2

Basic Principles of Numerical Integration

Preview

In this chapter, we shall discuss some basic ideas behind the algorithms that are used to numerically solve sets of ordinary differential equations specified by means of a state–space model. Following a brief introduction into the concept of numerical extrapolation that is at the heart of all numerical integration techniques, and after analyzing the types of numerical errors that all these algorithms are destined to exhibit, the two most basic algorithms, Forward Euler (FE) and Backward Euler (BE), are introduced, and the fundamental differences between explicit and implicit integration schemes are demonstrated by means of these two algorithms.

The reader is then introduced to the concept of numerical stability as opposed to analytical stability. The numerical stability domain is introduced as a tool to characterize an integration algorithm, and a general procedure to find the numerical stability domain of any integration scheme is presented. The numerical stability domain of an integration method is a convenient tool to assess some of its most important numerical characteristics.

2.1 Introduction

Given a state–space model of the form:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2.1}$$

where $\mathbf{x}$ is the *state vector*, $\mathbf{u}$ is the *input vector*, and t represents time, with a set of initial conditions:

$$\mathbf{x}(t = t_0) = \mathbf{x_0} \tag{2.2}$$

Let $x_i(t)$ represent the i^{th} state trajectory as a function of simulated time, t. As long as the state–space model does not contain any discontinuities in either $f_i(\mathbf{x}, \mathbf{u}, t)$ or any of its derivatives, $x_i(t)$ is itself a continuous function of time. Such function can be approximated with any desired precision by a Taylor–Series expansion about any given point along its trajectory, as long as the function does not exhibit a finite escape time, i.e., approaches

infinity for any finite value of time. Let t^* denote the point in time, about which we wish to approximate the trajectory using a Taylor Series, and let t^*+h be the point in time, at which we wish to evaluate the approximation. The value of the trajectory at that point can then be given as follows:

$$x_i(t^* + h) = x_i(t^*) + \frac{dx_i(t^*)}{dt} \cdot h + \frac{d^2x_i(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \dots \tag{2.3}$$

Plugging the state–space model into (2.3), we find:

$$x_i(t^* + h) = x_i(t^*) + f_i(t^*) \cdot h + \frac{df_i(t^*)}{dt} \cdot \frac{h^2}{2!} + \dots \tag{2.4}$$

Different integration algorithms vary in how they approximate the higher state derivatives, and in the number of terms of the Taylor–Series expansion that they consider in the approximation.

2.2 The Approximation Accuracy

Evidently, the accuracy with which the higher order derivatives are approximated should match the number of terms of the Series that are considered. If $n+1$ terms of the Taylor Series are considered, the approximation accuracy of the second state derivative $d^2x_i(t^*)/dt^2 = df_i(t^*)/dt$ should be of order $n-2$, since this factor is multiplied with h^2. The accuracy of the third state derivative should be of order $n-3$, since this factor is multiplied with h^3, etc. In this way, the approximation is correct up to h^n. n is therefore called the *approximation order* of the integration method, or, more simply, the integration method is said to be of n^{th} order.

The approximation error that is made because of the truncation of the Taylor Series after a finite number of terms is called *truncation error*. The truncation error contains terms in h^{n+1}, h^{n+2}, etc. It does not contain any terms in powers of h smaller than $n+1$. However, since the magnitude of the remaining terms usually decreases rapidly with increasing powers of h, the truncation error itself is often approximated by a single term, namely the term in h^{n+1}. In order to be able to assess the accuracy of the numerical integration, it is essential to be aware of this term. Therefore, many numerical integration codes actually estimate this term, and use this information for such purposes as step–size control.

The higher the approximation order of a method, the more accurate will be the estimation of $x_i(t^* + h)$. Consequently, when using a high–order method, we can afford to integrate with a large step size. On the other hand, the smaller the step size that we employ, the faster decreases the importance of the higher–order terms in the Taylor Series, and therefore, when using a small step size, we can afford to truncate the Taylor Series early.

The cost of integrating a state–space model across a single integration step depends heavily on the order of the method in use. High–order algorithms are much more expensive than low–order methods in this respect. However, this cost may be offset by the fact that we can use a much larger step size, and therefore require a considerably smaller overall number of integration steps to complete the simulation run. We have therefore always a choice between employing a low–order algorithm with a small step size integrating the system over many such steps, or using a high–order algorithm with a large step size integrating the system over much fewer steps.

Which of these choices is more economical in a given situation, depends on various factors. However, for now, the following simple rule of thumb may be used as an often quite decent indicator [2.4]:

> If the local relative accuracy required by an application, i.e., the largest error tolerated within a single integration step, is 10^{-n}, then it is best to choose at least a n^{th} order algorithm for the numerical integration.

For this reason, the simulation of problems from celestial dynamics requires the highest–order algorithms. We usually apply eighth–order algorithms to such problems. On the other hand, most simulations of economic systems call for first– or second–order methods, since the parameters of the models themselves are not more accurate than that. It makes no sense whatsoever to waste a superb integration algorithm on a garbage model. Garbage integrated with high precision still remains garbage.

Many engineering simulation applications require a global relative accuracy of approximately 0.001. We usually make the following assumption:

> If the local integration error, i.e., the error made during a single integration step, is proportional to h^{n+1}, then the global integration error, i.e., the error of the results at the end of the simulation run, is proportional to h^n.

This assumption is correct for a sufficiently small step size, h, i.e., in the so–called *asymptotic region* of the algorithm.

The above heuristic can be justified by the following observation. If the *local integration error* is of size e_ℓ, then the *per–unit–step integration error* assumes a value of $e_{p.u.s} = e_\ell/h$. The *global integration error* is proportional to the per–unit–step integration error, as long as the integration error does not accumulate excessively across multiple steps.

A global relative error of 0.001, as required by most engineering applications, calls for an algorithm with an approximation order of h^3 for the global error. In accordance with the previously made observation, this corresponds to an algorithm with an approximation order of h^4 for the local integration error. Therefore, we should require a local accuracy of 0.0001. This means that a fourth–order algorithm is about optimal, and, since engineers are

the most highly valued customers of continuous–system simulation software designers, this is what most such simulation software systems offer as their default integration algorithm, i.e., as the algorithm that is used by the system if the user doesn't specify explicitly, which technique he or she wishes to be employed.

A second type of approximation error to be looked at is caused by the finite word length of the computer, on which the simulation is performed. On a digital computer, real numbers can only be represented with a finite precision. This type of error is called the *roundoff error*. The roundoff error is important since, in numerical integration, invariably very small numbers are added to very large numbers.

For example, let us assume we employ a third-order algorithm:

$$x(t^* + h) \approx x(t^*) + f(t^*) \cdot h + \frac{df(t^*)}{dt} \cdot \frac{h^2}{2!} + \frac{d^2 f(t^*)}{dt^2} \cdot \frac{h^3}{3!} \tag{2.5}$$

to integrate a scalar state–space model:

$$\dot{x} = f(x, u, t) \tag{2.6}$$

across one second of simulated time. Let us assume for simplicity that the magnitude of x and its first three time derivatives is in the order of 1.0, thus:

$$\|x\| \approx \|f\| \approx \|\dot{f}\| \approx \|\ddot{f}\| \approx 1.0 \tag{2.7}$$

Let us further assume that a constant step size of $h = 0.001$ is employed throughout the simulation. The simulation is performed on a computer with a word length of 32 bits in single precision. Such machines usually offer a mantissa length of 24 bits, and an exponent of eight bits. On such a machine, the roundoff error is approximately:

$$\varepsilon_{\text{roundoff}} = 2^{-24} \approx 10^{-6} \tag{2.8}$$

Thus, a real number in single precision carries approximately six significant decimals. Applying this information to the process of numerical integration, we find:

$$\begin{aligned} \|x(t^* + h)\| \approx & \|x(t^*)\| + \|f(t^*) \cdot h\| + \|\frac{df(t^*)}{dt} \cdot \frac{h^2}{2!}\| + \|\frac{d^2 f(t^*)}{dt^2} \cdot \frac{h^3}{3!}\| \\ \approx & 1.0 + 0.001 + 10^{-6} + 10^{-9} \end{aligned} \tag{2.9}$$

Thus, while the constant term contributes six significant digits to the result of the addition, already the linear term contributes only three digits to the result, and the second–order term does not contribute anything of significance at all. We might just as well never have computed it in the first place. This fact is illustrated in Fig.2.1.

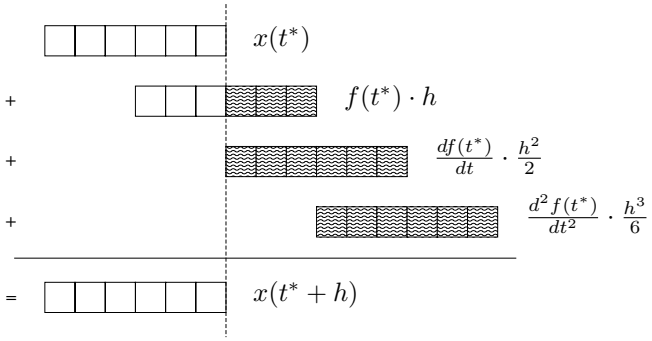

FIGURE 2.1. Effects of roundoff on numerical integration.

Consequently, using single precision on a 32 bit machine for numerical integration algorithms of order higher than two may be quite problematic. In reality, the effects of shiftout will not necessarily be as dramatic as shown in the above example, since higher–order algorithms allow use of a larger step size. Yet, double precision algorithms will be definitely more robust due to their reduced risk of shiftout, and they can meanwhile be implemented quite efficiently also. Therefore, there is no good reason anymore to use single precision on any integration algorithm but Euler. In this context, it is interesting to notice that many commercially available simulation software systems, such as ACSL [2.20], use a single–precision fourth–order variable–step Runge–Kutta algorithm as their default integration method, integrating happily –and dangerously– along.

A double precision representation will take care of roundoff errors as shown in Fig.2.2. A double precision representation on a 32 bit machine provides about 14 significant digits, since double precision words usually offer a 52 bit mantissa and a 12 bit exponent on such a machine.

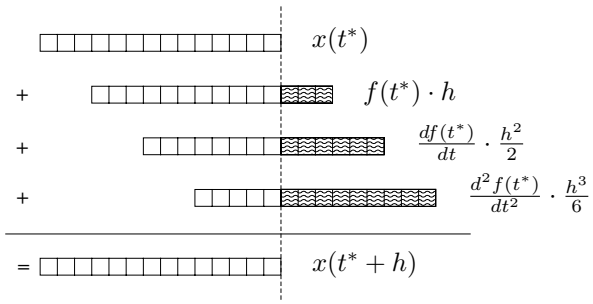

FIGURE 2.2. Roundoff in double precision.

Unfortunately, double precision operations are more time–consuming and therefore more expensive than single precision operations. This may still cause a problem especially in real–time applications. For this reason, Korn

and Wait introduced the concept of 1.5–fold precision [2.18]. The idea behind 1.5–fold precision is illustrated in Fig.2.3.

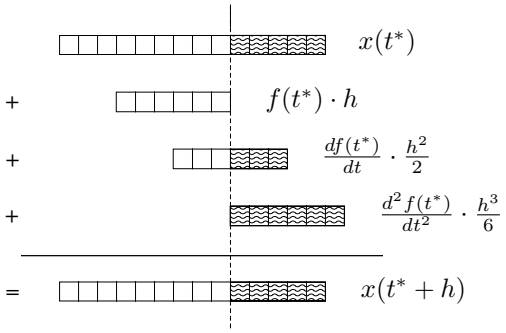

FIGURE 2.3. Roundoff in 1.5–fold precision.

It may not be necessary to store *all* real numbers in double precision. It may suffice to store only the state vector itself in double precision. In particular, this makes it possible to evaluate the nonlinear state–space model in single precision. Thereby, some of the accuracy of the state vector is compromised. The lost digits are shaded in Fig.2.3. However, these errors will not migrate to the left, i.e., a sufficiently large number of digits remains significant. The accuracy of the state vector is indeed roughly half way between that of single precision and that of double precision, but the overall price of the computation is closer to that of single precision.

Simulations of celestial dynamics problems should be performed in full double precision on a 64 bit machine, or, if only 32 bit machines are available, full four–fold precision should be used. It hardly ever makes sense to employ an algorithm of order higher than eight, since otherwise, the roundoff errors will dominate over the truncation errors even on the highest precision machines available.

For most engineering problems, double precision on a 32 bit machine is sufficient. With the advent of modern high–speed personal computers, producers of simulation software became less concerned with execution speed and more concerned with accuracy. For this reason, MATLAB, and with it also SIMULINK, perform routinely all numerical computations in double precision. Hence the roundoff error is today of a lesser concern than it used to be in the past.

A third type of error to be discussed is the *accumulation error*. Due to roundoff and truncation, $x(t^* + h)$ cannot be known precisely. This error will be inherited by the next integration step as an error in its initial conditions. Thus, errors accumulate when numerical integration proceeds over many steps. Fortunately, it can be observed that the effects of the initial conditions will eventually die out in the analytical solution of an analytically stable system. Consequently, it can be expected that a numerically

stable numerical integration (we shall present a proper definition of this term in due course) will dampen out the effects of initial conditions as well, and will thereby, as a side effect, also get rid of inaccuracies in the initial conditions.

This is very fortunate, since it indicates that errors in initial conditions of an integration step don't usually affect the overall simulation too much. However, this assumption holds only for analytically stable systems. This is the reason why numerical integration algorithms have a tendency to stall when confronted with analytically unstable systems even before any trajectory of the analytical solution has grown alarmingly large. While simulating an analytically unstable system, it can no longer be assumed that the global integration error is proportional to the per–unit–step integration error, since the integration error can accumulate excessively across multiple steps. In such a case, it may be better to start from the end, and integrate the system backward through simulated time.

On top of all these errors, the simulation practitioner is confronted with inaccuracies of the model itself. These can be decomposed into *parametric model errors*, i.e., errors that reflect inaccurately estimated model parameters, and *structural model errors*, i.e., unmodeled dynamics.

To summarize the above, the modeler and the simulation practitioner must deal with five different types of errors. Modeling errors can be subdivided into structural and parametric errors. The modeler must verify that his model reflects reality sufficiently well for the purpose of the study at hand. This process is commonly referred to as *model validation.* Techniques for model validation are discussed in detail in the companion book to this text: *Continuous System Modeling* [2.5]. Once it has been asserted that the model reflects reality sufficiently well, the simulation practitioner must now verify that the numerical trajectories obtained by means of a numerical simulation of the model decently replicate the analytical trajectories that would result if the model were computed with infinite precision. This process is referred to as *simulation verification.* Simulation verification plays a central role in this textbook. Simulation errors can be classified into truncation errors, roundoff errors, and accumulation errors. It is the conglomerate of all of these errors that makes the life of an applied mathematician interesting indeed.

2.3 Euler Integration

Let us now look at some actual numerical integration algorithms.

The simplest integration algorithm is obtained by truncating the Taylor Series after the linear term.

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \dot{\mathbf{x}}(t^*) \cdot h \tag{2.10a}$$

or:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^*), t^*) \cdot h \tag{2.10b}$$

It is obviously possible to write the integration algorithm in vector form, i.e., the entire state vector can be integrated in parallel. The above scheme is particularly simple, since it doesn't require the approximation of any higher–order derivatives. The linear term is readily available from the state–space model. This integration scheme is called *Forward Euler* algorithm, and will, from now on, be abbreviated as FE algorithm. Figure 2.4 depicts graphically how the FE integration method approximates a state trajectory.

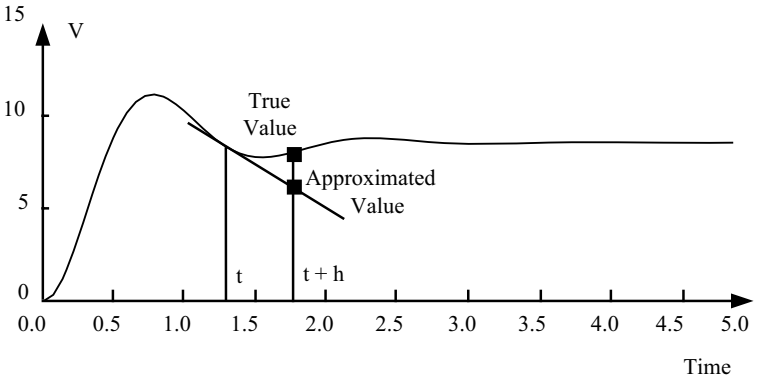

FIGURE 2.4. Numerical integration using Forward Euler.

Simulation using the FE algorithm is straightforward. Since the initial conditions, $\mathbf{x}(t = t_0) = \mathbf{x_0}$ are given, we can proceed as follows:

$$\begin{array}{lll}
\text{step 1a:} & \dot{\mathbf{x}}(t_0) & = \mathbf{f}(\mathbf{x}(t_0), t_0) \\
\text{step 1b:} & \mathbf{x}(t_0 + h) & = \mathbf{x}(t_0) \; + \; h \cdot \dot{\mathbf{x}}(t_0) \\
 & & \\
\text{step 2a:} & \dot{\mathbf{x}}(t_0 + h) & = \mathbf{f}(\mathbf{x}(t_0 + h), t_0 + h) \\
\text{step 2b:} & \mathbf{x}(t_0 + 2h) & = \mathbf{x}(t_0 + h) \; + \; h \cdot \dot{\mathbf{x}}(t_0 + h) \\
 & & \\
\text{step 3a:} & \dot{\mathbf{x}}(t_0 + 2h) & = \mathbf{f}(\mathbf{x}(t_0 + 2h), t_0 + 2h) \\
\text{step 3b:} & \mathbf{x}(t_0 + 3h) & = \mathbf{x}(t_0 + 2h) \; + \; h \cdot \dot{\mathbf{x}}(t_0 + 2h)
\end{array}$$

etc.

Simulation becomes a straightforward and quite procedural matter, since the numerical integration algorithm depends only on past values of state variables and state derivatives. An integration scheme that exhibits this property is called *explicit integration algorithm*. Most integration algorithms employed in today's general–purpose continuous–system simulation

languages, such as ACSL [2.20], are of this nature. However, this statement does not hold for special–purpose simulation software, such as electric circuit simulators.

Let us now introduce a different integration algorithm. Figure 2.5 depicts a slightly modified scheme.

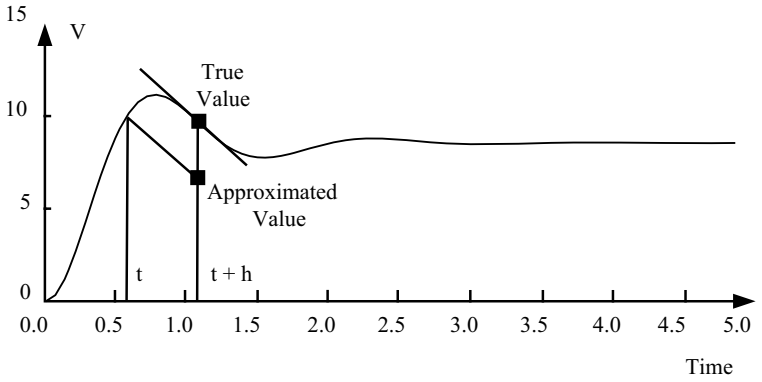

FIGURE 2.5. Numerical integration using Backward Euler.

In this scheme, the solution $\mathbf{x}(t^* + h)$ is approximated using the values of $\mathbf{x}(t^*)$ and $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h)$ using the formula:

$$\mathbf{x}(t^* + h) \approx \mathbf{x}(t^*) + \mathbf{f}(\mathbf{x}(t^* + h), t^* + h) \cdot h \tag{2.11}$$

This scheme is commonly referred to as the *Backward Euler* integration rule. It will, from now on, be abbreviated as BE algorithm.

As can be seen, this integration formula depends on current as well as past values of variables. This fact causes problems. In order to compute $\mathbf{x}(t^* + h)$ from Eq.(2.11), we need to know $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h))$, however, in order to compute $\mathbf{f}(\mathbf{x}(t^* + h), t^* + h))$ from Eq.(2.1), we need to know $\mathbf{x}(t^* + h)$. Thus, we are confronted with a nonlinear *algebraic loop*. Algorithms that are of this type are referred to as *implicit integration techniques*. The integration algorithms that are employed in electronic circuit simulators, such as PSpice [2.21], are of this type. Although implicit integration techniques are advantageous from a numerical point of view (as we shall learn later), the additional computational load created by the necessity to solve simultaneously a set of nonlinear algebraic equations at least once every integration step may make them undesirable for use in general–purpose simulation software except for specific applications, such as stiff systems.

2.4 The Domain of Numerical Stability

Let us now turn to the solution of autonomous, time–invariant linear systems of the type:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{2.12}$$

with initial conditions as specified in Eq.(2.2). The solution of such a system can be analytically given:

$$\mathbf{x}(t) = \exp(\mathbf{A} \cdot t) \cdot \mathbf{x_0} \tag{2.13}$$

The solution is called *analytically stable* if all trajectories remain bounded as time goes to infinity. The system of Eq.(2.12) is analytically stable if and only if all eigenvalues of $\mathbf{A}$ have negative real parts:

$$\mathbb{R}\mathrm{e}\{\mathrm{Eig}(\mathbf{A})\} = \mathbb{R}\mathrm{e}\{\lambda\} < 0.0 \tag{2.14}$$

The domain of analytical stability in the complex λ–plane is shown in Fig.2.6.

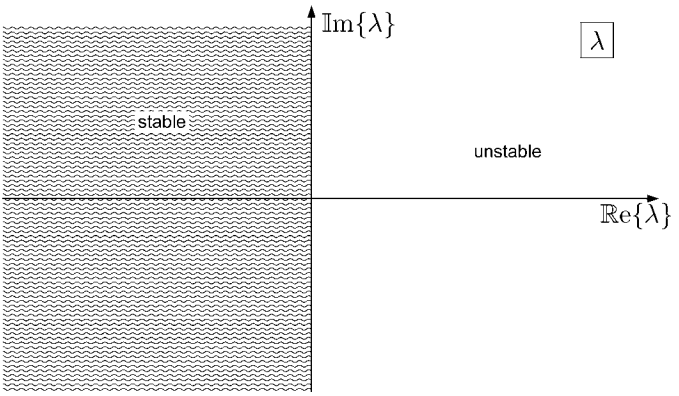

FIGURE 2.6. Domain of analytical stability.

Let us now apply the FE algorithm to the numerical solution of this problem. Plugging the system of Eq.(2.12) into the algorithm of Eq.(2.10), we obtain:

$$\mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^*) \tag{2.15}$$

which can be written in a more compact form as:

$$\mathbf{x}(k+1) = [\mathbf{I^{(n)}} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \tag{2.16}$$

$\mathbf{I^{(n)}}$ is an identity matrix of the same dimensions as $\mathbf{A}$, i.e., $n \times n$. Instead of referring to the simulation time explicitly, we simply index the time, i.e., k refers to the k^{th} integration step.

By plugging the state equations into the integration algorithm, we have converted the former continuous–time system into an "equivalent" discrete–time system:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{2.17}$$

where the discrete state matrix, $\mathbf{F}$, can be computed from the continuous state matrix, $\mathbf{A}$, and the step size, h, as:

$$\mathbf{F} = \mathbf{I^{(n)}} + \mathbf{A} \cdot h \tag{2.18}$$

The term "equivalence" is defined in the sense of the employed numerical integration algorithm. It does not mean that the converted discrete–time system behaves identically to the original continuous–time system. The two systems are "equivalent" in the same sense as the numerical trajectory is "equivalent" to its analytical counterpart.

For simplicity, we shall consistently employ the following notation in this book:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \tag{2.19a}$$

$$\mathbf{y} = \mathbf{C} \cdot \mathbf{x} + \mathbf{D} \cdot \mathbf{u} \tag{2.19b}$$

denotes the continuous–time linear system, where $\mathbf{A}$ is the state matrix, $\mathbf{B}$ is the input matrix, $\mathbf{C}$ is the output matrix, and $\mathbf{D}$ is the input/output matrix. $\mathbf{x}$ denotes the state vector. It is of length n ($\mathbf{x} \in \mathbb{R}^n$). $\mathbf{u}$ is the input vector, $\mathbf{u} \in \mathbb{R}^m$, and $\mathbf{y}$ is the output vector, $\mathbf{y} \in \mathbb{R}^p$.

The equivalent discrete–time linear system is written as:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} + \mathbf{G} \cdot \mathbf{u_k} \tag{2.20a}$$

$$\mathbf{y_k} = \mathbf{H} \cdot \mathbf{x_k} + \mathbf{I} \cdot \mathbf{u_k} \tag{2.20b}$$

where $\mathbf{F}$ now denotes the state matrix, $\mathbf{G}$ is the input matrix, $\mathbf{H}$ is the output matrix, and $\mathbf{I}$ is the input/output matrix.

The discrete–time system of Eq.(2.17) is analytically stable if and only if all its eigenvalues are located inside a circle of radius 1.0 about the origin, the so–called *unit circle*. From Eq.(2.18), we can conclude that all eigenvalues of $\mathbf{A}$ multiplied by the step size, h, must lie inside a circle of radius 1.0 about the point -1.0.

We define that the linear time–invariant continuous–time system integrated using a given fixed–step integration algorithm is *numerically stable* if and only if the "equivalent" linear time–invariant discrete–time system

(the term equivalence meant in the sense of the same integration algorithm) is analytically stable.

Figure 2.7 shows the domain of numerical stability of the FE algorithm.

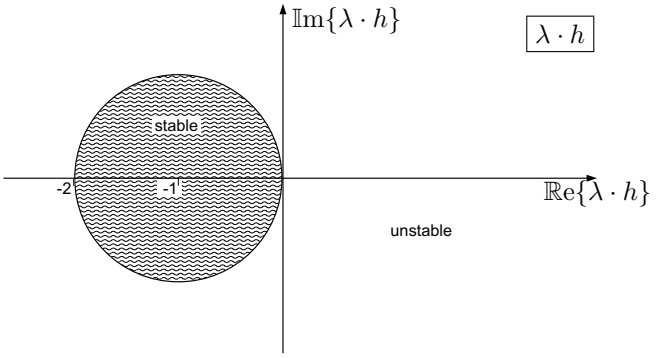

FIGURE 2.7. Domain of numerical stability of Forward Euler.

Notice that the numerical stability domain is, in a rigorous sense, only defined for linear time–invariant continuous–time systems, and applies only to fixed–step algorithms. Nevertheless, it is appealing that the numerical stability domain of an integration algorithm can be computed and drawn once and for all, and does not depend on any system properties other than the location of its eigenvalues.

The numerical stability domain of the FE algorithm tries to approximate the analytical stability domain, but evidently does a quite poor job at that.

Let us now try the following experiment. We simulate the scalar system $\dot{x} = a \cdot x$ with initial condition $x_0 = 1.0$ and a fixed step size of $h = 1.0$ over ten steps, i.e., from time $t = 0.0$ to time $t = 10.0$ using the FE algorithm. We repeat the experiment four times with different values of the parameter a. The results of this experiment are shown in Fig.2.8 The solid lines represent the analytical solutions, whereas the dashed lines represent the numerically found solutions. In the first case with $a = -0.1$, there exists a good correspondence between the two solutions. In the second case with $a = -1.0$, the numerical solution is still stable but bears little resemblance with the analytical solution, i.e., is very inaccurate. In the third case with $a = -2.0$, the numerical solution is marginally stable, and in the fourth case with $a = -3.0$, the numerical solution is unstable.

This result is in agreement with the numerical stability domain shown in Fig.2.7. We would have had to multiply the eigenvalue $\lambda = a = -3.0$ with a step size of $h = 2/3$, in order to obtain an even marginally stable solution, i.e., in order to get the eigenvalue into the stable region of the $\lambda \cdot h$–plane. In order to obtain an accurate result, a considerably smaller step size would have been needed. A 10% integration accuracy requires a step size of approximately $h = 0.1$ when applied to the system with $a = -3.0$, a 1%

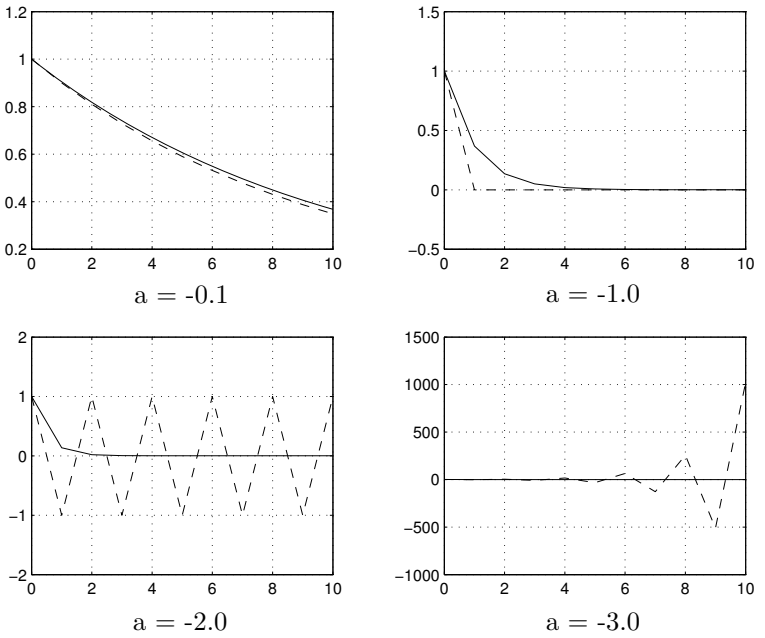

FIGURE 2.8. Numerical experiment using Forward Euler.

accuracy forces us to reduce the step size to $h = 0.01$, and a 0.1% accuracy calls for a step size of $h = 0.001$. In this case, we need already 10,000 steps to integrate this trivial system across 10 seconds. Quite obviously, the FE algorithm is not suitable if such high an accuracy is desired.

Moreover, the above experiment tested the FE algorithm on a very benign example. Systems with pairs of conjugate complex stable eigenvalues close to the imaginary axis are much worse. This fact is demonstrated in Fig.2.9.

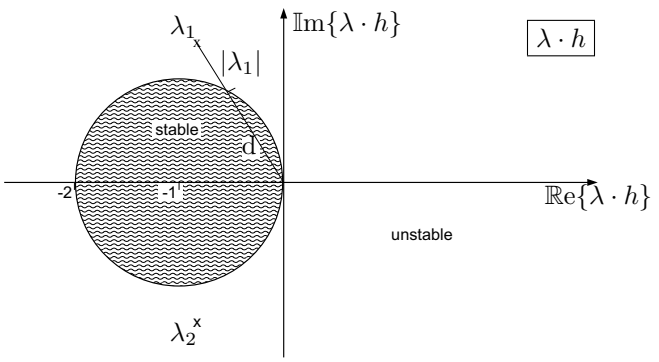

FIGURE 2.9. Determination of maximum step size with Forward Euler.

The location of the eigenvalues of the λ–plane are superimposed on the stability domain of the $\lambda \cdot h$–plane. A maximum step size of:

$$h_{max} = \frac{d}{|\lambda_1|} \tag{2.21}$$

must be used in order to guarantee a numerically stable solution. In the case, where the eigenvalues are on the imaginary axis itself, no step size can be found that will make the numerical solution exhibit the true undamped oscillation. The FE algorithm is not at all suited to integrate such models. Systems with their dominant eigenvalues on or close to the imaginary axis are quite common. They are either highly oscillatory systems with very little damping, or hyperbolic partial differential equation (PDE) systems converted to ordinary differential equation (ODE) form by means of the method–of–lines approximation.

Let us now look at the BE algorithm. We shall plug the state–space model of Eq.(2.12) into the algorithm of Eq.(2.10). We obtain:

$$\mathbf{x}(t^* + h) = \mathbf{x}(t^*) + \mathbf{A} \cdot h \cdot \mathbf{x}(t^* + h) \tag{2.22}$$

which can be rewritten as:

$$[\mathbf{I^{(n)}} - \mathbf{A} \cdot h] \cdot \mathbf{x}(t^* + h) = \mathbf{x}(t^*) \tag{2.23}$$

or:

$$\mathbf{x}(k+1) = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1} \cdot \mathbf{x}(k) \tag{2.24}$$

Thus:

$$\mathbf{F} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1} \tag{2.25}$$

Figure 2.10 shows the stability domain of this technique.

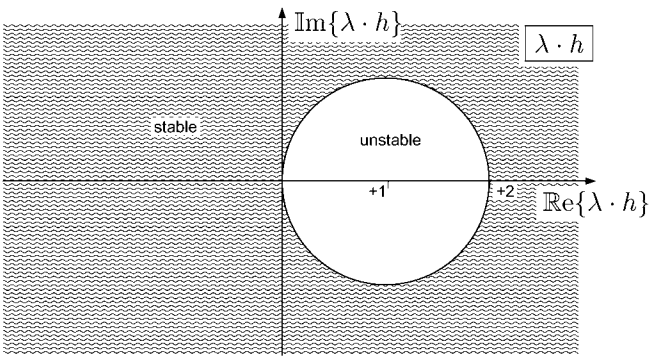

FIGURE 2.10. Stability domain of Backward Euler.

As in the case of the FE algorithm, BE tries to approximate the analytical stability domain, and does an equally poor job.

Let us repeat our previous experiment, this time with values of $a = -3.0$, and $a = +3.0$. It is of interest to us to simulate the system also in an analytically unstable configuration. Figure 2.11 shows the results of our efforts.

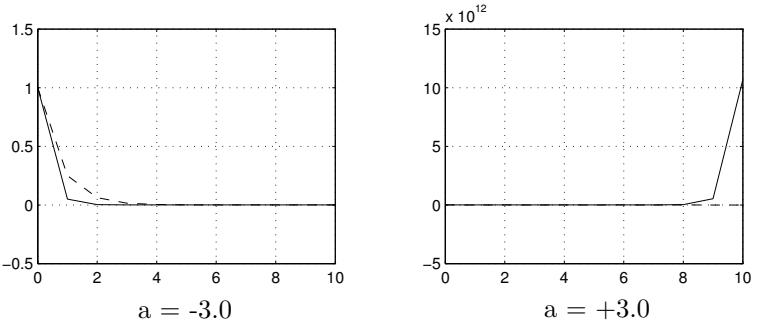

FIGURE 2.11. Numerical experiment using Backward Euler.

The results could have been predicted easily from the stability domain shown in Fig.2.10.

The BE algorithm does a fairly decent job on the problem with $a = -3.0$. The results are not very accurate with $h = 1.0$, but, at least, they bear some resemblance with reality. This type of algorithm is therefore better suited than the FE type to solve problems with eigenvalues far out on the negative real axis of the λ–plane. Systems with eigenvalues whose real parts are widespread along the negative real axis are called *stiff systems*. Stiff systems are quite common. In particular, they often result from converting parabolic PDEs to sets of ODEs using the method–of–lines approximation. Contrary to the situation when the FE algorithm is used, the step size will, in the case of the BE algorithm, be dictated solely by *accuracy requirements* of the system, and not by the *numerical stability domain* of the method.

The problem with $a = +3.0$ reveals yet a different type of problem. The analytical solution is unstable, but the numerical simulation suggests that the system is perfectly stable. This can be quite dangerous. Imagine that a nuclear reactor has been designed and simulated using the BE algorithm. The simulation makes the engineers believe that everything is fine, but in reality, the reactor will blow up on the first occasion. Traditionally, researchers have focused more on the simulation of analytically stable systems, and therefore, many simulation practitioners aren't fully aware of the dangers that might result from using implicit algorithms, such as BE, to simulate systems that are potentially unstable in the analytical sense.

The lesson to be learnt is the following: When it really matters, it may be a good idea to simulate the system twice, once with an algorithm that

exhibits a stability domain comparable to that of the FE algorithm, and once with an algorithm that behaves like the BE algorithm. If both simulations produce similar trajectories, the engineer may assume that the results are true to the model, though not necessarily to the physical plant. This is the most valuable simulation verification technique that exists, and the importance of this recommendation cannot be overestimated.

As in the case of the FE algorithm, BE has not much luck with marginally stable systems, i.e., with systems whose dominant eigenvalues are located on the imaginary axis. As before, no step size will predict the undamped oscillation of the true system.

How has the stability domain for the BE algorithm been found? Although there exist analytical techniques to determine the domain of numerical stability, they are somewhat cumbersome and error prone. Therefore, we prefer to go another route and devise a general–purpose computer program that can determine the domain of numerical stability of any integration algorithm.

We start out with a scalar problem with $|\lambda| = 1.0$, i.e., with its eigenvalue anywhere along the unit circle. In order to avoid complex numbers, we may alternatively use a second–order system with a complex conjugate pair of eigenvalues on the unit circle.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & 2\cos(\alpha) \end{pmatrix} \tag{2.26}$$

is a matrix with a pair of conjugate complex eigenvalues on the unit circle, where α denotes the angle of one of the two eigenvalues counted in the mathematically positive (i.e., counterclockwise) sense away from the positive real axis.

The following MATLAB routine computes $\mathbf{A}$ for any given value of α.

```
function [A] = aa (alpha)
   radalpha = alpha * pi/180;
   x = cos(radalpha);
   A = [ 0 , 1 ; -1 , 2 * x ];
return
```

We then compute the $\mathbf{F}$–matrix for this system. The *ff*–function accepts the $\mathbf{A}$–matrix, the step size, h, and a number representing the integration algorithm, *algor*, as input arguments, and returns the respective $\mathbf{F}$–matrix as output argument. The routine is here only shown with the code for the first two algorithms, the FE and BE algorithms.

```
function [F] = ff(A, h, algor)
   Ah = A * h;
   [n, n] = size(Ah);
   I = eye(n);
   %
   % algor = 1 : Forward Euler
   %
   if algor == 1,
      F = I + Ah;
   end
   %
   % algor = 2 : Backward Euler
   %
   if algor == 2,
      F = inv(I - Ah);
   end
return
```

Now, we compute the largest possible value of h, for which all eigenvalues of $\mathbf{F}$ are inside the unit circle. The *hh*–function calls upon the *aa*– and *ff*–functions internally. It accepts α and *algor* as input arguments. It also requires lower and upper bounds for the step size, h_{lower} and h_{upper}, such that the solution of the discretized problem is stable for one of them, and unstable for the other. The function returns the value of the step size, h_{max}, for which the discretized problem is marginally stable.

```
function [hmax] = hh(alpha, algor, hlower, hupper)
   A = aa(alpha);
   maxerr = 1.0e-6;
   err = 100;
   while err > maxerr,
      h = (hlower + hupper)/2;
      F = ff(A, h, algor);
      lmax = max(abs(eig(F)));
      err = lmax - 1;
      if err > 0,
         hupper = h;
      else
         hlower = h;
      end,
      err = abs(err);
   end
   hmax = h;
return
```

The *hh*–function, as shown above, works only for algorithms with stability domains similar to that of the FE algorithm. The logic of the *if*–statement must be reversed for algorithms of the BE type, but we didn't want to make the code poorly readable by including too many implementational details.

Finally, we need to sweep over a selected range of α values, and plot h_{max} as a function of α in polar coordinates. There certainly exist more efficient curve tracking algorithms than the one outlined above, but for the time being, this algorithm will suffice.

2.5 The Newton Iteration

One additional problem needs to be discussed. In the above example, it was easy to perform the simulation using the BE algorithm. Since the system to be simulated is linear, we were able to compute the **F**–matrix explicitly by means of matrix inversion.

This cannot be done in a nonlinear case. We need to somehow solve the implicit set of nonlinear algebraic equations that are formed by the state–space model and the implicit integration algorithm. To this end, we need an iteration procedure.

The first idea that comes to mind is to employ a *predictor–corrector technique*. The idea is quite simple. We start out with an explicit FE step, and use the result of that step (the predictor) for the unknown state derivative of the implicit BE step. We repeat by iterating on the BE step.

$$
\begin{array}{lll}
\text{predictor:} & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) & \\
& \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}_{\mathbf{k}} & \\
& & \\
1^{\text{st}} \text{ corrector:} & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) & \\
& \mathbf{x}^{\mathbf{C1}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} & \\
& & \\
2^{\text{nd}} \text{ corrector:} & \dot{\mathbf{x}}^{\mathbf{C1}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{C1}}_{\mathbf{k+1}}, t_{k+1}) & \\
& \mathbf{x}^{\mathbf{C2}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{C1}}_{\mathbf{k+1}} & \\
& & \\
3^{\text{rd}} \text{ corrector:} & \dot{\mathbf{x}}^{\mathbf{C2}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{C2}}_{\mathbf{k+1}}, t_{k+1}) & \\
& \mathbf{x}^{\mathbf{C3}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{C2}}_{\mathbf{k+1}} & \\
& & \\
\text{etc.} & &
\end{array}
$$

The iteration is terminated when two consecutive approximations of $\mathbf{x_{k+1}}$ differ less than a prescribed tolerance. Since the predictor step is explicit, the overall algorithm is explicit as well. This iteration scheme is called *fixed–point iteration*.

If we apply the linear system of Eq.(2.12) to this algorithm, and insert all the equations into each other, we find:

$$
\begin{aligned}
\mathbf{F^P} &= \mathbf{I^{(n)}} + \mathbf{A} \cdot h \\
\mathbf{F^{C1}} &= \mathbf{I^{(n)}} + \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 \\
\mathbf{F^{C2}} &= \mathbf{I^{(n)}} + \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 + (\mathbf{A} \cdot h)^3 \\
\mathbf{F^{C3}} &= \mathbf{I^{(n)}} + \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 + (\mathbf{A} \cdot h)^3 + (\mathbf{A} \cdot h)^4
\end{aligned}
$$

For infinitely many iterations, we obtain:

$$\mathbf{F} = \mathbf{I^{(n)}} + \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 + (\mathbf{A} \cdot h)^3 + \ldots \tag{2.27}$$

Thus:

$$(\mathbf{A} \cdot h) \cdot \mathbf{F} = \mathbf{A} \cdot h + (\mathbf{A} \cdot h)^2 + (\mathbf{A} \cdot h)^3 + (\mathbf{A} \cdot h)^4 + \ldots \tag{2.28}$$

and subtracting Eq.(2.28) from Eq.(2.27), we find:

$$[\mathbf{I^{(n)}} - \mathbf{A} \cdot h] \cdot \mathbf{F} = \mathbf{I^{(n)}} \tag{2.29}$$

or:

$$\mathbf{F} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1} \tag{2.30}$$

Thus, we are hopeful that we just found a (very expensive) explicit integration algorithm that behaves like the BE method. Unfortunately, nothing could be farther from the truth. Figure 2.12 depicts the resulting stability domain when plugging the **F**–matrix of Eq.(2.27) into the algorithm that generates stability domains.

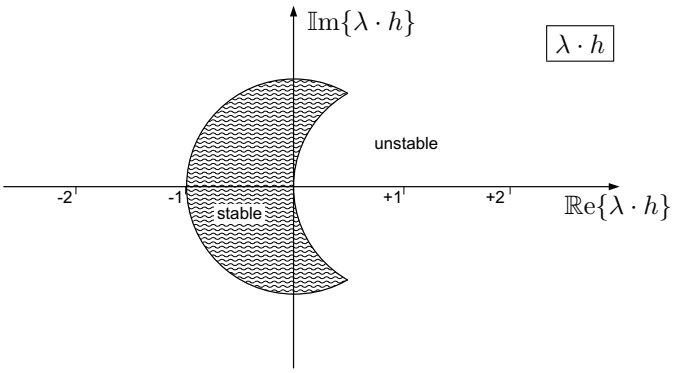

FIGURE 2.12. Stability domain of predictor–corrector FE–BE technique.

The reason for the half–moon domain obtained in this way is that the infinite series of Eq.(2.27) converges only if all eigenvalues of $\mathbf{A} \cdot h$ are inside the unit circle. The subtraction of the two infinite series of Eq.(2.27) and Eq.(2.28) is only legal if this is the case. Thus, the stability domain approaches that of BE only for sufficiently small values of $|\mathrm{Eig}(\mathbf{A} \cdot h)|$.

Let us try something else. Figure 2.13 shows how a zero–crossing of a function can be found using Newton iteration.

Given an arbitrary function $\mathcal{F}(x)$. We want to assume that we know the value of the function and its derivative $\partial\mathcal{F}/\partial x$ at some point x^ℓ. We notice

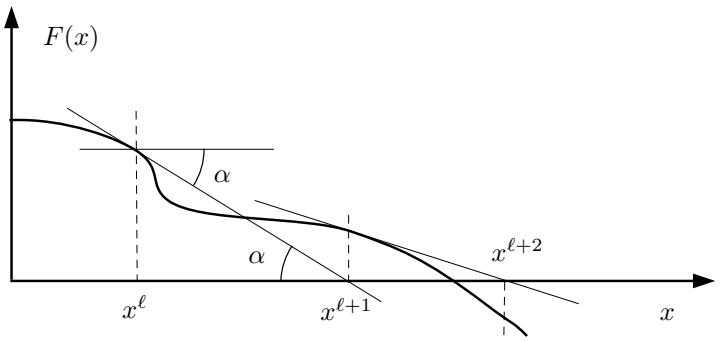

FIGURE 2.13. Newton iteration.

that:

$$\tan\ \alpha = \frac{\partial \mathcal{F}^{\ell}}{\partial x} = \frac{\mathcal{F}^{\ell}}{x^{\ell} - x^{\ell+1}} \tag{2.31}$$

Thus:

$$x^{\ell+1} = x^{\ell} - \frac{\mathcal{F}^{\ell}}{\partial \mathcal{F}^{\ell}/\partial x} \tag{2.32}$$

Let us apply this technique to the problem of iterating the nonlinear algebraic equation system at hand. Let us plug the scalar nonlinear state–space model evaluated at time t_{k+1}:

$$\dot{x}_{k+1} = f(x_{k+1}, t_{k+1}) \tag{2.33}$$

into the scalar BE algorithm:

$$x_{k+1} = x_k + h \cdot \dot{x}_{k+1} \tag{2.34}$$

We find:

$$x_{k+1} = x_k + h \cdot f(x_{k+1}, t_{k+1}) \tag{2.35}$$

or:

$$x_k + h \cdot f(x_{k+1}, t_{k+1}) - x_{k+1} = 0.0 \tag{2.36}$$

Equation (2.36) is in the desired form to apply Newton iteration. It describes a nonlinear algebraic equation in the unknown variable x_{k+1}, the zero–crossing of which we wish to determine. Thus:

$$x_{k+1}^{\ell+1} = x_{k+1}^{\ell} - \frac{x_k + h \cdot f(x_{k+1}^{\ell}, t_{k+1}) - x_{k+1}^{\ell}}{h \cdot \partial f(x_{k+1}^{\ell}, t_{k+1})/\partial x - 1.0} \tag{2.37}$$

where k is the integration step count, and ℓ is the Newton iteration count.

The matrix extension of the Newton iteration algorithm looks as follows:

$$\mathbf{x}^{\ell+\mathbf{1}} = \mathbf{x}^{\ell} - \left(\mathcal{H}^{\ell}\right)^{-1} \cdot \mathcal{F}^{\ell} \tag{2.38}$$

where:

$$\mathcal{H} = \frac{\partial \mathcal{F}}{\partial \mathbf{x}} = \begin{pmatrix} \partial \mathcal{F}_1/\partial x_1 & \partial \mathcal{F}_1/\partial x_2 & \dots & \partial \mathcal{F}_1/\partial x_n \\ \partial \mathcal{F}_2/\partial x_1 & \partial \mathcal{F}_2/\partial x_2 & \dots & \partial \mathcal{F}_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial \mathcal{F}_n/\partial x_1 & \partial \mathcal{F}_n/\partial x_2 & \dots & \partial \mathcal{F}_n/\partial x_n \end{pmatrix} \tag{2.39}$$

is the *Hessian matrix* of the iteration problem.

Applying this iteration scheme to the vector state–space model and the vector BE algorithm, we obtain:

$$\mathbf{x}_{\mathbf{k+1}}^{\ell+\mathbf{1}} = \mathbf{x}_{\mathbf{k+1}}^{\ell} - [h \cdot \mathcal{J}_{\mathbf{k+1}}^{\ell} - \mathbf{I}^{(\mathbf{n})}]^{-1} \cdot [\mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x}_{\mathbf{k+1}}^{\ell}, t_{k+1}) - \mathbf{x}_{\mathbf{k+1}}^{\ell}] \tag{2.40}$$

where:

$$\mathcal{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \dots & \partial f_1/\partial x_n \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \dots & \partial f_2/\partial x_n \\ \vdots & \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \partial f_n/\partial x_2 & \dots & \partial f_n/\partial x_n \end{pmatrix} \tag{2.41}$$

is the *Jacobian matrix* of the dynamic system.

Any implementation of this iteration scheme requires, in general, the computation of at least an approximation of the Jacobian matrix, as well as an inversion (refactorization) of the Hessian matrix. Since both operations are quite expensive, different implementations vary in how often they recompute the Jacobian (so–called modified Newton iteration). The more nonlinear the problem, the more frequently the Jacobian must be recomputed. Notice further that a modification of the step size does not require the computation of a new Jacobian, but it forces us to refactorize the Hessian.

Let us now analyze how this iteration scheme will affect the solution of linear problems, and, in particular, how it will influence the stability domain of the method.

The Jacobian of the linear state–space model is simply its state matrix:

$$\mathcal{J} = \mathbf{A} \tag{2.42}$$

Consequently, the Jacobian of a linear time–invariant model never needs to be updated, although a new inverse Hessian will still be required whenever the step size of the algorithm is modified.

Plugging the linear system into Eq.(2.40), we find:

$$\mathbf{x}_{\mathbf{k+1}}^{\ell+\mathbf{1}} = \mathbf{x}_{\mathbf{k+1}}^{\ell} - [\mathbf{A} \cdot h - \mathbf{I}^{(\mathbf{n})}]^{-1} \cdot [(\mathbf{A} \cdot h - \mathbf{I}^{(\mathbf{n})}) \cdot \mathbf{x}_{\mathbf{k+1}}^{\ell} + \mathbf{x_k}] \tag{2.43}$$

or:

$$\mathbf{x}_{\mathbf{k+1}}^{\ell+\mathbf{1}} = [\mathbf{I}^{(\mathbf{n})} - \mathbf{A} \cdot h]^{-1} \cdot \mathbf{x}_{\mathbf{k}} \tag{2.44}$$

Evidently, Newton iteration does not influence the stability properties of the linear system. This is generally true for all integration algorithms, not only when the Newton iteration is applied to the BE algorithm.

2.6 Semi–analytic Algorithms

As we have seen, numerical integration algorithms call at various places for the computation of derivatives. Time derivatives of $\mathbf{f}$ are needed for the higher–order terms of the Taylor–Series expansion. Spatial derivatives of $\mathbf{f}$ are required by the Newton iteration algorithm. More uses of derivatives will be met in due course.

However, the numerical computation of derivatives by explicit algorithms is notoriously ill–conditioned. An inaccurate evaluation of the Jacobian is relatively harmless. This will simply slow down the convergence of the Newton iteration. However, numerical errors in the higher–order terms of the Taylor Series are devastating. Therefore, numerical analysts have learnt to reformulate the problem so that a direct computation of the higher-order Taylor–Series terms can be avoided. We shall talk about this more in the following chapters of this book.

However, for now, we shall explore another avenue. The Taylor Series could easily be evaluated directly if only we had available analytical expressions for the higher derivatives. While analytical expressions for the higher derivatives can be derived fairly easily, it is painful for the user to have to manually derive those expressions. If the model is even only modestly complex, the user will probably make mistakes on the way.

However, techniques for algorithmic formulae manipulation have meanwhile been developed. In fact, this branch of computer science has been met with quite remarkable success over the past few years. Algorithmic differentiation of formulae has become a standard feature offered by many symbolic processing programs. However, many of these systems generate derivative formulae that expand, i.e., are much longer than the original formulae. This pitfall can be avoided. Joss developed a technique that avoids formulae expansion in symbolic differentiation [2.16]. The idea behind his technique is surprisingly simple. The original formulae are decomposed into primitives, each of which can be differentiated separately. The example shown below illustrates how the algorithm works in practice. Given the following function:

$$\dot{x} = \sin^2(\sqrt{x} + \frac{x^2 \cdot t}{2}) \tag{2.45}$$

Its algebraic differentiation can be computed in the following way:

$$\begin{array}{lcllcl}
\dot{x} & = & c_1^2 & \Rightarrow & \ddot{x} & = 2 \cdot c_1 \cdot \dot{c}_1 \\
c_1 & = & \sin(c_2) & \Rightarrow & \dot{c}_1 & = \cos(c_2) \cdot \dot{c}_2 \\
c_2 & = & c_3 + c_4 & \Rightarrow & \dot{c}_2 & = \dot{c}_3 + \dot{c}_4 \\
c_3 & = & \sqrt{x} & \Rightarrow & \dot{c}_3 & = \dot{x}/(2 \cdot \sqrt{x}) \\
c_4 & = & 0.5 \cdot c_5 \cdot c_6 & \Rightarrow & \dot{c}_4 & = 0.5 \cdot (c_5 \cdot \dot{c}_6 + \dot{c}_5 \cdot c_6) \\
c_5 & = & x^2 & \Rightarrow & \dot{c}_5 & = 2 \cdot x \cdot \dot{x} \\
c_6 & = & t & \Rightarrow & \dot{c}_6 & = 1.0
\end{array}$$

It can easily be verified that equations are available to compute all the unknown variables. The equations only need to be sorted into an executable sequence. The second time derivative of x is indeed being evaluated correctly. Since all possible primitive expressions can be tabulated together with their derivatives, the process of algorithmically generating derivatives is a fairly simple task. No formulae expansion takes place when differentiation is implemented in this fashion.

Joss also discovered that it is possible to compute derivatives not only of formulae, but even of entire programs. He developed an ALGOL program that can differentiate any ALGOL procedure or set of ALGOL procedures with respect to any variable or set of variables, generating new ALGOL procedures for the derivatives. Unfortunately, his dissertation was never translated into English. However, there exist newer references in English that can be consulted [2.17, 2.19, 2.22]. Kurz [2.19] used the algorithm of Joss for developing a PASCAL program that computes the derivative of any FORTRAN subroutine or set of FORTRAN subroutines with respect to any variable or set of variables, generating new FORTRAN subroutines for the derivatives. A treatise of these issues can be found in [2.11, 2.12].

In the context of simulation, symbolic differentiation was first employed by Halin [2.14, 2.15]. Halin was mostly concerned with real–time simulation, and therefore automatically generated code for a parallel multiprocessor. The run–time performance of his system was amazingly fast taking into account the primitive nature of the individual processors that he employed in his multiprocessor system. Moreover, his architecture is still valid. All that needs to be done is to replace the individual processors of his system by more modern architectures. One disadvantage of his approach to real–time simulation is that real–time simulators should be able to process external inputs, i.e., signals produced from a real plant by real–time sensors. Quite obviously, symbolic differentiation cannot find analytical expressions for time derivatives of such signals, since no formulae for the original signals are provided.

Modern modeling software, such as Dymola [2.5, 2.3], is able to choose from a rich palette of formulae manipulation algorithms when preprocessing the model in preparation of a simulation run. Algebraic differentiation is one of the tools that is being offered, and it is being used for a variety of different purposes.

This is clearly the current trend. Symbolic and numeric processing have both their strengths and weaknesses. A well–engineered combination of the two types of processing can preserve the best of both worlds, and can provide us with faster, more robust, and more user–friendly modeling and simulation environments.

2.7 Spectral Algorithms

Obviously, a Taylor–Series expansion is not the only way to approximate an analytic trajectory. Alternatively, the trajectory could be decomposed into a Fourier Series, and, at least in the case of marginally stable models, as they result from highly oscillatory systems and method–of–lines approximations to hyperbolic PDEs, this might even make a lot of sense.

Such techniques were investigated quite early by Brock and Murray [2.2]. However, at that time, no efficient techniques were known that would have allowed to generate algorithms that could compete with Taylor–Series methods in terms of run–time efficiency. However, the advent of the Fast Fourier Transform (FFT) and newly available FFT chips gave rise to a renewed interest in such techniques [2.10, 2.24]. New theoretical results were also reported by Bales *et al.* [2.1] and Tal–Ezer [2.23].

However, it is a fact that all numerical integration algorithms that are employed in today's commercially available simulation software make use of Taylor Series as a basis for their approximations, and therefore, we shall ignore other techniques in this book.

2.8 Summary

In this chapter, we have introduced the basic concepts of accuracy and stability as they relate to differential equation solvers. It turns out that, whenever we dealt with questions of *accuracy*, we were looking at nonlinear state–space models, whereas, whenever we were discussing *stability*, we were looking at linear state–space models. This is somewhat unsatisfactory. After all, accuracy is a local property of the algorithm, whereas stability is a more global facet of it. The reason for this inconsistency is simple. We dealt with accuracy in nonlinear terms, because it was easy to do, whereas we restricted our discussion of stability to the linear problem, since a general nonlinear treatise of stability issues is a very difficult subject indeed.

Linear stability considerations cannot always be extended to the nonlinear case, or, if they are, they may yield misleading answers. In fact, even linear time–variant systems may behave in surprising ways. To demonstrate this fact, let us look at the linear time–variant autonomous continuous–time system:

$$\dot{\mathbf{x}} = \mathbf{A}(t) \cdot \mathbf{x} = \begin{pmatrix} -2.5 & 1.5 \cdot \exp(8t) \\ -0.5 \cdot \exp(-8t) & -0.5 \end{pmatrix} \cdot \mathbf{x} \tag{2.46a}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 23 \\ 11 \end{pmatrix} \tag{2.46b}$$

The eigenvalues of the $\mathbf{A}$–matrix are -1.0 and -2.0, i.e., they are constant and negative real.

$$\begin{aligned} \text{Eig}(\mathbf{A}(t)) &= \text{Root}(\det(\lambda \cdot \mathbf{I^{(n)}} - \mathbf{A})) \\ &= \text{Root}((\lambda + 2.5) \cdot (\lambda + 0.5) + 0.75) = \text{Root}(\lambda^2 + 3 \cdot \lambda + 2) \end{aligned}$$

Yet, the analytical solution is:

$$x_1(t) = 5 \cdot \exp(7t) + 18 \cdot \exp(6t) \tag{2.47a}$$

$$x_2(t) = 5 \cdot \exp(-t) + 6 \cdot \exp(-2t) \tag{2.47b}$$

Evidently, $x_1(t)$ is unstable, although both eigenvalues of the system are in the left half λ–plane.

A similar discrete–time example can easily be constructed also. Let us look at the linear time–variant autonomous discrete–time system:

$$\mathbf{x_{k+1}} = \mathbf{F}(t) \cdot \mathbf{x_k} = \begin{pmatrix} -1 & 1.5 \cdot 8^k \\ -0.5 \cdot 8^{-k} & 1 \end{pmatrix} \cdot \mathbf{x_k} \tag{2.48a}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 11 \\ 5 \end{pmatrix} \tag{2.48b}$$

The eigenvalues of the $\mathbf{F}$–matrix are $+0.5$ and -0.5, i.e., they are constant and within the unit circle.

$$\begin{aligned} \text{Eig}(\mathbf{F}(t)) &= \text{Root}(\det(\lambda \cdot \mathbf{I^{(n)}} - \mathbf{F})) \\ &= \text{Root}((\lambda + 1) \cdot (\lambda - 1) + 0.75) = \text{Root}(\lambda^2 - 0.25) \end{aligned}$$

Yet, the analytical solution is:

$$x_1(k) = 2 \cdot 4^k + 9 \cdot (-4)^k \tag{2.49a}$$

$$x_2(k) = 2 \cdot 0.5^k + 3 \cdot (-0.5)^k \tag{2.49b}$$

Evidently, $x_1(k)$ is unstable, although both eigenvalues of the system are within the unit circle of the λ–plane.

This is bad news. In fact, let us assume that an analytically stable linear time–invariant continuous–time system is being integrated with an obscure

variable–step integration algorithm, whose stability region contains the entire left half $(\lambda \cdot h)$–plane (such a method is called *A–stable.*) Since the **F**–matrix is a function of the step size, h, it is entirely feasible that a sequence of h values can be chosen such that the numerical solution will blow up anyway. Such anomalies were reported in [2.6].

A general discussion of numerical stability in the nonlinear sense does exist. A major breakthrough in this research area was achieved by Dahlquist in two seminal papers published in the mid seventies [2.8, 2.7]. A mature discussion of the topic can be found in [2.9, 2.13]. The main idea behind Dahlquist's approach to nonlinear stability was to focus on a side effect of stability. In a stable system, trajectories that start out from neighboring initial conditions contract with time. Dahlquist focused on formulating conditions for when trajectories contract. Therefore, it has become customary to refer to nonlinear stability as *contractivity.* However, the theory is too involved to be dealt with in this book. The (linear) stability domain, that was introduced in this chapter, serves our purposes perfectly well, since our major goals are to help the simulation practitioner with this book to attain a feel for when which technique might have a decent chance of success, and if a technique fails to succeed, why this is, and what can be done about it.

2.9 References

[2.1] Laurence A. Bales. Cosine Methods for Second–Order Hyperbolic Equations with Time–Dependent Coefficients. *Mathematics of Computation*, 45(171):65–89, 1985.

[2.2] Paul Brock and Francis J. Murray. The Use of Exponential Sums in Step–by–Step Integration. *Math. Tables Aids Comput.*, 6:63–78, 1952.

[2.3] François E. Cellier and Hilding Elmqvist. Automated Formula Manipulation Supports Object–Oriented Continuous–System Modeling. *IEEE Control Systems*, 13(2):28–38, 1993.

[2.4] François E. Cellier and Peter J. Möbius. Toward Robust General Purpose Simulation Software. In Robert D. Skeel, editor, *Proceedings of the 1979 SIGNUM Meeting on Numerical Ordinary Differential Equations*, pages 18:1–5, Urbana, Ill., 1979. Dept. of Computer Science, University of Illinois at Urbana–Champaign.

[2.5] François E. Cellier. *Continuous System Modeling.* Springer Verlag, New York, 1991. 755p.

[2.6] Germund G. Dahlquist, Werner Liniger, and Olavi Nevanlinna. Stability of Two–Step Methods for Variable Integration Steps. *SIAM J. Numerical Analysis*, 20(5):1071–1085, 1983.

[2.7] Germund G. Dahlquist. Error Analysis for a Class of Methods for Stiff Nonlinear Initial Value Problems. In G. Alistair Watson, editor, *Proceedings* 6^{th} *Biennial Dundee Conference on Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics*, pages 60–72. Springer–Verlag, Berlin, 1975.

[2.8] Germund G. Dahlquist. On Stability and Error Analysis for Stiff Nonlinear Problems. Technical Report TRITA–NA–7508, Dept. of Information Processing, Royal Institute of Technology, Stockholm, Sweden, 1975.

[2.9] Kees Dekker and Jan G. Verwer. *Stability of Runge–Kutta Methods for Stiff Nonlinear Differential Equations*. North–Holland, Amsterdam, The Netherlands, 1984. 307p.

[2.10] David Gottlieb and Steven A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*, volume 26. SIAM Publishing, Philadelphia, Penn., 1977. 172p.

[2.11] Andreas Griewank. On Automatic Differentiation. In Masao Iri and Kunio Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Press, 1989.

[2.12] Andreas Griewank. *User's Guide for ADOL–C, Version 1.0*. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1990.

[2.13] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential–Algebraic Problems*, volume 14 of *Series in Computational Mathematics*. Springer–Verlag, Berlin, Germany, 2^{nd} edition, 1996. 632p.

[2.14] Hans Jürgen Halin, Richard Bürer, Walter Hälg, Hans Benz, Bernard Bron, Hans-Jörg Brundiers, Anders Isacson, and Milan Tadian. The ETH Multiprocessor Project: Parallel Simulation of Continuous Systems. *Simulation*, 35(4):109–123, 1980.

[2.15] Hans Jürgen Halin. The Applicability of Taylor Series Methods in Simulation. In *Proceedings 1983 Summer Computer Simulation Conference*, volume 2, pages 1032–1076, Vancouver, Canada, July 11–13, 1983. SCS Publishing, San Diego, Calif.

[2.16] Johann Joss. *Algorithmisches Differenzieren*. PhD thesis, Diss ETH 5757, Swiss Federal Institute of Technology, Zürich, Switzerland, 1976. 69p.

[2.17] Gershon Kedem. Automatic Differentiation of Computer Programs. *ACM Trans. Mathematical Software*, 6(2):150–165, 1980.

[2.18] Granino A. Korn and John V. Wait. *Digital Continuous–System Simulation*. Prentice–Hall, Englewood Cliffs, N.J., 1978. 212p.

[2.19] Eberhard Kurz. Algebraic Differential Processor. Technical report, Department of Electrical and Computer Engineering, University of Arizona, Tucson, Ariz., 1986.

[2.20] Edward E. L. Mitchell and Joseph S. Gauthier. *ACSL: Advanced Continuous Simulation Language — User Guide and Reference Manual*. Mitchell & Gauthier Assoc., Concord, Mass., 1991.

[2.21] James W. Nilsson and Susan A. Riedel. *Introduction to PSpice for Electric Circuits*. Prentice–Hall, Upper Saddle River, N.J., 6^{th} edition, 2002. 132p.

[2.22] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, 1981. 165p.

[2.23] Hillel Tal-Ezer. Spectral Methods in Time for Hyperbolic Equations. *SIAM J. Numerical Analysis*, 23(1):11–26, 1986.

[2.24] Robert Vichnevetsky and John B. Bowles. *Fourier Analysis of Numerical Approximations of Hyperbolic Equations*, volume 5 of *SIAM Studies in Applied Mathematics*. SIAM Publishing, Philadelphia, Penn., 1982. 140p.

2.10 Bibliography

[B2.1] George F. Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer–Verlag, Berlin, Germany, 2002. 459p.

[B2.2] C. William Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Series in Automatic Computation. Prentice–Hall, Englewood Cliffs, N.J., 1971. 253p.

[B2.3] Curtis F. Gerald and Patrick O. Wheatley. *Applied Numerical Analysis*. Addison–Wesley, Reading, Mass., 6^{th} edition, 1999. 768p.

[B2.4] John D. Lambert. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*. John Wiley, New York, 1991. 304p.

2.11 Homework Problems

[H2.1] Marginal Stability

Given the following linear time–invariant continuous–time system:

$$\begin{aligned}\dot{\mathbf{x}} &= \begin{pmatrix} 1250 & -25113 & -60050 & -42647 & -23999 \\ 500 & -10068 & -24057 & -17092 & -9613 \\ 250 & -5060 & -12079 & -8586 & -4826 \\ -750 & 15101 & 36086 & 25637 & 14420 \\ 250 & -4963 & -11896 & -8438 & -4756 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 5 \\ 2 \\ 1 \\ -3 \\ 1 \end{pmatrix} \cdot u \\ \mathbf{y} &= \begin{pmatrix} -1 & 26 & 59 & 43 & 23 \end{pmatrix} \cdot \mathbf{x} \end{aligned} \tag{H2.1a}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 1 & -2 & 3 & -4 & 5 \end{pmatrix}^T \tag{H2.1b}$$

Determine the step size, h_{marg}, for which FE will give marginally stable results.

Simulate the system across 10 seconds of simulated time with step input using the FE algorithm with the following step sizes: (i) $h = 0.1 \cdot h_{\text{marg}}$, (ii) $h = 0.95 \cdot h_{\text{marg}}$, (iii) $h = h_{\text{marg}}$, (iv) $h = 1.05 \cdot h_{\text{marg}}$, and (v) $h = 2 \cdot h_{\text{marg}}$. Discuss the results.

[H2.2] Integration Accuracy

For the system of Hw.[H2.1], determine the largest step size that will give you a global accuracy of 1%.

For this purpose, it is necessary to find the analytical solution of the given system. The easiest way to achieve this is to use the *spectral decomposition method*. The MATLAB statement:

$$[\mathbf{V}, \mathbf{\Lambda}] = \text{eig}(\mathbf{A}) \tag{H2.2a}$$

generates two matrices. $\mathbf{\Lambda}$ is the *eigenvalue matrix*, i.e., a diagonal matrix with the eigenvalues of $\mathbf{A}$ placed along its diagonal, and $\mathbf{V}$ is the *right modal matrix*, i.e., a matrix that consists of the right eigenvectors of $\mathbf{A}$ horizontally concatenated to each other. The i^{th} column of $\mathbf{V}$ contains the eigenvector associated with the eigenvalue located at the i^{th} diagonal element of the $\mathbf{\Lambda}$–matrix.

Apply a *similarity transformation*:

$$\xi(t) = \mathbf{T} \cdot \mathbf{x}(t) \tag{H2.2b}$$

with:

$$\mathbf{T} = \mathbf{V}^{-1} \tag{H2.2c}$$

This will put the system into diagonal form, from which the analytical solution can be read out easily.

If you don't trust the accuracy of the numerical algorithm, you can compute the transfer function of the system using:

$$\begin{aligned} \mathbf{Sys} &= \mathrm{ss}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}) && \text{(H2.2d)} \\ \mathbf{G} &= \mathrm{tf}(\mathbf{Sys}) && \text{(H2.2e)} \end{aligned}$$

The numerator and denominator polynomials of the transfer function can then be extracted by means of:

$$[\mathbf{p}, \mathbf{q}] = \mathrm{tfdata}(\mathbf{G}, 'v') \qquad \text{(H2.2f)}$$

Finally, the roots of the denominator polynomial can be found through:

$$\lambda = \mathrm{roots}(\mathbf{q}) \qquad \text{(H2.2g)}$$

You can then perform a *partial fraction expansion* on the transfer function, and read the analytical solution out by taking the *inverse Laplacian* thereof.

Simulate the original system using the FE algorithm across 10 seconds of simulated time. Repeat the simulation with different step sizes, until you obtain agreement between the analytical and the numerical solution with an accuracy of 1%:

$$\varepsilon_{\mathrm{global}} = \frac{\|\mathbf{x_{anal}} - \mathbf{x_{num}}\|}{\|\mathbf{x_{anal}}\|} \leq 0.01 \qquad \text{(H2.2h)}$$

Repeat the same experiment with the BE algorithm. Since the system is linear, you are allowed to compute the **F**–matrix using matrix inversion.

[H2.3] Method Blending

Given the following linear time–invariant continuous–time system:

$$\begin{aligned} \dot{\mathbf{x}} &= \begin{pmatrix} 0 & 1 \\ -9.01 & 0.2 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot u \\ \mathbf{y} &= \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \mathbf{x} + 2 \cdot u \end{aligned} \qquad \text{(H2.3a)}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 1 & -2 \end{pmatrix}^T \qquad \text{(H2.3b)}$$

Find the analytical solution using one of the techniques described in Hw.[H2.2]. Simulate the system across 25 seconds of simulated time using the FE algorithm. Determine the largest step size that will lead to a global accuracy of 1%. Repeat the experiment with the BE algorithm. You may compute the **F**–matrix using matrix inversion. What do you conclude?

Let us now design another algorithm. This time, we shall repeat each single integration step once with FE and once with BE, and we shall use the arithmetic mean of the two answers as the initial condition for the next step. Such an algorithm is called a *blended algorithm*. Determine again the maximum step size that will provide a 1% accuracy. Compare your results with those obtained by FE or BE alone.

[H2.4] Cyclic Method

Repeat Hw.[H2.3]. However, this time, we shall design another algorithm. Instead of using the mean value of FE and BE to continue, we shall simply toggle between one step of FE followed by one step of BE, followed by another step of FE, etc. Such an algorithm is called a *cyclic algorithm*.

Determine again the maximum step size that will provide a 1% accuracy. Compare your results with those obtained by FE or BE alone.

[H2.5] Stability Domain

For the *predictor–corrector method* of Eq.(2.27), find the stability domains if: (i) no corrector is used, (ii) one corrector is used, (iii) two correctors are used, (iv) three correctors are used, and (v) four correctors are used. Plot the five stability domains on top of each other, and discuss the results.

[H2.6] Stability Domain: Blended and Cyclic Methods

Find the stability domain for the blended method of Hw.[H2.3]. What do you conclude when comparing the stability domain of that method with those of FE and BE? How does the stability domain of the blended method explain the result of Hw.[H2.3]?

Find the stability domain for the cyclic method of Hw.[H2.4]. Instead of interpreting this method as switching to another algorithm after each step, we can think of this technique as one that is described by a single *macro–step* consisting of two *semi–steps*. Thus:

$$\mathbf{x}(k+0.5) = \mathbf{x}(k) + 0.5 \cdot h \cdot \dot{\mathbf{x}}(k) \tag{H2.6a}$$

$$\mathbf{x}(k+1) = \mathbf{x}(k+0.5) + 0.5 \cdot h \cdot \dot{\mathbf{x}}(k+1) \tag{H2.6b}$$

Don't despair, this one is tricky. What do you conclude when comparing the stability domain of that method with those of FE and BE? How does the stability domain of the cyclic method explain the result of Hw.[H2.4]?

[H2.7] Stability Domain Shaping: Blended Method

We wish to construct yet another method. It is derived from the previously discussed blended algorithm. Instead of using the mean value of the FE and BE steps, we use a weighted average of the two:

$$\mathbf{x}(k+1) = \vartheta \cdot \mathbf{x}_{\mathbf{FE}}(k+1) \; + \; (1-\vartheta) \cdot \mathbf{x}_{\mathbf{BE}}(k+1) \tag{H2.7a}$$

Such a method is called a *ϑ–method*. Plot the stability domains of these methods for:

$$\vartheta = \{0, 0.1, 0.2, 0.24, 0.249, 0.25, 0.251, 0.26, 0.3, 0.5, 0.8, 1\} \qquad \text{(H2.7b)}$$

Interpret the results. For this problem, it may be easier to use MATLAB's *contour* plot, than your own stability domain tracking routine.

[H2.8] Stability Domain Shaping: Cyclic Method

We shall now design another ϑ–method. This time, we start out with the cyclic method. The parameter that we shall vary is the step length of the two semi–steps. This is done in the following way:

$$\begin{aligned} \mathbf{x}(k+\vartheta) &= \mathbf{x}(k) + \vartheta \cdot h \cdot \dot{\mathbf{x}}(k) && \text{(H2.8a)} \\ \mathbf{x}(k+1) &= \mathbf{x}(k+\vartheta) + (1-\vartheta) \cdot h \cdot \dot{\mathbf{x}}(k+1) && \text{(H2.8b)} \end{aligned}$$

Determine the ϑ parameter of the method such that the overall method exhibits a stability domain similar to BE, but where the border of stability on the positive real axis of the $(\lambda \cdot h)$–plane is located at $+10$ instead of $+2$. Plot the stability domain of that method.

2.12 Projects

[P2.1] ϑ–Methods

For the two ϑ–methods described in Hw.[H2.7] and Hw.[H2.8], determine optimal values of the ϑ parameter as a function of the location of the eigenvalues of the $\mathbf{A}$–matrix (for linear time–invariant systems). To this end, vary the ϑ parameter until you get a maximum value of h that guarantees 1% accuracy. Repeat for different locations of the eigenvalues of $\mathbf{A}$, and come up with a recipe of how to choose ϑ for any given linear system.

[P2.2] Cyclic Methods

Do a library search on cyclic methods, and come up with a decision tree that characterizes the various cyclic methods that have been proposed.

2.13 Research

[R2.1] Simulation Verification

Study the problem of simulation verification. What techniques could a robust simulation run–time library offer to support the user in asserting the correctness of his or her simulation results?