Technische Universität München

Zentrum Mathematik

# Sparsing in Real Time Simulation

Diplomarbeit

von

Anton Schiela

| | |
|---|---|
| Themensteller: | Prof. Dr. Folkmar Bornemann |
| Betreuer: | Prof. Dr. Folkmar Bornemann |
| | Dr. Martin Otter |
| Abgabetermin: | Montag, 11.2.2002 |

Hiermit erkläre ich, daß ich die Diplomarbeit selbständig angefertigt und nur die angegebenen Quellen verwendet habe.

München, 11. Februar 2002

# Danksagung

Die vorliegende Diplomarbeit wurde am Institut für Robotik und Mechatronik des Deutschen Zentrums für Luft- und Raumfahrt in Zusammenarbeit mit dem Lehrstuhl für Wissenschaftliches Rechnen an der Technischen Universität München angefertigt.
Bedanken möchte ich mich für die Förderung und Betreuung meiner Arbeit bei Herrn Prof. Dr. Folkmar Bornemann von der Technischen Universität München, bei Herrn Dr. Martin Otter und Herrn Dr. Johann Bals vom Institut für Robotik und Mechatronik des Deutschen Zentrums für Luft- und Raumfahrt, sowie bei Herrn Dr. Hilding Elmqvist und Herrn Dr. Hans Olsson von Dynasim, Schweden. Ihre fachliche und freundliche Begleitung waren mir eine wertvolle Unterstützung bei der Anfertigung dieser Diplomarbeit.
Desweiteren gilt mein Dank meinen Kollegen und Freunden am Institut für Robotik und Mechatronik, und an der Universität. Ihre Unterstützung und ihre Anregungen waren für mich besonders wichtig.
Falls sich Fragen oder Anmerkungen ergeben, bin ich jederzeit per e-mail über die Adresse `Anton.Schiela@web.de` erreichbar.

# Contents

# Chapter 1

# Introduction

Real time simulation is a growing field of applications for simulation software. The possibility to replace a device in the real world by a virtual process makes this simulation technique a powerful tool for scientists and engineers.
Especially the so called "Hardware in the Loop" (HIL) simulation is an application that becomes more and more important, e.g., for testing and optimization of electronic control units. In such a scenario, a piece of hardware (e.g., a controller) is used in a virtual environment created by a simulation, that is running in real time. This controller communicates with the simulation software in short time cycles so that the software is required to provide results once in a cycle.
Using new modelling techniques, such as object oriented modelling, it is possible to describe more and more complex technical models (see, e.g., [34], [36], [17]). Many of those are multi-domain models, which means that they contain components from more than one physical domain. Mechanic, electric, hydraulic or thermodynamic components are often coupled together in one model. Consequently there are several different time scales present in the model, leading to stiff ordinary differential equations (ODEs) or differential algebraic equations (DAEs).
Besides that, we will see in Chapter 2 that stiff real time simulations require different algorithms than classical off-line simulations. These requirements will lead us to a special method, the linearly implicit Euler method and a special form of adaptivity, sparsing. Sparsing means zeroing out elements of the Jacobian matrix to accelerate sparse matrix factorizations.
In Chapter 3 the theoretical properties of this method, that are important for sparsing are reviewed. Especially the stability properties of the linearly implicit Euler method with an inexact Jacobian are studied. This leads us to a generalized eigenproblem. Special interest is taken on the extension of the linearly implicit Euler method to a higher order method by extrapolation. We will see how sparsing affects the order of extrapolation methods applied to DAEs.
In Chapter 4 we will derive a criterion to choose the proper elements to be zeroed out such that the dynamical properties of the linearly implicit Euler method are affected as little as possible. The approach taken is via perturbation theory for matrix pairs. We apply this theory to the generalized eigenproblem mentioned above and obtain a sparsing criterion based on a first order estimate for the eigenvalue changes.
Chapter 5 discusses the implementation of a stiff real time simulator. Especially, the solution of large sparse systems of equations in real time is studied and a row sequential method for a sparse QR decomposition is chosen.
This simulator and its applicability to real life problems is tested in Chapter 6. We observe the effects of sparsing on the structure of the matrix and the dynamics of the method.

# Chapter 2

# Real Time Simulation

Simulations become more and more important for many fields of applied and industrial science. The increasing available computing capacities make it possible to simulate more and more complex models. One large field of applications is the simulation of mechatronic systems to facilitate and accelerate the design process. Especially in flight and automobile industries these techniques gain growing interest.
One special industrial application is real time simulation. This is a simulation, that runs with the same speed as the real world process. Real time simulations make it possible to establish communication between devices and processes in the real world and the processes that are simulated. This "hardware-in-the-loop" application is of great importance for the testing of devices and the design of complex controllers. We will describe this application in Section 2.1.
To be able to simulate a process, one has to build a mathematical model of it. As the processes to be simulated become more and more complicated, efficient modelling techniques are crucial to be able to cope with the complexity. One interesting approach for this is object oriented modelling, that we want to describe briefly in Section 2.2.
In Section 2.3 we deal with the algorithmic implications of real time simulation. We formalize the requirements on the algorithm and discuss the applicability of common simulation techniques and methods to real time simulations. This will motivate the methods presented in the subsequent chapters.

## 2.1 Real Time Simulation in Industrial Applications

### 2.1.1 Applications for Real Time Simulation

Two application scenarios will be described, where real time simulations are crucial for the development and enhancement of industrial products.
The first application is the so called "hardware-in-the-loop simulation". It is for example used for the automatic testing and the optimization of hardware components, such as controllers. In [26] an automatic test environment is described for the testing of an electronic control unit for an automatic gear box. To test such a controller, a set of test situations has to be generated, in which the controller is observed. For safety reasons, these tests have to be extensive and standardized. Especially, they have to cover extreme situations. Therefore, these tests are expensive, if they are accomplished by test drives in a car prototype. Hardware in the loop simulations provide a comparably cheap alternative. The controller is embedded into a virtual car. This means that the controller is connected to a computer, where a simulation of the car is running. This simulation is of course required to run at the speed of the real world process. It is also clear

that this requirement has to be fulfilled strictly. Otherwise, the timing in the communication between the controller and the simulation breaks down. Similar applications arise also in other fields, e.g., in aircraft industries (c.f. [25]).
A second application is real time simulation as part of a controller, e.g., for an inverse model. Inverse models are used to compensate nonlinearities in the controlled process. This can best be explained with an example. The quantities of interest of an industrial robot are the joint angles. However, the manipulable quantities are the voltages at the drives. The relations between the controlled variables and the manipulated variables are nonlinear and complicated, especially because the joints are coupled dynamically. With an inverse model, this relation can be simplified, i.e., the process together with the inverse model can be controlled more easily. Obviously, the computations involved in evaluating the inverse model have to be performed in real time.

### 2.1.2 Practical Issues in Real Time Simulations

The most important practical issue in hardware-in-the-loop simulations is a well working communication between the hardware and the simulation, especially with the right timing. To achieve this, special hardware and special software based on special algorithms have to be employed. For a detailed introduction into the practical issues of real time computing see, e.g., the textbook [19].
The whole hardware set-up may contain the test hardware itself, the computer on which the simulation is running, a personal computer as the interface to an operator, sensors to measure the output of the test hardware and actuators to generate its input. Furthermore, all these components have to be connected with each other, so a special network has to be built.
Depending on the test hardware, the sensors and actuators are more or less expensive. The simplest case is the hardware being a digital controller. Then it may be sufficient to connect the test hardware to the simulation hardware properly. In the case of an analog controller, one has to employ digital-analog and analog-digital converters to be able to communicate. The most expensive case is, if the inputs and outputs of the test hardware are not signals, but physical quantities, such as forces. Then large and expensive sensor and actuator systems have to be used. One extreme example for such a system is the test bench for the gear of a car, that was built at the Technical University Munich by the chair "Elektrische Antriebssysteme" (see [28]). The "inputs" are the torque of the engine, the friction of the wheels, and the air resistance. To create these forces in a laboratory, large electrical drives have been mounted to the wheels and to the flange that normally connects the engine and the gear.
The connection to the simulation hardware is accomplished by a real time capable bus. Due to the wide applicability of such systems in control engineering, several standard systems are available for this task.
The simulation hardware also has to be real time capable. This does not only mean that it has to supply a sufficient amount of computing power, but that computations have to be accomplished at a guaranteed speed. This stands in contradiction with some classical speed up concepts such as cacheing, or branch prediction. Such features are designed to increase the overall computation speed, but in the worst case, the computations are slowed down. A cache can speed up computations considerably if most of the data needed can be stored in the cache. Otherwise, if data cannot be fetched from the cache, the cost is much higher. As this behavior is not predictable, it is very difficult to combine the concept of cacheing with real time computations. Due to this and for several similar reasons, there is special hardware necessary to provide real time capable computing power.

The same requirements as for real time hardware apply to the software. All operations have to be accomplished during a guaranteed time span. For this reason, there are special operation systems available equipped with special features, such as accurate timing, scheduling by priority and special I/O functions.
Similar requirements apply to real time capable algorithms. These requirements are described in more detail in Section 2.3.

## 2.2 Object Oriented Modelling

The following section will be a brief description of object oriented modelling, a modelling technique, that was explored first by H. Elmqvist [16] in the late seventies. For a more detailed introduction see [36], [35], [17]. We will refer to the modelling language Modelica [32] as an important example for an object oriented modelling language and to the modelling and simulation software Dymola [14] as an example for an implementation.

### 2.2.1 Object oriented modelling with Modelica

Object oriented modelling is a very general and powerful modelling technique especially suited for the modelling of complex modular systems. The idea is to define classes of models, that correspond to real physical objects. Classes of models can either be defined by their physical equations, as an extension of a base class, or they can be constructed hierarchically from submodels. For this purpose (sub-)models can be connected to each other, modelling a physical connection. The type of the connection is described by the definition of interfaces in each object. Interfaces contain two types of variables: "potential variables" and "flow variables". If two compatible interfaces are connected, then additional equations are generated: The sum of matching flow variables is required to be zero, and for all corresponding potential variables equality is required. A description of a model built up from submodels and connections is called an object diagram or composition diagram. Models can be stored in libraries and can so be reused easily. Due to these features, object oriented modelling allows a convenient construction of complex multi domain models, especially in industrial applications. Modelling can either be accomplished using a textual language, graphically, or by a mixture of both.

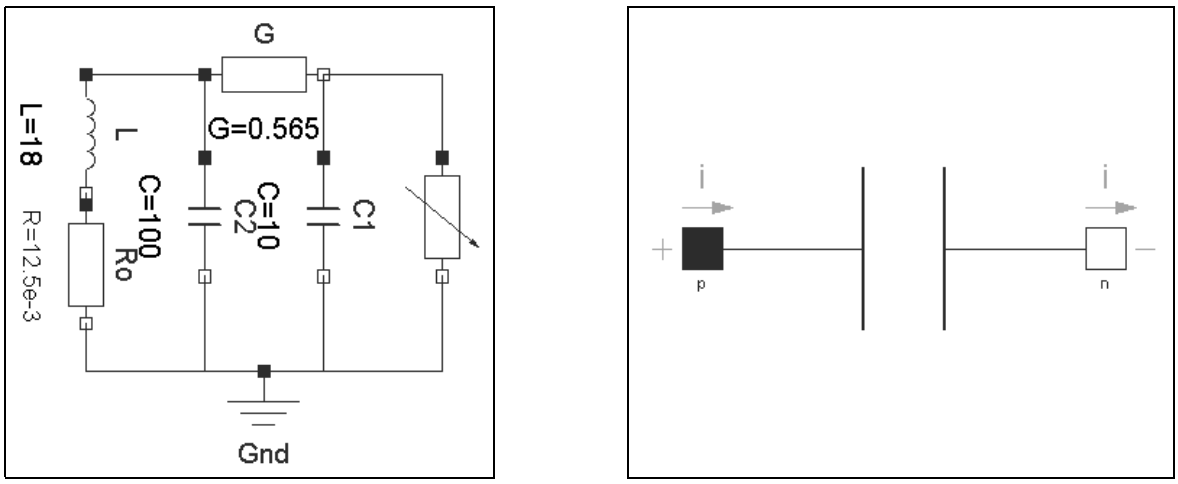


Figure 2.1: Object diagrams of an electrical circuit and of a capacitor.

For example if we want to model an electric circuit we can use a library, that contains models of electrical components, such as resistors, capacitors, and inductors. Each of them is defined by their physical equations. A model of a capacitor, for example, contains the equation $i = C\dot{u}$. A capacitor is defined as an extension of the base class "OnePort". Here the interface variables $u_1, u_2, i_1, i_2$ are defined, i.e., the voltage and the electrical current at each pin of the capacitor. It is also defined in "OnePort", that $u = u_2 - u_1$ and $i_1 = i_2$, and that $u_i$ are potential variables and $i_i$ are flow variables. Connecting the interfaces "1" and "2" of two electrical components "A" and "B" will now produce the additional equations $u_1^A = u_2^B$, and $i_1^A + i_2^B = 0$. These equations correspond to Kirchhoff's laws and can of course also be generated for multiple connections. If several electrical components are connected to an electric circuit containing a ground potential, and the model is physically meaningful, we obtain a non-singular system of differential algebraic equations by gathering all equations.
Roughly speaking, an object diagram containing submodels and connections between the submodels can be translated into a set of algebraic, differential, and discrete equations and a set of variables. Depending on the complexity of the object diagram, the size of such a system is between several hundred and several ten thousand equations, and the system is sparse, i.e., the corresponding incidence matrix is sparse. Especially, the algebraic part may be large, mostly because of the large number of connections between the submodels.

### 2.2.2 Preprocessing and Simulation with Dymola

To simulate such a large differential algebraic equation system efficiently, the system of equations has to be reduced and simplified by some symbolic preprocessing, as it is performed, e.g., by the software package Dymola [14]. Here, the preprocessing runs in several stages and affects mostly the algebraic equations. The goal is that for a given vector of "state variables" $x$, i.e., variables whose derivatives $\dot{x}$ occur in the equations, all algebraic variables $y$ and the state derivatives $\dot{x}$ can be computed efficiently.
First, redundancy is removed out of the system. Trivial equations, such as $y_i = p$, or $y_j = y_k$ are identified and removed together with the variables. Then an automatic index reduction is performed, using the "dummy-derivative" method by Mattson and Söderlind [31].
After this, the remaining equations and variables are sorted to obtain an incidence matrix in block lower triangular form. This "BLT-transformation" can be performed by an algorithm, that runs at a complexity $O(nN)$ (see, e.g., [13]). Here $n$ is the number of equations and $N$ is the number of non-zero elements of the incidence matrix. The result is a block lower triangular matrix with diagonal blocks, that cannot be further reduced by permutations of the rows and columns of the incidence matrix. The systems of equations corresponding to the diagonal blocks can now be treated separately and solved in a sequence.
To reduce the size of the equation systems even further, the preprocessing routine of Dymola performs a technique called tearing (see [27], [18], [3]). Tearing is a technique to split an irreducible block $B$ of an incidence matrix into four blocks

$$B = \begin{pmatrix} L & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \tag{2.1}$$

and sort the equations and variables such that $L$ is lower triangular, non-singular and as large as possible. In this case the solution of the equations corresponding to $B$ can essentially be reduced to the solution of a system of the size of $B_{22}$ and the solution of a sequence of one-dimensional equations corresponding to $L$. If the system to be solved is linear, then we can compute the

solution of

$$\begin{pmatrix} L & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2.2}$$

by solving the sequence of equation systems

$$(B_{22} - B_{21}L^{-1}B_{12})x_2 = b_2 - B_{21}L^{-1}b_1, \tag{2.3}$$

$$x_1 = L^{-1}(b_1 - B_{12})x_2. \tag{2.4}$$

In the non-linear case, we perform a Newton iteration over this sequence, using $x_2$ as iteration variables. Finding an optimal partitioning leads to non-polynomial algorithms. Therefore, heuristic algorithms are used. They also take care that the system of equations is actually solvable at all time instants in a stable way. This is possible, because the translation and preprocessing routines have access to additional information about the equations, e.g., if an equation is linear. Experience shows that after the preprocessing phase the size of the remaining equation systems to be solved numerically is rather small. Hence, dense numerical linear algebra routines can be used. The reason for this is that large irreducible blocks of algebraic equations correspond to complex submodels with purely algebraic equations ("algebraic loops"). However, in reasonable models these algebraic loops occur rather seldom.
The last stage of the preprocessing phase is the generation of C-code, that yields the right hand side of the reduced DAE system. Moreover, by default Dymola provides code that numerically transforms the DAE into an ODE, solving the remaining algebraic equations by Newton's method for each function evaluation.
Dymola provides a user interface for the interactive simulation of the models. After translation, models can be simulated using standard stiff and non-stiff integrators such as DASSL, DEABM, or explicit RK schemes. As many industrial system are stiff, DASSL is the default option. All integration methods are applied to the code in ODE form. Thus, in the standard setting a state space form method is used for the simulation.

### 2.2.3 Real Time Simulation in Dymola

With Dymola it is possible to generate code for the real-time simulation of stiff and non-stiff systems. To perform this efficiently, some additional features are needed.

**Inline integration.** The preprocessing routines in Dymola can be considered as a sparse solver applied to the algebraic part of the model only. This restriction is necessary, because the preprocessing routines have no information about the discretization method applied to the differential equations. The drawback is that using an implicit method, one step of the integration routine contains two loops: in the inner loop the algebraic equations are solved for each function evaluation, in the outer loop another system of equation is solved. This results in several function evaluations, calling the inner loop. This combination of a state space form method and an implicit integrator may lead to inefficient code.
If the preprocessing routines are applied after the discretization both algebraic and differential equations can be treated and there is only one single system of equations to be solved at each step. We obtain a direct method. As the preprocessing routines can only be applied before the integration starts, the discretization formula is fixed for the whole course of integration. However, for stiff real-time integration with its simple algorithms this technique, called "inline integration" (see [15]) is applicable and yields a very efficient method (see also Section 6.1). Every state derivative is replaced by its discretization formula, for example,

$$\dot{x} \to (x_{n+1} - x_n)/\tau, \tag{2.5}$$

where $x_{n+1}$ is unknown. The output variables are now $x_{n+1}, y_{n+1}$ and no longer $f(x_n), y_n$. The application of an explicit method means inserting $x_n$ into the right hand side. If $x_{n+1}$ is inserted into the right hand side we obtain an implicit method suitable for stiff systems. In any case the result is a purely algebraic system of equations, and the preprocessing routines described above can be applied.

**Mixed-mode integration.** In many cases, however, we observe that in connection with implicit inline integration, the preprocessing routines yield large systems of equations. The reason for this is that the systems of equations after preprocessing do not correspond to the physical algebraic loops in the model anymore, because the state variables subject to implicit discretization, have to be considered as unknowns now. Hence, equations containing state variables do not split algebraic loops any longer. So if we are forced to use an implicit discretization scheme, the systems of equations grow large and the simulation becomes inefficient.
One idea to cope with this situation is to insert the unknown $x_{n+1}$ only where it is necessary to obtain a stable integration scheme at a certain step size. Otherwise insert $x_n$. If a linearly implicit integration method is used, this approach is equivalent to sparsing of the Jacobian matrix, the main topic of this thesis. In the context of an implicit method and with the restriction that each component of the state vector is inserted into the right hand side either implicitly or explicitly, we obtain a scheme equivalent to a partitioned method (see [24]). This scheme is implemented in Dymola under the name "mixed-mode integration" [39].
It is not a trivial task to find out, where $x_{n+1}$ has to be inserted, and a large part of this thesis (Chapter 4) will deal with this question.

### 2.2.4 Example: an industrial robot

To obtain an idea of the possibilities of object oriented modelling, we will consider the model of the industrial robot Manutec r3. It will also be used as a test model in the subsequent chapters. A very detailed description of the model and its development can be found in [34].
The Manutec r3 is a classical six axis robot powered by six controlled electrical drives. To obtain

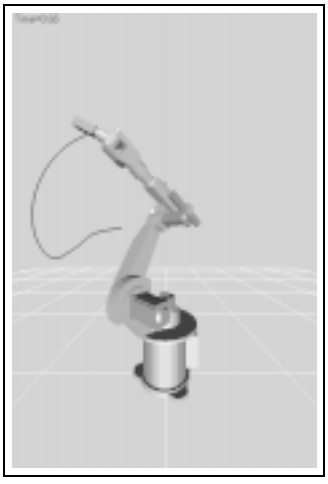

Figure 2.2: Screenshot of the simulated robot.

a detailed model it is necessary to model the dynamics of the robot as a multibody system, the joint friction, the gears, the dynamics of the electrical motors and the controllers. This leads to

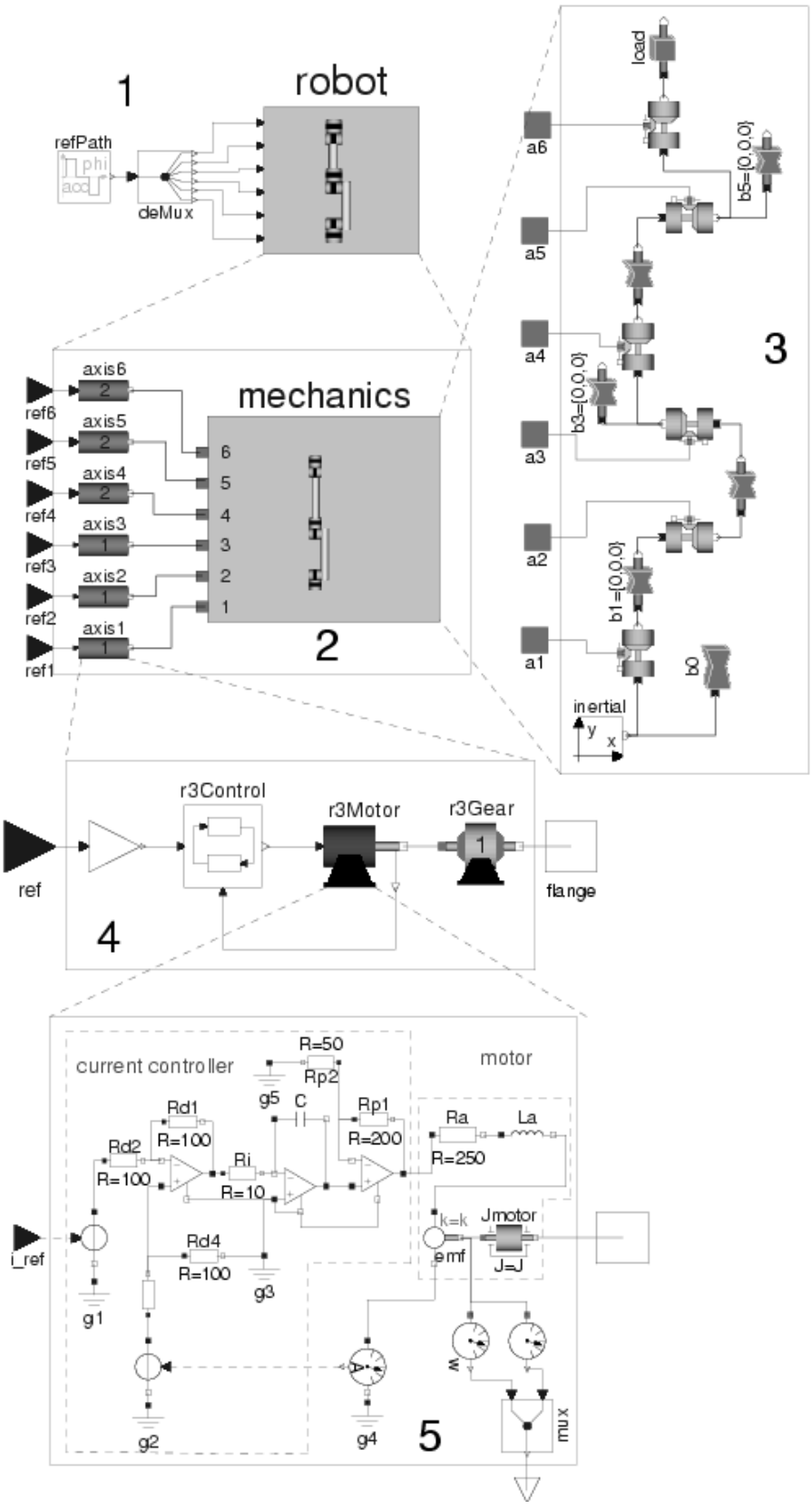


Figure 2.3: Object diagram of the robot.

an overall number of 7100 equations and 78 states, i.e., variables whose derivatives occur in the equations.
The preprocessing routines of Dymola can reduce this large system of equations in several stages. After removing the trivial equations and variables, there are 3612 algebraic and 78 differential variables left. This large system of equations is reduced by Dymola to an ODE of dimension 78. The largest system of equations to solve is 6×6. If this transformation is not desired, the resulting DAE system to be solved is of size 354×354 without tearing and 117×117 with tearing.
In Figure 2.3 we can see nicely, how the object oriented approach works. Figure 2.3.2 shows that the robot has got six axes, coupled together by a mechanical part, that we see in Figure 2.3.3. Each of the axes is modelled as shown in Figure 2.3.4. The large triangle to the left is a connector to a control signal from the outside. Here the desired angle and angular speed are provided. These values are the inputs of the controller for the motor. We see that the actual values of angular velocity and angle are fed into the controller, so that we obtain a closed loop control structure.
Zooming into the motor (see Figure 2.3.5) we see that it is a controlled direct current motor. A

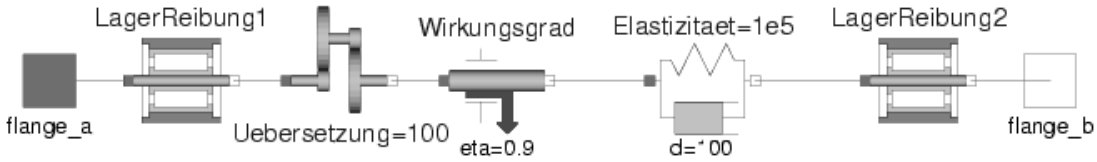


Figure 2.4: Object diagram of a gear.

PI controller (in the left part) is modelled as an analog circuit. Its purpose is to control the motor current and therefore the torque of the motor. The motor itself is implemented by a resistor, an inductor, the electric/mechanic transformer and an inertia. We see that some submodels have real physical counterparts, whereas other submodels are more of an abstract nature.
The motor is connected to a gear, whose object diagram is shown in Figure 2.4. It consists of several submodels, modelling bearing friction, elasticity, efficiency and the transmission coefficient of the gear. Here we realize a big advantage of object oriented modelling. We can very easily adjust the complexity of the model. If it is sufficient to model an ideal gear, we can leave out all objects, except for the transmission, if we want to model the friction differently, we can easily replace this submodel we use by a different one.

## 2.3 Numerical Issues in Real Time Simulation

In this section we are going to discuss how the special application "real time simulation" affects the design of numerical algorithms. We will see that especially conventional techniques for adaptivity cannot be used in real time simulations.

### 2.3.1 Specification of Real Time Simulation

We have described the special features of real time simulations in Section 2.1. The essential point was that the simulation communicates with devices from the real world during runtime. Therefore, real time simulation software has to meet certain requirements, that will be formalized in this section.

We consider the numerical solution of a quasi-linear differential algebraic initial value problem of index 1 with constant "mass matrix" $L$ and consistent initial values $x_0$.

$$L\dot{x} = f(x,t), \qquad x(t_0) = x_0. \tag{2.6}$$

This DAE is a model for a dynamic process, that starts at model time $t_0$ and runs for a certain amount of model time $\delta t$. For a given time grid $\Delta = \{t_0, t_1, \ldots, t_N = t_0 + \delta t\}$ on the interval $[t_0, t_0 + \delta t]$ and for consistent initial values $x_0$ we would like to obtain a numerical approximation $\{x_0, x_1, \ldots, x_N\}$ of the solution of (2.6) on $\Delta$. For this purpose we use an algorithm that starts to run on a computer at real time $T_0$. The special feature of "real time simulation" (in an abstract sense) is that for each grid point $t_n$ there is a specified "deadline" $\delta T_n$, a time span in "real time", and we require that the computation of $x_n$ is complete at the time $T_n = T_0 + \delta T_n$.
"Hardware in the Loop simulation" (in an abstract sense) requires that for each grid point $t_n$ there is a time span $\delta S_n$ such that $f(x_n, t_n)$ cannot be evaluated by the algorithm before the time $T_0 + \delta S_n$. This means that for given $x_n$ the computation of the next solution point $x_{n+1}$ cannot start before $S_n = T_0 + \delta S_n$ and is required to take no longer than the time span $D_n = \delta T_{n+1} - \delta S_n$. If this "real time" requirement is not met at each time instant the simulation fails.
In most applications we have an equidistant time grid $\Delta$ with a constant "communication interval" $\tau = t_1 - t_0 = \ldots = t_N - t_{N-1} = T_1 - T_0 = \ldots = T_N - T_{N-1}$, such that the intervals in model time and in real time are the same, and a constant "relative deadline" $\delta = D_0 = \ldots = D_{N-1}$. The fact that the simulation communicates with a real world device ,e.g., a controller has got

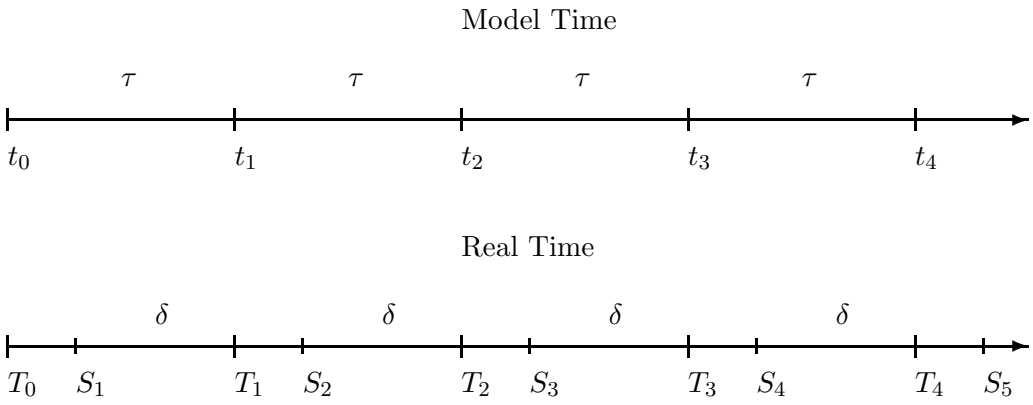


Figure 2.5: Time scales in real time simulation

several implications. First, the right hand side $f(x,t)$ cannot be assumed to be continuous, because $f(x,t)$ contains input from the outside. In most cases, this input is a step function, that changes at each time instant $T_i$.
Secondly, the communication interval $\tau$ is usually rather small, compared to the time scales of interest. This is because the real world device is constructed to cope with the original physical process in a reliable way. To obtain the necessary information the process is sampled using small time intervals.
A third point is that performance of real time simulation codes is measured in a special way, different from off-line simulation. We have seen that the success of real-time simulations depends on the adherence of the deadline $\delta$. The performance of a simulation code is therefore measured

by the minimal deadline $\delta_{min}$, for that the code is guaranteed to yield results before it has expired. This "worst case" measure affects both real time software and hardware design.

### 2.3.2 Appropriate numerical methods

To perform real time simulations in a reliable way we have to choose appropriate numerical methods, that reliably fulfill the requirements we have stated in Section 2.3.1.

**Multistep methods.** Multistep methods use information from the past to compute high order approximations of the solution. This implies the assumption that the differential equation to be solved is smooth. This is however not true in the case of real time simulations, as discontinuities at each time instant $T_i$ may occur. Therefore, multistep methods are not suitable for this type of application.

**Explicit One-step methods.** Explicit one-step methods meet all requirements stated in Section 2.3.1. Their computational effort is low and constant for each step and they are well suited for systems with discontinuities, as they do not use information from the past. So for non-stiff ordinary differential equations explicit one-step methods are the method of choice. In fact, currently there is hardly any alternative method used.
However, if the problem is stiff, explicit methods run into trouble. For mildly stiff problems one remedy is to use integration step sizes that are a fraction of the sample interval, but then efficiency decays for increasing stiffness. Another "solution" to this problem is to modify the model such that the stiffness decreases. It is of course a questionable tactic to change the dynamics of a model in order to get it simulated.
Differential algebraic systems of the form

$$\dot{x} = f(x, y) \tag{2.7}$$
$$0 = g(x, y) \tag{2.8}$$

are often changed to an ODE by explicitly introducing a perturbation parameter $\varepsilon$

$$\dot{x} = f(x, y) \tag{2.9}$$
$$\varepsilon\dot{y} = g(x, y). \tag{2.10}$$

The difficulty here is to choose the $\varepsilon$. There will always be a trade-off between model accuracy and the stiffness of the ODE. In many cases this will not lead to satisfactory results.

**Implicit one-step methods.** Implicit methods are designed to solve stiff and differential algebraic systems efficiently. For this purpose, a non-linear system of equations is solved at each step. This is performed by simplified Newton iterations. Therefore, the computational effort for each step cannot be estimated reliably, as it depends on the (theoretically unbounded) number of iterations. Hence, implicit methods as well as all methods solving non-linear equation systems are not feasible in real time simulation.
Nevertheless algorithms based on implicit methods can be used for real time simulations, if the number of Newton iterations is kept fixed.

**Linearly implicit one-step methods.** These methods only solve one linear system of equations at each step. This makes the computational effort predictable and therefore linearly implicit methods are well suited for the simulation of stiff and differential algebraic problems in real time. Linearly implicit methods share one drawback with implicit methods: if the size of the problem is very large, then the solution of the arising linear equation systems is computationally costly.

**Order of Convergence.** In hardware-in-the-loop applications, the sampling intervals $\tau$ are usually small compared to the time scales of interest. The required accuracy is usually rather low. On the one hand it is often sufficient to reflect the qualitative behavior of the simulated system, on the other hand, controlled systems are in general asymptotically stable, and so the error propagation is rather well behaved. Hence, we concentrate on low order methods especially the explicit Euler method for non-stiff ODEs:

$$x_{n+1} = x_n + \tau f(x_n, t_n) \tag{2.11}$$

and the linearly implicit Euler method:

$$x_{n+1} = x_n + \tau(L - \tau Df|_{(x_n,t_n)})^{-1} f(x_n, t_n) \tag{2.12}$$

for stiff ODEs and DAEs. Here $Df|_{(x_n,t_n)}$ denotes the Jacobian matrix of $f$ at $(x_n, t_n)$. In the following we will concentrate on the stiff and differential algebraic case, so we will deal with the linearly implicit Euler method.

### 2.3.3 Adaptivity

Classically, adaptivity adjusts the behavior of the algorithm while the simulation is running. Two examples are step size control and the reuse of the Jacobian matrix. The goal is making the algorithm more reliable (in terms of error control) and more efficient (in terms of overall computation time).
In real time simulations we do not have the possibility to change the step size because it is given or severely restricted by the real time specifications of the problem. The best thing we can do for "controlling" the error is to estimate it and log the estimates so that the value of the results can be judged a-posteriori.
If we want to use adaptivity to gain performance, we have to recall how performance is measured. (see Section 2.3.1) We see that every algorithmical device that reduces the computational work for *some* steps, but not for all will not lead to a better performance in real time simulations. This rules out techniques like reusing the Jacobian matrix, or the application of sparse LU-solvers. In off-line simulations such techniques are essential for efficient codes, as the average performance per unit step is improved. However, the "worst case performance" will usually decrease.
If we want to design an adaptive algorithm, we have to gather information before real time simulation starts and then use a form of adaptivity that relies on time independent properties of the problem. This requires an analysis of the problem performed by a preprocessing routine. As the overall computation time is secondary in real time simulations, such a strategy makes sense even if the preprocessing phase is quite expensive.

### 2.3.4 Industrial models and Stiffness

Real time simulations are mostly performed in industrial applications using models described by large differential algebraic equation systems with stiff components. In most cases stiffness is caused by subsystems with fast dynamics such as controllers or electric circuits compared to the

time scales of interest, e.g., the movement of mechanical parts. Systems with such a structure are sometimes called "separably stiff" (see, e.g., [24].). There are various ways to exploit this structure in off-line simulation (see, e.g., [24], [44]). These methods, however, are not suitable for real time simulation, because they use iterative techniques to adapt to the model. On the other hand we observe that this model inherent structure is often time invariant and therefore something we can also make use of in the real time case. Moreover, the qualitative behavior of the dynamical system generated by the numerical discretization concerning stiffness is easy to predict because the step size is kept constant.
We see that in industrial real-time simulation there can be a lot of time invariant structure in the model and the discretization. This can be exploited by a form of adaptivity as described in Section 2.3.3.

### 2.3.5 Sparsing

The linearly implicit Euler method for stiff simulations consumes a large part of the computation time by the decomposition of matrix $(L-\tau Df)$. If the Jacobian $Df$ is large and sparse this effort can be reduced by the use of direct sparse solvers (see, e.g., Duff [12]). The sparser the matrix, the faster the factorization of the matrix. However, in real time simulations such techniques cannot be applied directly, because the structure of the matrix may change and therefore the performance of the factorization. Moreover, the Jacobian matrix often contains more information than needed to stabilize the simulation.
This leads to the idea of setting selected elements of the Jacobian to zero such that - while the integration method is still stable - a sparse matrix factorization can be performed more efficiently. This technique leads to an approximation of the Jacobian matrix with reduced sparsity structure and is therefore called sparsing.
An example where sparsing was performed successfully in an off-line simulation is described by Nowak [33]. Here the matrix elements were zeroed out dynamically before each time step using a criterion based on the magnitude of the matrix elements.
In our approach for the real time case the Jacobian $Df|_{(x_n,t_n)}$ is analyzed for several $(x_n, t_n)$ and a fixed sparsity pattern is selected during a preprocessing phase. During the simulation only the remaining elements will be taken into account by the factorization routine. This will lead to a reduced and fixed sparsity pattern, that can now be exploited by an appropriate sparse factorization routine. The factorization routine itself has to meet real time requirements, too. We will therefore use a sparse orthogonal factorization, whose performance is only dependent on the fixed structure of the Jacobian matrix.
For a successful application of sparsing we have to deal with two questions: How does the approximation of the Jacobian affect the stability of the integration method? How is the accuracy of the method affected by sparsing? The next chapter deals with these questions considering the linearly implicit Euler method and extrapolation codes based on this method.

# Chapter 3

# Linearly implicit Euler, Sparsing, Extrapolation

## 3.1 The linearly implicit Euler method for DAEs of Index 1

The linearly implicit Euler method is the simplest example of the class of linearly implicit methods. This is a class of methods for the solution of stiff differential equations, with the common feature that one linear system of equations is solved during each step, which is a simplification compared to fully implicit methods. More detailed information about this class can be obtained, e.g., from the textbooks [9], [23], and [24].
The basic idea of the linearly implicit Euler method (and of linearly implicit methods in general) is to avoid the solution of non-linear systems of equations by splitting the differential equation

$$\dot{x} = f(x), \tag{3.1}$$

into two parts

$$\dot{x}(t) = Jx(t) + (f(x(t)) - Jx(t)), \qquad J = Df(x), \tag{3.2}$$

and discretizing the linear part implicitly and the non-linear part explicitly:

$$x_{n+1} = x_n + \tau J x_{n+1} + \tau(f(x_n) - Jx_n), \tag{3.3}$$

which yields

$$(I - \tau J)x_{n+1} = (I - \tau J)x_n + \tau f(x_n),$$

and hence

$$x_{n+1} = x_n + \tau(I - \tau J)^{-1} f(x_n). \tag{3.4}$$

We obtain a numerical method of convergence order 1. For linear problems the non-linear term vanishes and the linearly implicit Euler method is identical to the implicit Euler method. Hence, all results of the linear stability theory for the implicit Euler method can be applied immediately to its linearly implicit counterpart.
If we apply the linearly implicit Euler method to the test problem $\dot{x} = \lambda x$

$$\begin{aligned} x_{n+1} &= x_n + \tau(1 - \tau\lambda)^{-1}\lambda x_n \\ &= \frac{1}{1 - \tau\lambda} x_n, \end{aligned} \tag{3.5}$$

and set $z = \tau\lambda$ we obtain the stability function of the linearly implicit Euler method:

$$R(z) = \frac{1}{1-z}. \tag{3.6}$$

We will study the stability properties of the linearly implicit Euler method in Section 3.3 thoroughly. It is a well known fact that the method is L-stable. Therefore, it can be adapted for the solution of differential algebraic equations of Index 1.
To construct an efficient numerical method of higher order, the linearly implicit Euler method can be used as a basic step for an extrapolation method. For stiff ODEs such a method is implemented in the code EULSIM (see [7]). However, for stiff ODEs, extrapolation of the linearly implicit mid-point rule usually leads to more efficient codes.
Considering quasi-linear implicit differential equations of index 1

$$L(x)\dot{x} = f(x), \qquad x(0) = x_0, \tag{3.7}$$

with solution dependent matrix $L(x)$ we have to use a modification of 3.4 to take into account the left-hand-side matrix $L$. One possibility proposed by Deuflhard and Nowak (see [11]) is

$$x_{n+1} = x_n + \tau(L(x_n) - \tau\frac{d}{dx}(f(x_0) - L(x_0)\dot{x}_0))^{-1} f(x_n). \tag{3.8}$$

The extrapolation code LIMEX is based on this discretization scheme. As slightly different discretization is used in the code SEULEX by Lubich and Hairer. For more details on this topic see [7], [10], [11], [29], [9], and [24].

## 3.2 The linearly implicit Euler method with inexact Jacobian

The linearly implicit Euler method as the simplest linearly implicit method is a good candidate for sparsing. We are going to analyze this method for the case of a quasi-linear differential algebraic system with constant, possibly singular left hand side matrix $L$ and constant index 1.

$$L\dot{x} = f(x), \qquad x(0) = x_0. \tag{3.9}$$

In Section 3.5 we will discuss briefly the case of a solution dependent mass matrix $L(x)$. We assume that we use an inexact Jacobian, i.e., a matrix $J \approx Df$ for the discretization:

$$x_{n+1} = x_n + \tau(L - \tau J)^{-1} f(x_n, t_n) \tag{3.10}$$

The difference between the exact and the approximate Jacobian is denoted by $\Delta J := Df(0) - J$. For analysis we can always transform this system to separated form (see [24],VI.1), i.e, we can find transformation matrices $S, T$ such that

$$M = S \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix} T, \tag{3.11}$$

and transform (3.9) into

$$\begin{aligned} \dot{x} &= f(x,y), \qquad x(0) = x_0 \\ 0 &= g(x,y). \qquad y(0) = y_0 \end{aligned} \tag{3.12}$$

The condition for index 1 is now that

$$g_y(x,y) \qquad \text{is invertible.} \tag{3.13}$$

Linearly implicit methods are invariant under such coordinate transformations and therefore results about separated DAEs can be generalized to the case of a constant mass matrix in a straightforward way. The approximate Jacobian can now be written as a block matrix:

$$\begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix} = \begin{pmatrix} f_x(0) & f_y(0) \\ g_x(0) & g_y(0) \end{pmatrix} - \begin{pmatrix} \Delta J_{11} & \Delta J_{12} \\ \Delta J_{21} & \Delta J_{22} \end{pmatrix} \tag{3.14}$$

The inexact linearly implicit Euler method now reads:

$$\begin{pmatrix} I - \tau J_{11} & -\tau J_{12} \\ -\tau J_{21} & -\tau J_{22} \end{pmatrix} \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix} = \begin{pmatrix} f(x_i, y_i) \\ g(x_i, y_i) \end{pmatrix} + O(\tau^{M+2}). \tag{3.15}$$

## 3.3 Stability of the linearly implicit Euler method

We are now going to take a closer look on the stability properties of the linearly implicit Euler method. This will help us understand certain effects caused by sparsing of the Jacobian.
The basis of the following considerations are two well known theorems about stability of differential equations and difference equations, that we cite from the book [9].

**Theorem 3.1** *Let $x_* \in \Omega_0$ be a fixed-point of the autonomous differential equation $\dot{x} = f(x)$, with a right hand side $f$, that is continuously differentiable on $\Omega_0$. If the spectral abscissa of the Jacobian matrix in $x_*$ is negative, i.e.,*

$$\nu(Df(x_*)) := \max_{\lambda \in \sigma(Df(x_*))} \Re(\lambda) < 0, \tag{3.16}$$

*then $x_*$ is an asymptotically stable fixed point.*

**Proof**: This is a translation of [9] Satz 3.30. □

**Theorem 3.2** *Let $\Psi : \Omega_0 \to \Omega_0$ be a continuously differentiable mapping on the open set $\Omega_0 \in \mathbb{R}^d$ with a fixed-point $x^* = \Psi(x^*)$. If the spectral radius $\rho$ of the Jacobian matrix in $x^*$ is smaller than one, i.e.,*

$$\rho(D\Psi(x^*)) := \max_{\lambda \in \sigma(D\Psi(x_*))} |\lambda| < 1, \tag{3.17}$$

*then there is a $\delta > 0$ such that the iteration*

$$x_{n+1} = \Psi(x_n), \qquad n = 0, 1, 2, \ldots, \tag{3.18}$$

*converges for all $x_0 \in B_\delta(x^*) \in \Omega_0$. The fixed-point $x^*$ is therefore asymptotically stable.*

**Proof**: This is a translation of [9] Satz 3.38. □
If we perform a discretization of a differential equation $\dot{x} = f(x)$, we substitute it by a difference equation $x_{n+1} = \Psi_\tau(x_n)$. As a consequence we transform the eigenvalues of the Jacobian matrix. The stability function describes this transformation for a linear scalar equation.
We have derived the stability function of the linearly implicit Euler method to be:

$$R(z) = \frac{1}{1 - z}, \qquad z = \lambda\tau. \tag{3.19}$$

In Figures 3.1 and 3.2 we can study the main features of this mapping. First, we observe that the whole negative half plane (and even a large part of the positive half plane) is mapped onto the unit disc. Hence, the method inherits asymptotic stability from the differential equation in

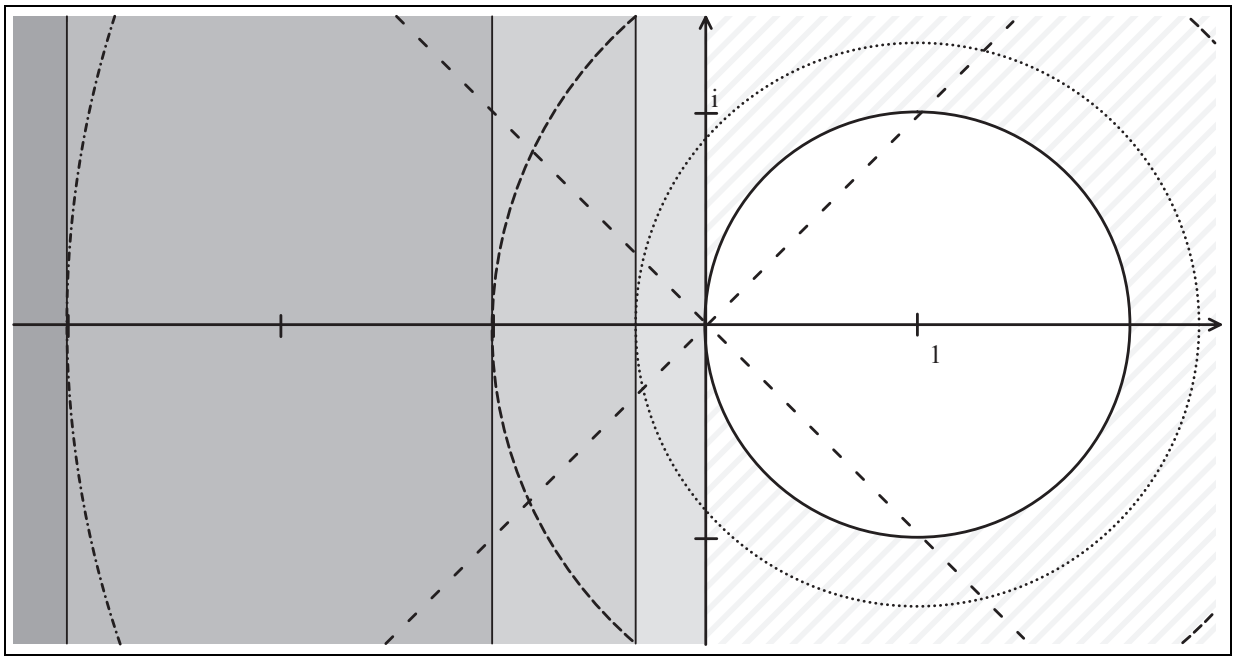


Figure 3.1: Some subsets of the complex plane.

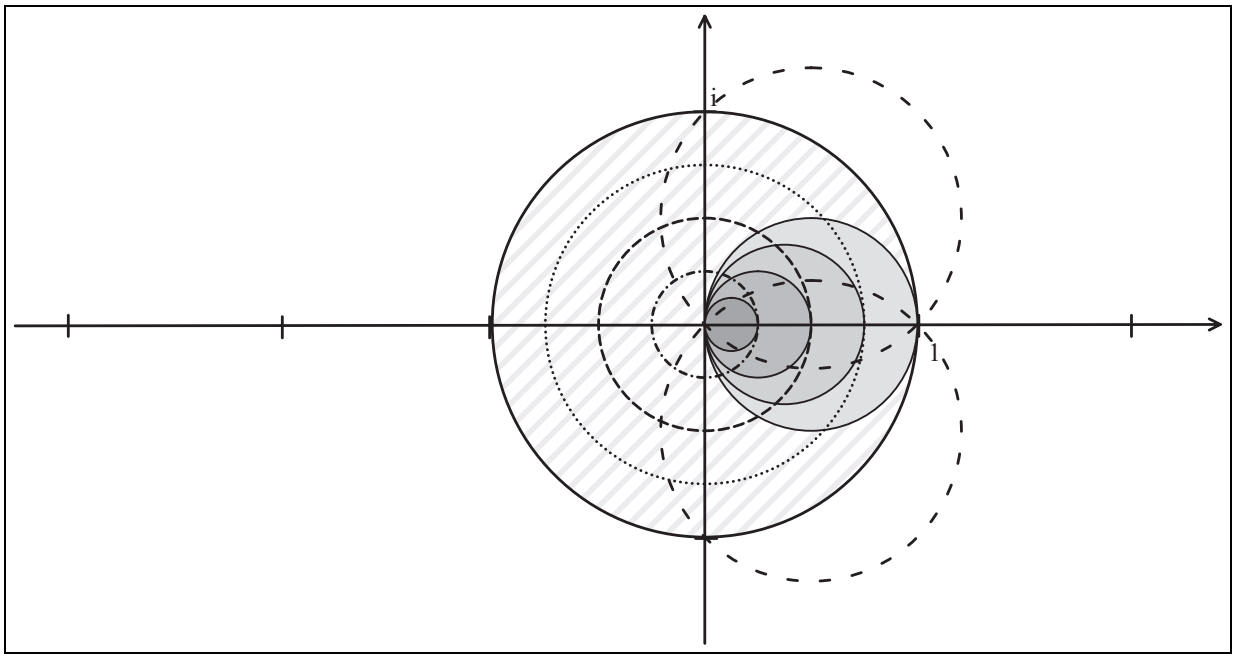


Figure 3.2: The same subsets transformed by the mapping $z \to R(z) = \frac{1}{1-z}$

all cases covered by Theorems 3.1 and 3.2, regardless of the step size $\tau$. This property is known as A-stability.
The second observation is that $\lim_{z\to\infty} R(z) = 0$. This feature is called L-stability. An L-stable method inherits the qualitative behavior of the test equation for large negative $\Re(z)$. Figures 3.1 and 3.2 illustrate this behavior: on the one hand the grey areas in Figure 3.1: $\{z \in \mathbb{C}|\ \Re(z) < -M\}$ with large positive $M$ are mapped onto small discs (which reflects the desired L-stability), on the other hand the subsets $\{z \in \mathbb{C}|\ |z-1| > R\}$ with large $R$ are also mapped onto small discs. As a consequence, the qualitative behavior of some unstable components of the solution is reflected in the wrong way. This often undesired property is called superstability.
L-stability is essential for the successful numerical treatment of singular perturbation problems and differential algebraic problems. It guarantees that for linear problems the numerical solution is consistent with the algebraic constraints after one single step.
If we are using the linearly implicit Euler method with an exact Jacobian matrix, then the features of this mapping generalize to higher dimensional problems: each eigenvalue is transformed by this mapping and the stability properties are accordingly.
If we use an inexact Jacobian, then this generalization is not possible anymore, and to study the stability of the discrete system we have to turn directly to the eigenvalues of the linearized difference equation

$$(L-\tau J)x_{n+1} = (L-\tau J)x_n + \tau Df x_n = (L+\tau\Delta J)x_n,$$

generated by the discretization with the linearly implicit Euler method , or in abstract notation

$$Bx_{n+1} = Ax_n. \tag{3.20}$$

As $B = L - \tau J$ is invertible due to the Index 1 assumption, we can directly apply Theorem 3.2 and conclude that we have to consider the generalized eigenvalue problem

$$Av - \lambda Bv = 0 \tag{3.21}$$

for stability considerations. Instead of the eigenvalue problem for an exact Jacobian

$$Lv - \lambda(L-\tau Df)v = 0, \tag{3.22}$$

we have to deal with

$$(L+\tau\Delta J)v - \lambda(L-\tau Df+\tau\Delta J)v = 0. \tag{3.23}$$

The investigation of stability for the extrapolated linearly implicit Euler method can be performed in a similar way. However, things become more complicated and we cannot avoid dealing with $(L-\tau J)^{-1}$ directly.

## 3.4 Asymptotic expansions for an inexact Jacobian matrix

In general, the use of an inexact Jacobian for linearly implicit methods leads to a loss of order in the method. However, one can construct linearly implicit methods without loss of order, so called W-methods. They were studied first by Steinhaug and Wolfbrandt [40]. The linearly implicit Euler method and extrapolation methods based on the linearly implicit Euler method are W-methods. The notion of W-methods is, however only meaningful in the context of ordinary differential equations. In this case convergence follows from consistency, because stability is

achieved for sufficiently small step sizes.
In the differential algebraic case we have to impose assumptions on the Jacobian matrix to achieve convergence. If we consider, for example, a purely algebraic equation, we see that the linearly implicit Euler method applied on this problem turns into Newton's method, which is independent of any step size $\tau$. Therefore, convergence can only be achieved if we assume some kind of stability. Moreover, we will see that concerning asymptotic expansions of the global error, an inexact Jacobian matrix has the consequence that perturbations may appear earlier than otherwise.
We will study this by computing the asymptotic expansion of the global error. For this purpose we will reproduce the proof of Theorem VI.5.3 in [24], that was developed originally in [10], and turn our attention to those parts of the proof, where the assumption of an exact Jacobian is crucial.

**Theorem 3.3** *Consider the problem (3.12) with consistent initial values $(x_0, y_0)$ and sufficiently smooth right hand side. Suppose that (3.13) is satisfied. The global error of the linearly implicit Euler method with inexact Jacobian (3.15) then has an asymptotic $\tau$-expansion of the form:*

$$\begin{aligned} x_i - x(t_i) &= \sum_{j=1}^{M} \tau^j (a_j(t_i) + \alpha_i^j) + O(\tau^{M+1}) \\ y_i - y(t_i) &= \sum_{j=1}^{M} \tau^j (b_j(t_i) + \beta_i^j) + O(\tau^{M+1}) \end{aligned} \tag{3.24}$$

*where $a_j(t), b_j(t)$ are smooth functions and the perturbation terms satisfy:*

$$\alpha_i^1 = 0, \qquad i \geq 0 \tag{3.25}$$

*If $\Delta J_{12} = 0, \Delta J_{21} = 0, \Delta J_{22} = 0$ and for exact Jacobian matrix, the perturbations satisfy:*

$$\alpha_i^1 = 0, \quad \alpha_i^2 = 0, \quad \alpha_i^3 = 0 \quad \beta_i^1 = 0 \qquad i \geq 0 \tag{3.26}$$
$$\beta_i^2 = 0 \qquad i \geq 1 \tag{3.27}$$
$$\alpha_i^j = 0 \qquad \textit{for} \quad i \geq j-4 \quad \textit{and} \quad j \geq 4 \tag{3.28}$$
$$\beta_i^j = 0 \qquad \textit{for} \quad i \geq j-2 \quad \textit{and} \quad j \geq 3 \tag{3.29}$$

*If $\Delta J_{21} = 0, \Delta J_{22} = 0$ then the perturbations satisfy:*

$$\alpha_i^1 = 0, \quad \alpha_i^2 = 0, \quad \beta_i^1 = 0 \qquad i \geq 0 \tag{3.30}$$
$$\alpha_i^3 = 0, \quad \beta_i^2 = 0 \qquad i \geq 1 \tag{3.31}$$
$$\alpha_i^j = 0 \qquad \textit{for} \quad i \geq j-3 \quad \textit{and} \quad j \geq 4 \tag{3.32}$$
$$\beta_i^j = 0 \qquad \textit{for} \quad i \geq j-2 \quad \textit{and} \quad j \geq 3 \tag{3.33}$$

*If $\Delta J_{12} = 0, \Delta J_{22} = 0$ then the perturbations satisfy:*

$$\alpha_i^1 = 0 \qquad i \geq 0 \tag{3.34}$$
$$\alpha_i^j = 0 \qquad \textit{for} \quad i \geq j-2 \quad \textit{and} \quad j \geq 2 \tag{3.35}$$
$$\beta_i^j = 0 \qquad \textit{for} \quad i \geq j \quad \textit{and} \quad j \geq 1 \tag{3.36}$$

*If $\Delta J_{22} = 0$ then the perturbations satisfy:*

$$\alpha_i^1 = 0 \qquad i \geq 0 \tag{3.37}$$
$$\alpha_i^j = 0 \qquad \textit{for} \quad i \geq j-1 \quad \textit{and} \quad j \geq 2 \tag{3.38}$$
$$\beta_i^j = 0 \qquad \textit{for} \quad i \geq j \quad \textit{and} \quad j \geq 1 \tag{3.39}$$

*If $\|I - J_{22}^{-1} g_y(0)\| \leq \gamma < 1$, then the error terms in (3.24) are uniformly bounded for $t_i = i\tau \leq T$ if $T$ is sufficiently small.*

**Proof**: The proof starts exactly like the proof of Theorem VI.5.3 in [24].
First we will recursively construct truncated expansions

$$\begin{aligned} \hat{x}_i &= x(t_i) + \sum_{j=1}^{M} \tau^j (a_j(t_i) + \alpha_i^j) + \tau^{M+1} \alpha_i^{M+1} \\ \hat{y}_i &= y(t_i) + \sum_{j=1}^{M} \tau^j (b_j(t_i) + \beta_i^j) \end{aligned} \tag{3.40}$$

such that the defect of $\hat{x}_i, \hat{y}_i$ inserted into the method is small. More precisely we require that

$$\begin{pmatrix} I - \tau J_{11} & -\tau J_{12} \\ -\tau J_{21} & -\tau J_{22} \end{pmatrix} \begin{pmatrix} \hat{x}_{i+1} - \hat{x}_i \\ \hat{y}_{i+1} - \hat{y}_i \end{pmatrix} = \begin{pmatrix} f(\hat{x}_i, \hat{y}_i) \\ g(\hat{x}_i, \hat{y}_i) \end{pmatrix} + O(\tau^{M+2}). \tag{3.41}$$

For the initial values we require $\hat{x}_0 = x_0, \hat{y}_0 = y_0$, or equivalently

$$a_j(0) + \alpha_0^j = 0, \qquad b_j(0) + \beta_0^j = 0, \tag{3.42}$$

and the perturbation terms are assumed to satisfy

$$\alpha_i^j \to 0, \qquad \beta_i^j \to 0 \qquad \text{for} \quad i \to \infty, \tag{3.43}$$

otherwise, these limits could be added to the smooth parts. The result will then follow from a stability estimate derived in the second part.

**Construction of truncated expansions.** For the construction of $a_j(x), b_j(x), \alpha_i^j, \beta_i^j$ we insert the truncated expansions (3.40) into the formula for the defect (3.41), and develop

$$\begin{aligned} f(\hat{x}_i, \hat{y}_i) &= f(x(t_i), y(t_i)) \\ &+ f_x(\tau a_1(t_i) + \tau \alpha_i^1 + \ldots) \\ &+ f_y(t_i)(\tau b_1(t_i) + \tau \beta_i^1 + \ldots) \\ &+ f_{xx}(t_i)(\tau a_1(t_i) + \tau \alpha_i^1 + \ldots)^2 + ..., \end{aligned}$$

$$\begin{aligned} \hat{x}_{i+1} - \hat{x}_i &= x(t_{i+1}) - x(t_i) + \tau(a_1(t_{i+1}) - a_1(t_i) + \alpha_{i+1}^1 - \alpha_i^1) + \ldots \\ &= \tau \dot{x}(t_i) + \frac{\tau^2}{2} \ddot{x}(t_i) + \ldots + \tau^2 \dot{a}_1(t_i) + \tau(\alpha_{i+1}^1 - \alpha_i^1) + \ldots, \end{aligned}$$

where $f_x(t) = f_x(x(t), y(t))$, etc. Similarly, we develop $g(\hat{x}_i, \hat{y}_i)$ and $\hat{y}_{i+1} - \hat{y}_i$, and compare coefficients of $\tau^{j+1}$ for $j = 0, \ldots, M$. Each power of $\tau$ will lead to two conditions - one containing

the smooth functions and the other containing the perturbation terms. The structure of the equations will in general be

$$\begin{aligned} f_x(t)a_j(t) + f_y(t)b_j(t) + r(t) &= \dot{a}_j(t) && (3.44)\\ g_x(t)a_j(t) + g_y(t)b_j(t) + s(t) &= 0 && (3.45)\\ \alpha_{i+1}^{j+1} - \alpha_i^{j+1} - J_{11}(\alpha_{i+1}^j - \alpha_i^j) - J_{12}(\beta_{i+1}^j - \beta_i^j) &= f_x(0)\alpha_i^j + f_y(0)\beta_i^j + \rho_i^j && (3.46)\\ -J_{21}(\alpha_{i+1}^j - \alpha_i^j) - J_{22}(\beta_{i+1}^j - \beta_i^j) &= g_x(0)\alpha_i^j + g_y(0)\beta_i^j + \sigma_i^j, && (3.47) \end{aligned}$$

with $i \geq 0, j \geq 0$. The terms $s(t), r(t)$ are smooth known functions depending on derivatives of $x(t), y(t)$ and on $a_{l-1}(t), b_{l-1}(t)$ with $l \leq j$. The terms $\rho_i^j, \sigma_i^j$ are linear combinations of expressions which contain as factors $\alpha_{i+1}^l, \alpha_i^{l-1}, \beta_i^{l-1}$ with $l \leq j$. In the first few steps and for each assumption $\Delta J_{ij} = 0$ some of the terms listed here drop out, so we will have a closer look on the first three steps.

*First step* ($j = 0$)*:* For $\tau^1$ we obtain the equations (3.12) for the smooth part, and $\alpha_{i+1}^1 - \alpha_i^1 = 0$. Because of (3.43) we get $\alpha_i^1 = 0$ for $i \geq 0$. As $J$ does not appear in the term for $\tau^1$, this result is independent of the quality of the approximation of $J$. However, we will need some assumptions on $J$ for further steps and in the second part of the proof.

*Second step* ($j = 1$)*:* The coefficients of $\tau^2$ give:

$$\begin{aligned} \dot{a}_1(t) + \frac{1}{2}\ddot{x}(t) - J_{11}\dot{x}(t) - J_{12}\dot{y}(t) &= f_x(t)a_1(t) + f_y(t)b_1(t) && (3.48)\\ -J_{21}\dot{x}(t) - J_{22}\dot{y}(t) &= g_x(t)a_1(t) + g_y(t)b_1(t) && (3.49)\\ \alpha_{i+1}^2 - \alpha_i^2 - J_{12}(\beta_{i+1}^1 - \beta_i^1) &= f_y(0)\beta_i^1 && (3.50)\\ -J_{22}(\beta_{i+1}^1 - \beta_i^1) &= g_y(0)\beta_i^1. && (3.51) \end{aligned}$$

The system (3.48)-(3.51) can be solved as follows. Compute $b_1(t)$ from (3.49) and insert it into (3.48), which is possible due to the index 1 assumption: $g_y(t)$ non-singular. This gives a linear differential equation for $a_1(t)$. Due to (3.42) and since $\alpha_0^1 = 0$, the initial values are $a_1(0) = 0$. Therefore, $a_1(t)$ and $b_1(t)$ are uniquely determined by (3.48), (3.49). Now the approximation error of the Jacobian matrix adds a perturbation term for the first time: the right hand side of (3.49) at $t = 0$ can be transformed as follows:

$$\begin{aligned} -J_{21}\dot{x}(0) - J_{22}\dot{y}(0) &= \Delta J_{21}\dot{x}(0) + \Delta J_{22}\dot{y}(0) - g_x(0)\dot{x}(0) - g_y(0)\dot{y}(0)\\ &= \Delta J_{21}\dot{x}(0) + \Delta J_{22}\dot{y}(0) - \frac{d}{dt}g(x(t), y(t))\Big|_{t=0}\\ &= \Delta J_{21}\dot{x}(0) + \Delta J_{22}\dot{y}(0) + 0. \end{aligned}$$

In the last step we used that $g(x(t), y(t)) \equiv 0$. Hence, (3.49) and (3.42) yield $b_1(0) = -\beta_0^1 = 0$ if $(\Delta J_{21}, \Delta J_{22}) = 0$. In this case we obtain $\beta_i^1 = 0$ (for all $i$) by (3.51) and $\alpha_i^1 = 0$ (for all $i$) by (3.50), (3.42).

For $\beta_0^1 \neq 0$ the recursion

$$\beta_{i+1}^1 = (I - J_{22}^{-1}g_y(0))\beta_i^1 \qquad (3.52)$$

is non-trivial.

Case $\Delta J_{22} = 0$: First $J_{22} = g_y(0)$ yields $\beta_i^1 = 0$ (for $i \geq 1$) and with (3.50) we obtain $\alpha_i^1 = 0$ (for $i \geq 1$). Case $\Delta J_{12} = 0, \Delta J_{22} = 0$: First $J_{22} = g_y(0)$ yields $\beta_i^1 = 0$ (for $i \geq 1$) and with $J_{12} = f_y(0)$ in (3.50) we obtain $\alpha_i^1 = 0$ (for $i \geq 0$). For the case of a fully perturbed Jacobian matrix, we cannot guarantee for any perturbation terms $\alpha_i^2, \beta_i^1$ to be non-zero. In the higher steps we have the same situation, so we will not consider this case anymore.

*Third Step* ($j = 2$)*:* Case $\Delta J_{12} = 0, \Delta J_{22} = 0$: For the first time we have non-zero $\rho_i^2 = f_y y(0)(\beta_i^1)^2$ and $\sigma_i^2 = g_y y(0)(\beta_i^1)^2$. This is the situation described in the "general step" below.
Case $\Delta J_{22} = 0$: Here we also have $\rho_i^2 \neq 0, \sigma_i^2 \neq 0$ and therefore the same situation as in the general case.
Taking into account the possible simplifications in the remaining cases we get

$$\begin{aligned} f_x(t)a_2(t) + f_y(t)b_2(t) + r(t) &= \dot{a}_2(t) && (3.53)\\ g_x(t)a_2(t) + g_y(t)b_2(t) + s(t) &= 0 && (3.54)\\ \alpha_{i+1}^3 - \alpha_i^3 - J_{12}(\beta_{i+1}^2 - \beta_i^2) &= f_y(0)\beta_i^2 && (3.55)\\ 0 &= g_y(0)\beta_{i+1}^2, && (3.56) \end{aligned}$$

where $r(t), s(t)$ are known functions depending on derivatives of $x(t), y(t)$ and on $a_1(t), b_1(t)$. We compute $a_2(t), b_2(t)$ as in step 2. However, $b_2(0) \neq 0$ in general, and therefore we are forced to introduce a perturbation term $\beta_0^2 \neq 0$.
Case $\Delta J_{21} = 0, \Delta J_{22} = 0$: We have $\alpha_i^2 = 0$ (for $i \geq 0$) and $\beta_i^1 = 0$ (for $i \geq 0$). Hence, (3.55), (3.43) yield $\alpha_0^3 = -(f_y(0) - J_{12})\beta_0^2 \neq 0$ and $\alpha_i^3 = 0$ (for $i \geq 1$)
If additionally $J_{12} = f_y(0)$ then $\alpha_i^3 = 0$ (for all $i \geq 0$) follows from (3.55).
*Fourth Step* ($j = 3$). Case $\Delta J_{12} = 0, \Delta J_{22} = 0$: We have the same situation as in the general step (see below) inserting $j = 3$. Therefore, $\alpha_i^4 = 0$ (for $i \geq 2$) and $\beta_i^3 = 0$ (for $i \geq 3$).
For the other cases we obtain (3.44), (3.45) for the smooth equations and

$$\begin{aligned} \alpha_{i+1}^4 - \alpha_i^4 + J_{11}\alpha_i^3 - J_{12}(\beta_{i+1}^3 - \beta_i^3) &= f_x(0)\alpha_i^3 + f_y(0)\beta_i^3 && (3.57)\\ 0 &= g_y(0)\beta_{i+1}^3, && (3.58) \end{aligned}$$

Case $\Delta J_{21} = 0, \Delta J_{22} = 0$: We have the same situation as in the third step. Therefore, $\alpha_i^4 = 0$ (for $i \geq 1$) and $\beta_i^3 = 0$ (for $i \geq 1$)
Case $\Delta J_{12} = 0, \Delta J_{21} = 0, \Delta J_{22} = 0$: We have the same situation as in the third step. Therefore, $\alpha_i^4 = 0$ (for $i \geq 0$) and $\beta_i^3 = 0$ (for $i \geq 1$)

| $\alpha$ | j-3 | j-2 | j-1 | j | j+1 |
|---|---|---|---|---|---|
| i-2 | 0 | 0 | 0 | * | * |
| i-1 | 0 | 0 | 0 | 0 | * |
| i | $\odot$ | $\odot$ | $\odot$ | $\odot$ | $\otimes$ |
| i+1 | $\odot$ | $\odot$ | $\odot$ | $\odot$ | ? |
| i+2 | 0 | 0 | 0 | 0 | ? |
| i+3 | 0 | 0 | 0 | 0 | ? |

| $\beta$ | j-3 | j-2 | j-1 | j | j+1 |
|---|---|---|---|---|---|
| i-2 | 0 | * | * | * | ? |
| i-1 | 0 | 0 | * | * | ? |
| i | $\odot$ | $\odot$ | $\odot$ | * | ? |
| i+1 | 0 | 0 | 0 | $\otimes$ | ? |
| i+2 | 0 | 0 | 0 | ? | ? |
| i+3 | 0 | 0 | 0 | ? | ? |

Table 3.1: The induction of the general step shown graphically. $\odot$: Element that is assumed to be zero. $\otimes$: Element that is shown to be zero.

*General Step.* In the first few steps, we had $\rho_i^j = \sigma_i^j = 0$ (for $i \geq 0$) in (3.46), (3.47) For the higher steps this assumption is not true anymore. The proof is now by induction on $j$.
Case $\Delta J_{22} = 0$ ($j \geq 2$): By the induction hypothesis we have $\rho_i^j = 0, \sigma_i^j = 0, \alpha_i^j = 0$ for $i \geq j-1$. (3.47) implies $\beta_{i+1}^j = 0$ (for $i \geq j-1$) and (3.46) together with (3.43) gives $\alpha_i^{j+1} = 0$ (for $i \geq j$).
Case $\Delta J_{12} = 0, \Delta J_{22} = 0$ ($j \geq 2$): By the induction hypothesis we have $\rho_i^j = 0, \sigma_i^j = 0, \alpha_i^j = 0$ for $i \geq j-1$. (3.47) implies $\beta_{i+1}^j = 0$ (for $i \geq j-1$) and (3.46) together with (3.43) gives $\alpha_i^{j+1} = 0$ (for $i \geq j-1$).
Case $\Delta J_{21} = 0, \Delta J_{22} = 0$ ($j \geq 4$): By the induction hypothesis we have $\rho_i^j = 0, \sigma_i^j = 0, \alpha_i^j = 0$ for

$i \geq j-3$. (3.47) implies $\beta_{i+1}^j = 0$ (for $i \geq j-3$) and (3.46) together with (3.43) gives $\alpha_i^{j+1} = 0$ (for $i \geq j-2$).
Case $\Delta J_{12} = 0, \Delta J_{21} = 0, \Delta J_{22} = 0$ ($j \geq 4$): By the induction hypothesis we have $\rho_i^j = 0, \sigma_i^j = 0, \alpha_i^j = 0$ for $i \geq j-3$. (3.47) implies $\beta_{i+1}^j = 0$ (for $i \geq j-3$) and (3.46) together with (3.43) gives $\alpha_i^{j+1} = 0$ (for $i \geq j-3$).

**Stability estimate.** We still have to estimate the remainder term, i.e., differences $\Delta x_i = x_i - \hat{x}_i$, $\Delta y_i = y_i - \hat{y}_i$. The important assumption here is that the algebraic part is discretized in a stable way. Subtracting (3.41) from (3.15) and eliminating $\Delta x_{i+1}, \Delta y_{i+1}$ yields

$$\begin{pmatrix} \Delta x_{i+1} \\ \Delta y_{i+1} \end{pmatrix} = \begin{pmatrix} \Delta x_i \\ \Delta y_i \end{pmatrix} + \begin{pmatrix} I + O(\tau) & O(\tau) \\ O(1) & J_{22}^{-1} \end{pmatrix} \begin{pmatrix} \tau(f(x_i, y_i) - f(\hat{x}_i, \hat{y}_i)) \\ g(x_i, y_i) - g(\hat{x}_i, \hat{y}_i) \end{pmatrix} + \begin{pmatrix} O(\tau^{M+2}) \\ O(\tau^{M+1}) \end{pmatrix}$$

The application of a Lipschitz condition for $Df_x, Df_y, Dg_x, Dg_y$ yields

$$\begin{pmatrix} \|\Delta x_{i+1}\| \\ \|\Delta y_{i+1}\| \end{pmatrix} \leq \begin{pmatrix} 1 + O(\tau) & O(\tau) \\ O(1) & \rho \end{pmatrix} \begin{pmatrix} \|\Delta x_i\| \\ \|\Delta y_i\| \end{pmatrix} + \begin{pmatrix} O(\tau^{M+2}) \\ O(\tau^{M+1}) \end{pmatrix} \tag{3.59}$$

where $|\rho| < 1$ if $T$ is sufficiently small: as $\|I - J_{22}^{-1} g_y(0)\| \leq \gamma < 1$, we also have $\|I - J_{22}^{-1} g_y(t)\| \leq \rho < 1$ for $t < T$. $\|\Delta x_i\| + \|\Delta y_i\| = O(\tau^{M+1})$ follows from an estimate based on the stability analysis of 3.59, bounding its eigenvalues. This technical lemma can be looked up at [24], Lemma VI.3.9. □

**Remark 3.1** *It is interesting to observe the role that the approximation error $\Delta J$ of the Jacobian matrix plays in the construction of the asymptotic expansion. In the second step we can see that $(\Delta J_{21}, \Delta J_{22}) = 0$ is essential to achieve $b_1(0) = 0$ in (3.49) and to delay the appearance of the first perturbation term by one. Otherwise, it is at least necessary that $\Delta J_{22} = 0$ to inhibit the propagation by (3.51) of this first perturbation term. The perturbation terms $\alpha_i^j$ appear indirectly via higher order terms $\rho_i^j$ or earlier, if $\Delta J_{12} \neq 0$ in (3.46). For inexact $J_{12}$, a non-zero $\beta_i^j$ directly leads to a non-zero $\alpha_i^{j+1}$. Otherwise this propagation is delayed and $\beta_i^j \neq 0$ only yields a non-zero $\alpha_{i-1}^{j+1}$. We further note that $J_{11}$ has no influence on the perturbed asymptotic expansion.*

**Remark 3.2** *If we analyze the proof we can see that we do not have any perturbations in the case of ordinary differential equations, whatever matrix $J$ is chosen. This is a justification of the statement above, that extrapolation methods based on the linearly implicit Euler method are W-methods.*

**Orders achieved by extrapolation** To construct the order tableaux for the extrapolation methods we proceed like [24] Theorem VI.5.4.

**Theorem 3.4** *If we consider the harmonic sequence $\{1, 2, 3, 4, \ldots\}$, then the extrapolated values $X_{jk}, Y_{jk}$ satisfy*

$$X_{jk} - x(t_0 + T) = O(T^{r_{jk}+1}), \qquad Y_{jk} - y(t_0 + T) = O(T^{s_{jk}}) \tag{3.60}$$

*where the differential-algebraic orders $r_{jk}, s_{jk}$ for the different cases are given in Tables 3.6-3.8. For the case of a fully perturbed Jacobian matrix we have $r_{ik} = s_{ik} = 1$ for all $i, k \geq 0$.*

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | * | * | * |
| 1 | 0 | 0 | 0 | 0 | 0 | * | * |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | * |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | * | * | * | * | * | * |
| 1 | 0 | 0 | 0 | * | * | * | * |
| 2 | 0 | 0 | 0 | 0 | * | * | * |
| 3 | 0 | 0 | 0 | 0 | 0 | * | * |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | * |

Table 3.2: Non-zero $\alpha's$ and non-zero $\beta's$ in the case $\Delta J_{12} = 0, \Delta J_{21} = 0, \Delta J_{22} = 0$ and for an an exact Jacobian matrix

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | X | X | * | * | * |
| 1 | 0 | 0 | 0 | 0 | X | * | * |
| 2 | 0 | 0 | 0 | 0 | 0 | X | * |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | X |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | * | * | * | * | * | * |
| 1 | 0 | 0 | 0 | * | * | * | * |
| 2 | 0 | 0 | 0 | 0 | * | * | * |
| 3 | 0 | 0 | 0 | 0 | 0 | * | * |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | * |

Table 3.3: Non-zero $\alpha's$ and non-zero $\beta's$ in the case $\Delta J_{21} = 0, \Delta J_{22} = 0$ compared to the case of an exact Jacobian matrix (additional perturbations are marked by "X")

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | X | X | * | * | * |
| 1 | 0 | 0 | 0 | X | X | * | * |
| 2 | 0 | 0 | 0 | 0 | X | X | * |
| 3 | 0 | 0 | 0 | 0 | 0 | X | X |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | X |

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | X | * | * | * | * | * | * |
| 1 | 0 | X | X | * | * | * | * |
| 2 | 0 | 0 | X | X | * | * | * |
| 3 | 0 | 0 | 0 | X | X | * | * |
| 4 | 0 | 0 | 0 | 0 | X | X | * |

Table 3.4: Non-zero $\alpha's$ and non-zero $\beta's$ in the case $\Delta J_{12} = 0, \Delta J_{22} = 0$ compared to the case of an exact Jacobian matrix (additional perturbations are marked by "X")

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | X | X | X | * | * | * |
| 1 | 0 | 0 | X | X | X | * | * |
| 2 | 0 | 0 | 0 | X | X | X | * |
| 3 | 0 | 0 | 0 | 0 | X | X | X |
| 4 | 0 | 0 | 0 | 0 | 0 | X | X |

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | X | * | * | * | * | * | * |
| 1 | 0 | X | X | * | * | * | * |
| 2 | 0 | 0 | X | X | * | * | * |
| 3 | 0 | 0 | 0 | X | X | * | * |
| 4 | 0 | 0 | 0 | 0 | X | X | * |

Table 3.5: Non-zero $\alpha's$ and non-zero $\beta's$ in the case $\Delta J_{22} = 0$ compared to the case of an exact Jacobian matrix (additional perturbations are marked by "X")

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 3 | | | | | |
| 1 | 2 | 3 | 4 | | | | |
| 1 | 2 | 3 | 4 | 4 | | | |
| 1 | 2 | 3 | 4 | 4 | 5 | | |
| 1 | 2 | 3 | 4 | 4 | 5 | 5 | |
| 1 | 2 | 3 | 4 | 4 | 5 | 6 | 5 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | | | | | | | |
| 2 | 2 | | | | | | |
| 2 | 2 | 3 | | | | | |
| 2 | 2 | 3 | 4 | | | | |
| 2 | 2 | 3 | 4 | 4 | | | |
| 2 | 2 | 3 | 4 | 5 | 4 | | |
| 2 | 2 | 3 | 4 | 5 | 5 | 4 | |
| 2 | 2 | 3 | 4 | 5 | 6 | 5 | 4 |

Table 3.6: Order tableaux for $r_{jk}, s_{jk}$ in the case $\Delta J_{12} = 0, \Delta J_{21} = 0, \Delta J_{22} = 0$ and in the case of an exact Jacobian matrix

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 2 | | | | | |
| 1 | 2 | 2 | 3 | | | | |
| 1 | 2 | 2 | 3 | 4 | | | |
| 1 | 2 | 2 | 3 | 4 | 4 | | |
| 1 | 2 | 2 | 3 | 4 | 5 | 4 | |
| 1 | 2 | 2 | 3 | 4 | 5 | 5 | 4 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | | | | | | | |
| 2 | 2 | | | | | | |
| 2 | 2 | 3 | | | | | |
| 2 | 2 | 3 | 4 | | | | |
| 2 | 2 | 3 | 4 | 4 | | | |
| 2 | 2 | 3 | 4 | 5 | 4 | | |
| 2 | 2 | 3 | 4 | 5 | 5 | 4 | |
| 2 | 2 | 3 | 4 | 5 | 6 | 5 | 4 |

Table 3.7: Order tableaux for $r_{jk}, s_{jk}$ in the case $\Delta J_{21} = 0, \Delta J_{22} = 0$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 2 | | | | | |
| 1 | 2 | 2 | 3 | | | | |
| 1 | 2 | 2 | 3 | 3 | | | |
| 1 | 2 | 2 | 3 | 4 | 3 | | |
| 1 | 2 | 2 | 3 | 4 | 4 | 3 | |
| 1 | 2 | 2 | 3 | 4 | 5 | 4 | 3 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 2 | | | | | |
| 1 | 2 | 3 | 2 | | | | |
| 1 | 2 | 3 | 3 | 2 | | | |
| 1 | 2 | 3 | 4 | 3 | 2 | | |
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | |
| 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 |

Table 3.8: Order tableaux for $r_{jk}, s_{jk}$ in the case $\Delta J_{12} = 0, \Delta J_{22} = 0$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 2 | | | | | |
| 1 | 2 | 2 | 2 | | | | |
| 1 | 2 | 2 | 3 | 2 | | | |
| 1 | 2 | 2 | 3 | 3 | 2 | | |
| 1 | 2 | 2 | 3 | 4 | 3 | 2 | |
| 1 | 2 | 2 | 3 | 4 | 4 | 3 | 2 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 1 | 2 | | | | | | |
| 1 | 2 | 2 | | | | | |
| 1 | 2 | 3 | 2 | | | | |
| 1 | 2 | 3 | 3 | 2 | | | |
| 1 | 2 | 3 | 4 | 3 | 2 | | |
| 1 | 2 | 3 | 4 | 4 | 3 | 2 | |
| 1 | 2 | 3 | 4 | 5 | 4 | 3 | 2 |

Table 3.9: Order tableaux for $r_{jk}, s_{jk}$ in the case $\Delta J_{22} = 0$

**Proof**: The proof is similar to the proof of [24] Theorem VI.5.4. It is based on two important observations: first $\alpha_0^j \neq 0, \beta_0^j \neq 0$ is equivalent to $a_j(0) \neq 0, b_j(0) \neq 0$. This leads to $a_j(t_0+T) = O(1), b_j(t_0+T) = O(1)$ and therefore to an order reduction of 1 for the whole column $j$ in the tableau.
Secondly, during the extrapolation process only the smooth parts of the expansion are eliminated but not the perturbation terms. So if $\alpha_i^j \neq 0, \beta_i^j \neq 0$ then after extrapolation using the values $X_{ji}, Y_{ji}$ an error of $O(T^j)$ remains. Therefore, the $i-1^{st}$ subdiagonal in the order tableaux for $r_{lk}$ will be not larger than $j-1$ and the $i-1^{st}$ subdiagonal in the order tableaux for $s_{lk}$ will be not larger than $j$. □

## 3.5 Quasi-Linear systems with a solution dependent mass matrix

If the mass matrix is not constant anymore as in (3.7), the question of convergence order becomes much more difficult than before. The local error of extrapolation methods in this case was studied in [29] and a proof of convergence for Rosenbrock methods in this case can be found in [30]. It can be applied to linearly implicit extrapolation methods as well if the assumption of an exact Jacobian matrix is made. However, the convergence theorem is valid only for methods of order 2 and higher. The reason is, that for convergence in this case it is necessary that the state derivatives $\dot{x}$ converge, too. But the derivatives are in general approximated with lower order than the states and therefore an order 1 method is not guaranteed to converge in this case. This situation is also reflected if we consider the asymptotic expansions of the linearly implicit Euler method (see [29]) in this case. In contrast to the expansion for a constant mass matrix, where we had the condition $\tau n \leq T$, we have now the additional condition $\tau n^2 \leq K$.
We have seen that for extrapolation methods applied to DAEs it is important that the algebraic part of the Jacobian matrix $J_{22}$ remains unperturbed. Otherwise, we obtain a method of order 1. In the case of a solution dependent mass matrix we cannot guarantee this property anymore. So if we perform static sparsing, we will at most obtain a method of consistency order 1 and this is not sufficient for convergence.
For these reasons, we cannot apply static sparsing in the case of a non-constant mass matrix, at least if no additional assumptions about the mass matrix can be made.

# Chapter 4

# A Sparsing Criterion

To find an appropriate sparsity structure of the Jacobian matrix for the linearly implicit Euler method, a preprocessing routine is necessary to analyze the effects of sparsing on the dynamical behavior of the numerical method. The core of such a routine is a cheap sparsing criterion that estimates those effects for each non-zero element of the Jacobian.
In the off-line case and for ODEs it may pay off to use a very simple criterion based on the magnitude of the entries and apply it dynamically during the simulation run (see [33]). However, concerning differential algebraic equations, the magnitude of an element is not directly related to its influence on the eigenvalues. A simple counter example is the linear equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \dot{x} = \begin{pmatrix} \alpha & 0 & \varepsilon \\ 0 & \beta & 0 \\ \varepsilon & 0 & (\alpha-\gamma)^{-1}\varepsilon^2 \end{pmatrix} x. \tag{4.1}$$

We can transform this equation to the ordinary differential equation

$$\dot{\tilde{x}} = \begin{pmatrix} \gamma & 0 \\ 0 & \beta \end{pmatrix} \tilde{x}. \tag{4.2}$$

We see that small matrix elements (for small $\varepsilon$) in the algebraic equations may have large influence on the elements of the underlying differential equation ($\alpha$ is replaced by an arbitrary $\gamma$) and therefore on the dynamical behavior of the continuous and the discrete system. So even if we only perform sparsing of the differential part, we have to take into account the influence of the algebraic equations. At the end of this chapter we will return to this point again, then equipped with a deeper theoretical understanding.
We have seen in Chapter 3 that the stability of the numerical integration scheme applied to a linear problem $L\dot{x} = Dfx$ depends on the eigenvalues of the discrete evolution of the inexact linearly implicit Euler method, i.e., the matrix pair

$$(L + \tau\Delta J, L - \tau Df + \tau\Delta J), \tag{4.3}$$

where the unperturbed matrix $L - \tau Df$ is non-singular, because we assume differential algebraic equations of index 1 or lower. We have seen that if $|\lambda| < 1$ for all eigenvalues $\lambda$ of (4.3) then the discrete evolution is asymptotically stable.
For small perturbation matrices $\Delta J$ it is feasible to estimate the changes of the eigenvalues with the help of linear perturbation theory. For this purpose the system is first transformed to block diagonal form and then the change of the eigenvalues is estimated for each block.

## 4.1 Simultaneous block diagonalization of matrix pairs

In this section we examine the generalized eigenproblem more closely and derive a numerical algorithm to obtain a block diagonal form of a matrix pair $(A, B)$, where both $A$ and $B$ are square matrices of order $n$, or - more generally - representations of mappings

$$\begin{aligned} A : V &\rightarrow W, \\ B : V &\rightarrow W, \end{aligned}$$

with $V, W \cong \mathbb{K}^n (\mathbb{K} = \mathbb{R}$, or $\mathbb{K} = \mathbb{C})$. The notation and the definitions are conformal to the textbook "Matrix Perturbation Theory" by Stewart and Sun ([41]) that contains a chapter about the generalized eigenvalue problem. The only major difference to the notation there is that we do not use the notion of a generalized eigenvalue as a pair of complex numbers. This notation is necessary if one wants to include the case of a singular matrix $B$. But as stated above, our matrix $B = L - \tau Df$ is always non-singular. We will use [41] as a starting point for the following and start with some basic definitions and standard results.

**Definition 4.1** *Let $(A, B)$ be a matrix pair with non-singular $B$. Then the solutions $\lambda$ of the "characteristic equation" $det(A - \lambda B) = 0$ are the "eigenvalues" of $(A, B)$. If $\lambda$ is an eigenvalue of $(A, B)$ then a solution $x \neq 0$ of*

$$Ax = \lambda Bx \tag{4.4}$$

*is a "right eigenvector" of $(A, B)$ and a solution $y \neq 0$ of*

$$y^H A = \lambda y^H B \tag{4.5}$$

*is a "left eigenvector" of $(A, B)$. The set of all eigenvalues of $(A, B)$ is called the spectrum $\mathcal{L}[(A, B)]$ of $(A, B)$.*

As $B$ is non-singular we can reduce the generalized eigenproblem (4.4) to the standard eigenproblem

$$B^{-1}Ax = \lambda x \tag{4.6}$$

that has the same eigenvalues and eigenvectors. Therefore, all standard results that apply to (4.6) also apply to (4.4). Just like in the standard eigenproblem we define a class of transformations that preserve the eigenvalues and transform the eigenvectors in a simple manner.

**Definition 4.2** *If $X$ and $Y$ are non-singular, then the pairs $(A, B)$ and $(Y^H AX, Y^H BX)$ are "equivalent".*

If we interpret $A, B$ as mappings $V \rightarrow W$, then an equivalence transformation can be interpreted as a coordinate transformation of the spaces $V$ and $W$.

**Theorem 4.3** *Let $\lambda$ be an eigenvalue of $(A, B)$ with right and left eigenvectors $x$ and $y$. Then $\lambda$ is an eigenvalue of the equivalent pair $(Y^H AX, Y^H BX)$ with right eigenvector $X^{-1}x$ and left eigenvector $Y^{-1}y$.*

**Proof**: The equivalence transformation for (4.4) reduces to a similarity transformation for (4.6) which yields the proposition for right eigenvectors. For left eigenvectors we reduce (4.4) to $y^H AB^{-1} = \lambda y^H$ to obtain the proposition. □

As our main interest is on block diagonal forms we define

**Definition 4.4** *Let $(A, B)$ be a regular matrix pair. The subspace $\mathcal{X}$ is an eigenspace if*

$$\dim(A\mathcal{X} \oplus B\mathcal{X}) \leq \dim(\mathcal{X}) \tag{4.7}$$

As $B$ is non-singular this inequality is clearly an equality and $A\mathcal{X} \subset B\mathcal{X}$. If we restrict the pair of mappings $(A, B)$ on $\mathcal{X}$, then $B|_{\mathcal{X}} : \mathcal{X} \to B\mathcal{X}$ is invertible and $\mathcal{X}$ is an invariant subspace of $B^{-1}A$. Therefore, $\mathcal{L}[(A|_{\mathcal{X}}, B|_{\mathcal{X}})] \subset \mathcal{L}[(A, B)]$ and intersections and direct sums of eigenspaces are eigenspaces again.

**Definition 4.5** *Let $\mathcal{X}$ be an eigenspace of $(A, B)$. $\mathcal{X}$ is called a simple eigenspace of $(A, B)$ if for any eigenspace $\tilde{\mathcal{X}}$*

$$\mathcal{L}[(A|_{\mathcal{X}}, B|_{\mathcal{X}})] = \mathcal{L}[(A|_{\tilde{\mathcal{X}}}, B|_{\tilde{\mathcal{X}}})] \neq \emptyset \Rightarrow \tilde{\mathcal{X}} \subset \mathcal{X}.$$

For later use we note that for two eigenspaces $\tilde{\mathcal{X}}, \mathcal{X}$ with disjunct spectrum we have the relation

$$\tilde{\mathcal{X}} \cap \mathcal{X} = 0, \tag{4.8}$$

because $\mathcal{L}[(A|_{\tilde{\mathcal{X}}\cap\mathcal{X}}, B|_{\tilde{\mathcal{X}}\cap\mathcal{X}})] = \mathcal{L}[(A|_{\mathcal{X}}, B|_{\mathcal{X}})] \cap \mathcal{L}[(A|_{\tilde{\mathcal{X}}}, B|_{\tilde{\mathcal{X}}})] = \emptyset$.

### 4.1.1 Triangularization

The numerical solution of the non-symmetric standard eigenvalue problem is performed by unitary similarity transformations to upper triangular form. The standard algorithm for this is the QR-algorithm and the result is called Schur form. For the generalized eigenvalue problem we use unitary equivalence transformations to obtain an upper triangular form for the matrix pair $(A, B)$, i.e., both matrices are in upper triangular form. The algorithm used is called QZ-algorithm (an extension of the QR-algorithm) and the result is called generalized Schur form. For a more detailed introduction to this field including references to original papers, see [22]. For existence of the generalized Schur form we cite [41] Theorem VI.1.9

**Theorem 4.6** *Let $(A, B)$ be a regular matrix pair. Then there are unitary matrices $Q$ and $Z$ such that the components of the equivalent pair*

$$(A_T, B_T) = (Q^H AZ, Q^H BZ) \tag{4.9}$$

*are upper triangular. The quantities $\lambda_i = (A_T)_{ii}/(B_T)_{ii} (i = 1 \ldots n)$ are the eigenvalues of $(A, B)$ and may be made to appear in any order on the diagonals of $A_T, B_T$.*

**Proof**: See [41] Theorem VI.1.9. The proof is similar to the proof for the standard Schur form. □

The columns of $Q$ and $Z$ are called left and right Schur vectors. They have the following property:

**Lemma 4.7** *Let (4.9) be the generalized Schur form of the matrix pair $(A, B)$. Then the subspace $\mathcal{Z}$ spanned by the first $k$ $(1 \leq k \leq n)$ columns of $Z$ is a right eigenspace of $(A, B)$ and accordingly the subspace $\mathcal{Q}$ spanned by the last $k$ $(1 \leq k \leq n)$ columns of $Q$ is a left eigenspace of $(A, B)$. The spectrum $\mathcal{L}[(A|_{\mathcal{Z}}, B|_{\mathcal{Z}})]$ of the restriction of $(A, B)$ to $\mathcal{Z}$ is given by the quotient of the first $k$ diagonal elements of $(A_T, B_T)$:*

$$\mathcal{L}[(A|_{\mathcal{Z}}, B|_{\mathcal{Z}})] = \{\lambda_i | \lambda_i = (A_T)_{ii}/(B_T)_{ii},\ i = 1 \ldots k\} \tag{4.10}$$

*and analogously*

$$\mathcal{L}[({}_{\mathcal{Q}}|A, {}_{\mathcal{Q}}|B)] = \{\lambda_i | \lambda_i = (A_T)_{ii}/(B_T)_{ii},\ i = n - k + 1 \ldots n\}. \tag{4.11}$$

**Proof**: Let

$$r = \sum_{i=1}^{k} \sigma_i z_i = Zs$$

be a linear combination of the first columns of $Z$. Then we compute:

$$Ar = QA_T Z^H Zs = QA_T s.$$

As $A_T$ is upper triangular, the vector $t = A_T s$ has the structure $t = (\tau_1, \ldots, \tau_k, 0, \ldots, 0)^T$, too. Hence,

$$Ar = \sum_{i=1}^{k} \tau_i q_i$$

. The same considerations are true for $B$ and its triangular form $B_T$. Thus

$$A\mathcal{Z} \oplus B\mathcal{Z} \in \text{span}(q_1, \ldots, q_k)$$

and therefore

$$\dim(A\mathcal{Z} \oplus B\mathcal{Z}) \leq \dim(\mathcal{Z}),$$

which proofs that $\mathcal{X}$ is an eigenspace of $(A, B)$. The result about the eigenvalues is due to the fact that the left upper block of $(A_T, B_T)$ is a representation of $(A|_{\mathcal{Z}}, B|_{\mathcal{Z}})$.
If we do the same with $(A^H, B^H) = (ZA_T^H Q^H, ZB_T^H Q^H)$ we obtain the same results for the last $k$ columns of $Q$, as $(A_T^H, B_T^H)$ is lower triangular. □

We see that we obtain a pair of sequences of right and left $i$-dimensional eigenspaces:

$$\begin{array}{rl} \mathcal{Z}_1 \subset \mathcal{Z}_2 \subset \ldots & \subset \mathcal{Z}_i \subset \quad \ldots \subset \mathcal{Z}_{n-1} \subset V \\ W \supset \mathcal{Q}_{n-1} \supset \ldots & \supset \mathcal{Q}_i \supset \quad \ldots \supset \mathcal{Q}_2 \supset \mathcal{Q}_1 \end{array} \tag{4.12}$$

such that

$$(A\mathcal{Z}_i, B\mathcal{Z}_i) \subset (\mathcal{Q}_{n-i}^{\perp}, \mathcal{Q}_{n-i}^{\perp}), \tag{4.13}$$

i.e.,

$$z \in \mathcal{Z}_i,\ q \in \mathcal{Q}_{n-i} \quad \Rightarrow \quad q^H Az = 0,\ q^H Bz = 0. \tag{4.14}$$

On the other hand, if we have a sequence (4.12) with (4.13) then the representation of $(A, B)$ corresponding to this sequence is upper triangular.

### 4.1.2 Ordering Eigenvalues in the generalized Schur form

We have seen in the last section that in the generalized Schur form the first $k$ columns of $Z$ span a $k$-dimensional eigenspace $\mathcal{Z}$ of $(A, B)$ and $\mathcal{L}[(A|_{\mathcal{Z}}, B|_{\mathcal{Z}})] = \{(A_T)_{ii}/(B_T)_{ii},\ i = 1 \ldots k\}$. So far we had to accept the ordering of the diagonal elements as given by the QZ-algorithm restricting the possibilities to choose $\mathcal{Z}$. Theorem 4.6 states, however, that the pairs of diagonal elements can be made to appear in any order. In this section we are going to realize this proposition algorithmically. An algorithm for the (standard) Schur form by Givens rotations is described in [22] Section 7.6.2 (original papers [38], [42]). This algorithm can be extended for the generalized Schur form in a straightforward way (see [43]). We are going to derive this generalization along the lines of [22].

The core of the algorithm, exchanging two subsequent eigenvalues, can be derived looking at the $2 \times 2$ problem. Suppose

$$(Q^H AZ, Q^H AZ) = (A_T, B_T) = \left( \begin{pmatrix} \alpha_1 & a \\ 0 & \alpha_2 \end{pmatrix}, \begin{pmatrix} \beta_1 & b \\ 0 & \beta_2 \end{pmatrix} \right) \qquad \frac{\alpha_1}{\beta_1} \neq \frac{\alpha_2}{\beta_2} \tag{4.15}$$

and that we wish to reverse the order of the pairs $(\alpha_i, \beta_i)$. Note that $\beta_2 A_T x = \alpha_2 B_T x = \alpha_2 \beta_2 y$, where

$$x = \begin{pmatrix} \alpha_2 b - \beta_2 a \\ \alpha_1 \beta_2 - \alpha_2 \beta_1 \end{pmatrix}, \tag{4.16}$$

and

$$y = \begin{pmatrix} \alpha_1 b - \beta_1 a \\ \alpha_1 \beta_2 - \alpha_2 \beta_1 \end{pmatrix}. \tag{4.17}$$

Let $G$ be a Givens rotation such that $G^H x_2 = 0$, and $H$ be a Givens rotation such that $H y_2 = 0$. Set $\tilde{Z} = G^H Z$,$\tilde{Q} = H^H Z$, then

$$\begin{aligned} \tilde{Q}^H A \tilde{Z} \|x\| e_1 = H A_T G^H \|x\| e_1 = H A_T x = H \alpha_2 y = \alpha_2 \|y\| e_1, \\ \tilde{Q}^H B \tilde{Z} \|x\| e_1 = H B_T G^H \|x\| e_1 = H B_T x = H \beta_2 y = \beta_2 \|y\| e_1. \end{aligned}$$

Therefore, $(\tilde{Q}^H A \tilde{Z}, \tilde{Q}^H B \tilde{Z})$ must have the form

$$(\tilde{Q}^H A \tilde{Z}, \tilde{Q}^H B \tilde{Z}) = \left( \begin{pmatrix} \alpha_2 \rho & * \\ 0 & \alpha_1 / \rho \end{pmatrix}, \begin{pmatrix} \beta_2 \rho & * \\ 0 & \beta_1 / \rho \end{pmatrix} \right), \tag{4.18}$$

with $\rho = \frac{\|y\|}{\|x\|}$. We see that the diagonal elements are scaled by this algorithm. However, as we use orthogonal transformations, the norms of the matrices $A$ and $B$ are preserved, and therefore the size of the diagonal elements is bounded.

We can now use this technique to interchange subsequent diagonal elements to order these elements in a specified way. A simple strategy is implemented in Algorithm 4.1. As input it requires the generalized Schur factorization $(A_T, B_T) = (Q^H AZ, Q^H BZ)$ of the matrix pair $(A, B)$, namely the matrices $Q^H =$ `Q`, $Z =$ `Z`, $A_T =$ `AT`, $B_T =$ `BT` and a set $J \subset \{1, \ldots, n\}$ of the indices of the diagonal elements to be sorted to the top. $J$ is represented by a vector `J` of flags: `J(i)` $= 1 \Rightarrow i \in J$, `J(i)` $= 0 \Rightarrow i \in \bar{J}$.

As long as not all diagonal elements corresponding to $J$ are on top, Algorithm 4.1 exchanges two pairs of diagonal elements, if the index of the upper pair is in $\bar{J}$ and the index of the lower pair is in $J$. This is accomplished by two Givens rotations applied to $(A_T, B_T)$ and $Z, Q$. Then the lower index is removed from $J$ and the upper index is included into $J$.

### 4.1.3 Block diagonalization

If the eigenvalues are known, the next step would be to compute the eigenvectors as they contain information about the behavior of the eigenvalues under perturbations. Unfortunately, for non-symmetric matrix pairs the eigenvector problem is in general badly conditioned. Especially if eigenvalues are close together eigenvector sensitivity becomes large (see again [41] or [22]).

In technical applications, multiple eigenvalues and non-trivial Jordan blocks are very common. Moreover, algebraic equations lead to a multiple eigenvalue 0 with sometimes very high multiplicity. Hence, a basis of eigenvectors does not exist, or cannot be computed reliably in such cases. To construct a robust algorithm we have to be satisfied with less: To given disjunct subsets of eigenvalues (with well conditioned corresponding eigenspaces) compute right and left

**Algorithm 4.1** *Algorithm in MATLAB notation to sort the diagonal entries of a matrix pair in generalized Schur form by Givens rotations.*

```
function [AT,BT,Q,Z]=sortqz(AT,BT,Q,Z,J)

sz=sum(J);
n=size(AT,2);

while sum(J(1:sz)) < sz
     for k=1:n-1
      if J(k)==0 & J(k+1)==1

         G=planerot([BT(k+1,k+1)*AT(k,k+1)  - AT(k+1,k+1)*BT(k,k+1);
                     BT(k,k)    *AT(k+1,k+1)- BT(k+1,k+1)*AT(k,k)   ]);
         H=planerot([BT(k,k)    *AT(k,k+1)  - AT(k,k)    *BT(k,k+1);
                     BT(k,k)    *AT(k+1,k+1)- BT(k+1,k+1)*AT(k,k)]);

         AT(k:k+1,k:n)  =H*AT(k:k+1,k:n);
         AT(1:k+1,k:k+1)=  AT(1:k+1,k:k+1)*G';

         BT(k:k+1,k:n)  =H*BT(k:k+1,k:n);
         BT(1:k+1,k:k+1)=  BT(1:k+1,k:k+1)*G';

         Z(1:end,k:k+1)=Z(1:end,k:k+1)*G';
         Q(k:k+1,1:end)=H*Q(k:k+1,1:end);

         J([k,k+1])=J([k+1,k]);
      end
   end
end
```

orthogonal bases. The unions of these bases yield two transformation matrices $X$ and $Y$ such that

$$Y^H A X = A_B, \qquad Y^H B X = B_B, \tag{4.19}$$

where $A_B$ and $B_B$ have the same block diagonal structure. This kind of factorization is sometimes called "spectral resolution". In the following we will derive and justify an algorithm that produces such a spectral resolution. The standard approach to obtain a block diagonal form is via solving a Sylvester equation which is also used in [41] to proof the existence of the spectral resolution. The columns of $X$ and $Y$ can be chosen in such a way that subsets of columns corresponding to the blocks are orthogonal. However, solving the Sylvester equation does not lead to such a structure. Hence, to obtain the block diagonal form, we use a different algorithm that preserves orthogonality inside the blocks. To derive this algorithm we start with the generalized Schur form (4.9) and use the eigenspaces of the matrix pair $(A, B)$ that appear in Lemma 4.7. We can choose the diagonal elements of $(A_T, B_T)$ in arbitrary order. Therefore, we obtain:

**Corollary 4.8** *For a given subset $\mathcal{S} \subset \mathcal{L}[(A, B)]$ there is a unique simple right eigenspace $\mathcal{X}_\mathcal{S}$ and a unique simple left eigenspace $\mathcal{Y}_\mathcal{S}$ such that $\mathcal{L}[(A|_\mathcal{X}, B|_\mathcal{X})] = \mathcal{L}[(_\mathcal{Y}|A, _\mathcal{Y}|B)] = \mathcal{S}$.*

**Proof**: Existence of $\mathcal{X}_\mathcal{S}$ follows from Lemma 4.7, as we can choose the generalized Schur form in a way that all pairs of diagonal elements with $\lambda_i = (A_T)_{ii}/(B_T)_{ii} \in \mathcal{S}$ are sorted to the

top. Then the corresponding first columns of $X$ are a basis of $\mathcal{X}_\mathcal{S}$.
If there are two simple eigenspaces $\mathcal{X}_\mathcal{S}$ and $\tilde{\mathcal{X}}_\mathcal{S}$ then by definition we have $\mathcal{X}_\mathcal{S} \subset \tilde{\mathcal{X}}_\mathcal{S}$ and $\mathcal{X}_\mathcal{S} \supset \tilde{\mathcal{X}}_\mathcal{S}$. Therefore, they are equal.
The same argumentation is true for the left eigenspace $\mathcal{Y}_\mathcal{S}$. □

We call $\mathcal{X}_\mathcal{S}$ ($\mathcal{Y}_\mathcal{S}$) the simple right (left) eigenspace corresponding to $\mathcal{S}$. With Lemma 4.7 and Corollary 4.8 we can now proof a theorem about block diagonalization.

**Theorem 4.9** *Let $\mathcal{S} \subset \mathcal{L}[(A,B)]$. Then there are nonsingular matrices $X = (X_1\ X_2)$ and $Y = (Y_1\ Y_2)$ such that*

$$\begin{aligned} \begin{pmatrix} Y_1^H \\ Y_2^H \end{pmatrix} A \begin{pmatrix} X_1 & X_2 \end{pmatrix} &= \begin{pmatrix} A_1 & 0 \\ 0 & A_2 \end{pmatrix} \\ \begin{pmatrix} Y_1^H \\ Y_2^H \end{pmatrix} B \begin{pmatrix} X_1 & X_2 \end{pmatrix} &= \begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} \end{aligned} \tag{4.20}$$

*and*

$$\mathcal{L}[(A_1,B_1)] = \mathcal{S}, \qquad \mathcal{L}[(A_2,B_2)] = \bar{\mathcal{S}} = \mathcal{L}[(A,B)] \setminus \mathcal{S}.$$

*The matrices $X_1, X_2, Y_1, Y_2$ can be chosen such that their columns form orthogonal bases for the simple right and left eigenspaces $\mathcal{X}_\mathcal{S}, \mathcal{X}_{\bar{\mathcal{S}}}, \mathcal{Y}_\mathcal{S}, \mathcal{Y}_{\bar{\mathcal{S}}}$ corresponding to $\mathcal{S}$.*

**Proof**: We consider two Schur forms of $(A,B)$,$A, B \in \mathbb{K}^{n\times n}$.

$$\begin{aligned} (A_T^{(1)}, B_T^{(1)}) &= (Q^{(1)H} A Z^{(1)}, Q^{(1)H} B Z^{(1)}) \\ (A_T^{(2)}, B_T^{(2)}) &= (Q^{(2)H} A Z^{(2)}, Q^{(2)H} B Z^{(2)}) \end{aligned}$$

The first with all $\lambda \in \mathcal{S}$ on top of the diagonal (the first $k$ diagonal entries) and the second with all $\lambda \in \mathcal{S}$ at the bottom (the last $k$ diagonal entries). According to Lemma 4.7 we choose the matrices

$$\begin{aligned} X_1 &= Z^{(1)}(:, 1:k), \\ X_2 &= Z^{(2)}(:, 1:n-k), \\ Y_1 &= Q^{(2)}(:, k+1:n), \\ Y_2 &= Q^{(1)}(:, n-k+1:n), \end{aligned}$$

(MATLAB notation) whose orthogonal columns span the unique simple right and left eigenspaces $\mathcal{X}_\mathcal{S}, \mathcal{X}_{\bar{\mathcal{S}}}, \mathcal{Y}_\mathcal{S}, \mathcal{Y}_{\bar{\mathcal{S}}}$ due to Corollary 4.8. As $A_T^{(1)}, B_T^{(1)}, A_T^{(2)}, B_T^{(2)}$ are upper triangular matrices, we have

$$\begin{aligned} Y_2^H A X_1 = 0 \qquad & Y_2^H B X_1 = 0 \\ Y_1^H A X_2 = 0 \qquad & Y_1^H B X_2 = 0 \end{aligned}$$

which proofs that we obtain block diagonal matrices. Furthermore $(X_1, X_2)$ and $(Y_1, Y_2)$ are invertible due to (4.8), because $\mathcal{X}_\mathcal{S}$ and $\mathcal{X}_{\bar{\mathcal{S}}}$ as well as $\mathcal{Y}_\mathcal{S}$ and $\mathcal{Y}_{\bar{\mathcal{S}}}$ have disjunct spectrum. □

We call $\mathcal{X}_2$ the complementary eigenspace and $\mathcal{Y}_1$ the corresponding left eigenspace to $\mathcal{X}_1$. Obviously $\mathcal{Y}_1$ is an eigenspace of the matrix pair $(A^H, B^H)$ as it is equivalent to the pair $(X^H A^H Y, X^H B^H Y)$, which is block diagonal as well. Comparing the diagonal blocks we also see

that $\mathcal{L}[(A|_{\mathcal{X}_1}, B|_{\mathcal{X}_1})] = \overline{\mathcal{L}[(A^H|_{\mathcal{Y}_1}, B^H|_{\mathcal{Y}_1})]}$. This theorem generalizes to multiple block structures in a straightforward way.
We observe that in Algorithm 4.1 columns of $Z$ and $Q$ corresponding to elements of $J$ are only rotated against elements of $\bar{J}$, which gives rise to the following tightening of Theorem 4.9.

**Theorem 4.10** *Notation and assumptions as in Theorem 4.9. If we construct the matrices $X^{(1)}, X^{(2)}, Y^{(1)}, Y^{(2)}$ in the proof of Theorem 4.9 with the help of Algorithm 4.1, then the pairs of blocks $(A_1, B_1)$ and $(A_2, B_2)$ are upper triangular.*

**Proof**: With Algorithm 4.1 we construct the two sequences of eigenspaces

$$\begin{aligned} \mathcal{Z}_1^{(1)} \subset \mathcal{Z}_2^{(1)} \subset \ldots \subset \mathcal{Z}_k^{(1)} = \mathcal{X}_S \subset \ldots \subset \mathcal{Z}_{n-1}^{(1)} \subset \mathcal{Z}_n^{(1)} = V \\ W = \mathcal{Q}_n^{(1)} \supset \mathcal{Q}_{n-1}^{(1)} \supset \ldots \supset \mathcal{Y}_{\bar{S}} = \mathcal{Q}_{n-k}^{(1)} \supset \ldots \supset \mathcal{Q}_2^{(1)} \supset \mathcal{Q}_1^{(1)} \end{aligned} \tag{4.21}$$

and

$$\begin{aligned} \mathcal{Z}_1^{(2)} \subset \mathcal{Z}_2^{(2)} \subset \ldots \subset \mathcal{Z}_{n-k}^{(2)} = \mathcal{X}_{\bar{S}} \subset \ldots \subset \mathcal{Z}_{n-1}^{(2)} \subset \mathcal{Z}_n^{(2)} = V \\ W = \mathcal{Q}_n^{(2)} \supset \mathcal{Q}_{n-1}^{(2)} \supset \ldots \supset \mathcal{Y}_S = \mathcal{Q}_k^{(2)} \supset \ldots \supset \mathcal{Q}_2^{(2)} \supset \mathcal{Q}_1^{(2)}, \end{aligned} \tag{4.22}$$

both with the property (4.13). The first $i$ columns of $Z^{(l)}$ span $\mathcal{Z}_i^{(l)}$ and the last $i$ columns of $Q^{(l)}$ span $\mathcal{Q}_i^{(l)}$. Now we choose the matrices $X_1, X_2, Y_1, Y_2$ as in the proof of Theorem 4.9. Hence, we obtain the sequences

$$\begin{aligned} \mathcal{Z}_1^{(1)} \subset \mathcal{Z}_2^{(1)} \subset \ldots \subset \mathcal{Z}_k^{(1)} = \mathcal{X}_S \subset \mathcal{X}_S \oplus \mathcal{Z}_1^{(2)} \subset \ldots \subset \mathcal{X}_S \oplus \mathcal{Z}_{n-k}^{(2)} = V \\ W = \mathcal{Y}_{\bar{S}} \oplus \mathcal{Q}_k^{(2)} \supset \ldots \supset \mathcal{Y}_{\bar{S}} \oplus \mathcal{Q}_1^{(2)} \supset \mathcal{Y}_{\bar{S}} = \mathcal{Q}_{n-k}^{(1)} \supset \ldots \supset \mathcal{Q}_2^{(1)} \supset \mathcal{Q}_1^{(1)}, \end{aligned} \tag{4.23}$$

for which we have to proof the relation (4.13).
For this purpose it is sufficient to show that

$$\mathcal{Z}_{k+i}^{(1)} = \mathcal{X}_S \oplus \mathcal{Z}_i^{(2)}, \qquad (i = 0, \ldots n-k), \tag{4.24}$$

and the corresponding result for the left eigenspaces

$$\mathcal{Q}_{n-k+i}^{(1)} = \mathcal{Y}_{\bar{S}} \oplus \mathcal{Q}_i^{(2)}, \qquad (i = 0, \ldots k). \tag{4.25}$$

Then we obtain immediately that (4.23) inherits the property (4.13) form (4.21) which proofs that we obtain upper triangular matrices.
The relation (4.24) can be shown inductively, if we observe how Algorithm 4.1 constructs (4.22) from (4.21). Initially, we have $J = \{k+1, \ldots n\}$. The algorithm moves the diagonal elements corresponding to $J$ one by one to the top, i.e., to the row with the lowest index, that is not already an element of $J$. The intermediate transformation matrices at the point of computation where the $i^{th}$ index of $J$ has just been moved to the top shall be denoted by $Z_i$ and $Q_i$.
Suppose that the subspace $\mathcal{Z}_{i-1}^{(2)}$ has been constructed by Algorithm 4.1 and (4.24) is true (for $i-1$). Assume furthermore that the first $k+i-1$ columns of the intermediate matrix $Z_{i-1}$ still span $\mathcal{Z}_{k+i-1}^{(1)}$. This is trivial for $i = 1$, because $\mathcal{Z}_0^{(2)}$ is the trivial subspace and $Z_0 = Z$ is still unchanged.
To construct $\mathcal{Z}_i^{(2)}$, Algorithm 4.1 changes $Z_{i-1}$ to $Z_i$ while rotating $z_{k+i}^{(1)}$ into $z_i^{(2)}$. This is performed by Givens rotations involving the columns $i, \ldots k+i$ of the intermediate $Z_{i-1}$ (due to the

logic of the algorithm) and therefore involving elements of $\mathcal{Z}_{k+i-1}^{(1)}$ and $z_{k+i}^{(1)}$ (due to the second part of the induction hypothesis). Hence, the orthogonal set of the first $i$ columns of the new $Z_i$ still spans $\mathcal{Z}_{k+i}^{(1)}$. For the same reason we have

$$\mathcal{Z}_{k+i}^{(1)} \supset \mathcal{Z}_{k+i-1}^{(1)} \oplus \mathbb{K} z_i{}^{(2)}. \tag{4.26}$$

Moreover,

$$\mathcal{Z}_{k+i}^{(1)} = \mathcal{Z}_{k+i-1}^{(1)} \oplus \mathbb{K} z_i{}^{(2)}, \tag{4.27}$$

because $\mathcal{Z}_i^{(2)}$ is a simple eigenspace of $(A|_{\mathcal{Z}_{k+i}^{(1)}}, B|_{\mathcal{Z}_{k+i}^{(1)}})$. Now (4.24) (for $i$) follows from (4.24) (for i-1):

$$\mathcal{Z}_{k+i}^{(1)} = \mathcal{Z}_{k+i-1}^{(1)} \oplus \mathbb{K} z_i{}^{(2)} = \mathcal{X}_S \oplus \mathcal{Z}_{i-1}^{(2)} \oplus \mathbb{K} z_i{}^{(2)} = \mathcal{X}_S \oplus \mathcal{Z}_i^{(2)}. \tag{4.28}$$

Relation (4.25) can be shown analogously. □

### 4.1.4 Block diagonalization for large systems with many algebraic equations

In object oriented modelling we often have large differential algebraic systems of equations with a specially structured left hand side matrix $L$:

$$L = \begin{pmatrix} L_1 & 0 \\ 0 & 0 \end{pmatrix}. \tag{4.29}$$

The dimension $d$ of $L_1$ is much lower than the dimension $n$ of $L$ and the Jacobian matrix $J = Df$ is assumed to be sparse. The matrix pair $(A, B)$ inherits this structure:

$$(A, B) = \left( \begin{pmatrix} L_1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} L_1 - \tau J_{11} & -\tau J_{12} \\ -\tau J_{21} & -\tau J_{22} \end{pmatrix} \right). \tag{4.30}$$

The problem is now that the QZ-algorithm applied on this matrix pair is inefficient due to the high dimension of the system. Moreover, we know a lot about the eigenvalues. The eigenvalue 0 has got the algebraic multiplicity $n-d$. This means that the QZ-algorithm takes a lot of time to compute information we already know. In the following we derive an algorithm that transforms $(A, B)$ into block diagonal form by equivalence transformations

$$(S^H A T, S^H B T) = \left( \begin{pmatrix} \tilde{A} & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} \tilde{B} & 0 \\ 0 & -\tau J_{22} \end{pmatrix} \right). \tag{4.31}$$

The matrices $\tilde{A}, \tilde{B}$ are both of order $d$ and the matrix pair $(\tilde{A}, \tilde{B})$ can now be treated with the methods described in Section 4.1.3.
The idea is to transform $B$ first to block upper triangular form without destroying the corresponding block upper triangular structure of $A$. In this step we obtain a matrix $Z$. Then we do the same with $(A^H, B^H)$ and obtain a matrix $Q$. Then we combine those matrices in a way similar to the procedure in Theorem 4.9 and obtain (4.31).
To transform $B$ to block-triangular form, it is convenient to transform it to triangular form. As we are looking for equivalence transformations, the transformations have to be applied to $A$ as well. Standard ways of triangularization are Gaußian elimination or elimination by Householder- or Givens transformations. In our case none of these methods is directly applicable. Householder transformations destroy any structure in the first step. Linear combination of rows results in

fill-in in the right lower block $A_{21}$ of $A$. But this is the block that has to be zeroed. There are some ways to avoid this effect, e.g., permutation of the left upper block to the right lower block, computing a lower triangular form, or using linear combinations of columns instead of rows. Following the concept of orthogonal bases for invariant subspaces, we decided for Givens rotations applied to the columns of $(A, B)$ that yield the transformation matrix $Z$. Givens rotations applied to the columns of $(A^H, B^H)$ yield the transformation matrix $Q$. How to apply Givens rotations to sparse matrices is discussed in Section 5.4, (see also [2], [20]).
Now we are in the following situation:

$$(AZ, BZ) = \left( \begin{pmatrix} * & * \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} * & * \\ 0 & * \end{pmatrix} \right), \tag{4.32}$$

$$(Q^H A, Q^H B) = \left( \begin{pmatrix} * & 0 \\ * & 0 \end{pmatrix}, \begin{pmatrix} * & 0 \\ * & * \end{pmatrix} \right). \tag{4.33}$$

We have to combine these two results so that we obtain (4.31). Straightforward considerations lead to the choice of the following column matrices

$$T = \Big( z_1, \ldots, z_d, e_{d+1}, \ldots, e_n \Big), \tag{4.34}$$

$$S = \Big( e_1, \ldots, e_{n-d}, q_{n-d+1}, \ldots, q_n \Big). \tag{4.35}$$

Here $z_i, q_i$ are the $i^{th}$ columns of $Z, Q$ and $e_i$ is the $i^{th}$ unit vector. Performing block-wise matrix multiplication we obtain

$$(S^H AT, S^H BT) = \left( S^H \begin{pmatrix} * & 0 \\ 0 & 0 \end{pmatrix}, S^H \begin{pmatrix} * & B_{12} \\ 0 & B_{22} \end{pmatrix} \right) \tag{4.36}$$

$$= \left( \begin{pmatrix} \tilde{A} & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} \tilde{B} & 0 \\ 0 & B_{22} \end{pmatrix} \right). \tag{4.37}$$

Of course, we do not have to perform this matrix multiplication explicitly, but we only have to compute the left upper blocks $\tilde{A}, \tilde{B}$ and so we only need the first $d$ columns of $T$ and the last $d$ columns of $S$.

## 4.2 Results from perturbation theory

In this section the theoretical basis for the sparsing criterion is derived. We study the behavior of the eigenvalues of a matrix pair $(A, B)$ if a small perturbation $(E, F)$ is applied. For the theoretical foundation of the following we mainly refer to the book of Stewart and Sun [41]. In contrast to the results presented in [42] our main goal is to achieve first order approximations of eigenvalue perturbations rather than bounds. This is because a sparsing criterion is used to comparing elements with each other rather than estimating a worst case.
As a starting point of our considerations we will cite a very general result of [41] (Theorem 4.12), that contains all the information we will need for our purpose.
The theorem contains some technical quantities, that we have to define first: a norm $\|\cdot\|_{\mathcal{F}}$ for matrix pairs

$$\|(P, Q)\|_{\mathcal{F}} := \max\{\|P\|_F, \|Q\|_F\}, \tag{4.38}$$

and the quantity

$$\mathrm{dif}[(A_1, B_1), (A_2, B_2)] := \inf_{\|(P,Q)\|_{\mathcal{F}}=1} \|(QA_1 + A_2P, QB_1 + B_2P)\|_{\mathcal{F}}. \tag{4.39}$$

The next lemma, cited from [41] shows that dif $\neq 0$ if $(A_1, B_1)$ and $(A_2, B_2)$ have disjunct spectrum.

**Lemma 4.11** *Let $(A_1, B_1)$ and $(A_2, B_2)$ be regular pairs. Then the mapping*

$$\boldsymbol{T} = (P, Q) \mapsto (A_1 P + Q A_2, B_1 P + Q B_2) \tag{4.40}$$

*is non-singular if and only if*

$$\mathcal{L}[(A_1, B_1)] \cap \mathcal{L}[(A_2, B_2)] = \emptyset. \tag{4.41}$$

**Proof**: See [41] Theorem VI.1.11. □

Note that dif$[\cdot]$ is not invariant against multiplication of $(A, B)$ with a common factor. However, if the perturbations are scaled with the same factor, the common factor cancels out in all relevant quantities mentioned in Theorem 4.42.

**Theorem 4.12 (Perturbations of block diagonal matrix pairs)** *Let the regular matrix pair $(A, B)$ have a spectral resolution (4.20). Given a perturbation $(E, F)$, let*

$$\begin{aligned} \begin{pmatrix} Y_1^H \\ Y_2^H \end{pmatrix} E \begin{pmatrix} X_1 & X_2 \end{pmatrix} &= \begin{pmatrix} E_{11} & E_{12} \\ E_{21} & E_{22} \end{pmatrix} \\ \begin{pmatrix} Y_1^H \\ Y_2^H \end{pmatrix} F \begin{pmatrix} X_1 & X_2 \end{pmatrix} &= \begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix} \end{aligned} \tag{4.42}$$

*Set*

$$\begin{aligned} \gamma &= \|(E_{21}, F_{21})\|_{\mathcal{F}}, \\ \eta &= \|(E_{12}, F_{12})\|_{\mathcal{F}}, \\ \delta &= \mathrm{dif}[(A_1, B_1), (A_2, B_2)] - \max\{\|E_{11}\|_F + \|E_{22}\|_F, \|F_{11}\|_F + \|F_{22}\|_F\}. \end{aligned}$$

*If $\delta > 0$ and*

$$\frac{\eta\gamma}{\delta^2} < \frac{1}{4}, \tag{4.43}$$

*Then there are matrices $P$ and $Q$ satisfying*

$$\|(P, Q)\|_{\mathcal{F}} \leq \frac{2\gamma}{\delta + \sqrt{\delta^2 - 4\gamma\eta}} < 2\frac{\gamma}{\delta} \tag{4.44}$$

*such that the columns of*

$$\tilde{X}_1 = X_1 + X_2 P \quad \text{and} \quad \tilde{Y}_2 = Y_2 + Y_1 Q^H$$

*span left and right eigenspaces of $(A + E, B + F)$ corresponding to the regular pairs*

$$(\tilde{A}_1, \tilde{B}_1) = (A_1 + E_{11} + E_{12} P, B_1 + F_{11} + F_{12} P) \tag{4.45}$$

*and*

$$(\tilde{A}_2, \tilde{B}_2) = (A_2 + E_{22} + Q E_{12}, B_2 + F_{22} + Q F_{12}). \tag{4.46}$$

**Proof**: This is a citation of [41] Theorem VI.2.15. The proof is a discussion of the solution $(P, Q)$ of the system of equations

$$\begin{aligned} Q(A_1 + E_{11}) + (A_2 + E_{22})P &= -E_{21} - QE_{12}P \\ Q(B_1 + F_{11}) + (B_2 + F_{22})P &= -F_{21} - QF_{12}P, \end{aligned}$$

that originates from the requirement

$$\tilde{Y}_2 A \tilde{X}_1 = \tilde{Y}_2 B \tilde{X}_1 = 0,$$

which is equivalent to the requirement that the transformed perturbed matrix pair is block upper triangular. □

The transformation matrices $X$ and $Y$ only appear in the theorem due to (4.42). If we consider perturbations $(\varepsilon E_0, \varepsilon F_0)$ we obtain the following asymptotic result.

**Corollary 4.13** *Notation as in Theorem 4.12. Consider for a sufficiently small $\varepsilon_0 > 0$ and perturbations $(E, F) = (\varepsilon E_0, \varepsilon F_0)$ the case $\varepsilon_0 > \varepsilon \longrightarrow 0$ $(\varepsilon > 0)$. Then there is also a block upper triangular form of $(\tilde{A}, \tilde{B}) = (A + E, B + F)$ with pairs of diagonal blocks:*

$$\begin{aligned} (\tilde{A}_1, \tilde{B}_1) &= (A_1 + E_{11} + O(\varepsilon^2), B_1 + F_{11} + O(\varepsilon^2)) \\ (\tilde{A}_2, \tilde{B}_2) &= (A_2 + E_{22} + O(\varepsilon^2), B_2 + F_{22} + O(\varepsilon^2)). \end{aligned}$$

**Proof**: Consider Theorem 4.12. Obviously we have

$$\gamma = O(\varepsilon), \quad \eta = O(\varepsilon), \quad \delta = O(1). \tag{4.47}$$

Therefore, $\|(P, Q)\|_{\mathcal{F}} = O(\varepsilon)$ and $QE_{12}, E_{12}P, QF_{12}, F_{12}P = O(\varepsilon^2)$. □

**Remark 4.1** *Corollary 4.13 states that the non-diagonal blocks of the transformed perturbation matrix can be neglected up to first order. Dealing with finite perturbations the question about the quality of this approximation arises.*
*One obvious factor is the norm of the transformation matrices $Y_i^H, X_j$. The lower it is the smaller are the transformed perturbations $E_{ij}$. This should have an impact on the way of computing these transformations.*
*We also note that for the transformed matrix pair $(SAT, SBT)$ and transformed perturbations $(SET, SFT)$ we obtain first order block perturbations transformed in the same way.*

If we consider only $1 \times 1$ blocks we obtain a result about perturbations of eigenvalues:

**Corollary 4.14** *Consider the diagonalizable matrix pair $(A, B)$ with invertible $B$ and no multiple eigenvalues. Let $Y^H$ and $X$ be the transformation matrices containing the left and right eigenvectors such that $Y^H AX$ and $Y^H BX$ are diagonal matrices. Furthermore consider a sufficiently small $O(\varepsilon)$ perturbation $(E, F)$. Then the diagonal matrix $\tilde{\Lambda}$ of eigenvalues of the perturbed matrix pair $(A + E, B + F)$ can be approximated as follows:*

$$\tilde{\Lambda} = \operatorname{diag}((Y^H(B + F)X)^{-1} Y^H(A + E)X) + O(\varepsilon^2); \tag{4.48}$$

**Proof**: From Theorem 4.12 and the non-singularity of $B$ follows that

$$\tilde{\lambda}_i = \frac{a_i + e_{ii} + O(\varepsilon^2)}{b_i + f_{ii} + O(\varepsilon^2)} = \frac{a_i + e_{ii}}{b_i + f_{ii}} + O(\varepsilon^2);$$

or written in matrix form:

$$\tilde{\Lambda} = \mathrm{diag}((Y^H(B+F)X))^{-1}\mathrm{diag}(Y^H(A+E)X)) + O(\varepsilon^2);$$

Lemma 4.15 below shows that:

$$\mathrm{diag}((Y^H(B+F)X))^{-1}\mathrm{diag}(Y^H(A+E)X)) = \mathrm{diag}((Y^H(B+F)X)^{-1}Y^H(A+E)X) + O(\varepsilon^2);$$

□

**Lemma 4.15** *Let $D_1, D_2$ be diagonal matrices and $E_1, E_2$ sufficiently small $O(\varepsilon)$ perturbations. Then we have:*

$$\mathrm{diag}((D_1+E_1)^{-1}(D_2+E_2)) = (\mathrm{diag}(D_1+E_1))^{-1}\mathrm{diag}(D_2+E_2) + O(\varepsilon^2); \tag{4.49}$$

**Proof**: Without loss of generality assume that $E_1, E_2$ have zero diagonal. Otherwise the diagonal elements of $E_i$ can be moved to the $D_i$. Then it remains to show:

$$\mathrm{diag}((D_1+E_1)^{-1}(D_2+E_2)) = D_1^{-1}D_2 + O(\varepsilon^2);$$

Neumann expansion of the leftmost factor yields:

$$\begin{aligned}\mathrm{diag}((D_1+E_1)^{-1}(D_2+E_2)) &= \mathrm{diag}(D_1^{-1}(I - E_1D_1^{-1} + O(\varepsilon^2))(D_2+E_2)) \\ &= \mathrm{diag}(D_1^{-1}(D_2 - E_1D_1^{-1}D_2 + E_2)) + O(\varepsilon^2);\end{aligned}$$

Due to the fact that $E_i$ multiplied with diagonal matrices still has zero diagonal, all the terms containing $E_i$ drop out and the proof is complete. □

With Corollary 4.14 we can proof a theorem about symmetric perturbations. It will be the theoretical basis for the sparsing criterion in the next section.

**Theorem 4.16** *Let $(A, B)$ be a diagonalizable matrix pair with eigenvalues $\lambda_i$ and with left and right eigenvectors $y_i$ and $x_i$. Assume all eigenvalues to be algebraically simple and $B$ to be invertible. Let $E$ be a sufficiently small $O(\varepsilon)$ perturbation matrix. Let $\tilde{\lambda}_i$ be the eigenvalues of the perturbed matrix pair $(A+E, B+E)$. Then a first order approximation of the sum of the differences between the original and the perturbed eigenvalues is given by the trace of $B^{-1}E(I - B^{-1}A)$:*

$$\sum_i(\tilde{\lambda}_i - \lambda_i) = \mathrm{tr}(B^{-1}E(I - B^{-1}A)) + O(\varepsilon^2) \tag{4.50}$$

**Proof**: Corollary 4.14 and the invariance of the trace under coordinate transformation yield

$$\begin{aligned}\sum(\tilde{\lambda}_i - \lambda_i) &= \mathrm{tr}((Y^H(B+E)X)^{-1}Y^H(A+E)X - (Y^HBX)^{-1}Y^HAX) + O(\varepsilon^2) \\ &= \mathrm{tr}((X^{-1}(B+E)^{-1}(A+E)X - X^{-1}B^{-1}AX) + O(\varepsilon^2) \\ &= \mathrm{tr}((B+E)^{-1}(A+E) - B^{-1}A) + O(\varepsilon^2).\end{aligned}$$

If we linearize this formula using the first order Neumann expansion and sort out higher order terms we obtain:

$$\begin{aligned}\sum(\tilde{\lambda}_i - \lambda_i) &= \operatorname{tr}(B^{-1}(I - EB^{-1} + O(\varepsilon^2))(A + E) - B^{-1}A) + O(\varepsilon^2) \\ &= \operatorname{tr}(B^{-1}(A - EB^{-1}A + E + O(\varepsilon^2) - A) + O(\varepsilon^2) \\ &= \operatorname{tr}(B^{-1}E(I - B^{-1}A)) + O(\varepsilon^2);\end{aligned}$$

□

**Remark 4.2** *With this theorem we deduce information about the behavior of the eigenvalues* without *using information about the eigenvectors. Moreover, as we see in the proof the estimate is invariant under equivalence transformations applied to both* $(A, B)$ *and* $(E, E)$.
*The drawback is that if the terms* $\tilde{\lambda}_i - \lambda_i$ *of the sum cancel out, the sum might be small but the changes in the eigenvalues might be large anyhow.*

## 4.3 Computation of a first order sparsing criterion

For the linearly implicit Euler method the matrix pair under consideration is $(L, L - \tau Df)$. Recall that the matrix $B = L - \tau Df$ is invertible for reasonable step sizes $\tau$. To this matrix pair special perturbations $\Delta J_{ij}$ (zeroing out the matrix element $Df_{ij}$) are applied symmetrically (recall Section 3.3). So the perturbed matrix pair is $(L + \Delta J_{ij}, L - \tau Df + \Delta J_{ij})$.
We will now use the numerical methods and theoretical results of the last two sections to derive a sparsing criterion, that estimates the impact of zeroing out a matrix element on a predefined cluster of eigenvalues.
In the following we will consider the two cases that occur: sparsing in the case of an ordinary differential equation, where the Jacobian matrix is assumed to be of medium size such that dense methods can be applied, and the case of a large, sparse algebraic part. The first case appears also if we partition a DAE as described in Section 4.1.4.

### 4.3.1 Sparsing of the differential part

The computation of the criterion is split into two phases: first a block diagonalization as described in Section 4.1 is performed:

$$Y^H L X = A_B \qquad Y^H (L - \tau Df) X = B_B \tag{4.51}$$

and we obtain the matrices $Y^H, X, A_B$ and $B_B$. Recall that $A_B$ and $B_B$ are block diagonal matrices with upper triangular blocks $A_k$ and $B_k$. The resulting blocks can now be treated separately which is justified by Theorem 4.12.
To evaluate equation (4.50) for each block we will now compute the matrices on the left and the right of the perturbation matrix. Included are the transformation matrices that lead to the block diagonal form.

$$U^{(k)} = B_k^{-1} Y_k^H; \tag{4.52}$$

$$V^{(k)} = X_k (I - B_k^{-1} A_k); \tag{4.53}$$

As $B_k$ is upper triangular this can be done by solving upper triangular matrix equations. Up to now all computations can be performed without knowledge about the perturbation. This means that this $O(n^3)$ computation can be done once for all elements.

The second phase, the computation of a sparsing criterion for each element and each block is a simple matter now and also - as we will show - computationally cheap. To apply Theorem 4.16 we have to compute :

$$c_{ij}^{(k)} = |\mathrm{tr}(U^{(k)} \tau \Delta J_{ij} V^{(k)})| \tag{4.54}$$

for the $k^{th}$ cluster of, say, $m$ eigenvalues. Because $\Delta J_{ij}$ contains only one non-zero element the matrix whose trace is computed is a rank 1 matrix:

$$U^{(k)} \tau \Delta J_{ij} V^{(k)} = (U_{1i}^{(k)}, \ldots, U_{mi}^{(k)})^T (-\tau Df_{ij})(V_{j1}^{(k)}, \ldots, V_{jm}^{(k)}). \tag{4.55}$$

The trace of such a rank 1 matrix can be computed very cheaply in $O(m)$ operations:

$$|\mathrm{tr}(U^{(k)} \tau \Delta J_{ij} V^{(k)})| = |\tau Df_{ij} \cdot (U_{1i}^{(k)}, \ldots, U_{mi}^{(k)})(V_{j1}^{(k)}, \ldots, V_{jm}^{(k)})^T| = |\tau Df_{ij} \cdot \sum_{l=1}^{m} U_{li}^{(k)} V_{jl}^{(k)}|. \tag{4.56}$$

### 4.3.2 Sparsing of the algebraic part

After transforming (4.30) to a block diagonal form (4.31) we obtain the two matrix pairs $(\tilde{A}_1, \tilde{B}_2)$ and $(A_{22}, B_{22}) = (0, -\tau Df_{22})$. The first pair can be treated with the methods described above, the second pair will be considered here. To treat this case successfully, we have to consider some additional factors. As we have seen in Section 3.4 sparsing of the algebraic part $J_{22}$ of the Jacobian matrix has the consequence that we cannot achieve orders higher than one. So if sparsing shall be used in the context of an extrapolation method, it is necessary to keep this block exact.

The application of the linearly implicit Euler method is only possible, if $L - \tau J$ is non-singular for reasonable $\tau$, which is guaranteed for an exact Jacobian matrix and index 1 systems. If we use sparsing of the algebraic part, this property has to be preserved, and therefore $J_{22}$ still has to be non-singular after sparsing. So a sparsing criterion has to do both estimating the perturbations of the eigenvalues and preserving non-singularity.

Another difficulty is that in contrast to the case of an ordinary differential equation, where $I - \tau J$ is well conditioned for small $\tau$, the condition of $Df_{22}$ may be bad and cannot be influenced by a parameter such as $\tau$. But if $J_{22}$ is near a singular matrix, the first order theory breaks down, due to large higher order terms. This is obvious already in the scalar case: $(b-e)^{-1} - (b^{-1} + b^{-2}e) = e^2/((b-e)b^2)$. Obviously, the error between the exact and the approximate solution is only small if $e \ll b$. So for small $b$ the first order approximation is valid only in a very small neighborhood of $b$. In higher dimensions we will have the case that the badly conditioned mapping restricted to some subspaces is nearly singular, on others not. A robust sparsing criterion will have to take such phenomena into account.

Additionally, if we want to apply sparsing to the algebraic part, we have to consider the size of this block. Although the performance in the analysis phase is not crucial in real time simulation, it might be the case that the algebraic part is just too big for a kind of analysis similar to the one described above. A first try might be using (4.50), that reads now

$$\sum_k (\tilde{\lambda}_k^0 - \lambda_k^0) = \mathrm{tr}(B_{22}^{-1} E) + O(\varepsilon^2). \tag{4.57}$$

The corresponding sparsing criterion is then

$$c_{ij} = |\mathrm{tr}((\tau Df_{22})^{-1}(-\tau(\Delta J_{22})_{ji}))| = |(Df_{22}^{-1})_{ij}(Df_{22})_{ij}|. \tag{4.58}$$

This criterion can be evaluated by one (sparse) LU-Decomposition of $Df_{22}$ and $d-n$ forward-backward- substitutions to evaluate the columns $(Df_{22}^{-1})(:,i)$.
If the algebraic part is small enough to be treated with dense linear algebra methods in the analysis phase, we can take the effort to transform $(0, Df_{22})$ to an equivalent pair. This is facilitated by the fact that we can choose the transformation matrices freely, as $Y^H 0 X = 0$.
In view of the discussion above a lot of information can be obtained via the singular value decomposition. Then we have orthogonal transformation matrices $X$ and $Y$ such that $S = Y^H Df_{22} X$ is diagonal, and we can apply (4.50) to each singular value and obtain for the changes of eigenvalues

$$\tilde{\lambda}_k^0 - \lambda_k^0 = S_{kk}^{-1} E + O(\varepsilon^2), \tag{4.59}$$

which leads to a sparsing criterion

$$c_{ij}^{(k)} = |S_{kk}^{-1} (Y^H)_{ki} \Delta J_{ij} X_{jk}| \tag{4.60}$$

for each singular value. We see that a small singular value causes a large criterion, as required.

## 4.4 Discussion of the criterion

In Section 4.2 some issues concerning approximation errors and the sparsing criterion itself were mentioned briefly and shall be discussed more thoroughly in this section. Some conclusions will be drawn on how to use the degrees of freedom that are still left in the design of the algorithm.

### 4.4.1 Errors introduced by the block diagonalization and cancellation

As we have already mentioned, our criterion does not take into account higher order terms. Therefore, algorithmic decisions should be made in order to keep those terms small. This mainly affects the choice of the blocks.
Concerning the choice of the blocks - or equivalently the clustering of the eigenvalues - there is a tendency that close eigenvalues in different blocks lead to large higher order terms. So blocks should be chosen in such a way that eigenvalues that are grouped together stay together in one block. The existence of such clusters is a property of the model and models with distinct clustering are especially well suited for sparsing.
From a theoretical point of view the least we have to require is that $\mathrm{dif}[\cdot] \neq 0$. By Theorem 4.11 this is achieved, if the spectra of all blocks are mutually disjunct. So multiple eigenvalues have to be gathered in one block. To obtain small error bounds in Theorem 4.12 we also have to require that $\mathrm{dif}[\cdot]$ is large compared to the norm of the perturbations.
The other source of errors is cancellation. It is possible that

$$\left|\sum(\tilde{\lambda}_i - \lambda_i)\right| \ll \sum |(\tilde{\lambda}_i - \lambda_i)|.$$

This means that it may happen that an element with a strong impact on the eigenvalues receives a small criterion and is classified in the wrong way. This also suggests to use rather small blocks when performing the block diagonalization so that this effect happens less probable.
However, at least if we consider the stiff eigenvalues, there is an heuristic argument that suggests that the magnitude of the sum is a good estimate for the sum of magnitudes. Sparsing was described as a way of blending implicit and explicit methods such that the result is a numerically stable algorithm. So, the more elements of the Jacobian are deleted, the more will the method

behave like an explicit one. Hence, the stiff eigenvalues will move to the left of the complex half-plane. Therefore, they will all move into the same direction, and so the effects of cancellation might not be so grave. This effect was also observed in numerical experiments. Concerning oscillatory modes, this tendency to move into one direction is not that strong and it might well happen that cancellation plays a role in the choice of an element.
These observations give a hint how to choose the clusters of eigenvalues. For the stiff eigenvalues we can use few clusters but the oscillatory eigenvalues should be divided into as many clusters as possible to avoid cancellation.
One important aspect of cancellation is that it smooths out the large first order estimates of eigenvalues that are very close together, because the matrix is a numerically perturbed Jordan matrix. In this case, first order estimates are very large but meaningless, because the higher order terms are large too. Clearly a sufficiently small perturbation will not cause multiple eigenvalues to jump to infinity as suggested by the first order criterion. To sum up, cancellation makes the criterion more robust dealing with multiple eigenvalues. In fact the presence of multiple eigenvalues was the reason to retreat to block diagonalization.
Obviously, we have to perform a trade-off between these two sources of errors. Experience has shown that it is a good choice to use small blocks such that only multiple eigenvalues (or very close eigenvalues) are inside one block. In this case cancellation plays a minor role, and only smooths out the first order terms of Jordan blocks. But for eigenvalues that are not too close the risk of cancellation weighs higher than possible second order errors.

### 4.4.2 Invariance against scaling

It is a well known fact that the qualitative behavior of a solution of a differential equation $\dot{x} = f(x)$ near a fixed point $x^* : \ f(x^*) = 0$ has the following invariance property. The transformation

$$x \longrightarrow Tx \Longrightarrow \dot{x} \longrightarrow T\dot{x} \Longrightarrow Df(x^*) \longrightarrow TDf(x^*)T^{-1} \tag{4.61}$$

does not change the qualitative behavior of the solution near the fixed point if $T$ is non-singular. This gives rise to the analysis of the qualitative behavior of the solution by eigenvalue analysis of the Jacobian. The same "invariance against similarity transformations" is valid for difference equations, which gives rise to the linear stability theory of numerical methods for differential equations. An introduction to invariance principles in a more general framework is given in [8]. Any analysis routine that aims on this kind of qualitative behavior (like our sparsing method) should also show such an invariance property.
Analogously, the qualitative behavior of differential algebraic equation systems $B\dot{x} = f(x)$ at a fixed point is invariant against equivalence transformations. The transformation

$$x \longrightarrow Tx, f(x) \longrightarrow Sf(x) \Longrightarrow (B, Df(x^*)) \longrightarrow (SBT, SDf(x^*)T) \tag{4.62}$$

does not change the qualitative behavior of the solution if $S$ and $T$ are non-singular.
However, as our sparsing criterion considers element-wise perturbations and their similarity transforms are not elements anymore. Hence, we restrict our considerations to scaling transformations, i.e., $S, T$ are diagonal matrices. Invariance against scaling is clearly obtained considering Remark 4.1 and Remark 4.2, because the elements of the matrix $J$ and therefore the perturbations are also subject to the scaling transformations.

### 4.4.3 Sparsing by magnitude

In this section we are going to review the technique of dynamic sparsing by magnitude as described in [33]. It has been applied successfully to large systems of stiff ordinary differential equations. The proposed criterion reads:

$$|\tilde{Df}_{ij}| < \sigma/\tau \rightarrow J_{ij} := 0 \tag{4.63}$$

Here $\tilde{Df}$ is the scaled Jacobian matrix, $\sigma < 1$ is a safety factor, and $\tau$ is the step size of one step of the extrapolated linearly implicit Euler method. If we consider (4.50) for the case of an ODE and if we do not perform a spectral resolution, we obtain:

$$\sum_i (\tilde{\lambda}_i - \lambda_i) = \mathrm{tr}((I - \tau Df)^{-1} \tau \Delta J_{ij} (I - (I - \tau Df)^{-1})) + O(\varepsilon^2). \tag{4.64}$$

If the problem is assumed to be stiff, then we cannot simplify this formula anymore, and we can especially see that the change in the eigenvalues is not related directly to the size of the element $Df_{ij}$.
If we assume the non-stiff case: $\tau Df \ll I$, then we can expand $(I - \tau Df)^{-1}$ with Neumann series, drop higher order terms, and obtain:

$$\begin{aligned} \sum_i (\tilde{\lambda}_i - \lambda_i) &= \mathrm{tr}((I + \tau Df) \tau \Delta J_{ij} (I - (I + \tau Df))) + O(\varepsilon^2) \\ &= \mathrm{tr}(\tau \Delta J_{ij} \tau Df) + O(\varepsilon^2) = -\tau^2 \Delta J_{ij}^2 + O(\varepsilon^2) = O(\varepsilon^2). \end{aligned}$$

In this case, the sparsing criterion results in values that have about the size of the neglected terms. This result was to be expected, because the linearly implicit Euler method is a W-method. If we had a first order change of the eigenvalues, the order of the method would decrease for an inexact Jacobian. If the matrix has some block structure such that a stiff and a non-stiff part are present and are only weakly coupled, then the size of the elements plays a role in the following sense: if $\Delta J_{ij}$ is large, then it can be associated with the stiff part, but in general not the other way round.
Clearly, in both cases (4.63) is not related to a first order criterion. We can, however, regard (4.63) as a worst case estimate for (4.64). With the relation

$$\mathrm{tr}(A) \leq n \|A\| \tag{4.65}$$

(the factor $n$ reflects that we estimate the *sum* of the changes of the eigenvalues) we can conclude that

$$\begin{aligned} \sum_i (\tilde{\lambda}_i - \lambda_i) &\doteq \mathrm{tr}((I - \tau Df)^{-1} \tau \Delta J_{ij} (I - (I - \tau Df)^{-1})) \\ &\leq n \|(I - \tau Df)^{-1}\| \tau |\Delta J_{ij}| \, \|(I - (I - \tau Df)^{-1}))\| \\ &= C \tau |\Delta J_{ij}|. \end{aligned}$$

We see that we can interpret $1/C$ as a conservative estimate for $\sigma$. However, sparsing will only be successful, if we can choose $\sigma \ll C$, e.g., if the problem has some special structure, or if there are many very small elements present. We also see that the weak point of sparsing by magnitude is, that this technique neglects the interdependence between the elements and therefore estimates the importance of each element only insufficiently, which is especially a drawback if differential algebraic equations are considered. Of course, if dynamic sparsing is the goal, the evaluation of (4.50) is too expensive, but it might be possible to design a dynamic sparsing criterion, that at least reflects an interdependence approximately.

# Chapter 5

# Implementation of a real-time simulator

The goal of the next two chapters is to study the practical applicability of sparsing in real-time simulation. This includes several issues. On the one hand we can study the effects and benefits of sparsing on the structure and the eigenvalue distribution of a given test matrix pair. On the other hand, we have to explore, how the reduced structure can actually be exploited by a real-time simulation method.

## 5.1 Design overview

We have seen in the preceding sections, especially in Section 2.3.1, that a simple design with little overhead is important for a real-time integrator, and that as much work as possible should be performed before the real-time simulation starts. Here, we will give a short outline of the design of the simulator. In the subsequent sections, more details are presented.
We will base our integrator on the linearly implicit Euler method without extrapolation. As already noted, this method is suitable for real-time simulation with small communication intervals.

**Preprocessing.** The main goal of the preprocessing routine is to analyze the model to be simulated and especially the systems of equations to be solved during the course of integration. The efficiency of the real-time simulator depends largely on the preprocessing. Hence, emphasis is placed on the quality of the results, rather than the speed of the routines. Therefore, this part of the integrator will mostly be implemented in MATLAB, what allows us to use high quality numerical subroutines and simple notation.
The preprocessing has the following tasks:

- Find a suitable sparsity pattern such that stable simulation is still possible, and the remaining linear equation systems can be solved efficiently.
- Analyze the remaining sparse system of equations such that the sparsity structure of this system can be exploited.
- Allocate the memory needed during the course of integration and other preparations.

**Simulation.** In this part, the integrator efficiency is crucial. The routines must be simple and contain as little overhead as possible. All routines have to be real-time capable and should be portable to special real-time hardware. Therefore, the programming language of choice is C. During the simulation, the following tasks have to be performed:

- efficient evaluation of the right hand side $f$ and the sparse Jacobian $Df$,
- solution of the systems of equations,
- computation of the solution for the next time instant,
- estimation of the error.

The following operations cannot be performed in real-time simulations:

- allocation or deallocation of memory and similar system calls, as they take an unpredictable amount of time,
- changes in the step size or similar forms of adaptivity,
- pivoting for stability in sparse matrix factorizations, because this may change the fill-in and therefore the performance.

## 5.2 Implementation of sparsing

Equipped with a first order estimate for eigenvalue perturbations we turn to the implementation of sparsing. For this purpose we use a pseudo-MATLAB notation. Furthermore we want to point out only the principal features of the algorithm, so many details of the actual implementation are omitted.

**Algorithm 5.1** *Algorithm in pseudo MATLAB notation to perform sparsing.*

```
function [S]=Sparsing(DAE,x0,t,tau,rho,Smin)
    [Sf,L]=Get_sparsity_structure(DAE);
    X=Sample_consistent_states(DAE,Sf,SampleTimes);
    S=0;
    for(i=1:nSamples)
        Jac=Get_jacobian(DAE,X(:,i),SampleTimes(i));
        if(~Satisfactory_Approximation(Jac,S,rho)){
            Sold=S;
            C=Calculate_sparsing_criterion(Jac,L,Sold,tau);
            S=Choose_structure(Jac,L,C,rho,Sold);
        }
    end
    Prepare_for_simulation(S);
```

Algorithm 5.1 is a sketch of the sparsing routine as implemented in MATLAB. It requires a description `DAE` of the DAE, consistent initial values `x0`, the time span `t` (as a vector) the step size `tau`, and parameters `rho` (as a vector) to control the tradeoff between sparsity and the quality of the Jacobian approximation (see below). The user can provide a matrix `Smin` to force

the algorithm to preserve those non-zero elements of Df that are also non-zero in `Smin`. The result is an incidence matrix S, that describes an acceptable sparsity structure.

### 5.2.1 Initialization

Before the start of the sparsing routine we have to collect some information about the DAE. This includes the full sparsity structure of the Jacobian and the left hand side matrix `L`, that is obtained in `Get_sparsity_structure`. In an object oriented framework this may be performed by symbolic analysis. Otherwise we may compute the Jacobian and scan the non-zero structure. This has to be performed several times with several arguments to minimize the risk of an element being zero accidentally.
The subroutine `Sample_consistent_states` provides several consistent states, e.g., by one or several off-line simulations, where the Jacobian is evaluated. The Jacobian matrices at those states are then subject to the sparsing routine. This is done in the `for` loop.

### 5.2.2 Testing of the sparsed Jacobian matrix

For each sample vector of states, the Jacobian is evaluated. Then it is tested, if the eigenvalues $\tilde{\lambda}_i$ of the difference equation with sparse Jacobian are close enough to the original eigenvalues $\lambda_i$. This is determined in the routine `Satisfactory_Approximation`. The criterion for this is the following: form a set of pairs from two sets of eigenvalues such that $\max|\lambda_i - \tilde{\lambda}_{\tilde{i}}|$ is minimal. Then with `rho`$= (\rho, \rho_m)$ check the relation:

$$|\lambda_i - \tilde{\lambda}_{\tilde{i}}| \leq r_i := \max(\rho(1 - |\lambda_i|), \rho_m). \tag{5.1}$$

If this inequality is not fulfilled for at least one $i$ we have to reconsider the sparsity structure of $J$ and allow for some further non-zero elements. This criterion has the effect that eigenvalues near the stability border are judged more restrictively than eigenvalues near the origin. This is necessary, because the errors of the latter are damped quickly, whereas errors in the larger eigenvalues can severely affect the long time behavior of the solution. However, there are always some "non-stiff" eigenvalues at or very close to 1. To treat them properly we introduce a minimal radius $\rho_m$ to avoid that very small changes in these eigenvalues force the sparsing algorithm to produce a structure with too many elements.

### 5.2.3 Sparsing

If necessary we recompute the sparsity structure of the Jacobian. We only have the possibility to add non-zero elements, because the previous sparsity structure is necessary for the stability of the differential equation in the previous steps. Therefore, we only have to consider the elements that are not already part of the sparsity structure and calculate their criterion.
The sparsing procedure consists of two parts. First the sparsing criteria `C` are computed in the routine `C=Calculate_sparsing_criterion(Jac,L,Sold,tau)`. Then the elements to be sparsed are selected by the routine `S=Choose_structure(Jac,C,rho,Sold)`.
The first routine is sketched in Algorithm 5.2. Its details are derived and described in Chapter 4. Its core is the subroutine `Evaluate_Criterion(S,T,Q,Z,k,i,j)` that evaluates equation (4.50) for the block $k$ and the element $J_{ij}$ of the Jacobian. We do not evaluate the criterion for elements that were already included in the sparsity structure.

The second routine chooses the sparsity structure according to Algorithm 5.3. Recall that for

**Algorithm 5.2** *Algorithm in pseudo MATLAB notation to compute the sparsing criterion.*

```
function C=Calculate_sparsing_criterion(Jac,L,Sold,tau)
    [S,T,Q,Z,blocks]=Perform_block_diagonalization(Jac,L,tau);
    for(i=1:size(blocks))
        [I,J]=find(Jac & ~Sold);
        for(j=1:nnz(Jac & ~Sold))
            C(i,j)=Evaluate_Criterion(S,T,Q,Z,blocks[i],Jac(I(j),J(j)));
        end
    end
```

**Algorithm 5.3** *Algorithm in pseudo MATLAB notation to compute the sparsing criterion.*

```
function S=Choose_structure(Jac,L,C,rho,Sold)
     w=Find_initial_weight(rho,Jac);
     S=spones(Jac & ~Sold);
     while(1)
         c=C*w;
         [eigenvalues, errors]=Compare_Eigs(Jac.*(S | Sold),Jac);
         S_test=spones(c<1);
         if(BreakCriterion(S | Sold, w)
            break;
         end
         if(Satisfactory_Approximation(eigenvalues, errors, rho) AND
             nnz(S_test | S_old) < nnz(S | Sold)
         S=S_test;
     end
     w=Choose_new_weights(w,errors);
 end
```

each non-zero element, `C` provides estimates for the sum of the eigenvalue changes in each block. To obtain a single criterion for each element, we compute a weighted sum of these. Algorithm 5.3 chooses and modifies the weights iteratively. In this way, a sparsity structure is determined such that the eigenvalues of the perturbed difference equation satisfy equation (5.1).
This basic algorithm can be implemented in several ways. It can be regarded as an optimization problem: "Minimize the weights $w$=`w` under the condition that the errors $\varepsilon$=`errors` satisfy equation (5.1)". We assume that the components $\varepsilon_i(w)$ are step functions and that $\varepsilon_i(w_j)$ is increasing monotonically.
Due to this non-smooth structure, we use a simple bisection strategy to choose the new weights. For each block we store the maximal weight $w_+$ for which (5.1) is fulfilled and the minimal weight $w_-$ for which (5.1) is violated. If there is no $w_+$ yet, we set $w = 2w_-$. If there is no $w_-$, we set $w = 0.5w_+$. Otherwise, we set $w = 0.5(w_+ + w_-)$. Then we test $w$ for the adherence of (5.1) and set $w_+ = w$ if $w$ fulfills (5.1) and $w_- = w$ otherwise. All computations are performed with a test matrix `S_test`. The algorithm stores the matrix with the lowest number of non-zero elements that fulfills (5.1) in `S`.

We choose the starting values according to equation (5.1) as

$$w_i = \frac{1}{r_i},$$

and we stop the search, if the sparsity structure of $S$ has not improved for several times.

## 5.3 Evaluation of the Jacobian matrix

To evaluate the sparse Jacobian matrix of the right hand side we can use the well known technique of column grouping (see [5]), as it is, e.g., used in the MATLAB routine `numjac`.
We consider the non-zero structure of the columns of the Jacobian and build subsets of columns such that their non-zeros do not overlap, i.e., each row index for a non-zero element appears only once in each subset. There are several good heuristic algorithms available for the NP-hard task of finding an optimal grouping. We use the MATLAB routine `colgroup` for this task and pass the result vector to the real-time simulator.
Assuming that we obtained $k$ subsets of columns, we can now evaluate the sparse Jacobian using only $k$ additional function evaluations. This is because we can now compute the sum of all columns in one of the subsets by one additional function evaluation. As each element of this vector can be associated uniquely with an entry in the Jacobian matrix, we obtain all columns of the Jacobian contained in this group. The obtained entries can now be stored in a sparse matrix structure, possibly performing some column pivoting.
In the context of sparsing we remark that the column grouping cannot be applied directly to the sparsed Jacobian structure. Otherwise non-zero elements, that are sparsed out, may add to other non-zero elements, that are important for stability. Of course, the computation of these elements would then be wrong. However, to save function evaluations, we can use the following method during the preprocessing phase. For each row, compare each element of the unsparsed Jacobian with the smallest non-zero element of the sparsed Jacobian in this row. If the element is some orders of magnitude smaller, it can be dropped. This can be especially useful, if there are small errors perturbing the right hand side, e.g., because during its evaluation there are equation systems to be solved.
In an integrated modelling and simulation environment, such as Dymola, more information about the right hand side is known to the preprocessing routine. For example, linear equations can be recognized and therefore it is possible to spot constant elements of the Jacobian. These elements do not have to be evaluated at each step, but only at the start of the simulation. As the equations in object oriented modelling usually do not contain very complex expressions, this concept can be extended such that the non-zero elements of the Jacobian matrix are evaluated symbolically. Then only those elements that are left after the sparing routine have to be computed.

## 5.4 Sparse factorizations in real-time

Much effort has been taken to achieve that the Jacobian matrix to be factorized during the integration is as sparse as possible. But how can the sparsity be exploited in the real-time case? Typical factorization routines perform sparse Gaussian elimination to obtain an LU-factorization of the matrix. Their core feature is an intelligent pivoting strategy to reduce the fill-in, i.e., the number of additional non-zero elements generated during the course of the factorization. Unfortunately, this fill-in cannot be predicted by structural considerations alone, because Gaussian elimination needs pivoting for numerical stability. As a consequence, time and storage requirements cannot be predicted exactly, too, although there are algorithms to compute upper bounds

(see [21]). Sparse LU-solvers are therefore equipped with sophisticated pivoting strategies, and with dynamic memory management. Both features are drawbacks in real-time simulation, as the pivoting logic amounts in a large overhead and does not improve the worst case estimate. Moreover, memory allocations and deallocations take an unpredictable amount of time. Subroutines for real-time applications need a simpler structure.

### 5.4.1 Methods for sparse QR decompositions.

The situation is different with orthogonal factorizations. Due to the inherent stability of orthogonal transformations, pivoting for stability is not necessary. Therefore, it is possible to construct a sequence of elimination steps utilizing only the structure. Certainly, the number of required floating point operations will usually increase, compared to Gaussian elimination. But we can implement a factorization routine for a constant sparsity structure that takes constant effort, uses static memory management, and has low overhead, because the pivoting strategy can be fixed before the integration starts. In the rest of this section we will summarize briefly the facts presented in the monograph [2] that are relevant for the design of our factorization routine.
There are three main approaches mentioned in [2] for sparse QR decompositions. The first approach is via solving the normal equation:

$$A^T Ax =: Bx = c := A^T b. \tag{5.2}$$

Here we have the well known problem that the condition number of $B = A^T A$ is the square of the condition number of $A$.
The second approach uses Givens rotations and pivoting strategies to reduce the fill-in. This approach has been taken, e.g., in [20] performing row by row elimination. Alternatively, elimination can be performed column by column, as described in [37].
The third approach is the multifrontal QR decomposition a more complex, but in general more efficient approach.

### 5.4.2 Predicting the structure of R in the QR Decomposition.

For real time simulations it is essential to use static storage. Therefore, we have to allocate all the required memory before the real time simulation starts. For this purpose we need the structure of the upper triangular factor $R$ of the QR decomposition of $A = QR$. As described in [2] this can be performed by "symbolic factorization" of $A^T A$ or by symbolic Givens rotations applied to $A$. Application of a symbolic method means here that the fill-in of the numerical method is computed, assuming that no numerical cancellation occurs. This can be performed in cases where pivoting for stability is not necessary. Due to [20], we have the following theorem, cited from [2].

**Theorem 5.1** *The structure of $R$ as predicted by symbolic factorization of $A^T A$ includes the structure of $R$ as predicted by the symbolic Givens method, which includes the structure of $R$.*

**Proof**: For details see [20]. The proof is based on a connection between the Cholesky factorization of $\tilde{R}^T \tilde{R} = B = A^T A$ and the QR decomposition of $A = QR$.

$$\tilde{R}^T \tilde{R} = B = A^T A = A^T Q Q^T A = R^T R \tag{5.3}$$

Due to the uniqueness of the Cholesky factor $\tilde{R}$ of $B$, the structures of $R$ and $\tilde{R}$ are the same. However, the symbolic factorization routine applied to $A^TA$ may produce more non-zero elements than $\tilde{R}$ actually has, independent of the numerical values of the entries of $A$. The reason for this is that there are correlations between the numerical values of $C = A^TA$. This information is lost for the symbolic factorization routine, because it is designed to handle matrices $C$ with arbitrary and independent entries. □

According to [4] we can improve the result, if we assume that $A$ has the "strong Hall property":

**Definition 5.2** *A matrix of order n is said to have the strong Hall property if for every subset of k columns, $0 < k < n$, the corresponding submatrix has non-zeros in at least $k + 1$ rows.*

**Theorem 5.3** *Let A have the strong Hall property. Then the structure of $A^TA$ will correctly predict that of R, excluding numerical cancellations.*

**Proof**: See [4]. □

It is an interesting and useful fact that the diagonal blocks of the block lower triangular form mentioned in Section 2.2 have the strong Hall property (see [2], originally [4]), provided the matrix $A$ is square and structurally non-singular.

### 5.4.3 Row oriented QR decomposition.

As stated above, a row sequential algorithm for a sparse QR decomposition was developed by George and Heath in [20]. We will summarize the most important features (see also [20], [2]) with special emphasis on real-time capability.
When solving the system $Ax = b$ for sparse matrices, we can improve efficiency by pivoting of columns and rows. This means that we can replace $A$ by $P_rAP_c$, where $P_r$ describes the row permutations and $P_c$ the column permutations. Then we solve the system $P_rAP_cP_c^Tx = P_rb$. We can now factorize $P_rAP_c = QR$, while we may choose $P_r$ and $P_c$ properly. Concerning the minimization of fill-in, we note that $(P_rAP_c)^TP_rAP_c = P_c^TA^TAP_c$. Hence, the structure of $R$ is not dependent on row permutations. The column permutations can be obtained with the help of one of the well known column orderings, such as the minimum degree ordering, or the approximate minimum degree ordering. Experience shows that the approximate minimum degree ordering usually yields the best results, so it is used in our algorithm. By symbolic factorization of $P_c^TA^TAP_c$ we obtain the structure of $R$ to be able to allocate the needed amount of memory. Then we can choose $P_r$ (see next paragraph) and perform the following algorithm, that we cite from [2].

**Algorithm 5.4 (Row Sequential QR Algorithm.)** *Assume that $R_0$ is initialized to have the structure of the final R and has all elements equal to zero. The rows $a_k^T$ of A are processed sequentially, $k = 1, 2, \ldots, m$, and we denote by $R_{k-1} \in \mathbb{R}^{n\times n}$ the upper triangular matrix obtained after processing rows $a_1^T, \ldots, a_{k-1}^T$. The $k^{th}$ row $a_k^T = (a_{k1}, a_{k2}, \ldots, a_{kn})$ is processed as follows: we uncompress this row into a full vector v and scan the non-zero elements from left to right. For each $v_j \neq 0$ a Givens rotation involving row j in $R_{k-1}$ is used to annihilate $v_j$. This may create new non-zeros both in $R_{k-1}$ and in v. We continue until the whole row v has been annihilated. If $r_{jj} = 0$, this means that this row of $R_{k-1}$ has not yet been touched by any rotation and hence the entire $j^{th}$ row must be zero. When this occurs, the remaining part of v is inserted as the $j^{th}$ row in R.*

Although the vector $v$ is stored as a full vector, we can make use of the sparsity of $v$. This is easily possible if we use the non-zero structure of $R$ to find potential non-zeros in $v$.

### 5.4.4 Row ordering

We still have the freedom to choose $P_r$. This will not affect the fill-in finally obtained, but it may affect the time, this fill-in occurs and therefore the number of arithmetic operations necessary for the decomposition.
For row ordering we use the algorithm described in [2] (Algorithm 6.6.4). Let the first non-zero index of row $i$ be denoted by $f_i$ and the last non-zero index be denoted by $l_i$. Sort the rows according to increasing $f_i$. All groups of rows with equal $f_i$ are sorted after increasing $l_i$.

### 5.4.5 Implementation details

As standard software for the sparse QR decomposition is not primarily designed for real-time simulations, we implement a QR decomposition routine for ourselves. To minimize the overhead and the programming effort, the design is chosen in a hybrid way. As much as possible is implemented in MATLAB. This is the symbolic analysis of the sparse matrix and the choice of the permutation matrices $P_r$ and $P_c$. Both can be performed once, before real-time simulation starts. The rest of the routine is written in C to ensure efficiency and real-time capabilities.
In MATLAB we first compute permutations such that the matrix is in block upper triangular form using the MATLAB standard routine `dmperm`. The diagonal blocks have the strong Hall property and are now treated separately by column and row permutations. They are applied to each block as described above to obtain sparse $R$ factors. The vectors of permutations, the structures of $A$ and $R$ and the block sizes are passed to the factorization routine.
In each time step, the factorization routine obtains the numerical entries of the sparsed Jacobian matrix $A = L - \tau J_n$ and the right hand side $b = f(x_n)$. After permutation, the solution of the system $Ax = b$ is obtained by solving a sequence of smaller systems corresponding to the diagonal blocks of $A$ with the help of the row sequential QR-decomposition. For this we use recursively that the blocked system

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{5.4}$$

can be solved as follows:

$$A_{22} = Q_2 R_2 \longrightarrow R_2 x_2 = Q_2 b_2 \longrightarrow A_{11} = Q_1 R_1 \longrightarrow R_1 x_1 = Q_1 (b_1 - A_{12} x_2). \tag{5.5}$$

So the routine performs a sequence of QR-decompositions, back-substitutions and block matrix multiplications.
Consequently, we do not really obtain an orthogonal factorization of $A$. We will later talk about "the factor $R$ of the QR-factorization of $A$", but this is rather the sequence of right upper triangular matrices obtained by our block algorithm, displayed as a block diagonal matrix.

## 5.5 Damping of algebraic variables

Newton's method is the standard way of solving non-linear algebraic equations $f(x) = 0$. However, convergence is only guaranteed in a neighborhood of the solution, whose size depends on a Lipschitz constant for the Jacobian. To obtain convergence for starting values outside this

neighborhood, various globalization techniques have been developed (see, e.g., [6], [8]).
One important technique is damping: the Newton correction $\Delta x$ is multiplied by a factor $\lambda \leq 1$ to prevent the method from diverging.
A simple, but effective strategy for choosing a damping factor is the Amiijo strategy.

Select an optimal $\lambda$ from a set $\{1, 1/2, 1/4, 1/8, \ldots, \lambda_{min}\}$ such that the monotonicity test $\|f(x + \lambda\Delta x)\| \leq (1 - \lambda/2)\,\|f(x)\|$ is fulfilled.

Implicit and linearly implicit methods for differential algebraic systems may suffer from similar difficulties. Especially, if the algebraic equations are highly non-linear, the intrinsic Newton iteration may have difficulties to converge. The standard cure in off-line simulation is step size reduction. The consequence is that the starting values are always close enough to the solution to allow convergence. In real-time simulation we do not have this possibility and so we have to resort to other techniques, such as introducing a damping factor $\lambda$.
The situation of highly non-linear equations occurs especially in the simulation of hydraulic systems (see, e.g., [1]). Here the flow $q$ through an orifice is proportional to the square root of the pressure difference $\Delta p$ between both sides of the orifice: $q = k \cdot \sqrt{\Delta p}$.
For small $\Delta p$ this leads to a high Lipschitz constant of the right hand side and the Jacobian matrix. The consequence is that Newton's method becomes unstable for small $\Delta p$.
In the following, we describe a simple damping technique and our experience we made applying this technique to the simulation of a hydraulic system with the linearly implicit Euler method. Simulation with constant step-size and without damping leads to large oscillations in the solution trajectory that originate from the square root terms in the algebraic equations.
Of course, we have to adapt the damping strategy to the special structure of the DAE. We know that for ordinary differential equations damping is not necessary. On the other hand, Newton's method for non-linear algebraic equations relies heavily on damping. If we combine algebraic and differential equations to a DAE and assume that algebraic and differential variables are only weakly coupled, then we may apply damping only to the algebraic part. For the computation of the residual, we also consider only the algebraic part. Then we can interpret our damping strategy as a damping strategy for the solution of a slowly changing algebraic equation system. To construct an efficient damping strategy for real-time applications, we have to use a constant number of function evaluations, and so we cannot compute an optimal damping parameter $\lambda$ in each step. However, if we assume that the behavior of the solution does not change much from one step to the other, we may use the damping factor from the last step as a starting point for the selection of the next damping factor. The damping factor must have the opportunity to increase or to decrease and so we arrive at the strategy:

Choose $\lambda_{k+1}$ from the set $\{\min\{1, 2\lambda_k\}, \lambda_k, \max\{\lambda_k/2, \lambda_{min}\}\}$, such that $\|f(x_k + \lambda_{k+1}\Delta x_k)\|$ is minimal.

We cannot simply adapt the monotonicity test to DAEs, as the equations to be solved change from step to step. This strategy needs two additional function evaluations compared to the linearly implicit Euler method without damping.
A damping strategy along these lines was implemented in MATLAB, and applied to a model of a hydraulic gear box. The strategy was able to cope with the numerical oscillations and a smooth trajectory was computed. This is an encouraging result, considering the simple algorithm that was used. Especially, for integrated simulation environments with knowledge about the structure of the equations, refined damping strategies may be the solution to some difficult problems that

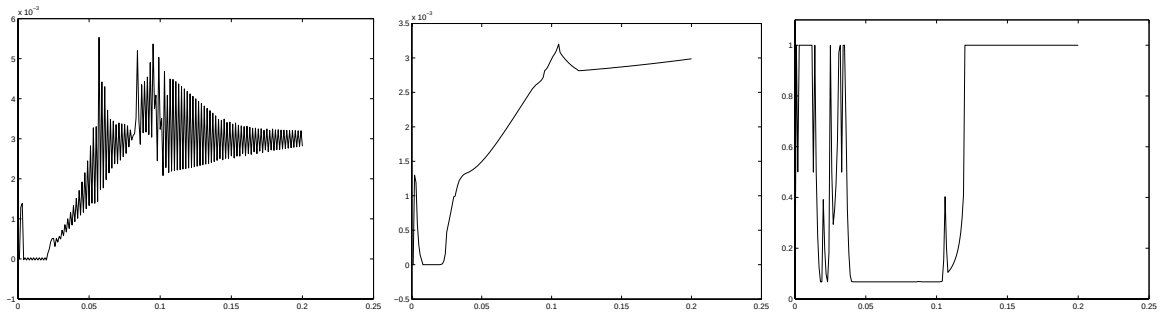

Figure 5.1: Numerical solution trajectory without damping, with damping and the damping factor $\lambda$.

occur especially in hydraulic simulations.

## 5.6 Error estimation

In real time simulations, accuracy is only a minor consideration, because in most cases it is sufficient to reproduce the qualitative behavior of the model. Furthermore, the error propagation is well behaved in most cases of industrial simulations, because the model is stable or stabilized by a controller. This leads to the choice of low order methods, such as the linearly implicit Euler method.
However, to be able to judge the quality of the solution and to detect potential instabilities in the solution, it is necessary to implement a local error estimator, at least in a simple form. In the context of higher order methods this is performed by comparing the integration method with a method of lower order, which leads to embedded methods, such as Runge-Kutta-Fehlberg methods (see, e.g., [9], [24]). If we apply this idea to a first order method we have to compare it to a zero order method, which is:

$$x_{n+1} = x_n.$$

We therefore can estimate the error by considering the absolute change in the variables

$$err_{est} = \|x_{n+1} - x_n\|. \tag{5.6}$$

To obtain a relative error concept, we divide each component $err_{est}^{(i)}$ by $x_n^i$ if $x_n^i$ is not zero. Then we compare the estimated error with a user prescribed relative tolerance and with a user prescribed absolute tolerance. If both tests fail, then a warning will be issued, or optionally the simulation will be stopped. The absolute tolerance can be scaled by the user.

# Chapter 6

# Numerical experiments

To test sparsing on a practical problem we consider the detailed model of an industrial robot with six degrees of freedom, that we discussed in Section 2.2.4. In this test example many aspects of simulation with sparsing can be observed. Especially, the robot shows the structure discussed in Section 2.3.4. The dynamics of the components are on different time scales, and so we obtain a separably stiff system. Our model is available as a C-code in three versions.

- As a differential algebraic system with 354 variables (without "tearing").
- As a differential algebraic system in 117 variables. (with "tearing", see Section 2.2.2).
- As an ordinary differential equation in 78 variables. The transformation from a DAE to an ODE is performed numerically.

The three versions of the model will help us to draw conclusions about the performance of sparsing in the context of stiff ODEs, of DAEs with a small algebraic part and of DAEs with a large algebraic part. Our aim is to simulate the movement of the robot for one second at a step size of 1 ms. At this step size, the simulation of the ODE model by an explicit method fails due to stiff components in the model.
We use a personal computer with an AMD Athlon processor, that runs at 700 MHz. The code is generated by the Visual C++ 6.0 compiler for a Windows NT environment with optimization for speed. Time measurements were performed with the help of the C-routine `clock()`, that measures time with an accuracy of 1 ms. To obtain more reliable and accurate timing results, we measured the overall execution time over a large number of steps. This makes it possible to measure the very short times that occur here and average out delays caused by the operating system. This technique is feasible, because in our algorithm every step takes essentially the same time.

## 6.1 Preliminary considerations

With our numerical experiments we address two main issues. How is the quality of the simulation affected by sparsing? How much performance can be gained by sparsing? Both questions can only be answered properly if we consider the setting.
As described in Section 2.2.3 Dymola provides a feature called inline integration, that is equivalent to a direct approach for solving differential algebraic systems, namely the implicit Euler method. The symbolic preprocessing routines generate a sequence of equation systems to be

solved. Due to the special problem structure many of these equation systems are small, and a lot of them is even one dimensional and linear and can be solved symbolically. Calling the resulting C-routine is performing one time step, rather than evaluating a right hand side. Time measurements show that in this case most of the computing time during one call is spent solving the larger systems of equations.
As an example, we consider the simulation of the robot model. Simulation of the robot for one second with inline integration at a step size of 1 ms takes 2.7 seconds on our personal computer. 2.5 seconds are spent solving the largest equation system, which is of order 39. We see that linear algebra constitutes the dominant part of the computational effort if Dymola and inline integration are used. The rest of the computational time is spent to evaluate the residual and the approximately 3700 algebraic variables. Although the time to perform this is comparatively small, namely about 0.2 ms per function call, it is crucial for performance that this time is not spent too often during one integration step. It is a big advantage of inline integration that this only has to be performed once in one time step, as the Jacobian matrices for the systems of equations can be evaluated locally. This means that in our case of a system of order 39, only 39 and not 3700 variables have to be evaluated to compute one column of the Jacobian. Unfortunately, this optimization cannot be accessed by external programs. This means that at each function evaluation 3700 variables are computed, even if only a small part of them is used. Therefore the cost for the evaluation of the Jacobian is very expensive.
The unpleasant consequence for us is that we cannot obtain meaningful time measurements about the overall performance, as the interface to the model includes only the possibility to evaluate the whole right hand side. In this case, the evaluation of the Jacobian is much more expensive than the linear algebra. To obtain meaningful results, one would have to combine sparsing, the sparse solver and inline integration into one method. However, this can only be performed by the developers of Dymola.
The best thing we can do is measuring the time needed for the solution of the remaining systems of equations and adding an estimated effort for the evaluation of the right hand side and the Jacobian.

## 6.2 Differential algebraic systems with a large algebraic part

If we modify the code generated by Dymola such that the transformation to ODE form is not performed and turn off tearing, we obtain a DAE of size 354 with 78 differential and 276 algebraic equations. The matrix $L - \tau Df$ contains 1306 non-zero elements before sparsing.
To study the effects of sparsing we perform sparsing with different parameters.

$$\rho \in \{0.01, 0.05, 0.1, 0.5, 1.0\} \qquad \rho_{min} = 0.01 \cdot \rho.$$

Table 6.1 shows the performance gains, of sparsing applied to this system. We observe that up to one third of the elements can be set to zero if $\rho$ is large. As a consequence the number of non-zero elements of $R$ (to its definition see Section 5.4.5) is halved and its largest block is 20 percent smaller. Observe the changes in the non-zero structure in Figures 6.3, 6.4, and 6.5. Due to the reduced structure, the factorization times are halved. The details are listed in Table 6.1. Here $\text{nnz}_A$ denotes the number of non-zeros of a matrix $A$, $n_{LB}$ is the size of the largest block of $R$ and $t_{sol}$ is the time needed to obtain a solution of the system $(L - \tau J)x = b$.
If we look at the eigenvalue plot (Figure 6.1), we can see how the sparsing parameter $\rho$ affects the change of the eigenvalues. Especially, the changes in the oscillatory and the non-stiff eigenvalues are only very small. Consequently, the long time behavior of the simulation is changed

| | $\text{nnz}_{L-\tau J}$ | $\text{nnz}_R$ | $n_{LB}$ | $t_{sol}$ |
|---|---|---|---|---|
| no sparsing | 1306 | 2107 | 302 | 0.63 ms |
| $\rho = 0.01$ | 907 | 1671 | 295 | 0.45 ms |
| $\rho = 0.05$ | 893 | 1622 | 293 | 0.45 ms |
| $\rho = 0.1$ | 897 | 1598 | 264 | 0.43 ms |
| $\rho = 0.5$ | 858 | 1353 | 246 | 0.34 ms |
| $\rho = 1.0$ | 822 | 1144 | 234 | 0.29 ms |

Table 6.1: Results for sparsing with different parameters (DAE with large algebraic part)

only minimally. On the other hand, the stiff eigenvalues change considerably and some new eigenvalues appear due to perturbations of the algebraic part. In simulations, these eigenvalues play a minor role, because they represent rapidly decaying modes. So changes in the stiff eigenvalues are a desired degree of freedom.
If we take a look at one component of the solution trajectory at Figure 6.2 (the relative angle between two flanges of an elastically modelled gear in the first joint of the robot) we see that the differences between the sparsed and the unsparsed trajectory are small, in this example smaller than 6 percent of the numerical range of the solution. This accuracy is sufficient for real time applications. If we take a closer look on the errors we can see that they are phase errors, clearly a consequence of the small changes in the eigenvalues. We also note that there is no substantial difference in the size of the errors corresponding to $\rho = 0.01$ and the errors corresponding to $\rho = 1$.
Concerning the overall performance, we have to say that 17 function evaluations per time step are necessary. Each evaluation takes about 0.15 ms, so the time needed for one step is between 2.8 ms and 3.2 ms. If we were able to compute the Jacobian matrix locally in the frame of inline integration, we might achieve times between 0.8 ms per step (without sparsing) and 0.45 ms per step (with sparsing).

## 6.3 Differential algebraic systems with a small algebraic part

After the BLT-Transformation (see Section 2.2.3) Dymola can reduce the sizes of the equation system even further by "tearing". If tearing is applied to the robot model, we obtain a differential algebraic equation system with 78 differential and 39 algebraic variables. Due to its reduced size, the time for the solution of one resulting equation system is lower than without tearing. With the help of sparsing we can reduce the time for solving the system to about two thirds.
A closer look at Table 6.2 shows that the factorization time does not necessarily decrease to-

| | $\text{nnz}_{L-\tau J}$ | $\text{nnz}_R$ | $n_{LB}$ | $t_{fac}$ |
|---|---|---|---|---|
| no sparsing | 449 | 746 | 105 | 0.14 ms |
| $\rho = 0.01$ | 342 | 533 | 89 | 0.12 ms |
| $\rho = 0.05$ | 326 | 484 | 89 | 0.13 ms |
| $\rho = 0.1$ | 319 | 454 | 88 | 0.12 ms |
| $\rho = 0.5$ | 283 | 334 | 34 | 0.09 ms |
| $\rho = 1.0$ | 277 | 322 | 34 | 0.09 ms |

Table 6.2: Results for sparsing with different parameters (DAE with small algebraic part)

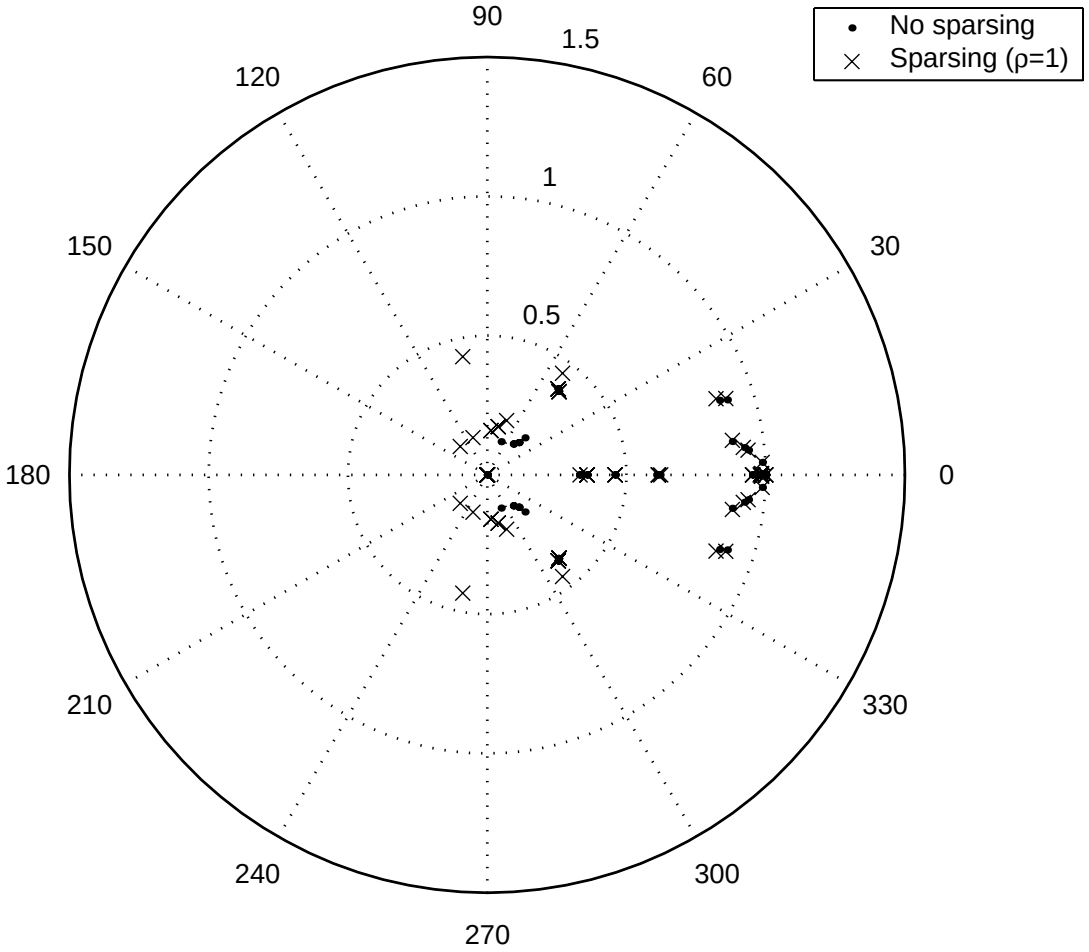


Figure 6.1: Eigenvalue changes due to sparsing.

gether with the number of non-zero elements. It seems that some non-zero structures allow faster factorizations than others.
The eigenvalues change in a similar way as they do in Section 6.2, and the changes in the solution trajectory are also of similar kind.
Concerning the overall performance, we have a similar picture. 19 function evaluations per time step are necessary. Each evaluation takes about 0.12 ms, so the time needed for one step is between 2.7 ms and 2.8 ms. If we were able to compute the Jacobian matrix locally in the frame of inline integration, we might achieve times between 0.3 ms per step (without sparsing) and 0.2 ms per step (with sparsing).

## 6.4 Ordinary differential equations

Considering the ODE model, we observe two new aspects. First, there are no algebraic variables. This suggests that sparsing will become easier and more efficient than in the case of a DAE. Secondly, during the evaluation of the right hand side systems of equations are solved. This means that the Jacobian contains small non-zero elements that originate from errors in the solution of the equation systems. Hence, the Jacobian matrix is rather dense, and therefore the performance

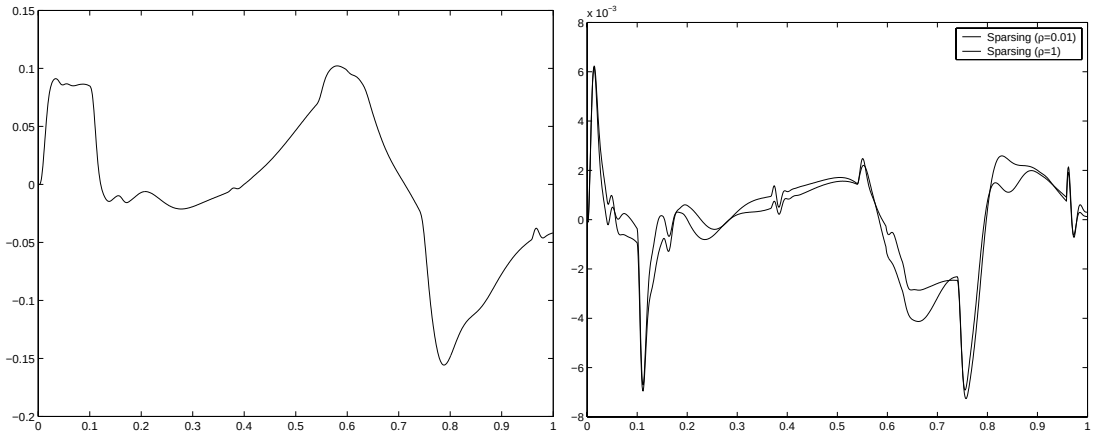


Figure 6.2: Test trajectory, computed with an unsparsed Jacobian and deviations from this trajectory introduced by sparsing with two different parameters $\rho$.

| | $\text{nnz}_{L-\tau J}$ | $\text{nnz}_R$ | $n_{LB}$ | $t_{fac}$ |
|---|---|---|---|---|
| no sparsing | 924 | 2223 | 66 | 0.45 ms |
| $\rho = 0.01$ | 233 | 314 | 56 | 0.08 ms |
| $\rho = 0.05$ | 226 | 311 | 56 | 0.09 ms |
| $\rho = 0.1$ | 200 | 244 | 34 | 0.07 ms |
| $\rho = 0.5$ | 181 | 160 | 10 | 0.05 ms |
| $\rho = 1.0$ | 181 | 160 | 10 | 0.05 ms |

Table 6.3: Results for sparsing with different parameters (ODE case)

of the sparse factorization routine in the unsparsed case is rather poor. Furthermore, there are dense rows in the matrix, which means that column grouping does not have any benefits here. Therefore, sparsing is expected to lead to a good speedup in this case. It can be seen in Table

| | $\text{nnz}_{L-\tau J}$ | $\text{nnz}_R$ | $n_{LB}$ | $t_{fac}$ |
|---|---|---|---|---|
| $\rho = 0.1, \tau = 0.1ms$ | 140 | 150 | 8 | 0.037 ms |
| $\rho = 0.1, \tau = 1.0ms$ | 200 | 244 | 34 | 0.07 ms |
| $\rho = 0.1, \tau = 10.0ms$ | 222 | 304 | 46 | 0.08 ms |
| $\rho = 1.0, \tau = 0.1ms$ | 88 | 83 | 2 | 0.014 ms |
| $\rho = 1.0, \tau = 1.0ms$ | 181 | 160 | 10 | 0.05 ms |
| $\rho = 1.0, \tau = 10.0ms$ | 192 | 212 | 22 | 0.05 ms |

Table 6.4: Results for sparsing with different parameters and step-sizes

6.3 that the number of non-zeros was reduced by a factor of five and the speedup for the solution of the equation systems was about a factor of ten. The very sparse structure of the resulting matrix shows also that sparsing is more efficient for ODEs than for DAEs. Here sparsing even helps us to save function evaluations, because we can leave out several small non-zero elements in the dense row, even for the estimation of the Jacobian matrix. Consequently, we do not have to evaluate the right hand side 78 times, but only 16 times (see Section 5.3).
It is interesting to observe, how the sparsity structure changes for different steps-izes. We expect that the number of non-zero elements decreases with the step-size. Numerical experiments verify this conjecture. The results of this experiment can be looked up at Table 6.4. For $\rho = 1.0$ and

$\tau = 0.1ms$ we obtain a tridiagonal system with only 10 non-diagonal elements. Of course, the diagonal is full anyway, because $L$ is the identity matrix.

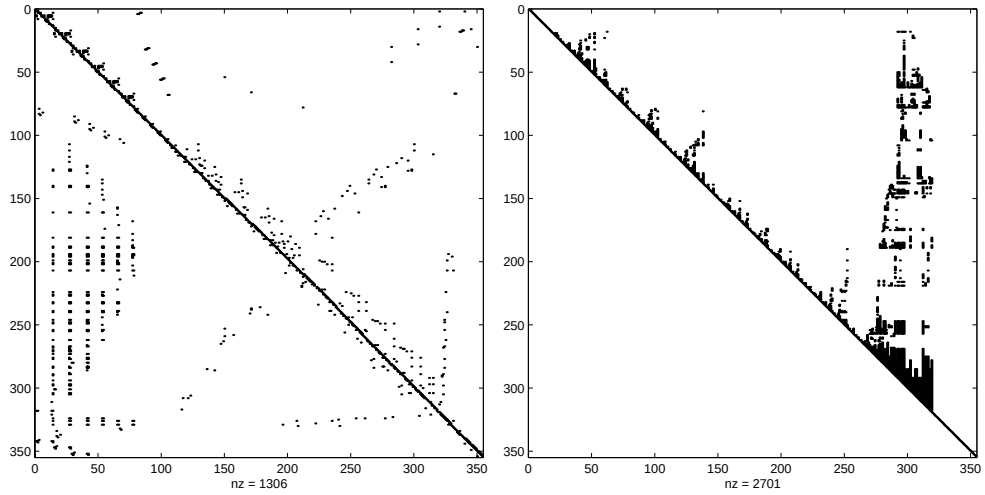


Figure 6.3: The Jacobian $Df$ of the robot model before sparsing and $R$

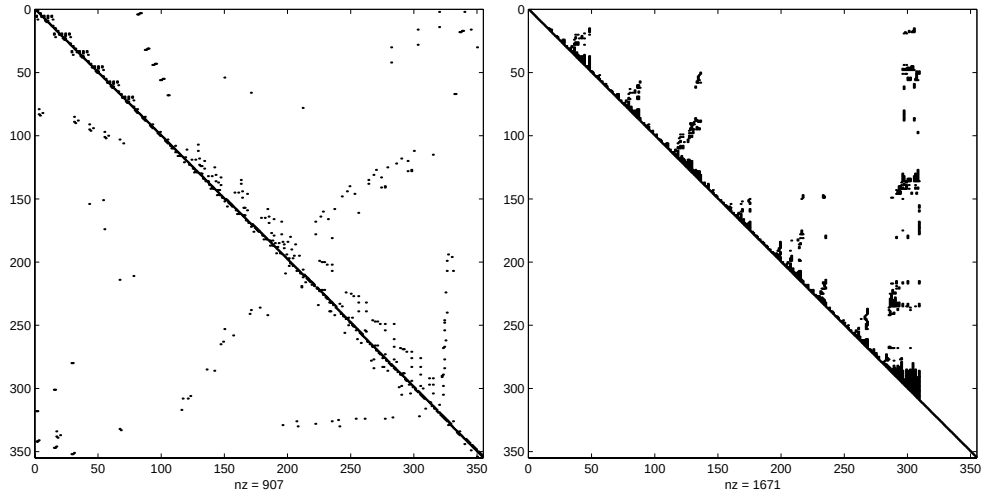


Figure 6.4: The Jacobian $Df$ of the robot model after sparsing ($\rho = 0.01$) and $R$

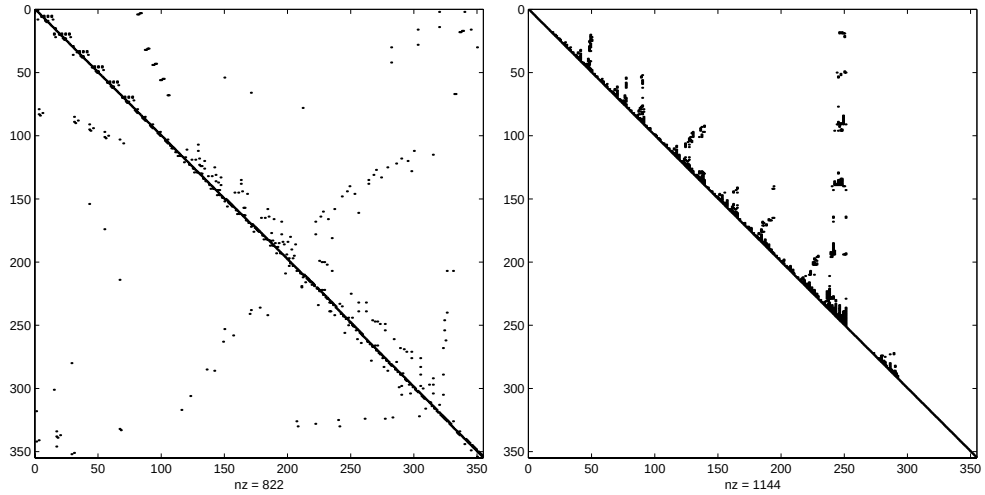


Figure 6.5: The Jacobian $Df$ of the robot model after sparsing ($\rho = 1.0$) and $R$

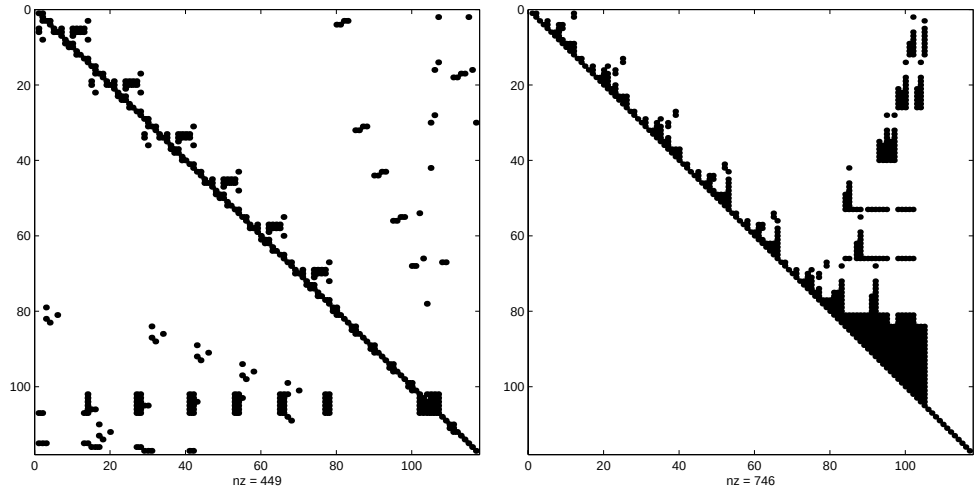


Figure 6.6: The Jacobian $Df$ of the robot model before sparsing and $R$.

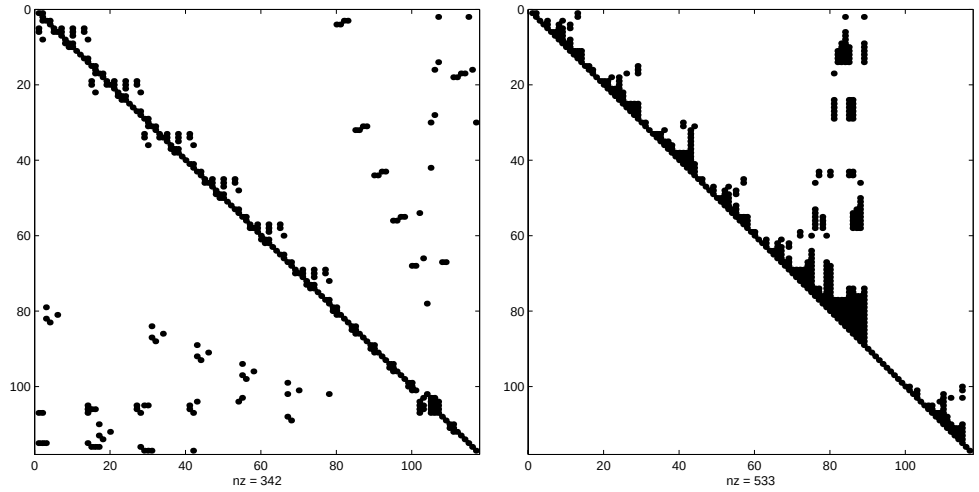


Figure 6.7: The Jacobian $Df$ of the robot model after sparsing ($\rho = 0.01$) and $R$.

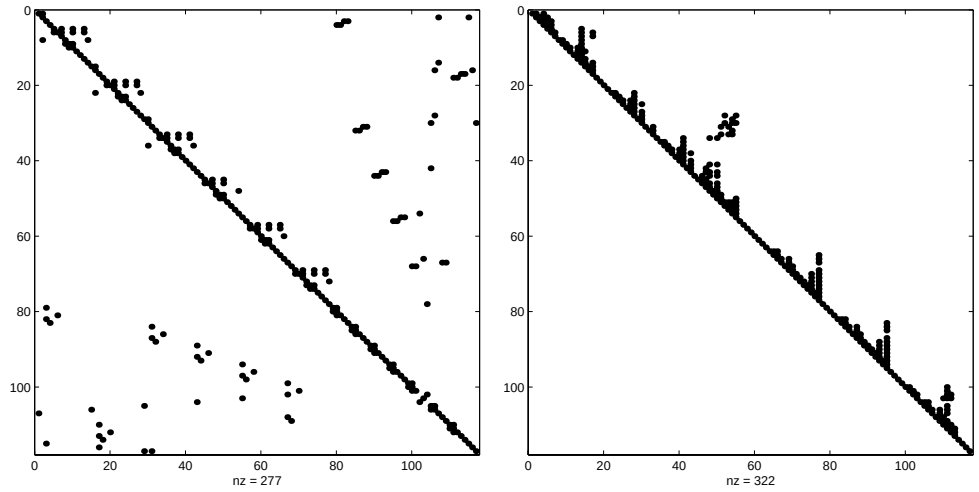


Figure 6.8: The Jacobian $Df$ of the robot model after sparsing ($\rho = 1.0$) and $R$.

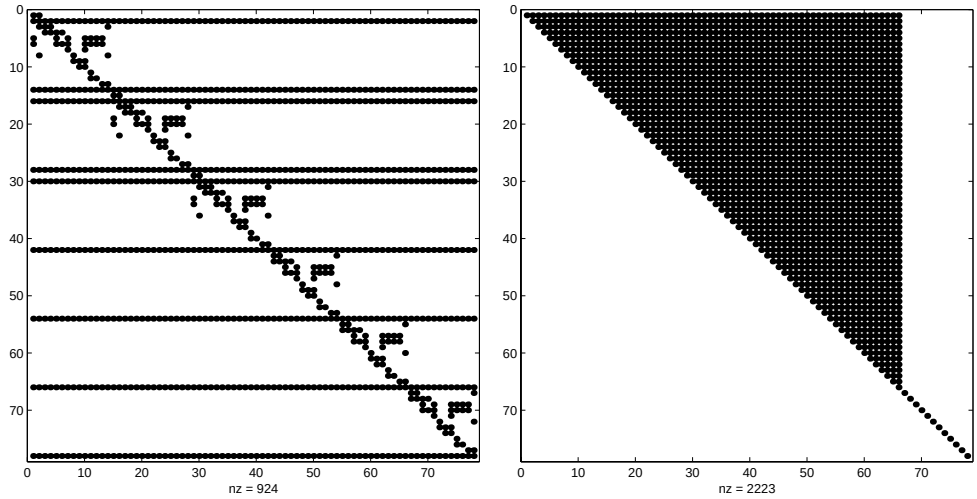


Figure 6.9: The Jacobian $Df$ of the robot model before sparsing and $R$.

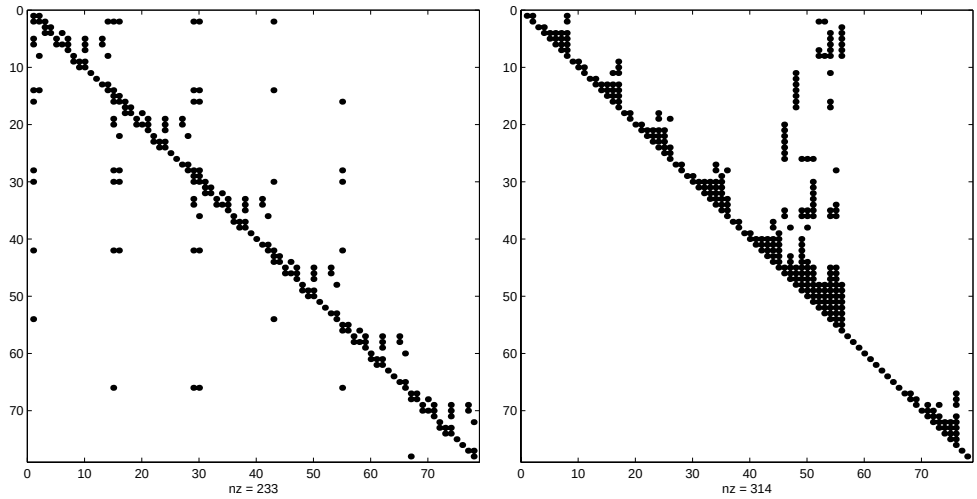


Figure 6.10: The Jacobian $Df$ of the robot model after sparsing ($\rho = 0.01$) and $R$.

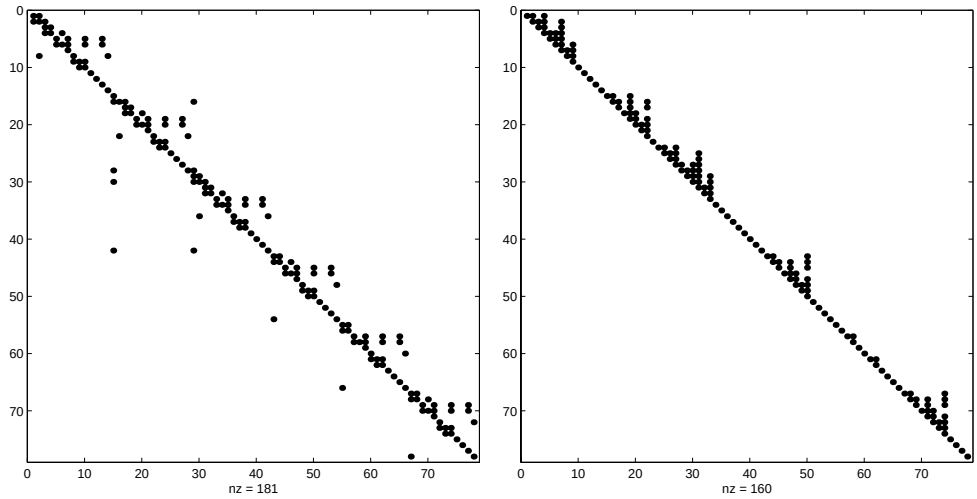


Figure 6.11: The Jacobian $Df$ of the robot model after sparsing ($\rho = 1.0$) and $R$.

# Chapter 7

# Conclusions and Outlook

We have shown that sparsing of the Jacobian is a practicable way of improving efficiency in real time simulation using one of the few possibilities of adaptivity in this field.
We have explored the theoretical relationship between sparsing and extrapolation methods based on the linearly implicit Euler method in the case of a differential algebraic system. Results about stability and the order of these methods were achieved.
We have derived and investigated a first order sparsing criterion, that relies on first order perturbation theory of matrix pairs, and that is suitable for ODEs as well as for DAEs. We have developed and tested an algorithm for sparsing, that is based on this criterion. Experiments have demonstrated the efficiency of this algorithm.
We have shown that it is possible to implement an efficient real time simulator for stiff differential equations and differential algebraic systems, that relies on sparse matrix methods. This was achieved by the combination of preprocessing and simple real time capable routines with low overhead. A key issue in this respect is the usage of linearly implicit methods together with a sparse orthogonal factorization routine.
It may be possible to carry over the theoretical results about sparsing to classical integration methods and to develop an efficient refined sparsing criterion for dynamic sparsing.
Moreover, the encouraging experimental results about damping of algebraic variables may lead to further investigations in this field.

# Bibliography


[1] P. Beater. *Entwurf hydraulischer Maschinen*. Springer, 1999.

[2] Åke Björk. *Numerical Methods for Least Squares Problems*. SIAM, 1996.

[3] E. Carpanzano and R. Girelli. The tearing problem: Definition, algorithm and application to generate efficient computational code from DAE systems. In *Proceedings of 2nd Mathmod Vienna*. IMACS Symposium on Mathematical Modelling, 1997.

[4] T. F. Coleman, A. Edenbrandt, and J.R. Gilbert. Predicting fill for sparse orthogonal factorization. *J. Assoc. Comput. Mach.*, 33:517–532, 1986.

[5] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software*, 10(3):329–345, 1994.

[6] John E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, 1983.

[7] Peter Deuflhard. Recent progress in extrapolation methods for ordinary differential equations. *SIAM Review*, Vol. 27, No 4:505–535, 1985.

[8] Peter Deuflhard. *Newton Methods for Nonlinear Problems. Affine Invariance Principles and Adaptive Algorithms*. Springer-Verlag, to be published.

[9] Peter Deuflhard and Folkmar Bornemann. *Numerische Mathematik II : Integration gewöhnlicher Differentialgleichungen*. de Gruyter, 1994.

[10] Peter Deuflhard, Ernst Hairer, and J. Zugck. One-step and extrapolation methods for differential-algebraic systems. *Numerische Mathematik*, 51:501–516, 1987.

[11] Peter Deuflhard and Ulrich Nowak. Extrapolation integrators for quasilinear implicit ODEs. In Peter Deuflhard and Engquist B., editors, *Large-Scale scientific computing*. Birkhäuser, Boston, 1987.

[12] Iain S. Duff and Ulrich Nowak. On sparse solvers in a stiff integrator of extrapolation type. *IMA Journal of Numerical Analysis*, 7:391–405, 1987.

[13] I.S. Duff, A.M. Erismann, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.

[14] Dymola. Dynasim AB, Lund, Sweden. Homepage: http://www.dynasim.se.

[15] H. Elmqvist, F. Cellier, and M. Otter. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In *Proceedings: European Simulation Multiconference Prague*, pages XXIII–XXXIV, 1995.

[16] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978. Report CODEN:LUTFD2/(TFRT–1015).

[17] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Object-oriented and hybrid modelling in Modelica. *Journal Européen des systémes automatisés*, 35,1:1 á X, 2001.

[18] Hilding Elmqvist and Martin Otter. Methods for tearing systems of equations in object-oriented modelling. In *Proceedings ESM'94 European Simulation Multiconference*, pages 326–332, 1994.

[19] Georg Färber. *Prozessrechentechnik.* Springer, 3. edition, 1994.

[20] Alan George and Michael T. Heath. Solution of sparse linear least squares problems using givens rotations. In Åke Björk, Robert J. Plemmons, and Hans Schneider, editors, *Large Scale Matrix Problems.* Elsevier North Holland, 1981.

[21] John R. Gilbert and Esmond G. Ng. Predicting structure in sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 15(1):62–79, 1994.

[22] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, third edition, 1996.

[23] Ernst Hairer, S. P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems.* Series in Computational Mathematics. Springer, third edition, 2001.

[24] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential Algebraic Problems.* Series in Computational Mathematics. Springer, second edition, 1996.

[25] K. Kautzsch. Hardware- / Computer-in-the-loop-Simulation. Werkzeuge zur Entwicklng und Analyse gelenkter Flugsysteme. Technical Report 94-D4-073, DIEHL Gmbh & Co., Röthenbach / Peg., 1994.

[26] Frank Kessler and Gebert Jürgen. Testautomatisierung und Antriebsmodellierung an HIL-Steuergeräteprüfständen in der BMW Getriebeentwicklung. *ATZ Automobiltechnische Zeitschrift*, 102(5):312–323, 2000.

[27] G. Kron. Diakoptics - the piecewise solution of linear systems. *MacDonald & Co.*, 1962.

[28] Christian Leimegger and Dierk Schröder. Hochdynamische Verbrennungsmotor- und Rad-Straße-Simulation an einem modellgeführten hardware-in-the-loop PKW-Antriebsstrangprüfstand. In *4. VDI Mechatronik Tagung 2001 - Innovative Produktentwicklung*, pages 389–409. VDI-Gesellschaft - Entwicklung Konstruktion Vertrieb, 2001.

[29] Christian Lubich. Linearly implicit extrapolation methods for differential-algebraic systems. *Numerische Mathematik*, 55:197–211, 1989.

[30] Christian Lubich and M. Roche. Rosenbrock methods for differential-algebraic systems with solution dependent singular matrix multiplying the derivative. *Computing*, 43:325–342, 1990.

[31] S.E. Mattsson and G. Söderlind. Index reduction in differential algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[32] Modelica. a unified object-oriented language for physical systems modelling. Modelica homepage: http://www.modelica.org.

[33] Ulrich Nowak. Dynamic sparsing in stiff extrapolation methods. *Impact of Comput. in Science and Engrg.*, 5:53–74, 1993.

[34] Martin Otter. Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter. *Fortschrittsberichte VDI Reihe 20*, 147, 1995.

[35] Martin Otter. Objektorientierte Modellierung von Antriebssystemen. In Dierk Schröder, editor, *Elektrische Antriebe - Regelung von Antriebssystemen*, chapter 20, pages 894–1009. Springer, 2001.

[36] Martin Otter et al. Objektorientierte Modellierung Physikalischer Systeme 1-17. *at Automatisierungstechnik*, 47/1 - 48/12, 1999/2000.

[37] Thomas H. Robey and Deborah L. Sulsky. Row ordering for a sparse QR decomposition. *SIAM Journal for Matrix Analysis and Applications*, 15(4):1208–1225, 1994.

[38] A Ruhe. An algorithm for numerical determination of the structure of a general matrix. *BIT*, 10:196–216, 1970.

[39] Anton Schiela and Hans Olsson. Mixed-mode integration for real-time simulation. In *Modelica 2000 Workshop Proceedings*, pages 69–75. The Modelica Association, 2000.

[40] T. Steinhaug and A. Wolfbrandt. An attempt to avoid exact Jacobian and nonlinear equations in the numerical solution of stiff differential equations. *Math. Comp.*, 33:521–534, 1979.

[41] G. W. Stewart and J. Sun. *Matrix Perturbation Theory*. Computer Science and Scientific Computing. Academic Press, 1990.

[42] G.W. Stewart. Algorithm 406: HQR3 and EXCHNG: Fortran subroutines for calculating and ordering the eigenvalues of a real upper Hessenberg matrix. *ACM Trans. Math. Soft.*, 2:275–280, 1976.

[43] P. Van Dooren. Algorithm 590: DSUBSP and EXCHQZ: FORTRAN subroutines for computing deflating subspaces with specified spectrum. *ACM Transactions on Mathematical Software*, 8(4):376–382, December 1982.

[44] M. Weiner et al. Partitioning strategies in Runge-Kutta type methods. *IMA Journal of Numerical Analysis*, 13:303–319, 1993.