9 Numerical methods for complex ODE / DAE systems

Jan F. Broenink
University of Twente, EE Department, Enschede, Netherlands

9.1 Introduction

In this chapter, we will further elaborate on numerical methods used for simulation of engineering systems described by ordinary differential equations, either explicit (ODE) or implicit (DAE).

Before going into detail on numerical integration methods and model properties relevant for simulation, first some basics are recalled:

Numerical integration is an *approximation*

It is crucial to consider that numerical integration is an *approximation* of the real integration process. Besides errors due to the time discretization, errors are generated because numerical integration formulas are approximations of the real integration function. The solution generated by numerical solver has thus a certain *accuracy*. This implies that different numerical integration methods exist; each of them having their own class of models for which they are suitable. This means that we have to analyze both models as well as numerical integration methods on aspects relevant for simulation.

Time discretization

To compute the variables in the model as a time function on a digital computer, time will be discretized. Otherwise, we have an infinite amount of values of time between t_0 and t_e, since t is a real variable. Only at discrete points t_k the model is computed. The result is that the model variables are available as a discrete series of values, and not as a continuous function. Such series are shown on a screen as dots. To give an idea of a continuous curve (of which the series of points is an approximation), straight lines interconnect the dots. This is in fact *linear interpolation*. In general this is accurate enough, because the distance between two adjacent points is small enough.

time step, h_k, *integration interval*

The time distance between two adjacent points is called the *time step*, h_k, or *integration interval*. This h_k need not necessarily be constant. As notation we use $x_k = x(t_k)$ in stead of $x(t)$. In the following we use the shorthand notation to denote a time-discretized variable.

Simulation Model

The model suitable for simulation consists of a set of *Ordinary Differential Equations*, and is called a *simulation model*. These differential equations describe the system as a set of *state equations*, where the initial values of the state variables (also called *initial conditions*) are known. In numerical mathematics, this is called an *Initial Value Problem* (IVP)*:*

Initial Value Problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{u},t) \tag{1}$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x},\mathbf{u},t) \tag{2}$$

State variables

where $\mathbf{x}$ is the array of *state variables*, also called the state vector; $\mathbf{x}(0)$, the initial values, which are known. $\mathbf{u}$ is the array of input variables. $\mathbf{f}$ are the state equations. $\mathbf{y}$ is the array of *output variables*, computed from the states and inputs using the functions $\mathbf{g}$.

Output variables

Model equations

$\mathbf{y}$ and $\mathbf{g}$ are not necessary to compute the state of the model as function of time t. Only equation (1) is needed. The equations $\mathbf{f}$ and $\mathbf{g}$ together form the *model equations*. $\mathbf{f}$ can be implicit or explicit.

Explicit model

Equations (1) and (2) denote an explicit model: all the information to compute $\dot{\mathbf{x}}$ is available. When, however, this is not the case, the model is

implicit. This means that **f** contains one or more implicit equations. The general form of equation (1) changes to:

$$\mathbf{f}^{\#}(\mathbf{x},\mathbf{u},t) = 0 \tag{3}$$

ODE
DAE

If we rewrite equation (3) to a kind of explicit form, we get a set of ODEs with algebraic constraint equations, which is called a set of *Differential Algebraic Equations* (DAEs). Its most general form, including the update of equation (1) is:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\dot{\mathbf{x}},\mathbf{u},t) \tag{4}$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x},\dot{\mathbf{x}},\mathbf{u},t) \tag{5}$$

Implicit model

The occurrence of $\dot{\mathbf{x}}$ in the argument of **f** makes the model implicit. The set of output equations is basically the same, but also dependent of $\dot{\mathbf{x}}$.

Models containing implicit equations need special provisions in the numerical integration method, namely iteration. It is also possible to eliminate $\dot{\mathbf{x}}$ by rewriting the state equations. The implicit model *changes* to an explicit model. Explicit models mostly are easier and faster to simulate. However, interesting variables can disappear due to the rewriting process, and the accuracy of the simulation result might get too worse.

In the next sections, first numerical integration methods are discussed, including their stability and accuracy regions. After that, model properties relevant for simulation are presented, including how these properties can already be discovered in the bond graph (so equations need *not* be derived). Furthermore, possibilities are discussed on adapting the model in order to be able to use the simplest numerical integration methods.

9.2 Numerical integration methods

A numerical integration method specifies how $\mathbf{x}_{k+1}$ is approximated in terms of **x** itself and its derivatives, both at previous, current or future moments of time. Practical numerical methods consist of a linear combination of these terms $\mathbf{x}^{(i)}_{k+1-j}$ using parameters α_{ij}, n and r ($\dot{x} = x^{(1)}$):

$$\sum_{i=0}^{n}\sum_{j=0}^{r} \alpha_{ij} h^{i}_{k+1-j} \mathbf{x}^{(i)}_{k+1-j} = 0 \tag{6}$$

The method is explicit if all α_{i0}=0 for i=1...n, otherwise the method is implicit.

Truncation errors
Round-off errors

Equation (6) is an approximation, so errors are made:

- *Truncation errors*, due to the non-idealness of the approximation formula.
- *Round-off errors*, due to the limited machine precision.

At smaller timesteps, the truncation error usually becomes smaller, whereas the round-off errors become larger. The round-off error only becomes dominant, if the numbers calculated by the method cannot be represented in the computer with sufficient accuracy. Adaptive changes of the time step can ensure that, at least locally per time step, the errors remain within specified bounds. This is achieved by a numerical integration method, which adapts to local changes of eigenvalues of the model. Such strategies also save computation time by allowing the largest time steps possible.

For a useful approximation, the time step h_k depends on both the model and the integration method.

The form of the numerical integration method, either explicit or implicit, the number of steps and the order of derivatives involved, significantly influence numerical stability and effective applicability of the method. Explicit numerical integration methods have less computational load than implicit methods, but unfortunately they also have a small stability region, and thus relatively small time steps h_k are necessary. The disadvantage of implicit numerical integration methods is that iterative calculations involving matrix inversion or its equivalence (such as a Newton-Raphson scheme) are needed to obtain $\mathbf{x}_{k+1}$ out of Equation (6).

Most numerical integration methods have been developed and evaluated for linear equations or for nonlinear equations only in the vicinity of the point of linearization. Nevertheless, numerical integration methods can be applied to sets of nonlinear equations, because the local properties of nonlinear equations are, in general, similar to those of linear or linearized equations, and changes of the solution at each time step can be assumed to be within the local area (except for strong linearities such as a jump).

Basically there are 3 general classes of integration methods, categorized by the number of steps and order of derivatives:

a) Multistep, high-order derivative methods
   the full version of equation (6).
b) Multistep, first-order derivatives methods
   n=1, only x and its derivative are used. Higher order methods take more values of the past, until x_{k+1-n}.
c) One-step higher-order derivatives methods
   r=1, only values of x_k and $x^{(i)}_k$, the i^{th} derivative of x_k, are used. These methods are Runge-Kutta methods. No values of previous moments are used.

Category a) is the most general one: b) and c) are special cases of a). Only b) and c) are of practical importance because of the simplicity of such algorithms, and together they cover most of the existing numerical integration methods used in practice for the solution of first-order ordinary differential equations. However, if the set of equations to be solved contains higher-order differential equations, multistep higher-order methods can be efficient.

9.2.1 Multistep, first-order derivative methods

For multistep, first-order derivative methods, the parameter *n* is equal to one Equation (6) can be written as:

$$\mathbf{x}_{k+1} = \alpha_{10} h_{k+1} \dot{\mathbf{x}}_{k+1} + \sum_{j=1}^{r} \alpha_{0j} \mathbf{x}_{k+1-j} + \sum_{j=1}^{r} \alpha_{1j} h_{k+1-j} \dot{\mathbf{x}}_{k+1-j} \qquad (7)$$

If $\alpha_{10} = 0$, the method is explicit, otherwise the method is implicit, as $\dot{\mathbf{x}}_{k+1}$ is the only term at $\mathbf{x}_{k+1}$ at the right-hand side. The first summation represents **x** at previous moments of time; the second summation represents the derivative of **x** at previous moments of time. If *r* = 1, then it is a *one*-step method: only **x**(*t*) and its derivative are used. At *r* > 1, it is a multistep method, which is not self-starting: at the beginning, old values of $\mathbf{x}_{k-1}$ etc. are not yet known. A one-step method is used to start. Some examples are presented in Table 1 ($\mathbf{f}(\mathbf{x}_k)$ represents the model equations):

AB-1	Adams Bashforth = explicit Euler	$\mathbf{x}_{k+1} = \mathbf{x}_k + h_k\dot{\mathbf{x}}_k = \mathbf{x}_k + h_k\mathbf{f}_k$
AB-2	Adams Bashforth = trapezium rule	$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{3}{2}h_k\dot{\mathbf{x}}_k - \frac{1}{2}h_k\dot{\mathbf{x}}_{k-1}$
AM-1	Adams Moulton = implicit Euler	$\mathbf{x}_{k+1} = \mathbf{x}_k + h_{k+1}\dot{\mathbf{x}}_{k+1}$
AM-2	Adams-Moulton = trapezium rule	$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{2}h_{k+1}\dot{\mathbf{x}}_{k+1} + \frac{1}{2}h_k\dot{\mathbf{x}}_k$
BDF	Backward Differentiation Formula	$\mathbf{x}_{k+1,m+1} = \mathbf{P}^{-1}\varepsilon + \mathbf{x}_{k+1,m}$ $\mathbf{P} = \frac{\partial}{\partial \mathbf{x}}\left(\dot{\mathbf{x}} - \mathbf{f}\right)$ $\dot{\mathbf{x}}_{k+1} = \frac{1}{h_k}\sum_{j=0}^{r}\alpha_i\mathbf{x}_{k+1-j} \quad 1 \le r \le 6$

Table 1 Multistep first-order derivative methods

The specifications of the coefficients α_{ij} for the listed methods up to r=6 can be found in chapter 7 and 11 of (Gear, 1971). Also chapter 7 of (Atkinson, 1989) presents details of integration methods.

The Adams-Bashforth methods are explicit method, require one evaluation of the model per time step, and have a limited stability region (conditionally stable). The Adams-Moulton methods are implicit in which $\dot{\mathbf{x}}_{k+1}$ is substituted as a fraction of $\mathbf{x}_{k+1}$, and require one evaluation of the model. The second order method is A-stable (absolute stable, the stability region is the whole left half plane), and can therefore be used for stiff models. The local truncation error of these methods is of order $O(h^{r+1})$.

BDF method

The BDF formula is implicit. The residue ε is calculated as the difference of $\dot{\mathbf{x}}$ from the integration method and $\dot{\mathbf{x}}$ from the model: the index *m* indicates the number of iterations (see Figure 9-1**)**.

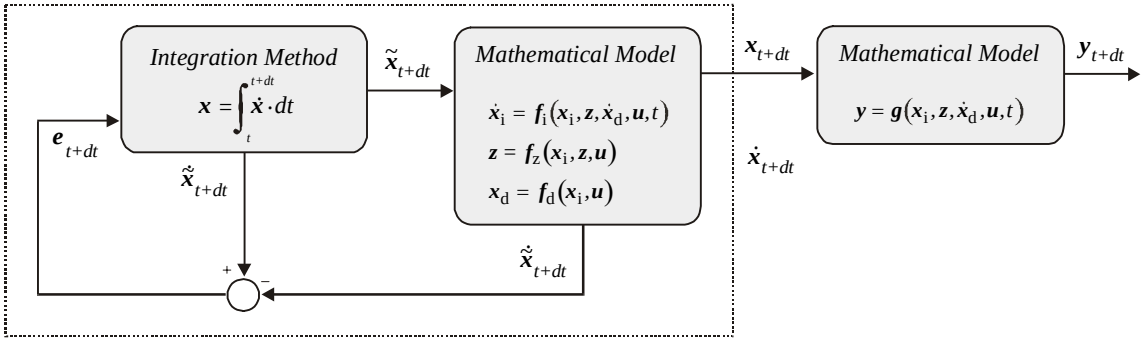

Figure 9-1: One integration step for a DAE method. The division of the state vector x into three parts is explained in section 9.3.1.

The matrix **P** is the Jacobian of the DAE system. $\dot{\mathbf{x}}_{k+1}$ is approximated by a weighted sum of previous values, via a backward difference; this is the reason of the name of the method. Up to 5 previous values can be used. More terms from the past result in an unstable BDF method. The factors α_I depend on the time step. The updating of these parameters as a consequence of variable time step is rather computational intensive, but this is rather well optimized in the DASSLRT code, a well-known implementation of the BDF method. This implementation is also used in 20-SIM.

Predictor Corrector methods

At a *predictor-corrector* method, first a *predictor* step is done (= a forecast) of $\mathbf{x}_{k+1}$ and thus of $\mathbf{f}(\mathbf{x}_{k+1})$. Then, this result is used to compute a *corrector* step (= a better approximation of $\mathbf{x}_{k+1}$). As predictor, an explicit method is used and as corrector an implicit one.

Variable step methods

At *variable-step* methods, the step size h_{k+1} is adjusted every time step to the dynamics of the system, such that $h_k\lambda_{i,k}$ stays in the stability region of the numerical integration method used.

9.2.2 One-step higher-order derivative methods

These are the Runge-Kutta methods. The formula is:

$$\mathbf{x}_{k+1} = \alpha_{01}\mathbf{x}_k + \sum_{i=1}^{n}\alpha_{i0}h_{k+1}^{i}\mathbf{x}_{k+1}^{(i)} + \sum_{i=1}^{n}\alpha_{i1}h_k^{i}\mathbf{x}_k^{(i)} \tag{8}$$

If one of the parameters α_{i0} is *not* equal to zero, the method is implicit. The first summation denotes the derivatives of $\mathbf{x}_{k+1}$. The second summation denotes the derivatives of $\mathbf{x}_k$. Because $\dot{\mathbf{x}}$ is computed via the model equations, the higher derivatives of $\mathbf{x}$ can be obtained by subsequent differentiation of the model equations $\mathbf{f}(\mathbf{x},\mathbf{u},t)$. However, this is a very computational-intensive process. Therefore, they are computed via extrapolation. Writing $\mathbf{x}_{k+1}$ as a Taylor's series, the coefficients of the numerical method, α_{ij}, can be determined. This way is useful for theoretical research, but for a practical algorithm, the α_{ij}'s are approximated via *extrapolation*. $\mathbf{x}$ is calculated at a number of points $t_k + \eta_i h_k$ where $0 \le \eta_i \le 1$ The values $\mathbf{x}(t_k + \eta_i h_k)$ and $\mathbf{x}(t_k)$ are used to match terms in the Taylor's series, which result in a specification of a numerical integration method ((Gear, 1971), section 2.4). This specification allows some freedom: not all coefficients are determined. These methods are called *r-stage Runge-Kutta* methods. They need r model computations per time step. The general r-stage Runge-Kutta method is given by:

$$\begin{aligned} x_Q &= x_k + \sum_{j=1}^{q}\beta_{qj}\mathbf{z}_j \quad q = 1,2,\ldots r \\ t_Q &= t_k + h_k\sum_{j=1}^{q}\beta_{qj} \\ \mathbf{z}_q &= h_k\mathbf{f}(\mathbf{x}_Q,\mathbf{t}_Q) \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \sum_{q=1}^{r}\gamma_q\mathbf{z}_q \end{aligned} \tag{9}$$

Equations for the parameters γ_q and β_{qj} are derived by matching terms with the Taylor's series. If β_{qj} =0 for j $\geq$ q, the method is explicit, otherwise implicit. As indicated above, there are more parameters to determine than equations available, so that a family of methods results. For explicit methods, the order is equal to the stage r for r$\leq$4. Otherwise, the order is less than the stage. The implicit methods can have a maximal order of 2r (Hairer and Wanner, 1996). See for some formulas Table 2.

The explicit Runge-Kutta methods perform calculations straightforwardly, whereas the implicit methods require a set of equations to be solved simultaneously, using iteration. In general it is difficult to justify the additional work of the implicit methods by the increased accuracy, so their use is restricted to some special problem types, like stiff systems.

Name	Coefficients	Algorithm
RK1, explicit Euler	$\gamma_1 = 1$	$\mathbf{q}_1 = h_k\mathbf{f}(\mathbf{x}_k,t_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{q}_1$
RK2, Heun	$\gamma_1 = ½$ $\gamma_2 = ½ \quad \beta_{21} = ½$	$\mathbf{q}_1 = h_k\mathbf{f}(\mathbf{x}_k,t_k)$ $\mathbf{q}_2 = h_k\mathbf{f}(\mathbf{x}_k + ½\mathbf{q}_1, t_k + ½h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + ½\mathbf{q}_1 + ½\mathbf{q}_2$

RK4, Classical	$\gamma_1 = 1/6$ $\gamma_2 = 1/3 \quad \beta_{21} = 1/2$ $\gamma_3 = 1/3 \quad \beta_{32} = 1/2$ $\gamma_4 = 1/6 \quad \beta_{43} = 1$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{q}_2 = h_k \mathbf{f}(\mathbf{x}_k + \frac{1}{2}\mathbf{q}_1, t_k + \frac{1}{2}h_k)$ $\mathbf{q}_3 = h_k \mathbf{f}(\mathbf{x}_k + \frac{1}{2}\mathbf{q}_2, t_k + \frac{1}{2}h_k)$ $\mathbf{q}_4 = h_k \mathbf{f}(\mathbf{x}_k + \mathbf{q}_3, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{6}\mathbf{q}_1 + \frac{1}{3}\mathbf{q}_2 + \frac{1}{3}\mathbf{q}_3 + \frac{1}{6}\mathbf{q}_4$
RK1, implicit Euler	$\gamma_1 = 1 \quad \beta_{11} = 1$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k + \mathbf{q}_1, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{q}_1$
RK2, trapezodial	$\gamma_1 = 1/2 \quad \beta_{11} = 0$ $\gamma_2 = 1/2 \quad \beta_{21} = 1/2$ $\beta_{22} = 1/2$	$\mathbf{q}_1 = h_k \mathbf{f}(\mathbf{x}_k, t_k)$ $\mathbf{q}_2 = h_k \mathbf{f}(\mathbf{x}_k + \frac{1}{2}\mathbf{q}_1 + \frac{1}{2}\mathbf{q}_2, t_k + h_k)$ $\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{1}{2}\mathbf{q}_1 + \frac{1}{2}\mathbf{q}_2$

Table 2 Runge-Kutta methods

Variable time step Runge-Kutta's do exist. Their time step is controlled by calculating each time step twice by either using two Runge-Kutta algorithms of different order denoted as RKi(j), or by means of step doubling. For the first method, computations are minimized by choosing the coefficients such, that some intermediate results are common to both algorithms (Gear, 1971). Two algorithms which are base on the multiple use of intermediate results, are often used: the Runge-Kutta-Merson algorithm and the Runge-Kutta-Fehlberg algorithm, of which the latter is probably the best and a good choice for non-stiff models. In the step doubling method, each basic time step of size h_k is done twice, once as two time steps of size $h_k/2$ and once as one time step of size h_k. The comparison of the two results is an indication for the local truncation error, which is used to compute the new time step. The RKi(j) methods are slightly more efficient as they need fewer evaluations per time step.

Runge Kutta Fehlberg

In 20-SIM the Runge-Kutta-Fehlberg algorithm is implemented, a RK4(5) method, which needs 7 model computations per time step.

9.2.3 General remarks on integration formulas

Some remarks on the order of derivatives, the number of steps, and the coefficients of the numerical integration methods:

1. Order of derivatives

The higher the order of derivatives used in the formula, the more accurate and more flexible the formula is. But at each time step, an accompanying heavy computation load exists, i.e. the computation of the high order derivatives of **f**(**x**,t) must be done.

2. Number of steps

The larger the number of steps, the more accurate and more flexible the formula is. But estimates of the unknown initial values of $\mathbf{x}_k$ are needed, which makes the multiple step methods not really suitable for models with discontinuities.

3. Implicit formula versus explicit formula

Implicit formulas always require the computation of an inverse of a matrix or its equivalent. Explicit formulas are easy to apply to complicated non-linear differential equations, and do not involve the calculation of a matrix inverse, but have a smaller stability region.

4. Interative implicit versus noniterative implicit formulas

As described above, implicit formulas always require a matrix inversion. An implicit algorithm that includes direct matrix inversion is called a non-

iterative algorithm. Of course, if the model is linear, the calculation involves no iteration, but if they are nonlinear, iterative calculations including matrix inversion are needed. On the other hand, an implicit method with iterative calculation equivalent to matrix inversion is often called an iterative algorithm. Iterative calculations without including matrix inversion are needed for both linear and nonlinear equations. The iterative algorithms are easy to apply to nonlinear models but may require more computation time than the noniterative methods when applied to small sets of linear equations and/or large-scale linear equations with systematic equations structures such as a band structure.

5. Coefficients of the integration method

Selection of the coefficients in an integration method depend on the model, e.g. for a stiff model, the coefficients shown be chosen such that the integration method becomes Absolute stable (i.e. the stability region of the method is the whole left-half plane). In case high accuracy is needed, the coefficients should be chosen such that the approximations generate little error.

9.2.4 Stability and accuracy of numerical integration methods

A numerical integration method is stable for all values of $\lambda_i h_k$ where the error in a next step will *not* grow larger.:

$$\varepsilon_{k+1} = z\varepsilon_k \quad \text{with } |z| \leq 1 \tag{10}$$

Numerical instability

The stability region are all those values for $\lambda_i h_k$ for which $|z| \leq 1$ holds. This should cover a considerable part of the left half of the complex plane, because a stable system (model) has its eigenvalues in the left half of the complex plane. Stability regions in the right half complex plane can be useful for nonlinear systems, when the stable eigenvalues penetrate a little in the right half complex plane.

Figure 9-2 shows some stability regions. Inside the curves the methods are stable. The inner circle is the stability region of RK1 or Euler (it should be a circle with radius of 1); the most outer one represents qualitatively the stability region of the classical RK4.

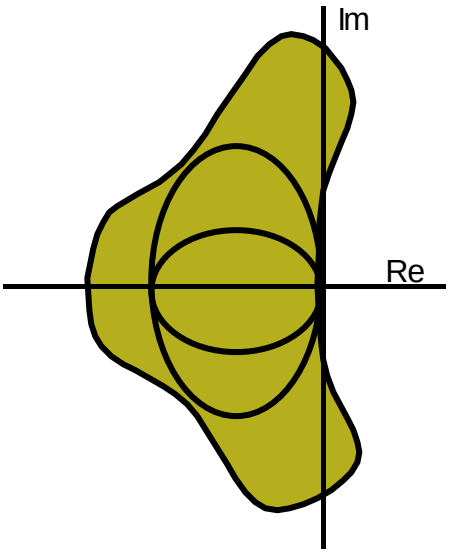

Figure 9-2: Stability regions of RK methods (RK1, RK2, RK4)

Numerical Distortion

Furthermore, the *distortion* caused by the numerical integration method due to the approximation can also be displayed in the complex plane. We can see when a stable eigenvalue moves to the right half plane, and as such causes

numerical instability. Since mostly a model has several eigenvalues, and a numerical integration method only has one integration time step, it is sometimes very difficult to get *all* eigenvalues placed within the stability region.

We will use a test equation to reason about accuracy of the methods. Consider the test model:

$$\dot{x}(t)=\lambda x(t) \quad x(0)=x_0 \tag{11}$$

where λ is an arbitrary complex number, a so-called eigenvalue. The solution, and its time discrete version are:

$$x(t) = e^{\lambda t} \qquad x_{k+1} = E(\lambda h)x_k \tag{12}$$

where E() is the approximation of the e-power series due to the integration method. This approximation shifts the eigenvalue in the left half of the complex plane. This approximated eigenvalue is:

$$e^{\tilde{\lambda} h} = E(\lambda h) \approx e^{\lambda h} \tag{13}$$

Expressing the eigenvalue in an eigenfrequency ω and relative damping ζ, yields the following formulas:

$$\lambda h = -\zeta\omega h \pm j\omega h\sqrt{1-\zeta^2} \tag{14}$$

pseudo natural frequency

pseudo relative damping

$$-\tilde{\zeta}\tilde{\omega} h \pm j\tilde{\omega} h\sqrt{1-\tilde{\zeta}^2} = \ln\left[E\left(-\zeta\omega h \pm j\omega h\sqrt{1-\zeta^2}\right)\right] \tag{15}$$

where $\tilde{\omega}$ is the *pseudo natural frequency* that is computed from the output of the integration method, and $\tilde{\zeta}$ is the *pseudo relative damping*.

Showing some eigenvalues in the complex plane, together with the deformed ones, the effect of approximation by the numerical integration method can be visualized. See Figure 9-3 for a generic deformation diagram, and the appendix for exact deformation diagrams.

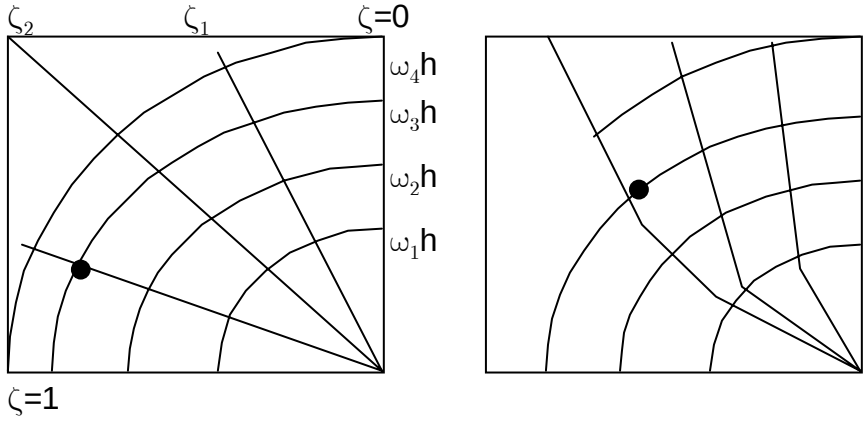

Figure 9-3: Original eigenvalues (left) and deformed eigenvalues due to the integration method.

Interpretation of the stability and deformation diagrams of the numerical integration methods lead to the following remarks:

1. Euler = RK1

The trajectory error causes a *higher* frequency and *less* damping of the pseudo eigenvalue (i.e. the deformed eigenvalue moves higher and more to the imaginary axis). A rule of thumb is to choose the stepsize ten times smaller then the lowest time constant ($h \approx 0.1\tau$).

2. Backward Euler = stiffly stable 1st order
The trajectory error causes a *lower* frequency and *less* damping than the corresponding analytical solution. Some unstable eigenvalues are solved stably. A rule of thumb is to choose the stepsize ten times smaller then the lowest time constant.

3. Stiffly stable 2nd order method
The solution has almost the same properties as that of the backward Euler, but has a slightly higher accuracy. Note, however, undamped eigenvalues may play some undesirable role if excited by an erroneous setting of the initial values.

4. Adams Bashforth, 2nd order method
Models with eigenvalues $0 \leq \lambda h \leq 0.3$ and $0.1 \leq \zeta \leq 1.0$ have only small trajectory errors. At the beginning of the simulation, there may exist some undesirable behavior due to an erroneous setting of the initial values.

5. Runge-Kutta 4th order method
Models with eigenvalues $0 \leq \lambda h \leq 0.5$ and $0 \leq \zeta \leq 1.0$ have only small trajectory errors. Models with eigenvalues $1.5 \leq \lambda h \leq 2.7$ ζ=0, that are undamped models, get an extra damping (numerical damping).

6. Trapezium rule
Models with eigenvalues $0 \leq \lambda h \leq 0.5$ and $0 \leq \zeta \leq 1.0$ have only small trajectory errors. For $\lambda h \leq 0.5$ the result has less damping and a lower frequency that the analytical solution. Models with $0 \leq \zeta \leq 0.5$ have less trajectory errors than those with $0.5 \leq \zeta \leq 1$.
This means that this method is useful for undamped or slightly damped models.

9.3 Model properties for simulation

The form of the equations (explicit or implicit) is very important for simulation: Implicit models can only be simulated using implicit models, whereas explicit models can be simulated by both explicit and implicit integration methods. Causal analysis indicates what form the equations have.

Another model property important for simulation is the parameter values, being concentrated in the eigenvalues.

Furthermore, it is relevant to know whether discontinuities are present in the model.

The three items mentioned here (form of equations, eigenvalues and presence discontinuities) will be discussed next.

9.3.1 Form of equations

We write implicit state equations in a kind of semi explicit form, and divide the state vector **x** into three parts. The model equations (16) (recall of (4))

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}, t) \tag{16}$$

are rewritten to a kind of explicit form, thus we get a set of ODEs with algebraic constraint equations, which is called a set of *Differential Algebraic Equations* (DAEs):

$$\begin{aligned} \dot{\mathbf{x}}_i &= \mathbf{f}_i(\mathbf{x}_i, \mathbf{x}_z, \dot{\mathbf{x}}_d, \mathbf{u}, t) \\ \mathbf{x}_z &= \mathbf{f}_z(\mathbf{x}_i, \mathbf{x}_z, \dot{\mathbf{x}}_d, \mathbf{u}, t) \\ \mathbf{x}_d &= \mathbf{f}_d(\mathbf{x}_i, \mathbf{x}_z, \mathbf{u}, t) \end{aligned} \tag{17}$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x}_i, \mathbf{x}_z, \dot{\mathbf{x}}_d, \mathbf{u}, t) \tag{18}$$

The state vector is divided into three parts:

- $\mathbf{x}_i$ independent state variables
- $\mathbf{x}_z$ additional variables for breaking algebraic loops (also called zero-order causal paths).
- $\mathbf{x}_d$ dependent state variables.

Implicit model

The occurrence of either $\mathbf{x}_z$ or $\mathbf{x}_d$ makes the model implicit. The state equations are also divided into three parts: $\mathbf{f}_i$ are the ODEs and $\mathbf{f}_z$ and $\mathbf{f_d}$ are the algebraic constraint equations. The set of output equations (18) is basically the same, but also dependent of $\mathbf{x}_z$ or $\mathbf{x}_d$.

Matrix Form

If the system is linear, we can write the resulting set of state equations in the standard form, namely,

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{19}$$

where **A** the *system matrix*, and **B** the *input matrix*. The order of the system is the dimension of the square matrix **A** and the order of the set of state equations is the *rank* of **A**. When dependent states or algebraic loops are present, the matrix description is as follows:

$$\mathbf{E}\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{20}$$

where **E** a square matrix. For each differential storage element and for each algebraic loop, **E** contains one row of zeros, appearing at the bottom of **E** when x is filled according to equation (17). Then **E** contains an Identity submatrix in the upper left part (interrelations of $\mathbf{x}_i$). The upper right part of **E** denotes the couplings of $\mathbf{x}_z$ and $\mathbf{x}_d$ to $\mathbf{x}_i$.

9.3.1.1 Causal analysis to determine form of equations

The form of equations can be determined from a bond graph using causal analysis. This means that the equations need *not* be derived first.

Furthermore, some insight on possible wrongly modeled phenomena can be obtained during the causal analysis. The place in procedure where a conflict appears or the bond graph gets completely causally augmented, can give insight in the correctness of the model. The decision whether to adapt the model or not, is left to the modeler.

First, we recall the causality assignment algorithm:

1a. Chose a fixed causality of a *source* element, assign its causality, and propagate this assignment through the graph using the causal constraints. Go on until all sources have their causalities assigned.

1b. Chose a not yet causal port with *fixed* causality (non-invertable equations), assign its causality, and propagate this assignment through the graph using the causal constraints. Go on until all ports with fixed causality have their causalities assigned.

2. Chose a not yet causal port with *preferred* causality (storage elements), assign its causality, and propagate this assignment through the graph using the causal constraints. Go on until all ports with preferred causality have their causalities assigned.

3. Chose a not yet causal port with *indifferent* causality, assign its causality, and propagate this assignment through the graph using the causal constraints. Go on until all ports with indifferent causality have their causalities assigned.

No causal conflict

Often, the bond graph is completely causal after *step 2*, *without* any causal conflict (all causal conditions are satisfied). Each storage element represents a state variable, and the set of equations is an explicit set of ordinary

differential equations (not necessarily linear or time invariant). So, the equations are of form (16) or for the linear case of form (19).

Complete after 1a

When the bond graph is completely causal after *step 1a*, the model does *not* have any dynamics. The behavior of all variables now is determined by the fixed causalities of the sources. The equations are *only* algebraic equations, *no* differential equations!

Step 1 conflict: Ill-defined model

Arises a causal conflict at *step 1a* or at *step 1b*, then the problem is ill posed. The model must be changed, by adding some elements. An example of a causal conflict at *step 1a* is two effort sources connected to one 0-junction. Both sources ‘want’ to determine the one effort variable.

At a conflict at *step 1b*, a possible adjustment is changing the equations of the fixed–causality element such that these equations become *invertable*, and thus the *fixed*ness of the constraint disappears. An example is a diode or a valve having zero current resp. flow while blocking. Allowing a small resistance during blocking, the equations become invertable.

Step 2 conflict: dependent storage elements

When a conflict arises at *step 2*, a storage element receives a *non*-–preferred causality. This means that this storage element does *not* represent a *state variable*. The initial value of this storage element cannot be chosen freely. Such a storage element often is called a *dependent storage element*. This indicates that a storage element was *not* taken into account during modeling, which should be there from physical systems viewpoint. It can be deliberately omitted, or it might be forgotten. The equations are of form (17) or for the linear case of form (20).

Step 3 ‘conflict’: algebraic loops

When *step 3* of the causality algorithm is necessary, a so—called *algebraic loop* is present in the graph. This loop causes the resulting set differential equations to be *implicit*. Often this is an indication that a storage element was not modeled, which should be there from a physical systems viewpoint. The equations are also of form (17) or for the linear case of form (20).

Concluding, by interpreting the result of causal analysis, one can determine what form the equations will have, *without* deriving equations.

9.3.2 Algebraic loops (Zero-order Causal Paths)

To determine the exact form of equations, i.e. how easy can the implicit equations be simulated by ‘normal’ implicit numerical integration methods, a further detail of the form of implicit model equations is needed. To do so, algebraic loops are further classified and for each class the resulting equation structure is determined.

Zero-order Causal Path ZCP

A *Zero-order Causal Path* (ZCP) is a causal path in which the number of integrations is equal to the number of differentiations. So, the order of the path is zero. Consequently, the loop gain is a pure algebraic relation, thus causing an implicit equation. Note that a causal path is equivalent to a signal loop. The two most important and most frequently occurring ZCPs are algebraic loops and loops between a dependent and an independent storage element. Besides these two kinds, there are 3 other kinds, having an increasing complexity and resulting in more complex equations. These occur for instance in rigid–body mechanical systems. More info on ZCPs is in (Dijk, J. van and Breedveld, 1991) and (Dijk, J. van 1994).From these texts the examples are copied.

First, the 5 different classes of ZCPs are discussed, of which class 1 en class 2 are the most common ones. After that, some modeling insight that can be concluded from these ZCPs is discussed.

Dependent storage elements (C – C)

$\mathbf{x}_d$: dependent state variable

A causal path between two ports of storage elements, of which one has integral causality and the other differential causality. The port of the storage element with differential causality is the *dependent storage element*, and its state variable is a *dependent state variable*. The choice is free which of the two storage elements becomes dependent. It is the signal loop (causal path) that is important. Now, we can specify equation (17) to (21) to obtain the specific form for this kind of ZCP. The output equations **g** (18) stays the same. Examples are given in Figure 9-4.

$$\begin{aligned}\dot{\mathbf{x}}_i &= \mathbf{f}_i(\mathbf{x}_i, \dot{\mathbf{x}}_d, \mathbf{u}, t)\\ \mathbf{x}_d &= \mathbf{f}_d(\mathbf{x}_i, \mathbf{u}, t)\end{aligned} \tag{21}$$

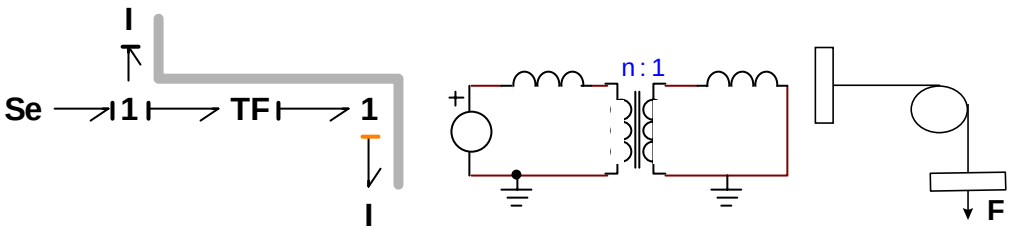

Figure 9-4: Example Class-1 ZCP (dependent storage elements)

Algebraic Loop, classical (R – R)

$\mathbf{x}_z$: algebraic loop breaker

A causal path between to ports with indifferent causality; the formulas of the elements are pure algebraic. It is a causal path, and thus a signal loop, between two resistors. This is the classical algebraic loop, because this type of ZCP was recognized in the early days of bond graphs. To break this loop, we can use one of the variables in the loop as x_z, i.e. an iteration variable in the iteration procedure of the simulator to solve this implicit equation. X_z is a part of the state vector. Now, we can specify equation (17) to (22) to obtain the specific form for this kind of ZCP. The output equations **g** (18) stays the same. Examples are given in Figure 9-5.

$$\begin{aligned}\dot{\mathbf{x}}_i &= \mathbf{f}_i(\mathbf{x}_i, \mathbf{x}_z, \mathbf{u}, t)\\ \mathbf{x}_z &= \mathbf{f}_z(\mathbf{x}_i, \mathbf{x}_z, \mathbf{u}, t)\end{aligned} \tag{22}$$

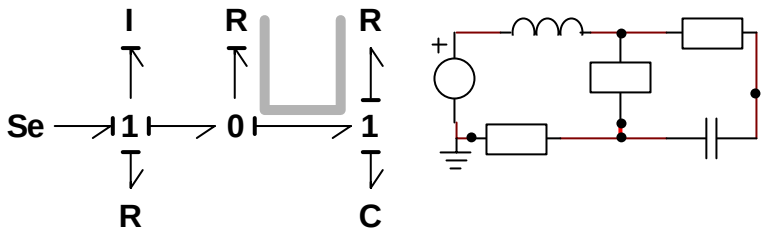

Figure 9-5: Example Class-2 ZCP (classical algebraic loop)

Causal mesh

The causal path between two indifferent causality ports is a so-called *causal mesh*. The signal loop is now open, which means of some bonds in the path, only *one* variable is involved in the signal loop. The resulting state equations have the same form as those of the class-2 ZCP, equation (22). An example is shown in Figure 9-6.

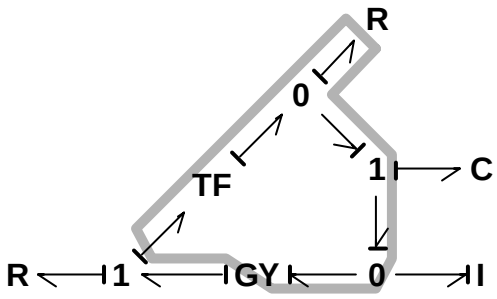

Figure 9-6: Example Class-3 ZCP (causal mesh)

Causal bond loop (class 4 and 5)
The causal path has the form of a bond loop. This implies that only TF, GY and junctions are involved, or equivalent own-made submodels. These causal path result in *two* signal loops, running in opposite directions. If the loop gain of the signal loops are unequal to ± 1, the causal path is a class-4 ZCP. Otherwise – loop gain equals ± 1 – it is a class-5 ZCP. The form of equations is again as in (22). An example is shown in Figure 9-7.

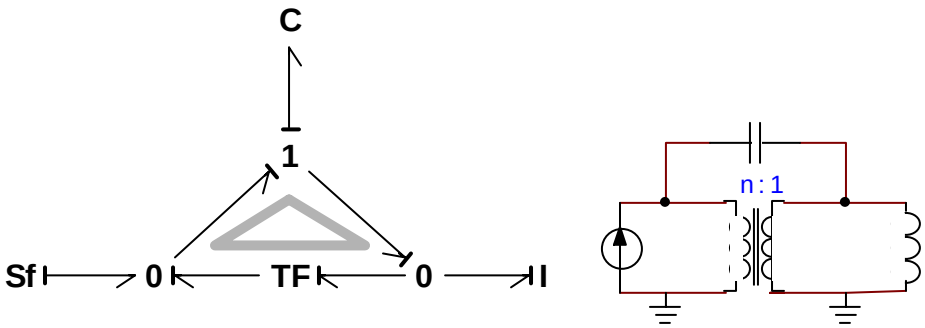

Figure 9-7: Example of Class 4/5 ZCP

The occurence of class-5 ZCPs can be prevented by assigning causalities in the right order. The standard causality assignment procedure needs to be adapted. This will not be treated here.

Touching ZCPs
Combinations of ZCPs to complex signal loops also occurs in physical systems. ZCPs of different classes are entwined. Here, two regurarly occuring examples are shown.

Touching class-1 and class-2 ZCP
The ZCP is a combination of two storage elements in a path (class-1) and a classical algebraic loop (class-2). The form of equations is as in (23). An example is shown in Figure 9-9.

$$\begin{aligned} \dot{\mathbf{x}}_\mathbf{i} &= \mathbf{f}_\mathbf{i}(\mathbf{x}_\mathbf{i}, \mathbf{x}_\mathbf{z}, \dot{\mathbf{x}}_\mathbf{d}, \mathbf{u}, t) \\ \mathbf{x}_\mathbf{z} &= \mathbf{f}_\mathbf{z}(\mathbf{x}_\mathbf{i}, \mathbf{x}_\mathbf{z}, \mathbf{u}, t) \\ \mathbf{x}_\mathbf{d} &= \mathbf{f}_\mathbf{d}(\mathbf{x}_\mathbf{i}, \mathbf{u}, t) \end{aligned} \tag{23}$$

Touching class-1 and class-4 ZCP
Here, the ZCP is a combination of two storage elements in a path (class-1) and a causal bond loop (class–4). The resulting model equations are *not* simulatable with a standard implicit numerical integration method such as BDF. The form of equations is as in (24). An example is shown in Figure 7-8.

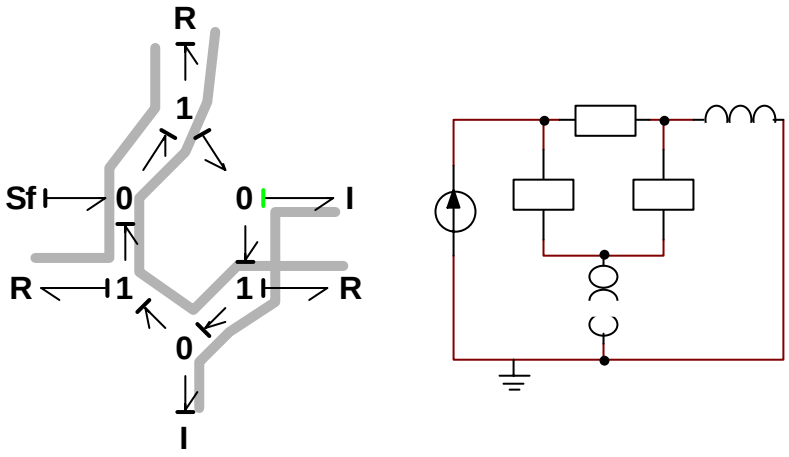

Figure 9-9: Example of touching class-1 and class-2 ZCP

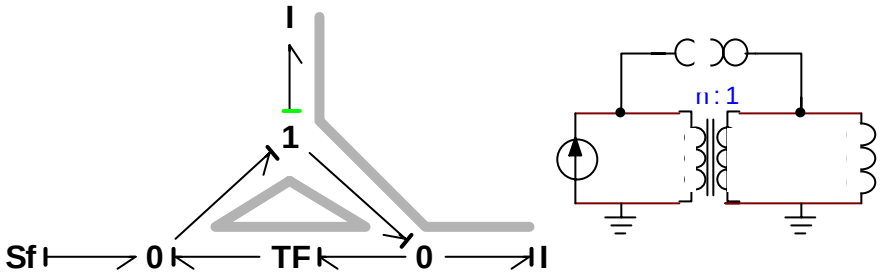

Figure 9-8: Example of touching class-1 and class-4 ZCP

$$\begin{aligned} \dot{\mathbf{x}}_i &= \mathbf{f}_i(\mathbf{x}_i, \mathbf{x}_z, \dot{\mathbf{x}}_d, \mathbf{u}, t) \\ \mathbf{x}_z &= \mathbf{f}_z(\mathbf{x}_i, \mathbf{x}_z, \dot{\mathbf{x}}_d, \mathbf{u}, t) \\ \mathbf{x}_d &= \mathbf{f}_d(\mathbf{x}_i, \mathbf{x}_z, \mathbf{u}, t) \end{aligned} \tag{24}$$

9.3.2.1 Model insight from ZCPs

Besides how the structure of the equations will be, also some insight in the modeling process, i.e. quality of the model can be obtained:

- **Class-1 or class-2 ZCPs**
  As indicated earlier, these ZCPs indicate that some physical phenomenon was not modeled, which should be there from a physical systems viewpoint. Either the phenomenon was forgotten and can be still included, or it was deliberately omitted. In the latter case, the model can stay unchanged, thus needing implicit integration methods for simulation or the model can be adapted (see also section 9.4).
- **Closed bond loop (class 3, 4, 5 ZCP)**
  The model contains a kinematic loop, or an electrical circuit with floating reference voltage.
- **Not modeled part (class 3, 4 ZCP)**
  Compare to the occurrence of class-1 or class-2 ZCPs. For simulation this class of ZCPs is no problem. However, a review of the model seems wise (cf. section 9.4).

- **Class-5 ZCP**
  A global causal constraint has *not* been taken into account. In principle, this is due to a weakness in the standard causality assignment procedure. In 20-SIM, this weakness has been repaired. See Van Dijk and Breedveld, 1991)

9.3.3 Parameter values

Eigen values λ

Parameter values really influence the characteristics of the model. The time constants and eigenfrequencies of a model are a function of the parameters. Using *eigenvalues* $\lambda_i(t)$, we can indicate these dependencies very compact. Since, in principle, non-linear models are concerned, the eigenvalues are not constant. They are the eigenvalues of the linearized model, the eigenvalues of the Jacobian $\mathbf{J} = \delta\mathbf{f}/\delta\mathbf{x}$, where $\mathbf{f}()$ are the model equations. The eigenvalues are computed by solving the following equation, for an explicit model:

$$\det(\mathbf{J} - \lambda_i \mathbf{I}) = 0$$

If the model is implicit, then the following equation needs to be solved:

$$\det(\mathbf{J} - \lambda_i \frac{d\mathbf{f}}{d\dot{\mathbf{x}}}) = 0$$

The amount of eigenvalues is equal to the rank of **J** and is also equal to the size of **x**. Eigenvalues are in principle complex numbers. For a stable system, the real parts of λ are negative, $\mathrm{Re}(\lambda_i)<0$.

The eigenvalues indicate whether a certain integration method can be used for the model at hand. We have to determine if the model is *numerical stiff* or contains *undamped components*.

9.3.3.1 Numerical Stiffness

Numerical stiffness can be determined by computing the *stiffness ratio, S*::

$$\frac{\mathrm{Max}_i \left|\mathrm{Re}\,\lambda_i(t)\right|}{\mathrm{Min}_i \left|\mathrm{Re}\,\lambda_i(t)\right|} = S(t) >> 1 \qquad (25)$$

The stiffness ratio is a measure of how far the real parts of the eigenvalues are apart from each other. Stiffness ratios of real-world problems have the order of magnitude of about 10^6, and arise in areas such as chemical engineering, control engineering, robotics, and electric circuits. For explicit methods, such as forward Euler, or Runge-Kutta, it turns out the time step h_k must be chosen very small to avoid instability. The product of the eigenvalue and the time step, $\lambda_i h_k$, should be in both the accuracy region as well as the stability region of the numerical integration method (See Figure 9-10).

9.3.3.2 Undamped components

An eigenvalue λ_i represents an oscillatory component (i.e. undamped component) if the imaginary part of λ_i is large with respect to its real part (see also Figure 9-10):

$$\frac{\left|\mathrm{Im}\,\lambda_i(t)\right|}{\left|\mathrm{Re}\,\lambda_i(t)\right|} >> 1 \qquad (26)$$

This eigenvalue is located close to the imaginary axis, of course in the left half plane. This means that this eigenvalue is almost undamped. This

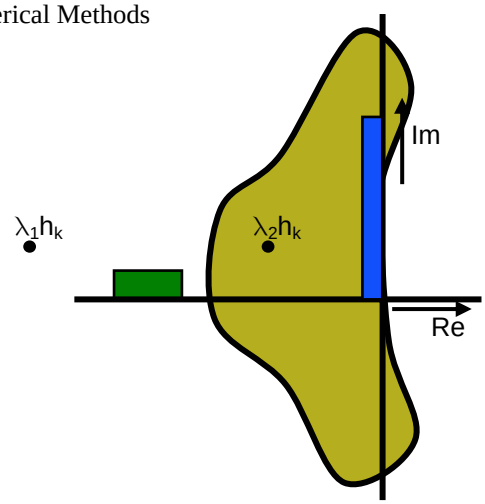

Figure 9-10Complex λh plane

location of λ_i restricts the chose of a numerical integration method, because many integration methods are inaccurate or unstable close to the imaginary axis. Euler methods and methods for stiff systems, like BDF, cannot be used for models containing undamped parts.

9.3.3.3 Discontinuities

Discontinuities in the model **f**() or the excitation signals **u** cause so-called *events* to occur, which must be treated in a special way during simulation. The normal numerical methods expect continuous differentiations of the state vector x (so, **f**() is supposed to be continuous). Fixed-step methods (h_k is constant) large errors can occur, depending on the chosen size of h_k. At variable-step methods, the simulation will 'creep' to the discontinuity: h_k becomes smaller and smaller while reaching the discontinuity, which often causes unnecessary much computation time.

Event

An *event* is defined *as an instantaneous change to another mode of continuous behavior*. A *mode of continuous behavior* is that part of the state space of **x** where the model equations **f**() are continuous. A *mode* is a kind of *logical state* of the system. Consequently, an event is *a logical state transition*. In stead of mode, sometimes the term mode switch is used.

Events are classified as follows:

1. Time Event
The discontinuity happens at equidistant time stamps, and are known on beforehand. Examples are the sampling of signals (A/D-conversion) in a physical system that is controlled by a digital computer. Another example is a time-discrete system.

2. State event
The discontinuity happens when a certain variable passes a threshold. The moment of happening is *not* known on beforehand. The discontinuity causes a change in the model, where even the number of state variables can change. It is quite obvious that this effect needs special treatment at a simulation. Examples are collision, grab or release of a load by a robot or crane, an ideal switch.

Another case of non-continuous model equations is *piecewise continuous functions*. These can often efficiently be described by a linear interpolation table, sometimes called *look-up table*. An example is a vessel with fluid, of which the content cannot become negative. These discontinuities do *not* need a special treatment during simulation. However, depending on which part of

the piecewise continuous function is used, the eigenvalue can change abruptly.

Note that from a physical point of view, there are *no* discontinuities, at least at macroscopic level (we restrict ourselves to the macroscopic view). This means that events are abstractions, chosen by the modeler. The behavior in the neighbourhood of the discontinuity is now described as an instantaneous change of physical variables. Of course, a more precise model without discontinuities can always be made. But, then it might *not* be competent! An example is two bouncing balls, where the compression of the balls is neglected. The complete collision takes place in *one* time step. The transfer of the impulse from one ball to the other is now specified in one formula.

Deciding whether certain behavior will be modeled as a *state event*, depends on the next two arguments:

- The physical (thus continuous) change of the variables involved is *not* important for the model: the time scale on which the event takes place is much smaller than the time scale of the effects we are interested in.
- The dynamic behavior at the logical state change is such that the behavior we are interested in is not influenced by this choice. In other words, we can freely use state events.

The simulation of state events needs special treatment: First, the state event must be *detected*, it is not known on beforehand when the effect will take place. For the detection, a so-called *constraint function* has to be defined, which detects a state event (Figure 9-11) while crossing zero. After that, the event must be *localized*, since in general events will occur between two simulation steps. Now the logical change can be performed, which is actually nothing more than begin a new continuous simulation again. The initial values need to be determined out of the values of the states before the event. This is a part of the model, and thus, the modeler has to build it. Furthermore, the integration method has to be re-initialized, because previous info on the simulation is not applicable anymore.

In most modern sophisticated modeling and simulation packages, these facilities are available. 20-SIM also supports this.

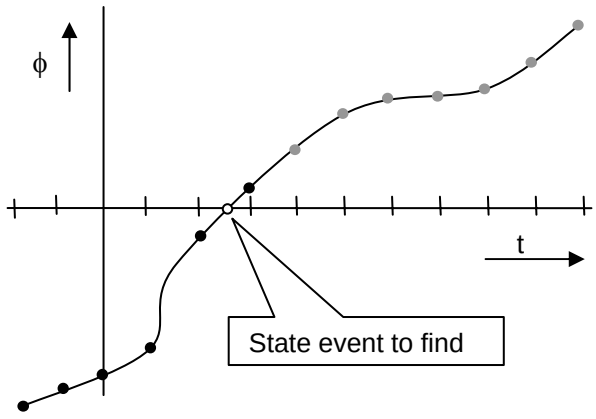

Figure 9-11: Constraint function φ for a state event

9.4 Adapt the model or numerical method

In general, different ways to handle the causal conflicts arising at step 2 or step 3 are possible:

1. Add elements.
   For example, you can withdraw the decision to neglect certain elements. The added elements can be parasitic, for example, to add elasticity (C—element) in a mechanical connection, which was modeled as rigid. Additionally adding a damping element (R) reduces the simulation time considerably, which is being advised.

2. Change the bond graph such that the conflict disappears.
   For a step 2 conflict, the dependent storage element is taken together with an independent storage element, having integral causality. For a step 3 discrepancy, sometimes resistive elements can be taken together to eliminate the conflict. This can be performed via transformations in the graph. The complexity of this operation depends on the size and kind of submodels along the route between the storage elements or resistors under concern.
   Also, in a block-diagram part, the loop can be '*cut*' by adding a one-timestep delay, which is a rather pragmatic solution, but can be useful, when no implicit integration methods are available. The accuracy might get too low. Note that the amplification of all elements in the loop must be smaller than one to obtain a stable solution.

3. The bond graph is *not* changed and during simulation, a special integration routine is needed. The implicit equations are solved by means of the iteration schemes present in the implicit integration methods, or added to the standard method. In 20-SIM, the BDF method is the only implicit method. At the other methods, the iteration scheme is added. Thus, with *all* integration methods, implicit models can be simulated.

4. The model (part) is modeled completely different, and the interface should contain power ports, to be able to connect it to other submodels.

References

Atkinson, K.E. (1989), *An introduction to Numerical Analysis*, John Wiley & Sons, 0-471-62489-6.

Dijk, J.v. (1994), *On the role of bond graph causality in modelling mechatronic systems*, PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands.

Dijk, J.v. and P.C. Breedveld (1991), Simulation of System Models containing zero-order causal paths - part I: Classification of zero-order causal paths, *J. Franklin Institute*, **329,** (5/6), pp. 959-979, ISSN:

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall,

Hairer, E. and G. Wanner (1996), *Solving Ordinary Differential Equations II*, Springer, 3-540-60452-9.

Appendix: Stability regions and Distortion diagrams

Drawings are taken from Watanabe and Himmelblau (1982): Analysis of trajectory errors in integrating ordinary differential equations, Journal of the Franklin Institute, Vol. 314, No 5, pp. 283-321, Nov 1982

First, the relation between the wave form of a signal and the positions of the poles in the complex plane.

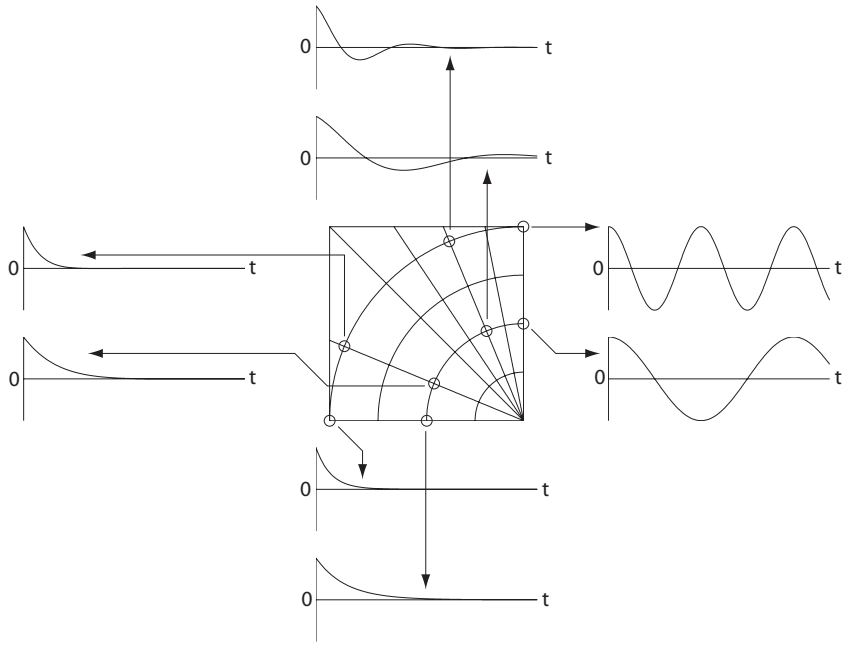

Figure 9-12 Waveforms related to poles.

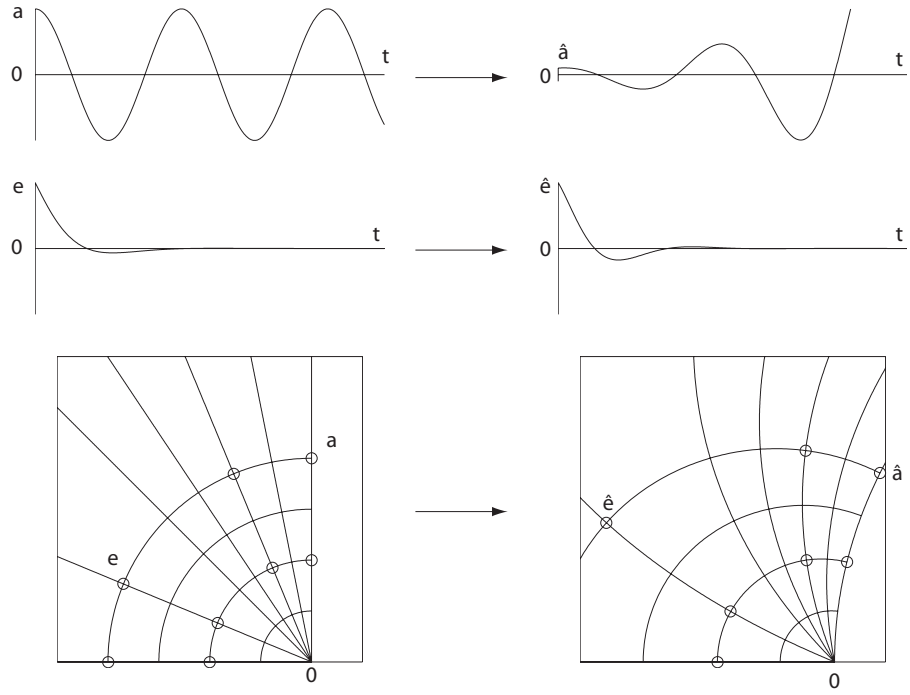

Figure 9-13 Deformation of a eigenvalue (pole) as result of numerical integration