Construction and benchmarking of adaptive parametrized linear multistep methods

Josefine Olander & Erik Jonsson-Glans

Master's thesis
2016:E49

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

A recent publication introduced a new way to define all k-step linear multistep methods of order k and $k+1$, in a parametric form that builds in variable step-size. In this framework it is possible to continuously change method and step-size, making it possible to create better behaving adaptive numerical solvers. In this thesis general numerical solvers based on this framework have been implemented, utilizing variable step-size and variable order, based on control theory and digital filters. To test and analyze the solvers, libraries of test problems, methods and filters have been implemented. In the analysis, the solvers were also compared to commercial (Matlab) solvers. The conclusion of this investigation is that the solvers show potential to become competitive in the field.

Acknowledgments

We want to thank our supervisors, Carmen Arévalo and Gustaf Söderlind, for all the inspiration and support you have given us. It has been a truly amazing opportunity to be a part of this project.

Notation and indexing

The following indexing and notation will be used in this report (a bold typeface indicates a vector):

Notation	*Definition*
$\boldsymbol{x}(t)$	The exact solution.
t_i	Time point i. The indexing is starting at 0.
t_0	The initial time point.
t_f	The final time point.
$\boldsymbol{x}_i$	The numerical approximation of $\boldsymbol{x}(t_i)$.
p	The order of a method.
k	The number of steps used by a method (k-step method).
h_i	The i:th step-size, where $h_i = t_{i+1} - t_i$.
r_i	The i:th step-size ratio, where $r_i = h_{i+1}/h_i$.
$\boldsymbol{P}_i$	The defining polynomial belonging to step i, see Section II.2.
TOL	The tolerance supplied to the solver.
θ_j	The j:th component of the parameter vector defining a particular method.
E_k	The class of explicit methods of order k, see Section II.2.
I_k	The class of implicit methods of order k, see Section II.2.
I_k^+	The class of implicit methods of order $k+1$, see Section II.2.

Special terms and abbreviations

The following abbreviations and special terms will be used in this report:

Abbreviation/Term	*Definition*
ODE	Ordinary Differential Equation
RHS-function	Right Hand Side Function
LMM	Linear Multistep Method
AB	Adams–Bashforth family of methods
AM	Adams–Moulton family of methods
BDF	Backward differentiation formula
EDF	Explicit differentiation formula
SMFP-combination	Solver-Method-Filter-Problem combination
Starter	The method/methods used in the start phase to be able to start the multistep method.
Initial predictor	Special predictor used only in the start phase.
Main predictor	The predictor used during the whole integration (except the start phase).
Reference solution	The solution we use as reference to the solution created by our solver such that we can calculate, for example, the global error. If an analytical solution exists, then this is used as reference solution, otherwise a reference solver is used to create a reference solution.
Reference solver	A solver (including settings) used to calculate a reference solution. The supplied tolerance is chosen very strict.

Contents

List of Figures

List of Tables

Part I.

Introduction

Many interesting and important problems in science and engineering, involve the task of solving initial value problems, that is, problems consisting of one initial value and one differential equation. These problems may either be the main focus of a scientific or engineering endeavor, or smaller parts of a much more daunting problem.

With a little imagination one can to conjure up a large number of scenarios where the corresponding problems either lack analytical solutions, or have analytical solutions that are too time-consuming too derive. This creates the need for ways to approximate the solution, which can be done using numerical methods.

Numerical methods for solving initial value problems have been known for a long time. One of the most famous examples, the *forward Euler method*, was introduced as early as 1768 by Leonard Euler in his *Institutionum calculi integralis* [7]. Since then, more sophisticated methods have been developed. Especially two groups have been coming to play a major role in various numerical solver implementations, namely *Runge–Kutta methods* and *linear multistep methods*, of which the latter is the focus of this thesis.

Traditional linear multistep methods utilize a pre-determined equidistant grid, on which the solution points are calculated. Since the methods only produce approximate solutions, a difference between the exact solution and the solution given by a certain method will exist. However, this error varies at the different time steps depending both on which part of the solution the method approximates, and on the size of the grid spacing. It is desirable to be able to control the size of the error throughout the integration process, which has led to various extensions of these traditional methods, and methods using an adaptive grid (variable step-size) have been constructed. When discussing this subject it is important to separate between *variable step-size linear multistep methods* and *numerical solvers using fixed step-size linear multistep methods, but that can vary the grid.* In the former case, the variability of the step-size is built into the numerical method. In the latter case, it is not, but instead a regridding is done by using interpolation.

Until now, no unified way, or framework, to construct these variable step-size linear multistep methods have existed. However, in recent work by Arévalo and Söderlind [1], such a framework was constructed. In this framework, traditional fixed step-size methods appear as special cases by restricting the step-size, to a pre-fixed value.

In addition to methods utilizing variable step-size, there is a need for a mechanism to regulate this step-size, such that a user demanded accuracy is achieved. Traditionally, this has been done using heuristic schemes lacking a sound foundation. A better approach is to utilize discrete control theory to create step-size regulators, which both have a satisfactory behavior, and can be analyzed using tools developed within the field of control theory. This has previously been done for Runge–Kutta methods with great success [8, 9]. The regulators are built using digital filters, which ensures an appropriate behavior of the step-size regulation if chosen wisely.

Nowadays, numerical solvers usually are *order-adaptive* as well, meaning that during the integration the current method is changed to a method of a different order. By changing the order in this fashion, it is possible to achieve better efficiency and accuracy. A new algorithm, based on control theory and step-size regulators, to change order mid-integration, has been proposed by Söderlind [24]. This algorithm has previously only been tested on specially crafted test data.

The purpose of this thesis is to investigate the concepts mentioned above, and in the process construct a software package containing solvers and tools that implement these concepts, and in which the user is able to choose among many numerical methods and filters. Specifically we

want to answer the following questions:

- Does the use of digital filters to control the step-size give good results when implemented in conjunction with linear multistep methods?
- Do different combinations of solvers, methods, filters and problems behave equally well, or are some combinations preferred?
- Is the proposed algorithm used for order change working as expected when implemented in a numerical solver based on linear multistep methods?
- Have the implemented solvers — based upon the concepts investigated in this thesis — the potential to compete with already widely used numerical solvers?

The work involved to answer these questions can be divided into four parts: one background part (containing both theoretical background and implementation decisions not specifically concerning step-size or order regulation), one part investigating the step-size regulation process, one part investigating the order regulation process, and a last part containing conclusions and areas of interest in future work. The first and last parts were written by both authors, whereas the other two parts were written separately.

A complete multistep solver is a complex device. It consists of many different parts that have to work smoothly together. By adjusting one part, it is easy to undo another. This has led to a tight connection between what started out as two separate theses. This and the large amount of shared theoretical background are the main reasons for this being one unified work, with two main parts: one dealing with the step-size regulation and the other one with order regulation. We must stress though, that the parts dealing with the two different regulation processes have been written separately, and the results within these parts are due to their respective author.

A major part of our work is the implementation part, in which we constructed a Matlab software package built upon a prototype which was written by Arévalo and Söderlind [1] as a proof of concept. This software package mainly consists of:

- Three types of solvers using variable step-size and fixed order
- Three corresponding solvers using variable order
- A problem library
- A method library
- A filter library

After completing the implementation part, experimental work was done, consisting in running tests on different problems. These results were used to analyze our solvers, finding strengths and weaknesses, and suggesting further development.

In addition, theoretical work has been done, consisting in filter stability analysis, and analysis coupled to the constructed solvers. Both these theoretical contributions were necessary parts for making implementation decisions, but in addition to this the analysis of the filters directly contributes to fulfilling the goal of answering the two questions related to step-size regulation; by using theoretical tools the author has immediately excluded large classes of filters with unwanted behavior.

Part II.

Background

II.1. Linear Multistep Methods

Linear multistep methods (LMMs) are a specific class of numerical methods used to solve ordinary differential equations (ODEs). We will consider initial value problems in semi-explicit form,

$$\begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \\ \boldsymbol{x}(0) = \boldsymbol{x}_0 \end{cases}, \ t \in [t_0, t_\mathrm{f}]. \tag{II.1}$$

A method belonging to this class, calculates an approximation, x_i, of the exact solution, $\boldsymbol{x}(t_i)$, to the ODE at a time point t_i. The function $\boldsymbol{f}(t, \boldsymbol{x})$ — which may be scalar or vector valued — is customarily placed on the right-hand side of the equality sign, and is therefore called *the right-hand side function* (*RHS-function*).

The idea behind these methods is to use several previously calculated values to increase the quality of the solution. Common for all multistep methods (with a fixed step-size[1]) is that they can be written on the form

$$\sum_{i=0}^{k} \alpha_{k-i} \boldsymbol{x}_{n-i} = h \sum_{i=0}^{k} \beta_{k-i} \boldsymbol{f}(t_{n-i}, \boldsymbol{x}_{n-i}), \tag{II.2}$$

where k indicates the number of previous solution points used to calculate the new solution point $\boldsymbol{x}_n$. Assuming that $\alpha_k \neq 0$, we call this a *k-step method*. Such a method can be either *explicit*, if $\beta_k = 0$, or *implicit*, if $\beta_k \neq 0$.

Let the order of consistency of a LMM be denoted p. There are multiple equivalent ways to define the order, but some of them can only be applied to fixed step-size methods. The definition used in this thesis is as follows: *An LMM is of order p, if for all systems of equations to which the solutions are polynomials of degree p or less, the method generates the exact solution.*

II.1.1. Interpolation conditions

One technique used to construct LMMs is to use *interpolation*. Here we construct a polynomial, $\boldsymbol{P}$, which interpolates another function $\boldsymbol{g}$ at several points such that

$$\forall m \in \{0, 1, \ldots, m_j\} \ : \ \boldsymbol{P}^{(m)}(t_{n-j}) - \boldsymbol{g}^{(m)}(t_{n-j}) = \boldsymbol{0}, \text{ for some } j \in \{1, 2, \ldots, k\} \tag{II.3}$$

where m_j is the highest order derivative used for interpolation in the point t_{n-j}. This kind of interpolation is called *Hermite interpolation*, or in the case when all $m_j = 0$, *Lagrange interpolation*. The constructed polynomial, $\boldsymbol{P}$, is then used as an approximation of $\boldsymbol{g}$ in the time interval $t \in [t_{n-1}, t_n]$, and the approximated solution point is defined as $\boldsymbol{x}_n = \boldsymbol{P}(t_n)$.

A famous example of such numerical methods are the Adams–Bashforth methods, which use the identity

$$\boldsymbol{x}(t_n) = \boldsymbol{x}(t_{n-1}) + \int_{t_{n-1}}^{t_n} \dot{\boldsymbol{x}}(\tau) d\tau = \boldsymbol{x}(t_{n-1}) + \int_{t_{n-1}}^{t_n} \boldsymbol{f}(\tau, \boldsymbol{x}) d\tau \tag{II.4}$$

[1]There exists multiple ways of extending LMMs so that they use a variable step-size [11, p. 109, 10, p. 397], but how this is done is outside the scope of this thesis. The formulation used in this thesis described in Section II.2 has no need of such an extension, as the formulation from the beginning is based upon a variable step-size being used.

where $\boldsymbol{f}(t, \boldsymbol{x})$ is approximated as a polynomial by using interpolation conditions on the points $\boldsymbol{x}(t_{n-j})$ with $j = \{1, 2, \dots, k\}$, and every $m_j = 0$. The true solution $\boldsymbol{x}(t_{n-j})$ is then replaced by the approximated values $\boldsymbol{x}_{n-j}$ for $j = \{0, 1, \dots, k\}$. From this algorithm, the coefficients in Equation II.2 may be derived [11, pp. 19-20].

II.1.2. Implicit collocation condition

In addition to interpolation conditions, there are *implicit collocation conditions*, that is, conditions on solution points that are not yet approximated. A numerical method constructed by an implicit collocation condition is called *implicit*; however, note that this type of condition alone is never enough to uniquely determine the polynomial $\boldsymbol{P}(t)$. In the case of LMMs, the only available implicit condition, according to Equation II.2, is

$$\dot{\boldsymbol{P}}(t_n) - \boldsymbol{f}(t_n, \boldsymbol{P}(t_n)) = 0. \tag{II.5}$$

A simple example of a linear multistep method using an implicit collocation condition is the *implicit Euler method*, which can be derived by approximating the solution by a first degree polynomial fulfilling the following collocation conditions

$$\boldsymbol{P}(t_{n-1}) - \boldsymbol{x}_{n-1} = 0, \tag{II.6}$$
$$\dot{\boldsymbol{P}}(t_n) - \boldsymbol{f}(t_n, \boldsymbol{P}(t_n)) = 0. \tag{II.7}$$

Traditionally, one area where implicit collocation conditions are extensively used, is when dealing with *implicit Runge-Kutta methods* [11, pp. 43-47]. Such methods often have more than one implicit collocation condition, unlike LMMs.

II.2. Parameterization of multistep methods

As already mentioned, this thesis is based on the theory developed by Carmen Arévalo and Gustaf Söderlind [1]. In their article they create a general approach to the construction of linear multistep methods, where variable step-size is built in from the start. Each method is characterized using a fixed number of interpolation and implicit collocation conditions that define a piecewise polynomial $\boldsymbol{P}$, here called *the defining polynomial*. The theory covers all linear k-step methods of maximal order, i.e., $p = k$ and $p = k + 1$. These methods can be divided into three classes:

Class I E_k methods. These methods are explicit and of order $p = k$.

Class II I_k methods. These methods are implicit and of order $p = k$.

Class III I_k^+ methods. These methods are implicit and of order $p = k + 1$.

Every method is associated with a parameter vector $\boldsymbol{\theta}$ with elements $\theta_i \in (-\pi/2, \pi/2]$. The dimension of this vector, n_θ, depends on the method class. Every method is uniquely determined by this vector in combination with the method class, as explained later.

The indexing of $\boldsymbol{x}$ and h is shown in Figure II.1, where $h_i = t_{i+1} - t_i$. All filled points are previously calculated, and the unfilled, x_n, is the next point to be calculated.

To further specify these classes we need some notation and a few definitions. Let $\boldsymbol{\Pi}_p$ denote the space of polynomials of degree p. We define the following:

State slack

$$\boldsymbol{s}_{n-j} = \boldsymbol{P}_n(t_{n-j}) - \boldsymbol{x}_{n-j}, \quad j = 0, \dots, k \tag{II.8}$$

Derivative slack

$$\boldsymbol{s}'_{n-j} = \dot{\boldsymbol{P}}_n(t_{n-j}) - \boldsymbol{f}(t_{n-j}, \boldsymbol{x}_{n-j}), \quad j = 0, \dots, k \tag{II.9}$$

Slack balance condition

$$\boldsymbol{s}_{n-j-1} \cos\theta_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1} \sin\theta_j = 0, \quad \theta_j \in (-\pi/2, \pi/2], \quad j = 1, \dots, k-1 \tag{II.10}$$

where $\boldsymbol{P}_n$ is the defining polynomial of the method used in the n:th point, see the illustrations in Figure II.2 and Figure II.3. With these definitions, a further specification of the three method classes can now be made:

Class I E_k methods. Explicit of order k, $n_\theta = k-1$. The defining polynomial at step n, $\boldsymbol{P}_n \in \boldsymbol{\Pi}_k$, is uniquely determined by the following conditions

$$\begin{cases} \boldsymbol{s}_{n-1} = 0 \\ \boldsymbol{s}'_{n-1} = 0 \\ \boldsymbol{s}_{n-j-1} \cos\theta_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1} \sin\theta_j = 0, \quad \theta_j \in (-\pi/2, \pi/2],\ j = 1, \dots, k-1 \end{cases} \tag{II.11}$$

Class II I_k methods. Implicit of order k, $n_\theta = k$. The defining polynomial at step n, $\boldsymbol{P}_n \in \boldsymbol{\Pi}_k$, is uniquely determined by the following conditions

$$\begin{cases} \dot{\boldsymbol{P}}_n(t_n) = \boldsymbol{f}(t_n, \boldsymbol{P}_n(t_n)) \\ \boldsymbol{s}_{n-1} \cos\theta_0 + h_{n-1}\boldsymbol{s}'_{n-1} \sin\theta_0 = 0, \quad \theta_0 \in (-\pi/2, \pi/2] \\ \boldsymbol{s}_{n-j-1} \cos\theta_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1} \sin\theta_j = 0, \quad \theta_j \in (-\pi/2, \pi/2],\ j = 1, \dots, k-1 \end{cases} \tag{II.12}$$

Class III I_k^+ methods. Implicit of order $k+1$, $n_\theta = k-1$. The defining polynomial at step n, $\boldsymbol{P}_n \in \boldsymbol{\Pi}_{k+1}$, is uniquely determined by the following conditions

$$\begin{cases} \dot{\boldsymbol{P}}_n(t_n) = \boldsymbol{f}(t_n, \boldsymbol{P}_n(t_n)) \\ \boldsymbol{s}_{n-1} = 0 \\ \boldsymbol{s}'_{n-1} = 0 \\ \boldsymbol{s}_{n-j-1} \cos\theta_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1} \sin\theta_j = 0, \quad \theta_j \in (-\pi/2, \pi/2],\ j = 1, \dots, k-1 \end{cases} \tag{II.13}$$

It can be shown that the coefficients of the defining polynomials are uniquely determined by the ratio between the step-sizes (how this is done for I_k^+ can be seen in [1], and the process is similar for the other method classes). Therefore another useful quantity, *the step-size ratio*, is introduced. It is defined as

$$r_i \equiv \frac{h_{i+1}}{h_i}. \tag{II.14}$$

and will be used throughout this report.

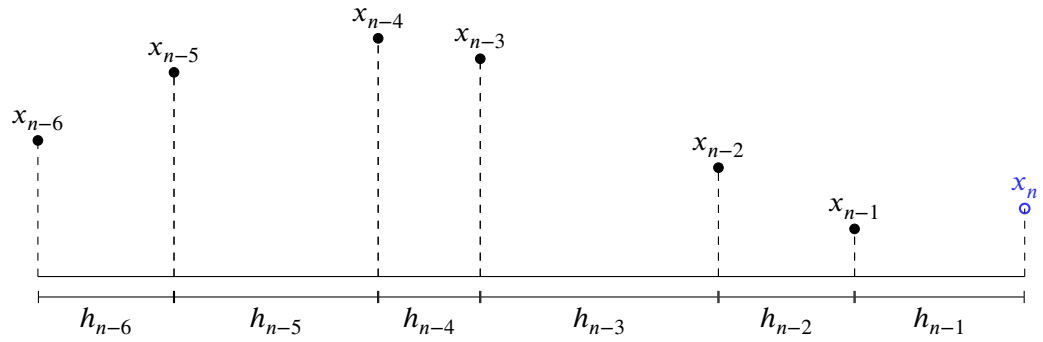

Figure II.1. The indexing of the calculated points x_i and the step-sizes h_j. The filled points are already calculated, and the unfilled one, x_n, is the next point in line to be calculated.

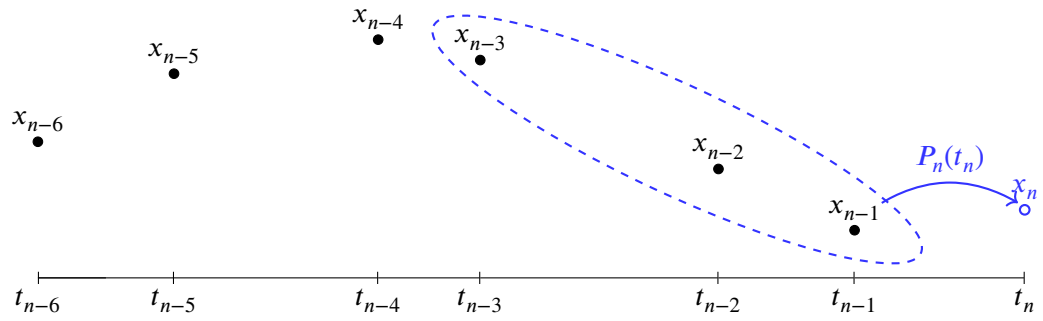

Figure II.2. *(Explicit case)* P_n is the defining polynomial used to calculate the new point x_n. It is constructed according to the conditions given in Equation II.11, using k previous solution points, and the corresponding time derivatives. Here $k = 3$. The three points inside the dashed ellipse, and the corresponding time derivatives, are used to construct P_n. To get the value of x_n, the polynomial is evaluated in the corresponding time point, i.e., $x_n \equiv P_n(t_n)$.

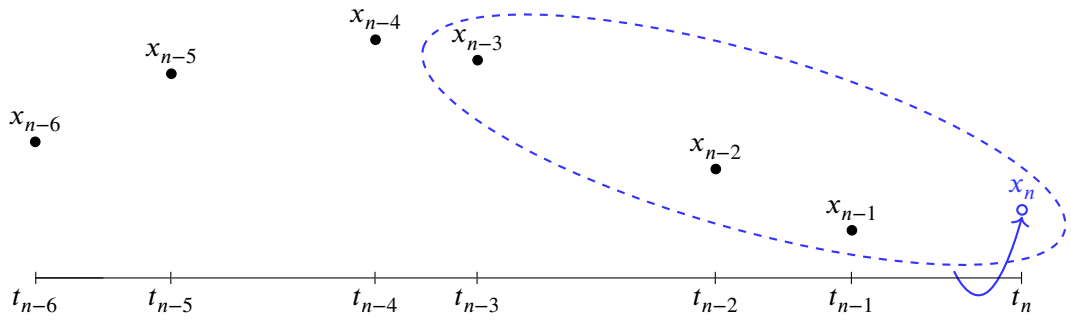

Figure II.3. *(Implicit case)* P_n is the defining polynomial used to calculate the new point x_n. It is constructed according to the conditions given in Equation II.12 or Equation II.13 depending on method class, using k previous solution points, the corresponding time derivatives, and the time derivative corresponding to x_n. Here $k = 3$. The three known points inside the dashed ellipse, their corresponding time derivatives, and the time derivative of x_n (an implicit collocation condition), is used to construct P_n. To get the value of x_n, the polynomial is evaluated in the corresponding time point, i.e., $x_n \equiv P_n(t_n)$.

II.3. Stability region of LMMs

Stability analysis is generally done for the fixed step-size case. As our solvers require very smooth step-size changes, it is possible to extend the stability analysis to our case. In the rest of the article we will assume fixed step-size when talking about stability regions.

An LMM has two *generating/characteristic polynomials* ρ and σ, defined as [10, p. 370]

$$\rho(\xi) = \alpha_k \xi^k + \alpha_{k-1}\xi^{k-1} + \cdots + \alpha_0, \tag{II.15}$$

$$\sigma(\xi) = \beta_k \xi^k + \beta_{k-1}\xi^{k-1} + \cdots + \beta_0. \tag{II.16}$$

With these two we can further construct the so-called *stability polynomial* Q of an LMM, given by [6]

$$Q(\xi, z) = \rho(\xi) - z\sigma(\xi), \tag{II.17}$$

where $\xi, z \in \mathbb{C}$.

A crucial requirement for a method to be usable, is *zero-stability*, since only then is it possible for the numerical solution to converge to the exact solution when the step-size tends to zero. An LMM is called *zero-stable*, if the following two conditions are satisfied:

1. The roots of ρ lie on or within the unit circle.
2. The roots of ρ on the unit circle are simple.

Another important concept is *absolute stability*. Belonging to every method is a region of *absolute stability*, $\mathscr{R}$. This is a region in the complex plane, basically telling us how to choose our step-size h, such that the result stays stable, meaning that the error is kept at a reasonable value throughout the integration. This notion is derived using the test equation $\dot{x} = \lambda x$, and defined as all $z = \lambda h$ in the complex plane, where every root ξ_i of the stability polynomial Q — corresponding to the LMM in question — satisfies $|\xi_i| \leq 1$ [6]. Stiff problems require methods with large stability regions so that the step-size is not restricted by stability requirements.

The stability region of a method, can be depicted by drawing the corresponding *root locus curve*. This is the set of all $z = \lambda h$ in the complex plane, where exactly one root ξ_i of the stability polynomial Q satisfies $|\xi_i| = 1$, and the other roots satisfy $|\xi_j| < 1$. This is a closed curve, constituting the boundary of the stability region, given by

$$z = \frac{\rho(e^{-i\phi})}{\sigma(e^{-i\phi})}, \quad \phi = [0, 2\pi[. \tag{II.18}$$

The root locus curve divides the complex plane into two or more areas — more than two when the curve is crossing itself. To deduce which belong to the stability region, one has to check the roots of the stability polynomial Q in all areas. As said before, the roots shall satisfy $|\xi_i| \leq 1$ inside the stability region. As an example we show the root locus curves and the stability regions for a few explicit LMMs (see Figure II.4 and Figure II.5).

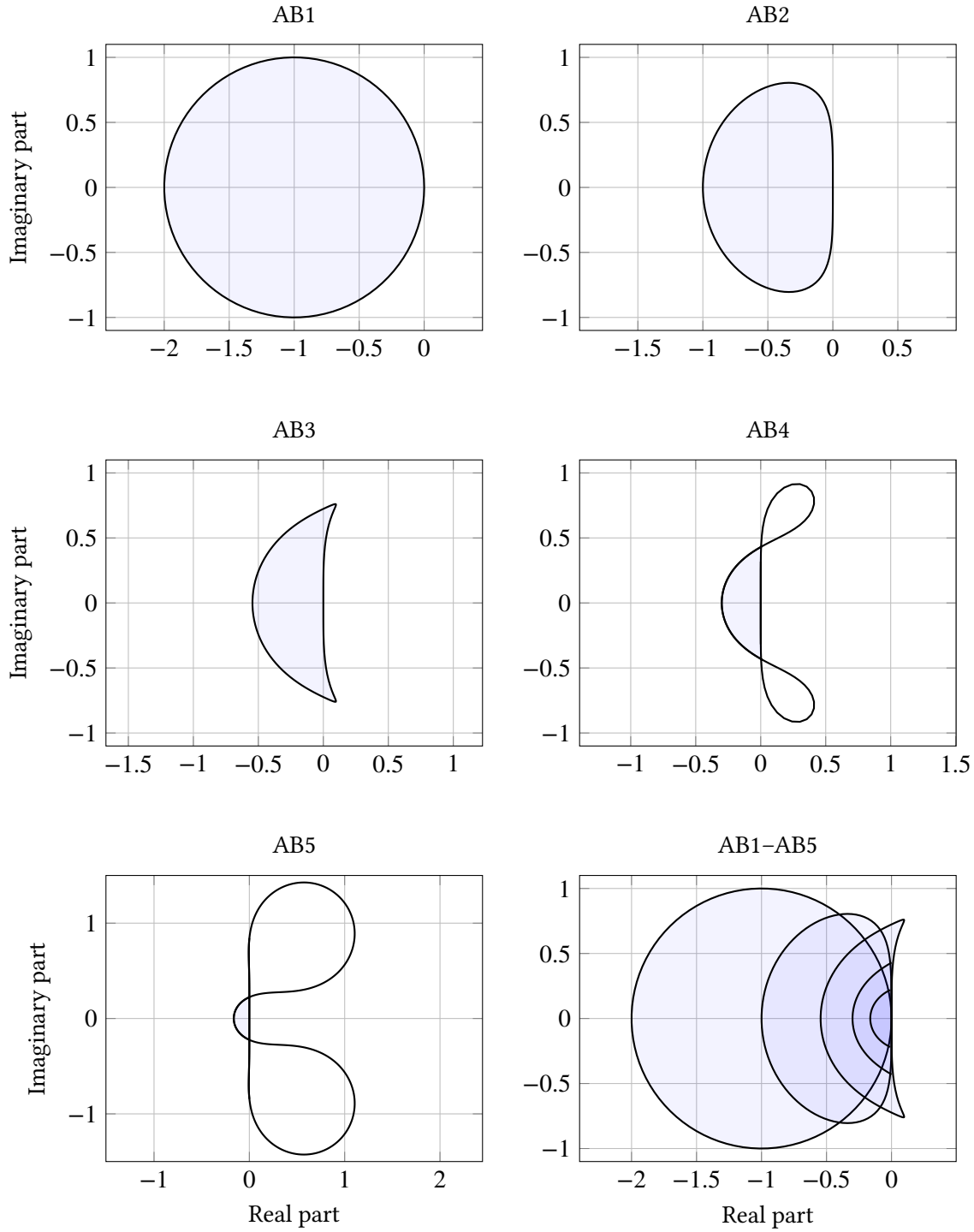

Figure II.4. The first five figures show the root locus curves and the stability regions for AB1–AB5. The root locus curves are the black lines, while the stability regions are the shaded areas only. The last figure (row 3, col 2) shows the stability regions related to each other. The higher the order of the AB-method, the smaller the stability region. Notice the special y-scaling in the sub-figure showing AB5.

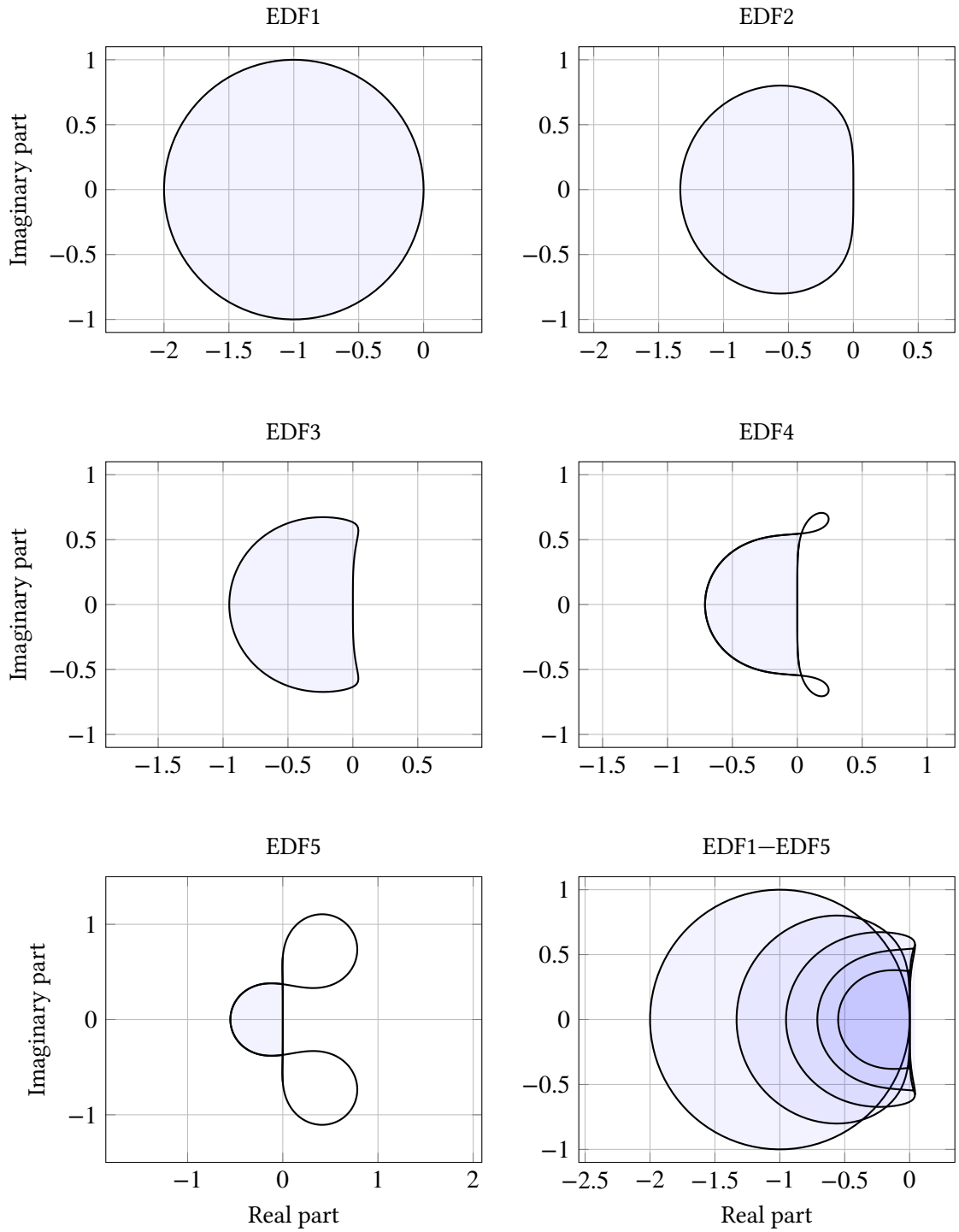

Figure II.5. The first five figures show the root locus curves and the stability regions for the family of methods EDF1–EDF5 [2]. The root locus curves are the black lines, while the stability regions are the shaded areas only. The last figure (row 3, col 2) shows the stability regions related to each other. The higher the order of the EDF-method, the smaller the stability region. Notice the special y-scaling in the sub-figure showing EDF5.

II.4. Errors and error constants of LMMs

II.4.1. Types of errors

In the context of multistep methods there are three types of errors which are of interest [20]:

- The global error
- The local error
- The local truncation error

The *global error* at a point is the difference between the solution obtained by the solver and the true solution at the same point. Due to errors introduced in each point the solution will drift from the true solution. The global error is the quantity one usually wants to control; however, due to the error accumulation, it is hard to do so [20].

Instead the local behavior is monitored. The *local error* at a point is the difference between the solution obtained by the solver, and the solution to the differential equation which passes through the previously calculated point. By controlling this error the global error may indirectly be controlled, as the global error is an accumulation of local errors. However, since a multistep method uses more than one previously calculated point, and these points generally do not belong to the same solution curve, a direct estimation of this quantity will be hard make. Instead, one normally estimates *the local truncation error*, which is the local error assuming that all k previously calculated solution points are exact. In this thesis when the term *local error* is used it always refers to *the local truncation error*.

II.4.2. Error constants in the fixed step-size case

When creating a linear multistep method, there is a lot of different properties to consider. One important accuracy property is *the order of the method*, p, which tells us at what rate the error tends to zero as h tends to zero. However, it is also important to consider *the error constant* C, which is a measure of the magnitude of the principal error term (see first term on the RHS in II.19).

It can be shown that the local error $\boldsymbol{x}(t_i) - \boldsymbol{x}_i$ at time point t_i of an LMM can be written as [10, p. 372]

$$\boldsymbol{x}(t_i) - \boldsymbol{x}_i = \alpha_k^{-1} C_{p+1} h^{p+1} \boldsymbol{x}^{(p+1)}(t_{i-k}) + \mathcal{O}(h^{p+2}), \tag{II.19}$$

assuming that the k previous solution points $\boldsymbol{x}_{i-k}, \boldsymbol{x}_{i-k+1}, \ldots, \boldsymbol{x}_{i-1}$ are exact. In the constant step-size case C_{p+1} is given by

$$C_{p+1} = \frac{1}{(p+1)!} \left(\sum_{i=0}^{k} \alpha_i i^{p+1} - (p+1) \sum_{i=0}^{k} \beta_i i^p \right), \tag{II.20}$$

where k is the number of steps used by the method.

The constant C_{p+1} is often used when measuring the global error, though not directly, one usually scales it according to [10, p. 373]

$$C = \frac{C_{p+1}}{\sigma(1)}. \tag{II.21}$$

In Figure II.6 we see the value of the scaled error constant C for a few different method classes.

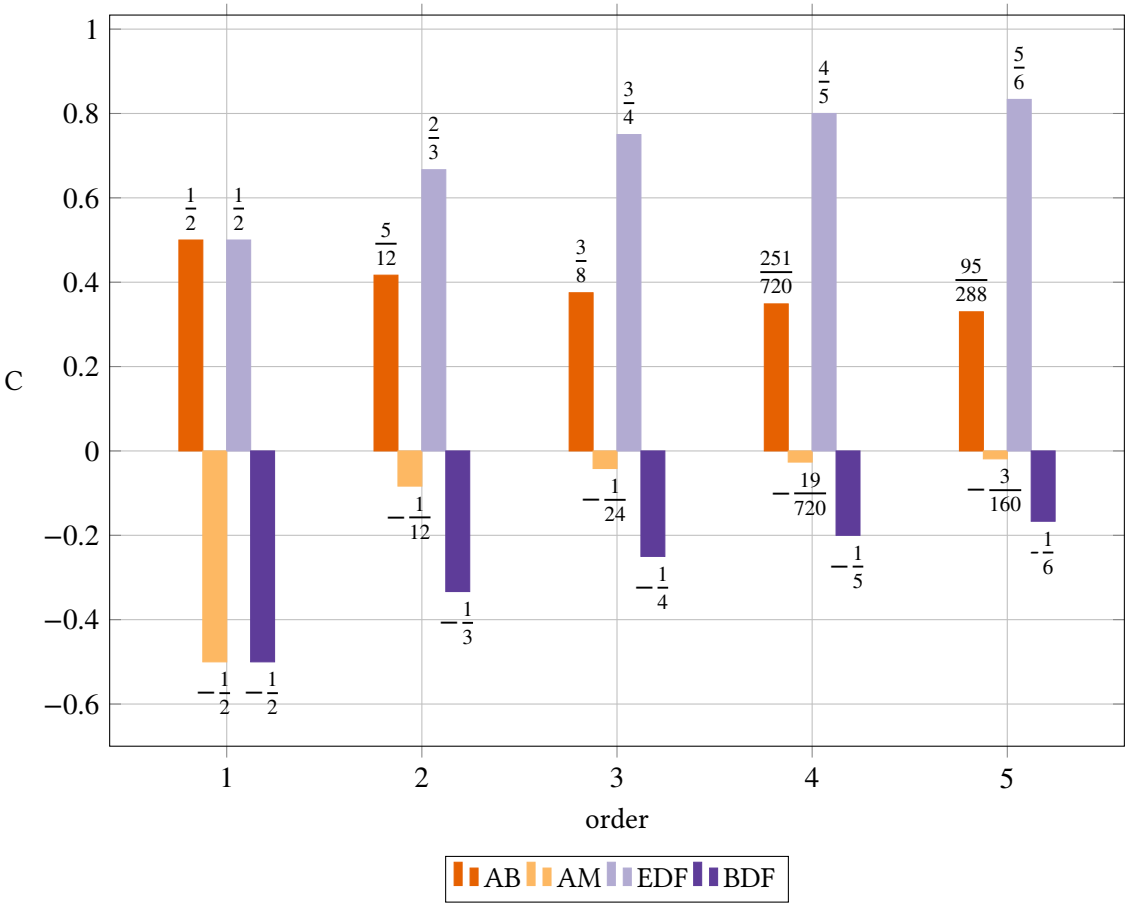

Figure II.6. The scaled error constants C for methods of order 1 to 5 of the method families AB (Adams—Bashforth), AM (Adams—Moulton), EDF (Explicit Differentiation Formula) [2] and BDF (Backward Differentiation Formula).

II.4.3. Error constants in the variable step-size case

The concepts above can also be extended to cover LMMs with variable step-size. We will consider methods written on the following form

$$\sum_{i=0}^{k} \alpha_{n,k-i} \boldsymbol{x}_{n-i} = h_{n-1} \sum_{i=0}^{k} \beta_{n,k-i} \boldsymbol{f}(t_{n-i}, \boldsymbol{x}_{n-i}) \tag{II.22}$$

where the coefficients, $\alpha_{n,k-i}$ and $\beta_{n,k-i}$, now depend on the step-size ratios given by the step-size sequence $\{h_j\}_{j=n-k}^{n-1}$.

We then define the *linear difference operator*, belonging to a method as

$$\mathscr{L}(\boldsymbol{x}, t, \boldsymbol{h}) \equiv \sum_{i=0}^{k} \left(\alpha_{n,k-i} \boldsymbol{x} \left(t - \sum_{j=1}^{i} \frac{h_{n-1}}{R_{n-j}^{n-2}} \right) - h_{n-1} \beta_{n,k-i} \dot{\boldsymbol{x}} \left(t - \sum_{j=1}^{i} \frac{h_{n-1}}{R_{n-j}^{n-2}} \right) \right) \tag{II.23}$$

where $\boldsymbol{h}$ is a vector containing the elements $\{h_j\}_{j=n-k}^{n-1}$, and R_i^j is defined as

$$R_i^j \equiv \frac{h_{j+1}}{h_i} = \Pi_{k=i}^{j} r_k. \tag{II.24}$$

By using Taylor expansion around t we get

$$\mathscr{L}(\boldsymbol{x}, t, \boldsymbol{h}) = \sum_{i=0}^{k} \left(\alpha_{n,k-i} \sum_{q \geq 0} \frac{\left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^q}{q!} h_{n-1}^q \boldsymbol{x}^{(q)}(t) \right. \tag{II.25}$$

$$\left. -\beta_{n,k-i} \sum_{q \geq 0} \frac{\left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^q}{q!} h_{n-1}^{q+1} \boldsymbol{x}^{(q+1)}(t) \right) = \sum_{i=0}^{k} \alpha_{n,k-i} \boldsymbol{x}(t) + \tag{II.26}$$

$$+ \sum_{q \geq 1} \frac{\boldsymbol{x}^{(q)}}{q!} h_{n-1}^q \left(\sum_{i=0}^{k} \alpha_{n,k-i} \left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^q - q \sum_{i=0}^{k} \beta_{n,k-i} \left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^{q-1} \right).$$

For a method of order p, this means that all terms where $q \leq p$ disappear, and we may rewrite the expression above as

$$\mathscr{L}(\boldsymbol{x}, t, \boldsymbol{h}) = \boldsymbol{x}^{(p+1)}(t) h_{n-1}^{p+1} \tilde{C}_{p+1} + \mathcal{O}(h_{n-1}^{p+2}) \tag{II.27}$$

where $\tilde{C}_{p+1}$ is a constant, only depending on the method and the step-size ratios, given by [2]

$$\tilde{C}_{p+1} = \left(\sum_{i=0}^{k} \frac{\alpha_{n,k-i}}{(p+1)!} \left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^{p+1} - \sum_{i=0}^{k} \frac{\beta_{n,k-i}}{p!} \left(-\sum_{j=1}^{i} \frac{1}{R_{n-j}^{n-2}} \right)^{p} \right). \tag{II.28}$$

The following lemma states an important fact about the local error:

Lemma II.4.1. *Assume that*

$$\boldsymbol{x}_{n-i} = \boldsymbol{x} \left(t - \sum_{j=1}^{i} \frac{h_{n-1}}{R_{n-j}^{n-2}} \right), \quad \text{for } 0 < i \leq k,$$

i.e., the previous solution points are exact. Then

$$\boldsymbol{x}(t_n) - \boldsymbol{x}_n = \left(\alpha_{n,k} \mathrm{I} - h_{n-1} \beta_{n,k} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(t_n, \boldsymbol{\eta}) \right)^{-1} \mathscr{L}(\boldsymbol{x}, t_n, \boldsymbol{h}), \tag{II.29}$$

where $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(t_n, \boldsymbol{\eta})$ is the Jacobian of $\boldsymbol{f}$, row-wise evaluated at possibly different points on the line segment $\boldsymbol{x}(t_n) - \boldsymbol{x}_n$.

[2] Note that this error constant will not be the same, even for the fixed step-size case, as the one used in Equation II.19. This difference is due to the higher order derivative being evaluated in different points.

Proof. The proof follows the same outline as used for fixed step-size methods in [10, p. 369]. By using the assumption that the previous solution points are exact, and inserting Equation II.23 into Equation II.22, we get

$$\mathscr{L}(\boldsymbol{x}, t_n, \boldsymbol{h}) = \alpha_{n,k}(\boldsymbol{x}(t_n) - \boldsymbol{x}_n) - h_{n-1}\beta_{n,k}\left(\boldsymbol{f}(t_n, \boldsymbol{x}(t_n)) - \boldsymbol{f}(t_n, \boldsymbol{x}_n)\right) \tag{II.30}$$

Now, by applying the mean value theorem (in the case of $\boldsymbol{f}$ being a vector valued function, this is done element-wise, where each vector element is regarded as a scalar function of multiple variables) we get

$$\mathscr{L}(\boldsymbol{x}, t_n, \boldsymbol{h}) = \alpha_{n,k}(\boldsymbol{x}(t_n) - x_n) - h_{n-1}\beta_{n,k}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(t_n, \boldsymbol{\eta})(\boldsymbol{x}(t_n) - \boldsymbol{x}_n) \tag{II.31}$$

$$\Longleftrightarrow$$

$$\boldsymbol{x}(t_n) - \boldsymbol{x}_n = \left(\alpha_{n,k}\mathrm{I} - h_{n-1}\beta_{n,k}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}(t_n, \boldsymbol{\eta})\right)^{-1}\mathscr{L}(\boldsymbol{x}, t_n, \boldsymbol{h}) \tag{II.32}$$

□

By combining Equation II.27 and II.29, we can approximate the local error for small h_{n-1} as

$$\boldsymbol{x}(t_n) - \boldsymbol{x}_n \approx \alpha_{n,k}^{-1}\tilde{C}_{p+1}h_{n-1}^{p+1}\boldsymbol{x}^{(p+1)}(t_n). \tag{II.33}$$

II.5. Demands on a solver

When designing an initial value problem (IVP) solver — that is, stand-alone software used to solve IVPs — a number of things need to be taken into consideration for the end result to turn out well. The different demands we have on a solver often affect each other and trade-offs have to be made, so a list of demands and their priority is of great importance. There are different opinions on this subject, but in this section we present and argue for our view. The following list contains what we regard as the most important issues, in order of importance:

1. Stability
2. Accuracy
3. Robustness
4. Tolerance proportionality
5. Efficiency
6. User friendliness

II.5.1. Stability

First and foremost in the list is stability. Stability is the most important aspect, because without it the approximate solution will not converge to the exact solution. If the underlying method is not zero-stable we can not achieve convergence [10, p. 392], which is the aim of any numerical

method. Nevertheless, not only the numerical method needs to be stable, but also the implementation of all parts of the solver need to work stably together. For example, step-sizes must be chosen such that they remain inside the stability region.

II.5.2. Accuracy

A numerical solution needs to have a certain degree of accuracy for it to be useful. For example, let us assume that we have a solver that always returns the value 1. This solver is stable and efficient, however not useful since it says that the solution to every problem is 1 at all times. Therefore, an inability to achieve good accuracy makes for a poor solver.

Another important aspect on the subject accuracy, is for the user to have the ability to change the degree of accuracy, since a decrease in demanded accuracy means that less work has to be done. In some situations the user might need quick results, but not a high degree of accuracy, and in other cases as high an accuracy as possible might be needed.

II.5.3. Robustness

Here the term *robustness* or *computational stability* is used to mean that small changes in the experimental setup (such as using different compilers or underlying platforms) will not introduce large changes in the result, and as has previously been argued in [27], this improves reproducibility and is therefore important when evaluating numerical software. A good example of robustness is the ability to give trustworthy results even when the tolerance is adjusted. For example, when we choose a stricter tolerance, we want the accuracy of the solution to improve and the work to increase. If this is not the case, then the solver is said to be non-robust. We argue that this also has value in production environments, where deployment in different hardware or software settings may be performed or where underlying infrastructure may be replaced with time.

II.5.4. Tolerance proportionality

As a user of a particular numerical solver, the only tool you may have to influence the accuracy of the solution, is the value of the tolerance. What a user expects is that when the supplied tolerance is made more stringent, the solution will be more accurate. A good solver should be able to recreate the relation

$$\log(E) = k \log(\mathrm{TOL}) + \log(A), \tag{II.34}$$

where E is the global error of the solution, $k \in \mathbb{R}^+$, $\log(A) \in \mathbb{R}$ and TOL is the supplied tolerance. Optimally $k = A = 1$, though if this is not the case it is easy to scale and calibrate the tolerance internally such that it becomes the case. As long as the global error produced by the solver is proportional to $\mathrm{TOL}_\mathrm{e}^\alpha$ — where the index 'e' stands for *external*, i.e., the supplied tolerance, and $\alpha \in \mathbb{R}^+$ — this is simply a matter of rescaling the tolerance according to

$$\mathrm{TOL}_1 = \mathrm{TOL}_\mathrm{e}^{1/\alpha}, \tag{II.35}$$

and use TOL_1 internally. This will theoretically result in the relation

$$\log(E) = \log(\mathrm{TOL}) + \log(B), \tag{II.36}$$

where $\log(B) \in \mathbb{R}$. By scaling the tolerance by B according to

$$\mathrm{TOL}_\mathrm{i} = \frac{\mathrm{TOL}_1}{B}, \tag{II.37}$$

the new internally used tolerance TOL_i should result in the relation

$$\log(E) = \log(\mathrm{TOL}_\mathrm{e}) \iff E = \mathrm{TOL}_\mathrm{e}. \tag{II.38}$$

II.5.5. Computational efficiency

Needless to say there are multiple reasons for wanting fast computations with a low demand on resources. However, this aspect has to take a backseat, since the previous ones affect the degree of confidence in the computational result, whereas this does not. However, this is of course not unimportant. One can easily imagine multiple situations where a lack of efficiency renders a solver unsatisfactory for a certain task.

II.5.6. User friendliness

Instead of computational resources, this issue deals with human resources. Whether or not computational efficiency should be prioritized over user friendliness is largely a matter of the specific use case of the solver. In many situations, a user is not interested in the details of the code, but only in a trustworthy, easy-to-use tool to solve the real problem at hand. By creating a solver that lacks user friendliness, one problem at hand for the user might become two.

A factor we stress here, is that user friendliness in many situations may not only involve how easy to use the solver is, but also how easily the code can be modified. This is an area where optimizing the code for computational efficiency often results in a code base which is harder to read and modify. We argue that a code like the one constructed in this thesis, meant for numerical research purposes, should be made as modular as possible, such that special parts of the code (for example, the step-size control) can be replaced easily. Using a modular code like this, it is easy to make comparisons between for example *error control system 1* and *error control system 2*. Instead of doing a lot of rewriting, two tests using the same main solver, but two different error control systems can be made, by only changing the function name of the control system in the main solver. This will not only make a change in error control system faster, but also make the tests more reliable since the rest of the code is untouched.

II.6. Error estimation

To be able to control the step-size in an ODE solver, the solver needs an indication of the size of the error, such that it can decide whether to decrease or increase the step-size. This indication is

given by an *error estimator* which produces an *error estimate*. As has been discussed in Section II.4, the quantity that is normally monitored is the local error. The rest of this section will describe how this error estimation process works.

II.6.1. Construction of error estimators

The estimation of the error follows the same basic principle in all three types of solvers considered here. The polynomial constructed to calculate the previous step, $\boldsymbol{P}_{n-1}(t)$, is evaluated at t_n to create a prediction value, $\boldsymbol{x}_n^{\text{pred}} = \boldsymbol{P}_{n-1}(t_n)$. The norm of the difference between this value and the solution value, $\boldsymbol{x}_n = \boldsymbol{P}_n(t_n)$, is then used as an error estimate

$$e_n = \|\boldsymbol{x}_n - \boldsymbol{x}_n^{\text{pred}}\|. \tag{II.39}$$

Although this principle is used for all three solvers, we will show that the underlying mechanisms making this a possible error estimate are different.

II.6.2. Error estimation for explicit solvers

The following part will, for the sake of clarity, only deal with the one-dimensional case, however the arguments are easily extended to the multidimensional case.

In the case of an explicit method used in combination with the error estimator described in Equation II.39, the evaluation of the previous polynomial $\boldsymbol{P}_{n-1}$ in the new time point t_n, is equivalent to using the same method as in point $n-1$ but taking a longer step. This is due to the lack of implicit conditions, and therefore only previously calculated solution points will affect the construction of $\boldsymbol{P}_{n-1}(t)$. The step-size used to calculate the prediction value will be given by

$$h^{\text{pred}} = h_{n-1} + h_n = \left(\frac{1}{r_{n-1}} + 1\right) h_n. \tag{II.40}$$

Let $x(t_n)$ denote the exact solution for the problem with initial condition $x(t_{n-2}) = x_{n-2}$. Now, under the assumption that all previous solution points, including x_{n-1}, are exact, the local errors for the new solution point x_n and the predictor point x^{pred} can be modeled as

$$\begin{cases} x(t_n) - x_n = \tilde{K} h_n^{p+1} & \text{(II.41a)} \\ x(t_n) - x_n^{\text{pred}} = \hat{K}\left(h_n^{\text{pred}}\right)^{p+1} = \hat{K}\left(\dfrac{1}{r_{n-1}} + 1\right)^{p+1} h_n^{p+1} & \text{(II.41b)} \end{cases}$$

by using the approximation of the local error in Equation II.33, where $\tilde{K}$ and $\hat{K}$ are constants given by (note that the α-coefficients will have different indexation, due to originally being used to calculate the solution at different time-steps)

$$\tilde{K} = \alpha_{n,k}^{-1} \tilde{C}_{p+1} x^{(p+1)}(t_n), \tag{II.42}$$

$$\hat{K} = \alpha_{n-1,k}^{-1} \hat{C}_{p+1} x^{(p+1)}(t_n), \tag{II.43}$$

and $\tilde{C}_{p+1}$ is the error constant belonging to the method with polynomial P_n and $\hat{C}_{p+1}$ is the error constant belonging to the method with polynomial P_{n-1} (since they are depending on the step-size ratio, they will not be constant throughout the integration).

Subtracting Equation II.41a from Equation II.41b results in

$$x_n - x_n^{\text{pred}} = \hat{K}\left(\frac{1}{r_{n-1}} + 1\right)^{p+1} h_n^{p+1} - \tilde{K} h_n^{p+1} = \left(\frac{\hat{K}}{\tilde{K}}\left(\frac{1}{r_{n-1}} + 1\right)^{p+1} - 1\right)\tilde{K} h_n^{p+1} \tag{II.44}$$

$$\Longleftrightarrow$$

$$\frac{x_n - x_n^{\text{pred}}}{\frac{\hat{K}}{\tilde{K}}\left(\frac{1}{r_{n-1}} + 1\right)^{p+1} - 1} = \tilde{K} h_n^{p+1}. \tag{II.45}$$

By substituting this back into Equation II.41a we get

$$x(t_n) - x_n = \frac{x_n - x_n^{\text{pred}}}{\frac{\hat{K}}{\tilde{K}}\left(\frac{1}{r_{n-1}} + 1\right)^{p+1} - 1}. \tag{II.46}$$

This can be compared to Equation II.39, where the error estimate is set to $\|x_n - x_n^{\text{pred}}\|$.

Now, under the assumption that $r_{n-1} \approx 1$, which is reasonable to assume for a well controlled process, this becomes

$$x(t_n) - x_n \approx \frac{\tilde{K}}{\hat{K} \cdot 2^{p+1} - \tilde{K}}(x_n - x_n^{\text{pred}}) \tag{II.47}$$

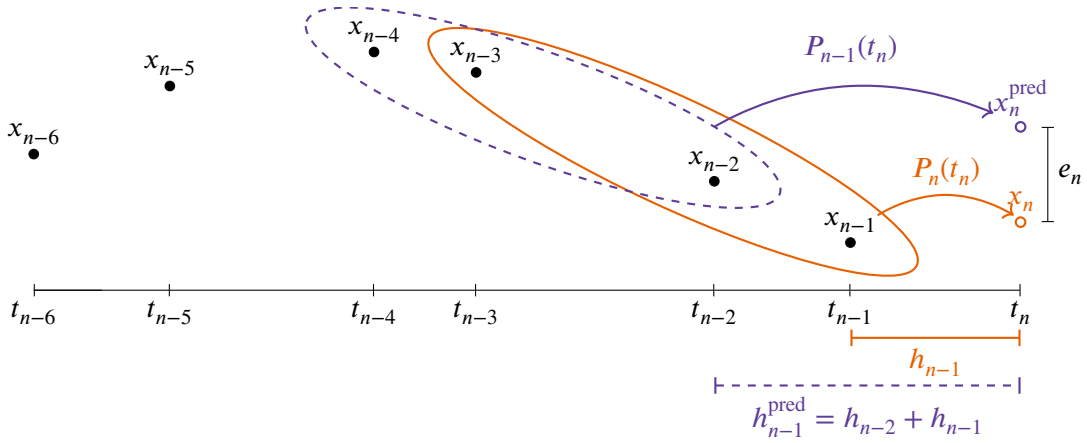

Figure II.7. An example showing how the error is estimated for a 3-step explicit method is calculated. First the point x_{n-1} is calculated by using the points encircled by the dashed line. Then the same points are used to calculate another solution point, x_n^{pred}. Then the correct new solution point, x_n, is calculated by using the points encircled by the solid line. The distance, e_n, between x_n and x_n^{pred} is calculated by using some appropriate norm, and this is then used as the error estimate.

An illustration of this error estimation process can be seen in Figure II.7.

II.6.3. Error estimation for implicit solvers

The implicit solvers use a different mechanism for estimating the error. Instead of using the same method, as in the case of the explicit solvers, the estimation process will be equivalent to using an explicit method of the same order to calculate the prediction value, which is essentially a *Milne device*. Why this is the case will be stated below in the form of two theorems with accompanying proofs.

Theorem II.6.1. *Let I_k be an implicit k-step method of order k, defined by the parameter vector $\boldsymbol{\theta}$. Then the way of estimating the error described in Equation II.39, is equivalent to using an explicit method of order k to calculate the prediction value. The method's parameter vector, $\hat{\boldsymbol{\theta}}$, is the vector corresponding to the first $k-1$ elements of $\boldsymbol{\theta}$.*

Proof. Let $\boldsymbol{P}_m(t)$ be the defining polynomial of I_k, constructed to calculate the solution $\boldsymbol{x}_m$ at time t_m. We will now show that this polynomial fulfills all conditions for an E_k-method (Equation II.11), with the change of index $n = m+1$. A polynomial fulfilling the conditions in Equation II.11 is guaranteed to be unique, and therefore $\boldsymbol{P}_m(t)$ is the same polynomial one would have got if it was constructed by using the conditions for the explicit method.

The slack condition is trivial, because $\boldsymbol{x}_m = \boldsymbol{P}_m(t_m)$ we have

$$\boldsymbol{s}_{n-1} = \boldsymbol{x}_{n-1} - \boldsymbol{P}_m(t_{n-1}) = \boldsymbol{x}_{n-1} - \boldsymbol{x}_m = \boldsymbol{x}_{n-1} - \boldsymbol{x}_{n-1} = 0. \tag{II.48}$$

The slack derivative condition is also fulfilled, because from the implicit collocation condition in Equation II.12 we have $\dot{\boldsymbol{P}}_m(t) = \boldsymbol{f}(t_m, \boldsymbol{x}_m)$, which gives us

$$\boldsymbol{s}'_{n-1} = \dot{\boldsymbol{P}}_m(t_{n-1}) - \boldsymbol{f}(t_{n-1}, \boldsymbol{x}_{n-1}) = \boldsymbol{f}(t_{n-1}, \boldsymbol{x}_{n-1}) - \boldsymbol{f}(t_{n-1}, \boldsymbol{x}_{n-1}) = 0. \tag{II.49}$$

The slack balance conditions are also fulfilled

$$\boldsymbol{s}_{n-j-1}\cos\theta_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1}\sin\theta_j = \boldsymbol{s}_{m-j}\cos\theta_j + h_{m-j}\boldsymbol{s}'_{m-j}\sin\theta_j = 0, \quad j = 1, \dots, k-1 \tag{II.50}$$

where the last equality follows from the slack balance conditions in Equation II.12. Lastly, the polynomial is of degree k, which gives the explicit method the order k. □

One thing to be noted is that in Equation II.50 the index j could have been allowed to take the value k as well, but because this does not increase the degree of the polynomial, no order is gained from this.

Theorem II.6.2. *Let I_k^+ be an implicit k-step method of order $k+1$, defined by the parameter vector $\boldsymbol{\theta}$. Then the way of estimating the error described in Equation II.39 is equivalent to using an explicit method of order $k+1$ defined by $\hat{\boldsymbol{\theta}}$ to calculate the prediction value, where the first component in $\hat{\boldsymbol{\theta}}$ can be arbitrarily chosen, and the rest of the components are the same as the ones in $\boldsymbol{\theta}$.*

Proof. Let $\boldsymbol{P}_m(t)$ be the polynomial constructed to calculate the solution $\boldsymbol{x}_m$ at time t_m. We will now show that this polynomial fulfills all conditions in Equation II.11, with the change of index $n = m+1$. A polynomial fulfilling the conditions in Equation II.11 is guaranteed to be unique,

and therefore $\boldsymbol{P}_m(t)$, is the same polynomial one would have got if it was constructed by using the conditions for the explicit method.

In the same way as in the proof of Theorem II.6.1 we get that

$$\boldsymbol{s}_{n-1} = 0 \tag{II.51}$$
$$\boldsymbol{s}'_{n-1} = 0 \tag{II.52}$$
$$\boldsymbol{s}_{n-j-1} \cos\hat{\theta}_j + h_{n-j-1}\boldsymbol{s}'_{n-j-1} \sin\hat{\theta}_j = 0, \quad j \in [2, k] \tag{II.53}$$

from Equation II.13. It remains to show that the slack balance condition for $j = 1$ is fulfilled.

The interpolation conditions in Equation II.13 forces $\boldsymbol{P}_m(t)$ to both pass through, $\boldsymbol{x}_{n-2}$ and have the same derivative $\dot{\boldsymbol{P}}_m(t_{n-2}) = \boldsymbol{f}(t_{n-2}, \boldsymbol{x}_{n-2})$ in this point. This is the point used in the first slack balance condition in Equation II.11. This in turn means that in this case the chose of parameter for this slack balance condition does not matter. The first slack balance condition is automatically fulfilled due to how the sequence of solution points was calculated.

Describing this with formulas

$$\boldsymbol{s}_{m-1} = \boldsymbol{s}_{n-2} = 0, \tag{II.54}$$
$$\boldsymbol{s}'_{m-1} = \boldsymbol{s}'_{n-2} = 0, \tag{II.55}$$

which gives us

$$\boldsymbol{s}_{n-2} \cos\hat{\theta}_1 + h_{n-2}\boldsymbol{s}'_{n-2} \sin\hat{\theta}_1 = 0 \cdot \cos\hat{\theta}_1 + h_{n-2} \cdot 0 \cdot \sin\hat{\theta}_1 = 0, \tag{II.56}$$

and therefore the slack balance condition is fulfilled, regardless of the choice of $\hat{\theta}_1$. The order follows from the degree of $\boldsymbol{P}_m(t)$ which is $k + 1$, and therefore gives the order $k + 1$. □

II.7. Construction of test problems

When working with ODEs it might be hard to test your solver on them due to the fact that most of the usual ODEs used for tests have no analytical solution. If you for example want to investigate the global error created by your solver, your *reference solution* will be the solution created by another (or the same) solver using very tight tolerances. Nevertheless, since this is not the real solution, you can not always be certain that the results reflect reality. Sometimes it might be the *reference solver* doing a bad job. A good idea in these cases, is to use ODEs with known analytical solutions. For this purpose, it is possible to construct ODEs, such that we know the analytical solution and at the same time are able to change the stiffness and nonlinearity of the system [26].

The method consists in choosing a function $\boldsymbol{f}(t)$, and constructing the following IVP:

$$\begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{\varphi}(\boldsymbol{x} - \boldsymbol{f}(t)) + \dot{\boldsymbol{f}}(t), \\ \boldsymbol{\varphi}(\boldsymbol{y}) = \mu A\boldsymbol{y} + \gamma \boldsymbol{g}(\boldsymbol{y}), \\ \boldsymbol{g}(0) = \boldsymbol{0}, \\ \boldsymbol{x}(0) = \boldsymbol{f}(0), \end{cases} \tag{II.57}$$

where $\mu, \gamma \in \mathbb{R}$, $A \in \mathbb{R}^{n\times n}$ and is constant, $\boldsymbol{g}$ is a function of the form $\boldsymbol{g} : \mathbb{R}^{n\times 1} \to \mathbb{R}^{n\times 1}$ and $\boldsymbol{y} \in \mathbb{R}^{n\times 1}$. It can be shown that the solution to this IVP is $\boldsymbol{x} = \boldsymbol{f}(t)$ by testing it:

$$\begin{aligned}\dot{\boldsymbol{x}} &= \boldsymbol{\varphi}(\boldsymbol{f}(t) - \boldsymbol{f}(t)) + \dot{\boldsymbol{f}}(t)\\ &= \boldsymbol{\varphi}(0) + \dot{\boldsymbol{f}}(t)\\ &= \dot{\boldsymbol{f}}(t)\\ &= \dot{\boldsymbol{x}},\\ \boldsymbol{x}(0) &= \boldsymbol{f}(0)\end{aligned}$$

The first term in $\boldsymbol{\varphi}$ is the linear term. By changing the matrix A and the scalar μ, you are able to change the stiffness of the problem. This due to the fact that the eigenvalues of the system will change. The second term is the nonlinear one. By changing $\boldsymbol{g}$ and γ you are able to change the nonlinearity of the system.

When using an implicit solver, we need the Jacobian of the problem in question. The Jacobian to Equation II.57 is given by

$$J = J_{\boldsymbol{\varphi}}(\boldsymbol{y}) = A\mu + \gamma J_{\boldsymbol{g}}(\boldsymbol{y}). \tag{II.58}$$

II.8. Construction of the solvers

One essential part of this thesis, is the implementation part. This section explains the most crucial parts of our solvers (except the changing order part) in a structured way. For more information about the structuring of our Matlab package see Section A.1.

Our Matlab software package consists of three fixed order solvers and corresponding three variable order solvers. Those of variable order are built on the same principles as those of fixed order, with the addition that they use one more *phase*, the *order control phase*. More about how the order control works can be read in Section IV.2. This phase was developed entirely by the second author.

The six solvers are called `pmme`, `pmmeVarOrd`, `pmmi`, `pmmiVarOrd`, `pmmip` and `pmmipVarOrd`, where those ending in `VarOrd` are the corresponding variable order codes. The first pair, `pmme` and `pmmeVarOrd`, are explicit non-stiff solvers using a class I method. Their defining polynomial at step i in our implementation is defined as

$$\boldsymbol{P}_i(t) = \boldsymbol{c}_k(t - t_{i-1})^k + \cdots + \boldsymbol{c}_1(t - t_{i-1}) + \boldsymbol{c}_0. \tag{II.59}$$

The second pair, `pmmi` and `pmmiVarOrd`, are implicit stiff solvers using a class II method. Their defining polynomial at step i in our implementation is defined as

$$\boldsymbol{P}_i(t) = \boldsymbol{c}_k(t - t_i)^k + \cdots + \boldsymbol{c}_1(t - t_i) + \boldsymbol{c}_0. \tag{II.60}$$

The third pair, `pmmip` and `pmmipVarOrd`, are implicit non-stiff solvers using a class III method. The defining polynomial of these in step i is in our implementation defined as

$$\boldsymbol{P}_i(t) = \boldsymbol{c}_{k+1}(t - t_{i-1})^{k+1} + \cdots + \boldsymbol{c}_1(t - t_{i-1}) + \boldsymbol{c}_0. \tag{II.61}$$

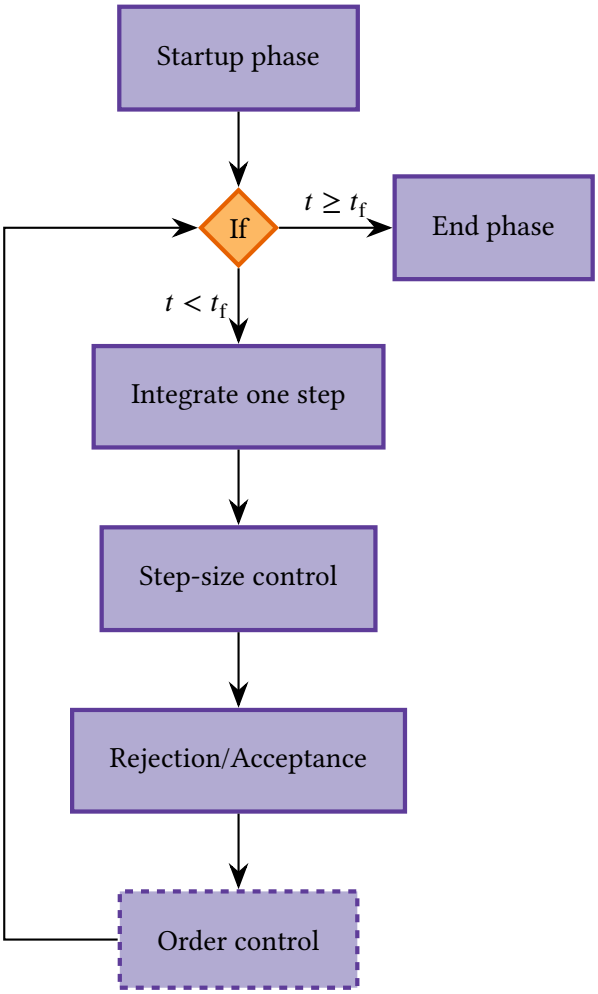

Figure II.8. Flow chart depicting a high level view of the solvers

For all six solvers the whole integration process can be divided into five/six major parts (which are depicted in Figure II.8):

- Startup phase
- Integration of one step
- Step-size control
- Rejection/Acceptance
- *Order control (is omitted for fixed order solvers)*
- End phase

The implementation details of each of these parts (and a few other things) are described below (except for the part dealing with order control which is described in Section IV.2). The *startup phase* and the *end phase* are just special cases of the other phases, and because of this we choose to describe the other parts first, even though they do not appear first in the control flow of the solvers.

In this implementation section we will use the term *acceptance sequence*, meaning the sequence of steps that are already calculated and accepted. With the term *step* we mean the solution, the step-size and the error estimate belonging to one iteration in the solver. In the beginning of the integration process, the acceptance sequence will be empty. Every time a step is *accepted*, the sequence will be updated with information about the newly accepted step. In the end of the integration process, the acceptance sequence is what will be returned to the user.

II.8.1. Integrating one step

It is important to note that in this phase the calculated solution is not necessarily *accepted*. Let us call it a *trial integration*. If the step is accepted or not will be determined in the next phase. It is not until we know that a step is accepted that we add it to the acceptance sequence.

Let us assume that we are at step n, i.e., $n-1$ steps have already been accepted and added to the acceptance sequence. The purpose of this phase is to calculate the coefficient vector, $\bar{\boldsymbol{c}}_n$, containing all coefficients $\boldsymbol{c}_0, \boldsymbol{c}_1 \ldots$ belonging to the defining polynomial at step n, $\boldsymbol{P}_n$ (see Equation II.59, II.60 and II.61). This is done by solving the equation system consisting of the equations defining the specific class used (see page 27). In the case of the two solvers belonging to class $\boldsymbol{E}_k$, this will lead to a linear system of equations. In this case Matlab's backslash(\)-operator[3], which uses different types of solvers depending on the matrix in question [14], is used to solve the system.

In the case of the implicit solvers, the equations will lead to a non-linear system of equations. This system is solved by using a modified version of Newton's method, where the modification consists in the Jacobian only being evaluated once — which takes place before the iterations are started — and then reused in all iterations. The implemented function uses two stopping criteria:

$$\|\bar{\boldsymbol{c}}_n^i - \bar{\boldsymbol{c}}_n^{i-1}\|_\infty < \frac{\mathrm{TOL}}{10} \quad \text{or} \quad i \geq 12, \tag{II.62}$$

[3]The \-operator can also be used by calling the function `mldivide()`.

where i is the iteration index, $\bar{c}_n^i$ is the coefficient vector calculated in the ith iteration, and TOL is the supplied tolerance described in Subsection II.8.2. For this method to work, we need the Jacobian of the RHS-function and an initial guess.

The evaluation of the Jacobian is done in one of two ways: The user has the possibility of supplying the Jacobian; if this is done, then this function is used (which makes it possible to use the analytical Jacobian). Otherwise the Jacobian is calculated numerically by using the following *first order central difference scheme* on every component:

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(t, \boldsymbol{x} + h_j \cdot \boldsymbol{e}_j) - f_i(t, \boldsymbol{x} - h_j \cdot \boldsymbol{e}_j)}{2h_j}, \tag{II.63}$$

where f_i is the i:th component of $\boldsymbol{f}$, h_j is the chosen step size in the x_j-direction and $\boldsymbol{e}_j$ is the natural basis vector in direction j.

The initial guess supplied to the numerical method is the coefficient vector, $\bar{c}_{n-1}$, belonging to the defining polynomial in the previously accepted step, i.e., the coefficient vector belonging to $\boldsymbol{P}_{n-1}$. In the current implementation, the constructed polynomials do not share the same local coordinate system (see Figure II.9 and II.10), which might slow down the rate of convergence of Newton's method. This problem could be solved either by using the same local coordinate system for all polynomials when constructing them, or by transforming the coefficients of $\boldsymbol{P}_{n-1}$ such that they are expressed using the same local coordinate system as $\boldsymbol{P}_n$. Further investigations would have to be made about this subject, to see whether or not these approaches are profitable.

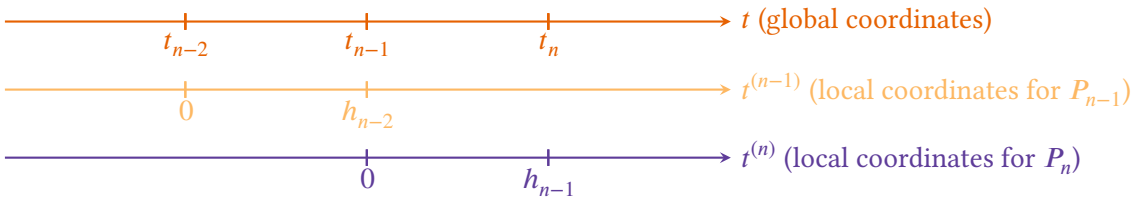

Figure II.9. A depiction of the local coordinate systems used by the constructed polynomials P_{n-1} and P_n, and their relation to the global coordinate system, where the polynomials belong to the class E_k or I_k^+.

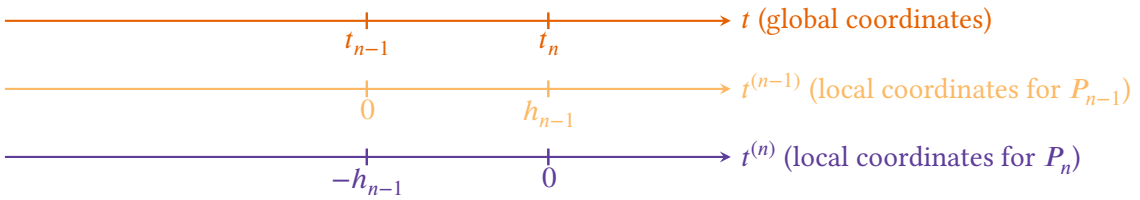

Figure II.10. A depiction of the local coordinate systems used by the constructed polynomials P_{n-1} and P_n, and their relation to the global coordinate system, where the polynomials belong to the class I_k.

When the coefficient vector is calculated, we evaluate the polynomial $\boldsymbol{P}_n$ at t_n, giving us the new solution point $\boldsymbol{x}_n$, i.e., $\boldsymbol{x}_n = \boldsymbol{P}_n(t_n)$. Once again, the solution point is not yet accepted.

II.8.2. Step-size control

A commonly used approach when it comes to controlling the error in numerical solvers, is to use the elementary controller [25]

$$h_{n+1} = \left(\frac{\theta \cdot \text{TOL}}{e_n} \right)^{1/p} h_n, \quad \theta < 1, \tag{II.64}$$

where θ is a safety factor, h_n is the step-size at step n, e_n is the error estimate at step n, TOL is the supplied tolerance and p is the order of the method used. As long as e_n is smaller than the tolerance, the step will be accepted. When the error estimate instead grows larger than the tolerance, the step will be rejected. This choice of implementation will often result in many rejections, which is unwanted since it will disturb the control process. This way of controlling the error will often lead to solvers with poor robustness. Another way of controlling the error in a numerical solver, is explained below and used in our implementation.

The step-size control phase of the solvers is done using digital filters based on control theory. These filters use previously calculated errors and step-sizes to decide a new step-size for the next step — digital filters are described in Section III.1.

After a new solution, $\boldsymbol{x}_n$, has been calculated (see previous section), the defining polynomial used to calculate the previous step, $\boldsymbol{P}_{n-1}$, is evaluated at the current time point, t_n, to get a prediction, $\boldsymbol{x}_n^{\text{pred}}$. This prediction is then used to calculate an estimation of the local error, e_n, by taking the difference between it and the new point $\boldsymbol{x}_n$ (this is described in more detail in Section II.6). The *error estimate* is given by

$$e_n = \|\boldsymbol{x}_n - \boldsymbol{x}_n^{\text{pred}}\| = \|\boldsymbol{P}_n(t_n) - \boldsymbol{P}_{n-1}(t_n)\|, \tag{II.65}$$

where $\|\bullet\|$ is a user supplied norm — which defaults to $\|\bullet\|_\infty$ if no norm is supplied by the user. This error estimate, together with previous error estimates and previous step-size ratios are fed to the filter, which returns a new step-size ratio r_{n-1} (not yet accepted).

In some cases — for example in the beginning — there are not enough previous error estimates to feed to the filter. In these cases the so called *elementary controller* is used instead — since it only needs the latest error estimate, e_n — giving us the new step-size ratio as[4]

$$r_{n-1} = \left(\frac{\text{TOL}}{e_n} \right)^{1/p} \tag{II.66}$$

where p is the order of the multistep method being used.

II.8.3. Rejection/Acceptance

Even when a proper step-size filter is used, situations might arise in which the estimated error corresponding to a step becomes unacceptably large. In these cases we have to reject the step and start anew.

[4]The index of r is as correctly stated $n-1$ at step n. See Page 7 for more information about notation.

This is the phase where we decide if x_n should be rejected or not. We do not use the actual error estimate e_n to make this decision, but instead we use the step-size ratio r_{n-1} as an indicator of the size of the error. Depending on the size of r_{n-1}, three different paths can be taken, i.e., the r-interval is divided into three subintervals r_I, r_{II} and r_{III}, defined according to

$$r_I = \{r < r_{\min}\}, \tag{II.67}$$

$$r_{II} = \{r_{\min} \leq r \leq r_{\max}\}, \tag{II.68}$$

$$r_{III} = \{r > r_{\max}\}, \tag{II.69}$$

where $r_{\min} \leq 1$ and $r_{\max} \geq 1$. In our code the default values of these are $r_{\min} = 0.8$ and $r_{\max} = 1.2$, which means that the controller is allowed to decrease/increase a step-size by at most 20% at every step. These three cases are discussed below.

Case I: $r_{n-1} \in r_I$

In the case when $r_{n-1} \in r_I$, we have an indication of a large error, which results in a rejection of x_n. This is therefore called *the rejection interval*.

If step n is rejected one time, the previous step-size ratio r_{n-2} is overridden by $r_{\min}$ (see Figure II.11), and the step is then recalculated using the new step-size $\tilde{h}_{n-1} = r_{\min} h_{n-2}$. If the filter now suggests an acceptable step-size ratio $\tilde{r}_{n-1}$, the step is accepted and the integration continues as before. If instead the new step-size ratio $\tilde{r}_{n-1}$ also is smaller than the limit $r_{\min}$, the previous step-size ratio $\tilde{r}_{n-2}$ is once again decreased, but this time by 5%. This is repeated until the new step is accepted, or the new step-size becomes too small, which happens when

$$\frac{\tilde{h}_{n-1}}{t_{\mathrm{f}} - t_0} < 10^{-16}, \tag{II.70}$$

in which case the integration is aborted and reported as failed to the user.

The method of regulating the step-size after rejections, by decreasing it 5% at a time, will sometimes result in a value for the step-size ratio which deviates largely from the original value proposed by the controller. Because of this extreme override, the control sequence is regarded as destroyed, and further regulation using the same step-size sequence is inadvisable. Instead the regulation process is restarted by using the filter described in Equation II.66 until enough new error estimates and step-sizes have been calculated, and then the use of the main filter is resumed (in the fixed order solvers the option to turn off this behavior, and continue to use the main filter all the time, is available).

Case II: $r_{n-1} \in r_{II}$

The case $r_{n-1} \in r_{II}$ is an indication that the error is within acceptable bounds. In this interval the step is accepted, which means that the step is not on trial anymore. Now $\boldsymbol{x}_n$, e_n and h_{n-1} are added to the acceptance sequence.

Case III: $r_{n-1} \in r_{III}$

The case $r_{n-1} \in r_{III}$ is an indication of an error that is too small. There are reasons (e.g., stability) not to allow too large increases of the step-size. The difference between this interval and the

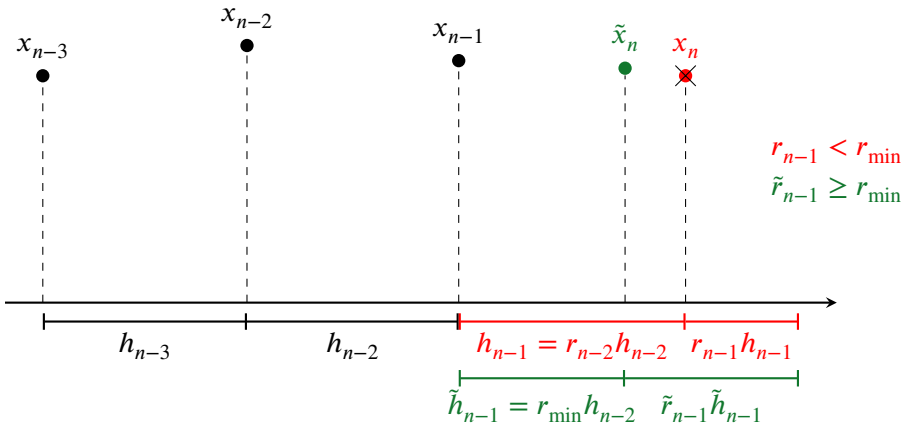

Figure II.11. A situation when the step with the solution x_n is rejected, because the new step-size ratio r_{n-1} proposed by the filter is too small. In this case a new solution $\tilde{x}_n$ must be calculated. The solver stops and overrides the previous suggested step-size ratio r_{n-2} by $r_{\min}$. It then calculates $\tilde{x}_n$, and asks the filter for a new step-size ratio $\tilde{r}_{n-1}$. If $\tilde{r}_{n-1} \geq r_{\min}$ the new solution is accepted and the solver continues as normal. If the step still is rejected the previous step-size ratio is decreased by 5%.

previous one is that instead of adding r_{n-1} to calculate the new step-size, we use $r_{\max}$ to do it. Everything else is exactly as in the previous case.

In contrast to the case when the step was rejected, the interference with the controller is not as large in this case, and therefore the regulation using the main filter is continued.

The intervals r_{II} and r_{III} form the acceptance interval.

II.8.4. Start up phase

There are three major problems which need to be solved in the startup phase of the solver:

1. An initial step-size must be determined
2. For a k-step method, k starting values must be provided.
3. There is no previous polynomial which can be used to calculate an error estimate.

How to estimate the initial step-size is outside the scope of this thesis. We use an algorithm (and code) which among other things estimates the Lipschitz constant to calculate an initial step-size [1]. One important thing to note about this algorithm is that it is *not* deterministic, but contains a part that uses a small random perturbation, which can lead to slightly varying results in some cases. This randomness will affect the reproducibility of results, but the user is able to set the initial step-size as an option.

The second problem is one that affects all multistep solvers, and one common approach is to either start with a 1-step method, then change to a 2-step method and continue increasing the number of steps like this until the k-step method can be used. Another common approach is to use a Runge-Kutta method to integrate until there are k steps, and then switch to the multistep method. The solvers `pmme`/`pmmeVarOrd` and `pmmip`/`pmmipVarOrd` use the latter solution, which

is implemented by calling Matlab's solver `ode45` with the following parameters

$$\text{relTol} = \text{TOL} \tag{II.71}$$

$$\text{absTol} = \text{TOL} \cdot 10^{-3} \tag{II.72}$$

where TOL is the tolerance used by our solver.

For some stiff problems this does not produce a good enough accuracy for the first steps, and therefore `ode15s` is used instead with the same settings in `pmmi/pmmiVarOrd`.

The last problem is not common to all multistep solvers, but is a consequence of the way the error is estimated in this case (see Section II.6). In the case of solvers of the type $\boldsymbol{I}_k$ this problem has an easy and natural solution. From Theorem II.6.1 on page 40, we know that there exists an explicit k-step method which could be used instead of the old polynomial, and therefore we construct this in accordance with the theorem.

In the cases of solvers of types $\boldsymbol{E}_k$ and $\boldsymbol{I}_k^+$ there are unfortunately no such simple solutions. There is no corresponding theorem for solvers of type $\boldsymbol{E}_k$ and although Theorem II.6.2 is a corresponding theorem for solvers of type $\boldsymbol{I}_k^+$ the explicit method which can be constructed in this case needs $k+1$ steps. In these two cases we have chosen to not enable the step-size control mechanism until $k+1$ steps have been generated, at which point there exists a previously constructed polynomial.

The lack of a previous polynomial also has another consequence for the implicit solvers; these solvers need polynomial coefficients which can be fed as a starting guess into Newton's method. Once again the case is simple for solvers of the type $\boldsymbol{I}_k$: the coefficients of the polynomial which was constructed by using an explicit method to estimate the error are used. For solvers of the type $\boldsymbol{I}_k^+$ the problem is solved in the following way:

1. A solver of the type $\boldsymbol{E}_k$ (in this case using the Adams–Bashforth method) is used to construct a polynomial of degree k. Let the coefficients be denoted $\{\boldsymbol{c}_i\}_{i=0}^k$, where $\boldsymbol{c}_0$ is the coefficient of the 0-degree term.

2. The polynomial is then extended to a polynomial of degree $k+1$, by adding another term $\boldsymbol{c}_{k+1}$, which initially is set to $\boldsymbol{0}$.

3. The extended polynomial is then fed as an initial guess to Newton's method.

The scheme above is chosen because it seems to generate a guess which is good enough to make the solver produce an acceptable step (not always on the first try though, but after adjusting the step-size); no detailed investigation of this issue has been done.

II.8.5. End phase

The end phase is reached when $t \geq t_\text{f}$. In the case of $t > t_\text{f}$, the last step in the acceptance sequence is too long, since we want to hit the end point t_f. A new point is calculated at the time point t_f, replacing the too long step.

II.8.6. Division by zero protection

In some parts of the code the division operation will be performed with divisors which may be equal to 0 (for example when the relative error is calculated). To protect against this, a small number ε, with the same sign as the divisor, is added to it, i.e.,

$$\frac{\text{numerator}}{\text{divisor} + \varepsilon} \tag{II.73}$$

where $|\varepsilon|$ is set to 10^{-16}.

II.8.7. Anti-windup

A controller works best when it is not tampered with. If you override a controller, it will start to compensate for the externally imposed instructions. Take our error controller as an example. We let it run free until the output r_{new} passes one of the limits $r_{\max}$ and $r_{\min}$. In these cases, instead of using the recommendation of the controller, we override it. If no *anti-windup scheme* is applied at this point, the output signal r might behave badly, since the controller will try to regain control of the process.

In our case, when $r_{\text{new}} < r_{\min}$, we have an indication of a large error. The step is rejected, which means that the step-size is decreased. Our anti-windup scheme is to *restart* the controller. What happens is that the information about previous errors and previous step-size ratios is eliminated, which means that the main controller can not be used. Instead *the elementary controller* (see Equation II.66) is used until we have enough information to go back to using the main controller.

In the case of $r_{\text{new}} > r_{\max}$ we currently do not use any anti-windup scheme. The reason is that the error is *too small* and *not too large*, which means that we do not expect stability problems. A different approach would be to use the same anti-windup scheme as in the case of $r_{\text{new}} < r_{\min}$.

II.8.8. Solver options

At this state, the solvers are *experimental*, meaning that they are meant for numerical research and used by scientists in the same field. For this reason, the user is able to change a lot of parameters and settings. An interesting thing to investigate is, for example, if some solver-method-filter-problem (SMFP) combinations are better than others. The options available are the following:

Tolerance The user is able to supply the tolerance. The step-size controller will use this as a set-point for the local error at every step.

Method name The user is able to choose what method to be used by the solver, by supplying the name of it. Available method names are given in Table II.1.

Method vector The user is also able to choose a method by supplying the corresponding method vector. This makes it possible to run the solvers with infinitely many unique methods, due to this new way of parameterize LMMs. Another useful aspect is that newly created LMMs can be tested immediately, given that the θ-vector is known, without any new implementation. This makes it possible to experiment with various SMFP-combinations.

Filter name The user is able to choose what filter to be used by the step-size controller, by supplying the corresponding name. Available filter names are given in Table II.2.

Filter vector The user is able to choose a filter by supplying the corresponding filter vector, which gives the possibility of running the solvers with infinitely many unique filters.

Step-size ratio interval The user is able to choose the step-size ratio boundaries $r_{\min}$ and $r_{\max}$. It can be shown (theoretically) that different methods may become unstable at different step-size ratio values [10, 1]. This option creates the opportunity to investigate the values of these instability barriers further.

Error mode The user is able to choose what *error mode* to be used internally when calculating the error estimation at every time step. The available choices are *absolute error mode* and *relative error mode*.

Filter mode The user is able to choose between two filter modes: *Error Per Unit Step (EPUS) mode* and *Error Per Step (EPS) mode*. More information on this subject can be found in Section III.1.

Initial step-size The user is able to set the initial step-size. This allows the user to investigate how long it takes for different SMFP-combinations to stabilize when initial step-sizes are set too large and too small.

Error norm The user is able to choose how to calculate the error estimation by supplying a general norm function. As a default, the infinity norm is used, but measuring the error of a multidimensional function is not straightforward. Some problems need special treatment. In some cases there might be advantages in measuring the error of only one component, while in other cases the different components might need different error weights.

No anti-windup scheme The user is able to turn off the anti-windup scheme. This means that after a step is rejected, the previously calculated error estimates will not be thrown away, but instead used, as usual, to calculate the future error estimates. This option makes it possible to investigate whether this anti-windup scheme is a good choice of algorithm.

Supply analytical solution The user is able to supply the exact solution of the RHS-function, which will then be used to estimate the error at every step. This option makes it possible to give the controller the actual local error. By using this option, it is possible to investigate the behavior of the controller in more detail.

Supply analytical Jacobian In the implicit solvers, the user is able to supply the analytical Jacobian. If no analytical Jacobian is supplied, the Jacobian needed by the solver will be calculated numerically (for more information see Subsection II.8.1).

Table II.1. The methods that are implemented in our method library. The name under *Method* is what the user supplies to the solver in order to choose a particular method.

Method I_k	*order*	k	n_θ		$\tan(\theta_j)$, $j = 0 : k-1$			
BDFk [10]	$k \le 6$	k	k	$\{0\}_{0:k-1}$				
Kregel [5]	3	3	3	154/543	−11/78	0	—	—
Rockswold [21]	3	3	3	1/3	2/3	1	—	—
Method I_k^+	*order*	k	n_θ		$\tan(\theta_j)$, $j = 1 : k-1$			
AMk [10]	$k+1$	k	$k-1$	$\{\infty\}_{1:k-1}$				
dcBDFk [2]	$k+1$	k	$k-1$	$\{(j+1)/(k+1)\}_{1:k-1}$				
Milne2 [10]	3	2	1	1/3	—	—	—	—
Milne4 [10]	5	4	3	4/15	∞	∞	—	—
IDC23 [2]	4	3	2	7/6	∞	—	—	—
IDC24 [2]	5	4	3	26/15	∞	∞	—	—
IDC34 [2]	5	4	3	4/5	33/20	∞	—	—
IDC45 [2]	6	5	4	28/45	11/10	32/15	∞	—
IDC56 [2]	7	6	5	43/84	6/7	29/21	55/21	∞
Method E_k	*order*	k	n_θ		$\tan(\theta_j)$, $j = 1 : k-1$			
ABk [10]	k	k	$k-1$	$\{\infty\}_{1:k-1}$				
EDFk [2]	k	k	$k-1$	$\{j+1\}_{1:k-1}$				
Nyström3 [10]	3	3	2	−2/3	∞	—	—	—
Nyström4 [10]	4	4	3	−5/3	∞	∞	—	—
Nyström5 [10]	5	5	4	−133/45	∞	∞	∞	—
EDC22 [2]	3	3	2	14/3	∞	—	—	—
EDC23 [2]	4	4	3	49/6	∞	∞	—	—
EDC33 [2]	4	4	3	7/2	39/4	∞	—	—
EDC24 [2]	5	5	4	1121/90	∞	∞	∞	—
EDC34 [2]	5	5	4	53/10	219/10	∞	∞	—
EDC45 [2]	6	6	5	193/45	121/10	692/15	∞	∞

Table II.2. The filters that are implemented in our filter library. The *Filter name* is what the user supplies to the solver in order to choose a particular filter. Filter classes and parameters are defined and explained in Section III.1.

Filter name	*Filter class*	$\kappa\beta_1$	$\kappa\beta_2$	$\kappa\beta_3$	$-\alpha_2$	$-\alpha_3$	*Classification*
H211D [25]	H_0211	1/2	1/2	—	−1/2	—	low-pass, dead-beat
H211*b* [25]	H211	1/*b*	1/*b*	—	−1/*b*	—	low-pass
H211PI [25]	H211PI	1/6	1/6	—	0	—	low-pass, PI
PI3333 [26]	H210PI	2/3	−1/3	—	0	—	PI
PI3040 [26]	H210PI	7/10	−4/10	—	0	—	PI
PI4020 [26]	H210PI	3/5	−1/5	—	0	—	PI
H312D [25]	H_0312	1/4	1/2	1/4	−3/4	−1/4	low-pass, dead-beat
H312*b* [25]	H312	1/*b*	2/*b*	1/*b*	−3/*b*	−1/*b*	low-pass
H312PID [25]	H312PID	1/18	1/9	1/18	0	0	low-pass, PID
H321D [25]	H_0321	5/4	1/2	−3/4	1/4	3/4	low-pass, dead-beat
H321 [25]	H321	1/3	1/18	−5/18	5/6	1/6	low-pass

Part III.

Construction and benchmarking of an error regulation algorithm

Author: Josefine Olander

III.1. Variable step-size implementation using digital filters

One way to implement variable step-size in a numerical solver is by the use of control and filter theory [25]. A discrete closed loop control system (see Figure III.1) is built making it possible to control the step-size and therefore also the error in a good way.

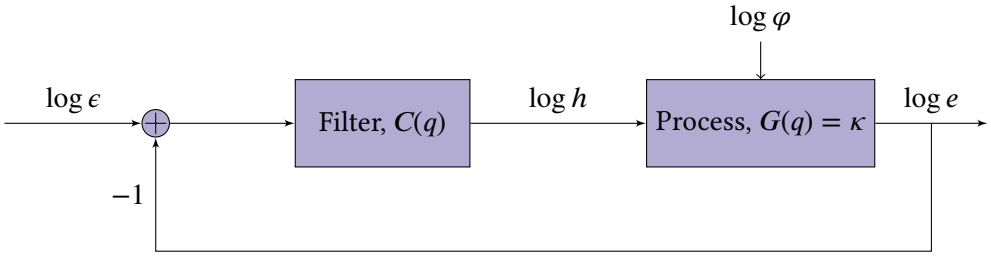

Figure III.1. Block diagram of a discrete closed loop control system, making it possible to control the step-size in a numerical solver.

This control action can be described by the following formula:

$$\log h = C(q)(\log \epsilon - \log e), \tag{III.1}$$

where h is the step-size sequence, q is *the forward-shift operator* ($qh_n = h_{n+1}$), $C(q)$ is *the control transfer function*, ϵ is the set-point sequence which is usually set to TOL and e is the error sequence (the sequence of error estimates). Furthermore, we assume that the local error of our multistep method is described by the *asymptotic model* [25]

$$e_n = \varphi_n h_n^{\kappa}, \tag{III.2}$$

where the subscript n denotes the number of the current step, φ_n denotes the *norm of the principal error function*, and κ is a scalar taking one of two values depending on what *error mode* we use. If p is the order of convergence of the numerical method, then $\kappa = p + 1$ when *error per step* — the error for the specific step-size h_n, which makes it harder to compare it to other methods, problems etc., since the steps taken vary in length — (EPS) mode is used and $\kappa = p$ when *error per unit step* — the error for a unit step, which gives us the ability to compare the error to other methods etc., since the same step length is used — (EPUS) mode is used.

Taking the logarithm of Equation III.2, will give us the following relation:

$$\log e = \kappa \ \log h + \log \varphi. \tag{III.3}$$

Combining Equation III.1 and Equation III.3 will make it possible to uncouple $\log e$ and $\log h$, resulting in:

$$\log e = E_{\epsilon}(q) \log \epsilon + E_{\varphi}(q) \log \varphi, \tag{III.4}$$

$$\log h = H_{\epsilon}(q) \log \epsilon + H_{\varphi}(q) \log \varphi. \tag{III.5}$$

where *the closed loop transfer functions* E_ϵ, E_φ, H_ϵ and H_φ are given by:

$$E_\epsilon(q) = \frac{\kappa C(q)}{1 + \kappa C(q)}, \tag{III.6}$$

$$E_\varphi(q) = \frac{1}{1 + \kappa C(q)}, \tag{III.7}$$

$$H_\epsilon(q) = \frac{C(q)}{1 + \kappa C(q)}, \tag{III.8}$$

$$H_\varphi(q) = -\frac{C(q)}{1 + \kappa C(q)}. \tag{III.9}$$

Of these functions, we are mostly interested in $E_\varphi(q)$ and $H_\varphi(q)$, since these are coupled to $\log \varphi$. $E_\varphi(q)$ is called *the error transfer map*, and $H_\varphi(q)$ is called *the step-size transfer map*.

From basic control theory it is well known that the control function, $C(q)$, has to include the factor $1/(q-1)$ for the controller to be able to adapt to the *set-point* [25] — the desired/target value. This factor is called *the integral part* or *the integrator*. Let us now write the control function as

$$C(q) = \frac{P(q)}{(q-1)Q(q)}, \tag{III.10}$$

where P and Q are polynomials of order $p_\mathrm{D} - 1$, and p_D is the order of the closed loop dynamics. P and Q can generally be written as

$$P(q) = \sum_{j=1}^{p_\mathrm{D}} \beta_j q^{p_\mathrm{D}-j}, \quad Q(q) = q^{p_\mathrm{D}-1} + \sum_{j=2}^{p_\mathrm{D}} \alpha_j q^{p_\mathrm{D}-j}. \tag{III.11}$$

Sometimes we will use the short version "r:th order filter" to mean "a filter where $p_D = r$". Substituting Equation III.11 and Equation III.10 into Equation III.1 gives us

$$\log h = \frac{1}{(q-1)} \frac{\sum_{j=1}^{p_\mathrm{D}} \beta_j q^{p_\mathrm{D}-j}}{q^{p_\mathrm{D}-1} + \sum_{j=2}^{p_\mathrm{D}} \alpha_j q^{p_\mathrm{D}-j}} (\log \epsilon - \log e) \tag{III.12}$$

$$\Longleftrightarrow$$

$$\left(q^{p_\mathrm{D}-1} + \sum_{j=2}^{p_\mathrm{D}} \alpha_j q^{p_\mathrm{D}-j} \right) (q-1)\ \log h = \sum_{j=1}^{p_\mathrm{D}} \beta_j q^{p_\mathrm{D}-j} \log\left(\frac{\epsilon}{e}\right) \tag{III.13}$$

By letting the latest index in the sequences be n, we will get

$$\left(q^{p_D-1}+\sum_{j=2}^{p_D}\alpha_j q^{p_D-j}\right)(q-1)\ \log h_{n-p_D+1}=\sum_{j=1}^{p_D}\beta_j q^{p_D-j}\log\left(\frac{\epsilon_{n-p_D+1}}{e_{n-p_D+1}}\right) \tag{III.14}$$

$$\Longleftrightarrow$$

$$\left(q^{p_D-1}+\sum_{j=2}^{p_D}\alpha_j q^{p_D-j}\right)(q\ \log h_{n-p_D+1}-\log h_{n-p_D+1})=\sum_{j=1}^{p_D}q^{p_D-j}\log\left(\frac{\epsilon_{n-p_D+1}}{e_{n-p_D+1}}\right)^{\beta_j} \tag{III.15}$$

$$\Longleftrightarrow$$

$$\left(q^{p_D-1}+\sum_{j=2}^{p_D}\alpha_j q^{p_D-j}\right)(\log h_{n-p_D+2}-\log h_{n-p_D+1})=\sum_{j=1}^{p_D}\log\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.16}$$

$$\Longleftrightarrow$$

$$\left(q^{p_D-1}+\sum_{j=2}^{p_D}\alpha_j q^{p_D-j}\right)\log\left(\frac{h_{n-p_D+2}}{h_{n-p_D+1}}\right)=\sum_{j=1}^{p_D}\log\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.17}$$

$$\Longleftrightarrow$$

$$q^{p_D-1}\log\left(\frac{h_{n-p_D+2}}{h_{n-p_D+1}}\right)+\sum_{j=2}^{p_D}\alpha_j q^{p_D-j}\log\left(\frac{h_{n-p_D+2}}{h_{n-p_D+1}}\right)=\sum_{j=1}^{p_D}\log\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.18}$$

$$\Longleftrightarrow$$

$$\log\left(\frac{h_{n+1}}{h_n}\right)+\sum_{j=2}^{p_D}q^{p_D-j}\log\left(\frac{h_{n-p_D+2}}{h_{n-p_D+1}}\right)^{\alpha_j}=\sum_{j=1}^{p_D}\log\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.19}$$

$$\Longleftrightarrow$$

$$\log\left(\frac{h_{n+1}}{h_n}\right)+\sum_{j=2}^{p_D}\log\left(\frac{h_{n+2-j}}{h_{n+1-j}}\right)^{\alpha_j}=\sum_{j=1}^{p_D}\log\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.20}$$

$$\Longleftrightarrow$$

$$\left(\frac{h_{n+1}}{h_n}\right)\cdot\prod_{j=2}^{p_D}\left(\frac{h_{n+2-j}}{h_{n+1-j}}\right)^{\alpha_j}=\prod_{j=1}^{p_D}\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j} \tag{III.21}$$

$$\Longleftrightarrow$$

$$\left(\frac{h_{n+1}}{h_n}\right)=\prod_{j=1}^{p_D}\left(\frac{\epsilon_{n+1-j}}{e_{n+1-j}}\right)^{\beta_j}\cdot\prod_{j=2}^{p_D}\left(\frac{h_{n+2-j}}{h_{n+1-j}}\right)^{-\alpha_j} \tag{III.22}$$

In our case the set-point ϵ will be constant, it will be kept at the tolerance, giving us:

$$h_{n+1}=\prod_{j=1}^{p_D}\left(\frac{\epsilon}{e_{n+1-j}}\right)^{\beta_j}\cdot\prod_{j=2}^{p_D}\left(\frac{h_{n+2-j}}{h_{n+1-j}}\right)^{-\alpha_j}h_n. \tag{III.23}$$

If we substitute Equation III.10 into Equation III.7 and Equation III.9 we get

$$E_\varphi(q)=\frac{(q-1)Q(q)}{(q-1)Q(q)+\kappa P(q)},\quad H_\varphi(q)=-\frac{P(q)}{(q-1)Q(q)+\kappa P(q)}, \tag{III.24}$$

where the denominator of $E_\varphi(q)$ and $H_\varphi(q)$ is called *the characteristic polynomial* $K(q)$, and $K(q) = 0$ is called *the characteristic equation.* In order for the filter to be stable, the roots q_i of this equation have to be strictly inside the unit circle in the complex plane, i.e., $|q_i| < 1$ where q_i is the i:th root of $K(q) = 0$ [25].

Another important thing when it comes to controllers is to choose what frequencies to filter. Let us assume that the objective is to remove the frequency ω. The solution to this problem is to let the transfer function in question, T, have a root at $e^{i\omega}$. In our case we can choose how to filter the step-size sequence and the error sequence. What we would like is to make $H_\varphi(q)$ a low pass filter, since this will result in a smooth step-size curve. This is done by forcing $H_\varphi(e^{i\omega})$ to be zero at $\omega = \pi$.

III.1.1. The categorization and notation of digital filters

A digital filter can be categorized according to the following basic notation system

$$\mathrm{H}_{[\ ]}p_\mathrm{D}p_\mathrm{A}p_\mathrm{F}[\ ],$$

where p_D is called *the order of dynamics*, p_A is called *the order of adaptivity* at $q = 1$, p_F is called *the filter order* at $q = -1$ and [ ] is a placeholder that might be empty. Given the first three parameters we know the following about the closed control function C, and the polynomials P and Q:

- The order of dynamics p_D – This value tells us that the order of the polynomials P and Q is $p_\mathrm{D} - 1$.
- The order of adaptivity p_A – This value tells us that $(q-1)^{p_\mathrm{A}-1}$ is a factor of the polynomial Q, i.e., that $C(q)$ has p_A poles at $q = 1$.
- The filter order p_F – This value tells us that $(q+1)^{p_\mathrm{F}}$ is a factor of the polynomial P, i.e., that $C(q)$ has p_F roots at $q = -1$ (when $p_\mathrm{F} > 0$ we have a step-size sequence low pass filter).

Some special kinds of filters worth mentioning are:

Dead-beat controllers

These controllers have all its poles at $q = 0$. They are denoted by

$$\mathrm{H}_0p_\mathrm{D}p_\mathrm{A}p_\mathrm{F},$$

where the zero subscript indicates that the controller is a dead-beat controller.

PID controllers

A PID controller is a Proportional–Integral–Derivative controller, which means that the *feedback control signal* — the signal steering the process — is determined by a weighted sum of three terms: one term proportional to the error, one term proportional to the integrated error, and the last term proportional to the differentiated error. The controller has the general transfer function

$$C(q) = q^{-1}\left(k_\mathrm{I}\frac{q}{q-1} + k_\mathrm{P} + k_\mathrm{D}\frac{q-1}{q}\right), \tag{III.25}$$

where k_I, k_P, and k_D are *the integral gain*, *the proportional gain*, and *the derivative gain*. By restructuring the function to have the form in Equation III.10, we can identify the parameters α_i and β_i, and use the recursive formula for the step-size sequence given in Equation III.23. This will result in

$$h_{n+1} = \left(\frac{\epsilon}{e_n}\right)^{k_\mathrm{I}+k_\mathrm{P}+k_\mathrm{D}} \left(\frac{\epsilon}{e_{n-1}}\right)^{-(k_\mathrm{P}+2k_\mathrm{D})} \left(\frac{\epsilon}{e_{n-2}}\right)^{k_\mathrm{D}} h_n, \tag{III.26}$$

i.e., $p_\mathrm{D} = 3$, $\alpha_2 = \alpha_3 = 0$, $\beta_1 = k_\mathrm{I} + k_\mathrm{P} + k_\mathrm{D}$, $\beta_2 = -(k_\mathrm{P} + 2k_\mathrm{D})$, and $\beta_3 = k_\mathrm{D}$. This type of controller is denoted by

$$\mathrm{H3}p_\mathrm{A}p_\mathrm{F}\mathrm{PID}.$$

PI controllers

A PID controller with $k_\mathrm{D} = 0$ becomes a PI controller with the following functions

$$C(q) = q^{-1}\left(k_\mathrm{I}\frac{q}{q-1} + k_\mathrm{P}\right), \tag{III.27}$$

$$h_{n+1} = \left(\frac{\epsilon}{e_n}\right)^{k_\mathrm{I}+k_\mathrm{P}} \left(\frac{\epsilon}{e_{n-1}}\right)^{-k_\mathrm{P}} h_n, \tag{III.28}$$

and the following parameters

$$p_\mathrm{D} = 2, \tag{III.29}$$

$$\alpha_2 = 0, \tag{III.30}$$

$$\beta_1 = k_\mathrm{I} + k_\mathrm{P}, \tag{III.31}$$

$$\beta_2 = -k_\mathrm{P}. \tag{III.32}$$

This type of controller is denoted by

$$\mathrm{H2}p_\mathrm{A}p_\mathrm{F}\mathrm{PI}.$$

III.1.2. Filters of 1st order dynamics

A filter of 1st order dynamics denoted by $\mathrm{H1}p_\mathrm{A}p_\mathrm{F}$ will have the following polynomials $\boldsymbol{P}$ and $\boldsymbol{Q}$:

$$\boldsymbol{P}(q) = \beta_1, \quad \boldsymbol{Q}(q) = 1, \tag{III.33}$$

which will result in the following transfer functions:

$$\boldsymbol{C}(q) = \frac{\beta_1}{(q-1)}, \quad \boldsymbol{H}_\varphi(q) = \frac{-\beta_1}{(q-1)+\kappa\beta_1}, \quad \boldsymbol{E}_\varphi(q) = \frac{(q-1)}{(q-1)+\kappa\beta_1}. \tag{III.34}$$

This will further result in the following step-size sequence:

$$h_{n+1} = (\epsilon/e_n)^{\beta_1} h_n. \tag{III.35}$$

The only possibility for p_A and p_F is $p_\mathrm{A} = 1$ and $p_\mathrm{F} = 0$, i.e., there is only one type of filter of 1st order dynamics, namely H110. The root of its characteristic equation is

$$q = 1 - \kappa\beta_1. \tag{III.36}$$

For this type of filter to be stable the following has to be fulfilled

$$-1 < 1 - \kappa\beta_1 < 1 \tag{III.37}$$

$$\Longleftrightarrow$$

$$\kappa\beta_1 \in]0, 2[. \tag{III.38}$$

The smaller we choose $\kappa\beta_1$, the smoother the step-size sequence becomes and the slower the homogeneous solution decays. Notice that this is not a low pass filter, hence this will not be used in the simulations later on. When $\beta_1 = 1/\kappa$ we call the controller *the elementary controller*.

III.1.3. Filters of 2nd order dynamics

A filter of 2nd order dynamics, denoted by H2$p_A p_D$, will have the following polynomials P and Q:

$$P(q) = \beta_1 q + \beta_2, \quad Q(q) = q + \alpha_2, \tag{III.39}$$

which will result in the following transfer functions:

$$C(q) = \frac{\beta_1 q + \beta_2}{(q-1)(q+\alpha_2)}, \tag{III.40}$$

$$H_\varphi(q) = \frac{-(\beta_1 q + \beta_2)}{q^2 + (\alpha_2 + \kappa\beta_1 - 1)q + (\kappa\beta_2 - \alpha_2)}, \tag{III.41}$$

$$E_\varphi(q) = \frac{(q-1)(q+\alpha_2)}{q^2 + (\alpha_2 + \kappa\beta_1 - 1)q + (\kappa\beta_2 - \alpha_2)}. \tag{III.42}$$

This will lead to the following step-size sequence:

$$h_{n+1} = (\epsilon/e_n)^{\beta_1}(\epsilon/e_{n-1})^{\beta_2}(h_n/h_{n-1})^{-\alpha_2} h_n. \tag{III.43}$$

The only possibilities for p_A and p_F are $p_A = 1, 2$ and $p_F = 0, 1$, i.e., there are only four types of filters of 2nd order dynamics namely H210, H211, H220 and H221. Since H210 and H220 are not low pass filters, we will only analyze H211 and H221.

The roots of all 2nd order filters are

$$x = \frac{1}{2}\left(-\alpha_2 - \beta_1\kappa + 1 \pm \sqrt{(\alpha_2 + \beta_1\kappa - 1)^2 - 4(\beta_2\kappa - \alpha_2)}\right). \tag{III.44}$$

Stability analysis of H221

H221 has the parameters $\alpha_2 = -1$ and $\beta_1 = \beta_2$. From Equation III.44 we get the roots

$$\begin{aligned} x_\pm &= \frac{1}{2}\left(1 - \beta_1\kappa + 1 \pm \sqrt{(-1 + \beta_1\kappa - 1)^2 - 4(\beta_1\kappa + 1)}\right) \\ &= \frac{1}{2}\left(2 - \beta_1\kappa \pm \sqrt{(\beta_1\kappa)^2 - 8\beta_1\kappa}\right). \end{aligned} \tag{III.45}$$

The root x_+ only takes values ≥ 1, which means that no stable H221 filter exists.

Stability analysis of H211

H211 has the parameters $\alpha_2 \neq -1$ and $\beta_1 = \beta_2$. From (III.44) we get the roots

$$x_{\pm} = \frac{1}{2}\left(-\alpha_2 - \beta_1\kappa + 1 \pm \sqrt{(\alpha_2 + \beta_1\kappa - 1)^2 - 4(\beta_1\kappa - \alpha_2)}\right). \tag{III.46}$$

Let us use the following notations

$$A_+ = \{(\alpha_2, \beta_1\kappa) : |x_+| < 1\}, \tag{III.47}$$
$$A_- = \{(\alpha_2, \beta_1\kappa) : |x_-| < 1\}, \tag{III.48}$$

i.e., A_+ (A_-) is the stability region in which x_+ (x_-) is stable. The intersection of these two, i.e.,

$$A = A_+ \cap A_- \tag{III.49}$$

is the stability region for all existing H211 filters. This region is given by

$$A = \{(\alpha_2, \beta_1\kappa) : \quad \beta_1\kappa > 0, \quad \alpha_2 < 1, \quad \beta_1\kappa < \alpha_2 - 1\}, \tag{III.50}$$

see Figure III.2.

One special subfamily of H211 is H211b. These filters have good frequency responses and well situated poles. Corresponding α's and β's are

$$\beta_1 = \beta_2 = \frac{1}{b\kappa}, \quad \alpha_2 = \frac{1}{b}, \tag{III.51}$$

resulting in the following roots of the characteristic equation

$$q = 0, 1 - 2/b. \tag{III.52}$$

These filters are stable for $b > 1$. If the second pole is negative, we get an oscillatory closed loop impulse response. This is highly unwanted, so for our simulations later on we choose $b \geq 2$. The stability domain of this family is marked with a continuous line in Figure III.2.

Stability analysis of PI filters

H2$p_{\mathrm{A}}p_{\mathrm{F}}$PI has the parameters $\alpha_2 = 0$, $\beta_1 = k_{\mathrm{I}} + k_{\mathrm{P}}$ and $\beta_2 = -k_{\mathrm{P}}$. From Equation III.44 we get the roots

$$\begin{aligned} x_{\pm} &= \frac{1}{2}\left(-\beta_1\kappa + 1 \pm \sqrt{(\beta_1\kappa - 1)^2 - 4\beta_2\kappa}\right) \\ &= \frac{1}{2}\left(-k_{\mathrm{I}} - k_{\mathrm{P}}\kappa + 1 \pm \sqrt{(k_{\mathrm{I}}\kappa + k_{\mathrm{P}}\kappa - 1)^2 + 4k_{\mathrm{P}}\kappa}\right). \end{aligned} \tag{III.53}$$

The corresponding stability region is given by

$$\begin{aligned} A &= \{(k_{\mathrm{I}}\kappa, k_{\mathrm{P}}\kappa) : |x_{\pm}| \leq 1\} \\ &= \{(k_{\mathrm{I}}\kappa, k_{\mathrm{P}}\kappa) : \quad k_{\mathrm{P}}\kappa < -\frac{k_{\mathrm{I}}\kappa}{2} + 1, \quad k_{\mathrm{P}}\kappa > -1, \quad k_{\mathrm{I}}\kappa > 0\}, \end{aligned} \tag{III.54}$$

and can be seen seen in Figure III.3.

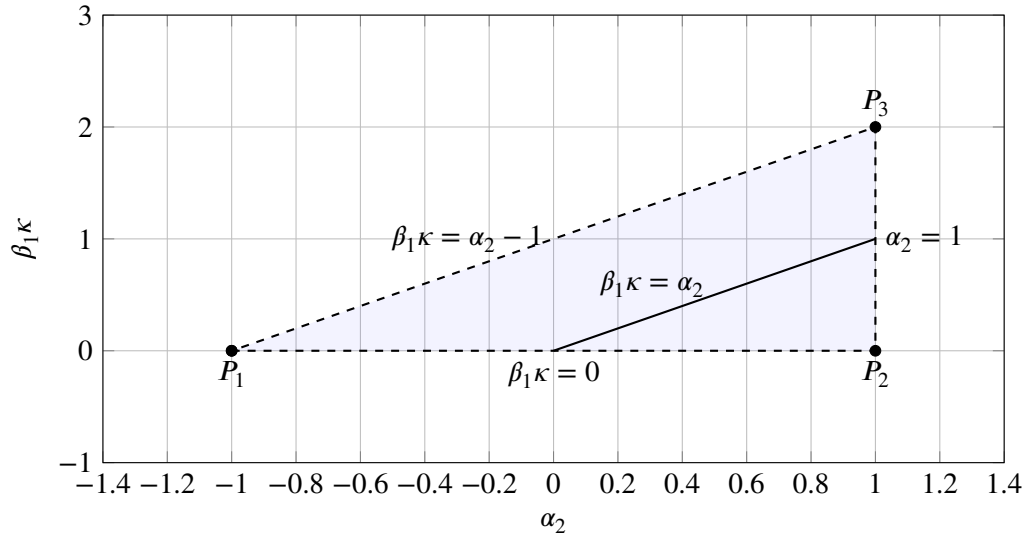

Figure III.2. The stability region of the family H211 (The dashed borderline is not included). The continuous line represents the domain of all stable H211*b* filters.

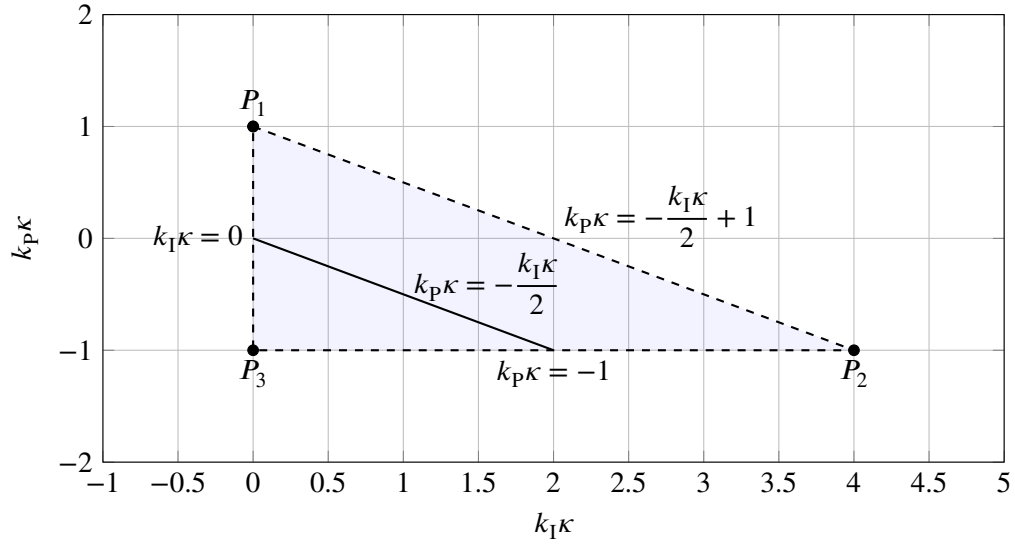

Figure III.3. The stability region of PI controllers (The dashed line is not included). The continuous line is the domain of all stable H211PI filters.

III.1.4. Filters of 3rd order dynamics

A filter of 3rd order dynamics denoted H3$p_A p_D$ will have the following polynomials P and Q:

$$P(q) = \beta_1 q^2 + \beta_2 q + \beta_3, \quad Q(q) = q^2 + \alpha_2 q + \alpha_3, \tag{III.55}$$

which will result in the following transfer functions:

$$C(q) = \frac{\beta_1 q^2 + \beta_2 q + \beta_3}{(q-1)(q^2 + \alpha_2 q + \alpha_3)}, \tag{III.56}$$

$$H_\varphi(q) = \frac{-(\beta_1 q^2 + \beta_2 q + \beta_3)}{q^3 + (\alpha_2 + \kappa\beta_1 - 1)q^2 + (\alpha_3 - \alpha_2 + \kappa\beta_2)q + (\kappa\beta_3 - \alpha_3)}, \tag{III.57}$$

$$E_\varphi(q) = \frac{(q-1)(q^2 + \alpha_2 q + \alpha_3)}{q^3 + (\alpha_2 + \kappa\beta_1 - 1)q^2 + (\alpha_3 - \alpha_2 + \kappa\beta_2)q + (\kappa\beta_3 - \alpha_3)}, \tag{III.58}$$

which further will result in the following step-size sequence:

$$h_{n+1} = (\epsilon/e_n)^{\beta_1}(\epsilon/e_{n-1})^{\beta_2}(\epsilon/e_{n-2})^{\beta_3}(h_n/h_{n-1})^{-\alpha_2}(h_{n-1}/h_{n-2})^{-\alpha_3} h_n. \tag{III.59}$$

The only possibilities for p_A and p_F are $p_A = 1, 2, 3$ and $p_F = 0, 1, 2$, i.e., there are nine types of 3rd order filters. Though, we only focus on the ones that are low pass filters, i.e., H311, H312, H321, H322, H331 and H332. It is easily shown that no stable H322, H331 and H332 filters exist, therefore we focus on stability analysis of H311, H312 and H321.

The characteristic polynomial of a 3rd order filter is always real — the coefficients are real — and of degree 3, which means that it can be factored in the following way:

$$(q + c)(q^2 + aq + b) = q^3 + (a + c)q^2 + (b + ac)q + cb, \tag{III.60}$$

where a, b, and c are real constants. To have a stable filter, all roots have to be strictly inside the unit circle, which in the terms of a, b, and c means that

$$|q_1| = |-a/2 + \sqrt{a^2/4 - b}| < 1, \tag{III.61}$$

$$|q_2| = |-a/2 - \sqrt{a^2/4 - b}| < 1, \tag{III.62}$$

$$|q_3| = |c| < 1. \tag{III.63}$$

Equation III.61 and Equation III.62 gives us the stability region T for root 1 and 2 (see Figure III.4)

We now wish to express c as a function of a, b, and either α_i or β_i.

Stability analysis of H321

The parameters of the filters H321 are

$$\begin{cases} \alpha_2 = y - 1 \\ \alpha_3 = -y \\ \kappa\beta_1 = x_1 \\ \kappa\beta_2 = x_1 + x_2 \\ \kappa\beta_3 = x_2 \end{cases} \tag{III.64}$$

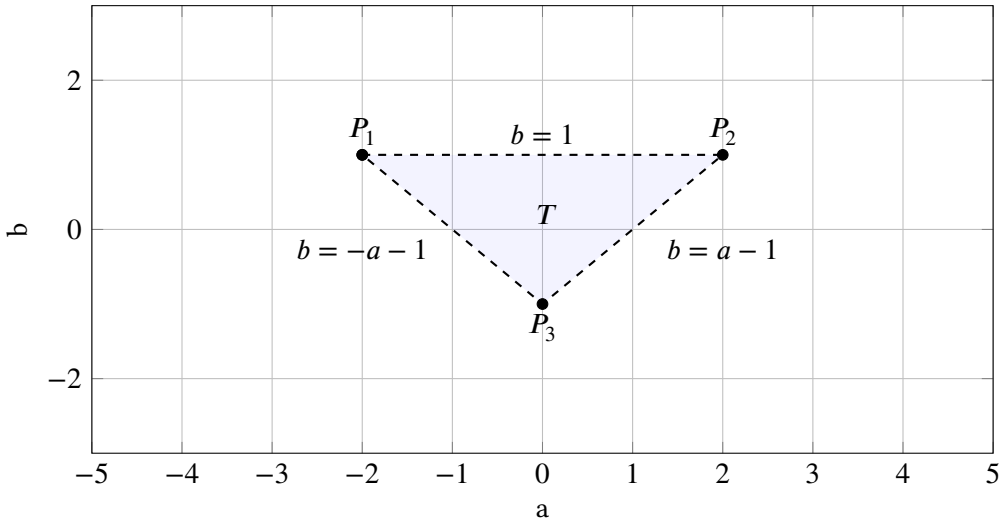

Figure III.4. The stability region of root 1 and 2 for a 3rd order filter (the dashed line is not included).

This gives us the characteristic equation

$$q^3 + (y - 2 + x_1)q^2 + (x_1 + x_2 - 2y + 1)q + (y - x_2). \tag{III.65}$$

By comparison with Equation III.60, we get the following system of equations:

$$\begin{cases} y - 2 + x_1 = a + c \\ x_1 + x_2 - 2y + 1 = b + ac \\ y + x_2 = cb \end{cases} \iff \begin{cases} c = \dfrac{-3 + b - a - 4x_2}{1 - a - 3b} \\ y = cb - x_2 \\ x_1 = b + (a + 2b)c - 1 - 3x_2 \end{cases} \tag{III.66}$$

The stability region for the filters H321, $A(x_2)$, is the intersection of T and B, where B is the stability region for the third root c, i.e.,

$$A(x_2) = T \cap B \quad \text{where} \quad B = \left\{ (a, b) : \frac{|-3 + b - a - 4x_2|}{|1 - a - 3b|} < 1 \right\}. \tag{III.67}$$

Calculation of the linear constraints in B will result in four cases

Case I

$$(-3 + b - a - 4x_2) < (1 - a - 3b)$$

where

$$-3 + b - a - 4x_2 > 0, \quad 1 - a - 3b > 0$$

Case II

$$(-3 + b - a - 4x_2) < -(1 - a - 3b)$$

where

$$-3 + b - a - 4x_2 > 0, \quad 1 - a - 3b < 0$$

Case III

$$-(-3+b-a-4x_2) < (1-a-3b)$$

where

$$-3+b-a-4x_2 < 0, \quad 1-a-3b > 0$$

Case IV

$$-(-3+b-a-4x_2) < -(1-a-3b)$$

where

$$-3+b-a-4x_2 < 0, \quad 1-a-3b < 0$$

After some calculations we get that $B = B_1 \cup B_2$, where

$$B_1 = \{(a,b) : b > x_2 + 1, b > -a - 1 - 2x_2\}, \tag{III.68}$$

$$B_2 = \{(a,b) : b < x_2 + 1, b < -a - 1 - 2x_2\}, \tag{III.69}$$

and

$$\mathrm{cl}(B_1) \cap \mathrm{cl}(B_2) = \{P_\mathrm{M}\} = \{(-3x_2 - 2, x_2 + 1)\}, \tag{III.70}$$

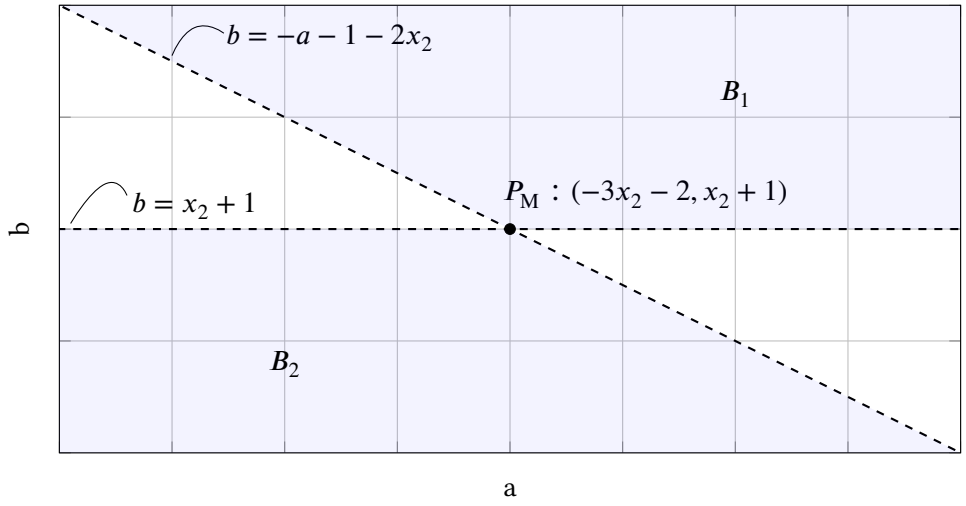

Figure III.5. The stability region for root 3. The dashed lines do not belong to the set.

where cl is the *closure operator*. It is easy to see that if none of the points P_1, P_2 and P_3 belongs to B, then the stability region $A(x_2) = T \cap B$ will be empty. We check if and when the points P_1, P_2 and P_3 are inside B_1 and B_2.

$P_1 = (-2, 1) \in B_1$ if the following is true

$$\begin{cases} 1 > x_2 + 1 \\ 1 > 2 - 1 - 2x_2 \end{cases} \implies \text{No solution} \tag{III.71}$$

$P_1 = (-2, 1) \in B_2$ if the following is true

$$\begin{cases} 1 \leq x_2 + 1 \\ 1 \leq 2 - 1 - 2x_2 \end{cases} \implies \text{No solution} \tag{III.72}$$

$P_2 = (2,1) \in B_1$ **if the following is true**

$$\begin{cases} 1 > x_2 + 1 \\ 1 > -2 - 1 - 2x_2 \end{cases} \iff -2 < x_2 < 0 \tag{III.73}$$

$P_2 = (2,1) \in B_2$ **if the following is true**

$$\begin{cases} 1 < x_2 + 1 \\ 1 < -2 - 1 - 2x_2 \end{cases} \implies \text{No solution} \tag{III.74}$$

$P_3 = (0,-1) \in B_1$ **if the following is true**

$$\begin{cases} -1 > x_2 + 1 \\ -1 > -1 - 2x_2 \end{cases} \implies \text{No solution} \tag{III.75}$$

$P_3 = (0,-1) \in B_2$ **if the following is true**

$$\begin{cases} -1 < x_2 + 1 \\ -1 < -1 - 2x_2 \end{cases} \iff -2 < x_2 < 0 \tag{III.76}$$

I.e., for the stability region $A(x_2)$ to be nonempty, $-2 < x_2 < 0$. In this case $P_2 \in B_1$ and $P_3 \in B_2$. Further we check if and when $P_{\mathrm{M}} = (-3x_2 - 2, x_2 + 1) \in T$.

$$\begin{cases} x_2 + 1 < 1 \\ x_2 + 1 > -3x_2 - 2 - 1 \\ x_2 + 1 > 3x_2 + 2 - 1 \end{cases} \iff \begin{cases} x_2 < 0 \\ x_2 > -1 \end{cases} \tag{III.77}$$

This gives us two cases:

Case I When $-1 < x_2 < 0$ the stability region (see Figure III.6) is given by $A = A^I = A_1^I \cup A_2^I$, where

$$A_1^I = \{(a,b) : \quad b < 1, \quad b > a - 1, \quad b > -a - 1 - 2x_2, \quad b > x_2 + 1\}, \tag{III.78}$$
$$A_2^I = \{(a,b) : \quad b < x_2 + 1, \quad b > a - 1, \quad b > -a - 1, \quad b < -a - 1 - 2x_2\}. \tag{III.79}$$

Case II When $-2 < x_2 \leq -1$ the stability region (see Figure III.7) is given by $A = A^{II} = A_1^{II} \cup A_2^{II}$, where

$$A_1^{II} = \{(a,b) : \quad b < 1, \quad b > a - 1, \quad b > -a - 1 - 2x_2\}, \tag{III.80}$$
$$A_2^{II} = \{(a,b) : \quad b < x_2 + 1, \quad b > a - 1, \quad b > -a - 1\}. \tag{III.81}$$

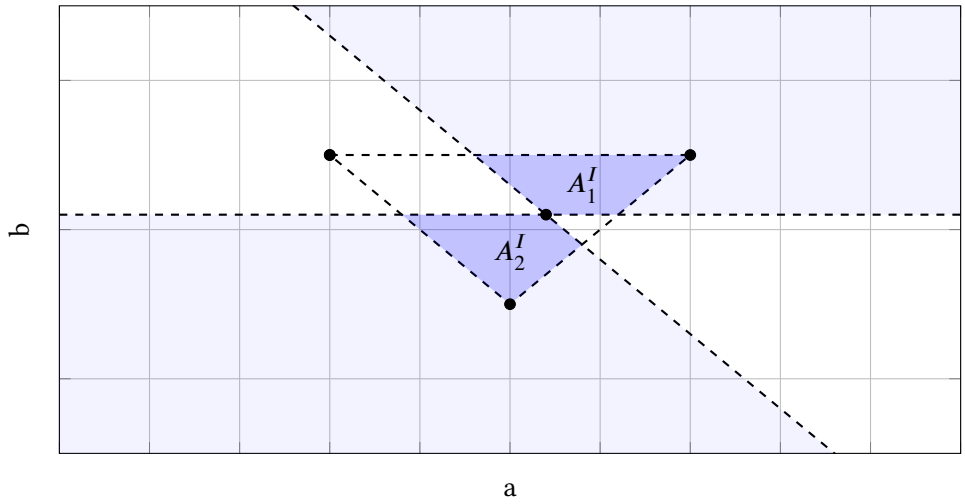

Figure III.6. The stability region of a H321 filter when $-1 < x_2 < 0$. The dashed lines do not belong to the set.

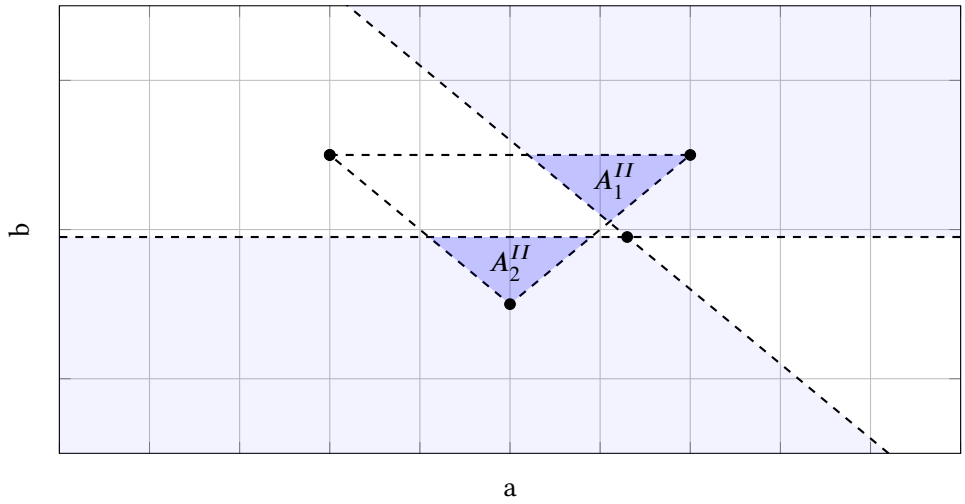

Figure III.7. The stability region of a H321 filter when $-2 < x_2 \leq -1$. The dashed lines do not belong to the set.

Substituting everything back, will result in that all stable H321 will have coefficients of the following form

$$\begin{cases} \alpha_2 = \left(\dfrac{-3+b-a-4x_2}{1-a-3b}\right) b - x_2 - 1 \\ \alpha_3 = -\left(\dfrac{-3+b-a-4x_2}{1-a-3b}\right) b + x_2 \\ \\ \kappa\beta_1 = b + (a+2b)\left(\dfrac{-3+b-a-4x_2}{1-a-3b}\right) - 1 - 3x_2 \\ \kappa\beta_2 = b + (a+2b)\left(\dfrac{-3+b-a-4x_2}{1-a-3b}\right) - 1 - 2x_2 \\ \kappa\beta_3 = x_2 \end{cases} \quad \text{where} \quad \begin{cases} (a,b) \in A(x_2) = A_1 \cup A_2 \\ -2 < x_2 < 0 \end{cases} \tag{III.82}$$

Stability analysis of H312
The parameters for H312 are given by

$$\begin{cases} \alpha_2 = y_1 \\ \alpha_3 = y_2 \\ \kappa\beta_1 = x \\ \kappa\beta_2 = 2x \\ \kappa\beta_3 = x \end{cases} \quad \text{where} \quad \alpha_2 \neq -\alpha_3 - 1. \tag{III.83}$$

By doing the corresponding analysis as in the previous part we get that for a filter of this class to be stable $-5/4 < y_2 < 3/4$. Furthermore, we get two cases:

Case I When $-5/4 < y_2 \leq -1/4$ the stability region (see Figure III.8) is given by $A = A^I = A_1^I \cup A_2^I$, where

$$A_1^I = \{(a,b) : \quad b < 1, \quad b > a - 1, \quad b > a + 0.5 - 2y_2\}, \tag{III.84}$$
$$A_2^I = \{(a,b) : \quad b < 0.25 + y_2, \quad b > a - 1, \quad b > -a - 1\}. \tag{III.85}$$

Case II When $-1/4 < y_2 < 3/4$ the stability region (see Figure III.9) is given by $A = A^{II} = A_1^{II} \cup A_2^{II}$, where

$$A_1^{II} = \{(a,b) : \quad b < 1, \quad b > a - 1, \quad b > 0.25 + y_2, \quad b > a + 0.5 - 2y_2\}, \tag{III.86}$$
$$A_2^{II} = \{(a,b) : \quad b < 0.25 + y_2, \quad b < a + 0.5 - 2y_2 \quad b > -a - 1 \quad b > a - 1\}. \tag{III.87}$$

Substituting everything back will result in that all stable H312 will have coefficients of the fol-

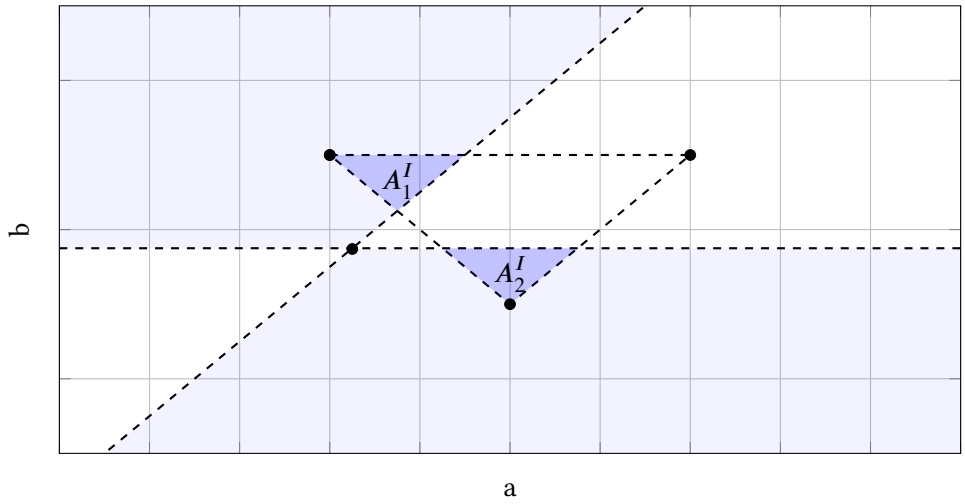

Figure III.8. The stability region for H312 when $-5/4 < y_2 \leq -1/4$. The dashed lines do not belong to the set.

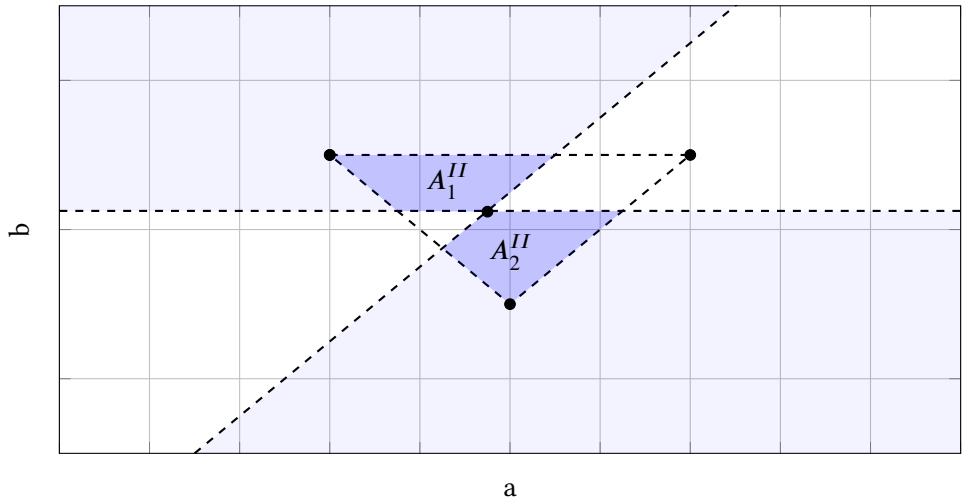

Figure III.9. The stability region for H312 when $-1/4 < y_2 < 3/4$. The dashed lines do not belong to the set.

lowing form:

$$\begin{cases} \alpha_2 = 3y_2 + (2b-a)\left(\dfrac{4y_2-b-a}{1+a-3b}\right) - b \\ \alpha_3 = y_2 \\ \\ \kappa\beta_1 = \left(\dfrac{4y_2-b-a}{1+a-3b}\right) b + y_2 \\ \kappa\beta_2 = 2\left(\dfrac{4y_2-b-a}{1+a-3b}\right) b + 2y_2 \\ \kappa\beta_3 = \left(\dfrac{4y_2-b-a}{1+a-3b}\right) b + y_2 \end{cases} \quad \text{where} \quad \begin{cases} (a,b) \in A(y_2) \\ -5/4 < y_2 < 3/4 \end{cases} \tag{III.88}$$

The family H312*b*

H312*b* are filters of third order dynamics with

$$\beta_1 = \beta_3 = \frac{1}{b\kappa}, \quad \beta_2 = \frac{2}{b\kappa}, \quad \alpha_2 = \frac{3}{b}, \quad \alpha_3 = \frac{1}{b}. \tag{III.89}$$

The closed loop poles of this family are

$$q = 0, 0, 1 - 4/b. \tag{III.90}$$

For stability we need $b \in]2, \infty[$, but if we do not want it to oscillate, we need to restrict it further with $b \geq 4$ (recommended value $b = 8$).

Stability analysis of H311

The parameters are given by

$$\begin{cases} \alpha_2 = y_1 \\ \alpha_3 = y_2 \\ \\ \kappa\beta_1 = x_1 \\ \kappa\beta_2 = x_1 + x_2 \\ \kappa\beta_3 = x_2 \end{cases} \quad \text{where} \quad y_1 \neq -y_2 - 1. \tag{III.91}$$

Let us denote $y_1 - y_2$ with y. By doing the same kind of analysis we did for H321, we will get the result that $-5/2 < y < 3/2$ for the stability region to be nonempty. This gives us a stability region $B(\alpha_3, \alpha_2)$ (see Figure III.10) given by

$$B(\alpha_3, \alpha_2) = \left\{ (\alpha_3, \alpha_2) : \quad \alpha_2 < \alpha_3 - \frac{1}{2}, \quad \alpha_2 > \alpha_3 - \frac{5}{2}, \quad \alpha_2 \neq -\alpha_3 - 1 \right\}. \tag{III.92}$$

Further analysis will result in three different cases

Case I When $-5/2 < y < -1/2$ the stability region (see Figure III.11) is given by $A = A^I$, where

$$A^I = \left\{ (a,b) : \quad b > \frac{a-2y-1}{2}, \quad b > -a-1, \quad b < 1 \right\}. \tag{III.93}$$

Case II When $-1/2 \leq y < 1/2$ the stability region (see Figure III.12) is given by $A = A^{II}$, where

$$A^{II} = \left\{ (a,b) : \quad b > \frac{a-2y-1}{2}, \quad b > -a-1, \quad b < 1, \quad a < 1-2y \right\}. \tag{III.94}$$

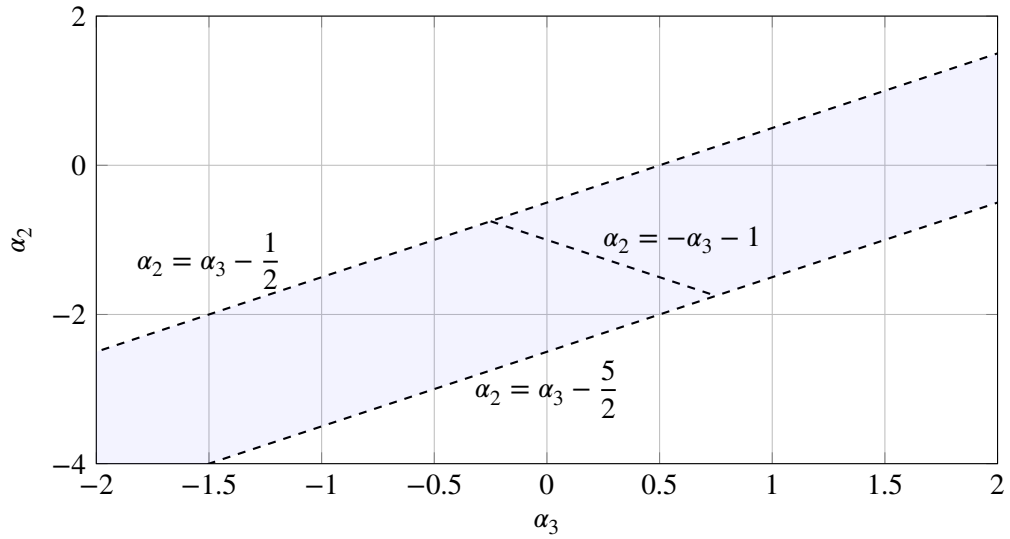

Figure III.10. The stability region for α_2 and α_3. The dashed lines do not belong to the set.

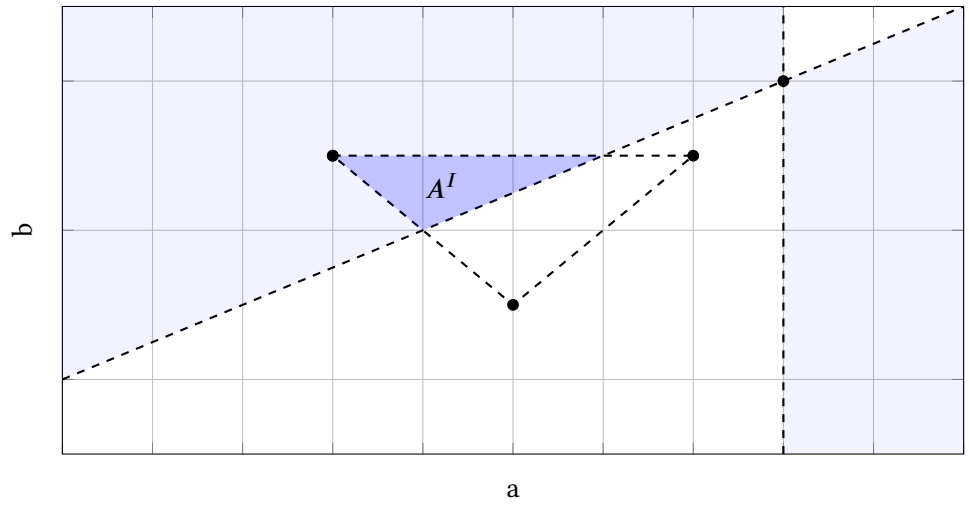

Figure III.11. The stability region for H311 when $-5/2 < y < -1/2$. The dashed lines do not belong to the set.

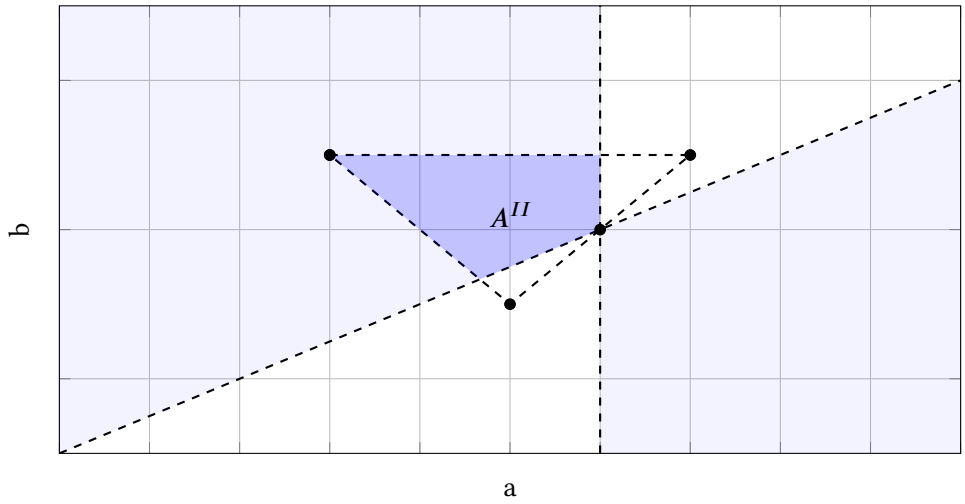

Figure III.12. The stability region for H311 when $-1/2 \leq y < 1/2$. The dashed lines do not belong to the set.

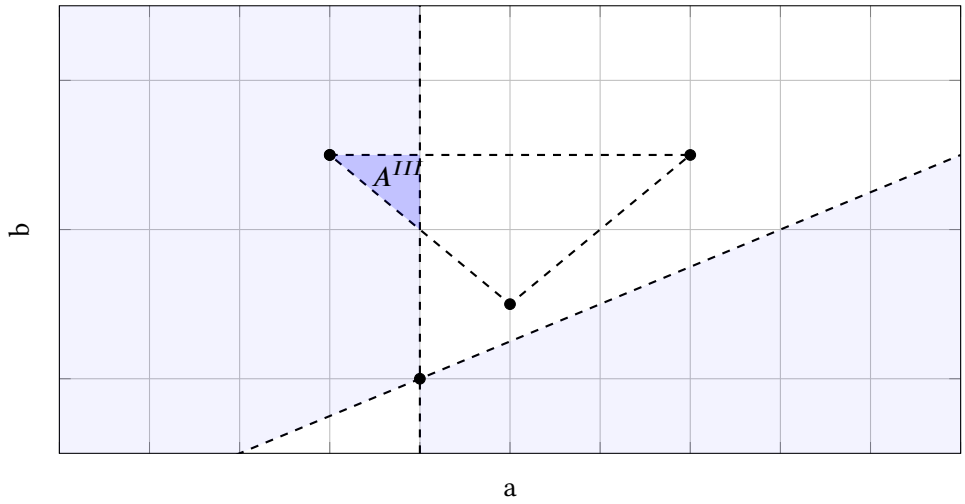

Figure III.13. The stability region for H311 when $1/2 \leq y < 3/2$. The dashed lines do not belong to the set.

Case III When $1/2 \leq y < 3/2$ the stability region (see Figure III.13) is given by $A = A^{III}$, where

$$A^{III} = \{(a,b) : \quad b > -a - 1, \quad b < 1, \quad a < 1 - 2y\}. \tag{III.95}$$

Substituting everything back will result in that all stable H311 will have coefficients of the following form

$$\begin{cases} \alpha_2 = y_1 \\ \alpha_3 = y_2 \\ \\ \kappa\beta_1 = b + (a-b)\left(\dfrac{2(y_1 - y_2) + b}{1 - a + b}\right) + y_1 - 2y_2 \\ \kappa\beta_2 = a\left(\dfrac{2(y_1 - y_2) + b}{1 - a + b}\right) + b + y_1 - y_2 \\ \kappa\beta_3 = \left(\dfrac{2(y_1 - y_2) + b}{1 - a + b}\right) b + y_2 \end{cases} \quad \text{where} \quad \begin{cases} (a,b) \in A(y_1 - y_2) \\ (y_2, y_1) \in B(y_2, y_1) \end{cases} \tag{III.96}$$

III.1.5. Summary of available filters

The filters discussed in the previous sections of part III, are summarized in Table III.1. This table shows all available filters of order one to three.

Table III.1. The whole set of filters where $1 \leq p_D \leq 3$. The classes marked by ▭ are non-empty, containing low pass filter, and those marked ▭ are less interesting since the corresponding filters are not low pass filters.

p_D	p_A	p_F	*name*	*low pass filter?*	*empty or not?*
1	1	0	H110	No	Not empty
2	1	0	H210	No	Not empty
2	1	1	H211	Yes	Not empty
2	2	0	H220	No	Not empty
2	2	1	H221	Yes	EMPTY
3	1	0	H310	No	Not empty
3	1	1	H311	Yes	Not empty
3	1	2	H312	Yes	Not empty
3	2	0	H320	No	Not empty
3	2	1	H321	Yes	Not empty
3	2	2	H322	Yes	EMPTY
3	3	0	H330	No	Not empty
3	3	1	H331	Yes	EMPTY
3	3	2	H332	Yes	EMPTY

III.2. Stiffness

When numerically integrating an ODE, it is expected that the step-size at a time point is depending on how much the system varies at that point, since too long steps where a lot of variations occur will result in bad resolution. Numerical solvers using error control will take care of this. However, sometimes the error control system will get the indication of a large error even when the solution to the system is smooth, which also will result in a smaller step-size. This phenomenon is what we call *stiffness*.

The *stiffness* of a system of equations has for long been a phenomenon intuitively understood. Numerical analysts know the symptoms, and how to deal with them. They generally agree upon the fact that it has something do with stability, rather than accuracy, however no precise mathematical definition has been accepted. Various stiffness measures have been proposed using the eigenvalues $\lambda_i, i = 1, 2, \dots, n$ of the local Jacobian of the ODE, for example,

1. *The stiffness index* L given by [4]

$$L = \max_i |\mathrm{Re}(\lambda_i)|. \tag{III.97}$$

   If L is large, then the problem is considered stiff.

2. *The stiffness ratio* S given by [22, 4]

$$S = \frac{\max_i |\mathrm{Re}(\lambda_i)|}{\min_i |\mathrm{Re}(\lambda_i)|}. \tag{III.98}$$

   If S is large, then the problem is considered stiff.

These stiffness measurements are adequate in some cases, but have been shown to not be general enough.

A new proposal to the phenomenon is given in [23]. This definition is what is used throughout the report.

Let a general ordinary differential equation (ODE) be given by:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^n, \quad t_0 \leq t \leq t_\mathrm{f}.$$

The stiffness of it can differ with every time point t; we call $s(t)$ the *stiffness indicator* at t. This measure can be calculated as follows [23]:

Let $\boldsymbol{J}$ be the Jacobian belonging to f. The first step is to symmetrize $\boldsymbol{J}$:

$$\boldsymbol{D}(t) = \frac{\boldsymbol{J}(t) + \boldsymbol{J}(t)^T}{2}. \tag{III.99}$$

Let the ordered eigenvalues of $\boldsymbol{D}(t)$ be $\lambda_1(t) \leq \lambda_2(t) \leq \ldots \leq \lambda_n(t)$. The stiffness indicator s is then defined as the mean value of the two extreme eigenvalues, i.e.,

$$s(t) = \frac{\lambda_1(t) + \lambda_n(t)}{2}. \tag{III.100}$$

If $s(t)$ is nonnegative, we say that the problem is non-stiff at this time point. Otherwise, the problem *might* be stiff depending on the magnitude of $|s(t)|$. As the value of $|s(t)|$ grows, the stiffness increases. The question is: What is 'small' and 'large' in this context?

To be able to quantify *stiffness*, we need a new stiffness measure. Since stiffness depends on both eigenvalues and integration time $T = t_f - t_0$, a new measure $\hat{s}$ can be introduced by normalizing the integration interval. We introduce the new variable $\theta = t/T$, and call it *the interval unit variable*. With the use of this variable, P is expressed as

$$\frac{d\boldsymbol{x}}{d\theta} = T\boldsymbol{f}(\theta T, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \quad \theta \in [0, 1], \tag{III.101}$$

which further gives us

$$\hat{s}(t) = T\frac{\lambda_1(t) + \lambda_n(t)}{2} = Ts(t), \tag{III.102}$$

where $\hat{s}$ is called the *normalized stiffness indicator* [23]. There is a caveat though: the dependency of $\hat{s}$ on the integration time T, makes it possible to make any problem infinitely stiff by integrating for a very long time. Also, this makes it possible to say that, for example, *the Lotka–Volterra problem* is stiffer than *the Oregonator*, by integrating the Lotka–Volterra problem over many cycles. By only using this measure for an appropriate integration time (for example, in the case of a periodic problem, letting the integration time be limited to one period), problems like these can be avoided.

A problem is called stiff, if any part of the corresponding integration interval is stiff according to this theory.

III.3. Test library

To test the three different solvers, a test library implemented by the author was used (the implementation can be found at [3]). This library consists of both stiff and non-stiff problems, using the definition of stiffness given in [23]. Throughout this report we are going to use the following definition of *problem*:

Definition III.3.1. *A problem consists of one function $\boldsymbol{f}$, one time span $[t_0, t_f]$, and one initial condition $\boldsymbol{x}_0$, making up the initial value problem (IVP)*

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \quad t \in [t_0, t_f] \quad \textit{where} \quad \boldsymbol{x} \in \mathbb{R}^n.$$

This means that when we refer to the problem *HIRES*, we mean the RHS-function, the time span (where the integration is required) and the initial condition, as given in Subsection III.3.1. In some cases, the RHS-function contains a parameter. For example, *problem 8, Van der Pol*, has the parameter μ. Thus, when we refer to this problem, we allow any value of μ, unless specifically mentioned, as in *Van der Pol using* $\mu = 1$.

Problems 1-11 are chosen from Bari's test library (see [12]), problem 12 — Lotka–Volterra — is a well known test problem, problem 13 — The flame propagation problem — is taken from [15] and the problems 14-17 are chosen from ODELab [18]. The theoretical background and mathematical descriptions of all of these problems, can be found in this section.

III.3.1. Problem 1: The HIRES problem

The name *HIRES* is short for *High Irradiance RESponse*. The problem originates from plant physiology, modeling the involvement of light in morphogenesis. More precisely, it models the high irradiance responses of photomorphogenesis on the basis of phytochrome, by means of a chemical reaction involving eight reactants. This problem consists of 8 non-linear ODEs.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^8, \quad 0 \leq t \leq 321.8122\,\text{s}.$$

Component x_i models the concentration (unit: mol) of the i:th reactant in the process over time (unit: seconds) (see Table III.2 for reactant correspondence). The RHS-function f is defined as:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} -k_1x_1 + k_2x_2 + k_6x_3 + o_{k_s} \\ k_1x_1 - (k_2 + k_3)x_2 \\ -(k_1 + k_6)x_3 + k_2x_4 + k_5x_5 \\ k_3x_2 + k_1x_3 - (k_2 + k_4)x_4 \\ -(k_1 + k_5)x_5 + k_2x_6 + k_2x_7 \\ k_4x_4 + k_1x_5 - k_2x_6 + k_-x_7 - k_+x_6x_8 \\ -(k_2 + k_- + k_*)x_7 + k_+x_6x_8 \\ (k_2 + k_- + k_*)x_7 - k_+x_6x_8 \end{pmatrix}, \tag{III.103}$$

where $k_1, ..., k_6, k_+, k_-, k_*$ and o_{k_s} are reaction parameters given in Table III.2. The initial condition is given by:

$$\boldsymbol{x}_0 = \left(1, 0, 0, 0, 0, 0, 0, 0.0057\right)^{\mathrm{T}}. \tag{III.104}$$

Table III.2. Reaction parameters, compound correspondence and description of the different compounds.

Reaction parameters			*Correspondence*			*Description*
k_1	=	1.71	x_1	$[P_r]$	P_r	Red absorbing form of phytochrome
k_2	=	0.43	x_2	$[P_{fr}]$	P_{fr}	Far red absorbing form of phytochrome
k_3	=	8.32	x_3	$[P_rX]$	X	Receptor I
k_4	=	0.69	x_4	$[P_{fr}X]$	X’	Receptor II
k_5	=	0.035	x_5	$[P_rX']$	E	Enzyme
k_6	=	8.32	x_6	$[P_{fr}X']$	P_rX	P_r bound by X
k_+	=	280	x_7	$[P_{fr}X'E]$	$P_{fr}X$	P_{fr} bound by X
k_-	=	0.69	x_8	[E]	P_rX'	P_r bound by X’
k_*	=	0.69			$P_{fr}X'$	P_{fr} bound by X’
o_{k_s}	=	0.0007			$P_{fr}X'E$	P_{fr} bound by X’ and partially influenced by E

The eight solution components to this problem can be seen in Figure III.14. Notice the different y-scaling, and the fact that we in the case of some of the components have zoomed in on the time interval $[0, 5]$ with the reason that this is the interval where the components in question are varying.

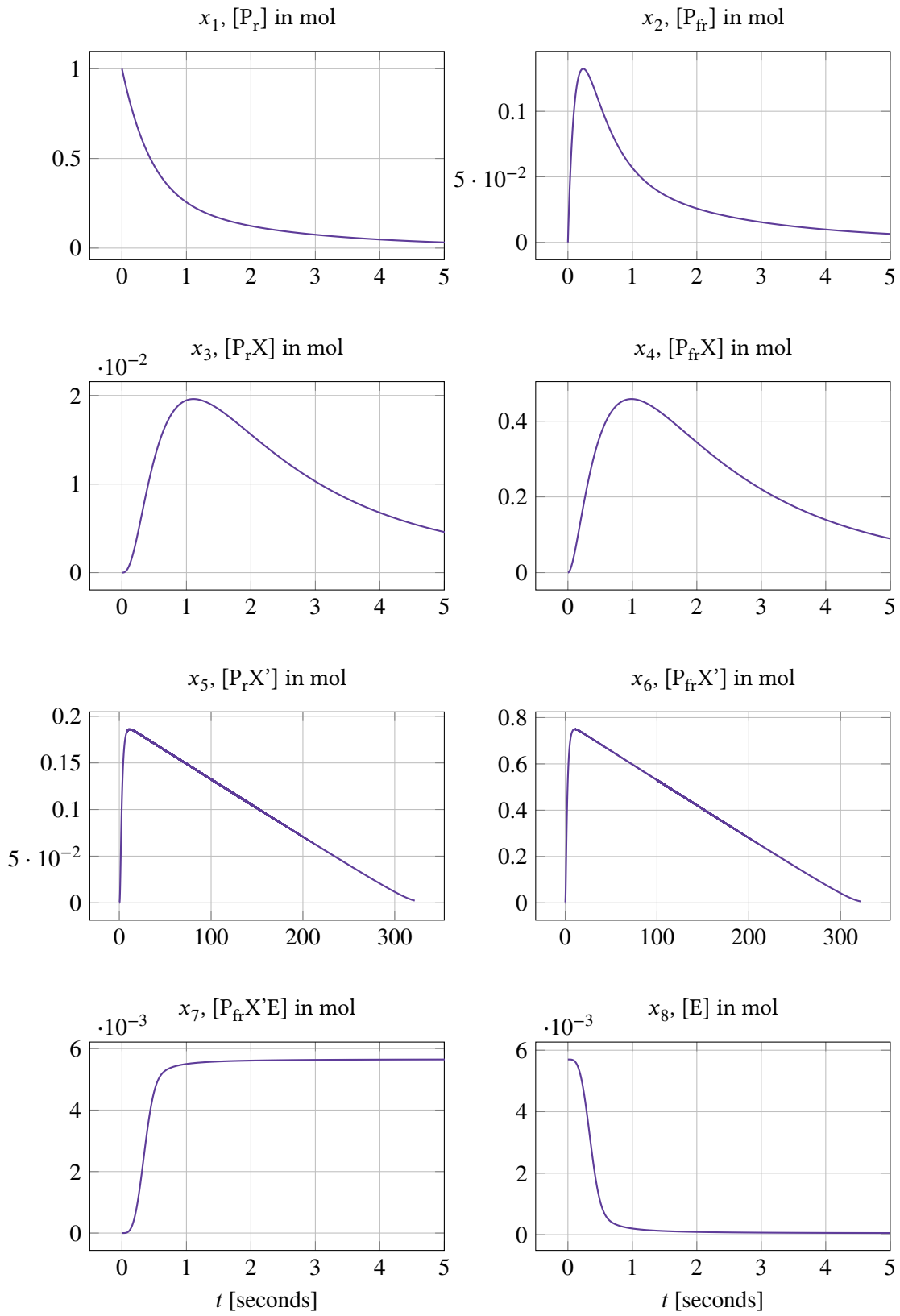

Figure III.14. The 8 solution components of *HIRES* as a function of time. Notice the different y-scaling and the fact that some of the components are zoomed in on the time interval $[0, 5]$ (the part where these components vary).

III.3.2. Problem 2: The pollution problem

This IVP is connected to the air pollution model developed at The Dutch National Institute of Public Health and Environmental Protection (RIVM). More precisely it models the chemical reaction part of this air pollution model, and consists of 25 chemical reactions and 20 reacting compounds.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^{20}, \quad 0 \leq t \leq 60\,\text{s}.$$

Component x_i models the concentration (unit: mol) of the i:th substance in the process as a function of time (unit: seconds) (see Table III.3 for substance correspondence). The RHS-function $\boldsymbol{f}$ is defined as:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} -\sum_{j\in\{1,10,14,23,24\}} r_j + \sum_{j\in\{2,3,9,11,12,22,25\}} r_j \\ -r_2 - r_3 - r_9 - r_{12} + r_1 + r_{21} \\ -r_{15} + r_1 + r_{17} + r_{19} + r_{22} \\ -r_2 - r_{16} - r_{17} - r_{23} + r_{15} \\ -r_3 + 2r_4 + r_6 + r_7 r_{13} + r_{20} \\ -r_6 - r_8 - r_{14} - r_{20} + r_3 + 2r_{18} \\ -r_4 - r_5 - r_6 + r_{13} \\ r_4 + r_5 + r_6 + r_7 \\ -r_7 - r_8 \\ -r_{12} + r_7 + r_9 \\ -r_9 - r_{10} + r_8 + r_{11} \\ r_9 \\ -r_{11} + r_{10} \\ -r_{13} + r_{12} \\ r_{14} \\ -r_{18} - r_{19} + r_{16} \\ -r_{20} \\ r_{20} \\ -r_{21} - r_{22} - r_{24} + r_{23} + r_{25} \\ -r_{25} + r_{24} \end{pmatrix}, \tag{III.105}$$

where r_i is the i:th auxiliary variable defined in Table III.3. The initial condition is given by:

$$\boldsymbol{x}_0 = \left(0, 0.2, 0, 0.04, 0, 0, 0.1, 0.3, 0.01, 0, 0, 0, 0, 0, 0, 0, 0.007, 0, 0, 0\right)^{\mathrm{T}}. \tag{III.106}$$

The 20 solution components to this problem can be seen in Figure III.15—III.17. Notice the different y-scaling, and the fact that we in some of the sub-figures have zoomed in on the time interval $[0, 1]$ with the reason that this is the only part of the interval where the components in question are varying.

Table III.3. The auxiliary variables r_i, the parameter values k_i and the compound correspondence to the variables x_i

Auxiliary variables			*Parameter values*			*Compounds*	
r_1	=	$k_1 \cdot x_1$	k_1	=	0.350	x_1	$[NO_2]$
r_2	=	$k_2 \cdot x_2 \cdot x_4$	k_2	=	$0.266 \cdot 10^2$	x_2	$[NO]$
r_3	=	$k_3 \cdot x_5 \cdot x_2$	k_3	=	$0.123 \cdot 10^5$	x_3	$[O_3P]$
r_4	=	$k_4 \cdot x_7$	k_4	=	$0.860 \cdot 10^{-3}$	x_4	$[O_3]$
r_5	=	$k_5 \cdot x_7$	k_5	=	$0.820 \cdot 10^{-3}$	x_5	$[HO_2]$
r_6	=	$k_6 \cdot x_7 \cdot x_6$	k_6	=	$0.150 \cdot 10^5$	x_6	$[OH]$
r_7	=	$k_7 \cdot x_9$	k_7	=	$0.130 \cdot 10^{-3}$	x_7	$[HCHO]$
r_8	=	$k_8 \cdot x_9 \cdot x_6$	k_8	=	$0.240 \cdot 10^5$	x_8	$[CO]$
r_9	=	$k_9 \cdot x_{11} \cdot x_2$	k_9	=	$0.165 \cdot 10^5$	x_9	$[ALD]$
r_{10}	=	$k_{10} \cdot x_{11} \cdot x_1$	k_{10}	=	$0.900 \cdot 10^4$	x_{10}	$[MEO_2]$
r_{11}	=	$k_{11} \cdot x_{13}$	k_{11}	=	$0.220 \cdot 10^{-1}$	x_{11}	$[C_2O_3]$
r_{12}	=	$k_{12} \cdot x_{10} \cdot x_2$	k_{12}	=	$0.120 \cdot 10^5$	x_{12}	$[CO_2]$
r_{13}	=	$k_{13} \cdot x_{14}$	k_{13}	=	$0.188 \cdot 10$	x_{13}	$[PAN]$
r_{14}	=	$k_{14} \cdot x_1 \cdot x_6$	k_{14}	=	$0.163 \cdot 10^5$	x_{14}	$[CH_3O]$
r_{15}	=	$k_{15} \cdot x_3$	k_{15}	=	$0.480 \cdot 10^7$	x_{15}	$[HNO_3]$
r_{16}	=	$k_{16} \cdot x_4$	k_{16}	=	$0.350 \cdot 10^{-3}$	x_{16}	$[O_1D]$
r_{17}	=	$k_{17} \cdot x_4$	k_{17}	=	$0.175 \cdot 10^{-1}$	x_{17}	$[SO_2]$
r_{18}	=	$k_{18} \cdot x_{16}$	k_{18}	=	$0.100 \cdot 10^9$	x_{18}	$[SO_4]$
r_{19}	=	$k_{19} \cdot x_{16}$	k_{19}	=	$0.444 \cdot 10^{12}$	x_{19}	$[NO_3]$
r_{20}	=	$k_{20} \cdot x_{17} \cdot x_6$	k_{20}	=	$0.124 \cdot 10^4$	x_{20}	$[N_2O_5]$
r_{21}	=	$k_{21} \cdot x_{17} \cdot x_6$	k_{21}	=	$0.210 \cdot 10$		
r_{22}	=	$k_{22} \cdot x_{19}$	k_{22}	=	$0.578 \cdot 10$		
r_{23}	=	$k_{23} \cdot x_1 \cdot x_4$	k_{23}	=	$0.474 \cdot 10^{-1}$		
r_{24}	=	$k_{24} \cdot x_{19} \cdot x_1$	k_{24}	=	$0.178 \cdot 10^4$		
r_{25}	=	$k_{25} \cdot x_{20}$	k_{25}	=	$0.312 \cdot 10$		

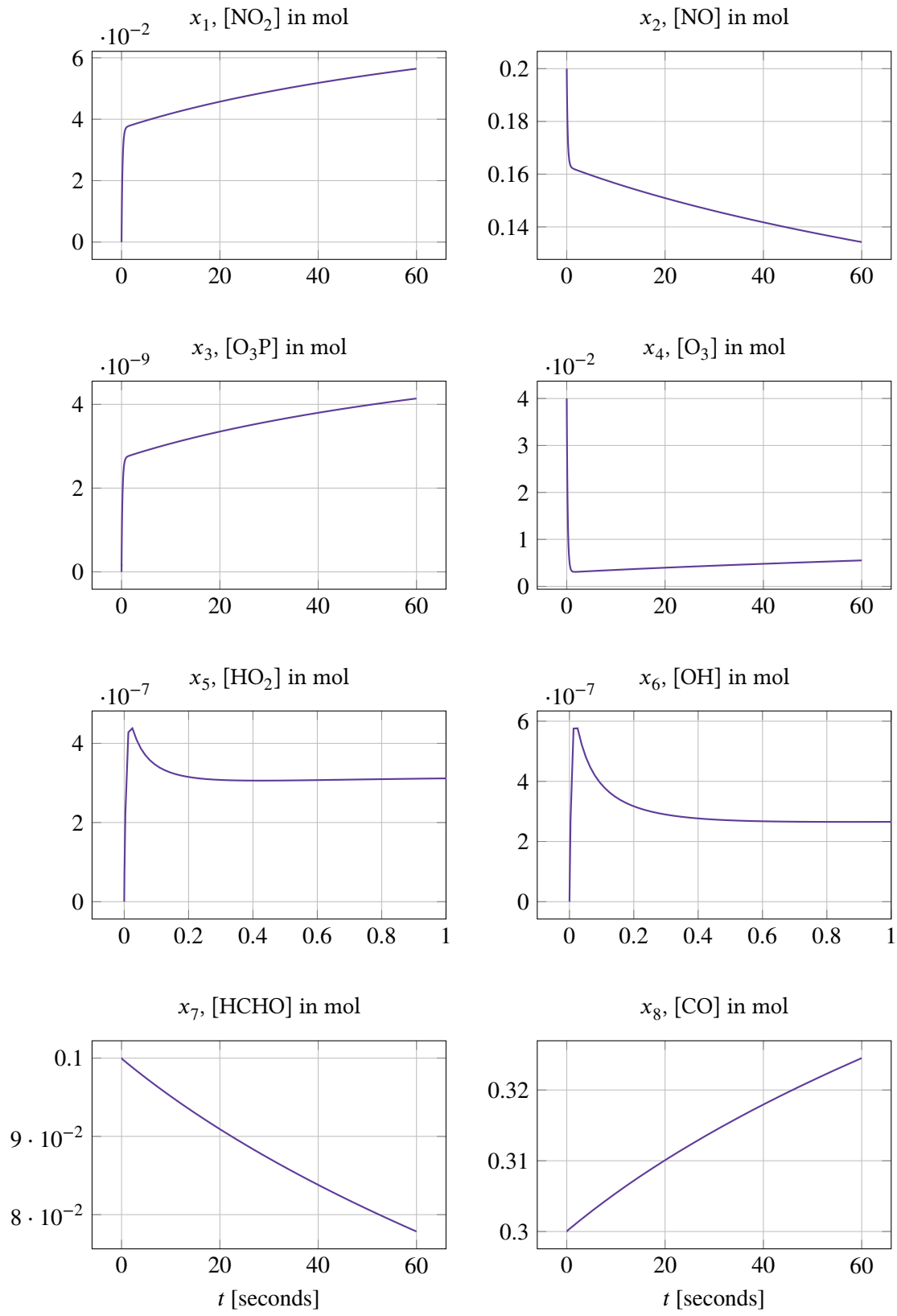

Figure III.15. The solution components x_1 – x_8 of *the pollution problem* as a function of time. Notice the different y-scaling and that, in the case of some of the components, we have zoomed in on the sub-interval $[0, 1]$ since this is the part where these components vary.

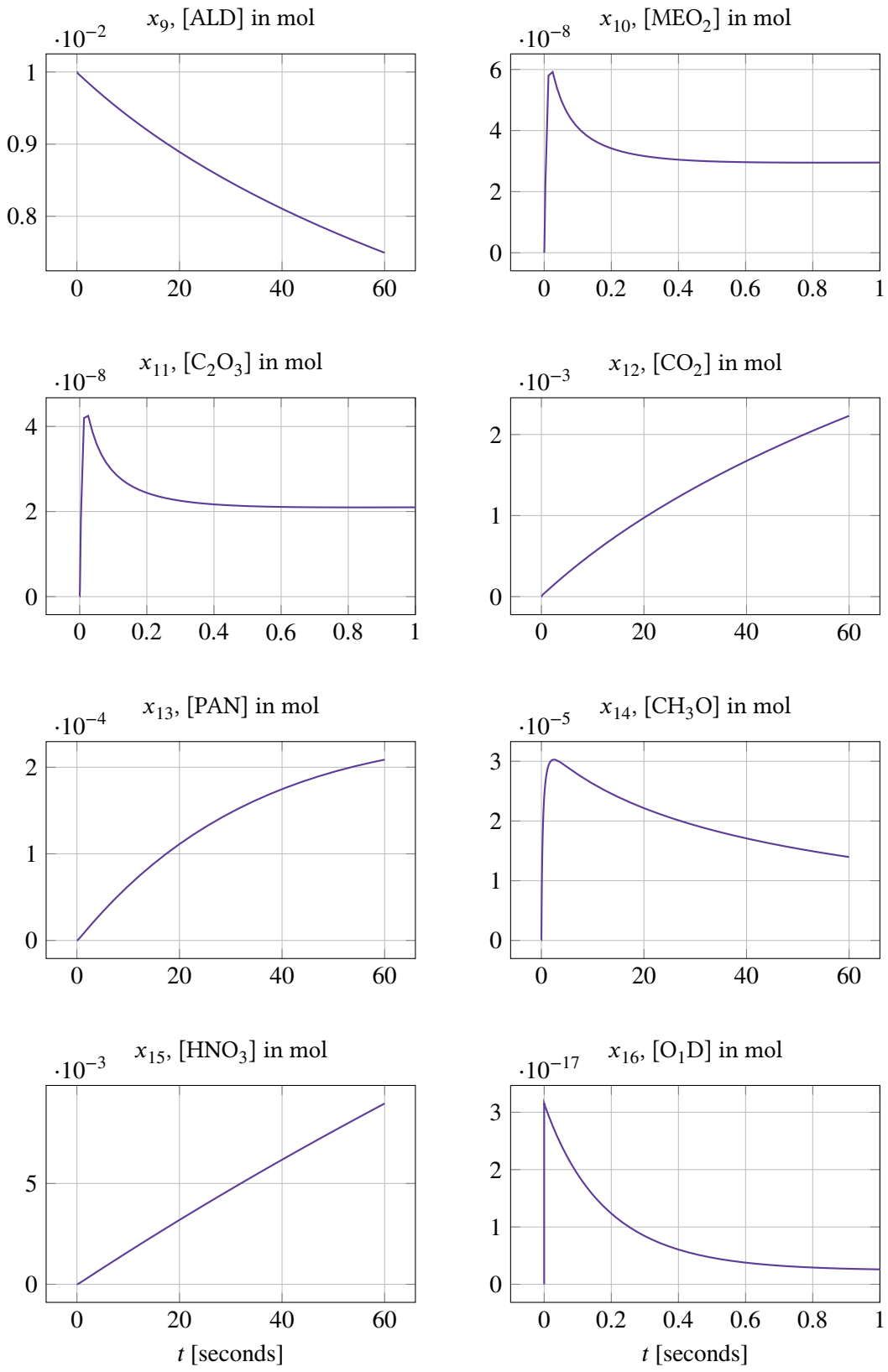

Figure III.16. The solution components x_9 – x_{16} of *the pollution problem* as a function of time. Notice the different y-scaling and that, in the case of some of the components, we have zoomed in on the sub-interval $[0, 1]$ since this is the part where these components vary.

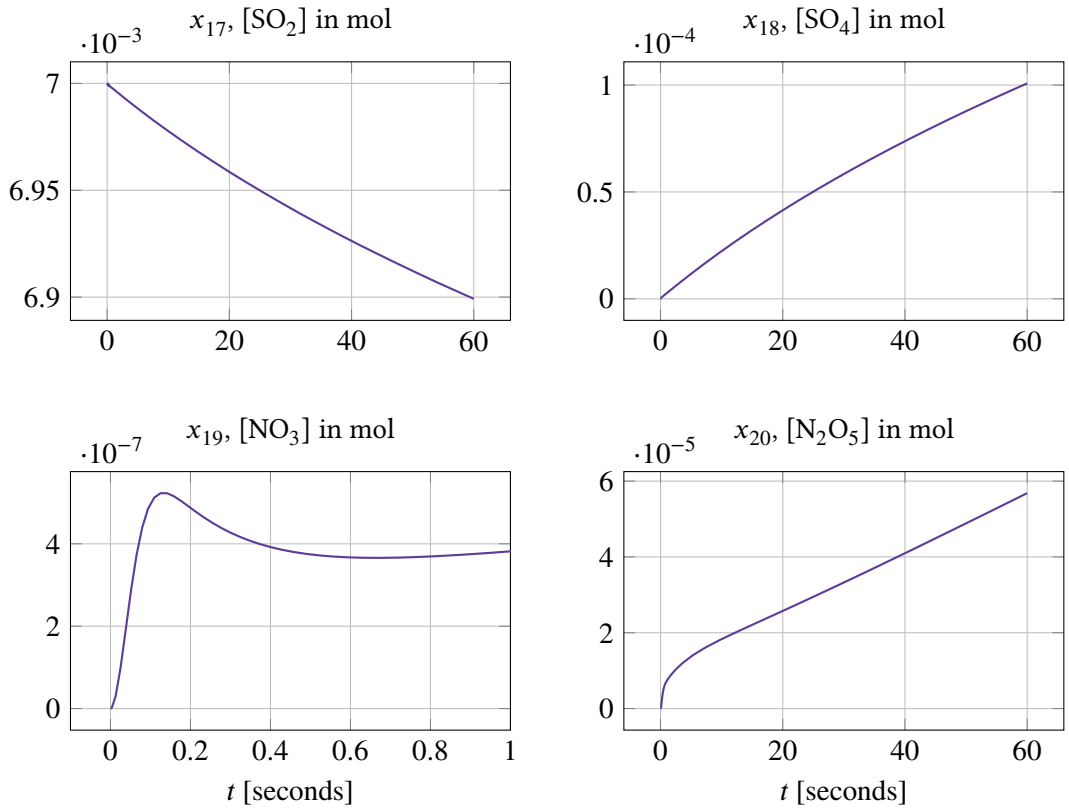

Figure III.17. The solution components $x_{17} - x_{20}$ of *the pollution problem* as a function of time. Notice the different y-scaling and that, in the case of x_{19}, we have zoomed in on the sub-interval $[0, 1]$ since this is the part where the component is varying.

III.3.3. Problem 3: The ring modulator problem

This IVP originates from electrical circuit theory. The system of equations consists of 15 differential equations and models the voltages and currents in a special kind of ring modulator.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^{15}, \quad 0 \leq t \leq 10^{-3}\,\text{s}.$$

The 7 first components $x_i, i \in \{1, \ldots, 7\}$, correspond to the 7 voltages $U_i, i \in \{1, \ldots, 7\}$ (unit: volts) in the circuit, and other 8 components $x_i, i \in \{8, \ldots, 15\}$ correspond to the 8 currents $I_i, i \in \{1, \ldots, 8\}$ (unit: amperes) in the circuit. The RHS-function $\boldsymbol{f}$ is defined as:

$$\boldsymbol{f}(t, \boldsymbol{x}) = \begin{pmatrix} \left(x_8 - 0.5x_{10} + 0.5x_{11} + x_{14} - x_1/R\right)/C \\ \left(x_9 - 0.5x_{12} + 0.5x_{13} + x_{15} - x_2/R\right)/C \\ \left(x_{10} - q(U_{\text{D1}}) + q(U_{\text{D4}})\right)/C_{\text{s}} \\ \left(-x_{11} + q(U_{\text{D2}}) - q(U_{\text{D3}})\right)/C_{\text{s}} \\ \left(x_{12} + q(U_{\text{D1}}) - q(U_{\text{D3}})\right)/C_{\text{s}} \\ \left(-x_{13} - q(U_{\text{D2}}) + q(U_{\text{D4}})\right)/C_{\text{s}} \\ \left(-x_7/R_{\text{p}} + q(U_{\text{D1}} + q(U_{\text{D2}} - q(U_{\text{D3}} - q(U_{\text{D4}})\right)/C_{\text{p}} \\ -x_1/L_{\text{h}} \\ -x_2/L_{\text{h}} \\ \left(0.5x_1 - x_3 - R_{\text{g2}}x_{10}\right)/L_{\text{s2}} \\ \left(-0.5x_1 + x_4 - R_{\text{g3}}x_{11}\right)/L_{\text{s3}} \\ \left(0.5x_2 - x_5 - R_{\text{g2}}x_{12}\right)/L_{\text{s2}} \\ \left(-0.5x_2 + x_6 - R_{\text{g3}}x_{13}\right)/L_{\text{s3}} \\ \left(-x_1 + U_{\text{in1}}(t) - (R_i + R_{\text{g1}})x_{14}\right)/L_{\text{s1}} \\ \left(-x_2 - (R_{\text{c}} + R_{\text{g1}})x_{15}\right)/L_{\text{s1}} \end{pmatrix}, \tag{III.107}$$

where

$$U_{\text{D1}} = x_3 - x_5 - x_7 - U_{\text{in2}}(t), \tag{III.108}$$
$$U_{\text{D2}} = -x_4 + x_6 - x_7 - U_{\text{in2}}(t), \tag{III.109}$$
$$U_{\text{D3}} = x_4 + x_5 + x_7 + U_{\text{in2}}(t), \tag{III.110}$$
$$U_{\text{D1}} = -x_3 - x_6 + x_7 + U_{\text{in2}}(t), \tag{III.111}$$
$$q(U) = \gamma\left(e^{U\delta} - 1\right), \tag{III.112}$$
$$U_{\text{in1}}(t) = 0.5\sin(2000\pi t), \tag{III.113}$$
$$U_{\text{in2}}(t) = 2\sin(20000\pi t), \tag{III.114}$$

and the constants given in Table III.4. The corresponding initial condition is given by:

$$\boldsymbol{x}_0 = \left(0, 0.2, 0, 0.04, 0, 0, 0.1, 0.3, 0.01, 0, 0, 0, 0, 0, 0, 0, 0.007, 0, 0, 0\right)^{\text{T}}. \tag{III.115}$$

The 15 solution components to this problem can be seen in Figure III.18—III.19. Notice the different y-scaling in the sub-figures.

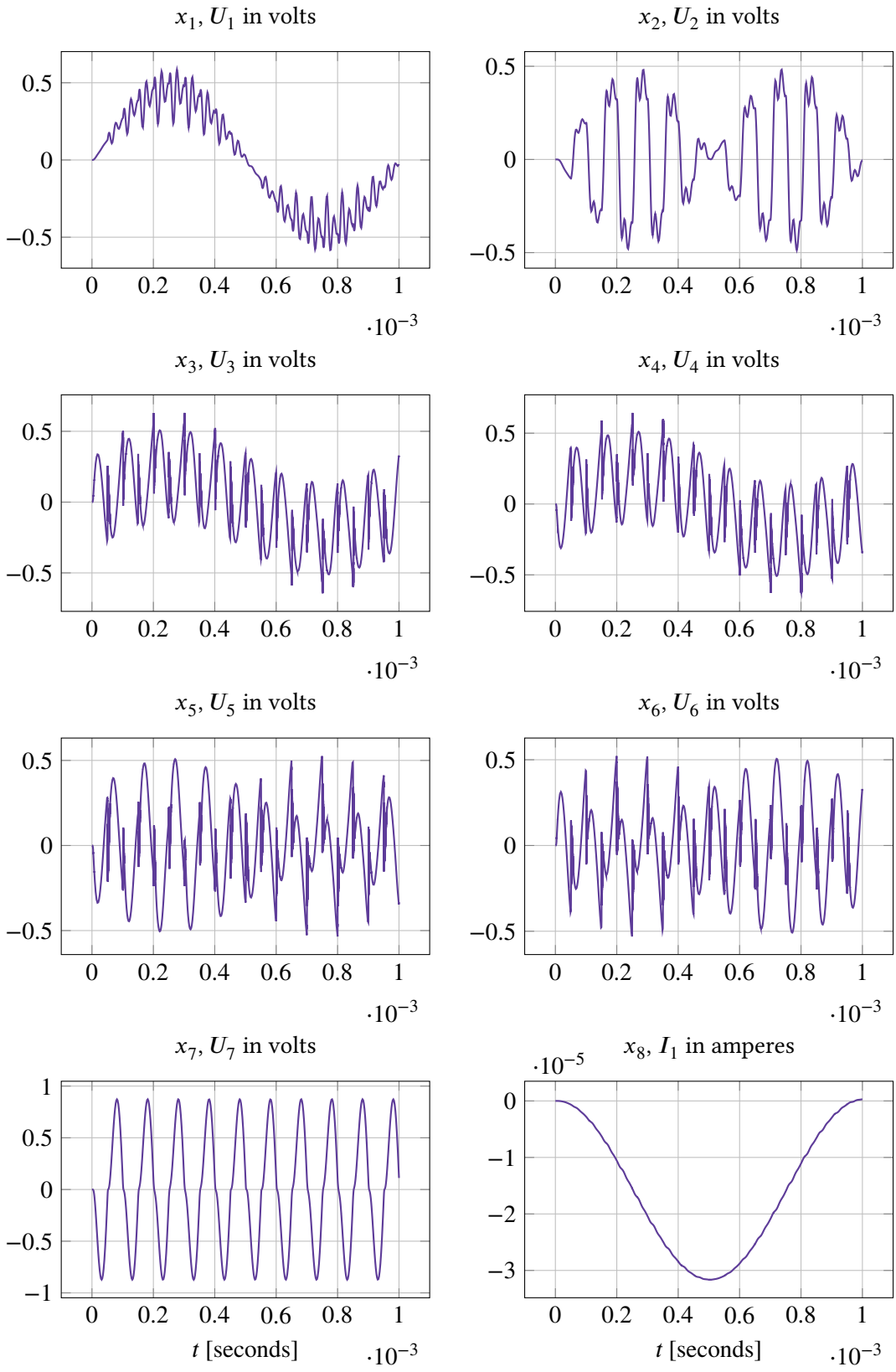

Figure III.18. The solution components x_1 – x_8 of *the ring modulator problem* as a function of time. Notice the different y-scaling.

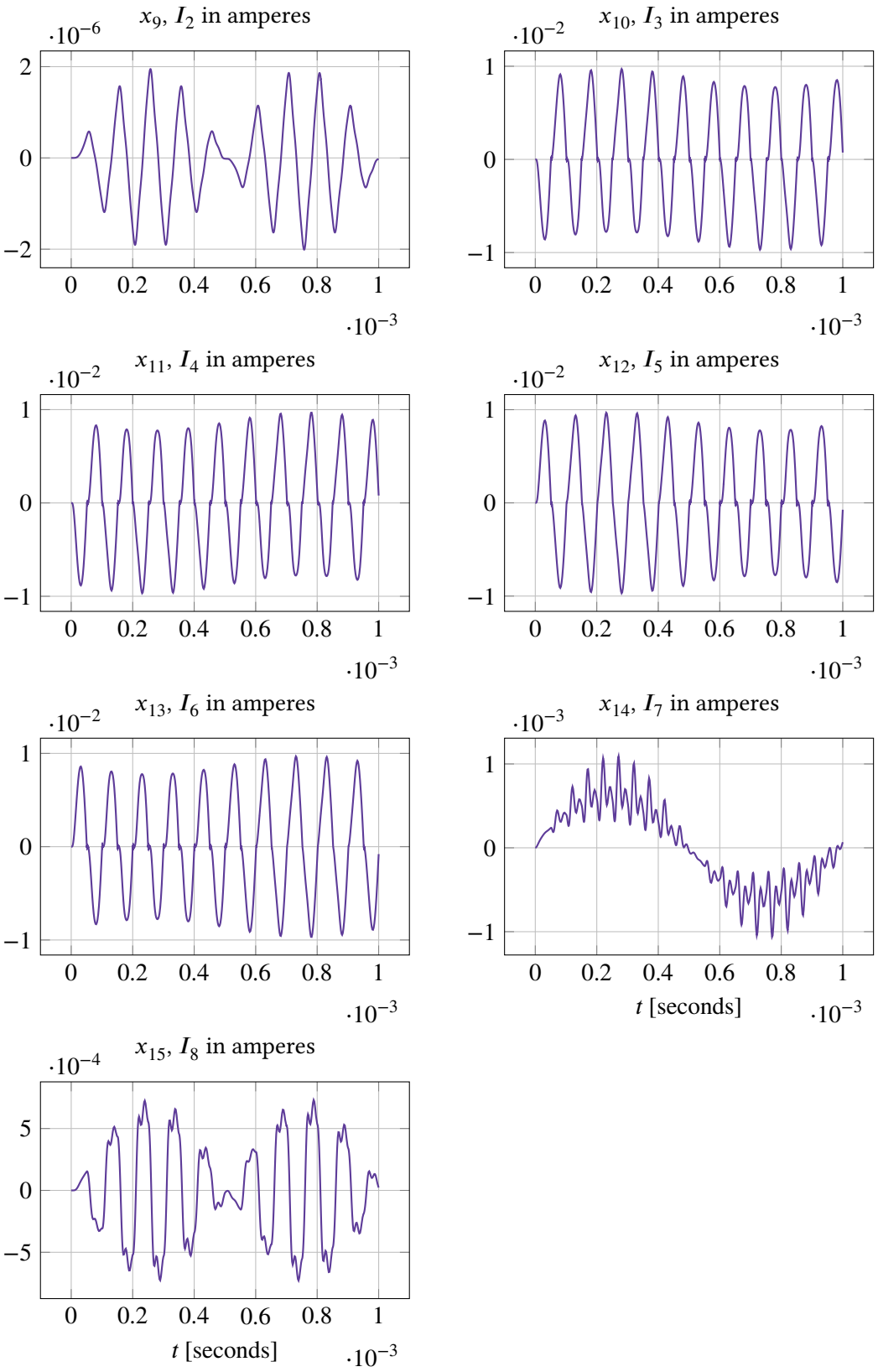

Figure III.19. The solution components $x_9 - x_{15}$ of *the ring modulator problem* as a function of time. Notice the different y-scaling.

Table III.4. All constants belonging to *the ring modulator problem.*

Resistances [Ω]			*Inductances* [H]			*Capacitances* [F]			*Unit-less constants*		
R	=	25000	L_{h}	=	4.45	C	=	$1.6 \cdot 10^{-8}$	γ	=	111
R_{p}	=	50	L_{s1}	=	0.002	C_{s}	=	$2 \cdot 10^{-12}$	δ	=	111
R_{g1}	=	36.3	L_{s2}	=	$5 \cdot 10^{-4}$	C_{p}	=	10^{-8}			
R_{g2}	=	17.3	L_{s3}	=	$5 \cdot 10^{-4}$						
R_{g3}	=	17.3									
R_{i}	=	50									
R_{c}	=	600									

III.3.4. Problem 4: The Medical Akzo Nobel problem

The Medical Akzo Nobel problem has its origin in medicine. It was formulated during a study about the penetrability of radio-labeled antibodies into a tumor-infected tissue. The study was performed by the Akzo Nobel research laboratories, and the problem formulated by the same.

The system originates from the chemical reaction:

$$A + B \xrightarrow{k} C,$$

where A is the radio-labeled anti-body, B is tumor-infected tissue, C is the chemical product and k is the rate constant for the reaction. The concentration of A, u, and the concentration of B, v, are functions of time t and space x ($x \in \mathbb{R}$). This problem assumes a semi-infinite, one-dimensional slab:

$$S_T(x,t) = \{(x,t) : 0 < x < \infty,\ 0 < t < T\}, \tag{III.116}$$

consisting of uniformly distributed tissue B. When the surface of the slab ($x = 0$) is exposed to the chemical A, this chemical starts to penetrate into the slab ($x > 0$). To be able to solve the problem numerically, the semi-infinite slab is transformed into a finite one according to the following rule:

$$\zeta = \frac{x}{x+c}, \quad c > 0, \tag{III.117}$$

resulting in

$$S_T(\zeta,t) = \{(\zeta,t) : 0 < \zeta < 1,\ 0 < t < T\}. \tag{III.118}$$

By using the method of lines, ζ is discretized into N points:

$$\zeta_i = i \cdot \Delta\zeta, \quad i = 1, 2, \ldots, N, \quad \Delta\zeta = \frac{1}{N}. \tag{III.119}$$

At each of these points, the concentration of A and B are measured, creating an equation system of size $2N$.

The mathematical formulation is the following:

$$\dot{\boldsymbol{y}} = \boldsymbol{f}(t, \boldsymbol{y}), \quad \boldsymbol{y}(0) = \boldsymbol{y}_0,$$

where

$$\boldsymbol{y} \in \mathbb{R}^{2N}, \quad 0 \le t \le 20\,\text{s}.$$

The component $y_i, i \in \{1, 3, \ldots, 2N-1\}$ corresponds to the concentration (unit: mol) of A at the grid point $\zeta_{(i+1)/2}$, and the component $y_i, i \in \{2, 4, \ldots, 2N\}$ corresponds to the concentration (unit: mol) of B at grid point $\zeta_{i/2}$. The components of the RHS-function $\boldsymbol{f}$ is given by

$$f_{2j-1} = \alpha_j \frac{y_{2j+1} - y_{2j-3}}{2\Delta\zeta} + \beta_j \frac{y_{2j-3} - 2y_{2j-1} + y_{2j+1}}{(\Delta\zeta)^2} - k y_{2j} y_{2j-1}, \tag{III.120}$$

$$f_{2j} = -k y_{2j} y_{2j-1}, \tag{III.121}$$

where $j = 1, \ldots, N$ and

$$y_{-1}(t) = \phi(t), \qquad \alpha_j = \frac{2(j\Delta\zeta - 1)^3}{c^2}, \quad \phi(t) = \begin{pmatrix} 2 \text{ for } t \in [0,5] \\ 0 \text{ for } t \in [5,20] \end{pmatrix}, \tag{III.122}$$

$$y_{2N+1} = y_{2N-1}, \quad \beta_j = \frac{(j\Delta\zeta - 1)^4}{c^2}, \qquad \Delta\zeta = \frac{1}{N}. \tag{III.123}$$

Values of the constants and initial condition of this problem are $k = 100$, $c = 4$ and

$$\boldsymbol{y}_0 = \big(0, 1, 0, 1, \ldots, 0, 1\big)^{\mathrm{T}}. \tag{III.124}$$

The 12 solution components to this problem, using $N = 100$, can be seen in Figure III.20—III.21. Notice the different y-scaling.

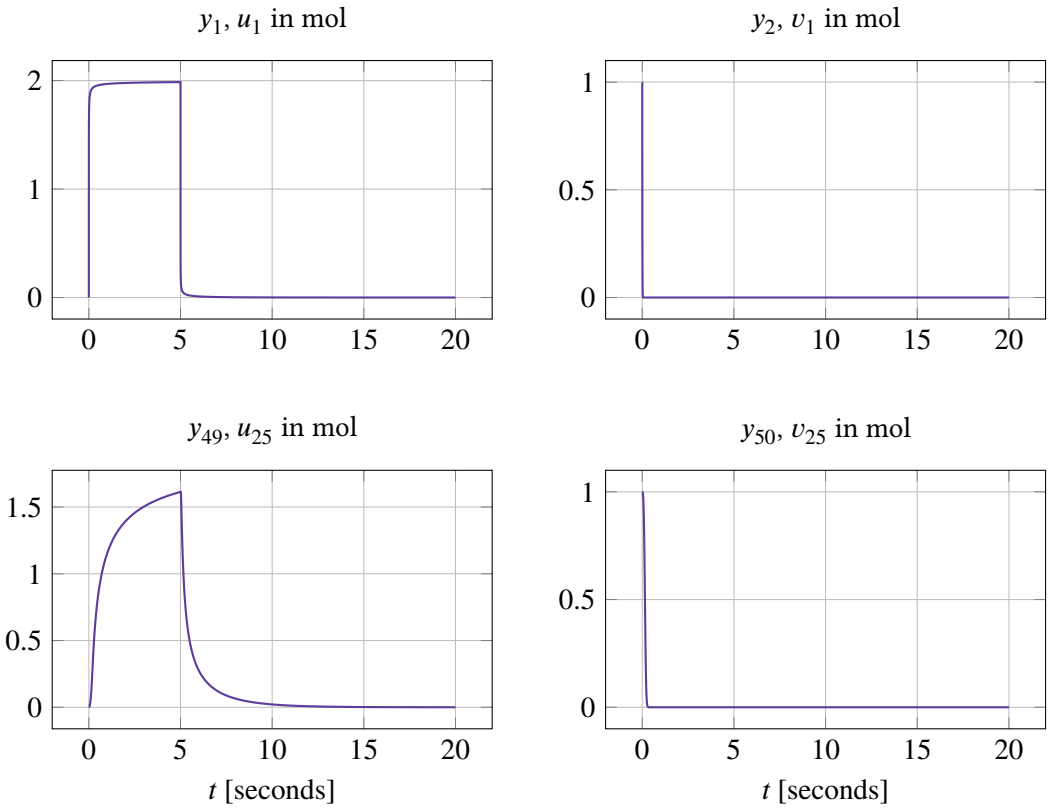

Figure III.20. Four of the solution components of *Medical Akzo Nobel* as a function of time. The sub-figures represent the concentration of A and B in the grid points ζ_1 and ζ_{25} as a function of time.

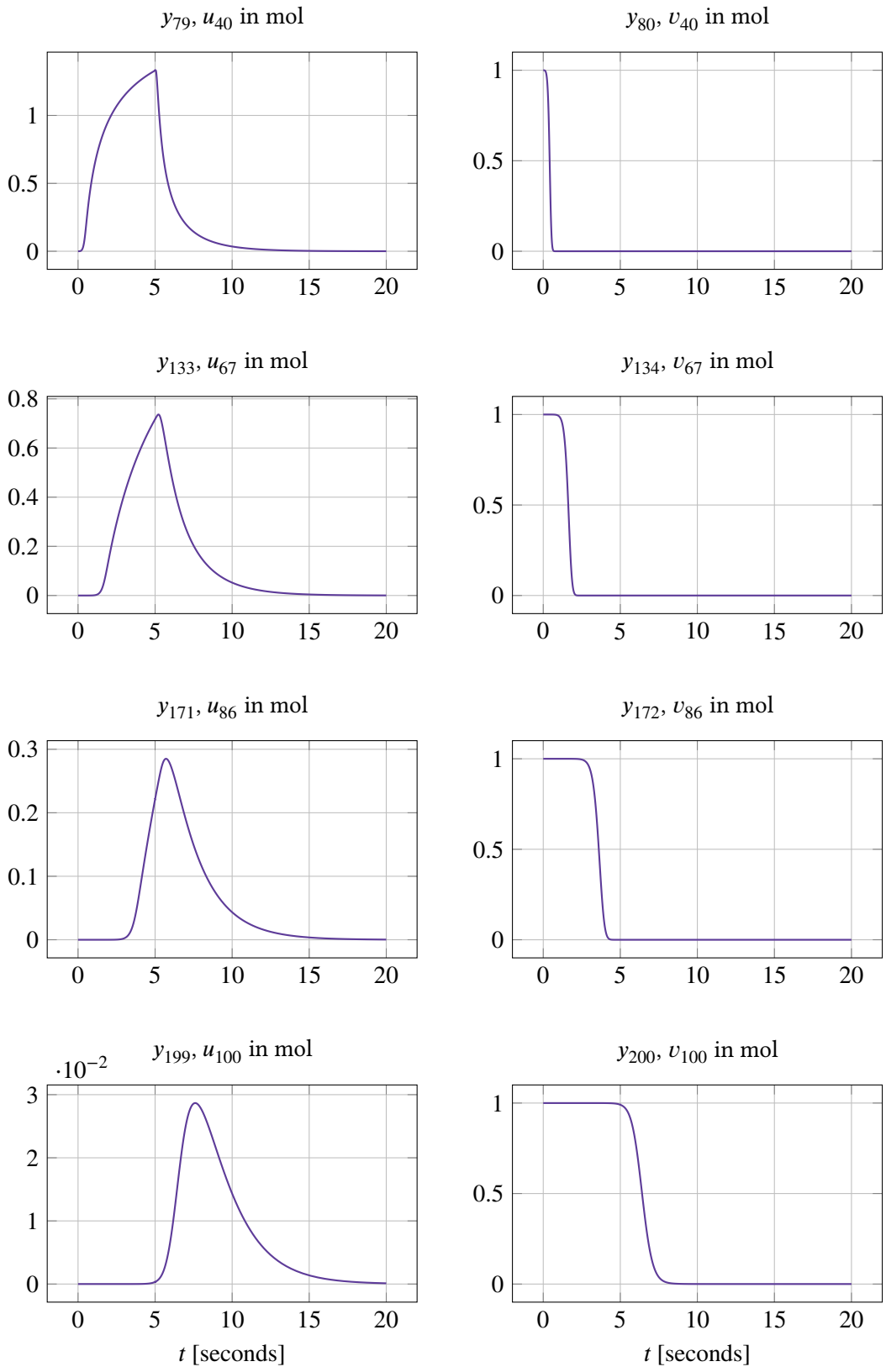

Figure III.21. Eight of the solution components of *Medical Akzo Nobel* as a function of time. The sub-figures represent the concentration of A and B in the grid points ζ_{40}, ζ_{67}, ζ_{86} and ζ_{100} as a function of time. Notice how the cancer concentration decreases, when the grid point receives the anti-body.

III.3.5. Problem 5: The EMEP problem

This IVP is connected to the EMEP MSC-W ozone chemistry model developed at the Norwegian Meteorological Institute in Oslo, Norway. More precisely it models the chemistry part of this ozone model consisting of about 144 chemical reactions and 66 reacting compounds.

The mathematical formulation of the problem is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^{66}, \quad 14400 \leq t \leq 417600\,\text{s}.$$

Component x_i, $i \in \{1, \ldots, 66\}$ models the concentration (unit: molecules per cm^3) of substance i (see Table III.5 for substance correspondence) as a function of time (unit: seconds). The RHS-function $\boldsymbol{f}$ of this problem can not be displayed here due to its size, though the implementation of it can be found at [3].

The initial condition is given by:

$$\boldsymbol{x}_0 = \begin{pmatrix} 1.0 \cdot 10^9 \text{ for } i = 1 \\ 5.0 \cdot 10^9 \text{ for } i \in \{2, 3\} \\ 3.8 \cdot 10^{12} \text{ for } i = 4 \\ 3.5 \cdot 10^{13} \text{ for } i = 5 \\ 1.0 \cdot 10^7 \text{ for } i \in \{6, 7, \ldots, 13\} \\ 5.0 \cdot 10^{11} \text{ for } i = 14 \\ 1.0 \cdot 10^2 \text{ for } i \in \{15, 16, \ldots, 37\} \\ 1.0 \cdot 10^{-3} \text{ for } i = 38 \\ 1.0 \cdot 10^2 \text{ for } i \in \{39, 40, \ldots, 66\} \end{pmatrix} \tag{III.125}$$

The 10 first solution components of this problem, can be seen in Figure III.22–III.23. Notice the different y-scaling.

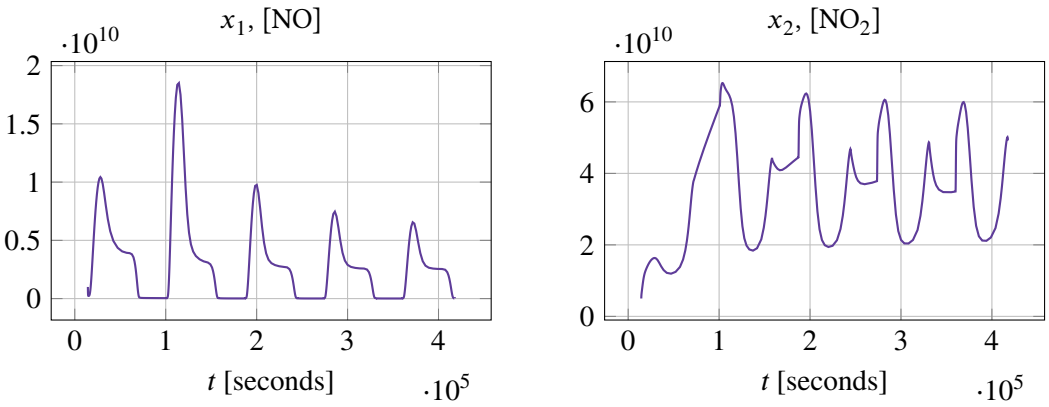

Figure III.22. The two first solution components of *EMEP* as a function of time. Notice the different y-scaling.

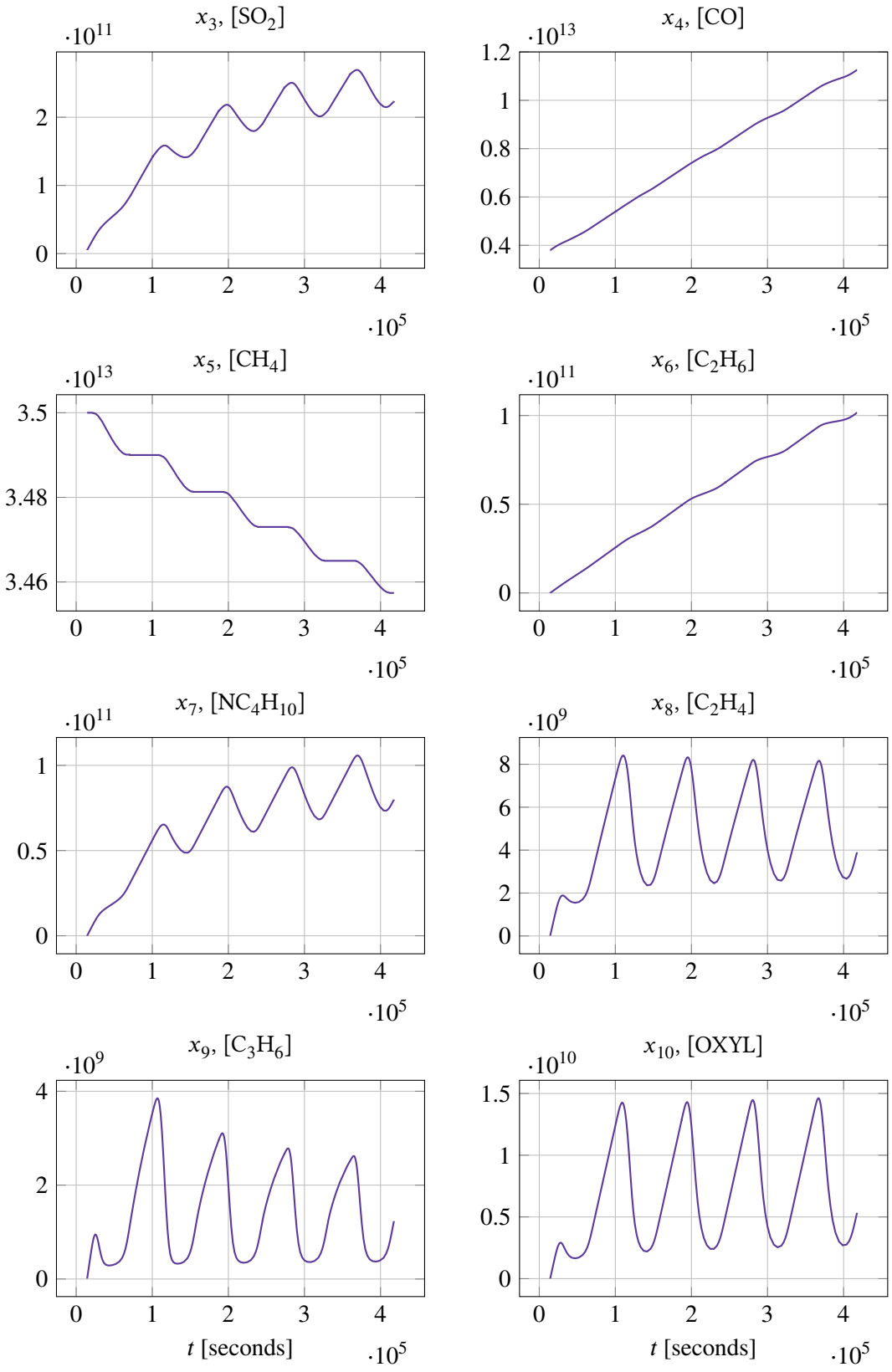

Figure III.23. Eight solution components of *EMEP* as a function of time. Notice the different y-scaling.

Table III.5. Compound-variable correspondence in *the EMEP problem.*

The compounds corresponding to the variables x_i					
x_1	[NO]	x_{23}	$[C_2H_5O_2]$	x_{45}	$[MNKO_2]$
x_2	$[NO_2]$	x_{24}	$[CH_3COO]$	x_{46}	$[CH_3OH]$
x_3	$[SO_2]$	x_{25}	[PAN]	x_{47}	$[RCO_3H]$
x_4	[CO]	x_{26}	$[SECC_4H]$	x_{48}	$[OXYO_2H]$
x_5	$[CH_4]$	x_{27}	$[MEKO_2]$	x_{49}	$[BURO_2H]$
x_6	$[C_2H_6]$	x_{28}	$[R_2OOH]$	x_{50}	$[ETRO_2H]$
x_7	$[NC_4H_{10}]$	x_{29}	$[ETRO_2]$	x_{51}	$[PRRO_2H]$
x_8	$[C_2H_4]$	x_{30}	[MGLYOX]	x_{52}	$[MEKO_2H]$
x_9	$[C_3H_6]$	x_{31}	$[PRRO_2]$	x_{53}	$[MALO_2H]$
x_{10}	[OXYL]	x_{32}	[GLYOX]	x_{54}	[MACR]
x_{11}	[HCHO]	x_{33}	$[OXYO_2]$	x_{55}	[ISNI]
x_{12}	$[CH_3CHO]$	x_{34}	[MAL]	x_{56}	$[ISRO_2H]$
x_{13}	[MEK]	x_{35}	$[MALO_2]$	x_{57}	$[MARO_2]$
x_{14}	$[O_3]$	x_{36}	[OP]	x_{58}	[MAPAN]
x_{15}	$[HO_2]$	x_{37}	[OH]	x_{59}	$[CH_2CCH_3]$
x_{16}	$[HNO_3]$	x_{38}	[OD]	x_{60}	$[ISONO_3]$
x_{17}	$[H_2O_2]$	x_{39}	$[NO_3]$	x_{61}	[ISNIR]
x_{18}	$[H_2]$	x_{40}	$[N_2O_5]$	x_{62}	$[MVKO_2H]$
x_{19}	$[CH_3O_2]$	x_{41}	[ISOPRE]	x_{63}	$[CH_2CHR]$
x_{20}	$[C_2H_5OH]$	x_{42}	[NITRAT]	x_{64}	$[ISNO_3H]$
x_{21}	[SA]	x_{43}	$[ISRO_2]$	x_{65}	[ISNIRH]
x_{22}	$[CH_3O_2H]$	x_{44}	[MVK]	x_{66}	$[MARO_2H]$

III.3.6. Problem 6: The Pleiades problem

This IVP has its origin in celestial mechanics. It describes the planar movement of seven stars in space, where star i has coordinates (x_i, y_i) and mass m_i.

The mathematical formulation of the problem is the following:

$$\boldsymbol{z}'' = \boldsymbol{f}(\boldsymbol{z}), \quad \boldsymbol{z}(0) = \boldsymbol{z}_0, \quad \boldsymbol{z}'(0) = \boldsymbol{z}_0',$$

where

$$\boldsymbol{z} \in \mathbb{R}^{14}, \quad 0 \leq t \leq 3\,\mathrm{s}.$$

Let z be given by:

$$\boldsymbol{z} = \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix}, \quad \boldsymbol{x},\ \boldsymbol{y} \in \mathbb{R}^7.$$

The RHS-function $\boldsymbol{f}$ is then given by

$$f_i^{(1)} = \sum_{j \neq i} m_j (x_j - x_i)/r_{ij}^{3/2}, \quad i = 1, \ldots, 7, \tag{III.126}$$

$$f_i^{(2)} = \sum_{j \neq i} m_j (y_j - y_i)/r_{ij}^{3/2}, \quad i = 1, \ldots, 7, \tag{III.127}$$

where $m_i = i$ and

$$r_{ij} = (x_i - x_j)^2 + (y_i - y_j)^2. \tag{III.128}$$

We rewrite this problem as an order 1 IVP in the following way:

$$\boldsymbol{w} = \begin{pmatrix} \boldsymbol{z} \\ \boldsymbol{z}' \end{pmatrix}' = \begin{pmatrix} \boldsymbol{z}' \\ \boldsymbol{f}(\boldsymbol{z}) \end{pmatrix}.$$

The initial condition is given by:

$$\boldsymbol{w}_0 = \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{y}_0 \\ \boldsymbol{x}_0' \\ \boldsymbol{y}_0' \end{pmatrix} \quad \text{where} \quad \begin{cases} \boldsymbol{x}_0 = (3, 3, -1, -3, 2, -2, 2) \\ \boldsymbol{y}_0 = (3, -3, 2, 0, 0, -4, 4) \\ \boldsymbol{x}_0' = (0, 0, 0, 0, 0, 1.75, -1.5) \\ \boldsymbol{y}_0' = (0, 0, 0, -1.25, 1, 0, 0) \end{cases} \tag{III.129}$$

The 14 first solution components to this problem can be seen in Figure III.24 – III.25. Notice the different y-scaling.

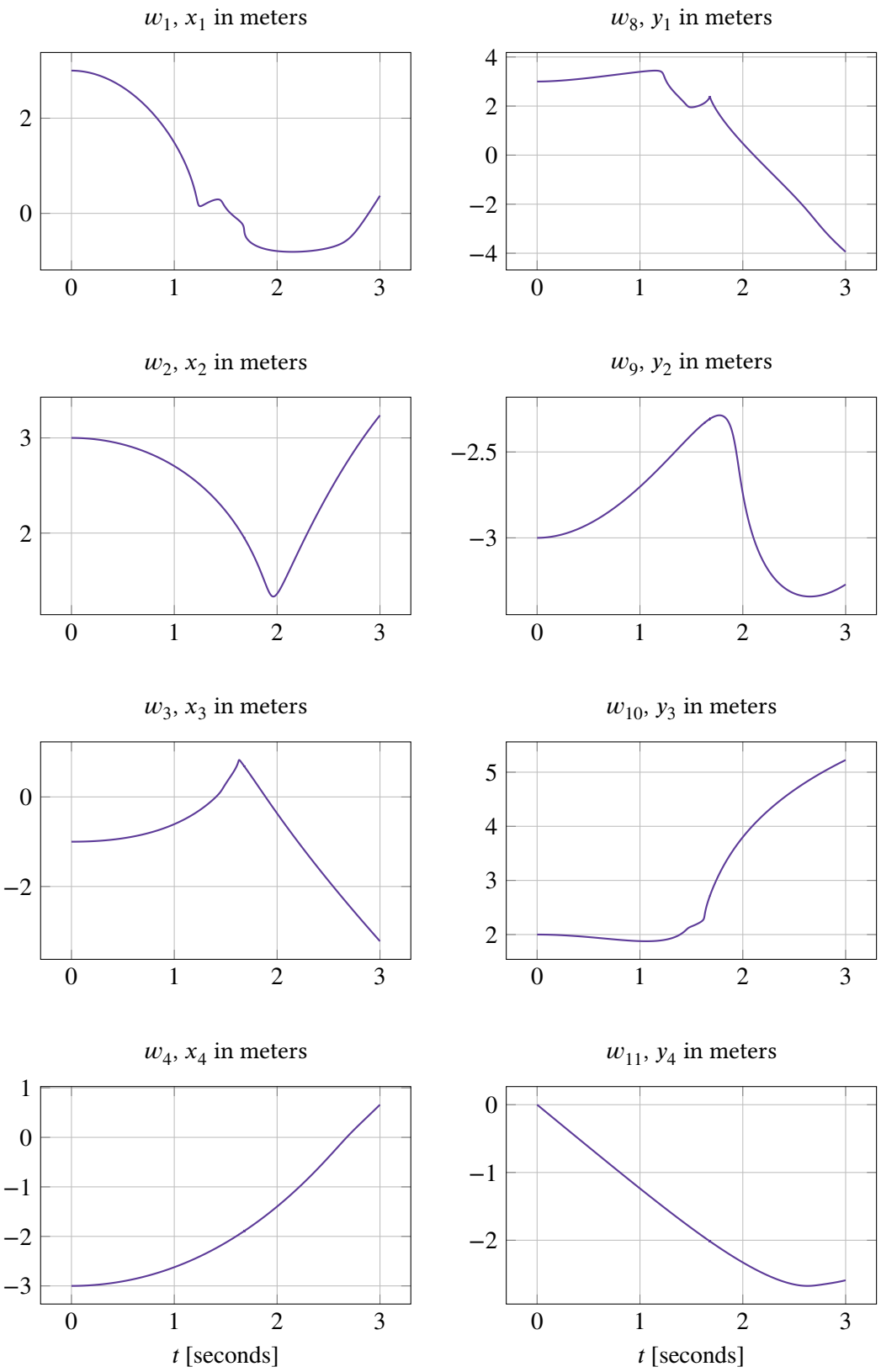

Figure III.24. Eight solution components of *Pleiades* as a function of time. The lines represent the coordinates of star 1,2,3 and 4 as a function of time. Notice the different y-scaling.

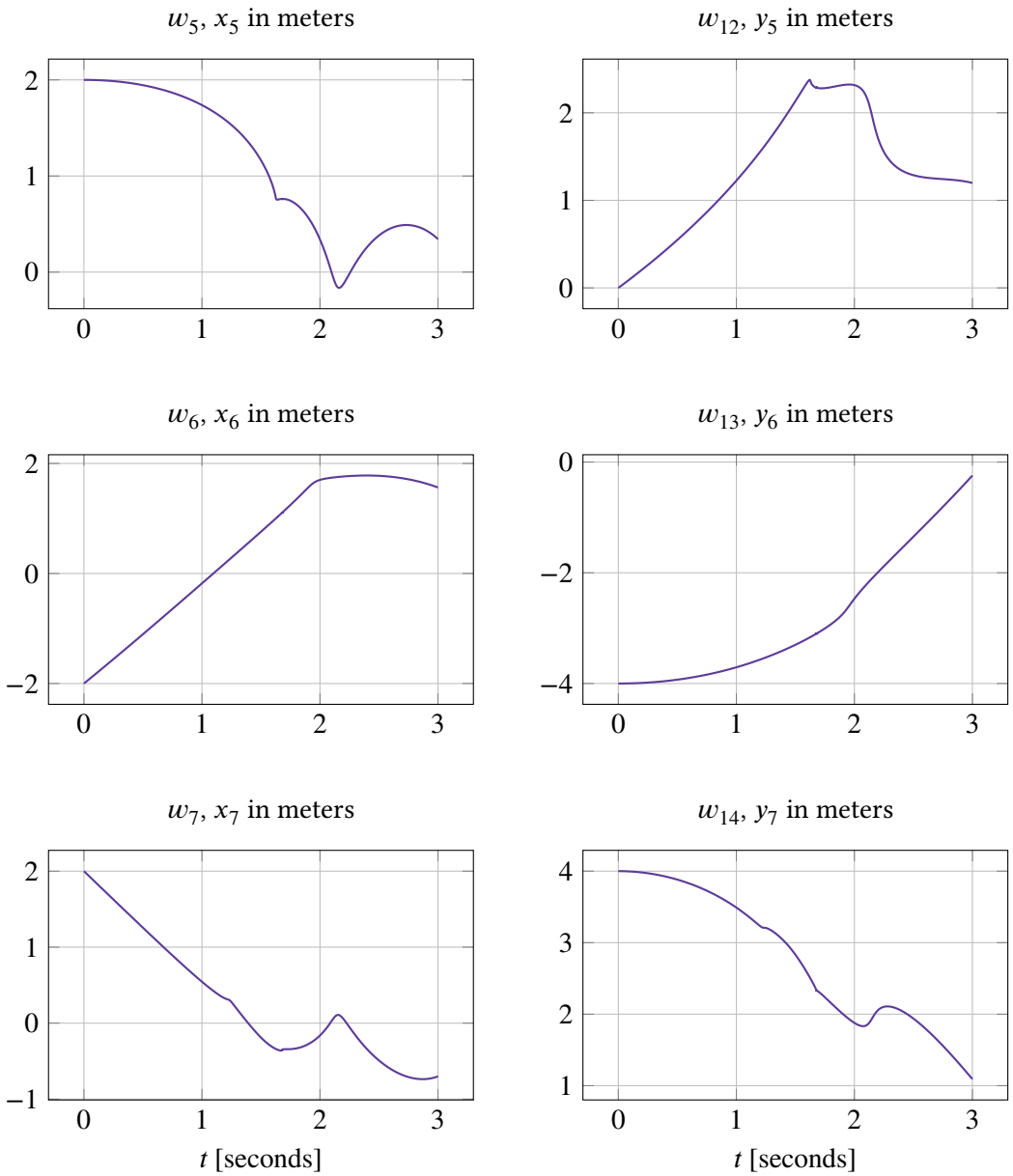

Figure III.25. Six solution components of *Pleiades* as a function of time. The lines represent the coordinates of star 5,6 and 7 as a function of time. Notice the different y-scaling.

III.3.7. Problem 7: The beam problem

This is a mechanical problem that models a thin beam of constant length 1, with one end clamped at ground and the other end free. At the free end a force $\boldsymbol{F} = (F_{\mathrm{u}}, F_{\mathrm{v}})$ is applied causing the beam to oscillate around its equilibrium. The model uses the coordinate $\boldsymbol{z}(t, s)$ defined as the angle (unit: degrees) to the equilibrium at time t and arc length s. The model is discretized in N equidistant points along the beam, creating the discretized variable

$$\boldsymbol{z}\left(t, \frac{k-0.5}{N}\right), \quad k = 1, \ldots, N.$$

The mathematical formulation of the problem is the following:

$$\boldsymbol{z}'' = \boldsymbol{f}(t, \boldsymbol{z}, \boldsymbol{z}'), \quad \boldsymbol{z}(0) = \boldsymbol{z}_0, \quad \boldsymbol{z}'(0) = \boldsymbol{z}'_0,$$

where

$$\boldsymbol{z} \in \mathbb{R}^N, \quad 0 \leq t \leq 5\,\mathrm{s}.$$

The function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(t, \boldsymbol{z}, \boldsymbol{z}') = \boldsymbol{C}\boldsymbol{v} + \boldsymbol{D}\boldsymbol{u}, \tag{III.130}$$

where $\boldsymbol{C}$ is a tridiagonal matrix of size $N \times N$ with the following values of the non-zero elements c_{ij} (row i and column j):

$$c_{11} = 1, \tag{III.131}$$
$$c_{NN} = 3, \tag{III.132}$$
$$c_{ii} = 2, \qquad l = 2, \ldots, N-1, \tag{III.133}$$
$$c_{i,i+1} = -\cos(z_i - z_{i+1}), \qquad l = 1, \ldots, N-1, \tag{III.134}$$
$$c_{i,i-1} = -\cos(z_i - z_{i-1}), \qquad l = 2, \ldots, N, \tag{III.135}$$

$\boldsymbol{D}$ is a bidiagonal matrix of size $N \times N$ with the following values of the non-zero elements d_{ij}:

$$d_{i,i+1} = -\sin(z_i - z_{i+1}), \qquad l = 1, \ldots, N-1, \tag{III.136}$$
$$d_{i,i-1} = -\sin(z_i - z_{i-1}), \qquad l = 2, \ldots, N, \tag{III.137}$$

$\boldsymbol{v}$ is a vector of size N with the elements:

$$v_i = N^4(z_{i-l} - 2z_i + z_{i+1}) + N^2(\cos(z_i)F_{\mathrm{y}} - \sin(z_i)F_{\mathrm{x}}), \quad l = 1, \ldots, N, \tag{III.138}$$

where

$$z_0 = -z_1, \tag{III.139}$$
$$z_{N+1} = z_N, \tag{III.140}$$

and $\boldsymbol{u}$ is a vector of size N, and the solution to the system:

$$\boldsymbol{C}\boldsymbol{u} = \boldsymbol{g}, \quad \text{where} \quad \boldsymbol{g} = \boldsymbol{D}\boldsymbol{v} + (z_1'^2, z_2'^2, \ldots, z_N'^2)^T. \tag{III.141}$$

We reformulate the system such that we get an ODE on first order form in the following way:

$$\boldsymbol{w} = \begin{pmatrix} \boldsymbol{z} \\ \boldsymbol{z}' \end{pmatrix}' = \begin{pmatrix} \boldsymbol{z}' \\ \boldsymbol{f}(t, \boldsymbol{z}, \boldsymbol{z}') \end{pmatrix}.$$

The initial condition is given by:

$$\boldsymbol{w}_0 = \begin{pmatrix} \boldsymbol{z}_0 \\ \boldsymbol{z}_0' \end{pmatrix}, \quad \text{where} \quad \begin{cases} \boldsymbol{z}_0 = (0, \dots, 0) \\ \boldsymbol{z}_0' = (0, \dots, 0) \end{cases}. \tag{III.142}$$

The solution of *Beam*, $N = 10$ can be seen in Figure III.26.

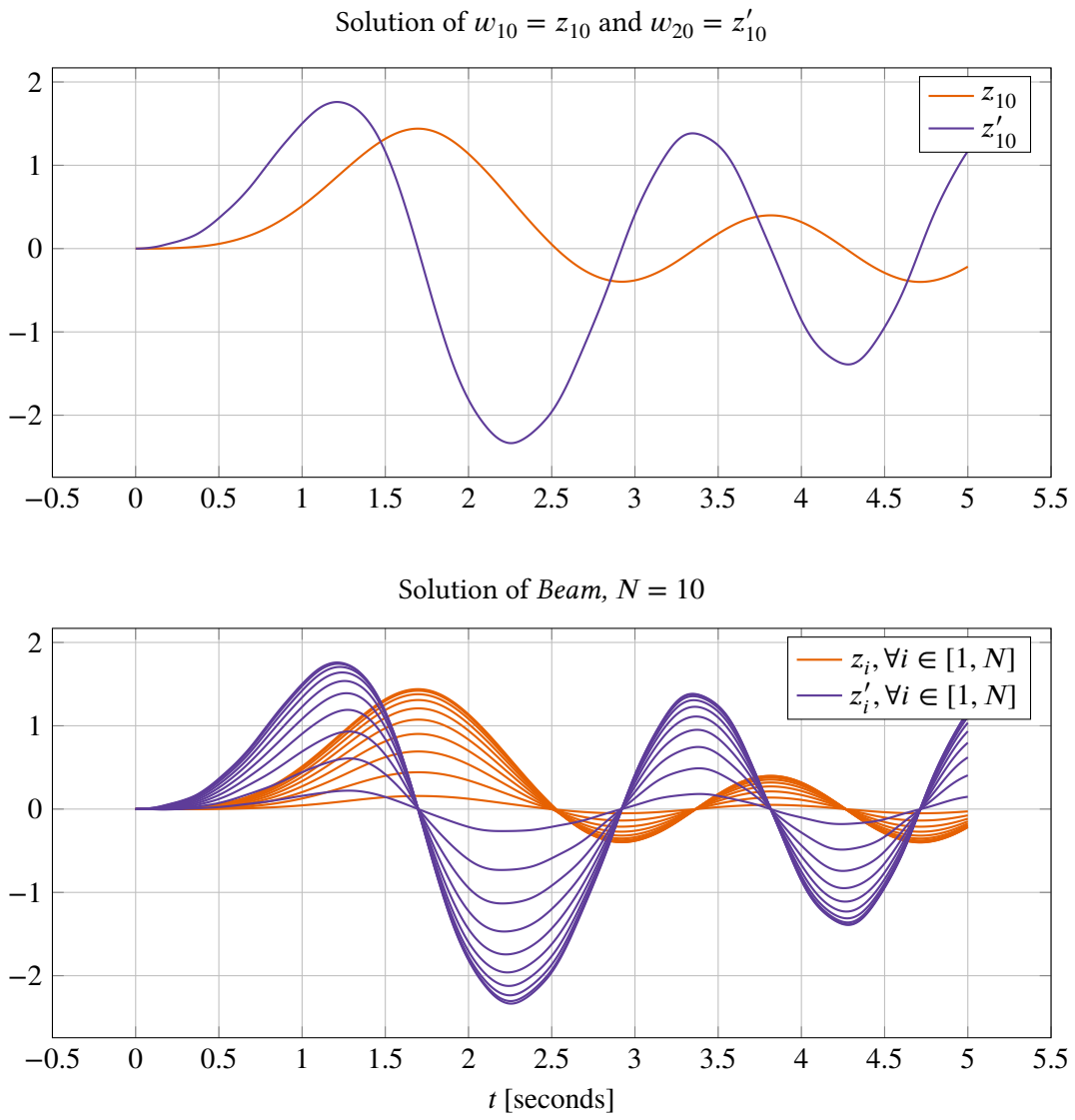

Figure III.26. Solutions to *Beam*, $N = 10$. (Top) solution of $w_{10} = z_{10}$ and $w_{20} = z'_{10}$. (Bottom) Solution of all 20 components.

III.3.8. Problem 8: The Van der Pol problem

The Van der Pol problem originates from electronics and models the scaled current $z(t) = kI(t)$ in vacuum tube circuits with non-linear damping. This is a periodic problem.

The mathematical formulation of the problem is the following:

$$z'' = f(z, z'), \quad z(0) = z_0, \quad z'(0) = z_0',$$

where

$$z \in \mathbb{R}, \quad 0 \leq t \leq 4\mu \text{ s}.$$

The period time T is approximately 2μ. The function f is given by:

$$f(z, z') = \mu(1 - z^2)z' - z, \quad \mu > 0. \tag{III.143}$$

The problem is transformed onto first order form, by rewriting it in the following way:

$$\boldsymbol{w} = \begin{pmatrix} z \\ z' \end{pmatrix}' = \begin{pmatrix} z' \\ f(z, z') \end{pmatrix}.$$

The initial values are:

$$\boldsymbol{w}_0 = \begin{pmatrix} z_0 \\ z_0' \end{pmatrix} \quad \text{where} \quad \begin{cases} z_0 = 2 \\ z_0' = 0 \end{cases}. \tag{III.144}$$

By varying μ, we can vary the stiffness of the system (the greater the μ, the stiffer the system).

The solution to *Van der Pol* with $\mu = 5, 10$ and 100 can be seen in Figure III.28. In Figure III.27 we see the phase plot of *Van der Pol* with $\mu = 10, 20, \ldots, 70$.

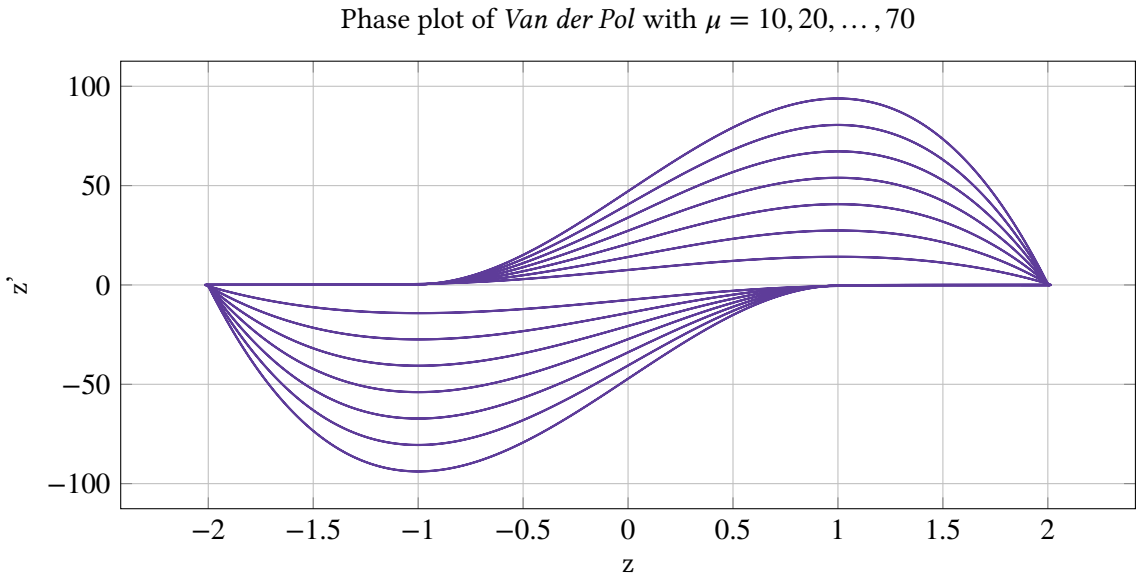

Figure III.27. The phase plot of *Van der Pol* with $\mu = 10, 20, \ldots, 70$. The larger the μ, the bigger the lobes of the phase plot.

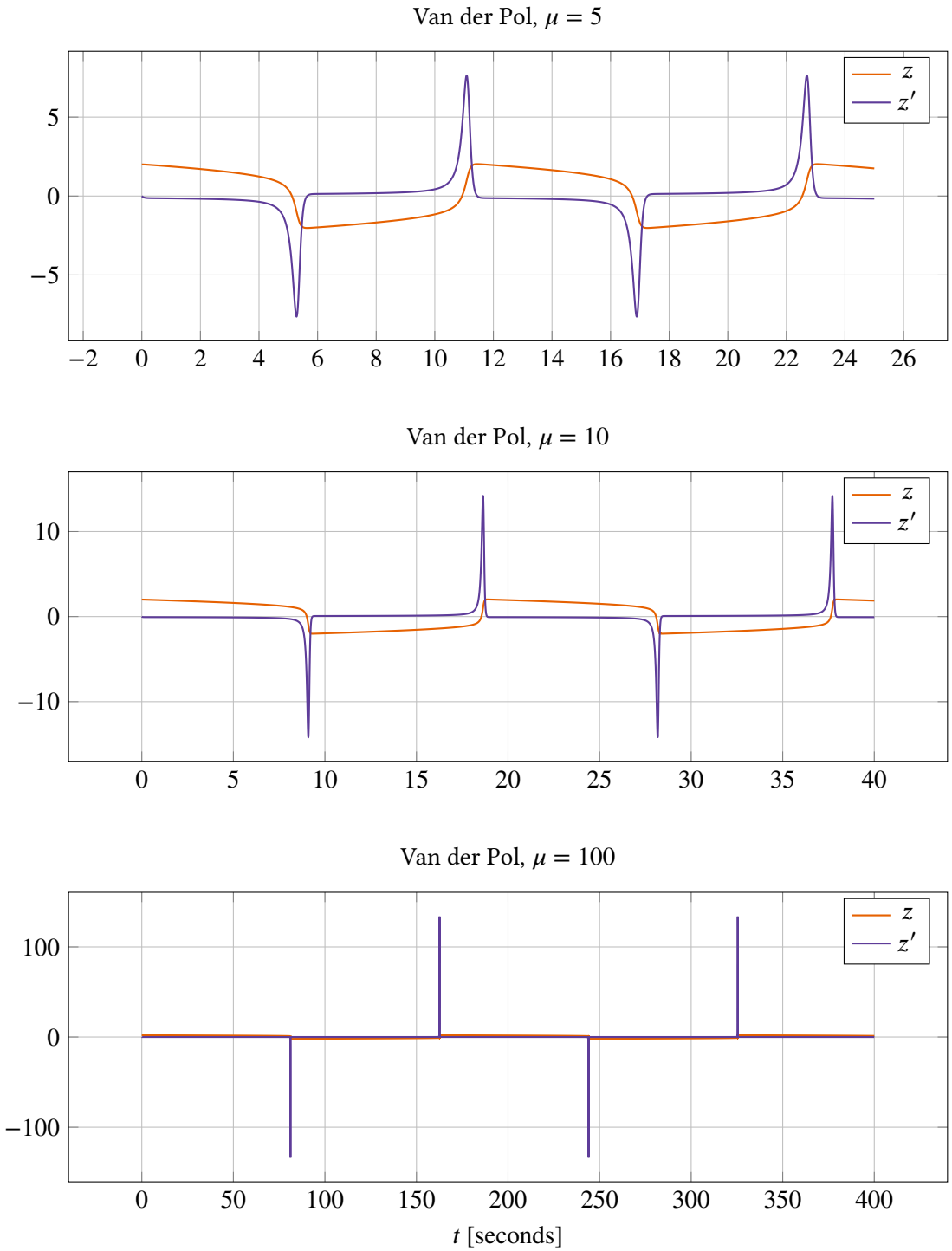

Figure III.28. The solution to *Van der Pol* with $\mu = 5, 10$ and 100 as a function of time. The larger the μ, the stiffer the equation and the longer the period T.

III.3.9. Problem 9: The Oregonator problem

This problem originates from chemistry. It models the famous oscillatory Belousov–Zhabotinskii (BZ) reaction in the simplest way possible. The model consists of 6 different substances, where the effective concentration of three of them is measured.

The mathematical formulation of the problem is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^3, \quad 0 \leq t \leq 360\,\mathrm{s}.$$

The three components x_1, x_2 and x_3 model the effective concentrations (unit: mol) of hypobromous acid, bromide and cerium-4 over time (unit: seconds). The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} s\left(x_2 - x_1 x_2 + x_1 - q x_1^2\right) \\ \left(-x_2 - x_1 x_2 + x_3\right)/s \\ w\left(x_1 - x_3\right) \end{pmatrix}, \tag{III.145}$$

where the value of the constants are

$$s = 77.27, \quad w = 0.161, \quad q = 8.375 \cdot 10^{-6}. \tag{III.146}$$

The initial values are given by:

$$\boldsymbol{x}_0 = \left(1, 2, 3\right). \tag{III.147}$$

The solution to *Oregonator* can be seen in Figure III.29.

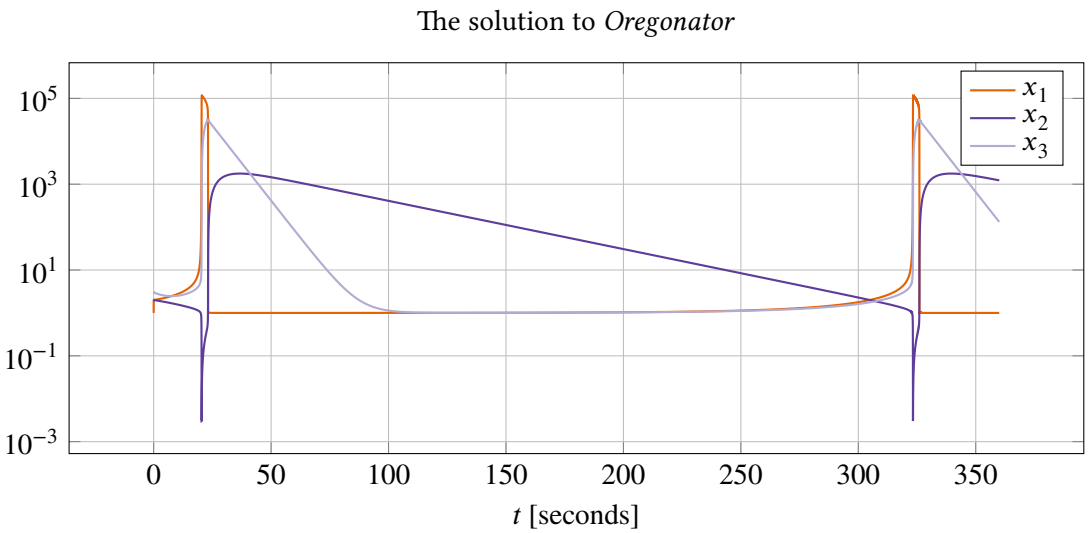

Figure III.29. The solution to *Oregonator* as a function of time.

III.3.10. Problem 10: The Robertson problem

Robertson is a chemical problem that models a special auto-catalytic reaction given by H.H. Robertson. The reaction consists of three different chemical substances A, B, and C.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^3, \quad 0 \leq t \leq 10^{11} \text{ s}.$$

The three components x_1, x_2 and x_3 model the concentrations (unit: mol) of substance A, B and C over time (unit: seconds). The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} -0.04x_1 + 10^4 x_2 x_3 \\ 0.04x_1 - 10^4 x_2 x_3 - 3 \cdot 10^7 x_2^2 \\ 3 \cdot 10^7 x_2^2 \end{pmatrix}. \tag{III.148}$$

The initial condition is given by:

$$\boldsymbol{x}_0 = \left(1, 0, 0\right). \tag{III.149}$$

The solution to *Robertson* can be seen in Figure III.30.

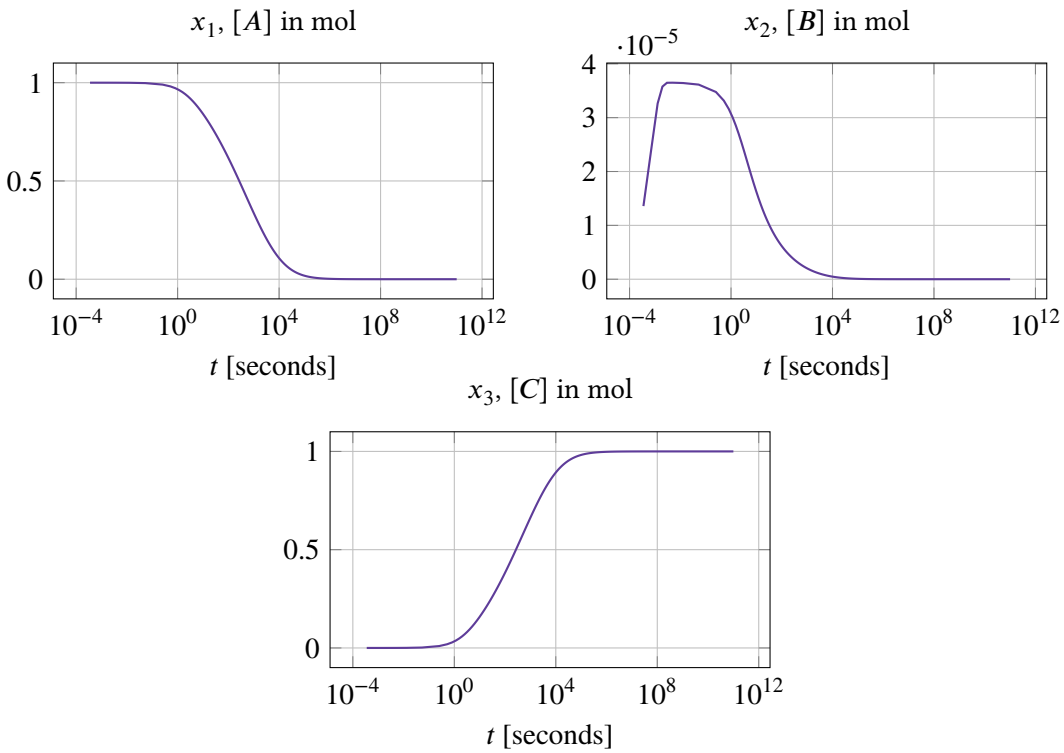

Figure III.30. The 3 solution components of *Robertson* as a function of time.

III.3.11. Problem 11: The E5 problem

E5 originates from chemistry. It models a chemical pyrolysis consisting of 6 reactants $R_1, \ldots, R_6$ where the concentration of four of them $R_1, \ldots, R_4$ is measured.

The mathematical formulation of the problem is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^4, \quad 0 \leq t \leq 10^{13}\,\text{s}.$$

The 4 components x_1, x_2, x_3 and x_4 model the concentration (unit: mol) of the compounds R_1, R_2, R_3 and R_4 over time (unit: seconds). The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} -Ax_1 - Bx_1x_3 \\ Ax_1 - MCx_2x_3 \\ Ax_1 - Bx_1x_3 - MCx_2x_3 + Cx_4 \\ Bx_1x_3 - Cx_4 \end{pmatrix}, \tag{III.150}$$

with the constants:

$$A = 7.89 \cdot 10^{-10}, \quad B = 1.1 \cdot 10^{7}, \quad C = 1.13 \cdot 10^{3}, \quad M = 10^{6}. \tag{III.151}$$

The initial condition is given by:

$$\boldsymbol{x}_0 = \left(1.76 \cdot 10^{-3}, 0, 0, 0\right). \tag{III.152}$$

The solution to *E5* can be seen in Figure III.31.

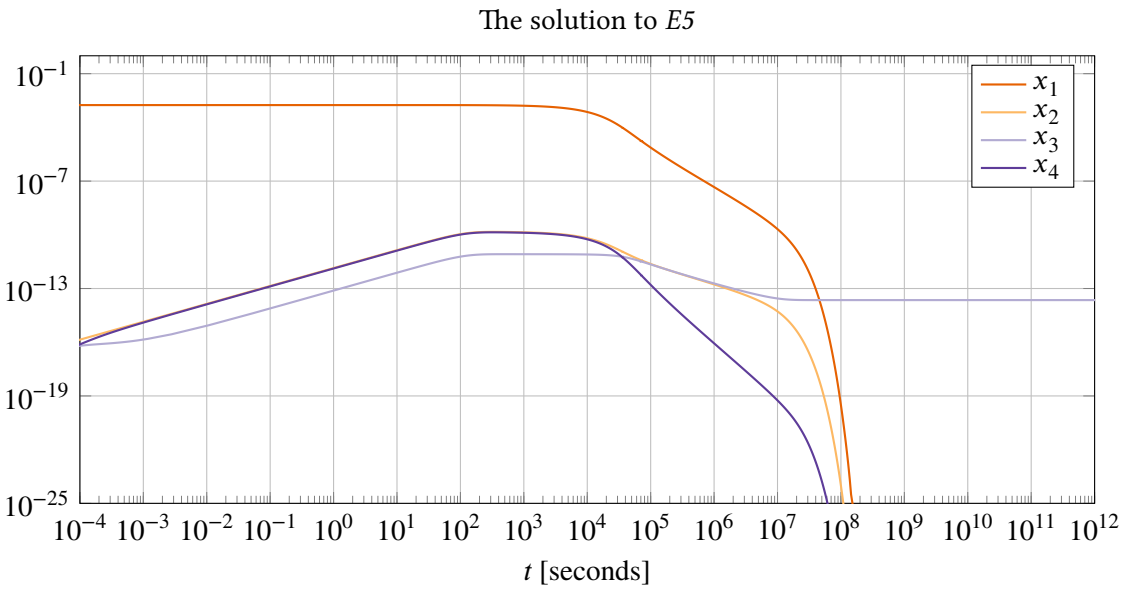

Figure III.31. The solution to *E5* as a function of time.

III.3.12. Problem 12: The Lotka–Volterra problem

This problem originates from biology. The model describes the dynamic of the amount of a special kind of predator (e.g., foxes) and a special kind of pray (e.g., rabbits) at a certain time t.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^2, \quad 0 \leq t \leq c\,\mathrm{s}.$$

Component x_1 models the amount of prays and x_2 models the amount of predators. The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} \alpha x_1 - \beta x_1 x_2 \\ \delta x_1 x_2 - \gamma x_2 \end{pmatrix}, \tag{III.153}$$

with the constants:

$$\alpha = 0.1, \quad \beta = 0.3, \quad \gamma = 0.5, \quad \delta = 0.5. \tag{III.154}$$

The initial condition is given by:

$$\boldsymbol{x}_0 = \begin{pmatrix} 1, 1 \end{pmatrix}. \tag{III.155}$$

This problem contains one parameter c, which corresponds to the number of cycles to be integrated. The solution to *Lotka–Volterra, $c = 4$* can be seen in Figure III.32.

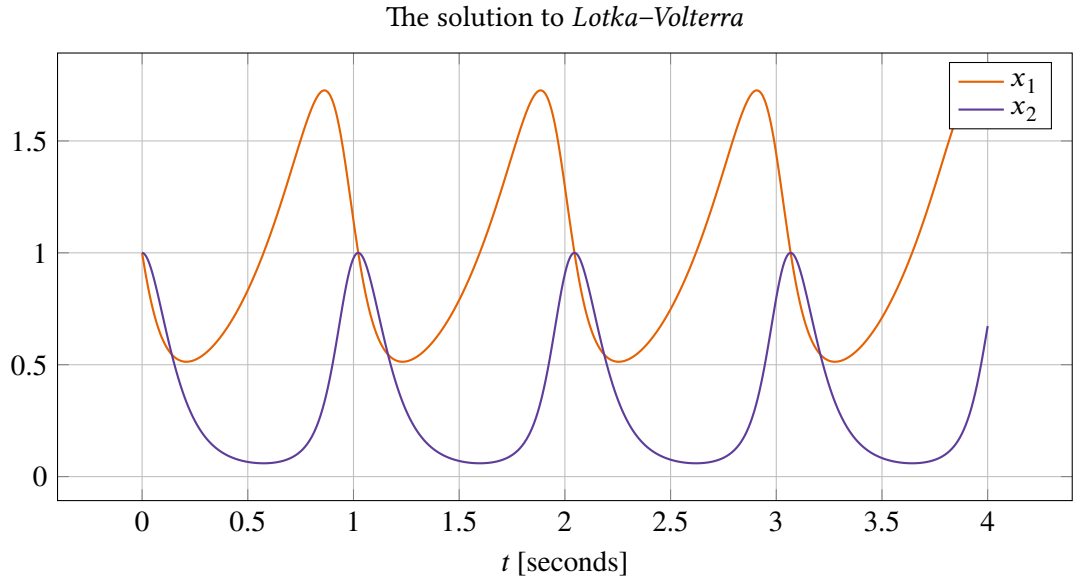

Figure III.32. The solution to *Lotka–Volterra, $c = 4$* as a function of time.

III.3.13. Problem 13: The flame propagation problem

This problem is a simple model of how the radius of a flame changes when a match is lit. In the beginning the radius grows rapidly until the flame reaches a steady state due to oxygen consumption limitations.

The mathematical formulation is the following:

$$\dot{x} = f(x), \quad x(0) = x_0,$$

where

$$x \in \mathbb{R}, \quad 0 \leq t \leq \frac{2}{x_0}.$$

The RHS-function f is given by:

$$f(x) = x^2 - x^3. \tag{III.156}$$

The initial condition is given by:

$$x_0 = 10^{-\alpha}, \quad \alpha > 0. \tag{III.157}$$

By varying α, we can vary the stiffness of the equation (the greater the α, the stiffer the equation).

The solution to *Flame propagation* for four different α's can be seen in Figure III.33.

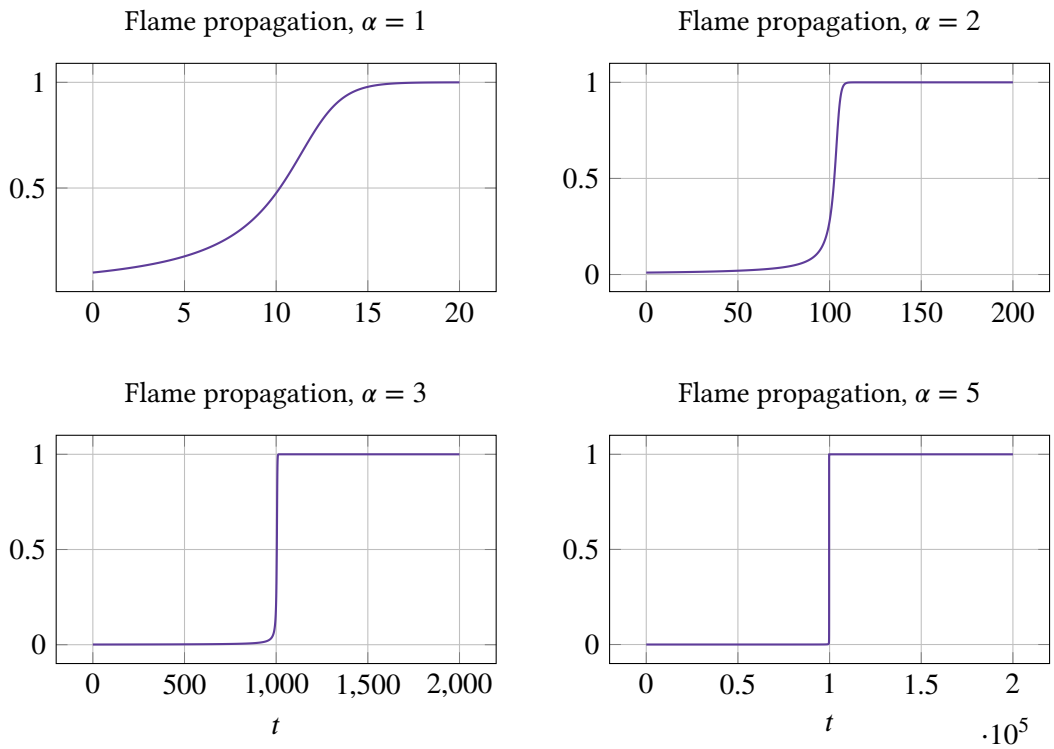

Figure III.33. The solution to *Flame propagation* with $\alpha = 1, 2, 3$ and 5 as a function of time.

III.3.14. Problem 14: The decaying exponential problem

Decaying exponential is commonly known as *the test equation*. It is an easy problem, used to make basic tests. In physics it is used to describe radioactive decay.

The mathematical formulation is the following:

$$\dot{x} = f(x), \quad x(0) = x_0,$$

where

$$x \in \mathbb{R}, \quad 0 \leq t \leq 10.$$

The RHS-function f is given by:

$$f(x) = -dx. \tag{III.158}$$

By varying d, we can vary the stiffness of the equation (the greater the d, the stiffer the equation). The initial condition is given by:

$$x_0 = 1. \tag{III.159}$$

The solution to *Decaying exponential* for two different d's can be seen in Figure III.34.

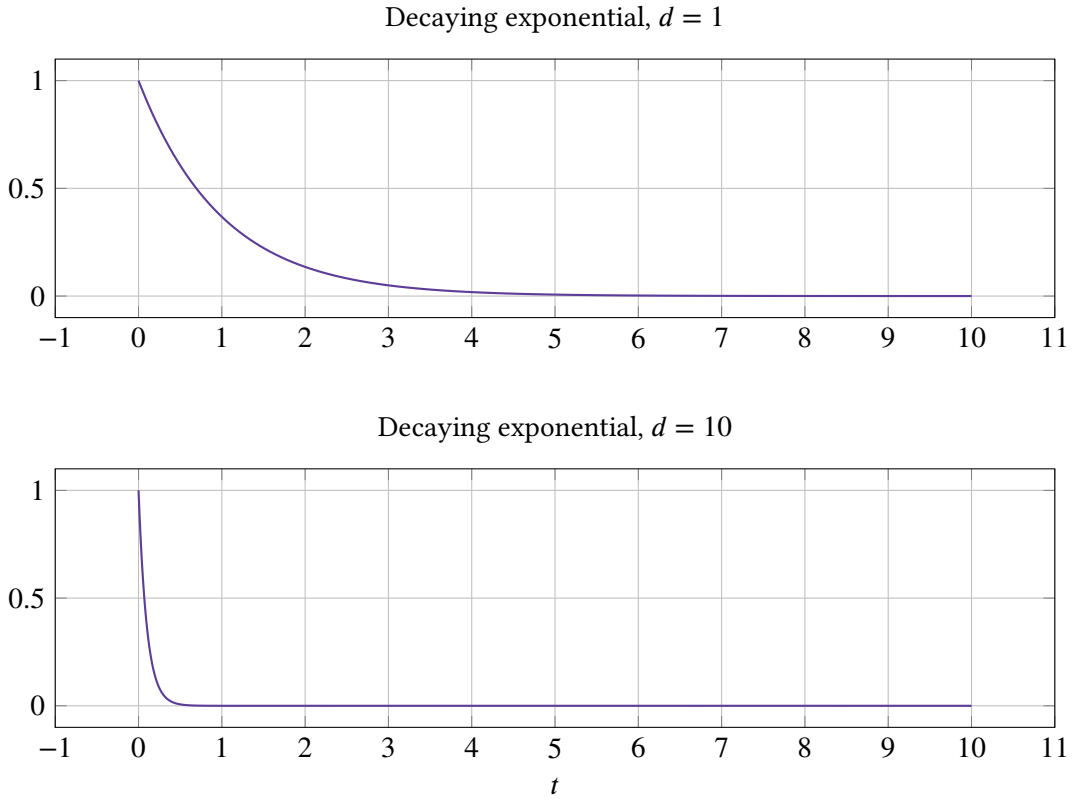

Figure III.34. The solution to *Decaying exponential* for $d = 1$ and $d = 10$ as a function of time.

III.3.15. Problem 15: The two exponentials problem

Two exponentials (TE) is the two dimensional version of *decaying exponential*, which makes it a little harder, though is still regarded as an easy problem to solve.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^2, \quad 0 \leq t \leq 10.$$

The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} \frac{\lambda_1 + \lambda_2}{2} x_1 + \frac{-\lambda_1 + \lambda_2}{2} x_2 \\ \frac{-\lambda_1 + \lambda_2}{2} x_1 + \frac{\lambda_1 + \lambda_2}{2} x_2 \end{pmatrix}, \tag{III.160}$$

where λ_1 and λ_2 are the two corresponding eigenvalues. Belonging to this problem is the initial condition:

$$\boldsymbol{x}_0 = (0, 2)^T. \tag{III.161}$$

The solution to *TE* for four different combinations of λ_1 and λ_2 can be seen in Figure III.35.

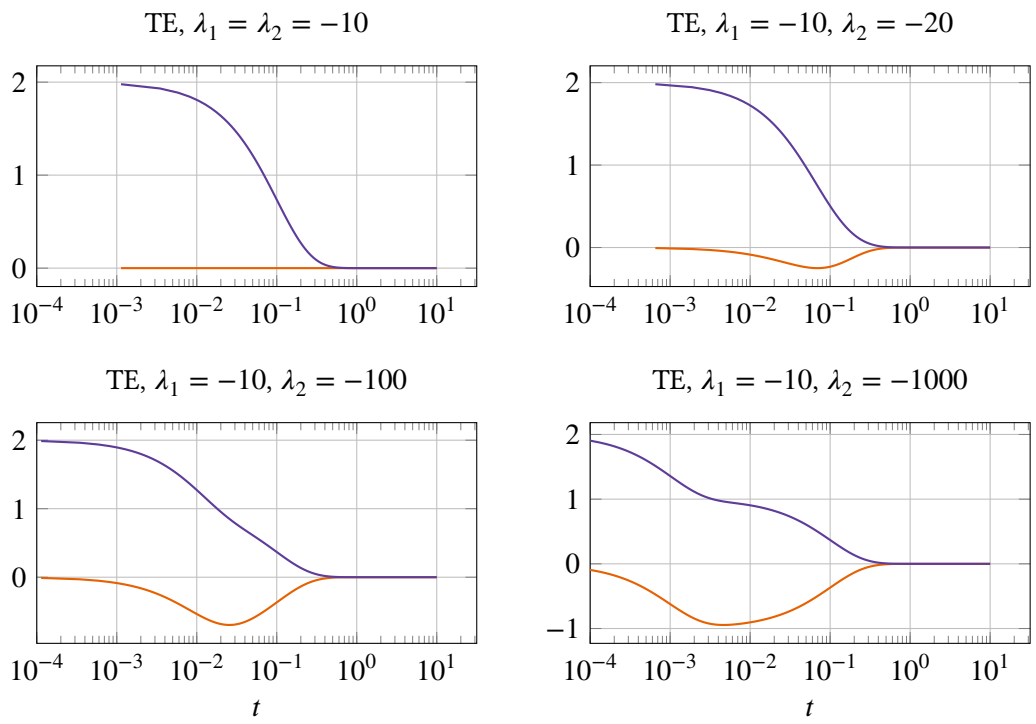

Figure III.35. The solution to *Two Exponentials* for four different combinations of λ_1 and λ_2 as a function of time.

III.3.16. Problem 16: The Lorenz problem

The Lorenz problem was developed in 1963 by Edward Lorenz, and is a simplified model of atmospheric convection. The solution is known to be chaotic.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^3, \quad 0 \leq t \leq 7\,\text{s}.$$

The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} -\sigma(x_1 - x_2) \\ x_1(\lambda - x_3) - x_2 \\ x_1 x_2 - b x_3 \end{pmatrix}, \tag{III.162}$$

where $b = 8/3$, $\lambda = 28$ and $\sigma = 10$. The initial condition is:

$$\boldsymbol{x}_0 = (-8, 8, 27)^T. \tag{III.163}$$

The solution to *Lorenz* can be seen in Figure III.36.

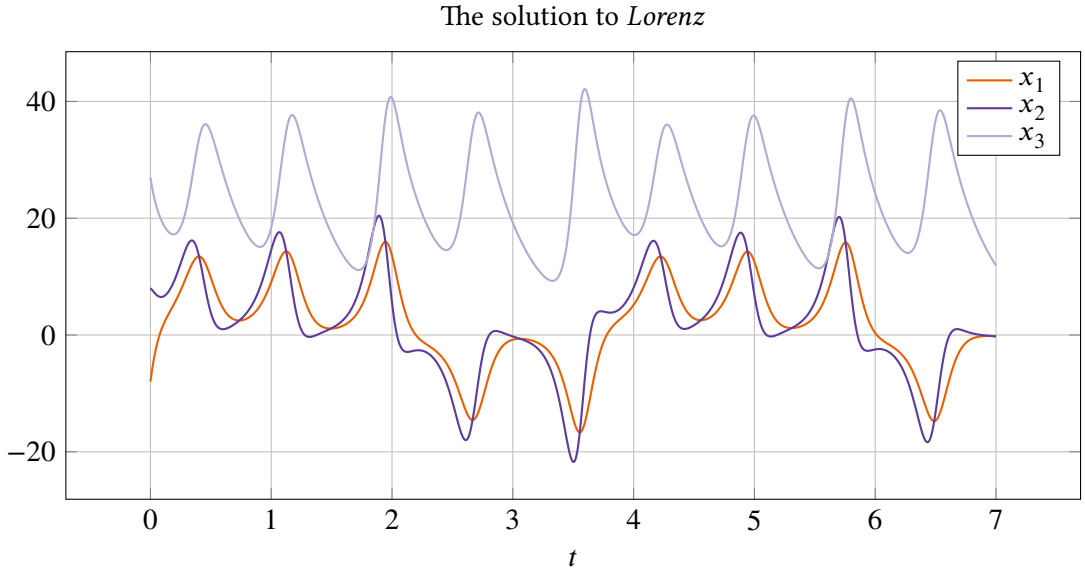

Figure III.36. The solution to *Lorenz* as a function of time.

III.3.17. Problem 17: The Brusselator problem

This problem originates from chemistry. More precisely, it models a type of autocatalytic reaction, for example, *the clock reaction*. The system was developed by Ilya Prigogine and his team at the Université Libre de Bruxelles.

The mathematical formulation is the following:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0,$$

where

$$\boldsymbol{x} \in \mathbb{R}^2, \quad 0 \leq t \leq 20.$$

The RHS-function $\boldsymbol{f}$ is given by:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix} A + x_1^2 x_2 - (B+1)x_1 \\ Bx_1 - x_1^2 x_2 \end{pmatrix}, \tag{III.164}$$

where $A = 2$ and $B = 8.533$. The initial condition is given by:

$$\boldsymbol{x}_0 = (1, 4.2665)^T. \tag{III.165}$$

The solution to *Brusselator* can be seen in Figure III.37.

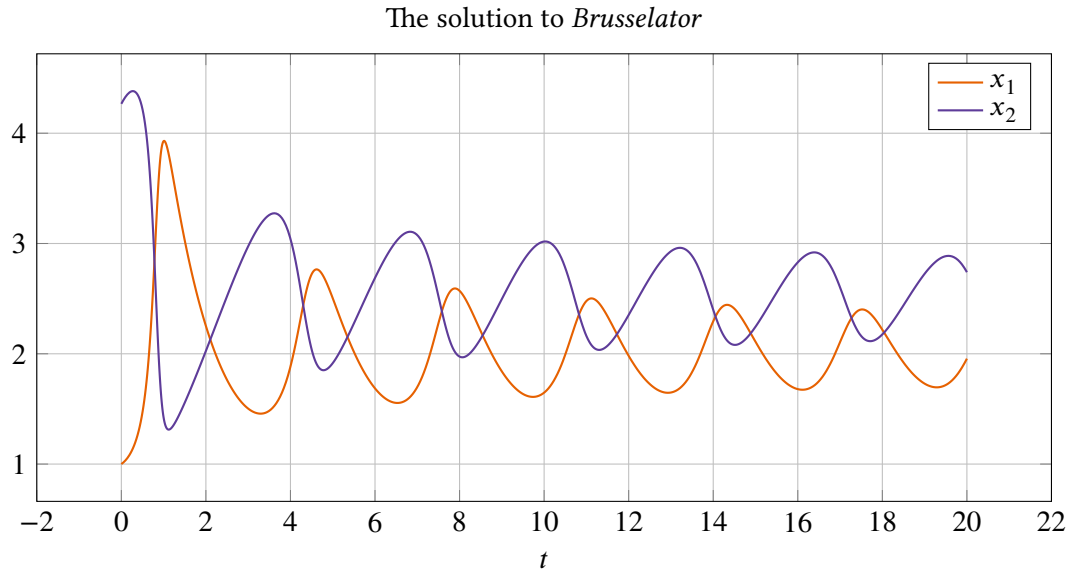

Figure III.37. The solution to *Brusselator* as a function of time.

III.3.18. Overview of the test problems

The test problems are categorized into three categories:

Cat. 1: stiff – Test problems that are always stiff. We solve these with a stiff solver.

Cat. 2: both – These problems have one parameter. Depending on the value of the parameter, the problem can go from being stiff to being non-stiff.

Cat. 3: non-stiff – Test problems that are always non-stiff. We solve these with a non-stiff solver.

To get an overview of all the test problems and their most important qualities see Table III.6, Table III.7 and Table III.8. Listed in the tables are the following things:

Nbr – The number of the test problem corresponding to this article.

Name – The name of the test problem corresponding to this article.

n – The size of the equation system corresponding to the problem.

$s_{\min}$ – The minimum value of the stiffness indicator $s(t)$ in the interval $[t_0, t_f]$.

$\hat{s}_{\min}$ – The minimum value of the normalized stiffness indicator $\hat{s}(t)$ in the interval $[t_0, t_f]$.

δt – The integration time.

Free par. – The free parameter belonging to the problem. This is only listed in Table III.7.

Table III.6. Summary of all test problems belonging to the category *stiff*. The following can be found here: Problem number, problem name, system size, stiffness indicator, normalized stiffness indicator and integration time.

Nbr	*Name*	*Free par.*	n	$s_{\min}$	$\hat{s}_{\min}$	δt
1	HIRES	—	8	$-1.1 \cdot 10^2$	$-3.5 \cdot 10^4$	322
2	Pollution	—	20	$-2.2 \cdot 10^{11}$	$-1.32 \cdot 10^{13}$	60
3	Ring modulator	—	15	$-5.4 \cdot 10^{10}$	$-5.4 \cdot 10^7$	10^{-3}
4	Medical Akzo Nobel	$N = 200$	400	$-4.5 \cdot 10^3$	$-9.0 \cdot 10^4$	20
5	EMEP	—	66	$-4.3 \cdot 10^8$	$-1.7 \cdot 10^{14}$	403200
9	Oregonator	—	3	$-6.8 \cdot 10^4$	$-2.4 \cdot 10^7$	360
10	Robertson	—	3	$-5.0 \cdot 10^3$	$-5.0 \cdot 10^{14}$	10^{11}
11	E5	—	4	$-1.0 \cdot 10^4$	$-1.0 \cdot 10^{17}$	10^{13}

Table III.7. Summary of all test problems belonging to the category *both*. The following can be found here: Problem number, problem name, system size, stiffness indicator, normalized stiffness indicator and integration time.

Nbr	*Name*	*Free par.*	n	$s_{\min}$	$\hat{s}_{\min}$	δt
8	Van der Pol	$\mu = 1$	2	-1.5	-6.0	4
		$\mu = 10$	2	$-1.5 \cdot 10^{1}$	$-6.0 \cdot 10^{2}$	40
		$\mu = 100$	2	$-1.5 \cdot 10^{2}$	$-6.0 \cdot 10^{4}$	400
		$\mu = 1000$	2	$-1.5 \cdot 10^{3}$	$-6.0 \cdot 10^{6}$	4000
13	Flame propagation	$\alpha = 1$	1	-1.0	$-2.0 \cdot 10^{1}$	$2 \cdot 10^{1}$
		$\alpha = 2$	1	-1.0	$-2.0 \cdot 10^{2}$	$2 \cdot 10^{2}$
		$\alpha = 3$	1	-1.0	$-2.0 \cdot 10^{3}$	$2 \cdot 10^{3}$
14	Decaying exponential	$p = 1$	1	-1.0	$-1.0 \cdot 10^{1}$	10
		$p = 10$	1	-10.0	$-1.0 \cdot 10^{2}$	10
		$p = 100$	1	-100.0	$-1.0 \cdot 10^{3}$	10
		$p = 1000$	1	-1000.0	$-1.0 \cdot 10^{4}$	10
15	Two exponentials	$\lambda = -1$	2	-1.0	$-1.0 \cdot 10^{1}$	10
		$\lambda = -10$	2	-10.0	$-1.0 \cdot 10^{2}$	10
		$\lambda = -100$	2	-100.0	$-1.0 \cdot 10^{3}$	10
		$\lambda = -1000$	2	-1000.0	$-1.0 \cdot 10^{4}$	10

Table III.8. Summary of all test problems belonging to the category *non-stiff*. The following can be found here: Problem number, problem name, system size, stiffness indicator, normalized stiffness indicator and integration time.

Nbr	*Name*	*Free par.*	n	$s_{\min}$	$\hat{s}_{\min}$	δt
6	Pleiades	—	28	$-7.3 \cdot 10^{-11}$	$-2.2 \cdot 10^{-10}$	3
7	Beam	$N = 40$	80	$-3.1 \cdot 10^{-4}$	$-1.5 \cdot 10^{-3}$	5
12	Lotka–Volterra	$c = 2$	2	-4.7	-9.4	2
16	The Lorenz problem	—	3	-6.3	$-4.4 \cdot 10^{1}$	7
17	Brusselator	—	2	-7.2	$-1.4 \cdot 10^{2}$	20

III.4. Benchmarking – Tests

This section consists of different kinds of tests created to analyze the solvers (those of fixed order) in different ways. It can be used as a guide for how our software may by used to analyze different aspects of the behavior of a method and a filter, and also the combination of them. The first two tests will evaluate their overall functionality and answer questions about numerical stability, accuracy, filter behavior, robustness, and tolerance and work proportionality. We will use Section II.5 as a checklist.

The last test will show what happens in the explicit solver when a problem (decaying exponent) is made stiffer. Interesting to see is how the filters behave when stability becomes an issue, and find out if any of the filters behave better than others.

Table II.1 and Table II.2 show the collection of methods and filters that our libraries supply, and Section III.3 contains all problems implemented in our problem library. Due to space limitation, we only submit solutions of a few combinations.

III.4.1. Test 1: Stability and accuracy

For this test we need three periodic problems; one for each solver. We choose *Lotka–Volterra*, *Van der Pol,* $\mu = 10$ and *Oregonator*, mostly since the solutions of these problems are well-known. In all three cases, we solve the problems for a few periods. The chosen settings are given in Table III.9.

Table III.9. Settings for the solvers `pmme`, `pmmip` and `pmmi` in test 1. For plots of the results see Figure III.38, III.39 and III.40.

	`pmme`	`pmmip`	`pmmi`
problem	Van der Pol, $\mu = 10$	Lotka–Volterra	Oregonator
filtername	H211PI	H211*b*	H312*b*
methodname	AB5	IDC34	BDF5
tol	1e-6	1e-7	1e-7
perc	[0.8, 1.2]	[0.8, 1.2]	[0.8, 1.2]
relerr	false	false	false
unit	false	false	false
usebypass	true	true	true
norm	$\|\| \bullet \|\|_\infty$	$\|\| \bullet \|\|_\infty$	$\|\| \bullet \|\|_\infty$
reference	`ode113`	`ode113`	`ode15s`

In this test we study the six following things (see Figure III.38, III.39 and III.40 for plots of the results):

1. *The solution over time,* $\boldsymbol{x}(t)$.
2. *The stiffness over time,* $s(t)$: Calculated as explained in Section III.2.
3. *The step-size over time,* $h(t)$: Discretized version, h_n, is calculated according to

$$h_n = t_{n+1} - t_n. \tag{III.166}$$

4. *The step-size ratio over time, $r(t)$*: Discretized version, r_n, is calculated according to

$$r_n = h_{n+1}/h_n. \tag{III.167}$$

5. *The absolute error over time, $E_{\text{abs}}(t)$*: This is calculated post-integration and is given by

$$E_{\text{abs}}(t) = \|\boldsymbol{x}(t) - \boldsymbol{x}^{\text{ref}}(t)\|_\infty, \tag{III.168}$$

   where x is the solution given by the solver and $\boldsymbol{x}^{\text{ref}}$ is the *reference solution*, that is, the analytical solution, if this exists, otherwise the solution of the *reference solver* used.

6. *The control error over time, $E_{\text{cont}}(t)$*: This is calculated mid-integration, in every time step. It is an estimation of the local error, and if the filter is working properly, this should be kept at the supplied tolerance TOL. This quantity is calculated according to

$$E_{\text{cont}}(t) = \|\boldsymbol{x}(t) - \boldsymbol{x}^{\text{pred}}(t)\|_\infty, \tag{III.169}$$

   where x is the solution given by the main method and $\boldsymbol{x}^{\text{pred}}$ is the solution given by the predictor. Note that this is only the case when the option `relerr` in the solver is set to false. In the case of this being true, the control error is further devided by $\|\boldsymbol{x}(t)\|_\infty$. More about this quantity can be found at Section II.6 where it is referred to as *error estimate*.

The first step is to check that the overall appearance of the solutions is correct. As can be seen, this is the case, which can be verified by comparison with the figures in Subsection III.3.8, Subsection III.3.12 and Subsection III.3.9. The rest of the problems, given in Section III.3, have been tested as well, but due to space limitations these results will be omitted.

Since the solutions of the given problems are periodic, we expect the corresponding step-size curves, step-size ratio curves and control error curves to behave periodically [5] as well, with the period

$$T = \frac{T_{\text{sol}}}{k}, \tag{III.170}$$

where T_{sol} is the period of the solution and $k \in \mathbb{Z}^+$. If this is not the case, the solver is not working properly. By studying the graphs, we see that our results are satisfactory in this regard.

The task of the error controller is to keep the control error at the set-point ϵ, in our case the supplied tolerance TOL, at all times, which for the solution ideally means that the local error is kept at the given tolerance. In the case of *Van der Pol* and *Lotka–Volterra*, the control error is kept in the interval $\sim [\text{TOL} \cdot 0.1, \text{TOL} \cdot 10]$, which is a proper behavior. In the case of *Oregonator*, the filter has a little bit of trouble at the stiffness peaks, however, it stabilizes around $E_{\text{cont}} = \text{TOL}$ quickly afterwards, which is a good sign. The most problematic part for the filter is where the stiffness changes rapidly as in Figure III.40. This due to the fact that the filter will try to stabilize the step-size h such that $\lambda_i h$ for all i, where λ_i is the ith eigenvalue of the problem, is stable in relation to the stability region and the error constant of the method. When the eigenvalues of the problem then moves rapidly from this "stable spot" in the complex plane, the error will also change rapidly creating those peaks. The faster this change happens, the harder it is for the filter to counteract it. If the big error change is created in one time step, then the filter has no

[5] Since we deal with numerics, periodic in this sense means *almost periodic*, that is, a few *bumps* may exist.

chance of adapting in the same step due to how we choose to reject. It can only prevent it from being even worse in the next time step. Interesting to investigate further, is what happens when different filters and methods are used on this problem to see if the peaks can be reduced.

By comparing the stiffness plots with the corresponding step-size plots, we see that they conform very well. When the problem gets stiffer, the step-size gets smaller, and the other way around. This is what we expect will happen according to the "symptom" of stiffness mentioned in Section III.2.

A control system is working at best when not tampered with. If limits are supplied, something called windup can arise making the controller behave badly and even making the process unstable. To prevent this, one can apply some anti-windup scheme, and it is important to check that this scheme will help the controller to reach a stable state again. In our control system we have two in-signals, previous *step-size ratios* and previous *control errors*, and one out-signal, the new *step-size ratio*. Two limits are applied to the out-signal, namely $r_{\min}$ and $r_{\max}$. These two values are denoted in the step-size ratio graphs as dashed horizontal lines. When the step-size ratio hits one of the given limits, special parts of the code are entered. To avoid bad behavior in the controller, our anti-windup scheme is activated (see Subsection II.8.7) [6]. In Figure III.40 the step-size ratio hits the roof, $r_{\max}$, but as can be seen, the filter handles this adequately. An interesting thing to further experiment with is the anti-windup scheme. We have only tried the one we are using right now, but it might be interesting to compare this scheme to other schemes.

[6] note that the anti-windup scheme can be turned off by setting the solver option `usebypass` to false

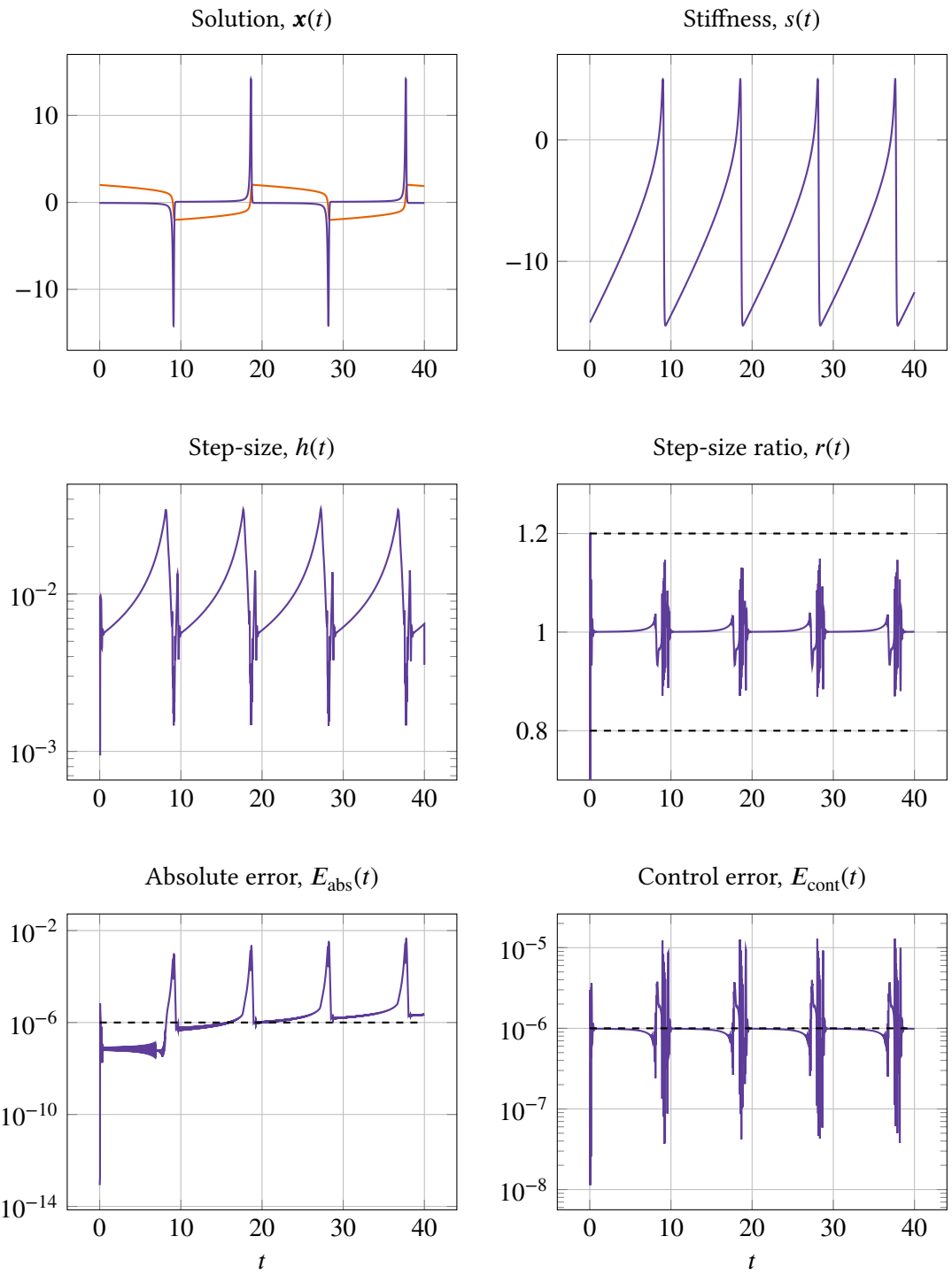

Figure III.38. Results for *Problem 8 – Van der Pol, $\mu = 10$* using `pmme`, AB5, H211PI and TOL $= 10^{-6}$. The rest of the solver settings can be found in Table III.9, Column 1. The dashed lines in the sub-figure titled *step-size ratio, $r(t)$* are the values of the limits r_{min} and r_{max} (see Subsection II.8.3 for more information), and the dashed lines in the error figures correspond to the supplied tolerance. The step-size, step-size ratio and control error is behaving periodically with an appropriate period. When the problem gets stiffer, the step-size becomes smaller and the other way around. The control error is kept around the set-point $\epsilon =$ TOL. The global absolute error stays around the tolerance as well (except at the stiffness peaks), however one can see a small drift-off.

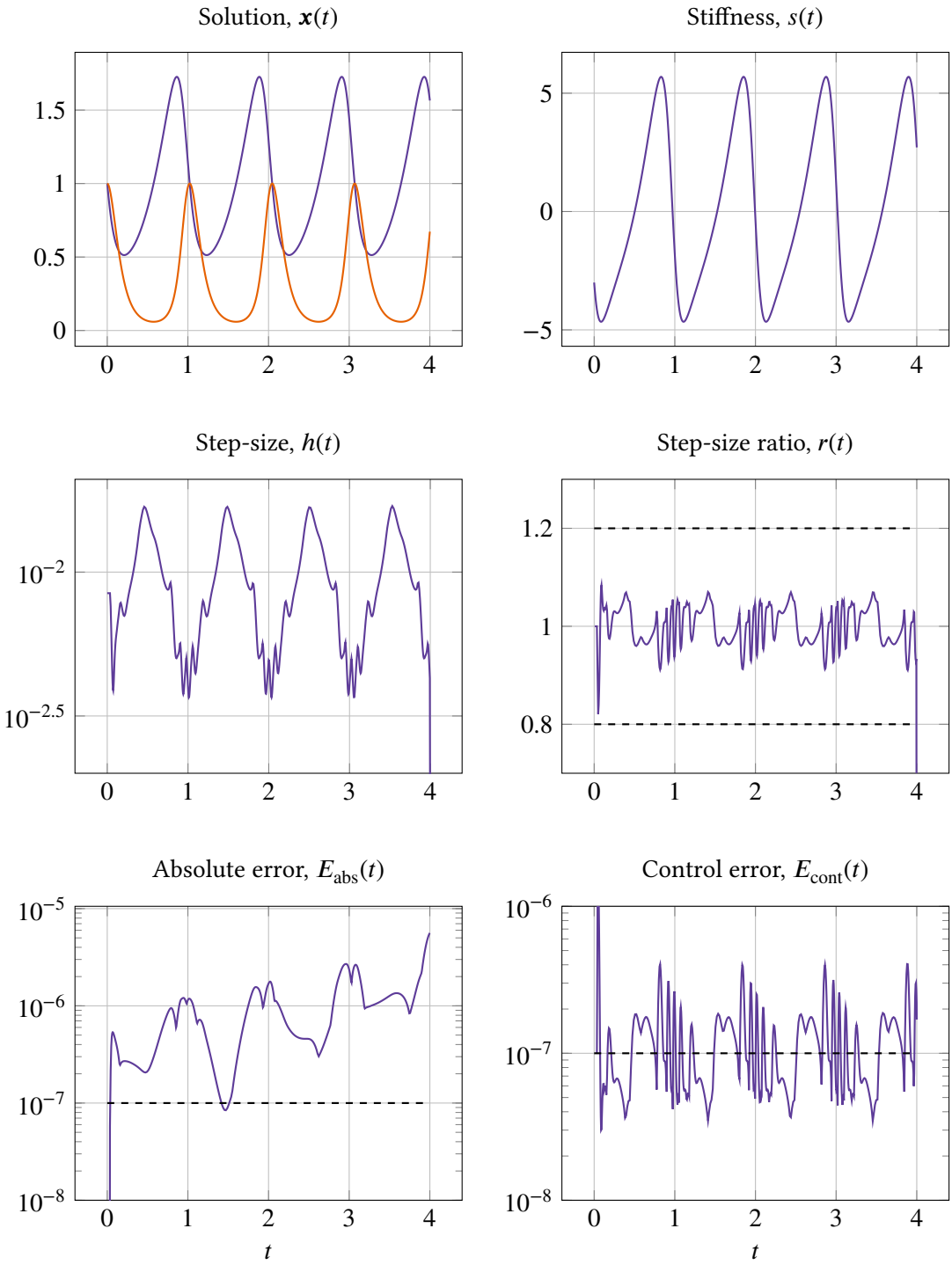

Figure III.39. Results for *Problem 12 – Lotka–Volterra* using `pmmip`, IDC34, H211*b* and TOL $= 10^{-7}$. The rest of the solver settings can be found in Table III.9, Column 2. The dashed lines in the sub-figure titled *step-size ratio,* $r(t)$ are the values of the limits r_{min} and r_{max} (see Subsection II.8.3 for more information), and the dashed lines in the error figures correspond to the supplied tolerance. The step-size, step-size ratio and control error is behaving periodically with an appropriate period. When the problem gets stiffer, the step-size becomes smaller and the other way around. The control error is kept around the set-point $\epsilon =$ TOL. The global absolute error one can see drift-off.

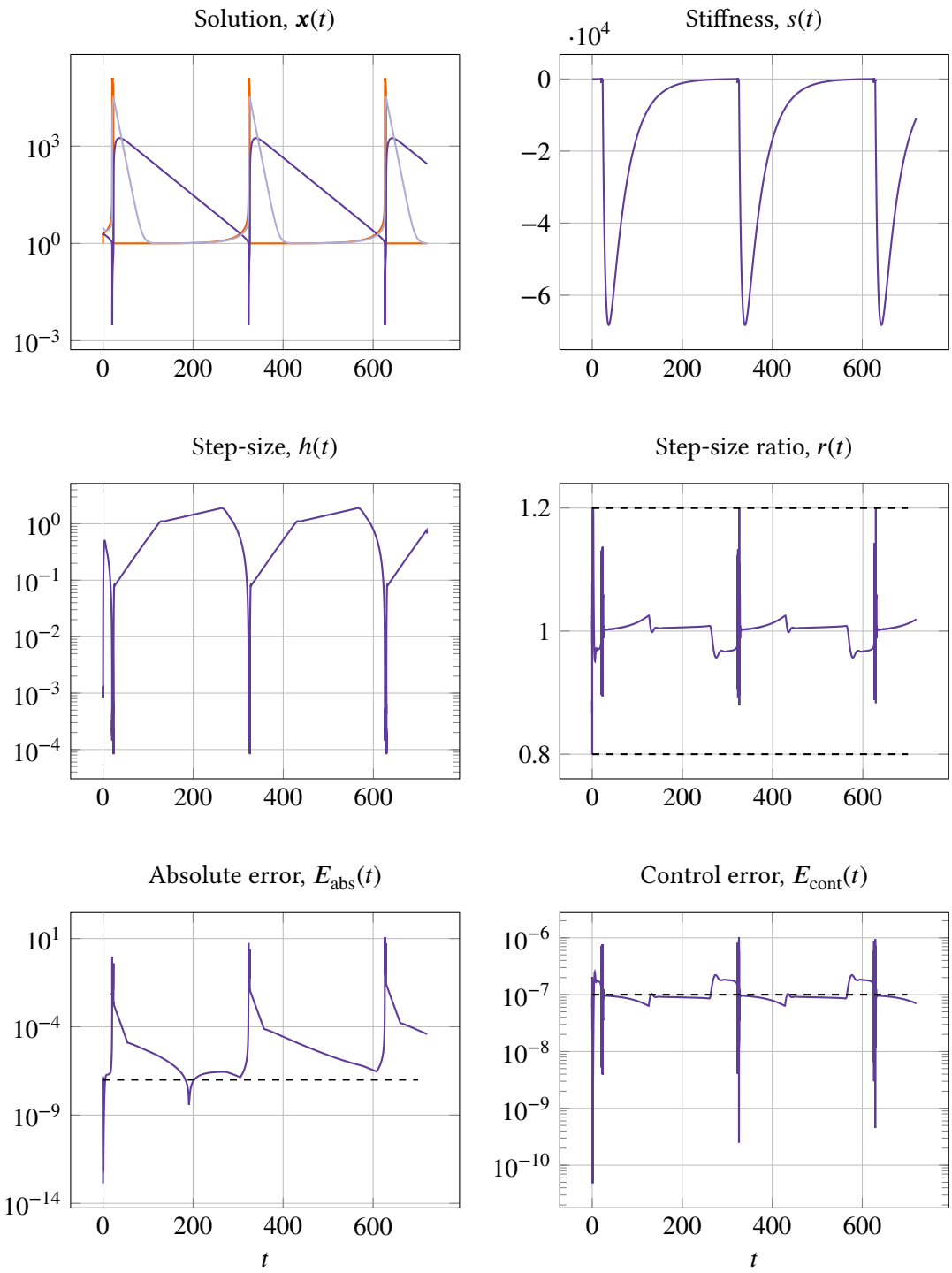

Figure III.40. Results for *Problem 9 – Oregonator* using `pmmi`, BDF5, H312*b* and TOL = 10^{-7}. The rest of the solver settings can be found in Table III.9, Column 3. The dashed lines in the sub-figure titled *step-size ratio, r(t)* are the values of the limits r_{min} and r_{max} (see Subsection II.8.3 for more information), and the dashed lines in the error figures correspond to the supplied tolerance. The step-size, step-size ratio and control error is behaving periodically with an appropriate period. When the problem gets stiffer, the step-size becomes smaller and the other way around. The control error is kept around the set-point ϵ = TOL. The global absolute error stays around the tolerance as well (except at the stiffness peaks), however one can see a small drift-off.

III.4.2. Test 2: Tolerance and work proportionality

In this test we will examine how the accuracy of a solution and the work to create it, when using different combinations of solvers, methods, filters and problems, behaves as a function of the supplied tolerance. The accuracy will be measured by the absolute error E (in the discrete L_1-norm) and the work will be measured by the total number of steps n.

The error E in this test is calculated using a *reference solution*, which is either the analytical solution of the problem, or the solution created by a *reference solver* (Matlab's `ode113` in the non-stiff case and Matlab's `ode15s` in the stiff case). Let us call the solution from our solver at time t_i, $\boldsymbol{x}_i$, the reference solution at the same time point $\boldsymbol{x}_i^{\text{ref}}$, and the absolute error between these two values measured in $\| \bullet \|_\infty$, e_i. The accuracy of a solution at one specific tolerance is then given by the *middle Riemann sum* of the values e_i, divided by the total time interval of the problem, that is,

$$e_i = \|\boldsymbol{x}_i^{\text{ref}} - \boldsymbol{x}_i\|_\infty, \quad i = 1, \dots, n, \tag{III.171}$$

$$E = \frac{1}{t_n - t_1} \cdot \sum_{i=1}^{n-1} \left(\frac{e_i + e_{i+1}}{2} \right) \cdot (t_{i+1} - t_i). \tag{III.172}$$

The tolerances are chosen according to

$$\text{TOL} = 10^{-y}, \quad y \in \mathbb{R}^+, \tag{III.173}$$

where the values of y are chosen equally spaced, starting at $y_{\min}$ and ending at $y_{\max}$. The number of tolerances is set to 100.

We choose a few of all combinations of solvers, methods, filters and problems to show here. The chosen combinations and corresponding setting can be found in Table III.10, whereas plots of the results are found in Figure III.41 – III.52. It is important to note that during this test, almost all combinations of methods, filters and problems were tried. However, since this results in a very large amount of combinations, all of them could not be displayed here.

As discussed in Section II.5, we know that the relation between the accuracy E and the supplied tolerance TOL of a well behaving solver should be approximately

$$\log(E) = k \log(\text{TOL}) + \log(m), \tag{III.174}$$

where k and m ideally equals 1. Furthermore, the important part is that the relation between $\log(E)$ and $\log(\text{TOL})$ is approximately linear, since the rest, i.e., k and $\log(m)$ can be adjusted by *tolerance scaling* and *tolerance calibration*. As can be seen in the accuracy vs tolerance plots, all curves are approximately linear. But, the slope of the curves are not exactly 1, neither are the slopes the same for methods of different orders, or for different problems. The key to this is that *error per step* is used and not *error per unit step*. First of all, what we supply is a tolerance which we would like to be equal to the global error at every time step. But, we can not estimate the global error mid-integration, so what we do is that we estimate the local error instead and hope that this will be a good enough estimation of the global error. This local error estimation is given to the controller with which it is trying to match the tolerance. Since the filter is actually controlling the local error — and not the global error as we would like — these local errors will accumulate for every step taken. This will lead to a *drift-off* from

$$\log(E) = \log(\text{TOL}) \tag{III.175}$$

Table III.10. Solver settings for `pmme`, `pmmip` and `pmmi` during test 2. *AS* is short for *Analytical Solution.*

Settings	`pmme`	`pmmip`	`pmmi`
filters	H211PI, H211*b*, PI3333, PI3040, PI4020, H312*b*	H211PI, H211*b*, PI3333, PI3040, PI4020, H312*b*	H211PI, H211*b*, PI3333, PI3040, PI4020, H312*b*
methods	AB2–AB5 EDF2–EDF5	AM2–AM5	BDF2–BDF5
nbroftols	100	100	100
$k_{\min}$	4	4	4
$k_{\max}$	10	10	10
perc	[0.8, 1.2]	[0.8, 1.2]	[0.8, 1.2]
unit	false	false	false
usebypass	true	true	true
reference	`ode113` (or AS)	`ode113` (or AS)	`ode15s` (or AS)

to

$$\log(E) = k \log(\text{TOL}) \tag{III.176}$$

where the slope k is roughly only depending on the number of steps taken. This would also explain the trend that the higher the order of the method, the higher the slope of the curve. Of course, this is only a simplified model of the whole process. The reality is a bit more complicated.

The hypothesis is that the term $\log(m)$ mainly comes from the fact that we are not using the real local error, but a rough estimation of it. As a local error estimate, we use (in our implementation)

$$\|\boldsymbol{x}_n - \boldsymbol{x}_n^{\text{pred}}\|_\infty, \tag{III.177}$$

though, as can be read about in Subsection II.6.2, a better approximation of the local error in the case of `pmme`, is in fact Equation II.46. By using this estimate instead, the term $\log(m)$ would probably be reduced.

As told before, k and $\log(m)$ can be adjusted for afterwards by *tolerance scaling* and *tolerance calibration.* But, since the different problems and different methods will create different values of k and $\log(m)$, to make it perfect, this scaling and calibration must be applied to every combination of method and problem.

The curves corresponding to one specific method in combination with different filters, should, if working properly, look about the same. The number of steps should be approximately the same, leading to the same relation between $\log(E)$ and $\log(\text{TOL})$. This behavior can be seen in the figures. The more stringent the tolerance, the more alike the different filter curves are, and in most figures they can not be separated at all. Of course, if the combination of solver, method, filter and problem is bad, this would not be the case any more, however, so far no such combination has been found. The only things found so far, that are not working properly, are the methods ABx ($x > 5$), EDFx ($x > 5$) and BDF6. It is believed that the reason for the explicit methods (AB and EDF) behaving badly, is that they have relatively small stability regions. In

the case of BDF6, we know that this method is barely zero-stable, which might be the reason for the improper behavior when used in combination with the solver. These methods have been used in combination with different filters and problems, however no special connection between which combinations work and which do not work has been found. They have also been used in combination with a smaller step size ratio interval, however this did not seem to improve the results. A further investigation using these methods needs to be done.

The corresponding work vs tolerance plots (see Figure III.41—III.52) are good. The lines are straight with a negative slope. The lower the order of the method, the more steps are needed just as expected. It can also be seen that the more stringent the tolerance is, the bigger the difference between the work of the methods with different order is. For example, in the case of Figure III.41, at $\mathrm{TOL} = 10^{-10}$ AB4 takes about $4,5$ times as many steps as AB5, and at $\mathrm{TOL} = 10^{-4}$ only about 2 times as many steps. Also, (in the case of `pmme`) a small difference when using two different methods of the same order (ABx and EDFx) can be seen, which can be explained by the fact that the different methods have slightly different stability regions and error constants.

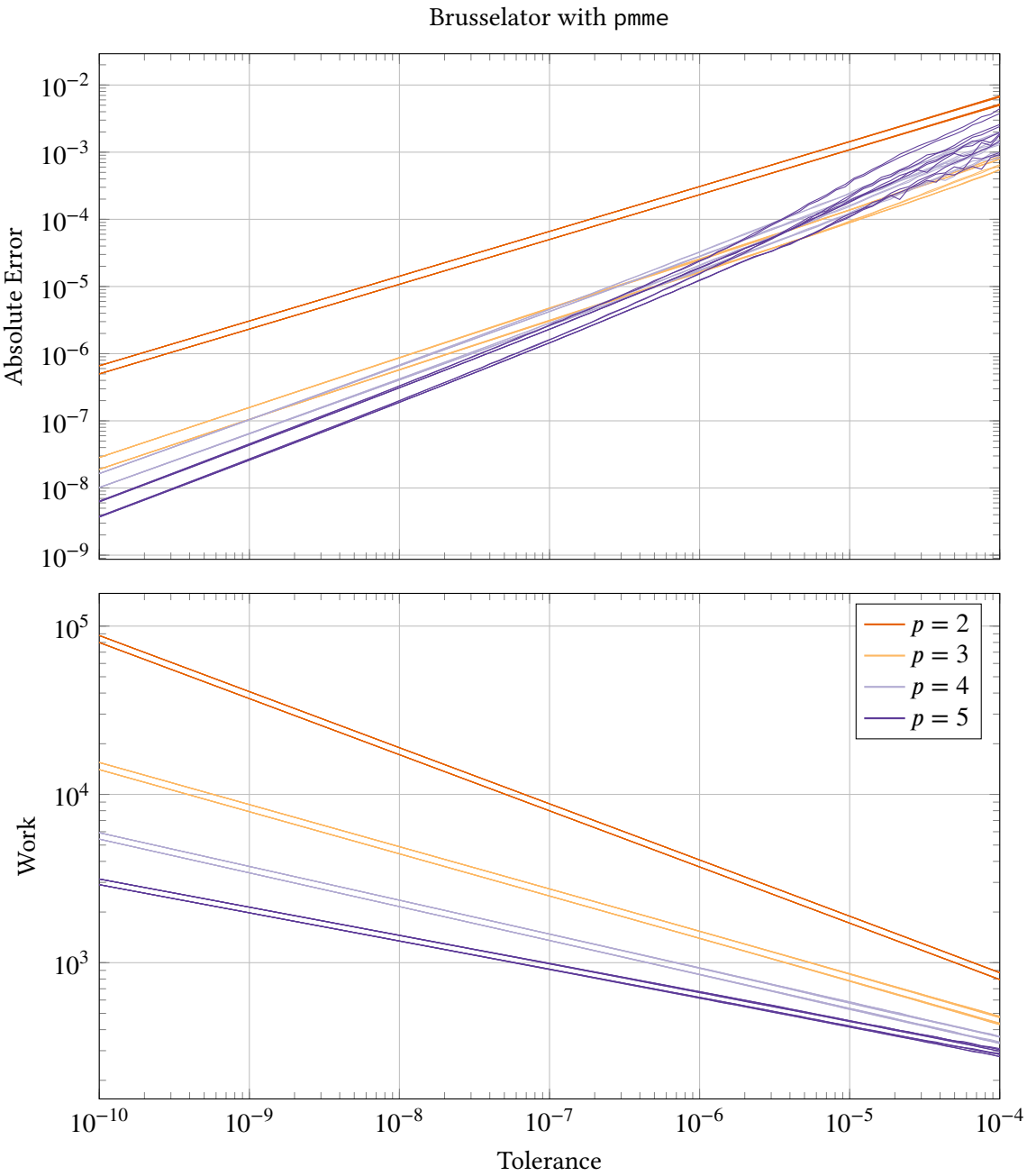

Figure III.41. Accuracy vs tolerance and work vs tolerance of *Brusselator* with solver pmme. The methods used are AB2–AB5 and EDF2–EDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the methods AB2 and EDF2 in combination with the different filters stated above. This means that in every shade there are 12 lines, however in most cases only two can be seen: one for ABx and one for EDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 5$ in the accuracy vs tolerance plot, one can see that the lines are starting to wobble at looser tolerances in the accuracy vs tolerance plot.

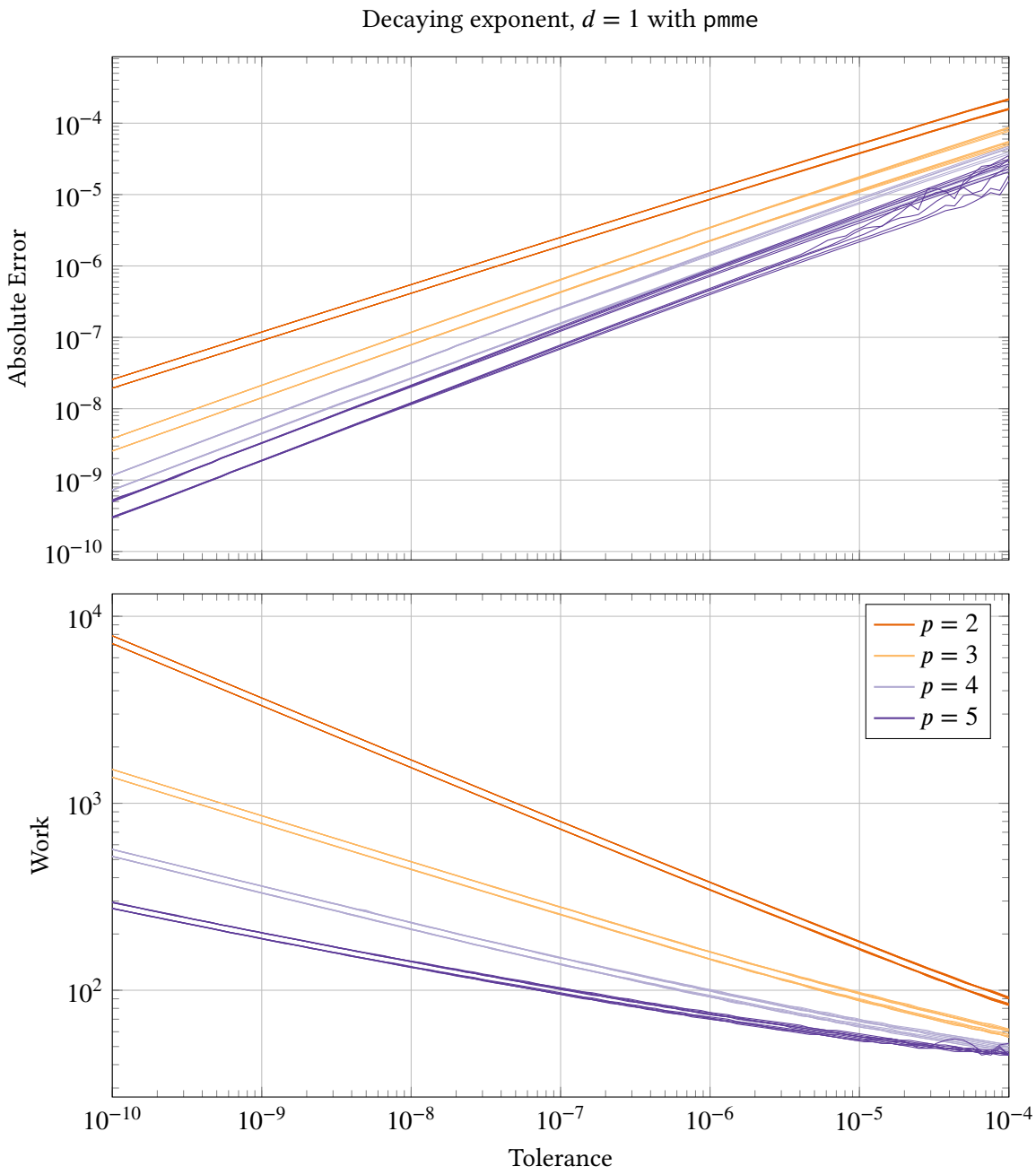

Figure III.42. Accuracy vs tolerance and work vs tolerance of *Decaying exponent, $d = 1$* with solver pmme. The methods used are AB2–AB5 and EDF2–EDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. As a reference, the analytical solution is used. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the methods AB2 and EDF2 in combination with the different filters stated above. This means that in every shade there are 12 lines, however in most cases only two can be seen: one for ABx and one for EDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 5$ in the accuracy vs tolerance plot, one can see that the different filter lines are starting to separate at looser tolerances.

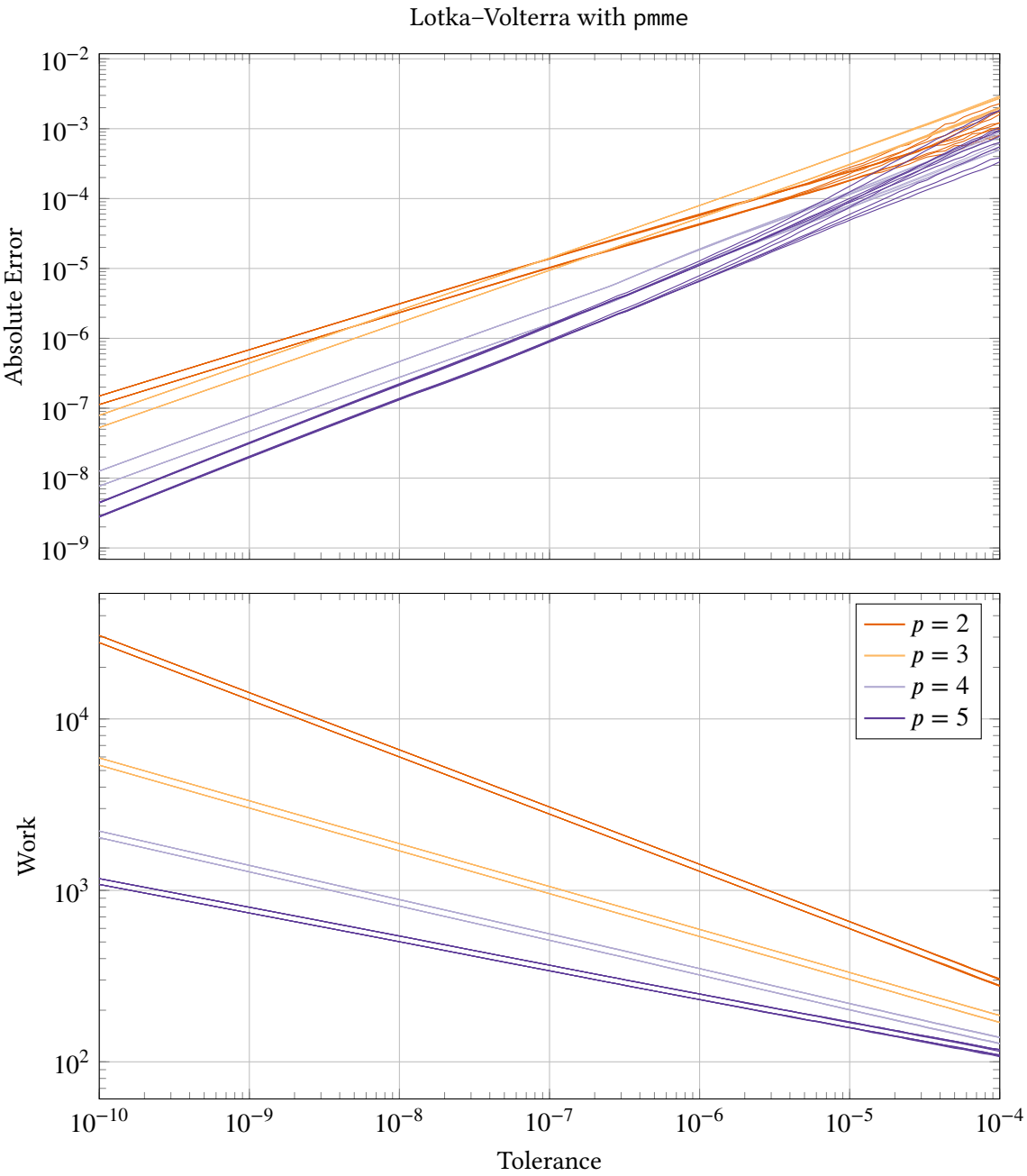

Figure III.43. Accuracy vs tolerance and work vs tolerance of *Lotka–Volterra* with solver pmme. The methods used are AB2–AB5 and EDF2–EDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the methods AB2 and EDF2 in combination with the different filters stated above. This means that in every shade there are 12 lines, however in most cases only two can be seen: one for ABx and one for EDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the accuracy vs tolerance plot one can start to notice splitting of the filter lines at looser tolerances.

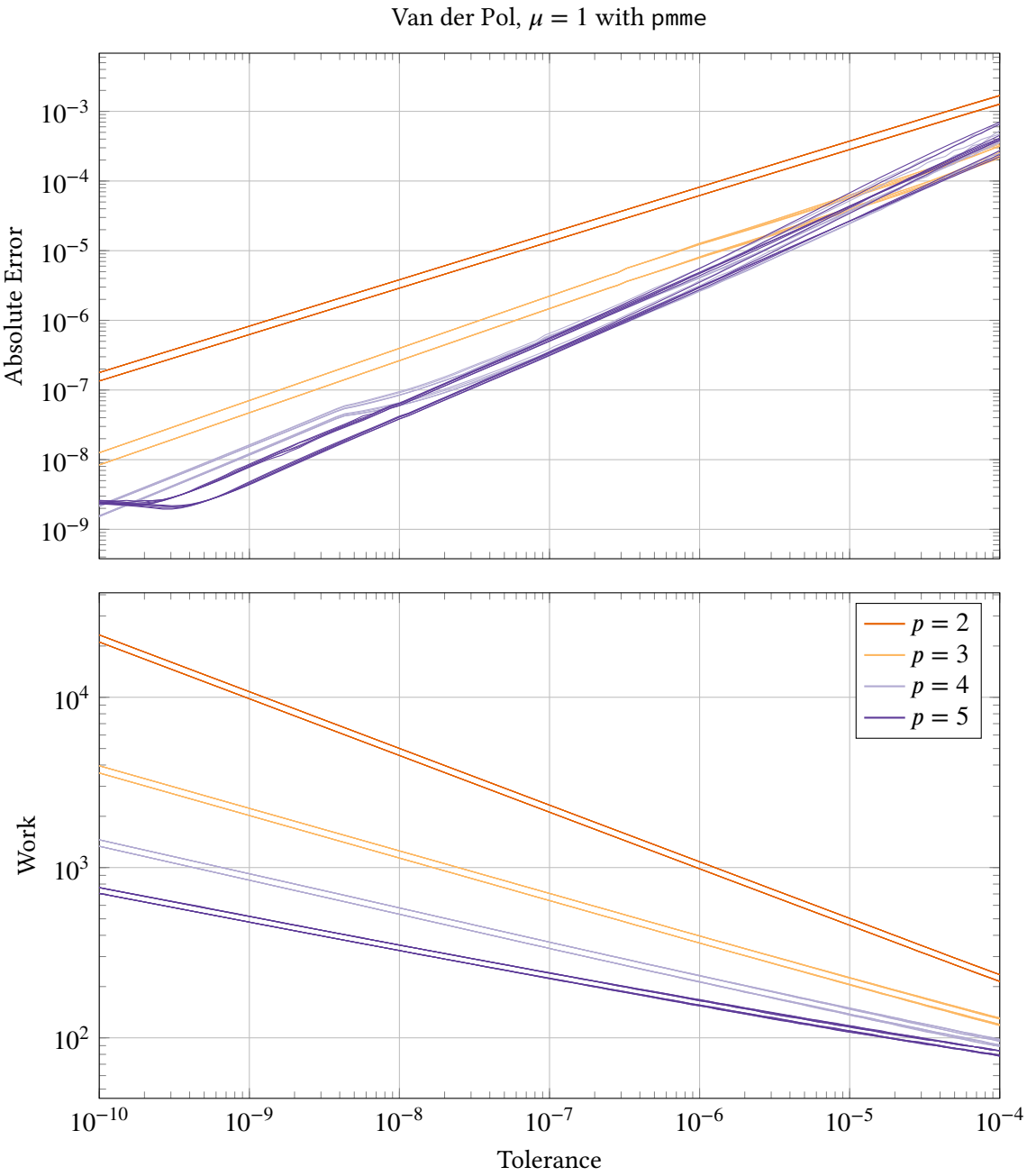

Figure III.44. Accuracy vs tolerance and work vs tolerance of *Van der Pol,* $\mu = 1$ with solver pmme. The methods used are AB2–AB5 and EDF2–EDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the methods AB2 and EDF2 in combination with the different filters stated above. This means that in every shade there are 12 lines, however in most cases only two can be seen: one for ABx and one for EDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness.

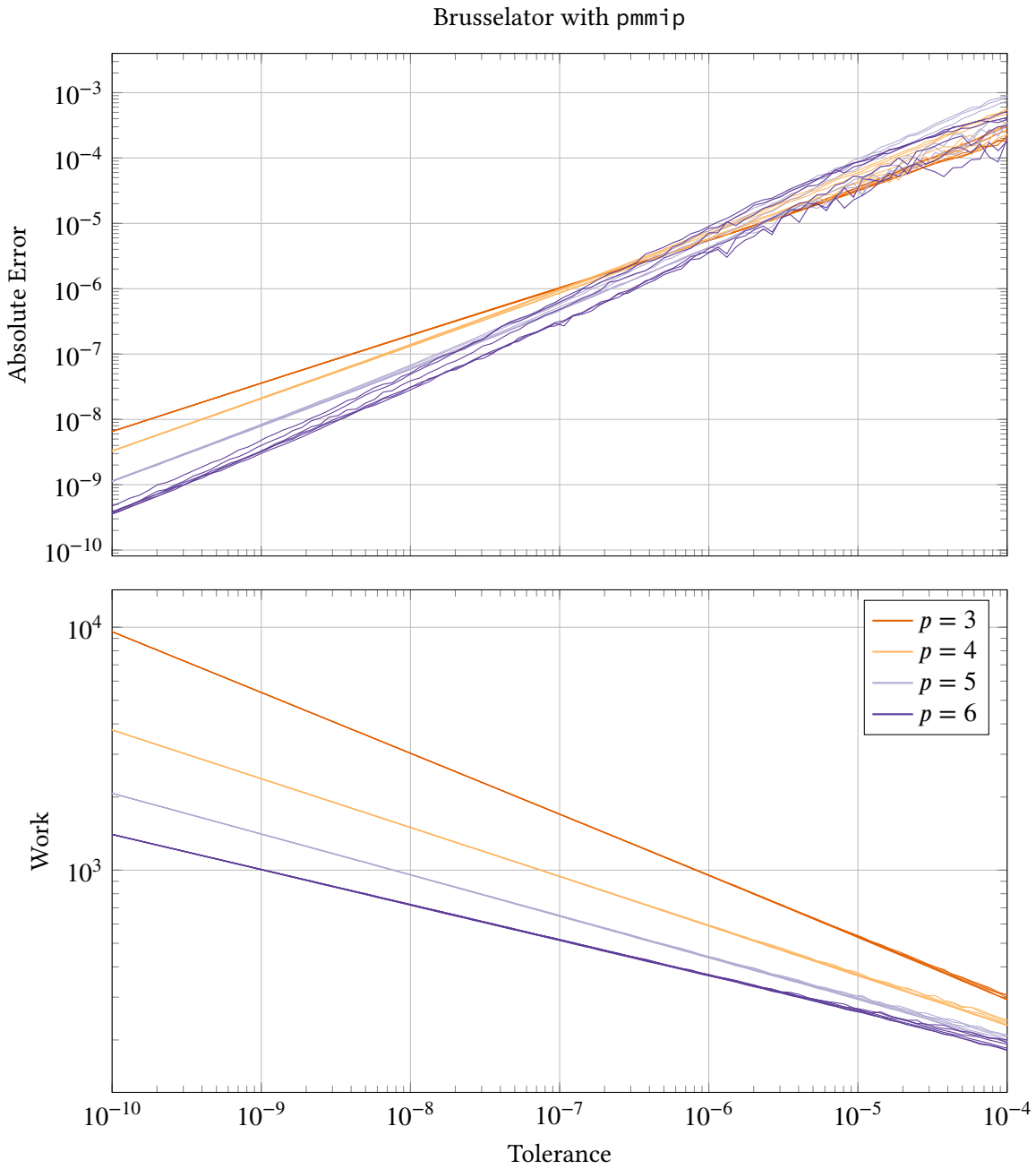

Figure III.45. Accuracy vs tolerance and work vs tolerance of *Brusselator* with solver `pmmip`. The methods used are AM2–AM5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method AM2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for AMx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 6$ in the accuracy vs tolerance plot, one can see the splitting of the different filter lines. The looser the tolerance, the more the splitting and wobble of the lines.

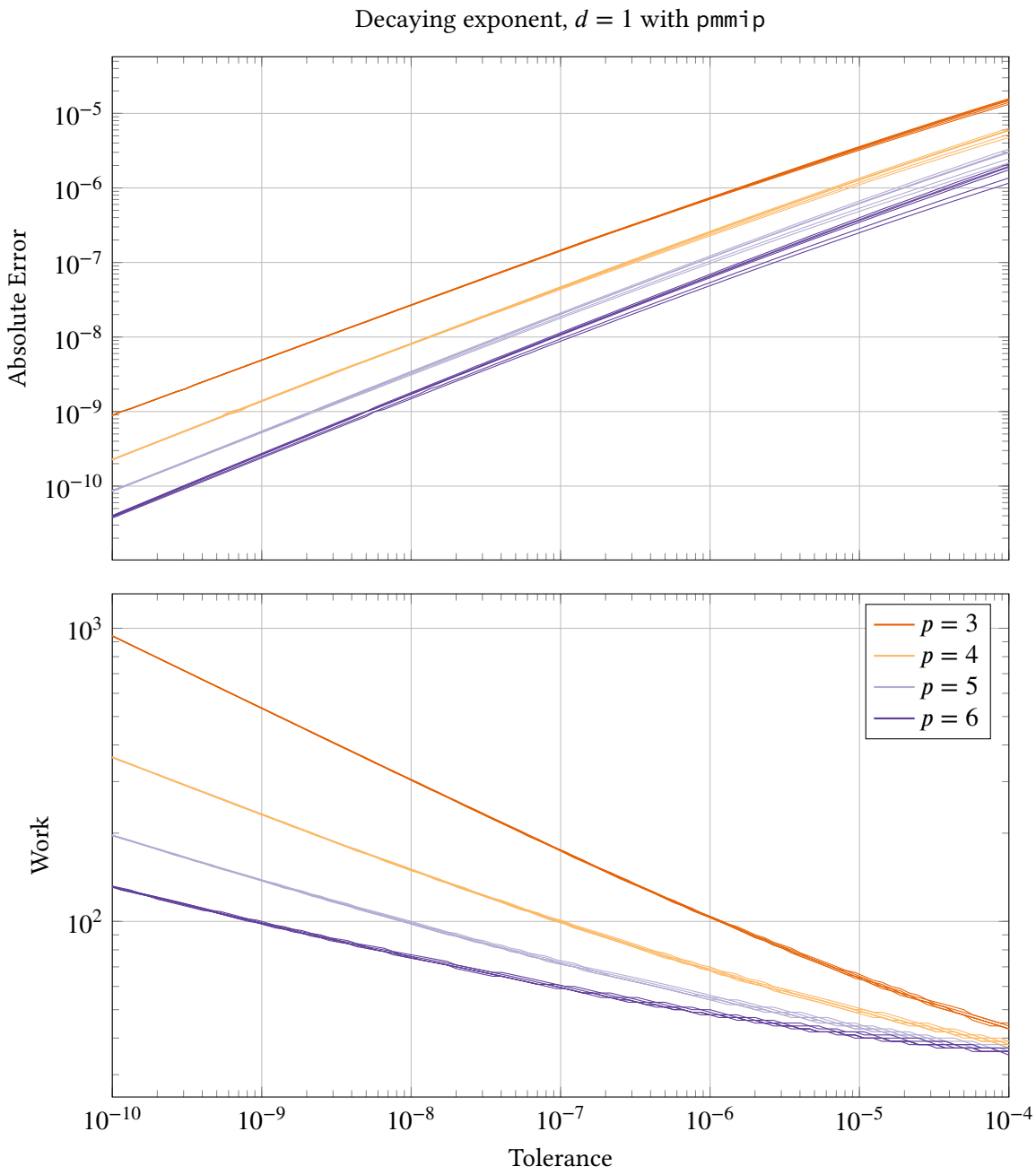

Figure III.46. Accuracy vs tolerance and work vs tolerance of *Decaying exponent, $d = 1$* with solver `pmmip`. The methods used are AM2–AM5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. As a reference, the analytical solution is used. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method AM2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for AMx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. At looser tolerances in the accuracy vs tolerance plot, one can start to see splitting between the filter lines. In the work vs tolerance plot, one start to notice that AM5 lines get flattened out at looser tolerances.

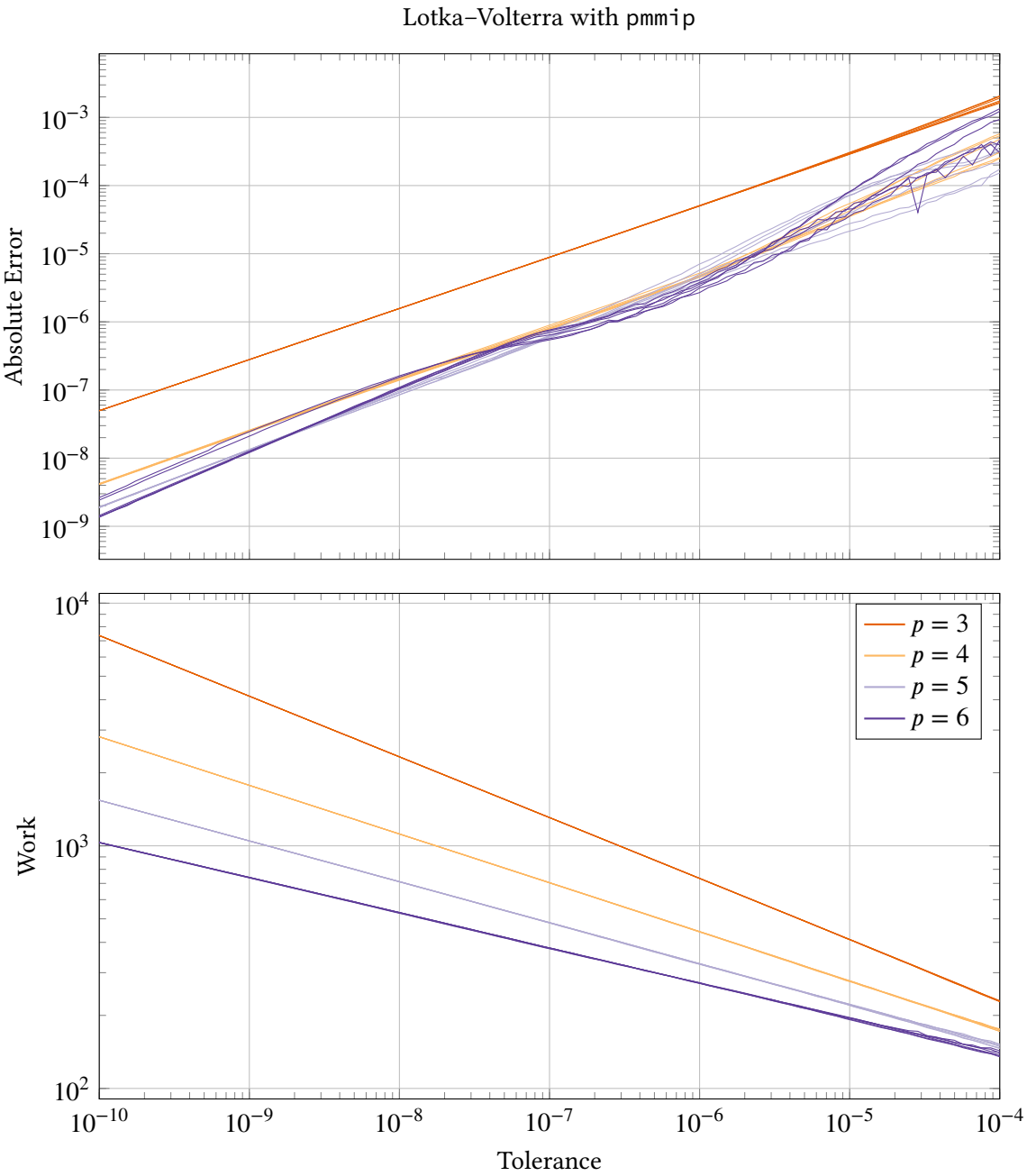

Figure III.47. Accuracy vs tolerance and work vs tolerance of *Lotka–Volterra* with solver `pmmip`. The methods used are AM2–AM5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method AM2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for AMx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 6$ in the accuracy vs tolerance plot, one can see that the filter lines are a little bend around TOL $= 10^{-9}$. One can also notice that the same lines are starting to wobble at looser tolerances.

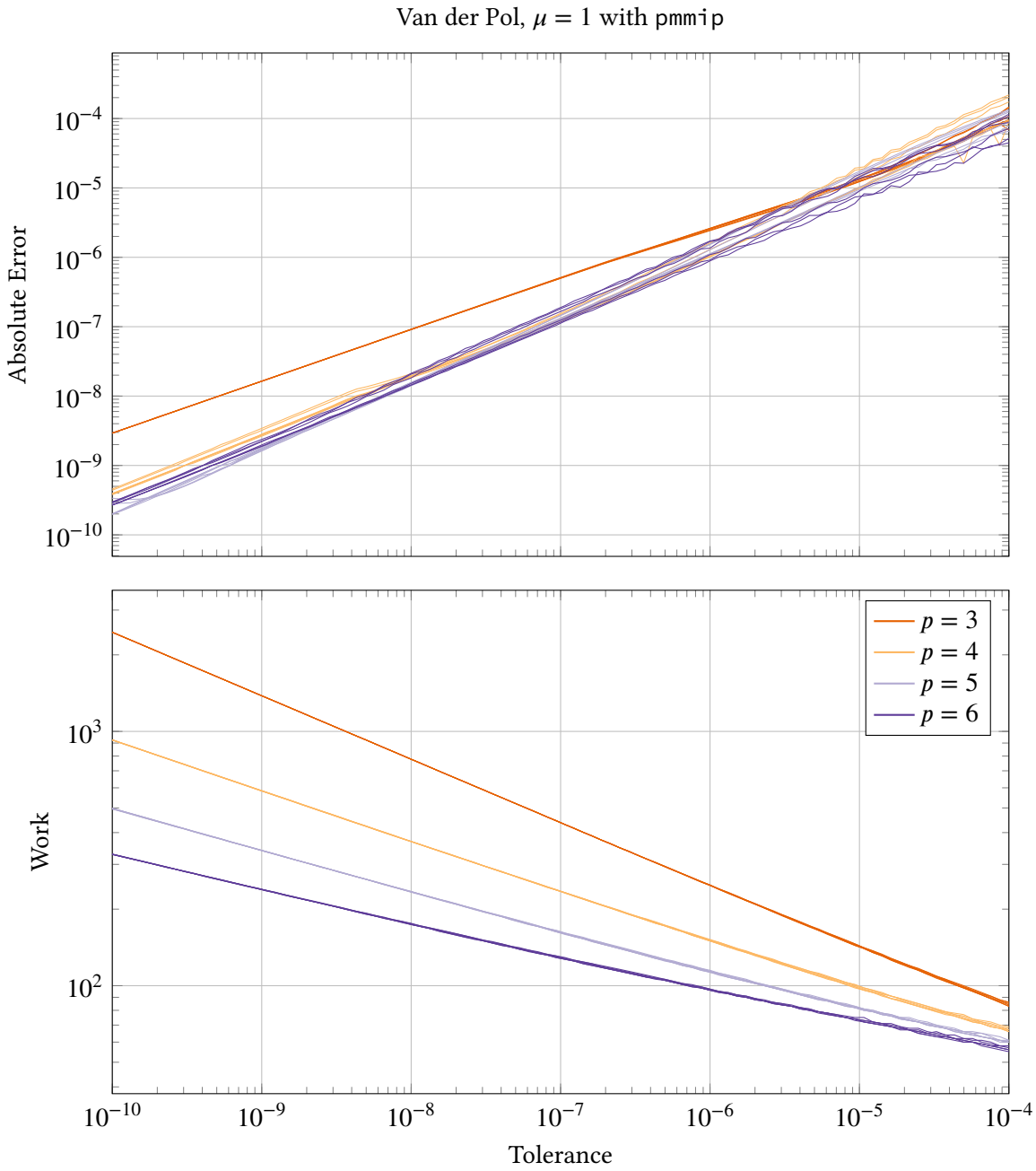

Figure III.48. Accuracy vs tolerance and work vs tolerance of *Van der Pol*, $\mu = 1$ with solver pmmip. The methods used are AM2–AM5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method AM2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for AMx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 6$ in the accuracy vs tolerance plot, one can see that the lines are starting to wobble at looser tolerances.

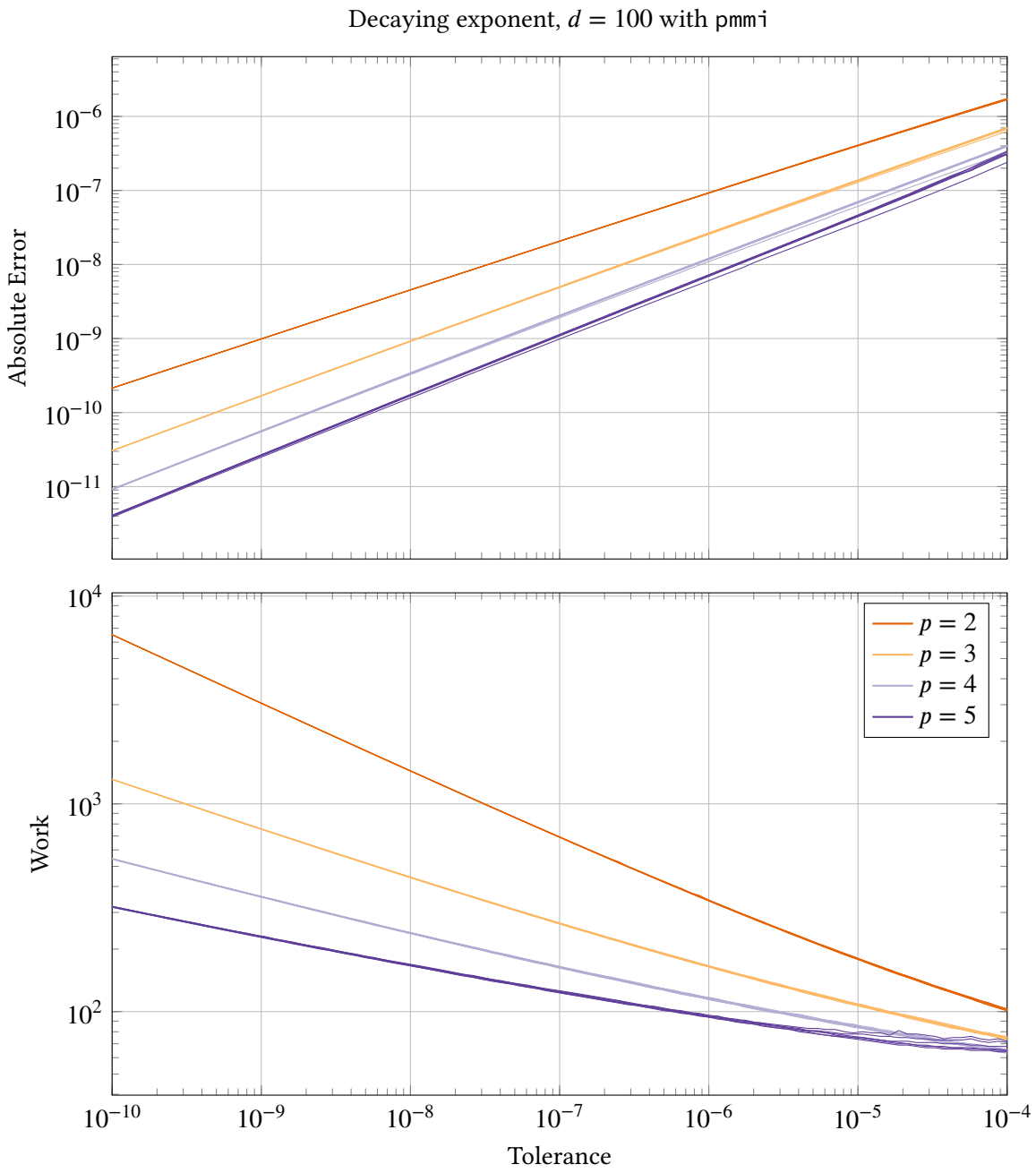

Figure III.49. Accuracy vs tolerance and work vs tolerance of *Decaying exponent, $d = 100$* with solver `pmmi`. The methods used are BDF2–BDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. As a reference, the analytical solution is used. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the method BDF2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for BDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 5$ in the work vs tolerance plot, one can see that the lines are getting flattened out at looser tolerances.

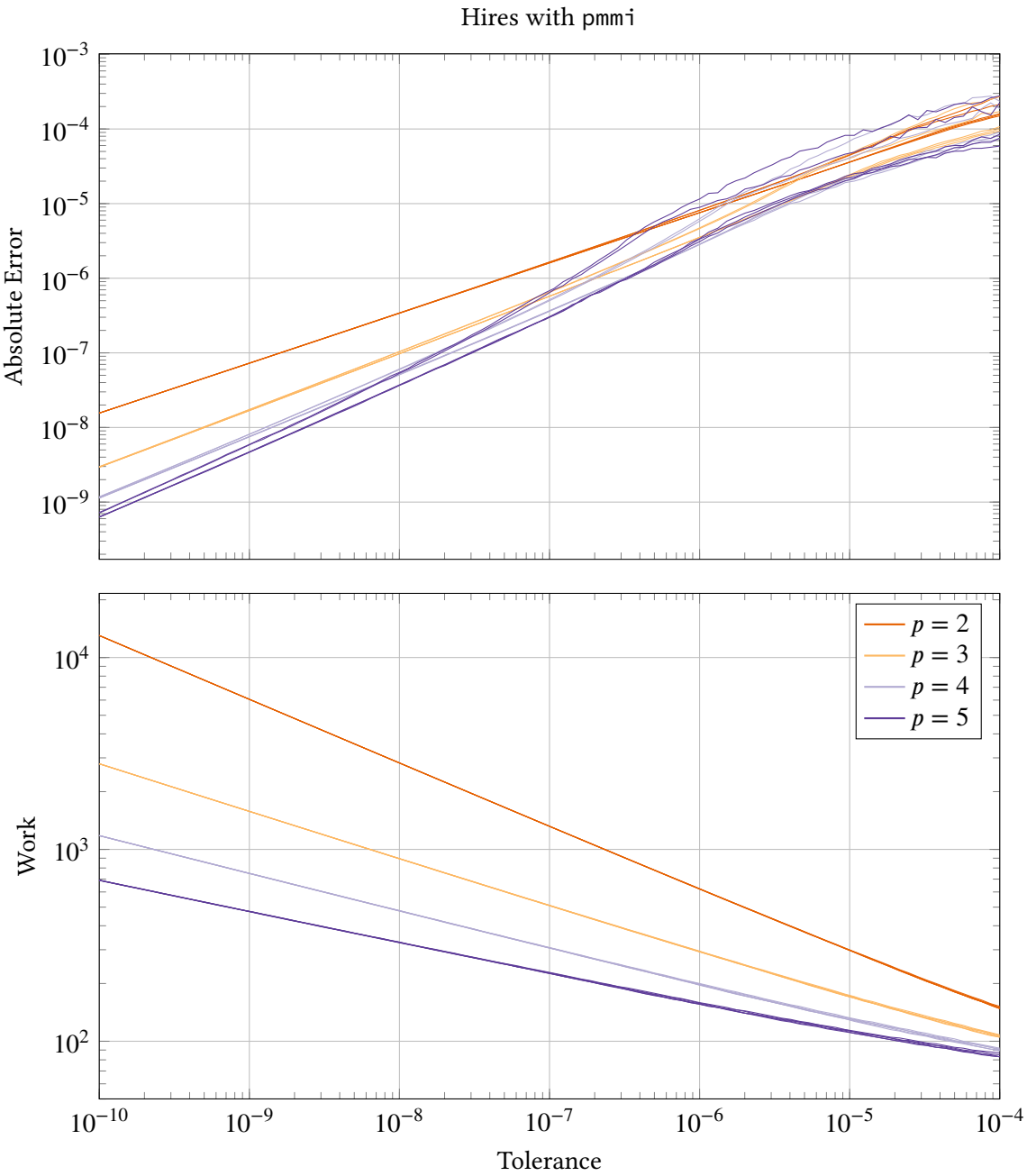

Figure III.50. Accuracy vs tolerance and work vs tolerance of *Hires* with solver `pmmi`. The methods used are BDF2–BDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 2 displays the method BDF2 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for BDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 5$ in the accuracy vs tolerance plot, one can see that the filter lines are splitting, and at looser tolerances they even start to wobble a little. This phenomenon can also be seen in the case of $p = 4$, but it is not as clear.

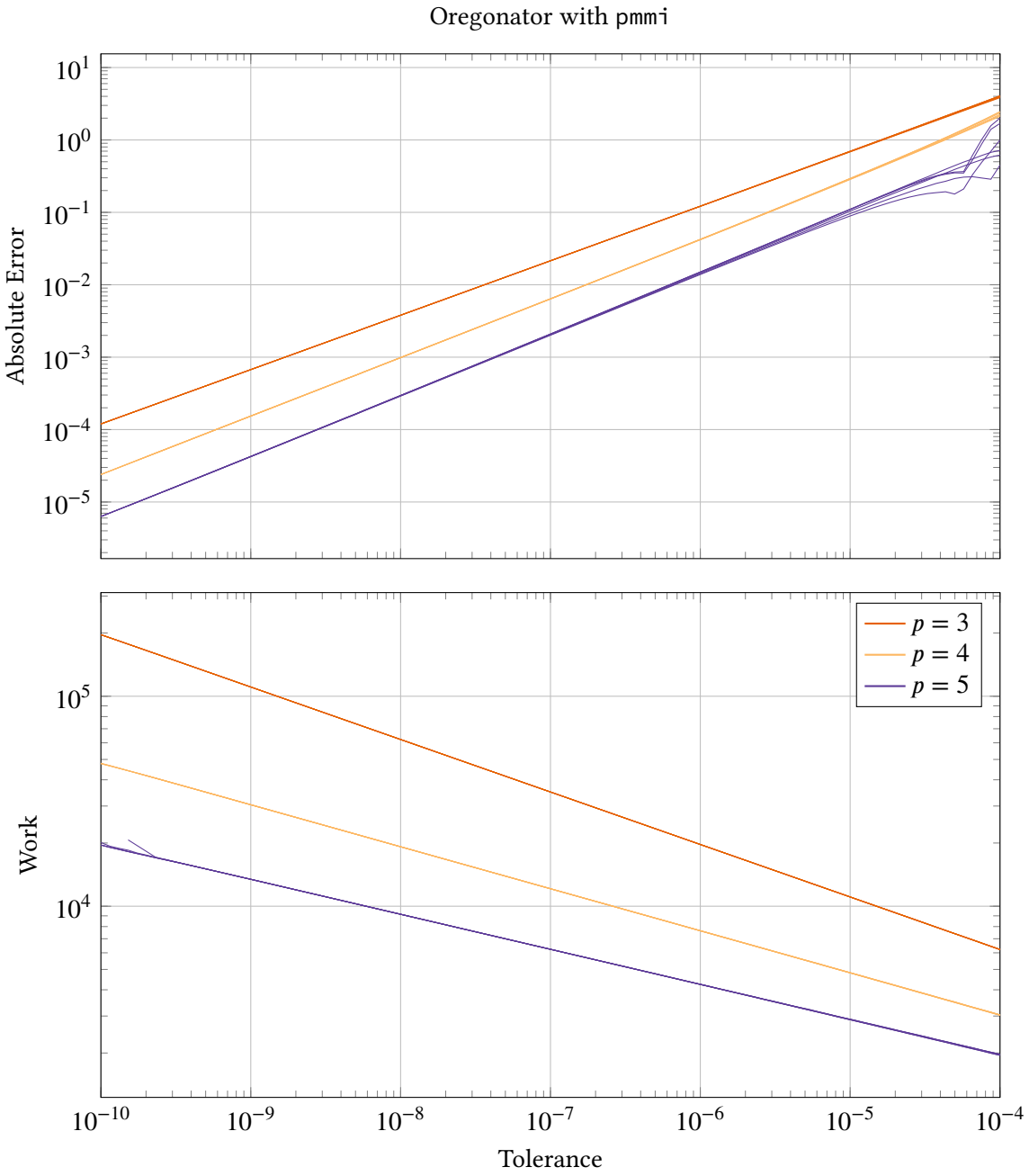

Figure III.51. Accuracy vs tolerance and work vs tolerance of *Oregonator* with solver `pmmi`. The methods used are BDF3–BDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method BDF3 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for BDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, which is a sign of robustness. In the case of $p = 5$ one can see that the lines are starting to wobble at looser tolerances in the accuracy vs tolerance plot.

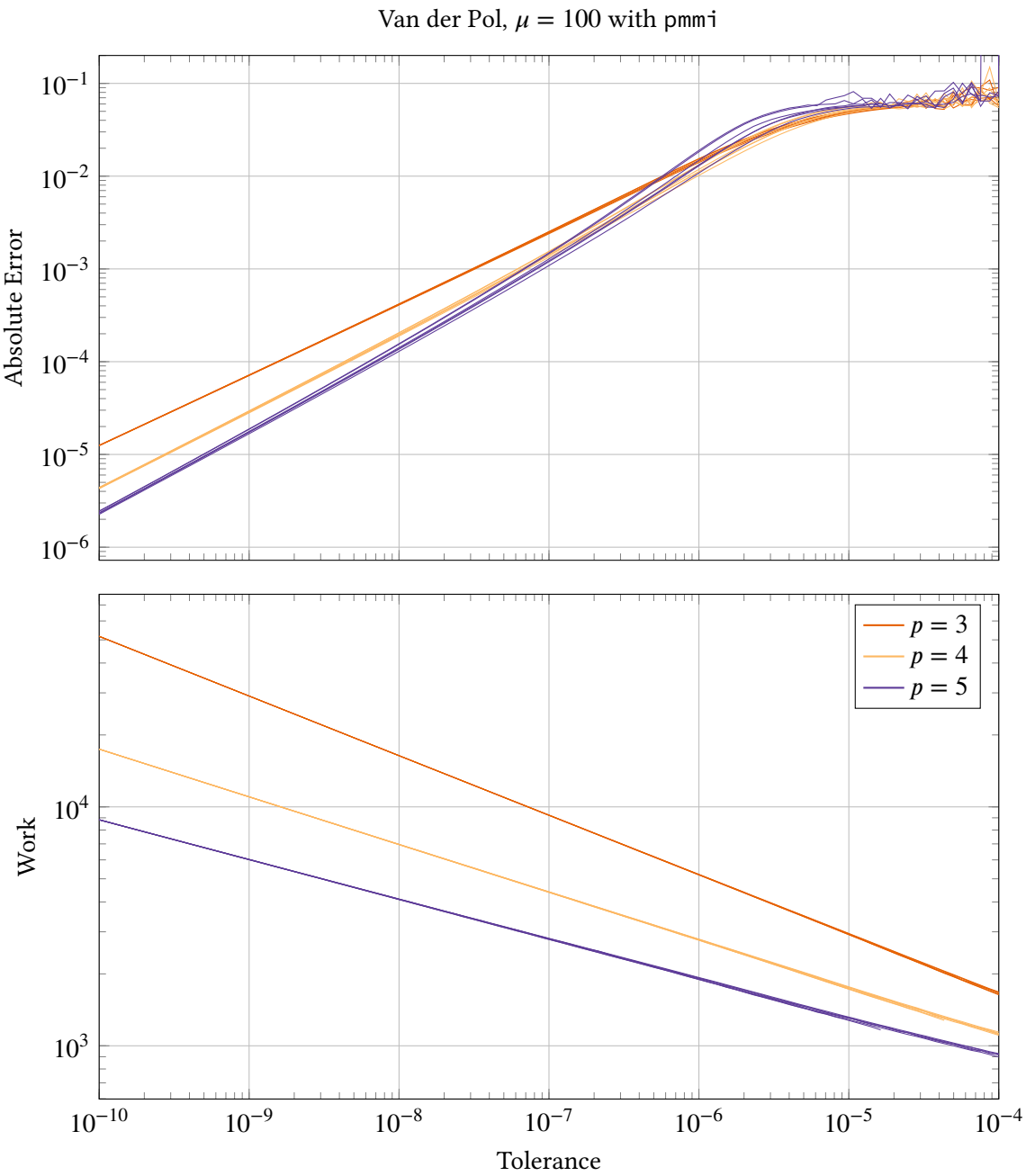

Figure III.52. Accuracy vs tolerance and work vs tolerance of *Van der Pol,* $\mu = 100$ with solver `pmmi`. The methods used are BDF3–BDF5 and the filters used are H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*. More solver settings can be found in Table III.10. The different line shades, separate the different method orders, meaning that, for example, the line shade corresponding to method order 3 displays the method BDF3 in combination with the different filters stated above. This means that in every shade there are 6 lines, however in most cases only one can be seen: one for BDFx. The reason for this is that the different filters in combination with a special method will behave in about the same way in these regards and therefore no difference between the lines can be seen. The lines in both plots are almost perfectly straight, except in the accuracy vs tolerance plot at looser tolerances. Here we see that we reach a plateau, however, notice that the corresponding accuracy at these tolerances is really low ($\sim 10^{-1}$).

III.4.3. Test 3: Stiffness test on solver `pmme`

In this section we will analyze the solver `pmme`. We will only use the decaying exponent (see problem in Subsection III.3.14), though we will vary the stiffness of it. The reason for using this problem is that it has only one eigenvalue, $\lambda = -d$, which is constant over time, and it has a known analytical solution which makes it possible to calculate the actual global error created by the solver. The methods used are *AB2 – AB5*, *EDF2 – EDF5*, and the corresponding (fixed step-size) stability regions and (fixed step-size) error constants can be found in Figure II.4, II.5 and II.6.

The questions we would like to answer in this section are "What happens with the accuracy vs tolerance and work vs tolerance when we decrease the eigenvalue from $\lambda = -1$ to $\lambda = -10$?", "Are these changes the same for all methods and filters?" and "Can these changes be explained by theory?". The solver settings used during this test, can be found in Table III.11.

Table III.11. Solver settings in Test 3

Settings	*Values*
solver	`pmme`
methods	AB2 - AB5, EDF2 - EDF5
filters	H211PI, H211*b*, PI3333, PI3040, PI4020 and H312*b*
nbroftols	100
perc	[0.8, 1.2]
unit	false
usebypass	false
reference	analytical solution

The accuracy will be measured in absolute error E and the work will be measured in total number of steps n just as in the previous test, and will be calculated accordingly as well.

According to Figure III.53 (Row 1), we obtain a nice result in the non-stiff case. However, when the problem is made stiffer, we are starting to see some strange behavior. We start to analyze the *work vs tolerance plot* (Figure III.53, Row 2, Column 2). For the methods of order 2 (AB2 and EDF2), the result is still good. We are given a straight line with a constant slope just as in the non-stiff case. For the higher order methods, we see that the slope of the lines are approaching 0 when the tolerance is made looser. The higher the order of the method, the sooner (at stricter tolerance) this phenomenon occurs. This fact can be explained by the following: The looser the tolerance, the longer the steps. For higher order methods the steps taken are generally longer and the stability region smaller, which means that the stability region border will be reached "earlier" when the tolerance is increased. When this border is hit, the steps can not be made any longer, which creates the flattening of the curves. Already at $\text{TOL} = 10^{-10}$, EDF5 and AB5 take as long steps as possible (the stability boarder is hit). By using the stiff solver instead, this effect should disappear. Further, in the case of $p = 5$, the method curves — all curves corresponding to AB5 and all curves corresponding to EDF5 — are not at all the same unlike the non-stiff case. The lines corresponding to EDF5 (the lines going through the point $(1 \cdot 10^{-10}, 3 \cdot 10^{2})$) reach a lower number of steps when the tolerance is increased, than the lines corresponding to AB5 (the lines going through the point $(1 \cdot 10^{-10}, 8 \cdot 10^{2})$). This can be explained by the fact that the stability region border for EDF5 is about 0.6, while the corresponding value in the case of AB5 is about

0.2. This method splitting phenomenon can also be seen in the case of $p = 4$, however it is not as clear.

In the case of the accuracy vs tolerance plot (Figure III.53, Row 2, Column 1) we see straight lines with small wiggles. In the case of $p = 2$, we do not see much wiggling, however, for the higher order methods we start to see a lot of wiggles. A hypothesis is that the wiggles can be explained by the fact that we reach the step-size ratio limits during the integration leading to rejections. At different tolerances these rejections will be reached at different time points, leading to both a different number of rejections and a variation in the behavior after the rejections. In Figure III.54 and Figure III.55, one can see a comparison of all filters used and the methods AB3 and AB4. According to these results, PI3333, PI3040 and PI4020, seem to be better at stabilizing the error in this case. My hypothesis is that this can be explained by the negative β_2 parameter (see Table II.2), the same parameter is positive in the other filters. This parameter will act as a damper when the stability region is hit. The other filters will react too fast, and get an overshoot creating an oscillation with high amplitude and high frequency. It would be interesting to see if PI3333, PI3040 and PI4020 would give better results also in the combination of other explicit methods and moderately stiff problems.

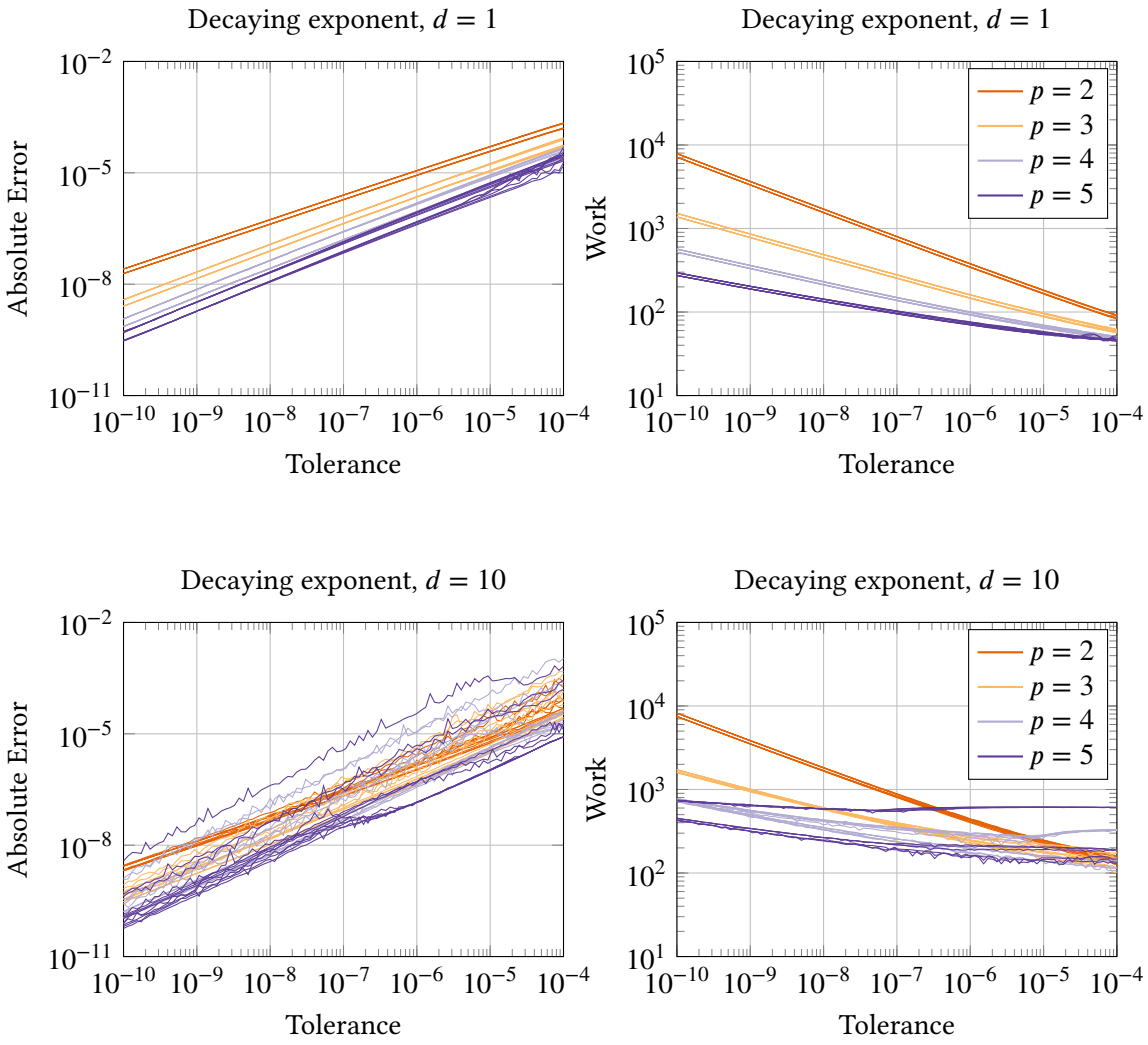

Figure III.53. Decaying exponent with $d = 1$ vs $d = 10$. Solved with solver `pmme`. For a list of all settings used see Table III.11. In the upper subfigures (Row 1), $d = 1$ is used. In the lower subfigures (Row 2), $d = 10$ is used.

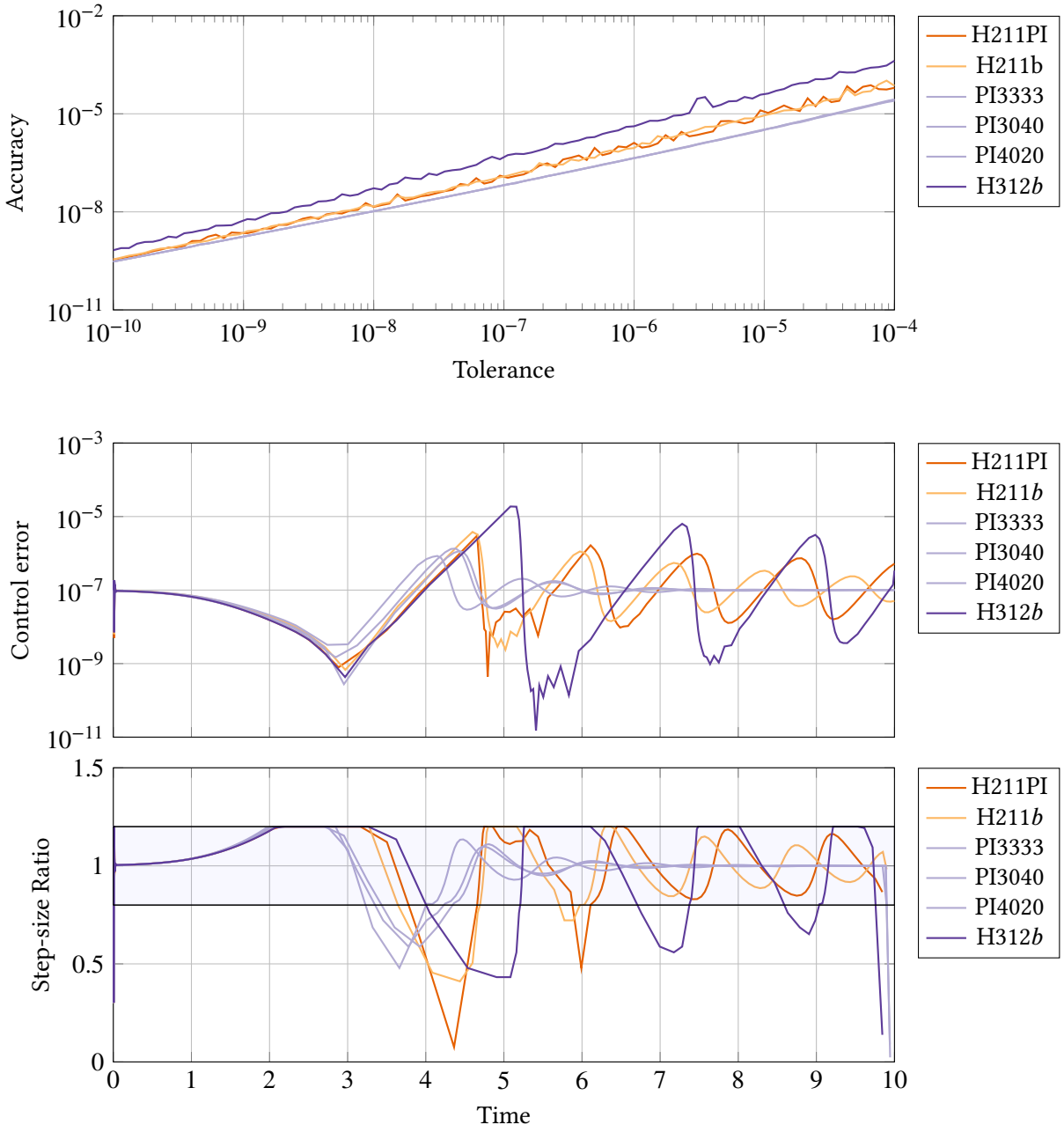

Figure III.54. Problem: The test equation with $d = 10$. Filters: H211PI, H211b, PI3333, PI3040, PI4020 and H312b. Method: AB3. Stability region limit for AB3 is -0.5455. The first figure shows the accuracy vs tolerance, the second one shows the control error as a function of time at TOL $= 10^{-7}$ and the third one shows the step-size ratio as a function of time at TOL $= 10^{-7}$. The filters PI3333, PI3040 and PI4020 behaves better than the rest. The same shade is used for these well behaving filters.

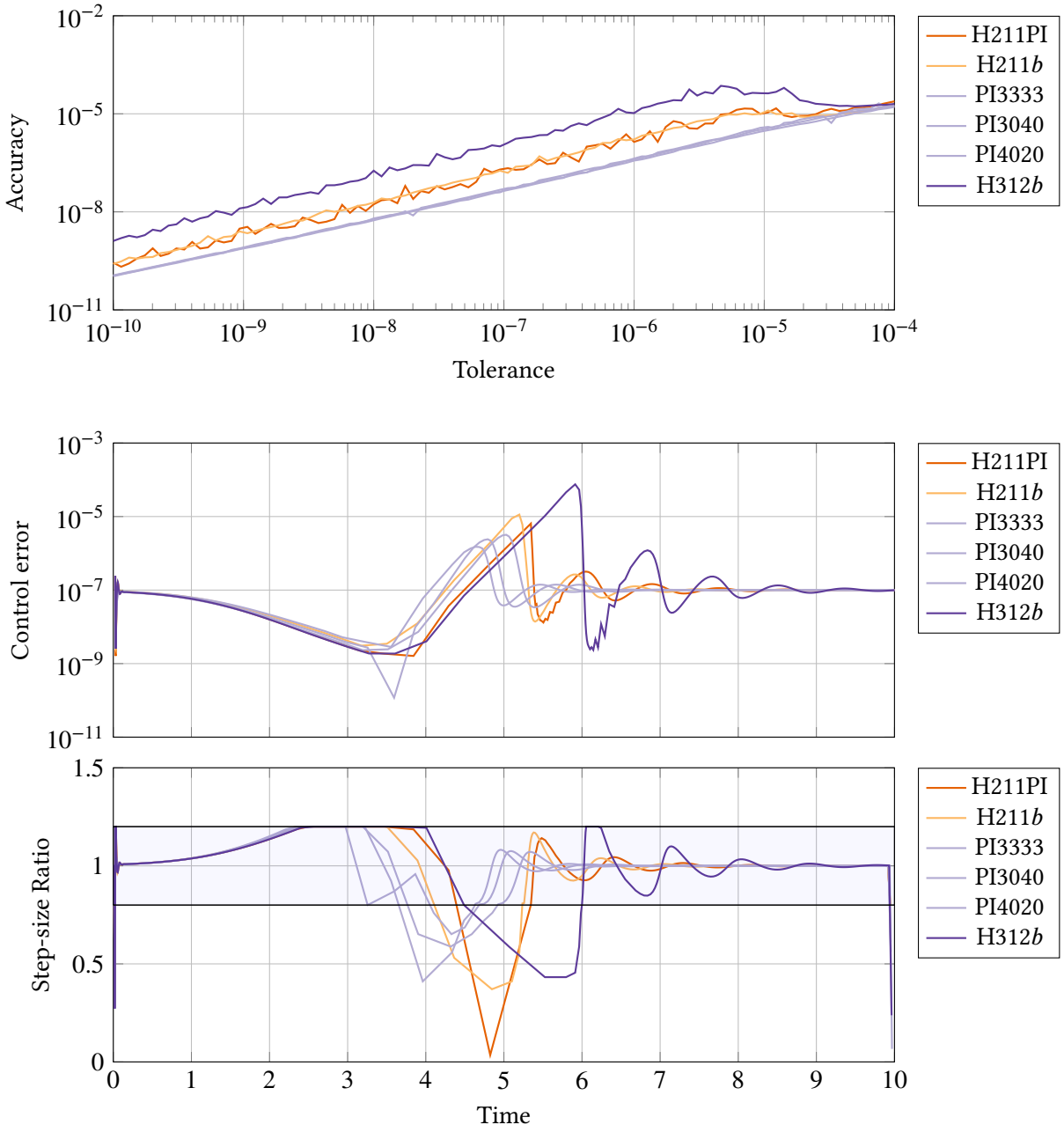

Figure III.55. Problem: The test equation with $d = 10$. Filters: H211PI, H211b, PI3333, PI3040, PI4020 and H312b. Methods: AB4. Stability region limit for AB4 is -0.3. The first figure shows the accuracy vs tolerance, the second one shows the control error as a function of time at TOL $= 10^{-7}$ and the third one shows the step-size ratio as a function of time at TOL $= 10^{-7}$. The filters PI3333, PI3040 and PI4020 behaves better than the rest. The same shade is used for these well behaving filters.

Part IV.

Construction and benchmarking of an order regulation algorithm

Author: Erik Jonsson-Glans

IV.1. Theory

The reason behind changing the order in a multistep solver is to decrease the amount of work needed to be done, and still be able to calculate a solution within a given precision. Below is a description of an algorithm used to change order, proposed by Söderlind [24], to which some modifications have been made.

IV.1.1. Order change for multistep methods

The main idea behind the algorithm for changing the order is to calculate how large a step can be taken if the order is increased or decreased, and then compare this to the step-size which can be used with the current order. If this comparison indicates a favorable outcome of changing the order, this is done. If no performance gain is predicted, the integration continues using the same order. In addition to the fact that a larger step-size decreases the work load, one has to factor in that a change in order also means a change in the number of calculations needed to advance the integrator one step, which is reflected in the algorithm by the inclusion of a work factor w_p.

When discussing the variable order solvers there will be multiple methods involved in each integration step: one which is used to perform the actual integration, from which we get the solution point that is output to the user, and two others which produce results only used to calculate the possible step-sizes which could have been used, if those were the methods performing the actual integration. To distinguish between these, the method which performs the actual integration will be called the *current order method*, and the other two will be called the *lower order method* and the *higher order method*.

The error produced by a method of order p is given by the error model

$$e_p = \varphi_p h_p^{\kappa} \tag{IV.1}$$

where h_p is the step-size used to compute the last step, and φ_p is a scalar dependent on the problem and the method used. κ will take different values, depending on how the local error is estimated. If error per unit step is used it will be $\kappa = p$, and if error per step is used $\kappa = p + 1$. During the calculations, the step-size filter will produce step-size sequences for the three methods mentioned above, and the index p in h_p denotes the order of the current order method.

Since only one step-size can be used to calculate the step, the error estimation for orders $p \pm 1$ will be given by

$$e_{p\pm1} = \varphi_{p\pm1} h_p^{\kappa\pm1} \tag{IV.2}$$

instead of

$$e_{p\pm1} = \varphi_{p\pm1} h_{p\pm1}^{\kappa\pm1}. \tag{IV.3}$$

This can be compensated for by multiplying with a factor

$$\bar{e}_{p\pm1} \equiv e_{p\pm1} \left(\frac{h_{p\pm1}}{h_p} \right)^{\kappa\pm1} = \varphi_{p\pm1} h_p^{\kappa\pm1} \left(\frac{h_{p\pm1}}{h_p} \right)^{\kappa\pm1}. \tag{IV.4}$$

This measure prohibits a *wind-up* of $h_{p\pm1}$ to an erroneous magnitude by making it appear as if a step-size $h_{p\pm1}$ has been used. By saving these results, it can also be used in the filter recursion if a change of order is made, which means that no restart of the control process is needed. With these error estimations, the error filter can calculate step-size suggestions for the orders p and $p \pm 1$.

The next step is to maximize the quantity h_p/w_p. This is normalized by the value obtained for the current order in use, and the relative advantage of changing order is then

$$\sigma_{p\pm1} = \frac{h_{p\pm1}/w_{p\pm1}}{h_p/w_p}. \tag{IV.5}$$

Changing the order directly based on this measurement though would lead to problems, like *chatter* (the repeated changing of order up and down), and therefore further processing of this quantity is done.

First, to measure the relative efficiency the weighted average order is defined (note that $\sigma_p = 1$ by definition) as

$$s_{p\pm1} \equiv \frac{(p \pm 1)\sigma_{p\pm1} + p\sigma_p}{\sigma_{p\pm1} + \sigma_p} = \frac{(p \pm 1)\sigma_{p\pm1} + p}{\sigma_{p\pm1} + 1}. \tag{IV.6}$$

This is a weighted mean, using the relative advantages as weights. This gives a natural way to express whether or not an order change is advantageous, by a quantity using order as the unit. If, for example, a change to the order $p + 1$ is advantageous (i.e. $\sigma_{p+1} > 1$), then $s_{p+1} > p + 1/2$. A complete summary of what $s_{p\pm1}$ indicates can be seen in Table IV.1.

Table IV.1. This table shows how the values of s_{p+1} and s_{p-1} indicate whether or not an increase or decrease in order is advantageous.

order change	if advantageous	if equally effective	if disadvantageous
$p + 1$	$s_{p+1} > p + 1/2$	$s_{p+1} = p + 1/2$	$s_{p+1} < p + 1/2$
$p - 1$	$s_{p-1} < p - 1/2$	$s_{p-1} = p - 1/2$	$s_{p-1} > p - 1/2$

It is desirable, though, to have a quantity which can easily be used as part of an integrator, which then makes the decision to change the order when its value reaches a certain threshold (this it to make the order changing algorithm robust and to avoid chatter). Therefore the fractional order increment is defined as

$$d_{p\pm1} \equiv s_{p\pm1} - p \mp \frac{1}{2}. \tag{IV.7}$$

From Table IV.1 we see that if $d_{p+1} > 0$, then an order increase is advantageous, and if $d_{p-1} < 0$ then an order decrease is advantageous. If one of these inequalities is reversed, this indicates that the current order is more advantageous than the one it is compared to. For example, if d_{p+1} is negative, the method order p is more advantageous than the one of order $p + 1$; it does not indicate anything about the advantage of changing to order $p-1$. Because of this, these quantities should only have an influence on the integrator if they belong to the intervals indicated by the inequalities above.

Therefore, the order increments due to a possible advantage of changing to a lower or higher order are calculated as

$$\delta p_+ \equiv \max(0, 4d_{p+1}),\ \delta p_- \equiv \min(0, 4d_{p-1}). \tag{IV.8}$$

By using the scaling factor 4, we have that for a small ε,

$$\sigma_{p\pm1} = 1 + \varepsilon \implies \delta p_\pm \approx \pm\varepsilon. \tag{IV.9}$$

This construction also ensures that $\delta p_\pm$ have upper bounds.

These measurements only compare the advantage of increasing or decreasing the order, over maintaining the current order in use. To compare if a change to order $p-1$ is more advantageous than a change to order $p+1$, the weighted average between the two is defined as

$$s_\pm \equiv \frac{(p+1)\sigma_{p+1} + (p-1)\sigma_{p-1}}{\sigma_{p+1} + \sigma_{p-1}}, \tag{IV.10}$$

the fractional order change is defined as

$$d_\pm = s_\pm - p, \tag{IV.11}$$

and the incremental order change is defined as

$$\delta p_\pm \equiv \begin{pmatrix} d_\pm, & \text{if } (\sigma_{p-1} - 1)(\sigma_{p+1} - 1) < 0 \\ 0, & \text{if } (\sigma_{p-1} - 1)(\sigma_{p+1} - 1) \geq 0 \end{pmatrix}. \tag{IV.12}$$

The last definition ensures that if it is advantageous to change to one order, but not the other, then $\delta p_\pm$ gets a value with the correct sign. If there is an advantage in both increasing and decreasing the order at the same time, then no clear decision can be made, and the incremental change is set to zero. This makes $\delta p_\pm$ act as a reinforcement when there is a clear advantage in choosing one order change over the other. No scaling factor is included in this case because if, for example, $\sigma_{p+1} = 1 + \varepsilon$ and $\sigma_{p-1} \in [1 - \varepsilon, 1]$, then the lower bound of $\delta p_\pm$ is approximately $\varepsilon/2$ and the upper bound is ε. A scaling factor would then, for some values of ε, give this comparison more influence than the main one, which is not desirable.

To avoid chatter the decision to change order is delayed, and multiple steps are needed to reinforce the decision to change order. To do this, the following integrator is used

$$\delta p_n = \delta p_{n-1} + (\delta p_+ + \delta p_- + \delta p_\pm). \tag{IV.13}$$

When $p + \delta p_n$ rounds to a different value than p (when $|\delta p_n| > 1/2$), the order is changed to $p + \text{sign}(\delta p_n)$. However, this integrating process may in some cases indicate an order change, even if the advantage measurements are inconclusive. To avoid this difficulty, an extra check is added, which is that the order is only changed to $p \pm 1$ if $\sigma_{p\pm1} > 1.1$. This heuristic value of 10 % is based on experiments [24].

The above way to calculate order increments assumes that a lower and higher order method exist all the time. There is an obvious limitation to the lower order, which is $p = 1$, and in practice one usually wants to set a higher limit as well (even if a convergent method exists). Therefore, special considerations must be made to deal with the cases when an increase or decrease of order is impossible.

In the case that an increase is impossible, the following variables are overridden and set to

$$\delta p_{+} = 0, \tag{IV.14}$$
$$\delta p_{\pm} = 0. \tag{IV.15}$$

In the case that a decrease is impossible, the following variables are overridden and set to

$$\delta p_{-} = 0, \tag{IV.16}$$
$$\delta p_{\pm} = 0. \tag{IV.17}$$

IV.1.2. Different step-size sequences

It should be noted that the step-size sequences discussed in this section are used to control the step-size and are fed into the step-size controller, but are not used to perform any actual calculations. There is a need for a fourth step-size sequence which corresponds to the steps between the time points of the calculated solution. To distinguish between these, the one used for calculations is called *the definitive step-size sequence*, and the other three used for step-size control are called *the referential step-size sequences*.

In the beginning, the definitive step-size sequence will correspond to the referential step-size sequence belonging to the current order, until an order change is made. When this happens, the previous part of the definitive step-size sequence remains the same, and the next step-size in this sequence will be the one previously proposed to the new order. If no additional order change is made after this, the part of the referential step-size sequence which is fed to the controller will eventually be equal to the corresponding elements in the definitive sequence. An example of this process can be viewed in Figure IV.1.

IV.2. Implementation

The basics of the implementation are the same as for the solver with a fixed order (see Section II.8), but some special considerations must be made which will be described in this section.

IV.2.1. The work factor

It has been mentioned that a work factor, w_p, is included when the relative efficiency is calculated. Because this thesis focuses on step-size regulation, and uses the amount of steps as a measure of work performed by the solver, this work factor has been set to 1 for all cases. Other work factors could be used, and one example could be to base the work factor on the amount of time it takes to solve the different systems of equations which are constructed when a step is integrated.

IV.2.2. Step-size rejection and large step-size ratios

When the step-size ratio proposed by the filter for any of the lower, current or higher methods is larger than the acceptable upper limit (set by the user), the ratio is replaced by this upper limit

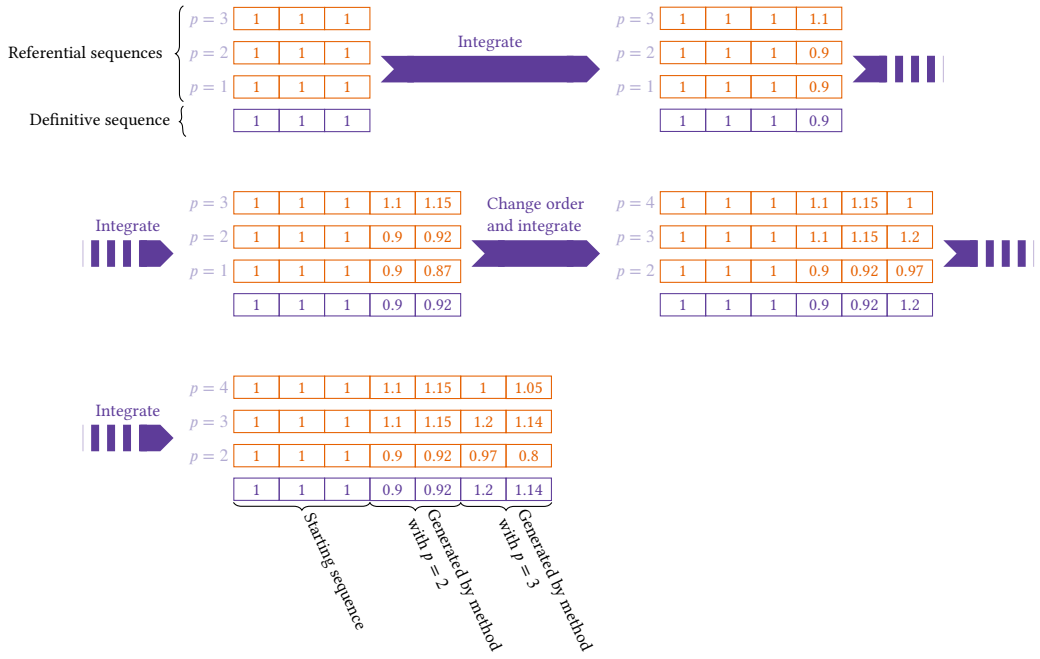

Figure IV.1. This figure shows an example of the relations between the different step-size sequences. All methods start with the same step-size sequence (this is a simplified example, and in practice the process will not be exactly the same, see Subsection IV.2.3) with all step-sizes set to 1, after which the solver integrates with the different methods, and the controller suggests new step-sizes for all three methods. Each time an integration is performed, the definitive step-size sequence is used (for all methods), and then the referential step-size sequences are used for each method to determine the new step-size. After 2 steps the order control system initiates an order change, and then the integration continues. In the end the definitive sequence is made up by different parts corresponding to different referential sequences. Note that if the same order is used long enough, then the part of the definitive sequence that is used to integrate the next step (the length of this part of the sequence depends on the order of the method) will be the same as the corresponding part of the referential sequence belonging to the current order method.

instead. In the case of the current order this is done so as to not introduce instabilities, and for the other two orders it is because otherwise there is a risk of overestimating the benefits of an order change. If, for example, the current and the higher order methods both originally had a step-size ratio of 3, and only the current order was capped to 1.2, it is clear that there is a great risk for the algorithm initiating an unwanted increase of the order.

When it comes to the step-size rejection, the current order method will be treated a bit differently compared to the lower and higher order methods, and this is because the current order method is the one performing the actual calculations.

A step is only rejected when the proposed step-size ratio for the current order method is lower than the acceptable limit (set by the user). When this happens, the control sequence is regarded as void, so the old error estimates are discarded for all three methods, and the main filter is bypassed in all three cases until sufficient new error data have been gathered. This is the same procedure as used in the fixed order solvers. Also, the order control mechanism is temporarily disabled until the step is accepted, after which it is resumed as normal.

If the lower or higher order method produces step-size ratios below the limit, nothing is done. There are no reasons to stop the integration and recalculate a step for these methods, because the results from them are not used in the next step. Also, keeping the small step-size ratio (and the large error which produced it) reinforces the message to the order change algorithm that it should not make a change to this method, which is desirable when it has shown to produce results with an unacceptable error.

IV.2.3. Startup phase

Just as in the case of the solvers using fixed order, the variable order solvers also suffer from the problem where the standard way of using the previously calculated polynomial to predict the next step is unavailable in the beginning of the integration. This problem is further increased by the fact that the higher order method is also affected by this (for the lower order method, though, there are enough points to construct a prediction polynomial).

Also, for the variable order case, the solver for which this problem is easiest to solve, is the solver using methods belonging to the class I_k. Just as in the case with the fixed order solver of this class, the corresponding explicit methods for the lower and current order are used to create prediction polynomials. After this, the order control system is disabled when solution point number $k+1$ is calculated (the previous solution points were calculated using the startup routine, see Subsection II.8.4), at which point the higher order method is started in the same way as the other two methods. Note that by doing this, the bypass counter for the higher order method is one less than the other two bypass counters. This will have the effect that the higher order method will start to use the correct filter one step after the lower and current order methods, but because the order control system is designed to not change order too rapidly, a difference of one step should not have a detrimental effect. The referential step-size sequence for the higher order method is also set to be the same as the definitive step-size sequence. Now all methods are enabled and solution point $k+2$ is calculated, after which the order control is started (see Figure II.8).

Solvers of class E_k or I_k^+ lack, as we have seen in Subsection II.8.4, the same natural way of starting the error estimation directly as solvers of class I_k, and therefore no step-size control

is enabled directly after the startup routine has calculated the initial points for the current and higher order methods. The difference though, compared to the fixed order case, is that there is also the lower order method which only needs $k-1$ steps to perform a calculation, and therefore a polynomial is constructed using the first $k-1$ points, which is then used in the usual fashion to predict the next step for this method. Because of this, step-size control is enabled directly for the lower order method. This will also enable the main filter for the lower order method one step before the main filter is activated for the current order method. Compare this to solvers of class $\boldsymbol{I}_k$, where the main filter for the current order method was activated one step before the higher order method; the motivations behind why this is acceptable are also the same in this case. Also in this case will the main filter for the higher order method be activated after it has been activated for the current order method. Table IV.2 contains a summary of when the main filter is activated for different methods. The reason for the delay of the higher order method is also the same as in the case of solvers of class $\boldsymbol{I}_k$: There is no method available at all after k solution points have been calculated. Just as in the case of $\boldsymbol{I}_k$-solvers, the order control mechanism is disabled until the error control has been enabled for the higher order method (a summary of this is found in Table IV.3).

Table IV.2. This table shows after how many calculated solution points the main filter is enabled, where p_D is the closed dynamic order (see Subsection III.1.1).

Solver class	lower order	current order	higher order
$\boldsymbol{E}_k$	$k+p_D$	$k+p_D+1$	$k+p_D+2$
$\boldsymbol{I}_k$	$k+p_D$	$k+p_D$	$k+p_D+1$
$\boldsymbol{I}_k^+$	$k+p_D$	$k+p_D+1$	$k+p_D+2$

Table IV.3. This table shows after how many calculated solution points the order control is enabled.

Solver class	Calculated solution points
$\boldsymbol{E}_k$	$k+3$
$\boldsymbol{I}_k$	$k+2$
$\boldsymbol{I}_k^+$	$k+3$

The next question is how the first Newton iterations are started. Solvers of class $\boldsymbol{I}_k$ just use the polynomials created by the corresponding explicit methods as initial guesses for all three orders. Solvers of class $\boldsymbol{I}_k^+$ though, use another scheme similar to the one described in Subsection II.8.4. This scheme consists of 5 steps:

1. A method of type $\boldsymbol{E}_k$ (in this case using the Adams–Bashforth method) is used to construct a polynomial of degree $k-1$. Let the coefficients be denoted $\{\hat{\boldsymbol{c}}_i\}_{i=0}^{k-1}$, where $\hat{\boldsymbol{c}}_0$ is the coefficient of the 0-degree term. This polynomial is then extended to a polynomial of degree k, by adding another term $\boldsymbol{c}_k$ which is initially set to $\boldsymbol{0}$.

2. The extended polynomial is then fed as an initial guess to Newton's method, when constructing the prediction polynomial to the lower order method. Let the resulting coefficients of the prediction polynomial be denoted $\{\tilde{\boldsymbol{c}}_i\}_{i=0}^{k-1}$.

3. The coefficients $\{\tilde{\boldsymbol{c}}_i\}_{i=0}^{k-1}$ are now used as an initial guess for the lower order method.

4. To get an initial guess for the current order method, the prediction polynomial is then extended to a polynomial of degree k, by adding another term $\tilde{\boldsymbol{c}}_k$, which initially is set to $\mathbf{0}$. Let the resulting coefficients of the current order polynomial be denoted $\{\boldsymbol{c}_i\}_{i=0}^{k}$.
5. Lastly, an initial guess for the higher order method is gotten by extending the coefficients from the previous step by adding a term, $\boldsymbol{c}_{k+1}$, and initially setting this to $\mathbf{0}$.

A flow chart depicting this process can be found in Figure IV.2.

In the special case where the initial order is the lowest order (as is the case when starting with a one step method), instead of using the coefficients of the prediction polynomial of the lower order method and extending them, Adams–Bashforth of order k is used to produce a polynomial which is extended in the same fashion and fed to the Newton iteration of the current order method.

Some final details should be mentioned in regards to the startup phase. Firstly, in Subsection II.8.4 it was mentioned that one way of starting a multistep method is to start with a one step method, and then increase the order. A version of this can be had with this implementation by starting with an initial order corresponding to a one step method. Secondly, if the initial order is the highest possible order, then one could enable the order control one step earlier. The choice to not implement this is a choice of convenience and to have one less special case to deal with when programming the solver.

IV.2.4. Handling order change

When an order change is initiated, one of the previously used methods will disappear and a new method will be introduced. If the order is increased, then the previously used lower order method will no longer be used and the previous current order method will take its place. The place of the previous current order method is taken by the previous higher order method and a new higher order method is introduced. The opposite is then done if the order is decreased instead. This means that a new method, without error estimations, referential step-size sequence and a predicting polynomial must be handled.

To deal with the lack of error estimations the main filter is just bypassed until the appropriate number of error estimates has been gathered (compare this with the startup phase).

When it comes to the missing previous referential step-size sequence, what is done is that the old referential step-size sequence, belonging to the method which is replaced, is kept. So, for example, directly after an order increase, the referential step-size sequences of the current and higher order methods will be the same.

Lastly, a predicting polynomial must to be procured, so that a new error estimate can be calculated for the first step after the order change. The solution to this is to use the solution points previous to the one most recently calculated, and to construct a polynomial using the new method as if it had actually been used to calculate the most recent solution point. This polynomial can then be used to predict the next step, and this is implemented for solvers of class E_k. The fact that the order control is delayed, guarantees that there will always be enough solution points to do this.

Solvers belonging to one of the implicit classes have an even easier solution to the missing prediction polynomial. Here there are corresponding explicit methods which can be used to construct a polynomial, and this is also how it is done for the implicit solvers. This makes for both an

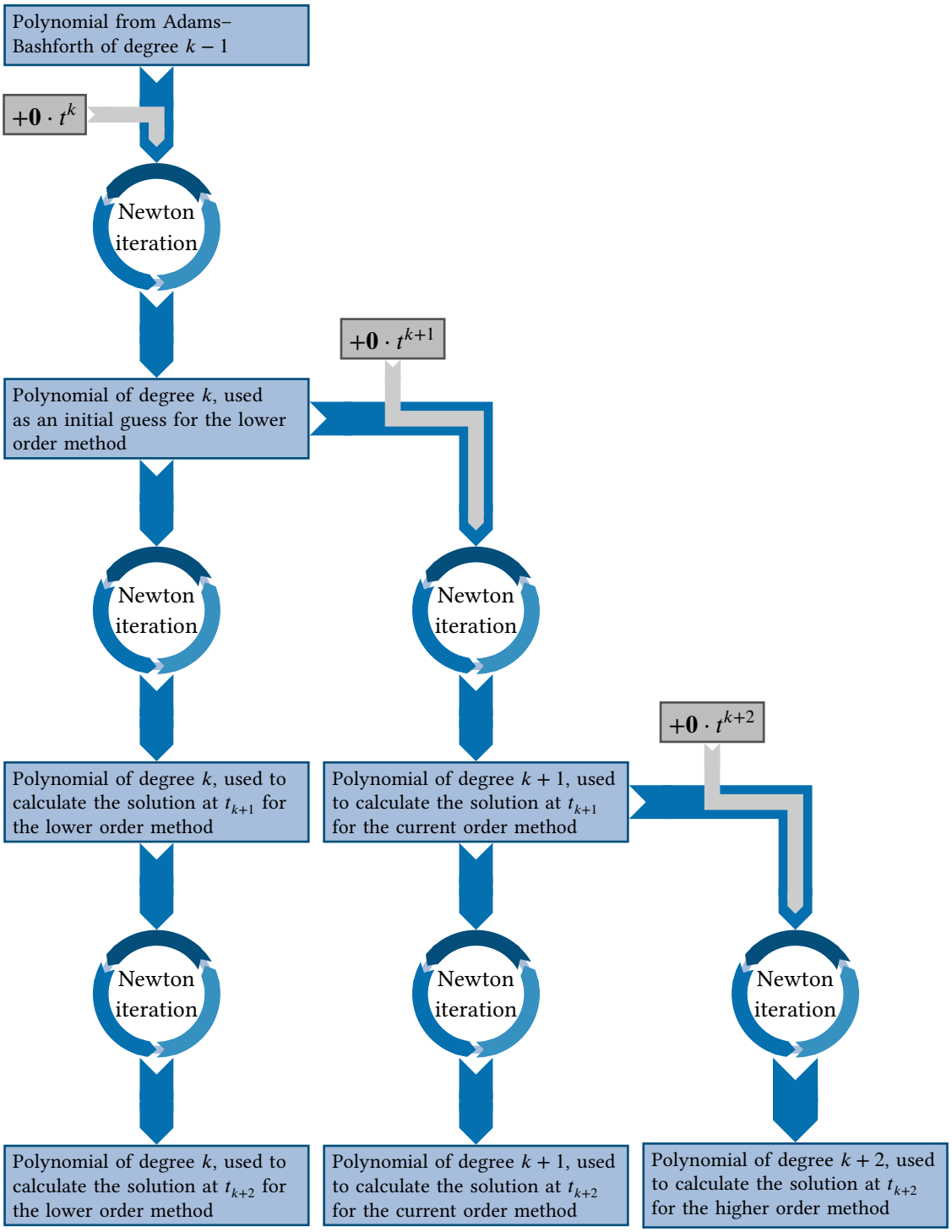

Figure IV.2. This flow chart illustrates how `pmmipVarOrd` initiates the Newton iteration processes for the methods of different order. The basic principle is, that when no polynomial of high enough degree is available, a polynomial of a lower degree is used by adding a higher order term for which the coefficients are set to $\mathbf{0}$. This is indicated by the small boxes and arrows in the chart. Each large box represents a polynomial, and its purpose is described by the text in the box.

easier implementation, and a very small performance gain, because there is no need to solve a non-linear system of equations.

IV.2.5. A note on the explicit methods corresponding to the prediction polynomials of the implicit methods

Throughout this thesis, the use of an explicit method in accordance with Theorem II.6.1 and II.6.2 have been treated as equivalent with using the previously calculated polynomial of the implicit method in use. This treatment is correct, but only under the assumption that the method used is also the method which calculated the previous solution points.

The problem can be seen if one studies the proofs of the theorems. If another method is used to calculate the most recent solution point, then the slack condition of the explicit method is not fulfilled for the previous polynomial constructed with an implicit method. This is not an issue when only one order is used, but in the variable order case, only the current order is also the one which produces the solution points which are saved. To make this clear, let $\boldsymbol{P}_m(t)$ be the polynomial constructed by the current order method, and $\hat{\boldsymbol{P}}_m(t)$ be the polynomial constructed by the lower or higher order method, at step number m. The solution points produced by these methods are then

$$\boldsymbol{x}_m = \boldsymbol{P}_m(t_m), \tag{IV.18}$$

$$\hat{\boldsymbol{x}}_m = \hat{\boldsymbol{P}}_m(t_m), \tag{IV.19}$$

where only $\boldsymbol{x}_m$ is saved. Now, if one wants to predict the next step, $n = m + 1$, then $\boldsymbol{P}_m(t)$ would also be the polynomial produced by the corresponding explicit method, and we can specifically see that the slack condition is fulfilled (as can also be seen in the proofs of Theorem II.6.1 and II.6.2) by $\boldsymbol{P}_m(t)$

$$\boldsymbol{s}_{n-1} = \boldsymbol{x}_{n-1} - \boldsymbol{P}_m(t_m) = \boldsymbol{x}_{n-1} - \boldsymbol{x}_m = \boldsymbol{x}_{n-1} - \boldsymbol{x}_{n-1} = \boldsymbol{0}. \tag{IV.20}$$

If one then tries to see if the same is true for the other polynomial (constructed by a method of different order), one gets

$$\hat{\boldsymbol{s}}_{n-1} = \boldsymbol{x}_{n-1} - \hat{\boldsymbol{P}}_m(t_m) = \boldsymbol{x}_{n-1} - \hat{\boldsymbol{x}}_m = \boldsymbol{x}_{n-1} - \hat{\boldsymbol{x}}_{n-1}, \tag{IV.21}$$

which is not necessarily equal to $\boldsymbol{0}$ (remember that only $\boldsymbol{x}_{n-1}$ is saved, so an explicit method would use this value for calculations in both cases).

When this case appears in the implementation no corrections are made, and this is because the assumption that

$$\boldsymbol{x}_{n-1} \approx \hat{\boldsymbol{x}}_{n-1} \implies \hat{\boldsymbol{s}}_{n-1} \approx \boldsymbol{0} \tag{IV.22}$$

is made.

IV.3. Results and discussion

Different types of benchmarks have been used to evaluate how the order changing algorithm performs. To minimize the risk of the method itself performing poorly, and thereby obscuring

the performance of the order change algorithm, one family of methods which has already been well studied in the literature was chosen for each type of solver. The families in question are Adams–Bashforth, Adams–Moulton, and BDF. Only one filter, PI3333, which was deemed to be generally well behaved was chosen, as the evaluation of filters has been done in Part III. The other settings used (when applicable) are chosen to be the same as in Part III to make comparison between fixed and variable order solvers easier. All settings used can be seen in Table IV.4.

An upper order limit of 5 was chosen for `pmmiVarOrd`, which uses the BDF-family of methods. There are no BDF-methods which are zero stable with an order above 6, and order 6 has a very small stability region. In the cases of the other two solvers, both families of methods used (Adams–Bashforth and Adams–Moulton) have no upper limits for which the methods stop being zero stable. The upper limits here are chosen to be 8, and this is because of performance issues, which will be discussed in detail in Subsection IV.3.5. The lower limit is chosen to be the lowest possible for all three solvers, and except for `pmmipVarOrd` this is also the initial order used. This removes the need of using another method to generate steps in the beginning (as the lowest order methods are all 1-step methods, and the initial condition is then enough). There are some problems, discussed below in Subsection IV.3.1, unrelated to the order changing algorithm when starting `pmmipVarOrd` with order 2, and as this is primarily an evaluation of the algorithm for changing order, the choice was made to bypass this problem by using order 3 instead.

Table IV.4. Settings used for benchmarking with the variable order solvers

Setting	`pmmeVarOrd`	`pmmipVarOrd`	`pmmiVarOrd`
Error per unit step	false	false	false
Relative error	false	false	false
Filter	PI3333	PI3333	PI3333
$r_{\min}$	0.8	0.8	0.8
$r_{\max}$	1.2	1.2	1.2
Family of methods used	Adams–Bashforth	Adams–Moulton	BDF
Minimal order	1	2	1
Maximal order	8	8	5
Initial order	1	3	1

The set of problems used is almost the same as the one used when evaluating the filters, also for the reason of making comparisons easier. The differences are that a moderately stiff problem, the flame propagation problem with the parameter $\alpha = 2$ (see Subsection III.3.13), was added because it showed some especially interesting behavior when the order was varied. Also, the Van der Pol problem was used with the parameter $\mu = 10$ (see Subsection III.3.8), which makes it a bit stiffer and also increases the integration time, making the solution contain more periods. This change was introduced to be better able to study the periodic behavior of the solvers in this case.

In all cases, except for the decaying exponent and flame propagation problems, there are no analytical solutions and therefore the reference solution, $\boldsymbol{x}^{\text{ref}}$, had to be calculated, and to do this the solvers `ode113` and `ode15s` were used with very strict tolerance settings (see Table IV.5). All settings except the tolerances were used with their default values. What solver was used for each problem can be seen in Table IV.6.

The first set of benchmarks consists of solutions to the problems calculated with 3 different tol-

Table IV.5. Settings used for `ode15s` and `ode113` when calculating a reference solution

Setting	Value for `ode15s`/`ode113`
absTol	10^{-16}
relTol	10^{-13}

Table IV.6. The method used to calculate the reference solutions for different problems. Settings for `ode15s` and `ode113` can be found in Table IV.5

Problem	Reference solution
Decaying exponent ($d = 1$)	Analytical solution
Van der Pol ($\mu = 10$)	`ode113`
Lotka–Volterra ($c = 2$)	`ode113`
Brusselator	`ode113`
Flame propagation ($\alpha = 2$)	Analytical solution
Decaying exponent ($d = 100$)	Analytical solution
Van der Pol ($\mu = 100$)	`ode15s`
Robertson	`ode15s`
Oregonator	`ode15s`
HIRES	`ode15s`

erances. Here the actual solution is shown (sometimes only one tolerance though, because the difference between the results of the various tolerance settings are very small) in order to make sure that the solvers produce reasonable solutions, and so that the reader may see how the actual solution looks like. In addition to this the error, step-sizes and orders used throughout the integration processes are shown. This gives a detailed view of how the solver behaves during calculations.

The next set of benchmarks consists of 100 test runs (for each problem) with tolerances spaced equidistantly when plotted using a logarithmic scale. The achieved accuracy is then measured by taking the discrete L_1-norm of the error throughout the solution (for details on how this is done see Equation III.171 and III.172). The amount of work is then measured by counting the number of successful steps used to produce the solution. Lastly, the mean order used is measured by integrating the order used throughout the solution according to the following formula

$$\bar{o} = \sum_{i=k_0}^{n} \frac{o_i \cdot h_{i-1}}{t_f - t_0} \tag{IV.23}$$

where o_i is the order used to calculate $\boldsymbol{x}_i$, k_0 is the amount of steps needed for the starting order, and n is the total number of steps. This gives a high level view of the order preferences of the solvers when using different tolerances.

The last set of benchmarks are comparisons between our solvers and two established solvers. `pmmipVarOrd` is compared to `ode113` and `pmmiVarOrd` is compared to `ode15s`. The choice of these methods are based upon the fact that they use (or can be set to use) the same underlying families of methods as the ones used to benchmark our solvers. `ode113` uses Adams–Moulton methods [16] and `ode15s` uses BDF [17]. It is important to note here that when speaking about

the underlying families of methods being the same, this means that when using a fixed step-size they will produce the same α- and β-coefficients used to define the methods in Equation II.2. When these methods are extended to use a variable step-size they are no longer necessarily the same.

By setting the absolute tolerance of `ode113` and `ode15s` to the one as used by our solvers, and the relative tolerance to 10^{-30}, `ode113` and `ode15s` in practice use absolute error to control the step-size. This is equivalent to measure the absolute error in the $\| \bullet \|_\infty$-norm and using this to control the step-size (which is done in our solvers, but the step-size control and rejection mechanism works differently than in Matlab) [19]. The purpose of this set of benchmarks is to see whether or not our solvers seem to have the potential to be competitive against solvers widely used in practice.

IV.3.1. The choice of initial order for `pmmipVarOrd`

As has been previously mentioned there are some performance issues when starting `pmmipVarOrd` with the lowest possible order. The problem is mainly that the achieved accuracy is poor, and that there are a lot of rejected steps. This is only a problem though if the solver chooses the initial step-size, by using its step-size estimation algorithm. If the user chooses an appropriate step-size both of the problems mentioned disappears. If one would look at where these step-size rejections take place, one sees that they are at the beginning of the integration process. An example, using Lotka–Volterra as a test problem, showing how the accuracy and number of rejected steps are affected by these settings when solving with different tolerances can be seen in Figure IV.3. This figure also shows that when using an initial order of 3 these problems do not appear.

The fact that the majority of the rejected steps are in the beginning of the integration process, and that setting a new appropriate initial step-size manually solves this issue, makes for the conclusion that there is a problem in the startup process of the solver. One probable explanation for this is that the step-size estimation algorithm is unable to propose a good initial step-size for the trapezoidal rule (Adams–Moulton of order 2), and then because the initial step is never rejected (due to the fact that the step-size regulation has not yet been enabled, see Subsection II.8.4 and IV.2.3) an unacceptably large local error is introduced, which then affects the accuracy for the rest of the solution. Note that this is not a problem in all cases, but enough test cases exhibit this behavior so that it should be dealt with.

For testing purposes, finding appropriate initial step-sizes for all different cases is not practical, but as can be seen in Figure IV.3, starting with an initial order of 3 is a simple solution to this problem and therefore this is done. Another effect this has is that the amount of steps taken decreases somewhat, which is not at all a strange behavior, because when starting at a higher order, the initial step-size in most cases can be allowed to be larger.

IV.3.2. Single runs

If we look at the results from `pmmeVarOrd`, which can be seen in Figures IV.4-IV.8, we see that the solver produces good solutions, and that it actively changes order. The error curves for a problem are similar to each other with the main difference being that they are shifted vertically, with the

Performance differences for pmmipVarOrd, when different startup settings are used for solving the Lotka–Volterra problem

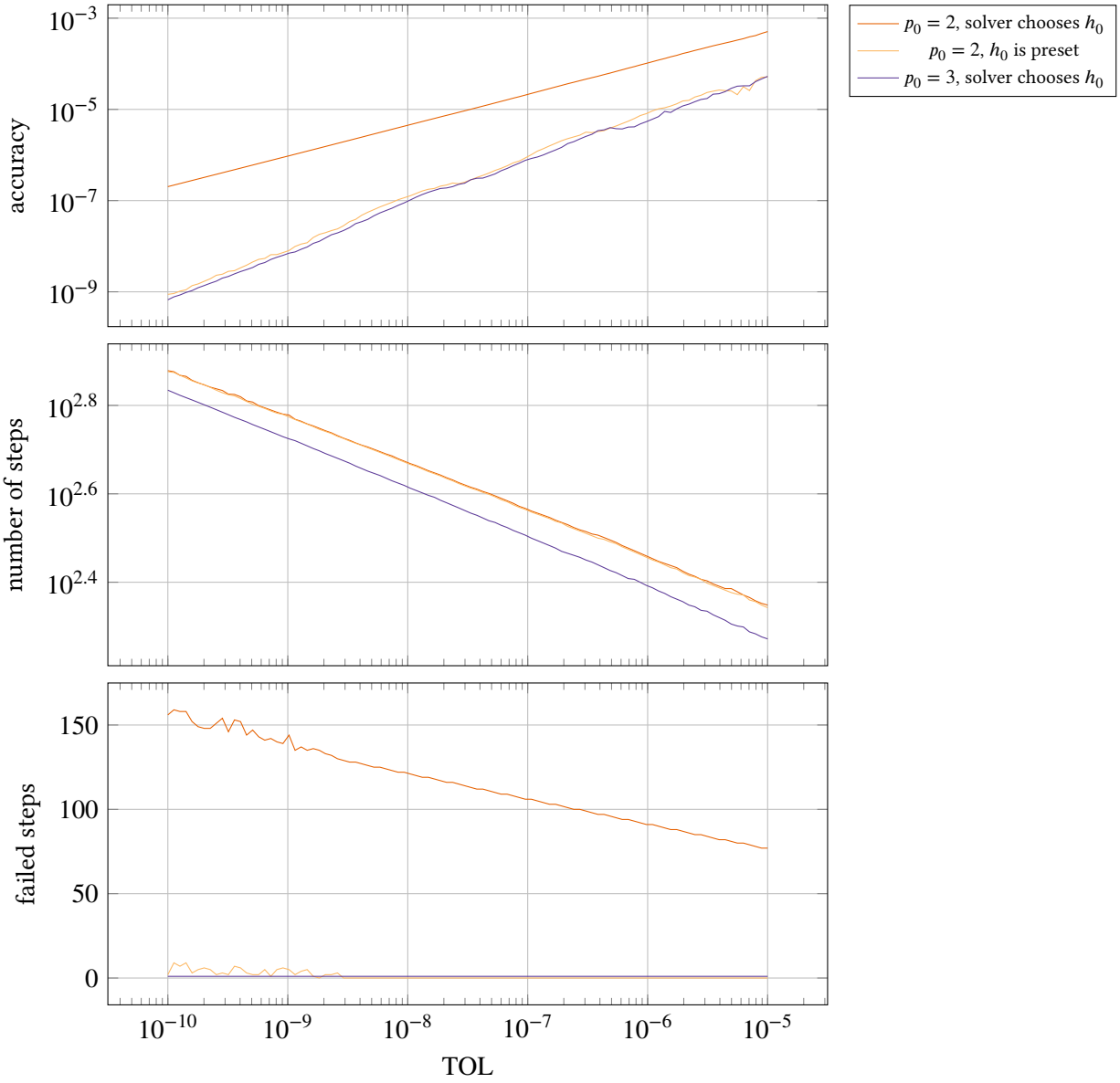

Figure IV.3. This shows the differences in performance when using different startup settings for `pmmipVarOrd`, and solving the Lotka–Volterra problem (with $c = 2$). In all cases are Adams–Moulton methods of order 2–8 used, but the initial order, p_0, is different. There is also a difference in how the initial step-size, h_0, is chosen. In two of the cases the solver uses its step-size estimation algorithm, but in the other case the step-size is preset to an appropriate value. Except for these changes, the settings are the same as shown in Table IV.4. Note how the accuracy is almost the same for the curve generated by using $p_0 = 3$ and the one where h_0 is preset. Compare this to the curve generated by the method with $p_0 = 2$ and where the solver chooses h_0. The accuracy when using these settings is much worse, and there are many more failed steps.

most noteworthy examples being Figure IV.5, IV.6 and IV.7. This is very desirable, because it will increase the possibilities of the solver being tolerance proportional and computationally stable.

In the two periodic problems, Lotka–Volterra and Van der Pol (Figure IV.6 and IV.7), we can see that both the error and the orders used have very periodic behaviors; this is expected as there is no reason for the solver to start behaving differently from period to period as it is essentially the same problem solved over and over again. Each period should also be more similar to the previous one in the later parts of the solution, and this is mainly because the solver has no influence over the order for the initial step, as this is set by the user who will probably not choose the same order as the solver uses after it has gained control. Also, the way the initial step-size is chosen, will contribute to the effect of the first periods being less similar to each other. Nevertheless, in our case, the behavior is very similar even from the beginning, and therefore this effect is not very noticeable. The step-size sequences also exhibit periodic behaviors, although this is much less prevalent than in the case of the behaviors of the order changes and the errors.

When comparing the variable order solver to the fixed order equivalent, one very noticeable difference is that the step-size sequences for the variable order solver are much less smooth than the ones which are produced by fixed order methods. This is especially apparent when comparing the sequences for Van der Pol in Figure IV.7 and Figure IV.9. Not only is there a difference in smoothness, but the curves have almost no similarities. The main similarity is that the step-size decreases rapidly before $t = 10, 20, 30, 40$.

That the variable order solver produces less smooth step-size sequences is expected, as the main point of the algorithm is to change to another order when there is a significant decrease of the work needed to compute a step, which in our case is measured by the length of the step, and therefore it promotes sudden and rapid increases in the step-size. In the case of fixed order solvers this would be a very undesirable behavior, as it increases the risk of the solver becoming unstable and also the risk of the error to deviate violently from the desired value. In this case, though, it should not pose a serious problem, because one of the main features of the algorithm employed for changing the order is that it should predict the behavior of the orders not currently in use, and regulate them so that they are stable and produce solution points with the correct accuracy. A good indication showing that this is working can be seen in Figure IV.7. Directly after $t = 10$, the solver using $\mathrm{TOL} = 10^{-10}$ changes order multiple times in short order, but still the error curve at this interval is extremely smooth. Another indication, which can be seen in the same figure (and other figures as well), is the fact that even though the order change is not initiated at the same time when using different tolerances, the shapes of the error curves are still very similar to each other.

When looking at the results from `pmmipVarOrd` (Figure IV.10-IV.14) one of the main differences from the results produced by `pmmeVarOrd` is that, except for the cases of the Van der Pol and flame propagation problem (Figure IV.13 and IV.14), is that the order mainly increases throughout the problem. This is an indication that in these cases a higher order is performing better throughout the whole problem. The fact that the two problems which are moderately stiff, and for which the stiffness has large changes throughout the problem, are showing changes up and down in order supports the conclusion that the algorithm is working as intended. How the stiffness varies in these problems can be seen in Figure III.38 and IV.15. Note that the stiffness values in these figures are not normalized, and that the peak magnitude of the normalized stiffness values can be seen in Table III.7. A largely varying stiffness should have an influence on which method is to be preferred, as it greatly changes the stability properties of the integration process (and in

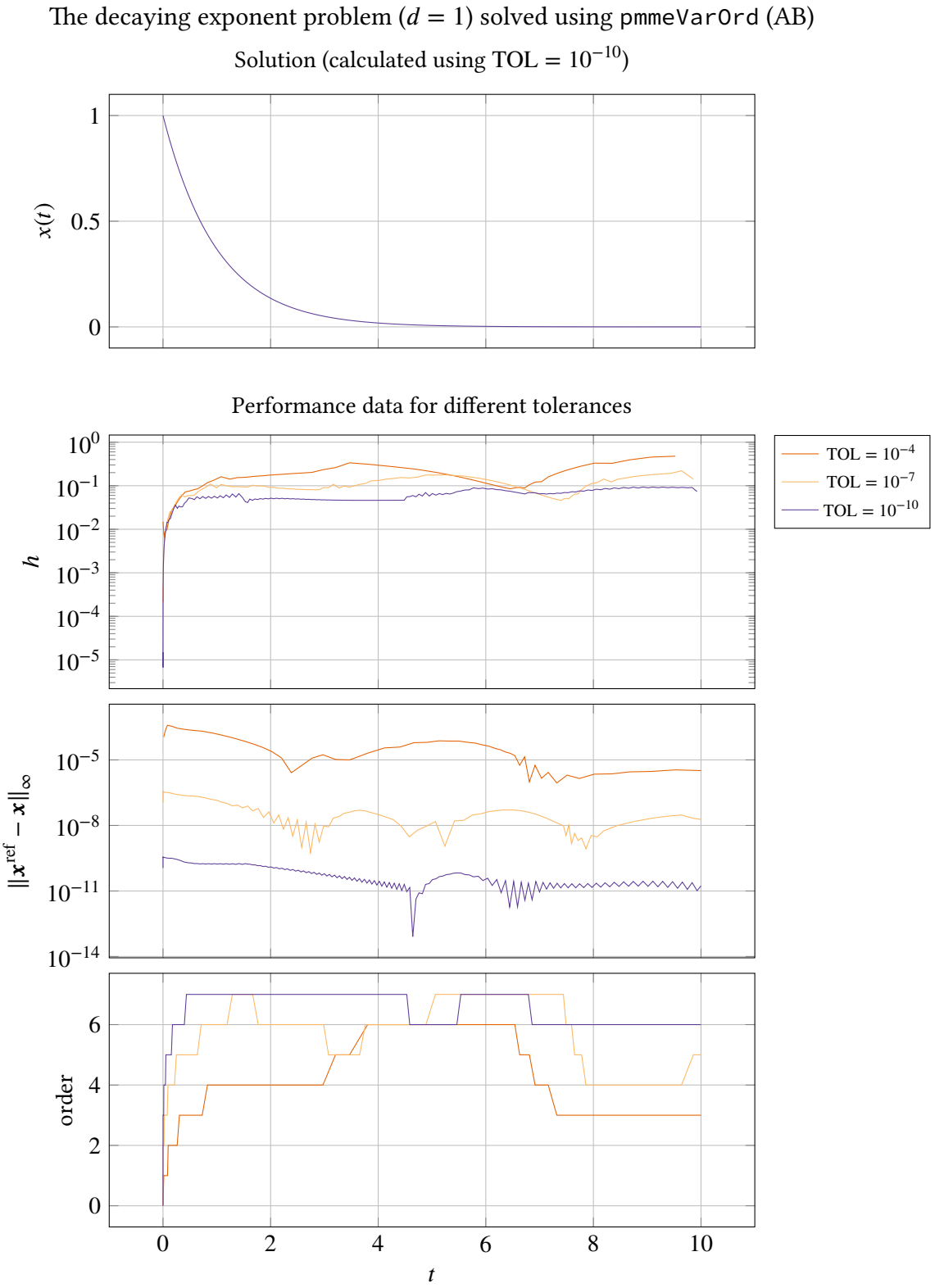

Figure IV.4. The decaying exponent problem solved by using `pmmeVarOrd` with 3 different tolerance settings. Note how the order increases more rapidly for stricter tolerances.

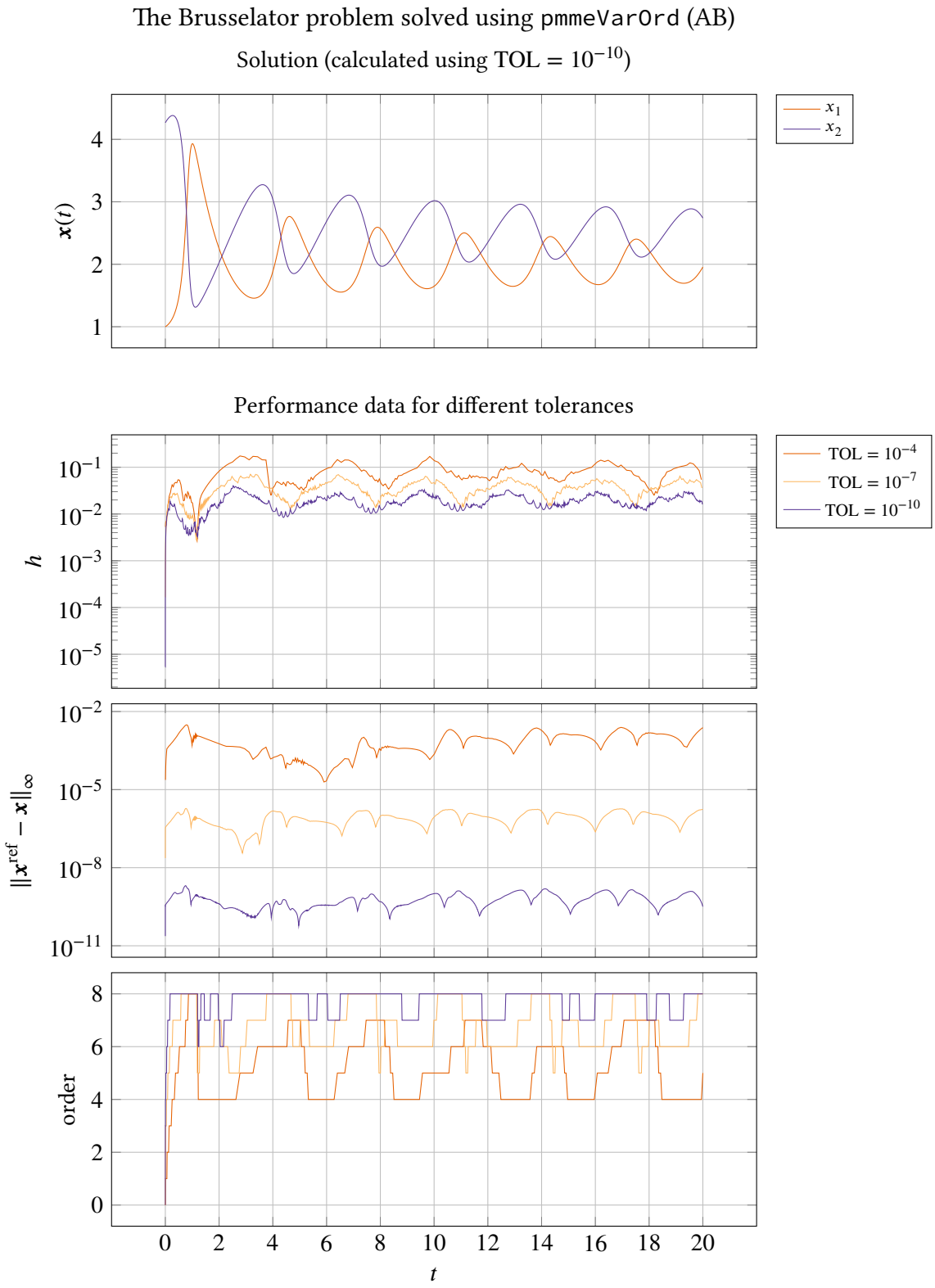

Figure IV.5. The Brusselator problem solved by using `pmmeVarOrd` with 3 different tolerance settings. Note the similarities of the shapes of the error curves, and that they are shifted vertically.

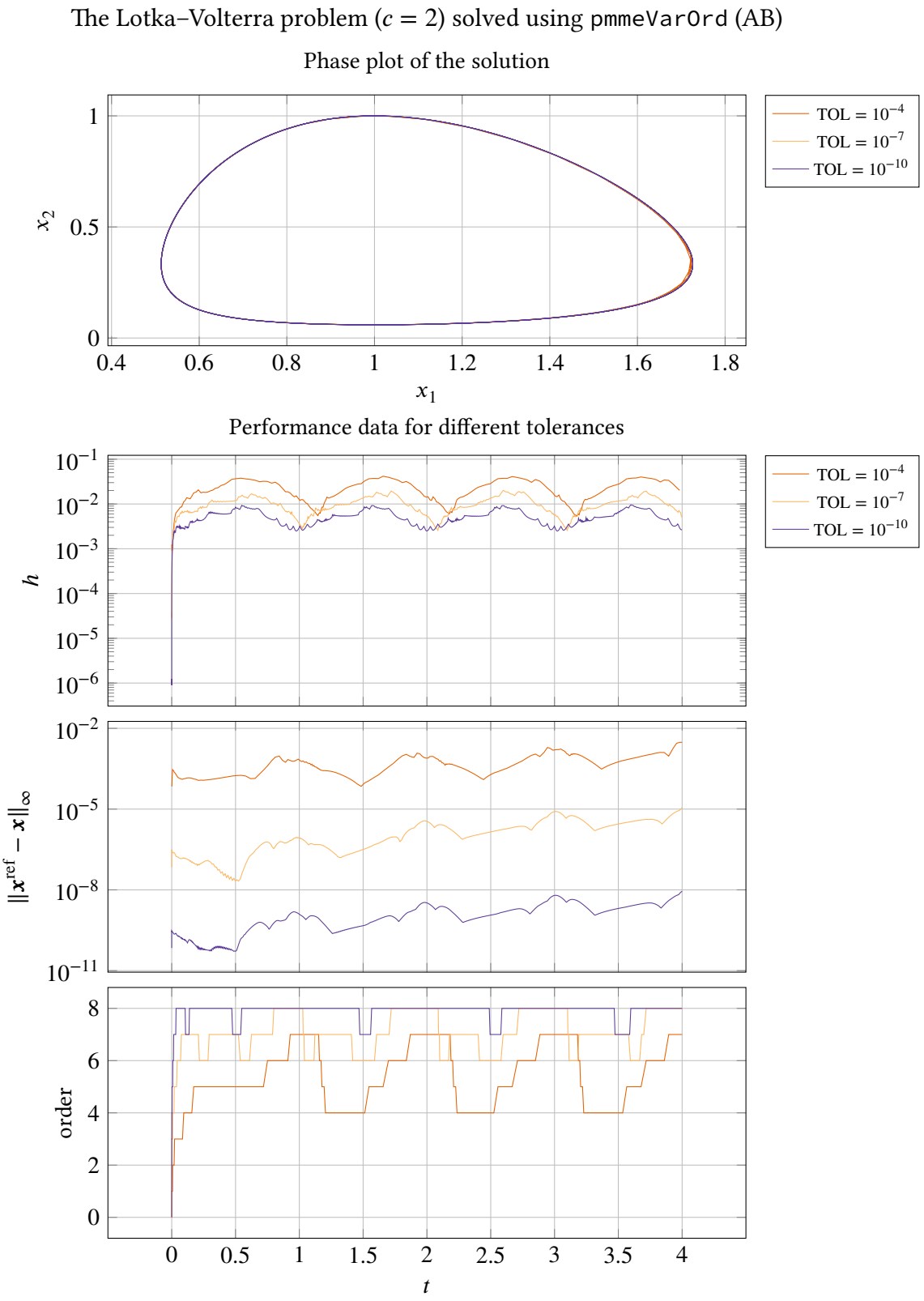

Figure IV.6. Lotka–Volterra problem solved by using `pmmeVarOrd` with 3 different tolerance settings. Both the order changes and the errors exhibits periodic behaviors. In the error we can see a small drift-off effect.

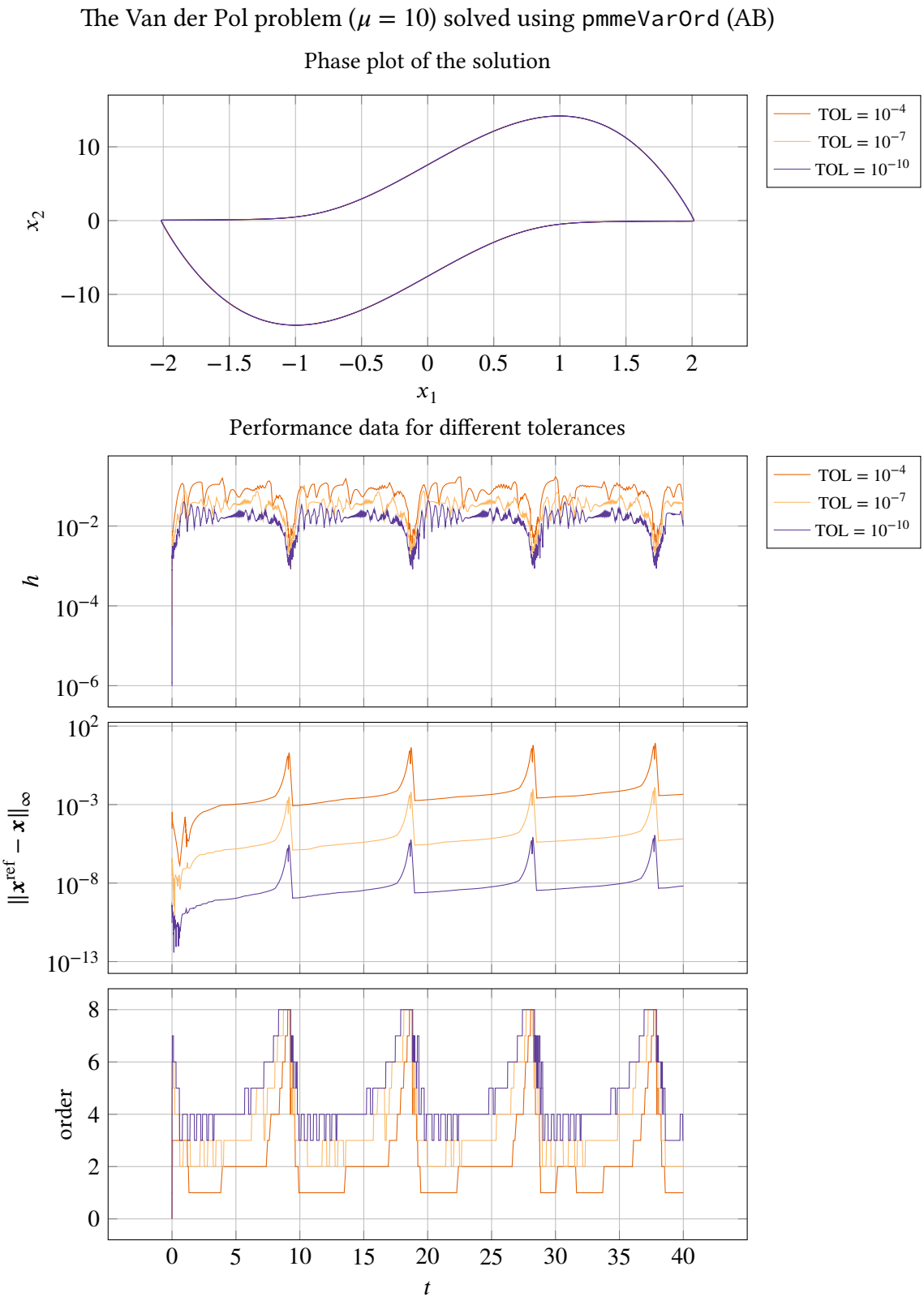

Figure IV.7. The Van der Pol problem solved by using `pmmeVarOrd` with 3 different tolerance settings. Both the order changes and the errors exhibits periodic behaviors. In the error we can see a small drift-off effect, although this is not visible in the phase plot. Note that even though the step-size sequence is not smooth, the shapes of the error curves are very smooth and similar to each other.

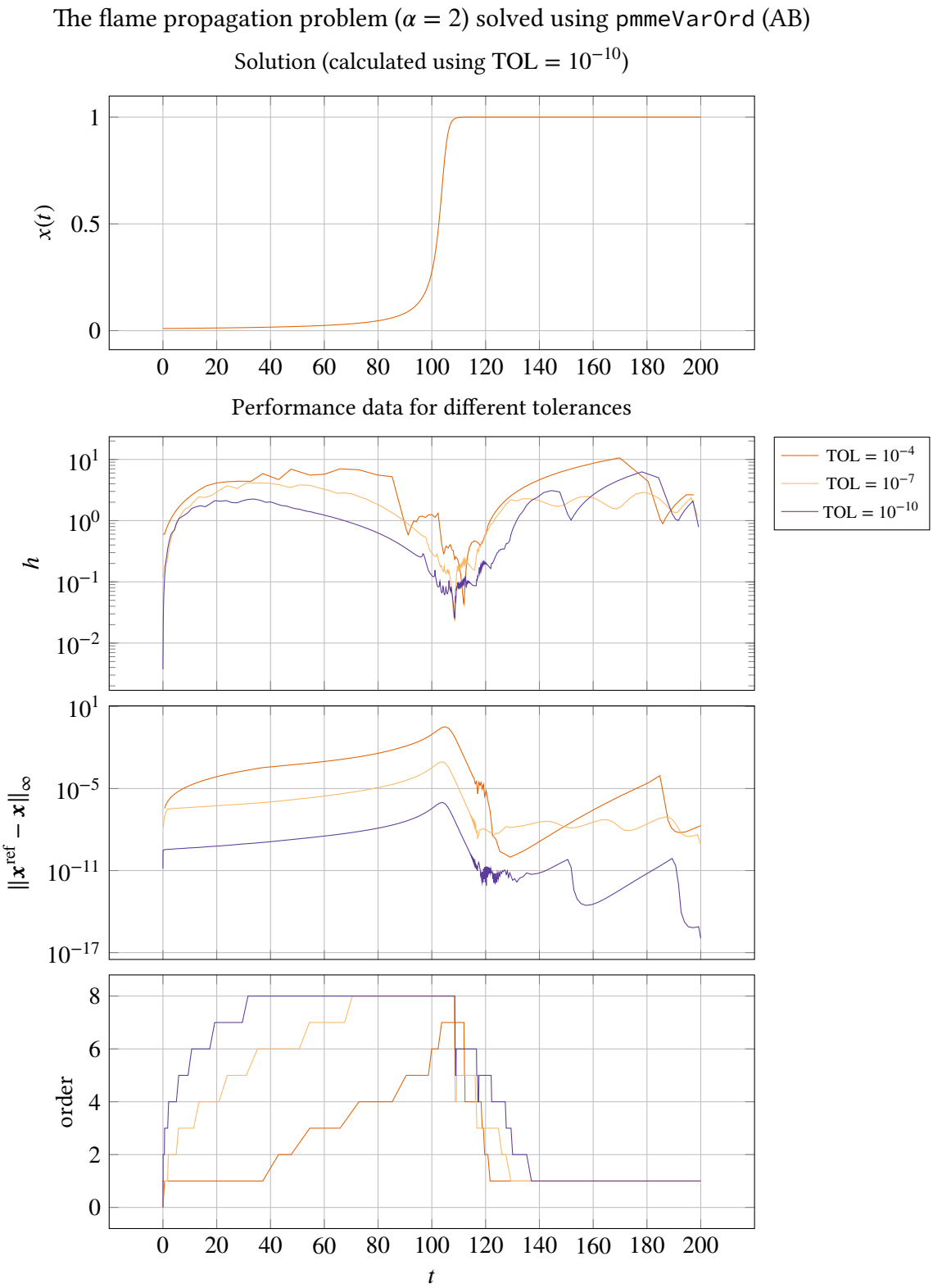

Figure IV.8. The flame propagation problem solved by using pmmeVarOrd with 3 different tolerance settings. When the problem becomes stiff (for a detailed view of the stiffness behavior of this problem see Figure IV.15) the order decreases down to the lowest possible order. Here the regulation of the error also becomes more difficult, and we can see that the shape of the error curves are starting to become dissimilar.

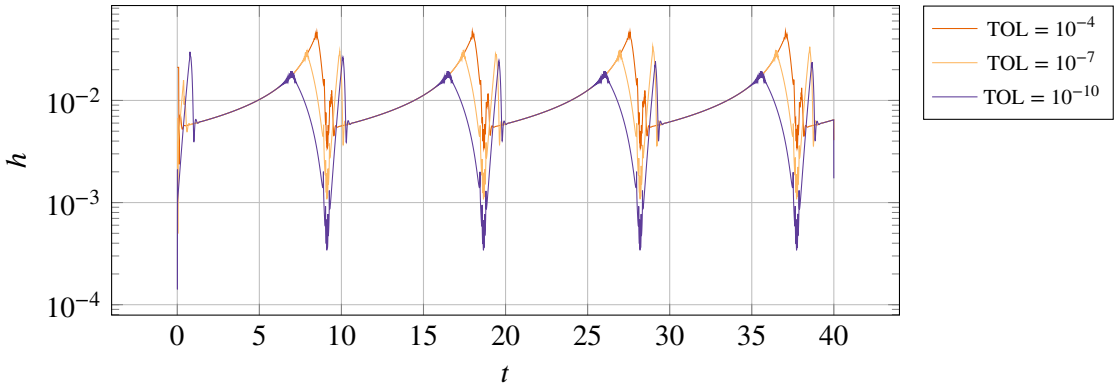

Figure IV.9. This plot shows the step-size sequences produced by the fixed order solver pmme, when solving the Van der Pol problem with $\mu = 10$. pmme uses Adams–Bashforth of order 5, and the same settings (except for the ones regarding order) as shown in Table IV.4. Note how the step-size sequences are smoother for the fixed order solver than the ones produced by the variable order solver in Figure IV.7, and that the sequences have few similarities to each other.

their fixed order versions, they have greatly varying stability region, which is something that can be expected to influence the stability even in the variable order case), and in these cases we see that at least the order is changing actively. The exception to this is in the flame propagation problem, where the order curve for TOL $= 10^{-7}$, shows that the order does not change. This is an unwanted behavior, and a problem which is discussed in Subsection IV.3.5. The periodic behavior of the order changes in the Van der Pol problem is also a good indication that the algorithm is working as expected (in the case of the other periodic problem, Lotka–Volterra, the behavior when using the two stricter tolerances is of course also periodic but constant, which unfortunately does not give as much information about the performance of the algorithm).

The last set of results in this section are from the tests using `pmmiVarOrd` to solve stiff problems (Figure IV.16-IV.20). Here the results are also very good. The error curves show the same behavior as previously mentioned: they have the same shape but are shifted. Two noticeable exceptions to this are the decaying exponent and Robertson problems (Figure IV.16 and IV.19). In the case of the decaying exponent the explanation is that the error becomes very small quickly, and is essentially 0 throughout most of the solution, and the regulator will have problem regulating at such small errors. In the majority of the solution the error is also so small, that we expect the limitations of floating point numbers to have an influence over the process. These are not problems with the algorithm, and we can see that in the beginning where the error is larger it performs as expected. In the case of the Robertson problem, the accuracy is not proportional to the tolerance, which can be seen by two the error curves crossing each other, and this is due to the performance of the solver. This is a particularly hard problem, and comparisons to another well established solver will be made later, that will give some context to these results.

Lastly, note that in the two periodic problems, the Oregonator and Van der Pol problems (Figures IV.18 and IV.20), the behavior of the order changes are periodic. This may not be obvious in the case of the Oregonator problem, as the periods are far apart and the solver uses the highest order for the most part, but if one looks in the vicinity of $t = 25$ and $t = 325$, the order decreases

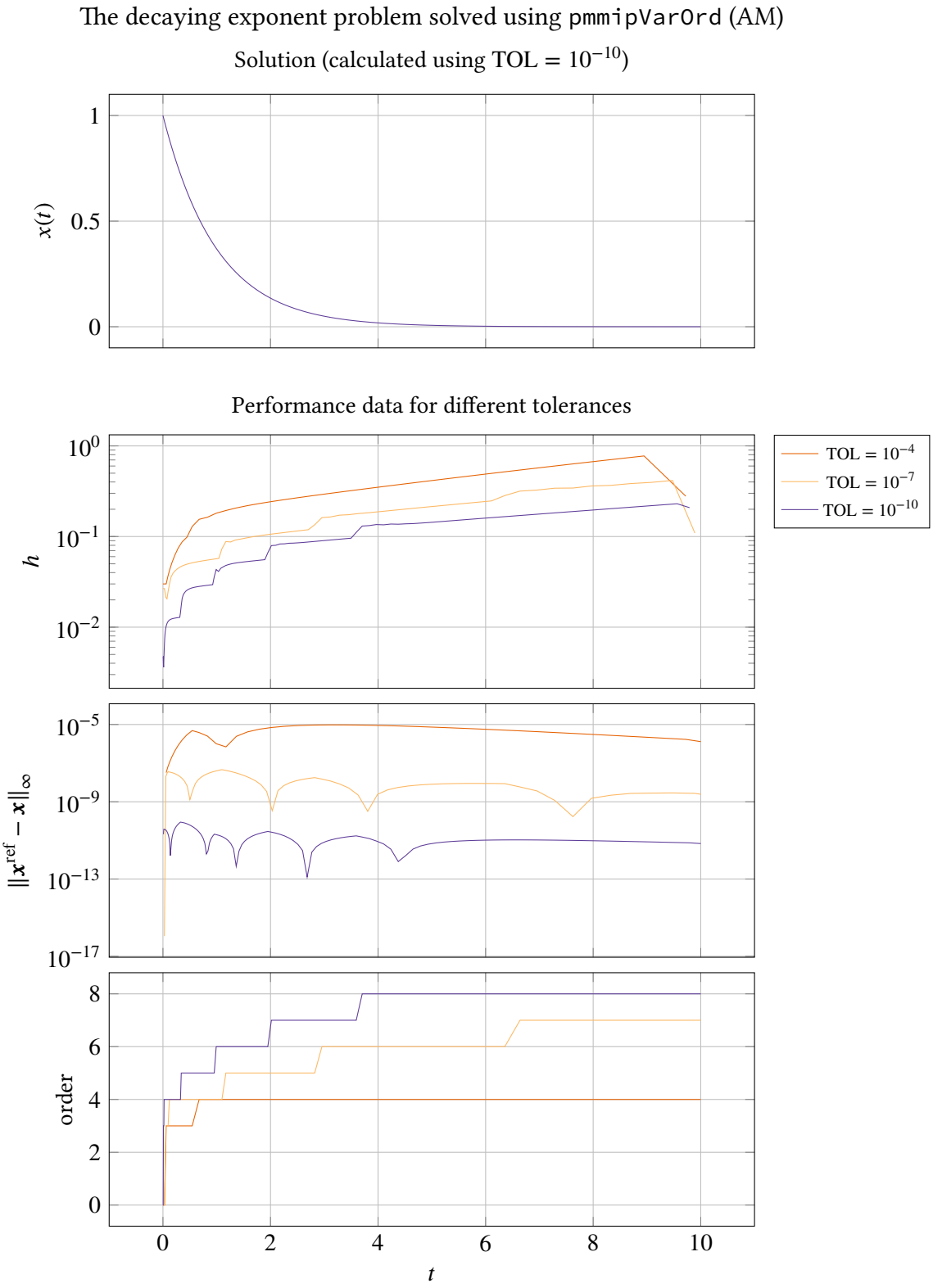

Figure IV.10. The decaying exponent problem ($d = 1$) solved by using `pmmipVarOrd` with 3 different tolerance settings. Here the behavior with regards to order changes is the same as for `pmmeVarOrd` (Figure IV.4), but the increase is slower.

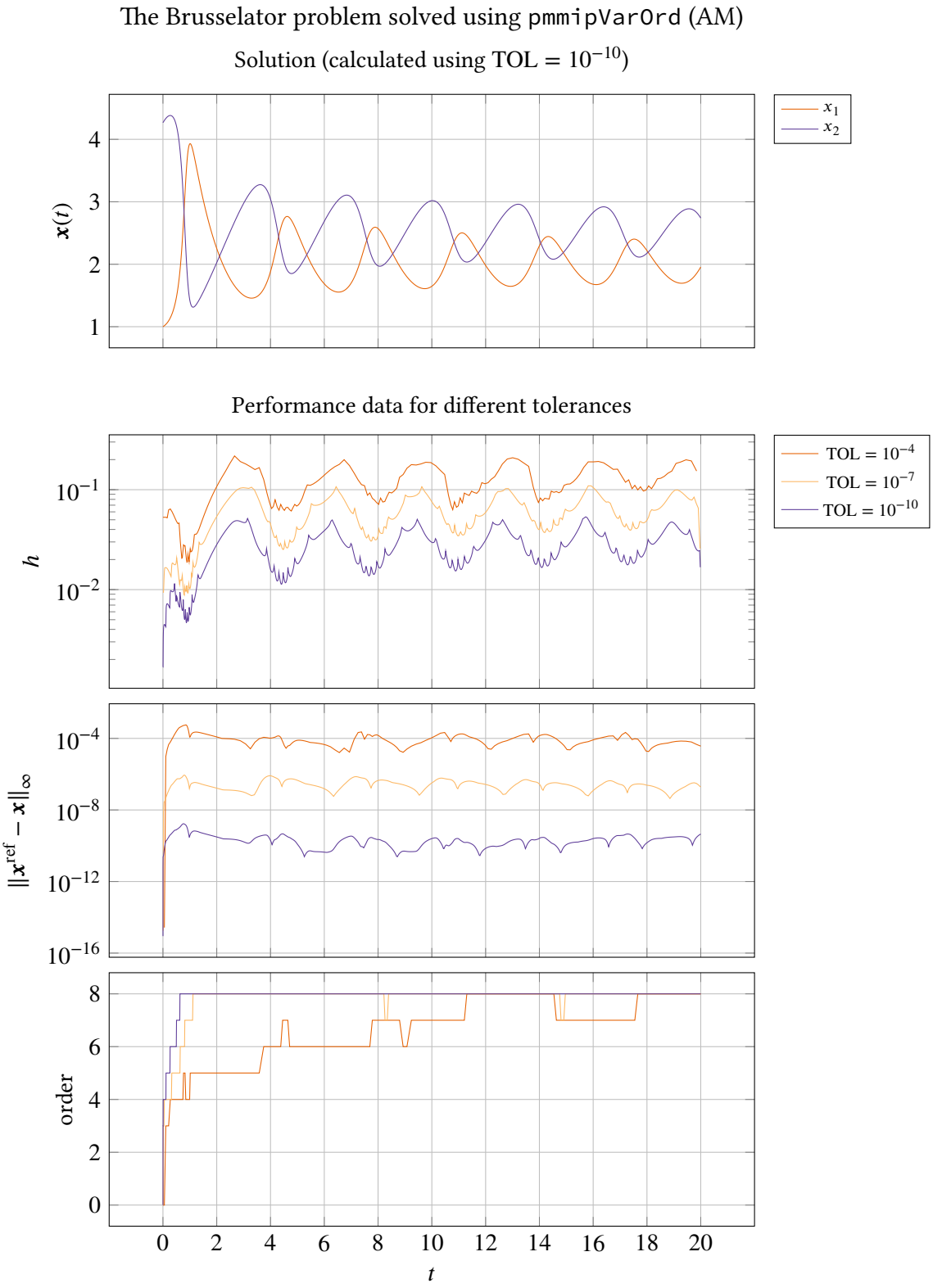

Figure IV.11. The Brusselator problem solved by using `pmmipVarOrd` with 3 different tolerance settings. Here the order is increasing almost throughout the whole integration process.

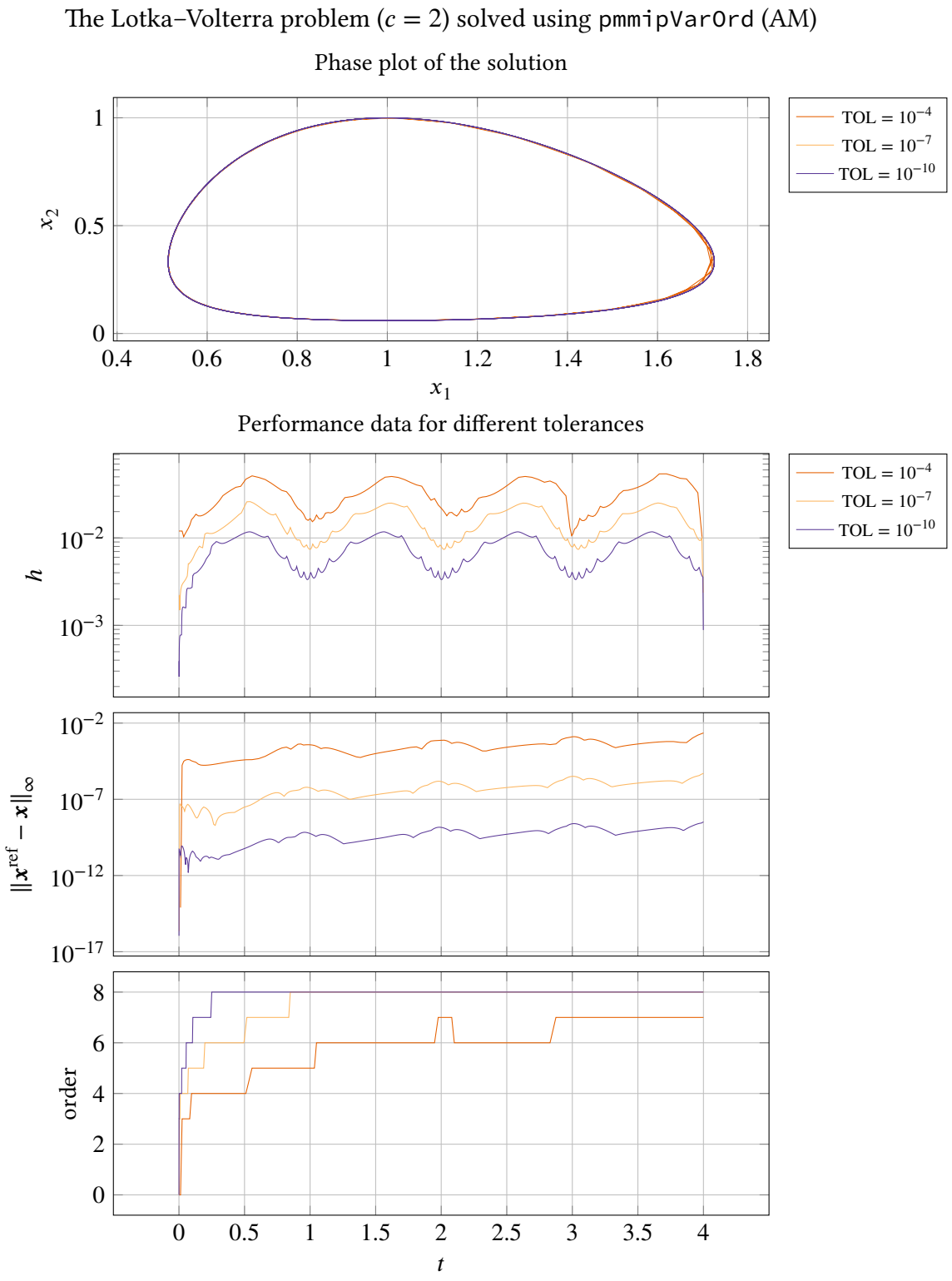

Figure IV.12. The Lotka–Volterra problem solved by using `pmmipVarOrd` with 3 different tolerance settings. The order increases throughout the problem (except for a little bump for TOL $= 10^{-4}$, at $t = 2$), even though the problem is periodic. This gives less information about the order changing algorithm's performance than if it had both increased and decreased the order, but does not necessarily mean that it is bad. It still (except for TOL $= 10^{-4}$) behaves periodically, as a constant behavior is periodic, and the higher order methods are probably the most effective which results in the order increase shown. A small drift-off effect can be noticed both in the error and phase plots.

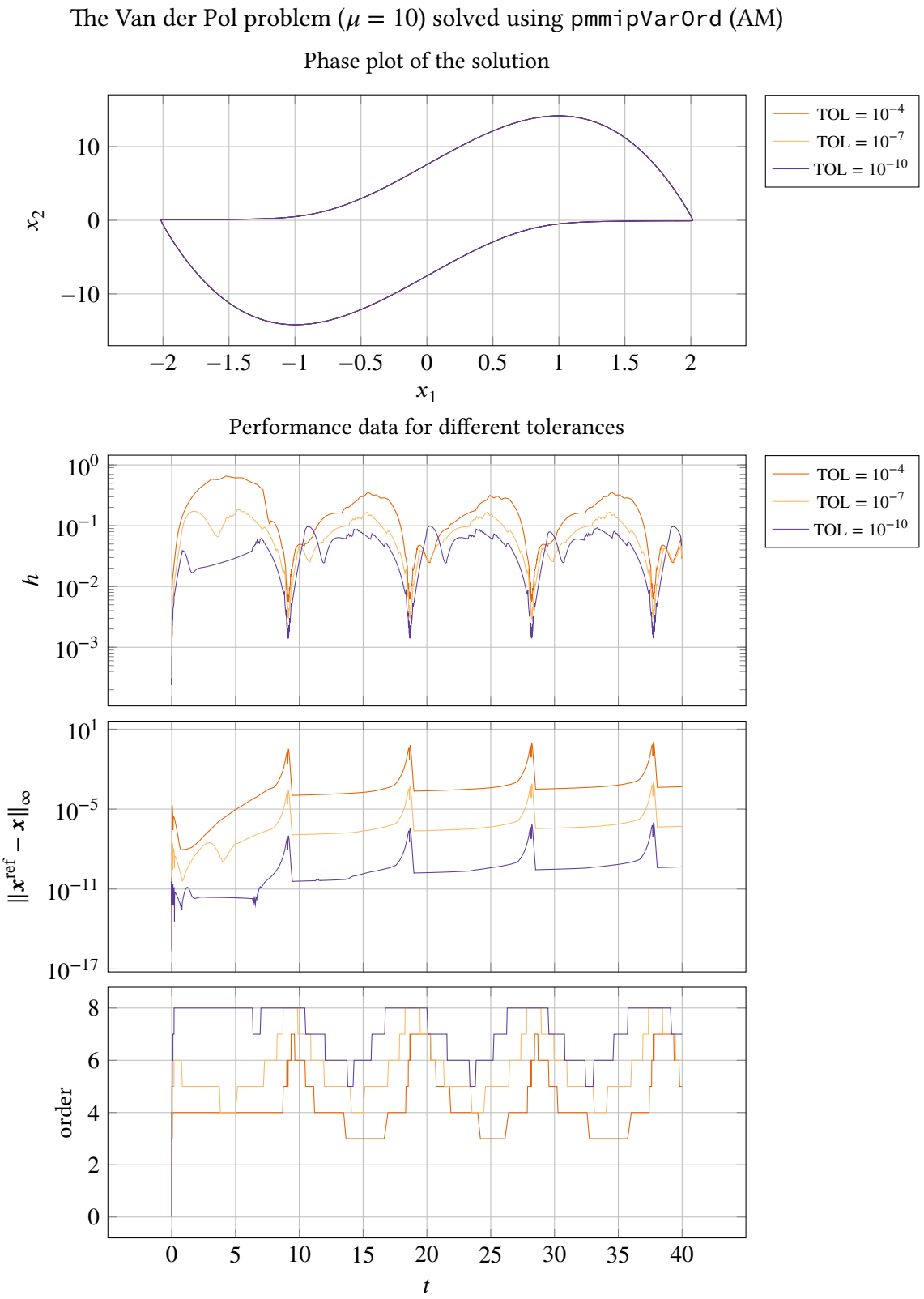

Figure IV.13. The Van der Pol problem solved by using `pmmipVarOrd` with 3 different tolerance settings. The behavior of the step-size sequences, the errors and the orders used all have good periodic behaviors. Note how the error curves show only a small discernible drift-off.

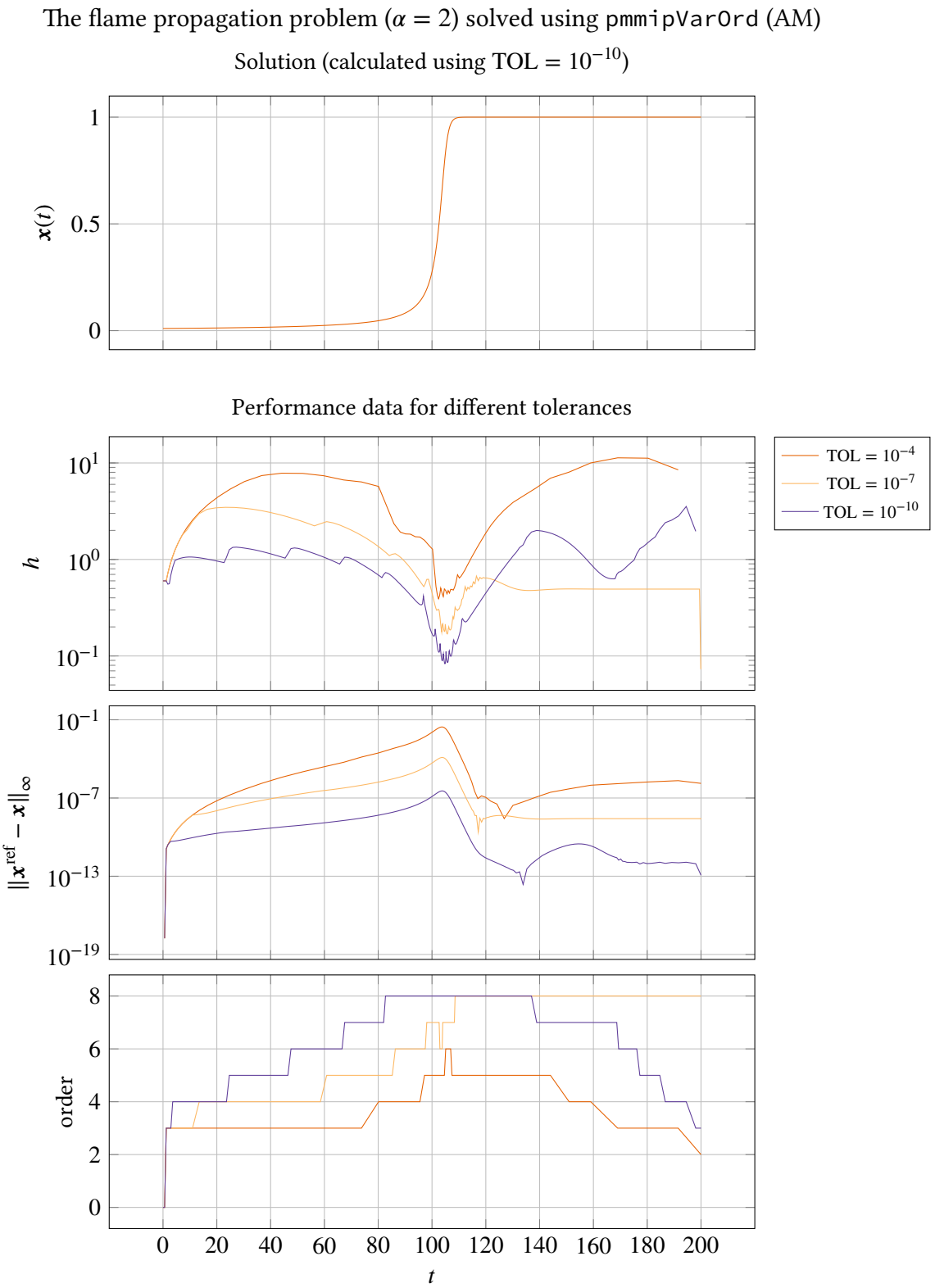

Figure IV.14. The flame propagation problem solved by using `pmmipVarOrd` with 3 different tolerance settings. For TOL = 10^{-4} and TOL = 10^{-10}, the order increases before the problem becomes stiff (for a detailed view of the stiffness behavior of this problem see Figure IV.15), and then decreases again. Compared to the explicit method (Figure IV.8) both the increase and decrease takes place over a longer time. For TOL = 10^{-7}, the order sticks at $p = 8$, a phenomena which is discussed in Subsection IV.3.5.

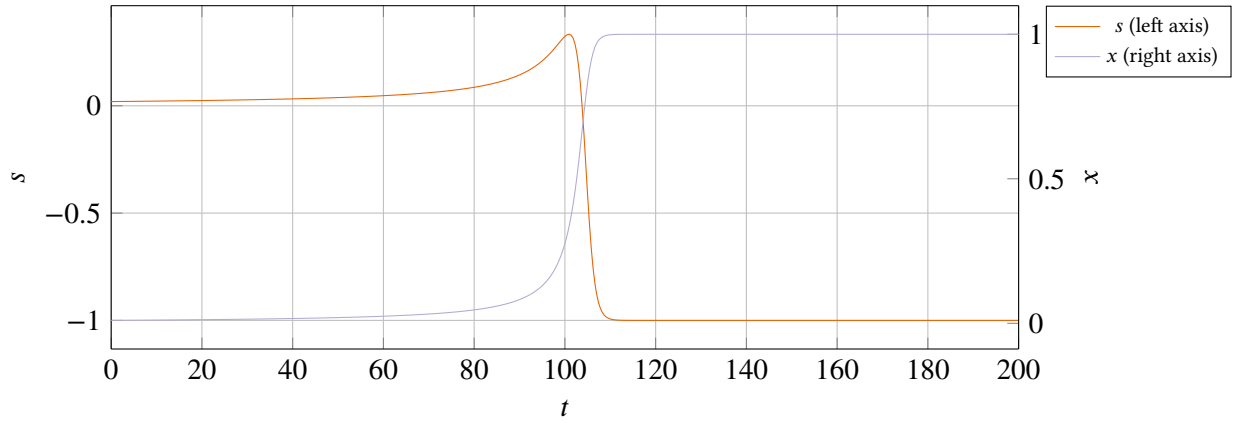

Figure IV.15. The solution and stiffness value of the flame propagation. Note that the stiffness value is not normalized. The monotonically increasing curve is the solution, and its value is shown by the right axis. The other curve is the stiffness, and its value is shown by the left axis.

rapidly and then increases rapidly, and this behavior is almost identical at both places. In the case of the Van der Pol problem, the solver shows excellent results.

There is another interesting thing to note, which touches upon the error curves when the order is changed. Both in the explicit and implicit case the analysis of the error estimate shows that the local error is scaled by a factor when directly using the difference of the predictor and the calculated solution to estimate the local error. In the case of the explicit methods, this can be seen in Equation II.46, and for the implicit methods, analysis shows that there would be a similar factor:

$$\|\boldsymbol{x}(t_n) - \boldsymbol{x}_n\| = \left| \frac{\tilde{C}_{p+1}}{\tilde{C}_{p+1} - \hat{C}_{p+1}} \right| \|\boldsymbol{x}_n - \boldsymbol{x}_n^{\text{pred}}\|. \tag{IV.24}$$

We see that the error estimate used is scaled, and is dependent among other things on the error constants. In practice this could potentially lead to the step-size controller trying to regulate for different set-points for different orders, as the error estimates are scaled differently. But when looking at the error curves this is not an effect which is noticed. Indications that this could be a problem should show up as bumps on the error curves when the order is changed.

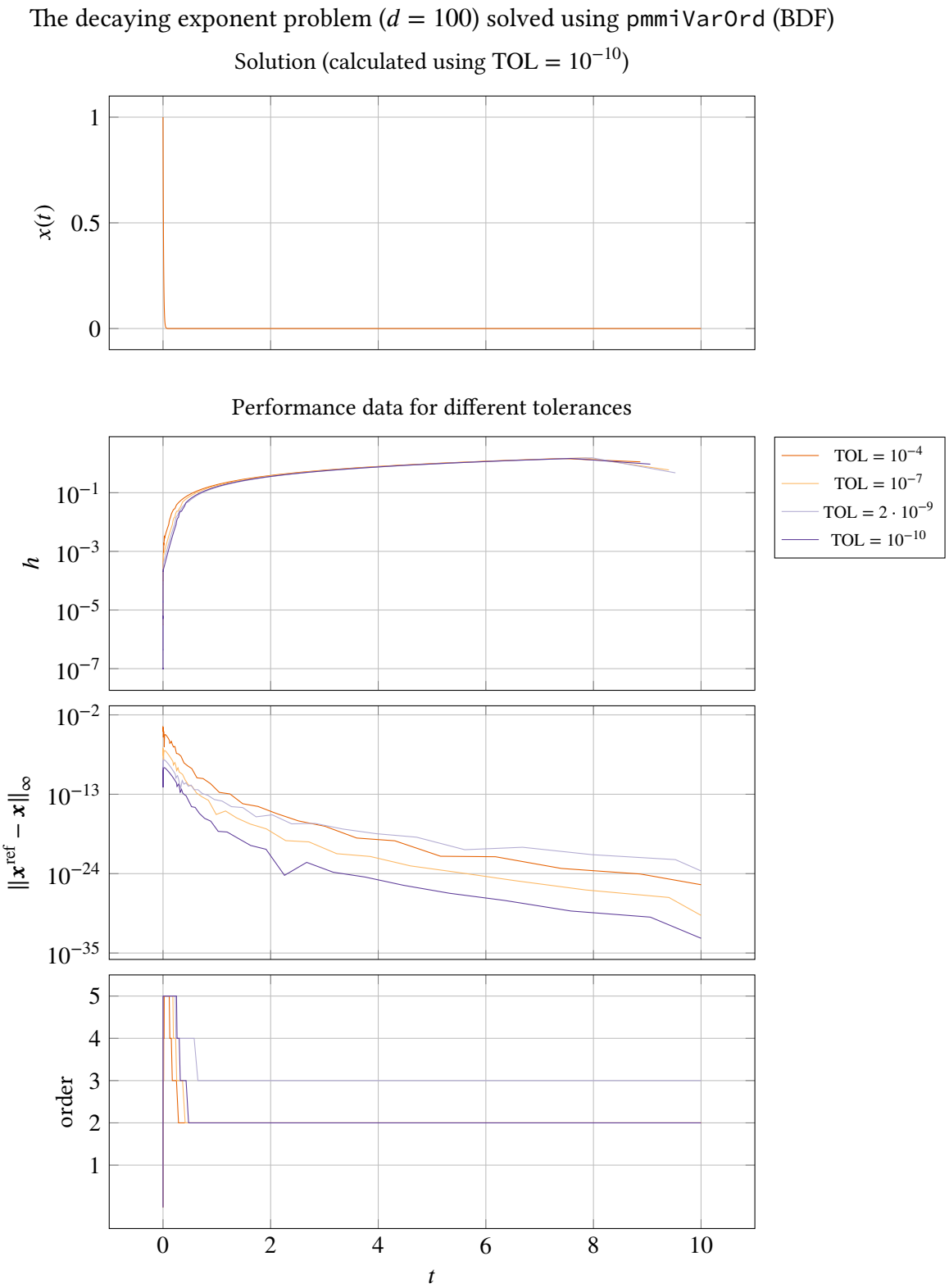

Figure IV.16. The decaying exponent problem solved by using `pmmiVarOrd` with 4 different tolerance settings. The error quickly becomes very small, because the solution goes very rapidly to 0. Here no real benefits are gained by changing order, which is reflected by the step-size curves going to the same value, and sometimes this leads to the order not going down to the lowest level, as is the case with TOL = $2 \cdot 10^{-9}$. In the beginning when the error is larger, all three curves exhibit an expected behavior.

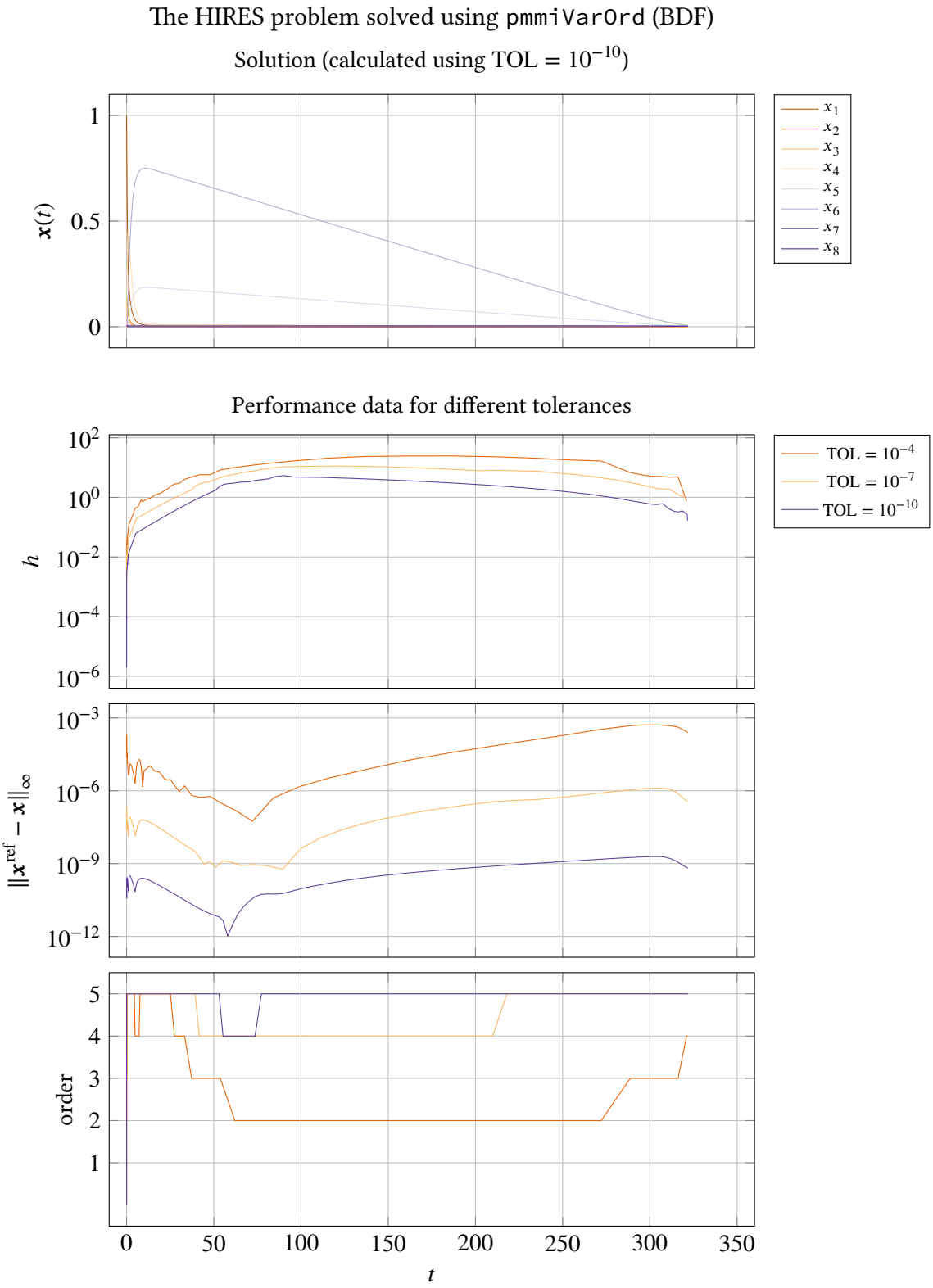

Figure IV.17. The HIRES problem solved by using `pmmiVarOrd` with 3 different tolerance settings. The error curves have similar shape and are evently spaced, which indicates computational stability.

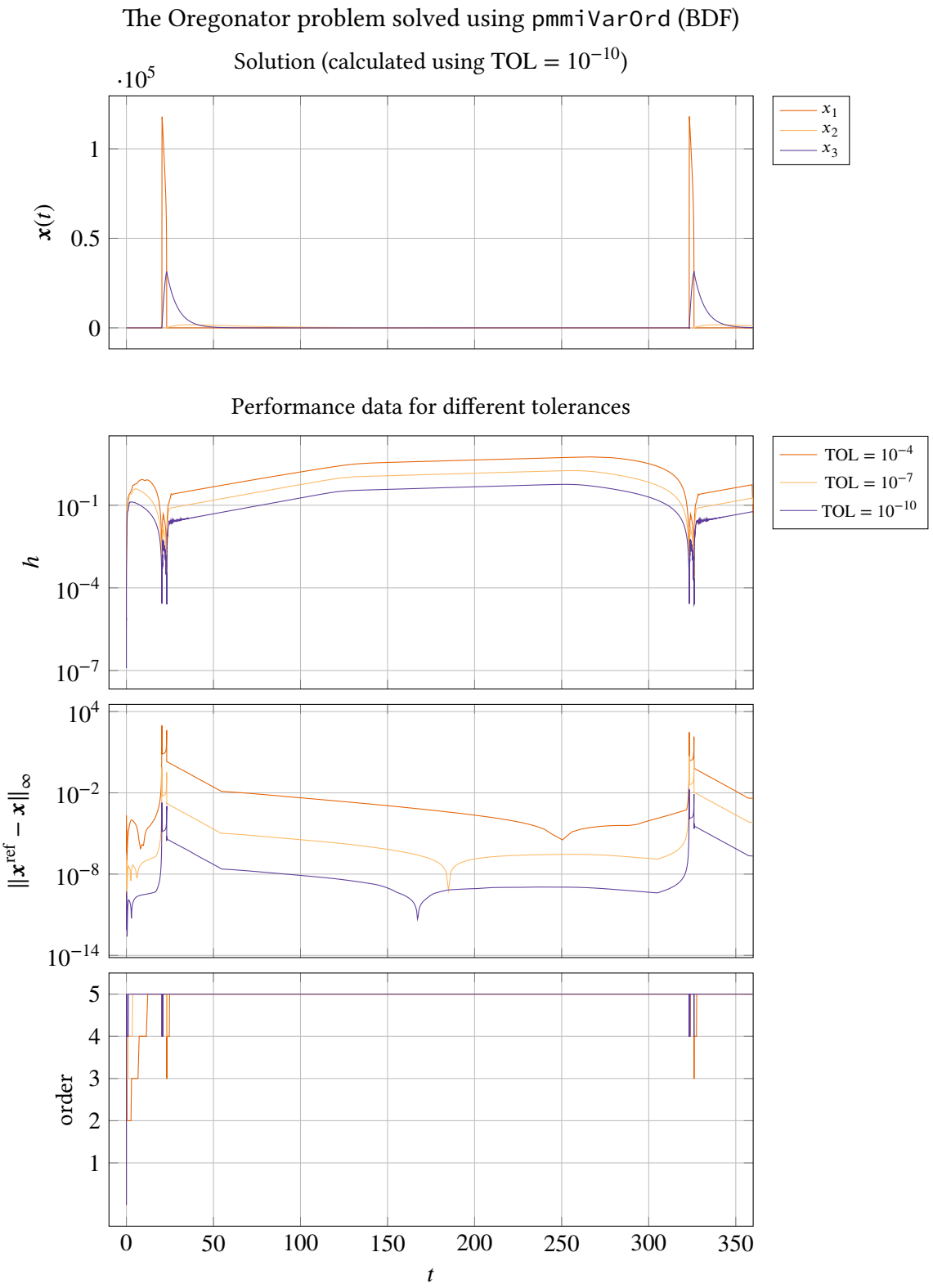

Figure IV.18. The Oregonator problem solved by using `pmmiVarOrd` with 3 different tolerance settings. This problem is periodic, and the solver exhibits periodic behavior at the beginning of each cycle (the problem does not go through two full cycles). Notice especially how the order changes at $t = 25$ and $t = 325$ are very similar, except that there is an extra dip for TOL = 10^{-10} at $t = 325$.

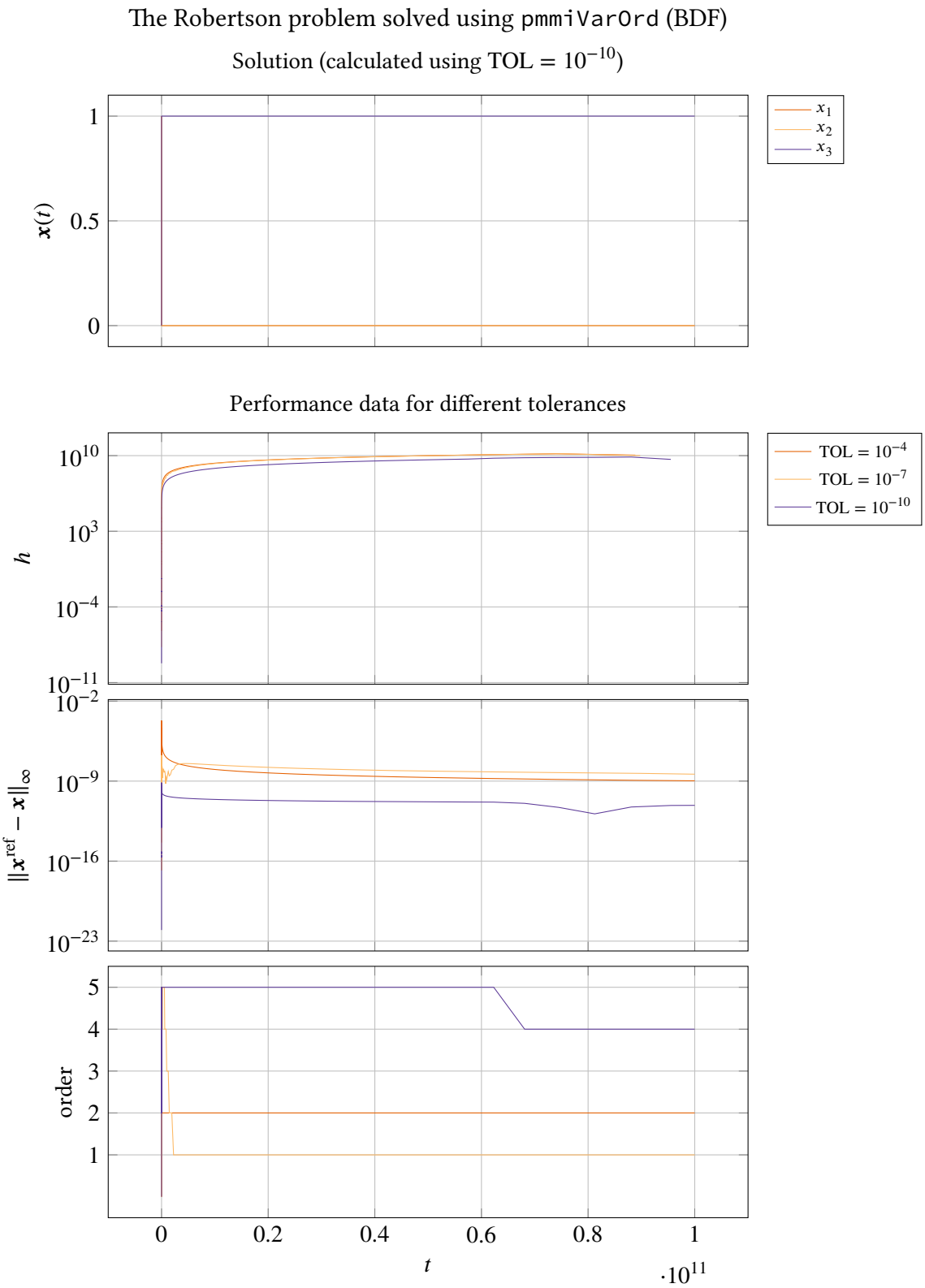

Figure IV.19. The Robertson problem solved by using `pmmiVarOrd` with 3 different tolerance settings. Here the error curves exhibit a bad behavior, as they cross each other and the stricter tolerance setting performs worse than the less strict one. Note that this is a very hard problem, and a comparison with another well established solver (see Figure IV.38) reveals that the result here may not necessarily be regarded as bad.

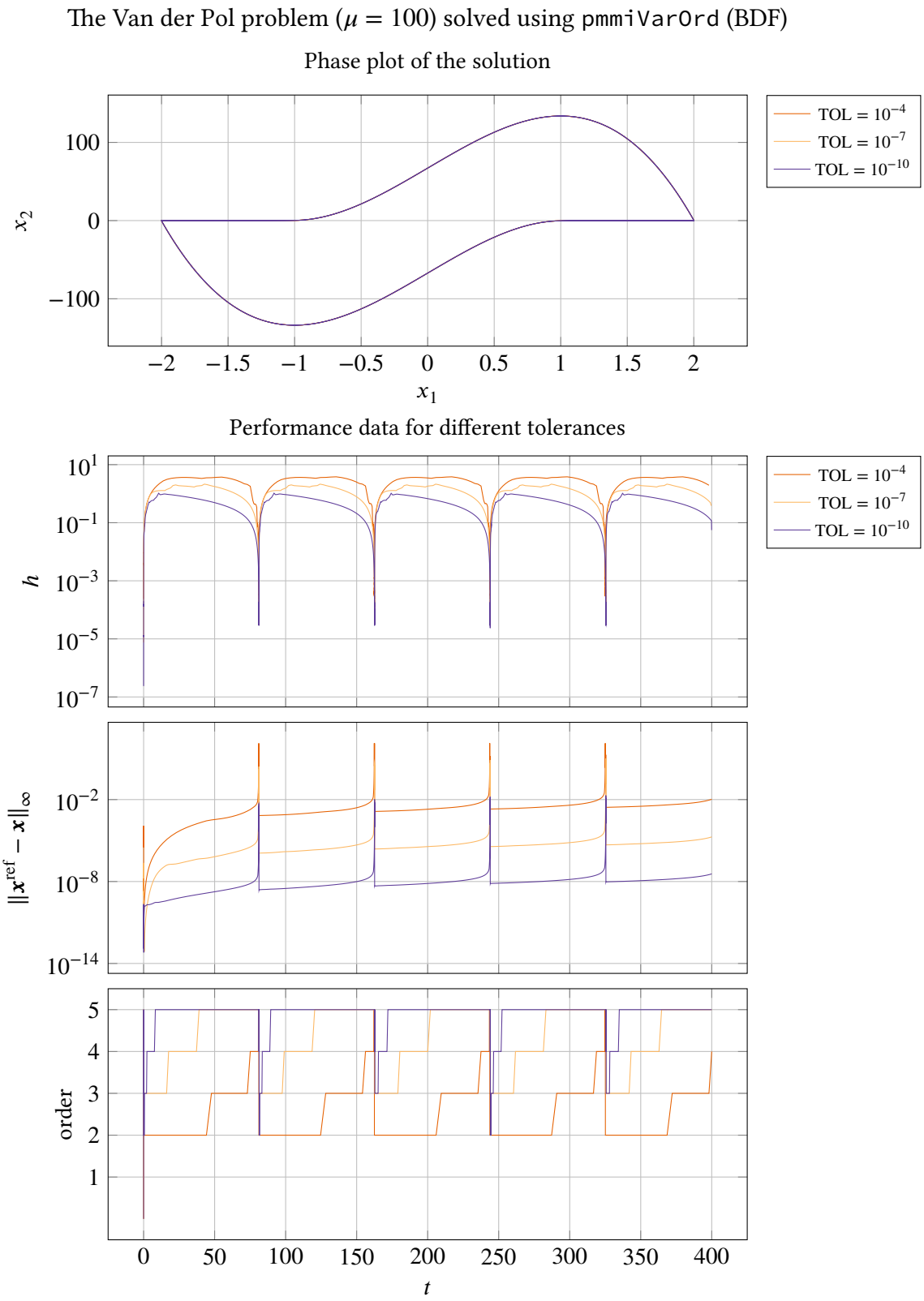

Figure IV.20. The Van der Pol problem solved by using `pmmiVarOrd` with 3 different tolerance settings. The solver exhibits a very periodic behavior in all plots. There is a small drift-off which can be seen in the error plot.

IV.3.3. Accuracy, work and order over multiple tolerances

For the non-stiff solvers the same set of test problems were used, and therefore they are shown in the same plots (Figure IV.21-IV.25). No calibration to achieve the same accuracy when a certain tolerance is demanded has been done. This means that even though a solver may have achieved a greater accuracy for a certain tolerance, the work done (measured by the number of steps in this case) must be taken into account as well.

The way the achieved accuracy responds to changes of the demanded tolerance is good for both `pmmeVarOrd` and `pmmipVarOrd`. The curves for `pmmeVarOrd` are less straight, especially when solving the Van der Pol problem (Figure IV.24), and they are generally less smooth and here there is room for improvement; the results for `pmmipVarOrd` are much better in this regard.

All tests also show that `pmmipVarOrd` achieves a greater accuracy for a less amount of work. This is not unexpected as the Adams–Moulton methods should outperform the Adams–Bashforth methods in most cases. The fact that `pmmipVarOrd` does not start at the lowest order also contributes to the lower amount of work done (as discussed in Subsection IV.3.1), but even if this were not the case, the amount of work done to achieve a certain accuracy is expected to be less for the Adams–Moulton methods; if the method, using a starting order of 2 and a preset initial step-size, in Figure IV.3 is compared to the explicit method in Figure IV.23, one can see that this is the case for the Lotka–Volterra problem.

In all cases there is also a trend, when looking at the mean order, which indicates that at stricter tolerances, higher order methods are preferred. In the case of `pmmipVarOrd` and the Brusselator problem (Figure IV.22) the difference is very small though, and the curve indicates that order 8 is used almost throughout the problem, something which can also be seen in Figure IV.11. That the solvers start favoring higher order methods at stricter tolerances is in agreement with the results in Figure III.41–III.47, which shows that when using more strict tolerances the difference in the amount of work a higher and lower order method uses increases.

We saw in the previous section, in Figure IV.14, that the order sometimes does not change as expected, but stays high even though it is beneficial to change to a lower order. This also shows up in these tests, both in Figure IV.25 (flame propagation) and IV.24 (Van der Pol), where we can see peaks in the mean order, and corresponding peaks in the amount of work performed. In the Van der Pol problem, there is also an indication that this phenomena has a small effect on the accuracy response, when changing the tolerance. There are small dips in the accuracy curves at places corresponding to the peaks of the mean order curve.

When comparing these results to the fixed order cases (Figure III.41–III.43 and Figure III.45–III.47), one can see that for the explicit solvers the accuracy curves are generally less smooth when using variable order. When comparing `pmmip` and `pmmipVarOrd`, the curves are generally of the same quality (they are straight and smooth). The achieved accuracy and the amount of work can in these cases not be compared fairly, as the variable order solvers allow for higher order methods than are shown in the fixed order cases, and when looking at the results from the fixed order solvers one clearly sees that there is a trend which indicates that higher order methods achieve a better accuracy with a less amount of work done.

When looking at the results from the stiff problems (Figure IV.27–Figure IV.30), one can see that the accuracy curves are very smooth and straight, except for the Robertson problem (see Figure IV.29). The accuracy curve for this problem is worse than any other, but this is also a very hard problem and some context for this will be given in Subsection IV.3.4.

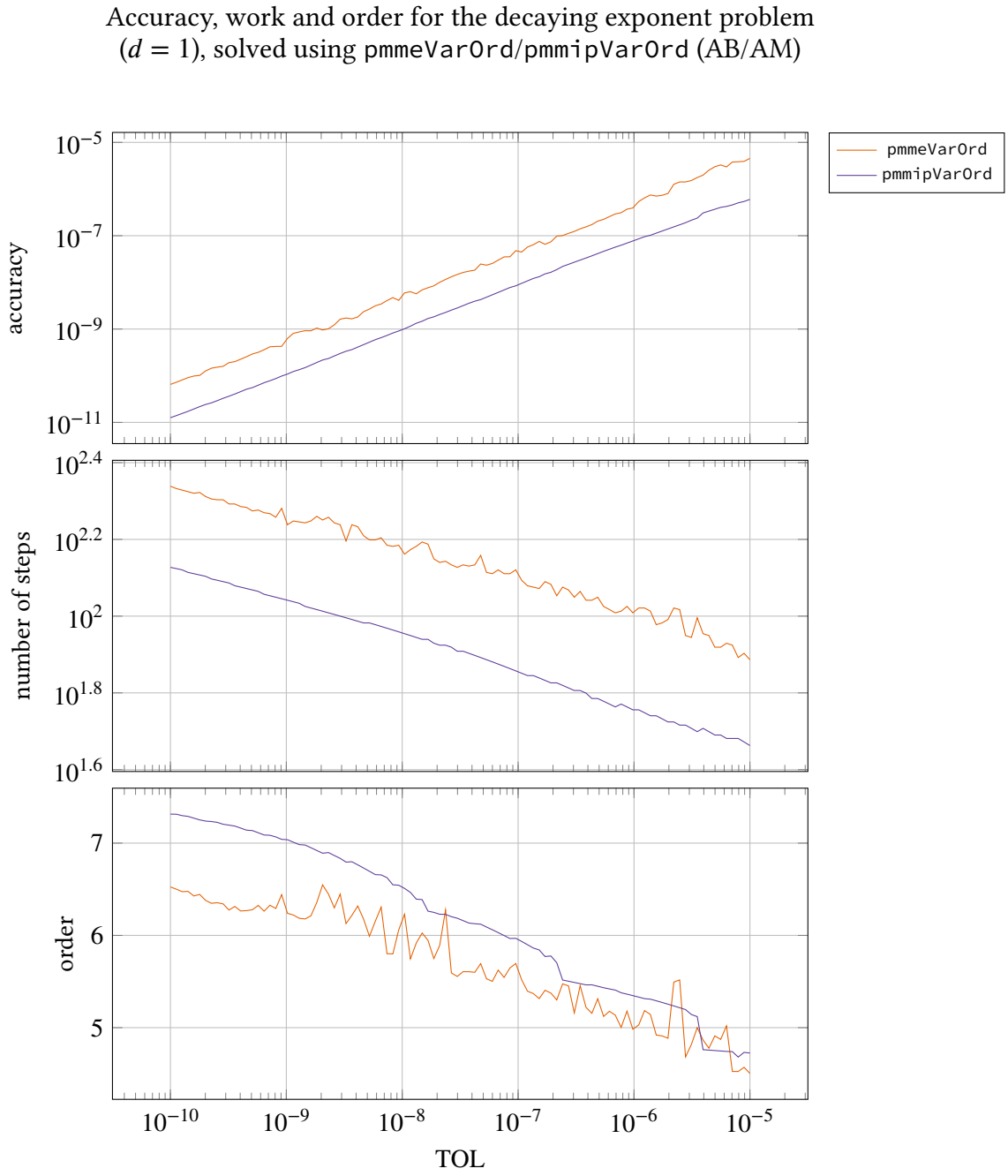

Figure IV.21. Performance data for `pmmeVarOrd` and `pmmipVarOrd` when solving with 100 different tolerance settings for the decaying exponent problem. The performance of `pmmipVarOrd` is better than the performance of `pmmeVarOrd`, both in regards to achieved accuracy for a certain amount of work and the smoothness of the accuracy curves. Both solvers favors higher order methods for stricter tolerances, compared to less strict tolerances.

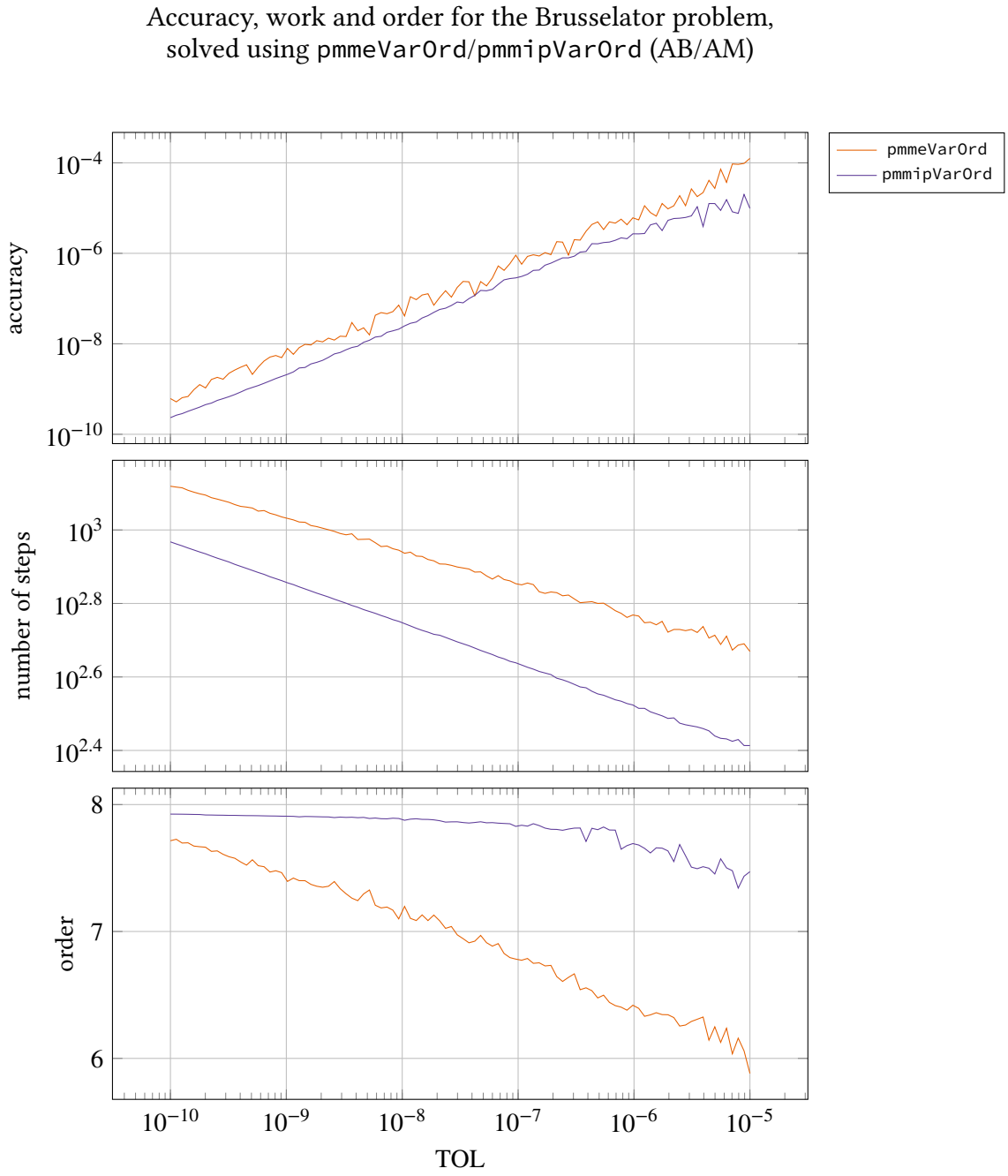

Figure IV.22. Performance data for `pmmeVarOrd` and `pmmipVarOrd` when solving with 100 different tolerance settings for the Brusselator problem. The achieved accuracies are comparable, but `pmmipVarOrd` produces a much smoother curve which is preferable and uses less work. Note that the difference in mean order for `pmmipVarOrd` varies less as the tolerance varies, then when using `pmmeVarOrd`, indicating that the implicit solver probably uses mostly higher order methods throughout the whole integration process.

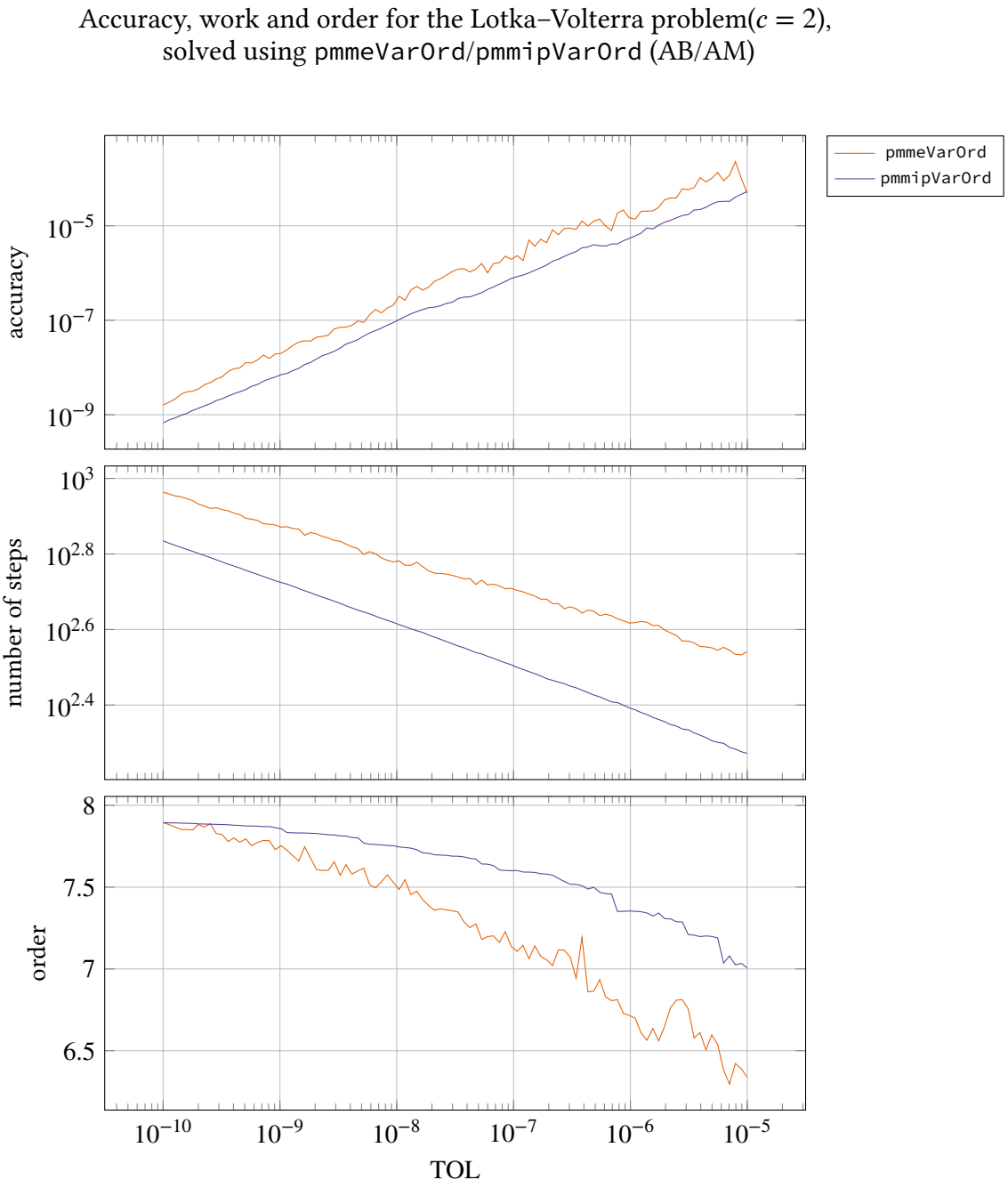

Figure IV.23. Performance data for `pmmeVarOrd` and `pmmipVarOrd` when solving with 100 different tolerance settings for the Lotka–Volterra problem. The performance of `pmmipVarOrd` is better than the performance of `pmmeVarOrd`, both in regards to achieved accuracy for a certain amount of work and the smoothness of the accuracy curves. Note that the difference in mean order for `pmmipVarOrd` varies less as the tolerance varies, then when using `pmmeVarOrd`, indicating that the implicit solver probably uses mostly higher order methods throughout the whole integration process.

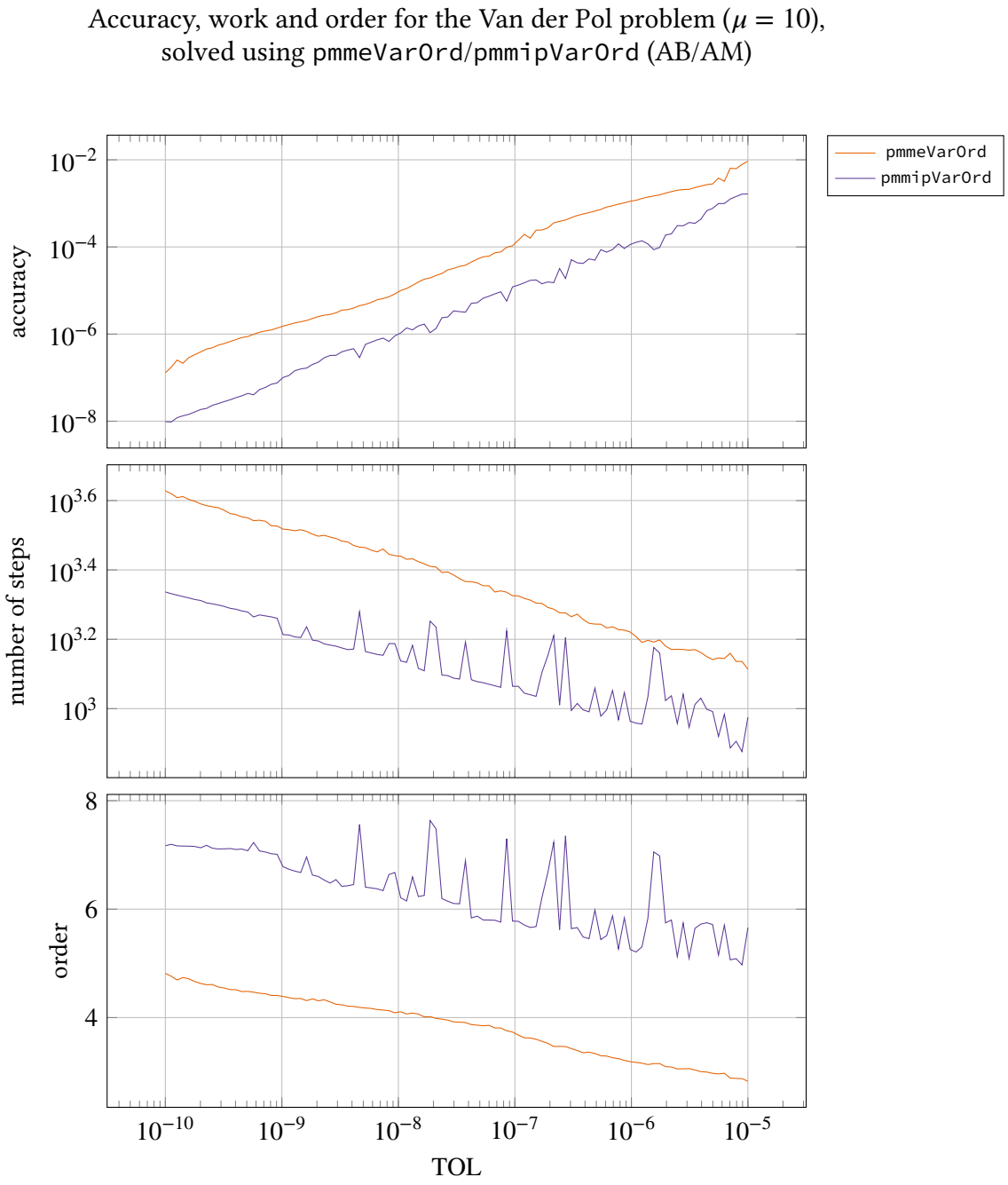

Figure IV.24. Performance data for `pmmeVarOrd` and `pmmipVarOrd` when solving with 100 different tolerance settings for the Van der Pol problem. Both solvers produces accuracy curves with an acceptable smoothness, although the one produced by `pmmipVarOrd` is somewhat straighter, which is preferable. The implicit solver also achieves a greater accuracy and uses less work. Both solvers favors higher order methods for stricter tolerances, compared to less strict tolerances. The peaks in work and order for the implicit solver, indicates that the order gets stuck sometimes. These peaks also corresponds to the small dips in the accuracy curve.

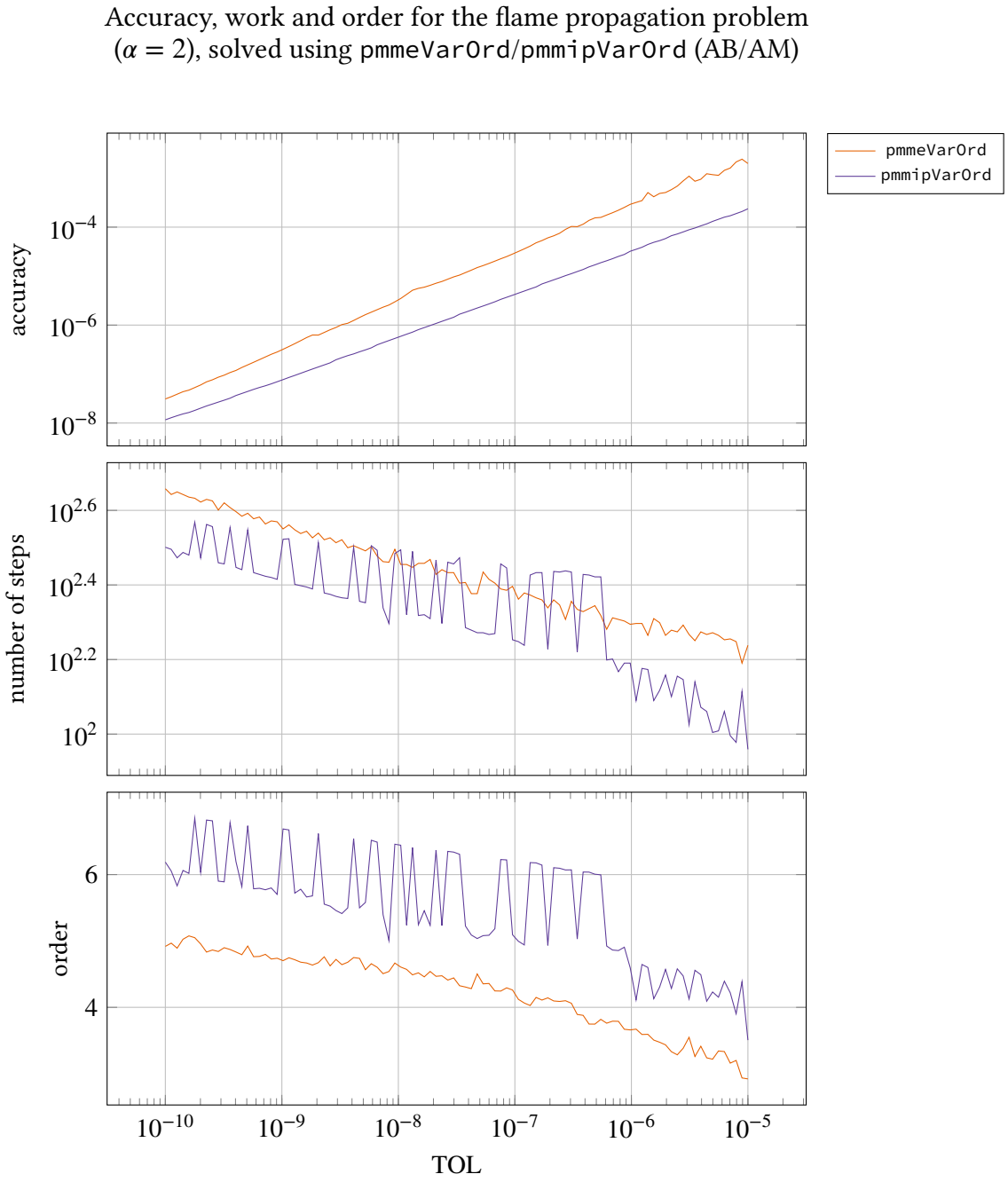

Figure IV.25. Performance data for `pmmeVarOrd` and `pmmipVarOrd` when solving with 100 different tolerance settings for the flame propagation problem. Both solvers shows very good accuracy curves, which are both straight and smooth. We can see that, for `pmmipVarOrd`, the order often gets stuck, which is indicated by the peaks in the work and order curves. This could also be seen in the more detailed view in Figure IV.14. Both solvers favor higher order methods for stricter tolerances, compared to less strict tolerances.

The trend which indicates that higher orders are preferred at stricter tolerance settings is present for the stiff problems as well (which once again agrees with the results shown for fixed order solvers in Figure III.49–III.52). In the case of the Oregonator problem (Figure IV.28), this difference is very small, and the mean order is almost 5 for all tolerances. This indicates that the order 5 is used mostly throghout the whole integration process, for all tolerance settings. This behavior can be observed when looking at the orders used in Figure IV.18. The exception to the trend is the decaying exponent problem (Figure IV.26), where the mean order changes up and down between approximately 2, 3, and 4. When looking at the detailed behavior of the order algorithm in Figure IV.16, one sees that the order stays the same in the later parts of the integration process, and that the error is very small here. This indicates that because the error is so small, there are no real benefits to changing order, but the greatest addition to the mean order comes from this later part of the integration process. If one looks at the detailed behavior of the order used in Figure IV.16, it indicates that in the beginning of the integration process, where almost all error is accumulated, the solver uses higher order methods for a longer period when using stricter tolerances, which is in agreement with the results shown in Figure III.49. Note also that even though the error curves in Figure IV.16 cross each other and do not exhibit a good behavior, this is not noticed in the plot showing the achieved accuracy in Figure IV.26, and this is, as previously stated, because the great majority of the integrated error is accumulated in the beginning of the integration, where the curves exhibit a good behavior.

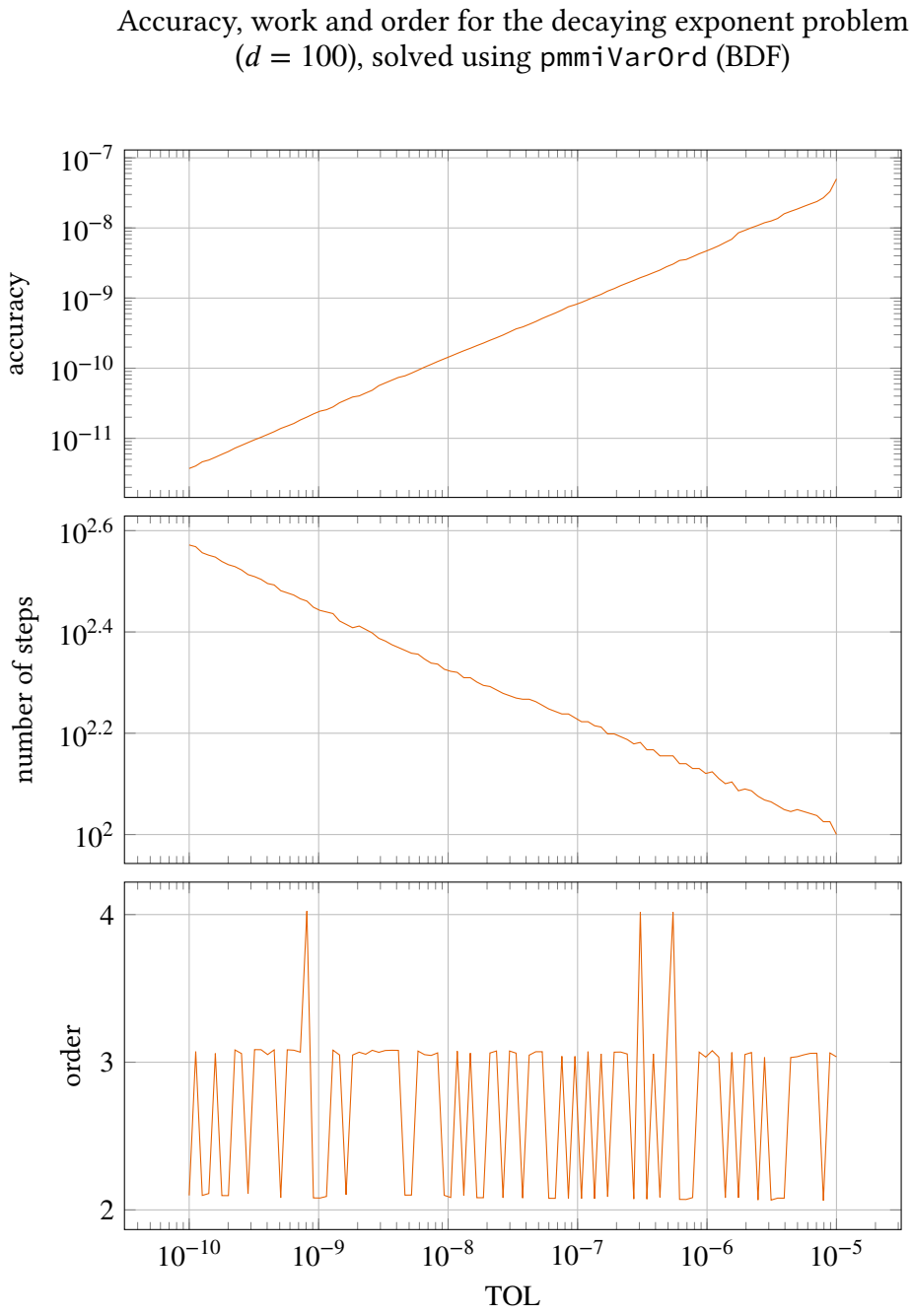

Figure IV.26. Performance data for `pmmiVarOrd` when solving with 100 different tolerance settings for the decaying exponent problem. The accuracy curve is very straight and smooth, indicating good computational stability. The jumps in the mean order can be explained when looking at the detailed view of the orders used in Figure IV.16, which shows that either order 2, 3, or 4, is used in the part of the solution where the error becomes very small. There is probably no benefit in changing order due to the small error, and because this constitutes the majority of the solution the mean order will be either approximately 2, 3, or 4.

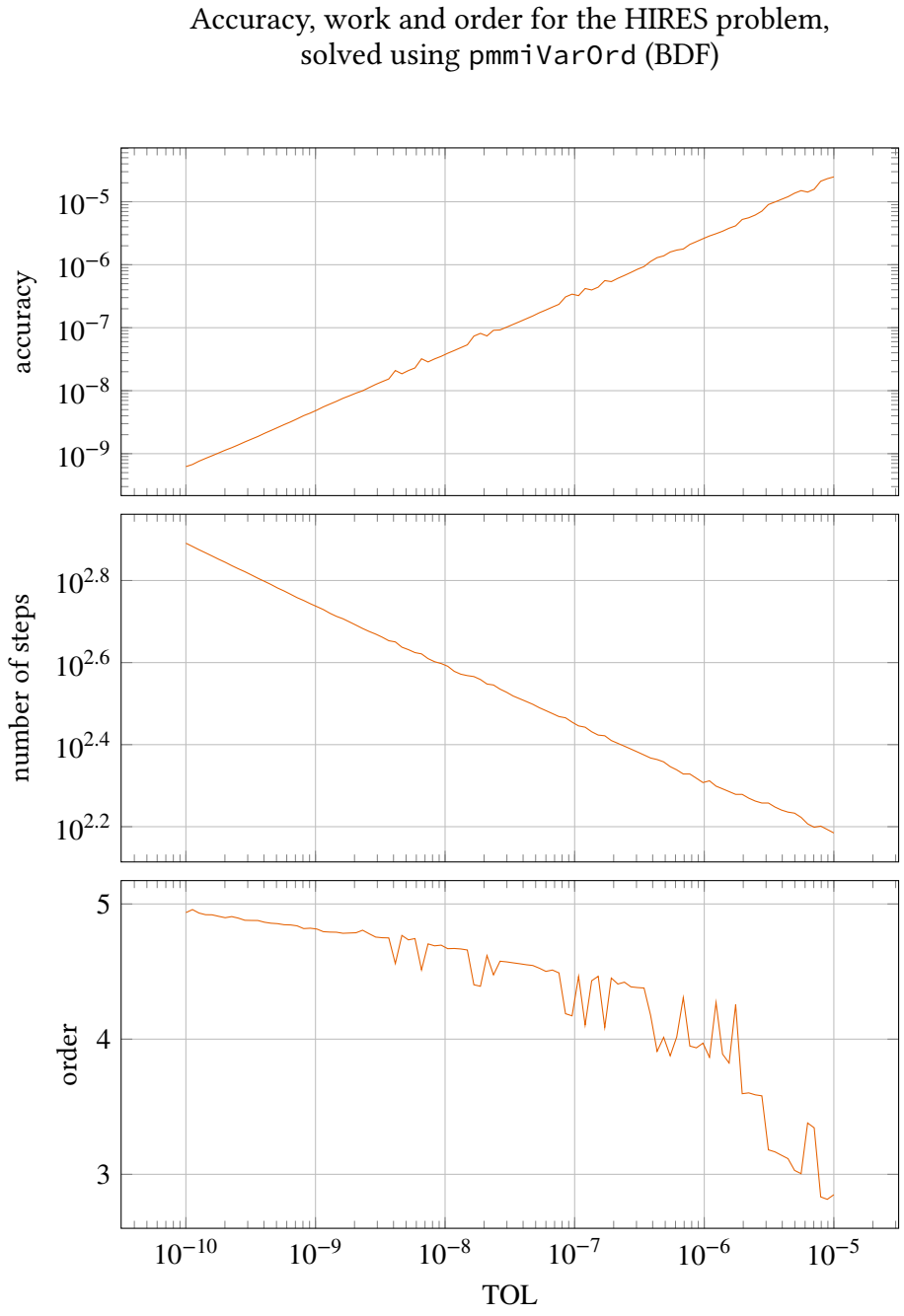

Figure IV.27. Performance data for `pmmiVarOrd` when solving with 100 different tolerance settings for the HIRES problem. The accuracy curve is very straight and smooth, indicating good computational stability. The mean order shows that the solver favors an increasing use of higher order methods when the tolerance becomes more strict.

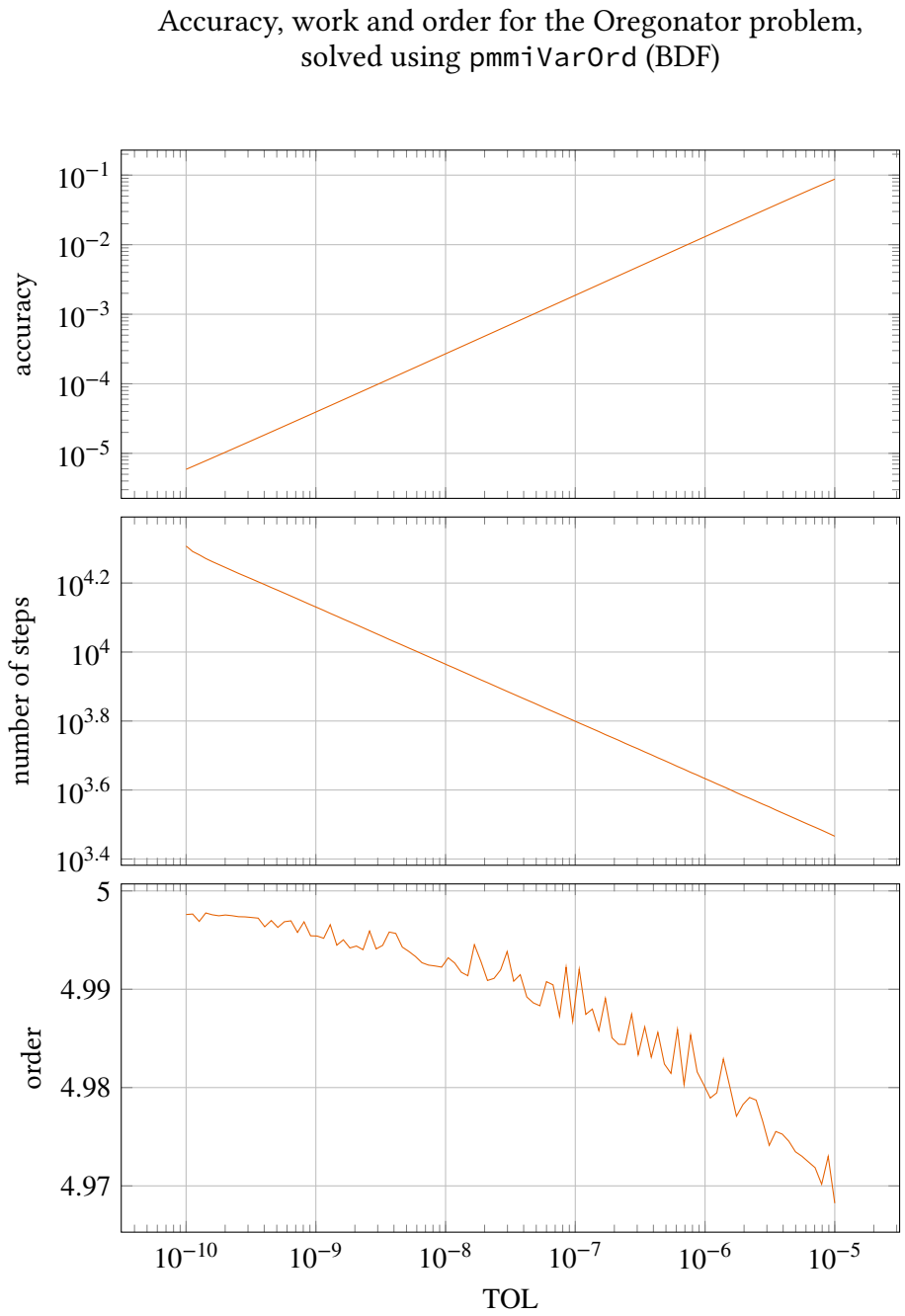

Figure IV.28. Performance data for `pmmiVarOrd` when solving with 100 different tolerance settings for the Oregonator problem. The accuracy curve is very straight and smooth, indicating good computational stability. The difference in mean order is extremely small, indicating that only one order is primarily used throughout the integration process (making the solver behave as a fixed order solver).

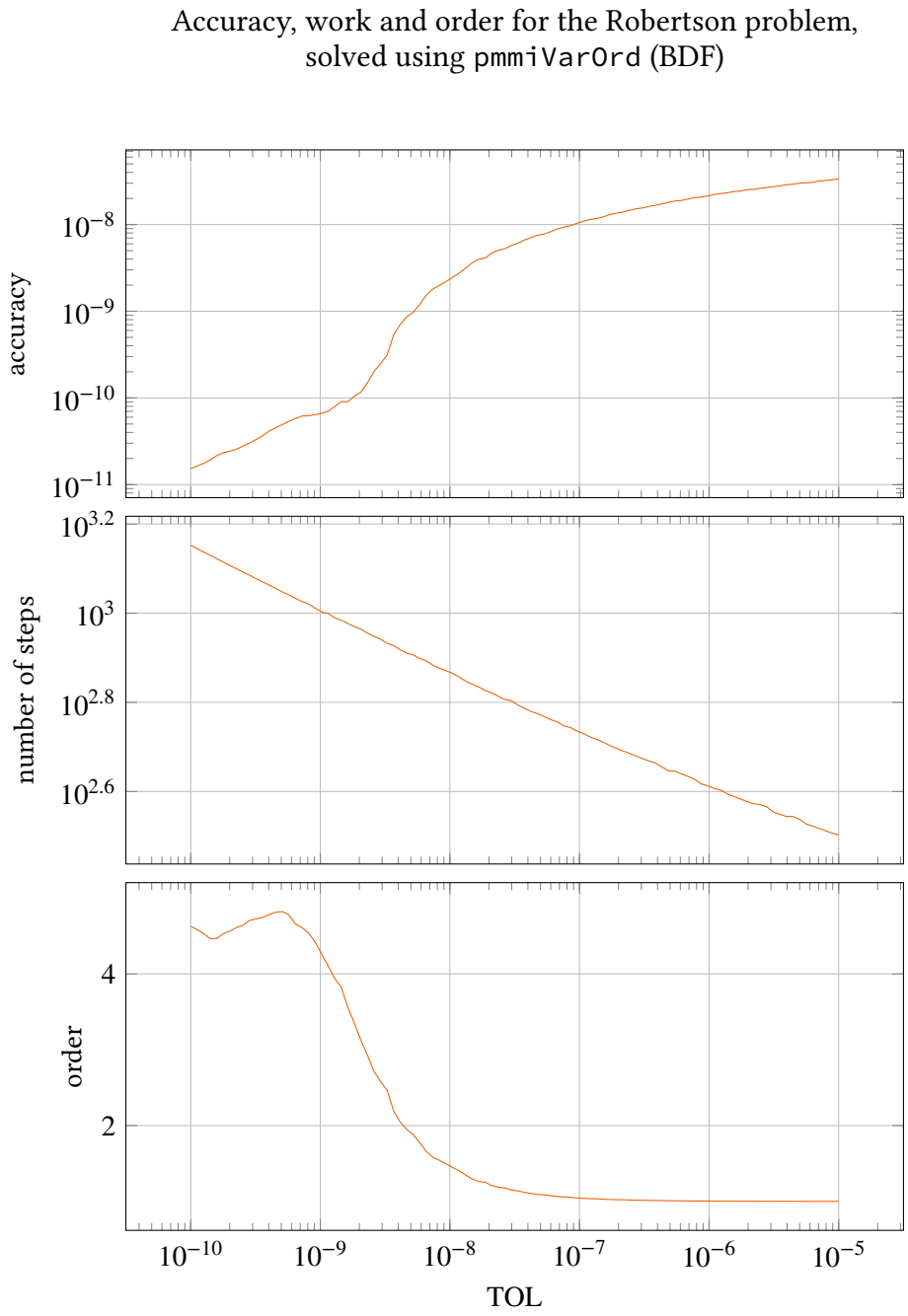

Figure IV.29. Performance data for `pmmiVarOrd` when solving with 100 different tolerance settings for the Robertson problem. The accuracy curve is much less smooth and straight than for the other test problems. This considerable difference is due to the Robertson problem being very difficult to solve. The results in Figure IV.38 gives some context to this by showing the performance of a well established solver.

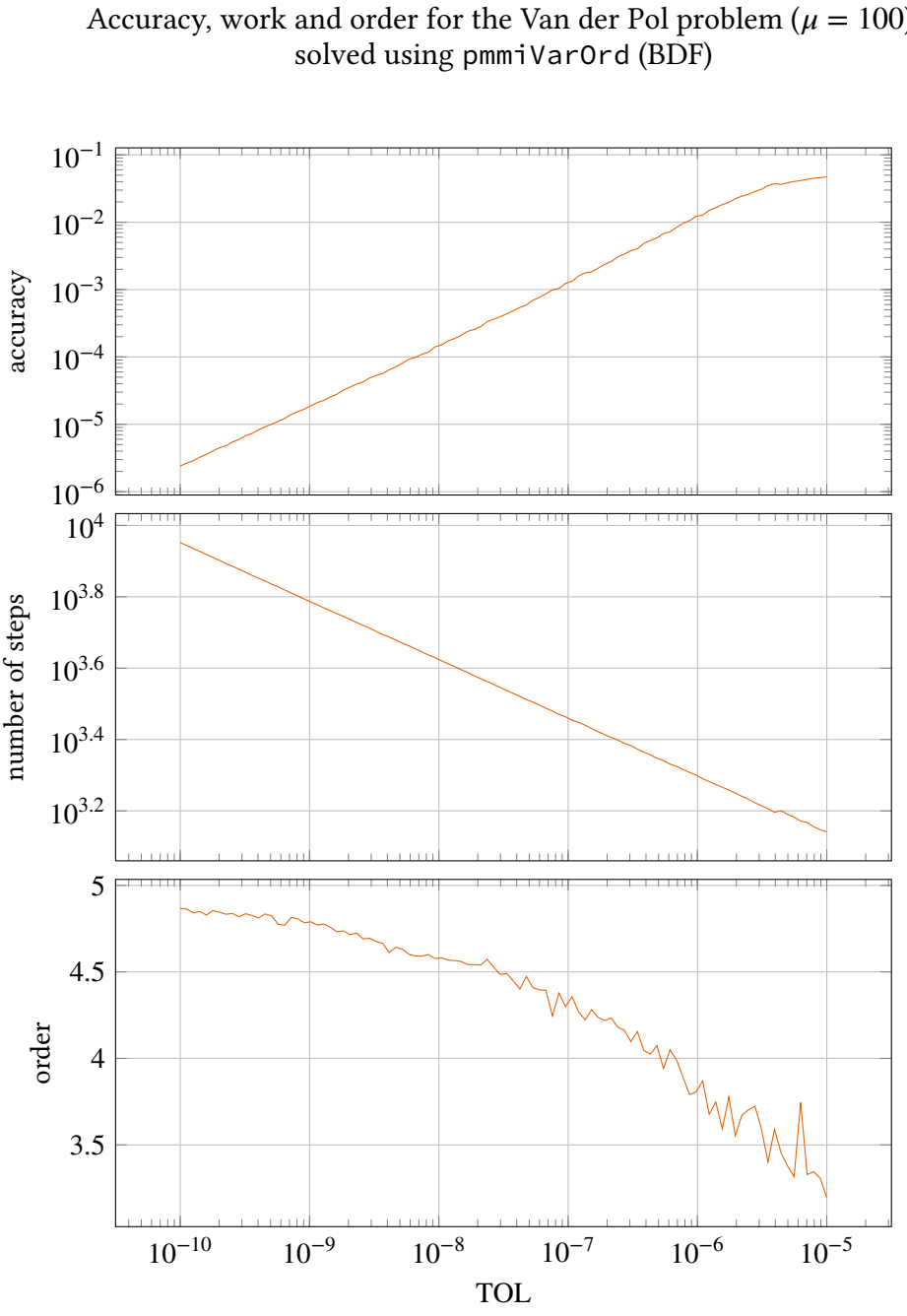

Figure IV.30. Performance data for `pmmiVarOrd` when solving with 100 different tolerance settings for the Van der Pol problem. The accuracy curve is very straight and smooth, indicating good computational stability. The mean order shows that the solver favors an increasing use of higher order methods when the tolerance becomes more strict.

IV.3.4. Comparison with other solvers

Our solvers were compared to Matlab's `ode113` (non-stiff) and `ode15s` (stiff). The latter was called with the 'BDF' option. When comparing the non-stiff solvers (Figure IV.31–IV.35), with the exception of the results for the Van der Pol problem (Figure IV.34), one sees that the accuracy achieved and the amount of work done by the solvers are comparable at less strict tolerances, and the amount of work done by `ode113` is less at stricter tolerances. The results show that in all cases the slope of the work curve for `pmmipVarOrd` has a larger negative value. One possible reason for this is that `ode113` utilizes methods with an order up to and including 12 [16]. As has been previously shown, at stricter tolerances higher order methods are preferred by our solvers, and because the allowed maximum order is 8 this may influence the slope of the work curve.

One can see that `ode113` generally has a larger amount of failed steps (especially at stricter tolerances). This is probably due to fundamental differences in the step-size regulation algorithms.

A significant difference in the results is how smooth the accuracy curves are. The curves for `pmmipVarOrd` are smoother than those for `ode113` every single time.

As we saw in the last section, `pmmipVarOrd` sometimes stays at a higher order, even though it would be beneficial to go down to a lower order. This greatly affects the amount of work needed compared to `ode113`. In the case of the Van der Pol problem, where we have already seen that the accuracy curve seems to be somewhat affected by this phenomena (Figure IV.13), we see that the accuracy curve is smoother for `pmmipVarOrd` than for `ode113` (Figure IV.34).

The results for the stiff solvers (Figure IV.36–IV.41) show that the accuracy achieved and the amount of work done by the solvers are comparable in all cases. The main difference can be found when looking at the amount of failed steps and how smooth the accuracy curves are.

In all cases, except for the decaying exponent problem (Figure IV.36), `ode15s` has many more failed steps, and for example in the case of the Van der Pol problem Figure IV.37 the difference is very large.

When looking at the smoothness of the accuracy curves, one can see that in the case of the decaying exponent problem the two solvers produces curves with comparable smoothness. In all other cases the curves produced by `pmmiVarOrd` are smoother than those produced by `ode15s`.

The Robertson problem stands out, because for the less strict tolerances the error curve produced by `ode15s` has very large jumps. This is also accompanied by a large amount of steps taken and a large amount of failed steps. The error is on such a scale that the solver, at least partly, has failed to compute a solution. To get a better view of how `ode15s` performs for stricter tolerances the results are also shown in Figure IV.39, but here the least strict tolerances have been excluded. Here it can be seen that the accuracy curve produced by `pmmiVarOrd` is much better than the one produced by `ode15s`, with regards to smoothness.

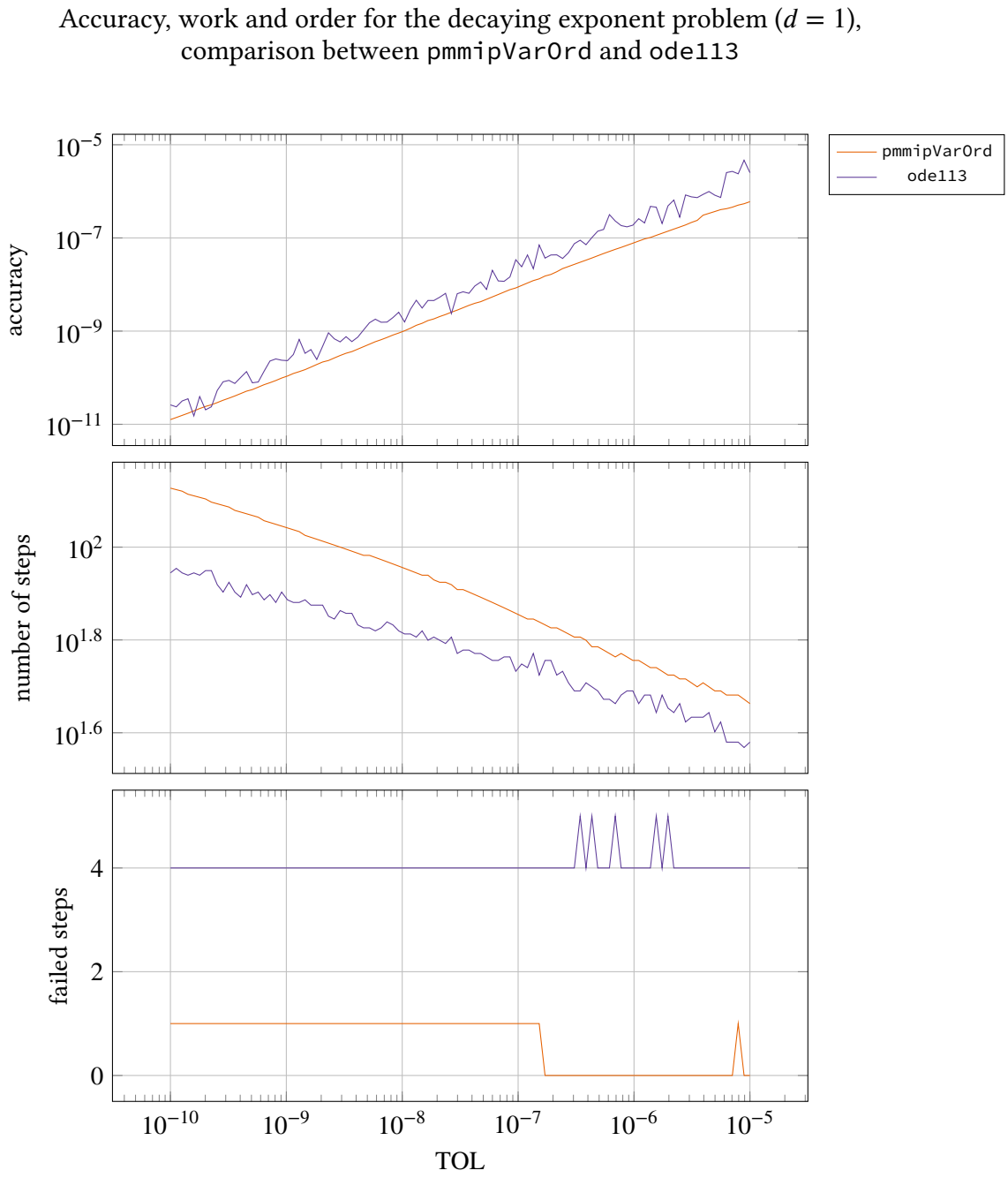

Figure IV.31. Performance data comparing `pmmipVarOrd` and `ode113`, when solving the decaying exponent problem with 100 different tolerance settings. The accuracy achieved by `pmmipVarOrd` is somewhat better, but `ode113` uses less amount of work. Note the different slopes of the work curves. The accuracy curve produced by `pmmipVarOrd` is much smoother than the one produced by `ode113`, indicating a better computational stability.

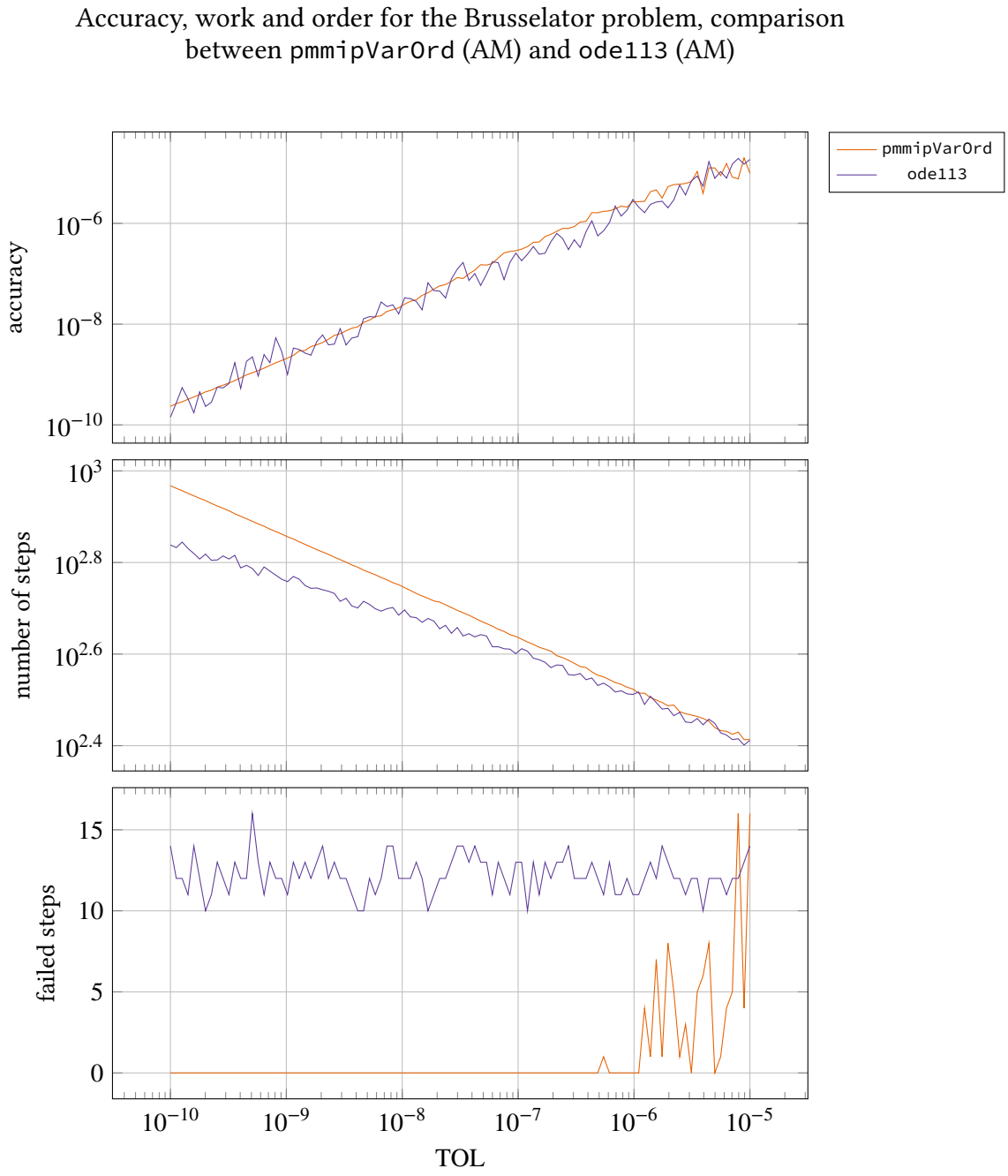

Figure IV.32. Performance data comparing `pmmipVarOrd` and `ode113`, when solving the Brusselator problem with 100 different tolerance settings. The accuracy achieved by both solvers is the same, but `ode113` uses less amount of work for stricter tolerances. The accuracy curve produced by `pmmipVarOrd` is much smoother than the one produced by `ode113`, indicating a better computational stability.

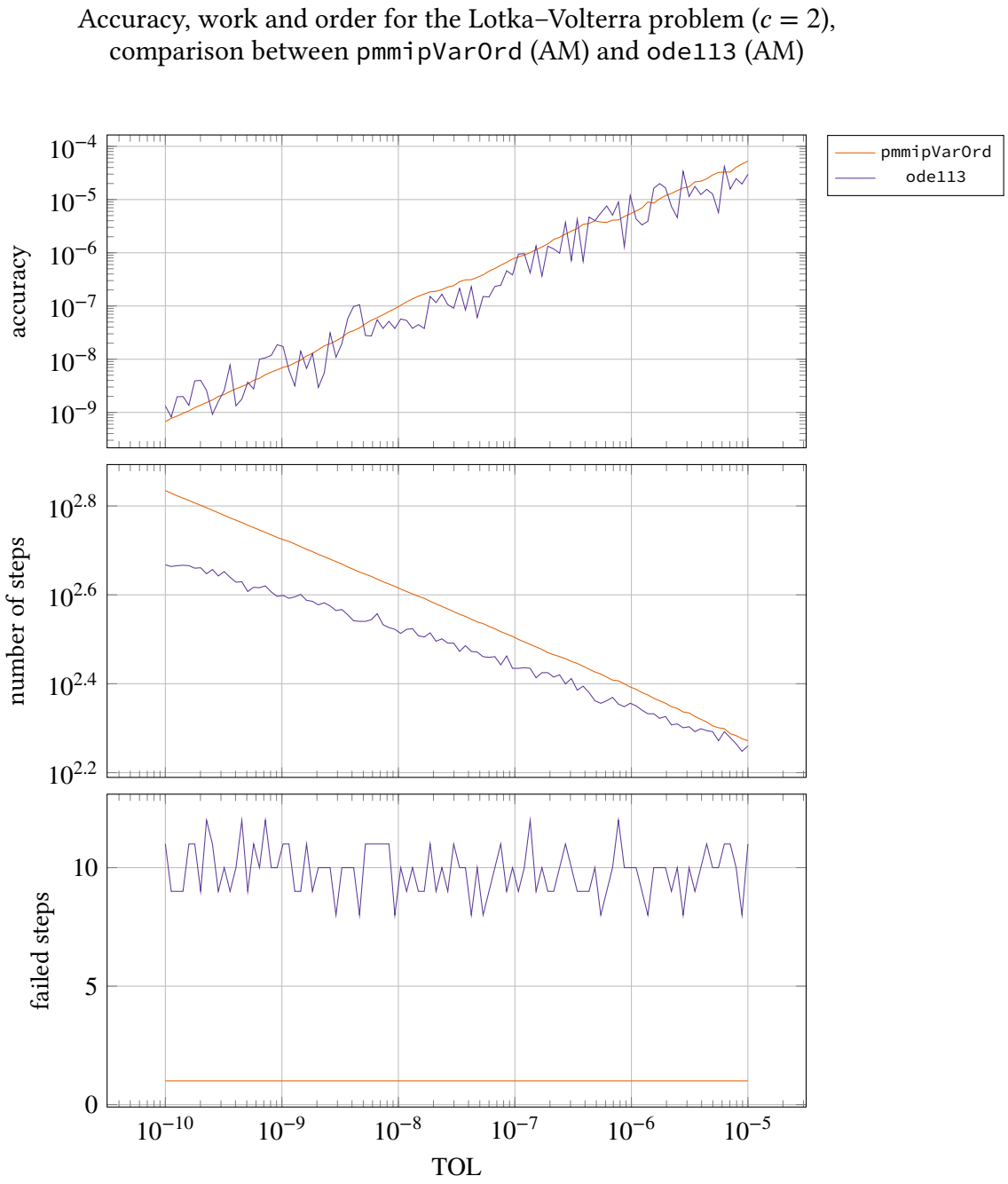

Figure IV.33. Performance data comparing `pmmipVarOrd` and `ode113`, when solving the Lotka–Volterra problem with 100 different tolerance settings. The accuracy achieved by both solvers is the same, but `ode113` uses less amount of work for stricter tolerances. The accuracy curve produced by `pmmipVarOrd` is much smoother than the one produced by `ode113`, indicating a better computational stability.

Accuracy, work and order for the Van der Pol problem ($\mu = 10$),
comparison between `pmmipVarOrd` (AM) and `ode113` (AM)

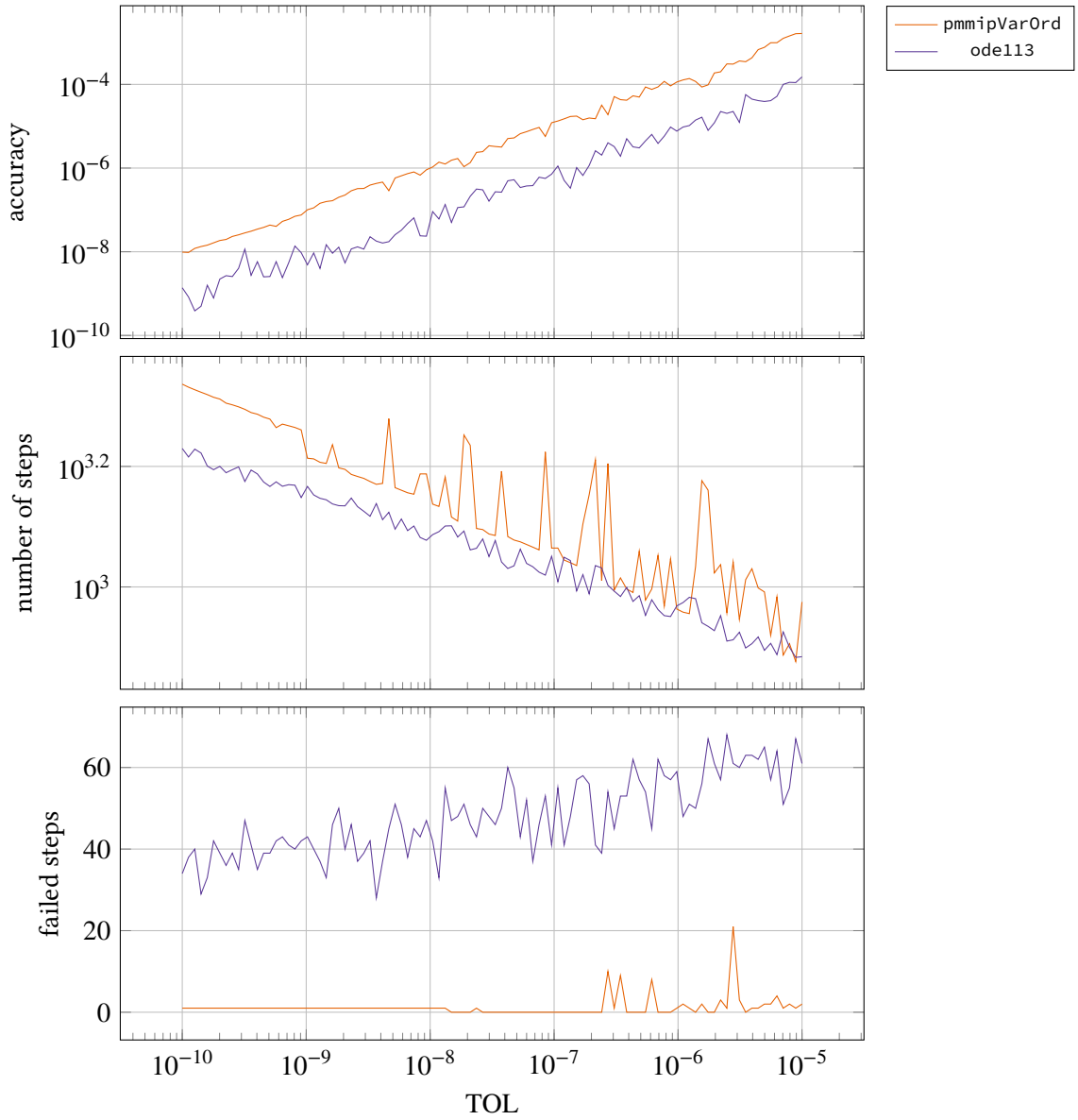

Figure IV.34. Performance data comparing `pmmipVarOrd` and `ode113`, when solving the Van der Pol problem with 100 different tolerance settings. The accuracy achieved by `ode113` is better and `ode113` uses less amount of work for stricter tolerances. The accuracy curve produced by `pmmipVarOrd` is much smoother than the one produced by `ode113`, indicating a better computational stability.

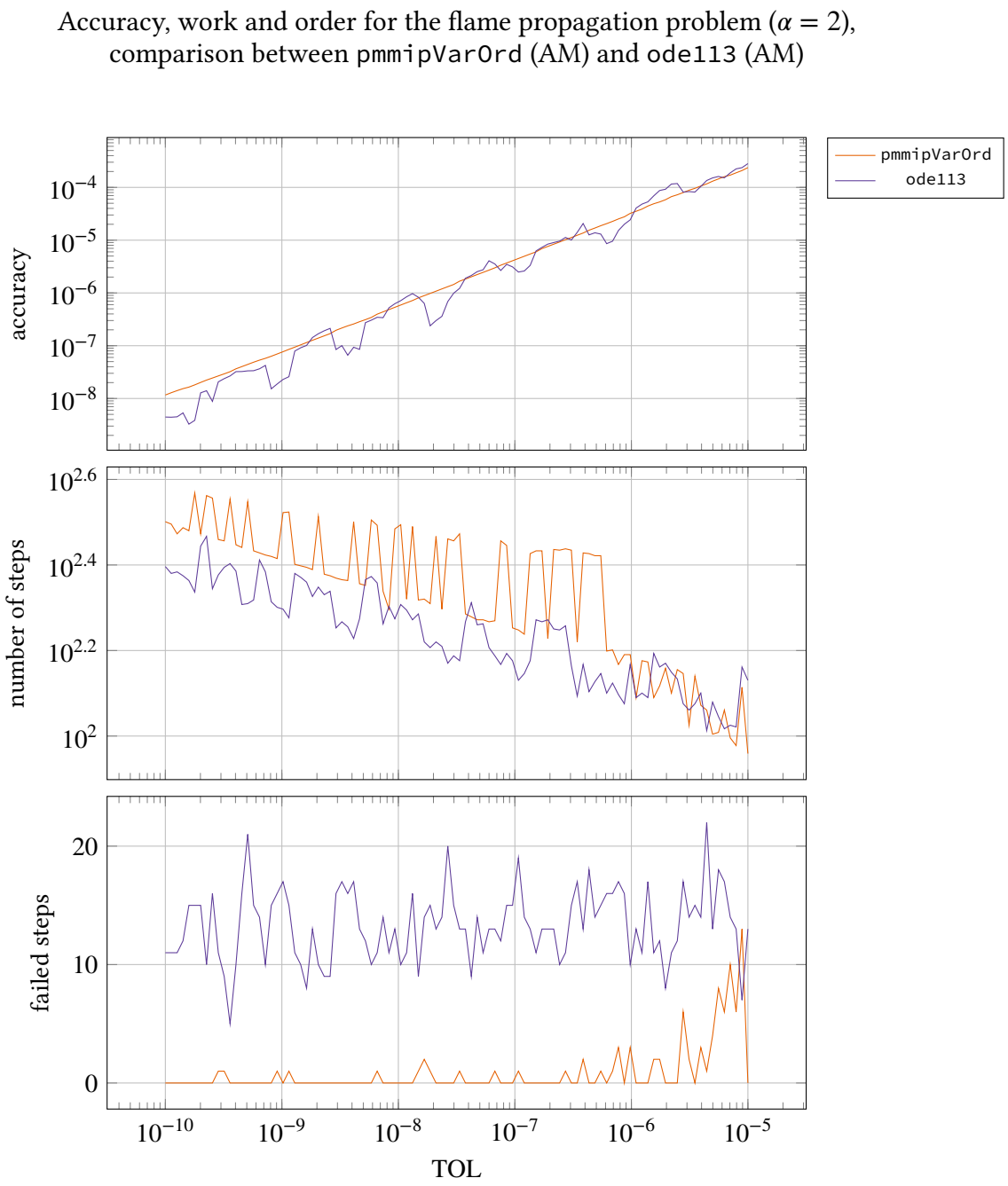

Figure IV.35. Performance data comparing `pmmipVarOrd` and `ode113`, when solving the flame propagation problem with 100 different tolerance settings. The accuracy achieved by both solvers is the same, but `ode113` uses less amount of work for all but the looser tolerances. The accuracy curve produced by `pmmipVarOrd` is much smoother than the one produced by `ode113`, indicating a better computational stability.

Accuracy, work and order for the decaying exponent problem ($d = 100$), comparison between `pmmiVarOrd` (BDF) and `ode15s` (BDF)

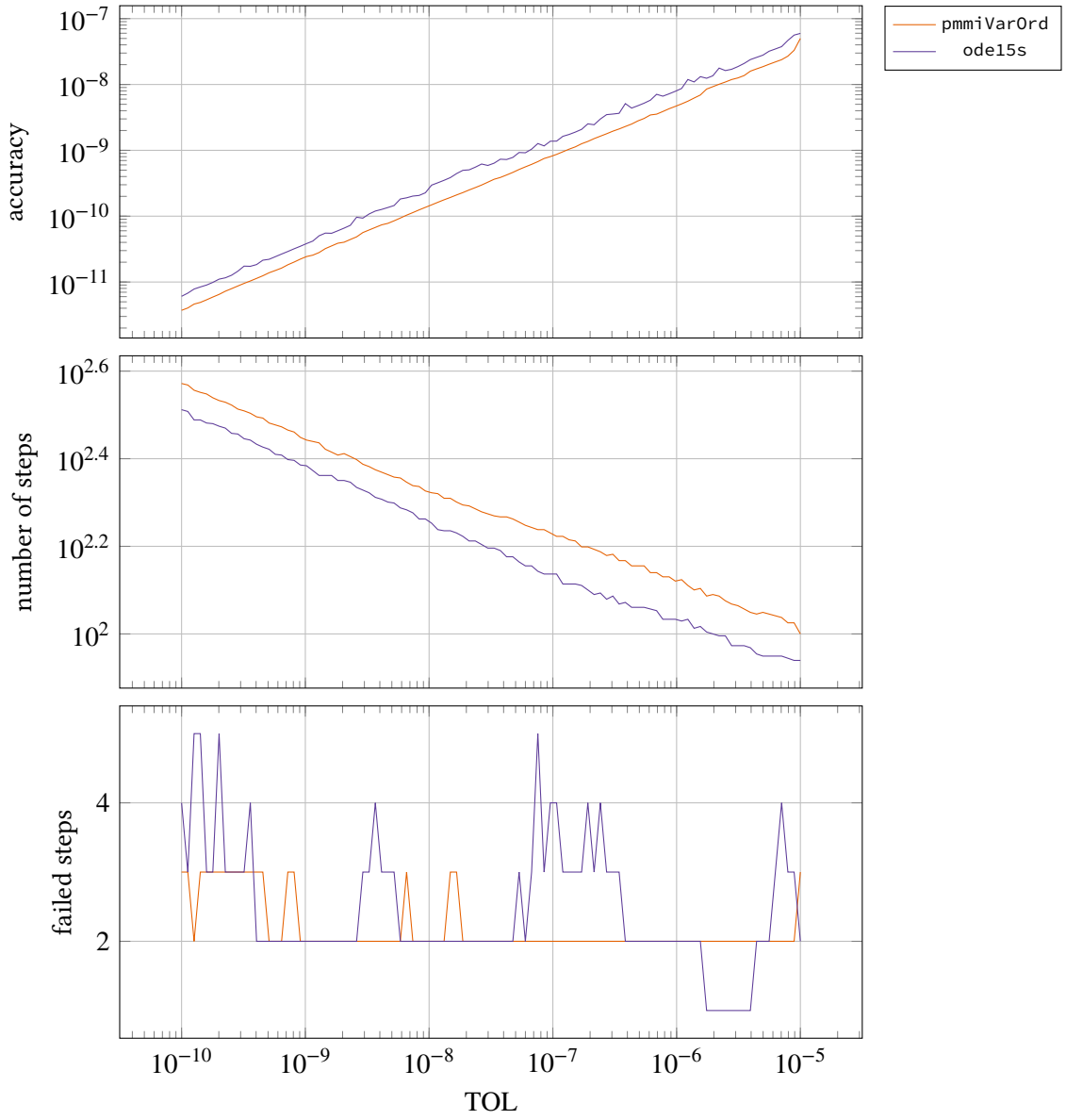

Figure IV.36. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the decaying exponent problem with 100 different tolerance settings. `pmmiVarOrd` achieves a slightly better accuracy, but also uses slightly less work. When taking this into account the performance is about the same for both solvers. Both accuracy curves are straight and smooth, indicating computational stability.

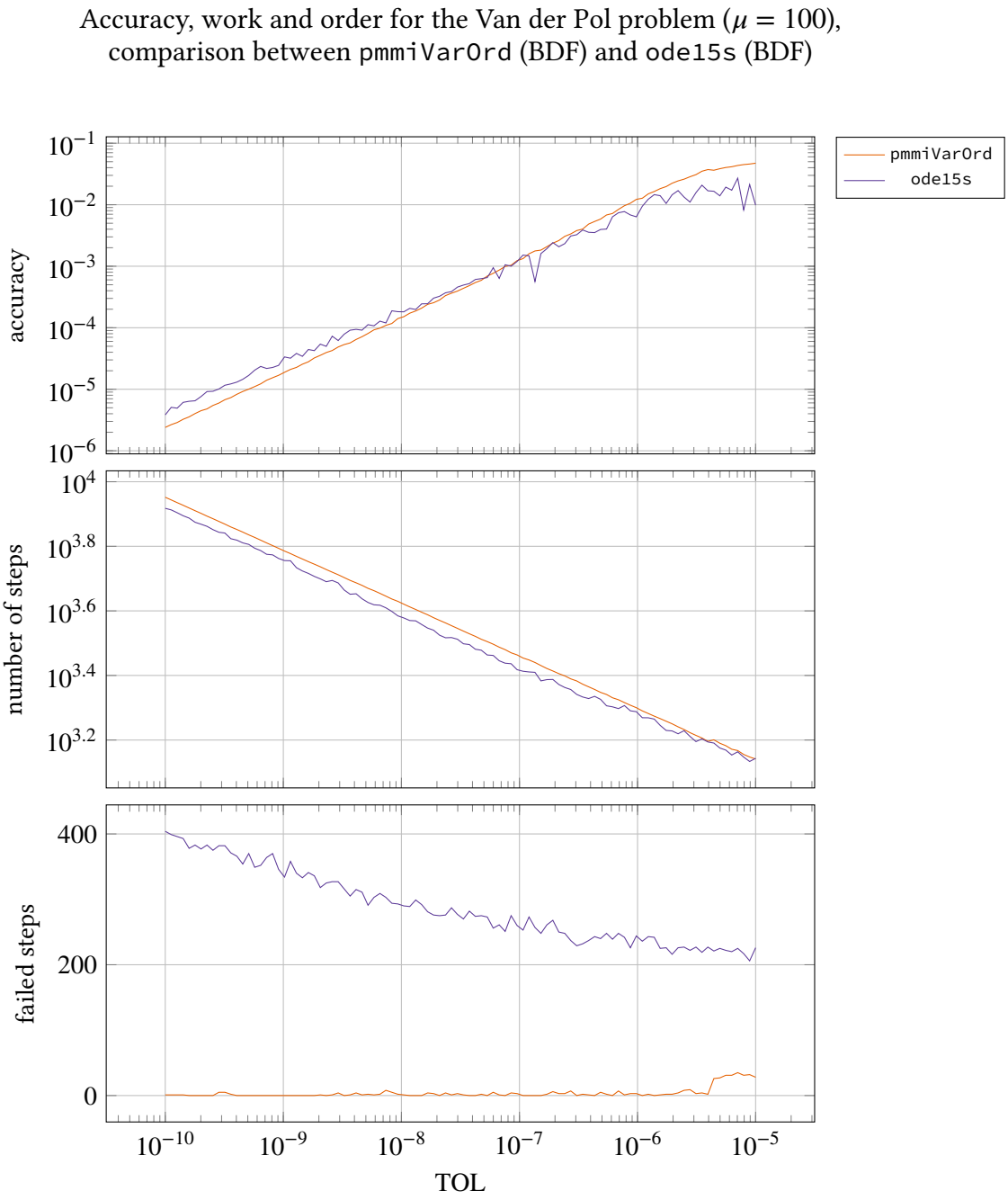

Figure IV.37. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the Van der Pol problem with 100 different tolerance settings. The achieved accuracy and amount of work is almost the same for both solvers. `pmmiVarOrd` produces a much smoother accuracy curve, indicating a better computational stability. `ode15s` has a much larger amount of failed steps compared to `pmmiVarOrd`.

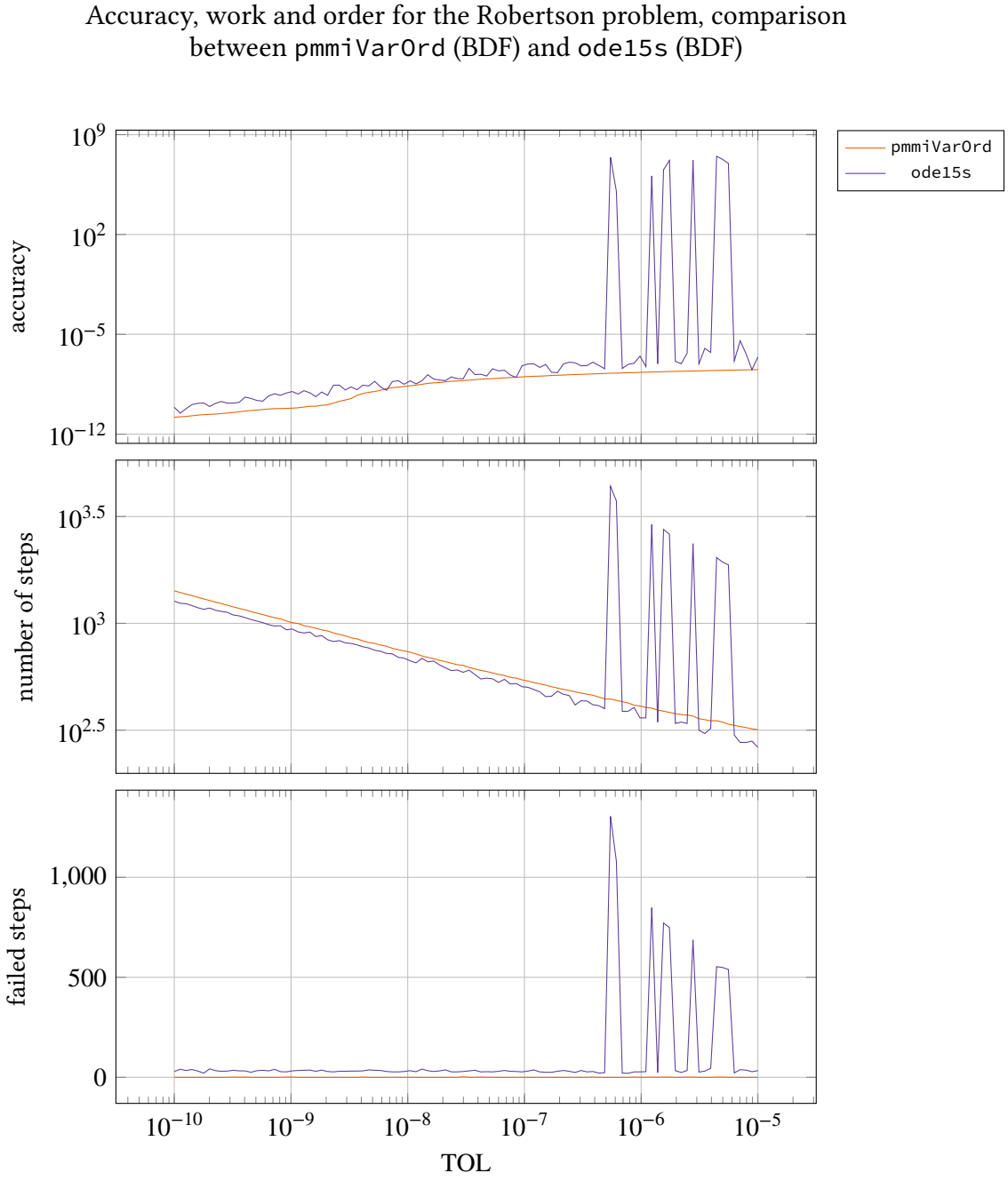

Figure IV.38. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the Robertson problem with 100 different tolerance settings. `ode15s` produces very large jumps in the error curve for the less strict tolerances, indicating that the solver is not able to produce a solution. This is accompanied by a large amount of work and failed steps. Due to the scaling the quality of the curves for the stricter tolerances can not be evaluated, and therefore these parts are shown in Figure IV.39.

Accuracy, work and order for the Robertson problem, comparison between `pmmiVarOrd` (BDF) and `ode15s` (BDF) (excluding the least strict tolerances)

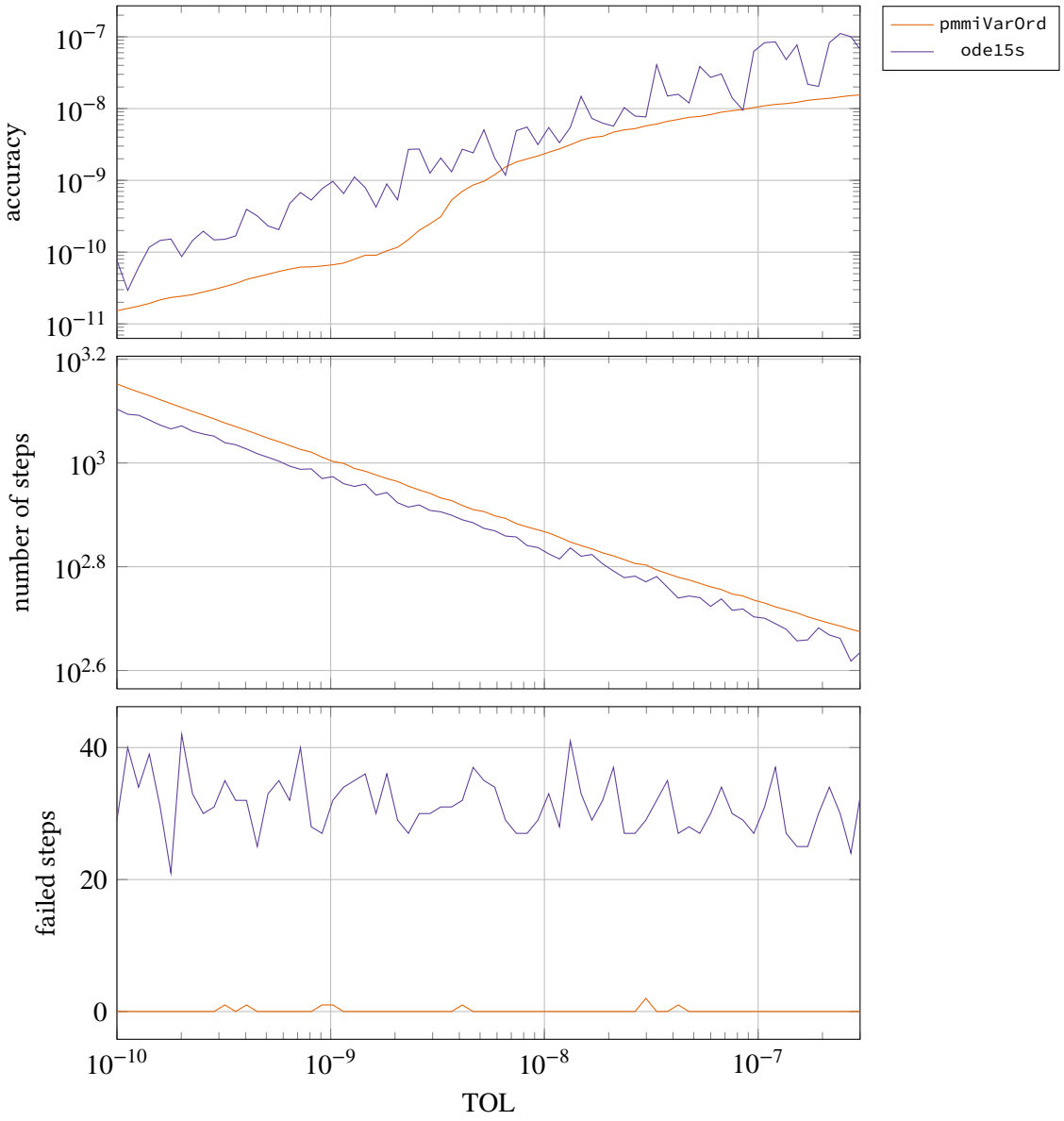

Figure IV.39. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the Robertson problem and excluding the least strict tolerance settings. The amount of work is comparable and `pmmiVarOrd` achieves a greater accuracy. None of the curves are smooth, although the quality of the curve produced by `pmmiVarOrd` is somewhat better. The Robertson problem is, as can be seen here, a hard problem to solve.

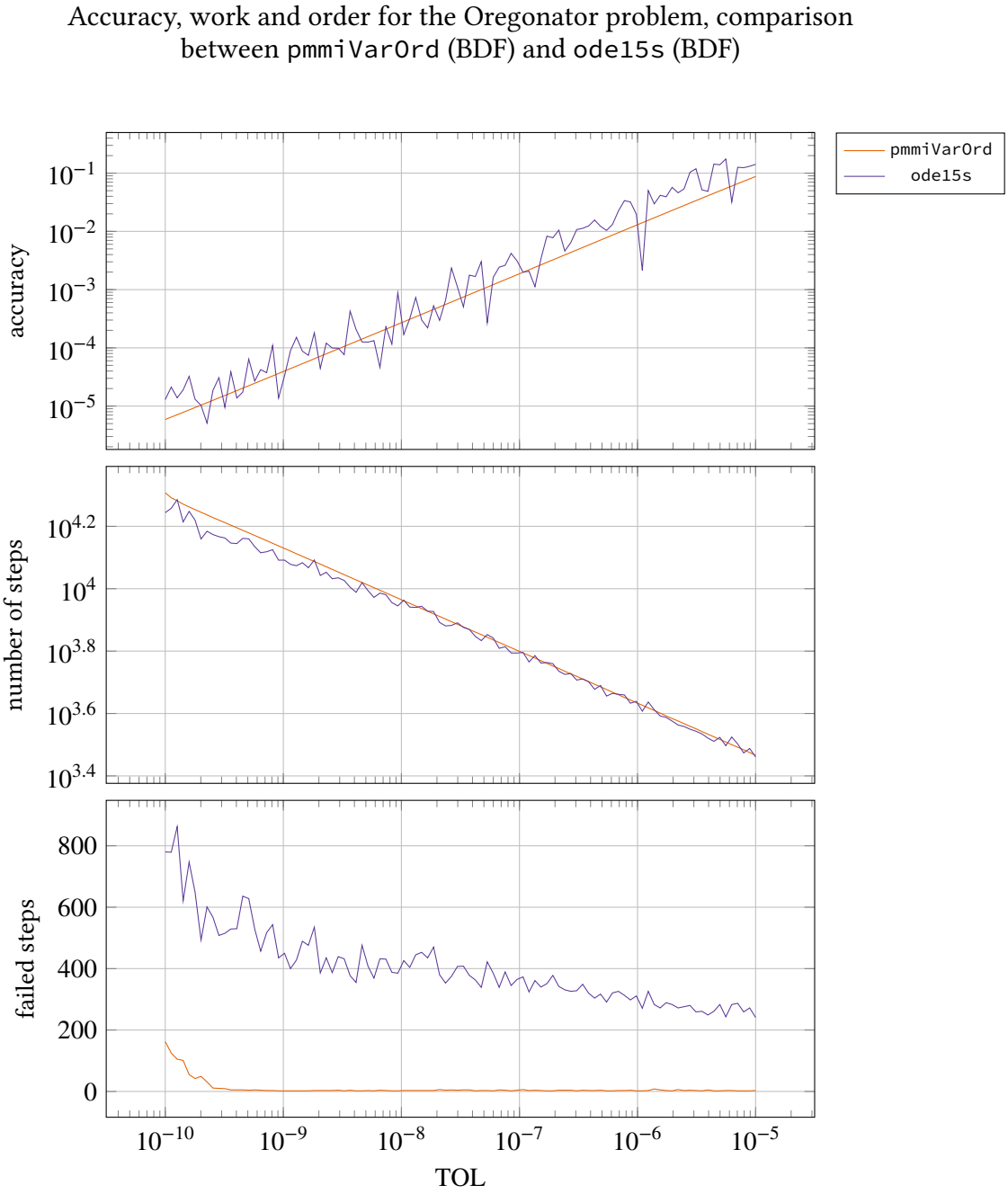

Figure IV.40. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the Oregonator problem. The achieved accuracy and amount of work done is the same for both solvers. `pmmiVarOrd` produces a much smoother accuracy curve, indicating a better computational stability. `ode15s` has a much larger amount of failed steps compared to `pmmiVarOrd`.

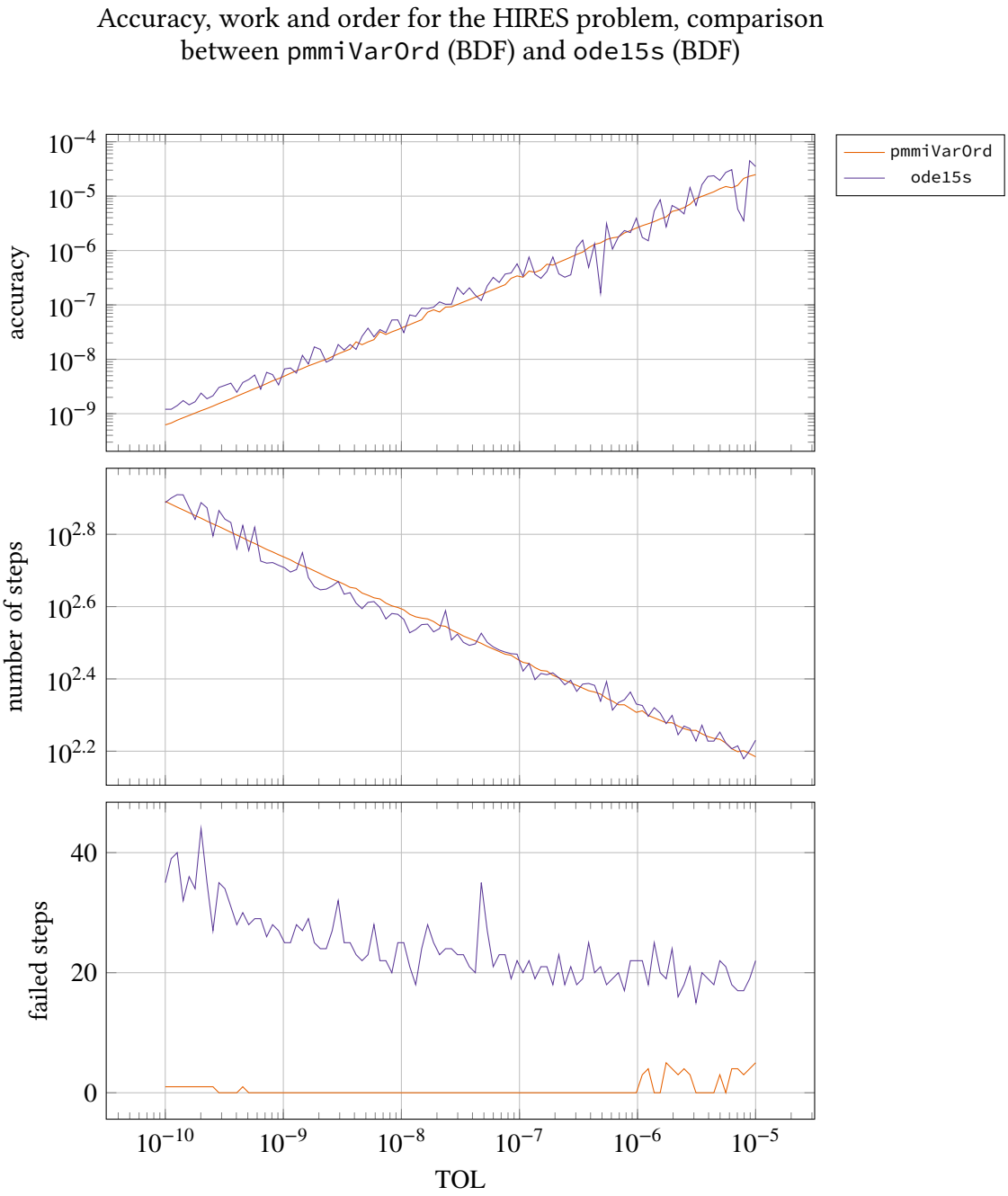

Figure IV.41. Performance data comparing `pmmiVarOrd` and `ode15s`, when solving the HIRES problem. The achieved accuracy and amount of work done is the same for both solvers. `pmmiVarOrd` produces a much smoother accuracy curve, indicating a better computational stability.

IV.3.5. Performance when allowing higher order methods

It was previously mentioned that due to performance reasons the maximum order was limited to 8 for the non-stiff solvers. It sometimes happens that when higher orders are allowed the solver increases the order, and then stays at this order even though it would be beneficial to use lower order methods. Such a behavior can be seen in Figure IV.42 (this example is only for `pmmipVarOrd`, but `pmmeVarOrd` exhibits similar behavior when using higher orders), where data is shown from two tests: one where the maximum order is limited to 8 and one where the maximum order is limited to 10. As can be seen in this figure the test run where the higher limit was allowed gets stuck on order 9 and 10. The accuracy for this test is somewhat better, but at the cost of a large increase in the amount of work (which can be seen by looking at the step-size sequence).

The most probable explanation is that at the higher orders the difference in step-size, between the order currently used and its neighboring orders, is so small that the algorithm does not change the order. The algorithm is designed to avoid chatter and therefore it does not change order when the benefits are too small. Also, the algorithm does not compare any other orders than those adjacent to the one currently in use, so even if there would be a great advantage to go down to a very low order it would lack information about this.

Some possible solutions to this could be to allow comparisons with more than 2 other orders, or adding some compensating factor which makes the algorithm change orders faster when a high order is in use. The first approach though will demand more computational power. The second approach is better in this regard, but it is not certain that it is possible to design one algorithm which fits all types of methods. Different methods have different characteristics which affect the error, e.g., their different error constants. It is not even the case that the error constant always decreases with an increase in order, as can be seen in Figure II.6, where it is shown that the error constants of EDF-methods actually increase.

There is not enough data to draw firm conclusions about the following, but from the data we have there are two interesting observations that can be made. The first is that the difficulties with the order changes, seem to be more prevalent in the case of `pmmipVarOrd`, than with `pmmeVarOrd`. This may indicate that the method used is relevant, which was discussed in the previous paragraph. The other observation is that both of the problems where we see this issue are moderately stiff problems.

The limit chosen for the maximum allowed order in the tests is not a hard limit, it depends on the problem and the settings of the solver. For these tests, the limit of 8 seems to be working well for `pmmeVarOrd`. For `pmmipVarOrd` it still poses problems in two cases, see Figure IV.24 and IV.25.

The difference between using a maximum order limit of 8 and 10, when solving the Van der Pol problem ($\mu = 10$) using `pmmipVarOrd`

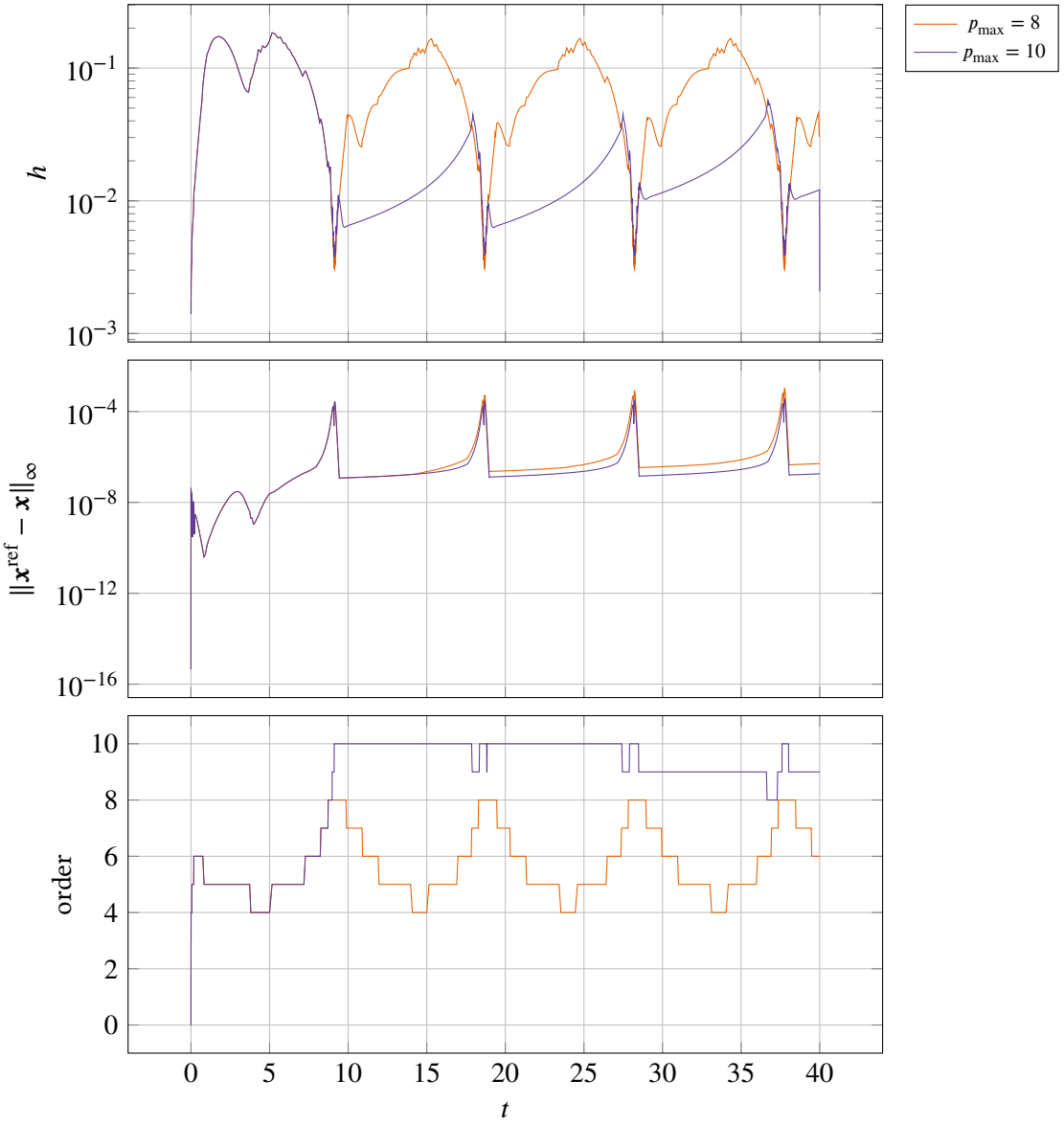

Figure IV.42. The difference when solving the Van der Pol problem using different maximum order limits, $p_{\max}$, with `pmmiVarOrd`. Except for the change in maximum order, the tests where done using the settings shown in Table IV.4, and using the tolerance 10^{-7}. Note how when $p_{\max} = 10$ the order gets stuck at 9 and 10. This is accompanied by a rapid decrease in step-size which leads to a large increase in the amount of work done.

Part V.

Conclusions and future work

V.1. Conclusions

The first question stated in the introduction — *Does the concept of using digital filters to control the step-size, give good results when implemented in conjunction with linear multistep methods?* — can be answered (in the case of the fixed order solvers) by the results in Subsection III.4.1 and Subsection III.4.2. When the solvers are given a periodic problem, they produce both periodic step-size sequences and periodic control error sequences which is a good sign. The controller keeps the control error at appropriate values around the specified tolerance and the step-size ratio sequence inside the allowed interval resulting in only a few (if any) rejections, which is a really good result since a rejection means a restart of the controller.

The fixed order solvers are very robust upon tolerance changes, at least in the case of work and accuracy according to what is seen in Subsection III.4.2, which is an important quality. Higher order methods need less work for the same accuracy, and the same is true when comparing the non-stiff implicit solver to the non-stiff explicit solver, which is what we expect as long as the problems are non-stiff.

The answer to question two in the introduction — *Do different combinations of solvers, methods, filters and problems behave equally well, or are some combinations preferred?* — is: Generally, no significant difference is found. According to what we have seen in the fixed-order tests, the different solver-method-filter-problem-combinations seem to behave about equally well in most cases. The only time we have gotten really bad results is when we have tried to use the methods ABx ($x > 5$), EDFx ($x > 5$) and BDF6. However, using these methods we have not found any special combinations that seem to be worse than others, instead they seem to behave pooly in general. In addition to running them in combination with most of the different filters, we have tried to run them using a stricter step-size ratio interval, however, this did not seem to improve the results at all. A further investigation using these methods is needed.

As said above, in general most SMFP-combinations seem to work well. However, in Subsection III.4.3, we see that an increase in stiffness (d goes from 1 to 10, making the problem moderately stiff) seem to give indications that the filters PI3040, PI3333, and PI4020 are better suited for moderately stiff problems. These results are not conclusive, as more tests on other moderately stiff equations have to be performed, but it shows that this may be an interesting avenue for further research.

There were two questions stated in the introduction regarding the order regulation, and the first of these questions dealt specifically with whether or not the proposed algorithm worked as intended. The first tests, in Subsection IV.3.2, shows that the variable order solver manages to produce accurate solutions, while actively regulating both the step-size and order. When used to integrate periodic problems the solver shows a good periodic behavior in regards to which orders are used, and this indicates that the algorithm is stable and manages to regulate in the same way, even though there are small differences as is the case when comparing different cycles in the same problem.

When looking at the flame propagation problem, which is a moderately stiff problem where there is a clear point at which the problem goes from being non-stiff to being stiff, the explicit solver decreases its order in the stiff region, which is something which can be expected from what we know about fixed step-size stability analysis. The same thing happens for the implicit solver, with the exception that it gets stuck at a higher order, for some tolerances.

The second set of tests, in Subsection IV.3.3, shows a good accuracy response when changing the tolerance, especially for the two implicit solvers for which the results are good (and in the case of `pmmiVarOrd`, excellent). Even though the results for the explicit solver are good, they need to be improved a bit to reach the same quality as the results produced by the implicit solvers. Up until now, the work curves have only been discussed when looking at the amount of work, but it is also worth mentioning that it is desirable that they also are smooth and straight, as this gives a more predictable efficiency behavior and also indicates computational stability. As discussed in Section II.5 this should not be prioritized above the accuracy behavior, but as can be seen from the results the curves are smooth and straight, with the exception of Figure IV.25 and Figure IV.24.

When comparing the performance of the two non-stiff solvers it is also shown that the implicit solver outperforms the explicit one in regards to the accuracy achieved for a certain amount of work done. This is an expected outcome, as the implicit methods do the same in the fixed order case.

The mean order also shows the trend that it increases when the tolerance becomes more strict (with the exception of Figure IV.36 for which this behavior has already been explained). As shown by the results from the fixed order solvers, the difference in the amount of work needed to reach a certain accuracy increases when the tolerance becomes stricter, and therefore a preference for higher order methods at these tolerances are expected even in the variable order case.

During the tests, two problems were discovered: One regarding how the methods are started, which a further test indicates is not due to problems with the order regulation and this will discussed in Section V.2, and the other regarding the behavior when allowing high order methods for solving problems. Tests showed that for certain problems the non-stiff solvers increased the order, but did not decrease it even though it would be beneficial. This indicates that the algorithm needs to modified in some way, as to also allow the use of higher order methods without exhibiting this behavior. Almost all the problems noted in the discussion about the results in Section IV.3, can be traced to this.

All the things mentioned above are on their own not enough to draw a conclusion, but together they paint a picture which shows that the main idea behind the algorithm is sound. If there is a solution to the problem with the order getting stuck, the algorithm should work as intended. Most importantly, even when the order gets stuck, this has a very small effect on the accuracy, which is a very good property and shows some of the benifits to using control theory when regulating the step-size.

The last question in the introduction, about whether or not the solvers constructed have the potential to compete will already established solvers, was investigated by comparing our variable order solvers with two of the solvers implemented in Matlab. When looking at the curves showing the accuracy, the curves produced by `pmmipVarOrd` and `pmmiVarOrd` are in all cases, except one, smoother than those produced by `ode113` and `ode15s`. As has been discussed in Section II.5 this is an important feature as this indicates robustness.

When comparing how much work is done for an achieved level of accuracy, `ode15s` and `pmmiVarOrd` performs equally well, whereas `ode113` outperforms `pmmipVarOrd`, especially at stricter tolerances. It has already been noted that the difference in slope here may be explained by the difference in the maximum order limit. Still, as long as the underlying problem for limiting the order at 8 for `pmmipVarOrd` the conclusion must be that this solver is less efficient. In the stiff case though, where the methods have a lower natural limit, the solvers are equally efficient.

We conclude, based on the reasons stated above, that the principles upon which the solvers tested in this thesis have been built are sound and have the potential to compete with already established solvers. This is not to say that they are production ready today, which is a conclusion that can not be made. These are early tests, this version of the software package is meant for scientific purposes, and there are as noted some outstanding problems.

V.2. Future work

It has previously been shown that the project of implementing a numerical solver involves a lot of different parts. However, the focus of this thesis has been to investigate the regulation of step-size and order, which means that the other parts have been subjected to less examination. In this section we will outline some of these subjects, and give ideas to interesting areas for future work.

As already mentioned, we have found a problem in the variable order solvers regarding the solvers getting stuck at higher order methods. We feel confident that this problem is solvable, but further work is required. A couple of solution proposals have already been discussed, the use of a compensation factor at higher orders, or to calculate the step-size sequence for more than 3 orders at a time, but these are not necessarily the only, or the best, solutions. Changes could, for example, also be made to the integrator (Equation IV.13). The introduction of a forgetting mechanism, could make the algorithm place greater weight on more recently calculated values.

Another issue already mentioned, touches upon the start-up phase in the non-stiff solvers. At the moment, we always accept solution point $k + 1$, i.e., the first solution point created by the k-step LMM. The reason for this is that we need to calculate $k + 1$ solution points before we can use the main predictor, and we do not have any good initial predictor to use instead. An example of a case when this has caused problem, can be seen in Figure IV.3. Here the proposed initial step-size is too large, however, since the first step taken by the LMM is always accepted, this leads to a severely diminished accuracy. In the stiff solvers, this is not a problem, since the main predictor is available already at step k (during which solution point $k+1$ is calculated). A couple of possible solutions worth trying out are the following:

1. The *starter*, i.e., the solver/method that generates $k - 1$ solution points in the beginning enabling the main method (k-step method) to start, could generate one additional solution point. This would make it possible to start the LMM and the main predictor at the same time (at step $k + 1$).

2. The use of an *initial predictor*, that is, a special predictor used exclusively at the first step of the LMM. This would enable an error estimate to be calculated at step k.

3. The error controller is still turned off at step k, but if step $k + 1$ is rejected, both step k and step $k + 1$ are rejected and recalculated using a shorter step-size.

The first proposal has the merit of being very easy to implement, however it is desirable to start the main method as soon as possible. This solution would also prohibit the variable order non-stiff solvers to be *self-starting* since no error prediction can be made at the first step by the 1-step method. The third proposal is harder to implement than the first, and it might not solve everything. What if the local error in step k is large (and therefore should be rejected), but the

local error in step $k + 1$ is small? This would lead to no rejection being made, and accuracy problems will still be introduced. Alternative 2 seems to be the best solution. It is both easy to implement, and would make it possible to use the main method already at step k. However, it is important that a good initial predictor is chosen, otherwise it might result in an under-estimated error, which could lead to an even worse accuracy.

An interesting area to investigate is the area of making the implementations more efficient. For example, one could try to parallelize the order regulation, which would be possible since the integration process of one step using different methods are independent of each other. This could especially be interesting if only one ODE is to be solved. If the solver is part of a larger problem that involves solving multiple ODEs, all available calculation resources might already be in use. Though, in the case of only one ODE, this parallelization could enable the use of more simultaneous order comparisons. For example, when using the BDF-family in the stiff variable order solver, one could integrate and do comparisons between all stable BDF-methods at the same time, enabling the solver to directly change from the highest to the lowest order. This would, of course, require modifications to the order regulation algorithm.

Another interesting idea, would be to create a solver that is *stiffness-adaptive*, meaning that the solver mid-integration can change method from one being stiff to one being non-stiff depending on the stiffness of the problem at that particular time. This would especially be beneficial when solving a problem containing both non-stiff parts and very stiff parts. Such an implementation could be based on the order regulation algorithm, which already compares different methods (although with the focus on the methods being of different orders) and decides which is most effective. Another approach could be to monitor the stiffness throughout the problem, and change method when the character of the problem changes. Calculating the stiffness involves finding the extreme eigenvalues of a symmetric matrix (the symmetrized Jacobian), and as the linear system of equations, which is to be solved in the Newton iterations, in most cases has a larger dimension than the Jacobian, this would lead to less calculations, because both the solving of a linear system of equations (by LU-factorization) and the calculation of the extreme eigenvalues (by transforming the matrix to Hessenberg form using Householder reflectors, and then using the bisectional algorithm and Sturm sequence to find the two eigenvalues of interest) can be done in cubic time [28]. An easy way to try the latter method is to explicitly calculate the stiffness function for a specific problem and during the integration of this problem evaluate this function. Start the integration using the non-stiff solver, and evaluate the stiffness at every time step. If the stiffness is less than a value x, then change to the stiff solver, and change back if the stiffness gets greater than x again. However, one problem with the approach to calculate the stiffness at every time step, is that the step-size regulation process when changing method has to be reset somehow, since no parallel step-size regulation process has been run.

When the step-size is regulated, it is sometimes necessary to override the controller, for example, upon rejections, or when the controller suggests a step-size increase that is too large. In these situations an anti-windup scheme may be employed to improve the results. In the current implementation the bypass functionality, which is employed upon rejections, works as an anti-windup, but besides this there are no other schemes applied. The results shown in this thesis are good, but a more sophisticated anti-windup scheme may have the benefit of improving on these even more. Further work can be done here by developing and testing such a scheme.

The way the implicit solvers solve the non-linear systems of equations that arise, are currently only done by Newton's method. Here other schemes may be tried, for example, fixed point itera-

tion. The implementation and evaluation of predictor-iteration schemes would also be interesting to experiment with, especially since the fact that reusing old polynomials are equivalent to using explicit methods as predictors in the implicit solvers. The use of Adams–Moulton methods, for example, automatically give Adams–Bashforth methods as predictors.

Further, the current software package implements the different solvers as 6 separate functions (3 of fixed order and 3 of variable order), all using slightly different interfaces. It might be beneficial to the user, if the solvers were unified to one solver, or at least if the variable order and fixed order solvers were unified to 3 solvers. Another improvement beneficial to the user, would be to make the solvers faster. This could be done by implementing the most time-consuming parts in a compiled language (as is possible with Matlab). Especially the functions constructing the polynomials for the implicit solvers are slow in the current implementation. By rewriting these parts in for example C, we could probably make the solvers faster. This is something which could be done in Matlab by constructing so called MEX-files [13].

Apart from the work ideas focused on implementation given above, we have other suggestions on areas to investigate. In this thesis the fixed order solvers were tested with many of the methods and filters implemented in the libraries. In these tests no significant difference in how the combinations of solvers, methods and filters performed could be found. All combinations seemed to work very well together, except in the case of the methods BDF6, AB6 and EDF6. All of these have been problematic to use, and there do not seem to be any link between what filter is used in conjunction with the methods, and their behavior. Neither did a tighter step-size ratio interval seem to make any difference. However, the subject is not thoroughly investigated. It needs to be further analyzed.

The allowed step-size ratio interval is another area we think should be further investigated. As for now, the default interval is set to $[0.8, 1.2]$, regardless of method, filter and other settings used. We believe that it might be beneficial to choose different default intervals depending on settings. For example, the order of the method, or the tolerance, could have an influence on what interval to choose. Different methods are also subject to becoming unstable at different changes of step-size, which motivates the limits of the interval to be method-dependent as well.

Bibliography

[1] C. Arévalo and G. Söderlind. “Grid-independent construction of multistep methods”. To appear in: J. Comp Math. 2016.

[2] C. Arévalo, G. Söderlind, and C. Führer. “Regular and singular β-blocking of difference corrected multistep methods for monstiff index-2 DAEs”. In: *Appl. Numer. Math.* 35.4 (2000), pp. 293–305. ISSN: 0168-9274.

[3] C. Arévalo et al. *VOSS software package (thesis version)*. 2016. URL: `https://gitlab.com/examensarbete/voss-thesis-version`.

[4] K. Atkinson, W. Han, and D. Stewart. *Numerical Solution of Ordinary Differential Equations*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. Wiley, 2011. ISBN: 9781118164525.

[5] T. P. Coffee, J. M. Heimerl, and M. D. Kregel. *A Numerical Method To Integrate Stiff Systems Of Ordinary Differential Equations*. Tech. rep. US Army Armament research and development command ballistic research laboratory arberdeen proving ground, Maryland, 1980.

[6] N. S. Dattani. *Linear Multistep Numerical Methods for Ordinary Differential Equations*. arXiv:0810.4965. 2008.

[7] L. Euler. *Institutionum calculi integralis*. Institutionum calculi integralis v. 1. imp. Acad. imp. Saènt., 1768. URL: `https://books.google.se/books?id=Vg8OAAAAQAAJ`.

[8] K. Gustafsson. “Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods”. In: *ACM Trans. Math. Softw.* 17.4 (1991), pp. 533–554. ISSN: 0098-3500.

[9] K. Gustafsson. “Control-theoretic techniques for stepsize selection in implicit runge-Kutta methods”. In: *ACM Trans. Math. Softw.* 20.4 (1994), pp. 496–517. ISSN: 0098-3500.

[10] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I – Nonstiff Problems*. 2nd. Springer, 2008. ISBN: 978-3-540-56670-0.

[11] A. Iserles. *A first course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 2009.

[12] F. Mazzia and F. Iavernaro. *Test Set for Initial Value Problem Solvers*. Tech. rep. Department of Mathematics, University of Bari, Italy, 2003.

[13] *mex*. Matlab version R2016b. URL: `https://se.mathworks.com/help/matlab/ref/mex.html` (visited on 2016-10-14).

[14] *mldivide, *. URL: `http://se.mathworks.com/help/matlab/ref/mldivide.html` (visited on 2016-08-19).

[15] C. Moler. *Stiff Differential Equations*. 2003. URL: http://se.mathworks.com/company/newsletters/articles/stiff-differential-equations.html (visited on 2016-04-07).

[16] *ode113*. Matlab version R2016b. URL: https://se.mathworks.com/help/matlab/ref/ode113.html (visited on 2016-09-16).

[17] *ode15s*. Matlab version R2016b. URL: https://se.mathworks.com/help/matlab/ref/ode15s.html (visited on 2016-09-16).

[18] ODELab. *ODELab*. 2015. URL: http://num-lab.zib.de/public/cgi-bin/odelabnew?formid=1&userid=josefine (visited on 2016-04-07).

[19] *odeset*. Matlab version R2016b. URL: https://se.mathworks.com/help/matlab/ref/odeset.html (visited on 2016-09-16).

[20] O. Østerby. "Step change strategies for multistep methods". In: *DAIMI Report Series* 14.196 (1985). ISSN: 2245-9316.

[21] G. K. Rockswold. "Implementation of α-type multistep methods for stiff differential equations". In: *J. Comput. Appl. Math.* 22.1 (1988), pp. 63–69. ISSN: 0377-0427.

[22] L. Shampine. *Numerical Solution of Ordinary Differential Equations*. Chapman & Hall mathematics v. 4. Taylor & Francis, 1994. ISBN: 9780412051517.

[23] G. Söderlind, L. Jay, and M. Calvo. "Stiffness 1952–2012: Sixty years in search of a definition". In: *BIT Numerical Mathematics* 55.2 (2015), pp. 531–558. ISSN: 1572-9125.

[24] G. Söderlind. "A specification of a control system for adaptive time-stepping". Manuscript. 2005.

[25] G. Söderlind. "Digital filters in adaptive time–stepping". In: *ACM Trans. Math. Softw.* 29.1 (2003), pp. 1–26. ISSN: 0098-3500.

[26] G. Söderlind. *Personal communication*. 2016.

[27] G. Söderlind and L. Wang. "Evaluating numerical ODE/DAE methods, algorithms and software". In: *J. Comput. Appl. Math.* 185.2 (2006), pp. 244–260. ISSN: 0377-0427.

[28] L. N. Trefethen and D. Bau, III. *Numerical Linear Algerbra*. Society for Industrial and Applied Mathematics, 1997. ISBN: 978-0-898713-61-9.

Appendix

A.1. File structure and organization

All the code generated during the scope of this thesis project is contained in a Matlab software package called `VOSS`. Further, this is organized into multiple sub-packages, where each sub-package contains functions controlling the same aspects of the solver. The reason for the code structure division, is that we want to make the solvers modular such that the user is able to change some part of it, for example the system controlling the variable step-size, without altering anything else in the solver. The whole package is available for download on the web [3]. The file structure of the package is as follows:

```
VOSS
├── +solvers
│   ├── pmme.m
│   ├── pmmeVarOrd.m
│   ├── pmmi.m
│   ├── pmmiVarOrd.m
│   ├── pmmip.m
│   └── pmmipVarOrd.m
├── +init
│   ├── start.m
│   └── autostart.m
├── +control
│   ├── getWorkFactors.m
│   ├── errorController.m
│   └── getFilterVec.m
├── +solverFunctions
│   ├── polE.m
│   ├── polI.m
│   ├── polIp.m
│   ├── getMethodThetaFunc.m
│   ├── getJac.m
│   ├── getMethodVec.m
│   └── +thetas
│       ├── AB.m
│       ├── BDF.m
│       ├── EDF.m
│       ├── AM.m
│       ├── dcBDF.m
│       └── Nyström.m
└── +ode
    ├── +both
    │   └── …
    ├── +nonStiff
    │   └── …
    └── +stiff
        └── …
```

A.1.1. Package: solvers

This package consists of the actual solvers, i.e., the main functions. All together there are 6 solvers. The three numerical method types, E_k, I_k and I_k^+ (see Section II.2), have to be handled in slightly different ways, and are therefore implemented in separate solvers. Each type is also implemented in two different versions, one that uses fixed order, and one that uses variable order (indicated by the `VarOrd` extension in the function name):

- `pmme.m` – Solver using a type E_k method (e.g. Adams–Bashforth)
- `pmmeVarOrd.m` – Variable order version of the previous solver
- `pmmi.m` – Solver using a type I_k method (e.g. BDF methods)
- `pmmiVarOrd.m` – Variable order version of the previous solver
- `pmmip.m` – Solver using a type I_k^+ method (e.g. Adams–Moulton)
- `pmmipVarOrd.m` – Variable order version of the previous solver

All the above solvers have the following signature

```
function [t,x,statistics] = pmm*(f,tSpan,x0,varargin)
```

where `t` and `x` are the calculated time points and corresponding solution points, and `statistics` is a Matlab-struct containing information about the calculation, e.g., what method was used, what filter was used and CPU-time. The 6 different solvers have a few different optional in-parameters as well.

A.1.2. Package: solverFunctions

For the solvers to work, we have implemented a few help functions, all contained in this sub-package. One notable thing is that the actual functions calculating the polynomial coefficients, are implemented here as separate functions, instead of being implemented directly in the solvers. This gives a wider flexibility both in this project, and future ones based on solvers using this multistep method parameterization. The functions contained in this sub-package are the following:

```
function [jac] = getJac(f,t,x)
```

This function calculates the Jacobian `jac` of a function `f(t,x)` using central finite differences. This method is used in the implicit solvers when no analytical Jacobian is supplied.

```
function [theta] = getMethodVec(method)
```

This function returns the parameter vector, the θ-vector, for a given method. It is used by all fixed order solvers, which enables the user to request a method by name, instead of by parameter vector. The library consists of the following multistep methods:

- Explicit methods of order k (method class E_k):
  - AB1, AB2, AB3, AB4,...
  - EDF1, EDF2, EDF3, EDF4,...
  - Nystrom3, Nystrom4, Nystrom5

  - EDC22, EDC23, EDC33, EDC24, EDC34, EDC45
- Implicit methods of order k (method class I_k):
  - Kregel
  - BDF1, BDF2, BDF3, BDF4, BDF5, BDF6
  - Rockswold
- Implicit methods of order $k+1$ (method class I_k^+):
  - Milne2, Milne4
  - IDC23, IDC24, IDC34, IDC45, IDC56
  - AM2, AM3, AM4,...
  - dcBDF2, dcBDF3, dcBDF4,...

```
function [thetaFunction] = getMethodThetaFunc(method)
```

This is a wrapper function that returns a function handle `thetaFunction` taking the order p as an argument. Given p, the handle returns the parameter vector for a method of this order. It plays the same role as `getMethodVec` above, but for variable order solvers.

In contrast to the fixed order solvers, the variable order solvers need to be able to accommodate many different methods, and not necessarily from the same traditional family (the parameterized formulation of multistep methods makes it less meaningful to talk about families). Therefore, instead of using a fixed parameter vector, the variable order solvers use a function which returns the parameter vector for a given order. These functions also indicate — by throwing an exception, which can be caught — when a method is not available. For example if one uses a variable order scheme based upon BDF-methods, then there are no zero-stable methods above order 6, and the function therefore throws an exception.

This implementation makes it easy to add other variable order schemes, either in the form of traditional families or as a mix of methods between different families. One just has to write a function returning the parameter vectors for a given order (and which throws the correct exception when an order is unavailable), and then add it to the wrapper function.

Already implemented in the software package are:

- Explicit methods of order k (method class E_k):
  - Adams–Bashforth
  - EDF
  - Nyström
- Implicit methods of order k (method class I_k):
  - BDF
- Implicit methods of order $k+1$ (method class I_k^+):
  - Adams–Moulton
  - dcBDF

`function [p] = polE(theta,h,x,xDot)`

This function calculates the coefficients for the polynomial in a specific point `x` and a specific explicit method (defined by the parameter vector `theta`). This function was written by C. Arévalo and G. Söderlind [1].

`function [p] = polI(f,jac,theta,h,x,xDot,t,coeffs,TOL)`

This function calculates the coefficients for the polynomial in a specific point `x` and a specific implicit method (defined by the parameter vector `theta`) of order k. It uses Newton iteration to solve the system of equations which the implicit method gives rise to. This function was written by C. Arévalo and G. Söderlind [1].

`function [p] = polIp(f,jac,theta,h,x,xDot,t,coeffs,TOL)`

This function calculates the coefficients for the polynomial in a specific point `x` and a specific implicit method (defined by a parameter vector `theta`) of order $k+1$. It uses Newton iteration to solve the system of equations which the implicit method gives rise to. This function was written by C. Arévalo and G. Söderlind [1].

Package: thetas This package contains the actual functions returned by the wrapper function `getMethodThetaFunc`.

A.1.3. Package: control

This package contains functions related to the step-size and order control of the solver. The sub-package contains:

```
function [r] = errorController(controllerVec, order, errorVec, hVec,
  unit)
```

This is the error controller used to produce smooth step-size sequences in the solver. It can use arbitrary filters defined by a filter vector `controllerVec`.

`function [parvec] = getFilterVec(filter)`

This function gives you the filter vector of a specific filter, and is used by all solvers. It acts as a small library of filters consisting of the following:

- Second order filters:
  - H211D, H211*b*, H211PI
  - PI3333
  - PI3040
  - PI4020
- Third order filters:
  - H312D, H312*b*, H312PID
  - H321D, H321

`function [workFactors] = getWorkFactors(orders)`

This function returns the work factors used to determine the efficiency of different orders in the variable order solvers. By rewriting this function one may experiment with different models for determining how efficient a method of a certain order is. For example, just returning the vector $(1, 1, 1)$ makes the solver only factor in the step-size, and not how much computing power is necessary to take a step.

A.1.4. Package: init

```
function [h] = autostart(f, tspan, u0, tol, p)
```

This function calculates an appropriate initial step-size. This step-size is used by both `start` and the solver itself in the calculation of the first step after the steps created by `start`. The algorithm is written by C. Arévalo and G. Söderlind [1].

```
function [h,t,x,xDot] = start(f,h0,t0,x0,xDot0,TOL,k)
```

This function initializes the multistep method by calculating the first k steps using Matlab's solver `ode15s`. This function is used in all solvers and is written by C. Arévalo and G. Söderlind [1].

A.1.5. Package: ode

This package contains a library of test problems that can be used to evaluate the performance of these and other solvers. The problems are organized into three different categories, namely:

nonStiff Contains non-stiff test problems (see Table III.8).

stiff Contains stiff test problems (see Table III.6).

both Contains problems which can be made both stiff and non-stiff, by either adjusting a parameter or the initial condition of the problem (see Table III.7).

Every problem consists of one RHS-fucntion called `problemname_rhs.m`. Most of them also has an accompanying analytical Jacobian called `problemname_jac.m`. Further, a few of them also have an analytical solution called `problemname_sol.m`.

Master's Theses in Mathematical Sciences 2016:E49
ISSN 1404-6342

LUTFNA-3039-2016

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/