A User-Friendly Fortran BVP Solver

L.F. Shampine
Mathematics Department
Southern Methodist University
Dallas, TX 75275, U.S.A.
lshampin@smu.edu,

P.H. Muir*and H. Xu
Department of Mathematics and Computing Science
Saint Mary's University
Halifax, Nova Scotia B3H 3C3, Canada
muir@smu.ca, h_xu@cs.stmarys.ca

June 7, 2006

Abstract

MIRKDC is a FORTRAN 77 code widely used to solve boundary value problems (BVPs) for ordinary differential equations (ODEs). A significant issue with this package and similar packages is that the user interfaces are so complicated that potential users may be reluctant to invest the time needed to learn how to use them properly. We have applied our experience in writing user interfaces for ODE solvers in MATLAB and Fortran 90/95 to develop a user-friendly Fortran 90/95 BVP solver from MIRKDC. In the course of developing a completely new user interface, we added significantly to the algorithmic capabilities of MIRKDC. In particular, the new solver, BVP_SOLVER, extends the class of BVPs solved by MIRKDC to problems with unknown parameters and problems with ODEs having a singular coefficient. It uses more effective Runge-Kutta formulas and continuous extensions. We have also written a number of auxiliary routines to provide further convenience to users of this package. For instance, there are routines for evaluating the solution and its derivative, routines for saving and retrieving solution information, and a routine that facilitates continuation in the length of the interval.

1 Introduction

This paper describes a large project resulting in new software for the numerical solution of boundary value problems (BVPs) for first order systems of ordi-

*Corresponding author

nary differential equations (ODEs). MIRKDC [12] is a FORTRAN 77 code widely used to solve BVPs. This code and other popular BVP solvers such as PASVA3/BVPFD [5, 17], COLSYS/COLNEW [2, 4], and TWPBVP [7] can appear formidable to new users because the user interfaces, i.e., argument lists and subroutines that must be written by the user, are quite long and complicated. For this reason it is all too frequent that users are reluctant to try the packages or they try them, get into trouble, and give up. Exploiting the capabilities of the MATLAB programming environment [18], Kierzenka and Shampine [16, 22] found that it was possible to simplify greatly the user interface of BVP solvers. The algorithms of the resulting code, `bvp4c`, have much in common with MIRKDC. However, to approach the convenience of its user interface in Fortran, it is necessary to have the capabilities of Fortran 90/95. For the sake of brevity we shall write F77 for FORTRAN 77 and F90 for Fortran 90/95 from now on. We learned how to do something along these lines in developing the user–friendly Fortran delay differential equation solver DDE_SOLVER [23].

Here we describe a user–friendly BVP solver called BVP_SOLVER. This package was developed through a massive modification of MIRKDC. By exploiting capabilities of F90 we were able to reduce radically the number of arguments and the number of subroutines that the user must write to describe the problem. In addition, we were able to reduce substantially the complexity of the source code by replacing static work arrays with arrays allocated dynamically, removing all common blocks, and replacing all low–level linear algebra subroutines with calls to intrinsic array functions. This is important because it improves greatly the maintainability of the code and facilitates further development. The emphasis on user-friendliness is further supported through auxiliary routines that perform common tasks such as evaluating the solution and its derivative, saving and retrieving solution information, and continuation in the length of the interval. The solver and auxiliary functions are available for download at `http://cs.stmarys.ca/~muir/`. Also available are a number of example programs that can be used as templates.

We consider the solution of BVPs for a first order system of NODE ordinary differential equations of the form

$$y' = \frac{1}{x-a}\,Sy + f(x,y,p), \qquad a \le x \le b, \tag{1}$$

subject to general nonlinear separated boundary conditions (BCs)

$$0 = g_a(y(a),p), \qquad 0 = g_b(y(b),p). \tag{2}$$

The constant matrix S is optional. This extension of the form typical of BVP solvers makes it easy to solve problems with solutions that are well-behaved at $x = a$ despite the singular coefficient. Such problems arise most often when partial differential equations are reduced to ODEs by cylindrical or spherical symmetry. The vector p of unknown parameters is also optional. This is a much more important extension of the typical form. Unknown parameters arise naturally, as in eigenvalue problems, but most often when asymptotic expansions

are used to deal with singular behavior, either due to singular coefficients or infinite intervals.

In the course of developing a completely new user interface for MIRKDC, we added significantly to its capabilities. We have already mentioned unknown parameters and a class of singular BVPs. A number of extensions to the capabilities of MIRKDC were developed in the Master's thesis of H. Xu [24]. One that we include in BVP_SOLVER is a set of improved Runge–Kutta formulas and associated continuous extensions. Although the formulas originally developed for the MIRKDC package perform well, it was shown in [19] that optimized formulas can lead to significant improvements in the overall performance of the code. MIRKDC requires users to provide subroutines for analytical partial derivatives. An algorithmic development described in [24] and further developed for BVP_SOLVER is a subroutine to approximate these partial derivatives by finite differences. This is already a convenience for users, but a more important convenience is to use it by default because then a user need not even consider this technical matter when solving an easy problem. Of course the new interface allows users to provide subroutines for analytical partial derivatives when this is worth the trouble. Provision of working storage for the F77 code is at best inconvenient. A more serious issue is that the storage needed to solve a given problem is not known in advance and the computation will fail if the user has not provided enough. Exploiting the dynamic storage capabilities of F90 in BVP_SOLVER, we have relieved the user entirely of such concerns. BVP_SOLVER computes an approximate solution that is $C^1[a, b]$. It uses a backward error analysis approach to error control and so controls the amount by which the approximation fails to satisfy the ODEs and the boundary conditions, the *defect* or *residual* of the approximation. However, the new solver has an option for estimating the error in the solution itself at mesh points.

The development of BVP_SOLVER has benefitted greatly from our experience with the MATLAB solver `bvp4c`. Although solving a given problem with BVP_SOLVER is not as easy as solving it with `bvp4c`, it is not much harder and the difference is largely due to writing programs in F90 instead of MATLAB. Because it is written in a compiled language and implements methods of higher order, BVP_SOLVER can be *much* faster than `bvp4c`, so it is to be preferred whenever run time is important.

2 Simple Calls for Simple Problems

One of our goals was to make solving simple problems as easy as possible. As it has turned out, just about all you have to do is define the problem. Before explaining how to do that, we need to explain that we have encapsulated all information about a numerical solution in a derived type, namely TYPE(BVP_SOL). A solution can be called anything, but in this paper we generally use the name SOL. Although quantities of interest can be accessed directly by means of fields in the structure, we have generally preferred to work with the solution by means of auxiliary functions.

We have split solving a BVP into two phases. An initial guess (structure) for the solution is created by a call that is typically of the form

```
SOL = BVP_INIT(NODE,LEFTBC,X,Y_GUESS)
```

Recall that NODE is the number of ODEs. LEFTBC is the number of boundary conditions at the left end of the interval, which is to say, it is the length of the vector $g_a(y(a), p)$ in (2). The vector X is a guess for a mesh that reveals the behavior of the solution; i.e., the mesh should have more points where the solution changes most rapidly. The NPTS entries in X must be strictly increasing with the first entry equal to a and the last equal to b. The argument Y_GUESS provides a guess for the solution on the mesh X. It can have three forms. If it is a vector of length NODE, the function BVP_INIT interprets the guess to be constant, i.e., the guess is Y_GUESS at all NPTS mesh points. If it is an array of size NODE by NPTS, a column of the array is a guess for the solution at the corresponding entry of X. If it is the name of a subroutine of the form FCN(X,Y), the subroutine is called for each mesh point to obtain the approximate solution there. BVP_INIT is a generic subroutine that calls one of three subroutines selected at run time by matching the types of the input arguments. This nice facility of F90 allowed us to provide the user with the most important possibilities for a guess and still have a simple and uniform interface.

The first four arguments of BVP_INIT are required, but there are two optional arguments. If there are unknown parameters, an initial guess for the parameters must be supplied as the vector value of an argument with keyword P and the call will have the form

```
SOL = BVP_INIT(NODE,LEFTBC,X,Y_GUESS,P)
```

Indeed, the solver uses the presence of this argument to recognize that there are unknown parameters and in particular that p is present in the differential equations (1) and the boundary conditions (2). The other optional argument is described below.

Having supplied minimal information about the problem and required guesses for mesh, solution, and if present, unknown parameters, the problem is solved with a call of the form

```
SOL = BVP_SOLVER(SOL,FSUB,BCSUB)
```

The input argument SOL is a guess structure typically formed with a call to BVP_INIT. FSUB is the name of a subroutine for evaluating $f(x, y, p)$ in (1) and BCSUB is the name of a subroutine for evaluating the boundary conditions functions in (2). Certainly we can say no less about the BVP. This is made possible by heavy use of defaults.

On a successful return from BVP_SOLVER, the solution and optionally its first derivative can be evaluated with an auxiliary subroutine called BVP_EVAL. A typical call to BVP_EVAL has the form

```
CALL BVP_EVAL(SOL,XPLOT,YPLOT)
```

For a solution encapsulated in a structure named SOL, this call with a scalar XPLOT returns the NODE components of the approximate solution at XPLOT in the vector YPLOT. If XPLOT is a vector of NPLOT components, the subroutine returns a matrix YPLOT that is NODE by NPLOT with each column of YPLOT being the approximate solution at the corresponding entry of XPLOT. A call of the form

```
CALL BVP_EVAL(SOL,XPLOT,YPLOT,DYPLOT)
```

similarly returns the derivative of the solution in the array DYPLOT. The solver approximates the solution components by polynomials (based on continuous Runge-Kutta methods) on subintervals of $[a, b]$. The degree of these polynomials corresponds to the order of the method. Solution approximations are computed in BVP_EVAL by evaluating these polynomials and optionally their first derivatives. It is sometimes useful to approximate the solution outside of $[a, b]$. For this reason arguments outside the interval are permitted, but it is to be appreciated that the approximations are obtained by polynomial extrapolation. It is well–known that extrapolation can be unsatisfactory and that this becomes more likely as the order increases. If there are unknown parameters, the computed approximations are available from a call of the form

```
CALL BVP_EVAL(SOL,P)
```

or directly as the field SOL%PARAMETERS in the solution structure.

Once the computed solution is no longer needed, it is important to release the memory that has been allocated to the array fields of SOL. This is done by calling BVP_TERMINATE which deallocates these fields.

There are a number of optional arguments for BVP_SOLVER. Several are discussed in other sections, specifically singular terms in §8, analytical partial derivatives in §4 and §9, and monitoring the computation in §5. As mentioned earlier, BVP_SOLVER employs a set of improved Runge–Kutta formulas [19, 24]. The default method is of order 4, but there are also methods of order 2 and 6. The solver is instructed to use one of the other methods with the optional argument METHOD. For example, if we wanted to use the method of order 6, the basic call above that includes only required arguments would be extended to

```
SOL = BVP_SOLVER(SOL,FSUB,BCSUB,METHOD=6)
```

Optional arguments must follow the required arguments, but by using keywords to identify them, they can be specified in any order. An important option is the scalar tolerance τ that can be specified by means of the optional argument TOL. BVP_SOLVER controls the size of the residual (defect) of an approximate solution $u(x)$ in a mixed relative-absolute sense. Specifically, it attempts to compute $u(x)$ so that

$$\frac{|u'(x) - f(x, u(x), p)|}{1 + |f(x, u(x), p)|} \leq \tau$$

holds for all components and all $a \leq x \leq b$, and so that the boundary conditions are satisfied to within τ. The default tolerance is 10^{-6}. To change this to, say, 10^{-3}, the preceding call would be changed to either

```
SOL = BVP_SOLVER(SOL,FSUB,BCSUB,TOL=1D-3,METHOD=6)
```

or

```
SOL = BVP_SOLVER(SOL,FSUB,BCSUB,METHOD=6,TOL=1D-3)
```

As explained above, the solver controls the defect in the ODEs and the boundary conditions. Providing the optional argument YERROR causes the solver to estimate the error in the solution itself at the mesh points. It uses global extrapolation to estimate the maximum error in a solution component $y_i(x_m)$ relative to $1 + |y_i(x_m))|$ for all i and all mesh points x_m and returns this quantity as YERROR. The same estimate applies to unknown parameters. Examples are found at the end of §8.

The solver adapts and refines the initial mesh until it manages to compute an approximation that satisfies the accuracy requirement or it gives up because it believes more than the allowed number of subintervals will be necessary. The maximum number of subintervals is 3000, but this can be changed using the optional argument MAX_NUM_SUBINTERVALS in the call to BVP_INIT. We illustrate this in §10. The solver terminates the run with an explanatory message when it fails, but this behavior can be changed with an appropriate value for the optional argument STOP_ON_FAIL as explained in §5 and illustrated in §10.

In simplest use BVP_INIT has 4 arguments and BVP_SOLVER has 3. This is remarkably fewer arguments than seen in popular BVP solvers. For instance, MIRKDC has 20 arguments, COLSYS/COLNEW has 17, and TWPBVP has 26. This is partly due to the language because in F77 it is necessary for users to deal with storage and there is no provision for optional arguments. It is also due to differences in design. For instance, COLSYS/COLNEW allows for systems of mixed order, for points that are to be kept fixed when adapting meshes to the solution, and allows the user to tell the code if the problem is linear. TWPBVP also provides the last two of these options. Both COLSYS/COLNEW and TWPBVP have an error control that involves a vector of weights. BVP_SOLVER does not provide these options so COLSYS/COLNEW and TWPBVP have additional arguments that might well be considered worthwhile. All of the other solvers require users to provide subroutines for evaluating partial derivatives; the default in BVP_SOLVER avoids this by approximating partial derivatives with finite differences. None of the other codes provides for unknown parameters. In simplest use, our design requires only one additional argument in BVP_INIT when there are unknown parameters.

3 Unknown Parameters

Often it is necessary to determine some unknown parameters as part of solving a BVP. An obvious example is computing an eigenvalue along with an eigenfunction. When an asymptotic expansion is used to represent a solution at

an end point where a coefficient is singular or the end point is at infinity, the expansion typically involves unknown parameters that must be determined as part of solving the BVP. A nonlinear eigenvalue problem of lubrication theory studied in §6.1 of H.B. Keller's text [15] is a concrete example. The equation has the form

$$\epsilon y'(x) = \sin^2(x) - \lambda \frac{\sin^4(x)}{y(x)}. \tag{3}$$

There is a known parameter $\epsilon > 0$ in this first order equation. Because of the unknown parameter λ, there are two boundary conditions,

$$y(-\pi/2) = 1, \qquad y(\pi/2) = 1. \tag{4}$$

Our example program LUBRICATION solves this problem for $\epsilon = 0.1$. Because BVP_SOLVER provides for unknown parameters, the program is completely straightforward. Some details are found in §9. Another example in that section shows how an asymptotic expansion can be used to compute a solution with singular behavior. It is a difficult problem that requires some analysis to determine the behavior of the solution and continuation in the length of the interval as explained in §7.

The Fortran BVP solvers in wide use do not provide for unknown parameters and in particular, MIRKDC does not. The main reason for this is the way that the codes solve linear systems. With separated boundary conditions (2) the linear systems that arise with standard methods can be written so that the matrices are of a form called "almost block diagonal" (ABD) [1]. Special methods have been developed to solve such systems using Gaussian elimination with alternate row and column pivoting to obtain a stable decomposition of the matrix without introducing fill-in. A quality implementation of such an algorithm is the COLROW program [11]. MIRKDC uses this program and so does BVP_SOLVER. The trouble with this approach and the reason most solvers do not provide for unknown parameters is that they lead to matrices that are not ABD. That is not an issue with `bvp4c` because it solves its linear systems with a program for general sparse matrices already available in MATLAB. At some cost in storage, sparse matrix technology made it easy to provide the convenience of unknown parameters in `bvp4c`.

It is straightforward to introduce new variables so as transform a problem with unknown parameters into a bigger problem without unknown parameters. There is an additional ODE for each unknown parameter, but generally there are few unknown parameters, so this increase in the size of the system is harmless. This way of using standard BVP solvers to deal with unknown parameters is suggested in standard texts like [3], but it is at best an annoyance for users. To provide the convenience of unknown parameters and still take advantage of software for ABD systems, we transform the problem inside BVP_SOLVER and so extend the class of problems solved by MIRKDC to (1),(2). If there are unknown parameters, p, a user must naturally provide a guess for them. As mentioned earlier, this is done with an optional argument of BVP_INIT. From the user's point of view, that is really all that is different about solving a

problem with unknown parameters with BVP_SOLVER, at least if the default finite difference approximations are used for partial derivatives. Naturally the subroutine FSUB for evaluating the ODEs must accept arguments X,Y,P and evaluate $f(x, y, p)$ and correspondingly for the boundary conditions subroutine. A programming issue is how to arrange that the solver call subroutines with the correct sets of arguments. In F90 there is an intrinsic for testing whether an optional argument is present. The solver defines a global variable NPAR in BVP_M that is the length of the vector p if a guess was provided to BVP_INIT and otherwise, zero. The solver evaluates the ODEs by calling a subroutine P_FSUB. This subroutine calls FSUB with arguments X,Y,P if NPAR is positive and otherwise with arguments X,Y. The boundary conditions subroutine is handled in the same way.

4 Finite Difference Jacobians

BVP codes are more likely to converge and converge faster if provided the analytical partial derivatives

$$\frac{\partial f}{\partial y}, \quad \frac{\partial g_a}{\partial y(a)}, \quad \frac{\partial g_b}{\partial y(b)},$$

and if there are unknown parameters,

$$\frac{\partial f}{\partial p}, \quad \frac{\partial g_a}{\partial p}, \quad \frac{\partial g_b}{\partial p}.$$

Furthermore, it may be (much) cheaper to evaluate the derivatives analytically than to approximate them by finite differences. This is why MIRKDC requires users to supply subroutines for analytical partial derivatives. However, it is our experience that the most common mistake users make is the incorrect preparation of these subroutines. To make it as easy as possible to solve easy problems, the default in BVP_SOLVER is therefore to approximate these partial derivatives by finite differences. Naturally the solver allows users to provide subroutines when this is convenient or desirable. For example, if it is convenient to provide a subroutine for evaluating analytically the partial derivatives of $f(x, y, p)$, it can be passed to the solver with the keyword DFDY. Note that this keyword is used whether or not p is present. Similarly the keyword DBCDY is used to communicate the name of a subroutine for evaluating partial derivatives of the boundary conditions functions. An example is found in §9.

We assume that the number of ODEs is not so large that it is important to take the structure of $f(x, y)$ into account. Correspondingly our function for approximating $\partial f/\partial y$ is an F90 implementation of a standard algorithm for dense Jacobians much like the FDJAC1 routine of MINPACK [13]. Dealing with boundary conditions is more complicated. Before explaining how partial derivatives are approximated, we need to discuss how the user defines the boundary conditions. MIRKDC asks the user to evaluate both $g_a(y(a))$ and $g_b(y(b))$ in a

single subroutine of the form GSUB(NODE,YA,YB,GYAYB). The user provides the solver with an integer LEFTBC that specifies the number of boundary conditions imposed at $x = a$, i.e., the length of the vector $g_a(y(a))$. In this design the subroutine returns a vector GYAYB of NODE components. The first LEFTBC components are g_a(YA) and the remaining components are g_b(YB). Because g_a(YA) and g_b(YB) are computed and used separately, it seemed more natural to us to return the vectors directly. Accordingly, BVP_SOLVER expects a subroutine of the form BCSUB(YA,YB,BCA,BCB). The output argument BCA is the vector g_a(YA) and BCB is the vector g_b(YB).

The subroutine of [24] for approximating $\partial g_a/\partial y(a)$ and $\partial g_b/\partial y(b)$ forms both these rectangular matrices in a matrix that is NODE × NODE. The subroutine approximates the two matrices one after the other at a cost of 2*NODE calls to GSUB. In the corresponding function of BVP_SOLVER we take advantage of the structure of the square matrix of partial derivatives to form the two rectangular submatrices simultaneously at a cost of only NODE calls. This is a straightforward application of the ingenious scheme of Curtis, Powell, and Reid [9]. It might be remarked that the numerical approximations are the same in both programs, it is just that the approach we employ in BVP_SOLVER is less expensive. When there are unknown parameters, partial derivatives with respect to p are approximated in the same way.

5 Tracing the Progress of the Computation

The progress of the computation can be monitored by means of an optional argument of BVP_SOLVER with keyword TRACE. The default value for this parameter, TRACE = 0, corresponds to no output when the code executes successfully. However, if the computation is unsuccessful, the code will write a message to the standard output channel and STOP. Other values of TRACE increase the amount of information returned by the code: When TRACE = 1, the code returns the number of subintervals of each mesh and the number of (full) Newton iterations for each mesh. When TRACE = 2, it also returns the mesh points and the norm of each Newton correction.

Sometimes it is necessary to change the way the solver behaves on a failure. This is the case when the solver is embedded in a package that cannot permit a subroutine to output a message directly or to STOP. An example is discussed in §10. If the optional argument STOP_ON_FAIL is set to .FALSE., the solver will not output a message and it will not stop on a failure. If this is done, the calling program must be able to determine whether the computation was successful. This is done by testing whether the field SOL%INFO is 0. This value indicates success and any non-zero value indicates failure.

6 Saving and Retrieving a Solution

BVP_SOLVER encapsulates a solution as a structure of derived type. Auxiliary functions make it easy to work with a solution in this form. However, sometimes we wish to save a solution for later use. This is not entirely straightforward in F90 because the BVP_TOL derived type uses pointers for arrays of a size that cannot be determined in advance. It is not difficult to deal with this, but it is enough trouble that we have written two auxiliary functions to facilitate the task. The subroutine BVP_SAVE(UNUM,FNAME,SOL) saves the solution structure SOL in a file. UNUM specifies the unit to be OPENed. The unit is CLOSEd after use. The string FNAME specifies the name of the file. The structure can be retrieved from this file with BVP_GET, which has a similar syntax. For example, we could save a solution structure in one program with

```
CALL BVP_SAVE(8,"SaveSOL",SOL)
```

and then retrieve it in another with

```
CALL BVP_GET(9,"SaveSOL",SOL)
```

The general form for a call to BVP_SAVE is

```
CALL BVP_SAVE(UNUM,FNAME,SOL)
```

where UNUM specifies the unit to be opened and the string FNAME specifies the name of the file where the information in the solution structure SOL will be saved. The argument list for BVP_GET is identical. In BVP_SAVE we first determine the sizes of the pointer arrays. Some of these dimensions are already available as fields in the structure SOL and the rest we get from the fields themselves with the SIZE intrinsic. We write the sizes to an unformatted file and then use the sizes to write all the fields of the structure to the file. In BVP_GET we reverse this process by reading the sizes and then reconstruct the solution by reading the data for the various fields.

7 Continuation

A principal difficulty in solving BVPs is finding guesses for the mesh and solution good enough that an approximate solution can be computed. Continuation is an important approach to securing a good guess. Often a BVP involves a known parameter and we are interested in solutions for a range of parameter values. Generally, but not always, the solution of a BVP will change by a small amount when a parameter is changed by a small amount. The solution for one value of the parameter will then provide a good initial guess for the solution of the BVP with a slightly different value of the parameter. The lubrication problem (3),(4) makes the point. Keller computes and plots solutions of this singularly perturbed ODE for a range of ϵ. The BVP is easy to solve with $\epsilon = 0.1$ and by a succession of modest changes in ϵ, we can compute solutions for quite small values of the parameter. Some BVP solvers automate continuation;

early examples are [10, 17] and a recent one is [8]. MIRKDC does not have this capability and neither does BVP_SOLVER. On the other hand, we have designed BVP_SOLVER to facilitate continuation. This is accomplished by using the same structure for guesses and solutions, meaning that the solution for one problem can be used as guess for another.

Continuation can take many forms. A form not included in the previous discussion is particularly helpful when the interval is infinite, namely continuation in the length of the interval. We provide a function, BVP_EXTEND, to facilitate this. It can be called as

```
SOLOUT = BVP_EXTEND(SOLIN,ANEW,YANEW,BNEW,YBNEW)
```

Here SOLIN is a solution structure previously computed for a system of NODE differential equations on an interval [A,B]. If ANEW is less than A, the mesh SOLIN%X is extended in SOLOUT%X to include ANEW. Specifically, the SOLOUT%X array has one more entry than the SOLIN%X array, ANEW is the first entry of this array, and the mesh points of SOLIN%X are the remaining entries. Similarly, the array of approximate solutions SOLIN%Y is extended in SOLOUT%Y to include the approximation YANEW at ANEW, i.e., the array SOLOUT%Y has one more column than SOLIN%Y. The first column has the value YANEW and the remaining columns are equal to those of SOLIN%Y. If ANEW is not less than A, the arguments ANEW, YANEW are ignored by the function. After processing ANEW, the right end of the interval is treated in the same way based on the values of BNEW and YBNEW.

Generally the best way to obtain a value at a new end point, say YBNEW, is to use an asymptotic approximation. If such an approximation is not available, a natural approach is to extrapolate the computed solution of SOLIN by

```
CALL BVP_EVAL(SOLIN,BNEW,YBNEW)
```

and then input this YBNEW to BVP_EXTEND. However, the dangers of high-order polynomial extrapolation to points far outside the interval are well-known, so it may be preferable to extrapolate then with polynomials of very low degree. This is an option built into BVP_EXTEND. When called in the form

```
SOLOUT = BVP_EXTEND(SOLIN,ANEW,BNEW,ORDER)
```

with ORDER = 1, values YANEW and YBNEW are formed by linear extrapolation. If ORDER = 0 or this optional argument is not present, the values are formed by constant extrapolation.

If there are unknown parameters, the default in BVP_EXTEND is to use the values stored in SOLIN%PARAMETERS. A different guess can be supplied to BVP_EXTEND with an optional argument P . An example in §9 shows how continuation in the length of the interval can be used to compute a solution with singular behavior. Continuation is discussed further in §10.

8 Singular BVPs

Solving BVPs with solutions that are not well–behaved is not a routine matter, as we illustrate in §9. On the other hand, there is a class of problems with singular coefficients that can be treated in a routine way. These problems are sufficiently common that it is worth providing for them. We have in mind a system of the form (1) with non-zero S that is to be solved on $[a, b]$ with $a < b$. It is assumed that the solution $y(x)$ satisfies

$$\lim_{x \to a} \left(S \frac{y(x)}{x - a} \right) = S y'(a). \tag{5}$$

Some of the most popular BVP solvers, e.g., COLSYS/COLNEW, implement methods that do not evaluate the differential equation at the ends of the interval, so they can be applied to equations with a coefficient that is singular at an end without giving any special attention to the matter. However, Cash and Silva [6] provide several examples with well–behaved solutions for which COLSYS returned numerical solutions "which had no resemblance to the true solution" for a range of tolerances. We assume that the BVP has the form (1) with well–behaved solution and impose certain restrictions on the matrix S because it has been shown [14] that standard methods converge for such problems. BVP_SOLVER and `bvp4c` implement methods that evaluate the differential equation at the ends of the interval, so the codes must account for the singular term. The paper [21] considers how to do this and illustrates the points by modifying `bvp4c`, modifications later made to the version of MATLAB itself. This is mainly a matter of evaluating properly the differential equation. Passing to the limit in (1), we see that $y'(a)$ must be a solution of

$$(I - S) y'(a) = f(a, y(a), p).$$

de Hoog and Weiss [14] solve this equation for $y'(a)$ in an implementation of the trapezoidal rule. However, it is pointed out in [21] that the system can be singular, so Shampine takes a different approach in `bvp4c`, computing the pseudoinverse $(I - S)^+$ and then

$$y'(a) = (I - S)^+ f(a, y(a), p). \tag{6}$$

This approach is not expensive because the number of ODEs is generally quite small. The pseudoinverse is computed only once and saved for computing the limit values with a matrix multiplication.

To solve a singular BVP, the user supplies the matrix S as the value of an optional argument SINGULARTERM to BVP_INIT. From the user's point of view, this is the *only* thing different about solving a singular BVP of this kind. The function $f(x, y, p)$ is evaluated by a subroutine FSUB as usual. On entry to the solver the pseudoinverse $(I - S)^+$ is formed and made available throughout the module as a globally defined matrix. A logical variable SINGULAR with global scope is used to indicate whether S has been supplied. The coding

introduced for unknown parameters turns out to be quite convenient here. Recall that the solver calls a subroutine P_FSUB that calls FSUB in a way that takes account of the presence of unknown parameters to get $f(x, y, p)$. When SINGULAR is .TRUE., that evaluation is followed by a test on the independent variable. If $x = a$, the value $y'(a)$ is computed using (6) and otherwise $y'(x)$ is computed by adding $Sy/(x - a)$ to $f(x, y, p)$. When the Jacobian is computed by finite differences, no further action is necessary, but when a subroutine is provided for analytical partial derivatives of $f(x, y, p)$, the partial derivative with respect to y must be modified by adding $S/(x - a)$ when $x > a$ and multiplied by $(I - S)^+$ when $x = a$.

For the limit (5) to exist, it is necessary that

$$Sy(a) = 0. \tag{7}$$

It may not be convenient for a user to supply a guess that satisfies this condition and it is not clear that successive approximations computed by the solver will satisfy it automatically. We follow [21] and impose the condition by multiplying approximations to $y(a)$ by $I - S^+S$. This requires computing another pseudoinverse, but again it is done only once and then this matrix is available to impose the necessary condition on the guess and all approximations both easily and inexpensively.

Emden's equation

$$y'' + \frac{2}{x}y' + y^n = 0 \tag{8}$$

models a spherical body of gas. Russell and Shampine [20] use this equation with $n = 5$ and boundary conditions

$$y'(0) = 0, \qquad y(1) = \sqrt{3/4} \tag{9}$$

as a test problem because it has the analytical solution $(1 + x^2/3)^{-1/2}$. It is also used as a test problem in [21] where it is written as a system of first order equations with $(y_1, y_2) = (y(x), y'(x))$, namely

$$\frac{d}{dx}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \frac{1}{x}\begin{pmatrix} 0 & 0 \\ 0 & -2 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} + \begin{pmatrix} y_2 \\ -y_1^5 \end{pmatrix}.$$

This system has the form (1) and numerical solution of the BVP as coded in the example program EMDEN is completely straightforward. The program uses a constant initial guess for the solution of $(\sqrt{3/4}, 10^{-4})$ and a default initial mesh of 10 equally spaced points. This guess does *not* satisfy the necessary condition (7). With default tolerance the problem was solved easily and the computed solution satisfied the necessary condition perfectly.

Because the BVP has an analytical solution, we used it to illustrate the option of estimating the error in the solution itself at the mesh points. Table 1 shows the maximum relative error in the two solution components and all mesh points. In our experiments with other BVPs the estimated maximum was often quite good, but it was unusually accurate for this problem. It is perhaps

worth remarking that when there are unknown parameters, the same estimate of the maximum error applies. This means that if the maximum error occurs in a solution component, the errors in the unknown parameters will be smaller than the estimate. For instance, we computed a reference value for the unknown parameter λ of the problem that is discussed in §3 and solved with the LUBRICATION example program in §9. With this reference value and default tolerance, we obtained the results of Table 2.

METHOD	2	4	6
Estimate	1.05D-6	9.58D-8	3.89D-9
True	1.05D-6	9.58D-8	3.89D-9

Table 1: Maximum relative error at mesh points for problem (8), (9).

METHOD	2	4	6
Estimate	1.46D-7	3.70D-8	6.23D-9
True	6.30D-8	3.76D-8	3.56D-9

Table 2: Maximum relative error in λ for problem (3), (4) with $\epsilon = 0.1$.

9 Numerical Examples

Solving a BVP with BVP_SOLVER requires two files: The module BVP_M.f90 contains the solver and its supporting programs written in F90. The file BVP_LA.f contains legacy software for linear algebra computations written in F77. The two files and programs that solve the problems of this paper and other illustrative problems are available from `http://cs.stmarys.ca/~muir/`.

With one exception, all our examples are written in a single file that begins with a module containing the subroutines for the differential equations and the boundary conditions and followed by a program for the computation and output of the solution. It is convenient to have the module defining the BVP in the same file as the main program, but, of course, it could be placed in a separate file if it is lengthy. That is the case for the example discussed in §10. We extract some critical lines from the LUBRICATION example discussed briefly in §3 to show the typical form.

```
MODULE DEFINE_FCN
  . . .
  INTEGER, PARAMETER :: NODE=1,NPAR=1,&
                        LEFTBC=1,RIGHTBC=NODE+NPAR-LEFTBC
CONTAINS
  . . .
```

```
END MODULE DEFINE_FCN
PROGRAM LUBRICATION
  USE DEFINE_FCN
  USE BVP_M
  . . .
  X = BVP_LINSPACE(A,B,NSUB+1)
  Y_GUESS = (/ 0.5D0 /)
  P = (/ 1D0 /)
  SOL = BVP_INIT(NODE,LEFTBC,X,Y_GUESS,P)
  SOL = BVP_SOLVER(SOL,FSUB,BCSUB,DFDY=DFSUB,DBCDY=DGSUB)
  . . .
  DO I = 1,100
    XPLOT = A + (I-1)*( (B - A)/99D0 )
    CALL BVP_EVAL(SOL,XPLOT,YPLOT)
  . . .
  END DO
  CALL BVP_TERMINATE(SOL)
END PROGRAM LUBRICATION
```

It is convenient in the declarations part of the DEFINE_FCN module to define some parameters for global use, namely NODE, the number of ODEs; NPAR, the number of unknown parameters; LEFTBC, the number of boundary conditions at the left end point; and RIGHTBC, the number of boundary conditions at the right end point. For the sake of clarity we define RIGHTBC in terms of the other parameters. These parameters are used, e.g., to define the sizes of arrays in the subroutines appearing after the CONTAINS statement. The program LUBRICATION begins with the USE of the modules defining the BVP and the solver. Obviously the module BVP_M.f90 must be compiled and linked with the main program, but so must the file BVP_LA.f with the F77 linear algebra routines. An initial mesh is defined in terms of parameters defined in the program itself, namely the end points A and B and the number of subintervals NSUB in the initial mesh. The BVP_LINSPACE function used to form the initial mesh has the same basic functionality as the LINSPACE function of MATLAB. It is available in BVP_M as a convenience in dealing with this common task. A reasonable alternative for this problem is the default mesh of 10 equally spaced points that would be used if the call were

```
SOL = BVP_INIT(NODE,LEFTBC,(/A,B/),Y_GUESS,P)
```

In this example BVP_INIT is called with a constant guess for the solution and a guess is supplied for an unknown parameter. Note that both must be supplied as vectors even though they have only one component. By making use of defaults, only minimal information must be supplied by the user. Optional subroutines for the partial derivatives of the differential equations and boundary conditions are supplied to BVP_SOLVER in this example program to show how this is done using the keywords DFDY and DBCDY. This example uses BVP_EVAL to evaluate the approximate solution at 100 equally spaced points in the interval

and write these results to a file. The call to BVP_TERMINATE releases the memory allocated to the array fields of SOL.

Neng Wang, a professor of finance and economics at Columbia Business School, communicated to us an interesting BVP that he formulated when studying an optimal consumption problem with learning. He needed the solution $y(x)$ of a BVP defined on $[0, 1]$ by the ODE

$$\begin{aligned} ry(x) = \frac{\delta x}{r} \quad &+ \quad \frac{\delta^2}{2\sigma^2} x^2 (1-x)^2 y''(x) \\ &- \quad \frac{\gamma r \delta^2}{2\sigma^2} x^2 (1-x)^2 \left(y'(x)\right)^2 - \gamma \delta x (1-x) y'(x) \end{aligned} \tag{10}$$

and boundary conditions

$$y(0) = 0, \qquad y(1) = \frac{\delta}{r^2} \tag{11}$$

Typical values of the constants that were used in the example program ECON are $r = 0.04, \delta = 0.2, \sigma = 0.2, \gamma = 5$. Solving this BVP is not a routine matter because the ODE is singular at both ends of the interval. An asymptotic analysis shows that more than one kind of behavior is possible at the right end. When $\gamma\delta > r$, as it is for our example program, the quadratic equation

$$\frac{\delta^2}{2\sigma^2} c(c-1) + \gamma\delta c - r = 0$$

has a negative root and a root that satisfies $0 < c < 1$. For the positive root, the ODE has a solution that behaves like

$$y(x) \sim \frac{\delta}{r^2} + p(1-x)^c \tag{12}$$

as $x \to 1-$, where p is a undetermined parameter at this order in the asymptotic analysis. Notice that if $p \neq 0$, then $y'(x) \to +\infty$ as $x \to 1-$ because $c < 1$. At the left end of the interval,

$$y(x) \sim \frac{\delta}{r(r+\gamma\delta)}\, x \tag{13}$$

as $x \to 0+$. We see that the solution and its first derivative are well-behaved at this end.

We use a standard approach to the numerical solution of a singular BVP in the ECON example program. For $\Delta > 0$, we solve the ODE (written as a first order system) on $[\Delta, 1-\Delta]$ where it is non-singular. The boundary conditions for the numerical problem are that $y(x)$ agree with the analytical approximation (13) at Δ and (12) at $1-\Delta$. The asymptotic expansion (12) involves an unknown parameter p. It is introduced into the numerical approximation by requiring that $y'(x)$ agree with the derivative of (12) at $1-\Delta$. This is easy because BVP_SOLVER provides for unknown parameters. An important issue with any

difficult BVP is finding an initial guess good enough to get convergence. If we can accomplish that for some Δ, we can expect continuation to provide a sufficiently good guess for a Δ that is a little smaller. We must also provide an initial guess for the unknown parameter p. An easy way to get reasonable guesses for both is to choose p so that the asymptotic approximation (12) at the right end satisfies the boundary condition at the left end, namely $p = -\delta/r^2$, and use the asymptotic solution as a guess on all of $[\Delta, 1-\Delta]$.

For $\Delta = 0.1$, the BVP is not hard to solve with the guesses stated and an initial mesh of 20 equally spaced points. BVP_EXTEND is used to extend the solution on one interval to form the guess for the bigger interval. Values at the new end points are obtained from the asymptotic approximations. We found that the solution for one value of Δ provides a good enough guess to get convergence when Δ is halved. In this approach we obtain a solution on all of $[0, 1]$ by using the analytical approximations at the ends of the interval and evaluating the numerical solution on $[\Delta, 1-\Delta]$ with BVP_EVAL. To gain confidence in our results, we test the consistency of the intermediate solutions. Specifically, we compute approximations to $y(x)$ at 100 points equally spaced in $[0, 1]$ and then compute the maximum change in $y(x)$ relative to the biggest value of the solution, namely $y(1)$. With the solver's default tolerance, we accept the computed solution when the results agree to 10^{-4}. This is achieved when $\Delta = 7.8 \times 10^{-4}$, at which time the maximum relative difference in successive solutions is about 3.6×10^{-5}. Despite the singular behavior of the first derivative evident in Figure 1, there are only 41 points in the final mesh. That is possible because the singular behavior is represented by an asymptotic expansion and a numerical method is applied where the solution is relatively smooth.

This problem illustrates an approach for solving BVPs with singular behavior. First we sort out the behavior of the solution near singularities by analytical means. That may not be not easy, but once it is done, a program to compute the solution is relatively straightforward because the design of BVP_SOLVER makes it easy to incorporate this behavior, even when it involves unknown parameters, and continue in either a known parameter or the length of the interval.

10 Continuation Revisited and a Test Set

Some experimentation was necessary with the ECON example of the last section. If we decrease Δ too fast, e.g., dividing it by 5 instead of 2, the computation fails. In this section we discuss how to make adjustment of the continuation parameter more automatic and the whole process more efficient by exploiting some capabilities of the solver. This is illustrated with an example program, CTS.f90, that solves all the problems of a test set formulated by J. Cash. The CTS program could serve as a template for solving difficult BVPs by continuation in a parameter. It demonstrates that all problems in a substantial test set that includes solutions exhibiting a variety of boundary, interior, and transition layers can be solved readily with BVP_SOLVER.

The test set can be downloaded from

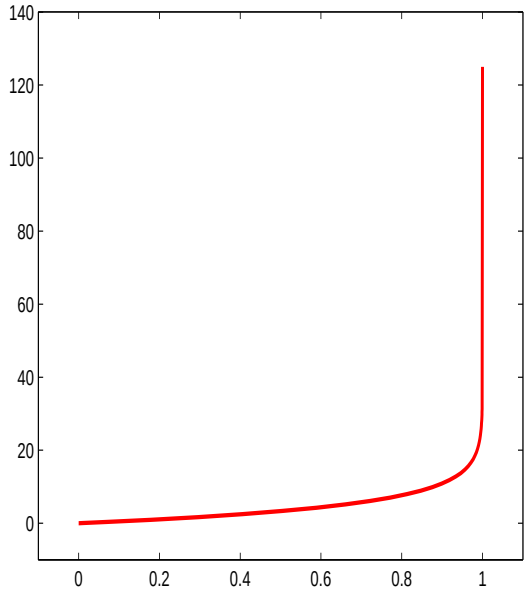

Figure 1: Solution $y(x)$ of (10), (11) with $r = 0.04, \delta = 0.2, \sigma = 0.2, \gamma = 5$.

`http://www.ma.ic.ac.uk/~jcash/BVP_software/readme.php`

There are 32 problems in the set. Problems 1–30 involve a scalar second order ODE written as a system of two first order equations and problems 31–32 involve systems of four first order equations. The ACDC code, which can be downloaded from the same site, has automatic continuation. It asks for starting and ending values of a known parameter EPS, so Cash specifies values for each problem called EPS0 and EPSF, respectively. Our CTS program solves these problems by continuation in EPS using BVP_SOLVER. Continuation was actually useful only for problems #24, 30, 32, so for the other problems we solve directly the problem with EPS equal to EPSF. We use default settings and in particular, use finite difference partial derivatives. The error tolerance is given a nominal value of 10^{-3}. In the last section we commented that it may be convenient to define the problem in a separate file if it is complicated. The thirty problems #1–30 are defined in the file CTS130.f90. Because of the difference in the sizes of the systems, it was convenient to define problems #31–32 in a separate file, CTS3132.f90. The CTS program can USE only one of these files at a time. At run time CTS asks which problem is to be solved. After the problem is solved, its solution is written to a data file. An M-file, CTS.m, is provided that imports this data into MATLAB and plots the solution.

The straightforward approach to continuation illustrated in §9 does not han-

dle a failed step in a satisfactory way. By default the solver will terminate the computation then. However, as explained in §5, we can prevent this by giving the optional argument STOP_ON_FAIL the value .FALSE. and testing the field SOL%INFO for success or failure. The solver will not quit until it has tried the maximum number of subintervals, which by default is 3000. The idea of continuation is to make a small change in the parameter so as to obtain a problem that is easy to solve. If the new problem requires many more subintervals, the change in the parameter was too large. In CTS we first solve the problem with the starting value of the parameter, EPS0. The number of mesh points required by BVP_SOLVER in order to solve the problem is available as SOL%NPTS. We use SOL%X and SOL%Y computed for one value of EPS as a guess for the mesh and solution when EPS is reduced by a FACTOR less than 1. The successive problems get harder, so we must allow more mesh points. If the solver is having trouble finding a suitable mesh, it may resort to halving its current mesh. In this situation we would prefer to give up and try again with less of a change in EPS. We can achieve this by using the option discussed in §2 to limit the number of subintervals to twice the number in the guess less 1 when we form the guess structure with BVP_INIT. CTS reduces EPS by a FACTOR that is initialized to 0.1. If a step fails, FACTOR is increased, the previous successful solution is restored, and the continuation step tried again with the larger value of EPS. However, if the step fails and the number of subintervals is equal to the maximum allowed of 3000, the program stops with a message to the effect that the continuation has failed. No provision has been made to reduce FACTOR. In part this is because we do not attempt to make continuation as efficient as possible and in part because we expect the problems to become more difficult for the solver as EPS approaches EPSF.

It is not easy to provide an automatic continuation facility because continuation is not an algorithm *per se*, rather an approach to solving difficult problems. We have not yet attempted to develop such a facility for BVP_SOLVER, but the CTS program outlined here shows how to use capabilities of the solver to implement continuation more effectively than in the ECON example. In particular, CTS shows how to recognize early when the parameter has been reduced too much and handle gracefully a computation that fails for this reason.

References

[1] P. Amodia, J.R. Cash, G. Roussos, R.W. Wright, G. Fairweather, I. Gladwell, G.L. Kraut, and M. Paprzycki, Almost block diagonal linear systems: sequential and parallel solution techniques, and applications, Numer. Linear Algebra Appl., 7 (2000) 275–317.

[2] U.M. Ascher, J. Christiansen, and R.D. Russell, Collocation software for boundary value ODEs, ACM Trans. Math. Softw., 7 (1981) 209–222.

[3] U.M. Ascher and R.D. Russell, Reformulation of boundary value problems into "standard" form, SIAM Rev., 23 (1981) 238–254.

[4] G. Bader and U.M. Ascher, A new basis implementation for a mixed order boundary value ode solver, SIAM J. Sci. Stat. Comp., 8 (1987) 483–500.

[5] BVPFD, subroutine in IMSL FORTRAN 77 Mathematics and Statistics Libraries v. 3, Visual Numerics Inc., Houston, TX, 2002.

[6] J.R. Cash and H.H.M. Silva, On the numerical solution of a class of singular two–point boundary value problems, J. Comp. Appl. Math., 45 (1993) 91–102.

[7] J.R. Cash and M.H. Wright, A deferred correction method for nonlinear two-point boundary value problems: implementation and numerical evaluation, SIAM J. Sci. Stat. Comput., 12 (1991) 971–989.

[8] J.R. Cash, G. Moore, and R.W. Wright, An automatic continuation strategy for the solution of singularly perturbed nonlinear boundary value problems, ACM Trans. Math. Softw., 27 (2001) 245–266.

[9] A.R. Curtis, M.J.D. Powell, and J.K. Reid, On the estimation of sparse Jacobian matrices, J. Inst. Math. Appl., 13 (1974) 117–119.

[10] P. Deuflhard, H.J. Pesch, and P. Rentrop, A modified continuation method for the numerical solution of nonlinear two-point boundary value problems by shooting techniques, Numer. Math., 26 (1976) 327–343.

[11] J.C. Diaz, G. Fairweather, and P. Keast, Algorithm 603. COLROW and ARCECO: FORTRAN packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination, ACM Trans. Math. Softw., 9 (1983) 376–380. COLROW, `http://www.mscs.dal.ca/~keast/`, 1992.

[12] W.H. Enright and P.H. Muir, Runge-Kutta software with defect control for boundary value ODEs, SIAM J. Sci. Comput., 17 (1996) 479–497.

[13] FDJAC1, subroutine in MINPACK, `http://www.netlib.org/minpack`.

[14] F.R. de Hoog and R. Weiss, Difference methods for boundary value problems with a singularity of the first kind, SIAM J. Numer. Anal., 13 (1976) 775–813.

[15] H.B. Keller, Numerical Methods for Two–Point Boundary–Value Problems, Dover, New York, 1992.

[16] J. Kierzenka and L.F. Shampine, A BVP solver based on residual control and the MATLAB PSE, ACM Trans. Math. Softw., 27 (2001) 299–316.

[17] M. Lentini and V. Pereyra, A variable order finite difference method nonlinear multipoint boundary value problems, Math. Comp., 28 (1974) 981–1004.

[18] MATLAB 6, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 2002.

[19] P.H. Muir, Optimal discrete and continuous mono-implicit Runge-Kutta schemes for boundary value ODEs, Adv. Comp. Math., 10 (1999) 135–167.

[20] R.D. Russell and L.F. Shampine, Numerical methods for singular boundary value problems, SIAM J. Numer. Anal., 12 (1975) 13–26.

[21] L.F. Shampine, Singular boundary value problems for ODEs, Appl. Math. and Comput., 138 (2003) 99–112.

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

[23] S. Thompson and L.F. Shampine, A Friendly Fortran DDE Solver, Appl. Numer. Math., 56 (2006) 503–516. `http://www.radford.edu/~thompson/ffddes/`.

[24] Hui Xu, Enhancements to a Runge-Kutta BVODE solver, M.Sc. Appl. Sci. Thesis, Dept. Math. and Comp. Sci., Saint Mary's University, Halifax, Nova Scotia, Canada, 2004.