DTU Compute
Department of Applied Mathematics and Computer Science

Exam hand-in

02686 Scientific Computing for Differential Equations

Anton Ruby Larsen (s174356)

Kongens Lyngby 2023

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

Introduction

In this assignment we will investigate one step methods for numerical approximation of differential equations. We will specifically focus on the class of methods called Runge-Kutta methods. In chapter 1 we will give a short review on the general theory underlying Runge-Kutta methods. In chapter 2 we will investigate the simplest explicit Runge-Kutta method called the explicit Euler method. We will test it on the Van Der Pol problem. In chapter 3 we will investigate the simplest implicit Runge-Kutta method called the implicit Euler method which we will also test on the Van Der Pol problem. In chapter 4 we will investigate the explicit-explicit and implicit-explicit methods for stochastic differential equations. These will be tested on two stochastic versions of the Van Der Pol problem. In chapter 5 we will look at a higher order explicit Runge-Kutta method called the classical Runge-Kutta method. We will again test on the Van Der Pol problem. In all previous chapters we have used an adaptive step size controller based on step doubling but in chapter 6 we will investigate the Dormand-Prince 5(4) which has an embedded error estimation which makes the control mechanism much cheaper. To test the Dormand-Prince 5(4) we will use the test equation, the Van Der Pol problem and the 1D and 3D CSTR problem. In chapter 7 we investigate a computational efficient implicit method called the ESDIRK23 method. This method we will also test on the Van Der Pol problem. In the final chapter we will compare the methods and discuss pros and cons for the different methods.

All code underlying the discussed models in this report has been created together with Andreas Heidelbach Engly, s170303, and Mads Esben Hansen, s174434. The code which is directly asked for in the exam paper is either shown in the text or given in the appendix. All other code is given in a zip file uploaded to learn.

Contents

CHAPTER 1

Test equation for ODEs

In this chapter we will go through some of the underlying theory for Runge-Kutta methods. To do this we will consider the linear initial value problem(IVP) called the test equation,

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

We will consider 1.1 with $\lambda = -1$ and $x_0 = 1$.

1.1 Exercise 1.1

The test equation is a linear IVP and hence we can solve it analytically. From [Per01], section 1.4 we know that all IVPs of the form,

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t), \qquad \mathbf{x}(0) = \mathbf{x}_0, \tag{1.2}$$

has a unique solution for all $t \in \mathbb{R}$ which is given by,

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0. \tag{1.3}$$

Hence 1.1 has the analytical solution,

$$x(t) = e^{\lambda t}x_0, \tag{1.4}$$

and with the specific values for λ and x_0 we have,

$$x(t) = e^{-t}. \tag{1.5}$$

We conclude hence that for our specific values, 1.1 should be asymptotically stable on the whole interval, $t \in [0, \infty]$.

1.2 Exercise 1.2

When given an IVP we can not always solve the system analytically and hence we must apply some approximate method. One family of such methods are Runge-Kutta methods, which will be the main focus of this assignment. As a first to understand these methods we will look at the the local truncation error and the global truncation error for a given Runge-Kutta method produce.

Local Truncation Error

We start with the local truncation error and how a Runge-Kutta method approximates an IVP. We can describe an arbitrary IVP on the interval $[t_0, t_0 + h]$ as

$$x(t_0 + h) = x_0 + \int_{t_0}^{t_0+h} f(t, x(t))\mathrm{dt}, \tag{1.6}$$

where $\dot{x}(t) = f(t, x(t))$. We can now approximate the LHS by the multivariate Taylor expansion which is given in theorem 2.2 in [Kre15] and here in Definition 1.

Definition 1. *Let $f \in C^{p+1}\left(\Omega, \mathbb{R}^d\right)$ with an open set Ω and $\mathrm{x} \in \Omega$. Then the multivariate Taylor expansion is given by*

$$f(\mathbf{x} + \mathbf{h}) = \sum_{k=0}^{p} \frac{1}{k!} f^{(k)}(\mathbf{x}) \cdot (\mathbf{h}, \ldots, \mathbf{h}) + O\left(\|\mathbf{h}\|^{p+1}\right)$$

for all sufficiently small $\mathbf{h} \in \mathbb{R}^m$.

We now introduce a generic autonomous IVP

$$\begin{cases} \dot{x}(t) = f(x(t)), & t \geq t_0 \\ x(t_0) = x_0 \end{cases} \tag{1.7}$$

We can now apply a first order Taylor expansion to 1.7 and obtain,

$$x(t_0 + h) = x_0 + h\dot{x}(t_0) + \mathcal{O}(h^2). \tag{1.8}$$

We recognize 1.8 as the explicit Euler method. Further we observe the method makes a local truncation error of order $\mathcal{O}(h^2)$. We define the local truncation error in Definition 2.

Definition 2. *Given an IVP as the one in 1.7 and a Runge-Kutta method, $\Phi(x_0, h, f, t_0)$, the local truncation error is given by,*

$$\boldsymbol{e}(h) = x(t_0 + h) - (x_0 + \Phi(x_0, h, f, t_0)) \tag{1.9}$$

$$= x(t_0 + h) - x_1. \tag{1.10}$$

If the Runge-Kutta method, $\Phi(\cdot)$, satisfy all terms up to order p in the multivariate Taylor expansion the local truncation error will be of order $\mathcal{O}(p + 1)$.

Global Truncation Error

It is though very uncommon that one only wants to integrate one step into the future and hence we would also like to know how the accumulated or global truncation error behaves. We define the global truncation error in Definition 3.

Definition 3. *Given an IVP as the one in 1.7 and a Runge-Kutta method, $\Phi(x_0, h, f, t_0)$, we integrate N steps ahead in time. The global truncation error is now given by,*

$$\boldsymbol{E} = x(t_0 + T) - x_N, \qquad T = t_N - t_0. \tag{1.11}$$

If the Runge-Kutta method, $\Phi(\cdot)$, satisfy all terms up to order p in the multivariate Taylor expansion the global truncation error will be of order $\mathcal{O}(p)$.

The local and global truncation error are illustrated in figure 1.1, which is taken from figure 2.1 in [Kre15].

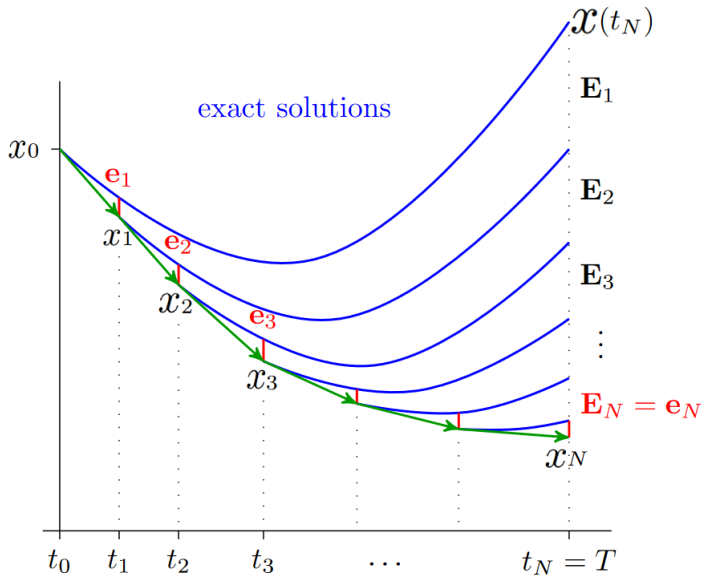

Figure 1.1: Lady Windemere's fan

Definition 3 gives us a definition of the global error but what we really are interested in is a upper bound of the error. This we have given and proved in theorem 2.16 in [Kre15], and here given in theorem 1.

Theorem 1. *Let U be a neighborhood of $\{(t, \mathbf{x}(t)) : t_0 \leq t \leq T\}$ such that the local error estimate, definition 2, and the Lipschitz constant*

$$\left\| \frac{\partial f}{\partial x} \right\| \leq L$$

hold in U. Then the global error, definition 3, satisfies

$$\|\mathbf{E}\| \leq h^p \frac{C}{L} \left(e^{L(T-t_0)} - 1 \right),$$

where C is some constant and $h = \max_i h_i$ is small enough for the numerical solution to remain in U.

Clearly, Theorem 1 indicates that we may run into severe problems as T grows, because of the exponentially growing error. This is though a very conservative estimate and ODEs are routinely integrated over relatively long time horizons.

1.3 Exercise 1.3

To be able to easier talk about different Runge-Kutta methods we will define a general framework.

Definition 4. *A method of the form:*

$$\begin{aligned}
\mathbf{k}_1 &= f\left(t_0 + c_1 h, \mathbf{y}_0 + h\sum_{\ell=1}^{s} a_{1\ell}\mathbf{k}_\ell\right)\\
&\vdots\\
\mathbf{k}_s &= f\left(t_0 + c_s h, \mathbf{y}_0 + h\sum_{\ell=1}^{s} a_{s\ell}\mathbf{k}_\ell\right)\\
\mathbf{y}_1 &= \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i
\end{aligned}$$

with all coefficients a_{il}, b_i, $c_i \in \mathbb{R}$, is called an s-stage Runge-Kutta method

The coefficients of a given method can be arranged in a matrix called a Butcher tableau. The general form of such a tableau is shown below.

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{1,1} & \cdots & a_{1,s} \\ \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & \cdots & a_{s,s} \\ \hline & b_1 & \cdots & b_s \end{array} \tag{1.12}$$

We are given the three specific methods given below.

$$\text{explicit Euler: } \begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{1.13}$$

$$\text{implicit Euler: } \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{1.14}$$

$$\text{classical Runge-Kutta: } \begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 2/6 & 2/6 & 1/6 \end{array} \tag{1.15}$$

We will now derive the local and global truncation error for the three methods. To do this we will use the relationship between x_1 and x_0 for the test equation given in 1.16. The relationship is from page 100 in [But02].

$$\begin{aligned} x_1 =& (1 + \lambda h b^T (I - \lambda h A)^{-1} \mathbf{1}_s) x_{n-1} \\ =& R(\lambda h) x_0 \end{aligned} \tag{1.16}$$

where $\mathbf{1}_s$ is a vector of s ones. We can hence calculate the local truncation error for the given Runge-Kutta methods by the following relation.

$$\begin{aligned} e_{testeq}(h) =& x(t_0 + h) - x_1 \\ =& \left(e^{\lambda(t_0+h)} - R(\lambda h) \right) x_0, \end{aligned} \tag{1.17}$$

and for an arbitrary number of steps into the numeric integration the local truncation error is given by,

$$e_{testeq}(h, t) = \left(e^{\lambda(t_0+h)} R(\lambda h)^{\frac{t-h}{h}} - R(\lambda h)^{\frac{t}{h}} \right) x_0. \tag{1.18}$$

The global truncation error is similarly given as

$$E_{testeq}(h, t) = \left(e^{\lambda(t_0+t)} - R(\lambda h)^{\frac{t}{h}} \right) x_0. \tag{1.19}$$

In figure 1.2, 1.3 and 1.4 we have respectively compared the analytical and empirical global and local truncation error for the explicit Euler, implicit Euler and classical Runge-Kutta method. We see that the observed global and local truncation error nicely matches with the analytical.

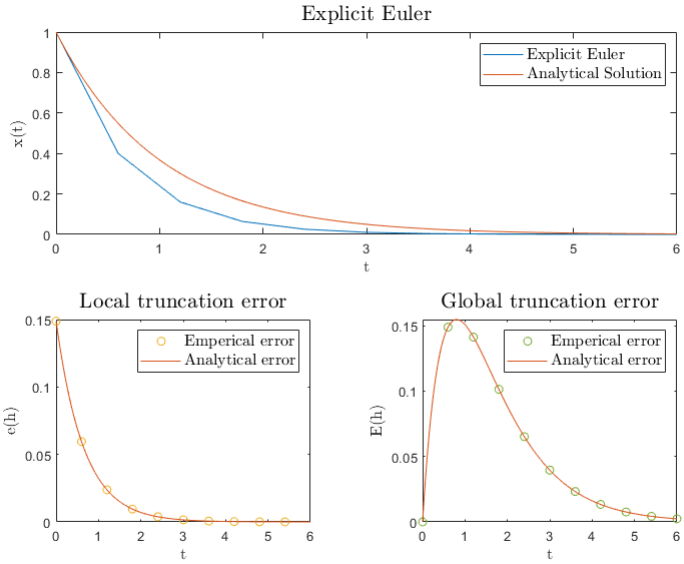

Figure 1.2: With a step size of $h = 0.6$

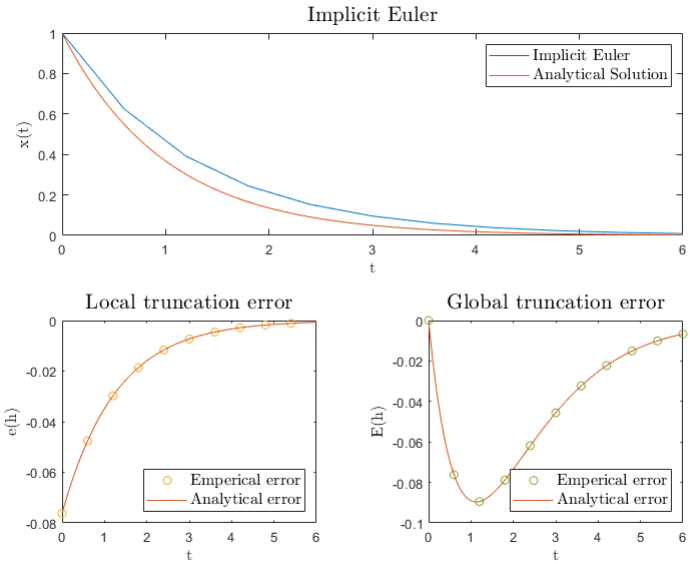

Figure 1.3: With a step size of $h = 0.6$

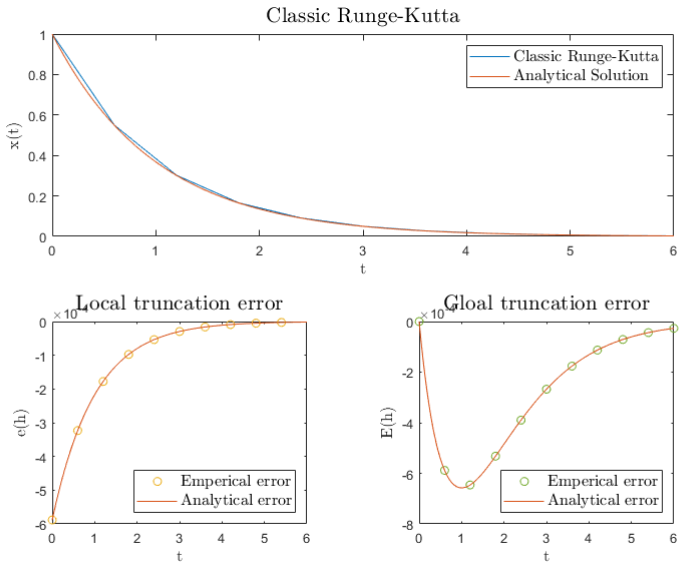

Figure 1.4: With a step size of $h = 0.6$

1.4 Exercise 1.4

We saw in figure 1.2, 1.3 and 1.4 that especially between the Euler methods and the classical Runge-Kutta method, there were a huge difference in the truncation error even though the step size was the same. From section 1.2 we know that the more terms in the Taylor expansion a method covers, the smaller an error it will make. It is though very tedious to write out all the terms in the Taylor expansion to check how many terms a method covers. This is also what is known as a methods order. We will hence introduce a easier way which uses the Butcher tableau.

Order conditions for Runge-Kutta methods

Before we dive into how one checks the order of a Runge-Kutta method we would also like our Runge-Kutta method to posses some other basic properties.

First the method should be consistent for all functions in the considered domain. This is given and proved in Lemma 2.6 in [Kre15] and here given in Theorem 2.

Theorem 2. *A Runge-Kutta method is consistent for all* $f \in C\left(\Omega, \mathbb{R}^d\right)$ *if and only if*

$$b^T \mathbf{1}_s = 1$$

Furthermore it would be nice if the Runge-Kutta method one designs both worked for autonomous and non-autonomous systems. This property is called invariant under autonomization and is given and proved in lemma 2.7 in [Kre15] and here stated in Theorem 3.

Theorem 3. *A Runge-Kutta method is invariant under autonomization if and only if it is consistent and satisfies*

$$c = A\mathbf{1}_s$$

Lastly we have the order of the method but to understand this theorem we must first understand rooted trees. We will not give a thorough explanation here but only explain a few quantities we need for the order theorem. For a deeper walk through we refer to [Kre15].

Rooted trees

When doing higher order Taylor expansions, and especially multivariate Taylor expansions, there are a lot of differentials of f to keep track of. A nice way to do this is by the use of rooted trees. To examplify how they work four simple rooted trees are depicted in figure 1.5.

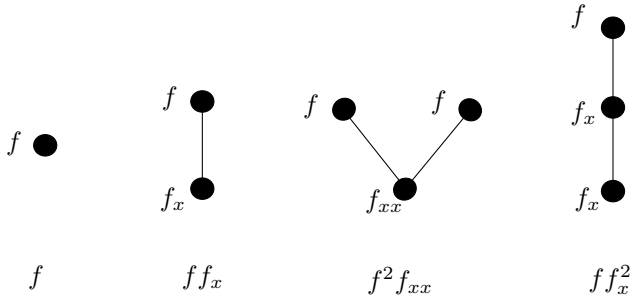

Figure 1.5: Example of four rooted trees

We see that the number of edges leaving a node corresponds to the number of times that node has been differentiated. Edges them self binds the nodes together by multiplication. When the root node in a tree, β, is deleted, the tree decomposes into k smaller tree, $\beta_1, ..., \beta_k$, as shown in figure 1.6.

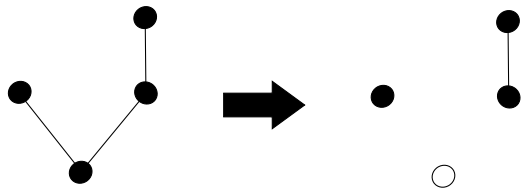

Figure 1.6: Deletion of the root node.

We will now define three quantities related to rooted trees we need, to be able the use the order theorem. The first is simply the number of nodes in a tree, β, which we will denote $\#\beta$. We define $\#\beta$ recursively as

$$\#\beta = 1 + \#\beta_1 + \cdots + \#\beta_k. \tag{1.20}$$

The second quantity is the factorial of a rooted tree which we will also define recursively as

$$!\beta = \#\beta \cdot \beta_1! \cdots \beta_k!. \tag{1.21}$$

The last quantity is the vector $A^{(\beta)} \in \mathbb{R}^s$, which is used when writing the Taylor expansion using rooted trees. To see more on the Taylor expansion for rooted trees we refer to page 20 in [Kre15]. Here we only state the recursive definition for $A^{(\beta)}$.

$$A_i^{(\beta)} = \left(A \cdot A^{(\beta_1)}\right)_i \cdots \left(A \cdot A^{(\beta_k)}\right)_i, \quad i = 1, \ldots, s \tag{1.22}$$

where s denote the number of stages in the respective method, A decodes for. Finally we can introduce the necessary and sufficient condition for a Runge-Kutta method to have order p. The theorem is from Theorem 2.12 in [Kre15] and here given in Theorem 4.

Theorem 4. *A Runge-Kutta method is of order p if and only if*

$$\mathbf{b}^\top A^{(\beta)} = \frac{1}{\beta!}$$

holds for all rooted trees β of order $\#\beta \leq p$.

We see from the explicit Euler, implicit Euler and the classical Runge Kutta methods Butcher tableaus stated in 1.13, 1.14 and 1.15 that they all are consistent and invariant under autonomization. To check their order we have used table 2.1 in [Kre15] where invariance under autonomization has been assumed to simplify $A^{(\beta)}$. We have stated all order condition up until order 4 in table 5.1.

1st Order	2nd Order	3rd Order	4th Order
$1 = \sum_{i=1}^{s} b_i$	$\frac{1}{2} = \sum_i b_i c_i$	$\frac{1}{3} = \sum_i b_i c_i^2$	$\frac{1}{4} = \sum_i b_i c_i^3$
		$\frac{1}{6} = \sum_{i,j} b_i a_{ij} c_j$	$\frac{1}{8} = \sum_{i,j} b_i c_i a_{ij} c_j$
			$\frac{1}{12} = \sum_{i,j} b_i a_{ij} c_j^2$
			$\frac{1}{24} = \sum_{i,j,k} b_i a_{ij} a_{jk} c_k$

Table 1.1: Order conditions for Theorem 4 up until 4th order.

We observe that the explicit and implicit Euler methods both are of only order 1 while the classical Runge-Kutta method is of order 4. To illustrate how an much higher order affects the local truncation error we plot the local truncation error against the step size for all three methods.

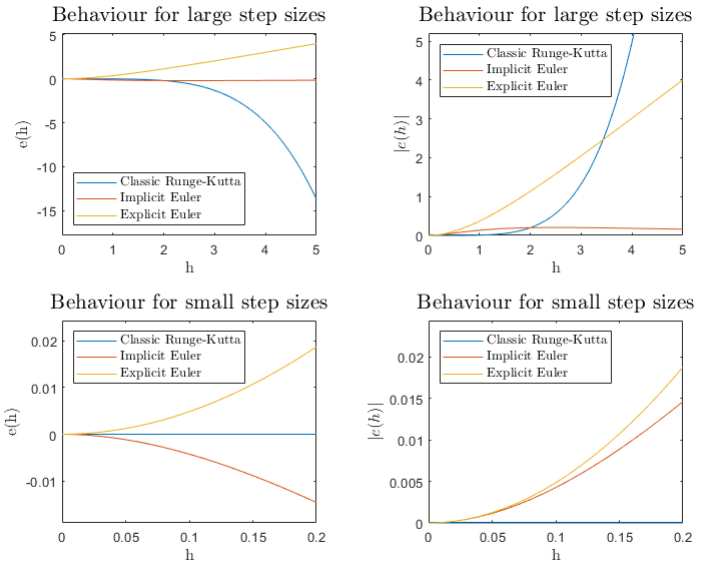

Figure 1.7: The local truncation error plotted against the step size.

We see in figure 1.7 that the classical Runge-Kutta method performs much better for larger step sizes compared to the Euler methods. We also see when we zoom in on the interval $h \in [0, 0.2]$ that the error of the implicit and explicit Euler methods grows approximately equally fast as their orders indicates.

1.5 Exercise 1.5

Having plotted the local truncation error we should also plot the global truncation error. We will plot the global truncation error for $T = 1.0$ against the step size.

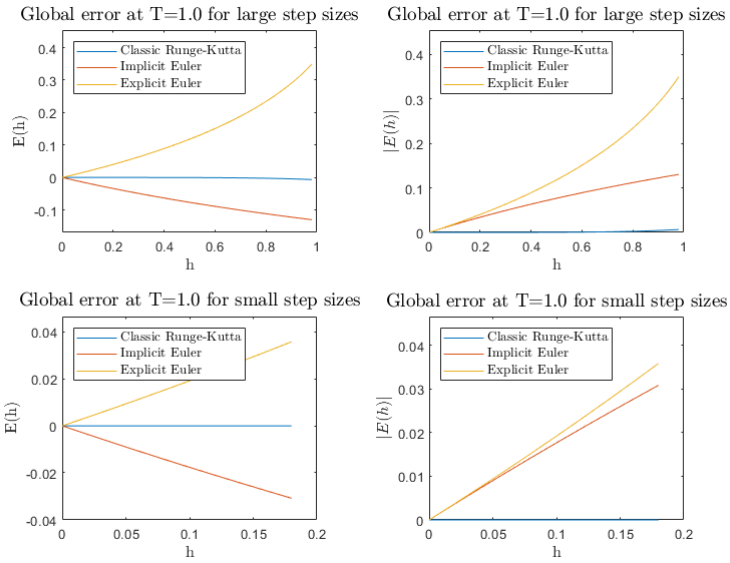

Figure 1.8: The global truncation error plotted for $T = 1.0$ against the step size.

We see a similar picture as for the local truncation error. We see that the classical Runge-Kutta method is out performing both the Euler methods. One other interesting observation is that on the smaller scale the implicit and explicit Euler methods are fairly similar but on the long scale the implicit method is out performing the explicit and it seems like it converges to some value. This has something to do with the stability of the methods which we will discuss next.

1.6 Exercise 1.6

In this section we will discuss the different stability concepts related to Runge-Kutta methods but before doing so we must first understand the stability of the continuous systems we try to approximate. Consider the following IVP

$$\dot{x}(t) = f(x(t)), \quad x(t_0) = x_0. \tag{1.23}$$

The stability for the IVP 1.23, is defined in Definition 5.

Definition 5. *Assume an IVP is given as in 1.23. Then the stability of the system is defined as follows:*

1. ***asymptotically stable*** *if* $\|\mathbf{x}(t)\| \to 0$ *as* $t \to \infty$ *for all initial values* $\mathbf{x}_0 \in \mathbb{R}^d$*;*
2. ***stable*** *if there is a constant* C *(independent of* t *and* $\mathbf{x}_0$ *) such that* $\|\mathbf{x}(t)\| < C\,\|\mathbf{x}_0\|$ *holds for all* $t \geq t_0$ *and* $\mathbf{x}_0 \in \mathbb{R}^d$
3. ***unstable****, otherwise.*

It is though very challenging to say anything about the stability of the approximation of so general a system as 1.23. We will hence limit our scope to only linear system as the one stated in 1.24.

$$\dot{x}(t) = Gx(t), \quad x(t_0) = x_0, \tag{1.24}$$

where G is a matrix. The stability of 1.24 is given by Theorem 5 which is proven in [Kre15].

Theorem 5. *The IVP 1.24 is asymptotically stable if and only if all eigenvalues* λ *of* G *satisfy* $\mathrm{Re}(\lambda) < 0$*. The IVP 1.24 is stable if and only if all eigenvalues* λ *of* G *satisfy* $\mathrm{Re}(\lambda) \leq 0$ *and if every eigenvalue* λ *with* $\mathrm{Re}(\lambda) = 0$ *has the same algebraic and geometric multiplicity.*

The system 1.24 is a generalization of the test equation, 1.1. For the test equation λ is the only eigenvalue for the one dimensional system. We have generalized the relationship 1.16 to a multidimensional form in 1.25. This relation is what we call a linear map of the linear IVP 1.24.

$$x_{n+1} = R(hG)x_n \tag{1.25}$$

The stability of a discrete map as 1.25 is given in Definition 6.

Definition 6. *Assume a discrete map as in 1.25. Then the stability of the map is defined as follows:*

1. ***asymptotically stable*** *if* $\|\mathbf{x}_k\| \to 0$ *as* $k \to \infty$ *for all initial values* $\mathbf{x}_0 \in \mathbb{R}^d$
2. ***stable*** *if there is a constant* C *(independent of* t *and* $\mathbf{x}_0$ *) such that* $\|\mathbf{x}_k\| < C\,\|\mathbf{x}_0\|$ *holds for all* $k \geq 0$ *and* $\mathbf{x}_0 \in \mathbb{R}^d$
3. ***unstable****, otherwise.*

Similarly to Theorem 5 for the linear continuous system we also have one for linear discrete maps. Theorem 6 is proved in [Kre15] under theorem 3.3.

Theorem 6. *The linear discrete map 1.25 is asymptotically stable if and only if all eigenvalues* λ *of* $R(hG)$ *satisfy* $|\lambda| < 1$*. The linear discrete map 1.25 is stable if and only if all eigenvalues* λ *of* $R(hG)$ *satisfy* $|\lambda| \leq 1$ *and if every eigenvalue* λ *with* $|\lambda| = 1$ *has the same algebraic and geometric multiplicity.*

Hence what we now want to answer is if our discrete map inherits the same stability as the continuous counter part. To answer this we need to investigate whether the eigenvalues of $R(hG)$ have absolute magnitude less than 1. We note that the eigenvalues of $R(hG)$ are given by $R(h\lambda)$ for every eigenvalue λ of G. We can hence define the stability region as

$$\mathcal{S} := \{h\lambda \in \mathbb{C} : |R(h\lambda)| \leq 1\}. \tag{1.26}$$

Below we have derived and plotted the stability regions for our three methods.

$$\text{The explicit Euler: } R(h\lambda) = 1 + h\lambda \tag{1.27}$$

$$\text{The implicit Euler: } R(h\lambda) = \frac{1}{1 - h\lambda} \tag{1.28}$$

$$\text{The classical Runge-Kutta: } R(h\lambda) = 1 + h\lambda + \frac{1}{2}(h\lambda)^2 + \frac{1}{6}(h\lambda)^3 + \frac{1}{24}(h\lambda)^4 \tag{1.29}$$

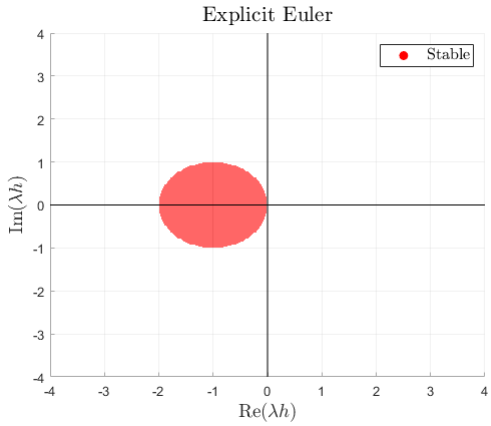

Figure 1.9: The stability region for the explicit Euler method

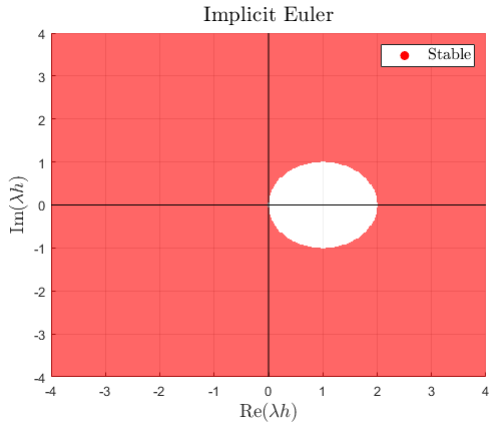

Figure 1.10: The stability region for the implicit Euler method

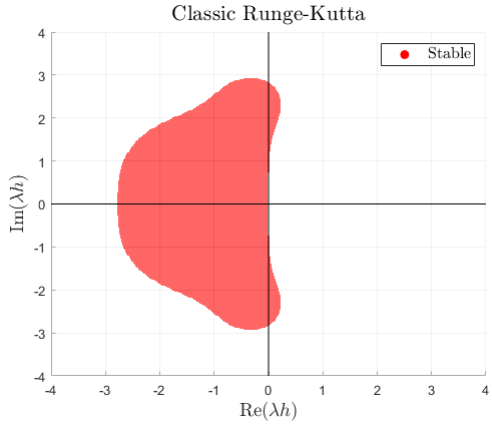

Figure 1.11: The stability region for the classical Runge-Kutta method

We observe that the stability region of the explicit Euler method and the classical Runge-Kutta method are polynomials and the stability region for the implicit Euler method is a rational function. The reason for this is that both the explicit Euler and the classical Runge-Kutta are explicit Runge-Kutta methods while the implicit Euler is an implicit Runge-Kutta methods. We will get to the difference between explicit and implicit methods in later chapters and hence only utilize page 31 in [Kre15] here to generalize our observation for the stability regions in Theorem 7.

Theorem 7. *Given a Runge-Kutta method we know that*

1. *If the method is an explicit method, then the stability region must be a polynomial.*

2. *If the method is an implicit method, then the stability region must be a rational function.*

A-stability

A very nice property for a method to posses is that no matter how large a step h we take, our method will still be stable if the original system is linear and stable. This property is called A-stability and is defined in Definition 7.

Definition 7. *A method is called A-stable if its stability region $\mathcal{S}$ satisfies $\mathbb{C}^- \subset \mathcal{S}$, where $\mathbb{C}^-$ denotes the left-half complex plane.*

We can therefore on the basis of the figures 1.9, 1.10 and 1.11 conclude that of our methods only the implicit Euler method is A-stable. Actually we can conclude something even stronger. From Lemma 3.5 in [Kre15] we know that if the stability region is given by a polynomial then the region must be compact. If the region is compact then it can not span the whole left plane and hence explicit Runge-Kutta methods can not be A-stable.

When we look at the stability region of the implicit Euler method we see that much more than the left plane is stable. This means it can act stable when the original system is not. To illustrate this we will consider the system

$$\dot{\mathbf{x}}(t) = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \mathbf{x}(t), \quad \mathbf{x}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{1.30}$$

We will simulate the system using the implicit Euler method using a step size of $h = 0.1$.

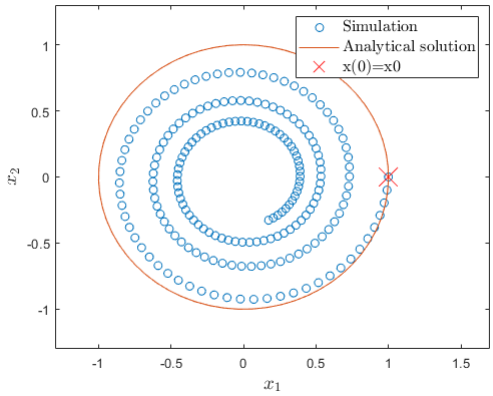

Figure 1.12: Simulation of 1.30 using the implicit Euler method with $h = 0.1$ and $T = 20$.

From figure 1.12 we see that the discrete map converges towards the origin even though the continuous system is stable on the unit circle. One method which solves this issue is the Trapezoidal method which is another implicit Runge-Kutta method. Its Buthcer tableau is given in 1.31 and its stability region is shown in figure 1.13.

$$\text{Trapezoidal:} \quad \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \tag{1.31}$$

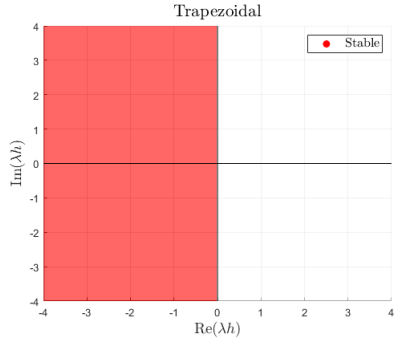

Figure 1.13: The stability region for the Trapezoidal method is given by $R(h\lambda) = \frac{1+1/2h\lambda}{1-1/2h\lambda}$

We see the stable region only covers the left-half plane and only the left-half plane. This gives some very nice stability properties regarding replicating the stability of the original system. This can be seen in figure 1.14 where 1.30 has been simulated using the Trapezoidal method with a step size of 4.

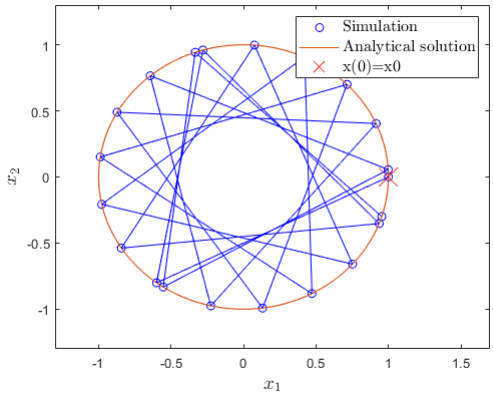

Figure 1.14: Simulation of 1.30 using the Trapezoidal method with $h = 4$ and $T = 80$.

We see that even for such a large step size it catches the right dynamic and the step size can be set as large as we want. It will not get thrown off the attractor.

L-stability

One problem with the Trapezoidal method though is how it damps errors in very stiff system. To illustrate this consider the system

$$\dot{x}(t) = -2000(x(t) - cos(t)), \quad x(0) = 0. \tag{1.32}$$

We will simulate the system for $t \in [0, 3]$ with $h = 1.5/40$ using the implicit Euler method and the Trapezoidal method. We see in figure 1.15 that the Trapezoidal method damps the initial error very poorly while the implicit Euler method jumps directly to the true trajectory.

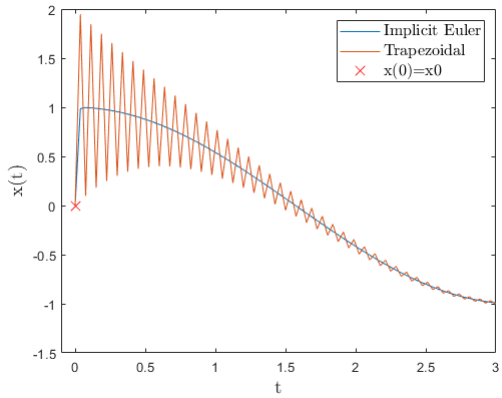

Figure 1.15: Simulation of 1.32 using the Trapezoidal and implicit Euler method with $h = 1.5/40$ and $t \in [0, 3]$.

The property of damping the initial error very quickly even in very stiff systems is called L-stability and is defined in Definition 8.

Definition 8. *A method is called L-stable if it is A-stable and if, additionally,*

$$\lim_{x \to \infty} R(x) = 0$$

From [Kre15] we have the following theorem regarding L-stability.

Theorem 8. *Given an implicit Runge-Kutta method, the method is L-stable if and only if,*

$$b^T A^{-1} \mathbf{1}_s = 1$$

We check the implicit Euler method.

$$b^T A^{-1} \mathbf{1}_s = 1^T 1^{-1} 1 = 1 \tag{1.33}$$

So the implicit Euler method is both A stable and L stable while the explicit Euler method and the classical Runge Kutta method is neither.

CHAPTER 2

Explicit ODE solver

In this chapter we will consider initial value problems of the form

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

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

2.1 Exercise 2.1

As described in chapter 1, equation 2.1 can on the interval $t \in [t_0, t_0 + h]$ be described as the integral equation

$$x(t_0 + h) = x_0 + \int_{t_0}^{t_0+h} f(t, x(t), p)\mathrm{dt}. \tag{2.2}$$

We can now approximate the integral by the rectangle $h \times f(t_0, x_0, p)$, also called a left Riemann sum, and obtain the approximation

$$x(t_0 + h) \approx x_1 = x_0 + hf(t_0, x_0, p). \tag{2.3}$$

This corresponds to the first order Taylor expansion of $x(t_0 + h)$ and is also known as the explicit Euler method which we also saw in chapter 1. It is the simplest Runge-Kutta method we have which fulfills both consistent and invariance under autonomization. Its Buthcer tableau was given in 1.13 and restated below.

$$\text{explicit Euler:} \quad \begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{2.4}$$

We will here give a pseudo code for the explicit Euler method with fixed step size.

Explicit Euler method:

1. Choose step size $h = (T - t_0)/N$.
2. for $j = 0, 1, \ldots, N - 1$ do

$$\begin{aligned} \mathbf{k}_1 &= f(t_j, \mathbf{x}_j) \\ \mathbf{x}_{j+1} &= \mathbf{x}_j + h\mathbf{k}_1 \\ t_{j+1} &= t_j + h \end{aligned}$$

end for

2.2 Exercise 2.2

We have implemented the explicit Euler method with fixed step size in Matlab. The method can be called from a common solver interface by

```
options = struct('step_control',false, 'initialStepSize', false);
[X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
    "Explicit Euler",options);
```

which inputs the relevant butcher tableau to a generic explicit Runge-Kutta code given below.

```
function [x,t,function_calls,hs] = ...
    explicitRungeKutta(f,param,h,t0,T,x0,A,b,c)
    N = ceil((T-t0)/h);
    t = zeros(1,N+1);
    hs = ones(1,N+1);
    hs = h.*hs;
    x = zeros(length(x0),N+1);
    s = length(b);
    k = zeros(length(x0),s)';
    t(1) = t0;
    x(:,1) = x0;
    function_calls = 0;

    Ah = h*A;
    bh = h*b;
    ch = h*c;

    for i = 2:N+1
        if t(i-1)+h > T
            h = T-t(i-1);
        end
        k = 0*k;
        for j = 1:s
            k(j,:) = f(t(i-1)+ch(j),x(:,i-1)+sum(k.*Ah(j,:)',1)', param);
            function_calls = function_calls +1;
        end
        x(:,i) = x(:,i-1)+sum(k.*bh,1)';
        t(i) = t(i-1)+h;
    end
    x = x';
end
```

Listing 2.1: The explicit Runge Kutta method with fixed step size

2.3 Exercise 2.3

A major disadvantage for methods using a fixed step size is that during the numeric integration the underlying dynamic is not equally challenging everywhere. A fixed step size approach forces the solver to use the step size that the hardest piece of the dynamic demands everywhere and hence waste a lot of compute power. It would hence be desirable to have an algorithm which could pick an h such that we were not wasting compute power but still maintained a local truncation error below a certain tolerance,

$$|e_i(h)| \leq \varepsilon, \forall i. \tag{2.5}$$

One problem though is how to estimate the local truncation error without access to the true solution. In [LW] they suggest to estimate the local truncation error by integrating the dynamics with one method of order p and one of order $p+1$. We could now use the more exact method as the true solution and the lower order as our numerical estimate when estimating the local truncation error.

We will refer to the methods by

1. Method A of order p: $\tilde{x}$
2. Method B of order $p+1$: $\hat{x}$

We now want to find a step size h_{new} such that

$$e_{i+1}(h_{new}) < \varepsilon \tag{2.6}$$

To ensure this we use the update rule given in 2.7 which is derived in [LW].

$$h_{new} = \gamma h \left(\frac{\varepsilon}{\|\tilde{x}_{i+1} - \hat{x}_{i+1}\|} \right)^{\frac{1}{p+1}} \tag{2.7}$$

Step doubling

We now have a way to control the error but it depends upon having access to one method of order p and one of order $p+1$. In chapter II.4 [EN08] they suggest using Richardson Extrapolation to solve this problem. Suppose that with a given initial value (t_0, x_0) and a step size h, we compute one step using a Runge-Kutta method of order p. This gives us now an estimate, $\tilde{x}_1$, for $x(t_0+h)$. We now reduce the step size to $h/2$, and compute two steps ahead with the same Runge-Kutta method of order p. This gives us another estimate, $\hat{x}_1$, for $x(t_0+h)$. We now know from Theorem 4.1 in [EN08] that $\tilde{x}_1$ is an estimate for $x(t_0+h)$ of order p, while $\hat{x}_1$ is of order $p+1$. We can hence use $\tilde{x}_1$ as our numerical estimate and $\hat{x}_1$ as the true solution to calculate the local truncation error.

PID control

The update rule 2.7 is an asymptotic error controller or an I-controller. From slideshow [Jør] and [SS18] more advanced control rules are presented. We will use the PI-control rule given in 2.8 and the PID-control rule given in 2.9. The coefficients in the exponents are from [SS18].

$$h_{new} = \gamma h \left(\frac{\varepsilon}{\|\tilde{x}_{i+1} - \hat{x}_{i+1}\|} \right)^{\frac{0.8}{p+1}} \left(\frac{\|\tilde{x}_{i+1} - \hat{x}_{i+1}\|}{\|\tilde{x}_i - \hat{x}_i\|} \right)^{\frac{0.31}{p+1}} \tag{2.8}$$

$$h_{new} = \gamma h \left(\frac{\varepsilon}{\|\tilde{x}_{i+1} - \hat{x}_{i+1}\|} \right)^{\frac{0.58}{p+1}} \left(\frac{\|\tilde{x}_{i+1} - \hat{x}_{i+1}\|}{\|\tilde{x}_i - \hat{x}_i\|} \right)^{\frac{0.21}{p+1}} \left(\frac{\|\tilde{x}_i - \hat{x}_i\|}{\|\tilde{x}_{i-1} - \hat{x}_{i-1}\|} \right)^{\frac{0.1}{p+1}} \tag{2.9}$$

To avoid dividing by zero we limit our observed local truncation error to be at least 10^{-10}.

The final algorithm

In the final adaptive step size algorithm we have incorporated a absolute and relative tolerance obtained from chapter II.4, section 'Automatic step control' in [EN08].

Adaptive step size:

1. Compute one step ahead with step size h and obtain $\tilde{x}_{i+1}$
2. Compute two steps ahead with step size $h/2$ and obtain $\hat{x}_{i+1}$
3. Compute
$$e_{i+1} = \tilde{x}_{i+1} - \hat{x}_{i+1}$$
$$sc_{i+1} = max_{k \in \{1,\dots,n\}} \left(Atol, |\hat{x}_{i+1}| Rtol \right)$$
$$r_{i+1} = \frac{e_{i+1}}{sc_{i+1}}$$
4. if $r_{i+1} \leq 1$
$$x_{i+1} = \hat{x}_{i+1}$$
$$t_{i+1} = t_i + h$$
$$h = controlRule(h, r_{i-1}, r_{i,i+1})$$
$$r_i = r_{i+1}$$
$$r_{i-1} = r_i$$

   else
$$h = controlRule(h, r_{i-1}, r_{i,i+1})$$
   Go to 1 and redo the step

We have implemented the explicit Euler method with adaptive time step and error estimation using step doubling in Matlab. The method can be called from a common solver interface with different controllers by

```
options = struct('step_control',true, 'initialStepSize', true, ...
    ,'control_type', "PID");
[X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
    "Explicit Euler",options);
```

which inputs the relevant butcher tableau to one of the codes given in appendix A.1 dependent on which of the control methods is chosen.

2.4 Exercise 2.4 and 2.5

In the following we will test on the Van Der Pol system. It is given as a two dimensional non-linear system of ODEs.

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= \mu \left(1 - x_1(t)^2\right) x_2(t) - x_1(t) \end{aligned} \tag{2.10}$$

The system gets stiffer and stiffer when the parameter μ is increased and we will through out this assignment test on a non-stiff version where $\mu = 3$ and a stiff version where $\mu = 20$. In this exercise we will test both the fixed and adaptive version of the explicit Euler but before doing so we will test the different control rules.

Control rules

The I, PI and PID controller was introduced in the last section. We will now test which of them gives the better performance on the Van Der Pol problem. We see from table 2.1 that for the non-stiff version the controllers are equally good. The PID controller takes 1.5% more steps than the I controller when the absolute and the relative tolerance is 10^{-3} but for higher tolerances there is no difference.

$\mu = 3$

Atol and Rtol	I	PI	PID
10^{-3}	21(1331)	23(1333)	37(1351)
10^{-4}	1(4136)	1(4136)	1(4134)
10^{-5}	1(13077)	1(13077)	1(13077)
10^{-6}	1(41344)	1(41344)	1(41344)
10^{-7}	1(130731)	1(130731)	1(130731)

Table 2.1: In the table rejected steps are shown out side the parenthesis and the total number of steps is shown inside the parenthesis, i.e. "rejected(total)"

In table 2.2 we have tested the controllers on the stiff version of the Van Der Pol system. Here we see that the PID controller rejects much fewer steps and takes much fewer steps in total. Hence we will use the PID controller in the rest of this assignment except the very last exercise which treats a method called ESDIRK23.

$\mu = 20$

Atol and Rtol	I	PI	PID
10^{-3}	135(1069)	133(1115)	98(960)
10^{-4}	123(2560)	86(2554)	79(2470)
10^{-5}	105(7347)	70(7296)	55(7272)
10^{-6}	3(22491)	1(22489)	1(22489)
10^{-7}	1(71094)	1(71095)	1(71097)

Table 2.2: In the table rejected steps are shown out side the parenthesis and the total number of steps is shown inside the parenthesis, i.e. "rejected(total)"

Fixed Step Size

We now move on to the testing of the explicit Euler method. First we test it with fixed step size. We initialize the solver at $x_0 = [1.0, 1.0]$ and run the simulation on the time interval $t \in [0, 4\mu]$. We first consider the explicit Euler with fixed step size on the non-stiff version of the Van Der Pol problem. We see in figure 2.1 that when the step size is $h = 0.1$, our method does not diverge but has a huge error compared to the Matlab native solvers. When the step size is decreased to $h = 0.01$ it gets much better but still a small error is visible and this will only get worse if we simulated over a longer time horizon. In the last column we have decreased the step size to $h = 0.001$ and we can not detect an error compared to Matlab's solvers anymore. We hence conclude that our method is correct and if used one should not use a step size over $h = 0.001$ for the non stiff system.

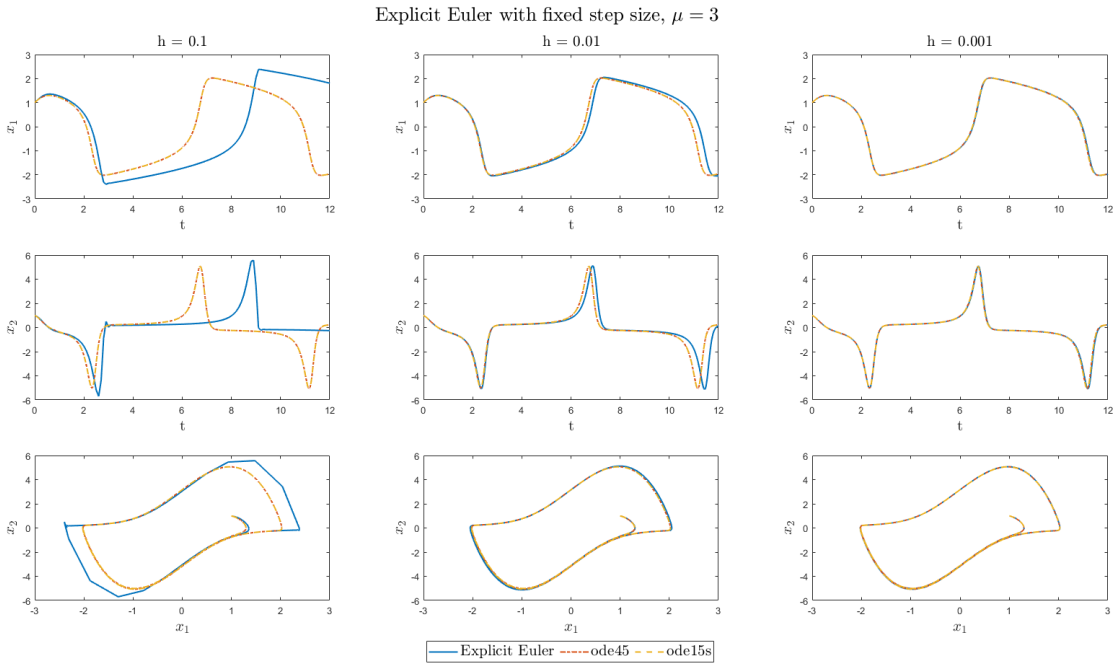

Figure 2.1: We have tested the explicit Euler method with fixed step size on the Van Der Pol system with $\mu = 3$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

We now proceed to the stiff version of the Van Der Pol system. In figure 2.2 we see that for the largest step size, $h = 0.1$, the explicit Euler method diverges. If we decrease the step size to $h = 0.01$ we see a similar picture to the non-stiff system with step size $h = 0.1$. A significant error is present compared to the Matlab solvers which would only get worse on longer time horizons. In the last column in figure 2.2 we decrease the step size to $h = 0.001$ and the error is similar to the second column in figure 2.1. Hence if one were to use the explicit Euler with fixed step size one should use a step size below $h = 0.001$ and from the pattern we have seen, $h = 0.0001$ would probably do it. This is though very computationally demanding because very many steps must be taken and only at the sharp spikes the system is stiff and the step size must be very low. We will hence proceed with the adaptive version which solves the problem of having one step size for the whole simulation.

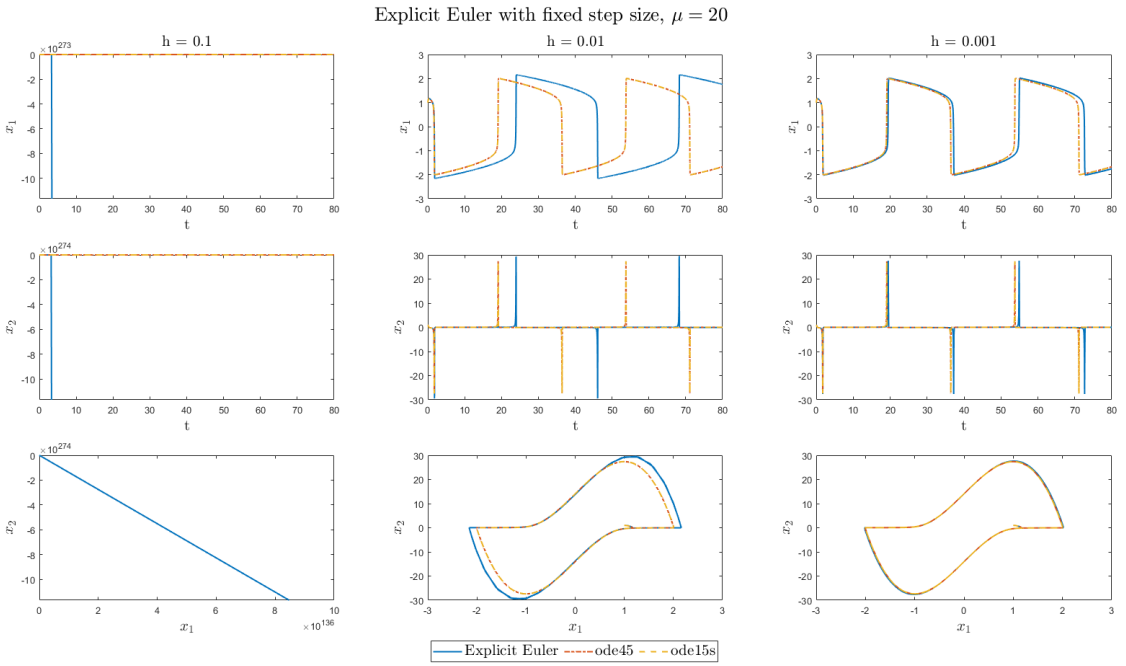

Figure 2.2: We have tested the explicit Euler with fixed step size on the Van Der Pol system with $\mu = 20$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

Adaptive Step Size

In figure 2.3 we have tested the explicit Euler with adaptive step size with different tolerances against the Matlab solvers on the non-stiff version of the Van Der Pol system. We see that when Atol = Rtol = 10^{-2} we have a significant error compared to both `ode45` and `ode15s`. When we decrease the tolerance to 10^{-4} we see that a very small error is visible and it completely disappears when we further decrease the tolerance to 10^{-6}.

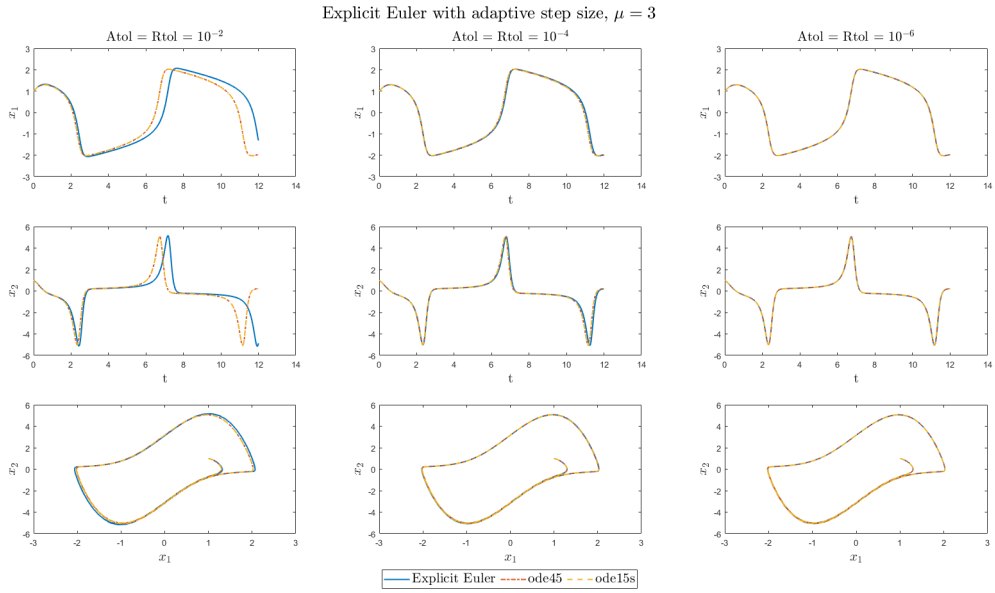

Figure 2.3: We have tested the explicit Euler with adaptive step size on the Van Der Pol system with $\mu = 3$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 2.4 we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism. We see that only when the tolerance is 10^{-2}, step sizes are being rejected.

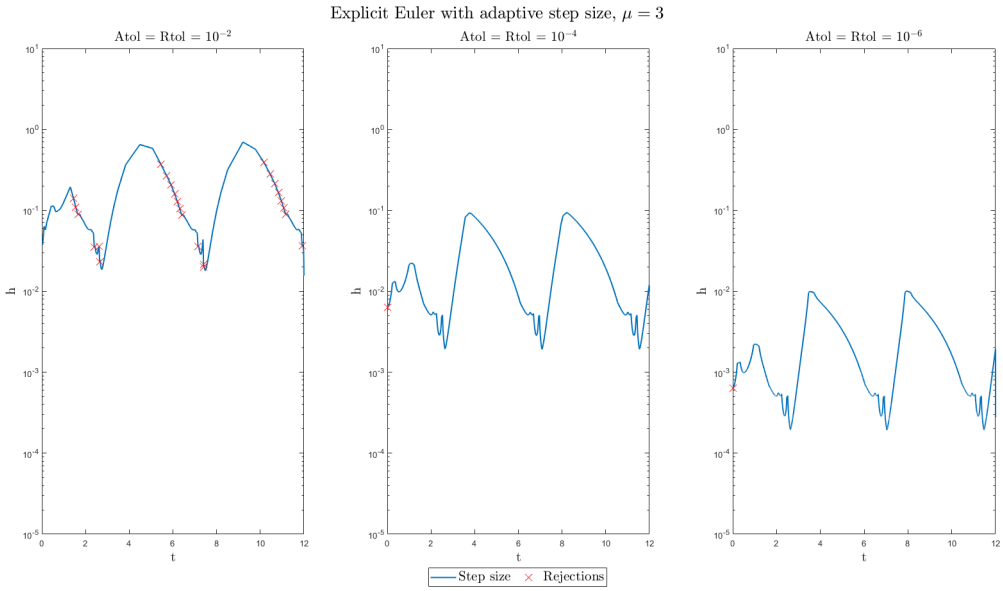

Figure 2.4: Here we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

We now proceed to testing on the stiff version of the Van Der Pol system. In figure 2.5 we see a similar picture as for the non-stiff system. When the tolerance is 10^{-2} we have a large error, when the tolerance is 10^{-4} a very small error is present and lastly for a tolerance of 10^{-6} no error is visible anymore.

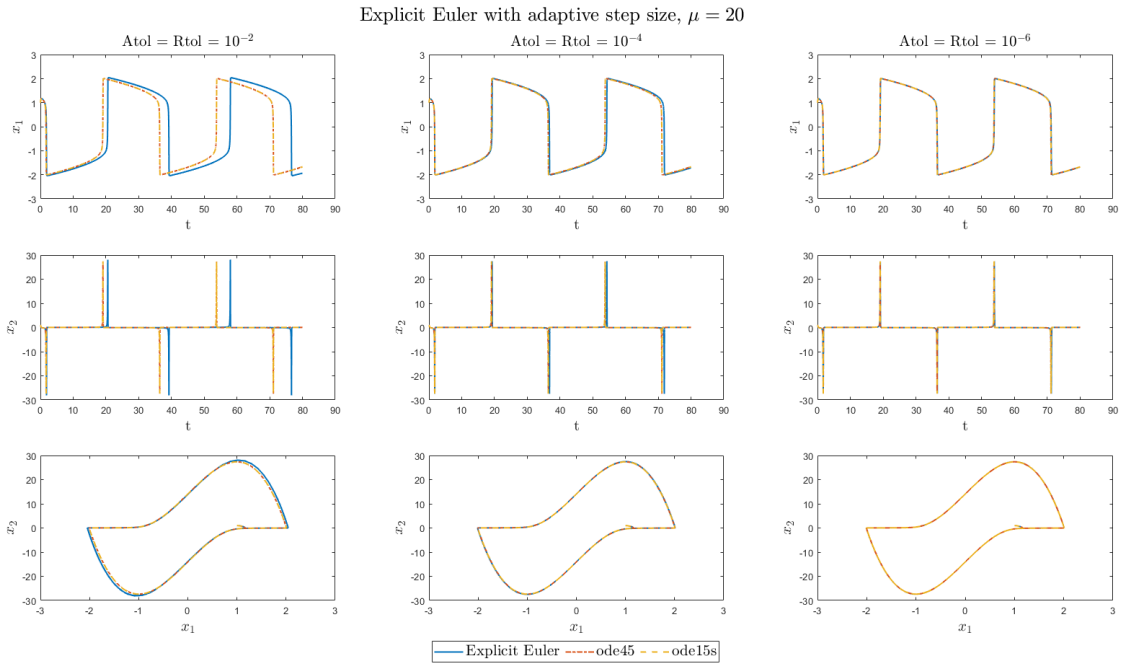

Figure 2.5: We have tested the explicit Euler with adaptive step size on the Van Der Pol system with $\mu = 20$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

For the acceptance of step sizes the picture is a bit different compared to the non-stiff system. We see that for both tolerances of 10^{-2} and 10^{-4} we have rejections and the behaviour of the control mechanism is not smooth but very jagged. If the tolerance is further decreased to 10^{-6}, the step size selection gets smooth and no rejections are made any more.

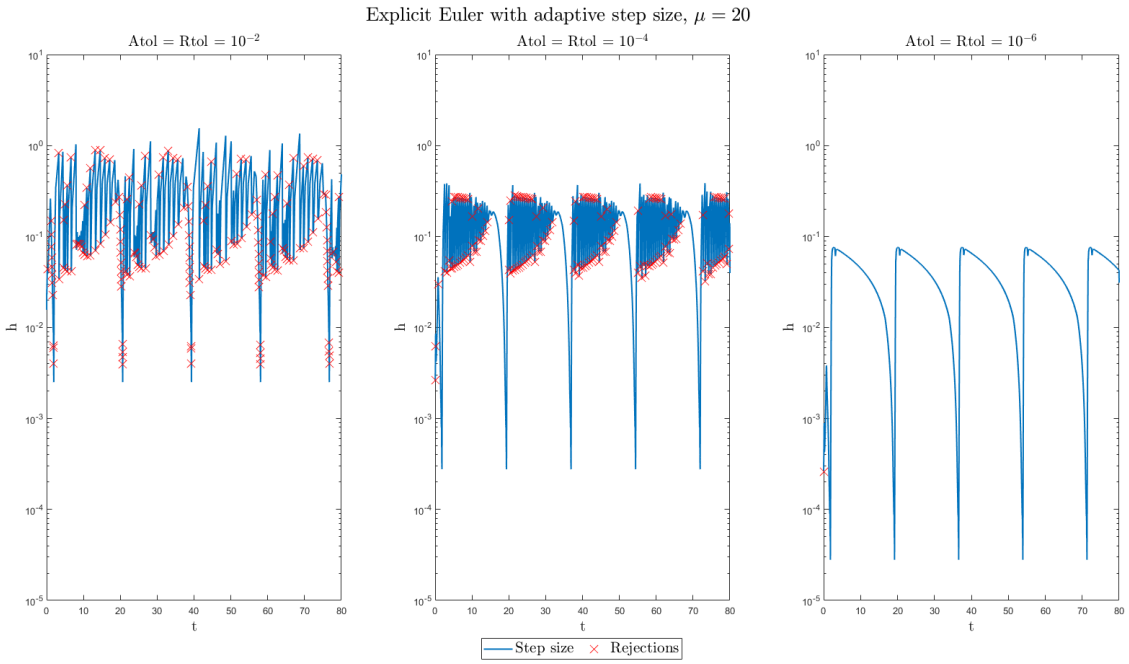

Figure 2.6: Here we have plotted the accepted step sizes for the test on the stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

Comparison

In this last section we willl compare the fixed and adaptive explicit Euler. In table 2.3 we have compared the methods on the non-stiff system. We saw that to obtain accurate simulations, we needed a step size of $h = 0.001$ or a tolerance at 10^{-6}. From the table we see that for a accurate simulation, the adaptive is slower for the non-stiff system but only a little. In the adaptive we need to do three solves per iteration and hence even though we do fewer steps in the adaptive than the fixed approach the simulation takes longer due to the increased number of function evaluations.

	Fixed Step			Adaptive Step		
h/tol	0.1	0.01	0.001	10^{-2}	10^{-4}	10^{-6}
Time	0	0	0.063	0.016	0.016	0.219
Function calls	$1.2 \cdot 10^3$	$1.2 \cdot 10^4$	$1.2 \cdot 10^5$	464	4178	41783
Steps	$1.2 \cdot 10^3$	$1.2 \cdot 10^4$	$1.2 \cdot 10^5$	154	1392	13927

Table 2.3: Comparison for the non-stiff system

In table 2.4 we compare the two methods on the stiff system. Here we saw that a step size of $h = 0.001$ was not enough to obtain an accurate simulation for the fixed step size approach. Hence we have included a step size of $h = 0.0001$. For the adaptive approach a tolerance of 10^{-6} was enough to gain an accurate simulation. We see that now the adaptive approach is much faster than the fixed approach for a accurate simulation. 6 times faster and almost 700000 fewer function calls. The reason that the adaptive approach really pays off for the stiff system compared to the non-stiff system is that for stiff systems the difference between "easy" and "hard" places is very large. Hence the fixed approach must decrease the step size every where while the adaptive is able to decrease the step size only where it is needed.

	Fixed Step				Adaptive Step		
h/tol	0.1	0.01	0.001	0.0001	10^{-2}	10^{-4}	10^{-6}
Time	0	0.078	0.359	3.625	0.016	0.063	0.563
Function calls	$8 \cdot 10^3$	$8 \cdot 10^4$	$8 \cdot 10^5$	$8 \cdot 10^6$	2639	12512	111641
Steps	$8 \cdot 10^3$	$8 \cdot 10^4$	$8 \cdot 10^5$	$8 \cdot 10^6$	879	4170	37213

Table 2.4: Comparison for the stiff system

CHAPTER 3

Implicit ODE solver

In this chapter we will again consider an IVP of the form

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

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

3.1 Exercise 3.1

We again write the IVP, 6.1, as an integral equation on the interval $[t_0, t_0 + h]$ as in section 2.1.

$$x(t_0 + h) = x_0 + \int_{t_0}^{t_0+h} f(t, x(t), p)\mathrm{dt}. \tag{3.2}$$

In section 2.1 we used a left Riemann sum to estimate 3.2. We now instead use a right Riemann sum which gives the approximation given in 3.3.

$$x(t_0 + h) \approx x_1 = x_0 + hf(t_1, x_1, p) \tag{3.3}$$

This method is called the implicit Euler method which is the simplest method in the family of implicit Runge Kutta methods. We also used the method in chapter 1 where the the Buthcer tableau was given but we restate it here for good measure.

$$\text{Implicit Euler: } \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{3.4}$$

One major increase in complexity we face when going from explicit to implicit methods is the reliance on future information. In the implicit Euler method, we for example do not know the value of x_1 at x_0 but to step from x_0 to x_1 we need this information. Hence for a general implicit Runge-Kutta method we need to solve the following system

$$k_i = x_0 + h \sum_{j=1}^{s} a_{ij} f(t_0 + c_j h, k_j, p), \quad i = 1, \dots s \tag{3.5}$$

This is a system of $s \times m$ coupled non-linear equations where m is the dimension of x. For a implicit Euler method $s = 1$ but we still need to solve it with Newtons method which is very expensive. Later we will get back to how we can reduce the computational cost but here we will just explain a very basic version of Newtons.

Newtons

We will not explain how to implement a general implicit Runge Kutta method but only focus on the implicit Euler. For a general theory we recommend section 3.2.1 in [Kre15]. For the implicit Euler we define the residual function

$$R(k) = k - x_0 - hf(t_0 + h, k, p) \tag{3.6}$$

To implement Newtons we also need the Jacobian of $R(k)$.

$$\frac{\partial R}{\partial k}(k) = I - h\frac{\partial f}{\partial k}(t_0 + h, k, p) \tag{3.7}$$

We now have the iteration

$$k_{l+1} = k_l - R(k_l)\left(\frac{\partial R}{\partial k}(k_l)\right)^{-1}, \tag{3.8}$$

and when

$$R(k_L)\left(\frac{\partial R}{\partial k}(k_L)\right)^{-1} \leq \varepsilon, \tag{3.9}$$

then

$$x_1 = k_L. \tag{3.10}$$

As an initial guess for k_0 we use an explicit Euler step

$$k_0 = x_0 + hf\left(t_0, x_0\right) \tag{3.11}$$

We will not provide a pseudo code in this exercise but just refer to the code given in listings 3.2.

3.2 Exercise 3.2

We have implemented the implicit Euler method with fixed step size in Matlab. The method can be called from a common solver interface

```
1 options = struct('step_control',false, 'initialStepSize', false);
2 [X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
      "Implicit Euler",options);
```

which calls the code given in

```
function [x,t,function_calls,hs] = ...
    implicitEulerFixed(f,jac,param,h,t0,T,x0,newtonTolerance,newtonMaxiterations)

    N = ceil((T-t0)/h);
    t = zeros(1,N+1);
    hs = ones(1,N+1);
    hs = h.*hs;
    x = zeros(length(x0),N+1);
    t(1) = t0;
    x(:,1) = x0;
    function_calls = 0;


% Fix 2:N+1
xdot = f(t(1), x(:,1), param); % first guess
function_calls = function_calls +1;
for i = 2:N+1
    if t(i-1)+h > T
        h = T-t(i-1);
    end
    t(i) = t(i-1)+h;
    xguess = x(:,i-1)+xdot*h; % We guess on a explicit euler step
    [x(:,i), xdot, function_calls] = newtonsMethod(f, jac,  t(:,i-1), ...
        x(:,i-1), h, xguess, newtonTolerance, newtonMaxiterations, ...
        param, function_calls);
end

t = t';
x = x';
end
```

Listing 3.1: The Implicit Runge Kutta method with fixed step size

with the associated Newton method given in

```
function [x, xdot, function_calls] = newtonsMethod(f, jac, told, ...
    Rterm, h, xguess, tolerance, maxiterations, params, function_calls)
%NEWTONSMETHODODE Does up to maxiterations rounds of newtons method to ...
    find
% the next step of the implicit within given tolerance.
% In case of slow convergence, it returns the guess it has reached at ...
    that
% point.
% f - the ODE
% jac - the jacobian
% told - the last time step
% Rterm - the term in the residual not encompassed by x_n+1 nor hf(x_n+1)
% dt - The step size
% xguess - The initial starting point (x_n+hf(x_n))
% tolerance - The size of the residual at which we exit
% maxiterations - Maximum allowed number of Newton iterations
% params - Parameters for f and jac
i = 0;
```

```
16 t = told + h;
17 x = xguess;
18 xdot = f(t, x, params);
19 J = jac(t,x,params);
20 function_calls(1) = function_calls +2;
21 R = x - xdot*h - Rterm;
22 I = eye(length(Rterm));
23 while (i < maxiterations) && (max(abs(R)) > tolerance) %Iteratively ...
       improve guess using Newton's method.
24
25     %The Jacobian tells us the change of the residual in x, and we ...
           then try
26     %to improve x using that information.
27     dRdx = I - J * h;
28     dx = dRdx\R;
29     x = x - dx;
30     xdot = f(t, x, params);
31     %Calculate jacobian and residual again
32     J = jac(t,x,params);
33     function_calls = function_calls +2;
34     R = x - h*xdot - Rterm;
35     i = i+ 1;
36 end
37 if i==maxiterations && (max(abs(R)) > tolerance)
38     disp("Not converging..") %big oh no if we're not converging
39 end
```

Listing 3.2: Basic Newton method

3.3 Exercise 3.3

We have implemented the implicit Euler method with adaptive step size in Matlab. For the adaptive step size we have used step doubling to estimate the error. This theory is described in chapter 2 and hence not repeated here. The method can be called from a common solver interface by

```
1 options = struct('step_control',true, 'initialStepSize', true);
2 [X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
      "Implicit Euler",options);
```

which calls the code given in

```
1 function [x,t,function_calls,hs,rs] = ...
      implicitEulerDoubling(f,jac,param,h,t0,T,x0,Atol,Rtol,hmin,hmax,eps_tol,initial_step_
      newtonMaxiterations)
2     N = ceil((T-t0)/hmin);
3     p = 1;
```

```
4       t = zeros(1,N+1);
5       hs = zeros(2,N+1);
6       rs = zeros(2,N+1);
7       x = zeros(length(x0),N+1);
8       t(1) = t0;
9       x(:,1) = x0;
10      accept_step = false;
11      function_calls = 0;
12      fac = 1;
13
14      if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
15          [h,function_calls] = initialStepSize(f,param,t0,x0);
16      end
17
18      i = 2;
19      l = 2;
20      runs = 1;
21      %h_old = h;
22      xdot = f(t(1), x(:,1), param); % first guess
23      function_calls = 3;
24      while t(i-1)≤T
25          while(¬accept_step)
26              if t(i-1)+h>T
27                  h = max(hmin,T-t(i-1));
28              end
29
30              t(i) = t(i-1)+h;
31              xguess = x(:,i-1)+xdot*h; % We guess on a explicit euler step
32              [oneAhead, ¬, function_calls] = newtonsMethod(f, jac,  ...
                    t(:,i-1), x(:,i-1), h, xguess, newtonTolerance, ...
                    newtonMaxiterations, param, function_calls);
33
34              hhalf = h/2;
35              xguess = x(:,i-1)+xdot*hhalf; % We guess on a explicit ...
                    euler step
36              [halfAhead, xdotHalf, function_calls] = newtonsMethod(f, ...
                    jac,  t(:,i-1), x(:,i-1), hhalf, xguess, ...
                    newtonTolerance, newtonMaxiterations, param, ...
                    function_calls);
37
38              xguess = halfAhead+xdotHalf*hhalf; % We guess on a explicit ...
                    euler step
39              [halfAhead, xdot, function_calls] = newtonsMethod(f, jac,  ...
                    t(:,i-1)+hhalf, halfAhead, hhalf, xguess, ...
                    newtonTolerance, newtonMaxiterations, param, ...
                    function_calls);
40
41              % Step doubling, see section II.4(s164) in Harier and slide ...
                    4c,3
42              e = abs(oneAhead-halfAhead);
43              r = max(e./max(Atol,abs(halfAhead).*Rtol));
44
45              if r≤1
46                  x(:,i) = halfAhead;
47                  hs(1,i-1) = h;
```

```
48                  hs(2,i-1) = runs;
49                  rs(1,l-1) = r;
50                  rs(2,l-1) = 1;
51                  runs = 1;
52                  t(i) = t(i-1)+h;
53                  accept_step = true;
54                  h = h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))); % ...
                        eq 4.13 in Harier
55              else
56                  runs = runs +1;
57                  accept_step = false;
58                  rs(1,l-1) = r;
59                  rs(2,l-1) = 2;
60                  h = h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))); % ...
                        eq 4.13 in Harier
61                  l = l+1;
62              end
63
64          end
65          accept_step = false;
66          i = i+1;
67          l = l+1;
68      end
69      x = x(:,1:i-1)';
70      t = t(1:i-1);
71      hs = hs(:,1:i-2);
72      rs = rs(:,1:l-2);
73  end
```

Listing 3.3: The implicit Runge Kutta method with adaptive step size

The method uses the same newton method as the fixed step size approach.

3.4 Exercise 3.4 and 3.5

In the following we will test on the Van Der Pol system as we also did in chapter 2. We have restated it here.

$$\begin{aligned} \dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= \mu \left(1 - x_1(t)^2\right) x_2(t) - x_1(t) \end{aligned} \tag{3.12}$$

We will also here test with $\mu = 3$ and $\mu = 20$.

Fixed Step Size

First we test it with fixed step size. We initialize the solver at $x_0 = [1.0, 1.0]$ and run the simulation on the time interval $t \in [0, 4\mu]$. We first consider the implicit Euler with fixed step size on the non-stiff version of the Van Der Pol problem.

We see in figure 3.1 that when the step size is $h = 0.1$, our method does not diverge but has a huge error compared to the Matlab native solvers. When the step

size is decreased to $h = 0.01$ it gets much better but still a small error is visible and this will only get worse if we simulated over a longer time horizon. In the last column we have decreased the step size to $h = 0.001$ and we can not detect an error compared to Matlabs solvers. We hence conclude that our method is correct and if used one should not use a step size over $h = 0.001$ for the non stiff system.

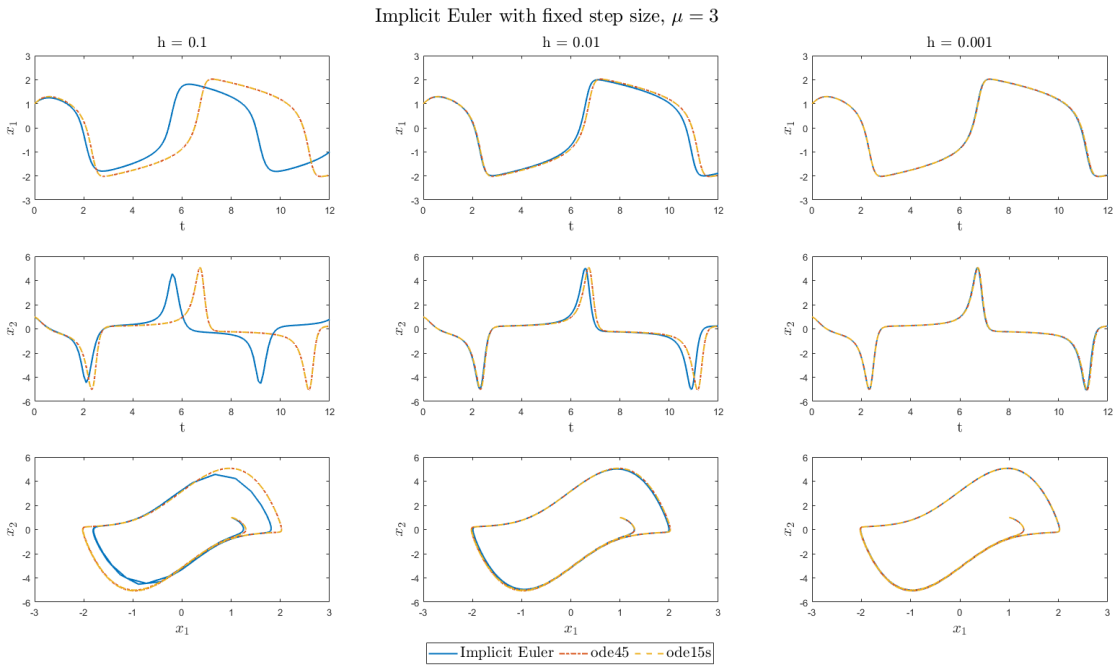

Figure 3.1: We have tested the implicit Euler with fixed step size on the Van Der Pol system with $\mu = 3$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

We now proceed to the stiff version of the Van Der Pol system. It is shown in figure 3.2 where we see that for the largest step size, $h = 0.1$, the implicit Euler makes a huge error. For this configuration we saw the explicit Euler diverge and hence we also see that the increased stability helps the implicit Euler. That said the error is still huge so we decrease the step size to $h = 0.01$. We see a similar picture to the non-stiff system with step size $h = 0.1$. A significant error is present compared to the Matlab solvers which would only get worse on longer time horizons. In the last column in figure 3.2 we decrease the step size to $h = 0.001$ and the error is similar to the second column in figure 3.1. Hence if one were to use the implicit Euler with fixed step size one should use a step size below $h = 0.001$ and from the pattern we have seen $h = 0.0001$ would probably do it. This is though very computationally demanding

because very many steps must be taken and only at the sharp turns the system is stiff and the step size must be very low. We will hence proceed with the adaptive version which solves the problem of having one step size for the whole simulation.

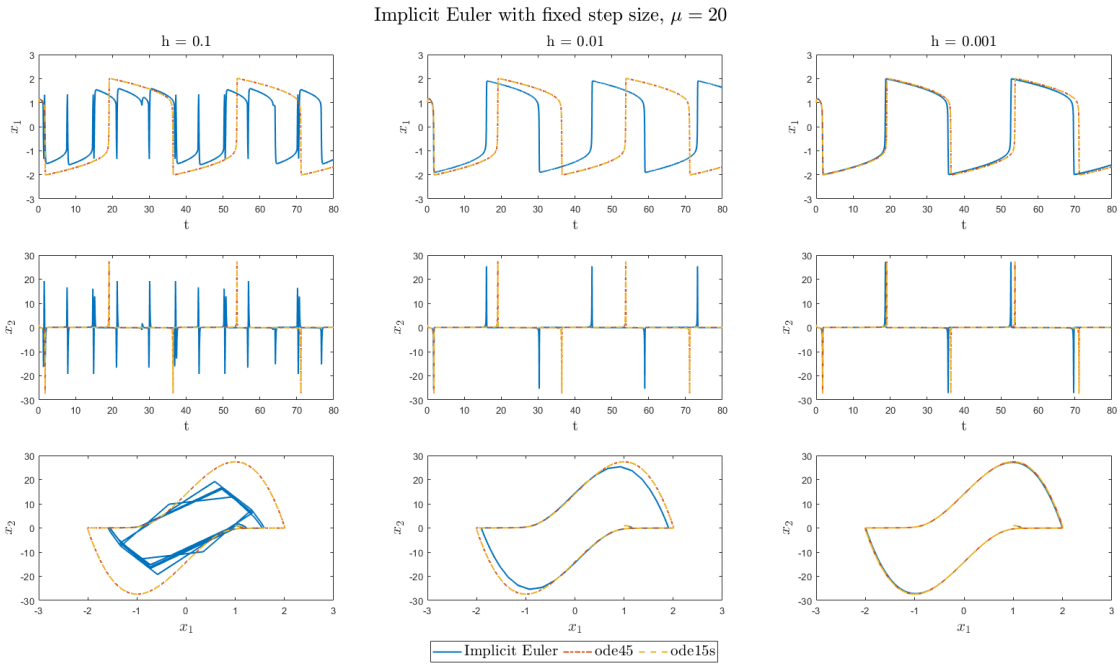

Figure 3.2: We have tested the implicit Euler with fixed step size on the Van Der Pol system with $\mu = 20$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

Adaptive Step Size

In figure 3.3 we have tested the implicit Euler with adaptive step size with different tolerances against the Matlab solvers on the non-stiff version of the Van Der Pol system. We see that when Atol = Rtol = 10^{-2} we have a significant error compared to both `ode45` and `ode15s`. When we decrease the tolerance to 10^{-4} we see that a very small error is visible and it completely disappears when we further decrease the tolerance to 10^{-6}.

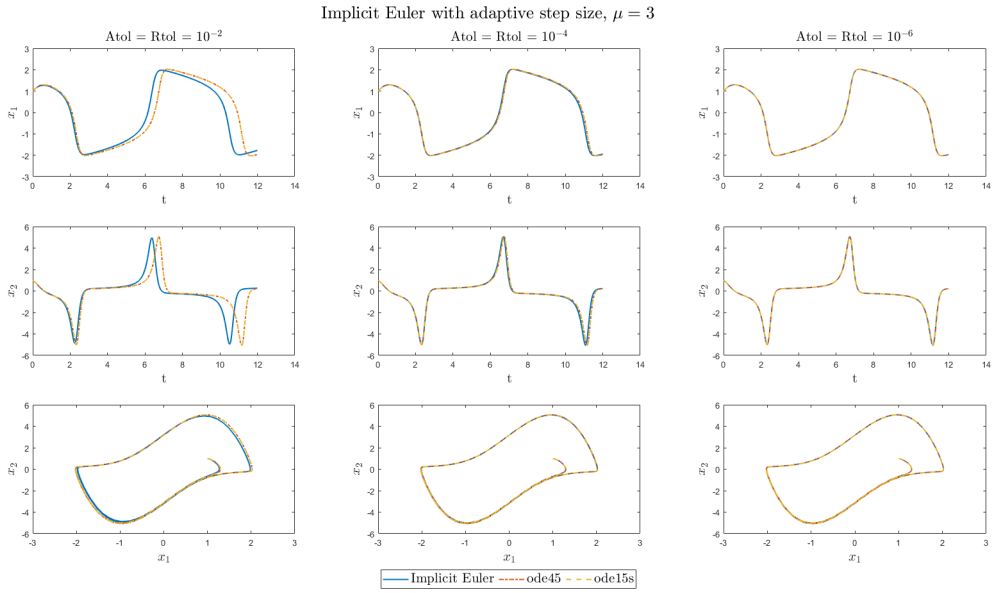

Figure 3.3: We have tested the implicit Euler with adaptive step size on the Van Der Pol system with $\mu = 3$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 3.4 we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size, 1 or more larger step sizes was proposed but rejected by the control mechanism. We see that only when the tolerance is 10^{-2}, step sizes are being rejected.

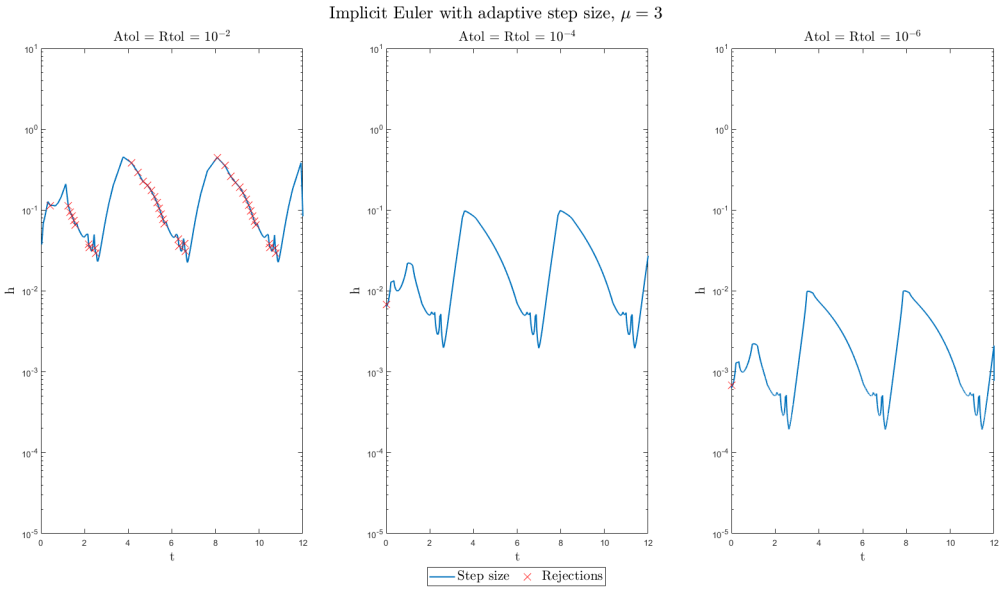

Figure 3.4: Here we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

We now proceed to testing on the stiff version of the Van Der Pol system. In figure 3.5 we see a similar picture as for the non-stiff system. When the tolerance is 10^{-2} we have a large error, when the tolerance is 10^{-4} a very small error is present and lastly for a tolerance of 10^{-6} no error is visible anymore.

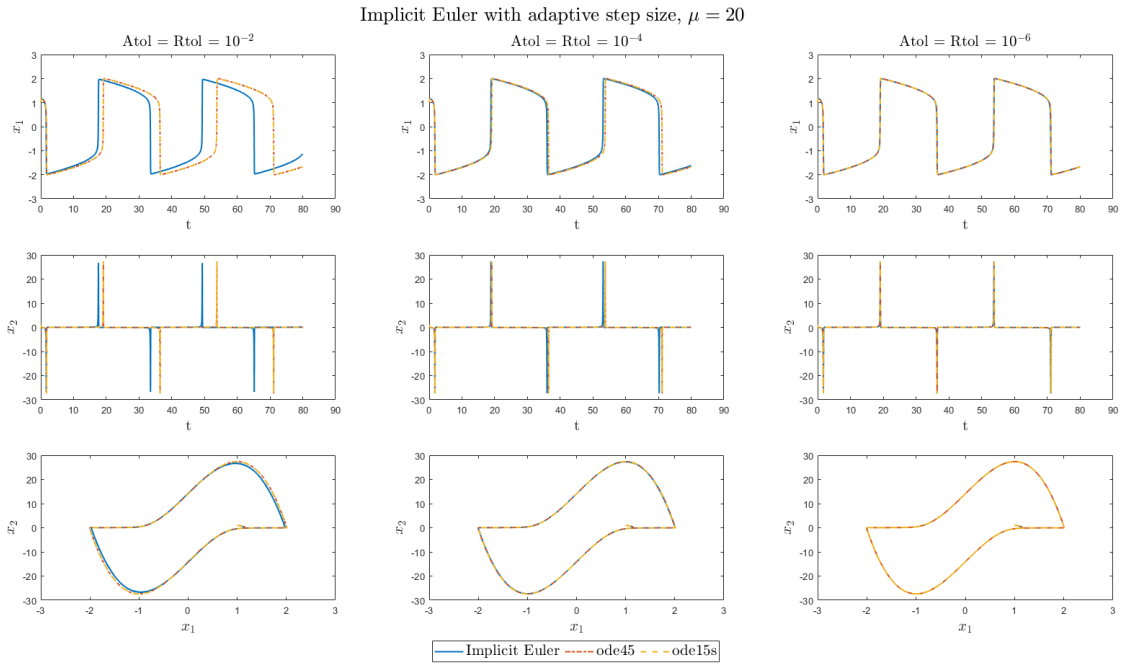

Figure 3.5: We have tested the implicit Euler with adaptive step size on the Van Der Pol system with $\mu = 20$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

For the acceptance of step sizes the picture is also very similar to the non-stiff system. We see that only for a tolerance of 10^{-2} steps are rejected and both for 10^{-4} and 10^{-6} no steps are rejected and the curve is nice and smooth.

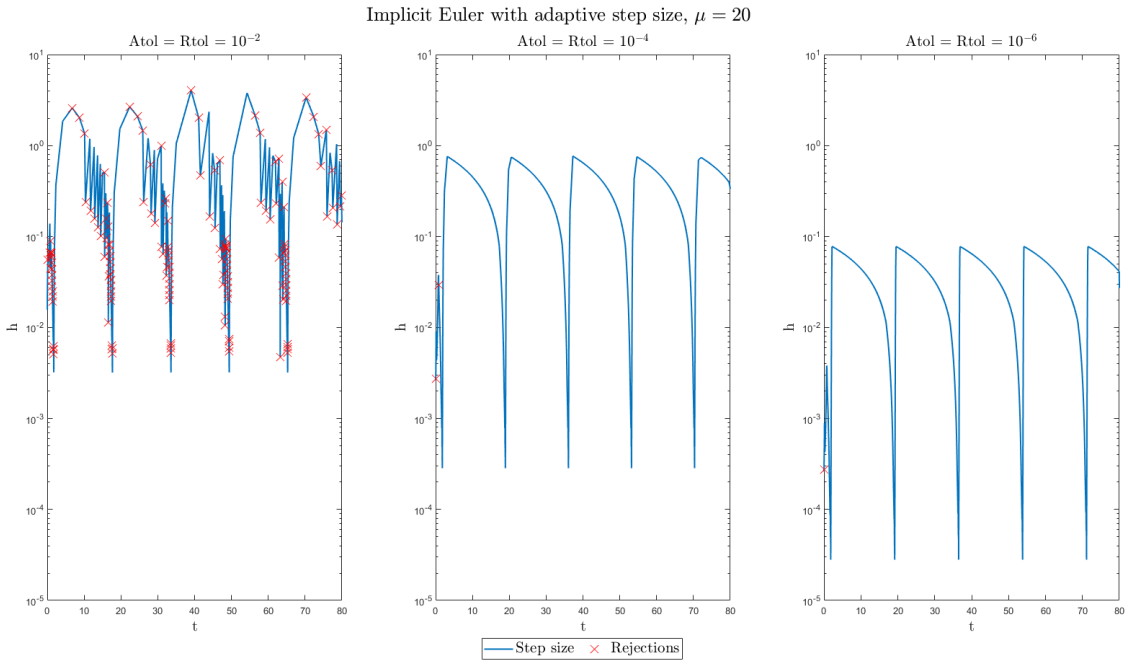

Figure 3.6: Here we have plotted the accepted step sizes for the test on the stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

Comparison

In this last section we will compare the fixed and adaptive implicit Euler. In table 3.1 we have compared them for the non-stiff system. We saw that to obtain accurate simulations we needed a step size at $h = 0.001$ and a tolerance at 10^{-6}. From the table we see that the adaptive is much slower than the fixed step size for these accuracies. The reason why we see so large a difference her compared to chapter 2 is that now we need to do a Newton solve each iteration. In the newton solve we need to call the gradient and the Jacobian each iteration and we need to inverse the Jacobian. This is very expensive and hence the many extra solves the adaptive method needs to do is punished very hard here.

	Fixed Step			Adaptive Step		
h/tol	0.1	0.01	0.001	10^{-2}	10^{-4}	10^{-6}
Time	0	0.031	0.437	0.031	0.391	1.578
Gradient calls	404	2711	24001	1788	8426	83492
Jacobian calls	403	2710	24000	1787	8425	83491
Steps	$1.2 \cdot 10^3$	$1.2 \cdot 10^4$	$1.2 \cdot 10^5$	192	1385	13915

Table 3.1: Comparison for the non-stiff system

In table 3.2 we compare the two methods on the stiff system. Here we saw that a step size of $h = 0.001$ was not enough to obtain an accurate simulation for the fixed step size approach. Hence we have included a step size of $h = 0.0001$. For the adaptive approach a tolerance of 10^{-6} was enough to gain an accurate simulation. We see that now the adaptive approach is much faster than the fixed approach. Because the fixed approach is wasting so much on the non-stiff regions of the stiff version of the Van Der Pol problem the adaptive method can justify solving the system three times per iteration.

	Fixed Step				Adaptive Step		
h/tol	0.1	0.01	0.001	0.0001	10^{-2}	10^{-4}	10^{-6}
Time	0.078	0.313	1.141	10.547	0.422	0.344	3.172
Gradient calls	3501	16588	99565	856210	10989	23452	223046
Jacobian calls	3500	16587	99564	856209	10988	23451	223045
Steps	$8 \cdot 10^3$	$8 \cdot 10^4$	$8 \cdot 10^5$	$8 \cdot 10^6$	644	3701	37174

Table 3.2: Comparison for the stiff system

CHAPTER 4

Solvers for SDEs

We now extend our scope to stochastic differential equations as the one given in 4.1

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

where $x \in \mathbb{R}^{n_x}$ and ω is a stochastic variable with dimension $n_w \cdot p_f$ 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 $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_{p_g}} \mapsto \mathbb{R}^{n_x \times n_w}$ (i.e. the result of g is a matrix of size $n_x \times n_w$).

4.1 Exercise 4.1

Before we dive into how we approximate the IVP, 4.1, we must understand how the stochastic differential equation differs from the non-stochastic differential equation. The last term in 4.1, $d\omega(t)$, is a diffusion term which introduces the stochasticity into the system. We will only work with diffusion distributed according the a Gaussian distribution which is also known as a Wiener process or a Brownian motion.

Definition 9. *A multivariate standard Brownian motion, over the time interval* $[0, T]$ *is a random variable* $\omega(t)$ *that depends continuously on* $t \in [0, T]$ *and satisfies the following conditions*

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

We have implemented a multivariate Brownian motion in Matlab and the code is given in Listing 4.1.

```
1 function [W,Tw,h] = brownian_motion(h,N,dim, seed)
2
3 if isnumeric(seed)
4   rng(seed);
5 end
6
7 W = sqrt(h)*randn(dim,N);
```

```
 8  W = cumsum(W,2);
 9  W = [zeros(dim,1) W];
10  Tw = 0:h:N;
11  end
```

Listing 4.1: Code for a multivariate Brownian motion.

We have in figure 4.1 plotted 3 paths of a 2D Brownian motion.

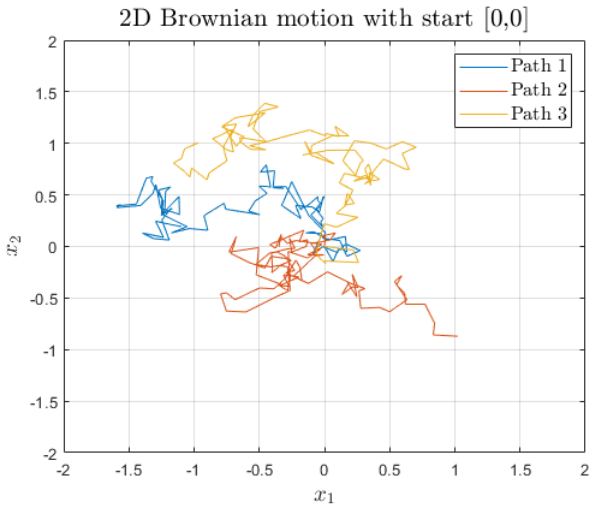

Figure 4.1: 3 paths of a 2D Brownian motion.

4.2 Exercise 4.2

We can now proceed approximating 4.1. We use Ito's integral to write up 4.1 as an integral equation on the interval $[t_0, t_0 + h]$ as we have done in the previous chapters.

$$\boldsymbol{x}(t_0+h) = \boldsymbol{x}_0 + \overbrace{\int_{t_0}^{t_0+h} f(t,\boldsymbol{x}(t),p_f)dt}^{\text{Riemann integral}} + \overbrace{\int_{t_0}^{t_0+h} g(t,\boldsymbol{x}(t)p_g)d\boldsymbol{\omega}(t)}^{\text{Ito integral}} \tag{4.2}$$

In chapter 1 we introduced rooted trees to design higher order Runge-Kutta methods for non-stochastic differential equations. For stochastic differential equations a similar theory exists. It is called colored rooted trees which we will not dive into here but if the reader is interested information can be found here [Röß10]. We will approximate 4.1 with rectangles based on the left end point as in chapter 2.

$$x(t_0+h) \approx x_1 = x_0 + hf(t_0,x_0,p_f) + \Delta\omega_h g(t_0,x_0,p_g) \tag{4.3}$$

where $\Delta\omega_h \sim N(0, hI)$. This method is called the Euler-Maruyama scheme or the explicit-explicit scheme. Because realizations of $\Delta\omega_h$ are random, each solution path will be different. Hence the order of a Runge-Kutta method for a stochastic differential equation is a bit more subtle than for a non-stochastic differential equation.

Convergence

We have two notions of convergence or order for stochastic differential equation. We will refer to a specific realization as x and to the random approximation scheme as X. First we have what is known as Strong Convergence. It is described in section 5 in [Rac17] and given as

$$\mathbb{E}\left[|x - X|\right] \leq Ch^{\gamma}. \tag{4.4}$$

and Weak Convergence given as

$$|\mathbb{E}\left[x\right] - \mathbb{E}\left[X\right]| \leq Ch^{\delta}. \tag{4.5}$$

We notice how different the two types of convergence are. Weak convergence means that the average path we simulate does h^{δ} good, where as strong convergence means that every path does h^{γ} good. For the explicit-explicit scheme the order of weak convergence is 1 and the order of strong convergence is $1/2$. The reason why the strong convergence is not 1, as the order was for the explicit Euler for non-stochastic differential equation, is that the stochastic 1st order Taylor expansion contains second order terms, [Rac17]. Ito's lemma corresponds to the stochastic 1st order Taylor expansion. A method which corrects for this is Milstein's method.

Implementation

We have implemented the explicit-explicit scheme with fixed step size in Matlab. The method can be called from a common solver interface with different controllers by

```
1 options = struct( 'paths' , 1, 'g', @VanPolDiffusion,'seed',1);
2 [X1,T1,function_calls1,hs1] = ...
      ODEsolver(@VanPol,[mu,sigma],h,t0,tend,x0, "Explicit-Explicit", ...
      options);
```

which calls the code given in

```
1 function [x, t,function_calls,hs] = SDEExplicitExplicit(x0, f, g, h, ...
      t0, T, params,paths,seed)
2     N = ceil((T-t0)/h);
3     hs = ones(1,N+1);
4     hs = h.*hs;
```

```
5       t = zeros(1,N+1);
6       dim = length(x0);
7       x = zeros(dim,N+1);
8       t(1) = t0;
9       x(:,1) = repmat(x0,1);
10      function_calls = 0;
11
12      W = brownian_motion(h,N+1,dim,seed);
13
14      for k=2:N+1
15          dW = W(:,k)-W(:,k-1);
16          dt = f(t(k-1), x(:,k-1), params);
17          dw = g(t(k-1), x(:,k-1), params);
18          x(:,k) = x(:,k-1) + dt*h + dw.*dW;
19          t(k) = t(k-1)+h;
20          function_calls = function_calls+2;
21      end
22      x = x';
23      t = t';
24
25  end
```

Listing 4.2: The Explicit-Explicit solver for SDEs

4.3 Exercise 4.3

We now use the right end point to approximate the Riemann integral in 4.2. This gives us the method

$$x(t_0 + h) \approx x_1 = x_0 + hf(t_1, x_1, p_f) + \Delta\omega_h g(t_0, x_0, p_g) \tag{4.6}$$

which for the deterministic dynamic corresponds to the implicit Euler method which we explored in chapter 3. This method is called the implicit-explicit scheme. It will not give us better convergence properties but if the deterministic dynamic shows stiff behaviour this method will be much better at handling it than the explicit-explicit scheme.

Implementation

We have implemented the implicit-explicit scheme with fixed step size in Matlab. The only implementation specific change that has not been discussed yet and is different from a normal implicit Euler method is the residual function. We have a stochastic contribution which will be constant through out the newton iteration and is given as

$$R(k) = k - x_0 - hf(t_0 + h, k, p_f) - g(t_0, x_0, p_g)\Delta\omega_h. \tag{4.7}$$

Besides that the theory regarding the newton iteration is identical to the one described in chapter 3. The method can be called from a common solver interface by

```
options = struct( 'paths' , 1, 'g', @VanPolDiffusion, 'Jac', ...
    @VanPolJac, 'seed', 1);
[X1,T1,function_calls1,hs1] = ...
    ODEsolver(@VanPol,[mu,sigma],h,t0,tend,x0, "Implicit-Explicit", ...
    options);
```

which calls the code given in

```
function [x, t,function_calls,hs] = SDEImplicitExplicit(x0, f,jac, g, ...
    h, t0, T, param,paths, newtonTolerance, newtonMaxiterations,seed)
    N = ceil((T-t0)/h);
    hs = ones(1,N+1);
    hs = h.*hs;
    t = zeros(1,N+1);
    dim = length(x0);
    x = zeros(dim,N+1);
    t(1) = t0;
    x(:,1) = repmat(x0,1);
    function_calls = 0;

    W = brownian_motion(h,N+1,dim,seed);
    dt = f(t0, x(:,1), param);
    for k=2:N+1
        dw = g(t(k-1), x(:,k-1), param);
        dW = W(:,k)-W(:,k-1);
        Rterm = x(:,k-1) + dw.*dW;
        t(k) = t(k-1)+h;
        xguess = Rterm + dt*h;
        [x(:,k), dt, function_calls] = newtonsMethod(f, jac, ...
            t(:,k-1), Rterm, h, xguess, newtonTolerance, ...
            newtonMaxiterations, param, function_calls);
        function_calls = function_calls+2;
    end
    x = x';
    t = t';

end
```

Listing 4.3: The Implicit-Explicit solver for SDEs

where the Implicit-Explicit solver uses the same Newton method as the one given in listings 3.2.

4.4 Exercise 4.4

We will now test the explicit-explicit and the implicit-explicit methods on the stochastic versions of the Van Der Pol system. They are given on slide 16 from the slideshow "Lecture06B" given in week 6. We have one version where the diffusion is state

independent,

$$\begin{aligned} d\boldsymbol{x}_1(t) &= \boldsymbol{x}_2(t)dt \\ d\boldsymbol{x}_2(t) &= \left[\mu\left(1-\boldsymbol{x}_1(t)^2\right)\boldsymbol{x}_2(t) - \boldsymbol{x}_1(t)\right]dt + \sigma d\omega(t) \end{aligned} \tag{4.8}$$

and a version where the diffusion is state dependent

$$\begin{aligned} d\boldsymbol{x}_1(t) &= \boldsymbol{x}_2(t)dt \\ d\boldsymbol{x}_2(t) &= \left[\mu\left(1-\boldsymbol{x}_1(t)^2\right)\boldsymbol{x}_2(t) - \boldsymbol{x}_1(t)\right]dt + \sigma\left(1+\boldsymbol{x}_1(t)^2\right)d\boldsymbol{\omega}(t) \end{aligned} \tag{4.9}$$

We will test both the methods with independent and dependent diffusion. The variance parameter σ will be set to 1 and 2 and we will test on a non stiff system where $\mu = 3$ and a stiff system where $\mu = 20$. Further more we will simulate 10 paths in each test. We will initialize all simulations at $x_0 = [0.5, 0.5]$ and integrate from time $t \in [0, 3\mu]$.

Explicit-Explicit

In figure 4.2 we see the simulations for the state independent diffusion. We see that the stochasticity is much more profound in the non-stiff system. The reason is that the values which the deterministic system takes are in the same range as the diffusion. Hence the observed dynamic will seem very disturbed. For the stiff system the values which the deterministic part generates are much larger than the diffusion and hence they will not be able to pollute the dynamics as much relative to the non-stiff system.

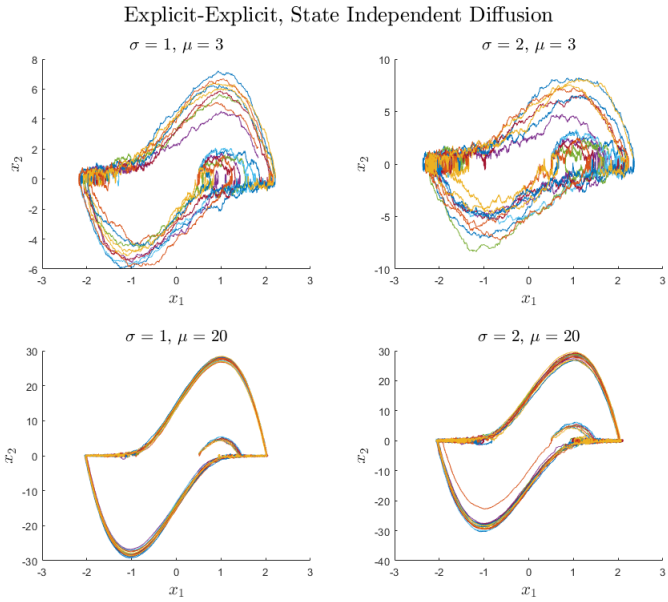

Figure 4.2: State independent simulations for the Explicit-Explicit method

In figure 4.3 we see the state dependent diffusion. Here the non-stiff system is even more polluted by the noise and in the stiff parts of the system, x_1 takes the largest values and hence the noise is also largest in these regions. For the stiff system we again do not see as large a pollution. This can seem weird because now the diffusion should be scaled by the magnitude of the system. The reason why we do not see as huge a disturbance is that the diffusion is dependent on x_1 and added to state x_2. State x_1 takes on quite small values compared to x_2 when the system is stiff and hence the disturbance remains quite small for stiff systems even when the diffusion is state dependent.

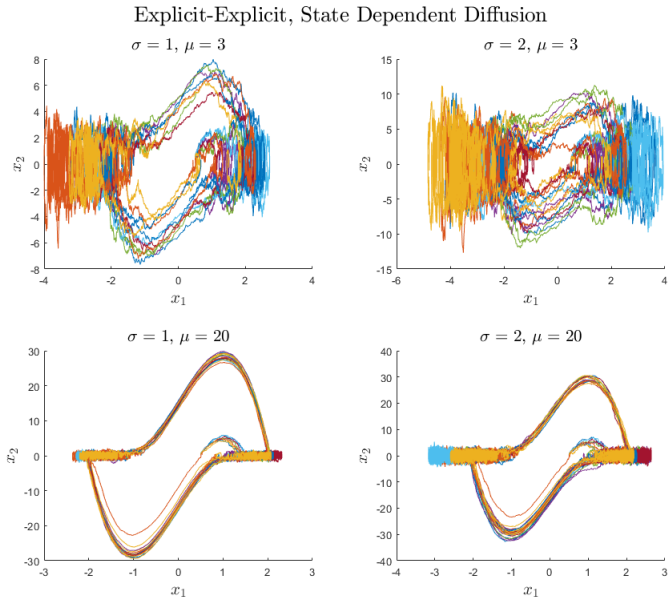

Figure 4.3: State Dependent simulations for the Explicit-Explicit method

Implicit-Explicit

The state independent simulations are shown in figure 4.4 and the state dependent are shown in figure 4.5. We see that there is not really any detectable difference between the Implicit-Explicit and the Explicit-Explicit simulations. We see that same things both for the dependent and the independent diffusion.

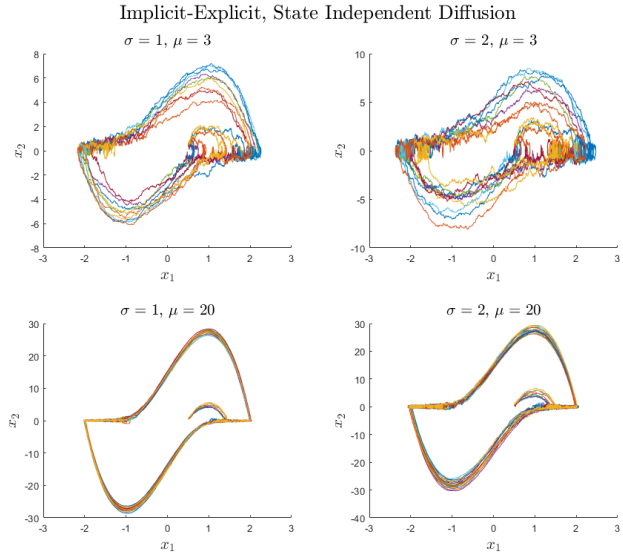

Figure 4.4: State independent simulations for the Implicit-Explicit method

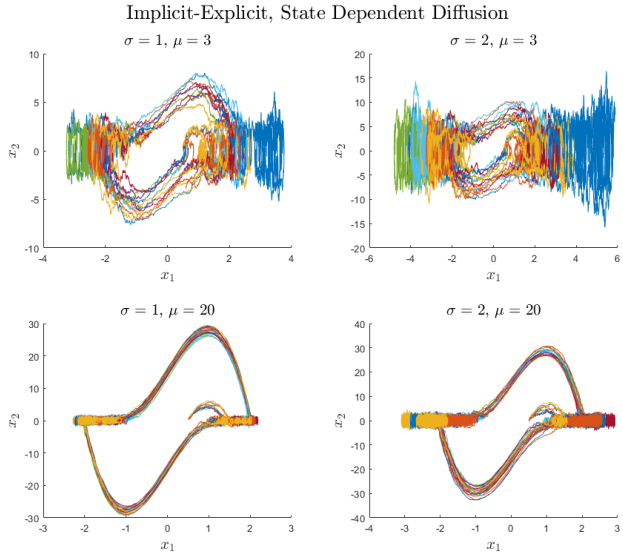

Figure 4.5: State Dependent simulations for the Implicit-Explicit method

Strong and Weak Convergence

Lastly we will take a look at the reduced order compared to the deterministic versions of the methods. Both explicit-explicit and implicit-explicit have a strong order of a $1/2$ while both explicit and implicit Euler have an order of 1. We can see this is figure 4.6 where the stochastic versions diverge both for step sizes of $h = 0.1$ and $h = 0.01$ while the deterministic version converge with a step size of $h = 0.01$.

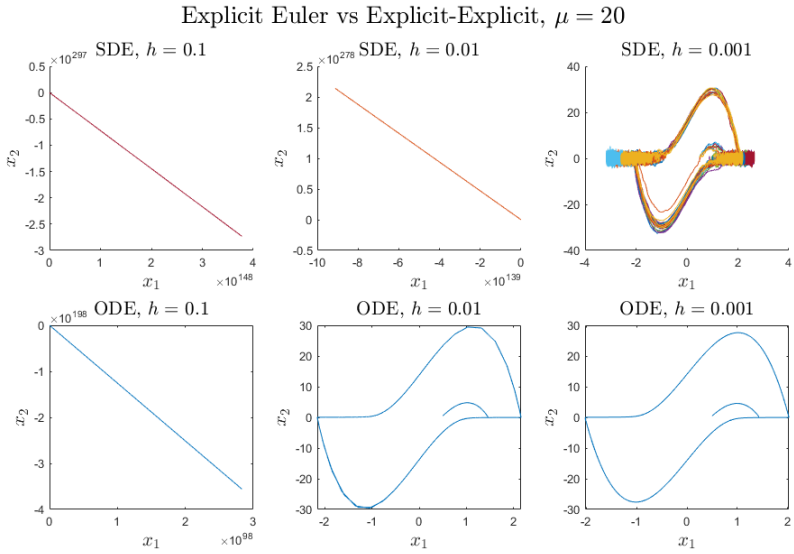

Figure 4.6: Illustration of the reduced order compared to the deterministic version.

CHAPTER 5

Classical Runge-Kutta

We now move back to non-stochastic dynamics and consider again a IVP of the form

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

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

5.1 Exercise 5.1

We again write 5.1 on the interval $t \in [t_0, t_0 + h]$ as as the integral equation

$$x(t_0 + h) = x_0 + \int_{t_0}^{t_0+h} f(t, x(t), p)\mathrm{dt}. \tag{5.2}$$

In the previous chapters we have used simple left and right Riemann sums to approximate the integral in 5.2. A more accurate method to approximate integrals is the 3 point Simpsons rule.

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \tag{5.3}$$

If we use this to approximate the integral in 5.2 we would obtain the classical Runge Kutta method. A more modern approach to designing Runge-Kutta methods would be to use rooted trees and Theorem 2, 3 and 4 in chapter 1. This would give the following equations to solve

1st Order	2nd Order	3rd Order	4th Order
$1 = \sum_{i=1}^{s} b_i$	$\frac{1}{2} = \sum_i b_i c_i$	$\frac{1}{3} = \sum_i b_i c_i^2$	$\frac{1}{4} = \sum_i b_i c_i^3$
		$\frac{1}{6} = \sum_{i,j} b_i a_{ij} c_j$	$\frac{1}{8} = \sum_{i,j} b_i c_i a_{ij} c_j$
			$\frac{1}{12} = \sum_{i,j} b_i a_{ij} c_j^2$
			$\frac{1}{24} = \sum_{i,j,k} b_i a_{ij} a_{jk} c_k$

Table 5.1: Order conditions for Theorem 4 up until 4th order.

By solving these we would cover up to 4th order of the Taylor expansion of the LHS in 5.2. This would also give the classical Runge-Kutta method.

The algorithm

Either way we obtain the following algorithm

Classical Runge-Kutta method with fixed step size:

1. Choose step size $h = (T - t_0)/N$.
2. for $j = 0, 1, \ldots, N-1$ do

$$\begin{aligned}
\mathbf{k}_1 &= f(t_j, \mathbf{x}_j) \\
\mathbf{k}_2 &= f(t_j + h/2, \mathbf{x}_j + h/2\mathbf{k}_1) \\
\mathbf{k}_3 &= f(t_j + h/2, \mathbf{x}_j + h/2\mathbf{k}_2) \\
\mathbf{k}_4 &= f(t_j + h, \mathbf{x}_j + h\mathbf{k}_3) \\
\mathbf{x}_{j+1} &= \mathbf{x}_j + \frac{h}{6}\mathbf{k}_1 + \frac{2h}{6}\mathbf{k}_2 + \frac{2h}{6}\mathbf{k}_3 + \frac{h}{6}\mathbf{k}_4 \\
t_{j+1} &= t_j + h
\end{aligned}$$

end for

The method is given as a Buthcer tableau in Chapter 1, equation 1.15 and restated here.

Classical Runge-Kutta:

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	2/6	2/6	1/6

(5.4)

5.2 Exercise 5.2

We have implemented a generic explicit Runge-Kutta with fixed step size code given in listings 2.1 in chapter 2. It can be called from the common solver interface by

```
1 options = struct('step_control',false, 'initialStepSize', false);
2 [X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
      "RK4",options);
```

5.3 Exercise 5.3

We have implemented the classical Runge-Kutta method with adaptive time step and error estimation using step doubling in Matlab. The theory is exactly the same as in chapter 2. Hence we refer to this chapter for theoretical considerations. The method can be called from a common solver interface by

```
1 options = struct('step_control',true, 'initialStepSize', ...
      true,'control_type', "PID");
2 [X1,T1,function_calls1,hs1,rs1] = ODEsolver(@VanPol,[mu],h,t0,tend,x0, ...
      "RK4",options);
```

which calls the code given in appendix A.1.

5.4 Exercise 5.4 and 5.5

We will test the classical Runge-Kutta method on the deterministic Van Der Pol problem which was introduced both in chapter 2 and 3. We will also here test on a non-stiff version where $\mu = 3$ and a stiff version where $\mu = 20$. All simulations will be initialized at $x_0 = [1.0, 1.0]$ and runned on the interval $t \in [0, 4\mu]$.

Fixed step size

We see from figure 5.1 that for the non-stiff system the Classical Runge-Kutta has no error compared to the in build Matlab solvers even for the largest tested step size. We clearly see the much higher order of the classical Runge-Kutta compared to the Euler methods which only showed no error for the smallest step size.

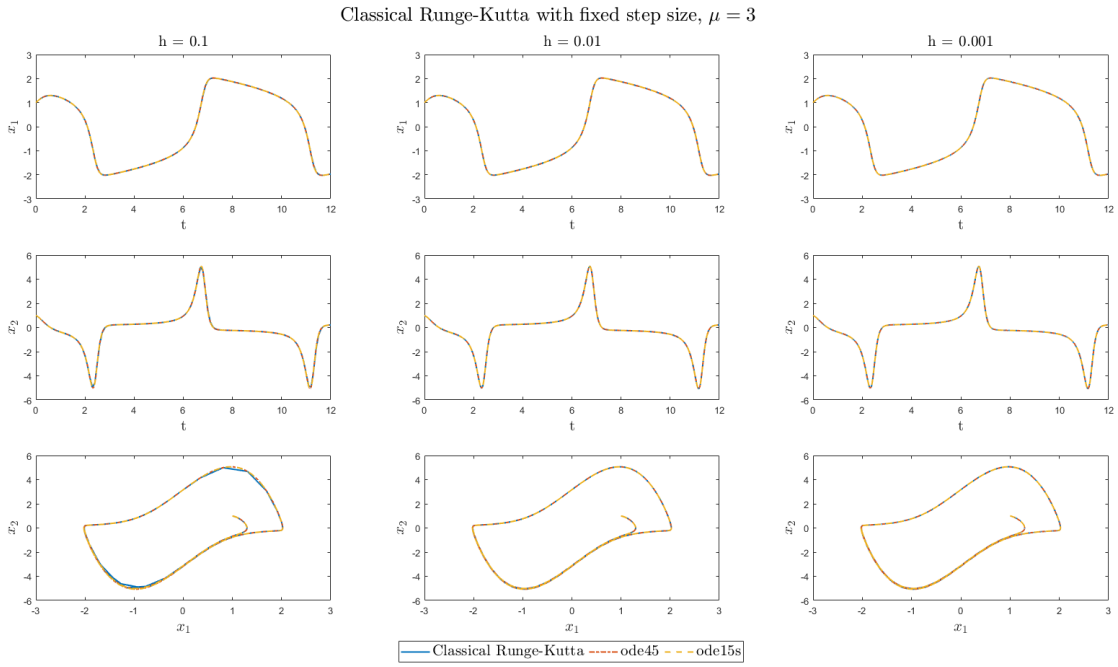

Figure 5.1: We have tested the classical Runge-Kutta with fixed step size on the Van Der Pol system with $\mu = 3$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

In figure 5.3 we see the simulations for the fixed step size on the stiff system. Here we also see another characteristic of a higher order method. Because we have covered the 4th Taylor expansion we will get quite dramatic behaviour the second we get out of the stability region. Hence we also see that no error is made for $h = 0.01$ and we diverge when $h = 0.1$.

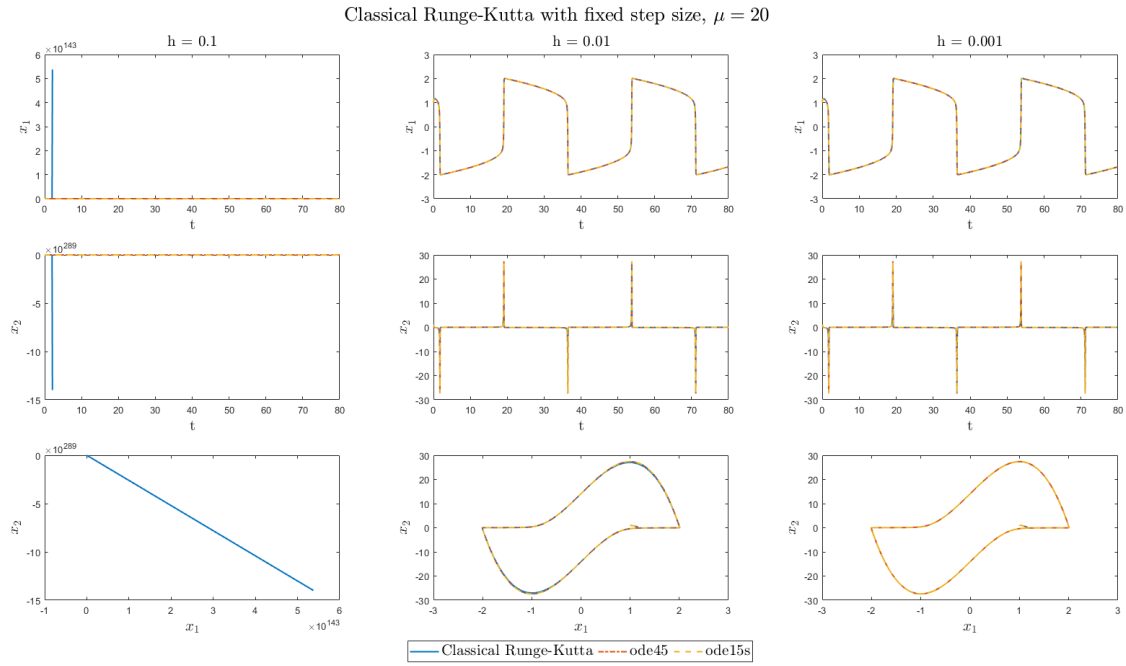

Figure 5.2: We have tested the classical Runge-Kutta with fixed step size on the Van Der Pol system with $\mu = 20$. This we have done for different step sizes against the Matlab native solvers, `ode45` and `ode15s`.

Adaptive step size

We now proceed with the adaptive version of the classical Runge-Kutta method. We will again not test the different controllers here but rely on the results from chapter 2. Hence we will use the PID controller introduced in chapter 2. In figure 5.3 we see the simulations for the non-stiff version of the Van Der Pol sysmte. We observe that for all tolerances no error is made. At first glance one could maybe think that an error is made for $Atol = Rtol = 10^{-2}$, but this is actually not an error. It is just the step size controller realizing it can take very large steps and hence when Matlab plots it is seems jagged but all the points are on the true attractor which Matlab's own solvers also produce.

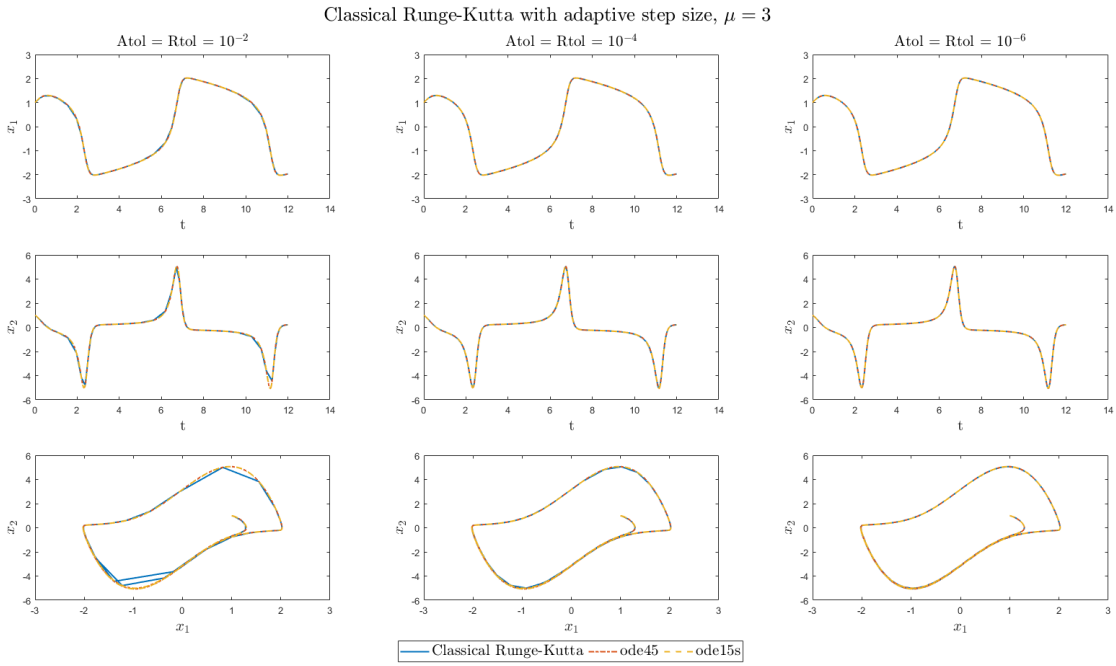

Figure 5.3: We have tested the Classical Runge-Kutta with adaptive step size on the Van Der Pol system with $\mu = 3$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 5.4 we see that every time we approach a spike in the graphs in the second row of figure 5.3 we have some rejections. This is because our controller is not fast enough in reducing the step size and hence some rejections are produced here.

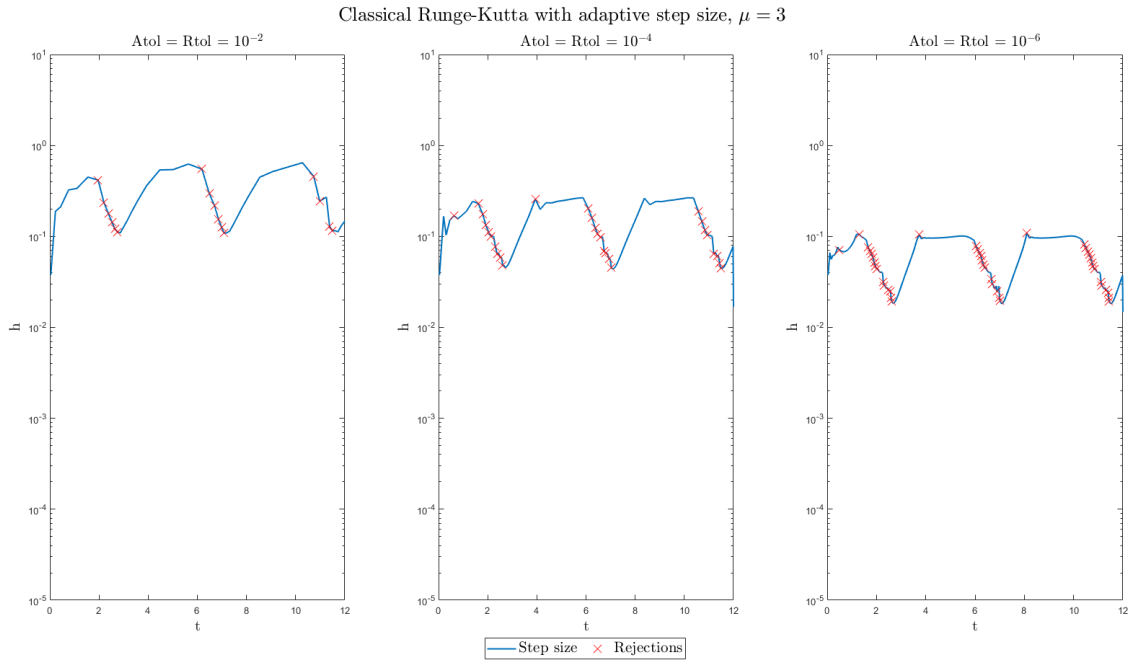

Figure 5.4: Here we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

We now proceed and test the adaptive version on the stiff system. In figure 5.5 we see the stiff system simulated for different tolerances and again we see no error being produced for all tolerances.

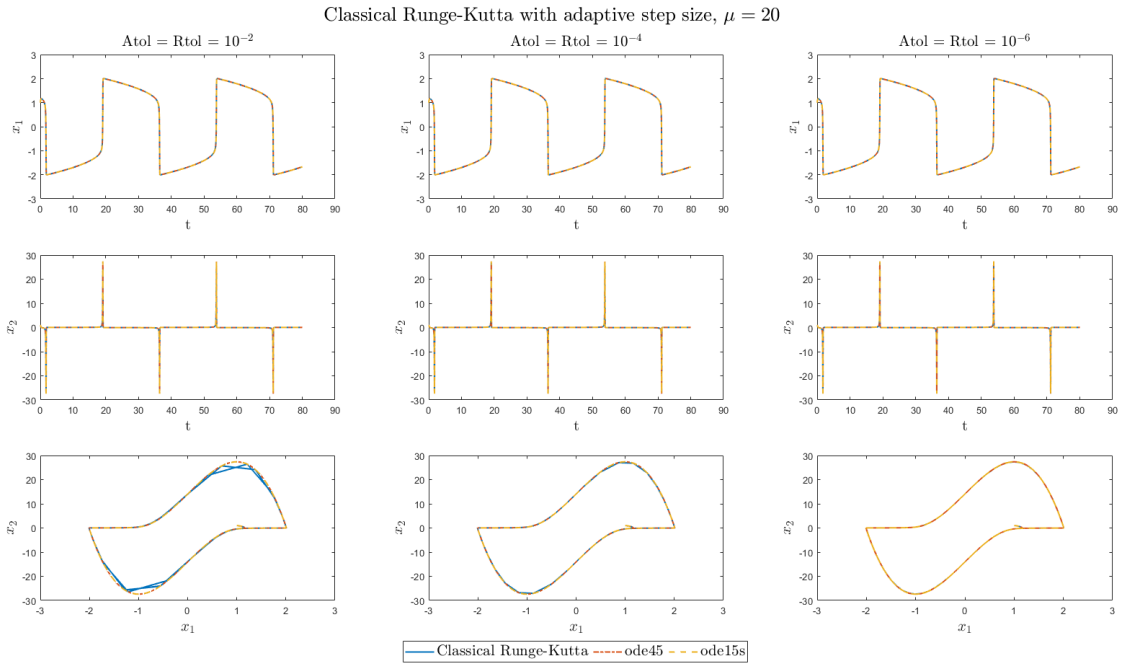

Figure 5.5: We have tested the classical Runge-Kutta with adaptive step size on the Van Der Pol system with $\mu = 20$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 5.6 we see that for a tolerance of 10^{-6} the picture is the same as for the non stiff system where we had rejections when we approached a spike. For tolerances of 10^{-2} and 10^{-4} we though see one more thing here. We see that when we leave a spike the controller tries to increase the step size to fast and hence get rejected in the attempt.

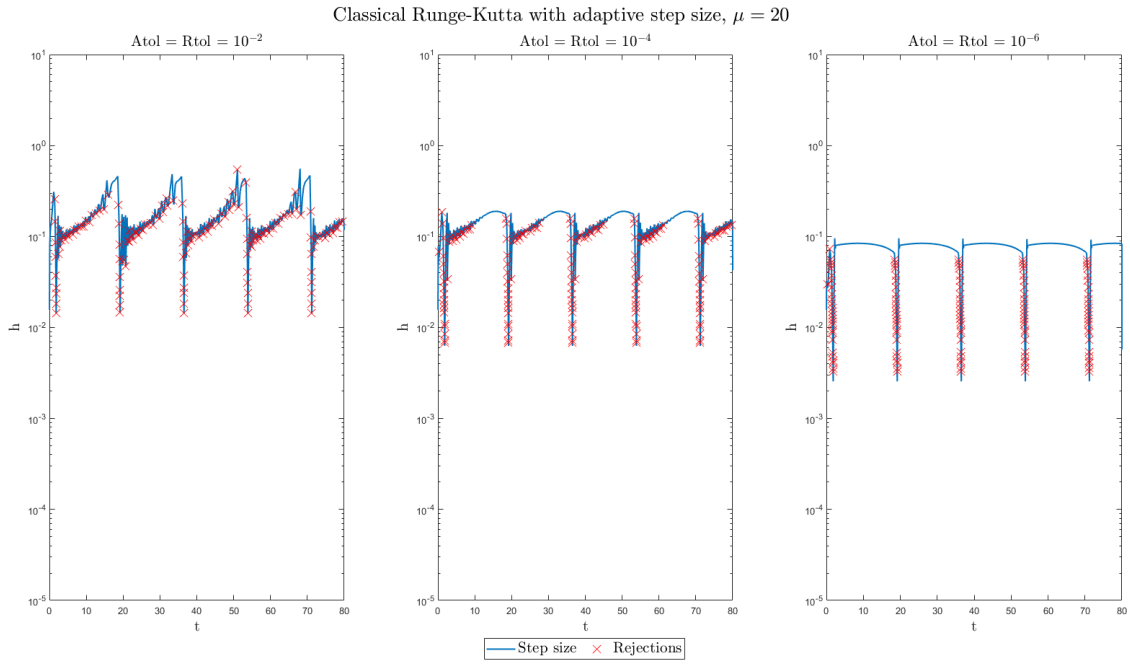

Figure 5.6: Here we have plotted the accepted step sizes for the test on the stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

Comparison

For the non-stiff system we saw that for the fixed step size a step size of $h = 0.1$ was actually fine and similarly for the adaptive method a tolerance of 10^{-2} was good enough. Hence we see that the fixed step size solves the system so fast that Matlab do not register a time. The adaptive method is not slow either and one could probably say it did not matter that much in this case. We tough get the security of not diverging if we use the adaptive method.

	Fixed Step			Adaptive Step		
h/tol	0.1	0.01	0.001	10^{-2}	10^{-4}	10^{-6}
Time	0	0.047	0.578	0.031	0.031	0.078
Function calls	$4.8 \cdot 10^3$	$4.8 \cdot 10^4$	$4.8 \cdot 10^5$	758	1562	3374
Steps	$1.2 \cdot 10^3$	$1.2 \cdot 10^4$	$1.2 \cdot 10^5$	63	130	281

Table 5.2: Comparison for the non-stiff system

For the stiff version of the Van Der Pol system we saw that the fixed step size needed a maximum step size of 0.01 and the adaptive method was still fine with a tolerance of 10^{-2}. Hence we see from table 5.3 that the adaptive method is 10 times faster than the fixed method.

	Fixed Step			Adaptive Step		
h/tol	0.1	0.01	0.001	10^{-2}	10^{-4}	10^{-6}
Time	0.031	0.578	3.359	0.047	0.078	0.125
Function calls	$3.2 \cdot 10^4$	$3.2 \cdot 10^5$	$3.2 \cdot 10^6$	9194	11306	18626
Steps	$8 \cdot 10^3$	$8 \cdot 10^4$	$8 \cdot 10^5$	766	942	1552

Table 5.3: Comparison for the stiff system

CHAPTER 6

Dormand-Prince 5(4)

We will again consider an IVP of the form

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

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

6.1 Exercise 6.1

In all the previous chapters which considered non-stochastic dynamics an adaptive step controller was implemented. They all relied on step doubling to obtain a method of a higher order to estimate the local error. This is a very computationally heavy approach and hence we would like some method which could give us the approximation of the local error without the extra computational burden. If we focus on explicit methods these kind of methods has a Butcher tableau of the following form

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \\ & \hat{b}_1 & \hat{b}_2 & \cdots & \hat{b}_{s-1} & \hat{b}_s \end{array} \tag{6.2}$$

The Dormand-Prince 5(4) is such an embedded explicit Runge-Kutta method with the first set of b's being of order 5 and the second set of order 4. To find the coefficients we can now use rooted trees which gives the following set of equations.

1st Order	2nd Order	3rd Order	4th Order
$1 = \sum_{i=1}^{s} \hat{b}_i$	$\frac{1}{2} = \sum_i \hat{b}_i c_i$	$\frac{1}{3} = \sum_i \hat{b}_i c_i^2$	$\frac{1}{4} = \sum_i \hat{b}_i c_i^3$
		$\frac{1}{6} = \sum_{i,j} \hat{b}_i a_{ij} c_j$	$\frac{1}{8} = \sum_{i,j} \hat{b}_i c_i a_{ij} c_j$
			$\frac{1}{12} = \sum_{i,j} \hat{b}_i a_{ij} c_j^2$
			$\frac{1}{24} = \sum_{i,j,k} \hat{b}_i a_{ij} a_{jk} c_k$

Table 6.1: Order conditions for the embedded 4th order method.

1st Order	2nd Order	3rd Order	4th Order	5th Order
$1 = \sum_{i=1}^{s} b_i$	$\frac{1}{2} = \sum_i b_i c_i$	$\frac{1}{3} = \sum_i b_i c_i^2$	$\frac{1}{4} = \sum_i b_i c_i^3$	$\frac{1}{5} = \sum_i b_i c_i^4$
		$\frac{1}{6} = \sum_{i,j} b_i a_{ij} c_j$	$\frac{1}{8} = \sum_{i,j} b_i c_i a_{ij} c_j$	$\frac{1}{10} = \sum_{i,j} b_i c_i^2 a_{ij} c_j$
			$\frac{1}{12} = \sum_{i,j} b_i a_{ij} c_j^2$	$\frac{1}{15} = \sum_{i,j} b_i c_i a_{ij} c_j^2$
			$\frac{1}{24} = \sum_{i,j,k} b_i a_{ij} a_{jk} c_k$	$\frac{1}{30} = \sum_{i,j,k} b_i c_i a_{ij} a_{jk} c_k$
				$\frac{1}{20} = \sum_i b_i \left(\sum_j a_{ij} c_j \right)^2$
				$\frac{1}{20} = \sum_{i,j} b_i a_{ij} c_j^3$
				$\frac{1}{40} = \sum_{i,j,k} b_i a_{ij} c_j a_{jk} c_k$
				$\frac{1}{60} = \sum_{i,j,k} b_i a_{ij} a_{jk} c_k^2$
				$\frac{1}{120} = \sum_{i,j,k,l} b_i a_{ij} a_{jk} a_{kl} c_l$

Table 6.2: Order conditions for the 5th order method.

Solving the equations in the above two tables will reveal the Dormand-Prince 5(4) which has the following Buthcer tableau.

Dormand-Prince 5(4):

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	−56/15	32/9				
8/9	$\frac{19372}{6561}$	$\frac{-25360}{2187}$	$\frac{64448}{6561}$	$\frac{-212}{729}$			
1	$\frac{9017}{3168}$	$\frac{-355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$\frac{-5103}{18656}$		
1	$\frac{35}{381}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	
b	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$\frac{-2187}{6784}$	$\frac{11}{84}$	0
$\hat{b}$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$\frac{-92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

(6.3)

We have written a pseudo code for the Dormand-Prince 5(4) with adaptive step size below.

Dormand-Prince 5(4):

1. Choose time horizon T.
2. while $t < T$ do

   a) Calculate

$$\mathbf{k}_s = f\left(t_0 + c_s h, \mathbf{x}_0 + h\sum_{\ell=1}^{s-1} a_{s\ell}\mathbf{k}_\ell\right), \quad s = 1\dots 7$$
$$\tilde{\mathbf{x}}_{j+1} = \mathbf{x}_j + h\sum_{i=1}^{7} b_i\mathbf{k}_i$$
$$\hat{\mathbf{x}}_{j+1} = \mathbf{x}_j + h\sum_{i=1}^{7} \hat{b}_i\mathbf{k}_i$$
$$\mathbf{e}_{j+1} = |\tilde{\mathbf{x}}_{j+1} - \hat{\mathbf{x}}_{j+1}|$$
$$sc_{i+1} = max_{k\in\{1,\dots,n\}}\left(Atol, |\tilde{\mathbf{x}}_{j+1}|Rtol\right)$$
$$r_{i+1} = \frac{e_{i+1}}{sc_{i+1}}$$

   b) if $r_{i+1} \leq 1$

$$\mathbf{x}_{i+1} = \tilde{\mathbf{x}}_{j+1}$$
$$t_{i+1} = t_i + h$$
$$h = controlRule(h, r_{i-1}, r_{i,i+1})$$
$$r_i = r_{i+1}$$
$$r_{i-1} = r_i$$

   else

$$h = controlRule(h, r_{i-1}, r_{i,i+1})$$

   end while

6.2 Exercise 6.2

We have implemented the Dormand-Prince 5(4) method with adaptive time step and embedded error estimation in Matlab. The only change in the adaptive time stepping is that now the error is estimated by the embedded method. Besides that the theory is exactly the same as in chapter 2 and hence we refer the this chapter for any theoretical explanations. The method can be called from a common solver interface by

```
1 options = struct('step_control',true, ...
      'initialStepSize',true,'control_type', ...
      "PID",'Rtol',Rtol,'Atol',Atol);
2 [X3,T3,function_calls12,hs3,rs12] = ...
      ODEsolver(@VanPol,[mu],h,t0,tend,x0, "DOPRI54", options);
```

which calls the code given in appendix A.2

6.3 Exercise 6.3

Before testing the Dormand-Prince 5(4) on non-linear systems we will test it on the test equation which we saw in chapter 1. We know also from chapter 1 that the error for the test equation of a Runge-Kutta method can be described by the stability function given in 1.16 and here restated for good measure.

$$R(\lambda h) = 1 + \lambda h b^T (I - \lambda h A)^{-1} \mathbf{1}_s \tag{6.4}$$

We now have the stability region given by

$$\mathcal{S} := \{h\lambda \in \mathbb{C} : |R(h\lambda)| \leq 1\}. \tag{6.5}$$

which for the forward integrating method in the Dormand-Prince 5(4) gives the stability region

$$1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \frac{(h\lambda)^4}{24} + \frac{(h\lambda)^5}{120} + \frac{(h\lambda)^6}{600}. \tag{6.6}$$

We have in figure 6.1 plotted the stability region for the 5th order method.

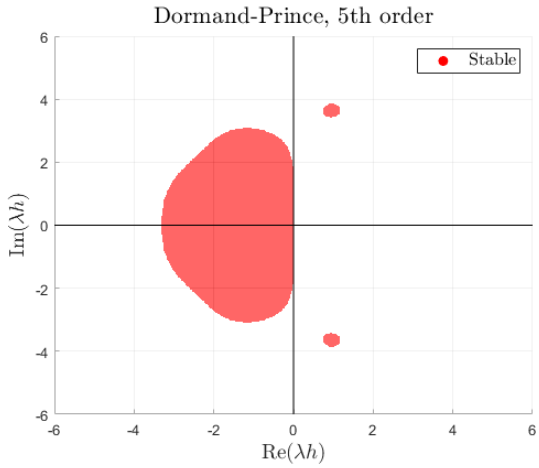

Figure 6.1: Stability region of the 5th order forward integrating method in Dormand-Prince 5(4).

First we test only the forward integrating method without any error control to see if the error made by the method matches the analytical error as we also did for the explicit Euler, implicit Euler and classical Runge-Kutta in chapter 1.

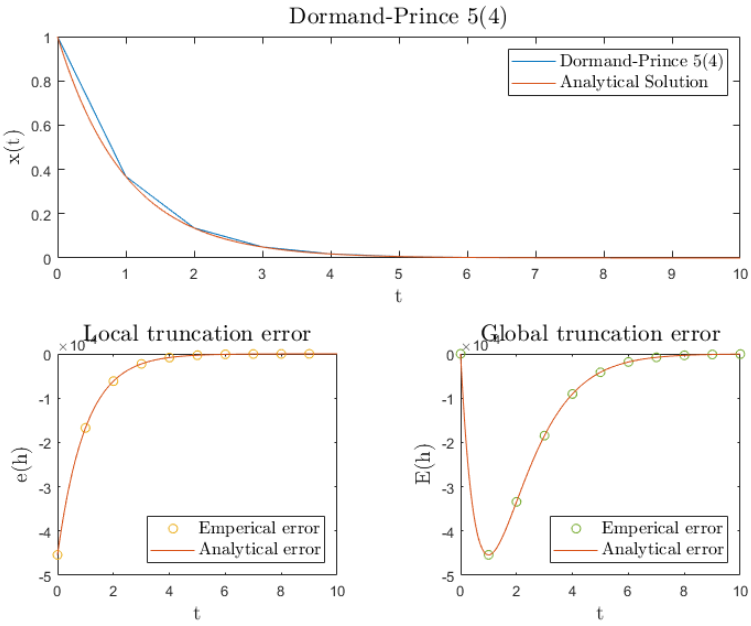

Figure 6.2: Only the forward integrating method without any error control tested on the test equation.

In figure 6.2 we have used a step size $h = 1$ on the interval $t \in [0, 10]$. We see that the observed error matches the analytical error which is calculated as in section 1.3. We now try to compare only the forward integrating method with the full Dormand-Prince 5(4) with error control.

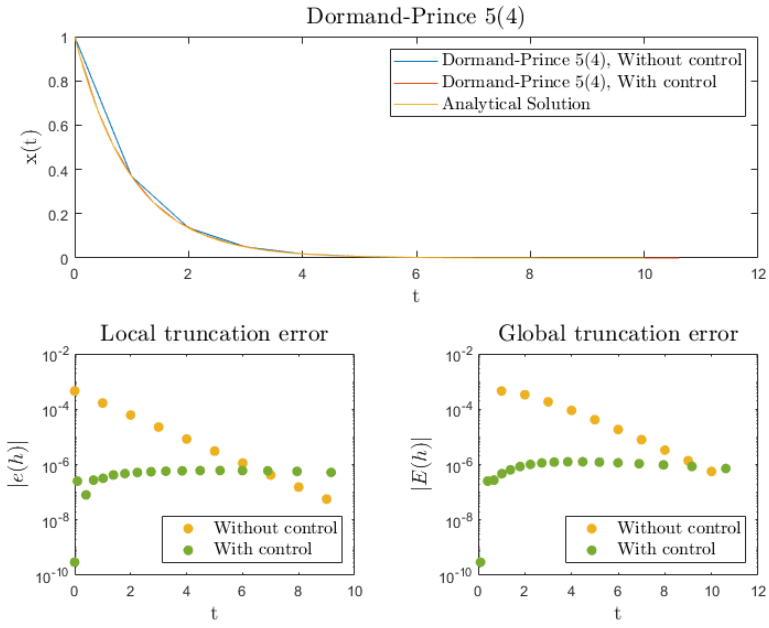

Figure 6.3: The Dormand-Prince 5(4) with and without error control tested on the Test equation.

We see in figure 6.3 that with error control the local and global error remains quite constant through out the integration indicating that our solver is adjusting the step size such that the largest possible steps are taken mean while the error is kept on an acceptable level. Lastly we will illustrate what happens on the boarder of the stability region. The boarder of the stability region is at $h_{lim} = 3.30656789263$ and in figure 6.4 we have tried to plot what happens just above that, on that limit and just under.

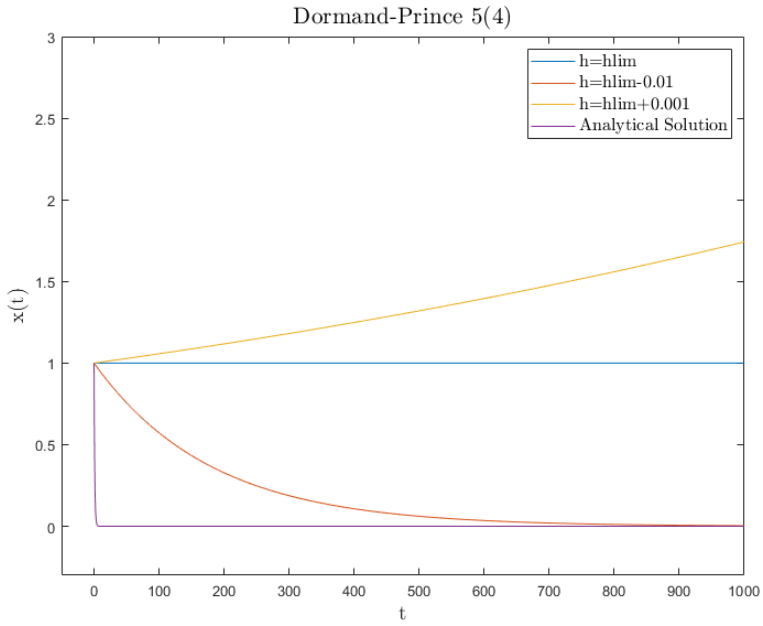

Figure 6.4: The Dormand-Prince 5(4) forward integrating method on the limit of the stability region.

We see that just within we converge to the true solution, on the limit we do not converge but neither we diverge and just above we diverge.

6.4 Exercise 6.4 and 6.6

We now proceed and test on the Van Der Pol problem. The controller in the Dormand-Prince 5(4) will again be the PID controller introduced in chapter 2. We will only test an adaptive method where the error estimation is based on the embedded method in the Dormand Prince 5(4) method. In figure 6.5 we see the simulations for the non stiff system. As for the classical Runge-Kutta method no error is made for all tolerances.

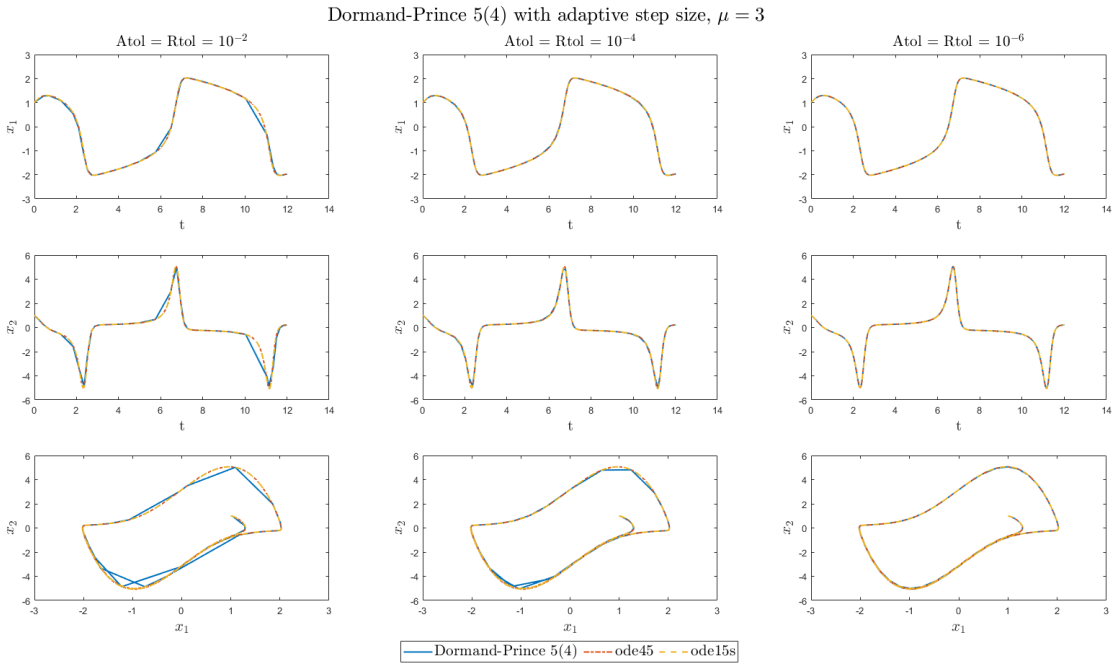

Figure 6.5: We have tested the Dormand-Prince 5(4) with adaptive step size on the Van Der Pol system with $\mu = 3$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 6.6 we see the same picture as for the Classical Runge Kutta. When we approach a spike some rejections are made.

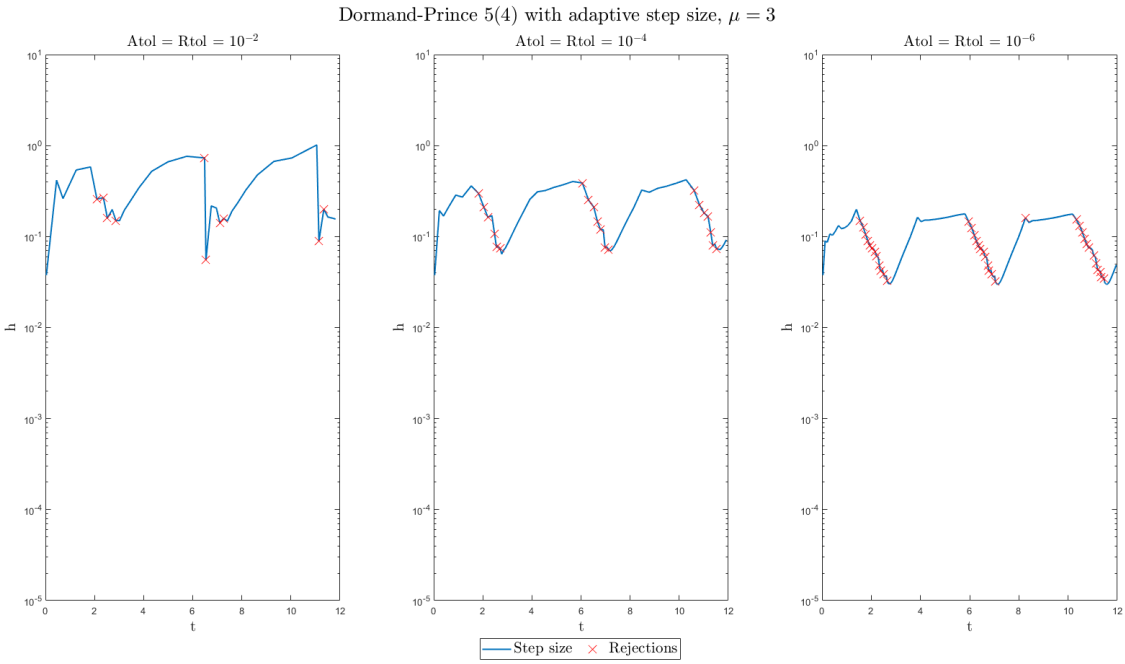

Figure 6.6: Here we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

In figure 6.7 we see that simulations for the stiff version of the Van Der Pol system. Here we see that for a tolerance of 10^{-2}, a small error is being made compared to the in build solvers. This is different compared to the classical Runge Kutta. A reason could maybe be that now we estimate the error with a 4th order method and stepping forward with a 5th order method. In the classical Runge-Kutta we was stepping forward with a 4th order method and due to the step doubling we were checking the error with a 5th order method. This could maybe be a bit more robust.

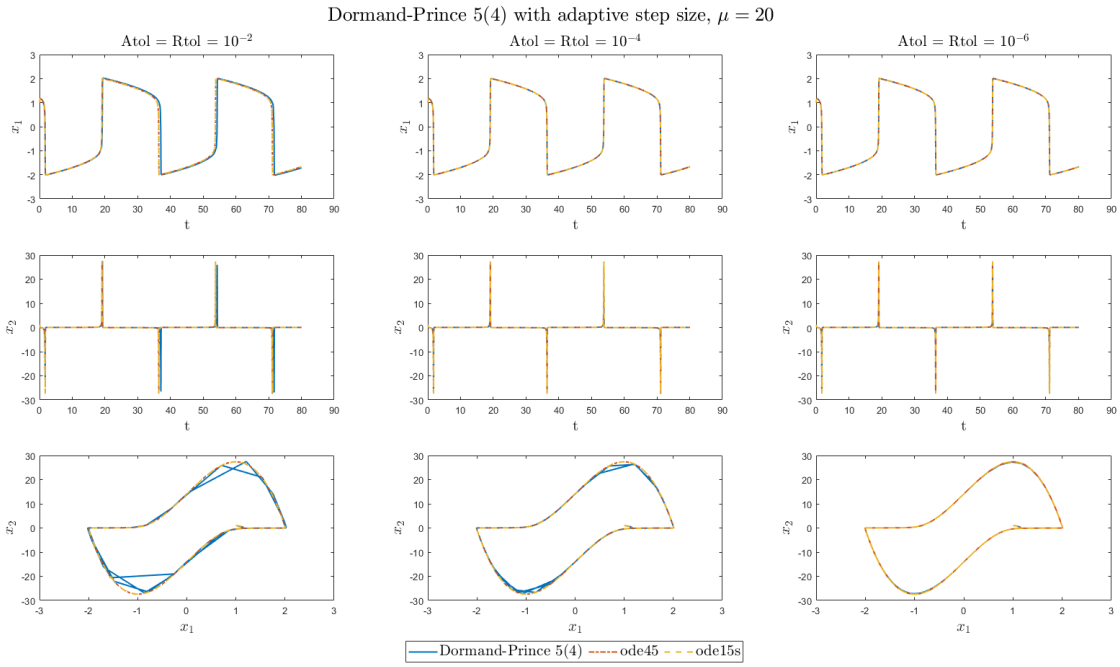

Figure 6.7: We have tested the Dormand-Prince 5(4) with adaptive step size on the Van Der Pol system with $\mu = 20$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 6.8 the picture is the same as for the classical Runge-Kutta and hence we will refer there for any comments.

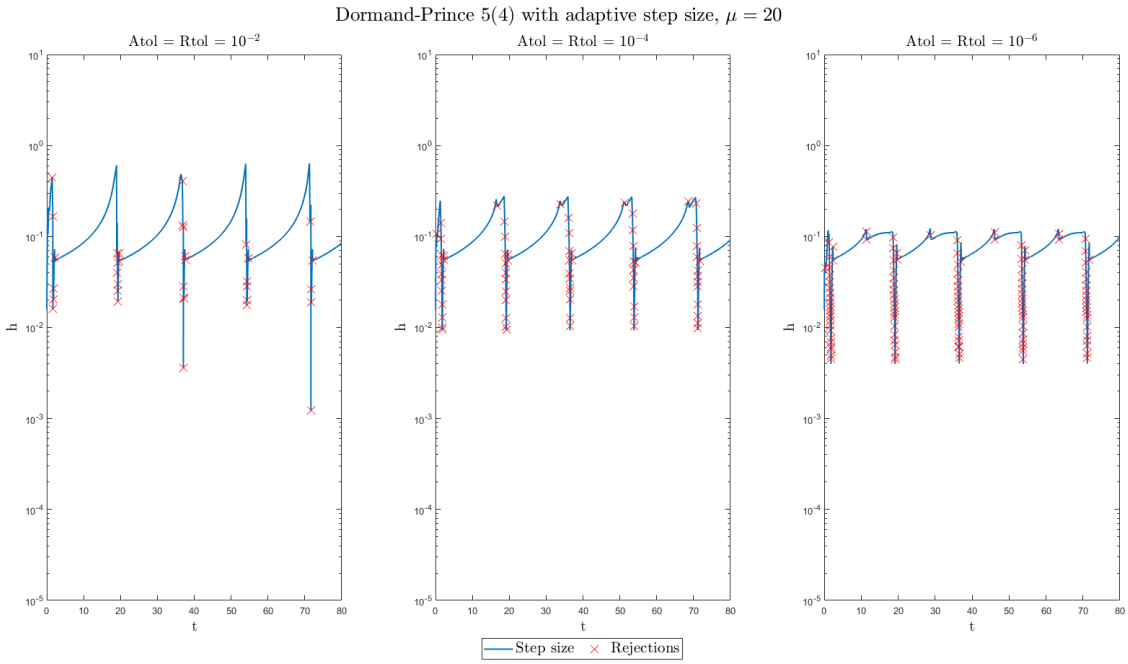

Figure 6.8: Here we have plotted the accepted step sizes for the test on the stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

If we compare table 6.3 and 6.4 with the numbers we got from the classical Runge-Kutta method we see that the simulations are faster for the Dormand-Prince 5(4). This also makes sense because the error estimation in the Dormand-Prince 5(4) is build in to the method and comes almost for free. For the classical Runge Kutta on the other hand the error estimation is done by step doubling that triples the amount of computation per iteration which is needed.

	Adaptive Step		
h/tol	10^{-2}	10^{-4}	10^{-6}
Time	0	0	0.031
Function calls	331	639	1325
Steps	47	91	189

Table 6.3: Comparison for the non-stiff system

	Adaptive Step		
h/tol	10^{-2}	10^{-4}	10^{-6}
Time	0.031	0.078	0.078
Function calls	6708	7422	9522
Steps	958	1060	1360

Table 6.4: Comparison for the stiff system

6.5 Exercise 6.5 and 6.6

Next we will test the Dormand-Prince 5(4) method on the CSTR 1D and 3D problems. The problems describe an exothermic reaction conducted in an adiabatic continuous stirred tank reactor. The 3D problem is a 3-state model describing temperature, mass balance and energy balance.

$$\begin{aligned} \dot{C}_\Lambda &= \frac{F}{V}\left(C_{A,in} - C_A\right) + R_A\left(C_\Lambda, C_B, T\right) \\ \dot{C}_B &= \frac{F}{V}\left(C_{B,in} - C_B\right) + R_B\left(C_A, C_B, T\right) \\ \dot{T} &= \frac{F}{V}\left(T_{in} - T\right) + R_T\left(C_\Lambda, C_B, T\right) \end{aligned} \tag{6.7}$$

The 1D problem is then an approximation which is given by

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

In both the 1D and 3D problem the R-functions are production rates which are given as

$$\begin{aligned} R_A\left(C_A, C_B, T\right) &= -r\left(C_A, C_B, T\right) \\ R_B\left(C_A, C_B, T\right) &= -2r\left(C_A, C_B, T\right) \\ R_T\left(C_A, C_B, T\right) &= \beta \mathrm{r}\left(C_A, C_B, T\right) \end{aligned} \tag{6.9}$$

where β is

$$\beta = -\frac{\Delta H_r}{\rho C_P}, \tag{6.10}$$

and $r(\cdot)$ is the rate of reaction

$$r\left(C_A, C_B, T\right) = k(T) C_A C_B. \tag{6.11}$$

In the rate of reaction the $k(\cdot)$ is the Arrhenius expression

$$k(T) = k_0 e^{\left(\frac{-E_a}{RT}\right)}. \tag{6.12}$$

In [WJ20] they simulate the two systems where they change F at specific points in the simulation. We will try to replicate the simulation and hence we have chosen the same parameter values as in the paper. The values are given in table 6.5 and the time points and values for F are given in figure 6.9 together with the temperature that our simulation should result in.

Density	ρ	1.0	kg/L
Specific heat capacity	c_P	4.186	kJ/(kg · K)
Arrhenius constant	k_0	exp(24.6)	L/(mol · s)
Activation cnergy	E_a/R	8500	K
Reaction enthalpy	ΔH_r	−560	kJ/mol
Reactor volume	V	0.105	L
Inlet concentration of A	$C_{A,in}$	1.6/2	mol/L
Inlet concentration of B	$C_{B,in}$	2.4/2	mol/L
Inlet temperature	T_{in}	273.65	K
		= 0.5	°C

Table 6.5: Parameter values

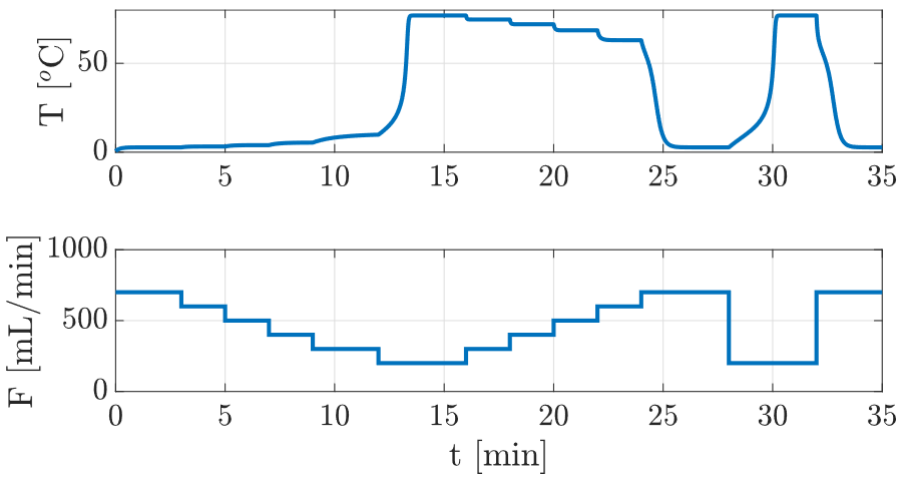

Figure 6.9: Values of F and the corresponding temperature from the paper [WJ20].

1D problem

We first simulate the 1D problem. In figure 6.10 we see the simulation. First of all we see that there is no difference between the Matlab solvers and our own implementation of Dormand-Prince 5(4). Next we see that our simulation is identical to the one from the paper shown in figure 6.9.

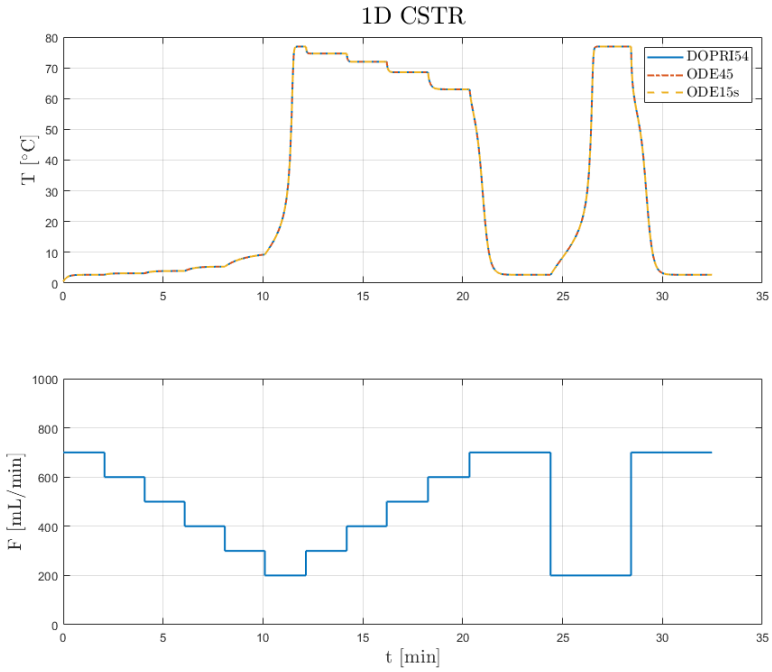

Figure 6.10: Simulation of the 1D CSTR problem.

In figure 6.11 we see that every time we change F the step size is reduced but then rapidly increased to the maximum step size. We also see that very few rejection are made.

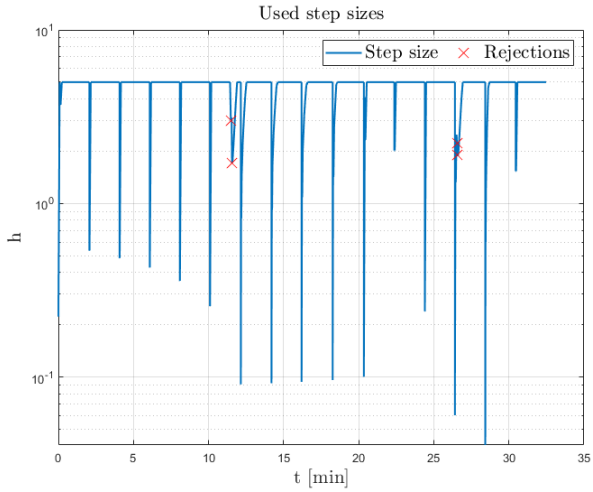

Figure 6.11: Used step sizes for the 1D simulation. The red crosses indicate that at the specific iteration larger step sizes than the one shown was proposed but rejected.

In table 6.6 we have stated some numbers related to the simulation. Also here we see very few rejections are made.

Function calls	Time	Total steps	Rejected steps
3091	0.047	453	4

Table 6.6: Summary statistics for the 1D simulation

3D problem

We now continue to the 3D problem. In figure 6.12 and 6.13 the simulation is shown. First of all we see that the is no difference between the Matlab solvers and our solver as for the 1D problem. We also see that figure 6.12 is identical with the simulation from the paper.

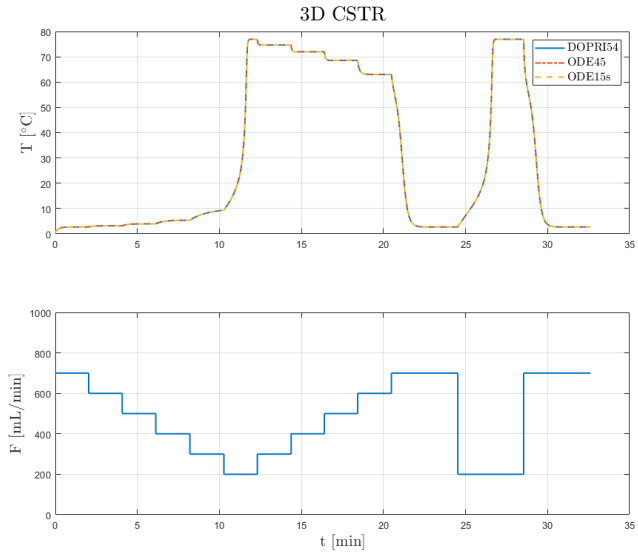

Figure 6.12: Simulation of the 3D CSTR problem showing the F-values and the simulated temperature.

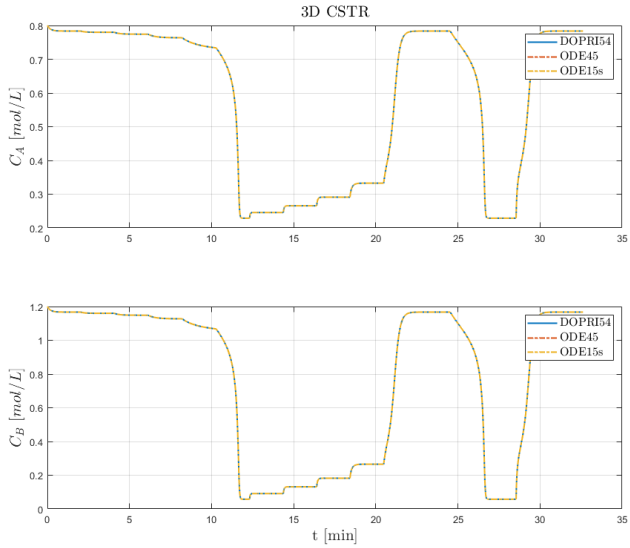

Figure 6.13: Simulation of the 3D CSTR problem showing the simulated mass balance and the simulated energy balance.

In figure 6.14 we see the used steps sizes and rejections for the 3D simulation. We see a similar picture as the one we saw for the 1D problem. We use the maximum step size most of the time and when we have some sudden shifts in the dynamics, i.e. temperature, mass balance or energy balance, then we have some rejections.

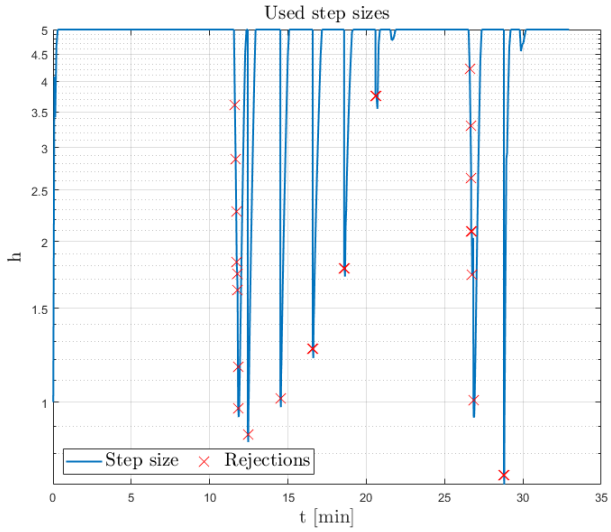

Figure 6.14: Used step sizes for the 3D simulation. The red crosses indicate that at the specific iteration larger step sizes than the one shown was proposed but rejected.

In table 6.7 we have stated some numbers related to the simulation. We see that we do not have many rejections but more than for the 1D simulation.

Function calls	Time	Total steps	Rejected steps
3402	0.125	515	39

Table 6.7: Summary statistics for the 3D simulation

CHAPTER 7

ESDIRK23

In this chapter we will investigate the last method of this assignment. To do this we will again use an IVP of the form

$$\dot{x}(t) = f(t, x(t), p), \quad x(t_0) = x_0, \tag{7.1}$$

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

7.1 Exercise 7.1

Runge-Kutta methods comes in many different types but in figure 7.1 we have tried to classify Runge-Kutta methods into 5 different categories which especially emphasis on differences between the implicit methods.

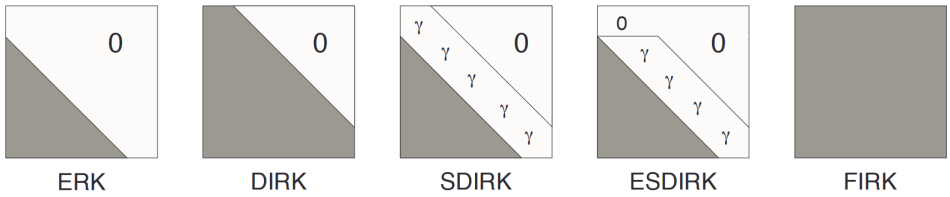

Figure 7.1: Structure of the A-matrix for different classes of Runge-Kutta methods. The figure is from [JT18]

Explicit Runge-Kutta(ERK) methods are the most computational efficient methods because they do not need to apply a newton algorithm within each step. This comes at the cost of poor stability properties. In the other end of the spectrum are the fully implicit Runge-Kutta(FIRK) methods which has excellent stability properties but are very computationally expensive due to the need of solving $s \times m$ coupled nonlinear equations as mentioned in chapter 3. In between we have the diagonally implicit Runge-Kutta(DIRK) methods which simplifies the task, to s systems of m nonlinear equations. Further we have the singly diagonally implicit Runge-Kutta(SDIRK) methods which can reuse the LU-factorization of the Jacobian for all s system because we solve for the same constant, γ, in each system. Lastly we have the most computationally efficient class within the implicit methods. This is the explicit singly diagonally

implicit Runge-Kutta(ESDIRK) methods which have an explicit first stage. In this exercise we will work with the method ESDRIK23 which is from the family of ESDIRK methods.

Order conditions

The ESDRIK23 is a stiffly accurate method and hence $a_{si} = b_i$ for $i = 1, 2, \ldots, s$. The method therefore have a Butcher tableau of the form given in 7.2.

$$\begin{array}{c|ccc} 0 & 0 & & \\ c_2 & a_{21} & \gamma & \\ 1 & b_1 & b_2 & \gamma \\ \hline & b_1 & b_2 & \gamma \\ & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 \end{array} \tag{7.2}$$

We want the ESDIRK23 to be A-stable, L-stable and of the highest possible order. To obtain this we could apply the theory described in chapter 1 but from [JT18] we know that this will not always determine the method uniquely. Hence we add the additional design criteria

$$C(q): \quad Ac^{k-1}e = \frac{1}{k}c^k e \quad k = 1, 2, \ldots, q. \tag{7.3}$$

This design criteria enforces order q for the internal stages in our method. The stability function for the ESDIRK23 method is given in 7.4.

$$R(z) = \frac{1 + (b_1 + b_2 - \gamma)\, z + (a_{21}b_2 - b_1\gamma)\, z^2}{(1 - \gamma z)^2} \tag{7.4}$$

For the method to be L-stable and hence also A-stable we need the 7.4 to go to 0 when z goes to ∞. For this to be true it must hold that $a_{21}b_2 - b_1\gamma = 0$. We could now try to design a method of 3rd order but in [JT18] they show that no siffly accurate ESDIRK method of order 3 with 3 stages is L-stable. Hence we aim at constructing a method of order 2 and with an internal order of 2. Lastly we also demand our embedded method to be of order 3. From [JT18] we hence knows that it gives the following order conditions.

$$\text{L-stability:} \quad \begin{cases} a_{21}b_2 - b_1\gamma = 0 \end{cases} \tag{7.5}$$

$$\text{1st order:} \quad \begin{cases} b_1 + b_2 + \gamma = 1 \end{cases} \tag{7.6}$$

$$\text{2nd order:} \quad \begin{cases} b_2c_2 + \gamma = \frac{1}{2} \end{cases} \tag{7.7}$$

$$\text{Internal order:} \quad \begin{cases} \gamma c_2 = \frac{1}{2}c_2^2 \\ c_2 = a_{21} + \gamma \end{cases} \tag{7.8}$$

$$\text{Embedded 1st order:} \quad \begin{cases} \hat{b}_1 + \hat{b}_2 + \hat{b}_3 = 1 \end{cases} \tag{7.9}$$

$$\text{Embedded 2nd order:} \quad \begin{cases} \hat{b}_2c_2 + \hat{b}_3 = \frac{1}{2} \end{cases} \tag{7.10}$$

$$\text{Embedded 3rd order:} \quad \begin{cases} \hat{b}_2c_2^2 + \hat{b}_3 = \frac{1}{3} \\ \hat{b}_2\left(c_2\gamma\right) + \hat{b}_3\left(c_2b_2 + \gamma\right) = \frac{1}{6} \end{cases} \tag{7.11}$$

In the above consistency is implicitly enforced by the 2nd order and Internal order conditions and hence not stated. If we solve the system we obtain the following Butcher tableau.

$$\text{ESDIRK23:} \quad \begin{array}{c|ccc} 0 & 0 & & \\ 2\gamma & \gamma & \gamma & \\ 1 & \frac{1-\gamma}{2} & \frac{1-\gamma}{2} & \gamma \\ \hline & \frac{1-\gamma}{2} & \frac{1-\gamma}{2} & \gamma \\ & \frac{6\gamma-1}{12\gamma} & \frac{1}{12\gamma(1-2\gamma)} & \frac{1-3\gamma}{3(1-2\gamma)} \end{array} \tag{7.12}$$

where $\gamma = \frac{2-\sqrt{2}}{2}$.

7.2 Exercise 7.2

The ESDIRK23 methods stability region is given by the equation 7.4 and plotted in figure 7.2. From definition 7 in chapter 1, we see that the method is A-stable.

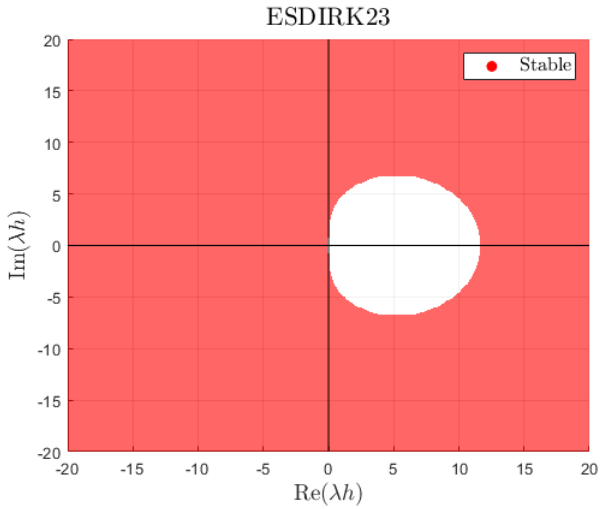

Figure 7.2: Stability region of the ESDIRK23 method

We though also see that a lot of the right hand plane is also stable like the implicit Euler method. We saw in figure 1.12 that the implicit Euler was too stable for the problem 1.30. Hence we try to solve the same system with the ESDIRK23.

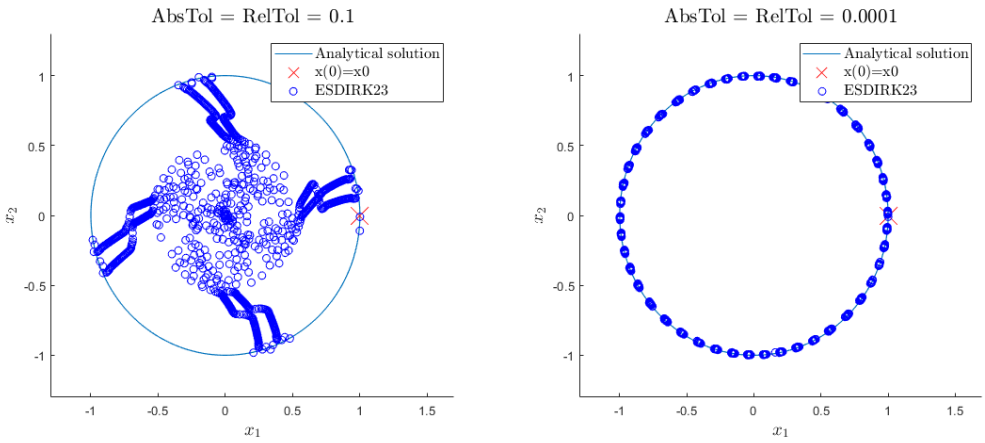

Figure 7.3: ESDIRK23 simulating the system 1.30 for different tolerances.

We see that for a low tolerance, ESDIRK23's step size controller can manage to stay on the stable attractor but when we relax the tolerance the method diverges from the stable attractor. We also saw that the implicit Euler was 'more' stable than the actual system due to being stable in the right half plane which is also the case

for the ESDIRK23 method. We saw in chapter 1 that the Trapezoidal method for example is not stable in the right plane but only the left plane and hence superior for this problem when it comes to stability. (see figure 1.14)

To investigate the L-stability properties of the ESDIRK23 method we will solve the system 1.32, given in section 1.6. We solve the system and make sure that the Trapezoidal method takes approximately the same size steps as the ESDIRK23 method.

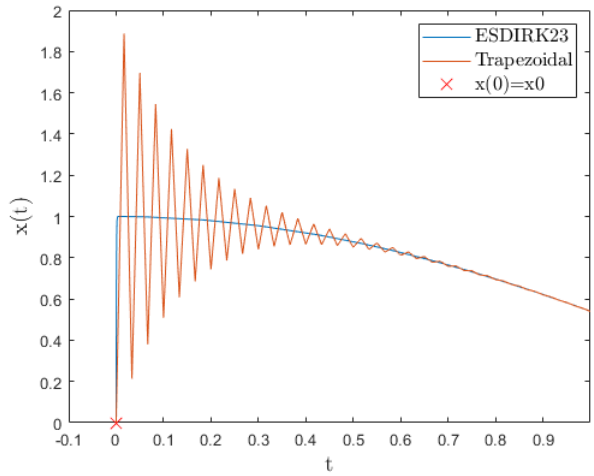

Figure 7.4: ESDIRK23 and the Trapezoidal method simulating the system 1.32

We see from figure 7.4 that the ESDIRK23 shows behaviour very similar to the implicit Euler method in figure 1.15 which were also L-stable. Hence we have given a piece of empirical evidence of the ESDIRK23 method being L-stable.

Embedded method

We will only just briefly comment on the embedded method. In figure 7.5 the stability region for the embedded method is plotted. We see that it is not A-stable and hence neither L-stable. In the paper [JT18] they show that a ESDIRK family designed by Anne Kværnø, posses embedded methods which are both stiffly accurate and A-stable. The embedded methods in this family are not L-stable but $|R(\infty)|$ is minimized. The down side of these methods is that they demand more stages. The method that corresponds to the ESDIRK23 has 4 stages instead of 3 and hence is more computational heavy.

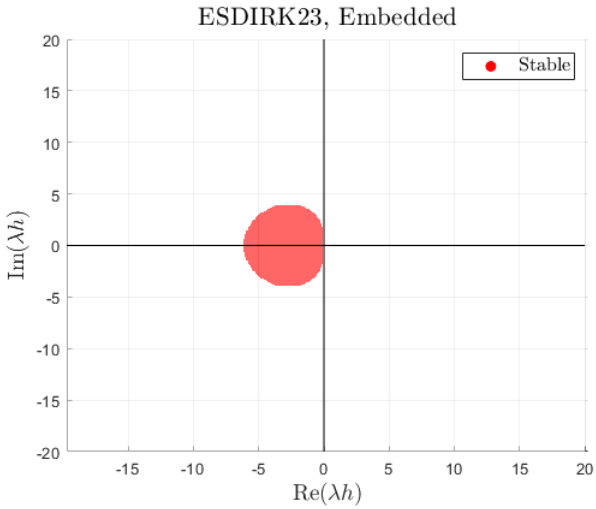

Figure 7.5: Stability region of the embedded method in ESDIRK23

7.3 Exercise 7.3

We have based our implementation on the provided code from lecture 10 and given our code in appendix A.3. Two new things are introduced that we have not yet discussed. They use another step size controller and an inexact newton method. In the paper [CS10] a combined pseudo algorithm for both the step size control and the factorizing and updating strategy for the Jacobian is given. We restate it here in figure 7.6.

Algorithm 5.1: The complete modified PI controller for an implicit Runge-Kutta method.

if *iterations converged* **then**
- $h_r \leftarrow \left(\frac{\epsilon}{r}\right)^{1/\hat{k}} h$
- **if** *step accepted* **then**
  - **if** *step restricted* **then**
    - $h_r \leftarrow \frac{h}{h_{acc}} h_r$
  - **else**
    - $h_r \leftarrow \frac{h}{h_{acc}} \left(\frac{r_{acc}}{r}\right)^{1/\hat{k}} h_r$
  - $r_{acc} \leftarrow r$
  - $h_{acc} \leftarrow h$
- $h \leftarrow \min\left(h_r, \left(\frac{\alpha_{ref}}{\alpha}\right)^{1/\hat{k}} h\right)$
- **if** $\alpha - |h - h_{LU}|/h_{LU} > \alpha_{Jac}$ **then**
  - Form new Jacobian and factorize iteration matrix.
  - $h_{LU} \leftarrow h$
- **else if** $|h - h_{LU}|/h_{LU} > \alpha_{LU}$ **then**
  - Factorize iteration matrix.
  - $h_{LU} \leftarrow h$

else
- **if** *new Jacobian* **then**
  - **if** $\alpha > \alpha_{ref}$ **then**
    - $h \leftarrow \left(\frac{\alpha_{ref}}{\alpha}\right)^{1/\hat{k}} h$
  - **else**
    - $h \leftarrow h/2$
  - Step restricted.
- **else**
  - Form new Jacobian.
- Factorize iteration matrix.
- $h_{LU} \leftarrow h$

Figure 7.6: Pseudo algorithm for the step size control and the inexact newton method introduced in the paper [CS10].

Step size control

We see that the first part of the algorithm updates the step size. If the Newton method has converged but the step is not accepted we update the step size using a asymptotic rule as we already know from chapter 2.

$$h_r = \left(\frac{\epsilon}{r_{n+1}}\right)^{1/(p+1)} h_n \tag{7.13}$$

If the step is accepted we use a PI adjusted rule proposed by Gustaffson, [GS97],

$$h_r = \frac{h_n}{h_{n-1}} \left(\frac{r_n}{r_{n+1}} \right)^{1/(p+1)} \left(\frac{\epsilon}{r_{n+1}} \right)^{1/(p+1)}. \tag{7.14}$$

If the Newton method does not converge we use the update rules given in the bottom of the algorithm. The reason by using these instead of just the asymptotic rule is that if the asymptotic rule was used the step size would fluctuate wildly. The parameter α_{ref} we set to 0.6 as they also do in [CS10].

Inexact Newton

The rest of the algorithm is about updating and refactorize the Jacobian. In the Newton method it self we monitor the convergence rate α by

$$\alpha_i = \frac{\left\| R_i \left(X_i^{[l+1]} \right) \right\|}{\left\| R_i \left(X_i^{[l]} \right) \right\|} \tag{7.15}$$

where $R(x)$ is the residual function introduced in chapter 2 and the norm $\|\cdot\|$ is given by

$$\left\| R_i \left(X_i^{[l]} \right) \right\| = \max_{j \in 1,\ldots,n_x} \frac{| \left(R_i \left(X_i^{[l]} \right) \right)_j |}{\max \left\{ \text{atol}_j, \ \text{rtol}_j X_i^{[l]} \right\}} \tag{7.16}$$

We can now use the convergence rate α to decide if we should update or refactorize the Jacobian to improve convergence. We set the parameter α_{Jac}, which controls if we should update the Jacobian, to 0.2 and the parameter α_{LU}, which controls if we should refactorize the Jacobian, to 0.2. These values are from [CS10].

7.4 Exercise 7.4 and 7.5

We now test on the Van Der Pol system again. We will test a stiff system where $\mu = 20$ and a non-stiff system where $\mu = 3$. We will test the implemented ESDIRK23 using three different tolerances, 10^{-2}, 10^{-4} and 10^{-6}. We start with the non-stiff system. In figure 7.7 we see that the ESDIRK23 method makes a small error for the largest tolerance but for the smaller tolerances no error is made compared to the in build Matlab solvers.

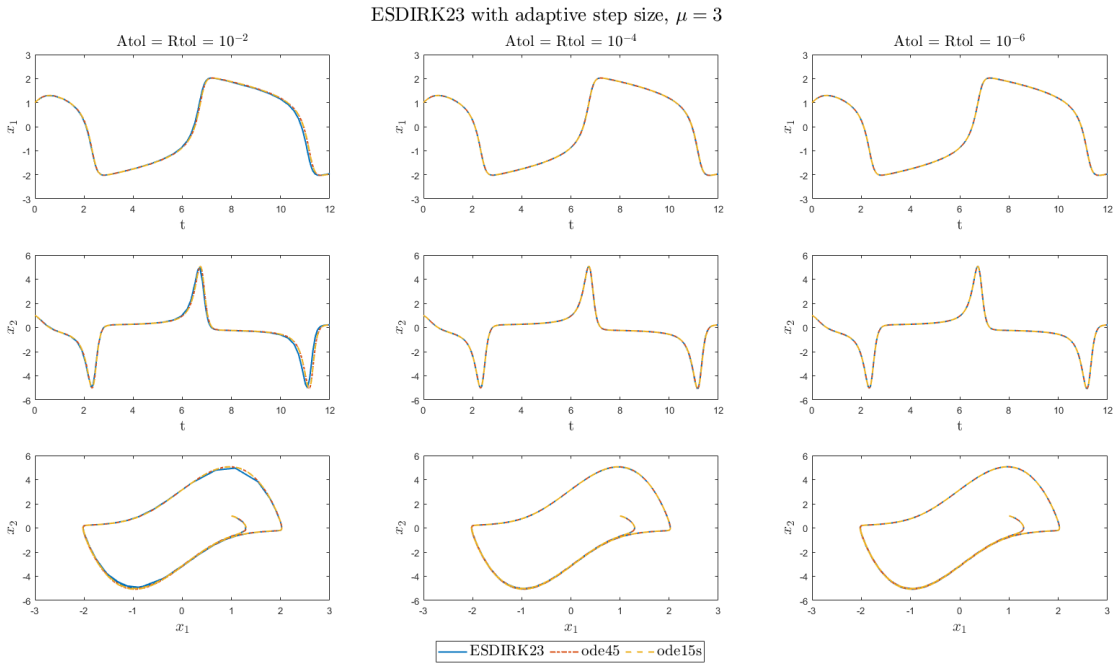

Figure 7.7: We have tested the ESDIRK23 with adaptive step size on the Van Der Pol system with $\mu = 3$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 7.8 we see the used step sizes for the non-stiff simulation. We see that some rejections are made when we leave a spike. This means our controller is not fast enough to decrease the step size and hence some rejections are made. Besides that very few rejections are made.

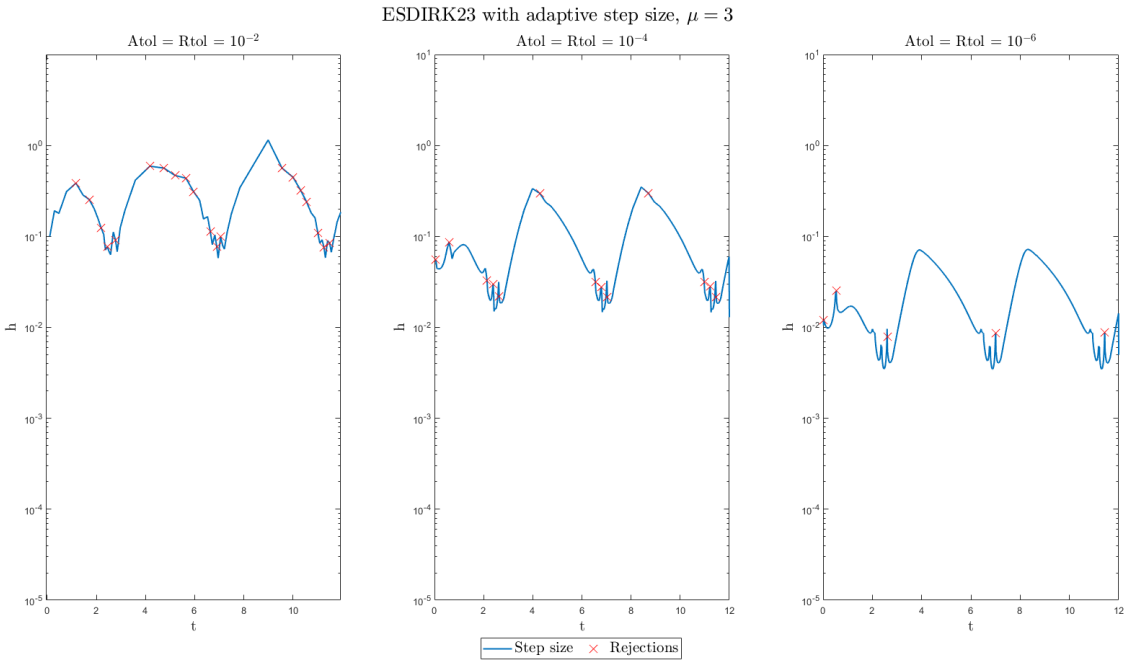

Figure 7.8: Here we have plotted the accepted step sizes for the test on the non-stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

We now proceed with the stiff system. In figure 7.9 the simulation is shown. We again see that an error is visible when the tolerance is set to 10^{-2} but for both 10^{-4} and 10^{-6} no error is visible compared to the in build solvers.

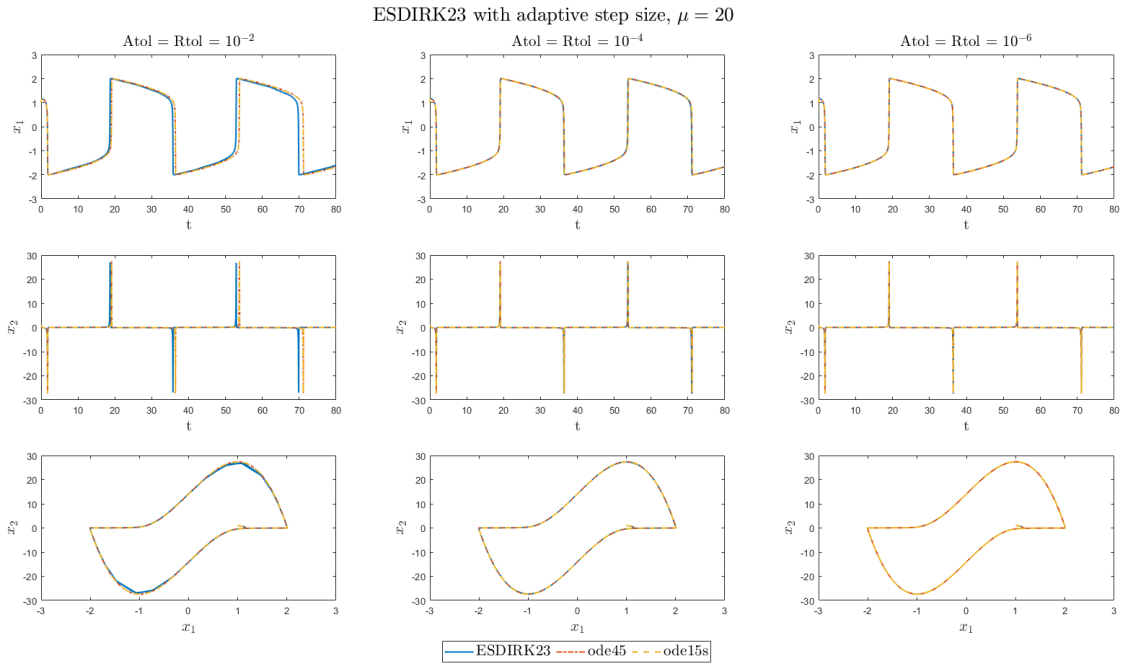

Figure 7.9: We have tested the ESDIRK23 with adaptive step size on the Van Der Pol system with $\mu = 20$. This we have done for different tolerances against the Matlab native solvers, `ode45` and `ode15s`.

In figure 7.10 we see the used step sizes for the stiff simulation. We see a similar picture to the one we saw for the non-stiff system. Just after a spike we have rejections. The difference between the non-stiff and the stiff system is that for the stiff system the number of rejections is higher. This also makes sense because the controller must change step size very rapidly for the stiff system.

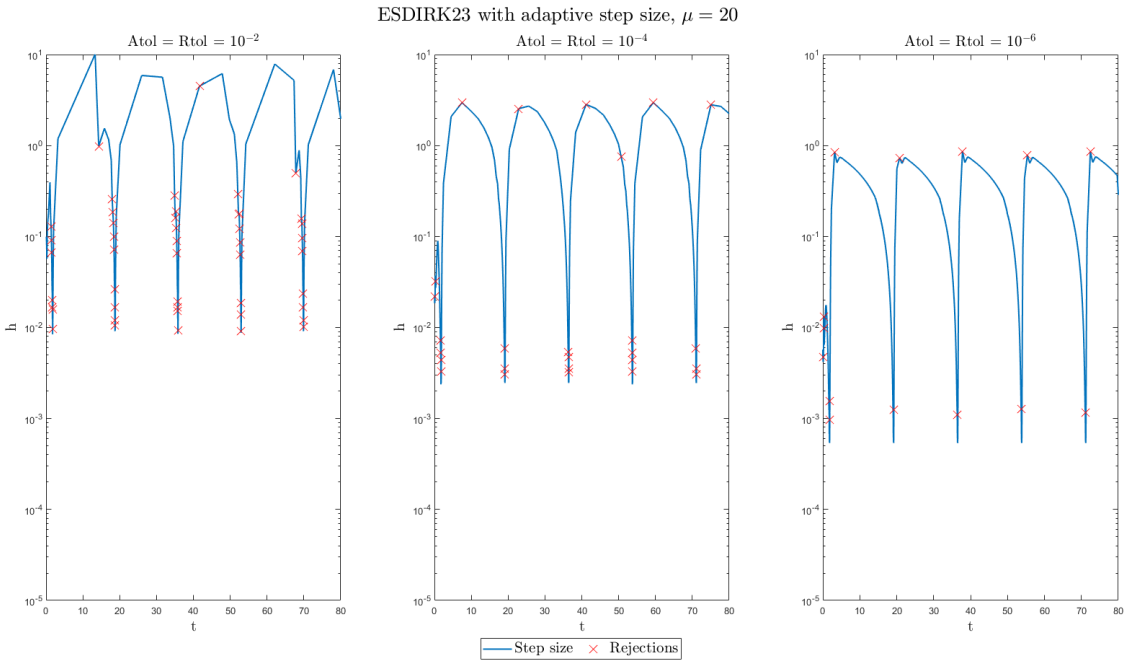

Figure 7.10: Here we have plotted the accepted step sizes for the test on the stiff version of the Van Der Pol system. The indications of a rejection means that before the shown accepted step size 1 or more larger step sizes was proposed but rejected by the control mechanism.

In table 7.1 and 7.2 we have summarized the simulations in some few numbers. We see that the stiff simulations take between 2 and 3 times more compute power and that we due to the inexact newton strategy saves a lot of Jacobian calls compared to the implicit Euler. Further we see that we do not have that many LU factorizations without also updating the Jacobian.

	Adaptive Step		
h/tol	10^{-2}	10^{-4}	10^{-6}
Time	0.016	0.016	0.063
Gradient calls	517	1199	4488
Jacobian calls	79	248	1066
LU factorizations	80	248	1066
Steps	79	247	1065

Table 7.1: Comparison for the non-stiff system

	Adaptive Step		
h/tol	10^{-2}	10^{-4}	10^{-6}
Time	0.031	0.203	0.422
Gradient calls	1384	3383	12560
Jacobian calls	207	662	2892
LU factorizations	210	664	2892
Steps	209	663	2891

Table 7.2: Comparison for the stiff system

CHAPTER 8

Discussion and Conclusions

In the past 7 chapters we have analyzed 7 different Runge-Kutta methods. 5 for deterministic ODEs and 2 for SDEs.

In chapter 1 and 6 we used the test equation which is primarily for theoretical considerations because the test equation is linear and hence easy to work with analytically. We saw all stability concepts considered in the report was based on the test equation and one should therefore be careful if carrying over any stability considerations to a non-linear domain.

In chapter 2-7 we tested on the Van Der Pol problem which is very easy to adjust in stiffness by turning the parameter μ up or down. We saw that the Euler methods needed a very low step size or tolerance to correctly integrate the problem. This was due to their very low order. In chapter 5 we worked with the classical Runge-Kutta method which is of order 4. It was much more accurate for large step sizes or high tolerances dependent on if one used a fixed or adaptive step size approach. In chapter 6 we worked with the Dormand-Prince 5(4) which order wise is only one higher compared to the classical Runge-Kutta method. The real gain of switching from the classical Runge-Kutta method to the Dormand-Prince 4(5) method is that now we have an embedded error estimation instead of paying a lot of compute power to obtain the error estimate from step doubling.

In chapter 7 we worked with the ESDIRK23 method which is the only implicit method besides the basic implicit Euler method we have worked with. The ESDIRK23 method is a very computationally optimized implicit method and hence we get a method which can handle very stiff systems but at a relatively low computational cost. In the report we have seen theoretical arguments for implicit methods should be superior to explicit methods for stiff problems. We have though here as a last test tried to compare the Dormand-Prince 5(4) method to the ESDIRK23 on the Van Der Pol problem with $\mu = 5, 10, 15, \ldots, 120$. We have used a absolute and relative tolerance of 10^{-6} and integrated over the time horizon $t \in [0, 20\mu]$. In figure 8.1 we see the test results. For around $\mu > 50$ the ESDIRK23 is faster than the Dormand-Prince 5(4) method. We see that the number of steps the Dormand-Prince 5(4) method uses grows much faster than the ESDIRK23 method and at $\mu \approx 50$ the Jacobian calls and non-linear solves the implicit method must do, pays of compared to the explicit not

doing them.

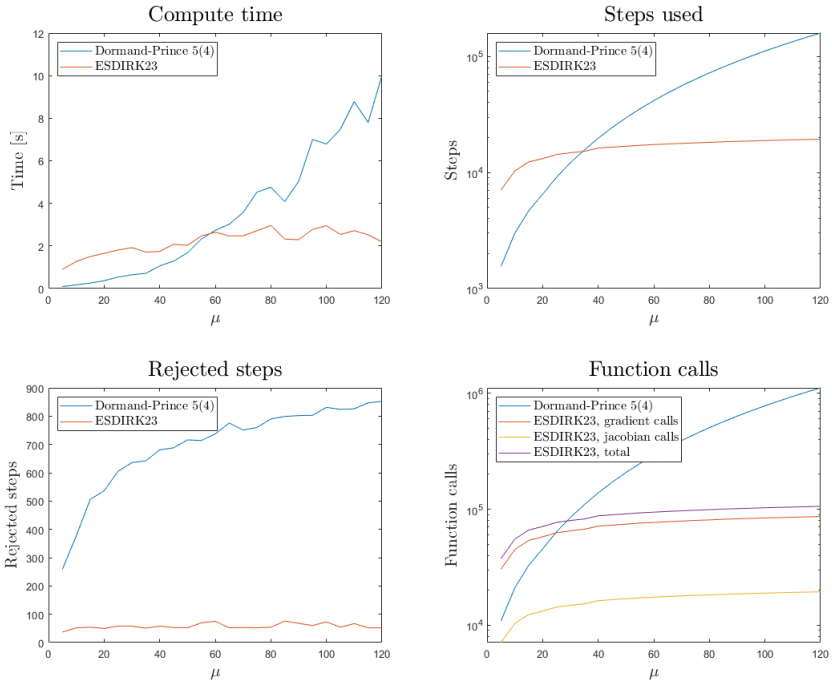

Figure 8.1: Comparison of the Dormand-Prince 5(4) and the ESDIRK23 methods on Van Der Pol problems of varying stiffness.

In chapter 4 we analyzed the state dependent and independent Van Der Pol problem and saw the stiff problem was much less prone to the stochasticity compared to the non-stiff system.

Lastly we also looked at the CSTR problem in chapter 6. This is a problem which describes an exothermic reaction conducted in an adiabatic continuous stirred tank reactor. The Dormand-Prince 5(4) handled it without any problems and the maximum step size was used every where except for the changes in F-value.

As a conclusion we have seen that the Euler methods are very easy to understand but not that useful. The classical Runge-Kutta method, the Dormand-Prince 5(4) and the ESDIRK23 though are all Runge-Kutta methods that has their problems and setting. One should identify which method to use in the specific setting.

APPENDIX A

Appendix

A.1 Adaptive Explicit Runge-Kutta Methods

I controller

```
function [x,t,function_calls,hs,rs] = ...
    explicitRungeKuttaDoubling(f,param,h,t0,T,x0,A,b,c,Atol,Rtol,hmin,hmax,eps_tol,p,
    N = ceil((T-t0)/hmin);
    s = length(b);
    d = length(x0);
    t = zeros(1,N+1);
    hs = zeros(2,N+1);
    rs = zeros(2,N+1);
    x = zeros(length(x0),N+1);
    k = zeros(d,s)';
    t(1) = t0;
    fac = 1;
    x(:,1) = x0;
    accept_step = false;
    function_calls = 0;

    if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
        [h,function_calls] = initialStepSize(f,param,t0,x0);
    end

    i = 2;
    l = 2;
    runs = 1;
    while t(i-1)≤T
       while(¬accept_step)
            if t(i-1)+h>T
                h = max(hmin,T-t(i-1));
            end

            k = 0*k;
            for j = 1:s
                k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                    param);
                function_calls = function_calls +1;
            end
            oneAhead = x(:,i-1)+sum(h*k.*b,1)';
```

```
35
36              hhalf = h/2;
37              k = 0*k;
38              for j = 1:s
39                  k(j,:) = ...
                        f(t(i-1)+hhalf*c(j),x(:,i-1)+sum(hhalf*k.*A(j,:)',1)', ...
                        param);
40                  function_calls = function_calls +1;
41              end
42              halfAhead = x(:,i-1)+sum(hhalf*k.*b,1)';
43
44              k = 0*k;
45              for j = 1:s
46                  k(j,:) = ...
                        f(t(i-1)+2*hhalf*c(j),halfAhead+sum(hhalf*k.*A(j,:)',1)', ...
                        param);
47                  function_calls = function_calls +1;
48              end
49              halfAhead = halfAhead+sum(hhalf*k.*b,1)';
50
51              % Step doubling, see section II.4(s164) in Harier and slide ...
                    4c,3
52              e = abs(oneAhead-halfAhead);
53              %r = max(e./max(Atol,abs(halfAhead).*Rtol));
54              r = max(e./max(Atol,abs(halfAhead).*Rtol));
55
56              if r<=1
57                  x(:,i) = halfAhead;
58                  hs(1,i-1) = h;
59                  hs(2,i-1) = runs;
60                  rs(1,l-1) = r;
61                  rs(2,l-1) = 1;
62                  runs = 1;
63                  t(i) = t(i-1)+h;
64                  accept_step = true;
65                  h = h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))); % ...
                        eq 4.13 in Harier
66              else
67                  runs = runs +1;
68                  rs(1,l-1) = r;
69                  rs(2,l-1) = 2;
70                  accept_step = false;
71                  h = h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))); % ...
                        eq 4.13 in Harier
72                  l = l+1;
73              end
74
75
76          end
77          accept_step = false;
78          i = i+1;
79          l = l+1;
80      end
81      x = x(:,1:i-1)';
82      t = t(1:i-1);
```

```
83      hs = hs(1:2,1:i-2);
84      rs = rs(1:2,1:l-2);
85  end
```

Listing A.1: The Explicit Runge Kutta method with step doubling using a I controller

PI controller

```
1   function [x,t,function_calls,hs,rs] = ...
        explicitRungeKuttaDoublingPI(f,param,h,t0,T,x0,A,b,c,Atol,Rtol,hmin,hmax,eps_tol,p
2       N = ceil((T-t0)/hmin);
3       s = length(b);
4       d = length(x0);
5       t = zeros(1,N+1);
6       hs = zeros(2,N+1);
7       rs = zeros(2,N+1);
8       x = zeros(length(x0),N+1);
9       k = zeros(d,s)';
10      fac = 1;
11      t(1) = t0;
12      x(:,1) = x0;
13      accept_step = false;
14      function_calls = 0;
15      p = p;
16
17      if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
18          [h,function_calls] = initialStepSize(f,param,t0,x0);
19      end
20
21      i = 2;
22      l = 2;
23      runs = 1;
24      while t(i-1)≤T
25          while(¬accept_step)
26              if t(i-1)+h>T
27                  h = max(hmin,T-t(i-1));
28              end
29
30              k = 0*k;
31              for j = 1:s
32                  k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                        param);
33                  function_calls = function_calls +1;
34              end
35              oneAhead = x(:,i-1)+sum(h*k.*b,1)';
36
37              hhalf = h/2;
38              k = 0*k;
39              for j = 1:s
```

```
40              k(j,:) = ...
                    f(t(i-1)+hhalf*c(j),x(:,i-1)+sum(hhalf*k.*A(j,:)',1)', ...
                    param);
41              function_calls = function_calls +1;
42          end
43          halfAhead = x(:,i-1)+sum(hhalf*k.*b,1)';
44
45          k = 0*k;
46          for j = 1:s
47              k(j,:) = ...
                    f(t(i-1)+2*hhalf*c(j),halfAhead+sum(hhalf*k.*A(j,:)',1)', ...
                    param);
48              function_calls = function_calls +1;
49          end
50          halfAhead = halfAhead+sum(hhalf*k.*b,1)';
51
52          % Step doubling, see section II.4(s164) in Harier and slide ...
                4c,3
53          %sc = Atol + max(abs(halfAhead),abs(oneAhead))*Rtol;
54
55          %e = norm((oneAhead-halfAhead)./sc,'Inf');
56          e = abs(oneAhead-halfAhead);
57          e = max(e./max(Atol,abs(halfAhead).*Rtol)); % r i john algo
58
59          if e<=1
60              x(:,i) = halfAhead;
61              hs(1,i-1) = h;
62              hs(2,i-1) = runs;
63              rs(1,l-1) = e;
64              rs(2,l-1) = 1;
65              runs = 1;
66              t(i) = t(i-1)+h;
67              accept_step = true;
68              if i==2
69                  h = ...
                        h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); ...
                        % eq 4.13 in Harier
70                  e1 = max(e,10^(-10));
71              else
72                  %h = ...
                        h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.4/(p+1))*(e1/e)^(0.3/(p+1))
                        % eq 4.13 in Harier
73                  h = ...
                        h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.8/(p+1))*(e1/e)^(0.31/(p+1)
                        % eq 4.13 in Harier
74                  e1 = max(e,10^(-10));
75              end
76
77          else
78              runs = runs +1;
79              accept_step = false;
80              rs(1,l-1) = e;
81              rs(2,l-1) = 2;
82              h = h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); % ...
                    eq 4.13 in Harier
```

```
83                  l = l+1;
84              end
85
86
87          end
88          accept_step = false;
89          i = i+1;
90          l = l+1;
91      end
92      x = x(:,1:i-1)';
93      t = t(1:i-1);
94      hs = hs(1:2,1:i-2);
95      rs = rs(1:2,1:l-2);
96  end
```

Listing A.2: The Explicit Runge Kutta method with step doubling using a PI controller

PID controller

```
1   function [x,t,function_calls,hs,rs] = ...
        explicitRungeKuttaDoublingPID(f,param,h,t0,T,x0,A,b,c,Atol,Rtol,hmin,hmax,eps_tol
2       N = ceil((T-t0)/hmin);
3       s = length(b);
4       d = length(x0);
5       t = zeros(1,N+1);
6       hs = zeros(2,N+1);
7       rs = zeros(2,N+1);
8       x = zeros(length(x0),N+1);
9       k = zeros(d,s)';
10      fac = 1;
11      t(1) = t0;
12      x(:,1) = x0;
13      accept_step = false;
14      function_calls = 0;
15      p = p;
16
17      if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
18          [h,function_calls] = initialStepSize(f,param,t0,x0);
19      end
20
21      i = 2;
22      l = 2;
23      runs = 1;
24      while t(i-1)≤T
25          while(¬accept_step)
26              if t(i-1)+h>T
27                  h = max(hmin,T-t(i-1));
28              end
29
30              k = 0*k;
```

```
31              for j = 1:s
32                  k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                        param);
33                  function_calls = function_calls +1;
34              end
35              oneAhead = x(:,i-1)+sum(h*k.*b,1)';
36
37              hhalf = h/2;
38              k = 0*k;
39              for j = 1:s
40                  k(j,:) = ...
                        f(t(i-1)+hhalf*c(j),x(:,i-1)+sum(hhalf*k.*A(j,:)',1)', ...
                        param);
41                  function_calls = function_calls +1;
42              end
43              halfAhead = x(:,i-1)+sum(hhalf*k.*b,1)';
44
45              k = 0*k;
46              for j = 1:s
47                  k(j,:) = ...
                        f(t(i-1)+2*hhalf*c(j),halfAhead+sum(hhalf*k.*A(j,:)',1)', ...
                        param);
48                  function_calls = function_calls +1;
49              end
50              halfAhead = halfAhead+sum(hhalf*k.*b,1)';
51
52              % Step doubling, see section II.4(s164) in Harier and slide ...
                    4c,3
53              %sc = Atol + max(abs(halfAhead),abs(oneAhead))*Rtol;
54
55              %e = norm((oneAhead-halfAhead)./sc,'Inf');
56              e = abs(oneAhead-halfAhead);
57              e = max(e./max(Atol,abs(halfAhead).*Rtol)); % r i john algo
58
59              if e<=1
60                  x(:,i) = halfAhead;
61                  hs(1,i-1) = h;
62                  hs(2,i-1) = runs;
63                  rs(1,l-1) = e;
64                  rs(2,l-1) = 1;
65                  runs = 1;
66                  t(i) = t(i-1)+h;
67                  accept_step = true;
68                  if i==2
69                      h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); ...
                            % eq 4.13 in Harier
70                      e1 = max(e,10^(-10));
71                  elseif i == 3
72                      h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.8/(p+1))*(e1/e)^(0.31/(p+1)
                             % eq 4.13 in Harier
73                      e1 = max(e,10^(-10));
74                      e2 = e1;
75                  else
```

```
76                      h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.58/(p+1))*(e1/e)^(0.21/
                            % eq 4.13 in Harier
77                      e1 = max(e,10^(-10));
78                      e2 = e1;
79                  end
80
81              else
82                  runs = runs +1;
83                  accept_step = false;
84                  rs(1,l-1) = e;
85                  rs(2,l-1) = 2;
86                  l = l+1;
87                  h = h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); % ...
                        eq 4.13 in Harier
88              end
89
90
91          end
92          accept_step = false;
93          i = i+1;
94          l = l+1;
95      end
96      x = x(:,1:i-1)';
97      t = t(1:i-1);
98      hs = hs(1:2,1:i-2);
99      rs = rs(1:2,1:l-2);
100 end
```

Listing A.3: The Explicit Runge Kutta method with step doubling using a PID controller

A.2 Adaptive Explicit Runge-Kutta Methods With Embedded Error Estimation

I controller

```
1  function [x,t,function_calls,hs,rs] = ...
       explicitRungeKuttaEmbedded(f,param,h,t0,T,x0,A,b,bhat,c,Atol,Rtol,hmin,hmax,eps_t
2      N = ceil((T-t0)/hmin);
3      s = length(b);
4      d = length(x0);
5      t = zeros(1,N+1);
6      hs = zeros(2,N+1);
7      rs = zeros(2,N+1);
8      x = zeros(length(x0),N+1);
9      k = zeros(d,s)';
10     fac = 1;
```

```
11      t(1) = t0;
12      x(:,1) = x0;
13      accept_step = false;
14      function_calls = 0;
15
16
17      if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
18          [h,function_calls] = initialStepSize(f,param,t0,x0);
19      end
20
21      runs = 1;
22      i = 2;
23      l = 2;
24      while t(i-1)≤T
25
26          while(¬accept_step)
27              k = 0*k;
28              for j = 1:s
29                  k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                        param);
30                  function_calls = function_calls +1;
31              end
32              oneAhead = x(:,i-1)+sum(h*k.*b,1)';
33              oneAheadhat = x(:,i-1)+sum(h*k.*bhat,1)';
34
35              % Step doubling, see section II.4(s164) in Harier and slide ...
                    4c,3
36              e = abs(oneAhead-oneAheadhat);
37              r = max(e./max(Atol,abs(oneAhead).*Rtol));
38
39              if r≤1
40                  x(:,i) = oneAhead;
41                  hs(1,i-1) = h;
42                  hs(2,i-1) = runs;
43                  rs(1,l-1) = r;
44                  rs(2,l-1) = 1;
45                  runs = 1;
46                  t(i) = t(i-1)+h;
47                  accept_step = true;
48                  h = ...
                        max(h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))),hmin); ...
                        % eq 4.13 in Harier
49              else
50                  runs = runs +1;
51                  accept_step = false;
52                  rs(1,l-1) = r;
53                  rs(2,l-1) = 2;
54                  h = ...
                        max(h*max(hmin,min(hmax,fac*(eps_tol/r)^(1/(1+p)))),hmin); ...
                        % eq 4.13 in Harier
55                  l = l+1;
56              end
57
58          end
59          accept_step = false;
```

```
60          i = i+1;
61          l = l+1;
62      end
63      x = x(:,1:i-1)';
64      t = t(1:i-1);
65      hs = hs(:,1:i-2);
66      rs = rs(:,1:l-2);
67  end
```

Listing A.4: The Explicit Runge Kutta method with embedded error estimation using a I controller

PI controller

```
1   function [x,t,function_calls,hs,rs] = ...
        explicitRungeKuttaEmbeddedPI(f,param,h,t0,T,x0,A,b,bhat,c,Atol,Rtol,hmin,hmax,eps_
2       N = ceil((T-t0)/hmin);
3       s = length(b);
4       d = length(x0);
5       t = zeros(1,N+1);
6       hs = zeros(2,N+1);
7       rs = zeros(2,N+1);
8       fac = 1;
9       x = zeros(length(x0),N+1);
10      k = zeros(d,s)';
11      t(1) = t0;
12      x(:,1) = x0;
13      accept_step = false;
14      function_calls = 0;
15
16
17      if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
18          [h,function_calls] = initialStepSize(f,param,t0,x0);
19      end
20
21      runs = 1;
22      i = 2;
23      l = 2;
24      while t(i-1)≤T
25
26          while(¬accept_step)
27              k = 0*k;
28              for j = 1:s
29                  k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                        param);
30                  function_calls = function_calls +1;
31              end
32              oneAhead = x(:,i-1)+sum(h*k.*b,1)';
33              oneAheadhat = x(:,i-1)+sum(h*k.*bhat,1)';
34
```

```
35              % Step doubling, see section II.4(s164) in Harier and slide ...
                    4c,3
36              e = abs(oneAhead-oneAheadhat);
37              e = max(e./max(Atol,abs(oneAhead).*Rtol)); % r i john algo
38
39              if e<=1
40                  x(:,i) = oneAhead;
41                  hs(1,i-1) = h;
42                  hs(2,i-1) = runs;
43                  rs(1,l-1) = e;
44                  rs(2,l-1) = 1;
45                  runs = 1;
46                  t(i) = t(i-1)+h;
47                  accept_step = true;
48                  if i==2
49                      h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); ...
                            % eq 4.13 in Harier
50                      e1 = max(e,10^(-10));
51                  else
52                      %h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.4/(p+1))*(e1/e)^(0.3/(p+1))
                            % eq 4.13 in Harier
53                      h = ...
                            h*max(hmin,min(hmax,fac*(eps_tol/e)^(0.8/(p+1))*(e1/e)^(0.31/(p+1)
                            % eq 4.13 in Harier
54                      e1 = max(e,10^(-10));
55                  end
56
57              else
58                  runs = runs +1;
59                  rs(1,l-1) = e;
60                  rs(2,l-1) = 2;
61                  accept_step = false;
62                  h = h*max(hmin,min(hmax,fac*(eps_tol/e)^(1/(p+1)))); % ...
                        eq 4.13 in Harier
63                  l = l+1;
64              end
65
66          end
67          accept_step = false;
68          i = i+1;
69          l = l+1;
70      end
71      x = x(:,1:i-1)';
72      t = t(1:i-1);
73      hs = hs(:,1:i-2);
74      rs = rs(:,1:l-2);
75  end
```

Listing A.5: The Explicit Runge Kutta method with embedded error estimation using a PI controller

PID controller

```
function [x,t,function_calls,hs,rs] = ...
    explicitRungeKuttaEmbeddedPID(f,param,h,t0,T,x0,A,b,bhat,c,Atol,Rtol,hmin,hmax,eps
    N = ceil((T-t0)/hmin);
    s = length(b);
    d = length(x0);
    t = zeros(1,N+1);
    hs = zeros(2,N+1);
    rs = zeros(2,N+1);
    x = zeros(length(x0),N+1);
    k = zeros(d,s)';
    fac = 1;
    t(1) = t0;
    x(:,1) = x0;
    accept_step = false;
    function_calls = 0;


    if initial_step_algo % slide 4b, 9 whic is from p169 Hairer
        [h,function_calls] = initialStepSize(f,param,t0,x0);
    end

    runs = 1;
    i = 2;
    l = 2;

    h = max(hmin,min(h,hmax));
    while t(i-1)≤T

        while(¬accept_step)
            k = 0*k;
            for j = 1:s
                k(j,:) = f(t(i-1)+h*c(j),x(:,i-1)+sum(h*k.*A(j,:)',1)', ...
                    param);
                function_calls = function_calls +1;
            end
            oneAhead = x(:,i-1)+sum(h*k.*b,1)';
            oneAheadhat = x(:,i-1)+sum(h*k.*bhat,1)';

            % Step doubling, see section II.4(s164) in Harier and slide ...
                4c,3
            e = abs(oneAhead-oneAheadhat);
            e = max(e./max(Atol,abs(oneAhead).*Rtol)); % r i john algo

            if e≤1
                x(:,i) = oneAhead;
                hs(1,i-1) = h;
                hs(2,i-1) = runs;
                rs(1,l-1) = e;
                rs(2,l-1) = 1;
                runs = 1;
                t(i) = t(i-1)+h;
```

```
49                  accept_step = true;
50                  if i==2
51                      h = ...
                            max(hmin,min(hmax,h*fac*(eps_tol/e)^(1/(p+1)))); ...
                            % eq 4.13 in Harier
52                      e1 = max(e,10^(-10));
53                  elseif i == 3
54                      h = ...
                            max(hmin,min(hmax,h*fac*(eps_tol/e)^(0.8/(p+1))*(e1/e)^(0.31/(p+1)
                             % eq 4.13 in Harier
55                      e1 = max(e,10^(-10));
56                      e2 = e1;
57                  else
58                      h = ...
                            max(hmin,min(hmax,h*fac*(eps_tol/e)^(0.58/(p+1))*(e1/e)^(0.21/(p+
                            % eq 4.13 in Harier
59                      e1 = max(e,10^(-10));
60                      e2 = e1;
61                  end
62
63
64              elseif runs > 5
65                  display(runs)
66                  x(:,i) = oneAhead;
67                  hs(1,i-1) = h;
68                  hs(2,i-1) = runs;
69                  rs(1,l-1) = e;
70                  rs(2,l-1) = 1;
71                  runs = 1;
72                  t(i) = t(i-1)+h;
73                  accept_step = true;
74                  h = hmin;
75              else
76                  runs = runs +1;
77                  accept_step = false;
78                  rs(1,l-1) = e;
79                  rs(2,l-1) = 2;
80                  h = max(hmin,min(hmax,h*fac*(eps_tol/e)^(1/(p+1)))); % ...
                        eq 4.13 in Harier
81                  l = l+1;
82
83              end
84
85          end
86          accept_step = false;
87          i = i+1;
88          l = l+1;
89      end
90      x = x(:,1:i-1)';
91      t = t(1:i-1);
92      hs = hs(:,1:i-2);
93      rs = rs(:,1:l-2);
94  end
```

Listing A.6: The Explicit Runge Kutta method with embedded error estimation using a PID controller

A.3 ESDIRK

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

%% ESDIRK23 Parameters
%=========================================================================
% 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;

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

```
43                 'nStep',       0, ...
44                 'nAccept',     0, ...
45                 'nFail',       0, ...
46                 'nDiverge',    0, ...
47                 'nSlowConv',   0);
48
49
50
51 %% Main ESDIRK Integrator
52 %================================================================
53 nx = size(x0,1);
54 F = zeros(nx,s);
55 t = t0;
56 x = x0;
57 h = h0;
58
59 [F(:,1),g]  = feval(fun,t,x,varargin{:});
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 % Output
79 chunk = 100;
80 Tout = zeros(chunk,1);
81 Xout = zeros(chunk,nx);
82 Gout = zeros(chunk,nx);
83
84 Tout(1,1) = t;
85 Xout(1,:) = x.';
86 Gout(1,:) = g.';
87
88 runs = 1;
89 while t<tf
90     info.nStep = info.nStep+1;
91     %================================================================
92     % A step in the ESDIRK method
93     i=1;
94     diverging = false;
95     SlowConvergence = false; % carsten
96     alpha = 0.0;
97     Converged = true;
```

```
98      while (i<s) && Converged
99          % Stage i=2,...,s of the ESDIRK Method
100         i=i+1;
101         phi = g + F(:,1:i-1)*(h*AT(1:i-1,i));
102
103         % Initial guess for the state
104          if i==2
105              dt = c(i)*h;
106              G = g + dt*F(:,1);
107              X = x + dgdx\(G-g);
108          else
109              dt = c(i)*h;
110              G  = g + dt*F(:,1);
111              X  = x + dgdx\(G-g);
112          end
113         T = t+dt;
114
115         [F(:,i),G] = feval(fun,T,X,varargin{:});
116         info.nFun = info.nFun+1;
117         R = G - hgamma*F(:,i) - phi;
118 %        rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
119         rNewton = norm(R./(absTol + abs(G).*relTol),inf);
120         Converged = (rNewton < tau);
121         %iter(i) = 0; % original, if uncomment then comment line 154: ...
                iter(:) = 0;
122         % Newton Iterations
123         while ¬Converged && ¬diverging && ¬...
                SlowConvergence%iter(i)<itermax
124             iter(i) = iter(i)+1;
125             dX = U\(L\(R(pivot,1)));
126             info.nBack = info.nBack+1;
127             X = X - dX;
128             rNewtonOld = rNewton;
129             [F(:,i),G] = feval(fun,T,X,varargin{:});
130             info.nFun = info.nFun+1;
131             R = G - hgamma*F(:,i) - phi;
132 %           rNewton = norm(R./(absTol + abs(G).*relTol),2)/sqrt(nx);
133             rNewton = norm(R./(absTol + abs(G).*relTol),inf);
134             alpha = max(alpha,rNewton/rNewtonOld);
135             Converged = (rNewton < tau);
136             diverging = (alpha ≥ 1);
137             SlowConvergence = (iter(i) ≥ itermax); % carsten
138             %SlowConvergence = (alpha ≥ 0.5); % carsten
139             %if (iter(i) ≥ itermax), i, iter(i), Converged, diverging, ...
                    pause, end % carsten
140         end
141         %diverging = (alpha ≥ 1); % original, if uncomment then ...
                comment line 142: diverging = (alpha ≥ 1)*i;
142         diverging = (alpha ≥ 1)*i; % carsten, recording which stage is ...
                diverging
143     end
144     %if diverging, i, iter, pause, end
145     nstep = info.nStep;
146     stats.t(nstep) = t;
147     stats.h(nstep) = h;
```

```
148     stats.r(nstep) = NaN;
149     stats.iter(nstep,:) = iter;
150     stats.Converged(nstep) = Converged;
151     stats.Diverged(nstep)  = diverging;
152     stats.AcceptStep(nstep) = false;
153     stats.SlowConv(nstep)  = SlowConvergence*i; % carsten, recording ...
            which stage is converging to slow (reaching maximum no. of ...
            iterations)
154     iter(:) = 0; % carsten
155     %=======================================================================
156     % Error and Convergence Controller
157     if Converged
158         % Error estimation
159         e = F*(h*d);
160 %        r = norm(e./(absTol + abs(G).*relTol),2)/sqrt(nx);
161         r = norm(e./(absTol + abs(G).*relTol),inf);
162         CurrentStepAccept = (r<=1.0);
163         r = max(r,eps);
164         stats.r(nstep) = r;
165         % Step Length Controller
166         if CurrentStepAccept
167             stats.AcceptStep(nstep) = true;
168             info.nAccept = info.nAccept+1;
169             if FirstStep || PreviousReject || ConvergenceRestriction
170                 % Aymptotic step length 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         CurrentStepAccept = 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         % Storage of variables for output
268
269         if CurrentStepAccept
270             nAccept = info.nAccept;
271             if nAccept > length(Tout)
272                 Tout = [Tout; zeros(chunk,1)];
273                 Xout = [Xout; zeros(chunk,nx)];
274                 Gout = [Gout; zeros(chunk,nx)];
275             end
276             Tout(nAccept,1) = t;
277             Xout(nAccept,:) = x.';
278             Gout(nAccept,:) = g.';
279         end
280
281
282     end
283     info.nSlowConv = length(find(stats.SlowConv)); % carsten
284     nAccept = info.nAccept;
285     Tout = Tout(1:nAccept,1);
286     Xout = Xout(1:nAccept,:);
287     Gout = Gout(1:nAccept,:);
```

Listing A.7: The code for the ESDIRK23 method.

Bibliography

[But02] John Charles Butcher. *Numerical Methods for Ordinary Differential Equations*. Second. Wiley, 2002.

[CS10] Per Grove Thomsen Carsten Völcker John Bagterp Jørgensen and Erling Halfdan Stenby. "Adaptive Stepsize Control in Implicit Runge-Kutta Methods for Reservoir Simulation." In: (2010).

[EN08] Gerhard Wanner Ernst Hairer and Syvert P. Nørsett. *Solving Ordinary Differential Equations I*. Third. Springer, 2008.

[GS97] Kjell Gustaffson and Gustaf Söderlind. "CONTROL STRATEGIES FOR THE ITERATIVE SOLUTION OF NONLINEAR EQUATIONS IN ODE SOLVERS." In: (1997).

[Jør] John Bagterp Jørgensen. $Lecture8B_{O}rderAndStabilityConditions_{E}rrorControl$. https://learn.inside.dtu.dk/d2l/le/content/103825/viewContent/423551/View.

[JT18] Morten Rode Kristensen John Bagterp Jørgensen and Per Grove Thomsen. "A FAMILY OF ESDIRK INTEGRATION METHODS." In: (2018).

[Kre15] Daniel Kressner. *Advanced Numerical Analysis*. EPFL, 2015.

[LW] Holde Lee and Jefferey Wong. *Match 361S Lecture Notes, Numerical solution of ODEs*. https://services.math.duke.edu/~holee/math361-2020/lectures/Lec7-ODEs.pdf.

[Per01] Lawrence Perko. *Differential Equations and Dynamical Systems*. Third. New York, NY, USA: Springer, 2001.

[Rac17] Chris Rackauckas. *An Intuitive Introduction For Understanding and Solving Stochastic Differential Equations*. Donald Bren School of Information and Computer Science, 2017.

[Röß10] Andreas Rößler. "RUNGE-KUTTA METHODS FOR THE STRONG APPROXIMATION OF SOLUTIONS OF STOCHASTIC DIFFERENTIAL EQUATIONS." In: (2010).

[SS18] Imre Fekete Sidafa Conde and John N. Shadid. "Embedded error estimation and adaptive step-size control for optimal explicit strong stability preserving Runge–Kutta methods." In: (2018).

[WJ20] Morten Ryberg Wahlgreen and John Bagterp Jørgensen. “Nonlinear Model Predictive Control for an Exothermic Reaction in an Adiabatic CSTR.” In: (2020).