3

Single–step Integration Methods

Preview

This chapter extends the ideas of numerical integration by means of a Taylor–Series expansion from the first–order (FE and BE) techniques to higher orders of approximation accuracy. The well–known class of explicit *Runge–Kutta techniques* is introduced by generalizing the predictor–corrector idea.

The chapter then explores special classes of single–step techniques that are well suited for the simulation of stiff systems and for that of marginally stable systems, namely the *extrapolation methods* and the *backinterpolation algorithms*. The stability domain serves as a good vehicle for analyzing the stability properties of these classes of algorithms.

We are then delving more deeply into the question of approximation accuracy. The *accuracy domain* is introduced as a simple tool to explore this issue, and the *order star* approach is subsequently introduced as a more refined and satisfying alternative.

The chapter ends with a discussion of the ideas behind *step–size control* and *order control*, and the techniques used to accomplish these in the realm of single–step algorithms.

3.1 Introduction

In Chapter 2, we have seen that predictor–corrector techniques can be used to merge explicit and implicit algorithms into more complex entities that are overall of the explicit type, while inheriting some of the desirable numerical properties of implicit algorithms.

In particular, we introduced the following predictor–corrector method:

$$\begin{array}{lll} \text{predictor:} & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) \\ & \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}_{\mathbf{k}} \\ \\ \text{corrector:} & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) \\ & \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} \end{array}$$

Let us now perform a nonlinear error analysis of this simple predictor–corrector technique. To this end, we plug all the equations into each other.

We obtain:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k} + h \cdot \mathbf{f_k}, t_k + h) \tag{3.1}$$

We wish to pursue the error analysis up to the quadratic term. Let us thus develop the expression $\mathbf{f}(\mathbf{x_k} + h \cdot \mathbf{f_k}, t_k + h)$ into a multidimensional Taylor Series around the point $< \mathbf{x_k}, t_k >$. Since this term in Eq.3.1 is multiplied by h, we may truncate the Taylor Series after the linear term.

Remember that:

$$f(x + \Delta x, y + \Delta y) \approx f(x, y) + \frac{\partial f(x, y)}{\partial x} \cdot \Delta x + \frac{\partial f(x, y)}{\partial y} \cdot \Delta y \tag{3.2}$$

Thus:

$$\mathbf{f}(\mathbf{x_k} + h \cdot \mathbf{f_k}, t_k + h) \approx \mathbf{f}(\mathbf{x_k}, t_k) + \frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial \mathbf{x}} \cdot (h \cdot \mathbf{f_k}) + \frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial t} \cdot h \tag{3.3}$$

where $\partial \mathbf{f} / \partial \mathbf{x}$ is the meanwhile well–known *Jacobian* of the system. Plugging Eq.(3.3) into Eq.(3.1), we find:

$$\mathbf{x_{k+1}} \approx \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k}, t_k) + h^2 \cdot (\frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial \mathbf{x}} \cdot \mathbf{f_k} + \frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial t}) \tag{3.4}$$

Let us compare this with the true Taylor Series of $\mathbf{x_{k+1}}$ truncated after the quadratic term:

$$\mathbf{x_{k+1}} \approx \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k}, t_k) + \frac{h^2}{2} \cdot \dot{\mathbf{f}}(\mathbf{x_k}, t_k) \tag{3.5}$$

where:

$$\dot{\mathbf{f}}(\mathbf{x_k}, t_k) = \frac{d\mathbf{f}(\mathbf{x_k}, t_k)}{dt} = \frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial \mathbf{x}} \cdot \frac{d\mathbf{x_k}}{dt} + \frac{\partial \mathbf{f}(\mathbf{x_k}, t_k)}{\partial t} \tag{3.6}$$

and:

$$\frac{d\mathbf{x_k}}{dt} = \dot{\mathbf{x}}_\mathbf{k} = \mathbf{f_k} \tag{3.7}$$

Comparing the true Taylor–Series expansion of $\mathbf{x_{k+1}}$ with the results obtained from the predictor–corrector method, we find that we almost got a match. Only the factor 2 in the denominator of the quadratic term is missing. Thus, the predictor–corrector technique can be written as:

$$\mathbf{x_{PC}}(k+1) \approx \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{x_k}, t_k) + h^2 \cdot \dot{\mathbf{f}}(\mathbf{x_k}, t_k) \tag{3.8}$$

We notice at once that a simple blending of FE and PC will give us a method that is second order accurate:

$$\mathbf{x}(k+1) = 0.5 \cdot (\mathbf{x_{PC}}(k+1) + \mathbf{x_{FE}}(k+1)) \tag{3.9}$$

or, in other words:

$$\begin{array}{lll} \text{predictor:} & \dot{\mathbf{x}}_{\mathbf{k}} & = \mathbf{f}(\mathbf{x_k}, t_k) \\ & \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} & = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k}} \\ & & \\ \text{corrector:} & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} & = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) \\ & \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} & = \mathbf{x_k} + 0.5 \cdot h \cdot (\dot{\mathbf{x}}_{\mathbf{k}} + \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}}) \end{array}$$

which is *Heun's method*. This method is sometimes also referred to under the name *modified Euler method*.

In the following section, we want to generalize the idea behind Heun's method by parameterizing the search strategy for higher–order algorithms of this kind.

3.2 Runge–Kutta Algorithms

Heun's method uses an FE step as a predictor, and then a blend of an FE and a BE step as a corrector. Let us generalize this idea somewhat:

$$\begin{array}{lll} \text{predictor:} & \dot{\mathbf{x}}_{\mathbf{k}} & = \mathbf{f}(\mathbf{x_k}, t_k) \\ & \mathbf{x}^{\mathbf{P}} & = \mathbf{x_k} + h \cdot \beta_{11} \cdot \dot{\mathbf{x}}_{\mathbf{k}} \\ & & \\ \text{corrector:} & \dot{\mathbf{x}}^{\mathbf{P}} & = \mathbf{f}(\mathbf{x}^{\mathbf{P}}, t_k + \alpha_1 \cdot h) \\ & \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} & = \mathbf{x_k} + h \cdot (\beta_{21} \cdot \dot{\mathbf{x}}_{\mathbf{k}} + \beta_{22} \cdot \dot{\mathbf{x}}^{\mathbf{P}}) \end{array}$$

This set of methods contains four different parameters. The β_{ij} parameters are weighting factors of the various state derivatives that are computed during the step, and the α_1 parameter specifies the time instant at which the first stage of the technique is evaluated.

Plugging the parameterized equations into each other and developing functions that are not evaluated at time t_k into Taylor Series, we obtain:

$$\mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} = \mathbf{x_k} + h \cdot (\beta_{21} + \beta_{22}) \cdot \mathbf{f_k} + \frac{h^2}{2} \cdot [2 \cdot \beta_{11} \cdot \beta_{22} \cdot \frac{\partial \mathbf{f_k}}{\partial \mathbf{x}} \cdot \mathbf{f_k} + 2 \cdot \alpha_1 \cdot \beta_{22} \cdot \frac{\partial \mathbf{f_k}}{\partial t}] \tag{3.10}$$

The Taylor Series of $\mathbf{x_{k+1}}$ truncated after the quadratic term can be written as:

$$\mathbf{x_{k+1}} \approx \mathbf{x_k} + h \cdot \mathbf{f_k} + \frac{h^2}{2} \cdot [\frac{\partial \mathbf{f_k}}{\partial \mathbf{x}} \cdot \mathbf{f_k} + \frac{\partial \mathbf{f_k}}{\partial t}] \tag{3.11}$$

A comparison of Eq.(3.10) and Eq.(3.11) yields three nonlinear equations in the four unknown parameters:

$$\beta_{21} + \beta_{22} = 1 \tag{3.12a}$$
$$2 \cdot \alpha_1 \cdot \beta_{22} = 1 \tag{3.12b}$$
$$2 \cdot \beta_{11} \cdot \beta_{22} = 1 \tag{3.12c}$$

Thus, there exist infinitely many such algorithms. Clearly, Heun's method belongs to this set of algorithms. Heun's method can be characterized by:

$$\alpha = \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \quad \beta = \begin{pmatrix} 1 & 0 \\ 0.5 & 0.5 \end{pmatrix} \tag{3.13}$$

α_2 characterizes the time when the corrector is evaluated, which obviously always happens at t_{k+1}, thus, $\alpha_2 = 1.0$. β is a lower triangular matrix.

Many references represent the method in a slightly different form:

0	0	0
1	1	0
x	1/2	1/2

which is called the *Butcher tableau* of the method. The first row of the Butcher tableau here indicates the function evaluation at time t_k. The second row represents the predictor, and the third row denotes the corrector.

Another commonly used algorithm of this family of methods is characterized by the following α–vector and β–matrix:

$$\alpha = \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}; \quad \beta = \begin{pmatrix} 0.5 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.14}$$

with the Butcher tableau:

0	0	0
1/2	1/2	0
x	0	1

This method is sometimes referred to as *explicit midpoint rule*. It can be implemented as:

$$\begin{aligned} \text{predictor:} \quad & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) \\ & \mathbf{x}^{\mathbf{P}}_{\mathbf{k+\frac{1}{2}}} = \mathbf{x}_{\mathbf{k}} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_{\mathbf{k}} \\ \\ \text{corrector:} \quad & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+\frac{1}{2}}} = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+\frac{1}{2}}}, t_{k+\frac{1}{2}}) \\ & \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+\frac{1}{2}}} \end{aligned}$$

This technique evaluates the predictor at time $t_k + h/2$. It is a little cheaper than Heun's algorithm due to the additional zero in the β–matrix.

The entire family of such methods is referred to as second–order Runge–Kutta methods, abbreviated as RK2.

The idea can be further generalized by adding more stages. The general explicit Runge–Kutta algorithm can be described as follows:

$$
\begin{aligned}
0^{th}\text{ stage:} \quad & \dot{\mathbf{x}}^{\mathbf{P_0}} = \mathbf{f}(\mathbf{x_k}, t_k) \\
j^{th}\text{ stage:} \quad & \mathbf{x}^{\mathbf{P_j}} = \mathbf{x_k} + h \cdot \textstyle\sum_{i=1}^{j} \beta_{ji} \cdot \dot{\mathbf{x}}^{\mathbf{P_{i-1}}} \\
& \dot{\mathbf{x}}^{\mathbf{P_j}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_j}}, t_k + \alpha_j \cdot h) \\
\text{last stage:} \quad & \mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \textstyle\sum_{i=1}^{\ell} \beta_{\ell i} \cdot \dot{\mathbf{x}}^{\mathbf{P_{i-1}}}
\end{aligned}
$$

where ℓ denotes the number of stages of the method. The most popular of these methods is the following fourth–order accurate Runge–Kutta (RK4) technique:

$$
\alpha = \begin{pmatrix} 1/2 \\ 1/2 \\ 1 \\ 1 \end{pmatrix} ; \quad \beta = \begin{pmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1/6 & 1/3 & 1/3 & 1/6 \end{pmatrix} \tag{3.15}
$$

or:

$$
\begin{aligned}
0^{\text{th}}\text{ stage:} \quad & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x_k}, t_k) \\
1^{\text{st}}\text{ stage:} \quad & \mathbf{x}^{\mathbf{P_1}} = \mathbf{x_k} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_{\mathbf{k}} \\
& \dot{\mathbf{x}}^{\mathbf{P_1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_1}}, t_{k+\frac{1}{2}}) \\
2^{\text{nd}}\text{ stage:} \quad & \mathbf{x}^{\mathbf{P_2}} = \mathbf{x_k} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}^{\mathbf{P_1}} \\
& \dot{\mathbf{x}}^{\mathbf{P_2}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_2}}, t_{k+\frac{1}{2}}) \\
3^{\text{rd}}\text{ stage:} \quad & \mathbf{x}^{\mathbf{P_3}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}^{\mathbf{P_2}} \\
& \dot{\mathbf{x}}^{\mathbf{P_3}} = \mathbf{f}(\mathbf{x}^{\mathbf{P_3}}, t_{k+1}) \\
4^{\text{th}}\text{ stage:} \quad & \mathbf{x_{k+1}} = \mathbf{x_k} + \tfrac{h}{6} \cdot [\dot{\mathbf{x}}_{\mathbf{k}} + 2 \cdot \dot{\mathbf{x}}^{\mathbf{P_1}} + 2 \cdot \dot{\mathbf{x}}^{\mathbf{P_2}} + \dot{\mathbf{x}}^{\mathbf{P_3}}]
\end{aligned}
$$

or yet more simply:

$$
\begin{aligned}
0^{\text{th}}\text{ stage:} \quad & \mathbf{k_1} = \mathbf{f}(\mathbf{x_k}, t_k) \\
1^{\text{st}}\text{ stage:} \quad & \mathbf{k_2} = \mathbf{f}(\mathbf{x_k} + \tfrac{h}{2} \cdot \mathbf{k_1}, t_k + \tfrac{h}{2}) \\
2^{\text{nd}}\text{ stage:} \quad & \mathbf{k_3} = \mathbf{f}(\mathbf{x_k} + \tfrac{h}{2} \cdot \mathbf{k_2}, t_k + \tfrac{h}{2}) \\
3^{\text{rd}}\text{ stage:} \quad & \mathbf{k_4} = \mathbf{f}(\mathbf{x_k} + h \cdot \mathbf{k_3}, t_k + h) \\
4^{\text{th}}\text{ stage:} \quad & \mathbf{x_{k+1}} = \mathbf{x_k} + \tfrac{h}{6} \cdot [\mathbf{k_1} + 2 \cdot \mathbf{k_2} + 2 \cdot \mathbf{k_3} + \mathbf{k_4}]
\end{aligned}
$$

This RK4 algorithm is particularly attractive due to the many zero elements in its β–matrix. As it is a four–stage algorithm, it involves four function evaluations. These are taken at t_k, $t_{k+1/2}$, $t_{k+1/2}$, and t_{k+1}. Thus, it is possible to think of this RK4 algorithm as a macro–step consisting of four micro–steps, two of length $h/2$, and two of length 0.

The Butcher tableau of this method can be written as:

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
x	1/6	1/3	1/3	1/6

In general:

$\mathbf{c}$	$\mathbf{A}$
x	$\mathbf{b}'$

The $\mathbf{c}$–vector contains the time instants when the various function evaluations are performed, the $\mathbf{A}$–matrix contains the weights of the various predictor stages, and the $\mathbf{b}'$–vector contains the weights of the corrector stage.

Notice that the number of stages and the approximation order are not necessarily identical. Higher–order RK algorithms require a larger number of stages to achieve a given order of accuracy. Table 3.1 provides a historic overview of the development of RK algorithms.

Developer	Year	Order	# of Stages
Euler [3.8]	1768	1	1
Runge [3.21]	1895	4	4
Heun [3.14]	1900	2	2
Kutta [3.17]	1901	5	6
Huťa [3.15]	1956	6	8
Shanks [3.22]	1966	7	9
Curtis [3.4]	1970	8	11

TABLE 3.1. History of Runge–Kutta Algorithms.

It is interesting to notice that, although the general mechanism for designing such algorithms had been known for quite some time, higher–order RK algorithms were slow in coming. This is due to the fact that the setting up of the nonlinear equations and their subsequent solution is an utterly tedious process. The original algorithm by Kutta contained an error that went unnoticed until it was corrected by Nyström [3.20] in 1925. Curtis finally had to deal with a large number of very awkward nonlinear equations in more than 200 unknowns. Symbolic formulae manipulation programs, such as Mathematica or Maple, would make it much easier today to set up and solve these sets of equations without making errors on the way, but such programs were unavailable at the time, and so, at least for these researchers, mathematics wasn't always fun ... but required lots of patience, perseverance, and suffering.

It is possible to design RK algorithms in the same number of stages as the approximation order only up to fourth order. It can be shown that no five–stage RK method can be found that is fifth–order accurate. Of course, it is important to keep the number of stages as small as possible, since each additional stage requires an extra function evaluation.

Additional requirements are usually formulated that are not inherent in the technique itself, but make a lot of practical sense. Obviously, we want to request that, in an ℓ–stage algorithm:

$$\alpha_\ell = 1.0 \tag{3.16}$$

since we wish to end the step at t_{k+1}. Also, we usually want to make sure that:

$$\alpha_i \in [0.0, 1.0] \quad ; \quad i = \{1, 2, \ldots, \ell\} \tag{3.17}$$

that is, all function evaluations are performed at times that lie between t_k and t_{k+1}.

If we want to prevent the algorithm from ever "integrating backward through time," we shall add the constraint that:

$$\alpha_j \geq \alpha_i \quad ; \quad j \geq i \tag{3.18a}$$

If we want to disallow micro–steps of length 0, we make this condition even more stringent:

$$\alpha_j > \alpha_i \quad ; \quad j > i \tag{3.18b}$$

The previously introduced classical RK4 algorithm violates Eq.(3.18b).

Why is this last condition important? Modelers sometimes wish to explicitly use derivative operations in their models. This is generally a bad idea, but it may not always be avoidable. For example, if u is a real–time input that stems from a measurement sensor, and the model requires $\dot{u}$, there is nothing in the world that can save us from actually having to differentiate the input. The typical simulationist would then approximate the derivative by:

$$\dot{u} \approx \frac{u - u_{\text{last}}}{t - t_{\text{last}}} \tag{3.19}$$

where t_{last} is the time of the previous function evaluation, and u_{last} is the value of the input u at that time. Therefore, if it should ever happen that $t = t_{\text{last}}$, the numerical differentiation algorithm would get itself into trouble.

Two caveats are called for. While we were able to develop Heun's method using a matrix–vector notation, this technique won't work anymore as we proceed to third–order algorithms. Let us explain.

We found that:

$$\frac{d\mathbf{f}}{dt} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \cdot \frac{d\mathbf{x}}{dt} + \frac{\partial \mathbf{f}}{\partial t} \tag{3.20}$$

or, in shorthand notation:

$$\dot{\mathbf{f}} = \mathbf{f}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_t \tag{3.21}$$

When we proceed to third–order algorithms, we need an expression for the second absolute derivative of $\mathbf{f}$ with respect to time. Thus, we are inclined to write formally:

$$\begin{aligned}
\ddot{\mathbf{f}} &= (\mathbf{f}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_t)\dot{} \\
&= \dot{\mathbf{f}}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_{\mathbf{x}} \cdot \dot{\mathbf{f}} + \dot{\mathbf{f}}_t \\
&= (\dot{\mathbf{f}})_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_{\mathbf{x}} \cdot (\dot{\mathbf{f}}) + (\dot{\mathbf{f}})_t \\
&= (\mathbf{f}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_t)_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_{\mathbf{x}} \cdot (\mathbf{f}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_t) + (\mathbf{f}_{\mathbf{x}} \cdot \mathbf{f} + \mathbf{f}_t)_t \\
&= \mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2 + 2 \cdot (\mathbf{f}_{\mathbf{x}})^2 \cdot \mathbf{f} + 2 \cdot \mathbf{f}_{\mathbf{x}t} \cdot \mathbf{f} + 2 \cdot \mathbf{f}_{\mathbf{x}} \cdot \mathbf{f}_t + \mathbf{f}_{tt}
\end{aligned} \tag{3.22}$$

but it is not clear, what this is supposed to mean. Obviously, $\ddot{\mathbf{f}}$ is a vector, and so is $\mathbf{f}_{tt}$, but what is $\mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2$ supposed to mean? Is it a tensor multiplied by the square of a vector? Quite obviously, the formal differentiation mechanism doesn't extend to higher derivatives in the sense of familiar matrix–vector multiplications. Evidently, we must treat the expression $\mathbf{f}_{\mathbf{xx}} \cdot (\mathbf{f})^2$ differently.

Butcher [3.3] developed a new syntax and a set of rules for how these higher derivatives must be interpreted. In essence, it turns out that, in this new syntax:

1. sums remain commutative and associative,
2. derivatives can still be computed in any order, i.e., $(\dot{\mathbf{f}})_{\mathbf{x}} = (\mathbf{f}_{\mathbf{x}})\dot{}$, and
3. the multiplication rule can be generalized, thus: $(\mathbf{f}_{\mathbf{x}} \cdot \mathbf{f})_{\mathbf{x}} = \mathbf{f}_{\mathbf{xx}} \cdot \mathbf{f} + (\mathbf{f}_{\mathbf{x}})^2$.

It is not necessary for us to learn Butcher's new syntax. It is sufficient to know that we can basically proceed as before, but must abstain from interpreting terms involving higher derivatives as consisting of factors that are combined by means of the familiar matrix–vector multiplication.

Prior to Butcher's work, all higher–order RK algorithms had simply been derived for the scalar case, and were then blindly applied to integrate entire state vectors. And here comes the second caveat. Butcher discovered that several of the previously developed and popular higher–order RK algorithms drop one or several orders of accuracy when applied to a state vector instead of a scalar state variable.

The reason for this somewhat surprising discovery is very simple. Already when computing the third absolute derivative of $\mathbf{f}$ with respect to time, the

two terms $\mathbf{f_x} \cdot \mathbf{f_{xx}} \cdot (\mathbf{f})^2$ and $\mathbf{f_{xx}} \cdot \mathbf{f} \cdot \mathbf{f_x} \cdot \mathbf{f}$ appear in the derivation. In the scalar case, these two terms are identical, since:

$$a \cdot b = b \cdot a \tag{3.23}$$

Unfortunately —and not *that* surprisingly after all— Eq.(3.23) does not extend to the vector case. Our new animals in the mathematical zoo of data structures and operations exhibit a property that we are already quite familiar with from matrix calculus, namely that multiplications are no longer commutative:

$$\mathbf{A} \cdot \mathbf{B} = (\mathbf{B}' \cdot \mathbf{A}')' \neq \mathbf{B} \cdot \mathbf{A} \tag{3.24}$$

where $\mathbf{A}'$ denotes the transpose of $\mathbf{A}$. So, algorithms that had been developed without grouping such terms together continued to work properly also in the vector case, whereas algorithms that had made use of the commutative nature of scalar multiplications did work well for scalar problems, but dropped one or several approximation orders when exposed to vector problems.

The details of the Butcher syntax are of no immediate concern to us, since we never plan to actually perform these new operations. All we need in order to develop new RK algorithms is to be able to extract their coefficients. To this end, we can pretend that the normal rules of matrix and vector calculus still apply.

3.3 Stability Domains of RK Algorithms

Since all the previously presented RK algorithms are explicit algorithms, we expect their stability domains to look qualitatively like that of the FE algorithm, or more precisely, we expect the contours of marginal stability to bend into the left half $(\lambda \cdot h)$–plane.

Let us plug the linear system of Eq.(2.12) into Heun's algorithm. We find:

$$\begin{array}{lll} \text{predictor:} & \dot{\mathbf{x}}_\mathbf{k} & = \mathbf{A} \cdot \mathbf{x_k} \\ & \mathbf{x}^\mathbf{P}_{\mathbf{k+1}} & = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_\mathbf{k} \\ \\ \text{corrector:} & \dot{\mathbf{x}}^\mathbf{P}_{\mathbf{k+1}} & = \mathbf{A} \cdot \mathbf{x}^\mathbf{P}_{\mathbf{k+1}} \\ & \mathbf{x}^\mathbf{C}_{\mathbf{k+1}} & = \mathbf{x_k} + 0.5 \cdot h \cdot (\dot{\mathbf{x}}_\mathbf{k} + \dot{\mathbf{x}}^\mathbf{P}_{\mathbf{k+1}}) \end{array}$$

or:

$$\mathbf{x}^\mathbf{C}_{\mathbf{k+1}} = [\mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2}] \cdot \mathbf{x_k} \tag{3.25}$$

i.e.,

$$\mathbf{F} = \mathbf{I}^{(\mathbf{n})} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2} \tag{3.26}$$

Since a two–stage algorithm contains only two function evaluations, no powers of h larger than two can appear in the **F**–matrix. Since the technique is second–order accurate, it must approximate the analytical solution:

$$\mathbf{F} = \exp(\mathbf{A} \cdot h) = \mathbf{I^{(n)}} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} + \frac{(\mathbf{A} \cdot h)^3}{3!} + \dots \tag{3.27}$$

up to the quadratic term. Consequently, all two–stage RK2 algorithms share the same stability domain, and the same holds true for all three–stage RK3s, and for all four–stage RK4s. The situation becomes more complicated in the case of the fifth–order algorithms, since there doesn't exist a five–stage RK5. Consequently, the **F**–matrices of RK5s necessarily contain a term in h^6 (with incorrect coefficient), and since there is no reason why these sixth–order terms should carry the same coefficient in different RK5s, their stability domains will look slightly different one from another.

Let us apply our general–purpose stability domain plotting algorithm that was presented in Chapter 2. We find:

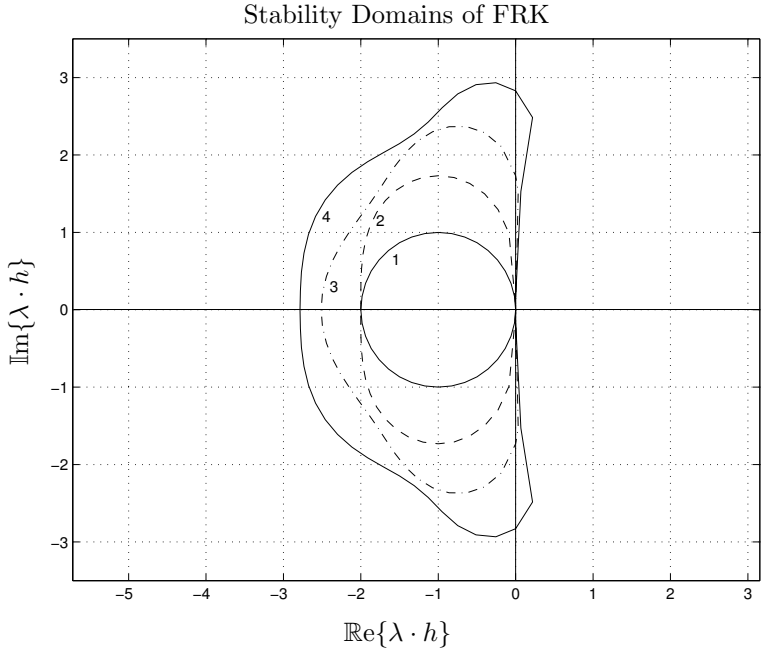

FIGURE 3.1. Stability domains of explicit RK algorithms.

Some of the RK5s are among those algorithms with small stable islands somewhere out in the unstable right half $(\lambda \cdot h)$–plane.

The reader may notice that these algorithms try indeed (and not surprisingly) to approximate the analytical stability domain, i.e., higher–order

RKs follow the imaginary axis better and better. Also, the stability domains grow with increasing approximation order. This is very satisfying, since higher–order algorithms call for larger step sizes.

3.4 Stiff Systems

Although the term "stiff system" has been popular at least since Gear's 1971 book [3.10] appeared, the numerical ODE literature still doesn't provide a crisp definition of what a stiff system really is. Even the 1991 book by Lambert [3.18] treats *"the nature of stiffness"* on as many as nine pages. Lambert observes that:

> *Statement # 1:* "A linear constant coefficient system is stiff if all of its eigenvalues have negative real part and the stiffness ratio is large."
>
> *Statement # 2:* "Stiffness occurs when stability requirements, rather than those of accuracy, constrain the steplength."
>
> *Statement # 3:* "Stiffness occurs when some components of the solution decay much more rapidly than others."
>
> *Statement # 4:* "A system is said to be stiff in a given interval of time if in that interval the neighboring solution curves approach the solution curve at a rate which is very large in comparison with the rate at which the solution varies in that interval."

The first statement is not overly useful since it relates to linear systems only. The second statement is not very precise since the accuracy requirements are not specified. Thus, one and the same system may be stiff, according to this statement, if the accuracy requirements are loose, and non–stiff if the accuracy requirements are tight. The third statement indirectly refers to the superposition principle, and is therefore, in a strict sense, again limited to linear systems. The fourth statement is basically a reformulation of the third.

Lambert concludes his exposé of the matter with the following definition:

> *Definition # 1:* "If a numerical method with a finite region of absolute stability, applied to a system with any initial conditions, is forced to use in a certain interval of integration a steplength which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be **stiff** in that interval."

Again, what exactly means "excessively small"?

Our remarks may sound critical of Lambert's work. They are not meant to be. Lambert's 1991 book represents a significant contribution to the numerical ODE literature. All we want to convey is that here is a term that has been around for more than a quarter of a century, and yet, the term is still fuzzy.

Let us attempt a more crisp definition of the term "stiff system":

> *Definition #2:* "An ODE system is called stiff if, when solved with any n^{th}–order accurate integration algorithm and a local error tolerance of 10^{-n}, the step size of the algorithm is forced down to below a value indicated by the local error estimate due to constraints imposed on it by the limited size of the numerically stable region."

Our definition comes closest to Lambert's statement #2, except that we added a definition of what we mean by accuracy requirements. Our definition is still somewhat fuzzy since it is possible that a system may fall under the category *stiff* when solved with one n^{th}–order accurate integration algorithm, and doesn't when solved with another. Yet, as we shall see, the grey zone of "marginally stiff" systems is fairly narrow, and moreover, this is exactly what these systems are: *marginally stiff*.

It is treacherous to rely on the eigenvalues of the Jacobian of a nonlinear or even linear but time–variant system to conclude anything about stiffness. Let us explain.

Given the system:

$$\dot{\mathbf{x}} = \mathbf{A}(t) \cdot \mathbf{x} = \begin{pmatrix} -2.5 & 1.5 \cdot \exp(-100t) \\ -0.5 \cdot \exp(100t) & -0.5 \end{pmatrix} \cdot \mathbf{x} \tag{3.28a}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 23 \\ 11 \end{pmatrix} \tag{3.28b}$$

Its analytical solution is:

$$x_1(t) = 5 \cdot \exp(-101t) + 18 \cdot \exp(-102t) \tag{3.29a}$$

$$x_2(t) = 5 \cdot \exp(-t) + 6 \cdot \exp(-2t) \tag{3.29b}$$

Therefore, the system is awfully stiff. Yet, the eigenvalues of its Jacobian are -1.0 and -2.0, i.e., they are perfectly tame at all times.

If we plug this system into the FE algorithm (or RK1, which is the same algorithm), we find:

$$\mathbf{F}(t) = \mathbf{I^{(n)}} + \mathbf{A}(t) \cdot h = \begin{pmatrix} -2.5 \cdot h + 1.0 & 1.5 \cdot h \cdot \exp(-100t) \\ -0.5 \cdot h \cdot \exp(100t) & -0.5 \cdot h + 1.0 \end{pmatrix} \tag{3.30}$$

Thus, the eigenvalues of the discrete–time system are at:

$$\lambda_1 = 1 - h \quad ; \quad \lambda_2 = 1 - 2h \tag{3.31}$$

which is what we would have expected from the locations of the eigenvalues of the continuous–time system and the stability domain of Fig.2.7. Thus, the stability domain doesn't indicate any foul play in this case. Codes that rely on the Jacobian for computing local error estimates will be fooled by this problem.

However, this is a particularly malignant problem, and fortunately one that physics doesn't usually prescribe. We may not truly want to simulate this system anyway since, already at simulated time $t = 10.0$, the element a_{21} has acquired a value of $-0.5 \cdot \exp(1000)$, something our simulator will most certainly complain bitterly about.

It is therefore still useful to search for methods that include in their numerically stable region the entire left half $(\lambda \cdot h)$–plane, or at least a large portion thereof.

> *Definition:* A numerical integration scheme that contains the entire left half $(\lambda \cdot h)$–plane as part of its numerical stability domain is called *absolute stable*, or, more simply, *A–stable*.

One way to obtain A–stable algorithms is to modify the recipe for developing RK algorithms by allowing non–zero elements also above the diagonal of the β–matrix [3.2] [3.13]. Such algorithms are invariably implicit. They are therefore called *implicit Runge–Kutta schemes*, abbreviated as IRK. A special role among those algorithms employ methods that limit the non-zero elements in their respective β–matrices to the first super–diagonal. Using the Butcher tableau representation, its **A**–matrix is still lower triangular, but contains nonlinear elements along its diagonal. Such algorithms are called *diagonally implicit Runge–Kutta schemes*, abbreviated as DIRK. They are implicit in each stage, but each stage can be iterated separately, and it is therefore fairly easy to implement a Newton iteration on them.

However, rather than looking at the problem of defining general IRK and DIRK algorithms through their α–vectors and β–matrices, we want to turn to two special classes of such algorithms that have interesting properties: the *extrapolation techniques*, and the *backinterpolation techniques*. We shall discuss some other classes of implicit Runge–Kutta algorithms in Chapter 8 in the context of solving sets of mixed differential and algebraic equations (DAEs).

3.5 Extrapolation Techniques

The idea behind the (Richardson) extrapolation techniques is quite straightforward. We repeat the same integration step with several low–order tech-

niques and blend the results to get a higher–order technique. Let us explain the concept by means of the linear system:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{3.32}$$

We shall integrate the system four times across one macro–step of length h each time using the FE algorithm with different micro–step sizes: $\eta_1 = h$, $\eta_2 = h/2$, $\eta_3 = h/3$, and $\eta_4 = h/4$. Accordingly, we need only one micro–step of length η_1, but we need four micro–steps of length η_4. The corresponding discrete–time systems are:

$$\begin{aligned}
\mathbf{x^{P_1}}(k+1) &= [\mathbf{I^{(n)}} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \\
\mathbf{x^{P_2}}(k+1) &= [\mathbf{I^{(n)}} + \frac{\mathbf{A} \cdot h}{2}]^2 \cdot \mathbf{x}(k) \\
\mathbf{x^{P_3}}(k+1) &= [\mathbf{I^{(n)}} + \frac{\mathbf{A} \cdot h}{3}]^3 \cdot \mathbf{x}(k) \\
\mathbf{x^{P_4}}(k+1) &= [\mathbf{I^{(n)}} + \frac{\mathbf{A} \cdot h}{4}]^4 \cdot \mathbf{x}(k)
\end{aligned} \tag{3.33a}$$

with the corrector:

$$\mathbf{x^C}(k+1) = \alpha_1 \cdot \mathbf{x^{P_1}}(k+1) + \alpha_2 \cdot \mathbf{x^{P_2}}(k+1) + \alpha_3 \cdot \mathbf{x^{P_3}}(k+1) + \alpha_4 \cdot \mathbf{x^{P_4}}(k+1) \tag{3.33b}$$

Multiplying the predictor formulae out, we find:

$$\begin{aligned}
\mathbf{x^{P_1}} &= [\mathbf{I^{(n)}} + \mathbf{A} \cdot h] \cdot \mathbf{x}(k) \\
\mathbf{x^{P_2}} &= [\mathbf{I^{(n)}} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{4}] \cdot \mathbf{x}(k) \\
\mathbf{x^{P_3}} &= [\mathbf{I^{(n)}} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{3} + \frac{(\mathbf{A} \cdot h)^3}{27}] \cdot \mathbf{x}(k) \\
\mathbf{x^{P_4}} &= [\mathbf{I^{(n)}} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{8} + \frac{(\mathbf{A} \cdot h)^3}{16} + \frac{(\mathbf{A} \cdot h)^4}{256}] \cdot \mathbf{x}(k)
\end{aligned} \tag{3.34a}$$

and for the corrector, we obtain:

$$\begin{aligned}
\mathbf{x^C}(k+1) = [&(\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4) \cdot \mathbf{I^{(n)}} \\
&+ (\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4) \cdot \mathbf{A} \cdot h \\
&+ (\frac{\alpha_2}{4} + \frac{\alpha_3}{3} + \frac{3\alpha_4}{8}) \cdot (\mathbf{A} \cdot h)^2 \\
&+ (\frac{\alpha_3}{27} + \frac{\alpha_4}{16}) \cdot (\mathbf{A} \cdot h)^3 + \frac{\alpha_4}{256} \cdot (\mathbf{A} \cdot h)^4] \cdot \mathbf{x_k}
\end{aligned} \tag{3.34b}$$

Comparing Eq.(3.34b) with the correct Taylor Series truncated after the fourth–order term, we obtain four linear equations in the four unknown α_i parameters:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1/4 & 1/3 & 3/8 \\ 0 & 0 & 1/27 & 1/16 \\ 0 & 0 & 0 & 1/256 \end{pmatrix} \cdot \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/6 \\ 1/24 \end{pmatrix} \tag{3.35}$$

which can be solved directly. We find:

$$\alpha_1 = -\frac{1}{6} \;\; ; \;\; \alpha_2 = 4 \;\; ; \;\; \alpha_3 = -\frac{27}{2} \;\; ; \;\; \alpha_4 = \frac{32}{3} \tag{3.36}$$

Thus, we just discovered another way to construct an RK4 algorithm since the extrapolation technique is fourth–order accurate ... at least for linear systems. We didn't bother to check whether the algorithm is also accurate up to fourth order for nonlinear systems, since unfortunately, the technique is quite inefficient. It took 10 function evaluations to complete a single macro–step. Compare this with the four function evaluations needed when performing an ordinary RK4 step.

Let us try another idea. The order of the algorithm wouldn't be all that important if we only could make the step size sufficiently small. Unfortunately, this would mean that we would have to perform many such steps in order to complete the simulation run ... or doesn't it?

We can write:

$$\mathbf{x_{k+1}}(\eta) = \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \eta + \mathbf{e_2} \cdot \frac{\eta^2}{2!} + \mathbf{e_3} \cdot \frac{\eta^3}{3!} + \ldots \tag{3.37}$$

where $\mathbf{x_{k+1}}$ is the true (yet unknown) value of $\mathbf{x}$ at time $t_k + h$, whereas $\mathbf{x_{k+1}}(\eta)$ is the numerical value that we find when we integrate the system from time t_k to time $t_k + h$ using the micro–step size η. Obviously, this value contains an error. We now develop the numerical value into a Taylor Series in η around the (unknown) correct value. The $\mathbf{e_i}$ vectors are error vectors [3.6].

We truncate the Taylor Series after the cubic term, and write Eq.(3.37) down for the same values of η_i as before. We find:

$$\begin{aligned}
\mathbf{x^{P_1}}(\eta_1) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot h + \frac{\mathbf{e_2}}{2!} \cdot h^2 + \frac{\mathbf{e_3}}{3!} \cdot h^3 \\
\mathbf{x^{P_2}}(\eta_2) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{2} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{2})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{2})^3 \\
\mathbf{x^{P_3}}(\eta_3) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{3} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{3})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{3})^3 \\
\mathbf{x^{P_4}}(\eta_4) &\approx \mathbf{x_{k+1}} + \mathbf{e_1} \cdot \frac{h}{4} + \frac{\mathbf{e_2}}{2!} \cdot (\frac{h}{4})^2 + \frac{\mathbf{e_3}}{3!} \cdot (\frac{h}{4})^3
\end{aligned} \tag{3.38}$$

or in matrix notation:

$$\begin{pmatrix} \mathbf{x^{P_1}} \\ \mathbf{x^{P_2}} \\ \mathbf{x^{P_3}} \\ \mathbf{x^{P_4}} \end{pmatrix} \approx \begin{pmatrix} h^0 & h^1 & h^2 & h^3 \\ (h/2)^0 & (h/2)^1 & (h/2)^2 & (h/2)^3 \\ (h/3)^0 & (h/3)^1 & (h/3)^2 & (h/3)^3 \\ (h/4)^0 & (h/4)^1 & (h/4)^2 & (h/4)^3 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x_{k+1}} \\ \mathbf{e_1} \\ \mathbf{e_2}/2 \\ \mathbf{e_3}/6 \end{pmatrix} \tag{3.39}$$

By inverting the Van–der–Monde matrix, we can solve for the unknown $\mathbf{x_{k+1}}$ and the three error vectors. Since we aren't interested in the errors, we only look at the first row of the inverted Van–der–Monde matrix. It turns out that the values in this row don't depend at all on the step size h. We find:

$$\mathbf{x_{k+1}} \approx \begin{pmatrix} -\frac{1}{6} & 4 & -\frac{27}{2} & \frac{32}{3} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x^{P_1}} \\ \mathbf{x^{P_2}} \\ \mathbf{x^{P_3}} \\ \mathbf{x^{P_4}} \end{pmatrix} \tag{3.40}$$

Obviously, $\mathbf{x_{k+1}}$ is no longer the truly correct solution since we had truncated the Taylor Series in η after the cubic term. However, the algorithm did the best it could to estimate the true value given the available data ... by raising the approximation order of the method to four.

Thus, we got precisely the same answers as before. We just found another way to derive the extrapolation method. Both approaches have their pros and cons. The first technique unveiled that the resulting method is indeed fourth–order accurate (at least for linear systems). The second method didn't show this fact explicitly ... it was more like swinging a magic wand. On the other hand, the first approach made explicitly use of the fact that each micro–step was performed by means of the FE algorithm. The second approach was not based on any such assumption.

Thus, we could now replace each of the micro–steps by a BE step of the same length, e.g. using Newton iteration if the system to be simulated is nonlinear, and still use the same corrector. The overall implicit extrapolation (IEX) technique then presents itself as:

$$
\begin{array}{lll}
1^{\text{st}} \text{ predictor:} & \mathbf{k_1} & = \mathbf{x_k} + h \cdot \mathbf{f}(\mathbf{k_1}, t_{k+1}) \\
2^{\text{nd}} \text{ predictor:} & \mathbf{k_{2a}} & = \mathbf{x_k} + \frac{h}{2} \cdot \mathbf{f}(\mathbf{k_{2a}}, t_{k+\frac{1}{2}}) \\
 & \mathbf{k_2} & = \mathbf{k_{2a}} + \frac{h}{2} \cdot \mathbf{f}(\mathbf{k_2}, t_{k+1}) \\
3^{\text{rd}} \text{ predictor:} & \mathbf{k_{3a}} & = \mathbf{x_k} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_{3a}}, t_{k+\frac{1}{3}}) \\
 & \mathbf{k_{3b}} & = \mathbf{k_{3a}} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_{3b}}, t_{k+\frac{2}{3}}) \\
 & \mathbf{k_3} & = \mathbf{k_{3b}} + \frac{h}{3} \cdot \mathbf{f}(\mathbf{k_3}, t_{k+1}) \\
4^{\text{th}} \text{ predictor:} & \mathbf{k_{4a}} & = \mathbf{x_k} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4a}}, t_{k+\frac{1}{4}}) \\
 & \mathbf{k_{4b}} & = \mathbf{k_{4a}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4b}}, t_{k+\frac{1}{2}}) \\
 & \mathbf{k_{4c}} & = \mathbf{k_{4b}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_{4c}}, t_{k+\frac{3}{4}}) \\
 & \mathbf{k_4} & = \mathbf{k_{4c}} + \frac{h}{4} \cdot \mathbf{f}(\mathbf{k_4}, t_{k+1}) \\
\text{corrector:} & \mathbf{x_{k+1}} & = -\frac{1}{6} \cdot \mathbf{k_1} + 4 \cdot \mathbf{k_2} - \frac{27}{2} \cdot \mathbf{k_3} + \frac{32}{3} \cdot \mathbf{k_4}
\end{array}
$$

A complete analysis of the nonlinear accuracy order of this technique is quite involved, and we have not attempted it. However, by following our initial approach at deriving the extrapolation method now for BE steps in place of FE steps, it is a simple exercise to verify that the method indeed carries fourth-order accuracy for solving linear systems. As was to be expected from the latter way of reasoning, the α_i–parameters turn out to be exactly the same for BE steps as for FE steps.

Let us look at the stability domain of this method. It is presented in Fig.3.2. The method is A–stable, and has a nicely large unstable region in the right half $(\lambda \cdot h)$–plane.

Implicit extrapolation techniques, such as the IEX4 technique explained above, have, in comparison with IRK or DIRK algorithms, the distinct advantage that they are easy to construct. They have the disadvantages that no formal nonlinear accuracy analysis is currently available, and that they are still fairly inefficient. IEX4 is a 10–stage algorithm. In contrast, a fourth–order fully-implicit IRK algorithm can be constructed with only two stages, as shall be demonstrated in Chapter 8.

3.6 Marginally Stable Systems

We have seen in Chapter 2 that neither the FE nor the BE algorithm will do a decent job when confronted with eigenvalues on or in the vicinity of the imaginary axis. Unfortunately, this situation occurs quite frequently, and there exists an entire class of applications, namely the hyperbolic PDEs that, when converted to sets of ODEs, exhibit this property as we shall see later. It seems thus justified to analyze what can be done to tackle such problems. Let us start with a proper definition:

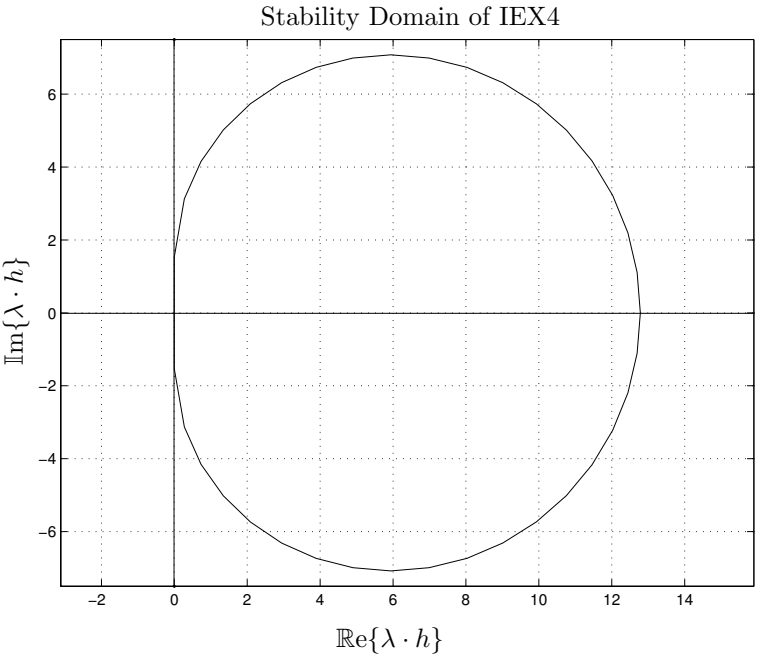

FIGURE 3.2. Stability domain of implicit extrapolation method.

> *Definition:* A dynamical system whose Jacobian has its dominant eigenvalues on or in the vicinity of the imaginary axis is called *marginally stable*.

The dominant eigenvalues of a matrix are those eigenvalues that have the most positive real parts, i.e., that are located most to the right in the λ–plane.

In order to tackle such problems decently, we require integration algorithms that approximate the imaginary axis particularly well. Such algorithms do exist, and, in fact, algorithms of arbitrary order can be constructed whose borders of numerical stability coincide with the imaginary axis. We shall study these algorithms in due course.

> *Definition:* A numerical integration scheme that contains the entire left half $(\lambda \cdot h)$–plane and nothing but the left half $(\lambda \cdot h)$–plane as its numerical stability domain is called *faithfully stable*, or, more simply, *F–stable*.

The reader may now be inclined to think that F–stable algorithms must be the answer to all our prayers. Unfortunately, this is not so. F–stable algorithms will perform poorly when asked to integrate stiff systems. The reason for this surprising disclosure is the following: If we think of the complex $(\lambda \cdot h)$–plane as an infinitely large plane, we are inclined to assume

that each point at infinity is infinitely far away from each other point at infinity. However, it turns out to be more accurate to think of the complex $(\lambda \cdot h)$–plane as an infinitely large globe. From wherever we stand on that globe, infinity is the single one spot that is farthest away from us. Thus, infinity is a "single spot" in the sense that the numerical properties of any integration algorithm based on Taylor–Series expansion will be exactly the same irrespective of the direction from which we approach infinity.

Consequently, since (by definition) the entire imaginary axis belongs to the margin of stability, so does the infinity "spot" itself. This means that, although the entire left half $(\lambda \cdot h)$–plane is indeed stable, as we approach infinity along the negative real axis, points along the negative real axis will become less and less stable until, at point infinity, stability is lost. Similarly, although the entire right half $(\lambda \cdot h)$–plane is indeed unstable, as we approach infinity along the positive real axis, points along the positive real axis will become less and less unstable until, at point infinity, stability is reconquered.

The λ–plane has different properties. As we move along a line parallel to the real axis to the left, the damping of an eigenvalue located at that position increases constantly until it reaches a value of infinity at point infinity. In fact, the damping of an eigenvalue is identical with its distance from the imaginary axis.

An F–stable algorithm can obviously not mimic this facet of the λ–plane, and consequently, it will perform poorly when exposed to eigenvalues located far out to the left on the λ–plane. The time response due to these eigenvalues will not properly be dampened out. The F–stably simulated system with eigenvalues at such locations will therefore behave more sluggishly than the real system.

> *Definition:* A numerical integration scheme that is A–stable, and, in addition, whose damping properties increase to infinity as $\mathbb{R}\mathrm{e}\{\lambda\} \to -\infty$, is called *L–stable.*

The various numerical stability definitions are, in a strict sense, only meaningful for linear time–invariant systems, but they are often good indicators when applied to nonlinear systems as well.

When dealing with stiff systems, it is not sufficient to demand A–stability from the integration algorithm. We need to look more closely at the damping behavior. L–stability may be a desirable property. Evidently, all F–stable algorithms are also A–stable, but never L–stable.

A system that is both marginally stable and stiff is difficult to cope with. Such systems exist, and we shall provide an example of one such system in due course. As of now, we are somewhat at a loss when asked which algorithm we recommend in this situation. What might possibly work best is an L–stable algorithm with an extra large unstable region in the right half $(\lambda \cdot h)$–plane, but best may still not be very good. We shall demonstrate how such algorithms can be constructed.

3.7 Backinterpolation Methods

We shall now look at yet another class of special IRK methods, called *backinterpolation techniques*, abbreviated as BI algorithms. Similar to the previously discussed *extrapolation techniques*, BI methods are easy to construct. However, they offer much better control over the accuracy order in comparison with the IEX algorithms even when applied to nonlinear systems. Also, they are considerably more efficient than IEX algorithms. BI algorithms can be made F–stable, L–stable, or anything in between, depending on the current needs of the user, and they lend themselves conveniently to stability domain shaping.

Let us look once more at the BE algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{3.41}$$

We can rearrange Eq.(3.41) as follows:

$$\mathbf{x_k} = \mathbf{x_{k+1}} - h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{3.42}$$

Thus, a step *forward* through time from time t_k to time t_{k+1} using the BE algorithm with a step size of h can also be interpreted as a step *backward* through time from time t_{k+1} to time t_k using the FE algorithm with a step size of $-h$.

Thus, one way to implement the BE algorithm is to start out from an estimate of the yet unknown value $\mathbf{x_{k+1}}$, and integrate backward through time to t_k. We then iterate on the unknown "initial" condition $\mathbf{x_{k+1}}$ until we hit the known "final" value $\mathbf{x_k}$ accurately. We accept the last guess of $\mathbf{x_{k+1}}$ as the correct value, and estimate $\mathbf{x_{k+2}}$. Now we integrate again backward through time to t_{k+1} until we hit $\mathbf{x_{k+1}}$.

This idea can, of course, be extended to any RK algorithm. For example, we can take any off–the–shelf RK4 algorithm to replace the former FE algorithm in integrating backward through time. This is the basic idea behind backinterpolation. These simplest of all BI algorithms are therefore sometimes called *backward Runge–Kutta methods*, or, abbreviated, BRK methods.

This gives us a series of algorithms of increasing order with the **F**–matrices:

$$\mathbf{F_1} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1} \tag{3.43a}$$

$$\mathbf{F_2} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!}]^{-1} \tag{3.43b}$$

$$\mathbf{F_3} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} - \frac{(\mathbf{A} \cdot h)^3}{3!}]^{-1} \tag{3.43c}$$

$$\mathbf{F_4} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} - \frac{(\mathbf{A} \cdot h)^3}{3!} + \frac{(\mathbf{A} \cdot h)^4}{4!}]^{-1} \tag{3.43d}$$

Their numerical stability domains are shown in Fig.(3.3). Evidently, these stability domains are the mirror images of the stability domains of the explicit RK algorithms. This is not further surprising, since they are the same algorithms with h replaced by $-h$.

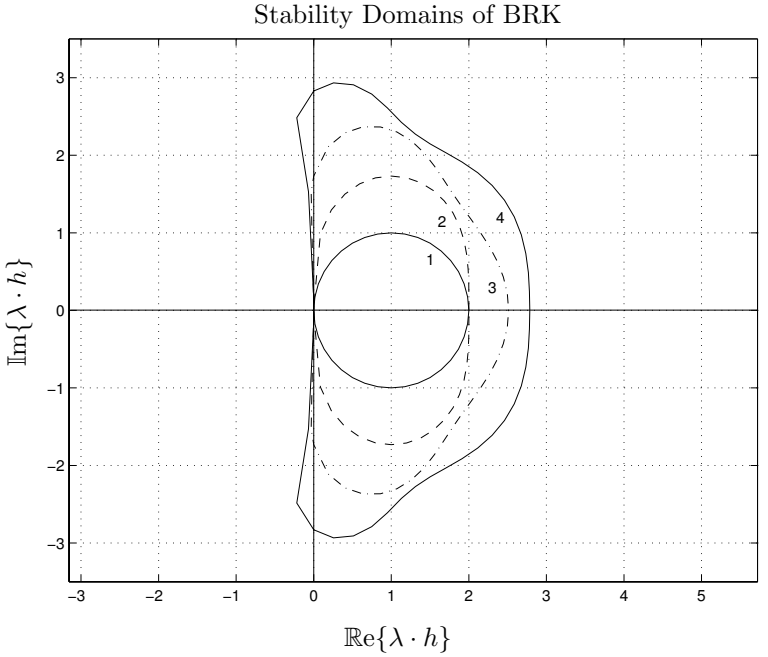

FIGURE 3.3. Stability domains of basic backinterpolation methods.

Let us now discuss whether we can exploit the backinterpolation idea to generate a set of F–stable algorithms of increasing order.

Several F–stable algorithms have been known for a long time. One of those algorithms is the *trapezoidal rule*:

$$1^{\text{st}} \text{ stage:} \quad \mathbf{x_{k+\frac{1}{2}}} = \mathbf{x_k} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_\mathbf{k}$$

$$2^{\text{nd}} \text{ stage:} \quad \mathbf{x_{k+1}} = \mathbf{x_{k+\frac{1}{2}}} + \tfrac{h}{2} \cdot \dot{\mathbf{x}}_\mathbf{k+1}$$

The trapezoidal rule is an implicit algorithm that can be envisaged as a cyclic method consisting of a semi–step of length $h/2$ using FE followed by another semi–step of length $h/2$ using BE.

Its **F**–matrix is thus:

$$\mathbf{F_{TR}} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \tag{3.44}$$

The trapezoidal rule exploits the symmetry of the stability domains of its two semi–steps.

Since we can implement the BE semi–step as a BRK1 step using the backinterpolation method, we can extend this idea also to higher–order algorithms. Their **F**–matrices will be:

$$\mathbf{F_1} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \tag{3.45a}$$

$$\mathbf{F_2} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8}] \tag{3.45b}$$

$$\begin{aligned}\mathbf{F_3} = &[\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} - \frac{(\mathbf{A} \cdot h)^3}{48}]^{-1} \cdot \\ &[\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} + \frac{(\mathbf{A} \cdot h)^3}{48}]\end{aligned} \tag{3.45c}$$

$$\begin{aligned}\mathbf{F_4} = &[\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} - \frac{(\mathbf{A} \cdot h)^3}{48} + \frac{(\mathbf{A} \cdot h)^4}{384}]^{-1} \cdot \\ &[\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{8} + \frac{(\mathbf{A} \cdot h)^3}{48} + \frac{(\mathbf{A} \cdot h)^4}{384}]\end{aligned} \tag{3.45d}$$

All these techniques are F–stable. $\mathbf{F_2}$ is not very useful, since $\mathbf{F_1}$ is, by accident, already second–order accurate.

The implementation of these algorithms is straightforward. For example, $\mathbf{F_4}$ can be implemented in the following way. We start out from time t_k and integrate forward through time across a semi–step from time t_k to time $t_{k+\frac{1}{2}}$ using any off–the–shelf RK4 algorithm. We store the resulting state $\mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}$ for later reuse. We then estimate the value $\mathbf{x_{k+1}}$, e.g. by letting $\mathbf{x_{k+1}} = \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}$, and integrate backward through time across the second semi–step from t_{k+1} to $t_{k+\frac{1}{2}}$ using the same off–the–shelf RK4 algorithm. The resulting state is $\mathbf{x}^{\text{right}}_{\mathbf{k+\frac{1}{2}}}$. We now iterate on the unknown state $\mathbf{x_{k+1}}$, until $\mathbf{x}^{\text{right}}_{\mathbf{k+\frac{1}{2}}} = \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}$. We then use the final value of $\mathbf{x_{k+1}}$ as the initial condition for the next integration macro–step.

We need to analyze the iteration process somewhat more. Chapter 2 taught us that a poor choice of the iteration algorithm can foul up our stability domain.

When applying Newton iteration to the BI1 algorithm, we can set:

$$\mathcal{F}(\mathbf{x_{k+1}}) = \mathbf{x}^{\text{right}}_{\mathbf{k+\frac{1}{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}} = 0.0 \tag{3.46}$$

and find:

$$
\begin{aligned}
\mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}} &= \text{FE}(\mathbf{x_k}, t_k, \tfrac{h}{2}) \\
\mathbf{x^0_{k+1}} &= \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}} \\
& \\
\mathbf{J^0_{k+1}} &= \mathcal{J}(\mathbf{x^0_{k+1}}, t_{k+1}) \\
\mathbf{x}^{\text{right\_1}}_{\mathbf{k+\frac{1}{2}}} &= \text{FE}(\mathbf{x^0_{k+1}}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{H^1} &= \mathbf{I^{(n)}} - \tfrac{h}{2} \cdot \mathbf{J^0_{k+1}} \\
\mathbf{x^1_{k+1}} &= \mathbf{x^0_{k+1}} - \mathbf{H^{1^{-1}}} \cdot (\mathbf{x}^{\text{right\_1}}_{\mathbf{k+\frac{1}{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}) \\
\varepsilon^{\mathbf{1}}_{\mathbf{k+1}} &= \|\mathbf{x^1_{k+1}} - \mathbf{x^0_{k+1}}\|_\infty \\
& \\
\mathbf{J^1_{k+1}} &= \mathcal{J}(\mathbf{x^1_{k+1}}, t_{k+1}) \\
\mathbf{x}^{\text{right\_2}}_{\mathbf{k+\frac{1}{2}}} &= \text{FE}(\mathbf{x^1_{k+1}}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{H^2} &= \mathbf{I^{(n)}} - \tfrac{h}{2} \cdot \mathbf{J^1_{k+1}} \\
\mathbf{x^2_{k+1}} &= \mathbf{x^1_{k+1}} - \mathbf{H^{2^{-1}}} \cdot (\mathbf{x}^{\text{right\_2}}_{\mathbf{k+\frac{1}{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}) \\
\varepsilon^{\mathbf{2}}_{\mathbf{k+1}} &= \|\mathbf{x^2_{k+1}} - \mathbf{x^1_{k+1}}\|_\infty \\
& \\
&\text{etc.}
\end{aligned}
$$

where $\mathcal{J}$ denotes the Jacobian. For Heun's method (BI2), we find:

$$
\begin{aligned}
\mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}} &= \text{Heun}(\mathbf{x_k}, t_k, \tfrac{h}{2}) \\
\mathbf{x^0_{k+1}} &= \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}} \\
& \\
\mathbf{J^0_{k+1}} &= \mathcal{J}(\mathbf{x^0_{k+1}}, t_{k+1}) \\
\mathbf{x}^{\text{right\_1}}_{\mathbf{k+\frac{1}{2}}} &= \text{Heun}(\mathbf{x^0_{k+1}}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{J}^{\mathbf{0}}_{\mathbf{k+\frac{1}{2}}} &= \mathcal{J}(\mathbf{x}^{\text{right\_1}}_{\mathbf{k+\frac{1}{2}}}, t_{k+\frac{1}{2}}) \\
\mathbf{H^1} &= \mathbf{I^{(n)}} - \tfrac{h}{4} \cdot (\mathbf{J^0_{k+1}} + \mathbf{J}^{\mathbf{0}}_{\mathbf{k+\frac{1}{2}}} \cdot (\mathbf{I^{(n)}} - \tfrac{h}{2} \cdot \mathbf{J^0_{k+1}})) \\
\mathbf{x^1_{k+1}} &= \mathbf{x^0_{k+1}} - \mathbf{H^{1^{-1}}} \cdot (\mathbf{x}^{\text{right\_1}}_{\mathbf{k+\frac{1}{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}) \\
\varepsilon^{\mathbf{1}}_{\mathbf{k+1}} &= \|\mathbf{x^1_{k+1}} - \mathbf{x^0_{k+1}}\|_\infty \\
& \\
\mathbf{J^1_{k+1}} &= \mathcal{J}(\mathbf{x^1_{k+1}}, t_{k+1}) \\
\mathbf{x}^{\text{right\_2}}_{\mathbf{k+\frac{1}{2}}} &= \text{Heun}(\mathbf{x^1_{k+1}}, t_{k+1}, -\tfrac{h}{2}) \\
\mathbf{J}^{\mathbf{1}}_{\mathbf{k+\frac{1}{2}}} &= \mathcal{J}(\mathbf{x}^{\text{right\_2}}_{\mathbf{k+\frac{1}{2}}}, t_{k+\frac{1}{2}}) \\
\mathbf{H^2} &= \mathbf{I^{(n)}} - \tfrac{h}{4} \cdot (\mathbf{J^1_{k+1}} + \mathbf{J}^{\mathbf{1}}_{\mathbf{k+\frac{1}{2}}} \cdot (\mathbf{I^{(n)}} - \tfrac{h}{2} \cdot \mathbf{J^1_{k+1}})) \\
\mathbf{x^2_{k+1}} &= \mathbf{x^1_{k+1}} - \mathbf{H^{2^{-1}}} \cdot (\mathbf{x}^{\text{right\_2}}_{\mathbf{k+\frac{1}{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k+\frac{1}{2}}}) \\
\varepsilon^{\mathbf{2}}_{\mathbf{k+1}} &= \|\mathbf{x^2_{k+1}} - \mathbf{x^1_{k+1}}\|_\infty \\
& \\
&\text{etc.}
\end{aligned}
$$

The algorithm stays basically the same, except that we now need two Jacobians evaluated at different points in time, and the formula for the Hessian becomes a little more involved.

If we assume that the Jacobian remains basically unchanged during one integration step (modified Newton iteration), we can compute both the Jacobian and the Hessian at the beginning of the step, and we find for BI1:

$$\mathbf{J} = \mathcal{J}(\mathbf{x_k}, t_k) \tag{3.47a}$$

$$\mathbf{H} = \mathbf{I^{(n)}} - \frac{h}{2} \cdot \mathbf{J} \tag{3.47b}$$

and for BI2:

$$\mathbf{J} = \mathcal{J}(\mathbf{x_k}, t_k) \tag{3.48a}$$

$$\mathbf{H} = \mathbf{I^{(n)}} - \frac{h}{2} \cdot \mathbf{J} + \frac{h^2}{8} \cdot \mathbf{J}^2 \tag{3.48b}$$

We recognize the pattern. Clearly, the sequence of **H**–matrices is:

$$\mathbf{H_1} = \mathbf{I^{(n)}} - \mathbf{J} \cdot \frac{h}{2} \tag{3.49a}$$

$$\mathbf{H_2} = \mathbf{I^{(n)}} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} \tag{3.49b}$$

$$\mathbf{H_3} = \mathbf{I^{(n)}} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} - \frac{(\mathbf{J} \cdot h)^3}{48} \tag{3.49c}$$

$$\mathbf{H_4} = \mathbf{I^{(n)}} - \mathbf{J} \cdot \frac{h}{2} + \frac{(\mathbf{J} \cdot h)^2}{8} - \frac{(\mathbf{J} \cdot h)^3}{48} + \frac{(\mathbf{J} \cdot h)^4}{384} \tag{3.49d}$$

We may even decide to keep the same Jacobian for several steps in a row, and, in that case, we won't need to compute a new Hessian either, unless we decide to change the step size in between.

Evidently, the sequence in which we execute the forward and the backward semi–steps can be interchanged. The **F**–matrix of the interchanged BI1 algorithm is:

$$\mathbf{F_{MP}} = [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \cdot [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \tag{3.50}$$

which corresponds to the algorithm:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}} \tag{3.51}$$

which is the well–known *implicit midpoint rule*, the one–legged twin of the *trapezoidal rule*. In the same manner, it is possible to generate algorithms of higher orders as well. The two twins are identical in their linear properties, but they behave differently with respect to their nonlinear characteristics. The original BI algorithms are a little more *accurate* than their one–legged twins, since we read out the value of the state at the end of the iteration

rather than after the forward semi–step. On the other hand, the one–legged variety has somewhat better *nonlinear stability* (contractivity) properties, as shown in [3.5].

BI techniques have a certain resemblance with *Padé approximation methods*, which we shall abbreviate as PA methods. The idea behind PA methods is the following: Every numerical ODE solver tries to somehow approximate the analytical **F**–matrix, which would be:

$$\mathbf{F} = \exp(\mathbf{A} \cdot h) \tag{3.52}$$

Equation (3.52) can be rewritten as:

$$\mathbf{F} = \exp(\mathbf{A}\frac{h}{2}) \cdot \exp(\mathbf{A}\frac{h}{2}) = [\exp(\mathbf{A}(-\frac{h}{2}))]^{-1} \cdot \exp(\mathbf{A}\frac{h}{2}) \tag{3.53}$$

According to [3.19], this can be approximated by:

$$\mathbf{F} \approx \mathbf{D}(p,q)^{-1} \cdot \mathbf{N}(p,q) \tag{3.54}$$

with:

$$\mathbf{D}(p,q) = \sum_{j=0}^{q} \frac{(p+q-j)!\ q!}{(p+q)!\ j!\ (q-j)!} \cdot (-\mathbf{A}h)^j \tag{3.55a}$$

$$\mathbf{N}(p,q) = \sum_{j=0}^{p} \frac{(p+q-j)!\ p!}{(p+q)!\ j!\ (p-j)!} \cdot (\mathbf{A}h)^j \tag{3.55b}$$

which, for $p = q$, leads to the following set of **F**–matrices:

$$\mathbf{F_2} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2}] \tag{3.56a}$$

$$\mathbf{F_4} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{12}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{12}] \tag{3.56b}$$

$$\mathbf{F_6} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{10} - \frac{(\mathbf{A} \cdot h)^3}{120}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{(\mathbf{A} \cdot h)^2}{10} + \frac{(\mathbf{A} \cdot h)^3}{120}] \tag{3.56c}$$

$$\mathbf{F_8} = [\mathbf{I^{(n)}} - \mathbf{A} \cdot \frac{h}{2} + \frac{3(\mathbf{A} \cdot h)^2}{28} - \frac{(\mathbf{A} \cdot h)^3}{84} + \frac{(\mathbf{A} \cdot h)^4}{1680}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A} \cdot \frac{h}{2} + \frac{3(\mathbf{A} \cdot h)^2}{28} + \frac{(\mathbf{A} \cdot h)^3}{84} + \frac{(\mathbf{A} \cdot h)^4}{1680}] \tag{3.56d}$$

As the indices indicate, these formulae are all accurate to the double order, i.e., while the individual semi–steps are no longer proper Runge–Kutta

steps (they are themselves only first–order accurate), the overall method attains a considerably higher order of linear accuracy. Due to the symmetry between $\mathbf{D}(p,q)$ and $\mathbf{N}(p,q)$, i.e., due to selecting $p=q$, all these methods are still F–stable.

PA techniques have been intensively studied in [3.13, 3.16]. The problem with them is that an accuracy analysis is only available for the linear case. When exposed to nonlinear systems, the methods may drop several orders of accuracy. Thereby, $\mathbf{F_8}$ may degenerate to an algorithm of merely second order.

The BI algorithms don't share this problem. While they are less accurate than their corresponding PA counterparts for the same computational effort when solving linear problems, their order of accuracy never drops. When using BI4, we shall retain fourth–order accuracy even when solving nonlinear problems, since each of its semi–steps itself is fourth–order accurate for nonlinear as well as linear problems.

Let us check whether we can transform our F–stable BI techniques into a set of more strongly stable BI techniques. The previous set of F–stable backinterpolation techniques did exploit the symmetry of the stability domains of its two semi–steps. However, there is no compelling reason why the two semi–steps have to meet exactly in the middle. The explicit semi–step could span a distance of $\vartheta \cdot h$, and the implicit semi–step could span the remaining distance $(1-\vartheta)\cdot h$. Such a technique is called *ϑ–method.* The resulting algorithm would still be accurate to the same order as its two semi–steps. Using this technique, the stability domain can be shaped.

The case with $\vartheta > 0.5$ is of not much interest, but the case with $\vartheta < 0.5$ is very useful. It produces a series of techniques with ever increasing stability until, at $\vartheta = 0.0$, we obtain a set of L–stable algorithms. The **F**–matrices of the ϑ–methods are:

$$\mathbf{F_1} = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h] \tag{3.57a}$$

$$\mathbf{F_2} = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!}] \tag{3.57b}$$

$$\mathbf{F_3} = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!} - \frac{(\mathbf{A}(1-\vartheta)h)^3}{3!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!} + \frac{(\mathbf{A}\vartheta h)^3}{3!}] \tag{3.57c}$$

$$\mathbf{F_4} = [\mathbf{I^{(n)}} - \mathbf{A}(1-\vartheta)h + \frac{(\mathbf{A}(1-\vartheta)h)^2}{2!} - \frac{(\mathbf{A}(1-\vartheta)h)^3}{3!} + \frac{(\mathbf{A}(1-\vartheta)h)^4}{4!}]^{-1} \cdot [\mathbf{I^{(n)}} + \mathbf{A}\vartheta h + \frac{(\mathbf{A}\vartheta h)^2}{2!} + \frac{(\mathbf{A}\vartheta h)^3}{3!} +$$

$$\frac{(\mathbf{A}\vartheta h)^4}{4!}] \tag{3.57d}$$

Figure 3.4 shows the stability domains of the BI algorithms that result for:

$$\vartheta = 0.4 \tag{3.58}$$

These methods result in very nice stability domains with large unstable

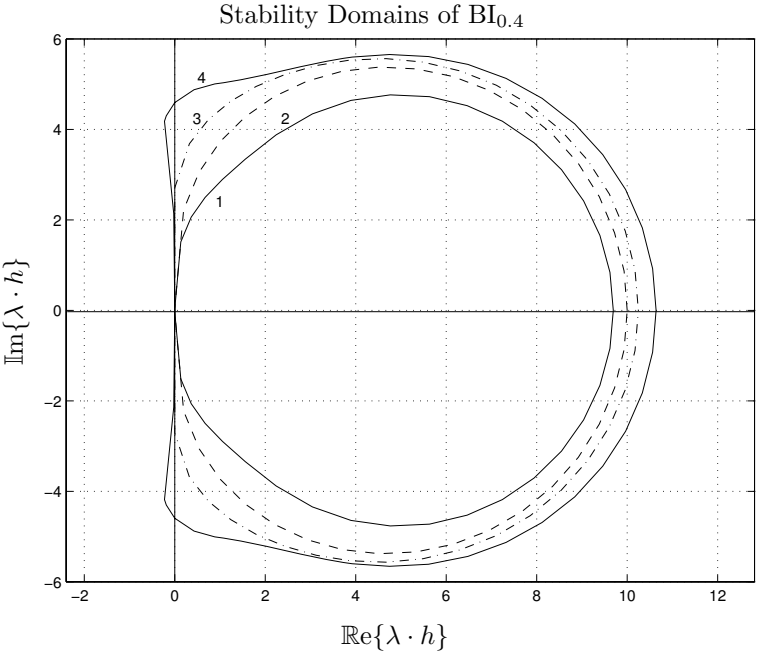

FIGURE 3.4. Stability domains of backinterpolation ϑ–methods.

regions in the right half $(\lambda \cdot h)$–plane. The selection of a good value for ϑ is a compromise. ϑ should be chosen large enough to generate meaningfully large unstable regions in the right half $(\lambda \cdot h)$–plane, yet small enough to dampen out the high frequency components appropriately in the left half $(\lambda \cdot h)$–plane. The fourth–order algorithm of Fig.3.4 is no longer A–stable. Its unstable region reaches slightly into the left half $(\lambda \cdot h)$–plane. Such a method is called (A,α)–stable, where α denotes the largest angle away from the negative real axis that contains only stable territory. $BI4_{0.4}$ is (A,86°)–stable.

The previously mentioned BRK algorithms are special cases of this new class of ϑ–methods with $\vartheta = 0$, and the explicit RK algorithms are special cases of this class of ϑ–methods with $\vartheta = 1$. The F–stable BI algorithms are special cases with $\vartheta = 0.5$.

A set of L–stable BI algorithms with $\vartheta > 0$ can be constructed by choosing the approximation order of the implicit semi–step one order higher than that of the explicit semi–step. For stiff engineering problems, a BI4 algorithm using an RK4 method for its explicit semi–step and a BRK5 method for its implicit semi–step together with $\vartheta = 0.45$ turns out to be generally an excellent choice [3.24]. This method will be abbreviated as $\mathrm{BI4}/5_{0.45}$.

3.8 Accuracy Considerations

It is now time to revisit the problem of the approximation accuracy, and discuss this issue with a little more insight and detail.

We start out with our standard linear test problem:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \quad ; \quad \mathbf{x}(t_0) = \mathbf{x_0} \tag{3.59}$$

with the same $\mathbf{A}$–matrix that we already used when constructing the stability domain:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 \\ -1 & 2\cos(\alpha) \end{pmatrix} \tag{3.60}$$

and with the standardized initial condition:

$$\mathbf{x_0} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{3.61}$$

Let us apply the following fourth–order Runge–Kutta algorithm:

```
function [x] = rk4(A, h, x0)
  %
  h2 = h/2;   h6 = h/6;
  x(:,1) = x0;
  %
  for i = 1 : 10/h,
    xx = x(:,i);
    k1 = A * xx;
    k2 = A * (xx + h2 * k1);
    k3 = A * (xx + h2 * k2);
    k4 = A * (xx + h * k3);
    x(:,i+1) = xx + h6 * (k1 + 2 * k2 + 2 * k3 + k4);
  end
return
```

to simulate this system across 10 seconds of simulated time.

We want to compare the simulated solution $\mathbf{x_{simul}}$ with the analytical solution:

$$\mathbf{x_{anal}} = \exp(\mathbf{A} \cdot (t - t_0)) \cdot \mathbf{x_o} \tag{3.62}$$

We define the global error as follows:

$$\varepsilon_{\text{global}} = \|\mathbf{x_{anal}} - \mathbf{x_{simul}}\|_\infty \tag{3.63}$$

and vary the step size, h, until the global error matches a prescribed tolerance:

```
function [hmax] = hh2(alpha, hlower, hupper, tol)
  %
  A = aa(alpha);
  x0 = ones(2,1);
  maxerr = 1E-6;  err = 100;
  while err > maxerr,
    h = (hlower + hupper)/2;
    xsimul = rk4(A,h,x0);
    for i = 0 : 10/h,
      xanal(:,i+1) = expm(A*h*i)*x0;
    end,
    eglobal = norm(xanal - xsimul,'inf');
    err = eglobal - tol;
    if err > 0,
      hupper = h;
    else
      hlower = h;
    end,
    err = abs(err);
  end
  hmax = h;
return
```

This routine looks very similar to the one that was presented in Chapter 2 for the computation of the stability domain.

We again sweep over a range of α values, and plot h_{max} as a function of α in polar coordinates. Figure 3.5 shows the results of our efforts.

The chosen error tolerance was $tol = 10^{-4}$.

Just like the stability domain, the *accuracy domain* can be plotted in the $(\lambda \cdot h)$–plane. If we were to select a pair of eigenvalues in the λ–plane twice as far away from the origin and use a step size, h, that is half as large, we would get exactly the same accuracy. This happens because both the analytical $\mathbf{F}$–matrix:

$$\mathbf{F_{anal}} = \exp(\mathbf{A} \cdot h) \tag{3.64}$$

and its numerical counterpart:

$$\mathbf{F_{RK4}} = \mathbf{I^{(n)}} + \mathbf{A} \cdot h + \frac{(\mathbf{A} \cdot h)^2}{2!} + \frac{(\mathbf{A} \cdot h)^3}{3!} + \frac{(\mathbf{A} \cdot h)^4}{4!} \tag{3.65}$$

are functions of $\mathbf{A} \cdot h$.

On first sight, this seems to be an important discovery — accuracy can be treated in the same way as stability. Unfortunately, this is not quite true.

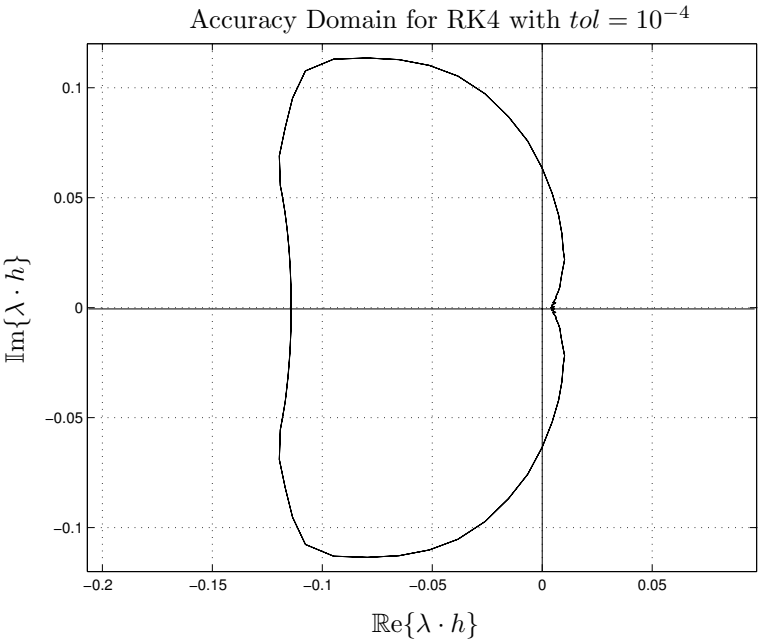

FIGURE 3.5. Accuracy domain of explicit fourth–order Runge–Kutta method.

The problem is that the accuracy domain depends heavily on the selected initial condition. The largest step size that can be chosen for a prescribed tolerance is approximately inverse proportional to the largest gradient in the simulation, and since:

$$\|\dot{\mathbf{x}}\|_\infty \approx \|\mathbf{A}\|_\infty \cdot \|\mathbf{x}\|_\infty \tag{3.66}$$

$h_{\max}$ is also approximately inverse proportional to any norm of the initial condition for stable systems. Notice the asymmetry of the accuracy domain with respect to the imaginary axis. If the poles are located in the analytically stable left half λ–plane, the transients die out with time, and the largest errors are committed early on in the game. On the other hand, if the poles are located in the analytically unstable right half λ–plane, the transients grow larger and larger, and the committed errors grow accordingly for any fixed step size. This is the major reason why an accurate simulation of analytically unstable systems is a quite expensive enterprize as we have to fight accumulation errors in that case.

Figure 3.6 shows the accuracy domains of the RK4 algorithm for three different error tolerances: $tol_1 = 10^{-4}$ (as before), $tol_2 = 10^{-3}$, $tol_3 = 10^{-2}$, and finally, $tol_4 = 10^{-1}$. The stability domain has been plotted on top of the three accuracy domains.

It can be noticed that all three accuracy domains are safely within the numerically stable region, at least as far as the left–half $\lambda \cdot h$–plane is

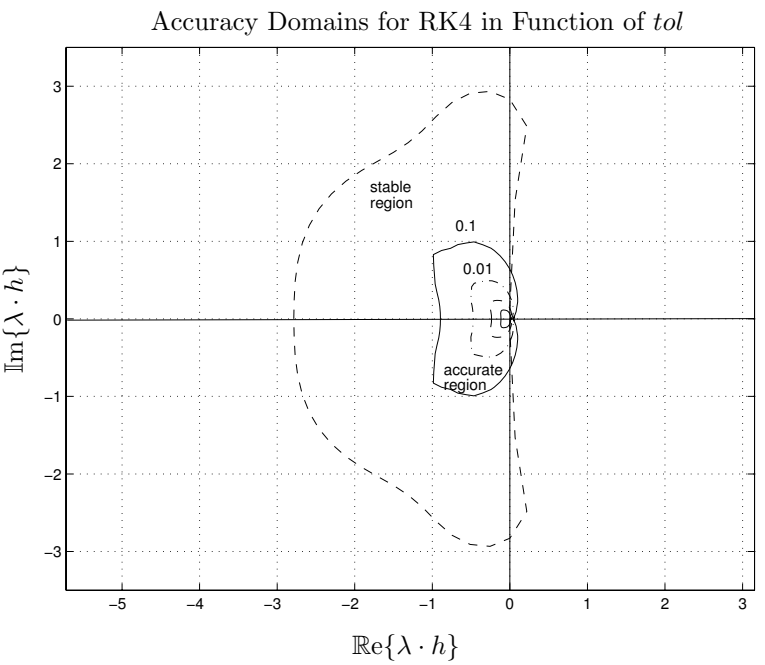

FIGURE 3.6. Accuracy domains of RK4 for different tolerance values.

concerned. However, this is deceiving. As the norm of the state vector decays for analytically stable eigenvalue locations, the step size could be chosen larger and larger to achieve the same tolerance.

The solution decays as $\exp(-\sigma \cdot t)$. Let us assume that the two eigenvalues are located at -1.0. In this case, the damping, σ, is 1.0. The norm of the chosen initial condition is $\|\mathbf{x_0}\| = 1.0$. With this initial condition, the accuracy domain intersects the negative real axis at about -0.1. Thus, when the norm of the state vector has decayed to roughly $\|\mathbf{x}\| = 0.04$, the accuracy domain has grown to the size of the stability domain, and from then onward, the step size will actually be controlled by the numerical stability requirements, and no longer by accuracy requirements. This happens after about 3.2 seconds of simulated time ... and this is approximately, how long we would usually simulate such a system before the trajectories become utterly uninteresting, as long as no input function adds to the "excitement."

If we now simulate a system, in which fast phenomena are superposed to slow phenomena, then the simulation run length will be determined by the slowest time constant whereas the step size will be dictated by the numerical stability requirements of the fastest component. This is the problem that was addressed under the heading *stiff system*. As the above calculation shows, it doesn't take a very large difference in time scales

before stiffness becomes a problem. A time scale factor of 10 is probably still acceptable, one of 100 is already quite problematic. Many engineering applications call for simulations with time scale differences of several orders of magnitude. A typical example of such applications are electronic circuits. This is one of the reasons why *all* circuit simulators use implicit integration schemes.

However, let us now return to the question of accuracy. As Fig.3.6 shows, the difference in step size needed to improve the accuracy by a factor of 10 is not very large. By cutting the step size in half, we can improve the accuracy by one order of magnitude.

Let us explore this thought a little further. To this end, we shall keep the eigenvalues at -1.0, and we shall vary the step size to see how accurate the simulation will be. Figure 3.7 shows the results of this experiment. I plotted the number of function evaluations, which equals the number of stages of the algorithm multiplied by the number of steps performed during the simulation as a function of the achieved accuracy.

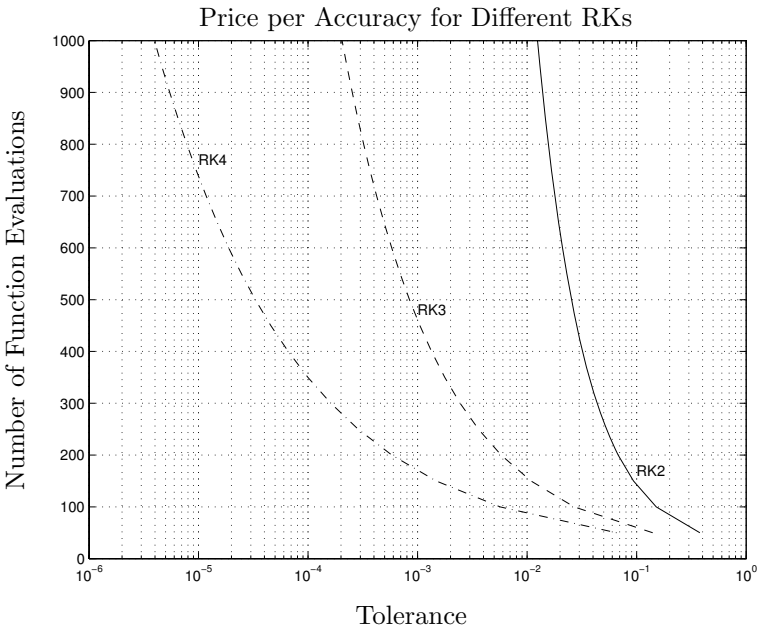

FIGURE 3.7. Simulation cost as a function of accuracy for different RKs.

It turns out that RK4 was cheaper than RK3, which in turn was cheaper than RK2 for *all* error tolerance values. This is not very surprising. By cutting the step size in half, we double the number of function evaluations needed to complete the simulation run. For the same "money," we could have kept the step size the same and instead doubled the accuracy order (at least for low–order algorithms). Thereby we would have gained two

orders of magnitude in improved accuracy as opposed to only one order by reducing the step size.

Euler's performance is not shown in Fig.3.7. We also tried RK1 (FE), but we couldn't get a global accuracy of 10% (corresponding to a local accuracy of roughly 1%) for below 1000 function evaluations.

For a required global accuracy of 10%, the three algorithms shown in Fig.3.7 have a quite comparable price. For a 1% accuracy, RK2 is already out of the question, whereas both RK3 and RK4 still perform decently. For a 0.1% global accuracy, RK3 has become expensive, while RK4 still performs acceptably well.

Remembering that we usually control the *local* integration error rather than the *global* integration error, which is roughly one order of magnitude better, we see that indeed RK4 will work well for local errors of up to about 10^{-4}. If we want to compute more accurately than that, we definitely should turn to higher–order algorithms.

Notice that all these computations were performed on a 32 bit machine in double precision, thus, the roundoff error is negligible in comparison with the truncation error. Just for fun, we repeated the same computations in simulated single precision by chopping eight digits off the state vector at the end of each integration step using the MATLAB statement:

$$\mathbf{x} = \text{chop}(\mathbf{x}, 8)$$

The results of this effort are shown in Fig.3.8.

If the accuracy requirements are low, the simulation can use large step sizes, and therefore, roundoff is not a problem. Consequently, the algorithms behave in the same way as before. However, for higher accuracy requirements, roundoff sets in (due to small step sizes), and accordingly, a further reduction of the step size will not help to meet the required accuracy. No algorithm of any order will get us a global accuracy of better than 10^{-5} in this case.

To summarize this discussion, problems with roundoff make simulation in single precision on a 32 bit machine quite problematic for *any* model using *any* integration algorithm. Accuracy requirements put a lower bound on the approximation order of the integration algorithm. A local error tolerance of $\varepsilon_{\text{local}} = 10^{-k}$ calls for at least a k^{th}–order integration algorithm. Lower–order algorithms aren't necessarily cheaper even if the accuracy requirements aren't stringent. Accuracy domains aren't quite as handy as one could hope for due to their heavy dependence on the chosen initial conditions.

Let us see whether we can come up with yet another tool to describe the accuracy of an integration algorithm, a tool that isn't plagued by the same flaw as the accuracy domain.

Let us look once more at our standard linear test problem of Eq.3.59 with the analytical solution of Eq.3.62. Obviously, the analytical solution

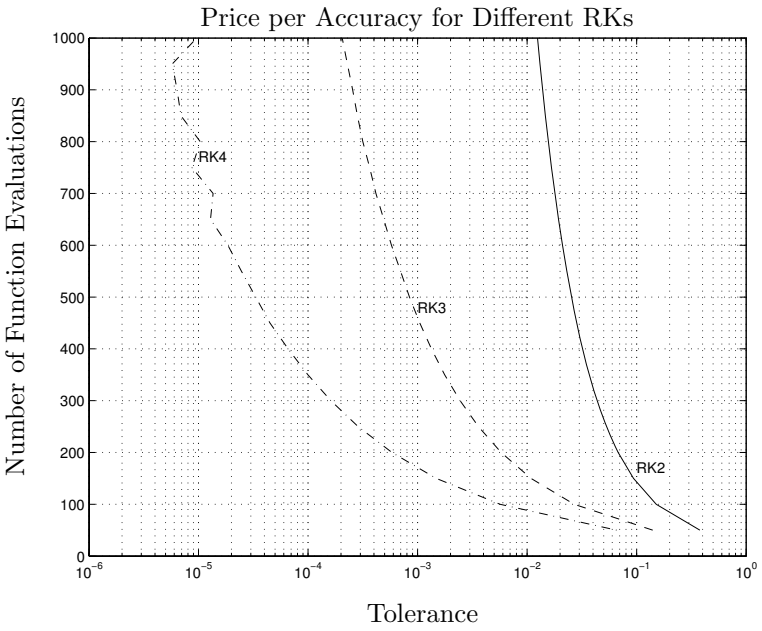

FIGURE 3.8. Cost for accuracy for different RKs in single precision.

is correct for any value of t, t_0, and $\mathbf{x_0}$, and in particular, it is true for $t_0 = t_k$, $\mathbf{x_0} = \mathbf{x_k}$ and $t = t_{k+1}$. With this substitution, we find:

$$\mathbf{x_{k+1}} = \exp(\mathbf{A} \cdot h) \cdot \mathbf{x_k} \tag{3.67}$$

Thus, the analytical $\mathbf{F}$–matrix of this system is:

$$\mathbf{F_{anal}} = \exp(\mathbf{A} \cdot h) \tag{3.68}$$

For a scalar problem, we can specialize the general solution of Eq.3.62 into:

$$x(t) = c_1 \cdot \exp(-\sigma \cdot t) \cdot \cos(\omega \cdot t) + c_2 \cdot \exp(-\sigma \cdot t) \cdot \sin(\omega \cdot t) \tag{3.69}$$

where σ is the distance of the eigenvalue from the imaginary axis and is called the *damping* of the eigenvalue, whereas ω is the distance of the eigenvalue from the real axis and is called the *eigenfrequency* of the eigenvalue.

It is easy to show that the eigenvalues of the analytical $\mathbf{F_{anal}}$–matrix are related to those of the $\mathbf{A}$–matrix through:

$$\text{Eig}\{\mathbf{F_{anal}}\} = \exp(\text{Eig}\{\mathbf{A}\} \cdot h) \tag{3.70}$$

or:

$$\lambda_{disc} = \exp(\lambda_{cont} \cdot h) = \exp((-\sigma + j \cdot \omega) \cdot h) = \exp(-\sigma \cdot h) \cdot \exp(j \cdot \omega \cdot h) \tag{3.71}$$

Consequently, the damping, i.e., the distance of an eigenvalue from the imaginary axis in the λ–plane maps into a distance from the origin in the $\exp(\lambda \cdot h)$–plane, whereas the eigenfrequency, i.e., the distance of an eigenvalue from the real axis in the λ–plane maps into an angle away from the positive real axis in the $\exp(\lambda \cdot h)$–plane.

Notice that we just introduced a new plane: the $\exp(\lambda \cdot h)$–plane. Control engineers call this plane the z–domain, where:

$$z = \exp(\lambda \cdot h) \tag{3.72}$$

Since even the analytical $\mathbf{F}_{\mathbf{anal}}$–matrix depends on the step size, h, it makes sense to introduce a *discrete damping*, $\sigma_d = h \cdot \sigma$, and a *discrete frequency*, $\omega_d = h \cdot \omega$. Obviously, we can write:

$$|z| = \exp(-\sigma_d) \tag{3.73a}$$
$$\angle z = \omega_d \tag{3.73b}$$

Now, let us replace the analytical $\mathbf{F}_{\mathbf{anal}}$–matrix by the one that belongs to the numerical integration routine, $\mathbf{F}_{\mathbf{simul}}$. The numerical $\mathbf{F}_{\mathbf{simul}}$–matrix is either a rational (for implicit integration algorithms) or a polynomial (for explicit integration algorithms) approximation of the analytical $\mathbf{F}_{\mathbf{anal}}$–matrix. We define:

$$\hat{z} = \exp(\hat{\lambda}_d) \tag{3.74}$$

with:

$$\hat{\lambda}_d = -\hat{\sigma}_d + j \cdot \hat{\omega}_d \tag{3.75}$$

Therefore:

$$|\hat{z}| = \exp(-\hat{\sigma}_d) \tag{3.76a}$$
$$\angle \hat{z} = \hat{\omega}_d \tag{3.76b}$$

As $\hat{z}$ approximates z, so must $\hat{\sigma}_d$ be an approximation of σ_d, and $\hat{\omega}_d$ must approximate ω_d. It makes sense to study the relationship between the analytical discrete damping, σ_d, on the one hand, and the numerical discrete damping, $\hat{\sigma}_d$, on the other. Similarly, we can study the relationship between the analytical discrete frequency, ω_d, and the numerical discrete frequency, $\hat{\omega}_d$.

We can define:

$$\varepsilon_\sigma = \sigma_d - \hat{\sigma}_d \tag{3.77a}$$
$$\varepsilon_\omega = \omega_d - \hat{\omega}_d \tag{3.77b}$$

where ε_σ denotes the *damping error*, and ε_ω denotes the *frequency error* committed by the numerical integration algorithm.

Since the case of all continuous eigenvalues being negative and real occurs so frequently (e.g. *all* thermal systems are of that nature), it is worthwhile to study the damping error when moving an eigenvalue left or right along the negative real axis. We can plot σ_d and $\hat{\sigma}_d$ as functions of σ_d itself. The following program will compute $\hat{\sigma}_d$ for any single–step integration algorithm.

```
function [sdhat] = damp(sd, algor)
  %
  f = ff(−sd, 1, algor);
  sdhat = −log(f);
return
```

We can then sweep across a range of σ_d–values, and plot both $-\sigma_d$ and $-\hat{\sigma}_d$ against $-\sigma_d$. Such a graph is called a *damping plot*. Figure 3.9 shows the damping plot of RK4.

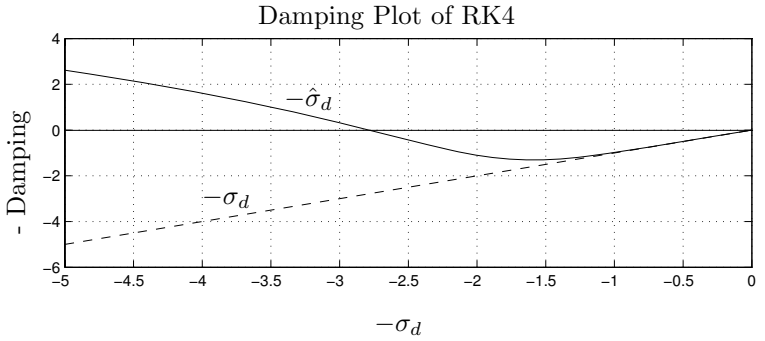

FIGURE 3.9. Damping plot of RK4.

In the vicinity of the origin, the numerical damping value, $\hat{\sigma}_d$, follows the analytical damping, σ_d, very well. However, already for very moderate eigenvalue locations, the numerical damping behavior deviates drastically from the analytical one, and somewhere around $\sigma_d = 2.8$, the numerical damping becomes negative, which coincides with the border of numerical stability. The area where the approximation of σ_d by $\hat{\sigma}_d$ is accurate, is called the *asymptotic region* of the integration algorithm.

Let us now look at the damping plot of BI4. This plot is shown in Fig.3.10.

Since BI4 is A–stable, the numerical damping stays positive for all values of λ. Since BI4 is F–stable, the numerical damping approaches zero as λ approaches $-\infty$. Figure 3.11 shows the damping plot of the ϑ–method with $\vartheta = 0.4$.

Now, the damping no longer approaches zero, but it doesn't go to infinity either. From Eq.3.57d, we can conclude that:

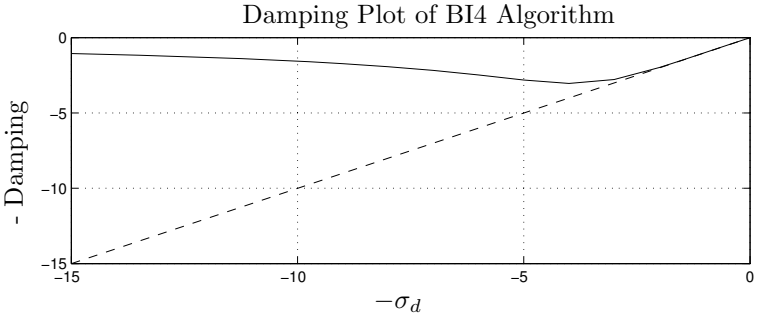

FIGURE 3.10. Damping plot of BI4.

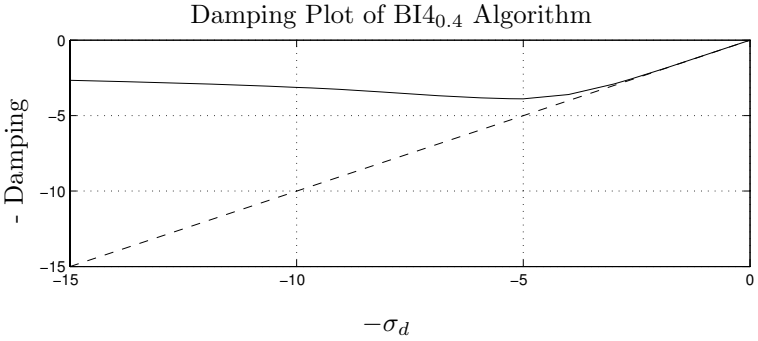

FIGURE 3.11. Damping plot of BI4 with $\vartheta = 0.4$.

$$\hat{\sigma}_d(-\infty) = -4 \cdot \log(\frac{\vartheta}{1-\vartheta}) \tag{3.78}$$

Consequently, for $\vartheta = 0.5$, we find that $\hat{\sigma}_d(-\infty) = 0.0$, a fact that we already knew (F–stability), and for $\vartheta = 0.4$, we find that $\hat{\sigma}_d(-\infty) = 1.6219$. The algorithm with $\vartheta = 0.0$, i.e., BRK4, is L–stable, since $\hat{\sigma}_d(-\infty) \to -\infty$. Unfortunately, the true power of L–stability is not as glamorous as one might think, as the BRK4 damping plot of Fig.3.12 demonstrates. Although BRK4 is L–stable, the increase in damping when moving the pole to the left is despairingly slow as a result of the logarithm function in Eq.(3.78). For $\sigma_d = -10^{-9}$, we find that $\hat{\sigma}_d \approx -80$. L–stability is thus somewhat overrated.

Just for completeness, let us draw the damping plot of IEX4 as well. It is shown inf Fig.3.13. Now, this is interesting. Somewhere around $\sigma_d = -6.7$, the numerical damping, $\hat{\sigma}_d$, intersects with the analytical damping, σ_d, thus the damping error is exactly equal to zero.

Let us extend our search to the entire complex $(\lambda \cdot h)$–plane. Figure 3.14 plots $\hat{\sigma}_d - \sigma_d$ over the complex plane.

Points on Fig.3.14 with positive amplitude represent a surplus in numer-

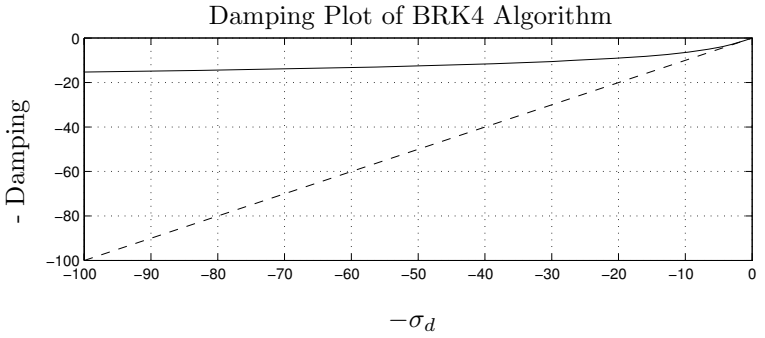

FIGURE 3.12. Damping plot of BRK4.

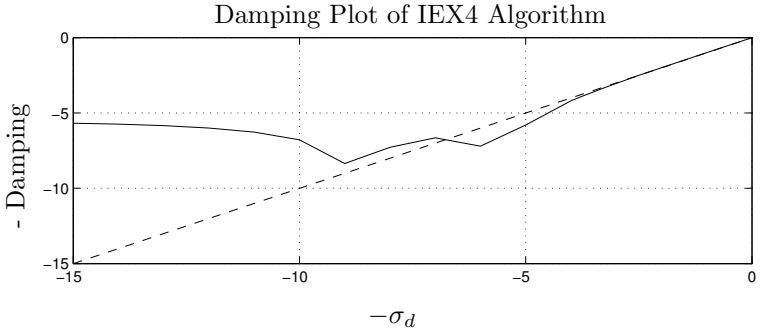

FIGURE 3.13. Damping plot of IEX4.

ical damping, whereas points with negative amplitude represent a lack in numerical damping. We can strip away the magnitude information, and only display the sign of the damping error. This is shown in Fig.3.15. '+' means that there is surplus damping at this point, whereas '−' indicates that there is not enough damping.

There obviously exists a locus of at least partially connected points of the $\lambda \cdot h$–plane, where the damping error is zero. This locus has been plotted in Fig.3.16 for the IEX4 method. Such a locus is called an *order star* [3.13, 3.16, 3.23].

Figure 3.14 shows that there exist points where the simulated damping $\hat{\sigma}_d$ is infinite. Contrary to $\mathbf{F_{anal}}$, which is a very smooth function, any rational function approximation $\mathbf{F_{simul}}$ has poles, i.e., points with infinite numerical damping.

The **F**–matrix of IEX4 can be written as follows:

$$\begin{aligned} \mathbf{F} \quad = \quad & -\frac{1}{6} \cdot [\mathbf{I^{(n)}} - \mathbf{A} \cdot h]^{-1} + 4 \cdot [\mathbf{I^{(n)}} - \frac{\mathbf{A} \cdot h}{2}]^{-2} \\ & -\frac{27}{2} \cdot [\mathbf{I^{(n)}} - \frac{\mathbf{A} \cdot h}{3}]^{-3} + \frac{32}{3} \cdot [\mathbf{I^{(n)}} - \frac{\mathbf{A} \cdot h}{4}]^{-4} \end{aligned} \tag{3.79}$$

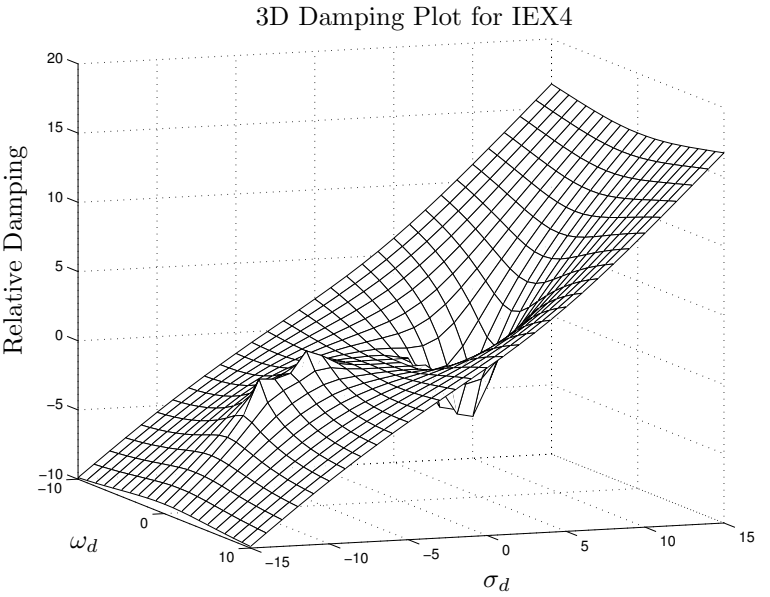

FIGURE 3.14. 3D–plot of damping error for IEX4.

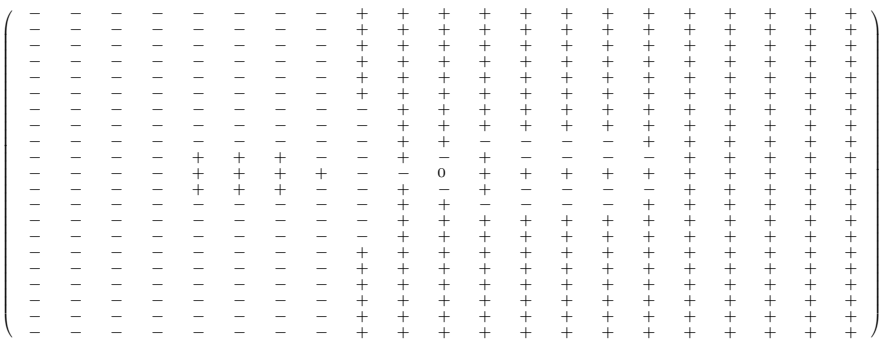

FIGURE 3.15. Damping error sign for IEX4.

Let us analyze the scalar case with:

$$q = \lambda \cdot h$$

We can write:

$$f = -\frac{1}{6} \cdot \frac{1}{1-q} + 4 \cdot \frac{1}{(1-q/2)^2} - \frac{27}{2} \cdot \frac{1}{(1-q/3)^3} + \frac{32}{3} \cdot \frac{1}{(1-q/4)^4} \quad (3.80)$$

f is a rational function with 10 poles located at +1 (single pole), +2 (double

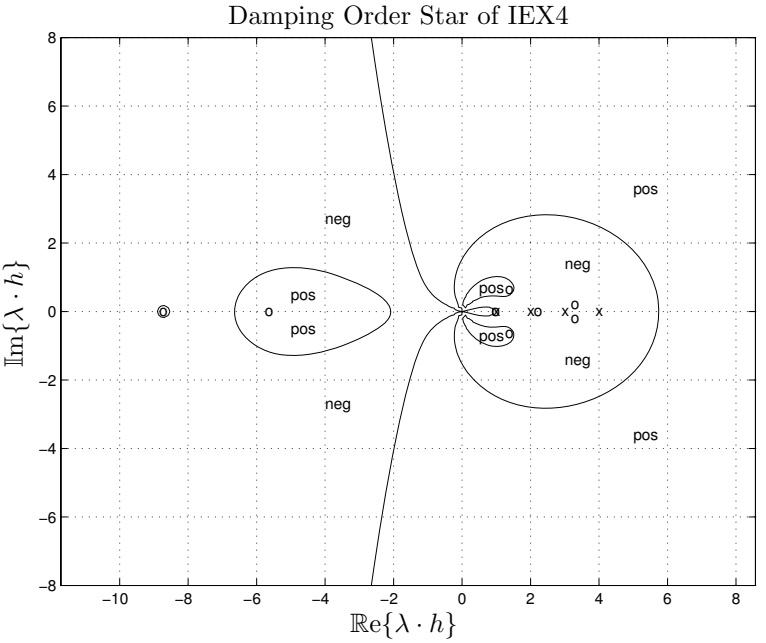

FIGURE 3.16. Damping order star for IEX4.

pole), +3 (triple pole), and +4 (quadruple pole). As with any respectable integration algorithm, all the poles are in the right half complex plane. The pole locations are marked on Fig.3.16 as '×'. f has also nine zeros, which are located at $3.277 \pm 0.2155j$, 2.1918, $1.3622 \pm 0.6493j$, 0.9562, −5.6654, −8.7506, and −65.0105. They were marked on Fig.3.16 as 'o'. Since:

$$\hat{\sigma}_d = -\log(|f|) \tag{3.81}$$

zeros show up on the damping plot (Fig.3.13) as negative poles, and on the 3D–plot (Fig.3.14) as positive poles.

Figure 3.14 shows a very rugged terrain just to the right of the origin of the complex plain. For this reason, extrapolation techniques surely aren't suitable for the integration of unstable systems.

In stiff system integration, we requested that $\hat{\sigma}_d \to -\infty$ as $\lambda \to \infty$. From Eq.(3.81), we conclude that $f \to 0$ as $q \to \infty$. This is obviously only possible if f is a strictly proper rational function. This is the reason why explicit integration algorithms can never be L–stable.

IEX4 also has many zeros, some of which are even in the left half complex plain. This can pose a problem. The BRK algorithms don't have any zeros. This may sometimes be beneficial.

Let us now look at the damping order star of a backinterpolation technique. It is shown for BI4 on Fig.3.17.

The terrain in the vicinity of the origin (the asymptotic region) is much

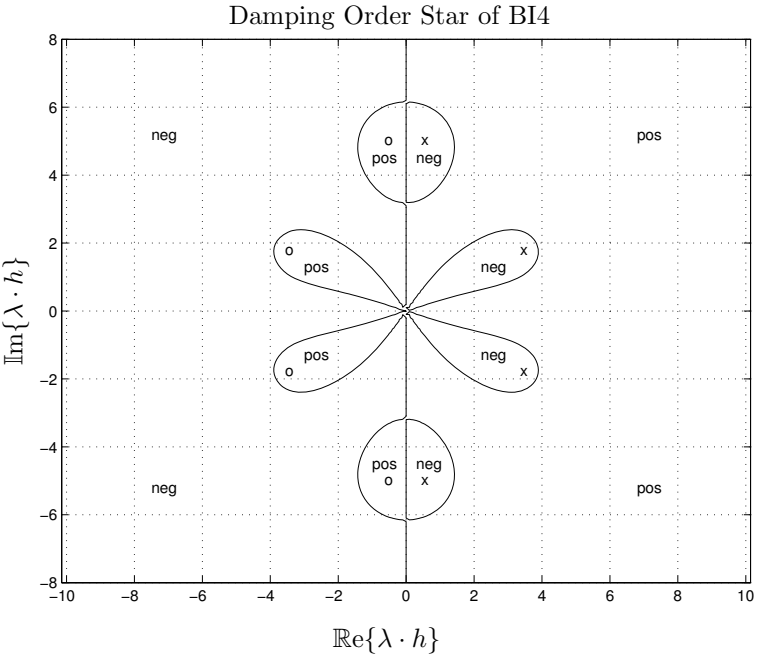

FIGURE 3.17. Damping order star for BI4.

smoother than in the case of IEX4. BI4 has four poles and four zeros that are marked on Fig.3.17. As in the case of IEX4, BI4 has zeros in the left half complex plane, but at least none on the negative real axis.

Let us now discuss the *frequency error*. It may be worthwhile to study the frequency error when moving an eigenvalue up or down along the positive imaginary axis. We can plot ω_d and $\hat{\omega}_d$ as functions of ω_d itself. The following program will compute $\hat{\omega}_d$ for any single–step integration algorithm.

function [*wdhat*] = freq(*wd*, *algor*)
  %
  f = ff(*wd*, 1, *algor*);
  wdhat = atan2(imag(*f*),real(*f*));
return

We can then sweep across a range of ω_d–values, and plot both ω_d and $\hat{\omega}_d$ against ω_d. Such a graph is called a *frequency plot*. Figure 3.18 shows the frequency plot of RK4.

The frequency plot of RK4 seems to exhibit a discontinuity around $\omega_d = 2.5$. Yet, the plot is misleading. The discrete frequency ω_d is 2π–periodic. The "discontinuity" simply represents a jump of $\hat{\omega}_d$ from $+\pi$ to $-\pi$. We could easily compensate for the jump, and get a $\hat{\omega}_d$ curve that is totally smooth, as long as the chosen path for ω_d doesn't lead through either a pole or a zero.

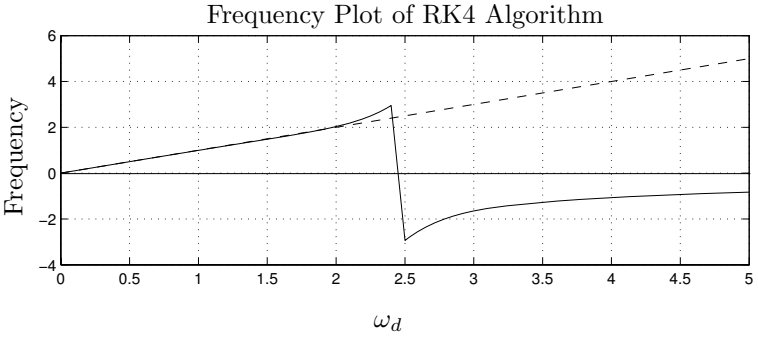

FIGURE 3.18. Frequency plot of RK4.

Does this mean that we can plot a *frequency order star* in the same fashion as we drew the damping order star? Unfortunately, there still is a problem. The damping order star is a contour plot, i.e., a plot of an equipotential line, namely that for the zero potential. Contour plots, however, can only be drawn for *potential fields*, i.e., single-valued functions in two real variables or one complex variable. The damping error function is indeed single–valued. Unfortunately, the same does not hold for the frequency error function. For each value of the complex independent variable $s_d = \sigma_d + j \cdot \omega_d$, the dependent variable $\hat{\omega}_d$ assumes infinitely many values, as $\hat{\omega}_d$ is 2π–periodic.

Can we fix the problem by eliminating the artificial discontinuities, as proposed above? Unfortunately, this does not solve the problem. If we choose a closed path in s_d that encircles either a pole or a zero, the total frequency contribution around the pole or zero is $\pm 2\pi$. The terrain of the $\hat{\omega}_d$ function in the vicinity of any pole or zero looks like an infinitely long spiral staircase. Consequently, the $\hat{\omega}_d$–function is not a potential field.

The $\hat{\omega}_d$–function can be turned into a potential field by limiting its range to e.g. $(-\pi, +\pi]$, in which case we can indeed plot a frequency order star just as easily as in the case of the damping order star.

Figure 3.19 shows the frequency order star of BI4, and Fig.3.20 exhibits the frequency order star of IEX4.

Frequency order stars aren't depicted often, although they should be, and were it only for their exquisite beauty.

What can we do with these tools? We have seen that both the damping plot and the frequency plot exhibit asymptotic regions, i.e., regions, in which $\mathbf{F_{simul}}$ deviates only little from $\mathbf{F_{anal}}$ both in terms of the absolute value (damping) and in terms of the phase value (frequency). The asymptotic region surrounds the origin. Figure 3.21 shows the asymptotic regions of the RK4 algorithm. The top graph shows the asymptotic region for $s_d = -\sigma_d$, whereas the bottom graph depicts the asymptotic region for $s_d = j \cdot \omega_d$.

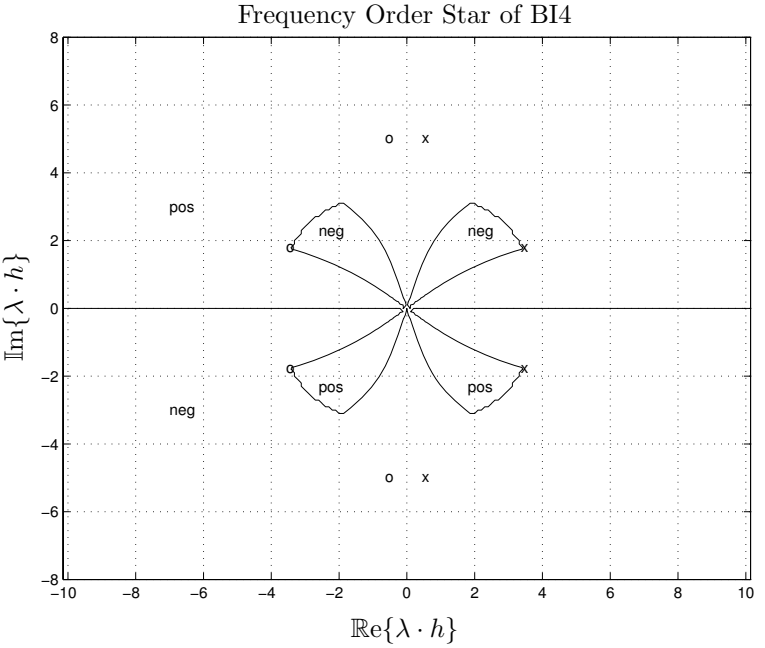

FIGURE 3.19. Frequency order star of BI4.

Hence it may make sense to define an error function that accounts for both types of errors, the damping error and the frequency error, simultaneously, e.g.:

$$os_{err} = |\sigma_d - \hat{\sigma}_d| + |\omega_d - \hat{\omega}_d| \tag{3.82}$$

If both the damping error and the frequency error are defined such that they form potential fields, then the *order star error* function, os_{err}, must also form a potential field. Consequently, we can draw equipotential lines of $os_{err} = 10^{-4}$, $os_{err} = 10^{-3}$, and $os_{err} = 10^{-2}$ as contour plots, and superpose them onto the same graph. Figure 3.22 depicts the order star accuracy domain of the RK4 algorithm.

The *order star accuracy domain* has an important advantage over the previously introduced *accuracy domain*. It is totally independent of the problem to be solved or the initial conditions being used. It only depends on the algorithm itself. It is a metric that is as "pure" as the stability domain.

Notice that the order star accuracy domain is not asymmetric w.r.t. the imaginary axis. The reason is simple. The order star accuracy domain compares the *analytical* solution of the original continuous–time problem with the equally *analytical* solution of the derived discrete–time problem. Consequently, it only accounts for the truncation error. It does not consider

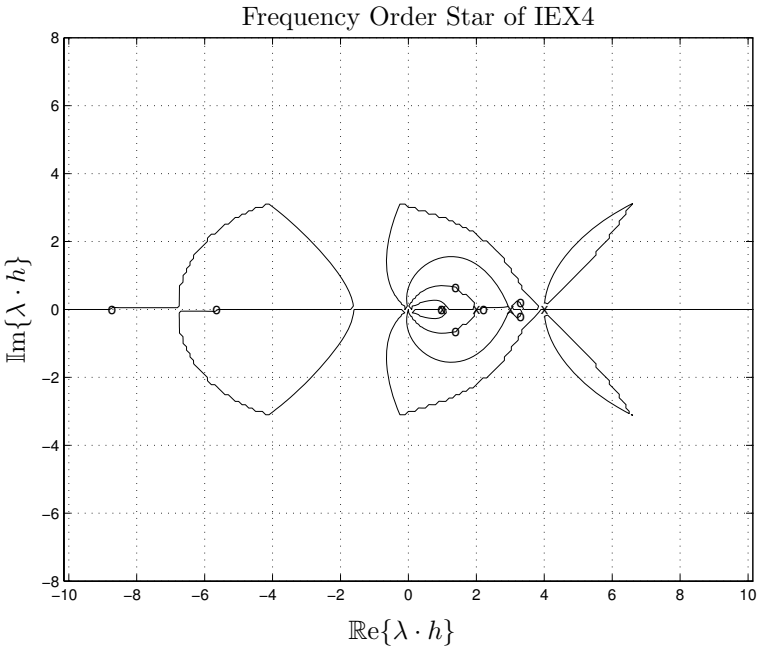

FIGURE 3.20. Frequency order star of IEX4.

either the roundoff or the accumulation errors. The roundoff error is relatively harmless, as it can be easily controlled by the length of the mantissa used in the numerical computation. The accumulation error, on the other hand, is anything but harmless. It is responsible for the narrow region of accurate simulations in the right half plane of the accuracy domain. As the order star accuracy domain doesn't account for accumulation errors, it allows for an equally large region of accurate computations in the right–half $\lambda \cdot h$–plane as in the left–half $\lambda \cdot h$–plane.

Hence and in spite of its other shortcomings, the previously introduced accuracy domain is a considerably more conservative measure of the ability of a code to perform accurate simulations than the newly introduced order star accuracy domain.

Figure 3.23 shows the order star accuracy domain of the IEX4 algorithm. This time, the order star accuracy domain is indeed asymmetrical to the imaginary axis. However, the reason here is not related to the accumulation errors, but rather to the poles and zeros of this algorithm that are located close to the origin in the right–half $\lambda \cdot h$–plane, leading to a very rugged terrain of the order star in this region. Hence the IEX4 algorithm has no chance of simulating accurately unstable systems, even irrespective of the problem of error accumulation.

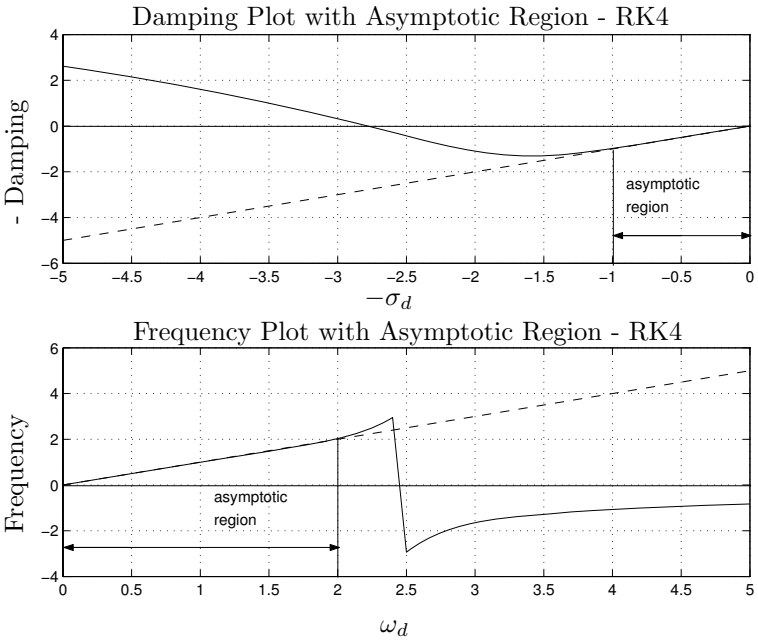

FIGURE 3.21. Asymptotic regions of RK4.

3.9 Step–size and Order Control

Although we have talked about the various sources of errors that can corrupt our numerical integration results, we have done nothing so far to contain them. We know that smaller step sizes will, in general, lead to smaller integration errors at a higher computational cost, whereas larger step sizes will lead to larger errors at a smaller cost. The right step size is a compromise between containment of error and cost. However, we don't know how to choose the most appropriate step size. As engineers, we certainly know how accurate we need our results to be, and we also know how much money we are willing to spend in order to get them, but while this knowledge indirectly determines the step size, we don't have a good algorithm yet that would translate the error/cost knowledge into an adequate value for the step size. This problem will be discussed next.

Since the relationship between error/cost on the one hand and the step size on the other depends heavily on the numerical properties of the system to be integrated, it is not clear that a fixed step size will lead to the same integration error throughout the simulation. It may well be that a variation of the step size during the simulation period in order to keep the error at a constant level close to the maximum allowed error tolerance can reduce the overall cost of the simulation. This observation leads to the demand for *variable–step integration algorithms*, which in turn call for a *step–size*

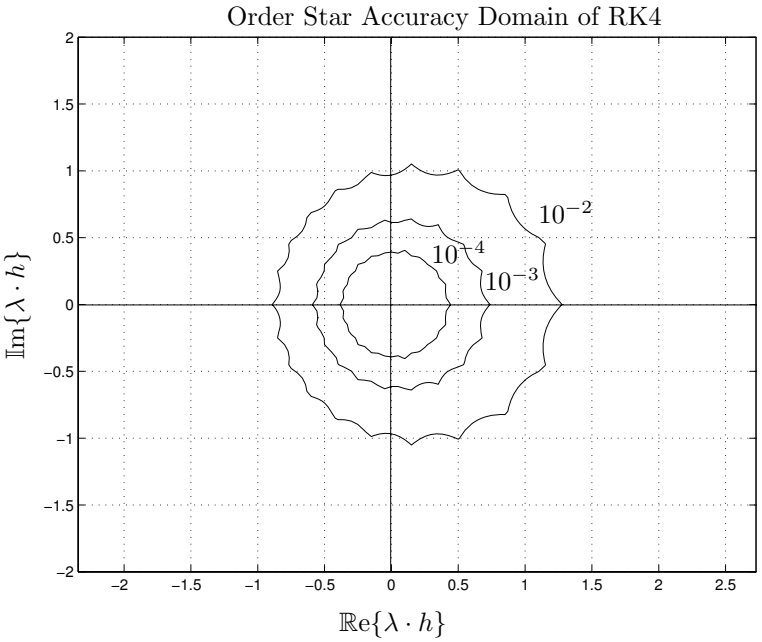

FIGURE 3.22. Order star accuracy domain of RK4.

control algorithm.

The following step–size control algorithm may work. Take any two arbitrary Runge–Kutta algorithms, and repeat the same step twice, once with each of the algorithms. The results of these two algorithms will differ by ε. If ε is larger than the tolerated error tol_{abs}, the step is rejected and repeated with half the step size. If ε is smaller than $0.5 \cdot tol_{\mathrm{abs}}$ during four steps in a row, the next step will be computed with $1.5 \cdot h$.

This algorithm is *very* heuristic and somewhat unsatisfactory on several counts, but the reader certainly has no problems in understanding how the algorithm is supposed to work. The difference between the two solutions, ε, is taken as an estimate for the local integration error and is compared against the tolerated error. If the estimate is larger than the tolerated error, the step size needs to be reduced, but if it is smaller, the step size can be increased.

The first objection that comes to mind is the use of the absolute error as a performance measure. Intuitively, if the state variable (i.e., the output of the integrator) has a value of the order of 10^6, we can tolerate a much larger absolute error than if the state variable has a value of 10^{-6}. It therefore may make sense to replace the absolute error by a relative error. The modified algorithm looks as follows:

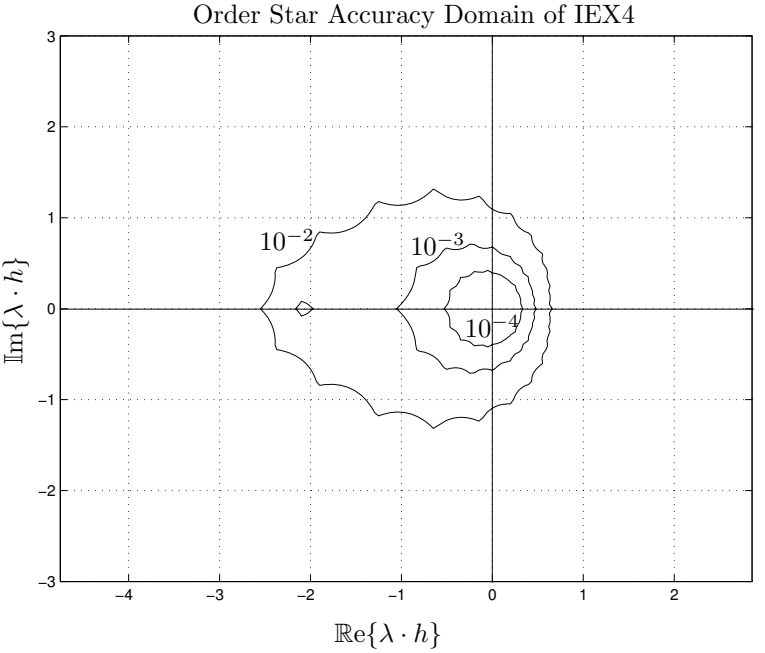

FIGURE 3.23. Order star accuracy domain of IEX4.

$$\varepsilon_{\mathrm{rel}} = \frac{|x_1 - x_2|}{|x_1|} \tag{3.83a}$$

$$\textbf{if } \varepsilon_{\mathrm{rel}} > tol_{\mathrm{rel}} \Rightarrow h_{\mathrm{new}} = 0.5 \cdot h \tag{3.83b}$$

$$\textbf{if } \varepsilon_{\mathrm{rel}} < 0.5 \cdot tol_{\mathrm{rel}} \text{ during four steps} \Rightarrow h_{\mathrm{new}} = 1.5 \cdot h \tag{3.83c}$$

where x_1 is the value of the state variable obtained by one of the two algorithms, whereas x_2 is the value obtained by the other algorithm.

Also this algorithm has its flaws. What if, by accident, $x_1 = 0.0$ at some point in time? This problem can be countered by modifying Eq.(3.83a) in the following fashion:

$$\varepsilon_{\mathrm{rel}} = \frac{|x_1 - x_2|}{\max(|x_1|, |x_2|, \delta)} \tag{3.84}$$

where δ is a fudge factor, e.g., $\delta = 10^{-10}$.

If an entire state vector needs to be integrated, the user may specify different relative error tolerances for each of the state variables separately. The above algorithm could then be applied to each of the state variables separately, resulting in different suggestions for the next step size, h_{new}, to be taken. The smallest of those values would then be applied.

The only remaining problem is the price tag associated with this procedure. Each step must be computed twice, i.e., the step–size control algorithm at least doubles the cost of the numerical integration. Is this truly necessary?

Edwin Fehlberg [3.9] didn't think so. He proposed to make use of the freedom in assigning the α and β parameters of the regular RK algorithms in designing a step–size controlled algorithm in which both RK methods share the early stages of the scheme, and vary only in the later stages. He created pairs of algorithms one order apart. The most commonly used among his methods is RKF4/5 with the Butcher tableau:

0	0	0	0	0	0	0
1/4	1/4	0	0	0	0	0
3/8	3/32	9/32	0	0	0	0
12/13	1932/2197	−7200/2197	7296/2197	0	0	0
1	439/216	-8	3680/513	−845/4104	0	0
1/2	−8/27	2	−3544/2565	1859/4104	−11/40	0
x_1	25/216	0	1408/2565	2197/4104	−1/5	0
x_2	16/135	0	6656/12825	28561/56430	−9/50	2/55

where:

$$f_1(q) = 1 + q + \frac{1}{2}q^2 + \frac{1}{6}q^3 + \frac{1}{24}q^4 + \frac{1}{104}q^5 \tag{3.85a}$$

$$f_2(q) = 1 + q + \frac{1}{2}q^2 + \frac{1}{6}q^3 + \frac{1}{24}q^4 + \frac{1}{120}q^5 + \frac{1}{2080}q^6 \tag{3.85b}$$

Thus, x_1 is a five–stage RK4, and x_2 is a six–stage RK5. However, the RK4 and RK5 algorithms have the first five stages in common. Therefore, the step–size controlled algorithm is overall still a six–stage RK5, and the only additional cost associated with step–size control is the computation of the corrector of RK4. Step–size control comes almost for free.

How about the heuristic algorithm for modifying the step size? Luckily, we can do better than that. Since we have explicit expressions for $f_1(q)$ and $f_2(q)$, we can provide an explicit formula for the error estimate:

$$\varepsilon(q) = f_1(q) - f_2(q) = \frac{1}{780}q^5 - \frac{1}{2080}q^6 \tag{3.86}$$

In a first approximation, we can write:

$$\varepsilon \sim h^5 \tag{3.87}$$

or:

$$h \sim \sqrt[5]{\varepsilon} \tag{3.88}$$

It makes sense to use the following step–size control algorithm:

$$h_{\text{new}} = \sqrt[5]{\frac{tol_{\text{rel}} \cdot \max(|x_1|, |x_2|, \delta)}{|x_1 - x_2|}} \cdot h_{\text{old}} \tag{3.89}$$

The rational behind this algorithm is very simple. As long as:

$$\frac{|x_1 - x_2|}{\max(|x_1|, |x_2|, \delta)} = tol_{\text{rel}} \tag{3.90}$$

we got the right step size, and the step size won't change. However, as soon as $|x_1 - x_2|$ becomes too large, the step size will be reduced. On the other hand, if $|x_1 - x_2|$ becomes too small, the step size will be enlarged.

Contrary to the previously proposed algorithm, no step will ever be repeated. The algorithm accepts too large errors in a single step and just tries to prevent mishap from repeating itself. Algorithms that operate in this fashion are called *optimistic algorithms*, whereas algorithms that repeat steps exhibiting errors that are too large are called *conservative algorithms*.

The above procedure might work very well indeed if only we could trust that one of the algorithms always *overestimates* the true value, while the other always *underestimates* it, with the additional constraint that both algorithms smoothly approximate the true value as $h \to 0$.

Unfortunately, such a guarantee cannot be given. It is entirely feasible that both algorithms agree, by accident, on the same incorrect result. Thus, the algorithm can be fooled. Luckily, this doesn't happen too often in practice, and consequently, engineers are usually quite happy with this algorithm.

Kjell Gustafsson [3.11] had an even better idea. Kjell had a background in control engineering, and therefore viewed the step–size control problem from a control engineering perspective. This view is shown in Fig.3.24.

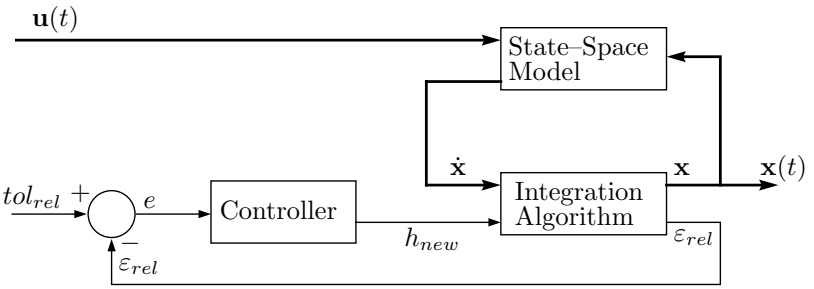

FIGURE 3.24. Step–size control viewed as a control problem.

The integration algorithm interacts with the state space model in a closed loop. We have seen that loop before. However, it also generates another output, namely the estimate of the relative integration error, ε_{rel}. This quantity is fed back and compared with the desired relative error, tol_{rel}. The resulting error signal, e, is then fed into a controller box that computes the next value of the step size, h_{new}.

The controller can be designed using standard control theory. It turns out that the previously proposed step–size adaptation rule of Eq.3.89 cor-

responds to a discrete proportional controller (P–controller). However, we can just as well implement either a discrete PI–controller or a discrete PID–controller. All we need to do is to modify Eq.3.89 accordingly.

Gustafsson found that a PI–controller can be implemented using the following modified step–size control algorithm:

$$h_{\text{new}} = \left(\frac{0.8 \cdot tol_{\text{rel}}}{\varepsilon_{\text{rel}_{\text{now}}}} \right)^{\frac{0.3}{n}} \cdot \left(\frac{\varepsilon_{\text{rel}_{\text{last}}}}{\varepsilon_{\text{rel}_{\text{now}}}} \right)^{\frac{0.4}{n}} \cdot h_{\text{old}} \tag{3.91}$$

where:

$$\varepsilon_{\text{rel}_{\text{now}}} = \frac{\|\mathbf{x_1} - \mathbf{x_2}\|_\infty}{\max(\|\mathbf{x_1}\|_2, \|\mathbf{x_2}\|_2, \delta)} \tag{3.92a}$$

$$\varepsilon_{\text{rel}_{\text{last}}} = \text{same quantity one time step back} \tag{3.92b}$$

and n is the approximation order of the integration algorithm, in the case of RKF4/5, $n = 5$.

In Fig.3.24, it was assumed that the same relative error applies to all state variables, i.e., we can operate on norms of the two state vectors rather than on individual state variables. Therefore, ε_{rel} is a scalar rather than a vector. However, the vector case could have been treated in exactly the same fashion.

In Chapter 2, we have seen that we always have a choice between a high–order algorithm with a larger step size and a low–order algorithm with a smaller step size. Although it has been shown in Chapter 3 that low–order algorithms aren't really suitable in most situations, the question remains whether order control might be a viable alternative to step–size control.

Several authors have extended the idea of embedded RK algorithms (RK algorithms of different orders sharing their early stages) for the purpose of order control [3.1]. However, beside from some interesting research papers, these efforts didn't go anywhere. The reason is simple. Low–order RK algorithms are dubious anyway. Step–size controlled low–order RKs are even worse, since the relative overhead paid for step–size control is larger for low–order algorithms. The additional overhead paid for the embedding makes these algorithms non–economic for practically all applications. Order control is a fashionable subject in multi–step integration, algorithms that will be discussed in the next chapter of this book. However, the order–control issue is, in our opinion, overrated even in the context of those algorithms.

3.10 Summary

This chapter has extended the general observations on numerical ODE integration made in Chapter 2 to a large class of higher–order integration

algorithms, namely the so–called Runge–Kutta methods. Both explicit [3.3, 3.12] and implicit versions [3.13] of these algorithms were discussed, and their stability as well as accuracy properties were analyzed.

Most engineering applications call for fourth–order algorithms. The step–size controlled RKF4/5 algorithm [3.9] is easily the most popular among the explicit RK techniques. More recently, DOPRI4/5 [3.7, 3.11] became also fairly popular due to its somewhat smaller error coefficient ($97/120,000$ in comparison with $1/780$ of RKF4/5), which is paid for by one additional stage, i.e., one additional function evaluation per step.

The most popular implicit RK algorithms for stiff system simulation are the fully–implicit Radau algorithms [3.13]. These algorithms are highly efficient, and good (robust) implementations are available. They shall be discussed in greater detail in Chapter 8 of this book. Backinterpolation techniques, and in particular $BI4/5_{0.45}$, are a new and viable alternative that, in our opinion, will receive more attention in the future.

Implicit extrapolation techniques have been advocated for use on parallel processor architectures [3.6]. Parallelization of these algorithms is trivial, since the predictor stages can be computed in parallel. In this way, the number of consecutive stages of a fourth-order extrapolation method can be reduced from 10 to four. These algorithms have been successfully applied to the simulation of complex chemical processing plants, such as distillation columns with 50 trays.

F–stable algorithms for the simulation of marginally stable systems are a more specialized breed of animals in the zoo of numerical ODE integration algorithms. They will never become as popular as either the non–stiff explicit algorithms or the stiffly–stable implicit algorithms, since the number of suitable applications for these algorithms is more limited. However, these algorithms should be looked at more closely in the context of method–of–lines solutions to hyperbolic PDE problems.

3.11 References

[3.1] Dale G. Bettis. Efficient Embedded Runge–Kutta Methods. In Roland Bulirsch, Rolf Dieter Grigorieff, and Johann Schröder, editors, *Numerical Treatment of Differential Equations*, volume 631 of *Lecture Notes in Mathematics*, pages 9–18. Springer–Verlag, New York, 1976.

[3.2] Kevin Burrage. Efficiently Implementable Algebraically Stable Runge–Kutta Methods. *SIAM J. Numerical Analysis*, 19:245–258, 1982.

[3.3] John C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge–Kutta and General Linear Methods.* John Wiley, Chichester, United Kingdom, 1987. 512p.

[3.4] Alan R. Curtis. An Eighth Order Runge–Kutta Process With Eleven Function Evaluations Per Step. *Numerische Mathematik*, 16:268–277, 1970.

[3.5] 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.

[3.6] Peter Deuflhard. Extrapolation Integrators for Quasilinear Implicit ODEs. In Peter Deuflhard and Björn Enquist, editors, *Large Scale Scientific Computing*, volume 7 of *Progress in Scientific Computing*, pages 37–50, Birkhäuser, Boston, Mass., 1987.

[3.7] John R. Dormand and Peter J. Prince. A Family of Embedded Runge–Kutta Formulae. *J. of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[3.8] Leonhard Euler. De integratione æquationum differentialium per approximationem. In *Opera Omnia*, volume 11 of *first series*, pages 424–434. Institutiones Calculi Integralis, Teubner Verlag, Leipzig, Germany, 1913.

[3.9] Edwin Fehlberg. Classical 5^{th}–, 6^{th}–, 7^{th}–, and 8^{th}–Order Runge–Kutta Formulas. Technical Report NASA TR R–287, NASA Johnson Space Center, Houston, Texas, 1968.

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

[3.11] Kjell Gustafsson. *Control of Error and Convergence in ODE Solvers*. PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1992.

[3.12] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*, volume 8 of *Series in Computational Mathematics*. Springer–Verlag, Berlin, Germany, 2^{nd} edition, 2000. 528p.

[3.13] 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.

[3.14] Karl Heun. Neue Methoden zur approximativen Integration der Differentialgleichungen einer unabhängigen Veränderlichen. *Zeitschrift für Mathematische Physik*, 45:23–38, 1900.

[3.15] Anton Huťa. Une amélioration de la méthode de Runge–Kutta–Nyström pour la résolution numérique des équations différentielles du premier ordre. *Acta Fac. Nat. Univ. Comenian. Math.*, 1:201–224, 1956.

[3.16] Arieh Iserles and Syvert P. Nørsett. *Order Stars*, volume 2 of *Applied Mathematics and Mathematical Computation*. Chapman & Hall, London, United Kingdom, 1991. 248p.

[3.17] Wilhelm Kutta. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschrift für Mathematische Physik*, 46:435–453, 1901.

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

[3.19] Cleve Moler and Charles van Loan. Nineteen Dubious Ways to Compute the Exponential of a Matrix. *SIAM Review*, 20(4):801–836, 1978.

[3.20] Evert Johannes Nyström. Über die numerische Integration von Differentialgleichungen. *Acta Socialum Scientarum Fennicæ*, 50(13):1–55, 1925.

[3.21] Carl Runge. Über die numerische Auflösung von Differentialgleichungen. *Mathematische Annalen*, 46:167–178, 1895.

[3.22] Baylis Shanks. Solutions of Differential Equations by Evaluations of Functions. *Mathematics of Computation*, 20:21–38, 1966.

[3.23] Gerhard Wanner, Ernst Hairer, and Syvert Nørsett. Order Stars and Stability Theorems. *BIT*, 18:475–489, 1978.

[3.24] Wei Xie. Backinterpolation Methods for the Numerical Solution of Ordinary Differential Equations and Applications. Master's thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, Ariz., 1995.

3.12 Homework Problems

[H3.1] Family of Explicit RK2 Algorithms

Verify that Eq.(3.10) is indeed the correct Taylor–Series expansion describing the parameterized family of all two–stage explicit RK2 algorithms.

[H3.2] Family of Explicit RK3 Algorithms

Derive the constraint equations in the α_i and β_{ij} parameters that characterize the family of all three–stage explicit RK3 algorithms.

To this end, the Taylor–series of $f(x+\Delta x, y+\Delta y)$ must now be expanded up to the quadratic terms:

$$f(x+\Delta x, y+\Delta y) \approx f(x,y) + \frac{\partial f(x,y)}{\partial x}\cdot\Delta x + \frac{\partial f(x,y)}{\partial y}\cdot\Delta y + \frac{\partial^2 f(x,y)}{\partial x^2}\cdot\frac{\Delta x^2}{2} + \frac{\partial^2 f(x,y)}{\partial x\cdot\partial y}\cdot\Delta x\cdot\Delta y + \frac{\partial^2 f(x,y)}{\partial y^2}\cdot\frac{\Delta y^2}{2} \quad \text{(H3.2a)}$$

[H3.3] Runge–Kutta–Simpson Algorithm

Given the four–stage Runge–Kutta algorithm characterized by the Butcher tableau:

0	0	0	0	0
1/3	1/3	0	0	0
2/3	-1/3	1	0	0
1	1	-1	1	0
x	1/8	3/8	3/8	1/8

Write down the stages of this algorithm. Determine the linear order of approximation accuracy of this method. How would you judge this method in comparison with other Runge–Kutta algorithms discussed in this chapter?

[H3.4] RK Order Increase by Blending

Given two separate n^{th}–order accurate RK algorithms in at least $(n+1)$ stages:

$$f_1(q) = 1 + q + \frac{q^2}{2!} + \cdots + \frac{q^n}{n!} + c_1\cdot q^{n+1} \quad \text{(H3.4a)}$$

$$f_2(q) = 1 + q + \frac{q^2}{2!} + \cdots + \frac{q^n}{n!} + c_2\cdot q^{n+1} \quad \text{(H3.4b)}$$

where $c_2 \neq c_1$.

Show that it is always possible to use blending:

$$\mathbf{x^{blended}} = \vartheta\cdot\mathbf{x^1} + (1-\vartheta)\cdot\mathbf{x^2} \quad \text{(H3.4c)}$$

where $\mathbf{x^1}$ is the solution found using method $f_1(q)$ and $\mathbf{x^2}$ is the solution found using method $f_2(q)$, such that $\mathbf{x^{blended}}$ is of order $(n+1)$.

Find a formula for ϑ that will make the blended algorithm accurate to the order $(n+1)$.

[H3.5] Stability Domains of RKF4/5

Find the stability domains of the two algorithms used in RKF4/5. Interpret the results. Is it better to use the fourth–order approximation to continue with the next step, or should the fifth–order approximation be used?

[H3.6] Runge–Kutta Integration

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

$$\begin{aligned}\dot{\mathbf{x}} &= \begin{pmatrix} 1250 & -25113 & -60050 & -42647 & -23999 \\ 500 & -10068 & -24057 & -17092 & -9613 \\ 250 & -5060 & -12079 & -8586 & -4826 \\ -750 & 15101 & 36086 & 25637 & 14420 \\ 250 & -4963 & -11896 & -8438 & -4756 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 5 \\ 2 \\ 1 \\ -3 \\ 1 \end{pmatrix} \cdot u \\ \mathbf{y} &= \begin{pmatrix} -1 & 26 & 59 & 43 & 23 \end{pmatrix} \cdot \mathbf{x} \end{aligned} \tag{H3.6a}$$

with initial conditions:

$$\mathbf{x_0} = \begin{pmatrix} 1 & -2 & 3 & -4 & 5 \end{pmatrix}^T \tag{H3.6b}$$

This is the same system that was used in Hw.[H2.1] Simulate the system across 10 seconds of simulated time with step input using the RK4 algorithm with the α–vector and β–matrix of Eq.(3.15). The following fixed step sizes should be tried:

1. $h = 0.32$,
2. $h = 0.032$,
3. $h = 0.0032$.

Plot the three trajectories on top of each other. What can you conclude about the accuracy of the results?

[H3.7] Implicit Extrapolation

Derive the α_1 and α_2 coefficients of the IEX2 method using the two approaches demonstrated in the chapter. Show that you indeed obtain the same coefficients using either of the two methods.

[H3.8] Implicit Extrapolation

Repeat Hw.[H3.6], this time using the IEX4 algorithm. Since the system to be simulated is linear, the implicit algorithm can be implemented by matrix inversion rather than by Newton iteration.

Which algorithm is more accurate for the same step size: RK4 or IEX4?

[H3.9] Explicit Integration Methods and Their Stability Domains

Prove that all explicit RK algorithms have stability domains that look qualitatively like that of FE, i.e., bend into the left–half $\lambda \cdot h$ plane. To this end, show that all explicit RK algorithms are characterized by a polynomial rather than rational $f(q)$, and analyze $f(q)$ for large values of $q \to \infty$.

[H3.10] Implicit Integration Methods and Their Stability Domains

Prove that all implicit RK algorithms with strictly proper rational $f(q)$ functions have stability domains that look qualitatively like that of BE, i.e., bend into the right–half $\lambda \cdot h$ plane.

Show furthermore that no integration algorithm with a non–strictly proper $f(q)$ function can exhibit infinite damping far away from the origin of the complex $\lambda \cdot h$ plane.

[H3.11] Stability Domains of BI4/5$_\vartheta$

Find the stability domain of BI4/5$_\vartheta$ using the approximation of Eq.(3.85a) for the explicit semi–step, and the approximation of Eq.(3.85b) for the implicit semi–step using the following ϑ values:

$$\vartheta = \{0.4, 0.45, 0.475, 0.48, 0.5\} \tag{H3.11a}$$

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

[H3.12] BI4/5$_{0.45}$ for Linear Systems

Repeat Hw.[H3.6] this time using BI4/5$_{0.45}$. The explicit semi–step uses the fourth–order approximation of RKF4/5. There is no need to compute the fifth–order corrector. The implicit semi–step uses the fifth–order corrector. There is no need to compute the fourth–order corrector. Since the system to be simulated is linear, the implicit semi–step can be implemented using matrix inversion. No step–size control is attempted.

Compare the accuracy of this algorithm with that of RK4 and IEX4.

[H3.13] Stability Domain and Newton Iteration

Show that Newton iteration indeed does not modify the stability domain of BI2.

[H3.14] BI4/5$_{0.45}$ for Nonlinear Systems

Repeat Hw.[H3.12]. This time, we want to replace the matrix inversion by Newton iteration. Of course, since the problem is linear and time–invariant, Newton iteration and modified Newton iteration are identical. Iterate until

$\delta_{\text{rel}} \leq 10^{-5}$, where:

$$\delta_{\text{rel}} = \frac{\|\mathbf{x}^{\text{right}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{2}}} - \mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{2}}}\|_\infty}{\max(\|\mathbf{x}^{\text{left}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{2}}}\|_2, \|\mathbf{x}^{\text{right}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{2}}}\|_2, \delta)} \tag{H3.14a}$$

Compare the results obtained with those found in Hw.[H3.12].

[H3.15] Backinterpolation With Step–Size Control

We want to repeat Hw.[H3.14] once more, this time using a step–size controlled algorithm. The step–size control to be used is the following. On the *explicit* semi–step, compute now both correctors, and find ε_{rel} according to the formula:

$$\varepsilon_{\text{rel}} = \frac{\|\mathbf{x_1} - \mathbf{x_2}\|_\infty}{\max(\|\mathbf{x_1}\|_2, \|\mathbf{x_2}\|_2, \delta)} \tag{H3.15a}$$

If $\varepsilon_{\text{rel}} \leq 10^{-4}$, use the Gustafsson algorithm to compute the step size to be used in the next step:

$$h_{\text{new}} = \left(\frac{0.8 \cdot 10^{-4}}{\varepsilon_{\text{rel}_{\text{now}}}}\right)^{0.06} \cdot \left(\frac{\varepsilon_{\text{rel}_{\text{last}}}}{\varepsilon_{\text{rel}_{\text{now}}}}\right)^{0.08} \cdot h_{\text{old}} \tag{H3.15b}$$

except during the first step, when we use:

$$h_{\text{new}} = \left(\frac{0.8 \cdot 10^{-4}}{\varepsilon_{\text{rel}_{\text{now}}}}\right)^{0.2} \cdot h_{\text{old}} \tag{H3.15c}$$

However, if $\varepsilon_{\text{rel}} > 10^{-4}$, we reject the step at once, i.e., we never even proceed to the implicit semi–step, and compute a new step size in accordance with Eq.(H3.15c).

If a step was repeated, the step size for the immediately following next step is also computed according to Eq.(H3.15c) rather than using Eq.(H3.15b).

Apply this step–size control algorithm to the same problem as before, and determine the largest global relative error by comparing the solution with the analytical solution of this linear time–invariant system.

Compute also the largest global relative error of the three solutions of Hw.[H3.14].

Compute the number of floating–point operations of the step–size controlled algorithm as well as the numbers of floating–point operations of the fixed–step algorithms of Hw.[H3.14].

Use the product of accuracy and cost:

$$Q = \#\text{ of floating point operations} \times \text{largest global relative error} \tag{H3.15d}$$

as a performance measure, and rank the four solutions accordingly. Is the step–size controlled algorithm economical when using this performance measure?

[H3.16] CSMP–III

The most widely used simulation software in the 70s was a program from IBM called *Continuous System Modeling Program III (CSMP–III)*. That software offered, as its default integration algorithm, the classical 4^{th}–order Runge–Kutta algorithm presented in Eq.(3.15). For step–size control, the algorithm used an implementation of *Simpson's rule*, also known under the name of 4^{th}*–order Milne algorithm*, an implicit linear multi–step method that we shall meet again in Chapter 4 of this book. The algorithm is usually written as:

$$\mathbf{x_{k+1}} = \mathbf{x_{k-1}} + \frac{h}{3} \cdot (\mathbf{f_{k+1}} + 4 \cdot \mathbf{f_k} + \mathbf{f_{k-1}}) \tag{H3.16a}$$

However, by shrinking the step size by a factor of two, it can also be written as:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{6} \cdot (\mathbf{f_{k+1}} + 4 \cdot \mathbf{f_{k+\frac{1}{2}}} + \mathbf{f_k}) \tag{H3.16b}$$

CSMP–III implemented this formula as a predictor–corrector technique, using two semi–steps of FE to estimate the unknown derivative values, $\mathbf{f_{k+\frac{1}{2}}}$ and $\mathbf{f_{k+1}}$.

Write down the Butcher tableau of the combined RK4/Simpson algorithm, sharing as many stages between the two algorithms as possible.

Find the linear order of approximation accuracy of this implementation of Simpson's rule. How would you characterize this method?

What can you conclude about the usefulness of this technique for step–size control of the RK4 algorithm?

[H3.17] Embedded RK Algorithms

Given the two embedded RK algorithms characterized by the following Butcher tableau:

0	0	0	0
1	1	0	0
1/2	1/4	1/4	0
x_1	1/2	1/2	0
x_2	1/6	1/6	2/3

Write down the stages of these two algorithms. Determine the linear order of approximation accuracy for each of them.

[H3.18] Accuracy Domains

Determine the accuracy domains (left–half plane only) of IEX4 and $BI4/5_{0.45}$ for $tol = 10^{-4}$, and compare them to the accuracy domain of RK4. What do you conclude?

[H3.19] Order Star

Find the damping order star for $BI4/5_{0.45}$, and plot it together with the pole and zero locations. Compare with the order star of Fig.3.17. Find the frequency order star for $BI4/5_{0.45}$, and plot it together with the pole and zero locations. Compare with the order star of Fig.3.19. Finally, compute and plot the order star accuracy domain of this method.

[H3.20] Lie-series Integration, Algebraic Differentiation

The Van–der–Pol oscillator can be described by the following 2^{nd}–order differential equation:

$$\ddot{x} - \mu \cdot (1 - x^2) \cdot \dot{x} + x = 0 \tag{H3.20a}$$

Write down a state–space model of the Van–der–Pol oscillator with $x_1 = x$, and $x_2 = \dot{x}$.

Create a MATLAB function:

$$[\mathbf{f}, \dot{\mathbf{f}}, \ddot{\mathbf{f}}] = vdp(\mathbf{x}) \tag{H3.20b}$$

that computes the first, second, and third state derivative vectors. Use *algebraic differentiation* to symbolically find expressions for the higher derivatives.

We wish to simulate the Van–der–Pol oscillator with $\mu = 2.0$ during 20 seconds using a step size of $h = 0.1$ by means of *Lie–series integration*, i.e., by making direct use of the Taylor–series expansion of the exponential function:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \mathbf{f_k} + \frac{h^2}{2} \cdot \dot{\mathbf{f}}_\mathbf{k} + \frac{h^3}{6} \cdot \ddot{\mathbf{f}}_\mathbf{k} \tag{H3.20c}$$

Use six different sets of initial conditions:

1. $x_{10} = 0.1$, $x_{20} = 0.1$,
2. $x_{10} = 0.1$, $x_{20} = -0.1$,
3. $x_{10} = -0.1$, $x_{20} = 0.1$,
4. $x_{10} = -0.1$, $x_{20} = -0.1$,
5. $x_{10} = -2.0$, $x_{20} = 2.0$,
6. $x_{10} = 2.0$, $x_{20} = -2.0$,

and plot $x_2(t)$ as a function of $x_1(t)$ in the phase plane, superposing the six solutions onto the same graph.

3.13 Projects

[P3.1] Accuracy Domain *vs.* Order–star Accuracy Domain

Draw the accuracy domains and order–star accuracy domains for different integration algorithms, and determine the relationship between these two approaches to characterizing the accuracy of an integration algorithm.

3.14 Research

[R3.1] ϑ–Method

Study the relationship between the locations of the eigenvalues of the Jacobian matrix of the system to be simulated, and the choice of the *theta*–parameter in BI4/5_ϑ.

Design a general–purpose control algorithm to modify *theta* as a function of the (usually time–dependent) eigenvalue locations of the Jacobian matrix of the system to be simulated.

[R3.2] L–Stability

Analyze the effects of the shape of the damping plot on the accuracy of a stiff system integrator. Quantify the importance of L–stability. Determine a method to systematically find L–stable algorithms that minimize the distance between σ_d and $\hat{\sigma}_d$ in a least–square sense within a reasonable range of the negative real axis of the damping plot.