The Numerical Solution of Large Systems of Stiff IVPs for ODEs*

KENNETH R. JACKSON

Computer Science Department,
University of Toronto,
Toronto, Ontario, Canada, M5S 1A4.
krj@cs.toronto.edu.

Original draft July 1995.
Revised September 1995.

Accepted for publication by the
J. Applied Numerical Mathematics.

Abstract

The application of the method of lines to a system of time-dependent partial differential equations gives rise to a system of initial-value problems (IVPs) for ordinary differential equations (ODEs). Such systems are often stiff and very large. The need to solve problems of this kind has affected the development of both formulas and codes for IVPs for ODEs. We survey some of these developments.

Key words: Method of Lines, MOL, time-dependent partial differential equations, PDEs, ordinary differential equations, ODEs, initial value problems, IVPs, large systems, stiff problems.

AMS(MOS) subject classification: 65L05.

---

* This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.

1 Introduction

Systems of initial value problems (IVPs) for linearly-implicit ordinary differential equations (ODEs),

$$A(t)\,y'(t) = f(t, y(t)), \qquad t \in [a, b], \tag{1}$$

$$y(a) = y_0, \tag{2}$$

where $y \in \mathcal{R}^m$, arise from the application of the method of lines (MOL) to time-dependent partial differential equations (PDEs). These ODE systems are often stiff and very large.

For some spatial discretizations of a time-dependent PDE, $A(t) = I$, in which case (1) reduces to an ODE in standard form:

$$y'(t) = f(t, y(t)), \qquad t \in [a, b]. \tag{3}$$

For other spatial discretizations, $A(t) \neq I$, but $A(t)$ is nonsingular and well-conditioned for all t. In this case, numerical methods for solving the standard IVP (2)–(3) can easily be extended to (1)–(2). Therefore, for simplicity throughout this paper we assume that $A(t) = I$.

For still other spatial discretizations of PDEs, $A(t)$ is singular or poorly conditioned. Many such problems can be solved by numerical methods for differential algebraic equations (DAEs). Although many of the techniques for ODEs discussed in this paper also apply to DAEs, we do not consider here the additional difficulties associated with DAEs.

The development of formulas and codes for IVPs for ODEs has been influenced by the need to solve such large stiff systems. We review several of these developments in this article. Choosing which topics to include in a brief review of this nature is, of course, a subjective activity. By and large, I have selected topics with which I have had some research involvement. See [35] for a more comprehensive survey.

The talk upon which this paper is based concluded with a discussion of parallel methods for IVPs for ODEs. However, we have omitted this material from our survey because good reviews of parallel ODE schemes are available. See, for example, Burrage's book [8], the four survey papers and twelve research articles in [1] or my older survey paper [28].

2 Solving Implicit Equations

All commonly used formulas for solving stiff IVPs are implicit in some sense. Therefore, stiff ODE codes must solve a linear or nonlinear system at each step of the integration. Throughout this section, we use the backward differentiation formulas (BDFs) to illustrate the effect that this has on the design of stiff ODE solvers. Much of what we say also applies to other formulas as well.

The k-step BDF has the well-known form

$$\frac{1}{h}\sum_{i=0}^{k} \alpha_i y_{n-i} = f(t_n, y_n) \tag{4}$$

where h is the stepsize and the coefficients α_i depend on k only. At each step of an integration with this scheme, we must solve the implicit equation

$$g(y_n) \equiv \frac{\alpha_0}{h} y_n - f(t_n, y_n) + \psi_n = 0 \tag{5}$$

for y_n, where $\psi_n = \frac{1}{h}\sum_{i=1}^{k} \alpha_i y_{n-i}$ represents known past values.

Some variant of Newton's method is typically used to solve (5), since simpler schemes, such as functional iteration, normally fail to converge without a severe restriction on the stepsize in stiff ODE codes. Given an initial guess $y_n^{(0)}$ for y_n, this class of Newton-like nonlinear equation solvers takes the form

$$\begin{aligned} W_n^{(l)} \Delta_n^{(l)} &= -g(y_n^{(l)}) && (6) \\ y_n^{(l+1)} &= y_n^{(l)} + \Delta_n^{(l)} && (7) \end{aligned}$$

for $l = 0, 1, \ldots$ For the full Newton method,

$$W_n^{(l)} = g'(y_n^{(l)}) = \frac{\alpha_0}{h} I - f_y(t_n, y_n^{(l)}) \tag{8}$$

is an $m \times m$ matrix. Typically, given $y_n^{(l)}$, an LU factorization of $W_n^{(l)}$ is computed, the factors are used to solve (6) for $\Delta_n^{(l)}$, and $y_n^{(l+1)}$ is computed by (7).

For large problems, such as those that arise from the MOL, evaluating the Jacobian $f_y(t_n, y_n^{(l)})$ and factoring the Newton iteration matrix $W_n^{(l)}$ are often by far the most computationally expensive operations performed during the integration. As a result, a simplified Newton method, in which $W_n^{(l)}$ is held constant for as many iterations of (6)–(7) and steps of the formula (4) as is efficient, is typically used instead of a full Newton method. Of course, while $W_n^{(l)}$ is held constant, the Jacobian $f_y(t_n, y_n^{(l)})$ is not re-evaluated and $W_n^{(l)}$ is not refactored. As a result, methods incorporating a simplified Newton scheme often enjoy significant savings in computational work compared to methods that use the full Newton iteration, particularly when applied to large problems. Because of this, most good modern ODE solvers incorporate sophisticated heuristics to decide when to update the scalar α_0/h, to re-evaluate the Jacobian $f_y(t_n, y_n^{(l)})$ and to refactor $W_n^{(l)}$. See [3] and the references therein for a further discussion of this point. Next, we consider the effect these heuristics have on the choice of the variable-stepsize extension of the BDFs.

3 Variable-Stepsize BDFs

In the preceding section, we used the standard textbook description of the BDFs that assumes that a *fixed stepsize* (FS) is used throughout the integration, but production codes vary the stepsize — often by several orders of magnitude. In this section, we consider three possible techniques of implementing *variable-stepsize* (VS) BDF codes and the influence that the need to solve large, stiff problems has had on the choice of VS implementation technique. The material presented here is discussed in more detail in [30], where additional references are cited and variable-order methods are also considered.

To develop a VS extension of (4), recall that the name backward differentiation formula comes from an equivalent formulation of these schemes in which we let $p_n(t)$ be the unique polynomial of degree k or less that interpolates (t_{n-i}, y_{n-i}) for $i = 0, \ldots, k$ and we determine y_n from the collocation condition

$$p_n'(t_n) = f(t_n, y_n). \tag{9}$$

Formula (4) follows immediately from (9) by writing $p_n(t)$ in Lagrangean form. Similarly, the divided-difference or Nordsieck forms of the BDFs follow from (9) by writing $p_n(t)$ in divided-difference or Taylor series form, respectively.

Using the formulation (9), it is natural to extend (4) to a VS formula by simply dropping the restriction that the past stepsizes $h_{n-i} = t_{n-i} - t_{n-i-1}$, $i = 0, \ldots, k-1$, are all equal. Writing $p_n(t)$ in Lagrangean form, (9) becomes

$$\frac{1}{h_n} \sum_{i=0}^{k} \alpha_{n,i} y_{n-i} = f(t_n, y_n), \tag{10}$$

where

$$\alpha_{n,i} = h_n \, l_{n,i}'(t_n) \quad \text{and} \quad l_{n,i}(t) = \prod_{\substack{j=0 \\ j \neq i}}^{k} \frac{t - t_{n-j}}{t_{n-i} - t_{n-j}} \, . \tag{11}$$

This is called the *variable-coefficient* (VC) implementation of the BDFs since the coefficient $\alpha_{n,i}$ depends on the stepsize ratios $h_{n-1}/h_n, \ldots, h_{n-k+1}/h_n$. Moreover, it is easy to see that (10) is an extension of (4) in the sense that $\alpha_{n,i} = \alpha_i$ if $h_n = h_{n-1} = \cdots = h_{n-k+1}$.

The VC implementation of the Adams formulas is used in several codes for nonstiff ODEs (see for example DE/STEP/INTERP[1] [36]), but EPISODE [13] is the only public-domain production-quality BDF code of which we know that uses this approach. One possible reason for this is that, although a VC implementation is in some sense more stable and robust than the other two commonly used VS schemes discussed below, it is more costly in a stiff ODE code because the Newton iteration matrix (8) becomes

$$g'(y_n^{(l)}) = \frac{\alpha_{n,0}}{h_n} I - f_y(t_n, y_n^{(l)}), \tag{12}$$

where, from (11),

$$\begin{aligned} \alpha_{n,0} &= h_n \, l_{n,0}'(t_n) \\ &= 1 + \frac{1}{1 + h_{n-1}/h_n} + \cdots + \frac{1}{1 + h_{n-1}/h_n + \cdots + h_{n-k+1}/h_n} \, . \end{aligned}$$

Therefore, since $\alpha_{n,0}$ typically varies for $k-1$ steps after a stepsize change even if h_n is not changed again, a Newton iteration matrix from a previous step may not be a good approximation to the Newton iteration matrix (12) on the current step. Consequently, the simplified Newton iteration described in §2 is often not as effective with the VC implementation as it is with the other two VS schemes described below.

[1] Many of the codes mentioned in this paper can be obtained from netlib [15, 16].

The *fixed-coefficient* (FC) implementation of the BDFs is used in DIFSUB [19], GEAR [23], LSODE [24] and several variants of these codes. This VS extension of the BDFs can be derived by letting $\tilde{p}_n(t)$ be the unique polynomial of degree k or less that interpolates the $k+1$ evenly spaced past y-values (t_n, y_n), $(t_n - h_n, y_{n-1})$, $(t_n - 2h_n, \tilde{y}_{n-2}), \ldots, (t_n - kh_n, \tilde{y}_{n-k})$ and choosing y_n to satisfy the collocation condition

$$\tilde{p}_n'(t_n) = f(t_n, y_n), \tag{13}$$

where $\tilde{y}_{n-2}, \ldots, \tilde{y}_{n-k}$ are computed by interpolating the $k+1$ evenly spaced past y-values (t_{n-1}, y_{n-1}), $(t_{n-1} - h_{n-1}, y_{n-2})$, $(t_{n-1} - 2h_{n-1}, \bar{y}_{n-3}), \ldots, (t_{n-1} - kh_{n-1}, \bar{y}_{n-k-1})$ used on the previous step. Note that, if $h_n = h_{n-1}$, then this "interpolation" requires no work. This process is a little more complicated after an order change, as described in [30].

A disadvantage of the FC implementation is that it is less stable and robust (in some sense) than the other two VS extensions of the BDFs considered here. One way to avoid this potential instability in an FC BDF code is to hold the stepsize and order constant for at least k steps following a stepsize change. This ensures that the interpolated values $\tilde{y}_{n-2}, \ldots, \tilde{y}_{n-k}$ are calculated from "computed" past values and not interpolated ones.

On the other hand, an advantage of the FC implementation over the VC approach is that it works effectively in conjunction with the simplified Newton iteration. To see why, note that (13) is equivalent to

$$\frac{1}{h_n}\left(\alpha_0 y_n + \alpha_1 y_{n-1} + \sum_{i=2}^{k} \alpha_i \tilde{y}_{n-i}\right) = f(t_n, y_n), \tag{14}$$

where the coefficients α_i, $i = 0, \ldots, k$, are the same as those that appear in the FS formula (4). Hence the name FC. Moreover, the Newton iteration matrix

$$g'(y_n^{(l)}) = \frac{\alpha_0}{h_n} I - f_y(t_n, y_n^{(l)}) \tag{15}$$

associated with (14) is the same as (8) for the FS case, except, of course, that the factor h_n is used instead of h. Therefore, as long as h_n stays constant, the scalar factor α_0/h_n in (15) remains fixed. Consequently, if the stepsize in an FC BDF code is restricted as described in the previous paragraph, then the simplified Newton iteration (6)–(7) usually converges satisfactorily with $W_n^{(l)} \approx g'(y_n^{(l)})$ held constant for several steps, often yielding a significant advantage over codes incorporating the VC approach.

Another reason for the initial popularity of the FC approach is that it is easy to implement in Nordsieck form: all that is required to change the stepsize is to rescale each component of the Nordsieck history array by the appropriate power of h_n/h_{n-1}. However, this is not a significant advantage over the other two VS extensions of the BDFs described here since they can be implemented efficiently as well. Moreover, in the solution of large problems, the computation of the coefficients of the method is not a significant cost for any of these techniques.

The *fixed-leading-coefficient* (FLC) implementation of the BDFs is used in the newer codes DASSL [2] and VODE [3]. This VS extension is similar to the FC implementation in that, when the stepsize is changed, both can be viewed as first interpolating past values to compute new evenly spaced past values and then applying the FS BDF formula to those evenly spaced values. However, the FC and FLC implementations differ in the choice of past values to interpolate. To be more specific, the FLC extension of the BDFs can be derived by letting $\hat{p}_n(t)$ be the unique polynomial of degree k or less that interpolates the evenly spaced past values (t_n, y_n), $(t_n - h_n, y_{n-1})$, $(t_n - 2h_n, \hat{y}_{n-2}), \ldots, (t_n - kh_n, \hat{y}_{n-k})$ and choosing y_n to satisfy the collocation condition

$$\hat{p}_n'(t_n) = f(t_n, y_n) \tag{16}$$

where (in one variant of the scheme) $\hat{y}_{n-2}, \ldots, \hat{y}_{n-k}$ are computed by interpolating the $k+1$ (possibly unequally-spaced) "true" past values (t_{n-i}, y_{n-i}), $i = 1, \ldots, k+1$. It is easy to see that (16) is equivalent to

$$\frac{1}{h_n}\left(\alpha_0 y_n + \alpha_1 y_{n-1} + \sum_{i=2}^{k} \alpha_i \hat{y}_{n-i}\right) = f(t_n, y_n), \tag{17}$$

where the coefficients α_i, $i = 0, \ldots, k$, are the same as those in the FS formula (4). It follows easily that (17) is equivalent to

$$\frac{1}{h_n}\left(\alpha_0 y_n + \sum_{i=1}^{k+1} \hat{\alpha}_{n,i} y_{n-i}\right) = f(t_n, y_n), \tag{18}$$

where the leading coefficient α_0 remains unchanged. Hence the name FLC implementation. However, the other coefficients $\hat{\alpha}_{n,i}$, $i = 1, \ldots, k+1$, depend on the past stepsize ratios, in much the same as do the $\alpha_{n,i}$. Note that the Newton iteration matrix associated with either (17) or (18) is (15), the same as for the FC implementation. As a result, the FC and FLC implementations share the advantage that the simplified Newton iteration (6)–(7) usually converges satisfactorily with $W_n^{(l)} \approx g'(y_n^{(l)})$ held constant for several steps, often yielding a significant advantage over the VC approach. However, when the stepsizes are changed frequently the FLC implementation tends to be more stable and robust than the FC implementation. Thus the FLC implementation combines (to some degree) the advantages of the VC and FC implementations. See [3] and [30] for further discussion.

4 Similarity Factorizations

As noted in §2, the cost of evaluating the Jacobian $f_y(t_n, y_n^{(l)})$ and refactoring the Newton iteration matrix $W_n^{(l)}$ (8) is often the dominant cost in solving a large stiff IVP. As a result, many codes use a simplified Newton method (6)–(7) to solve (5) in which $W_n^{(l)}$ is factored when the scalar α_0/h and/or the matrix $f_y(t_n, y_n^{(l)})$ are re-evaluated and then that factorization of $W_n^{(l)}$ is used repeatedly to solve (6) for as long as is efficient.

However, as alluded to in §3, it often happens that a change in the scalar factor α_0/h, rather than a change in the Jacobian $f_y(t_n, y_n^{(l)})$, forces the re-evaluation and subsequent re-factorization of $W_n^{(l)}$. Many codes, such as VODE [3], include heuristics to decide when to update only the scalar factor α_0/h before re-factoring $W_n^{(l)}$ and when to update both α_0/h and the Jacobian matrix $f_y(t_n, y_n^{(l)})$. (In the first case, a stored copy of f_y can be used in forming $W_n^{(l)}$.) The use of such sophisticated heuristics often results in significant savings when solving large stiff problems.

However, even greater savings could be achieved if there were a cheap way to update the factorization of $W_n^{(l)}$ when the scalar factor α_0/h is changed. This, though, does not seem possible if an LU factorization of $W_n^{(l)}$ is used. On the other hand, Enright [17] observed that a cheap update of a similarity-based factorization is possible. To explain this idea, let

$$W = \gamma I - J, \tag{19}$$

be the $m \times m$ Newton iteration matrix, where γ is a scalar corresponding to α_0/h in (8), I is the identity and J is an approximation to the Jacobian $f_y(t_n, y_n^{(l)})$. A similarity-based factorization

$$W = SHS^{-1} \tag{20}$$

is used, where S is typically either lower triangular or orthogonal and H is upper-Hessenberg. (Such factorizations are often used in the first step of computing the eigenvalues of a matrix.) To solve $Wx = b$,

1. compute $\tilde{b} = S^{-1}b$ (or equivalently solve $S\tilde{b} = b$),
2. solve $Hy = \tilde{b}$,
3. compute $x = Sy$.

One way to solve $Hy = \tilde{b}$ is first to perform an LU factorization of H and then to use the factors to solve this system. Note that the number of arithmetic operations required to compute the LU factorization of an $m \times m$ upper-Hessenberg matrix H is proportional to m^2, while the number required for the LU factorization of the general matrix W is proportional to m^3, assuming standard methods are used in each case.

If we change $W = \gamma I - J$ to $\hat{W} = \hat{\gamma} I - J$, then we can easily update the similarity factorization (20), since

$$\begin{aligned}
\hat{W} &= (\hat{\gamma} - \gamma)I + \gamma I - J \\
&= (\hat{\gamma} - \gamma)I + W \\
&= S((\hat{\gamma} - \gamma)I)S^{-1} + SHS^{-1} \\
&= S((\hat{\gamma} - \gamma)I + H)S^{-1} \\
&= S\hat{H}S^{-1},
\end{aligned}$$

where $\hat{H} = (\hat{\gamma} - \gamma)I + H$ is upper-Hessenberg also. Therefore, to solve a series of systems with $\hat{W}$, we need only compute a new LU factorization of $\hat{H}$. As noted above, this is much less costly than computing a new LU factorization of $\hat{W}$. On the other hand, each factorization and each solve is more expensive for the similarity-based schemes than for the standard LU approach. For example, if the factor γ in a general $m \times m$ matrix W is changed r times (resulting in r factorizations) and k solves are performed with each factorization, then the standard LU-based method would require approximately $\frac{1}{3}rm^3 + krm^2$ flops (combined floating-point adds and multiplies), while one variant of the similarity-based technique would require about $\frac{5}{6}m^3 + (r - \frac{1}{2} + \frac{3}{2}kr)m^2$ flops. For $r > 5/2$, these rough estimates suggest that the latter is cheaper than the former if $m > (3r(k+2) - 3)/(2r - 5)$. If r is large, this inequality reduces to $m > \frac{3}{2}(k+2)$, which is easily satisfied for many large systems.

Enright's numerical results [17] support the conclusion that a similarity-based factorization can be effective in the appropriate context. Unfortunately, similarity-based factorizations appear typically to suffer more fill-in than LU factorizations for sparse matrices and consequently the direct application of similarity-based techniques appears to offer less benefit in this important case.

Enright and Kamel [18] extended the idea outlined above to partition automatically ODE systems into their stiff and nonstiff components. Assume we use orthogonal transformations to factor W. If we stop the factorization early, then, instead of an upper-Hessenberg H, we are left with

$$S^T W S = H = \begin{pmatrix} H_1 & R_{12} \\ 0 \; v & R_{22} \end{pmatrix}$$

where H_1 is upper-Hessenberg, v is a vector with all components, except possibly the first, equal to zero, and R_{12} and R_{22} are possibly full matrices. If $\|v\|$ and $\|R_{22}\|$ are "small enough", we have essentially partitioned the transformed system into its stiff components (corresponding to H_1) and its nonstiff components (corresponding to R_{22}). Enright and Kamel suggest replacing H by

$$\tilde{H} = \begin{pmatrix} H_1 & R_{12} \\ 0 & \gamma I \end{pmatrix}$$

which corresponds to replacing W by $\tilde{W} = S\tilde{H}S^T$. (An alternative is to replace v by zero and R_{22} by its diagonal.) Solving $\tilde{W}\Delta = -g$, instead of $W\Delta = -g$ as in (6), can be viewed as applying a variant of Newton's method to the stiff components of the system and a predictor-corrector iteration to the nonstiff components. If the dimension of H_1 is much smaller than that of W, this approach can lead to significant savings. However, as for the full scheme, because of fill-in this method losses some of its attractiveness when applied naively to sparse systems.

Higham [22] observed the close connection between the Enright–Kamel scheme and Krylov subspace methods, and he used this observation to explain why the dimension of H_1 is typically not much larger than the number of stiff components in the IVP. His discussion can also be used to explain the effectiveness of (unpreconditioned) Krylov subspace methods for IVPs with only a few stiff components. However, because Krylov subspace schemes do not typically generate or store the matrices R_{12} and R_{22} or the associated submatrices of S, they may be significantly more cost effective — in terms of both computational work and storage — than the Enright-Kamel scheme, particularly for large sparse problems. Moreover, effective preconditioning seems more natural in the context of Krylov subspace methods.

5 Sparse Direct Linear Equation Solvers

In MOL applications, the matrix $W_n^{(l)}$ that occurs in the Newton iteration (6)–(7) is frequently large and sparse. Therefore, it is natural to consider replacing the direct linear equation solver in a stiff ODE code such as LSODE [24] by a sparse linear equation solver. In this section, we consider direct methods and, in the next, iterative schemes. Of course, in either case, an appropriate sparse matrix data structures must be incorporated into the ODE code, but also other important considerations arise.

If a direct sparse linear equation solver is used, one consideration that significantly affects the efficiency of the scheme is the choice of pivoting strategy. Typically the solution of a symmetric positive-definite (SPD) system $Ax = b$ is divided into four phases:

1. choose a symmetric re-ordering of A based on the location of the nonzeros of A, but ignoring their values, to minimize the fill-in generated in the numerical factorization in phase 3,

2. perform a symbolic factorization of A (essentially setting up the data structures that will be used in the numerical factorization in phase 3),

3. perform a numerical factorization of A using the re-ordering and data structures from phases 1 and 2,

4. use the numerical factorization computed in phase 3 to solve $Ax = b$.

We refer to this as the SPD scheme. It is guaranteed to be stable if A is SPD, but, for general matrices, it may be unstable and produce poor results or even break down completely.

Such an SPD scheme is used to great effect in LSODES [25] even though the matrices $W_n^{(l)}$ that arise in the Newton iteration (6)–(7) may not be SPD. Since typically $W_n^{(l)}$ has the same nonzero structure for all n and l, phases 1 and 2 of the SPD scheme outlined above need be performed once only, at the start of the integration. Phase 3 alone is performed whenever a new factorization is needed. Schemes that pivot for numerical stability as well as sparsity for general systems $Ax = b$ must combine phases 1, 2 and 3 in some sense to take account of the values of the nonzeros as well as their locations. As a result, in an ODE solver, the combined phases 1, 2 and 3 must be executed each time $W_n^{(l)}$ is factored. Therefore, these more general schemes are typically significantly more expensive than the SPD scheme.

However, this added expense might be warranted if the SPD scheme used in LSODES broke down frequently. In this case, the Newton iteration (6)–(7) would fail and the stepsize would be decreased until the factorization is sufficiently good for the Newton iteration to converge. This must eventually happen, since $hW_n^{(l)} \to \alpha_0 I$ as $h \to 0$. However, this could impose a restriction on the stepsize similar to that required for functional iteration for a nonstiff ODE method.

Surprisingly, the SPD scheme used in LSODES seems rarely to suffer the breakdown outlined above. If the Jacobian $f_y(t, y)$ associated with the problem (2)–(3) is symmetric negative-definite, as is often the case for stiff problems, the constant α_0 in (8) is positive, as it is for the BDFs, and $h > 0$, as we assume throughout this paper, then $W_n^{(l)}$ is SPD and the SPD scheme outlined above is guaranteed to be stable. Moreover, it can be argued (see §2.1 of [14]) that, if the Jacobian is symmetric, but not negative-definite, then a variable-stepsize method will naturally choose its stepsizes so that $W_n^{(l)}$ will remain SPD. However, it is not so clear why the SPD scheme works so well for more general stiff ODE problems. An explanation of the effectiveness of the SPD scheme in this context would be reassuring.

6 Iterative Linear Equation Solvers

As noted in the previous section, in MOL applications, the matrix $W_n^{(l)}$ that occurs in the Newton iteration (6)–(7) is frequently large and sparse. We discussed the use of sparse direct methods for solving (6) in the previous section; we consider iterative schemes here.

Recently, most attention has been focused on preconditioned Krylov subspace methods. For a discussion of this class of iterative linear equation solvers in the context of stiff ODE codes, see [6, 7, 14, 20, 31] and the references therein.

In the introductory comments of VODPK [4], Brown, Byrne and Hindmarsh state:

> "To achieve an efficient solution, the preconditioned Krylov methods in VODPK generally require a thoughtful choice of preconditioners. If the ODE system produces linear systems that are not amenable to solution by such iterative methods, the cost can be higher than with a solver that uses sparse direct methods. However, for many systems, careful use of VODPK can be highly effective."

A preconditioner for a linear system $Ax = b$ is a pair of nonsingular matrices P_1 and P_2 and the associated preconditioned system is $\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = P_1 A P_2$, $x = P_2\hat{x}$ and $\hat{b} = P_1 b$. Often one of P_1 or P_2 is I. In a preconditioned Krylov subspace method, the matrix $\hat{A} = P_1 A P_2$ is not normally formed explicitly, since it is necessary only to multiply an arbitrary vector v by $\hat{A} = P_1 A P_2$. Therefore, in this context, a preconditioner pair P_1 and P_2 is good if it is relatively inexpensive to compute $P_1 v$ and $P_2 v$ and the iterative method applied to the preconditioned system $\hat{A}\hat{x} = \hat{b}$ converges much more rapidly than when applied to the original system $Ax = b$.

Other authors have made similar observations about the potential of iterative linear equation solvers in stiff ODE codes. Such schemes nearly always save space and can often save computational work as well. Chan and Jackson [14] provide a rough estimate of the amount of space and computational work required to solve the heat equation in 1, 2 and 3 spatial-dimensions by the MOL and conclude that the linear equations are best solved by

- a band solver in 1–D,
- either a sparse direct or a preconditioned Krylov subspace method in 2–D,
- a preconditioned Krylov subspace method in 3–D.

The need for the user to choose an effective preconditioner when using a routine such as VODPK adds the requirement that the user be much more knowledgeable about the numerical methods used than is the case for most other ODE codes, which are normally treated as "black boxes".

Part of the reason for the effectiveness of iterative linear equation solvers in stiff ODE codes is that there is usually a good initial guess $y_n^{(0)}$ for y_n and the linear system (6) does not need to be solved particularly accurately since the Newton iteration is used in effect only to stabilize the explicit method that computes the initial guess $y_n^{(0)}$. However this leads to an important question: What is an appropriate stopping criterion for the iterative solution of (6)? Although this question has been addressed by several authors (see [6, 7, 14, 20, 31] and the references therein), I do not believe it has been satisfactorily resolved.

As noted in §4, Krylov subspace methods are particularly effective when the system of ODEs has only a few stiff components. In this context, preconditioning may not be needed and a *matrix-free* implementation [6, 7, 14], in which the Jacobian $f_y(t_n, y_n^{(l)})$ is never explicitly generated, may be highly effective.

Jackson and Seward [31] point out that iterative linear equation solvers can also be used to extend the idea of *type insensitive* codes in which the routine automatically switches between a stiff and a nonstiff scheme as the problem becomes locally stiff or nonstiff. Instead of switching between a nonstiff scheme and a stiff one that uses a direct linear equation solver, as suggested in [34], the code could switch between a nonstiff scheme and a stiff one that uses an iterative linear equation solver. On the mildly stiff segments of the integration, the iterative scheme in the stiff solver should require a few iterations only. If the iterative scheme is implemented in matrix-free form, for example, the stiff solver would be only a little more expensive than a nonstiff predictor-correct method on these mildly stiff segments. On the stiffer segments, the number of iterations required by the iterative scheme would typically increase. In addition, the results in [31] show that the iterative solver should switch from using no preconditioner, or a very cheap one, to using a more effective (and usually more expensive) preconditioned as the problem becomes more stiff. As a final resort for very stiff problems, the stiff method could switch to a direct linear equation. In a sense, this use of an iterative linear equation solver (possibly with different levels of preconditioning) in a type insensitive code makes the transition from a nonstiff to a stiff method much smoother, often reducing the computational cost by delaying the switch to an expensive stiff method until it is really needed.

Note that LSODPK [5] and VODPK [4], the two best-known stiff ODE solvers incorporating preconditioned Krylov subspace methods, employ many of the strategies and heuristics of LSODE [24] and VODE [3], respectively. A compelling reason for this is that it is natural to make as few changes as necessary when transforming one code in a family to another code in the same family. On the other hand, many of the strategies used in LSODE or VODE seem less appropriate in LSODPK or VODPK, respectively, since the use of an iterative linear equation solver makes it less important to keep the scalar factor γ fixed in the Newton iteration matrix $W = \gamma I - J$ for as long as possible. Therefore, in particular, it seems less natural to use the fixed-coefficient and fixed-leading-coefficient implementations of the BDFs in LSODPK and VODPK, respectively, than in LSODE and VODE. It would be interesting to determine if a variable-coefficient implementation of the BDFs is a better option in a stiff ODE code employing an iterative linear equation solver.

7 Diagonally Implicit Runge-Kutta Formulas

The need to solve large systems of stiff IVPs has also influenced the development of Runge-Kutta (RK) formulas. A general s-stage RK formula for the numerical solution of the problem (2)–(3) is

$$Y_{n,i} = y_n + h_n \sum_{j=1}^{s} a_{ij} f(t_n + c_j h_n, Y_{n,j}) \tag{21}$$

$$y_{n+1} \;=\; y_n + h_n \sum_{j=1}^{s} b_j f(t_n + c_j h_n, Y_{n,j}). \tag{22}$$

It is well-known that an RK formula must be implicit to be appropriate for (very) stiff problems. That is, we must solve a system

$$G(Y_n) \equiv Y_n - h_n(A \otimes I_m)F(Y_n) - e \otimes y_n = 0 \tag{23}$$

at each step, where $Y_n = (Y_{n,1}^T, \ldots, Y_{n,s}^T)^T \in \mathcal{R}^{sm}$, $A = [a_{ij}]$ is the $s \times s$ matrix of coefficients of the RK formula (21)–(22), I_m is the $m \times m$ identity matrix, $F(Y_n) = (f(t_n + c_1 h_n, Y_{n,1})^T, \ldots, f(t_n + c_s h_n, Y_{n,s})^T)^T \in \mathcal{R}^{ms}$, $e = (1, \ldots, 1)^T \in \mathcal{R}^s$ and $\otimes$ is the Kronecker product defined by

$$M \otimes N = \begin{pmatrix} m_{11}N & \cdots & m_{1q}N \\ \vdots & & \vdots \\ m_{p1}N & \cdots & m_{pq}N \end{pmatrix} \tag{24}$$

for any $p \times q$ matrix $M = [m_{ij}]$ and any matrix N.

Note that the system (23) is similar to (5), but s times bigger. For large systems of IVPs, this causes a dramatic increase in the computational work and storage required for the Newton iteration corresponding to (6)–(7), since, for (23), the associated Newton iteration matrix $W_n^{(l)}$ is $sm \times sm$, rather than $m \times m$, as it is for (5). As a result, a dense LU factorization, for example, of the matrix $W_n^{(l)}$ associated with (23) requires about $s^3m^3/3$ flops, rather than $m^3/3$ flops for the $W_n^{(l)}$ associated with (5), and each solve of the form (6) requires about s^2m^2 flops for (23), rather than m^2 flops for (5). Similar increases in computational work occur for direct solvers if the Jacobian is symmetric, banded or sparse.

As a result, several researchers in the 1970s proposed schemes to reduce the computational work required to solve (23). One approach was the development of diagonally-implicit Runge-Kutta (DIRK) formulas. These are schemes (21)–(22) in which $a_{ij} = 0$ for $i < j$. As a result, (21) can be rewritten

$$Y_{n,i} = y_n + h_n \sum_{j=1}^{i} a_{ij} f(t_n + c_j h_n, Y_{n,j}).$$

So, if we compute $Y_{n,i}$ in the natural order $i = 1, 2 \ldots, s$, we need only solve

$$G_i(Y_{n,i}) \equiv Y_{n,i} - h_n a_{ii} f(t_n + c_i h_n, Y_{n,i}) - \psi_{n,i} = 0 \tag{25}$$

for $Y_{n,i}$ at stage i, where $\psi_{n,i} = y_n + h_n \sum_{j=1}^{i-1} a_{ij} f(t_n + c_j h_n, Y_{n,j})$ represents known past values. That is, the system (23) of sm equations in sm unknowns decouples to s systems (25), each with m equations in m unknowns. Note the similarity between (25) and (5).

Moreover, singly-diagonally-implicit Runge-Kutta (SDIRK) formulas, for which $a_{ii} = \alpha$ for all i, offer additional savings. If the same approximation J is used for all the Jacobians $f_y(t_n + c_i h_n, Y_{n,i})$, then the same $m \times m$ approximate Newton iteration matrix $W_n^{(l)} = I - h_n \alpha J$ can be used to solve all s systems (25). Therefore, to solve the s systems (25), we need only factor one Newton iteration matrix of size $m \times m$, resulting in a significant savings in both computational work and storage compared to the more general scheme (23). Moreover, given the factorization, each solve of the form (6) for each of the s systems (25) requires about

m^2 flops, bringing the cost of using an SDIRK method to solve a large system of stiff IVPs more into line with BDF scheme costs.

Of course, the restricted form of a DIRK or an SDIRK formula does impose some constraints on the order and stability properties that can be obtained compared to the more general RK formula (21)–(22). For example, the order of an s-stage DIRK or SDIRK formula is at most $s+1$ and A-stable formulas of this order exist for $s = 1, 2, 3, 5$ only. For a further discussion of DIRK and SDIRK schemes, see [12, 21] and the references therein.

Nørsett and Thomsen developed the code SIMPLE [32] based on the third-order SDIRK formula and associated error estimate described in [33]. Numerical tests show that this code is effective for the solution of large stiff systems of IVPs at relaxed to moderate tolerances, typical of the systems that arise from the MOL. For another example of a code of this type, see SDIRK4 [21].

8 Singly Implicit Runge-Kutta Formulas

As noted in §7, several researchers in the 1970s proposed schemes to reduce the computational work and storage required to solve (23). We outlined one successful approach in §7 and consider another here.

If we apply a simplified Newton's method to (23) in which each Jacobian $f_y(t_n+c_j h_n, Y_{n,j}^{(l)})$ is approximated by the same matrix $J_n^{(l)}$, then we can write the resulting approximate Newton iteration matrix in Kronecker product form:

$$\mathcal{W}_n^{(l)} = I_s \otimes I_m - h_n A \otimes J_n^{(l)}, \tag{26}$$

where I_s and I_m are $s \times s$ and $m \times m$ identity matrices, respectively. Given an initial guess $Y_n^{(0)}$ for Y_n, the associated simplified Newton iteration is

$$\begin{aligned} \mathcal{W}_n^{(l)} \Delta_n^{(l)} &= -G(Y_n^{(l)}), & (27) \\ Y_n^{(l+1)} &= Y_n^{(l)} + \Delta_n^{(l)}, & (28) \end{aligned}$$

for $l = 0, 1, \ldots$.

Butcher [11] noted that the tensor product form of $\mathcal{W}_n^{(l)}$ can be exploited to great advantage in solving (27). For example, if A is nonsingular, then, by a variant of the Jordan Normal Form Theorem, there is a nonsingular matrix T such that $T^{-1}A^{-1}T = D^{-1}B$, where $D = \mathrm{diag}(\lambda_1, \ldots, \lambda_s)$ is a diagonal matrix with the eigenvalues $\lambda_1, \ldots, \lambda_s$ of A on its diagonal ordered so that all equal λ's are grouped together and

$$B = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \delta_2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & \delta_3 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \delta_s & 1 \end{pmatrix}$$

with $\delta_i = 0$ if $\lambda_{i-1} \neq \lambda_i$ and $\delta_i = 0$ or 1 if $\lambda_{i-1} = \lambda_i$, depending on the eigensystem of A. Letting $P = DT^{-1}A^{-1}$ and $Q = T$, we can transform (27)–(28) to

$$\begin{aligned} \hat{\mathcal{W}}_n^{(l)} \hat{\Delta}_n^{(l)} &= -\hat{G}(Y_n^{(l)}), & (29)\\ Y_n^{(l+1)} &= Y_n^{(l)} + (Q \otimes I)\hat{\Delta}_n^{(l)}, & (30) \end{aligned}$$

with

$$\begin{aligned} \hat{\mathcal{W}}_n^{(l)} &= (P \otimes I_m)\mathcal{W}_n^{(l)}(Q \otimes I_m) \\ &= (P \otimes I_m)(I_s \otimes I_m - h_n A \otimes J_n^{(l)})(Q \otimes I) \\ &= (PQ \otimes I_m - h_n PAQ \otimes J_n^{(l)}) \\ &= (B \otimes I_m - h_n D \otimes J_n^{(l)}), & (31)\\ \hat{\Delta}_n^{(l)} &= (Q^{-1} \otimes I_m)\Delta_n^{(l)}, \\ \hat{G}(Y_n^{(l)}) &= (P \otimes I_m)G(Y_n^{(l)}), \end{aligned}$$

where we have used $(M_1 \otimes N_1)(M_2 \otimes N_2) = M_1M_2 \otimes N_1N_2$ for matrices of compatible dimensions. (This follows immediately from the definition of the Kronecker product (24).)

The key observation is that this transformation decouples the system (27) of sm equations in sm unknowns into s systems of m equations in m unknowns, since, by (31) and the definitions of B and D, the i^{th} block-row of (29) can be rewritten

$$(I_m - h_n \lambda_i J_n^{(l)})\hat{\Delta}_{n,i}^{(l)} = -\hat{G}_i(Y_n^{(l)}) - \delta_i \hat{\Delta}_{n,i-1}^{(l)}, \tag{32}$$

where the term $\delta_i \hat{\Delta}_{n,i-1}^{(l)}$ is taken to be zero for $i = 1$. Since the coefficients P, Q, λ_i and δ_i can all be precomputed from the coefficients of the Runge-Kutta (RK) formula itself, this brings the cost of solving the linear systems associated with a fully-implicit RK formula (21)–(22) more into line with the cost of solving the linear systems associated with a DIRK formula. Important implementation questions, such as how to cope with complex λ_i, are discussed in [12, 21] and the references therein.

A significant saving can be realized if all the eigenvalues $\{\lambda_i\}$ of the matrix A are identical (and therefore real also), since, in this case, the matrix on the left side of (32) reduces to $I_m - h_n \lambda J_n^{(l)}$ and is common to all s systems. Therefore, one matrix only need be factored, significantly reducing the cost of solving these equations, particularly if the underlying system of IVPs is large.

RK formulas for which all $\lambda_i = \lambda \in \mathcal{R}$ are called singly-implicit Runge-Kutta (SIRK) formulas and were first investigated by Burrage [9]. Although SIRK formulas are in many ways more general than SDIRK schemes, their restricted form still imposes some constraints on the order and stability properties compared to the more general RK formula (21)–(22). For example, the order of an s-stage SIRK formula is at most $s + 1$ and A-stable formulas of this order exist for $s = 1, 2, 3, 5$ only. As noted in §7, DIRK and SDIRK formulas suffer the same constraint. However, it is much easier to construct high-order SIRK formulas with good stability properties than it is for SDIRK schemes. For a further discussion of SIRK formulas, see [12, 21] and the references therein.

Burrage, Butcher and Chipman [10] developed the variable-stepsize variable-order code STRIDE based on s-stage SIRK formulas of order s for $s = 1, 2, \ldots, 15$. These formulas are L–stable for $s = 1, 2, \ldots, 6$ and $s = 8$ and L(α)–stable with $\alpha \geq 1.45$ for $s = 7$ and $s = 9, \ldots, 15$.

Finally, note that, if all $\delta_i = 0$ in (32), these s equations completely decouple and consequently can be solved simultaneously. This is the basis for some parallel Runge-Kutta methods. For further details, see [29] and the references therein.

9 Explicit Methods with Extended Stability Regions

As noted in §1, the IVPs that arise from the MOL are often stiff, but many are only "mildly" stiff. Therefore, an explicit formula with a large region of absolute stability may be more cost effective than an implicit scheme for their numerical solution.

The Runge-Kutta-Chebyshev (RKC) formulas [38] constitute one class of explicit schemes with large stability regions. When applied to the IVP (2)–(3), an s-stage formula in this class is

$$
\begin{aligned}
Y_{n,1} &= y_n, \\
Y_{n,2} &= Y_{n,1} + h_n \tilde{\mu}_2 f(t_n + c_1 h_n, Y_{n,1}), \\
Y_{n,j} &= \mu_j Y_{n,j-1} + \nu_j Y_{n,j-2} + (1 - \mu_j - \nu_j) Y_{n,1} \\
&\quad + h_n \left[\tilde{\mu}_j f(t_n + c_{j-1} h_n, Y_{n,j-1}) + \tilde{\gamma}_j f(t_n + c_1 h_n, Y_{n,1}) \right], \\
&\qquad \text{for } j = 3, \ldots, s, \\
y_{n+1} &= Y_{n,s}.
\end{aligned} \tag{33}
$$

By induction on j, it is easy to see that the RKC formula (33) can be rewritten as an s-stage explicit Runge-Kutta formula in the standard form (21)–(22). However, (33) is an effective way to implement the scheme, since, in this form, it is clear that the calculation of each new stage vector $Y_{n,j}$ requires one additional f evaluation and at most 5 scalar-vector multiplies and 4 vector adds. In addition, the storage required is approximately 5 m-vectors, independent of s. Moreover, Verwer, Hundsdorfer and Sommeijer [38] show that, for any $s \geq 2$, you can choose the coefficients in (33) so that the resulting formula is first order, and, for any $s \geq 3$, you can choose a set of coefficients so that the resulting formula is second order. Furthermore, for both families of formulas, the associated stability function is a shifted Chebyshev polynomial of the first kind of degree $s-1$. Consequently, the associated stability region is a long narrow strip in the complex plane centered around the negative real axis. To be more specific, define the *real stability interval* of a formula to be the largest real interval $[-\beta_s, 0]$ contained in the formula's stability region. For an s-stage RKC formula of either family, β_s is proportional to s^2, with the constant of proportionality being about 2 for the first-order formulas and about 2/3 for the second-order ones. Moreover, for both families of formulas, if the stepsize is inside the stability region, then the computation of the internal stage values $\{Y_{n,j} : j = 1, \ldots, s\}$ is also stable.

The strategy recommended by Verwer, Hundsdorfer and Sommeijer [38] is to choose the stepsize h_n at step n based on accuracy considerations only and then choose s sufficiently large so that the formula is stable. This strategy is implemented in Sommeijer's routine RKC [37] which is based on the second-order family of RKC formulas. The routine is recommended for IVPs originating from the application of the MOL to diffusion-dominated parabolic PDEs with modest accuracy requirements.

Higher-order explicit formulas with extended stability regions can be developed in a similar way. See [26] and [27] for examples of higher-order Runge-Kutta schemes and predictor-corrector multistep methods, respectively.

In closing, note that there is some similarity between the schemes discussed here and implicit methods for which the associated Newton update formula (6) is solved by a Krylov subspace scheme, particularly if the Krylov subspace scheme is implemented in matrix-free form, as discussed in §6. Naturally, there are advantages and disadvantages to these two approaches. For example, the RKC schemes discussed in this section fall within the well-understood class of explicit RK formulas, so they can be easily analyzed. On the other hand, preconditioning, for example, seems more natural in the Krylov subspace setting, and, as noted earlier, the appropriate choice of a preconditioner can greatly increase the effectiveness of an implicit method that employs a Krylov subspace scheme to solve the Newton update formula (6).

References

[1] A. Bellen and M. Zennaro (eds.), Special Issue on Parallel Methods for Ordinary Differential Equations, *Appl. Numer. Math.*, **11(1–3)**, 1993.

[2] K. E. Brenan, S. L. Campbell and L. R. Petzold, *Numerical Solution of Initial-Value Problems in in Differential-Algebraic Equations*, North-Holland, Amsterdam, 1989.

[3] P. N. Brown, G. D. Byrne and A. C. Hindmarsh, VODE: A Variable Coefficient ODE Solver, *SIAM J. Sci. Stat. Comput.*, **10**, 1989, 1038–1051.

[4] P. N. Brown, G. D. Byrne and A. C. Hindmarsh, VODPK: A Variable-Coefficient Ordinary Differential Equation Solver with the Preconditioned Krylov method GMRES for the Solution of Linear Systems, available from Netlib, 1994.

[5] P. N. Brown and A. C. Hindmarsh, LSODPK: Livermore Solver for Ordinary Differential Equations with Preconditioned Krylov Iteration Methods for the Newton Correction of Linear Systems, private communication, 1987.

[6] P. N. Brown and A. C. Hindmarsh, Matrix-Free Methods for Stiff Systems of ODE's, *SIAM J. Numer. Anal.*, **23**, 1986, 610–638.

[7] P. N. Brown and A. C. Hindmarsh, Reduced Storage Matrix Methods in Stiff ODE Systems, *Appl. Math. Comp.*, **31**, 1989, 40–91.

[8] K. Burrage, *Parallel and Sequential Methods for Ordinary Differential Equations*, Oxford Science Publishers, Clarendon Press, Oxford, 1995.

[9] K. Burrage, A Special Family of Runge-Kutta Methods for Solving Stiff Differential Equations, *BIT*, **18**, 1978, 237–240.

[10] K. Burrage, J. C. Butcher and F. H. Chipman, An Implementation of Singly-Implicit Runge-Kutta Methods, *BIT*, **20**, 1980, 326–340.

[11] J. C. Butcher, On the Implementation of Implicit Runge-Kutta Methods, *BIT*, **16**, 1976, 237–240.

[12] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations*, John Wiley & Sons, New York, 1987.

[13] G. D. Byrne and A. C. Hindmarsh, A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations, *ACM Trans. Math. Software*, **1**, 1975, 71–96.

[14] T. F. Chan and K. R. Jackson, The Use of Iterative Linear-Equation Solvers in Codes for Large Systems of Stiff IVPs for ODEs, *SIAM J. Sci. Stat. Comput.*, **7**, 1986, 378–417.

[15] J. Dongarra and E. Grosse, Distribution of Mathematical Software Via Electronic Mail, *Communications of the ACM*, **30**, 1987, 403–407.

[16] J. Dongarra, T. Rowan and R. Wade, Software Distribution Using XNETLIB, *ACM Trans. Math. Software*, **21**, 1995, 79–88.

[17] W. H. Enright, Improving the Efficiency of Matrix Computations in the Numerical Solution of Stiff Ordinary Differential Equations, *ACM Trans. Math. Software*, **4**, 1978, 71–96.

[18] W. H. Enright and M. S. Kamel, Automatic Partitioning of Stiff Systems and Exploiting the Resulting Structure, *ACM Trans. Math. Software*, **5**, 1979, 374–385.

[19] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[20] C. W. Gear and Y. Saad, Iterative Solution of Linear Equations in ODE Codes, *SIAM J. Sci. Stat. Comput.*, **4**, 1983, 583–601.

[21] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer-Verlag, New York, 1991.

[22] D. J. Higham, Analysis of the Enright–Kamel Partitioning Method for Stiff Ordinary Differential Equations, *IMA J. Numer. Anal.*, **9**, 1989, 1–14.

[23] A. C. Hindmarsh, GEAR: Ordinary Differential Equation System Solver, UCID–30001, Rev. 2, Lawrence Livermore Laboratory, Livermore, CA, 1972.

[24] A. C. Hindmarsh, LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers, *ACM Signum Newsletter*, **15**, 1980, 10–11.

[25] A. C. Hindmarsh and A. H. Sherman, LSODES: Livermore Solver for Ordinary Differential Equations with General Sparse Jacobian Matrices, available from Netlib, 1987.

[26] P. J. van der Houwen, *Construction of Integration Formulas for Initial Value Problems*, North-Holland, Amsterdam, 1977.

[27] P. J. van der Houwen and B. P. Sommeijer, Predictor-Corrector Methods with Improved Absolute Stability Regions, *IMA J. Numer. Anal.*, **3**, 1983, 417–437.

[28] K. R. Jackson, A Survey of Parallel Numerical Methods for Initial Value Problems for Ordinary Differential Equations, *IEEE Trans. on Magnetics*, **27**, 1991, 3792–3797.

[29] K. R. Jackson and S. P. Nørsett, The Potential for Parallelism in Runge-Kutta Methods. Part 1: RK Formulas in Standard Form, *SIAM J. Numer. Anal.*, **32**, 1995, 49–82.

[30] K. R. Jackson and R. Sacks-Davis, An Alternative Implementation of Variable Stepsize Multistep Formulas for Stiff ODEs, *ACM Trans. Math. Software*, **6**, 1980, 295–318.

[31] K. R. Jackson and W. L. Seward, Adaptive Linear Equation Solvers in Codes for Large Stiff Systems of ODEs, *SIAM J. Sci. Comput.*, **14**, 1993, 800–823.

[32] S. P. Nørsett and P. G. Thomsen, User's Guide for SIMPLE, a Stiff System Solver, private communication.

[33] S. P. Nørsett and P. G. Thomsen, Embedded SDIRK Methods of Basic Order Three, *BIT*, **24**, 1984, 634–646.

[34] L. R. Petzold, Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations, *SIAM J. Sci. Stat. Comput.*, **4**, 1983, 136–148.

[35] W. L. Seward, G. Fairweather and R. L. Johnston, A Survey of High-Order Methods for the Numerical Integration of Semidiscrete Parabolic Problems, *IMA J. Numer. Anal.*, **4**, 1984, 375–425.

[36] L. F. Shampine and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[37] B. P. Sommeijer, RKC: A Runge-Kutta-Chebyshev Subroutine, available from Netlib, 1991.

[38] J. G. Verwer, W. H. Hundsdorfer and B. P. Sommeijer, Convergence Properties of the Runge-Kutta-Chebyshev Method, *Numer. Math.*, **57**, 1990, 157–178.