MATHEMATICS OF COMPUTATION
VOLUME 36, NUMBER 154
APRIL 1981

Type-Insensitive ODE Codes Based on Implicit A-Stable Formulas

By L. F. Shampine

Abstract. A special concept of stiffness is appropriate for implicit A-stable formulas. It is possible to recognize this kind of stiffness economically and reliably using information readily available during the integration of an ODE. Using this development, a variety of effective ODE solvers could be made insensitive to the type of problem, i.e. the code would automatically recognize and alter automatically its algorithm at any step depending on whether the problem is stiff there.

1. Introduction. Codes for the numerical solution of the initial value problem for a system of ordinary differential equations (ODEs) are divided into two types. One type is intended for the solution of stiff problems and the other type for nonstiff problems. The first thing every user must do is to decide which type is appropriate to his problem. The decision is important. If the problem is at all stiff, it is prohibitively expensive to use a code intended for nonstiff problems. If the problem is not stiff, it is feasible to use a code intended for stiff problems. However, the cost is, relatively speaking, very high because such codes form Jacobians, form and factor iteration matrices, and solve linear systems. These are expensive operations not performed in codes for nonstiff problems. The storage required is also very high because of the matrices used. In addition, formulas in codes for stiff problems are often of comparatively low order, and the nonstiff problems are the ones which might well be solved to high accuracy. Worst of all, from the user's point of view, is that he has to get much more involved in the solution because stiff problems are simply harder. The most distasteful matters are the provision of a Jacobian and of structure information about the Jacobian.

Naturally, the first question a user asks is how to recognize stiffness. Numerical analysts become evasive then because there is no simple answer. Stiffness depends on the formula used in the code as well as on the user's problem. It depends on the solution itself which is, of course, not available. Even a qualitative understanding of the solution behavior is not enough because stiffness depends on the equation too. It is typical that physical problems involve the solution of ODEs for a range of parameter values. Small changes of a parameter may change the overall type; see for example [1], [2]. Even if it were possible to provide the user with an answer adequate for current codes, this would still be unsatisfactory for the code developer because the type can, and ordinarily does, change in the course of the integration of a "stiff" problem. Almost all problems described as "stiff" have regions of sharp

Received January 9, 1980; revised August 11, 1980.
1980 *Mathematics Subject Classification.* Primary 65L05.
Key words and phrases. ODE codes, stiffness, A-stable.

0025-5718/81/0000-0059/$04.00

change (boundary layers) in which the integration is relatively expensive but not stiff. From this sketch of the situation, it is fair to say that the most serious defect in current *user interfaces* to codes for the initial value problem for ODEs is this type decision required of the user. There are several possible remedies; in this paper we describe one way to provide type-insensitive codes.

As we see it, there are certain minimum requirements for a truly type-insensitive code. If a problem is unequivocally nonstiff, we must avoid entirely the formation of Jacobians and its associated storage and matrix computations. If the problem is unequivocally stiff, we must use a method efficient for stiff problems. Of course, it would be desirable to use a method efficient for nonstiff problems when the problem is not stiff, but this is by no means essential. We are insisting that the worst inefficiencies be avoided, so if the method used is at all reasonable, the cost of solving these relatively cheap problems will be acceptable (or at least tolerable) to a great many users. In this paper we show how to achieve this minimum level of success and more. Most stiff problems will actually be solved more efficiently using our ideas, because we take advantage of portions of the integration which are not stiff. The solution of nonstiff problems will be reasonably efficient, although not in general comparable to that of the best codes for this specific task. However, we will describe how to solve the nonstiff portions about as well as possible, when we restrict ourselves to formulas of order two. Though limited in scope, this result could provide an extremely useful tool in contexts such as simulation languages where low order may be acceptable and an efficient type-insensitive code of obvious value.

The theoretical results are interesting quite aside from their practical implications. By restricting our attention to implicit A-stable (IA) formulas, we are able to characterize stiffness in a practical way. It turns out that one can recognize this kind of stiffness, which we term "IA-stiffness," economically and reliably using information readily available during the integration. This tidy development provides the foundation for a subsequent paper reporting our investigation of formulas which are not A-stable.

2. IA-Stiffness and its Recognition. The practical man frequently asks, "What *is* stiffness?" There is no simple answer. A little reflection about the theory and practice provides one reason–stiffness depends on the formula being considered as well as the problem. Implicit A-stable (IA) formulas are the object of this study. Although the word "implicit" may seem superfluous here, we insist on it, as well as "A-stable," so as to draw attention to the two attributes which are the foundation of our investigation. To remind the reader that we employ a special concept of stiffness, we shall speak of "IA-stiffness."

It may be puzzling that one even mentions stiffness in connection with A-stable formulas. Insofar as we have noticed, no one has asked straight out, what is stiffness for an A-stable formula, but the question has been answered implicitly many times. To see this we must review the situation. The review will also serve to state a number of items we shall need later. First let us recall a few familiar examples from the family of Adams-Moulton (AM) formulas and the family of backward differentiation formulas (BDF). The formulas of order 1, AM1 and BDF1, are the same formula, usually called the backward Euler formula. When

solving a system of differential equations,

$$y' = f(x, y),$$

it is

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1}).$$

Here y_n approximates $y(x_n)$ and $x_{n+1} = x_n + h$. The AM formula of order 2 is better known as the trapezoidal rule,

$$y_{n+1} = y_n + \frac{h}{2}\left[f(x_{n+1}, y_{n+1}) + f(x_n, y_n)\right].$$

The BDF of order 2 with constant step size h is

$$y_{n+1} = \frac{2}{3} hf(x_{n+1}, y_{n+1}) + \frac{4}{3} y_n - \frac{1}{3} y_{n-1}.$$

The formulas cited are implicit, as are all the common formulas intended for the solution of stiff problems. Suppressing the independent variable, we see that these formulas have the generic form

$$y = h\gamma f(y) + \psi, \qquad h\gamma > 0, \tag{1}$$

for the algebraic equations to be solved at each step for their solution y^*. Here y represents the new solution value, γ comes from the formula, h is the step size, and ψ lumps together previously computed quantities. Easy extensions of the form cover formulas with more complicated structure, such as implicit Runge-Kutta formulas. The standard way [3] of solving (1) is to approximate the Jacobian matrix at x_{n+1}, y_{n+1} by a matrix J and then to improve an approximate solution y^m by solving

$$y^{m+1} = \psi + h\gamma f(y^m) + h\gamma J(y^{m+1} - y^m), \tag{2a}$$

or, equivalently,

$$(I - h\gamma J)(y^{m+1} - y^m) = \psi + h\gamma f(y^m) - y^m. \tag{2b}$$

The choice $J = 0$ is called simple, or functional, iteration. If J is nontrivial, a lot more work is involved. In the order of (typically) decreasing cost, an approximate Jacobian J must be formed, an iteration matrix $I - h\gamma J$ must be formed and factored, and the systems of linear equations (2) must be solved. The iteration with $J \neq 0$ is called a simplified Newton or, sometimes, quasi-Newton iteration. Because a change of step size alters the iteration matrix and so induces a relatively expensive matrix decomposition, codes for stiff problems make such changes only when they must and when a significant increase of step size is possible. Codes for nonstiff problems take advantage of even modest alterations of step size.

The practical difficulty of stiffness is that a step size which would yield the desired accuracy must be severely restricted for other reasons, and so one has to work much harder than appears reasonable. With an implicit A-stable formula, the usual restriction on the step size to maintain absolute stability is not present. The only obvious restriction on the step size arises in the solution of the implicit formula. Simple iteration may require the step size to be severely restricted in order to get adequate convergence. In such a situation using a nontrivial approximation to the Jacobian in (2) may greatly ease this restriction. The simplified Newton iteration is much more expensive per step than simple iteration, but it can lead to

enormous increases in efficiency because the step size used can be so much larger. It is worth comment that how rapidly the true Jacobian changes along the solution, the accuracy of the approximation J, the way J is used, and the like may lead to other restrictions on the step size in order to secure adequate convergence.

It is quite clear from previous use of the example formulas that people consider a problem to be nonstiff for them if convergence of simple iteration does not restrict unduly the step size, and stiff otherwise. The backward Euler formula is especially illuminating. In its guise as AM1, it is evaluated by simple iteration and used for nonstiff problems. In its guise as BDF1, it is evaluated by a simplified Newton method and used for stiff problems. The famous DIFSUB code of Gear [4] implements both the Adams-Moulton family and the BDF and makes precisely this distinction for this formula.

For an implicit A-stable (IA) formula, we define the problem to be nonstiff at x_{n+1}, y_{n+1} if simple iteration works "satisfactorily" and otherwise stiff. This is in agreement with previous practice with such formulas. It is an unusually specific and practical definition for what we call IA-stiffness.

With mild smoothness assumptions on f, the condition [5] for convergence of simple iteration for all starting guesses y^0 in a neighborhood of a solution y^* of (2) is that

$$h\gamma\rho\left(\frac{\partial f}{\partial y}(y^*)\right) < 1,$$

where $\rho(M)$ means the spectral radius of the matrix M. If this condition holds, there is a norm in which the iteration is contracting. The criterion is not at all realistic. Codes work with specific norms and must observe convergence which must be rapid. It is of no interest that a process will eventually converge if the code permits no more than three or four iterations, as is typical. The practical condition is that

$$h\gamma\left\|\frac{\partial f}{\partial y}(y^*)\right\| < 1. \tag{3}$$

The condition (3) must be modified to take into account certain other practical issues. For one thing, simple iteration is so much cheaper than a simplified Newton iteration and permits so much more rapid variation of step size that it is cost-effective to reduce the step size h substantially to some δh if necessary to secure adequate convergence. The condition (3) merely says that simple iteration will contract in a sufficiently small ball about y^*. In practice we must have pretty rapid convergence and so must require that it be at least as fast as a selected number r, e.g., 0.25. Thus, we ask if

$$\max_{\|y-y^*\|<\nu} \delta h\gamma\left\|\frac{\partial f}{\partial y}\right\| \leqslant r < 1 \tag{4}$$

for a ball of radius ν containing y^* and y^0. We remark for later use that

$$L = \max_{\|y-y^*\|<\nu} \left\|\frac{\partial f}{\partial y}\right\| \tag{5}$$

is a Lipschitz constant for f on the ball. If (4) holds, we say the problem is locally nonstiff for the IA formula and otherwise it is stiff.

3. Changing Type from Stiff to Nonstiff. It has occurred to many people to use the norm of the Jacobian to decide if simple iteration is feasible. The trouble is that being able to make this decision is all but useless if one cannot decide when to switch back to a simplified Newton iteration. It is for this reason that the author has devoted his earlier research to the harder task. A way to accomplish this task for *IA* formulas is presented in the next section. Although deciding when to switch to simple iteration is relatively easy, it is not entirely straightforward. We are not aware of any previous discussion of the practical issues involved, so we take them up in this section.

The first issue to be addressed is the matrix norm. To apply the condition (2.4), it is necessary that we select a matrix norm compatible with the vector norm being employed in the error control. It would be preferable to use the subordinate norm so that the condition will be sharp. To be practical, the matrix norm must be a cheap, simple computation. The only popular vector norms in current codes are (weighted) maximum, Euclidean, and RMS norms. The last is a constant multiple of the Euclidean norm, so there are really only two norms which concern us. The matrix norm subordinate to the maximum norm is computationally convenient, but that for the Euclidean norm is impractical. The Frobenius matrix norm is a practical alternative compatible with the Euclidean vector norm. It has enjoyed some popularity in numerical linear algebra, but the fact that it can differ substantially from the desired subordinate norm is a serious disadvantage in this context. There are various arguments in favor of one vector norm or another, but none is convincing. Our preference is to work with the maximum norm and its subordinate matrix norm because of the advantages enjoyed by the matrix norm in this context.

The condition (2.4) refers to the maximum value of the Jacobian on a ball containing the predicted solution at x_{n+1}. Our intention is to estimate this quantity by means of the approximation J, formed in the code for the iteration (2.2). At best one has a matrix obtained by evaluation of analytical expressions for the partial derivatives at a predicted solution y^0_{n+1}. Most current codes retain an iteration matrix as long as convergence is at a satisfactory rate. Only when a substantial increase of step size is possible do they form a new J and iteration matrix. A few codes may not form a new J even then, using instead the old J in the new iteration matrix. (We advocate this in [3].) The point here is that the argument for the partial derivatives may be a solution obtained at some time in the past. In addition many, if not most, problems are solved by numerical approximation of the partial derivatives. Thus, it may well be that we do not have a particularly good approximation to the desired Jacobian. A related difficulty is that the norm may vary from step to step. For example, if a pure relative error control is prescribed, the weights are the reciprocals of the solution components. Although these observations should instill a sense of caution, the situation is not at all worrisome. An old J is used only when it is believed that the Jacobian is roughly constant along and near the solution. If evidence to the contrary, such as unsatisfactory convergence, is observed, a new J is formed. Thus, we expect that $\|J\|$ be a reasonable approximation to the size of the Jacobian on a ball about the predicted solution. It would be possible to recompute $\|J\|$ at each step should the norm change, but it hardly seems worth the effort, and we do not suggest it.

Computation of $\|J\|$ is quite cheap for the maximum norm and advantage can be taken of a known sparsity of J. The cost is comparable to solving a linear system with a matrix already factored. As we suggest it, the norm is calculated every time a new J is formed and not at the formation of a new iteration matrix (a rather more frequent event if one saves J, as we advocate). One factorization of an iteration matrix costs as much as many norm computations. There is almost always an initial transient present for problems showing some stiffness and, anyway, one should start the integration with a step size small enough so that simple iteration works, so as to get on scale. The type of code we propose will take, then, at least one step with simple iteration which the usual code aimed at stiff problems will handle with a simplified Newton iteration. Avoiding the formation of the Jacobian, the factorization of the corresponding iteration matrix, and the solution of several linear systems will compensate for the extra overhead during the remainder of the integration of the test we propose.

We want to make it easy to switch to simple iteration. Simple iteration is much cheaper and it is practical to adapt the step size to the solution much more precisely than when using a simplified Newton iteration. The scheme described in the next section for recognizing stiffness is cheap. There is an important reason for making it a lot easier to switch in one direction than the other. One would not expect the character of the problem to change frequently, but if one uses a simple threshold, one might possibly encounter a special correlation of problem and threshold which causes frequent switches. An asymmetry in switching controls the difficulty. We do not suggest forming a new approximate Jacobian J for a sharper test if, when using an old $\|J\|$, we should find that a switch is possible with the current step size. For the reasons outlined, a borderline case like this would be handled most efficiently by going to simple iteration.

If one forms a new J and finds that simple iteration is possible, one switches and makes no use of this approximate Jacobian. Ideally, we would like to make the decision before forming the Jacobian; we do not see how to do this. We do advocate saving this J. Should it appear desirable to go back to a simplified Newton iteration, one must decide whether to form a new Jacobian or to use a stored J. If we retain the arguments y_k and x_k at which the stored J was evaluated, or at least $\|y_k\|$ and x_k, we can compare them to the current arguments. If the arguments have not changed much, it would appear reasonable to try using the stored J and otherwise to form a new J. This simple test will be especially valuable in avoiding troubles due to potentially frequent switches resulting from the use of the less than ideal formulas we take up in another paper.

A final point is that the test proposed is rather sharp if the approximation to the Jacobian is current. It seems highly unlikely that one would make a "mistake" and so use Newton's method when simple iteration is feasible, unless $\|J\|$ comes from a previous step. This cannot happen unless convergence is satisfactory, for otherwise a new approximate Jacobian would be formed. If convergence is satisfactory and our recommendation about retaining a copy of J is accepted, a change of step size would not be accompanied by formation of a new Jacobian. Thus, even if an isolated mistake is made, the code would solve the problem there in a relatively cheap way.

4. Changing Type from Nonstiff to Stiff. Recognizing when a problem has changed from nonstiff to stiff is much harder than recognizing a change in the reverse direction because much less information is available. For some years the author has investigated this matter for the most popular methods for solving nonstiff problems, the explicit Runge-Kutta and the Adams methods. The studies have had some success, but the schemes proposed have all been carefully described as deciding whether the reason for a code working too hard is stiffness. The goal, however, is a scheme so sensitive that information gathered in a single step would suffice. We describe here how to do this. The new element in this investigation is to ask what kind of formula, reasonable for nonstiff problems, makes the decision easy, as opposed to asking how to make the decision for specific formulas. The author worked out the basic idea some years ago for the BIOS code alluded to in [6]. The code development had to be interrupted before the new test was implemented. However, we did apply one tool in the recent study [7] of detecting large Lipschitz constants and stiffness in Adams and Runge-Kutta codes. We mention this background because it is important to understand what was different about the situation in BIOS.

Although BIOS was intended for the solution of nonstiff problems, it was based on block implicit one-step formulas, which happen to be implicit A-stable formulas. The formulas were selected because of their suitability for constructing a variable order Runge-Kutta code, and no particular importance was attached to their stability properties nor to their being implicit. Naturally, in the context of their planned use, the implicit formulas were evaluated by simple iteration. We soon realized that with any IA formula we have two step sizes of interest. There is the largest step size which could achieve the desired local accuracy, h_{acc}. There is a step size h_{iter} such that for $h < h_{\text{iter}}$, simple iteration converges. IA-stiffness means that the code must use a step size smaller than that which would give the desired accuracy so as to make the cheap iteration for evaluating the formula converge. We thus define

$$\frac{h_{\text{acc}}}{h_{\text{iter}}} = IA\text{-stiffness index.}$$

If the index is large, it is cost-effective to resort to a more expensive way of evaluating the formula which allows us to use a step size more nearly h_{acc}.

Because all modern differential equation solvers estimate h_{acc}, it appears that we need only deal with the restriction on the step size due to the iteration method. Quite the contrary. There is a fundamental difficulty with h_{acc}. In our studies of explicit Runge-Kutta methods [8], [9] we avoided using h_{acc} just because of this difficulty. We also avoided its use in a test for Adams codes [10], but more recently did try to use it in [7]. The difficulty is that in the presence of a finite absolute stability region, the step size estimated by the code as appropriate for producing the desired accuracy, h_{est}, may be derived from a "rough" numerical solution contaminated by propagated error rather than from the underlying smooth true solution. In a subsequent paper dealing with formulas which are not A-stable, we shall have to discuss this fully. Here we simply note that it is only because we work with A-stable formulas that we can always approximate $h_{\text{acc}} \doteq h_{\text{est}}$.

Next, we consider how to estimate the restriction on the step size for the convergence of simple iteration. Here is where we use the implicitness of the formula to gather the information required. Suppose the code forms h_{est}. For a number of reasons a step size $h = \zeta h_{\text{est}}$ is actually tried, where ζ is a known quantity which may depend on a variety of computed quantities. Starting with a predicted value y^0, a sequence of iterates y^m is formed by simple iteration as in (2.2a) with $J = 0$. If L is the Lipschitz constant of (2.5) on a ball about y^* of radius ν containing y^0, we have

$$\begin{aligned}\|y^{m+2} - y^{m+1}\| &= \|h\gamma f(y^{m+1}) - h\gamma f(y^m)\| \\ &\leqslant h\gamma L\|y^{m+1} - y^m\| = \zeta h_{\text{est}}\,\gamma L\|y^{m+1} - y^m\|.\end{aligned}$$

The step size h_{iter} is defined by

$$h_{\text{iter}}\,\gamma L = 1,$$

so that $h < h_{\text{iter}}$ is the condition that the iteration contract in norm on the ball. These observations provide a computable lower bound for the IA-stiffness index:

$$(1) \qquad \frac{1}{\zeta}\left\|\frac{y^{m+2} - y^{m+1}}{y^{m+1} - y^m}\right\| \leqslant \frac{\gamma L h}{\zeta} = \frac{h_{\text{est}}}{h_{\text{iter}}} \doteq \frac{h_{\text{acc}}}{h_{\text{iter}}}.$$

The lower bound for the stiffness index tells us the penalty paid for using simple iteration. If it is "large," we surely want to switch to the more expensive iteration scheme. What if the bound is not large? Does this mean the problem is not stiff? *No*, the fact that simple iteration will diverge for *some* y^0 near y^* does not preclude it converging for other y^0. Thus, it may happen that we get convergence when we had no right to expect it.

There are several reasons why we might accept a step and not even have available the bound (1) on the IA-stiffness index. Some acceptance tests are described in [3], [11], [12]. A reliable and simple acceptance test based on the residual, which is recommended by Shampine [11] and by Williams [12], could accept y^0 or y^1. A much less reliable test described in [3] could sometimes accept y^1. In such situations, the information needed to form the bound is simply not available. If y^0 or a subsequent iterate should be about as accurate as possible in the precision of the computer, the quantities $\|y^{m+1} - y^m\|$ may be roundoff errors only and we cannot form the bound reliably. (The residual test will terminate the iteration before forming such a quantity. With other acceptance tests, one can spot such a quantity easily and with reasonable reliability as we did in [7].)

Thus, the index might not reveal the stiffness, the information might not be available to compute the index, or the information available to compute the index might not be reliable. This does not matter. Our object is to integrate the differential equation efficiently, not to determine stiffness per se. If we should evaluate the formula cheaply with simple iteration when we had no right to expect it, we can just enjoy our good fortune. Because we are using A-stable formulas, we need not worry about the integration remaining stable. This is a crucial issue with formulas which are not A-stable because of the effect on h_{est}, but it is not an issue at all with the kind of formulas we study here.

To recapitulate, the fact that the formula is implicit A-stable is used in several ways:

(i) stiffness in a practical sense is equivalent to the rapid divergence of simple iteration;

(ii) $h_{\text{est}} \doteq h_{\text{acc}}$;

(iii) the integration remains stable if we make a "mistake" by accepting a step computed with simple iteration when simple iteration is not in general convergent for this step size;

(iv) the step size restriction due to simple iteration can be estimated from available data.

We have already said that we believe every integration should be started off with a step size small enough that simple iteration converge, so as to get on scale. If the problem never exhibits IA-stiffness, then the procedure described in this section will never call for the formation of a Jacobian and its attendant costs of storage and linear algebraic computations. Thus, if the problem is unequivocally nonstiff, it will be solved using simple iteration and algorithmic tactics appropriate to the type. If the problem is unequivocally stiff, a switch to a simplified Newton iteration will be made just as soon as the IA-stiffness is evident.

5. Changing Formulas. There is no need to use the same formula for both stiff and nonstiff portions of the integration. What we want in a formula is a bit different in the two cases. A-stability is important to our way of recognizing stiffness when solving nonstiff problems, but damping at infinity is not important except when solving stiff problems. Accuracy is a critical matter for the solution of nonstiff problems, but it is of secondary importance for the solution of stiff problems. Changing formula at the same time one changes iteration method offers interesting possibilities for improved performance. To consider changing from one formula to another, we must be able to relate the truncation errors and the restrictions due to convergence of simple iteration for the two formulas so as to alter the step size appropriately. These questions are not germane to this paper and we are not going to take them up in general. However, to show what might be done, we will sketch an interesting possibility.

Few people would disagree that if one is working at order two, the trapezoidal rule (AM2) is a very attractive formula for nonstiff problems and the BDF2 a very attractive formula for stiff problems. The AM2 is not strongly damped at infinity as the BDF2 is and so is not nearly as suitable for solving stiff problems. On the other hand, it is considerably more suitable for solving nonstiff problems. The truncation errors of the AM2 and the BDF2 are

$$\frac{1}{12}h^3|y^{(3)}(\xi)| \quad \text{and} \quad \frac{1}{3}h^3|y^{(3)}(\nu)|, \tag{1}$$

respectively. This means that the AM2 could achieve the same accuracy as the BDF2 with a step size bigger by a factor of about 1.59. Furthermore, the conditions for the convergence of simple iteration are

$$\frac{1}{2}\|hJ\| < 1 \quad \text{and} \quad \frac{2}{3}\|hJ\| < 1, \tag{2}$$

respectively. This means that simple iteration can be used with the AM2 at a step size bigger by a factor of about 1.33.

We propose that one use the AM2 when simple iteration is feasible and the BDF2 when it is not. Because of the extremely simple relationships (1) and (2), we can at any time understand the effects of a change of formula and select an appropriate step size. It is easy to implement the two formulas in virtually identical fashion so that a change is very easy; compare, for example, the implementation of both in the DIFSUB code [4].

It is worth noting that Klopfenstein [13] has derived a second order formula, which we shall call K2, enjoying all the aforementioned properties of the BDF2. The condition for convergence of simple iteration is

$$\frac{3}{5}\|hJ\| < 1,$$

and the truncation error is

$$\frac{1}{6}h^3|y^{(3)}(\zeta)|.$$

The former is slightly better than the BDF2 and the latter is significantly better. We would implement K2 rather than the BDF2 in a type-insensitive code.

A code along the lines sketched would satisfy remarkably well the attributes one would hope for in a type-insensitive code. It is a pity that the order is only two, but there are important areas in which this would suffice.

6. Some Applications. There is a considerable variety of formulas and procedures to which our ideas apply. We shall cite here some effective codes for stiff problems which, in principle, could be altered easily to make them type-insensitive. Unfortunately, a change to an existing code which is simple in concept is often surprisingly difficult in practice. Providing an alternative iteration method is only part of the task. To properly solve nonstiff problems, some basic tactics, such as those for adjustment of step size, must be different from those for solving stiff problems. Providing an alternative set of tactics implies a substantial software development effort.

We shall not mention any specific one of the many codes based on the multistep formulas we have given as examples–BDF1, BDF2, AM2. Klopfenstein has a code STIFEQ which implements the interesting formula K2 described in Section 5. It needs no further comment.

Hulme and Daniel [14] have implemented two families of fully implicit Runge-Kutta formulas in COLODE. The Legendre family is A-stable and the Radau family strongly A-stable. This code makes several Jacobian evaluations in each step, the number depending on the order of the formula selected. It estimates the local error by doubling, meaning that two steps of length h are taken and compared to one of length $2h$. This implies two matrix factorizations at each step. As Hulme and Daniel properly point out, this code is extremely expensive in terms of Jacobian evaluations, storage, and overhead when applied to a nonstiff problem. Our ideas do away with all this expense and so make the code of acceptable efficiency for a nonstiff problem. Indeed, because Jacobian evaluations are made at every step, the decision about switching to simple iteration is sharper than it is in many codes.

Alexander [15] has implemented a number of semi-implicit Runge-Kutta methods involving one to three stages. All the formulas are at least A-stable. The local error is estimated by doubling. It is clear that using simple iteration, when feasible, greatly reduces the cost. Norsett has derived a second order semi-implicit formula with internal error estimate. It has been implemented nicely so as to take account of sparse Jacobians by Houbak and Thomsen [16]. The formula has three stages with an overlap of one stage into the next step if the step is a success and the next step is of the same size. The formula is L-acceptable, hence our ideas apply. The gain is not as dramatic as in Alexander's code because of the more efficient error estimation scheme, but it is still just what is needed to make the formula practical for nonstiff problems.

The idea of extrapolation is to solve a problem twice (or more) with the same formula using different step sizes and to combine the results to get a higher order result. Lindberg does this with the modified midpoint rule in his code IMPEX 2 [17]. When simple iteration is feasible it is very advantageous because it avoids two matrix factorizations coming from the various integrations. Extrapolation has been rather successful for the solution of nonstiff problems, so it is possible that a type-insensitive version of IMPEX 2 might be pretty competitive for such problems.

Defect correction methods resemble extrapolation in some respects. More than one integration is done, but the problem is altered rather than the step size. Ueberhuber [18] does the integrations with the backward Euler method. Because of the low order of the basic formula, several integrations are normally done. Recognizing when simple iteration is feasible is of obvious importance to make the method practical for nonstiff problems.

The examples cited show that quite a variety of effective codes for stiff problems could be altered to make them type-insensitive. It is not claimed that they would compete with the best codes for nonstiff problems, but they would be practical. Their performance on stiff problems would be improved, as they take advantage of a change of type in the course of the integration. It would be worth a lot to many users to have just one code for all their problems which would be efficient for expensive (stiff) problems and be of acceptable efficiency for relatively inexpensive (nonstiff) problems.

Numerical Mathematics Division 5642
Sandia National Laboratories
Albuquerque, New Mexico 87185

1. M. R. Scott & H. A. Watts, "A systemized collection of codes for solving two-point boundary-value problems," in *Numerical Methods for Differential Systems* (L. Lapidus and W. Schiesser, Eds.), Academic Press, New York, 1976, pp. 197–227.

2. A. E. Rodrigues & E. C. Beira, "Staged approach of percolation processes," *AIChE J.*, v. 25, 1979, pp. 416–423.

3. L. F. Shampine, "Implementation of implicit formulas for the solution of ODEs," *SIAM J. Sci. Stat. Comput.*, v. 1, 1980, pp. 103–118.

4. C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, N. J., 1971.

5. J. M. Ortega & W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

6. L. F. SHAMPINE, M. K. GORDON & J. A. WISNIEWSKI, "Variable order Runge-Kutta codes," in *Computational Techniques for Ordinary Differential Equations* (I. Gladwell and D. K. Sayers, Eds.), Academic Press, London, 1980.

7. L. F. SHAMPINE, "Lipschitz constants and robust ODE codes," in *Computational Methods in Nonlinear Mechanics* (J. T. Oden, Ed.), North-Holland, Amsterdam, 1980.

8. L. F. SHAMPINE, "Stiffness and non-stiff differential equation solvers, II: Detecting stiffness with Runge-Kutta methods," *ACM Trans. Math. Software*, v. 3, 1977, pp. 44–53.

9. L. F. SHAMPINE & K. L. Hiebert, "Detecting stiffness with the Fehlberg (4, 5) formulas," *Comp. & Maths. with Appls.*, v. 3, 1977, pp. 41–46.

10. L. F. SHAMPINE & M. K. GORDON, *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, Freeman, San Francisco, 1975.

11. L. F. SHAMPINE, "Evaluation of implicit formulas for the solution of ODEs," *BIT*, v. 19, 1979, pp. 495–502.

12. J. WILLIAMS, *The Problem of Implicit Formulas in Numerical Methods for Stiff Differential Equations*, Rep. 40, Dept. of Math., Univ. of Manchester, Manchester, England, 1979.

13. R. W. KLOPFENSTEIN, "Numerical differentiation formulas for stiff systems of ordinary differential equations," *RCA Rev.*, v. 32, 1971, pp. 447–462.

14. B. L. HULME & S. L. DANIEL, *COLODE: A Colocation Subroutine for Ordinary Differential Equations*, Rep. SAND74-0380, Sandia Laboratories, Albuquerque, N. M., 1974.

15. R. ALEXANDER, "Diagonally implicit Runge-Kutta methods for stiff O.D.E.'s," *SIAM J. Numer. Anal.*, v. 6, 1977, pp. 1006–1021.

16. N. HOUBAK & P. G. THOMSEN, *SPARKS, a FORTRAN Subroutine for the Solution of Large Systems of Stiff ODE's with Sparse Jacobians*, Rep. NI-79-02, Inst. for Numer. Anal., Tech. Univ. of Denmark, Lyngby, Denmark, 1979.

17. B. LINDBERG, *IMPEX 2, a Procedure for Solution of Systems of Stiff Differential Equations*, Rep. TRITA-NA-7303, Dept. of Inform. Processing, Royal Inst. of Tech., Stockholm, Sweden, 1973.

18. C. W. UEBERHUBER, "Implementation of defect correction methods for stiff differential equations," *Computing*, v. 23, 1979, pp. 205–232.