A BVP SOLVER BASED ON RESIDUAL CONTROL AND THE MATLAB PSE

JACEK KIERZENKA* AND LAWRENCE F. SHAMPINE†

Abstract. Our goal was to make it as easy as possible to solve a large class of boundary value problems (BVPs) for ordinary differential equations in the MATLAB problem solving environment (PSE). We present here theoretical and software developments resulting in `bvp4c`, a capable BVP solver that is exceptionally easy to use.

1. Introduction. Our goal was to make it as easy as possible to solve a large class of boundary value problems (BVPs) for ordinary differential equations (ODEs) in the MATLAB problem solving environment (PSE). We present here theoretical and software developments resulting in the BVP solver `bvp4c`. It solves first order systems of ODEs

$$y' = f(x, y, p), \qquad a \le x \le b \tag{1.1}$$

subject to two-point boundary conditions

$$g\left(y\left(a\right), y\left(b\right), p\right) = 0 \tag{1.2}$$

that can depend on a vector p of unknown parameters. It is unusual to allow problems with non-separated boundary conditions. Among general-purpose BVP solvers, the valuable capability of unknown parameters is unusual. Despite the generality of the problems allowed, `bvp4c` does not require users to provide analytical partial derivatives, a great convenience that is also unusual.

The numerical method of `bvp4c` can be viewed as collocation with a C^1 piecewise cubic polynomial $S(x)$ or as an implicit Runge-Kutta formula with a continuous extension (interpolant). The formula has been studied by several authors and implemented in a number of codes [3, 5, 9, 12, 13]. However, the codes evaluate the formula in different ways, define error in different ways, use different algorithms to estimate the error and select the mesh, and provide answers in different forms. We base error estimation and mesh selection on the residual of $S(x)$. Control of the size of the residual is unusual for BVPs, but Enright and Muir [9, 10] point out that it has important advantages. One of particular interest to us is its potential for coping with poor guesses for the mesh and the solution. Enright and Muir use the same formula, but a different interpolant in their solver MIRKDC. The residual is defined in terms of the interpolant and we show that the one we use has important advantages. We prove that the behavior of its residual is asymptotically local and depends only on a derivative of the solution. We develop inexpensive estimates of both L_∞ and L_2 norms of the residual that we prove to be asymptotically correct. We argue that the estimate of the L_2 norm is credible even when the asymptotic behavior of the residual is not evident. An interesting aspect of this estimate and our numerical method is that the acceptance test on the size of the residual automatically takes into account how well the collocation equations are satisfied.

This paper presents the theory of a solver that appears in MATLAB 6 (Release 12). In addition, we explain here how to exploit further the PSE to solve BVPs faster.

*The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760 (jkierzenka@mathworks.com)
†Math. Dept., SMU, Dallas, TX 75275. (lshampin@mail.smu.edu)

Hereafter we refer to the release version as `R12bvp4c` and the new one as `bvp4c`. By default `R12bvp4c` uses finite difference Jacobians. We reduce the cost of forming Jacobians in `bvp4c` by determining whether it is possible instead to use an averaged Jacobian. `bvp4c` is vectorized and has a new option that allows a user to reduce the run time significantly by vectorizing the evaluation of $f(t, y, p)$.

Some numerical examples illustrate the software and algorithmic developments of `bvp4c`. Other examples compare the standard interpreted MATLAB code with compiled code. Included in this comparison are both MIRKDC and a compiled version of `bvp4c`.

2. Simpson Method. Solving BVPs in a PSE is different from solving them in general scientific computing (GSC). Some of these differences direct our attention to certain kinds of formulas and implementations. A very important difference is that in a PSE, most solutions are viewed graphically, though we must be able to obtain accurate answers anywhere in $[a, b]$. This implies that some popular approaches are not suitable: Formulas of quite high order are popular, but a modest order is more appropriate at graphical accuracy. Some popular methods provide answers only at mesh points. Others provide answers everywhere, but the accuracy is not uniform. In this section we describe a method that satisfies our requirements and state some of its basic properties. Later we prove some new results that show it to be especially suitable for our purposes.

We assume throughout that the functions $f(x, y, p)$ and $g(u, v, p)$ of (1.1,1.2) are as smooth as necessary. In particular, f is continuous and satisfies a Lipschitz condition in y. For simplicity we suppress the argument p. The method we investigate can be viewed in several ways. Dickmanns and Well [8] view it as collocation with a piecewise cubic polynomial function $S(x)$. $S(x)$ satisfies the boundary conditions and for each subinterval $[x_i, x_{i+1}]$ of a mesh $a = x_0 < x_1 < \ldots < x_N = b$, it is a cubic polynomial that collocates at the ends of the subinterval and the midpoint. It is continuous at the end points of each subinterval, which along with collocation there implies that $S(x) \in C^1[a, b]$. This collocation method is equivalent to the 3-stage Lobatto IIIa implicit Runge-Kutta formula. When applied to a quadrature problem, it reduces to the Simpson formula, so we call it the *Simpson method*. Cash [4] draws attention to the method because its global discretization error contains only even powers of the step size, making it attractive for extrapolation and deferred correction. He points out that the formula can be evaluated efficiently using analytical condensation. Specifically, if $y_i = S(x_i) \approx y(x_i)$ and $h_i = x_{i+1} - x_i$, then

$$
\begin{aligned}
y_{i+1} = y_i &+ \frac{h_i}{6}(f(x_i, y_i) + f(x_{i+1}, y_{i+1})) \\
&+ \frac{2h_i}{3} f(x_i + \frac{h_i}{2}, \frac{y_i + y_{i+1}}{2} - \frac{h_i}{8}(f(x_{i+1}, y_{i+1}) - f(x_i, y_i)))
\end{aligned}
$$

The formula is viewed as a mono-implicit method in [2] and implemented in HAGRON [3]. It is used as the basic formula for evaluating higher order formulas by deferred correction in TWPBVP [6, 5]. It is viewed as a boundary value Runge-Kutta method in [12] and implemented in ABVRKS. All these codes implement the Simpson method as one of a family of finite difference methods and provide solutions only at mesh points. Other codes use continuous extensions (interpolants) to obtain solutions throughout $[a, b]$. We refer to the C^1 cubic spline used by Dickmanns and Well as the natural interpolant and denote it by $S(x)$. The Simpson formula is implemented with the natural interpolant in RWA2 [13]. It is implemented with a different interpolant

in MIRKDC [9]. This interpolant is a piecewise quartic polynomial function that is $C^1[a,b]$.

The text [1] uses the Simpson formula to illustrate its convergence results for implicit Runge-Kutta methods. It is shown that if $h = \max h_i$ and the mesh is such that h/h_i is bounded above, the formula is uniformly fourth order accurate at all the collocation points. If we let $y_i' = S'(x_i)$, it follows from this result and the Lipschitz condition on f that

$$y_i' - y'(x_i) = f(x_i, y_i) - f(x_i, y(x_i)) = O\left(h^4\right)$$

The text also shows that the collocation polynomial (the natural interpolant) $S(x)$ and its derivatives satisfy

$$S^{(j)}(x) = y^{(j)}(x) + O\left(h^{4-j}\right), \qquad j = 0, 1, 2, 3 \tag{2.1}$$

uniformly for x in $[a,b]$.

3. Residual. A serious practical difficulty with finite difference and collocation codes is finding a mesh for which the asymptotic arguments that justify error estimation and mesh selection are valid. Indeed, COLSYS [1] changes over to a more expensive error estimate for an acceptance test because the estimate used to select the mesh is not reliable when the mesh is coarse. This difficulty is especially serious when working at graphical accuracy with coarse meshes as we typically do in MATLAB. Enright and Muir [9, 10] point out advantages in basing these fundamental algorithms on the control of residuals, one being that it can be used both for mesh selection and for acceptance of a solution. We prove here that the Simpson formula with natural interpolant has some properties that make it a very attractive formula for a BVP solver based on control of the residual: To leading order the behavior of the residual is local, a result fundamental to mesh selection. We obtain an asymptotically correct estimate of an L_∞ norm of the residual with just one evaluation of the residual per subinterval and an asymptotically correct estimate of an L_2 norm in just two evaluations. The L_2 estimator takes into account how well the collocation equations are satisfied.

The residual of $S(x)$ in the differential equations is $r(x) = S'(x) - f(x, S(x))$, and the residual in the boundary conditions is $g(S(a), S(b))$. Put differently, $S(x)$ is a solution of the BVP

$$\begin{aligned} y' &= f(x,y) + r(x) \\ g(y(a), y(b)) &= g(S(a), S(b)) \end{aligned}$$

From the viewpoint of backward error analysis, $S(x)$ is a good solution to (1.1,1.2) when the residuals are small because it is the exact solution of a problem close to the one posed. When the residuals are small, the error in $y'(x)$ is also small: The Lipschitz condition on f and (2.1) imply that for $a \le x \le b$,

$$f(x, S(x)) = f(x, y(x)) + O\left(h^4\right)$$

hence that

$$\begin{aligned} r(x) &= (S'(x) - f(x, S(x))) - (y'(x) - f(x, y(x)) \\ &= S'(x) - y'(x) + O(h^4) \end{aligned}$$

Because the term $S'(x) - y'(x)$ is $O(h^3)$, the residual is asymptotically equal to the error in the first derivative. This term is $O(h^4)$ for the interpolant of MIRKDC. Accordingly, the residual of this interpolant is of higher order, but has a more complicated behavior. In choosing $S(x)$ for `bvp4c`, we gave up some accuracy to get a residual with a simple behavior, a behavior that makes possible an inexpensive and asymptotically correct estimate of the size of the residual.

In our implementation of the Simpson method, the residual is well-defined no matter what mesh is used and no matter how poorly the collocation conditions are satisfied. Further, the residual can be evaluated easily at any x, making it possible to compute a credible assessment of its size even when the mesh is coarse. We aim to develop algorithms based on control of the residual that will cope with poor guesses for the mesh and solution and still be effective when the asymptotic behavior is evident. To this end, we establish now the asymptotic behavior of $r(x)$.

Let $q(x)$ be the cubic polynomial interpolating $y(x)$ and $y'(x)$ at both ends of $[x_i, x_{i+1}]$. The Hermite representation of $q(x)$ is

$$q(x) = A_1(x)\, y(x_i) + A_2(x)\, y(x_{i+1}) + B_1(x)\, y'(x_i) + B_2(x)\, y'(x_{i+1})$$

where

$$A_1(x) = \frac{1}{h_i^2}(x - x_{i+1})^2 \left(1 + 2\,\frac{x - x_i}{h_i}\right), \quad B_1(x) = \frac{1}{h_i^2}(x - x_i)(x - x_{i+1})^2$$
$$A_2(x) = \frac{1}{h_i^2}(x - x_i)^2 \left(1 - 2\,\frac{x - x_{i+1}}{h_i}\right), \quad B_2(x) = \frac{1}{h_i^2}(x - x_i)^2(x - x_{i+1})$$

Subtracting $q(x)$ from a similar representation of $S(x)$ and differentiating leads to

$$\begin{aligned} S'(x) - q'(x) = {} & A_1'(x)\,(y_i - y(x_i)) + A_2'(x)\,(y_{i+1} - y(x_{i+1})) + \\ & B_1'(x)\,(y_i' - y'(x_i)) + B_2'(x)\,(y_{i+1}' - y'(x_{i+1})) \end{aligned}$$

It is easily seen that the $B_j'(x)$ are $O(1)$ which, along with $y_i' - y'(x_i) = O(h^4)$, leads to

$$S'(x) - q'(x) = A_1'(x)\,(y_i - y(x_i)) + A_2'(x)\,(y_{i+1} - y(x_{i+1})) + O(h^4)$$

The remaining terms are more difficult to assess because the $A_j'(x)$ are $O\left(h^{-1}\right)$. The fourth order Simpson formula

$$y_{i+1} = y_i + \frac{h_i}{6}\left(y_i' + 4y_{i+1/2}' + y_{i+1}'\right)$$

is satisfied by $y(x)$ with a local truncation error that is $O(h_i^5)$

$$y(x_{i+1}) = y(x_i) + \frac{h_i}{6}\left(y'(x_i) + 4y'(x_{i+1/2}) + y'(x_{i+1})\right) + O(h_i^5)$$

Subtracting these two equations and using the convergence results for the solution and its derivative at the collocation points, we find that

$$y_{i+1} - y(x_{i+1}) = y_i - y(x_i) + O\left(h^5\right)$$

Substituting this result into the expression for $S'(x) - q'(x)$ leads to

$$S'(x) - q'(x) = \left(A_1'(x) + A_2'(x)\right)(y_i - y(x_i)) + O\left(h^4\right)$$

The coefficients of the Hermite representation satisfy

$$A_1'(x) + A_2'(x) \equiv 0$$

hence

$$S'(x) - q'(x) = O\left(h^4\right)$$

Combining this with our earlier expression for the residual shows that on each subinterval

$$r(x) = q'(x) - y'(x) + O\left(h^4\right)$$

Interpolation theory [17] provides the representation

$$q'(x) - y'(x) = -\frac{1}{12} h_i^3 y^{(4)}(\zeta)\theta(\theta - 1)(2\theta - 1)$$

where $\theta = (x - x_i)/h_i$ and $\zeta \in (x_i, x_{i+1})$. It is convenient to expand the derivative about the midpoint to obtain finally

$$r(x) = -\frac{1}{12} h_i^3 y^{(4)}\left(x_{i+1/2}\right)\theta(\theta - 1)(2\theta - 1) + O\left(h^4\right)$$

We see that to leading order the residual depends on the problem only through $y^{(4)}(x)$ and that the behavior is local to each subinterval. It follows that

$$\|r(x)\|_i = C_1 h_i^3 \left\| y^{(4)}\left(x_{i+1/2}\right)\right\| + O(h^4)$$

For the L_∞ norm the constant C_1 is about 8.02×10^{-3} and for the L_2 norm, about 5.75×10^{-3}.

MIRKDC uses two samples (evaluations of r) per subinterval to estimate the L_∞ norm of the residual of its interpolant. It is not possible to obtain an asymptotically correct estimate in this way because the behavior of the residual depends on the problem [9]. The samples are taken at points that correspond to the local extrema of a simplified model of the residual. We have seen that the residual of $S(x)$ is asymptotically a cubic polynomial in θ. A little calculation shows that its maximum magnitude in $[0,1]$ is assumed at $\theta = (3 \pm \sqrt{3})/6$. Accordingly, a single sample taken at one of the points $x_i + \theta h_i$ provides an asymptotically correct estimate of the L_∞ norm of $r(x)$ on $[x_i, x_{i+1}]$.

We do not follow Enright and Muir in using the L_∞ norm for several reasons. We think that when the mesh is coarse and asymptotic behavior is not evident, a quadrature formula of a moderately high degree of precision provides a more credible estimate of an integral norm than the same number of samples provides for the maximum norm. Also, a solver based on an integral norm is more robust when the residual is not smooth. For the Simpson method with natural interpolant the choice matters only when the mesh is coarse and when f is not smooth because otherwise the L_∞ norm of $r(x)$ is asymptotically a constant multiple of the L_2 norm. On each subinterval $[x_i, x_{i+1}]$ we estimate

$$\|r(x)\|_i = \left(\int_{x_i}^{x_{i+1}} \|r(x)\|_2^2 \, dx \right)^{1/2}$$

with a 5-point Lobatto quadrature formula. We have seen that to leading order, $\|r(x)\|_2^2$ is a polynomial of degree 6. This quadrature formula has degree of precision 7, so we obtain an asymptotically correct estimate of the size of the residual. In the course of constructing $S(x)$ we evaluate the residual at both ends and at the midpoint of the subinterval, so this estimate requires only two additional evaluations of $r(x)$ per subinterval, the same as the scheme of MIRKDC.

When the Simpson formula is evaluated exactly, the residual is zero at both ends and the midpoint of each subinterval. In our implementation, the residual is zero at both ends, but at the midpoint its size measures how well the algebraic equations defining the method are satisfied. The residual at the midpoint of the computed $S(x)$ is available and we use it in the quadrature formula. In the next section we see how this takes into account the error made in evaluating the collocation equations.

4. Implementation. We have developed a code, `bvp4c`, based on the Simpson method with residual control for MATLAB. Our goal was to make it as easy as possible to solve a large class of BVPs. We believe that this means we cannot require users to supply analytical partial derivatives. Also, we believe that it is quite important to solve problems with unknown parameters and problems with general two-point boundary conditions. Although we want the solver to be efficient, speed is not a primary goal in MATLAB. We have taken advantage of experience with codes that implement variations of the Simpson method, specifically [3, 5, 9, 12, 13]. However, we handle some matters differently because computing in MATLAB differs from GSC in important ways. We use the Simpson method more effectively because of the theoretical developments of §3. More details about the issues and algorithms can be found in [14].

4.1. Evaluating the Formula. When the Simpson method is implemented as an implicit Runge-Kutta formula, the vector of unknowns includes the solution at all collocation points. The work required to solve this algebraic system can be reduced significantly by numerical or analytical condensation [1]. The approximate solution and its first derivative are then computed only at the mesh points, but $S(x)$ is a cubic polynomial between mesh points, so these values are all we need to evaluate it anywhere. We exploit the fact that in the condensed formulation of the Simpson method, the residual at the middle of each subinterval is directly related to the error of evaluating the formula.

The Simpson method applied to (1.1,1.2) with mesh $a = x_0 < x_1 < \ldots < x_N = b$ is evaluated by solving the algebraic equations

$$\Phi(X, Y) = 0 \tag{4.1}$$

where

$$X = [x_0, x_1, \ldots, x_N]^T$$
$$Y = [y_0, y_1, \ldots, y_N, p]^T$$

$$\Phi_0(X, Y) = g(y_0, y_N, p)$$
$$\Phi_i(X, Y) = y_i - y_{i-1} - \frac{1}{6} h_{i-1}(f_{i-1} + 4f_{i-1/2} + f_i)$$

for $i = 1, 2, \ldots, N$, and

$$f_i = f(x_i, y_i, p)$$
$$f_{i-1/2} = f(x_{i-1} + \frac{h_{i-1}}{2}, \frac{y_{i-1} + y_i}{2} - \frac{h_{i-1}}{8}(f_i - f_{i-1}), p)$$

We solve (4.1) with a simplified Newton (chord) method, hence require the global Jacobian $\partial\Phi/\partial Y$. For a fine mesh, a large system of linear equations is solved at each iteration, making the structure of the Jacobian crucial both to storage and efficiency. These matters are discussed fully in [1], so here we comment briefly about matters particular to `bvp4c`. When there are no unknown parameters and the boundary conditions are separated, popular solvers treat $\partial\Phi/\partial Y$ as a banded or staircase matrix. There are then a variety of effective ways to store the matrices and solve the linear equations. This is the most important reason for popular codes excluding unknown parameters and non-separated boundary conditions. However, excluding such problems shifts the burden to the user who must prepare them for the solver. Because sparse matrix technology is fully integrated into MATLAB, it is natural and efficient to treat $\partial\Phi/\partial Y$ as a general sparse matrix. It is important to appreciate that relative costs in this PSE do not correspond to those of GSC. In particular, the built-in linear algebra functions execute much faster than interpreted code. Even if this were not so, allowing general boundary conditions and especially unknown parameters is such a convenience for users that it would justify the approach we take. A similar argument about relative costs disposes of an objection to analytical condensation of the Simpson method, namely the cost of multiplying partial derivatives of f in forming terms like

$$\frac{\partial\Phi_i}{\partial y_i} = I - \frac{h_{i-1}}{6}\frac{\partial f_i}{\partial y} - \frac{2}{3}h_{i-1}\frac{\partial f_{i-1/2}}{\partial y}\left(\frac{1}{2}I - \frac{h_{i-1}}{8}\frac{\partial f_i}{\partial y}\right)$$

It is possible to reduce this cost by evaluating the Simpson formula as a deferred correction to the trapezoidal rule. However, in MATLAB the straightforward approach is better because matrix multiplication is relatively fast.

By default we use `numjac` [16] to compute finite difference approximations to partial derivatives of f and g. The algorithm is unusually robust: It retains scale information from the approximation of one Jacobian to assist in the selection of increments for the next. When computing a column of the Jacobian, it monitors the change in function value and recomputes the column with a different increment if this appears necessary for a meaningful approximation. Some codes reduce the cost of forming Jacobians numerically by approximating some of them. In the case of the Simpson formula, Cash [4] replaces $J_{i-1/2}$ by J_{i-1} in $\partial\Phi_i/\partial y_{i-1}$ and replaces it by J_i in $\partial\Phi_i/\partial y_i$. However, Enright and Muir [9] report that the Newton process is more likely to converge and converges faster when the Jacobian is approximated directly at all the collocation points. In developing `R12bvp4c` we experimented with $J_{i-1/2} \approx (J_i + J_{i-1})/2$. Our experience with this approximation was like that of Enright and Muir, so `R12bvp4c` approximates directly $J_{i-1/2}$. `bvp4c` tests whether

$$\|J_i - J_{i-1}\|_1 \leq 0.25 * (\|J_i\|_1 + \|J_{i-1}\|_1)$$

If so, $J_{i-1/2}$ is approximated by the average of J_i and J_{i-1} and otherwise, it is approximated directly. With this test we use an average value only where the Jacobian is not changing rapidly. The test is made practical by the fast built-in function for computing norms. In experiment with a large set of test problems we found with one exception that judiciously replacing the Jacobian at the midpoint by an average did not increase the run time and often reduced the longer run times by 20%. In particular, all the examples for which always averaging resulted in unsatisfactory performance in our earlier experiments were solved successfully. The exception arises from the fact that when the collocation equations are linear in all the unknowns and all the Jacobians are evaluated directly, the Newton iteration converges in a single

step. Averaging Jacobians results then in a set of equations that must be solved iteratively. On the other hand, the solver does not average when analytical partial derivatives are supplied. These partial derivatives are generally very easy to supply for linear problems, so it is generally very easy to avoid the inefficiency of averaging when the collocation equations are linear. This inefficiency is less common than it might at first seem. For averaging to be counterproductive, not only must the ODEs be linear, but also the boundary conditions, and unknown parameters must appear linearly.

Like `bvp4c`, the MATLAB programs for solving stiff initial value problems (IVPs) form Jacobians by finite differences. Often it is easy to vectorize the evaluation of $f(x,y)$ for a given x and an array of vectors y. This is obviously valuable in forming a finite difference Jacobian, so the IVP solvers have an option for vectorized functions. This is a natural option for BVPs, too, but important additional gains are possible in `bvp4c` if the function is vectorized with respect to x as well. That is because in forming (4.1), all the arguments are known in advance. Generalizing what is done for IVPs, we added an option for users to code $f(x,y)$ so that it accepts $x = [x_1, x_2, ...]$ and $y = [y_1, y_2, ...]$ and returns an array $[f(x_1, y_1), f(x_2, y_2), ...]$. The benefits of vectorizing $f(x,y)$ depend on the BVP, but in our experiments we have found that together with judicious averaging of the Jacobian, run times were often reduced by 40% and more.

Because guesses for a mesh and solution may be poor, it is important to enhance the global convergence of the simplified Newton iteration. We follow Deuflhard [7] and Gladwell [11] in using a weak line search for this purpose. Deuflhard emphasizes the importance of scaling in the practical solution of nonlinear algebraic equations and proposes an affine-invariant convergence test to deal with this difficulty; we adopted his proposal in `bvp4c`. A new global Jacobian is formed and factored only when the convergence rate is unsatisfactory. Enright and Muir [9] suggest that if the Newton iterations converge slowly, it may be more efficient to stop iterating and redistribute the mesh. That has been our experience, too, so we limit the number of iterations to four.

Finite difference and collocation methods all evaluate a formula by solving algebraic equations like (4.1). If the approximation to $y(x)$ is to have a certain accuracy, it is usual to solve the algebraic equations rather more accurately, enough so that the error in evaluating the formula can be neglected. For instance, Enright and Muir found in experiment [10] with MIRKDC that a tolerance of 1% of the tolerance on the residual was satisfactory. In `bvp4c` we use 10%, but we have found a way to take into account the error in evaluating the formula. A little calculation shows that

$$\begin{aligned} S\left(x_{i+1/2}\right) &= \frac{1}{2}\left(y_i + y_{i+1}\right) - \frac{1}{8} h_i \left(f_{i+1} - f_i\right) \\ S'\left(x_{i+1/2}\right) &= \frac{3}{2}\frac{y_{i+1} - y_i}{h_i} - \frac{1}{4}\left(f_{i+1} + f_i\right) \end{aligned}$$

When these expressions are used in the residual,

$$\begin{aligned} r\left(x_{i+1/2}\right) = &\frac{3}{2}\,\frac{y_{i+1} - y_i}{h_i} - \frac{1}{4}\,\left(f_{i+1} + f_i\right) \\ &- f\left(x_{i+1/2}, \frac{1}{2}\left(y_i + y_{i+1}\right) - \frac{1}{8}\, h_i \left(f_{i+1} - f_i\right)\right) \end{aligned}$$

which is recognized as

$$r(x_{i+1/2}) = \frac{3}{2h_i}\Phi_{i+1}$$

As we construct $S(x)$ with analytical condensation of the implicit Runge-Kutta formula, the residual vanishes at both ends of $[x_i, x_{i+1}]$ and the residual at the midpoint is the only one that we seek to make zero. This value of the residual is used by the quadrature formula that estimates $\|r(x)\|_i$. In this way our test for accepting a solution takes into account how well the algebraic equations (4.1) are satisfied.

4.2. Mesh Selection. A popular approach to mesh selection redistributes mesh points with a global strategy based on inverse interpolation. It assumes that the mesh is sufficiently fine and f is sufficiently smooth. We experimented with the approach and found it unsatisfactory when the mesh is crude and f is only piecewise smooth. Problem 7 of [15] was illuminating. At $x = 1.5$ there is a jump in $y'(x)$ and the residual is $O(1)$. Because it uses a maximum norm, MIRKDC is not able to solve this BVP. However, even with an integral norm, we found the global strategy to be unsatisfactory and so developed a local strategy for `bvp4c`. We add equally-spaced mesh points to each subinterval with $\|r(x)\|_i > tol$. We have found it best not to introduce more than two mesh points at a time. When the residual exhibits its asymptotic behavior, $\|r(x)\|_i$ is $O(h_i^{7/2})$. Two additional mesh points would then reduce the residual by a factor of $3^{-7/2} \approx 2 \times 10^{-2}$, so we introduce two points when $\|r(x)\|_i > 100\,tol$ and otherwise, one. This is a reasonable approach to reducing the error when the asymptotic behavior is evident and a plausible approach for any mesh. To reduce the cost, we also remove points. This is done in a way that is plausible for any mesh and when the asymptotic behavior is evident, assures us that the residual will remain acceptable with the coarser mesh. One of our tactics is to replace three consecutive subintervals by two because the residual is then evaluated at new points, increasing the robustness of the scheme for assessing the size of the residual. Another is to test the consistency of the behavior of the residual on successive subintervals.

5. Examples. In this section we consider a couple of examples that show the design and algorithms of `bvp4c` make it easy to solve a large class of BVPs. In particular, they show this to be rather easier than with popular solvers for GSC. We have proposed several changes to the algorithms of `R12bvp4c` that make `bvp4c` faster. A third example shows that these changes can reduce run times significantly. It is possible to compile codes written in FORTRAN or C as MEX files for use in a MATLAB program. With this in mind, it is natural to ask how one of the solvers for GSC would compare to `bvp4c`. We did this with MIRKDC and compare it to `bvp4c`. There is a MATLAB compiler that first translates an M-file into C and then compiles it. We did this with `bvp4c` and compare the compiled code to the M-file.

Example 1.10 of [1] models the spread of measles with the differential equations

$$\begin{aligned}
y_1' &= \mu - \beta(t)\, y_1\, y_3 \\
y_2' &= \beta(t)\, y_1\, y_3 - y_2/\lambda \\
y_3' &= y_2/\lambda - y_3/\eta
\end{aligned}$$

Here $\beta(t) = 1575\,(1 + \cos 2\pi t)$ and the constants are given as $\mu = 0.02$, $\lambda = 0.0279$, and $\eta = 0.01$. The solution is to satisfy the periodicity conditions $y(0) = y(1)$. Solving this BVP is straightforward with `bvp4c` because it accepts general two-point

boundary conditions, but most solvers, including TWPBVP and MIRKDC, do not accept problems with non-separated boundary conditions. Introducing new variables to obtain a problem with separated boundary conditions is not hard, but it is some trouble for the user and has computational implications, c.f. §4.1.

The following program solves this BVP and provides the solution as Figure 5.1.

```
function sol = measles
solinit = bvpinit(linspace(0,1,5),[0.01; 0.01; 0.01]);
sol = bvp4c(@odes,@bcs,solinit);
x = linspace(0,1,100);
y = bvpval(sol,x);
plot(x,y(1,:)-0.07,'-',x,y(2,:),'--',x,y(3,:),'-.');
legend('y_1(t) - 0.07  ','y_2(t)','y_3(t)',4);

%====================================================
function dydx = odes(x,y);
beta = 1575*(1 + cos(2*pi*x));
dydx = [     0.02 - beta*y(1)*y(3)
         beta*y(1)*y(3) - y(2)/0.0279
           y(2)/0.0279 - y(3)/0.01   ];

function res = bcs(ya,yb)
res = ya - yb;
```

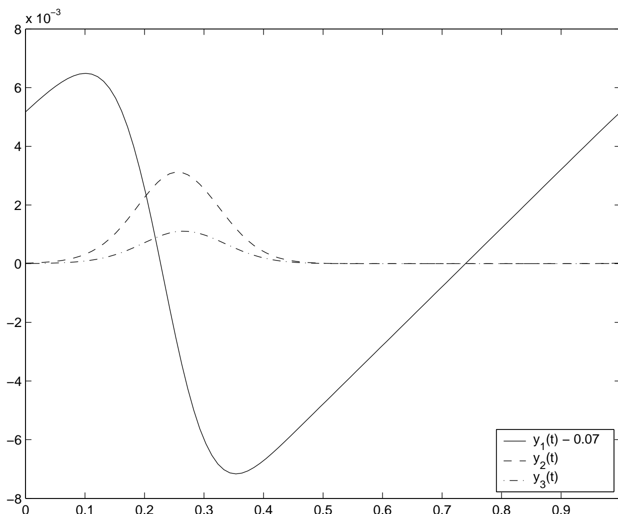

FIG. 5.1. *The measles problem.*

In simplest use, just about all you have to do to solve a BVP with **bvp4c** is define the problem. The ODEs and the boundary conditions are defined by providing functions for evaluating them. The ODEs are handled just as with the IVP solvers, so the subfunction `odes` requires no discussion. The boundary conditions subfunction `bcs` evaluates the residual $g(ya, yb)$ of arguments `ya` and `yb` that approximate $y(a)$ and $y(b)$, respectively. The solver makes no assumptions about the form of the boundary conditions, so from the user's point of view, there is nothing special about the periodic boundary conditions of this example.

BVPs can have more than one solution, so part of the definition of a BVP is a guess for the desired solution. It is provided to `bvp4c` as a structure that is usually formed with `bvpinit(x,v)`. The argument `x` is a guess for the mesh with either $a = x(1) < x(2) < \ldots < x(end) = b$ or $a = x(1) > x(2) > \ldots > x(end) = b$. The argument `v` provides a guess for $y(x)$ on $[a, b]$, either in the form of a function or a vector when the guess is constant. Here the guess structure `solinit` is formed with a mesh of five equally spaced points in $[0, 1]$ and a guess that all three components of the solution have the constant value 0.01.

The solution is returned by `bvp4c` as a structure called here `sol`. The code selects automatically an appropriate mesh that is returned in the field `sol.x`. Solution values are returned in the array `sol.y` with each column of `sol.y` corresponding to an entry of `sol.x`. This form of the output is convenient for plotting with the standard tools of MATLAB. For instance, all three components are plotted with `plot(sol.x,sol.y)`. `bvp4c` computes a solution $S(x)$ for the whole interval $[a, b]$. For some applications we need the solution at specific points, hence must evaluate $S(x)$ there. The cost of solving a BVP depends strongly on the number of mesh points, so `bvp4c` tries to use no more than necessary. Because of this, the values at mesh points alone may not provide a smooth graph. That is the case with this example. To get a smooth graph, we just need to evaluate $S(x)$ at more points. The function `bvpval(sol,xint)` evaluates a solution defined by the structure `sol` at all the points of the array `xint`. This capability contrasts sharply with BVP solvers for GSC cited earlier, including TWPBVP, that provide answers only on a mesh. Further, the continuous extension evaluated in `bvpval` is uniformly accurate and $C^1[a, b]$ in contrast to the $C^0[a, b]$ continuous extensions of COLSYS that have a (much) lower order of accuracy between mesh points. The Simpson method and encapsulation of the solution as a structure made it easy to provide the capability: In addition to the field `sol.y` that holds values of $S(x)$ at mesh points, there is a field `sol.yp` that holds values of $S'(x)$ there. For each entry ξ of `xint`, `bvpval` determines the subinterval for which $x_i \leq \xi \leq x_{i+1}$ and then evaluates the cubic polynomial $S(x)$ in Hermite form at $x = \xi$. Vectorization and built-in functions are used to reduce considerably the cost of evaluating $S(x)$ for long vectors `xint`.

As this example illustrates, `bvp4c` and its auxiliary functions `bvpinit` and `bvpval` make it easy to formulate a BVP, solve it, and plot the solution. Certainly the call list of `bvp4c` is remarkably short compared to the 24 arguments of TWPBVP and the 20 of MIRKDC. To achieve this, heavy use is made of defaults and features of the language. The MATLAB PSE allows us to relieve users almost entirely of the disagreeable and error-prone specification of sizes, types, and storage modes of arrays, especially working storage, that is typical of BVP solvers for GSC. Most BVP solvers, including TWPBVP and MIRKDC, require users to provide analytical partial derivatives, but `bvp4c` does not, a great convenience for the user. Like the IVP solvers of MATLAB, `bvp4c` provides for a scalar relative error tolerance and a vector of absolute error tolerances. This is more control of the error than provided by many BVP solvers, including TWPBVP and MIRKDC, but problems are typically solved to graphical accuracy in this PSE and the default values are generally satisfactory for this purpose. Optional information like the names of functions for evaluating analytical partial derivatives and error tolerances is specified in a convenient way using an optional (structure) argument. Because this is exactly like the IVP solvers, there is no need to discuss it here.

Example 1.4 of [1] describes flow in a long vertical channel with fluid injection:

$$f''' - R[(f')^2 - f\,f''] + R\,A = 0$$
$$h'' + R\,f\,h' + 1 = 0$$
$$\theta'' + P\,f\,\theta' = 0$$

Here R is a Reynolds number and $P = 0.7\,R$. The parameter A is unknown. Because of the presence of the unknown scalar A, there are eight boundary conditions

$$f(0) = f'(0) = 0, \quad f(1) = 1, \quad f'(1) = 1$$
$$h(0) = h(1) = 0, \quad \theta(0) = 0, \quad \theta(1) = 1$$

`bvp4c` accepts problems with unknown parameters, but most BVP solvers, including TWPBVP and MIRKDC, do not. Indeed, [1] uses this example to show how to prepare a BVP so that unknown parameters can be computed with solvers that do not provide for them. As with non-separated boundary conditions, this is not hard, but it is some trouble for the user and has computational implications, c.f. §4.1. Unknown parameters are more trouble than non-separated boundary conditions because after preparing the problem, the user of a code like TWPBVP or MIRKDC must also provide analytical partial derivatives of the ODEs and the boundary conditions with respect to the parameters. In designing `bvp4c` we considered the convenience of handling directly unknown parameters to be quite important because they are commonly introduced when preparing singular BVPs for numerical solution. Deciding to provide this capability reinforced our decision that `bvp4c` use numerical partial derivatives by default.

The following program solves this BVP for $R = 100$, provides the solution as Figure 5.2, and reports to the screen that $A = 2.7606$. As seen here, plotting solutions at just the mesh points typically provides a satisfactory graph.

```
function sol = injection
solinit = bvpinit(linspace(0,1,10),ones(7,1),1);
sol = bvp4c(@odes,@bcs,solinit);
A = sol.parameters
plot(sol.x,sol.y(2,:));

%=======================================================
function dydx = odes(x,y,A);
dydx = [ y(2); y(3); 100*(y(2)^2-y(1)*y(3)-A)
         y(5); -100*y(1)*y(5)-1; y(7); -70*y(1)*y(7) ];

function res = bcs(ya,yb,A)
res = [ ya(1); ya(2); yb(1)-1; yb(2)
        ya(4); yb(4); ya(6); yb(6)-1 ];
```

This program shows that unknown parameters cause very few complications. Naturally a guess must be supplied for the unknown parameters to identify the values of interest and to assist the solver in computing them. It is provided as a third (vector) argument of `bvpinit`. Here we guess that $A = 1$. The ODE and boundary conditions functions must have the vector of unknown parameters as an additional input argument, no matter whether the function uses any of the parameters. The values computed for the unknown parameters are returned in the solution structure `sol` as the field `sol.parameters`.

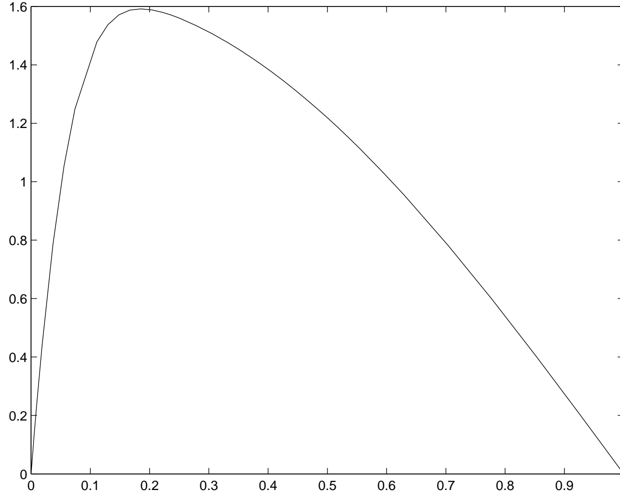

FIG. 5.2. *Flow in a vertical channel,* $R = 100$.

The `shockbvp` example provided with `R12bvp4c` solves

$$\epsilon y'' + xy' = -\epsilon\pi^2 cos(\pi x) - (\pi x) sin(\pi x)$$

with boundary conditions $y(-1) = -2$, $y(1) = 0$. This BVP is used in Example 9.2 of [1] to illustrate mesh selection. Details are provided there for COLSYS when computing the solution with $\epsilon = 10^{-4}$. To illustrate continuation, `shockbvp` solves this BVP by successively solving the problem with $\epsilon = 10^{-2}, 10^{-3}, 10^{-4}$. This is easy with `bvp4c` because a solution structure for one value of ϵ can be supplied as a guess structure for the next value. It also illustrates the use of analytical Jacobians to reduce run times. As with the previous examples, `shockbvp` uses the default tolerances of $RelTol = 10^{-3}$, $AbsTol = 10^{-6}$. We made two modifications for our experiments. To increase the run time, we did another step of continuation to solve the BVP with $\epsilon = 10^{-5}$. To include the costs of interpolation, we used `bvpval` to evaluate the solution at 100 equally spaced points in $[-1, 1]$ for the plot of Figure 5.3. All our experiments with this BVP were run on a PC with a Pentium II 450MHz processor and 128 MBytes of RAM. The run time of a function `shkbvp` was measured by issuing the command

```
>> clear all functions, shkbvp, tic, for i=1:10, shkbvp, end, toc
```

and dividing the elapsed time by 10.

First we solved the BVP with `R12bvp4c`. With default finite difference partial derivatives, this took 19.82s. With its judicious use of averaged Jacobians, `bvp4c` took only 13.31s. This is a substantial reduction in run time that requires no action on the part of the user. The run time can be reduced considerably by providing the solver with analytical partial derivatives. Averaged Jacobians are not used in `bvp4c` when analytical partial derivatives are supplied, so the run times of the two codes are then much more comparable, 9.56s for `R12bvp4c` and 8.83s for `bvp4c`. `bvp4c` takes advantage of a vectorized ODE function. Using this option, it solved the BVP in 6.49s with finite difference partial derivatives and in 3.50s with analytical partial

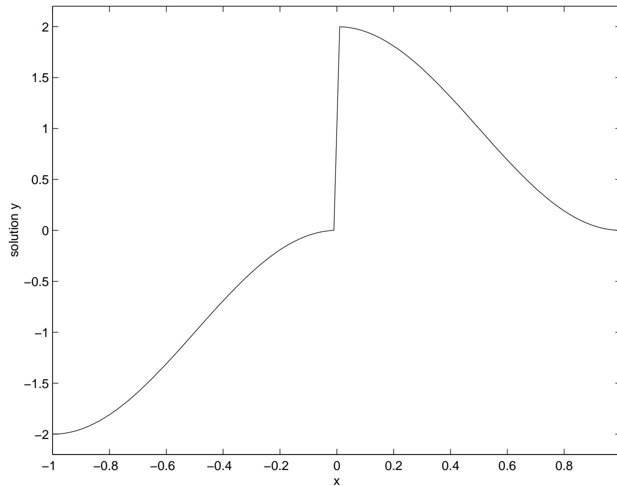

FIG. 5.3. *The shock layer problem with* $\epsilon = 10^{-5}$.

derivatives. If run time is sufficiently important to warrant the trouble of providing analytical partial derivatives and/or vectorizing the ODE function, the algorithmic developments of `bvp4c` make it possible to reduce the run time significantly. Of course, if run time is *very* important, the computations should not be performed in MATLAB at all.

Of the solvers written for GSC that we have cited, the only one that might be compared fairly with `bvp4c` is MIRKDC because it is the only one that controls the size of the residual. It is a natural choice anyway because the algorithms of `bvp4c` are developments of those of the method of order 4 in MIRKDC, developments tailored to MATLAB. MIRKDC can be used to solve the present BVP because it can do continuation and it has a subroutine for interpolation. It requires analytical partial derivatives, but this is an option in `bvp4c`. We created MEX files for MIRKDC and its interpolation subroutine that we used to solve the BVPs very much as we did with `bvp4c` and `bvpval`. A critical difficulty in making a fair comparison is that the two codes do not measure the size of the residual in the same way: MIRKDC measures the size of the residual relative to $\max(|f|, 1)$ and `bvp4c`, relative to $|f| + AbsTol/RelTol$. The error controls are *roughly* comparable if we take $AbsTol = RelTol$. For our experiments we took the tolerance *tol* of MIRKDC to be 10^{-3} and $RelTol = AbsTol =$ *tol* for `bvp4c`. It proved to be illuminating to solve the BVP for several values of ϵ. The run times are reported in Table 5.1. These times are not intermediate results of continuation to $\epsilon = 10^{-5}$; they are the results of independent runs with the ϵ of the table being the final value of continuation. In comparing the results we must consider the effects of compilation and algorithms. The former is illuminated by another set of computations displayed in the table. The Cbvp4c line in the table presents results obtained using compiled versions of `bvp4c` and `bvpval`. It might be surprising that compilation did not reduce the run time more. Certainly it would be possible to obtain a more efficient C version of the solver by coding it directly, but the fundamental issue is that only part of the computation is compiled. The solver must continually pass through an interface to MATLAB where the various functions are evaluated as interpreted code. The FORTRAN code of MIRKDC is coded more efficiently, but the expense of going through the interface is higher because the FORTRAN code is

TABLE 5.1
Run time in seconds.

Final ϵ	10^{-3}	10^{-4}	10^{-5}
bvp4c	1.24	2.88	8.41
Cbvp4c	1.02	2.38	6.92
MIRKDC	0.97	3.58	12.24
Vbvp4c	0.63	1.34	3.37

TABLE 5.2
Number of mesh points.

ϵ	10^{-2}	10^{-3}	10^{-4}	10^{-5}
bvp4c	36	56	113	235
MIRKDC	36	87	208	699

not nearly as well matched to the PSE. The differences in the run times for `bvp4c` and MIRKDC are so large for the smaller ϵ that they must be due to algorithmic differences in the codes. This is made clear by Table 5.2 which displays the number of mesh points used by the two codes. To complete our comparison of run times, we show in the Vbvp4c line of Table 5.1 results for `bvp4c` when the ODE function is vectorized. These runs with an M-file show that exploiting fully the PSE is more important than (partial) compilation.

6. Acknowledgements. We are grateful to P.H. Muir of Saint Mary's University for valuable discussions about residual control and MIRKDC and to I. Gladwell of Southern Methodist University for valuable advice about the design and implementation of `bvp4c`. Some of L.F. Shampine's research was done while he held the Ontario Research Chair in Computer Algebra at the University of Western Ontario.

REFERENCES

[1] U. M. ASCHER, R. M. M. MATTHEIJ, AND R. D. RUSSELL, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, Society for Industrial and Applied Mathematics, Philadelphia, 1995.

[2] J. R. CASH, *A variable order deferred correction algorithm for the numerical solution of nonlinear two point boundary value problems*, Comput. Math. Appl., 9 (1983), pp. 275–265.

[3] ———, *On the numerical integration of nonlinear two-point boundary value problems using iterated deferred corrections, Part 2: The development and analysis of highly stable deferred correction formulae*, SIAM J. Numer. Anal., 25 (1988), pp. 862–882.

[4] J. R. CASH AND D. R. MOORE, *A high order method for the numerical solution of two-point boundary value problems*, BIT, 20 (1980), pp. 44–52.

[5] J. R. CASH AND M. H. WRIGHT, *User's Guide for TWPBVP: A Code for Solving Two-Point Boundary Value Problems*, in Ode directory of Netlib.

[6] ———, *A deferred correction method for nonlinear two-point boundary value problems: Implementation and numerical evaluation*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 971–989.

[7] P. DEUFLHARD, *Nonlinear equation solvers in boundary value problem codes*, in Codes for Boundary-Value Problem in Ordinary Differential Equations, B. Childs, M. Scott, J. W. Daniel, E. Denman, and P. Nelson, eds., vol. 76 of Lecture Notes in Computer Science, Springer, 1979, pp. 40–66.

[8] E. DICKMANNS AND K. WELL, *Approximate solution of optimal control problems using third order Hermite polynomial functions*, in Optimization Techniques, G. I. Marchuk, ed., vol. 27 of Lecture Notes in Computer Science, Springer, 1975, pp. 158–166.

[9] W. H. ENRIGHT AND P. H. MUIR, *A Runge-Kutta type boundary value ODE solver with defect control*, Tech. Rep. 267/93, University of Toronto, Dept. of Computer Science, Toronto, Canada, 1993.

[10] ———, *Runge-Kutta software with defect control for boundary value ODEs*, SIAM J. Sci. Comput., 17 (1996), pp. 479–497.

[11] I. Gladwell, *Shooting methods for boundary value problems*, in Modern Numerical Methods for Ordinary Differential Equations, G. Hall and J. M. Watt, eds., Clarendon Press, Oxford, 1976, pp. 216–238.

[12] S. Gupta, *An adaptive boundary value Runge-Kutta solver for first order boundary value problems*, SIAM J. Numer. Anal., 22 (1985), pp. 114–126.

[13] M. Hanke, R. Lamour, and R. Winkler, *Program system "RWA" for the solution of two-point boundary value problems - foundations, algorithms, comparisons*, Seminarbericht nr. 67, Sektion Mathematik der Humboldt-Universität zu Berlin, Berlin, 1985.

[14] J. Kierzenka, *Studies in the Numerical Solution of Ordinary Differential Equations*, PhD thesis, Southern Methodist University, Dallas, TX, 1998.

[15] M. Lentini and V. Pereyra, *A variable order finite difference method for nonlinear multipoint boundary value problems*, Math. Comput., 28 (1974), pp. 981–1003.

[16] L. F. Shampine and M. W. Reichelt, *The Matlab ODE Suite*, SIAM J. Sci. Comput., 18 (1997), pp. 1–22.

[17] B. Wendroff, *Theoretical Numerical Analysis*, Academic Press, New York, 1966.