Accurate Parallel Integration of Large Sparse Systems of Differential Equations

Donald J. Estep[1] and Roy D. Williams[2]

Abstract

We describe a MIMD parallel code to solve a general class of ordinary differential equations, with particular emphasis on the large, sparse systems arising from space discretization of systems of parabolic partial differential equations. The main goals of this work are sharp bounds on the accuracy of the computed solution and flexibility of the software.

We discuss the sources of error in solving differential equations, and the resulting constraints on time steps. We also discuss the theory of *a posteriori* error analysis for the Galerkin finite element methods, and its implementation in error control and estimation.

The software is designed in a matrix-free fashion, which enables the solver to effectively tackle large sparse systems with minimal memory consumption and an easy and natural transition to MIMD (distributed memory) parallelism. In addition, there is no need for the choice of a particular representation of a sparse matrix. All memory is dynamically allocated, with a new expandable array object used for archiving results.

The implicit solution of the discrete equations is carried out by replaceable modules: the nonlinear solver module may be a full Newton scheme or a quasi-Newton; these in turn are implemented with a linear solver, for which we have used both a direct solver and QMR, an iterative (Krylov space) method.

Three example computations are presented: the Lorenz system, which has dimension three and the discretized versions of the (partial-differential) bistable equation in one and two dimensions. The Lorenz system demonstrates the quality of the error estimation. The discretized bistable examples provide large sparse systems, and our precise error estimation shows, contrary to standard error estimates, that reliable computation is possible for large times.

1 Introduction

We consider the numerical time integration of systems of ordinary differential equations (ODEs) of the form

$$\begin{cases} \dot{y} = f(y,t), & t > 0, \\ y(0) = y_0 \in I\!R^M. \end{cases} \tag{1}$$

We are particularly interested in large sparse systems, meaning that the Jacobian of f with respect to y is a sparse matrix. Such systems often arise from the space discretization of systems of ordinary and parabolic partial differential equations (PDEs) such as

$$u_t - \nabla \cdot (D \nabla u) = F(u), \tag{2}$$

[1]School of Mathematics, Georgia Institute of Technology, Atlanta, GA 30332, `estep@math.gatech.edu`
[2]Center for Advanced Computing Research, California Institute of Technology, `roy@caltech.edu`

posed on a domain that is a product of a time domain $t > 0$ with a spatial domain $\Omega \in I\!R^d$ of dimension $d = 1$, 2, or 3. Here, u represents a vector of m unknown functions, so $u : I\!R^d \to I\!R^m$; F is the *reaction* term; and D is the *diffusion* term. F is a vector-valued function of u and ∇u as well as space and time and D is a semi-positive definite $m \times m$ matrix-valued function of u and space and time. The value of u is specified at the beginning of the time interval, and by Dirichlet or Neumann boundary conditions on subsets of $\partial\Omega$ as required.

Many physical phenomena are described by systems of reaction-diffusion equations; well-known applications occur in chemistry [20], materials science [22], fluid flow [2], and population dynamics [19]. The interaction between the nonlinear reaction and the diffusion gives rise to interesting behavior such as finite time blowup, long time behavior such as metastability, and localized behavior such as fast transients, steep fronts, and pattern formation. The variety of applications and interesting solutions have excited much interest in the engineering and mathematical communities, but the source of the interesting behavior, i.e. the nonlinearity, also makes the mathematical analysis of solutions exceedingly difficult in general.

For this reason, it is tempting to turn to numerical analysis of the differential equations in order to determine something about the solutions. Yet, this poses a new set of difficulties because relatively little is known in mathematical terms about whether accurate numerical solutions can be produced for such problems. The nonlinear nature of the problems often deters convergence analysis, and even when possible, convergence analysis usually depends on unverifiable assumptions such as the existence of high-order derivatives of solutions. Moreover, standard error bounds are unsatisfactory in practical terms because they depend on derivatives of the solution, which are unknown, and usually include a factor that grows exponentially with time, making the bounds meaningful only for a short initial transient period. In short, most of the numerical results in the literature are missing even a rough quantitative estimate of the size of the error. This is particularly troublesome in this area because so little is known about the solutions themselves, increasing the reliance on numerical methods "working" as desired.

In some applications, the goal is to measure characteristics of a collection of solutions rather than following particular solutions accurately. An example is the class of highly chaotic systems obtained in molecular dynamics studies, where it is hoped that averages computed over numerical trajectories are approximations of averages of a thermodynamic ensemble of exact trajectories. There is little mathematical analysis to support this leap of faith however. Moreover, it appears that there are at least two ways to compute such averages: to follow one trajectory for an unrealistically long time period or to follow many trajectories over a period of time for which accuracy is guaranteed. Without further assumption, there is no reason to believe that these yield equivalent results. In the latter choice, accurate computation of individual trajectories is fundamentally important.

We attempt to deal with these problems by using adaptive finite element methods with error control based on feedback from the computation. The error control rests on a rigorous theory of *a posteriori* analysis in which the error is bounded by computable or approximatible quantities that depend on the numerical solution (rather than the unknown true solution). The error estimators indicate the proper choice of resolution to achieve the required accuracy, while the use of adaptive meshing enables the computational resources to be used where needed in order to be efficient. The theory has been worked out completely for linear parabolic problems [7]; there has been much progress on systems of ODEs of fixed dimension [9], [11] (these references also contain a comparison of this new theory to classical theory); and also some work on nonlinear parabolic problems [8]. However, there are important issues remaining to complete the theory for ODEs (which we describe below) and much work remaining to be done for systems of reaction-diffusion equations.

The main deterrence to the widespread use of adaptive methods in PDEs is the computational complexity required to code them. We believe that it is possible to reduce the implementation advantage of simple explicit methods over adaptive methods for the general user because many ingredients of adaptive codes for differential equations are common to all sorts of problems. Making these ingredients accessible in a general way reduces the overhead of implementation. In other work, we have shown that the complexity of adaptive spatial discretization can be managed by a code such as DIME [25], or more generally with a Voxel Database [24]. In this paper, we concentrate on adaptive time discretization, and in particular, accurately solving the ODE in time that results from the space discretization of a PDE.

In our view, it is proper for numerical analysis to address not only problems in approximation of functions and computations of numbers, but also the ways in which numerical algorithms are implemented on real hardware. Computer science and numerical analysis should evolve together. In particular, we have found that designing a code with parallelism in mind *ab initio* is no more difficult than designing a sequential code to implement the mathematical theory. This is in contrast to the usual difficult situation of 'parallelizing' an existing sequential code. Our adaptive ODE code is designed to be run on a MIMD parallel computer, [15] which includes massively parallel machines, as well as on clusters of workstations communicating by Ethernet.

This paper has three objectives. First, we describe an adaptive numerical method to solve sparse systems of ODEs. The second objective is to discuss the implementation of these procedures into a flexible and portable software structure, so that it runs efficiently on many processors of a MIMD parallel computer. Third, we describe the application of the *a posteriori* theory of adaptive error control for ODEs developed in [9], [11] to systems of large dimension and then present the results of numerical experiments on several reaction-diffusion problems that address the issue of whether accurate computation is indeed possible.

2 Theory

2.1 Computational Error

2.1.1 Computational Order and Time Step Restrictions

There are four factors that affect the accuracy obtained by an approximation of the solution of a nonlinear differential equation:

1. **The stability of the exact solution**. The stability of the solution is a global property determining how nearby perturbations behave as time elapses; in this case, the perturbations of concern are the errors induced by the numerical discretization.

2. **The nonlinear nature of the problem**. The nonlinear equations that determine the solution at a given point are solved by linearizing the equations around one or more points where the solution is known.

3. **The quality of the discretization**. The smoothness of the solution is a local property determining how well it can be approximated by functions with a finite number of degrees of freedom over a given interval, such as those used by a numerical method.

4. **The properties of the numerical method**. The fourth consideration has to do with the stability properties of the numerical method. Recall that in the classical theory of numerical methods for PDEs, numerical stability is a necessary condition for convergence.

In adaptive error control, we want to choose the time steps according to the smoothness and the stability properties of the solution, and we desire to avoid, as much as possible, restrictions due to solving a nonlinear problem and numerical stability. In consideration of (4), explicit methods require the Courant-Friedrichs-Levy (CFL) condition on the relation between the time step and the space mesh spacing to be satisfied, regardless of the pointwise behavior of the solution. Hence, we employ implicit methods and avoid restraints imposed by numerical stability altogether. Having chosen to use an implicit method, we generally must solve a set of nonlinear equations, and there is a choice between first-order, fixed point iterations that are easily implemented, and higher-order methods, such as Newton's method, that cost more per iteration. If we choose a stepwise-cheap, fixed-point iteration, there is again a severe step size restriction in order to obtain convergence. This restriction can be as severe as the CFL condition for explicit methods. On the other hand, Newton's method can converge for large steps, provided that a good initial guess for the iteration is provided. Of course, because such a guess is usually computed from previous values of the approximation, this in turn gives a new restriction on the step size. However, this choice at least gives the possibility of allowing adjustment of step sizes.

2.1.2 The Error of Interpolation

With regard to factors (1) and (3), the chief tool for estimating the error of an interpolant of a given function is Taylor's theorem. For example, if y has one continuous derivative on an interval $[t_1, t_2]$, with $k = t_2 - t_1$, and Y denotes the constant Taylor polynomial computed at t_1, then

$$\|y(t_2) - Y(t_2)\| \leq k \max_{\xi} \|y'(\xi)\|; \tag{3}$$

while if y has two continuous derivatives and Y denotes the linear Taylor polynomial computed at t_1,

$$\|y(t_2) - Y(t_2)\| \leq \frac{1}{2}\, k^2 \max_{\xi} \|y''(\xi)\|. \tag{4}$$

In each case, the bounding quantity has four factors:

- a constant which we shall call a 'stability factor', which is 1 in the examples above;
- an "interpolation constant" that depends only on the order of the approximation, it is 1 and 1/2 respectively;
- a power of the time step; and
- a quantity depending on derivatives of the solution.

A similar result holds for other polynomial-based interpolants of y on the interval, though the interpolation constant may have a different value. In general, the error is indeed the same order as error bound: in which case, it is said to be *sharp*. This is an important property of error bounds used for adaptive error control since a bound that is much larger than the error most of the time leads to inefficient computations. Error bounds for appropriate interpolants of the solution are the benchmark by which the error bounds of a finite element approximant are judged; bounds such as (3) and (4) are called *optimal* in the terminology of the finite element method.

2.1.3 *A Priori* Error Bounds

Taylor's theorem is also the main tool used to study the convergence properties of numerical methods for ODEs. The resulting *a priori* error bounds are similar in form to (3) and (4), bounding the error in terms of a quantity consisting of four factors: • the "stability factor"; • the "interpolation constant"; • a power of the mesh spacing; and • a quantity that depends on derivatives of the solution.

For example, the classic error bound for the backward Euler approximation Y_n for (1) computed on a set of nodes $t_0 = 0 < t_1 < t_2 < \cdots < t_N = T$ with step size k is

$$\|y(t_n) - Y_n\| \leq \frac{e^{LT} - 1}{L} \, \frac{1}{2} \max_{0 \leq \xi \leq T} \|y''(\xi)\| \, k, \quad 0 \leq n \leq N, \tag{5}$$

where L is the Lipschitz constant of f, which is a measure of the smoothness of f. *A priori* error bounds are derived by considering the question:

How well does the exact solution satisfy the discrete equations?

There is, however, an important difference between the interpolation of a given function and the approximation of the solution of a differential equation. In the first case, the interpolant is computed with full knowledge of the function, and the error in one interval has only a local effect. This is the reason that the stability factor is one. In the case of approximating a solution of a differential equation, errors generally propagate and accumulate throughout the domain. This difference is reflected in the error bounds. The stability factor in (5) grows exponentially in time. Note that the accumulation can also affect the number of derivatives required in the bound. For example, compare (3) and (5), both of which bound the error of an approximation with one degree of freedom on each interval. This is known as *loss of optimality* of the error bound.

An *a priori* bound depends on general properties of the solution and the approximation. For this reason, it is not computationally useful. Most obvious is the dependence of the bound on unknown derivatives of the solution. The *a priori* nature is also reflected in the exponentially increasing factor, which is determined by the most pessimistic rate of accumulation of errors. In general, this exponential factor is far too pessimistic for the bound to be computationally useful. For example, in linear parabolic homogeneous problems, there is no accumulation of error. In problems with a conserved quantity given by a norm of the solution, the rate of error accumulation appears to be only polynomial in time on average. In general problems, there may be relatively brief periods of exponential growth of error, but rarely as fast as suggested by the worst case. For example, in fluid flow problems, L depends on the Reynolds number and the term that is exponentiated reaches size $10^5 - 10^{10}$ in an extremely short time, implying that accurate computation is essentially impossible.

2.1.4 *A Posteriori* Error Bounds

Another approach to error control is based on *a posteriori* error bounds that involve **computable** quantities. This kind of bound is derived by answering the question:

How well does the numerical solution solve the differential equation?

A posteriori error bounds consist of four factors:

- a "stability factor" $S_1(t)$ that measures the accumulation of error;
- a constant C_i determined only by the order of the method;
- a power of the mesh size k_n; and
- a quantity that measures the *residual error* by the regularity of the approximation.

For example, an *a posteriori* error bound for the backward Euler scheme is:

$$\|y(t_n) - Y_n\| \leq S_1(t_n) \max_{1\leq m\leq n} C_i \left(\left\| \frac{Y_m - Y_{m-1}}{k_n} \right\| + \|f_t(Y_m, t_m)\| \right) k_n, \quad 0 \leq n \leq N, \tag{6}$$

where f_t denotes $\frac{\partial f}{\partial t}$. The residual error measures how well the approximation satisfies the differential equation over one step.

In the *a posteriori* theory outlined below, $S_1(t)$ is determined by the solution of a *dual problem* which is solved *backwards in time.* The dual problem is obtained by linearizing the weak formulation of the differential equation around the solution to be approximated. The linear dual problem reads as follows: for $1 \leq n \leq N$, find z_n such that

$$\begin{cases} -\dot{z}_n = f_y(y,t)^* z_n, & t_n > t > 0, \\ z_n(t_n) = e(t_n)/||e(t_n)||, \end{cases} \tag{7}$$

where f_y denotes the Jacobian of f with respect to y while f_y^* denotes its transpose and $e(t_n)$ denotes the error at time t_n. Note that this problem is computed "backwards", but there is a corresponding change in sign. The stability factor is defined as

$$S_1(t_n) = \int_0^{t_n} ||\dot{z}_n(t)||dt. \tag{8}$$

Since (7) and (8) depend on the solution, the stability factor is not computable. $S_1(t)$ generally varies with time greatly and using a crude bound on $S_1(t_n)$ in a computation is not practical for the reason above. Hence, in practice (7) and (8) are approximated during a computation.

We remark that the dual problem arises naturally when considering the weak formulation of a differential equation. The analysis is very general in the sense that effects of perturbations in initial data or in the nonlinearity can be associated to other stability factors through similar arguments.

2.2 Parallel Computing

The mathematical statement of the adaptive error control algorithm outlined below is written in terms of matrices, vectors, and scalars. These classes are related by the following operations: a matrix may be used to transform a vector into another, there are vector functions of vectors, and there is a norm function that converts vectors to scalars. We use this simple description to design a parallel ODE solver for a MIMD model of parallel computing in which each processor has its own local memory and communication between processors is by message passing.

A basis of the M-dimensional vector space of equation (1) is partitioned among the processors, so that each processor is responsible for a subspace of the full phase space. We refer to $\mathbb{R}^M$ as the *global vector space* and the subspaces for which a processor is responsible are the *local vector spaces.* Scalars are stored redundantly by each processor.

The chief issue in efficient implementation on a parallel computer is the communication between processors. For example, program flow for the ODE solver is determined by norms of vectors in $\mathbb{R}^M$. As long as the same value of the norm is delivered to all processors at appropriate times, they all follow the same flow of control. This property makes the ODE solver an SPMD (Single Program, Multiple Data) program. The norm of a global vector is the root mean square sum of the norms of the local vector spaces, thus the calculation of a norm requires communication between the processors. It is accomplished with a *combine*, or global sum operation [15], on a parallel computer.

The other operations requiring communication are the calculations of the function f and the Jacobian of f from equation (1). These in turn often require a matrix-vector product. Since efficient implementation of matrix-vector products and vector functions are completely problem dependent, we assume that these are handled by the user of the ODE solver. The upshot is that the code for the parallel ODE solver is identical to the code for the sequential ODE solver. This is also true for the nonlinear solver and the iterative linear solver, which are also pure SPMD programs.

In a fully space and time adaptive code for a PDE, the user interface modules are replaced by the adaptive space discretization module. This module has the responsibility for dealing with the complicated parallel issues such as redistributing the vector spaces across the processors (load balancing) when the space discretization is changed. In this setting, the splitting into subspaces is equivalent to domain decomposition of the PDE.

2.3 Software Structure

A partial differential equation solver has several components:

- Space discretization of the PDE yielding a large, usually sparse system of ODEs,
- An implicit integration scheme for the ODE system,
- A solver for the resulting nonlinear system of equations, such as Newton's method,
- A solver for the resulting system of linear equations.

A code that implements these mathematical techniques is "flexible" if it is straightforward to replace one module with an implementation of another technique. For example, it may be desirable to replace the nonlinear solver with a quasi-Newton method, or use a different implicit solution scheme for the ODE module. It also means that the problem is implemented in such a way that different patterns of sparsity can be handled with equal ease.

Flexibility reduces, as always, to a specification of interfaces. If the interfaces between software modules are well specified, then each module can function independently.

2.3.1 Matrix Representations

A critical issue in the interfaces between modules is the specification of the variables, and in particular, the large matrices resulting from discretization. A full matrix is usually defined by the number of rows, the number of columns, and a pointer to an array of the matrix elements. In FORTRAN, it is usual to supply another integer, the "fixed" size to which the (work) array is dimensioned. However, this representation of a full matrix, although easy to understand and implement, is not practical for the very large, sparse ODE systems.

When the matrix is sparse, the representation has to be more sophisticated if the sparseness is to be used to advantage. One representation is based on specifying which matrix elements are non-zero, then giving the values of those elements. For a given representation, there must be a library available to do such operations as multiplication by a scalar or addition of matrices, together with a method of constructing a matrix, a linear solver, and so on. Such sparse matrix representations generally take up a great deal of memory and put stress on the memory allocation system. In addition, since one of our objectives is to run the solver on massively parallel processors, such a sparse-matrix library must be available for massively parallel platforms.

Another possibility is to "hard-code" a particular sparsity structure into the code. For example, in solving a PDE that involves the Laplace operator on a square mesh, there might be code such as

```
Ax[i][j] = 0.25*(x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1]).
```

It is reasonable for this to appear in the software describing the space-discretization of the PDE. However, it should not be used in the software responsible for solving the ODE system, because it is then difficult to use the software to handle discretizations with a different stencil. Furthermore, it is difficult to write the software in a modular form because this structure is present throughout the code.

Coding the solution of large linear systems on a parallel computer raises another consideration for the choice of matrix representation. Direct methods such as LU decomposition are difficult to parallelize efficiently, and they also fill in the sparse matrix. On the other hand, iterative solvers parallelize well, since the only parts that require consideration are the matrix-vector product and the norm calculation. Iterative solvers require a good approximate guess to be efficient, but this is readily available when integrating an ODE system. For example, the solution at the last timestep, or the solution derived from a less accurate solution method, such as an explicit or multistep solver, are both good possibilities.

We have implemented the parallel ODE solver with *matrix-free* methods [18] [4], where matrices are not assembled as collections of numbers, but are instead passed around as *functions*. The kernel of matrix-free methods may be stated quite simply:

**The fundamental concept is a linear transformation,
not the matrix that represents it.**

Iterative methods access the matrix only as a transformation on the vector space, hence a matrix can passed between software modules as a *function*, not an array or a structure pointer. For example, the declaration for the QMR solver (an iterative solver for nonsymmetric linear systems) looks like:

```
QMR_linear_solve (
    int n,                   /* dimension of system              */
    int (*A)(),              /* matrix multiply method           */
    int (*At)(),             /* transpose of mmm                 */
    real (*norm)(),          /* vector norm function             */
    real *x,                 /* in: initial guess; out: solution */
    real *b,                 /* right hand side vector           */
    real tolres              /* tolerance criterion for solution */
)
```

The arguments `A` and `At` are *methods* of multiplying a vector by the matrix, and its transpose, respectively, and `norm` is the function that produces an appropriate norm for vectors. The linear solver has no knowledge of the structure of the matrix; it only needs to pass vectors to the matrix and then receive the transformed vectors back again.

Also note that the parallelism is contained in the matrix-multiply methods and in the norm method. It is in these functions that communication between processors occurs, not in the linear solver. Thus, the linear solver can be tested on a workstation, run on a massively parallel machine without change, and reused for other applications.

2.3.2 Linear Solvers

As noted above, an implicit scheme for a set of ordinary differential equations reduces to a set of nonlinear equations, and these are solved by a Newton or quasi-Newton algorithm, which in turn requires the solution of large, sparse sets of linear equations.

In the case of the PDEs we consider, the linear equations are in general • not symmetric, • not positive-definite, and • not diagonally dominant. To solve such systems efficiently, we use the QMR (Quasi-Minimal Residual) method [16].

QMR is a Krylov-subspace technique which requires only the operations of multiplication by the matrix and its transpose, and a scalar product. Because the matrix is non-symmetric, there is no short-recurrence sequence of orthogonal vectors. In the GMRES method [21], a long-sequence recurrence is used, which consumes significant memory unless restarts are frequent; alternatively, the BiCG [1] method uses two mutually orthogonal sequences of vectors as a basis, which can be done with a short recurrence, so that the matrix is reduced to a tridiagonal system. QMR follows BiCG by using two sequences of vectors, but QMR solves this reduced system in a least-squares sense, rather than with an implicit LU decomposition. This, together with lookahead techniques, provides more robustness than BiCG, but without the memory overhead of GMRES.

Note that the QMR method is simply a choice of one among many Krylov and direct methods for the solution of linear equations. The ODE code also has a full solver based on Gaussian elimination, and other methods can be easily substituted.

2.3.3 Differentiable Vector Fields

The matrix-free idea can be extended to non-linear problems. The nonlinear solver, for example, is designed to solve

$$F_{nonlin}(u) = 0, \qquad \text{with} \qquad F_{nonlin} : I\!R^n \to I\!R^n. \tag{9}$$

Besides the function F_{nonlin} itself and an initial guess for the solution, the solver also needs the Jacobian (derivative) matrix of F_{nonlin} and its transpose, and a norm function for the vector space that contains u. We shall call this collective object a *differentiable vector field* or Vfield.

The ODE solver works with a Vfield that we call F_{ode}, since the statement of the ODE is couched in such terms, as in equation (1). When the ODE is discretized in time, a set of nonlinear equations has to be solved: this entails passing a Vfield F_{nonlin} to the nonlinear solver. In the case of the backward Euler step, the relationship between these two is simply

$$F_{nonlin}(u) = \frac{(u - u_{old})}{k} - F_{ode}(u), \tag{10}$$

where k is the timestep and u_{old} is the known value of u at the current time level.

A Vfield object includes the Jacobian of the vector field. The Jacobian is a matrix in the sense of the section above: it is a linear transformation on vectors. For the above example, we simply have

$$J_{nonlin}(w) = w/k - J_{ode}(w) \tag{11}$$

so that the action of the Jacobian of the nonlinear problem on a vector is defined in terms of the action of the Jacobian of the differential equation. This separation seems trivial for a method as simple as the backward Euler, where it would be easy to combine directly the code for the differential equation ($f : I\!R^M \to I\!R^M$ in (1)) with the scheme used for numerical solution of the equation. However, more complex schemes, where the dimension of the system to be solved is a multiple of M, this direct approach becomes increasingly tedious and error-prone. Add to this the further difficulty of evaluating the Jacobian transformation, and the Vfield approach is clearly superior.

2.3.4 Preconditioning and Interface Expansion

The definition of a matrix purely in terms of its linear action on vectors yields a clean programming model in which each module has a well-defined function and the interfaces between modules are narrow and unambiguous. To implement a linear solver for symmetric matrices, such as Conjugate Gradient, this is all that is needed. For nonsymmetric systems, the transpose operation must be added to the module interface.

Predictably, these iterative linear solvers do not attain their full potential in their simplest forms. A preconditioner is needed, which is in some sense an approximation to the matrix that is easily inverted. For example, one simple preconditioner is *diagonal scaling*, where the matrix is "approximated" by ignoring its off-diagonal elements. To use this, the definition of a matrix must be expanded, so that there are three methods: the action of the matrix, the action of the transpose, and the action of the inverse of the diagonal.

Although preconditioners have been suggested [4] that are purely matrix-free, such as the Incomplete Orthogonalization Method [17], it is still generally true that more effective preconditioners require more complex software interfaces.

2.3.5 Archiving the Solution

To compute the error bound on the approximate solution of the ODE, we solve a linear "dual" system which is related to the solution of the original ODE. The dual system is solved backwards in time and requires storage of the time history of the approximation. We have implemented an archive object for the storage of this time history, using a dynamically allocated linked list of memory blocks for storage, with adjustable block size. The archive assumes that the data is accessed in time order, so it searches for data first in the block in which data was previously found, then resorts to a binary search.

2.4 The Galerkin Finite Element Methods for ODEs

We use Galerkin finite element methods for the numerical integration of (1). The Galerkin formulation makes the derivation of *a posteriori* error bounds natural. The details of these methods are discussed in [6], [9], and [11].

2.4.1 The Methods

The finite element method is based on a weak formulation of (1) that reads: find the differentiable function y on $[0,T]$ such that

$$\begin{cases} \int_0^T (\dot{y}, v)dt = \int_0^T (f(y(t),t), v(t))dt, \\ y(0) = y_0, \end{cases} \tag{12}$$

for all piecewise continuous functions v on $[0,T]$, where $(\cdot,\cdot)$ denotes the standard dot product. We partition $[0,T]$ into N intervals $(t_{m-1}, t_m]$ with timestep $k_m = t_m - t_{m-1}$ and compute the approximation Y as the polynomial on each interval that satisfies (12) for test functions in an appropriate finite dimensional test space of polynomials. There are two classes of method, distinguished by whether the approximation is continuous at interval boundaries or not: the continuous Galerkin (cG) approximation is continuous and the discontinuous Galerkin (dG) is discontinuous. We use the notation dG0, cG1, dG1, cG2, etc., where the last digit is the polynomial order of the basis.

The exact formulas are given in Appendix A. For example, the simplest scheme, dG0, gives a zero-order polynomial $Y = Y_m$ on the m^{th} interval, yielding

$$\begin{cases} Y_m = \int_{I_m} f(Y_m, t)\, dt + Y_{m-1}, \\ Y_0 = y(t_0). \end{cases} \tag{13}$$

Note that the backward Euler scheme is obtained by applying the rectangle quadrature rule to the integral in (13). In general, computing the Galerkin approximations involves analytically computing integrals of the form

$$\int f(\text{polynomial in } t, t)\, dt.$$

There is an advantage to computing these integrals analytically, if possible, because the errors may accumulate more slowly than if quadrature is used. However, it may be difficult or impossible to do this, and so a general-purpose code employs quadrature to evaluate the integrals numerically. Depending on the choice of quadrature rule, the result is a Runge-Kutta scheme. Conversely, many Runge-Kutta and multi-step schemes can be written as a dG or cG approximation with the appropriate choice of quadrature.

The two criteria for choosing a quadrature rule are to preserve the order of convergence and preserve the stability properties of the method. We also desire to use as few function evaluations as is consistent with these two points. The best choice of interpolatory quadrature rule for the dG method uses the Gauss points in each interval. The choice of quadrature for the cG method is not as clear because preserving the conservation property depends both on the form of the problem and on the conserved quantity. For the cG1 method, we employ the trapezoidal rule, but some problems may call for a different choice. Exact formulas for the quadrature rules we use are given in Appendix A.

2.4.2 Properties

We summarize the results in [9] and [11]. The cG and dG methods are implicit, stiffly A-stable, one-step methods. The stability properties of the dG methods make them effective for stiff problems in particular, while the cG approximations often inherit the property of preserving an conserved quantity if one is associated with the solutions of the differential equation.

The cGq and dGq methods converge with up to order $q+1$ on each interval I_m. More precisely,

$$\max_{0\le t\le t_n} \|e(t)\| \le C\left(1+Lt_n e^{CLt_n}\right)\max_{m\le n}\left\{\min_{p\le q+1} k_m^p \max_{I_m}\|y^{(p)}\|\right\}, \quad 1\le n\le N. \tag{14}$$

Note that these results are optimal in order. These methods also have a *superconvergence* property at time nodes when the mesh does not change too much. Namely, the dGq method converges with order $2q+1$ at the nodes $\{t_n\}$ when y has $2q+1$ continuous derivatives and the cGq method converges with order $2q$ at nodes when y has $2q$ continuous derivatives. The exact forms of the bounds are analogous to (14). See [6].

2.4.3 *A Posteriori* Error Bounds

A posteriori error bounds are the basis for adaptive error control decisions. We recall that z_n solves the dual problem (7) with initial data given at t_n, and we define the stability factors,

$$S(t_n) = S(t_n, y) = \|z(0)\|,$$

$$S_i(t_n) = S_i(t_n, y) = \textstyle\int_0^{t_n} \|z_n^{(i)}(s)\| ds, \quad i \ge 0.$$

In the following, we use q to denote the degree of the dG or cG approximation and r to denote the degree of the quadrature used to compute the approximation. We use p to denote the possible order of convergence of the approximation, and l to denote the possible order of convergence of the quadrature. We use $C_{q,p}$ and $C_{r,l}$ to denote *interpolation* constants that depend **only** on q, p and r, l respectively. These are introduced to make the stability factors dimensionless. Then,

$$|e_n^-| \le S_1(t_n) \max_{1\le m\le n}\left\{\min_{0\le p\le q} C_{q,p}\left\{||[Y]_{m-1}|| + k_m^{p+1}\left\|\frac{d^p}{dt^p} f(Y(t),t)\right\|_{I_m}\right\}\right\} \tag{15}$$

$$+ S_0(t_n) \max_{1\le m\le n} \left\{ \min_{0\le l\le r} C_{r,l}\, k_m^l \left\| \frac{d^l}{dt^l} f(Y(t),t) \right\|_{I_m} \right\}.$$

The first term on the right measures the error of the Galerkin discretization and is optimal in order. The second term on the right measures the error from using the quadrature. It is not optimal in order, and also the quadrature residual errors accumulate at a different rate than the original discretization error. See [14] for further discussion. The original discretization error arises because we seek an approximation of the solution y in a finite dimensional space. The quadrature error arises because we sample the flow only at discrete points (i.e., at the quadrature points). It is important for the efficiency of the error control to take into account the two sources of error independently, as this bound allows.

The superconvergence results have a similar form, except that high-order stability factors S_p, $p \ge 2$ are involved, see [6], [9], [11]. We summarize these results in a convenient form as

$$\|e_n^-\| \le \max_{1\le m\le n} \min_{1\le p\le \hat{q}} S_p(t_n)\, \mathcal{R}(q,p,Y,m) + \max_{1\le m\le n} \min_{\substack{0\le p\le q\\ 1\le l\le r}} S_p(t_n)\, \mathcal{Q}(r,l,q,p,Y,m), \tag{16}$$

where $\hat{q} = q$ for cGq and $q+1$ for dGq, and the local *discretization residual* $\mathcal{R}(q,p,Y,m)$ and the local *quadrature residual* $\mathcal{Q}(r,l,q,p,Y,m)$ are defined as appropriate.

These results hold if f is Lipschitz continuous with constant L, and in addition for the methods of order two and more, f_y is Lipschitz continuous as well. We also assume that k is sufficiently small, so that the local residual error is smaller than a fixed constant.

2.4.4 Approximation of the Stability Factors

The stability factors cannot be computed directly because (7) requires the solution y. It is possible to bound the stability factors *a priori*, however as explained above, the resulting bounds are too crude to be used in error control. Therefore, we compute approximate stability factors $S_p(t_n, Y)$, where we consider the linear problem (7) obtained by linearizing around the approximation Y and use a guess for the initial condition $e(t_n)/\|e(t_n)\|$. The resulting system to be solved is: for $1 \le n \le N$, find $Z = Z_n$ such that

$$\begin{cases} -\dot{Z} + f_y(Y,t)^* Z = 0, \quad t_n > t > 0, \\ Z(t_n) = d_n, \quad \|d_n\| = 1. \end{cases} \tag{17}$$

We then approximate Z using the same scheme and the same step sizes used to compute Y, with the steps possibly altered to take into account the convergence of the linear solver. Finally, we compute $S_p(t_n, Y)$ by using the approximate values of Z in quadrature formulas for the integrals defining $S_p(t_n, y)$.

The reliability of the error control hinges on the quality of the approximation $S_p(t_n, Y)$, which in turn depends on the effect of the two steps used to change (7) into the computable problem (17). Based on computations on many examples (see [9] and [11]), we believe that these two steps are justified. We express this as two conjectures:

Conjecture 1: The stability factors computed from (17) using the approximation Y instead of (7) using the solution y are good approximations to the true stability factors when Y is a good approximation of y.

Conjecture 2: The stability factors are relatively insensitive to the choice of initial values for (17) when computed over a sufficiently long time interval on many systems, and good values can be obtained using a small number of initial values.

We note that the stability factors do not need to be computed with great accuracy (order of magnitude is sufficient) for the purposes of accurate error estimation and control. Under general conditions, $S_p(t_n, Y)$ converges to $S_p(t_n, y)$ as the tolerance tends to zero, provided the true initial data $e(t_n)/\|e(t_n)\|$ is used in the backward computation. As far as Conjecture 2, on certain classes of problems, such as contractive problems, we can prove that the choice of initial data is immaterial. We discuss this issue further below.

The issue of the approximation of the stability factor is the last issue remaining to complete this theory for ODEs of fixed dimension. The analogous issues also exist for PDEs, with the added technical difficulties associated to approximating infinite dimensional systems.

The approximations of the stability factors represent the majority of the overhead that adaptive error control requires in terms of computing time. This overhead, typically representing 10-90 percent of the total time depending on the stability of the solution and the number of points at which the error control is imposed, appears to be necessary to achieve error control efficiently and reliably. However, neither of the alternatives appear reasonable to us: using *a priori* bounds on the stability factors, which are generally so large as to preclude computation altogether; or ignoring the effects of accumulation of error, which means that quantitative error control is simply absent. Moreover, it turns out that knowledge of the stability factors themselves indicate much useful information about the solutions, as we demonstrate below.

We recall that there are also interpolation constants $C_{q,p}$ and $C_{r,l}$ in the *a posteriori* bounds. These constants depend on the order of convergence, but the exact values are affected by the inequalities used in the derivation of the bounds. It would be very tedious to trace through the analysis and determine exact values. Instead, we compute linear problems with known solutions and numerically determine values that make the error bounds the same size as the error. We use these values in all subsequent computations.

2.5 The Algorithm for Adaptive Error Control

Given a tolerance TOL, the **ideal** goal of the adaptive error control is to satisfy

$$\|e_n^-\| \le \text{ TOL}, \tag{18}$$

for $n \ge 0$ while doing as little computational work as possible. We call TOL the *global error tolerance*. To achieve (18), we compute the approximation so that

$$\max_{1\le m\le n} \min_{1\le p\le \hat{q}} S_p(t_n)\, \mathcal{R}(q,p,Y,m) + \max_{1\le m\le n} \min_{\substack{0\le p\le q\\ 1\le l\le r}} S_p(t_n)\, \mathcal{Q}(r,l,q,p,Y,m) \le \text{ TOL}, \tag{19}$$

for $n \ge 1$. Provided the assumptions of the *a posteriori* analysis hold, (18) is guaranteed to hold if (19) holds.

Solving the practical optimization problem of minimizing the computational work while satisfying (19) is difficult and we simplify the problem in several ways. First, we program only the dG0, cG1, and dG1 methods, giving a range of first, second and third order schemes. In general, higher-order convergence in methods for (2) is not expected because of the difficulty of satisfying high-order compatibility conditions at the boundaries and the regularity constraints on the solutions. There are no superconvergence results for the dG0 and the cG1 methods, so in those cases, the only stability factors that occur are $S_0(t)$ and $S_1(t)$. $S_2(t)$ is involved in the dG1 superconvergence result, but we use a Lipschitz assumption to replace $S_2(t)$ by $LS_1(t)$. Equation (19) thus simplifies to

$$S_1(t_n) \max_{1\le m\le n} \min_{1\le p\le \hat{q}} \mathcal{R}(q,p,Y,m) \; + \; S_0(t_n) \max_{1\le m\le n} \min_{\substack{0\le p\le q \\ 1\le l\le r}} \mathcal{Q}(r,l,q,p,Y,m) \le \; \mathrm{TOL}, \qquad (20)$$

for $n \ge 1$. With these simplifications, minimizing the computational work is equivalent to maximizing the step size for each interval. We achieve (20) by a two-stage process.

2.5.1 Local Step Size Control

We introduce a local discretization residual tolerance RTOL and a local quadrature residual tolerance QTOL and compute Y so that on each interval I_m, $m \ge 1$,

$$\min_{1\le p\le \hat{q}} \mathcal{R}(q,p,Y,m) \; \le \; \mathrm{RTOL} \text{ and } \min_{\substack{0\le p\le q \\ 1\le l\le r}} \mathcal{Q}(r,l,q,p,Y,m) \; \le \; \mathrm{QTOL}. \qquad (21)$$

On I_m, we let p'_m denote the order of the step in the minimum of the residuals $\{\mathcal{R}(q,p,Y,m)\}_p$ and l'_m denote the order of the step in the minimum of the residuals $\{\mathcal{Q}(r,l,q,p,Y,m)\}_{l,p}$, and we write

$$\min_{1\le p\le \hat{q}} \mathcal{R}(q,p,Y,m) \; = k_m^{p'_m} \, \mathcal{R}'(Y,m) \text{ and } \min_{\substack{0\le p\le q \\ 1\le l\le r}} \mathcal{Q}(r,l,q,p,Y,m) \; = k_m^{l'_m} \, \mathcal{Q}'(Y,m).$$

We achieve (21) by a prediction-correction iteration. Suppose that $k_{m,pred}$ denotes a *predicted* time step for I_m. We compute $Y|_{I_m}$ and $\mathcal{R}'(Y,m)$ and $\mathcal{Q}'(Y,m)$. If (21) is satisfied, we accept the step and compute forward with a new predicted step. If either inequality is not satisfied, we recompute from the previous time node with a new predicted step. In both cases, the new predicted step is computed by

$$k_{m,pred} = \min \left\{ \left(\frac{\mathrm{RTOL}}{\mathcal{R}'(Y,m)} \right)^{1/p'_m}, \left(\frac{\mathrm{QTOL}}{\mathcal{Q}'(Y,m)} \right)^{1/l'_m} \right\}.$$

2.5.2 Global Step Size Control

The local tolerances are chosen so that (20) holds, i.e.,

$$S_1(t_n)\,\mathrm{RTOL} \; + \; S_0(t_n)\,\mathrm{QTOL} \; \le \mathrm{TOL}, \qquad (22)$$

for $n \ge 1$. We compute the tolerances iteratively. We begin by setting RTOL = QTOL = $\frac{1}{2}$TOL, assuming that $S_1(t_n) = S_0(t_n) = 1$. We compute to the final time, checking (22) at each time node. If (22) holds at every node, then the computation has the desired accuracy. If (22) is violated at some time steps, then we compute new tolerances

$$\mathrm{RTOL} = \min_n \frac{1}{2}\mathrm{TOL}/S_1(t_n) \text{ and } \mathrm{QTOL} = \min_n \frac{1}{2}\mathrm{TOL}/S_0(t_n).$$

This is tantamount to assuming that the work associated to computing the approximation and the quadrature is equal. We then recompute the entire approximation with the new tolerances.

We emphasize that obtaining global error control at a time node means integrating a linear problem of the same dimension as (1) over an interval of the same length as the node. In practice, it may be sufficient to require global error control at some subset of the time nodes, for example, only at the final point. In the code, the user defines a set of sample times at which the global error is checked. The local step size control is maintained at every time step, since the form of the *a posteriori* bounds requires this.

There are two other constraints on the choice of step size. If the Newton iteration does not reach the user defined tolerance on the residual within ten or so iterations, the step is recomputed with half the predicted step size. Likewise, if the linear solver fails to converge with a residual error less than the tolerance, the steps are halved. In addition, there is a user defined maximum step size.

3 Computational Results

We now describe some numerical experiments that illustrate aspects of the material presented above. First, we conduct various tests of the adaptive error control using the well-known Lorenz system of ODEs. This is a good test problem for checking the accuracy of the error control because it is low dimensional, yet is nontrivial. Then, we consider a system of ODEs arising from space discretization of the bistable problem in one and two space dimensions. The bistable problem is a well-known example of a reaction-diffusion equation that has interesting behavior over long time intervals. We discuss the speedup gained on a parallel computer for this problem, and present numerical evidence as to the "computability" of the bistable problem over long time intervals.

3.1 The Lorenz System

In the early 1960s, the meteorologist E. Lorenz derived a simple model in order to explain why weather forecasts over more than a couple of days are unreliable. The model is derived by taking a three-element finite element space discretization of the Navier-Stokes equations for fluid flow (the "fluid" being the atmosphere in this case). After a change of variables, this gives a three-dimensional system of ODEs in time:

$$\begin{cases} x' = -\sigma x + \sigma y, \\ y' = rx - y - xz, \\ z' = -bz + xy, \\ x(0) = x_0, y(0) = y_0, z(0) = z_0, \end{cases} \tag{23}$$

where σ, r, and b are positive constants. These were determined originally as part of the physical problem, but the interest among mathematicians quickly shifted to studying (23) for values of the parameters that make the problem *chaotic*.

A precise definition of chaotic behavior is difficult, but we point out two distinguishing features: while confined to a fixed region in space, the solutions do not "settle down" into a steady state or periodic state; and the solutions are *data sensitive*, which means that perturbations of the initial data of a given solution eventually causes large changes in the solution. In such a situation, numerical approximations always become inaccurate after some time and it is important to determine this time in order to determine valid information about solutions from computations. In fact, accurate computation can reveal much detail about the dynamical behavior of the solutions, see [14].

We choose standard values $\sigma = 10$, $b = 8/3$, and $r = 28$, and we compute with the dG1 method. In Figure 1, we plot two views of the solution corresponding to initial data $(1, 0, 0)$. The solutions always behave similarly: after some short initial time, they begin to "orbit"

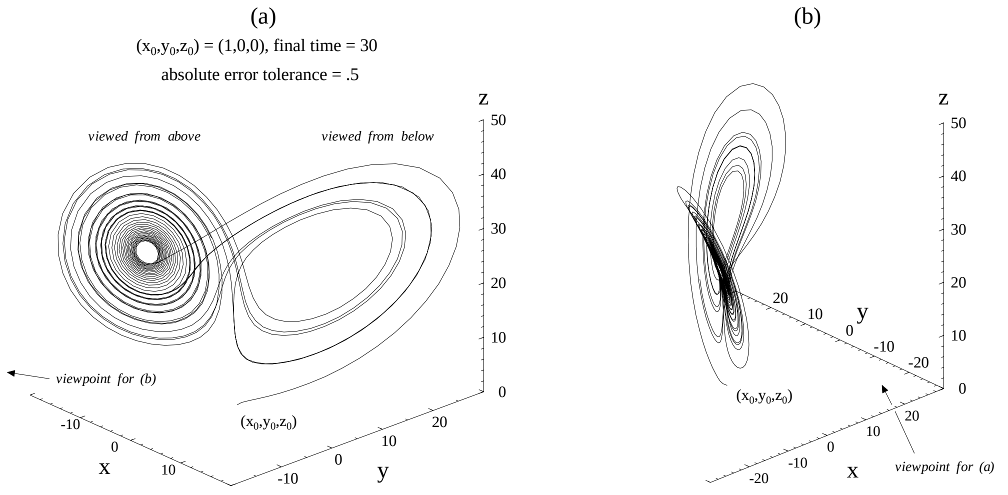

Figure 1: Two views of a solution of the Lorenz system

around one of two points, with an occasional "flip" back and forth between the points. The chaotic nature of the solutions is this flipping that occurs at apparently random times. In fact, accurate computation can reveal much detail about the behavior of the solutions, see [14].

We solve the Lorenz system using the dG1 method and compute the *a posteriori* error bounds at regular time intervals. To test the accuracy of the bounds, we compute an "approximate" error by comparing the approximation from this computation to an approximation computed with a residual tolerance that is 10^{-5} smaller. This brute force approach should yield a good approximation of the true error because the *a posteriori* error bound suggests that the approximation is accurate enough on the chosen interval to be within the asymptotic regime of convergence. In Figure 2, we plot the approximate error together with the *a posteriori* error bound versus time. There is remarkable agreement.

The standard *a priori* analysis yields a stability factor of size e^{99t}, precluding accurate computation beyond $t = 0.5$. This result bounds the exponential rate of error accumulation by the maximum norm of the Jacobian of the ODE system (which is about 99), assuming that the system is sensitive to the accumulation of error at the worst possible rate uniformly in the phase space. Actually, the system has this sensitivity only in a very small region of the attractor. In fact, the *a posteriori* error bound suggests that computations are meaningful up to $t = 30$. The precise error bounds are due to the fact that the computational *a posteriori* approach to error estimation measures the sensitivity of the system along the actual trajectory by integration.

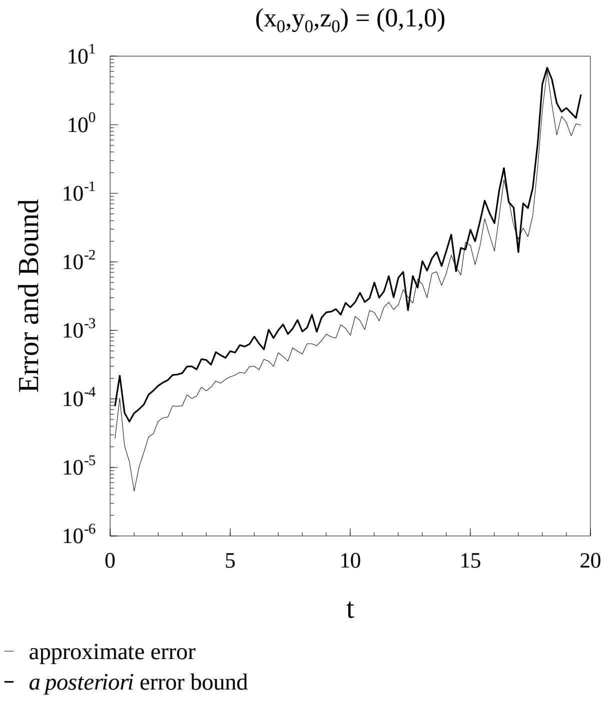

Figure 2: An approximation of the true error and the computed error bound

The error bound shown in Figure 2 is a linear combination of several sources of error. Each source is a product of a stability factor with the rate of production of the error; the stability factor expresses the growth rate of the error once it is formed. The three types of error growth that we consider here are:

- the effect of error in the initial conditions (measured by $S(t)$);
- the discretization error that arises because the solution is not a piecewise polynomial (measured by $S_1(t)$);
- the quadrature error that arises because the flow is sampled at only discrete quadrature points (measured by $S_0(t)$).

In Figure 3, we plot the stability factors on a logarithmic scale versus time. The data sensitivity of this problem is reflected in the overall exponential growth. Note that the factors do not grow uniformly rapidly and there are periods of time with different data sensitivity. Moreover, the overall average growth rate for $S_1(t)$ is approximately $e^{0.92t}$, see [14], nothing like e^{99t}.

We now show numerical evidence that supports Conjectures 1 and 2 of Section 2.4.4 concerning the approximation of stability factors. In Figure 4, we plot $S_1(t)$ for various trajectories that are computed with different tolerances. In support of Conjecture 1, the results suggest that it is sufficient to use the approximation instead of the (unknown) solution to compute the stability factor. We see that the stability factors computed from the trajectories with different accuracies are equal up to time 19. As less accurate trajectories begin to diverge grossly from the more accurate trajectories, differences in corresponding

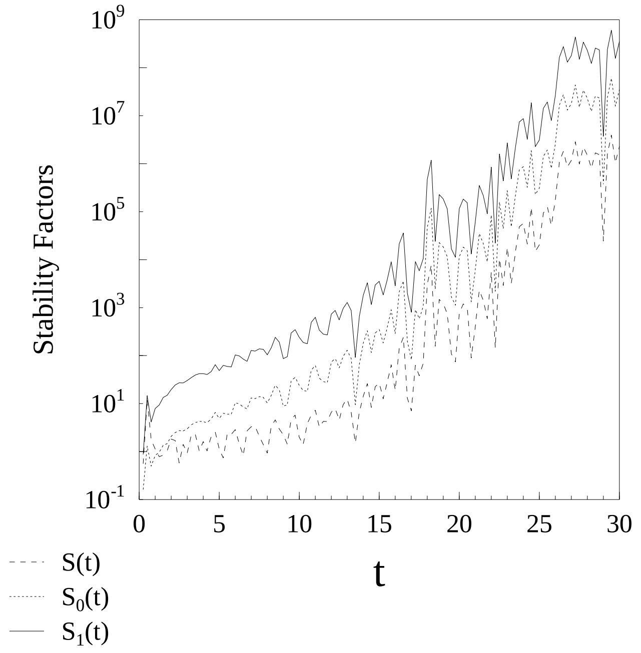

Figure 3: Three stability factors for the Lorenz system

stability factors become apparent. Even so, the stability factors remain roughly the same magnitude.

In Figure 4b, we plot the approximation to $S_1(t)$ computed for three different choices of initial data for the dual problem (17). These data support Conjecture 2, that the stability factor is relatively insensitive to the choice of initial data for the backward problems.

For small dense systems, it is sometimes more efficient to solve for the fundamental solution matrix of the stability system (17), with Z starting at t_n as a $M \times M$ identity matrix, rather than solving (17) for several initial values. This is because computing the fundamental solution means that (17) does not have to be integrated back to zero from each time.

3.2 The Bistable Equation in One Dimension

We now consider the bistable problem with Neumann boundary conditions in one dimension:

$$\begin{cases} u_t - \epsilon^2 u_{xx} = u - u^3, & 0 < x < 1, 0 < t, \\ u_x(0,t) = u_x(1,t) = 0, & 0 < t, \\ u(x,0) = u_0(x), & 0 < x < 1. \end{cases} \tag{24}$$

The bistable equation is one of the simplest problems that produce nonlinear relaxation to equilibrium in the presence of competing stable steady states. The stable steady states are $u \equiv 1$ and $u \equiv -1$, which are minimizers of the energy functional

$$\int_0^1 \left(\frac{\epsilon^2}{2} u_x^2 + \frac{1}{4}(u^2 - 1)^2 \right) dx.$$

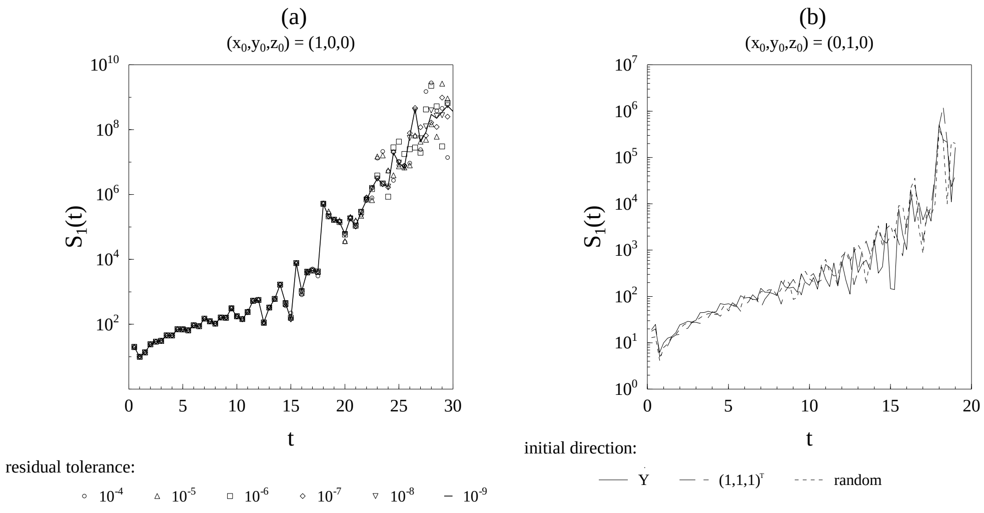

Figure 4: The stability factor for different trajectories and initial data

For generic initial data, $\lim_{t\to\infty} u(x,t)$ is one of these steady states. But, this convergence can be extremely slow because solutions of (24) can exhibit dynamic *metastability*. In general, u forms a pattern of transition layers between the values 1 and -1, where the layer thickness is of order ϵ. The subsequent time scale for substantial motion of the layers is $\exp(Cd/\epsilon)$, where $C = O(1)$ and d is the minimum distance between layers or between the layers and the boundaries. Metastable solutions are not local minimizers of the energy, and thus are always dynamic. After a metastable period, one or more of the layers disappear in a relatively quick transient and the system forms a new metastable pattern. This repeats until the eventual convergence to a steady state.

The main issue we address is whether accurate numerical computation is possible over the entire range of motion of a typical metastable solution. The standard *a priori* analysis, with an exponentially growing stability factor on the order of $\exp(Ct/\epsilon^2)$ ($\approx \exp(1000t)$ for the ϵ we use) rules this out quite definitely. It is possible to show[10] that accurate approximation is possible over the first metastable period, i.e. up to some time before the first transition when the wells have not collapsed too much, provided a certain "threshold" accuracy in space and time is maintained. This analysis does not indicate what happens during a transient, however. The computational *a posteriori* error bounds suggest that meaningful computation is possible for long times, including both transients and metastable periods. We discretize (24) in space using a second-order finite element method based on piecewise linear functions and lumped mass quadrature, resulting in the equation for the unknown U of dimension M:

$$\begin{aligned} U_t - \epsilon^2 AU &= F(U), \quad t > 0, \\ U(0) &= \text{given}, \end{aligned} \tag{25}$$

with

$$A = \begin{pmatrix} 1 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix} M^2 \text{ and } F(U) = (U_i - U_i^3)_i.$$

3.3 Parallel Speedup

We begin by addressing the practical computational issues, because this system is of large dimension and requires a lot of computational power and memory. The results for the bistable equation have been computed using workstations and also with an Intel Paragon parallel computer with up to 256 processors.

We assign each processor a consecutive subset of the M gridpoints, so that if there are P processors, each subset contains about M/P gridpoints. Since the bulk of the computation is with vectors of this size, the computational time is proportional to M/P. We let α denote the constant of proportionality representing the time taken per gridpoint.

The differential equation itself involves the one-dimensional discrete Laplacian (multiplication by the matrix A); to compute this, each gridpoint uses values from its neighbors so the resulting Jacobian is tridiagonal. To evaluate the discrete Laplacian in parallel, each processor (except those at the ends of the interval) communicates with its neighbor on either side. For this one-dimensional case, the extra time β taken by this communication is independent of the number of processors or the number of gridpoints. In the case of a single processor, however, this communication term is absent.

An additional overhead comes from computing scalar products. Each processor computes the part of the scalar product from the gridpoints that it controls. Finally, a sum over all processors is computed, which takes a time that is logarithmic in the number of processors. We write this time as $\gamma \log_2 P$.

Thus, we expect the time taken by the inner loop of the one-dimensional computation to be:

$$T(M,P) = \alpha M/P + \beta + \gamma \log_2 P. \qquad (26)$$

The parallel machine will be significantly faster than a uniprocessor so long as the overhead caused by the communication is a small fraction of the computational time itself. In other words, the code is efficient if:

$$M/P >> \frac{\beta}{\alpha} + \frac{\gamma}{\alpha} \log_2 P. \qquad (27)$$

We now show the times taken for 60 iterations of the QMR solver on (25) versus the size of the system. There are nine curves in the figure, corresponding to different numbers of (Intel Paragon) processors being used. The qualitative content of this figure is that there is no point in using a massively parallel machine without a sufficiently large problem to solve. Only when there are a hundred or more gridpoints per processor is the calculation efficient.

The experimental results are in agreement with the model above if the parameters satisfy $\beta/\alpha = 50$, and $\gamma/\alpha = 25$. This means that the code is efficient so long as the number of gridpoints per processor is significantly greater than $50 + 25 \log_2 P$.

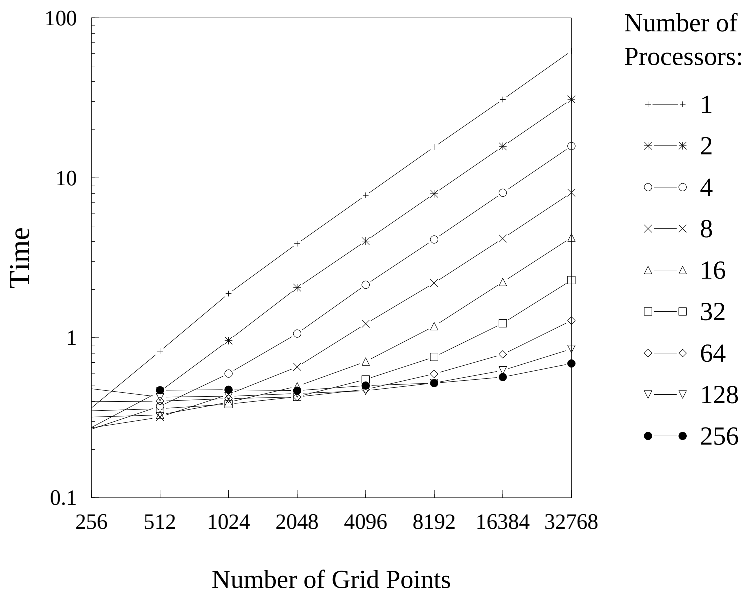

Figure 5: Times for a single time step on the Intel Paragon parallel computer

3.4 Computability of the Bistable Problem

We now discuss the *a posteriori* error analysis of numerical solutions to the one-dimensional bistable problem.

To study the different stability properties of the solution during metastable and transient times, we take initial data consisting of two "wells" of different thicknesses and of suitable shape so that the initial data is nearly metastable at the start. Namely, we take $\epsilon = 0.03$, and the initial condition

$$u_0(x) = \begin{cases} \tanh((.2 - x)/(2\epsilon)), & 0 \le x < .28, \\ \tanh((x - .36)/(2\epsilon)), & .28 \le x < .4865, \\ \tanh((.613 - x)/(2\epsilon)), & .4865 \le x < .7065, \\ \tanh((x - .8)/(2\epsilon)), & .7065 \le x \le 1. \end{cases}$$

In Figure 6a, we plot the evolution of the approximation from this initial condition. The left well is slightly thinner than the right and collapses by the sides coming together around time 41, while the well on the right collapses at time 141. The solution exhibits metastability during the time before 41 and between the two times. In Figure 6b, we plot the stability factor $S_1(t)$ reflecting the sensitivity of the solution to numerical approximation. In this case, $S_1(t)$ is of order one except during the transient periods, where it rises to 100 or so. This means that the solution can be accurately approximated with residual tolerances on the order of 10^{-3} or less.

The stability factor in Figure 6 shows two sharp peaks, one at each transient. The stability factor is quite small between the transients when the solution is almost stable with respect to numerical error. There is a sharp increase leading up to the transients; this growth is even faster than exponential. After the transient, the stability factor drops precipitously,

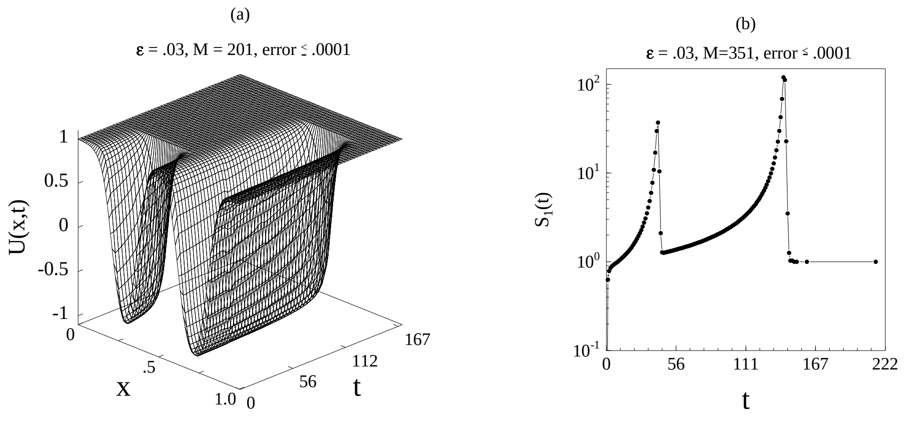

Figure 6: Evolution of a bistable solution and its stability factor

indicating that the subsequent solution is essentially independent of any previous error accumulation. When the solution finally converges to the uniform equilibrium state, the stability factor is one, and all previous error due to accumulation is removed.

In Figure 7, we show the effects of changing the residual tolerance. When the residual tolerance is 10^{-1} and 10^{-2}, (which give "accuracies" on the order of 500% and 50%, respectively), the second transient occurs at a later time. This is visible in the stability factor and the decrease in time steps during the second transient. We also note that decreasing the residual tolerance appears to cause a smooth decrease in the time steps used, an important property if systematic numerical experimentation is to be performed.

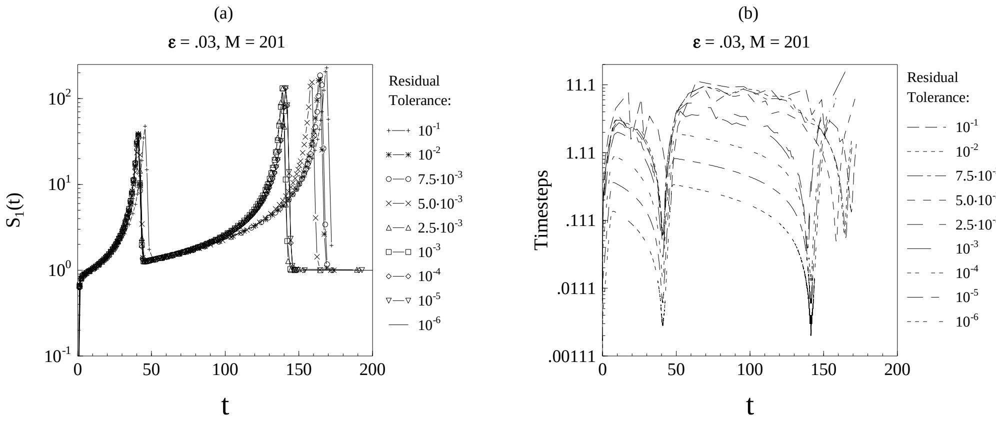

Figure 7: Effect of changing the residual tolerance in the bistable problem with $M = 201$

Next, we show the effect of choosing different initial directions for the dual computations in Figure 8. Ideally, we would use the direction of the (unknown) error at time t_n to compute the stability factor for that time. The stability factor measures the accumulation of error

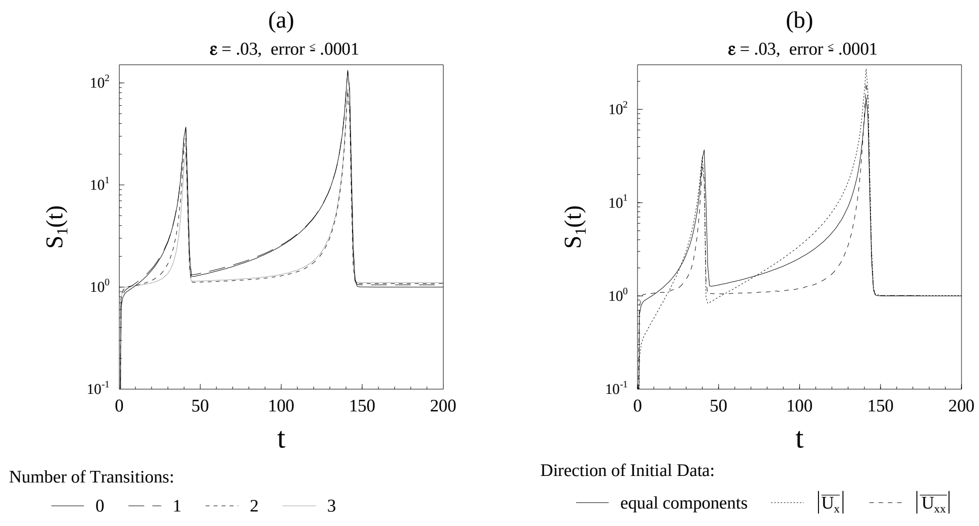

Figure 8: Effect of changing initial data in the dual problem for the bistable problem with $M = 201$

associated to that particular direction.

Conjecture 2 of Section 2.4.4 is that in many problems, the magnitude of the stability factor is relatively insensitive to the choice of initial data. Consider the case when the Jacobian of the system is a constant matrix. For generic initial data, the solution of the backward problem will rotate and point in the direction associated to the most unstable (or least stable) eigenvalue of the Jacobian. This in turn implies that most data will produce the same size stability factors, provided sufficient time has passed (the time scale depends on the distribution of eigenvalues). Only initial data that have no component in the most unstable mode will act differently. In the general non-autonomous case, the eigenvalues of the Jacobian matrix do not determine the stability factors, but rather it is the Lyapunov characteristic numbers. These are defined as limits over infinite time of average logarithmic growth rates of perturbations of solutions. We believe that an analogous analysis can be performed; however, the meaning of "sufficiently long time intervals" is not as clear. Certainly, variations in the stability factors result from varying the initial data, and the issue is whether this affects the overall error control.

The stability factor corresponding to the direction of the error at the current time can be bounded by the maximum of the stability factors computed for all directions at that time. For low-dimensional systems, it is feasible to compute this bound explicitly, however for large systems it is not. In the latter case, we can increase reliability by computing stability factors for several choices of initial data. We choose four arbitrary initial directions for the backward problem and plot the resulting stability factors in Figure 8a. The first uses a unit vector with all components equal, labelled as "0 transitions". The computation labelled "1 transition" has the first half of the components in the direction of $1/\sqrt{M}$ and the second half in the direction of $-1/\sqrt{M}$. The data labelled "2 transitions" and "3 transitions" are constructed similarly: unit vectors with components of equal magnitude but alternating sign. We see that the stability factors are roughly the same size, though there are variations in the metastable periods. On the other hand, the peak sizes of all the stability factors are very close. Since this coincides with the period of largest residual error, all four factors yield nearly the same residual tolerances used to achieve the same global error.

The correct initial data for the backward problem is the unknown true error, but it may be possible in some problems to determine a reasonable substitute for this. In the case of the bistable equation, it seems natural to conjecture that the error will be largest in the transition layers. In Figure 8b, we plot stability factors for three choices of initial data. The first is the stability factor corresponding to 0 transitions above. The second is a vector $\overline{U_x}$ whose i^{th} component is the average of $|U_x|$ in the elements on either side of the i^{th} node normalized to be a unit vector. The third choice is a vector $\overline{U_{xx}}$ whose i^{th} component is the discrete Laplacian matrix applied to U and normalized to be a unit vector. We see that as above, there is variation in the stability factors during the metastable regime, however the peak heights are nearly equal.

As the dimension M is decreased, the ODE system ceases to adequately represent the continuum behavior of the PDE, and solutions of the ODE system have qualitatively different time behavior than solutions of the PDE system. In particular, the ODE system undergoes "locking", which means that solutions that have the appearance of metastable solutions actually become stable. To illustrate the effects of this on the error control, we present results for a variety of M. In Figure 9a, we show $S_1(t)$ for various computations. In all cases, the smaller well collapses. However, when $M = 21$ the collapse occurs much sooner than for the other values of M, which is reflected in the stability factor. On the other hand, when $M = 21$, the second well becomes fixed for all time and the stability factor correspondingly remains 1. We note that once M is sufficiently large to prevent locking, the stability factor is relatively insensitive to M. This suggests that a coarse interpolant of the true solution could be used in the backward computation, an idea that we plan to explore in future work. In Figure 9a, we plot the time steps versus time for computations

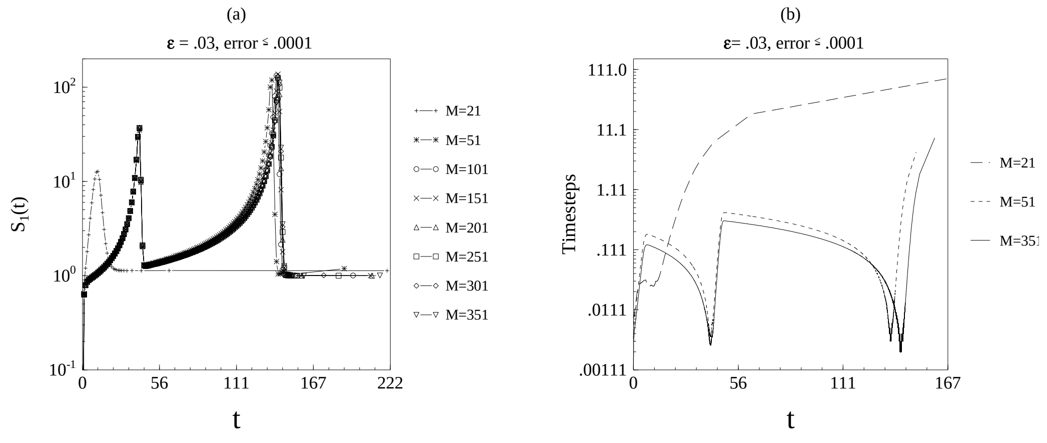

Figure 9: Effect of changing the (space) dimension in the bistable problem

with $M = 21$, $M = 51$, and $M = 351$. Since the solution locks when $M = 21$, we see that the time steps increase steadily as time passes, while for larger M, two sharp dips occur during the transients, with the time steps decreasing by a couple of orders of magnitude.

3.5 The Bistable Equation in Two Dimensions

In two dimensions, the problem reads

$$\begin{cases} u_t - \epsilon^2 \Delta u = u - u^3, & (x,y) \in \Omega, 0 < t, \\ \text{periodic boundary conditions}, & 0 < t, \\ u(x,y,0) = u_0(x,y), & (x,y) \in \Omega, \end{cases} \tag{28}$$

where $\Omega = [0,1] \times [0,1]$. As in the one-dimensional bistable equation, solutions starting from generic initial data evolve to form thin $O(\epsilon)$ transition layers separating regions of $u \approx \pm 1$. All solutions converge to one of the two stable equilibria, $u \equiv 1$ and $u \equiv -1$. In this case, the evolution is "motion by mean curvature" [3], [5], meaning that the normal velocity of a transition layer is of order κ/ϵ^2, where κ indicates the sum of the layer's principal curvatures. Thus, the time scales in this problem are much shorter than the exponential time scales ($O(e^{1/\epsilon})$) of the evolution of metastable solutions in one dimension.

We made a computation starting from the initial condition consisting of two circular "mesas":

$$u_0(x,y) = \left\{ \begin{array}{ll} 1, & \|(x,y) - (.25,.25)\| \leq .15, \\ 1, & \|(x,y) - (.75,.75)\| \leq .30, \\ -1, & \text{otherwise.} \end{array} \right\}$$

We compute with $\epsilon = 1/60$, $M = 64 \times 64$, and the computation has reported accuracy $\leq .0001$ using 16 processors of an Intel Paragon.

We plot the solution at four different times in Figure 10. At time $t = 18$, we see the two mesas with flat tops and steep, thin sides. The smaller one has already begun to leave the regime of motion by mean curvature because its radius is no longer large compared to ϵ. It will shortly disappear (at time $t = 43$). In the second panel at $t = 54$, the smaller mesa is gone. At $t = 144$, the remaining mesa is close to disappearing, which it does at $t = 158$. At $t = 180$, we see the solution has converged to $u \equiv 1$.

In Figure 11, we plot the three stability factors during the evolution. The left plot is $S(t)$ reflecting the effect of error in the initial conditions; the middle plot is $S_1(t)$ reflecting the accumulation of discretization error; and the right plot is $S_0(t)$ reflecting the accumulation of quadrature error. We see that $S(t)$ is essentially constant when the evolution is motion by mean curvature, whereas $S_0(t)$ and $S_1(t)$ show approximately linear growth during the same periods. During the transients when the mesas collapse, the error growth rate becomes superlinear, so that the stability factors reach a sharp peak. After both transients have passed, $S(t)$ tends rapidly to zero since the solution is now insensitive to small perturbations in the initial conditions. Similarly, $S_0(t)$ (quadrature error) also becomes very small; the solution is no longer evolving, so that integration by quadrature is exact. On the other hand, $S_1(t)$ achieves its minimum of 1, meaning that the error is bounded by the residual tolerance.

We conclude by using the stability factors to contrast the dynamics of the one-dimensional and two-dimensional cases. In Figure 12, we plot $S_1(t)$ for the one-dimensional computation with $\epsilon = .03$ and $M = 201$ using the two well data of Section 3.4 and $S_1(t)$ for the two-dimensional computation just described. We chose ϵ and the initial conditions so that the time of evolution was roughly the same for each computation. $S_1(t)$ in the one-dimensional case has periods of superexponential growth as compared to the weakly super-linear growth of the two-dimensional case, and consequently reaches a much higher

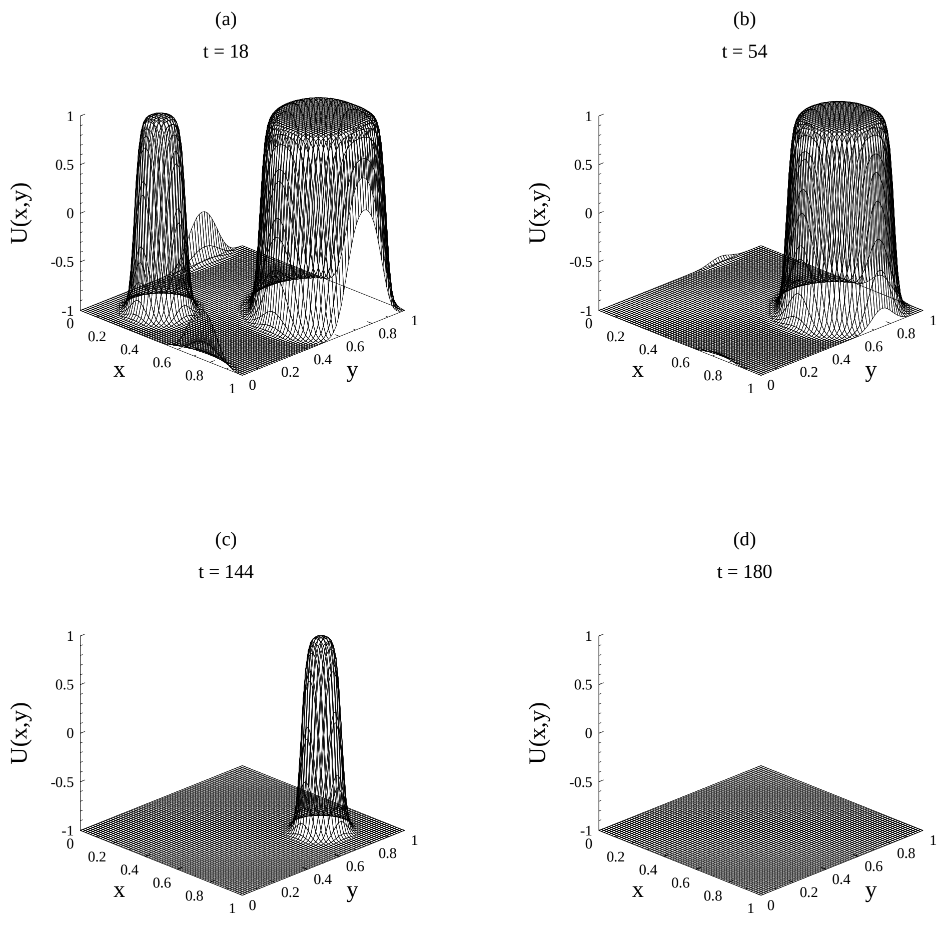

Figure 10: Evolution of a solution of the bistable equation in two dimensions

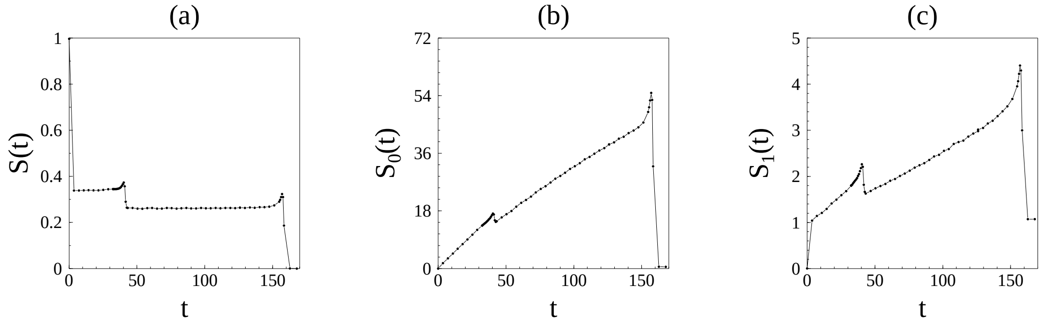

Figure 11: Stability factors for the bistable equation in two dimensions

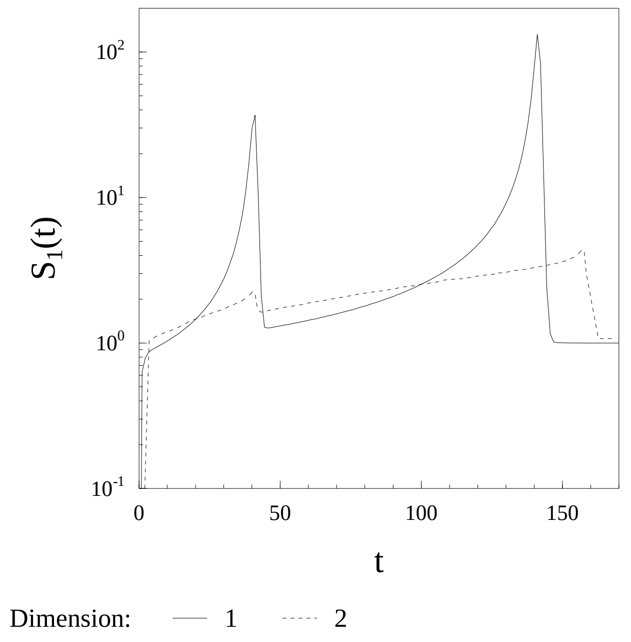

Figure 12: Stability factors for the bistable equations in one and two dimensions

value. We conclude that the one-dimensional case is much more sensitive to perturbations than the two-dimensional case.

4 Conclusions

In this paper, we described our implementation of a theory of *a posteriori* error control for the numerical integration of both large sparse and small dense ODE problems. The software is robust, modular, and efficient on machines ranging from a workstation to massively parallel processors.

We provided evidence that the error estimate is accurate and robust using the three-dimensional Lorenz system, and also gave evidence that the fundamentally important stability factors may be accurately approximated using the computed solution rather than the exact solution.

Next, we investigated the one-dimensional bistable problem and demonstrated that reasonable parallel speedup of the code with up to 32768 points on 256 processors of an massively parallel processor is possible. We studied a metastable solution that has two slowly evolving metastable periods punctuated by fast transients before it converges to a steady state. We not only computed the solution, but quantified the accuracy of the computations using the *a posteriori* estimates and by computing the stability factors. We gave further evidence that the stability factors themselves can be reliably and robustly approximated.

The last experiment we presented showed that we can compute using the code on a larger system of dimension 4096 while quantifying the accuracy using *a posteriori* error bounds. The behavior of this system is qualitatively different from its one-dimensional counterpart, and we illuminated the differences by comparing the stability factors.

4.1 Improvements to the ODE Solver

We plan several improvements and extensions of this work in the future.

It is expensive to require storage of the full history of the approximation. We will investigate ways of reducing the storage requirement using adaptive spline interpolants to compress the history data.

We shall implement higher-order Galerkin methods and change the adaptive algorithm to allow the code to choose between different methods and quadrature formulas, depending on the convergence rate and stability properties of the approximation.

We shall consider different choices of norm in analyzing the error representation and incorporating the convergence criteria for the nonlinear and linear solvers into the step size selection mechanism.

4.2 Toward an Accurate Adaptive Parallel PDE Solver

When solving a PDE, it is wise to adapt spatially by refining the mesh when new features require extra resolution, or by coarsening the mesh when complexity has reduced to save machine resources. Such changes to the discretization of the PDE may be from topological changes to the mesh (h-refinement), or from changing the nature of the finite-element basis associated with elements of the mesh (p-refinement).

Such mesh refinement and coarsening requires a considerable quantity of software support, especially so when it occurs on a distributed-memory parallel machine [24] [25]. There is a difficult optimization problem to solve in the assignment of mesh entities to processors known as the 'load-balance problem' [23]; in the context of the ODE solver as discussed above, it is a question of which basis vectors (of the phase-space of the solution) are owned by which processor.

In addition to further work on the ODE solver, we hope to implement such a parallel space-adaptive PDE solver for reaction-diffusion equations.

Obtaining the Software

The code discussed in this paper and used for the computations is available on the Internet. The anonymous ftp address is `ftp.ccsf.caltech.edu`, and the program is in the file `/roy/cards/cards-1.0.tar.Z`. There is a World Wide Web site including the software and other material at `http://www.ccsf.caltech.edu/~roy/cards/`.

Acknowledgments

The work of D. Estep is supported by the National Science Foundation, contract numbers DMS-9208684 and INT-9302016. The work of R. D. Williams is supported by Caltech, the Concurrent Supercomputing Consortium, and the Center for Research on Parallel Computation (a National Science Foundation Science and Technology Center, Grant 292-3-51393 under NSF CCR-8809615).

Appendix A

We partition $\mathbb{R}^+$ into $0 = t_0 < t_1 < \cdots$, letting $k_m = t_m - t_{m-1}$ denote the time step for the interval $I_m = (t_{m-1}, t_m]$, $k = \max k_m$, and $\mathcal{P}^q(I_m)$ the space of polynomials of degree q and less on I_m. The dG method is based on using the space of discontinuous piecewise polynomials for both the space of approximation and the test space. Since these functions possibly have two values at each node t_m, we use the notation $U_m^+ = \lim_{t\downarrow t_m} U(t)$, $U_m^- = \lim_{t\uparrow t_m} U(t)$ and $[U]_m = U_m^+ - U_m^-$. For $1 \le m$, the dG approximation $Y \in \mathcal{P}^q(I_m)$ satisfies

$$\textstyle\int_{t_{m-1}}^{t_m} (\dot{Y}, V)\, dt + (Y_{m-1}^+, V_{m-1}^+) = \int_{t_{m-1}}^{t_m} (f(Y(t), t), V(t))\, dt + (Y_{m-1}^-, V_{m-1}^+), \qquad \text{for all } V \in \mathcal{P}^q(I_m), \tag{29}$$

while $Y_0^- = y_0$. For $q = 0$, (29) reduces to

$$Y_m^- = \int_{I_m} f(Y_m^-, t)\, dt + Y_{m-1}^-, \tag{30}$$

which is recognizable as a variant of the backward Euler scheme. The cG method uses the space of continuous piecewise polynomials, while the test functions are discontinuous piecewise polynomials of one less degree, since enforcing continuity takes one degree of freedom on each interval. For $m \ge 1$, the cG approximation $Y \in \mathcal{P}^q(I_m)$ satisfies

$$\int_{t_{m-1}}^{t_m} (\dot{Y}, V)\, dt = \int_{t_{m-1}}^{t_m} (f(Y(t), t), V(t))\, dt, \qquad \text{for all } V \in \mathcal{P}^{q-1}(I_m), \tag{31}$$

while $Y_0 = y_0$. For $q = 1$, (31) reduces to

$$Y_m = \int_{I_m} f(Y(t), t)\, dt + Y_{m-1}, \tag{32}$$

where

$$Y|_{I_m} = Y_{m-1} \frac{t - t_m}{-k_m} + Y_m \frac{t - t_{m-1}}{k_m}. \tag{33}$$

For the dG method, the best choice of interpolatory quadrature rule uses the Gauss points in each interval, as this choice satisfies all of our criteria. For example, for $q = 0$ we use

$$\int_{I_m} f(Y_m^-, t)\, dt \to f(Y_m^-, t_{m-1} + k_m/2) k_m, \tag{34}$$

while for $q = 1$,

$$\begin{aligned}\int_{I_m} f(Y(t), t)\, dt \to{}& f\left(Y\left(t_{m-1} + k_m \frac{\sqrt{3}-1}{2\sqrt{3}}\right), t_{m-1} + k_m \frac{\sqrt{3}-1}{2\sqrt{3}}\right) \frac{k_m}{2} \\ &+ f\left(Y\left(t_{m-1} + k_m \frac{\sqrt{3}+1}{2\sqrt{3}}\right), t_{m-1} + k_m \frac{\sqrt{3}+1}{2\sqrt{3}}\right) \frac{k_m}{2}.\end{aligned} \tag{35}$$

For the cG $q = 1$ method, we employ the trapezoidal rule

$$\int_{I_m} f(Y(t), t)\, dt \to f(Y_{m-1}, t_{m-1}) \frac{k_m}{2} + f(Y_m, t_m) \frac{k_m}{2}. \tag{36}$$

Suitable quadratures can be derived for all orders.

References

[1] Bank, R. and Chan, T., *An analysis of the composite step bi-conjugate gradient method*, Numer. Math., 66 (1993), 295-319.

[2] Bertsch, M., Peletier, L. A., and Lunel, S. M. V., *The effect of temperature-dependent viscosity on shear-flow of incompressible fluids*, SIAM J. Math. 22 (1991), 328-343.

[3] Bronsard, L. and Kohn, R. V., *Motion by mean-curvature as the singular limit of Ginzburg-Landau dynamics*, J. Diff. Eq. 90 (1991), 211-237.

[4] Brown, P. N. and Hindmarsh, A. C., *Matrix-free methods for stiff systems of ODE's*, SIAM J. Numer. Anal., 23 (1986), 610-638.

[5] de Mottoni, P. and Schatzmann, M., *Évolution géometrique d'interfaces*, C. R. Acad. Sci. Paris Ser. I Math. 309 (1989), 453-458.

[6] Eriksson, K., Estep, D., Hansbo, P., and Johnson, C., *Adaptive Finite Element Methods*, North-Holland, Inc, to appear.

[7] Eriksson, K. and Johnson, C., *Adaptive finite-element methods for parabolic problems 1. A linear model problem*, SIAM J. Num. Anal., 28 (1991), 43-77.

[8] Eriksson, K. and Johnson, C., *Adaptive finite element methods for parabolic problems IV: nonlinear problems* preprint #1992-44, Chalmers University of Technology, 1992.

[9] Estep, D., *A posteriori error bounds and global error control for approximations of ordinary differential equations.* SIAM J. Numer. Anal. 32 (1995), 1–48.

[10] Estep, D., *An analysis of numerical approximations of metastable solutions of the bistable equation*, Nonlinearity 7 (1994), 1445–1662 .

[11] Estep, D. and French, D., *Global error control for the continuous Galerkin finite element method for ordinary differential equations*, RAIRO Modél. Math. Anal. Numér. 28 (1994), 815–852.

[12] Estep, D. and Johnson, C., *The computability of the Lorenz system.* submitted to J. Comput. Physics.

[13] Estep, D. and Stuart, A., *The dynamical behavior of the Galerkin method for ordinary differential equations and related difference schemes.* in preparation.

[14] Estep, D. and Johnson, C., *An analysis of quadrature in Galerkin finite element methods for ordinary differential equations*, in preparation.

[15] Fox, G. C., Williams, R. D., and Messina, P. C., *Parallel Computing Works!*, Morgan-Kaufman, Los Altos, California, 1994.

[16] Freund, R. and Nachtigal, N., *QMR: A quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math., 60 (1991), pp. 315-339.

[17] Gear, C. W. and Saad, Y., *Iterative solution of linear equations in ODE codes*, SIAM J. Sci. Stat. Comp., 4 (1983), 583-601.

[18] Johan, Z., Hughes, T.J., and Shakib, F., *A globally convergent matrix-free algorithm for implicit time-marching schemes arising in finite element analysis in fluids.* Comput. Meth. Appl. Mech. Engr. 87 (1991), 281-304.

[19] Meade, D. and Milner, F. A., *S-I-R epidemic models with directed diffusion*, in *Mathematical Aspects of Human Diseases*, G. Da Prato (ed.), Appl. Math. Monographs 3, Giardini Editori, Pisa, 1992.

[20] Pearson, J. E., *Complex patterns in a simple system*, Science 261 (1993), 189-192.

[21] Saad, Y., and Schultz, M., *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems,* SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856-869.

[22] Wheeler, A. A., Murray, B. T., and Schaefer, R. J., *Computation of dendrites using a phase field model*, Physica-D, 66 (1993), 243-262.

[23] Williams, R. D., *Performance of Dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency, 3 (1991), 457.

[24] Williams, R. D., *Voxel databases: A paradigm for parallel computing with meshes*, Concurrency, 4 (1992), p. 619.

[25] Williams, R. D., *DIME: Distributed Irregular Mesh Environment*, Source code and documentation available from `ftp://ftp.ccsf.caltech.edu/dime/`.