BIT 24 (1984), 484–502

MULTIRATE LINEAR MULTISTEP METHODS[1]

C. W. GEAR and D. R. WELLS[2]

Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, Illinois 61801, U.S.A.

IBM Corporation DSC Division Neighborhood Road Kingston, NY 12401, U.S.A.

Dedicated to Professor Germund Dahlquist on the occasion of his 60th birthday

Abstract

The design of a code which uses different stepsizes for different components of a system of ordinary differential equations is discussed. Methods are suggested which achieve moderate efficiency for problems having some components with a much slower rate of variation than others. Techniques for estimating errors in the different components are analyzed and applied to automatic stepsize and order control. Difficulties, absent from non-multirate methods, arise in the automatic selection of stepsizes, leading to a suggested organization of the code that is counter-intuitive. An experimental code and some initial experiments are described.

1. Introduction.

The principal objective of a multirate method is to reduce the integration time by using larger stepsizes for those variables in a system that have a behavior which is slow compared to the fastest variables. The ordinary differential equation system

$$y' = f(y,t), \qquad y(0) = y_0 \tag{1.1}$$

where $f \in \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$ may be partitioned into the M coupled systems

$$y_i' = f_i(y_j, t), \qquad y_i(0) = y_{i0} \tag{1.2}$$

where $f_i \in \mathbb{R}^N \times \mathbb{R}^{N_i}$, $1 \leqq i \leqq M$, $\sum N_i = N$, and the components of y_i are all to be treated with the same stepsize. For expository purposes in this paper, we will discuss the cases $M = 2$ and $M = 3$ using the systems

$$\begin{aligned} y' &= b(y,z,t) \\ z' &= c(y,z,t) \end{aligned} \tag{1.3}$$

[1] Supported in part by the Department of Energy under grant DOE DEAC0276ERO2383.

[2] Work done while attending the University of Illinois.

Received December 1983. Revised May 1984.

and

$$x' = a(x, y, z, t)$$
$$y' = b(x, y, z, t)$$
$$z' = c(x, y, z, t)$$

respectively, where letters later in the alphabet represent faster components (thus, z is faster than y). The results extend to the general case.

A system could be partitioned in this way to permit different methods to be used for different components. This has been examined in [1] and [5]. Different methods might be used to improve the stability properties of the combined method, to handle different continuity properties of different components, or for other reasons such as the availability of higher derivatives of some but not all components. However, our purpose in partitioning is to use as large a stepsize as possible for each component to gain efficiency. We will restrict ourselves to the use of the same general type of method for all components (namely, a linear multistep method), although the order and choice of stiff (BDF) versus nonstiff (Adams) formulas is component dependent.

The objective of a modern code is to select automatically as many of the parameters of the method as possible, making the code available to users in other areas of expertise. The use of multirate methods increases the number of parameter choices. There are the usual choices of stepsize and order for each component. These choices can be made automatically using extensions of well-known techniques, although it is the automatic choice of stepsize which leads to the greatest difficulties in multirate methods. An approach to the resolution of these difficulties is the principal topic of this paper. There is the choice of a stiff versus nonstiff method for each component. In the experimental code we discuss later, this choice is not made by the program, although the techniques being used by various authors (for example [6] and [7]) could be added without difficulty. There is the choice of which components are fast and which are slow. For example, if a code were presented with the system (1.3), it could attempt to determine automatically which component is fast and which is slow. We have not attempted to do this in the code reported here for reasons which will become evident in later discussion. However, it does not appear to present an insurmountable difficulty. Finally, there is the choice of partitioning into subsystems. This can either be done statically before the start of the integration or dynamically during the integration. Neither has been attempted automatically for several reasons. Multirate methods introduce a fair amount of inefficiency at the code level. Dynamic repartitioning would increase this inefficiency considerably. Static partitioning could be done only on the basis of initial values, which may give a false picture of the behavior over the bulk of the interval. Hence it is of little value unless done by the user who has additional knowledge of the physical system not available to the program prior to the integration. (Dynamic partitioning was tried in [4], but the cost was high. The subject is worth further exploration.)

The work reported here represents some first, tentative steps towards an automatic multirate method. Systems considered are assumed to be either nonstiff, or have their stiffness isolated in the separate components in the sense that components of eigenvectors corresponding to large eigenvalues are negligible except in single components of the system (1.2). Currently we do not know how to handle systems which violate this restriction.

2. Efficiency considerations.

If there were no coupling between different components in (1.2), their integrations could proceed independently. The size of the coupling affects the errors and stability of the methods, and its existence causes the bulk of the inefficiencies in the methods because of the need to know the value of one variable at a mesh point of another.

When system (1.3) is being integrated by a multirate method using stepsizes $H > h$ for y and z (the slow and fast components, respectively), the integration method for z will require the evaluation of $c(y, z, t)$ at values of t which are not mesh points in the y integration. Hence, the value of y will have to be approximated by interpolation from mesh values of y. This process will cost approximately the same amount of computer time as the prediction step in a predictor-corrector process because the prediction process is a linear combination of past values. However, y does not have to be corrected at these points, so the time for the correction of y, which is almost entirely due to evaluations of $b(y, z, t)$, can be saved in a multirate method. Unless this cost is significant compared to the predictor arithmetic, little can be gained from multirate methods. (If the problem is sparse, additional savings are possible because not all variables habe to be interpolated. See [2] for further analysis. The code discussed later allows the user to specify inter-component independencies to exploit the sparsity.)

If y is integrated on a mesh that is not a subset of the meshpoints of z, values of z will also have to be interpolated when $b(y, z, t)$ is evaluated. For this reason, *synchronization* of the meshes is desirable. If the mesh used for any component is a subset of the meshes used for all faster components, the stepsizes are said to be synchronized, and there will never be unnecessary interpolations. Achieving this synchronization in an automatic stepsize selection scheme without causing sharp reductions in stepsizes for some components is difficult. Such sharp reductions are known to cause inefficiency because errors introduced in the reduction cause a number of small steps to be taken before the stepsize can be increased again. For this reason we have chosen to limit stepsize changes to halving and doubling or powers of the same: a step may be halved at any time, but it may only be doubled when $(t-t_0)/h$ is even, where h is the current stepsize. (This scheme guarantees that $(t-t_0)/h$ is an integer and is a measure of the length of integration covered so far in units of the current stepsize.)

This scheme could be generalized to permit the stepsize of a component to be an integer multiple of the stepsize of the next faster component. Whereas the halving/doubling scheme described above maintains the required relationship (which generates synchronization) automatically, relaxing to any integer multiple could cause other inefficiences. For example, suppose system (1.4) is being integrated, and the stepsizes used for x, y, and z are currently r_1r_2h, r_2h, and h, respectively, where the r_i are integers. If we wish to change the stepsize for the y integration, we are limited to values r_3h where r_3 is a divisor of r_1r_2 unless we also change the stepsizes for the integrations of x and/or z. Since r_1r_2 may have no divisors other than r_1, r_2, and 1, we may not be able to reduce the step to other than h without changing other stepsizes.

Synchronization reduces inefficiencies by reducing unnecessary interpolations, but it does not avoid integrating one component before another. At any point in the integration process, the "simulated time" (the value of the independent variable t) for one component may be ahead of that of another. An automatic code uses some strategy for selection of the next component to be integrated and its stepsize. The objective of the selection strategy is to make sure that it is possible to interpolate for the needed values of other components at tolerable error levels. Since automatic stepsize control methods do not provide a final value for a stepsize until the corresponding step has been taken, the component selection strategy must make its decision in the absence of definitive information about the sizes of steps yet to be taken. Although most integration methods estimate the stepsize to be used for the next step somewhat conservatively so that there is a high probability that the step will be successful, the component selection strategy must allow for the possibility of rejection of an attempted step and subsequent reduction of stepsize. This means that the mesh points at which interpolation will be necessary are not known in advance.

The first approach tried for component selection (see [2]) used the intuitively appealing idea of *events*. Each mesh point is viewed as an event in time, and as in discrete simulation, a single global time is advanced to the next event. Thus, if system (1.3) were being integrated with stepsizes H and h starting from $t = t_0$ in both components, and $H = rh$, z would be integrated over $r-1$ steps of size h and then y and z would be simultaneously integrated over steps of H and h, respectively. This would advance both components to $t = t_0+H$. During the first $r-1$ integration steps for z, approximate values of y at $t_n = t_0+nh$, $1 \leqq n \leqq r-1$ would have to be *extrapolated* from prior values of y. This can be done with a predictor-like formula. Since the usual accuracy requirements on a multistep method require that the predictor-corrector difference be small and we are extrapolating over an interval *smaller* than the integration stepsize H to be used for y, the extrapolation error should be of tolerable size. (Note that the availability of extrapolation from past values is an advantage for multistep methods over Runge-Kutta methods in the multirate context.)

If the stepsizes were known in advance, the technique above, called the *fastest-first* method, seems to be the most appropriate. However, automatic stepsize control causes great difficulty because the stepsizes are not known in advance, as can be seen in a simple example: Suppose the stepsize selection algorithm has chosen a size H for y and $h < H$ for z at $t = t_0$. A fastest-first method will integrate z forward until it is about to reach t_0+H. At that time, y will also be integrated over one step. If that integration fails, H must be reduced. Now the integration of y requires values of z at times before $t+H$. However, the earlier z values have already been discarded and the value of z retained at $t+H$ is based on integrations using predicted values of y that are not sufficiently accurate. Therefore, it is necessary to provide some mechanism for backing z up to an earlier time. Reverse integration is out of the question (it may be unstable and is time consuming). Saving all values of z is not attractive because of space, and saving just one prior value of z at the last meshpoint to which all variables have been integrated implies a large computation cost for some step failures in multicomponent systems.

For these reasons we have abandoned the fastest-first method for automatic integration and changed to the *slowest-first* method. In this technique the following rules for selecting the next component to be integrated are used:

(1) The next component to be integrated over one step is one of those whose simulated time is minimum (that is, has been integrated over the shortest total interval).

(2) Of all components satisfying (1), the component with the largest stepsize is integrated next.

(3) If an integration step is rejected, the stepsize is reduced (by a power of two) and the rules are applied again.

For example, if system (1.4) is integrated with stepsizes $4h$, $2h$ and h, respectively, from t_0 to $t_4 = t_0+4h$, the order of computation of variable values would be

$$x_4,\ y_2,\ z_1,\ z_2,\ y_4,\ z_3,\ z_4.$$

In this scheme, the extrapolation of the fast variables to integrate the slow variables first will lead to large errors in the extrapolated values because the extrapolation is over many time steps in the fast variables. However, this is mitigated by the fact that the coupling from the fast values to the slow values is generally small. Thus, if in system (1.3) $\partial b/\partial z$ is large and z is rapidly changing, b is usually rapidly changing so y is not slow. (This is not always true along particular solutions, as can be seen by the example

$$y' = -y+\lambda(z-\sin \omega t)$$
$$z' = \omega \cos \omega t$$

where λ, the coupling, and ω, the "speed" of z, are large.)

The important property of the slowest-first method is that if a variable has to be backed up because of an integration failure in another variable. the backup is simply a reduction of the size of the last step taken, and can be done provided that one additional value is kept for all variables. Direct application of the three rules given above for the determination of which component to integrate next does not lead to any backup of steps already completed. Backup can only occur when we attempt to control the error in the extrapolation process. When a large step in a slow component is taken, an assumption (based on previous values) is made about the size of the extrapolation error. Coupled with a bound on the size of the coupling (due to off-diagonal blocks of the Jacobian such as $\partial b/\partial z$), this gives an estimate of the error introduced into the slow components (see the next section for the analysis). If the later integration of the fast component reveals that the extrapolation error is larger than expected, the slow steps can be repeated with a smallest stepsize. As long as the detection of the excess extrapolation error is done during the first step at which it occurs in the faster components, the slow components do not have to be backed up over more than the current step.

3. Convergence and error analysis.

The analysis of constant stepsize methods is straightforward but notationally tedious, so we will simply describe the results here. An analysis for the slowest-first method is given in [8]; the fastest-first analysis is similar.

In a constant stepsize model we assume that the stepsize of the ith component is $r_i h$ where the r_i are fixed integers such that r_i is divisible by r_{i+1} and the smallest stepsize, h, is a parameter. The standard questions concern the convergence of errors over a finite interval and its rate as $h \to 0$. Naturally, the method associated with each component (which may be different) must be zero stable and consistent – which for fixed coefficient multistep methods means an order of at least one. The interpolation used must have errors no larger than $o(1)$, so it is natural to call methods satisfying this condition *consistent* interpolations. For example, use of the most recently computed value is adequate, since it is $O(h)$. These three conditions: stability and consistency of the integration and consistency of the interpolation formula are necessary and sufficient for convergence if the problem is Lipschitz continuous. The necessity of the first two conditions follows directly from their necessity for a method for a single equation [3]. The necessity for the third condition follows by assuming otherwise: if we integrate the equation $\tilde{z}' = c(\hat{y}, \tilde{z}, t)$ setting $\hat{y}(t) = y(t)+d$ where y is the correct solution of system (1.3), it is clear that $\tilde{z}$ differs from the solution z of system (1.3) by an amount independent of h. Sufficiency follows from an extension of the usual argument. Note that interpolation errors in approximating y by $\hat{y}$ of size $o(1)$ over a finite interval contribute errors to the integration of $z' = c(\hat{y}, z, t)$ of size no greater than $o(1)$ provided that $\partial c/\partial y$ is bounded: this is the reason that the local error in the interpolation can be one order lower than that in the integration formula.

As usual the convergence result can be sharpened to show that if the minimum order of accuracy of the integration formulas is p (local errors $O(h^{p+1})$) and of the interpolation formula is q (local errors $O(h^q)$) then the global error for suitably differentiable problems is $O(h^{\min(p,q)})$.

Of more practical importance is the estimation and control of errors in a code. As with most codes, we attempt to control the local error. The local errors in a slowest-first multirate integration come from three sources: the truncation error associated with the linear multistep methods used, the interpolation of slower components and the extrapolation of the fast components. To illustrate this consider the system (1.4) of three classes x, y, and z which are of slow, medium and fast rates, respectively. We will assume that corrector equations are solved exactly.

At a y integration point, t_n, the computed value of y, satisfies an equation of the form

$$y_n = H\beta_0 b(\hat{x}_n, y_n, \bar{z}_n) + \Sigma$$

where H is the stepsize used by y, $\hat{x}_n$ is an interpolated value, $\bar{z}_n$ is an extrapolated value, and Σ is a linear sum of past y and y' values.

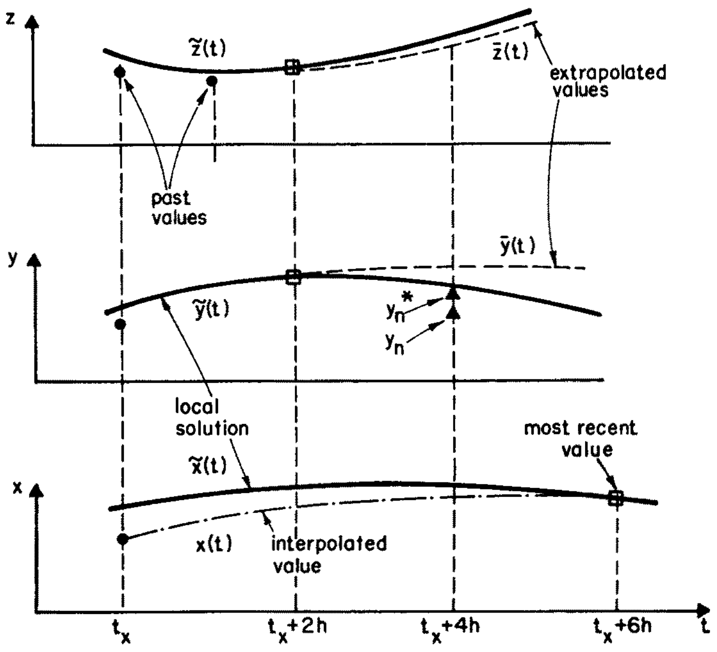

Fig. 1. Local solutions, interpolants and extrapolants.

Let $\tilde{x}(t)$, $\tilde{y}(t)$, and $\tilde{z}(t)$ be the components of the *local* analytic solution through the most recently computed values of x, y, and z, and let y_n^* be defined by

$$y_n^* = H\beta_0 b(\tilde{x}(t_n), y_n^*, \tilde{z}(t_n)) + \Sigma.$$

The functions $\tilde{x}$, $\tilde{y}$, and $\tilde{z}$ are actually solutions to a multipoint boundary value problem, as shown in Figure 1, where the most recent values of x, y, and z have been computed at t_x+6h, t_x+2h, and t_x+2h, respectively. It is true that if the computed values lie too far from the solution through the last set of points at which all components were synchronized (t_x in the example), there may not be a solution to this boundary value problem, but if the errors and couplings between components are small, a solution will exist.

From the point of view that $\tilde{x}, \tilde{y}, \tilde{z}$ represents our best approximation so far, it makes sense to define additional errors in terms of those components, so let the local truncation error, d_n, associated with the linear multistep method be $d_n = \tilde{y}(t_n) - y_n^*$ and the local error, e_n, of the multirate method be $e_n = \tilde{y}(t_n) - y_n$. Now

$$e_n = \tilde{y}(t_n) - y_n^* + y_n^* - y_n = d_n + y_n^* - y_n \quad \text{and} \tag{3.1}$$

$$\begin{aligned} y_n^* - y_n &= H\beta_0[b(\tilde{x}(t_n), y_n^*, \tilde{z}(t_n)) - b(\hat{x}_n, y_n, \bar{z}_n)] \\ &= H\beta_0[b_x\delta_x + b_y(y_n^* - y_n) + b_z\delta_z] \end{aligned}$$

where the partials b_x, b_y, and b_z are evaluated at appropriate points, $\delta_x = \tilde{x}(t_n) - \hat{x}_n$, and $\delta_z = \tilde{z}(t_n) - \bar{z}_n$. Hence,

$$y_n^* - y_n = (I - H\beta_0 b_y)^{-1} H\beta_0(b_x\delta_x + b_z\delta_z). \tag{3.2}$$

Combining this with (3.1) yields

$$e_n = d_n + (I - H\beta_0 b_y)^{-1} H\beta_0(b_x\delta_x + b_z\delta_z) \tag{3.3}$$

and thus e_n consists of a truncation error term and interpolation and extrapolation error terms as claimed.

To control the local error we proceed as follows:

First we try to ensure that the contributions from the interpolation and extrapolation are small in comparison to the truncation error term, i.e., the expression in (3.2) is small in comparison to d_n.

Next we estimate the truncation error by Milne's device and select H so that this estimate is smaller than a given error tolerance, ε. To justify this approach let $\bar{y}_n$ be the predictor for y_n. For an appropriate constant, C_n, we assume that

$C_n(y_n^* - \bar{y}_n)$ is an acceptable estimate of d_n. (This is as justified as it is in any multistep code! It or a minor variant is used in virtually all.) Now if (3.2) is small, then y_n is approximately equal to y_n^*, hence $C_n(y_n - \bar{y}_n)$ is an acceptable estimate of d_n.

Next we consider how to control the size of (3.2). Suppose the error in the last x integration step is bounded by ε_1 where ε_1 is proportional to ε. Since we are interpolating between the last two values of x, the $\hat{x}$ interpolation error, δ_x, satisfies* $\|\delta_x\| \leqq K\varepsilon_1$ where K depends on the interpolation method, but is typically no larger than 2 because the interpolation error is due to two effects: the errors in the interpolation formula which are significantly less than an extrapolation formula over the same interval, which in turn are a small multiple of ε_1, and errors due to errors in the mesh values, also of order ε_1. The contribution of the interpolation error to (3.2) is then bounded by

$$KH\beta_0\|(I - H\beta_0 b_y)^{-1}\| \, \|b_x\|\varepsilon_1.$$

The strategy is then to choose ε_1 to satisfy

$$H\|b_x\|\varepsilon_1 \leqq \alpha\varepsilon \tag{3.4}$$

where $\alpha K\beta_0$ is small (e.g., 0.1), so that the contribution to the local error in y due to the interpolation is bounded by

$$\alpha K\beta_0\|(I - H\beta_0 b_y)^{-1}\|\varepsilon.$$

Equation (3.4) implies that an upper bound for $H\|b_x\|$ be known before ε_1 is chosen. The restriction on the size of $H\|b_x\|$ is natural for absolute stability requirements discussed in the next section.

There are several possible ways to control the fast extrapolation error, none of which is entirely satisfactory. One way is to use the fact that

$$\delta_z = \Gamma H^{p+1} z^{(p+1)}(t_n) + O(H^{p+2})$$

for some Γ which depends on the extrapolation method, with p the order of the z integration. This can be estimated prior to the slow step by using the estimate

* Mathematically, the statements in this paragraph are not true unless a number of additional hypotheses are added because the automatic stepsize control mechanisms used in the underlying codes guarantee nothing absolutely without additional hypotheses. It is well known that any automatic code for ODES that operates entirely by numerical evaluation of the right-hand side of $y' = f(y,t)$ can be made to fail by appropriate choice of f (even for a polynomial $f(y,t) = P(t)$). Therefore, these statements have to be interpreted in the sense of "most of the time." Perhaps we need a concept of "Robust Mathematical Analysis" in the sense that we talk about "Robust Mathematical Software" – software that fails only in cases which are unusual from the user's point of view.

of the z error at time t_n. If the estimate is too large, H can be reduced accordingly. However, the stepsize used on z is probably much smaller than H and the $O(H^{p+2})$ term in the expression above is likely to be the dominant one, so the estimate of $z^{(p+1)}$ is of no use (unless we are working to very high accuracy so H is small.) Furthermore, we would like to verify the estimate of the extrapolation error after z has been integrated.

For a second approach we try to estimate the extrapolation error after z has been integrated. A direct way to do this is to evaluate

$$b(\hat{x}_n, y_n, z_n) - b(\hat{x}_n, y_n, \bar{z}_n) \tag{3.5}$$

after z_n has been computed. By this time it is too late to reduce the H for the y integration (because z would have to be backed up) so it provides an *a posteriori* error estimate which can be used only to control the next step. For simplicity, the code described in section 5 uses an approach of this form described below, although there is an alternate technique discussed later in this section which allows reduction of the y integration step.

This second error estimation approach uses an extra function evaluation for (3.5) which can be avoided since the estimate is used only to control the next stepsize, H, in y. We can illustrate this for a forward Euler/backward Euler predictor-corrector pair using their difference as an error estimate. When the corrected value of y_n is computed, a value of $b(\hat{x}_n, y_n^*, \bar{z}_n)$ is also computed (where y_n^* depends on the technique used to evaluate the corrector). When the predicted $\bar{y}_{n+r}$ is computed, this value of b is used to get

$$\bar{y}_{n+r} = y_n + Hb(\hat{x}_n, y_n^*, \bar{z}_n).$$

The corrected value, y_{n+r}, uses values of $b(\hat{x}_{n+r}, y_{n+r}^*, \bar{z}_{n+r})$ where $\bar{z}_{n+r}$ is an extrapolated value of z based on z_n, not on $\bar{z}_n$, because before this step for y is computed, z is integrated to t_n. Consequently, the error estimate for y in the step from t_n to t_{n+r} can be written as

$$\begin{aligned} e_y &= y_{n+r} - \bar{y}_{n+r} \\ &= Hb_x(\hat{x}_{n+r} - \hat{x}_n) + Hb_y(y_{n+r}^* - y_n^*) + Hb_z(\bar{z}_{n+r} - \bar{z}_n) \\ &= H^2 y_n'' + Hb_x(\hat{x}_{n+r} - \hat{x}_n - Hx_n') \\ &\quad + Hb_y(y_{n+r}^* - y_n^* - Hy_n') \\ &\quad + Hb_z(\bar{z}_{n+r} - \bar{z}_n - Hz_n'). \end{aligned} \tag{3.6}$$

The first term is a multiple of the local truncation error in the integration formula. The second and third terms will be no worse than $O(H^2)$ for reasonable formulas for $\hat{x}$ and y^*. The fourth term can be rewritten as

$$Hb_z(\bar{z}_{n+r}-z_n-Hz_n')+Hb_z(z_n-\bar{z}_n). \tag{3.7}$$

The first term in (3.7) will also be $O(H^2)$ for reasonable extrapolation formulas, while the second is $O(H)$ because $(z_n-\bar{z}_n)$ is now fixed. If the value of z_n is a reasonable approximation to $\tilde{z}_n$ (z_n should be much closer to $\tilde{z}_n$ than to $\bar{z}_n$), the second term in (3.7) is close to the term $Hb_z\delta_z$ in (3.3). Hence, control of e_y by use of the estimate (3.6) will cause a reduction of H so that the term $Hb_z\delta_z$ is small on the next step – unfortunately too late to undo any damage, but, for slowly changing functions, satisfactory when high accuracy is requested.

A third approach is to monitor the extrapolation error in $\bar{z}_n$ as z is advanced from t_{n-r} to t_n. When y_n was computed, it was hypothesized that the error δ_z in $\bar{z}_n$ was such that

$$\|[I-H\beta_0 b_y]^{-1}\|H\beta_0\|b_z\delta_z\| \tag{3.8}$$

was small compared to ε, the local error in each y step. If this had not been the case, we should reduce H which would reduce (3.8) in two major ways: directly because of the multiplicative factor H and via δ_z which would reduce as the interval of extrapolation was reduced. Therefore, at each step t_{n-i}, $i = r-1, r-2, \ldots, 0$ as z is advanced from t_{n-r} to t_n, we can estimate the size of

$$(t_{n-i}t_{n-r})\beta_0\|\delta_z^i\| \tag{3.9}$$

where δ_z^i is the extrapolation error that would occur if z were extrapolated from t_{n-r} to t_{n-i}. The latter can be calculated directly as $\bar{z}_{n-i}-z_{n-i}$ at the cost of an additional extrapolation. When we have estimated (3.9), we can combine it with an estimate of $\|b_z\|$ (either user-supplied or periodically calculated) to estimate the size of (3.8) if y had been integrated to t_{n-i} rather than to t_n. If this estimate exceeds the desired size, the last step for z can be rejected and the last step for y can be reduced to bring y to t_{n-i-1}. This affects no components faster than z because they have yet to be integrated past t_{n-i-1}, but does require a reduction of stepsize for any components between y and z in speed.

The third approach has not been coded and is almost certainly more expensive because of the additional estimates and the need for $\|b_z\|$ values, but should be more reliable for larger integration tolerances than the second approach.

4. Absolute stability.

The linear stability analysis of most methods is straightforward and can be understood by studying the scalar test equation $y' = \lambda y$ because the transformation $Sy \to z$ that takes the linear equation $y' = Ay$ to the Jacobian form $z' = Jz$ where $J = SAS^{-1}$ can be applied directly to the numerical method to show that if y_n is computed when the method is applied to $y' = Ay$, then the value computed when the method is applied to $z' = Jz$ satisfies $z_n = Qy_n$. The reason that the same transformation reduces both the differential and numerical operators to a triangular form is that when exactly the same method is applied to all components (same stepsize, order, etc.), the numerical operator corresponding to $y' = Ay$ is $y_n = R(hA)y_{n-1}$ where R is a rational function. Hence, $R(A)$ is diagonalized (triangularized) by any transformation which diagonalizes (triangularizes) A. Therefore, apart from minor differences in the coupling between terms when J is not diagonal, it suffices to consider the scalar equation $y' = \lambda y$, where λ is an eigenvalue of A.

This is not true in multirate methods, or in any method which does not treat all variables identically. We can see this by looking at a simple case: integrate the system

$$\begin{cases} y' = Ay + Bz \\ z' = Cy + Dz \end{cases} \tag{4.1}$$

by forward Euler in the fastest-first mode, using stepsizes h for z and $2h$ for y. If we compute only y_{2n} and set $\bar{y}_{2n+1} = y_{2n}$ (a consistent interpolant) we get

$$z_{2n+1} = z_{2n} + hCy_{2n} + hDz_{2n}$$

and

$$z_{2n+2} = z_{2n+1} + hCy_{2n} + hDz_{2n+1}$$

or

$$z_{2n+2} = z_{2n} + (2 + hD)(hCy_{2n} + hDz_{2n})$$

while

$$y_{2n+2} = y_{2n} + 2(hAy_n + hBz_{2n}).$$

Thus,

$$w_{2n+2} = Rw_{2n} \tag{4.2}$$

where $w = (y^T, z^T)^T$ and

$$\begin{bmatrix} I + 2A & 2hB \\ \left(I + \dfrac{hD}{2}\right)2hC & I + \left(I + \dfrac{hD}{2}\right)2hD \end{bmatrix}.$$

This is not a rational function of the original system matrix, and its eigenvalues *cannot* be expressed as functions of the eigenvalues of the original matrix.

Note that if B or C is zero, the eigenvalues of the operator in (4.1) can be split into two groups, those of A and D, and can be associated with the y and z components, respectively. The eigenvalues of the numerical operator in (4.2) can be split similarly, and are the eigenvalues of the one-step forward Euler operator applied to $y' = Ay$, namely $I+2hA$, and the two-step forward Euler operator applied to $z' = Dz$, namely, $(I+hD)^2$.

This result is generalized to multistep methods in [8]. It is evident that if the system is block triangular, where the blocks correspond to the components in system (1.2), then absolute stability of the multirate method is equivalent to simultaneous absolute stability of each method for each component. For example, if an implicit method is used separately for each component so that implicit equations of the form

$$y_n^* = h\beta_0 b(\hat{x}_n, y_n^*, \bar{z}_n) + \Sigma$$

are solved for y_n^* while $\hat{x}_n$ and $\bar{z}_n$ are kept fixed, the combined method would be $A(\alpha)$-stable for a triangular system if each method is $A(\alpha)$-stable. The moment the system is not triangular, any form of stability at infinity is lost because of the extrapolation which brings in off-diagonal blocks in the numerical operator which are polynomial in h rather than rational. These terms cause the norm of the numerical operator to be unbounded as $h \to \infty$ so that it is ultimately unstable.

The best type of result that we have been able to obtain is of the following form:

PROPOSITION 4.1. *If the multirate method applied to the linear system* (4.1) *with the off-diagonal blocks of the Jacobian set to zero has all the eigenvalues of the numerical operator strictly less than one in magnitude for all* $h > 0$, *then there exists a* K *such that the method is absolutely stable if* $h\|B\| < K$. (*Note that* K *depends on the matrices* A, C, *and* D.)

This type of result (which can be obtained by a continuity argument) tells us that if the system matrix is not too far from block triangular, we can expect stability properties close to those of the method used for individual components when h is not too large.

We have already seen that we cannot handle large coupling from the fast-to-slow components in the slowest-first technique, so we expect B to be small. (In fact, we expect $rhB = HB$ to be small.) Hence, in those situations in which our multirate techniques can be applied, the stability properties of the methods can be approximated by the stability properties of the separate components. The code discussed in the next section allows each component separately to be treated as stiff or nonstiff.

It should be realized that any transformation of the system, or any repartitioning, affects absolute stability: if an automatic partitioning could be done, it should probably be based on an "approximate triangularization." (This could be useful in reducing the matrix work for stiff equations even if multirate methods are not used.)

5. Some tests on an experimental code.

An experimental multirate code, MRATE, based on DIFSUB is listed in [8]. It requires the user to provide a partitioned system of equations. Some scalar parameters of DIFSUB (the tolerance and the method flag for stiff or nonstiff methods) are expanded to arrays with one entry for each component. The subroutine also keeps arrays for the stepsize, independent variable, and order. The user provides a function evaluation subroutine which can evaluate the right-hand sides of the ith component (i is a parameter) and an array to indicate which variables appear in the right-hand sides of a given component: the program MRATE performs the extrapolation and interpolation only for those variables actually required in the evaluation of a component in order to save time.

All variables are arranged in a single array with the first N_1 forming the slowest component, the next N_2 forming the next slowest component, and so on. The use provides an array of integers N_1, N_1+N_2, $N_1+N_2+N_3, \ldots$ which is used by the program to locate each component. The fact that members of a component are contiguous permits simple and rapid indexing. Repartitioning would either require reallocation of storage (which is time consuming) or use an indirect pointer for every variable (which is only slightly less time consuming) unless a variable is moved to its adjacent partition. The requirement that components always maintain the same relative speed (i.e., that component A is always slower than component B) is not of importance to the algorithm because stepsize halving and doubling is used, so that, before two components could change in relative speed, they would have equal stepsizes and would be synchronized. At that time, their positions could be switched. This would require a slightly more complicated data structure to keep track of the speed ordering, but as long as the size of each component is not too small, the overhead is reasonable. It is true that with this extension and components of size 1, we would effectively have dynamic repartitioning, but the overhead would negate any advantage of multirate methods in most cases. The history information is kept as a Nordsieck vector because the code is a modification of DIFSUB. If the components are all of significant size (number of variables), a modified divided difference representation would be preferable, but the Nordsieck vector has some advantages for small systems because the coefficients do not have to be recalculated after stepsize changes. (However, with step doubling and halving, a fairly small number of combinations of k past stepsize ratios would cover most cases, and coefficients for these cases could be precalculated and stored.)

The code described in [8] does not require the user to provide a Jacobian (although it does require the dependency matrix which indicates the sparsity structure of the off-diagonal blocks of the Jacobian). This code does not control the errors introduced by coupling in the current step, but requires the user to select the tolerances for each component cognizant of the effects of coupling – that is, so terms of the form $\|b_z\|\varepsilon_z$ are small compared to ε_y where ε_y and ε_z are the tolerances for the y and z components in (1.3).

The comparisons below were made between MRATE and DIFSUB because the codes use the same basic strategies and representation, not because DIFSUB represents a state-of-the-art integrator. Improvements found in modern codes could be added to MRATE in most cases, so we feel that the comparison does serve as an initial test of this multirate approach.

The experiments below are designed to show how the selection of the method options and the error tolerances for the different classes in MRATE affect the computation time. These experiments show that MRATE performs very well on certain problems, but not as well as one might expect on others. All calculations were done on a PRIME 650 and the CPU times are given in seconds.

The test equations are mainly of the form

$$y' = A(y-\phi(t))+\phi'(t), \qquad y(0) = \phi(0), \quad 0 \leqq t \leqq 4, \tag{5.1}$$

with

$$\phi(t) = (\sin(.05t), \cos(.05t), \sin(t), \cos(t), \sin(20t), \cos(20t))^T$$

and

$$A = \begin{bmatrix} -50 & 49 & a & a & a & a \\ 49 & -50 & a & a & a & a \\ b & b & -5 & 4 & a & a \\ b & b & 4 & -5 & a & a \\ b & b & b & b & -1 & 0 \\ b & b & b & b & 0 & -1 \end{bmatrix}.$$

The values of a and b will be varied to show how the size of the couplings between the fast and the slow components affect the performance of MRATE. The partitioning of the system for MRATE is the natural one with (y_1, y_2) the slow component, (y_3, y_4) the medium component, and (y_5, y_6) the fast.

The Jacobians for the components are the 2×2 matrices on the diagonal

$$\begin{pmatrix} -50 & 49 \\ 49 & -50 \end{pmatrix}, \begin{pmatrix} -5 & 4 \\ 4 & -5 \end{pmatrix}, \text{ and } \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

which have eigenvalues -99 and -1, -9 and -1, and -1 and -1, respectively. Thus, considered as systems by themselves, the slow component is stiff, the medium component is barely stiff, and the fast component is not stiff at all.

For the first experiments we take $a = 0$, $b = 1.0$, and run both DIFSUB and MRATE with their various stiffness options. The single step error tolerances are EPS $= K \cdot 10^{-6}$ for DIFSUB, and EPS(1) = EPS(2) = EPS(3) $= K \cdot 10^{-6}$ for MRATE where K is a constant dependent on the experiment that is chosen to make the actual maximum errors reasonably close to 10^{-6} so that a fair comparison of the methods can be made. The results are summarized in Table 5.1. The column headed MF indicates the stiffness option used. For DIFSUB, MF = 0 means a nonstiff option and MF = 1 indicates the stiff option with a user-supplied subroutine to compute the partial derivatives. For MRATE, the entry 110 under MF, for example, means that the first two classes were integrated by stiff methods and the third by a nonstiff method. Jacobians are always computed by numerical differencing with MRATE. The entries in the last column are the number of steps taken during integration for DIFSUB and the number taken by the respective classes in MRATE. Unsuccessful steps are included.

Table 5.1.

CODE	MF	MAX ERROR	CPU TIME	# STEPS
DIFSUB	0	0.975D-06	19	810
	1	0.962D-06	13	597
MRATE	000	0.946D-06	16	627 544 596
	100	0.973D-06	11	18 253 641
	110	0.992D-06	10	16 58 644
	111	0.920D-06	9	22 70 570

From Table 5.1 we can see that the choice of the method option can have a great effect on the performance of the code. With the proper selection of these options the performance of MRATE is considerably better than that of

Table 5.2.

Variable	Analytic Solution	Error	
		MRATE (MF = 110)	DIFSUB (MF = 1)
y_1	−0.19867	0.160D-06	−0.576D-12
y_2	0.98007	0.176D-06	−0.576D-12
y_3	−0.75680	0.166D-06	−0.230D-11
y_4	−0.65364	0.208D-06	−0.221D-11
y_5	−0.99389	−0.665D-06	0.963D-06
y_6	−0.11039	−0.992D-06	0.404D-06

DIFSUB even on this small system. Table 5.2 gives a breakdown by variables of the errors in a run by MRATE and one by DIFSUB.
Note that the errors from the DIFSUB computation vary greatly while those from MRATE are of the same order of magnitude.

As a second experiment we take $a = 0$ and $b = 10.0$ to show how MRATE behaves when the fast components are strongly dependent on the slow. The results are given in Table 5.3.

Table 5.3.

INTEGRATOR	EPS	MAX ERROR	CPU TIME
DIFSUB (MF = 1)	.6D-6	0.962D-06	13
MRATE (MF = 110)	.1D-6 .1D-6 .1D-6	0.162D-05	14
	.1D-7 .1D-6 .2D-5	0.402D-06	10

In this experiment it was necessary to use a very small error tolerance to achieve the desired accuracy in MRATE when all components use the same error tolerance. About the same accuracy can be obtained by using a smaller tolerance on the slower classes to compensate for the size of the couplings. With the last strategy the computation time was reduced significantly.

In the next experiments we fix $b = 1.0$ and vary a to study the effect of a moderate sized dependence of the fast on the slow and a small dependence of the slow on the fast. The values of a are given in the corresponding column of Table 5.4. Again we have taken $\text{EPS} = K \cdot 10^{-6}$ for DIFSUB and $\text{EPS}(1) = \text{EPS}(2) = \text{EPS}(3) = K \cdot 10^{-6}$ for MRATE with K chosen to get a reasonable comparison of the results.

Table 5.4.

a	MRATE (MF = 110)		DIFSUB (MF = 1)	
	CPU TIME	MAX ERROR	CPU TIME	MAX ERROR
0.1D-0	22	0.716D-6	12	0.796D-6
0.1D-1	22	0.799D-6	13	0.954D-6
0.1D-2	20	0.983D-6	13	0.962D-6
0.1D-3	20	0.875D-6	13	0.962D-6

The stiffness options chosen for the integrators were those which gave the lowest CPU time.

As the table shows, MRATE does not perform as well as DIFSUB, but its performance tends to improve as the couplings become smaller while DIFSUB's remains fairly constant. Part of the additional computation time required by

MRATE for these problems is due to the fact that every component is dependent on every other component and thus an interpolation or extrapolation of each variable must be done at every step. Many of these computations were unnecessary with $a = 0$. Also, some error feeds from the fast to the slow and tends to force the slower components to use a smaller stepsize than would be needed if $a = 0$. This is a small system with not a very high percentage of slow variables.

If there is a high enough percentage of slow components, MRATE can perform very well as the following experiment shows. Consider the system

$$y_i' = -10u_i + u_{i-1} + au_n + .1\cos(.1t), \qquad i = 1, 2, \ldots, n-1,$$
$$y_n' = -10u_n + u_{n-1} + 20\cos(20t),$$
$$u_0 \equiv 0, \quad u_i = y_i - \sin(.1t), \qquad i = 1, 2, \ldots, n-1,$$
$$u_n = y_n - \sin(20t),$$
$$y_i(0) = 0, \qquad i = 1, 2, \ldots, n.$$

We integrate this system from $t = 0$ to $t = 4$ and vary the size of a. For MRATE the first 20 components form the slow class and the last component is the fast class. The results are given in Table 5.5.

Table 5.5.

a	MRATE (MF = 00)			DIFSUB (MF = 0)		
	CPU TIME	MAX ERROR	EPS	CPU TIME	MAX ERROR	EPS
.1D 0	14	0.731D-6	.2D-4 .2D-4	21	0.783D-6	.2D-5
.1D-1	12	0.779D-6	.2D-4 .2D-4	20	0.116D-5	.1D-5
.1D-2	12	0.784D-6	.2D-4 .2D-4	20	0.603D-6	.1D-5
.1D-3	12	0.784D-6	.2D-4 .2D-4	21	0.907D-6	.1D-5
0	8	0.639D-6	.4D-4 .4D-4	21	0.894D-6	.1D-5

Thus, MRATE can perform quite well given the right type of problem.

In section 4 it was shown that multirate methods were absolutely stable for systems which were upper block triangular provided that the diagonal matrices in the Jacobian were in the region of absolute stability of the multistep methods used. For the final experiments in this section we take $a = 1.0$ and $b = 0$ to show that MRATE will integrate this type of system, but the performance is not particularly good. The experiment was performed with all of the possible choices for MF and different error tolerances were used on the classes to compensate for the error from the fast feeding into the slow. (If this was not done, the slow correctors had trouble converging). The results are shown in Table 5.6.

Table 5.6.

	MF	CPU TIME	MAX ERROR	EPS		
DIFSUB	0	20	0.173D-5			.10D-5
	1	13	0.963D-6			.60D-6
MRATE	000	18	0.744D-6	.10D-5	.50D-6	.10D-6
	001	25	0.789D-6	.10D-5	.50D-6	.10D-6
	010	19	0.743D-6	.10D-5	.50D-6	.10D-6
	011	25	0.790D-6	.10D-5	.50D-6	.10D-6
	100	23	0.116D-5	.10D-6	.50D-7	.10D-7
	101	25	0.949D-6	.20D-6	.10D-7	.20D-7
	110	25	0.890D-6	.50D-7	.25D-7	.50D-8
	111	18	0.926D-6	.18D-6	.90D-7	.18D-7

6. Conclusion.

In summary, the experiments in the previous section show that care must be used in selecting the method options and the error tolerances when using MRATE. If used correctly MRATE can perform well even on fairly small systems but is best suited for large systems with a high percentage of slow components. The code will not perform well if the slow components are strongly dependent upon the fast. Theory suggests that the improvements will be more noticeable at smaller tolerances because the effect of interpolation and extrapolation errors reduces with stepsize and the asymptotic theory is more appropriate. The major problem in making a code more automatic so that dynamic partitioning is possible seems to be one of software organization.

REFERENCES

1. J. F. Andrus, *Numerical solution of systems of ordinary differential equations separated into subsystems*, SIAM J. Num. Anal. 16(4), 1979, 605–611.
2. C. W. Gear, *Automatic multirate methods for ordinary differential equations*, Proc. IFIP 1980, 717–722. North-Holland Publishing Company.
3. P. Henrici, *Discrete Variable Methods in Ordinary Differential Equations*, John Wiley & Sons, Inc.: New York, 1962.
4. A. Orailoglu, *Software design issues in the implementation of hierarchical, display editors*, Rept. No. UIUCDCS-R-83-1139, Dept. Computer Sci., Univ. Illinois, 1983.
5. O. A. Paulusinski and J. V. Wait, *Simulation methods for combined linear and nonlinear systems*, Simulation 30(3), 1978, 85–94.
6. L. Petzold, *Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations*, SIAM J. Scientific and Statistical Computing 4(1), 1983, 136–148.
7. L. F. Shampine, *Type-insensitive ODE codes based on implicit $A(\alpha)$-stable formulas*, Math. Comp. 39 (159), 1982, 109–124.
8. D. R. Wells, *Multirate linear multistep methods for the solution of systems of ordinary differential equations*, Rept. No. UIUCDCS-R-82-1093, Dept. Computer Sci., Illinois, 1982.