4

Multi–step Integration Methods

Preview

In this chapter, we shall look at several families of integration algorithms that all have in common the fact that only a single function evaluation needs to be performed in every integration step, irrespective of the order of the algorithm. Both explicit and implicit varieties of this kind of algorithms exist and shall be discussed. As in the last chapter, we shall spend some time discussing the stability and accuracy properties of these families of integration algorithms.

Whereas step–size and order control were easily accomplished in the case of the single–step techniques, these issues are much more difficult to tackle in the case of the multi–step algorithms. Consequently, their discussion must occupy a significant portion of this chapter.

The chapter starts out with mathematical preliminaries that shall simplify considerably the subsequent derivation of the multi–step methods.

4.1 Introduction

In the last chapter, we have looked at integration algorithms that, in one way or other, all try to approximate Taylor–Series expansions of the unknown solution around the current time instant. The trick was to never compute the higher derivatives explicitly, but to replace these higher derivatives by additional function evaluations to be taken at various time instants inside the integration step.

One disadvantage of this approach is that, with every new step, we start out again with an empty slate, i.e., in each new step, we have to build up the higher–order algorithms from scratch. Isn't it a pity that, at the end of every step, all the higher–order information is thrown away again? Isn't that wasteful? Wouldn't it be possible to preserve some of this information so that, in the subsequent step, the number of function evaluations can be kept smaller? The answer to this question is a definite yes. In fact, it is possible to find entire classes of integration algorithms of arbitrary order of approximation accuracy that require only a single function evaluation in every new step, because they preserve the complete information from the previous steps. That is the topic of our discussion in this chapter.

There are many ways how these families of algorithms can be derived. However, in order to make their introduction and derivation easy, we need some additional mathematical apparatus that we shall introduce first. To this end, we shall initially not talk about numerical integration at all. Instead, we shall focus our interest on higher–order interpolation (extrapolation) polynomials.

4.2 Newton–Gregory Polynomials

Given a function of time, $f(t)$. We shall denote the values of this function at various points in time, t_0, t_1, t_2, etc. as f_0, f_1, f_2, etc. We shall introduce Δ as a *forward difference operator*, thus, $\Delta f_0 = f_1 - f_0$, $\Delta f_1 = f_2 - f_1$, etc.

Higher–order forward difference operators can be defined accordingly:

$$\Delta^2 f_0 = \Delta(\Delta f_0) = \Delta(f_1 - f_0) = \Delta f_1 - \Delta f_0 = f_2 - 2f_1 + f_0 \tag{4.1a}$$

$$\Delta^3 f_0 = \Delta(\Delta^2 f_0) = f_3 - 3f_2 + 3f_1 - f_0 \tag{4.1b}$$

etc.

In general:

$$\Delta^n f_i = f_{i+n} - n \cdot f_{i+n-1} + \frac{n(n-1)}{2!} \cdot f_{i+n-2} - \frac{n(n-1)(n-2)}{3!} \cdot f_{i+n-3} + \dots \tag{4.2}$$

or:

$$\Delta^n f_i = \binom{n}{0} f_{i+n} - \binom{n}{1} f_{i+n-1} + \binom{n}{2} f_{i+n-2} - \binom{n}{3} f_{i+n-3} + \dots \pm \binom{n}{n} f_i \tag{4.3}$$

Let us now assume that the time points at which the f_i values are given are a fixed distance h apart from each other. We wish to find an interpolation (extrapolation) polynomial of n^{th} order that passes through the $(n+1)$ given function values f_0, f_1, f_2, …, f_n at the given time instants t_0, $t_1 = t_0 + h$, $t_2 = t_0 + 2h$, …, $t_n = t_0 + n \cdot h$.

Let us introduce an auxiliary variable s defined as follows:

$$s = \frac{t - t_0}{h} \tag{4.4}$$

Thus, for $t = t_0 \leftrightarrow s = 0.0$, for $t = t_1 \leftrightarrow s = 1.0$, etc. The real–valued variable s assumes integer values at the sampling points. At those points, the value of s corresponds to the index of the sampling point.

The desired interpolation polynomial can be written as a function of s:

$$f(s) \approx \binom{s}{0} f_0 + \binom{s}{1} \Delta f_0 + \binom{s}{2} \Delta^2 f_0 + \cdots + \binom{s}{n} \Delta^n f_0 \tag{4.5}$$

This is called a *Newton–Gregory forward polynomial*. It is easy to prove that this polynomial indeed possesses the desired qualities. First of all, it is clearly an n^{th}–order polynomial in s. Since s is linear in t, it is also an n^{th}–order polynomial in t. By plugging in integer values of s in the range 0 to n, we can verify easily that the polynomial indeed passes through f_0 to f_n. Since there exists exactly one n^{th}–order polynomial that passes through any given set of $(n+1)$ points, the assertion has been proven.

Sometimes, it is more useful to have an n^{th}–order polynomial that passes through $(n+1)$ time points in the past. The *Newton–Gregory backward polynomial* can be written as:

$$\begin{aligned} f(s) \quad \approx \quad & f_0 + \binom{s}{1} \Delta f_{-1} + \binom{s+1}{2} \Delta^2 f_{-2} + \binom{s+2}{3} \Delta^3 f_{-3} + \ldots \\ & + \binom{s+n-1}{n} \Delta^n f_{-n} \end{aligned} \tag{4.6}$$

It is equally easy to show that this n^{th}–order polynomial passes through the $(n+1)$ points f_0, f_{-1}, f_{-2}, …, f_{-n} at the time instants t_0, t_{-1}, t_{-2}, …, t_{-n} by plugging in values of $s = 0.0$, $s = -1.0$, $s = -2.0$, …, $s = -n$.

It is common practice to also introduce a *backward difference operator*, ∇, defined as:

$$\nabla f_i = f_i - f_{i-1} \tag{4.7}$$

with the higher–order operators:

$$\begin{aligned} \nabla^2 f_i &= \nabla(\nabla f_i) = \nabla(f_i - f_{i-1}) = \nabla f_i - \nabla f_{i-1} \\ &= f_i - 2\ f_{i-1} + f_{i-2} \end{aligned} \tag{4.8a}$$

$$\nabla^3 f_i = \nabla(\nabla^2 f_i) = f_i - 3f_{i-1} + 3f_{i-2} - f_{i-3} \tag{4.8b}$$

etc.

or, in general:

$$\nabla^n f_i = \binom{n}{0} f_i - \binom{n}{1} f_{i-1} + \binom{n}{2} f_{i-2} - \binom{n}{3} f_{i-3} + \cdots \pm \binom{n}{n} f_{i-n} \tag{4.9}$$

The Newton–Gregory backward polynomial can be expressed in terms of the ∇–operator as:

$$f(s) \approx f_0 + \binom{s}{1}\nabla f_0 + \binom{s+1}{2}\nabla^2 f_0 + \binom{s+2}{3}\nabla^3 f_0 + \cdots + \binom{s+n-1}{n}\nabla^n f_0 \tag{4.10}$$

It is also quite common to introduce yet another operator, namely the *shift operator*, $\mathcal{E}$. It is defined as:

$$\mathcal{E} f_i = f_{i+1} \tag{4.11}$$

with the higher–order operators:

$$\mathcal{E}^2 f_i = \mathcal{E}(\mathcal{E} f_i) = \mathcal{E}(f_{i+1}) = f_{i+2} \tag{4.12a}$$
$$\mathcal{E}^3 f_i = \mathcal{E}(\mathcal{E}^2 f_i) = \mathcal{E}(f_{i+2}) = f_{i+3} \tag{4.12b}$$
etc.

It is obviously true that:

$$\Delta f_i = \mathcal{E} f_i - f_i = (\mathcal{E} - 1) f_i \tag{4.13a}$$
$$\nabla f_i = f_i - \mathcal{E}^{-1} f_i = (1 - \mathcal{E}^{-1}) f_i \tag{4.13b}$$
$$\mathcal{E}(\nabla f_i) = \mathcal{E}(f_i - f_{i-1}) = f_{i+1} - f_i = \Delta f_i \tag{4.13c}$$

By abstraction:

$$\Delta = \mathcal{E} - 1 \tag{4.14a}$$
$$\nabla = 1 - \mathcal{E}^{-1} \tag{4.14b}$$
$$\Delta = \mathcal{E}\nabla \tag{4.14c}$$

Since these are all linear operators, we can formally calculate with them as with other algebraic quantities. In particular:

$$\Delta^n = (\mathcal{E} - 1)^n = \mathcal{E}^n - n\mathcal{E}^{n-1} + \binom{n}{2}\mathcal{E}^{n-2} - + \cdots \pm \binom{n}{n-1}\mathcal{E} \mp 1 \tag{4.15}$$

Using this calculus, the derivation of the two Newton–Gregory polynomials becomes trivial.

$$f(s) \approx \mathcal{E}^s f_0 = (1 + \Delta)^s f_0 = \left[1 + \binom{s}{1}\Delta + \binom{s}{2}\Delta^2 + \binom{s}{3}\Delta^3 + \ldots\right] f_0 \tag{4.16}$$

is the Newton–Gregory forward polynomial, and:

$$f(s) \approx (1-\nabla)^{-s} f_0 = \left[1 + \binom{s}{1}\nabla + \binom{s+1}{2}\nabla^2 + \binom{s+2}{3}\nabla^3 + \dots\right] f_0 \tag{4.17}$$

is the Newton–Gregory backward polynomial.

Since differentiation is also a linear operation, we can find the first time derivative of $f(t)$ in the following manner:

$$\begin{aligned} \dot{f}(t) &= \frac{d}{dt} f(t) = \frac{\partial}{\partial s} f(s) \cdot \frac{ds}{dt} \\ &\approx \frac{1}{h} \cdot \frac{\partial}{\partial s} \left(f_0 + s\Delta f_0 + \frac{s(s-1)}{2!}\Delta^2 f_0 + \dots \right) \end{aligned} \tag{4.18}$$

and in particular:

$$\dot{f}(t_0) \approx \frac{1}{h} \cdot \left(\Delta f_0 - \frac{1}{2}\Delta^2 f_0 + \frac{1}{3}\Delta^3 f_0 - \dots \pm \frac{1}{n}\Delta^n f_0 \right) \tag{4.19}$$

We introduce a new operator, the *differentiation operator*, $\mathcal{D}$, as:

$$\mathcal{D} = \frac{1}{h} \cdot \left(\Delta - \frac{1}{2}\Delta^2 + \frac{1}{3}\Delta^3 - \dots \pm \frac{1}{n}\Delta^n \right) \tag{4.20}$$

Consequently, we can compute the second derivative as:

$$\begin{aligned} \mathcal{D}^2 &= \frac{1}{h^2} \cdot \left(\Delta - \frac{1}{2}\Delta^2 + \frac{1}{3}\Delta^3 - \dots \pm \frac{1}{n}\Delta^n \right)^2 \\ &= \frac{1}{h^2} \cdot \left(\Delta^2 - \Delta^3 + \frac{11}{12}\Delta^4 - \frac{5}{6}\Delta^5 + \dots \right) \end{aligned} \tag{4.21}$$

etc.

A more thorough discussion of these and other interpolation polynomials can be found in [4.7]. However, for our purpose, the material presented here will suffice.

4.3 Numerical Integration Through Polynomial Extrapolation

The idea behind multi–step integration is straightforward. We can employ a Newton–Gregory backward polynomial setting $t_k = t_0$ and evaluating for $s = 1.0$. This should give us an estimate of $x(t_{k+1}) = f_1$. The back values f_0, f_{-1}, f_{-2}, etc. are the previously computed solutions $x(t_k)$, $x(t_{k-1})$, $x(t_{k-2})$, etc. Until here, we have written the Newton–Gregory polynomials

for the scalar case, but the concept extends without complications also to the vector case.

The trick is to somehow modify the notation of the Newton–Gregory backward polynomial such that values of $\dot{f}$ are used beside from the values of f in order to incorporate the knowledge available through the state–space model, but such that higher derivatives, as $\ddot{f}$, are avoided, since they are difficult to compute accurately.

4.4 Explicit Adams–Bashforth Formulae

Let us write a Newton–Gregory backward polynomial for the state derivative vector $\dot{\mathbf{x}}(t)$ around the time point t_k:

$$\dot{\mathbf{x}}(t) = \mathbf{f}_k + \binom{s}{1}\nabla\mathbf{f}_k + \binom{s+1}{2}\nabla^2\mathbf{f}_k + \binom{s+2}{3}\nabla^3\mathbf{f}_k + \dots \tag{4.22}$$

where:

$$\mathbf{f}_k = \dot{\mathbf{x}}(t_k) = \mathbf{f}(\mathbf{x}(t_k), t_k) \tag{4.23}$$

is the state derivative vector at time t_k. We wish to find an expression for $\mathbf{x}(t_{k+1})$. Therefore, we need to integrate Eq.(4.22) in the interval $[t_k, t_{k+1}]$:

$$\begin{aligned}
&\int_{t_k}^{t_{k+1}} \dot{\mathbf{x}}(t)dt = \mathbf{x}(t_{k+1}) - \mathbf{x}(t_k) \\
&= \int_{t_k}^{t_{k+1}} \left[\mathbf{f}_k + \binom{s}{1}\nabla\mathbf{f}_k + \binom{s+1}{2}\nabla^2\mathbf{f}_k + \binom{s+2}{3}\nabla^3\mathbf{f}_k + \dots\right] dt \\
&= \int_{0.0}^{1.0} \left[\mathbf{f}_k + \binom{s}{1}\nabla\mathbf{f}_k + \binom{s+1}{2}\nabla^2\mathbf{f}_k + \binom{s+2}{3}\nabla^3\mathbf{f}_k + \dots\right] \cdot \frac{dt}{ds} \cdot ds
\end{aligned} \tag{4.24}$$

Thus:

$$\begin{aligned}
\mathbf{x}(t_{k+1}) =& \mathbf{x}(t_k) + h\int_0^1 \left[\mathbf{f}_k + s\nabla\mathbf{f}_k + \left(\frac{s^2}{2} + \frac{s}{2}\right)\nabla^2\mathbf{f}_k \right. \\
&\left. + \left(\frac{s^3}{6} + \frac{s^2}{2} + \frac{s}{3}\right)\nabla^3\mathbf{f}_k + \dots\right] ds
\end{aligned} \tag{4.25}$$

and therefore:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + h\left(\mathbf{f}_k + \frac{1}{2}\nabla\mathbf{f}_k + \frac{5}{12}\nabla^2\mathbf{f}_k + \frac{3}{8}\nabla^3\mathbf{f}_k + \dots\right) \tag{4.26}$$

If we truncate Eq.(4.26) after the quadratic term and expand the ∇–operators, we obtain:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{12}\left(23\mathbf{f}_k - 16\mathbf{f}_{k-1} + 5\mathbf{f}_{k-2}\right) \tag{4.27}$$

which is the well–known third–order Adams–Bashforth algorithm, abbreviated as AB3. Since the expressions on the right are multiplied by h, we obtain a third–order approximation by truncating the infinite series after the quadratic term.

If we truncate Eq.(4.26) only after the cubic term, we obtain:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{24}\left(55\mathbf{f}_k - 59\mathbf{f}_{k-1} + 37\mathbf{f}_{k-2} - 9\mathbf{f}_{k-3}\right) \tag{4.28}$$

which is the AB4 algorithm.

Also these algorithms can be represented through an α–vector and a β–matrix. These are:

$$\alpha = \begin{pmatrix} 1 & 2 & 12 & 24 & 720 & 1440 \end{pmatrix}^T \tag{4.29a}$$

$$\beta = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & -1 & 0 & 0 & 0 & 0 \\ 23 & -16 & 5 & 0 & 0 & 0 \\ 55 & -59 & 37 & -9 & 0 & 0 \\ 1901 & -2774 & 2616 & -1274 & 251 & 0 \\ 4277 & -7923 & 9982 & -7298 & 2877 & -475 \end{pmatrix} \tag{4.29b}$$

Here, the i^{th} row contains the coefficients of the ABi algorithm, i.e., the coefficients of the i^{th} order Adams–Bashforth algorithm. The i^{th} row of the β–matrix contains the multipliers of the $\mathbf{f}$–vectors at different time points, and the i^{th} row of the α–vector contains the common denominator, i.e., the divider of h.

All algorithms within the class of ABi algorithms are *explicit* algorithms. Of course, AB1 is:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{1}\left(1\mathbf{f}_k\right) \tag{4.30}$$

which is immediately recognized as the FE–algorithm. There exists only one explicit first–order algorithm, namely the *Forward Euler* algorithm.

Let us now look at the stability domains of the ABi algorithms. However, before we can do so, we must find the $\mathbf{F}$–matrices of the ABi algorithms. Let

us look at AB3 for example. Applying Eq.(4.27) to the linear homogeneous problem, we find:

$$\mathbf{x}(t_{k+1}) = \left[\mathbf{I^{(n)}} + \frac{23}{12}\mathbf{A}h\right]\mathbf{x}(t_k) - \frac{4}{3}\mathbf{A}h\ \mathbf{x}(t_{k-1}) + \frac{5}{12}\mathbf{A}h\ \mathbf{x}(t_{k-2}) \quad (4.31)$$

We can transform the third–order vector differential equation into three first–order vector differential equations with the substitutions:

$$\mathbf{z_1}(t_k) = \mathbf{x}(t_{k-2}) \quad (4.32a)$$
$$\mathbf{z_2}(t_k) = \mathbf{x}(t_{k-1}) \quad (4.32b)$$
$$\mathbf{z_3}(t_k) = \mathbf{x}(t_k) \quad (4.32c)$$

With these substitutions, we find:

$$\mathbf{z_1}(t_{k+1}) = \mathbf{z_2}(t_k) \quad (4.33a)$$
$$\mathbf{z_2}(t_{k+1}) = \mathbf{z_3}(t_k) \quad (4.33b)$$
$$\mathbf{z_3}(t_{k+1}) = \frac{5}{12}\mathbf{A}h\ \mathbf{z_1}(t_k) - \frac{4}{3}\mathbf{A}h\ \mathbf{z_2}(t_k) + \left[\mathbf{I^{(n)}} + \frac{23}{12}\mathbf{A}h\right]\ \mathbf{z_3}(t_k) \quad (4.33c)$$

or, in a matrix form:

$$\mathbf{z}(t_{k+1}) = \begin{pmatrix} \mathbf{O^{(n)}} & \mathbf{I^{(n)}} & \mathbf{O^{(n)}} \\ \mathbf{O^{(n)}} & \mathbf{O^{(n)}} & \mathbf{I^{(n)}} \\ \frac{5}{12}\mathbf{A}h & -\frac{4}{3}\mathbf{A}h & (\mathbf{I^{(n)}} + \frac{23}{12}\mathbf{A}h) \end{pmatrix} \cdot \mathbf{z}(t_k) \quad (4.34)$$

Thus, for a 2×2 $\mathbf{A}$–matrix, we obtain a $2i \times 2i$ $\mathbf{F}$–matrix for ABi. The stability domains that result when plugging these $\mathbf{F}$–matrices into the general routine of Chapter 2 are shown in Fig.4.1.

As the ABi methods are explicit algorithms, their borders of stability must loop into the left–half complex $\lambda \cdot h$–plane. This was to be expected. Unfortunately, the stability domains of the ABi algorithms look very disappointing. We proceed to higher orders of approximation accuracy, in order to use *larger* step sizes ... yet, the stability domains *shrink*! AB7 is even totally unstable.

As we proceed to higher orders, the step size will very soon be limited by the stability domain rather than by the accuracy requirements. In comparison with the RK algorithms, it is true that we need only one function evaluation per step, yet, we probably will have to use considerably smaller step sizes due to the disappointingly small stable regions in the left–half $\lambda \cdot h$–plane.

The reasons for these unfortunate results are easy to understand. It is not true that higher–order polynomials necessarily lead to a more accurate interpolation everywhere. They only allow us to fit more points precisely. In between these points, higher–order polynomials have a tendency

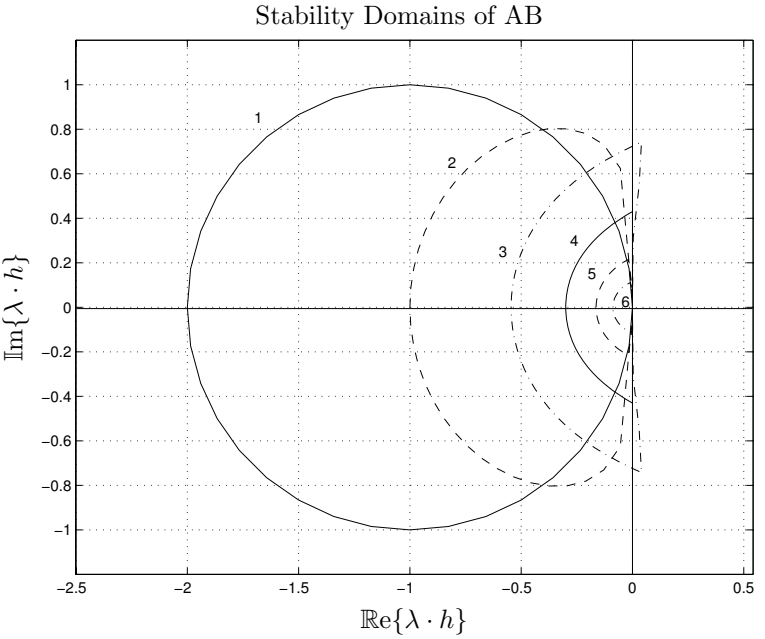

FIGURE 4.1. Stability domains of explicit AB algorithms.

to oscillate. Worse, while higher–order polynomial *interpolation* may still be acceptable, higher–order polynomial *extrapolation* is a disaster. These polynomials have a tendency to deviate quickly from the approximated curve outside the interpolation interval. Unfortunately, extrapolation is what numerical integration is all about.

The previous paragraph indicates that the discovered shortcoming of this class of algorithms will not be limited to the explicit Adams–Bashforth methods, but is an inherent disease of *all* multi–step integration algorithms.

4.5 Implicit Adams–Moulton Formulae

Let us check whether we have more luck with implicit multi–step algorithms. To this end, we again develop $\dot{\mathbf{x}}(t)$ into a Newton–Gregory backward polynomial, however this time, we shall develop the polynomial around the point t_{k+1}.

$$\dot{\mathbf{x}}(t) = \mathbf{f}_{k+1} + \binom{s}{1}\nabla\mathbf{f}_{k+1} + \binom{s+1}{2}\nabla^2\mathbf{f}_{k+1} + \binom{s+2}{3}\nabla^3\mathbf{f}_{k+1} + \dots \quad (4.35)$$

We integrate again from time t_k to time t_{k+1}. However, this time, $s = 0.0$ corresponds to $t = t_{k+1}$, thus, we need to integrate across the range $s \in$

$[-1.0, 0.0]$.

We find:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + h\left(\mathbf{f}_{k+1} - \frac{1}{2}\nabla\mathbf{f}_{k+1} - \frac{1}{12}\nabla^2\mathbf{f}_{k+1} - \frac{1}{24}\nabla^3\mathbf{f}_{k+1} + \dots\right) \tag{4.36}$$

Expanding the ∇–operators and truncating after the quadratic term, we find:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{12}\left(5\mathbf{f}_{k+1} + 8\mathbf{f}_k - \mathbf{f}_{k-1}\right) \tag{4.37}$$

which is the well–known implicit Adams–Moulton third–order algorithm, abbreviated as AM3. Truncating after the cubic term, we obtain:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + \frac{h}{24}\left(9\mathbf{f}_{k+1} + 19\mathbf{f}_k - 5\mathbf{f}_{k-1} + \mathbf{f}_{k-2}\right) \tag{4.38}$$

which is the AM4 algorithm. We can again represent the class of AMi algorithms through an α–vector and a β–matrix:

$$\alpha = \begin{pmatrix} 1 & 2 & 12 & 24 & 720 & 1440 \end{pmatrix} \tag{4.39a}$$

$$\beta = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 5 & 8 & -1 & 0 & 0 & 0 \\ 9 & 19 & -5 & 1 & 0 & 0 \\ 251 & 646 & -264 & 106 & -19 & 0 \\ 475 & 1427 & -798 & 482 & -173 & 27 \end{pmatrix} \tag{4.39b}$$

Clearly, AM1 is the same as BE. This was to be expected since there can exist only one implicit first–order integration algorithm. AM2 is the trapezoidal rule, thus while AM1 is L–stable, AM2 is F–stable.

Let us now look at the stability domains of the higher–order AMi formulae. Plugging the linear homogeneous system into AM3, we find:

$$\left[\mathbf{I}^{(\mathbf{n})} - \frac{5}{12}\mathbf{A}h\right]\mathbf{x}(t_{k+1}) = \left[\mathbf{I}^{(\mathbf{n})} + \frac{2}{3}\mathbf{A}h\right]\mathbf{x}(t_k) - \frac{1}{12}\mathbf{A}h\ \mathbf{x}(t_{k-1}) \tag{4.40}$$

Using the same substitution as in the case of the ABi formulae, we find:

$$\mathbf{F} = \begin{pmatrix} \mathbf{O}^{(\mathbf{n})} & \mathbf{I}^{(\mathbf{n})} \\ -[\mathbf{I}^{(\mathbf{n})} - \frac{5}{12}\mathbf{A}h]^{-1} \cdot [\frac{1}{12}\mathbf{A}h] & [\mathbf{I}^{(\mathbf{n})} - \frac{5}{12}\mathbf{A}h]^{-1} \cdot [\mathbf{I}^{(\mathbf{n})} + \frac{2}{3}\mathbf{A}h] \end{pmatrix} \tag{4.41}$$

Thus, for a 2×2 $\mathbf{A}$–matrix, we obtain a $2(i-1) \times 2(i-1)$ $\mathbf{F}$–matrix for AMi. The stability domains that result when plugging these $\mathbf{F}$–matrices into the general routine of Chapter 2 are shown in Fig.4.2.

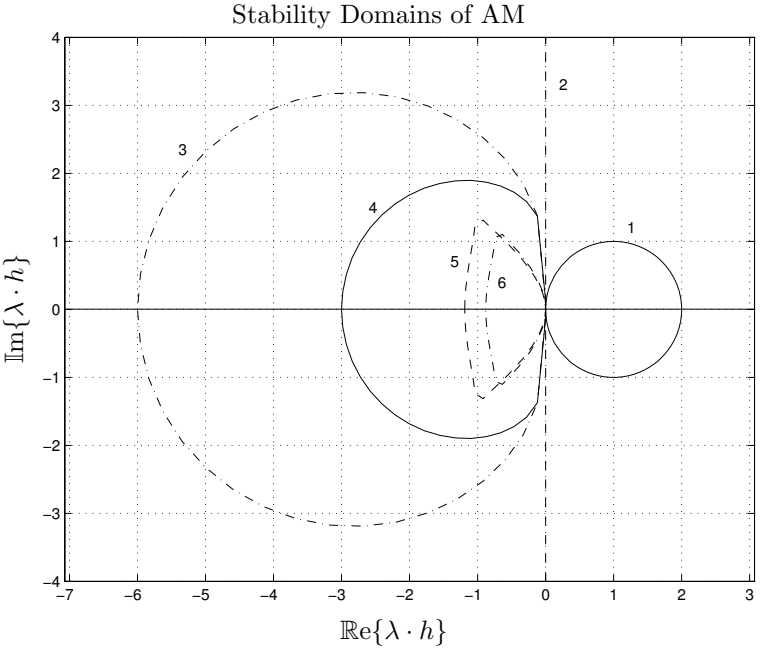

FIGURE 4.2. Stability domains of implicit AM algorithms.

As in the case of the ABi algorithms, the results are disappointing. AM1 and AM2 are useful algorithms ... but they were already known to us under different names. Starting from the third–order, the stability domains of the AMi algorithms loop again into the left–half $\lambda \cdot h$–plane. It is unclear to us why we should want to pay the high price of Newton iteration, if we don't get a stiffly–stable technique after all.

4.6 Adams–Bashforth–Moulton Predictor–Corrector Formulae

The ABi algorithms were rejected due to their miserably small stable regions in the left–half $\lambda \cdot h$–plane. The AMi algorithms, on the other hand, were rejected because they are implicit, yet not stiffly–stable. Maybe all is not lost yet. We can try a compromise between ABi and AMi. Let us construct a predictor–corrector method with one step of ABi as a predictor, and one step of AMi as a corrector. For example, the (third–order accurate) ABM3 algorithm would look as follows:

$$\begin{aligned}
\text{predictor:} \quad & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) \\
& \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + \tfrac{h}{12}(23\dot{\mathbf{x}}_{\mathbf{k}} - 16\dot{\mathbf{x}}_{\mathbf{k-1}} + 5\dot{\mathbf{x}}_{\mathbf{k-2}}) \\
\text{corrector:} \quad & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) \\
& \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + \tfrac{h}{12}(5\dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} + 8\dot{\mathbf{x}}_{\mathbf{k}} - \dot{\mathbf{x}}_{\mathbf{k-1}})
\end{aligned}$$

Evidently, the overall algorithm is explicit. Therefore, no Newton iteration is needed, and consequently, the fact that the method won't be stiffly–stable is of no concern. However, for the price of a second function evaluation per step, we may have bargained for a considerably larger stability domain than in the case of AB3.

Replacing the nonlinear problem by the linear homogeneous problem in the predictor–corrector technique, and plugging the predictor formula into the corrector, we find:

$$\begin{aligned}
\mathbf{x}(t_{k+1}) = \left[\mathbf{I}^{(\mathbf{n})} + \frac{13}{12}\mathbf{A}h + \frac{115}{144}(\mathbf{A}h)^2\right]\mathbf{x}(t_k) - \left[\frac{1}{12}\mathbf{A}h + \frac{5}{9}(\mathbf{A}h)^2\right]\mathbf{x}(t_{k-1}) \\
+ \frac{25}{144}(\mathbf{A}h)^2\mathbf{x}(t_{k-2})
\end{aligned} \tag{4.42}$$

with the **F**–matrix:

$$\mathbf{F} = \begin{pmatrix} \mathbf{O}^{(\mathbf{n})} & \mathbf{I}^{(\mathbf{n})} & \mathbf{O}^{(\mathbf{n})} \\ \mathbf{O}^{(\mathbf{n})} & \mathbf{O}^{(\mathbf{n})} & \mathbf{I}^{(\mathbf{n})} \\ \frac{25}{144}(\mathbf{A}h)^2 & -\left[\frac{1}{12}\mathbf{A}h + \frac{5}{9}(\mathbf{A}h)^2\right] & \left[\mathbf{I}^{(\mathbf{n})} + \frac{13}{12}\mathbf{A}h + \frac{115}{144}(\mathbf{A}h)^2\right] \end{pmatrix} \tag{4.43}$$

The stability domains of some ABMi algorithms are shown in Fig.4.3.

Indeed, the approach worked. The stability domains of the ABMi algorithms are considerably larger than those of the ABi algorithms, although they are still considerably smaller than those of the AMi algorithms — especially for orders three and four. Since Newton iteration takes usually about three iterations per step, i.e., three additional function evaluations in the case of these multi–step algorithms, ABMi is about twice as expensive as ABi, and AMi is about twice as expensive as ABMi. Thus, if ABMi allows us to use a step size that is at least twice as large as the step size we could employ when using ABi, the predictor–corrector method becomes economical. If AMi allows us to use a step size that is at least four times as large as the step size we could employ when using ABi, the implicit algorithm becomes economical in spite of the need for Newton iteration.

4.7 Backward Difference Formulae

Let us check whether we can find a set of multi–step formulae whose stability domains loop into the right–half $\lambda \cdot h$–plane. This time, we write the

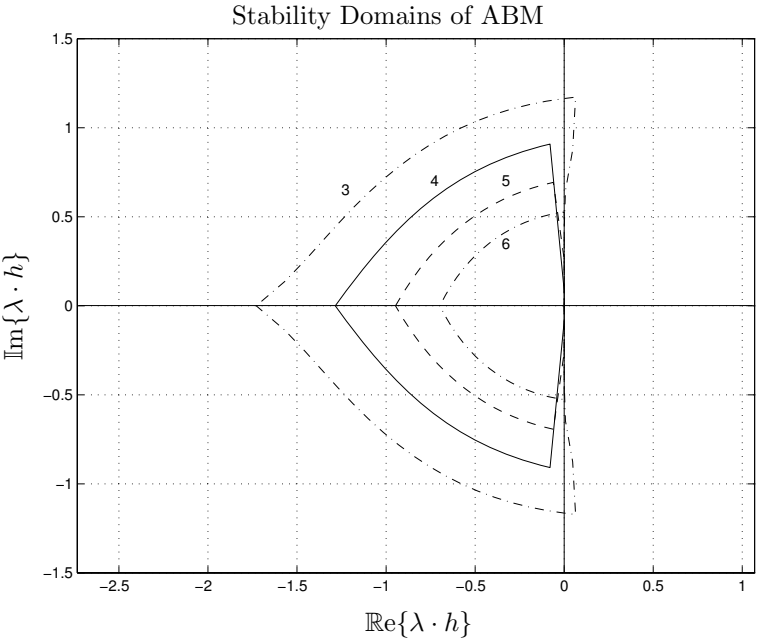

FIGURE 4.3. Stability domains of predictor–corrector ABM algorithms.

Newton–Gregory backward polynomial in $\mathbf{x}(t)$ rather than in $\dot{\mathbf{x}}(t)$ around the time instant t_{k+1}. Thus:

$$\mathbf{x}(t) = \mathbf{x}_{k+1} + \binom{s}{1}\nabla\mathbf{x}_{k+1} + \binom{s+1}{2}\nabla^2\mathbf{x}_{k+1} + \binom{s+2}{3}\nabla^3\mathbf{x}_{k+1} + \dots \quad (4.44)$$

or:

$$\mathbf{x}(t) = \mathbf{x}_{k+1} + s\nabla\mathbf{x}_{k+1} + \left(\frac{s^2}{2} + \frac{s}{2}\right)\nabla^2\mathbf{x}_{k+1} + \left(\frac{s^3}{6} + \frac{s^2}{2} + \frac{s}{3}\right)\nabla^3\mathbf{x}_{k+1} + \dots \quad (4.45)$$

We now compute the derivative of Eq.(4.45)) with respect to time:

$$\dot{\mathbf{x}}(t) = \frac{1}{h}\left[\nabla\mathbf{x}_{k+1} + \left(s + \frac{1}{2}\right)\nabla^2\mathbf{x}_{k+1} + \left(\frac{s^2}{2} + s + \frac{1}{3}\right)\nabla^3\mathbf{x}_{k+1} + \dots\right] \quad (4.46)$$

We evaluate Eq.(4.46) for $s = 0.0$, and obtain:

$$\dot{\mathbf{x}}(t_{k+1}) = \frac{1}{h}\left[\nabla\mathbf{x}_{k+1} + \frac{1}{2}\nabla^2\mathbf{x}_{k+1} + \frac{1}{3}\nabla^3\mathbf{x}_{k+1} + \dots\right] \quad (4.47)$$

Multiplying Eq.4.47 with h, truncating after the cubic term, and expanding the ∇–operators, we obtain:

$$h \cdot \mathbf{f}_{k+1} = \frac{11}{6}\mathbf{x}_{k+1} - 3\mathbf{x}_k + \frac{3}{2}\mathbf{x}_{k-1} - \frac{1}{3}\mathbf{x}_{k-2} \tag{4.48}$$

Eq.(4.48) can be solved for $\mathbf{x}_{k+1}$:

$$\mathbf{x}_{k+1} = \frac{18}{11}\mathbf{x}_k - \frac{9}{11}\mathbf{x}_{k-1} + \frac{2}{11}\mathbf{x}_{k-2} + \frac{6}{11} \cdot h \cdot \mathbf{f}_{k+1} \tag{4.49}$$

which is the well–known third–order backward difference formula, abbreviated as BDF3. We can obtain BDFi algorithms of other orders by truncating Eq.(4.47) after fewer or more terms.

Also the BDFi algorithms can be expressed through an α–vector and a β–matrix:

$$\alpha = \begin{pmatrix} 1 & 2/3 & 6/11 & 12/25 & 60/137 \end{pmatrix}^T \tag{4.50a}$$

$$\beta = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 4/3 & -1/3 & 0 & 0 & 0 \\ 18/11 & -9/11 & 2/11 & 0 & 0 \\ 48/25 & -36/25 & 16/25 & -3/25 & 0 \\ 300/137 & -300/137 & 200/137 & -75/137 & 12/137 \end{pmatrix} \tag{4.50b}$$

where the i^{th} row represents the BDFi algorithm. The coefficients of the β–matrix are here the multipliers of past values of the state vector $\mathbf{x}$, whereas the coefficients of the α–vector are the multipliers of the state derivative vector $\dot{\mathbf{x}}$ at time t_{k+1}. The BDF techniques are implicit algorithms, thus clearly, BDF1 is the same as BE.

The stability domains of the BDFi algorithms are presented in Fig.4.4.

It becomes evident at once that, finally, we have found a set of stiffly–stable multi–step algorithms. Unfortunately, they (not unexpectedly) also suffer from the high–order polynomial extrapolation disease. As the order of the extrapolation polynomials grows, the methods become less and less stable. Although the stability domains loop into the right–half $\lambda \cdot h$–plane, they are pulled over more and more into the left–half plane. BDF6 (not shown on Fig.4.4) has only a very narrow band of stable area to the left of the origin. BDF6 is thus only useful for simulation of problems with all eigenvalues strictly on the negative real axis, such as method–of–lines solutions to parabolic PDEs. BDF7, is unstable in the entire $\lambda \cdot h$–plane.

Yet, due to the simplicity of these techniques, the BDFi algorithms are today easily the most widely used stiff system solvers on the market. In the engineering literature, these algorithms are often called *Gear algorithms*, after Bill Gear who discovered their stiffly–stable properties [4.5]. The most widely used code based o the BDF formulae is DASSL. DASSL is the default simulation algorithm used in Dymola. We shall talk more about DASSL in Chapter 8 of this book.

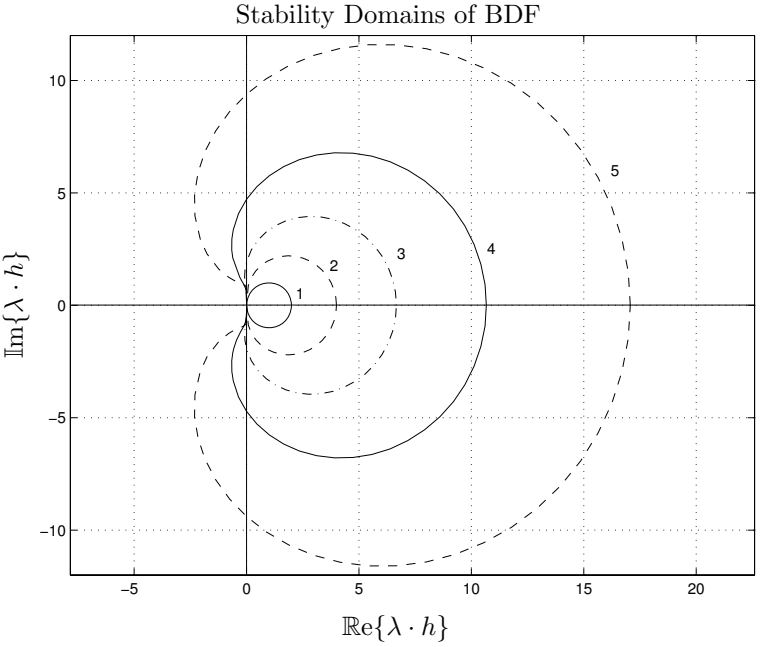

FIGURE 4.4. Stability domains of implicit BDF algorithms.

By evaluating Eq.(4.46) for $s = -1.0$, we can obtain a series of explicit BDFi algorithms. Unfortunately, they are not useful, since they are all unstable in the entire $\lambda \cdot h$–plane.

4.8 Nyström and Milne Algorithms

There exist two more classes of multi–step techniques that are sometimes talked about in the numerical ODE literature, the explicit *Nyström techniques* [4.10], and the implicit *Milne methods* [4.10]. Let us derive them and look at their stability behavior.

We start again out with Eq.(4.22). However this time, we integrate from t_{k-1} to t_{k+1}, thus, from $s = -1.0$ to $s = +1.0$. We find:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_{k-1}) + h\left(2\mathbf{f}_k + \frac{1}{3}\nabla^2\mathbf{f}_k + \frac{1}{3}\nabla^3\mathbf{f}_k + \dots\right) \tag{4.51}$$

The term in $\nabla\mathbf{f}_k$ drops out. Truncating Eq.(4.51) after the cubic term and expanding the ∇–operators, we obtain:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_{k-1}) + \frac{h}{3}\left(8\mathbf{f}_k - 5\mathbf{f}_{k-1} + 4\mathbf{f}_{k-2} - \mathbf{f}_{k-3}\right) \tag{4.52}$$

which is the fourth–order Nyström algorithm, abbreviated as Ny4.

The Nyi algorithms are characterized by the following α–vector and β–matrix:

$$\alpha = \begin{pmatrix} 1 & 1 & 3 & 3 & 90 \end{pmatrix}^T \tag{4.53a}$$

$$\beta = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 7 & -2 & 1 & 0 & 0 \\ 8 & -5 & 4 & -1 & 0 \\ 269 & -266 & 294 & -146 & 29 \end{pmatrix} \tag{4.53b}$$

The Nyström algorithms have unfortunately a serious drawback. They are unstable in the entire $\lambda \cdot h$–plane. Ny1 is the explicit midpoint rule with a double step size, which by accident is already 2^{nd}–order accurate. Ny2 is the same algorithm as Ny1. Even Ny2 (Ny1) is an unstable algorithm though, since it is interpreted here as a multi–step technique, rather than as a single–step algorithm with an FE predictor step, as proposed in Chapter 3.

If we start out with Eq.4.35, but integrate from time t_{k-1} to time t_{k+1}, corresponding to the interval $s \in [-2.0, 0.0]$, we get:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_{k-1}) + h\left(2\mathbf{f}_{k+1} - 2\nabla\mathbf{f}_{k+1} + \frac{1}{3}\nabla^2\mathbf{f}_{k+1} + 0\nabla^3\mathbf{f}_{k+1} + \dots\right) \tag{4.54}$$

This time around, the term in $\nabla^3\mathbf{f}_{k+1}$ drops out. Truncating Eq.4.54 after the cubic term (the quadratic term really) and expanding the ∇–operators, we find:

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_{k-1}) + \frac{h}{3}\left(\mathbf{f}_{k+1} + 4\mathbf{f}_k + \mathbf{f}_{k-1}\right) \tag{4.55}$$

which is the implicit fourth–order *Milne algorithm*, abbreviated as Mi4. The same algorithm is also known under the name of *Simpson's rule*.

The α–vector and β–matrix for the Mii algorithms are as follows:

$$\alpha = \begin{pmatrix} 1 & 1 & 3 & 3 & 90 \end{pmatrix}^T \tag{4.56a}$$

$$\beta = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 29 & 124 & 24 & 4 & -1 \end{pmatrix} \tag{4.56b}$$

Mi1 is recognizable as backward Euler with a double step size. Mi2 is by accident explicit, since the coefficient of $\mathbf{f}_{k+1}$ drops out. Mi2 is the same as Ny2, i.e., the explicit midpoint rule with a double step size.

Mi4 is truly remarkable. Due to a combination of "lucky" circumstances, a lot of terms dropped away, leading to a fourth–order accurate multi–step methods with only two memory elements (two past values are used in the algorithm). It is truly regrettable that our "good fortune" comes at a high price. The stability domain of Mi4 is extremely small — it consists of the origin only (!) Therefore, while Simpson's rule is very fashionable for quadrature problems (to numerically determine the integral of a function), it is entirely useless for solving differential equations. The higher–order Milne formulae are all unstable as well.

Nyström and Milne formulae are sometimes useful as partners within predictor–corrector methods. The fact that these formulae are unstable by themselves does not preclude the possibility that they may be combined with other formulae either in a predictor-corrector scheme, or in a blended method, or in a cyclic method, thereby leading to perfectly usable algorithms with appropriate stability properties.

4.9 In Search for Stiffly–stable Methods

Until now, we used the Newton–Gregory polynomials to derive multi–step algorithms. This technique has its advantages. In particular, it generates the integration algorithms using the ∇–operator, which is useful. However, the technique called for lots of symbolic or at least semi–symbolic operations that are hard to implement in MATLAB in search for new algorithms with pre–specified stability and/or accuracy properties.

To this end, let us introduce another technique that can alternatively be used to derive the coefficients of multi–step integration algorithms. Let us derive once again the BDF3 algorithm.

We know that every n^{th}–order multi–step algorithm is defined through an n^{th}–order polynomial fitted through $(n+1)$ points, a mixture of state values and state derivative values, at the time points t_{k+1}, t_k, t_{k-1}, etc. Let us write this polynomial once again in the variable s, assuming that $s = 1.0$ corresponds to $t = t_{k+1}$, $s = 0.0$ corresponds to $t = t_k$, etc. The polynomial can be written as:

$$p(s) = a_0 + a_1\ s + a_2\ s^2 + a_3\ s^3 + \cdots + a_n\ s^n \tag{4.57}$$

in the yet unknown coefficients a_i. The time derivative of $p(s)$ can be written as:

$$h \cdot \dot{p}(s) = a_1 + 2a_2\ s + 3a_3\ s^2 + \cdots + n\ a_n\ s^{n-1} \tag{4.58}$$

In the case of BDF3, we know that $p(0) = x_k$, $p(-1) = x_{k-1}$, $p(-2) = x_{k-2}$, and $h \cdot \dot{p}(+1) = h \cdot f_{k+1}$. This gives us four equations in the four unknowns a_0, a_1, a_2, and a_3. These are:

$$h \cdot f_{k+1} = a_1 + 2a_2 + 3a_3 \tag{4.59a}$$
$$x_k = a_0 \tag{4.59b}$$
$$x_{k-1} = a_0 - a_1 + a_2 - a_3 \tag{4.59c}$$
$$x_{k-2} = a_0 - 2a_1 + 4a_2 - 8a_3 \tag{4.59d}$$

or, in a matrix form:

$$\begin{pmatrix} h \cdot f_{k+1} \\ x_k \\ x_{k-1} \\ x_{k-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & -2 & 4 & -8 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \tag{4.60}$$

which can be solved for the unknown parameter vector by matrix inversion. We wish to evaluate:

$$x_{k+1} = p(+1) = a_0 + a_1 + a_2 + a_3 \tag{4.61}$$

Thus, we simply add up the elements in each column of the inverse matrix, and receive directly the desired coefficients of BDF3:

$$x_{k+1} = \frac{6}{11} h \cdot f_{k+1} + \frac{18}{11} x_k - \frac{9}{11} x_{k-1} + \frac{2}{11} x_{k-2} \tag{4.62}$$

This procedure can easily be automated in a MATLAB function.

You had been told that BDF6 is not a very useful technique due to its narrow corridor of stable territory to the left of the origin. The method is (A,α)–stable, but only with $\alpha = 19^o$. Let us ascertain whether the above outlined procedure allows us to find a better sixth–order stiffly–stable algorithm than BDF6.

Obviously, any sixth–order linear multi–step method can be written as:

$$p(s) = a_0 + a_1\ s + a_2\ s^2 + a_3\ s^3 + a_4\ s^4 + a_5\ s^5 + a_6\ s^6 \tag{4.63}$$

in seven unknown parameters. Consequently, we must provide seven solution points through which the polynomial will be fitted. In the past, we talked about the high–order extrapolation disease. Maybe, it will work better if we shorten the tail of the algorithm by providing both values for x and for f at $s = 0$, at $s = -1$, and at $s = -2$. Clearly, the list of data points *must* contain $f(t_{k+1})$ in order for the method to be implicit. The rationale for this idea is that the interpolated curve may look more like a polynomial over a shorter time span.

Thus:

$$\begin{pmatrix} h \cdot f_{k+1} \\ x_k \\ h \cdot f_k \\ x_{k-1} \\ h \cdot f_{k-1} \\ x_{k-2} \\ h \cdot f_{k-2} \end{pmatrix} = \mathbf{M} \cdot \mathbf{a} \tag{4.64}$$

where:

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & 1 & -2 & 3 & -4 & 5 & -6 \\ 1 & -2 & 4 & -8 & 16 & -32 & 64 \\ 0 & 1 & -4 & 12 & -32 & 80 & -192 \end{pmatrix} \tag{4.65}$$

Computing the inverse of $\mathbf{M}$ and adding up columns, we find:

$$\mathbf{x_{k+1}} = \frac{3}{11}h\cdot\mathbf{f_{k+1}} - \frac{27}{11}\mathbf{x_k} + \frac{27}{11}h\cdot\mathbf{f_k} + \frac{27}{11}\mathbf{x_{k-1}} + \frac{27}{11}h\cdot\mathbf{f_{k-1}} + \mathbf{x_{k-2}} + \frac{3}{11}h\cdot\mathbf{f_{k-2}} \tag{4.66}$$

This is a beautiful new method, it is clearly sixth–order accurate, and it has only one single drawback ... it is unfortunately unstable in the entire $\lambda \cdot h$–plane (!)

Thus, we need to expand our search. Let us allow values to be included as far back as $t = t_{k-5}$, corresponding to $s = -5$. Since we definitely want $\mathbf{f_{k+1}}$ to be included and since we can pick either $\mathbf{x}$–values or $\mathbf{f}$–values otherwise, we need to pick six items out of 12. This gives us 924 different methods to try.

I quickly programmed this problem in MATLAB, calculating the $\mathbf{F}$–matrices for all 924 techniques, and checking for each of them whether or not its stability domain intersects with the positive real axis, making it a potential candidate for a stiffly–stable algorithm.

Most of the 924 methods are entirely unstable. Others behave like Adams–Moulton. Only six out of the 924 methods have an intersection of their respective stability domains with the positive real axis. These are:

$$\begin{aligned} \mathbf{x_{k+1}} = & \frac{20}{49}h\cdot\mathbf{f_{k+1}} + \frac{120}{49}\mathbf{x_k} - \frac{150}{49}\mathbf{x_{k-1}} + \frac{400}{147}\mathbf{x_{k-2}} - \frac{75}{49}\mathbf{x_{k-3}} \\ & + \frac{24}{49}\mathbf{x_{k-4}} - \frac{10}{147}\mathbf{x_{k-5}} \end{aligned} \tag{4.67a}$$

$$\mathbf{x_{k+1}} = \frac{308}{745} h \cdot \mathbf{f_{k+1}} + \frac{1776}{745}\mathbf{x_k} - \frac{414}{149}\mathbf{x_{k-1}} + \frac{944}{447}\mathbf{x_{k-2}} - \frac{87}{149}\mathbf{x_{k-3}} - \frac{288}{745} h \cdot \mathbf{f_{k-4}} - \frac{2}{15}\mathbf{x_{k-5}} \tag{4.67b}$$

$$\mathbf{x_{k+1}} = \frac{8820}{21509} h \cdot \mathbf{f_{k+1}} + \frac{52200}{21509}\mathbf{x_k} - \frac{63900}{21509}\mathbf{x_{k-1}} + \frac{400}{157}\mathbf{x_{k-2}} - \frac{28575}{21509}\mathbf{x_{k-3}} + \frac{6984}{21509}\mathbf{x_{k-4}} + \frac{600}{21509} h \cdot \mathbf{f_{k-5}} \tag{4.67c}$$

$$\mathbf{x_{k+1}} = \frac{179028}{432845} h \cdot \mathbf{f_{k+1}} + \frac{206352}{86569}\mathbf{x_k} - \frac{34452}{12367}\mathbf{x_{k-1}} + \frac{26704}{12367}\mathbf{x_{k-2}} - \frac{65547}{86569}\mathbf{x_{k-3}} - \frac{83808}{432845} h \cdot \mathbf{f_{k-4}} + \frac{24}{581} h \cdot \mathbf{f_{k-5}} \tag{4.67d}$$

$$\mathbf{x_{k+1}} = \frac{12}{29} h \cdot \mathbf{f_{k+1}} + \frac{1728}{725}\mathbf{x_k} - \frac{81}{29}\mathbf{x_{k-1}} + \frac{64}{29}\mathbf{x_{k-2}} - \frac{27}{29}\mathbf{x_{k-3}} + \frac{97}{725}\mathbf{x_{k-5}} + \frac{12}{145} h \cdot \mathbf{f_{k-5}} \tag{4.67e}$$

$$\mathbf{x_{k+1}} = \frac{30}{71} h \cdot \mathbf{f_{k+1}} + \frac{162}{71}\mathbf{x_k} - \frac{675}{284}\mathbf{x_{k-1}} + \frac{100}{71}\mathbf{x_{k-2}} - \frac{54}{71}\mathbf{x_{k-4}} + \frac{127}{284}\mathbf{x_{k-5}} + \frac{15}{71} h \cdot \mathbf{f_{k-5}} \tag{4.67f}$$

Among those six methods, Eq.(4.67a) is the well-known BDF6 technique. The methods of Eq.(4.67b) and Eq.(4.67f) are useless, since their stability domains also intersect with the negative real axis. The stability domains of the survivors are shown in Fig.4.5.

How can we evaluate the relative merits of these four algorithms against each other? One useful criterium is the angle α of the $A(\alpha)$ stability. Gear [4.5] proposed an alternate method to judge the stability of a stiffly–stable method consisting of two parameters, the distance a away from the imaginary axis that the stability domain reaches into the left–half $\lambda \cdot h$–plane, and the distance c away from the negative real axis, which defines the closest distance of the stability domain to the negative real axis. These three parameters are shown in Fig.4.6.

Yet, we shall need to look at accuracy as well. It has become customary [4.10] to judge the accuracy of a multi–step method in the following way. Any multi–step method can be written in the form:

$$\mathbf{x_{k+1}} = \sum_{i=0}^{\ell} a_i \cdot \mathbf{x_{k-i}} + \sum_{i=-1}^{\ell} b_i \cdot h \cdot \mathbf{f_{k-i}} \tag{4.68}$$

where ℓ is a suitably large index to include the entire history needed for the method. We can take all terms to the left–hand side of the equal sign, and shift the equation by ℓ steps into the future. By doing so, we obtain:

$$\sum_{i=0}^{m} \alpha_i \cdot \mathbf{x_{k+i}} + h \cdot \sum_{i=0}^{m} \beta_i \cdot \mathbf{f_{k+i}} = 0 \tag{4.69}$$

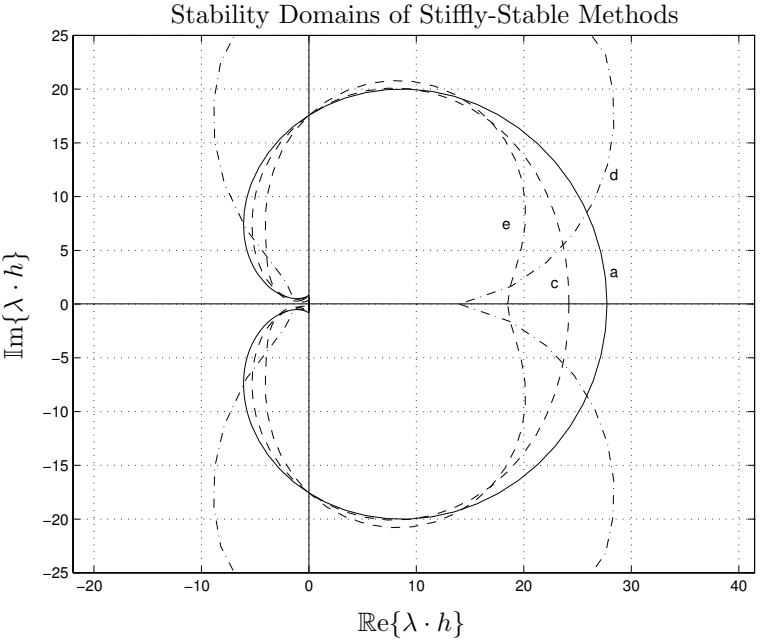

FIGURE 4.5. Stability domains of some stiffly–stable algorithms.

where α_i, β_i, and m can be easily expressed in terms of the previously used parameters a_i, b_i, and ℓ. Assuming that our system is *smooth*, i.e., the solution is continuous and continuously differentiable at least n times, where n is the order of the integration algorithm, we can develop $\mathbf{x_{k+i}}$ and $\mathbf{f_{k+i}}$ into Taylor series around $\mathbf{x_k}$ and $\mathbf{f_k}$, and come up with an expression in $\mathbf{x_k}$ and its derivatives:

$$c_0 \cdot \mathbf{x_k} + c_1 \cdot h \cdot \dot{\mathbf{x}}_\mathbf{k} + \cdots + c_q \cdot h^q \cdot \mathbf{x_k^{(q)}} + \ldots \tag{4.70}$$

where $\mathbf{x_k^{(q)}}$ is the q^{th} time derivative of $\mathbf{x_k}$. The coefficients can be expressed in terms of the previously used parameters α_i, β_i, and m as follows:

$$c_0 = \sum_{i=0}^{m} \alpha_i \tag{4.71}$$

$$c_1 = \sum_{i=0}^{m} (i \cdot \alpha_i - \beta_i) \tag{4.72}$$

$$\vdots \tag{4.73}$$

$$c_q = \sum_{i=0}^{m} \left(\frac{1}{q!} \cdot i^q \cdot \alpha_i - \frac{1}{(q-1)!} \cdot i^{q-1} \cdot \beta_i \right), \quad q = 2, 3, \ldots \tag{4.74}$$

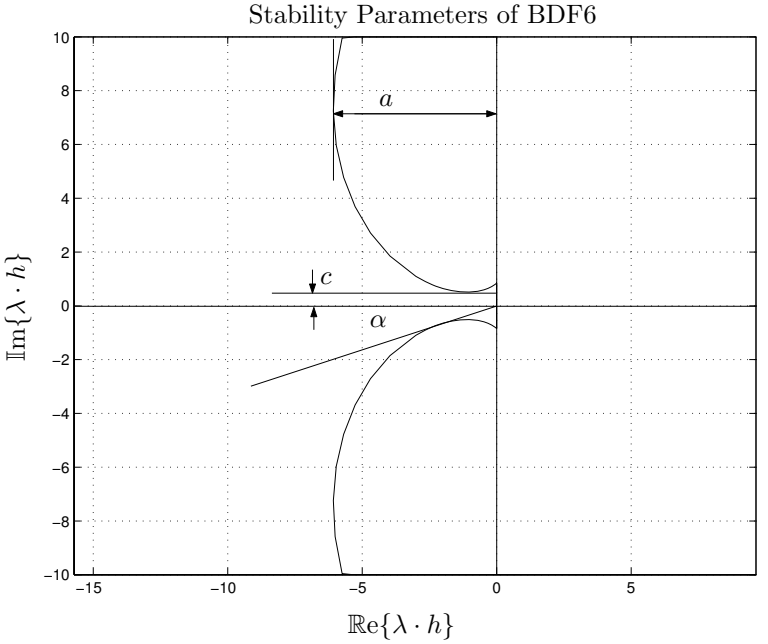

FIGURE 4.6. Stability parameters of a stiffly–stable algorithm.

Since the function that has been developed into a Taylor series is the zero function, all of these coefficients ought to be equal to zero. However, since the approximation is only n^{th}–order accurate, the coefficients for $q > n$ may be different from zero. Hence we can define the dominant of those coefficients as the *error coefficient* of the multi–step integration algorithm:

$$c_{err} = \sum_{i=0}^{m} \left(\frac{1}{(n+1)!} \cdot i^{n+1} \cdot \alpha_i - \frac{1}{n!} \cdot i^n \cdot \beta_i \right) \tag{4.75}$$

A small value of the error coefficient is indicative of a good n^{th}–order multi–step formula.

We may also wish to look at the damping properties of the algorithm. The *damping plot*, that had been introduced in Chapter 3 of this book, can easily be extended to multi-step methods by redefining the *discrete damping* as:

$$\sigma_d = -\log(\max(\text{abs}(\text{eig}(\mathbf{F})))) \tag{4.76}$$

The damping plots of BDF6 and the other three surviving algorithms are shown in Fig.4.7. The top graph shows the damping plots as depicted in the past. The bottom graph shows the same plots using a semi–logarithmic scale for the independent variable.

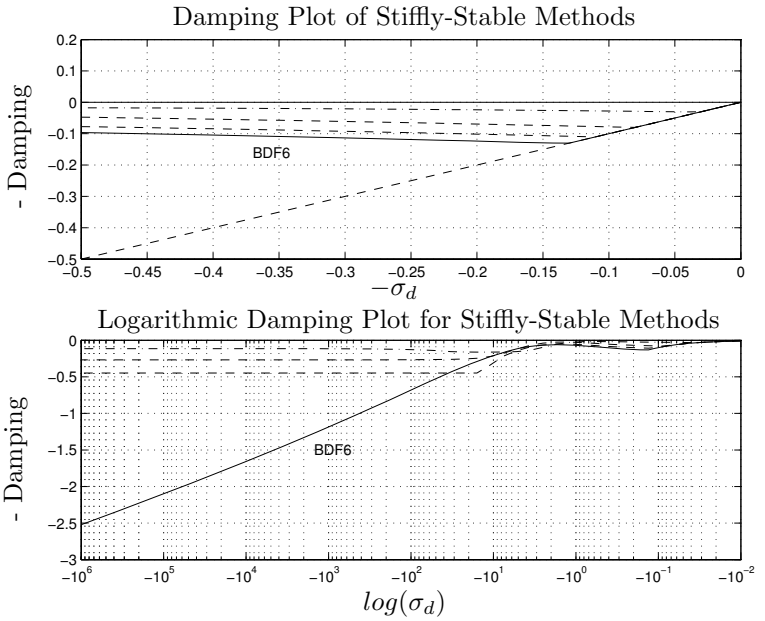

FIGURE 4.7. Damping plots of BDF6 and other 6^{th}–order stiffly–stable algorithms.

The top graph of Fig.4.7 shows that the *asymptotic region* of BDF6 is clearly larger than that of its three competitors. The graph shows furthermore that the discrete damping of the method exhibits a sharp bend at the place where it starts deviating from the analytical damping, i.e., at approximately $\sigma_d = -0.125$. This bend is caused by a *spurious eigenvalue* taking over at that point. This is a new phenomenon that we didn't observe in the case of the single–step algorithms. Since the **F**–matrix is of a larger size than the **A**–matrix, it has more eigenvalues, which may become dominant eventually.

The bottom graph shows that BDF6 is *L–stable*, whereas its three contenders are not. Although the concept of L–stability is somewhat overrated in the numerical ODE literature (at $\sigma_d = 10^6$, the discrete damping of the BDF6 method assumes only a value of $\hat{\sigma}_d = 2.5$), this is still a nice property for a stiffly–stable method to possess.

Hence, and in spite of all our efforts, we haven't hit a mark yet. BDF6 is still the winner.

Looking at the surviving algorithms, we notice at once that all of them make use of the entire interpolation span. Quite obviously, it was a lousy idea to try to shorten the tail of the algorithm. The reason for this result is also understandable. By extending the interpolation range, the relative distance of extrapolation necessary to predict $\mathbf{x_{k+1}}$ becomes shorter. This is beneficial. Thus, let us extend this idea, and allow also interpolation points

at time t_{k-6}, t_{k-7}, etc without increasing the order of the polynomial.

We decided to extend the search all the way to t_{k-11}. Although it would have been possible to include both previous state values and previous derivative values in the search, we limited our search to past state values only, since no stiffly–stable method makes use of past derivative values. We have to pick six values out of 12.

Some 924 methods later ...

314 methods were found that exhibit properties similar to those of BDF6. Their performance parameters are summarized in Table 4.1.

BDF6	Other stiffly–stable methods
$\alpha = 19°$	$\alpha \in [19°, 48°]$
$a = -6.0736$	$a \in [-6.0736, -0.6619]$
$c = 0.5107$	$c \in [0.2250, 0.8316]$
$c_{err} = -0.0583$	$c_{err} \in [-7.4636, -0.0583]$
$as.reg. = -0.14$	$as.reg. \in [-0.30, -0.01]$

TABLE 4.1. Properties of stiffly–stable 6^{th}–order algorithms.

Evidently, BDF6 exhibits the worst behavior of all of these algorithms w.r.t. its α and a values. Yet, BDF6 is characterized by the smallest error coefficient. Unfortunately, as the length of the tail of the algorithm grows, so does the error coefficient. The c parameter is somewhere in the middle range, and so is the asymptotic region, which we defined as the value of σ_d, where $|\hat{\sigma}_d - \sigma_d| = 0.01$.

How can these 314 algorithms be rank–ordered? To this end, we shall need to define a performance index, something along the lines of:

$$P.I._i = \frac{|\alpha_i|}{\|\alpha\|} - \frac{|a_i|}{\|a\|} + \frac{|c_i|}{\|c\|} - k \cdot \frac{|c_{err_i}|}{\|c_{err}\|} + \frac{|as.reg._i|}{\|as.reg.\|} = max! \tag{4.77}$$

where each of the five parameters is normalized to make them comparable with each other. A k–factor was assigned to the error coefficient to be able to vary the importance of the error coefficient within the overall performance index. We chose a value of $k = 20$.

The best three methods are now compared with BDF6. Their coefficients are given by:

$$\begin{aligned}\mathbf{x_{k+1}} =& \frac{72}{167}h \cdot \mathbf{f_{k+1}} + \frac{2592}{1169}\mathbf{x_k} - \frac{2592}{1169}\mathbf{x_{k-1}} + \frac{1152}{835}\mathbf{x_{k-2}} \\ & - \frac{324}{835}\mathbf{x_{k-3}} + \frac{81}{5845}\mathbf{x_{k-7}} - \frac{32}{5845}\mathbf{x_{k-8}} \qquad (4.78a) \\ \mathbf{x_{k+1}} =& \frac{420}{977}h \cdot \mathbf{f_{k+1}} + \frac{19600}{8793}\mathbf{x_k} - \frac{2205}{977}\mathbf{x_{k-1}} + \frac{1400}{977}\mathbf{x_{k-2}}\end{aligned}$$

$$- \frac{1225}{2931}\mathbf{x_{k-3}} + \frac{40}{2931}\mathbf{x_{k-6}} - \frac{7}{8793}\mathbf{x_{k-9}} \tag{4.78b}$$

$$\mathbf{x_{k+1}} = \frac{44}{103}h \cdot \mathbf{f_{k+1}} + \frac{5808}{2575}\mathbf{x_k} - \frac{242}{103}\mathbf{x_{k-1}} + \frac{484}{309}\mathbf{x_{k-2}} - \frac{363}{721}\mathbf{x_{k-3}} + \frac{242}{7725}\mathbf{x_{k-5}} - \frac{4}{18025}\mathbf{x_{k-10}} \tag{4.78c}$$

Table 4.2 summarizes the five performance parameters of these methods.

BDF6	SS6a	SS6b	SS6c
$\alpha = 19°$	$\alpha = 45°$	$\alpha = 44°$	$\alpha = 43°$
$a = -6.0736$	$a = -2.6095$	$a = -2.7700$	$a = -3.0839$
$c = 0.5107$	$c = 0.7994$	$c = 0.8048$	$c = 0.8156$
$c_{err} = -0.0583$	$c_{err} = -0.1478$	$c_{err} = -0.1433$	$c_{err} = -0.1343$
$as.reg. = -0.14$	$as.reg. = -0.21$	$as.reg. = -0.21$	$as.reg. = -0.21$

TABLE 4.2. Properties of stiffly–stable 6^{th}–order algorithms.

The stability domains of the four methods are presented in Fig.4.8. The damping plots are shown in Fig.4.9.

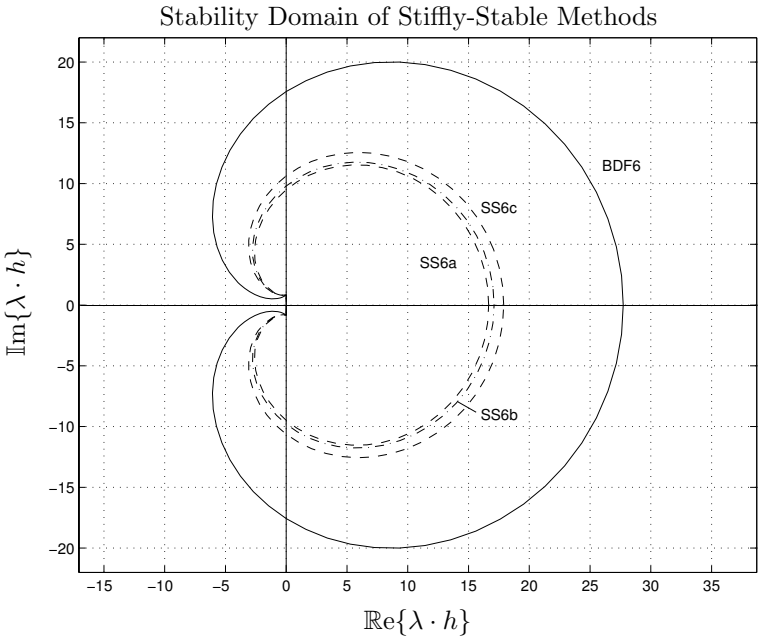

FIGURE 4.8. Stability domains of BDF6 and other 6^{th}–order stiffly–stable algorithms.

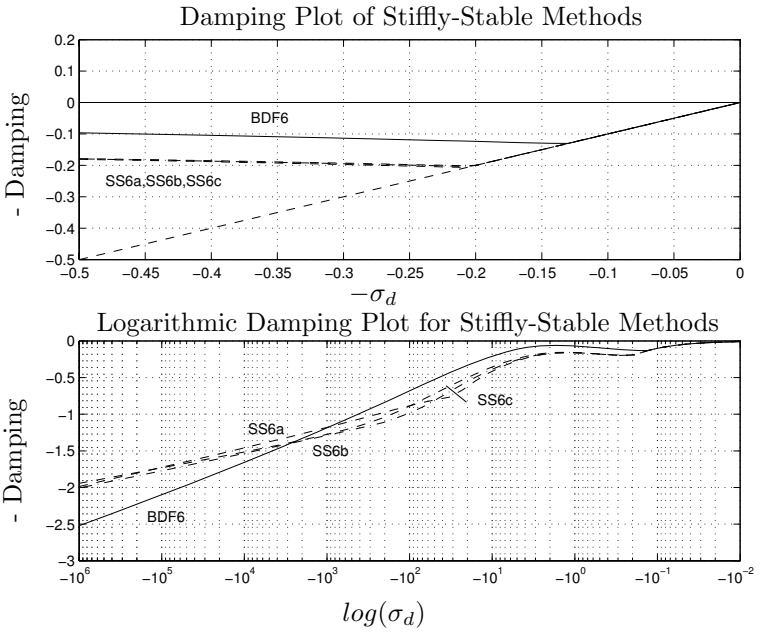

FIGURE 4.9. Damping plots of BDF6 and other 6^{th}–order stiffly–stable algorithms.

The asymptotic regions of these three algorithms are almost identical and considerably larger than that of BDF6. Consequently, it may be possible to use larger step sizes with any of these algorithms. Thus, either of these methods may be more economic than BDF6. A large asymptotic region may be considered an alternative to a small error coefficient as an indicator for a good algorithm from the point of view of integration accuracy.

The logarithmic decay rate of BDF6 is a little better than those of its three contenders. Yet, this issue may not be of much concern.

4.10 High–order Backward Difference Formulae

Although it is known that there are no stable BDF algorithms of orders higher than six, this statement only applies to the algorithms without extended memory tail. We can apply the same procedure as before to search for BDF algorithms of order seven, allowing the tail of the algorithm to reach back all the way to e.g. t_{k-13}. Hence we need to choose seven elements from a list of 14.

Of the possible 3432 algorithms, 762 possess properties similar to BDF6, i.e., they are $A(\alpha)$–stable and also L–stable. The search was limited to algorithms with $\alpha \geq 10°$. Their performance parameters are tabulated in Table 4.3.

BDF6	7^{th}–order stiffly–stable methods
$\alpha = 19°$	$\alpha \in [10°, 48°]$
$a = -6.0736$	$a \in [-6.1261, -0.9729]$
$c = 0.5107$	$c \in [0.0811, 0.7429]$
$c_{err} = -0.0583$	$c_{err} \in [-6.6498, -0.1409]$
$as.reg. = -0.14$	$as.reg. \in [-0.23, -0.01]$

TABLE 4.3. Properties of stiffly–stable 7^{th}–order algorithms.

The smallest error coefficient is now almost three times larger than in the case of the 6^{th}–order algorithms. The other parameters are comparable in their ranges with those of the 6^{th}–order algorithms.

This time, we used a value of $k = 15$ in Eq.(4.77). The best three algorithms are characterized by the following sets of coefficients:

$$\mathbf{x_{k+1}} = \frac{5148}{12161} h \cdot \mathbf{f_{k+1}} + \frac{552123}{243220}\mathbf{x_k} - \frac{200772}{85127}\mathbf{x_{k-1}} + \frac{184041}{121610}\mathbf{x_{k-2}} - \frac{184041}{425635}\mathbf{x_{k-3}} + \frac{20449}{1702540}\mathbf{x_{k-8}} - \frac{4563}{851270}\mathbf{x_{k-10}} + \frac{99}{121610}\mathbf{x_{k-12}} \tag{4.79a}$$

$$\mathbf{x_{k+1}} = \frac{234}{551} h \cdot \mathbf{f_{k+1}} + \frac{13689}{6061}\mathbf{x_k} - \frac{492804}{212135}\mathbf{x_{k-1}} + \frac{4056}{2755}\mathbf{x_{k-2}} - \frac{4563}{11020}\mathbf{x_{k-3}} + \frac{169}{19285}\mathbf{x_{k-8}} - \frac{507}{121220}\mathbf{x_{k-11}} + \frac{54}{30305}\mathbf{x_{k-12}} \tag{4.79b}$$

$$\mathbf{x_{k+1}} = \frac{3276}{7675} h \cdot \mathbf{f_{k+1}} + \frac{17199}{7675}\mathbf{x_k} - \frac{191646}{84425}\mathbf{x_{k-1}} + \frac{596232}{422125}\mathbf{x_{k-2}} - \frac{74529}{191875}\mathbf{x_{k-3}} + \frac{1183}{191875}\mathbf{x_{k-8}} - \frac{882}{422125}\mathbf{x_{k-12}} + \frac{2106}{2110625}\mathbf{x_{k-13}} \tag{4.79c}$$

Their performance parameters are tabulated in Table 4.4.

BDF6	SS7a	SS7b	SS7c
$\alpha = 19°$	$\alpha = 37°$	$\alpha = 39°$	$\alpha = 35°$
$a = -6.0736$	$a = -3.0594$	$a = -2.9517$	$a = -3.2146$
$c = 0.5107$	$c = 0.6352$	$c = 0.6664$	$c = 0.6331$
$c_{err} = -0.0583$	$c_{err} = -0.3243$	$c_{err} = -0.3549$	$c_{err} = -0.3136$
$as.reg. = -0.14$	$as.reg. = -0.15$	$as.reg. = -0.16$	$as.reg. = -0.15$

TABLE 4.4. Properties of stiffly–stable 7^{th}–order algorithms.

The error coefficients of these methods have grown quite a bit, but luckily, the asymptotic regions haven't shrunk yet significantly.

The stability domains of the three methods are presented in Fig.4.10, where they are also compared to that of BDF6. The damping plots are shown in Fig.4.11.

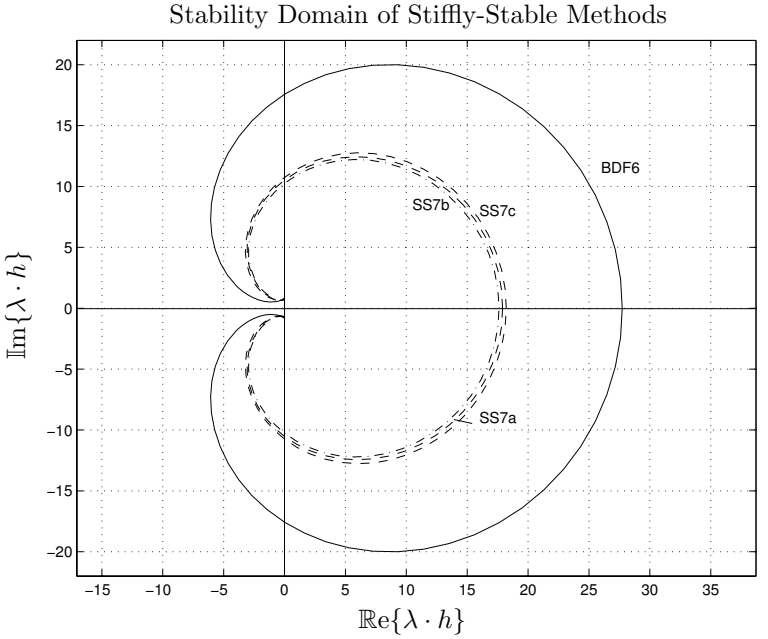

FIGURE 4.10. Stability domains of BDF6 and a set of 7^{th}–order stiffly–stable algorithms.

The three algorithms are very similar indeed in almost every respect, and they should also perform quite similarly in simulations.

We can now proceed to algorithms of 8^{th} order. We searched for algorithms with tails reaching all the way back to t_{k-15}. Hence we had to choose 8 elements out of a list of 16 candidates. Of the possible 12870 algorithms, 493 exhibit properties similar to those of BDF6.

Their performance parameters are tabulated in Table 4.5.

BDF6	8^{th}–order stiffly–stable methods
$\alpha = 19°$	$\alpha \in [10°, 48°]$
$a = -6.0736$	$a \in [-5.3881, -1.4382]$
$c = 0.5107$	$c \in [0.0859, 0.6485]$
$c_{err} = -0.0583$	$c_{err} \in [-6.4014, -0.4416]$
$as.reg. = -0.14$	$as.reg. \in [-0.16, -0.01]$

TABLE 4.5. Properties of stiffly–stable 8^{th}–order algorithms.

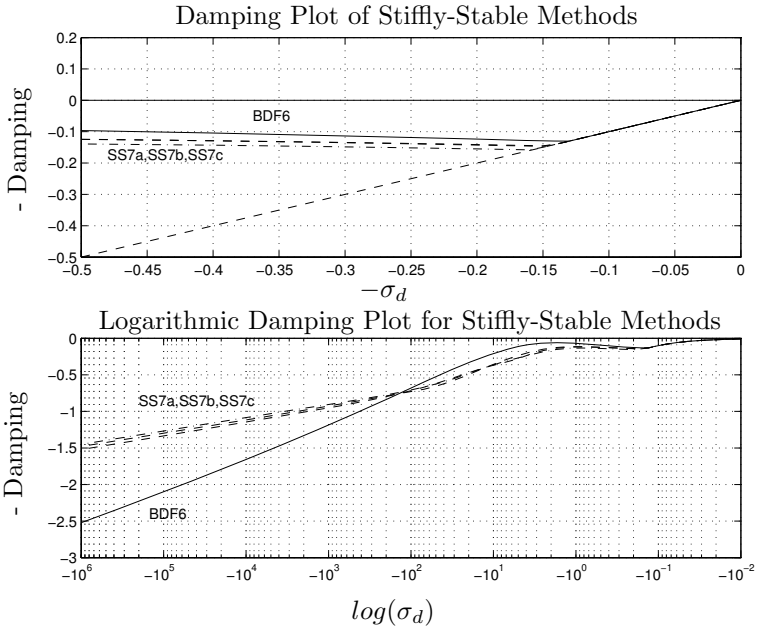

FIGURE 4.11. Damping plots of BDF6 and a set of 7^{th}–order stiffly–stable algorithms.

The smallest error coefficient has unfortunately again grown by about a factor of three, and this time, also the largest asymptotic region has begun to shrink.

This time around, we used a factor of $k = 10$ in Eq.(4.77). Two among the best of these algorithms are characterized by the following sets of coefficients:

$$\begin{aligned}\mathbf{x_{k+1}} =& \frac{112}{267}h\cdot\mathbf{f_{k+1}} + \frac{71680}{31239}\mathbf{x_k} - \frac{2800}{1157}\mathbf{x_{k-1}} + \frac{179200}{114543}\mathbf{x_{k-2}} \\ &- \frac{3920}{8811}\mathbf{x_{k-3}} + \frac{112}{12015}\mathbf{x_{k-9}} - \frac{160}{12727}\mathbf{x_{k-13}} + \frac{7168}{572715}\mathbf{x_{k-14}} \\ &- \frac{35}{10413}\mathbf{x_{k-15}} \end{aligned} \tag{4.80a}$$

$$\begin{aligned}\mathbf{x_{k+1}} =& \frac{208}{497}h\cdot\mathbf{f_{k+1}} + \frac{216320}{93933}\mathbf{x_k} - \frac{93600}{38269}\mathbf{x_{k-1}} + \frac{16640}{10437}\mathbf{x_{k-2}} \\ &- \frac{67600}{147609}\mathbf{x_{k-3}} + \frac{5408}{469665}\mathbf{x_{k-9}} - \frac{1280}{147609}\mathbf{x_{k-12}} + \frac{3328}{574035}\mathbf{x_{k-14}} \\ &- \frac{65}{31311}\mathbf{x_{k-15}} \end{aligned} \tag{4.80b}$$

Their performance parameters are tabulated in Table 4.6.

Their stability domains and damping plots look almost the same as for

BDF6	SS8a	SS8b
$\alpha = 19°$	$\alpha = 35°$	$\alpha = 35°$
$a = -6.0736$	$a = -3.2816$	$a = -3.4068$
$c = 0.5107$	$c = 0.5779$	$c = 0.5456$
$c_{err} = -0.0583$	$c_{err} = -0.9322$	$c_{err} = -0.8636$
$as.reg. = -0.14$	$as.reg. = -0.14$	$as.reg. = -0.13$

TABLE 4.6. Properties of stiffly–stable 8^{th}–order algorithms.

the 7^{th}–order algorithms. We therefore refrained from printing these plots.

Let us now proceed to 9^{th}–order algorithms. We decided to search for algorithms with their tails reaching back as far as t_{k-17}. Hence we had to choose 9 elements from a list of 18. Of the 48620 candidate algorithms, only 152 exhibit properties similar to those of BDF6.

Their performance parameters are tabulated in Table 4.7.

BDF6	9^{th}–order stiffly–stable methods
$\alpha = 19°$	$\alpha \in [10°, 32°]$
$a = -6.0736$	$a \in [-5.0540, -2.4730]$
$c = 0.5107$	$c \in [0.0625, 0.4991]$
$c_{err} = -0.0583$	$c_{err} \in [-5.9825, -1.2492]$
$as.reg. = -0.14$	$as.reg. \in [-0.10, -0.02]$

TABLE 4.7. Properties of stiffly–stable 9^{th}–order algorithms.

The smallest error coefficient has once again grown by about a factor of three, and also the largest asymptotic region has now shrunk significantly.

This time, we used a factor of $k = 5$ in Eq.(4.77). Two among the best of these algorithms are characterized by the following sets of coefficients:

$$\begin{aligned}\mathbf{x_{k+1}} =& \frac{4080}{9947} h \cdot \mathbf{f_{k+1}} + \frac{165240}{69629}\mathbf{x_k} - \frac{16854480}{6336239}\mathbf{x_{k-1}} + \frac{1664640}{905177}\mathbf{x_{k-2}} \\ & - \frac{5618160}{9956947}\mathbf{x_{k-3}} + \frac{23120}{1462209}\mathbf{x_{k-8}} - \frac{332928}{9956947}\mathbf{x_{k-14}} + \frac{351135}{6336239}\mathbf{x_{k-15}} \\ & - \frac{29160}{905177}\mathbf{x_{k-16}} + \frac{1360}{208887}\mathbf{x_{k-17}} \qquad (4.81a)\end{aligned}$$

$$\begin{aligned}\mathbf{x_{k+1}} =& \frac{1904}{4651} h \cdot \mathbf{f_{k+1}} + \frac{719712}{302315}\mathbf{x_k} - \frac{62424}{23255}\mathbf{x_{k-1}} + \frac{6214656}{3325465}\mathbf{x_{k-2}} \\ & - \frac{873936}{1511575}\mathbf{x_{k-3}} + \frac{18496}{1046475}\mathbf{x_{k-8}} - \frac{249696}{16627325}\mathbf{x_{k-13}} + \frac{7803}{302315}\mathbf{x_{k-15}} \\ & - \frac{6048}{302315}\mathbf{x_{k-16}} + \frac{952}{209295}\mathbf{x_{k-17}} \qquad (4.81b)\end{aligned}$$

Their performance parameters are tabulated in Table 4.8.

BDF6	SS9a	SS9b
$\alpha = 19°$	$\alpha = 18°$	$\alpha = 18°$
$a = -6.0736$	$a = -4.3280$	$a = -4.3321$
$c = 0.5107$	$c = 0.3957$	$c = 0.3447$
$c_{err} = -0.0583$	$c_{err} = -1.7930$	$c_{err} = -1.6702$
$as.reg. = -0.14$	$as.reg. = -0.10$	$as.reg. = -0.08$

TABLE 4.8. Properties of stiffly–stable 9^{th}–order algorithms.

Their stability domains and damping plots look almost the same as for the 7^{th} and 8^{th}–order algorithms.

In this section, a number of new algorithms have been developed and presented that extend the concept of BDF algorithms to higher orders.

4.11 Newton Iteration

As we have seen, many of the truly interesting multi–step algorithms are implicit. Let us look once more at BDF3.

$$\mathbf{x_{k+1}} = \frac{6}{11} h \cdot \mathbf{f_{k+1}} + \frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}} \tag{4.82}$$

Plugging in the α–vector and the β–matrix of the BDF3 method, we find:

$$\mathbf{x_{k+1}} = \alpha_3 \; h \cdot \mathbf{f_{k+1}} + \beta_{31} \; \mathbf{x_k} + \beta_{32} \; \mathbf{x_{k-1}} + \beta_{33} \; \mathbf{x_{k-2}} \tag{4.83}$$

Thus, the i^{th}–order BDFi algorithm can be written as:

$$\mathbf{x}_{k+1} = \alpha_i \; h \cdot \mathbf{f}_{k+1} + \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1} \tag{4.84}$$

Plugging in the linear homogeneous problem and solving for $\mathbf{x_{k+1}}$, we find:

$$\mathbf{x_{k+1}} = -\left[\alpha_i \cdot (\mathbf{A} \cdot h) - \mathbf{I^{(n)}}\right]^{-1} \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1} \tag{4.85}$$

On a nonlinear problem, we cannot apply matrix inversion. We can rewrite Eq.(4.84) as:

$$\mathcal{F}(\mathbf{x}_{k+1}) = \alpha_i \; h \cdot \mathbf{f}(\mathbf{x}_{k+1}, t_{k+1}) - \mathbf{x}_{k+1} + \sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1} = 0.0 \tag{4.86}$$

and use Newton iteration on Eq.(4.86):

$$\mathbf{x}_{k+1}^{\ell+1} = \mathbf{x}_{k+1}^{\ell} - [\mathcal{H}^{\ell}]^{-1} \cdot [\mathcal{F}^{\ell}] \tag{4.87}$$

where the Hessian $\mathcal{H}$ can be computed as:

$$\mathcal{H} = \alpha_i \cdot (\mathcal{J} \cdot h) - \mathbf{I^{(n)}} \tag{4.88}$$

and $\mathcal{J}$ is the Jacobian of the system. By plugging the linear homogeneous system into Eq.(4.87), it is easy to show that we get Eq.(4.85) back, i.e., Newton iteration doesn't change the stability domain of the method.

Most of the professional multi–step codes use modified Newton iteration, i.e., they do not reevaluate the Jacobian during the iteration. They usually don't evaluate the Jacobian even once every step. Instead, they use the error estimate of the method as an indicator when the Jacobian needs to be reevaluated. As long as the error estimate remains approximately constant, the Jacobian is still acceptable. However, as soon as the absolute gradient of the error estimate starts to grow, indicating the need for a change in step size, this is a clear indication that a new Jacobian computation is in order, and only if a reevaluation of the Jacobian doesn't get the error estimate back to where it was before, will the step size of the method be adjusted.

Even the Hessian is not reevaluated frequently. The Hessian needs to be recomputed either if the Jacobian has been reevaluated, or if the step size has just been modified.

Most professional codes offer several options for how to treat the Jacobian. The user can choose between (i) providing an analytical expression for the Jacobian, (ii) having the full Jacobian evaluated by means of a numerical approximation, and (iii) having only the diagonal elements of the Jacobian evaluated by means of a numerical approximation ignoring the off–diagonal elements altogether.

Both the convergence speed and the convergence range of the Newton iteration scheme are strongly influenced by the quality of the Jacobian. A diagonal approximation is cheap, but leads to a heavy increase in the number of iterations necessary for the algorithm to converge, and necessitates more frequent Jacobian evaluations as well. In our experience, it hardly ever pays off to consider this option.

The full Jacobian is usually determined by first–order approximations. The i^{th} state variable, x_i, is modified by a small value, Δx_i. The state derivative vector is then reevaluated using the modified state value. We find:

$$\frac{\partial \mathbf{f}(\mathbf{x}, t)}{\partial x_i} \approx \frac{\mathbf{f_{pert}} - \mathbf{f}}{\Delta x_i} \tag{4.89}$$

where $\mathbf{f}$ is the state derivative vector evaluated at the current nominal values of all state variables, whereas $\mathbf{f_{pert}}$ is the perturbed state derivative vector evaluated at $x_i + \Delta x_i$ with all other state variables being kept at their currently nominal values. Thus, an n^{th}–order system calls for n additional function evaluations in order to obtain one full approximation of the Jacobian.

Yet, even by spending these additional n function evaluations, we gain only a first–order approximation of the Jacobian. Any linear model can be converged in precisely one step of Newton iteration with the correct Jacobian being used, irrespective of the location of its eigenvalues or the size of the system. Using the first–order approximation, however, we may need three to four iterations in order to get the *iteration error* down to a value below the *integration error*. On a sufficiently nasty nonlinear problem, the ratio of the numbers of iterations needed to converge may be even worse.

For these reasons, we strongly advocate the analytical option. In the past, this option has rarely been used ... because we engineers are a lazy lot. An n^{th}–order model calls for n^2 additional equations in order to analytically describe its Jacobian. Moreover, these equations may be longer and more involved than the original n equations due to the analytical differentiation. Thus, deriving the Jacobian equations by hand is a tedious and error–prone process.

However, there is really no good reason why these equations should have to be derived by hand. As you already know if you read the companion book to this text *Continuous System Modeling* [4.2], engineers anyway don't usually write down their models by hand in a form that the numerical integration software could use directly. They employ a *modeling language*, such as Dymola [4.3], from which, by means of compilation, a simulation program is generated.

There is no good reason why the analytical Jacobian equations could not be generated automatically in this process, i.e., the Jacobian can be generated once and for all at compile time by means of symbolic differentiation. Indeed, Dymola [4.3] already offers such a feature. Symbolic differentiation is very useful also for other purposes that we shall talk about in due course [4.1].

We are fully convinced that mixed symbolic/numerical algorithms are the way of the future. Many problems can be tackled either numerically or symbolically. However in some cases, the numerical solution is more efficient, whereas in others, the symbolic approach is clearly superior. By merging these two approaches into one integrated software environment, we can preserve the best of both worlds. In continuous system modeling and simulation, models specified by the user in a form most convenient to him or her are symbolically preconditioned for optimal use by the subsequent numerical algorithms, such as the numerical integration software.

Even in this chapter, we have already made use of symbolic algorithms without explicitly mentioning it. We explained in Eqs.(4.60) and (4.61), how the coefficients of high–order stiffly–stable integration algorithms can be found. Yet, if this is done numerically in MATLAB, e.g. using the statement:

$$\mathbf{coef} = \text{sum}(\text{inv}(\mathbf{M})); \tag{4.90}$$

the resulting coefficient vector will be generated in a real–valued format, rather than as rational expressions. Numerical mathematicians prefer to be provided with rational expressions for these coefficients, so that the mantissa of the machine on which the integration algorithm is to be implemented can be fully exploited without leading to additional and unnecessary roundoff errors.

We could have tried to obtain rational expressions making use of the fact that both the determinant and the adjugate of an integer–valued matrix are integer–valued, i.e.:

$$\begin{aligned} Mdet &= \text{round}(\det(\mathbf{M})); \\ \mathbf{Madj} &= \text{round}(Mdet * \text{inv}(\mathbf{M})); \\ \mathbf{coef\_num} &= \text{sum}(\mathbf{Madj}); \end{aligned} \tag{4.91}$$

In this way, the numerators of the coefficients, **coef_num**, can be computed as an integer–valued vector, whereas the common denominator is the equally integer–valued determinant. We could then use any one among a number of well-known algorithms to determine the common dividers between the numerators and denominators of each coefficient.

This approach works well for BDF algorithms of orders three or four, but fails, when dealing with 8^{th}– or 9^{th}–order algorithms. The reason is that the determinant of $\mathbf{M}$ grows so rapidly with the size of $\mathbf{M}$ that the mantissa of a 32–bit machine is exhausted quickly, in spite of the fact that MATLAB computes everything in double precision.

For this reason, we generated the coefficient vectors of e.g. Eqs.(4.81) by means of the MATLAB statement:

$$\mathbf{coef} = \text{sum}(\text{inv}(\text{sym}(\mathbf{M}))); \tag{4.92}$$

i.e., making use of MATLAB's *symbolic toolbox*. The symbolic toolbox represents integers as character strings, and is thereby not limited by the length of the mantissa. The *sym*–operator converts the numeric integer–valued matrix $\mathbf{M}$ into a symbolic representation. The *inv*– and *sum*–functions are overloaded MATLAB functions that make use of different algorithms depending on the type declaration of their operand.

Using similar techniques, Gander and Gruntz [4.4] recently corrected a number of errors in well-known and frequently used numerical algorithms that had gone unnoticed for several decades.

4.12 Step–size and Order Control

We have seen in the previous chapter that the appropriate order of an RK algorithm is determined by the accuracy requirements. Therefore, since the

relative accuracy requirements usually are the same throughout the entire simulation run, order control makes little sense.

Let us ascertain whether the same is true in the case of the multi–step algorithms. To this end, we shall simulate our fifth–order linear test problem of Eq.(H4.8a) across 10 seconds, using zero input and applying an initial value of 1.0 to all five state variables. For different global relative error requirements, we find the largest step sizes that keep the numerical error just below the required error bounds, and plot the number of function evaluations needed to simulate the system across 10 seconds as a function of the required accuracy. The process is repeated for AB2, AB3, and AB4. The same quantity for RK4 is plotted on the same graph for comparison. The results of this analysis are shown on Fig.4.12.

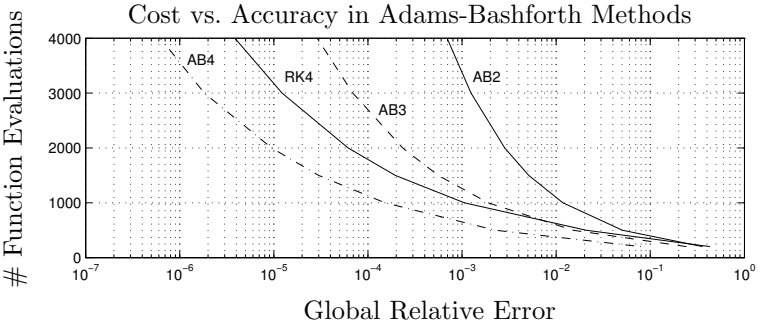

FIGURE 4.12. Cost versus accuracy for different ABi algorithms.

The results are a little deceiving, since only the number of function evaluations (one per step for all ABi algorithms) is plotted, not taking into account the higher cost associated with data management within the higher–order ABi algorithms.

If we decide that we are willing to spend about 500 function evaluations on this simulation, we can get a global relative accuracy of about 10% with AB2, we can get about 1% global accuracy with AB3, and we can get about 0.1% accuracy with AB4. Thus, just as in the case of the RKi algorithms, the accuracy requirements determine the minimum order of the algorithm that is economical to employ. Since the order of the algorithm is dictated by the (constant) accuracy requirements specified by the user, order control doesn't make too much sense.

As a little caveat: AB4 is about 25% cheaper than RK4 in this example. Notice that this is a linear time–invariant non–stiff problem, i.e., a problem where the ABi algorithms perform at their best. Although RK4 takes four function evaluations per step, whereas AB4 takes only one function evaluation per step, we never gain a factor of four in efficiency, since the asymptotic regions of the RKi algorithms are considerably larger than those of the ABi algorithms, forcing us to use a smaller step size in the

latter case. The situation gets worse with higher orders of approximation accuracy due to the detrimental influence of spurious eigenvalues.

Why is order control fashionable in multi–step algorithms? The answer is simple: because order control is *cheap*. Remember how multi–step algorithms work. At all times, we keep a record of back storage values of states and/or state derivatives. When we proceed from time t_k to time t_{k+1}, we simply throw the oldest values (the rear end of the tail) away, shift all the vectors by one into the past, and add the newest state information to the front end of the queue. If we decide to increase the order by one, we simply don't throw away anything. On the other hand, if the decide to decrease the order by one, we simply throw away the two oldest values. Thus, order control is trivial.

Step–size control is *not* cheap. If we change the step size at any time, we are faced with non–equidistantly spaced storage values, and although we could redesign our multi–step methods to work with non–equidistant spacing also (this has been done on some occasions), it is usually too expensive to do so. There are better ways to do step–size control, as we shall see.

Since step–size control is expensive and order control is cheap, why not use order control instead? If we are currently computing too accurately and wish to increase the step size, why can't we drop the order instead and keep using the same step size? The answer is that order control is very coarse. Dropping the order by one usually reduces the accuracy by about a factor of 10. This can be easily seen on Fig.4.12. Thus, we must be computing *much* too accurately, before dropping the order is justified. Moreover, we don't even save that much by doing so. After all, the number of function evaluations remains the same. The fact that everybody does order control, doesn't mean, it's the smart thing to do (!)

How then can we do step–size control efficiently? The trick is actually quite simple. Let us reconsider the Newton–Gregory backward polynomials. We start out with:

$$\mathbf{x}(t) = \mathbf{x_k} + s\nabla\mathbf{x}_k + \left(\frac{s^2}{2} + \frac{s}{2}\right)\nabla^2\mathbf{x}_k + \left(\frac{s^3}{6} + \frac{s^2}{2} + \frac{s}{3}\right)\nabla^3\mathbf{x}_k + \dots \quad (4.93)$$

Differentiation with respect to time yields:

$$\dot{\mathbf{x}}(t) = \frac{1}{h}\left[\nabla\mathbf{x}_k + \left(s + \frac{1}{2}\right)\nabla^2\mathbf{x}_k + \left(\frac{s^2}{2} + s + \frac{1}{3}\right)\nabla^3\mathbf{x}_k + \dots\right] \quad (4.94)$$

The second derivative becomes:

$$\ddot{\mathbf{x}}(t) = \frac{1}{h^2}\left[\nabla^2\mathbf{x}_k + (s+1)\nabla^3\mathbf{x}_k + \dots\right] \quad (4.95)$$

etc.

Truncating Eqs.(4.93)–(4.95) after the cubic term, expanding the ∇–operators, and evaluating for $t = t_k \quad (s = 0.0)$, we obtain:

$$\begin{pmatrix} x_k \\ h \cdot \dot{x}_k \\ \frac{h^2}{2} \cdot \ddot{x}_k \\ \frac{h^3}{6} \cdot x_k^{(\mathrm{iii})} \end{pmatrix} = \frac{1}{6} \cdot \begin{pmatrix} 6 & 0 & 0 & 0 \\ 11 & -18 & 9 & -2 \\ 6 & -15 & 12 & -3 \\ 1 & -3 & 3 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_k \\ x_{k-1} \\ x_{k-2} \\ x_{k-3} \end{pmatrix} \tag{4.96}$$

The vector to the left of the equal sign is called the *Nordsieck vector*, here written of third order. Using this trick, it has become possible to translate state information stored at different points in time by means of a simple multiplication with a constant matrix into state– and state derivative information stored at one time point only. The transformation was written here for a single state variable. The vector case works in exactly the same fashion, but the notation is less convenient.

This discovery allows us to solve the step–size control problem. If we wish to change the step size at time t_k, we simply multiply the vector containing the past state values with the transformation matrix, thereby transforming the state history vector to an equivalent Nordsieck vector. In this new form, the step size can be modified easily, e.g. by multiplying the Nordsieck vector from the left with the diagonal matrix:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{h_{\mathrm{new}}}{h_{\mathrm{old}}} & 0 & 0 \\ 0 & 0 & \left(\frac{h_{\mathrm{new}}}{h_{\mathrm{old}}}\right)^2 & 0 \\ 0 & 0 & 0 & \left(\frac{h_{\mathrm{new}}}{h_{\mathrm{old}}}\right)^3 \end{pmatrix} \tag{4.97}$$

We now have the Nordsieck vector expressed in the modified time step, h_{new}. We then multiply this modified Nordsieck vector from the left with the inverse transformation matrix. This operation results in a modified state history vector, where the "stored" $\mathbf{x}$ values represent the solution at the modified "sampling points."

This is today the preferred method for step–size control in multi–step integration algorithms. Step–size control is still fairly expensive. In the case of implicit algorithms, we need to also evaluate a new Hessian after modifying the step size using the above matrix multiplications. Since we are already in a "spending mood," we might just as well use the opportunity to get a new Jacobian also.

For this reason, the Gustafsson algorithm [4.8] is not practical for use in multi–step integration. We don't want to change the step size after every step. If we need to reduce the step size, we shall reduce it at once to at least one half of its former value to prevent the danger of having to reduce the step size immediately again. We don't want to increase the step size either

until we are fairly certain that we can at least double it without violating the accuracy constraints. How do we know that? The next section will tell.

4.13 The Startup Problem

One problem we have not discussed yet is how the integration process is started in the first place. Usually, the initial state vector is given at time t_0, but no back values are available. How can we compute estimates for these values such that the multi–step formulae become applicable?

Traditionally, applied mathematicians have chosen the easy route: applying order control. They employ a first–order method during the first integration step. This provides them with a second data point at time $t_1 = t + h$. Since, by now, they have two data points, they can raise the order by one, and perform the second integration step using a second–order formula, etc. After a suitable number of steps, the algorithm has acquired the desired state history vector in order to make an n^{th}–order multi–step method applicable.

This method "works," in the sense that it can be programmed into an algorithm. However, it is not acceptable on any other grounds. The problem is accuracy. In order to satisfy our accuracy requirements, we must use a very small step size during the first low–order steps. Even after we have built up the order, we should not immediately increase the step size by use of the Nordsieck transformation, since some of the back values currently in the storage area of the algorithm are low–order accurate. In the transformation, we may pick up bad numerical errors. It is better to wait for at least another n steps, before we even dream of changing the step size to a more decent value. This is utterly wasteful.

Bill Gear has shown another way [4.6]. We can use Runge–Kutta algorithms for startup purposes. In this way, also the early steps are of the correct order, and a more decent step size can be used from the beginning.

Let us explain our own version of this general idea. If we decide that we want to employ an i^{th}–order algorithm, we start out performing $(i-1)$ steps of RKi using a fixed–step algorithm. The step size doesn't matter too much as long as it is chosen sufficiently small to ensure that the accuracy requirements are met. For example, we can use step–size control during the first step to determine the right step size, and then disable the step–size control algorithm.

By now, we have i equidistantly–spaced storage values of the state vector, and we are able to start using the multi–step algorithm. However, what step size should we use? In order to determine the answer to this question, let us look once more at Fig.4.12. This time, we plotted the same curves as before using a double–logarithmic scale.

We notice that the logarithm of the step size is, for all practical purposes,

linear in the logarithm of the accuracy. Thus, we can perform one step of the multi–step technique using the step size h_1 from the RK starter, and obtain an error estimate ε_1. We then reduce the step size to $h_2 = h_1/2$, and repeat the step. We obtain a new error estimate ε_2.

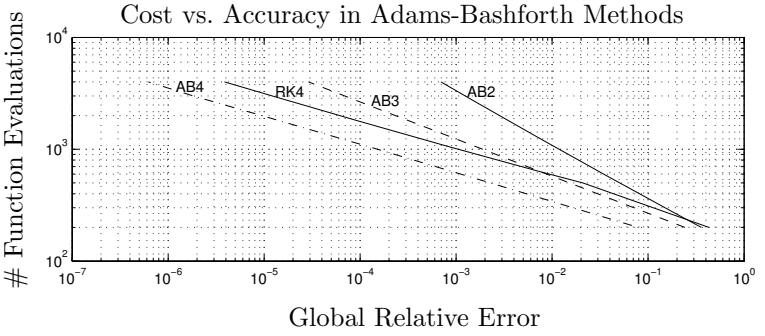

FIGURE 4.13. Cost versus accuracy for different ABi algorithms.

We now place a linear curve through the two points, and interpolate (extrapolate) with the desired accuracy ε_{des} to obtain a good value for the true step size to be used by the multi–step algorithm:

$$\begin{pmatrix} \ln(h_1) \\ \ln(h_2) \end{pmatrix} = \begin{pmatrix} \ln(\varepsilon_1) & 1 \\ \ln(\varepsilon_2) & 1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \tag{4.98a}$$

$$\ln(h_{\text{des}}) = a_1 \cdot \ln(\varepsilon_{\text{des}}) + a_2 \tag{4.98b}$$

$$h_{\text{new}} = 0.8 \cdot h_{\text{des}} \tag{4.98c}$$

Eq.(4.98a) solves a linear set of two equations for the unknown parameters a_1 and a_2, Eq.(4.98b) solves the interpolation (extrapolation) problem, and Eq.(4.98c) computes the new step size using a safety margin of 20%.

Runge–Kutta starters work very well in the case of non–stiff problems, i.e., as start–up algorithms for Adams–Bashforth or Adams–Bashforth–Moulton algorithms. They are more problematic in the case of the Backward Difference Formulae. The reason for this observation is simple: The RK start–up algorithm may need to use exceedingly small steps because of the stiffness of the problem to be solved. Once we switch over to the BDF algorithm, we may thus wish to increase the step size dramatically. The Nordsieck approach allows us to do so, but in the process, the sampling points get extended over a wide range, which corresponds to heavy extrapolation, a process that is invariably associated with sources of inaccuracy. We may thus prefer to limit the allowed step size enlargement to maybe a factor of 10, then perform n steps of BDF with that intermediate step size to gain a new accurate history vector, before increasing the step size once more by a factor of 10, and repeat this process, until the appropriate step

size has been reached.

In step–size control, we can use the same algorithm. We don't want to change the step size unless it needs to be changed by a large value. Therefore, if we have decided that a step–size change is truly in order, we can afford to calculate one additional test step with half the former step size (or double the former step size) in order to obtain a decent estimate of where the step size ought to be.

A yet better approach might have been to use a number of $\mathrm{BI4}/5_{0.45}$ steps during startup to avoid both the numerical stability problems haunting the RK starters and the numerical accuracy problems associated with the low–order BDF starters.

4.14 The Readout Problem

The last problem to be discussed in this chapter is the readout problem. If we simulate a system, we want to obtain results, i.e., we wish to obtain the values of one or several output variables at pre–specified points in time, the so–called *communication points*.

Often, the communication points are equidistantly spaced. In this case, the distance between neighboring communication points is referred to as the *communication interval*.

In single–step integration, we simply reduce the step size when approaching the next communication point in order to hit the communication point accurately. In multi–step integration, this approach is too expensive. We cannot afford to modify the step size for no other purpose than to deliver some output values.

The solution is simple. We integrate past the communication point using the actual step size. We then interpolate back to calculate an estimation of the state vector at the communication point. In multi–step integration, interpolation with the same order of approximation accuracy that the currently employed integration method uses is cheap. All we need to do is to convert the state history vector at the end of the integration step to the Nordsieck form, then correct the step size such that the end of the "previous step" coincides with the communication point, then record the new "immediate past value" as the readout value.

After the results have been recorded, the algorithm returns to the end of the integration step, and proceeds as if no interruption had taken place.

4.15 Summary

20 years ago, the chase for new integration algorithms was still on. In numerical integration workshops around the globe, new integration meth-

ods were presented by the dozen. Hardly any of them survived the test of time. The reason for this surprising fact is simple. To come up with a new algorithm is the easy part. To incorporate that algorithm in a robust general–purpose production code is an entirely different matter.

Most engineering users of simulation software use the numerical integration software as a black box. They don't have the foggiest idea of how the code works, and frankly, they couldn't care less. All they are interested in is that the code *reliably* and *efficiently* generates accurate estimates of the output variables at the communication points.

In a mature production code of a multi–step integration algorithm, the actual algorithm occupies certainly less than 5% of the code. All the rest is boiler plate: code for initializing the coefficient matrices; code for starting up the integration algorithm, e.g., using an RK starter; code for update, maintenance, and disposal of the state history information; code for interpolating the results at communication points; code for step–size and (alas!) order control; and finally, code for handling of discontinuities — a topic to be discussed in a separate chapter of this book.

Software engineers may be inclined to believe that the answer to this problem is software modularization. Let us structure the software in such a way that e.g. the step–size control is handled by one routine, interpolation is handled by another, etc., in such a fashion that these routines can be modularly plugged together. Unfortunately, even this doesn't work. A step–size control algorithm that is efficient for an RK algorithm, such as the Gustafsson method, could theoretically also be used for a multi–step algorithm, but it would be terribly inefficient.

What *has* happened is a certain standardization of the interfaces of numerical integration software (the parameter calling sequences), such that a user can fairly easily replace one entire code by another to check which one works better. In this context, it is important to mention the efforts of Alan Hindmarsh whose various LSODE codes are clearly among the survivors.

More could certainly be done. Today's multi–step codes are unnecessarily unreadable. What we would need is an efficient *MATLAB compiler* that would allow us to develop new production codes in MATLAB, making them easily readable, and once they are fully debugged, generate automatically efficient C–code for use as stand–alone programs. Availability of such a software design tool would make the life of applied mathematicians much easier. Although a C–compiler for MATLAB has been developed, the generated code is unfortunately anything but efficient, since it makes use of the generic data management tools of MATLAB. Consequently, C–compiled MATLAB code doesn't execute much faster than the interpreted MATLAB code itself.

You, the reader, may have noticed that we are somewhat critical vis–à–vis the multi–step integration methods. Runge–Kutta methods are, in our opinion, considerably more robust, and it is easier to design production codes for them. Multi–step techniques are fashionable because the algo-

rithms themselves are so much easier to design, but the price to be paid is dear. Mature multi–step codes are very difficult to design, and even with the best of all such codes, it still happens that it breaks down in the face of a nasty nonlinear simulation problem, and if it does, it may be very difficult for even knowledgeable users to determine what precisely it was that the algorithm didn't like, and which parameter to fiddle around with in order to get around the problem.

Single–step codes are much simpler to develop and maintain, and they offer a smaller number of tuning parameters that the user might need to worry about. They are considerably more robust. Whereas non–stiff RK codes are available and are being widely used, stiff implicit RK production codes are slow in coming. BDF codes are still far more frequently used in practice than IRK codes. However, this is true not because of the superiority of these algorithms, but due to the wider distribution of good production codes.

4.16 References

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

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

[4.3] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[4.4] Walter Gander and Dominik Gruntz. Derivation of Numerical Methods Using Computer Algebra. *SIAM Review*, 41(3):577–593, 1999.

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

[4.6] C. William Gear. Runge–Kutta Starters for Multistep Methods. *ACM Trans. Math. Software*, 6(3):263–279, 1980.

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

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

[4.9] Klaus Hermann. Solution of Stiff Systems Described by Ordinary Differential Equations Using Regression Backward Difference Formulae. Master's thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, Ariz., 1995.

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

[4.11] William E. Milne. *Numerical Solution of Differential Equations*. John Wiley, New York, 1953. 275p.

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

4.17 Homework Problems

[H4.1] The Differentiation Operator

Rewrite Eq.(4.20) and Eq.(4.21) in terms of the ∇–operator. Develop also a formula for $\mathcal{D}^3$.

[H4.2] Nyström–Milne Predictor–Corrector Techniques

Follow the reasoning of the Adams–Bashforth–Moulton predictor–corrector techniques, and develop similar pairs of algorithms using a Nyström predictor stage followed by a Milne corrector stage.

Plot the stability domains for NyMi3 and NyMi4. What do you conclude?

[H4.3] New Methods

Using the Gregory–Newton backward polynomial approach, design a set of algorithms of the type:

$$\mathbf{x_{k+1}} = \mathbf{x_{k-2}} + \frac{h}{\alpha_i} \cdot \left[\sum_{j=1}^{i} \beta_{ij} \; \mathbf{x}_{k-j+1} \right] \tag{H4.3a}$$

Plot their stability domains. Compare them with those of the Adams–Bashforth techniques and those of the Nyström techniques. What do you conclude?

[H4.4] Milne Integration

Usually, the term "Milne integration algorithm," when used in the literature, denotes a specific predictor–corrector technique, namely:

$$\begin{aligned}
\text{predictor:}\quad & \dot{\mathbf{x}}_{\mathbf{k}} = \mathbf{f}(\mathbf{x}_{\mathbf{k}}, t_k) \\
& \mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k-3}} + \tfrac{h}{3}(8\dot{\mathbf{x}}_{\mathbf{k}} - 4\dot{\mathbf{x}}_{\mathbf{k-1}} + 8\dot{\mathbf{x}}_{\mathbf{k-2}}) \\
\text{corrector:}\quad & \dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} = \mathbf{f}(\mathbf{x}^{\mathbf{P}}_{\mathbf{k+1}}, t_{k+1}) \\
& \mathbf{x}^{\mathbf{C}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k-1}} + \tfrac{h}{3}(\dot{\mathbf{x}}^{\mathbf{P}}_{\mathbf{k+1}} + 4\dot{\mathbf{x}}_{\mathbf{k}} + \dot{\mathbf{x}}_{\mathbf{k-1}})
\end{aligned}$$

The corrector is clearly Simpson's rule. However, the predictor is something new that we haven't seen yet.

Derive the order of approximation accuracy of the predictor. To this end, use the Newton–Gregory backward polynomial in order to derive a set of formulae with a distance of four steps apart between their two state values.

Plot the stability domain of the predictor–corrector method, and compare it with that of NyMi4. What do you conclude? Why did William E. Milne [4.11] propose to use this particular predictor?

[H4.5] Damping Plots of Adams–Bashforth Techniques

Find the damping plots of AB2, AB3, and AB4 for σ_d in the range [-1.0,0.0]. Compare them with the corresponding damping plots of RK2, RK3, and RK4. What do you conclude about the size of the asymptotic regions of these algorithms?

[H4.6] Damping Plots of Adams–Moulton Techniques

Find the damping plots of AM2, AM3, and AM4 for σ_d in the range $[-1.0, 0.0]$. Compare them with those of AB2, AB3, and AB4. What can you say about the comparative sizes of the asymptotic regions of these algorithms?

[H4.7] Damping Plots of Backward Difference Formulae

Find the damping plots of BDF2, BDF3, and BDF4 for σ_d logarithmically spaced between 10^{-1} and 10^6, and plot them on a logarithmic scale like in Fig.4.7. What do you conclude about the damping properties of these algorithms at large values of σ_d? Do all these methods share the desirable properties of BDF6, or was this a happy accident?

[H4.8] Cost Versus Accuracy

Compute the cost–versus–accuracy plots of AB4, ABM4, and AM4 for the linear non–stiff test problem:

$$\dot{\mathbf{x}} = \begin{pmatrix} 1250 & -25113 & -60050 & -42647 & -23999 \\ 500 & -10068 & -24057 & -17092 & -9613 \\ 250 & -5060 & -12079 & -8586 & -4826 \\ -750 & 15101 & 36086 & 25637 & 14420 \\ 250 & -4963 & -11896 & -8438 & -4756 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 5 \\ 2 \\ 1 \\ -3 \\ 1 \end{pmatrix} \cdot u \tag{H4.8a}$$

with zero input and with the initial condition $\mathbf{x_0} = \text{ones}(5,1)$, and plot them together on one graph. ABM4 consumes always two function evaluations per step. In the case of AM4, the situation is more involved, but for simplicity, we want to assume that AM4 needs, on the average, four function evaluations per step.

Although we plot the number of steps versus the accuracy, it is more efficient to vary the number of steps and check what accuracy we obtain in each case. We suggest that you select a set of values of steps in the range $[200, 4000]$, e.g. $nbrstp = [200, 500, 1000, 2000, 4000]$. You then need to compute the step sizes. In the case of ABi, they would be $h = 10/nbrstp$, since we want to integrate across 10 seconds of simulated time using $nbrstp$ steps in total. In the case of AMi, we would use the formula $h = 40/nbrstp$.

Since the test problem is linear, you can simulate the system using the **F**–matrices. In the case of the AMi algorithms, you can use matrix inversion rather than Newton iteration (we indirectly accounted for the iteration by allowing four function evaluations per step).

Since these methods are not self–starting, you need to start out with $(i-1)$ steps of RKi using the same step sizes. Of course, the RKi steps are also simulated using their respective **F**–matrices.

As a gauge, we need the analytical solution of the test problem. Since the input is constant between sampling points (in fact, it is zero), we can find the *exact* solution by converting the differential equations into a set of equivalent difference equations using MATLAB's $c2d$–function. This generates the analytical **F**–matrix. Theoretically:

$$\mathbf{F} = \exp(\mathbf{A} \cdot h) \tag{H4.8b}$$

but doesn't use MATLAB's *expm*–function. For sufficiently large step sizes, you'll get an overflow error. It would be asked a little too much to explain here why this happens. If you are interested to know more about this numerical problem, we refer you to Cleve Moler's excellently written paper on this subject [4.12].

The local relative error is computed using the formula:

$$\varepsilon_{\text{local}}(t_k) = \frac{\|\mathbf{x}_{\text{anal}}(t_k) - \mathbf{x}_{\text{simul}}(t_k)\|}{\max(\|\mathbf{x}_{\text{anal}}(t_k)\|, \text{eps})} \tag{H4.8c}$$

where *eps* is MATLAB's machine constant.

The global relative error is computed using the formula:

$$\varepsilon_{\text{global}} = \max_k(\varepsilon_{\text{local}}(t_k)) \tag{H4.8d}$$

What do you conclude about the relative efficiency of these three algorithms to solve the test problem?

[H4.9] Cost Versus Accuracy

We wish to repeat the same analysis as before, but this time for the stiff linear test problem:

$$\dot{\mathbf{x}} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -10001 & -10201 & -201 \end{pmatrix} \cdot \mathbf{x} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \cdot u \qquad \text{(H4.9a)}$$

We wish to compute the step response of this system across ten seconds of simulated time.

This time, we are going to use BDF2, BDF3, and BDF4, in order to compare their relative efficiency at solving this stiff test problem. As in the case of the previous homework, we are going to simulate the system using the **F**–matrices. As with the AMi algorithms, we use matrix inversion, and simply assume that each step consumes, on the average, four function evaluations.

As a reference, also compute the cost–versus–accuracy plot of the BI4/$5_{0.45}$ algorithm using RKF4/5 for its semi–steps. For reasons of fairness, we shall assume that the implicit semi–step uses four iterations. Together with the single explicit semi–step, one entire step of BI4/$5_{0.45}$ consumes five semi–steps with six function evaluations each, thus: $h = 300/nbrstp$.

Which technique is more efficient, BDF4 or BI4/5, to solve this stiff test problem?

[H4.10] The Nordsieck Form

Equation (4.96) showed the transformation matrix that converts the state history vector into an equivalent Nordsieck vector. Since, at the time of conversion, we also have the current state derivative information available, it is more common to drop the oldest state information in the state history vector, and replace it by the current state derivative information. Consequently, we are looking for a transformation matrix **T** of the form:

$$\begin{pmatrix} x_k \\ h \cdot \dot{x}_k \\ \frac{h^2}{2} \cdot \ddot{x}_k \\ \frac{h^3}{6} \cdot x_k^{(\text{iii})} \end{pmatrix} = \mathbf{T} \cdot \begin{pmatrix} x_k \\ h \cdot \dot{x}_k \\ x_{k-1} \\ x_{k-2} \end{pmatrix} \qquad \text{(H4.10a)}$$

The matrix **T** can easily be found by manipulating the individual equations of Eq.(4.96).

Find corresponding **T**–matrices of dimensions 3×3 and 5×5.

[H4.11] Backward Difference Formulae

We wish to simulate the stiff test problem of Eq.(H4.9a) once more using BDF4. However this time around, we no longer want to make use of the knowledge that the system is linear.

Implement BDF4 with Newton iteration in MATLAB. Use three steps of RK4 for startup. Use the outlined procedure for step–size control. We wish to record the values of the three state variables once every second. Solve the readout problem using the algorithm outlined in this chapter.

4.18 Projects

[P4.1] Stiffly–Stable Methods

Extend one of the widely used variable–order, variable–step size stiff system solvers to include methods of orders seven, eight, and nine, as developed in this chapter.

Compare the efficiency of the so modified code with that of the original code when solving a stiff system with high accuracy requirements.

[P4.2] Stiffly–Stable Methods

Study the effect of the start–up algorithm on a stiff system solver by comparing an order buildup approach with a single–step start–up approach.

Compare RK starters with BI starters.

[P4.3] Stiffly–Stable Methods

Study the importance of a small error coefficient *vs.* a large asymptotic region on the efficiency of a stiff system solver.

Compare different BDF algorithms of the same order using the same start–up and step–size control strategies against each other. The codes are supposed to differ only in the formulae being used. Choose some formulae with small error coefficients, and compare them with formulae with large asymptotic regions.

Draw cost *vs.* accuracy plots to compare their relative economy when solving identical stiff systems.

4.19 Research

[R4.1] Regression Backward Difference Algorithms

In the development of the stiffly–stable algorithms, we always made use of $n+1$ terms to define an n^{th}–order algorithm. It may be beneficial to allow more terms in the algorithm without increasing its order.

For example, we could allow a 3^{rd}–order accurate BDF algorithm to make use of the term $\mathbf{x_{k-3}}$ as well. In that case, Eq.(4.60) needs to be modified as follows:

$$\begin{pmatrix} h \cdot f_{k+1} \\ x_k \\ x_{k-1} \\ x_{k-2} \\ x_{k-3} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & -2 & 4 & -8 \\ 1 & -3 & 9 & -27 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \tag{R4.1a}$$

The **M**–matrix in this case is no longer square. The above equation can thus only be solved for the unknown parameter vector in a least square sense. We thus call these algorithms *Regression Backward Difference Formulae* [4.9].

We can multiply the equation:

$$\mathbf{z} = \mathbf{M} \cdot \mathbf{a} \tag{R4.1b}$$

from the left with $\mathbf{M}'$:

$$\mathbf{M}' \cdot \mathbf{z} = (\mathbf{M}' \cdot \mathbf{M}) \cdot \mathbf{a} \tag{R4.1c}$$

where $\mathbf{M}' \cdot \mathbf{M}$ is a square matrix of full rank. Hence, we can multiply the equation with its inverse:

$$\mathbf{a} = (\mathbf{M}' \cdot \mathbf{M})^{-1} \mathbf{M}' \cdot \mathbf{z} \tag{R4.1d}$$

where $(\mathbf{M}' \cdot \mathbf{M})^{-1} \mathbf{M}'$ is the *Penrose–Moore pseudoinverse* of the rectangular matrix **M**. It solves the over–determined linear system in a least square sense.

Extend the search for high–order stiffly–stable methods by allowing extra terms in the algorithm. The hope is that the added flexibility may enable us to either reduce the error coefficient or (even better!) enlarge the asymptotic region.