Using AD to solve BVPs in MATLAB

L.F. SHAMPINE
Southern Methodist University
and
ROBERT KETZSCHER and SHAUN A. FORTH
Cranfield University (Shrivenham Campus)

The MATLAB program bvp4c solves two–point boundary value problems (BVPs) of considerable generality. The numerical method requires partial derivatives of several kinds. To make solving BVPs as easy as possible, the default in bvp4c is to approximate these derivatives with finite differences. The solver is more robust and efficient if analytical derivatives are supplied. In this paper we investigate how to use automatic differentiation (AD) to obtain the advantages of analytical derivatives without giving up the convenience of finite differences. In bvp4cAD we have approached this ideal by a careful use of the MAD AD tool and some modification of bvp4c.

Categories and Subject Descriptors: G.1.4 [**Quadrature and Numerical Differentiation**]: Automatic differentiation; G.1.7 [**Ordinary Differential Equations**]: Boundary value problems; G.4 [**Mathematical Software**]: Efficiency; G.4 [**Mathematical Software**]: Matlab

General Terms: Performance

Additional Key Words and Phrases: AD, BVP, Matlab

1. INTRODUCTION

The MATLAB [Matlab 2000] program bvp4c [Kierzenka and Shampine 2001] solves two–point boundary value problems (BVPs) of considerable generality. In particular, it provides for unknown parameters and general nonlinear boundary conditions that may be non-separated. The collocation method of this program results in a system of nonlinear algebraic equations that is solved by a variant of Newton's method. This involves a good many partial derivatives of several kinds. To make solving BVPs as easy as possible, bvp4c approximates these partial derivatives with finite differences. There is an option for providing functions to evaluate partial derivatives analytically and the documentation [Matlab 2000; Kierzenka and Shampine 2001; Shampine, L., Kierzenka, J. and Reichelt, M. ; Shampine et al.

©ACM, (2004). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Mathematical Software , VOL 31, No. 1 (March 2005), pp79-94 http://doi.acm.org/10.1145/1055531.1055535

Author's addresses: L.F. Shampine, Mathematics Department, Southern Methodist University, Dallas, TX 75275, U.S.A. Robert Ketzscher and Shaun A. Forth, Applied Mathematics & Operational Research, Engineering Systems, Cranfield University (Shrivenham Campus), Swindon SN6 8LA, U.K.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2005 ACM 0098-3500/2005/1200-0001 $5.00

2003] emphasizes that this can reduce the run time greatly. Also, despite the exceptionally strong function that `bvp4c` uses for approximating partial derivatives by finite differences, the program is somewhat more robust when given analytical partial derivatives.

With advances in automatic (algorithmic) differentiation (AD) in MATLAB like MAD [Forth 2001], it is natural to ask if it is possible to obtain the advantages of analytical partial derivatives in `bvp4c` with AD. Convenience was a primary goal in the design of `bvp4c`, so it is of the greatest importance that these advantages be obtained with minimal demands placed on the user. Here we report our experience developing a solver based on `bvp4c` that uses the MAD package for partial derivatives. This effort has been quite successful, but there is still scope for futher development.

2. PRELIMINARIES

In this section we state certain aspects of `bvp4c` and MAD that must be taken into account when using AD to improve the solution of BVPs for ordinary differential equations.

2.1 bvp4c

The `bvp4c` function solves BVPs consisting of a system of n first order ordinary differential equations (ODEs),

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}, \mathbf{p}) \tag{1}$$

and a set of boundary conditions,

$$0 = \mathbf{g}(\mathbf{y}(a), \mathbf{y}(b), \mathbf{p}) \tag{2}$$

Here $\mathbf{p}$ is a vector of unknown parameters that may, or may not, be present.

The collocation method of `bvp4c` leads to a system of nonlinear algebraic equations that is solved by a variant of Newton's method. Because the linearization is itself only an approximation, it is consistent to use finite difference approximations to the partial derivatives. `bvp4c` computes these approximations with `numjac`. Its algorithm [Shampine and Reichelt 1997] is exceptionally strong. For instance, it monitors the changes induced in the dependent variables by the increments in the independent variables and adapts the increments for the next computation. Moreover, if it does not believe that a column of partial derivatives has been computed reliably, it will adjust the increments and try again. Even with such precautions it is hard to produce accurate approximations reliably. When numerical approximations are not very accurate, analytical evaluation of the partial derivatives may result in faster convergence of the iteration. It may also result in a larger domain of convergence. In extreme cases, analytical partial derivatives may be the difference between success and failure.

Providing a routine for evaluating partial derivatives analytically is often inconvenient, but it can be considerably faster than having the solver approximate partial derivatives. In this connection it is important to appreciate that `bvp4c` is intended only for BVPs involving a relatively small number of equations. For example, 10 equations would be a moderately large system. Indeed, the `numjac` function provides for the efficient computation by finite differences of sparse partial derivative

matrices, but this capability is not used in `bvp4c`. Generally it is possible to speed up the computation of finite differences substantially in MATLAB by vectorizing the evaluation of $\mathbf{f}(x, \mathbf{y})$, but this is also inconvenient. In the next section we discuss vectorization more fully.

The convenience of not asking users for partial derivatives is obvious and considered to be of primary importance. It is worth stating that users seem to have trouble supplying partial derivatives, especially when there are unknown parameters $\mathbf{p}$. Here are the partial derivatives that are needed when solving (1,2) with `bvp4c`:

—$\mathbf{f_y}$, and when there are unknown parameters, $\mathbf{f_p}$

—$\mathbf{g}_{\mathbf{y}(a)}$ and $\mathbf{g}_{\mathbf{y}(b)}$, and when there are unknown parameters, $\mathbf{g_p}$

`bvp4c` allows the user to supply functions for evaluating the partial derivatives of $\mathbf{f}$, or $\mathbf{g}$, or both. It is much more important to provide the partial derivatives of $\mathbf{f}$.

2.1.1 *Vectorization.* Vectorization plays two roles in `bvp4c`. One is to speed up the finite difference approximation of $\mathbf{f_y}$. The obvious way to evaluate the function in (1) is to input $x, \mathbf{y}^k, \mathbf{p}$ and output a vector $\mathbf{f}^k = \mathbf{f}(x, \mathbf{y}^k, \mathbf{p})$. Often it is not much trouble in MATLAB to code this so that you can input several vectors $\mathbf{y}^k$ as columns in a matrix $\mathbf{Y}$, compute efficiently all the $\mathbf{f}^k$, and output the $\mathbf{f}^k$ as columns in a matrix $\mathbf{F}$. Depending on the ODEs and how carefully the function is coded, this can be much faster than computing the $\mathbf{f}^k$ one at a time. Indeed, it can cost little more than computing just one. This can be very advantageous because a program like `numjac` evaluates $\mathbf{f}$ about n times for each approximation to $\mathbf{f_y}$.

When approximating $\mathbf{f_y}$ by finite differences, x is held fixed and $\mathbf{f}$ is evaluated for a number of $\mathbf{y}$. A fundamental computation in `bvp4c` is to evaluate the residual of the numerical solution at all the mesh points x_i. In this $\mathbf{f}$ is evaluated for a set of arguments $(x_i, \mathbf{y}_i)$. Generally there are more mesh points than differential equations, often a great many more, so reducing the cost of this computation by vectorization can be much more important than reducing the cost of approximating Jacobians. Accordingly, in the BVP solver `bvp4c`, vectorization of $\mathbf{f}$ means vectorization with respect to both independent and dependent variables.

The two roles of vectorization in `bvp4c` are independent, so it is valuable to vectorize $\mathbf{f}$ even when partial derivatives are evaluated exactly, whether analytically or by AD.

3. AUTOMATIC DIFFERENTIATION AND MAD

In the standard reference for the subject [Griewank 2000], Griewank states that

> Algorithmic, or Automatic Differentiation (AD) is concerned with the accurate and efficient evaluation of derivatives for functions defined by computer programs.

AD is based on the systematic application of the chain rule of differentiation to the floating point evaluation of a function and its derivatives. Unlike finite difference approximation, there are no discretisation or cancellation errors and the resulting derivative values are accurate to within roundoff. By virtue of working in floating point arithmetic, AD is more efficient than symbolic differentiation as found, e.g., in

the packages Mathematica and Maple. It also permits the use of control structures such as loops, branches, and subfunctions that are common to modern computer languages but not easily amenable to symbolic differentiation.

There are two fundamental approaches to AD. *Forward Mode AD* involves augmenting a program for evaluating a function so that a variable's directional derivatives are calculated along with its value. Let $\texttt{time}(\mathbf{f})$ be the time it takes to evaluate a function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. A standard result [Griewank 2000] states that the time it takes to calculate the $m \times n$ Jacobian matrix $\mathbf{Jf}$ by forward mode AD is $O\,(n\,\texttt{time}(\mathbf{f}))$. *Reverse (Adjoint) Mode AD* is a two stage process. First the original function code is run, perhaps augmented by statements to store data to enable the code to be run a second time in *reverse*, propagating the sensitivities of the function's outputs to each calculated variable. Such sensitivities are termed *adjoints*. A computational complexity analysis for reverse mode AD states that $\texttt{time}(\mathbf{Jf}) = O\,(m\,\texttt{time}(\mathbf{f}))$. When $m \ll n$, the reverse mode is advantageous, but in the circumstances of `bvp4c`, the forward mode is both efficient and simpler. With this in mind we give our attention to the implementation of forward mode AD in MATLAB.

AD is implemented in one of two ways—operator overloading or source transformation. *Operator Overloading* takes advantage of the possibility of defining new types or classes in languages such as Fortran 95, C++, and MATLAB. The new AD type is defined to have a component which holds a variable's value and components to hold derivative information. Arithmetic and intrinsic functions are extended to the AD type by operator and function overloading. In typed languages such as Fortran and C++, all that remains is for the user to redefine the types of all relevant variables within the function and all subfunctions to that of the AD type, initialize appropriate values and derivatives, invoke the function, and then extract the values of the derivatives. Examples of such implementations are the Fortran package of [Pryce and Reid 1998] and the C package of [Griewank et al. 1996]. *Source Transformation* requires the development of sophisticated compiler-like software to read in a computer program, determine which statements require differentiation, and then write a new version of the original program augmented with statements to calculate derivatives. Examples of such implementations are the Fortran packages of [Bischof et al. 1998; FastOpt 2003; Tapenade 2003] and the C package of [Bischof et al. 1997].

Compared with languages such as C and Fortran, there has been relatively little work concerning AD in MATLAB. Rich and Hill [Rich and Hill 1992] provided a limited facility that enabled AD of simple arithmetic expressions defined by a character string. Such strings, together with necessary values of variables were passed to an external routine written in Turbo-C for differentiation. The most significant work to date is that of Verma and Coleman [Verma 1998; Coleman and Verma 1998b; 1998a] who, in a monumental coding effort, produced an operator-overloading AD package named ADMAT that provides facilities for forward and reverse mode AD for both first and second derivatives and runtime Jacobian sparsity detection. These authors also interfaced ADMAT with ADMIT [Coleman and Verma 2000], a package for efficient sparse Jacobian calculation via various coloring algorithms. A recent development is the ADIMAT hybrid source-transformation/operator overloading AD

tool [Vehreschild 2001]. Comparisons [Bischof et al. 2003] have shown its forward mode to be more efficient than that of ADMAT. Although ADMAT's operation count appears to be in agreement with AD theory [Coleman et al. 2000], its run time does not [Bischof et al. 2003]. With the goal of improving the performance of AD in MATLAB, we developed the MAD package [Forth 2001].

3.1 The MAD AD Tool

Version 1 of MAD provides an operator-overloaded implementation of forward-mode AD using the object-oriented programming features of MATLAB. Our primary aim was to secure the advantages of numerically exact evaluation of derivatives in a time comparable to forming finite difference approximations. We designed the software so that its performance can be profiled and optimized. Further, we designed it so that users could extend the functionality of the package.

The vast majority of the functions for MAD version 1 is associated with two MATLAB classes—`fmad` and `derivvec`. At present the `fmad` class consists of over 60 functions. It includes a class constructor and overloaded versions of most of MATLAB's arithmetic operations and many of its intrinsic functions. Within variables of the `fmad` class, the value of the variable is stored as a MATLAB array component. Multiple directional derivatives are stored in a component of `derivvec` class. The functions of the `derivvec` class enable linear combinations of directional derivatives to be performed in an efficient manner.

3.1.1 *Evaluating Jacobians with MAD.* The use of MAD is straightforward. One of the example problems of [Shampine et al. 2003] requires evaluating the partial derivative with respect to `y` of a function like

```
function f = odes(x,y)
n = -0.1; s = 0.2; c = -(3 - n)/2;
f = [ y(2); y(3); 1+s*y(2); y(5); s*(y(4) - 1) ];
```

for arguments

```
x = 0;
y = [0; 0; -0.96631; 0; 0.65291];
```

Using the `fmad` constructor, we first initialize `y_ad`, an `fmad` equivalent of `y`, with the value given by `y` and the 5×5 identity matrix:

```
y_ad = fmad(y,eye(5));
```

Each column of the identity matrix defines a separate directional derivative for the 5 elements of `y`. With this definition the first directional derivative of `y` is $(1\,0\,0\,0\,0)^T$ and the first directional derivative of any output corresponds to the derivative with respect to `y(1)`. Similarly, the second directional derivative of `y` is $(0\,1\,0\,0\,0)^T$ and the second directional derivative of any output corresponds to the derivative with respect to `y(2)`. The set of all 5 directional derivatives of any output corresponds to its Jacobian with respect to `y`. It now remains to evaluate the function with the arguments x and `y_ad` by

```
f = feval(@odes,x,y_ad);
```

a) Original Code

```
function res = bcs(ya,yb,P)
global d
res = zeros(6,1);
res(1:4) = ya - series(d,P);
res(5:6) = [ yb(3); ...
  (yb(4)-(-yb(1)*yb(2))) ];
```

b) Modified Code

```
function res = bcs(ya,yb,P)
global d
res = zeros(size(ya));
res(1:4) = ya - series(d,P);
res(5:6) = [ yb(3); ...
   (yb(4)-(-yb(1)*yb(2))) ];
```

Fig. 1. Assigning an `fmad` object to a component of a class `double` object

This evaluation uses overloaded versions of MATLAB arithmetic operations and functions to propagate values of all variables and their 5 directional derivatives through the calculation of `f`. The optimized `getinternalderivs` function of the `fmad` class is then used to extract the 5×5 Jacobian of `f` with respect to `y`.

```
>> dfdy = getinternalderivs(f)

dfdy =

         0    1.0000         0         0         0
         0         0    1.0000         0         0
         0    0.2000         0         0         0
         0         0         0         0    1.0000
         0         0         0    0.2000         0
```

3.1.2 *Overloaded AD in* MATLAB *versus Typed Languages.* Overloaded AD is much more convenient in MATLAB than in typed languages such as Fortran and C. MATLAB is an object-oriented language which manipulates objects of various classes. Those unfamiliar with the object-oriented approach should consider a class as equivalent to the familiar type of a data item and an object an instance of the class or, in other words, a variable. In MATLAB an object's class is determined by the class of the data first assigned to it. No declarations are required; indeed, they are not possible. Consequently, an advantage of implementing AD by overloading in MATLAB is that there are no declarations to change as there would be in C or Fortran. Because of this, programmers rarely need to make any changes to their existing MATLAB code before applying MAD to calculate Jacobians. This makes possible the automated use of AD in software requiring derivatives such as `bvp4c`.

3.1.3 *Limitations of MAD.* There are two important limitations when using MAD. Because MATLAB has a rich set of intrinsic functions, there are still functions that have not been developed as overloaded `fmad` routines. We have pursued a policy of adding functionality as test cases have been supplied so that we can test properly the new routines.

The second limitation has to do with changing the class of an object. Suppose, for example, that an object is of class `double`. If we assign an `fmad` object to one of its components, we need to change the class of the object itself to `fmad`. Unfortunately, it is not possible to do this automatically in MATLAB. A particularly relevant example is preallocation of arrays using the `zeros` intrinsic function. An example from [Shampine et al. 2003] is considered in Figure 1(a). The function begins by

initializing `res` as a 6×1 vector of class `double`. If the parameter `ya` is of class `fmad`, the next command attempts to assign a variable of class `fmad` to a portion of a vector in class `double`. On encountering this, MATLAB uses the `double` function associated with the `fmad` class to convert (or cast) the result of the right hand side of the statement to class `double` prior to the assignment to the left hand side. As a consequence, derivative information is not propagated through the assignment and the `fmad double` function issues a warning. This limitation is unfortunate because it means that using MAD is not entirely automatic. Nevertheless, it is usually easy enough to recode a function to circumvent the difficulty. For example, in Figure 1(b) we use the `fmad` overloaded `size` function to return an object of `fmad` class with an appropriately sized zero matrix for its derivatives. This allows the number of derivatives to be passed to the `fmad` overloaded `zeros` function which also returns an appropriately sized zero matrix as its derivative component.

3.2 Finite differences versus AD

We use the example of §3.1.1 to contrast the approximation of Jacobians by finite differences and AD. It took 1.78s to calculate 1000 Jacobians using MAD with MATLAB 6.5. Correspondingly, it took 0.69s to approximate 1000 Jacobians with finite-differencing via `numjac`

```
f = odes(t,y);
[dfdy,fac] = numjac(@odes,t,y,f,threshold,fac);
```

Generally the cost of approximating a Jacobian with MAD is comparable to finite differences. This example shows as big a difference as any we encountered in our experiments. That is enough to make it interesting, but there are other points of interest.

The `numjac` routine requires a vector `threshold` which indicates the scale of the problem. In this computation we used the same values that `bvp4c` uses, namely,

```
threshval = 1e-6;
threshold = threshval(ones(n,1));
```

Two things are unusual about `numjac`. The vector `fac` is used to adapt the increments of the finite difference approximations to the columns of the Jacobian from one call to the next. Also, if a column should be computed as the zero vector, `numjac` will adjust the increment for that column and try again. The first column of this Jacobian is actually zero, so `numjac` does more work than most finite difference routines as it tries to decide whether the zero column is the result of an increment that is off scale. Two of the entries computed by `numjac` are not very accurate. For instance, the $(3, 2)$ entry should be 0.2, but it is approximated as 0.2012. The other non-zero entry in this column is the $(1, 2)$ entry. This entry is 1 and it is computed exactly. Because the increment is chosen to give an accurate value for the largest entry and the same increment is used for all components in a column, it is not surprising that a smaller component is not very accurate. A little experiment exposes the difficulties of approximating Jacobians with finite differences: If we reduce `threshold` by a factor of 0.01, the function computes an approximation of 0 for the (3,2) entry! An increment that is out of scale can result in a very bad approximation like this one. For this example, and the BVP that

gives rise to it, the choice made for `threshold` in `bvp4c` works well enough, but it is always possible that the choice results in a poor approximation. In contrast, use of MAD circumvents any such problems because it produces a Jacobian without the truncation errors associated with finite differences and is accurate to floating point roundoff.

It is very important to understand that BVP solvers need only approximate Jacobians because they are used in a linearization that is itself an approximation. Generally the larger entries of a Jacobian are the ones most important in a linearization. Fortunately, it is easier to approximate the larger entries of a Jacobian by finite differences because they correspond to components of the function that change by a relatively large amount when the increment is changed. Codes for solving BVPs and stiff IVPs depend on these facts. However, sometimes the smaller entries in an approximate Jacobian *are* important. Moreover, the domain of convergence of the linearization is affected adversely by a poor approximate Jacobian. As a consequence, the "exact" Jacobians provided by AD are by no means necessary, but they may lead to faster convergence and with a larger domain of convergence. Indeed, they may be critical to getting convergence when solving difficult problems.

Having introduced MAD, and in particular the `fmad` class, we now consider applying MAD to the MATLAB BVP solver `bvp4c`.

4. AD IN BVP4C

We have developed a solver, `bvp4cAD`, based on `bvp4c` that uses AD by default. It proved possible to do this in a nearly seamless way. In this section we explain some of the issues.

One issue is what to do about analytical partial derivatives supplied by the user. We decided that if the user supplies either of the functions for evaluating partial derivatives analytically, `bvp4cAD` should use it. This is done partly to respect the instructions of the user and partly because such functions should be faster than evaluating partial derivatives with MAD.

It is characteristic of an operator overloading implementation of AD that in the course of computing a partial derivative of a function, the function itself is evaluated. We modified `bvp4c` to exploit this characteristic, but it is not as advantageous as it might at first seem. That is because evaluation of the function at mesh points is often not accompanied by evaluation of partial derivatives: When a new mesh is formed, it is necessary both to evaluate $\mathbf{f}$ at all the mesh points and the boundary conditions function $\mathbf{g}$. We reorganized some of the computations in `bvp4c` so as to defer evaluation of the functions until they are obtained as a byproduct of the evaluation of the partial derivatives. The global system is then linearized using these partial derivatives and the matrix is factored. In a simplified Newton iteration, this matrix is used until convergence is achieved or a new linearization is made to improve the rate of convergence. The test on convergence of this iteration involves evaluation of the functions at each iterate. However, we benefit from the evaluation of functions whilst evaluating partial derivatives only when a new mesh is formed. This happens when the iteration fails to converge or converges, but the accuracy is unacceptable.

All the popular BVP solvers use a simplified Newton iteration in the manner just

described for `bvp4c`, even when they are provided analytical partial derivatives. That is because linear algebra is a considerable fraction of the total cost of solving a BVP. To reduce the number of times that the linearized system is formed and factored, a matrix decomposition is used as long as convergence is adequate. The price is slower (linear) convergence and a reduced domain of convergence. With the availability of exact partial derivatives by means of AD and the relatively fast linear algebra of MATLAB, this might not be the best way to proceed in `bvp4cAD`, but we have not yet investigated this matter.

To keep down the cost of approximating partial derivatives by finite differences, `bvp4c` obtains some by averaging. The scheme [Kierzenka and Shampine 2001] for deciding which Jacobians to approximate by averaging seems to work well, but averaging is not done when analytical partial derivatives are available and correspondingly, the algorithm is somewhat stronger then. Naturally, in `bvp4cAD` we use the stronger algorithm for both analytical and AD partial derivatives.

The generality of the functions accepted by `bvp4c` presents some difficulties. One is the presence of known parameters. Like many MATLAB functions, `bvp4c` allows users to pass known parameters as optional input arguments to the solver that are then passed to all the functions that it calls. Using `varargin`, `bvp4c` receives these arguments and loads them into a cell array `ExtraArgs` that is used for communication within the solver. In `bvp4cAD` we augment `ExtraArgs` with information that must be passed to the subfunctions that are differentiated with MAD. For example, we must pass the handle for the user's function for evaluating $\mathbf{f}$ to the subfunction that we actually use to evaluate the partial derivatives of $\mathbf{f}$ with MAD. Another difficulty is the presence of unknown parameters $\mathbf{p}$ that are an optional vector argument of `bvp4c`. When present it is necessary to compute partial derivatives of both $\mathbf{f}$ and $\mathbf{g}$ with respect to these parameters. There is a little complication simply because there are two cases. In the following we detail the more involved case of unknown parameters.

4.1 Calculating partial derivatives $\partial\mathbf{f}/\partial\mathbf{y}$, $\partial\mathbf{f}/\partial\mathbf{p}$

To form the Jacobian of the residual associated with the collocation method, `bvp4c` requires the Jacobian $\partial\mathbf{f}(x_i, \mathbf{y}_i, \mathbf{p})/\partial\mathbf{y}_i$ associated with the approximation $\mathbf{y}_i$ to the solution at the mesh point x_i. Also, if there are unknown parameters $\mathbf{p}$, the partial derivative $\partial\mathbf{f}(x_i, \mathbf{y}_i, \mathbf{p})/\partial\mathbf{p}$ is required. In `bvp4c` the user can supply analytical expressions for these partial derivatives by coding up a function `[dfdy,dfdp] = fjac(x,y,p)` and passing its handle to `bvp4c` using the `bvpset` utility. Though not shown here, the call list for `fjac` may include known parameters that are passed to the function as optional input to `bvp4c`. The MAD equivalent of `fjac` is

```
function [dfdy,f,dfdp] = AD_fjac(x,y,n,npar,ode,ExtraArgs)
```

Notice that the function value `f` is returned along with the partial derivatives. The cell array `ExtraArgs` holds $\mathbf{p}$ and any known parameters that are to be passed to `ode`. Unlike `fjac`, which must be coded afresh for each set of differential equations, `AD_fjac` is part of the `bvp4cAD` package and uses MAD to calculate automatically the required partial derivatives. We now discuss more fully how this is done.

Since the active variables are the n entries of $\mathbf{y} = \mathbf{y}_i$ and the $npar$ entries of $\mathbf{p}$, the initialization of `y_ad` and `p_ad`, the MAD equivalents of $\mathbf{y}$ and $\mathbf{p}$, is accomplished

by

```
y_ad = fmad(y,eye(n,n+npar));
p_ad = fmad(ExtraArgs{1},[zeros(npar,n),eye(npar)]);
```

Effectively this uses the first n rows of an $(n + npar) \times (n + npar)$ identity matrix for the derivatives of $\mathbf{y}$ and the last $npar$ rows for the derivatives of $\mathbf{p}$. By virtue of operator overloading, a call to `ode` produces not only the value of the function, but also the values of its derivatives. They are returned as an object, so we must first extract the values of the function and the derivatives and then the portions of the derivatives that correspond to $\partial\mathbf{f}/\partial\mathbf{y}$ and $\partial\mathbf{f}/\partial\mathbf{p}$:

```
f_all = feval(ode,x,y_ad,p_ad,ExtraArgs{2:end});
f = getvalue(f_all);
f_derivs = getinternalderivs(f_all);
dfdy = f_derivs(:,1:n);
dfdp = f_derivs(:,n+1:n+npar);
```

4.2 Taking Advantage of Vectorization

Suppose that there are N mesh points x_i. Corresponding to `AD_fjac` is a function that deals with all N vectors $\mathbf{y}_i$ assembled as an $n \times N$ matrix `Y`:

```
function [dFdy,F,dFdp] = AD_Fjacvec(x,Y,n,npar,ode,odevec,ExtraArgs)
```

This allows us to vectorize the initialization of all MAD variables and the extraction later of all derivatives. We can also take advantage of vectorized $\mathbf{f}$. Whether or not $\mathbf{f}$ is vectorized, this is more efficient than a loop that calls the `ADf_jac` function of §4.1 for each $(x_i, \mathbf{y}_i)$.

The `Y_ad` corresponding to the matrix `Y` needs to be initialized with N copies of the derivative matrix used in the non-vectorized case. This could easily be achieved using MATLAB's `repmat` function,

```
Y_ad = fmad(Y,repmat(eye(n,n+npar),[N,1]))
```

but it is more efficient to use

```
A = eye(n,n+npar);
n_ind = (1:n)';
Y_ad = fmad(Y,A(n_ind(:,ones(1,N)),1:n+npar));
```

with the parameter vector `p_ad` as before,

```
p_ad = fmad(ExtraArgs{1},[zeros(npar,n),eye(npar)]);
```

If the `ode` function is vectorized, `odevec = true`, all the necessary Jacobians can be computed in one call of the `ode` function with the `Y_ad` and `p_ad` as arguments. If not, they are computed in a loop with a preallocated `F_all fmad` variable as required for efficient MATLAB execution.

```
if odevec
   F_all = feval(ode,x,Y_ad,p_ad,ExtraArgs{2:end});
else
   F_all = fmad(zeros(n,N),zeros(n,N,n+npar));
```

```
   for i = 1:N
       F_all(:,i)=feval(ode,x(:,i),Y_ad(:,i),p_ad,ExtraArgs{2:end});
   end
end
```

In §2.1.1 we discussed the two roles of vectorization in `bvp4c`. We see here a third role in `bvp4cAD`. It is interesting that vectorization of the `ode` function is key to the efficient approximation of Jacobians by finite differences, but we are also able to use it to improve the efficiency of computing Jacobians by AD.

Derivatives extraction can also be performed using vector operations:

```
F = getvalue(F_all);
F_derivs = permute(getderivs(F_all),[1 3 2]);
dFdy = F_derivs(:,1:n,:);
dFdp = F_derivs(:,n+1:n+npar,:);
```

A slight complication is that because of the way derivatives are stored within the `fmad` class, we must permute the derivative arrays to get the right shape.

4.3 Functions Not Supported

A very important issue is that `bvp4cAD` handle gracefully the possibility that MAD cannot evaluate a partial derivative. A small problem that we constructed to illustrate this is

$$y'' = f(y), \quad y(0) = 1, \quad y(1) = 2$$

for $f(y) = \cosh(y)$. At present MAD cannot compute the partial derivative f_y of this special function, but it can if the function is written in the equivalent form $f(y) = (e^y + e^{-y})/2$. We begin the computation in `bvp4cAD` with an attempt to evaluate each of the two kinds of partial derivatives. We use an exception handling construct to trap errors. If there is an error, the solver recognizes that it must use finite differences to approximate the partial derivative. For this example, the solver finds that it must use finite differences to approximate partial derivatives of f, but it can use MAD to compute the partial derivatives of this second form of f. Solving the problem in this way resulted in the output

```
The solution was obtained on a mesh of 6 points.
The maximum residual is 5.690e-004.
There were 107 calls to the ODE function.
There were 7 calls to the BC function.
```

For the second form of $f(y)$, `bvp4cAD` is able to use MAD for all the partial derivatives. The output is then

```
The solution was obtained on a mesh of 6 points.
The maximum residual is 5.690e-004.
There were 71 calls to the ODE function.
There were 5 calls to the BC function.
```

The number of calls to the ODE function is reduced in the second computation mainly because the Jacobians are evaluated by AD rather than approximated by

finite differences. AD was used to approximate the partial derivatives of the boundary conditions in both computations, so a reduction in the number of calls to the BC function means that fewer iterations were done when more accurate Jacobians were used. This is an interesting observation, but the point of this example is to show that `bvp4cAD` will recognize and respond properly to a partial derivative that MAD is not able to form.

5. ILLUSTRATIVE COMPUTATIONS

5.1 Fluid Injection Problem

Example 1.4 of [Ascher et al. 1995] illustrates a number of the issues that we have discussed. To solve this problem with `bvp4c` we must first write its differential equations,

$$\begin{aligned} f''' - R[(f')^2 - ff''] + RA &= 0 \\ h'' + Rfh' + 1 &= 0 \\ \theta'' + P_e f \theta' &= 0 \end{aligned}$$

as a system of seven first order equations. In the range of sizes we think typical in applications of `bvp4c`, this is a moderately large number of equations. There are three physical parameters in these equations. The Reynolds number R and the Peclet number $P_e = 0.7R$ are known. The parameter A is unknown. Many BVP solvers have no provision for unknown parameters, but `bvp4c` does, so no further preparation of the problem is necessary. As we have explained, both known and unknown parameters present certain difficulties for the addition of AD to `bvp4c`. Because of the unknown parameter, there are eight boundary conditions,

$$\begin{gathered} f(0) = f'(0) = 0, \quad f(1) = 1, \quad f'(1) = 1 \\ h(0) = h(1) = 0, \quad \theta(0) = 0, \quad \theta(1) = 1 \end{gathered}$$

which happen to be linear and separated.

In a first set of numerical experiments we initialized all components of the solution and the unknown parameter to have constant value 1. We also believed that an initial mesh of 10 equally spaced points would reveal the behavior of the solution. This initial solution and mesh was formed into a structure for passing to the solver by

```
solinit = bvpinit(linspace(0,1,10),ones(7,1),1)
```

Using this structure we solved the BVP with both bvp4c and bvp4cAD in MATLAB 6.1 with default tolerances and a range of R. We found that `bvp4c` failed for Reynolds numbers greater than 200, but `bvp4cAD` was able to get convergence for Reynolds numbers up to 350. As with any scheme for computing the solution of a set of nonlinear algebraic equations, the details of how a particular initial solution interacts with the computation can determine whether or not convergence takes place. That said, we see here a worthwhile improvement in the domain of convergence when exact partial derivatives are available through AD.

This BVP is hard to solve for large Reynolds numbers, so it is suggested in the texts [Ascher et al. 1995] and [Shampine et al. 2003] that it be solved by continuation. This means that the solution and the corresponding mesh for one

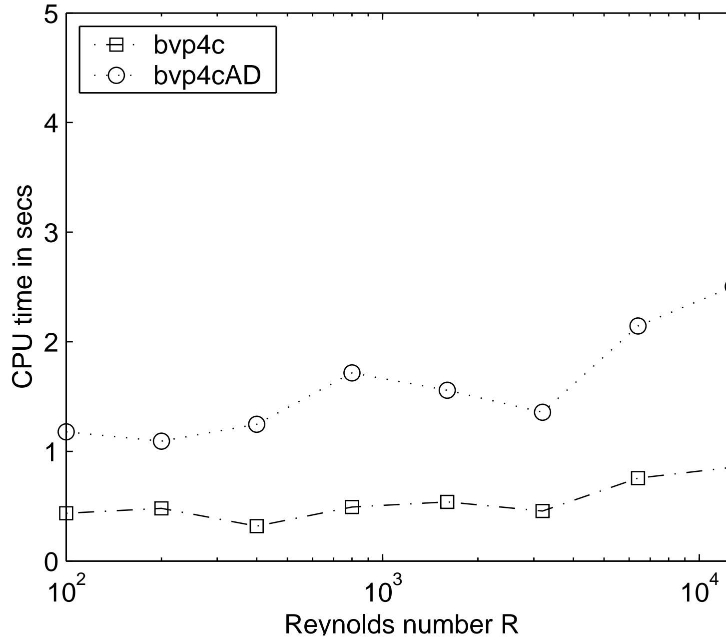

Fig. 2. Fluid Injection Problem - unvectorized

value of R are used as initial values for a bigger value of R. This is repeated until reaching the desired value of the Reynolds number. Continuation provides a way of solving hard BVPs and when continuing in a physical parameter like the Reynolds number, the intermediate solutions are also of interest. Continuation is so important in the practical solution of BVPs that `bvp4c` was designed to make it easy.

In a second set of experiments we started with a solution and mesh for $R = 100$ and doubled the Reynolds number at each stage of continuation. Figure 2 contrasts the default finite difference approximation of partial derivatives in `bvp4c` and the exact evaluation of partial derivatives by AD of `bvp4cAD`. Although the performance with AD is quite acceptable, it is not advantageous as regards run time for this problem.

In the text [Shampine et al. 2003] it is pointed out that analytical partial derivatives and vectorization are very advantageous for this problem. Our experience with this example is that users have had trouble working out the analytical partial derivatives. If the shapes of the matrices are wrong, the computation fails immediately and if the partial derivatives are wrong, the code may fail to converge or converge slowly. Vectorization is not always easy, but in our experience users have had less difficulty with this. In a third set of experiments we solved the BVP as in the previous set, but now with $\mathbf{f}$ vectorized. Figure 3 shows that supplying analytical partial derivatives reduces the run time significantly. The run times for `bvp4cAD` are significantly lower than those for `bvp4c` with default finite differencing and are close to those of `bvp4c` when it is provided functions for the partial derivatives.

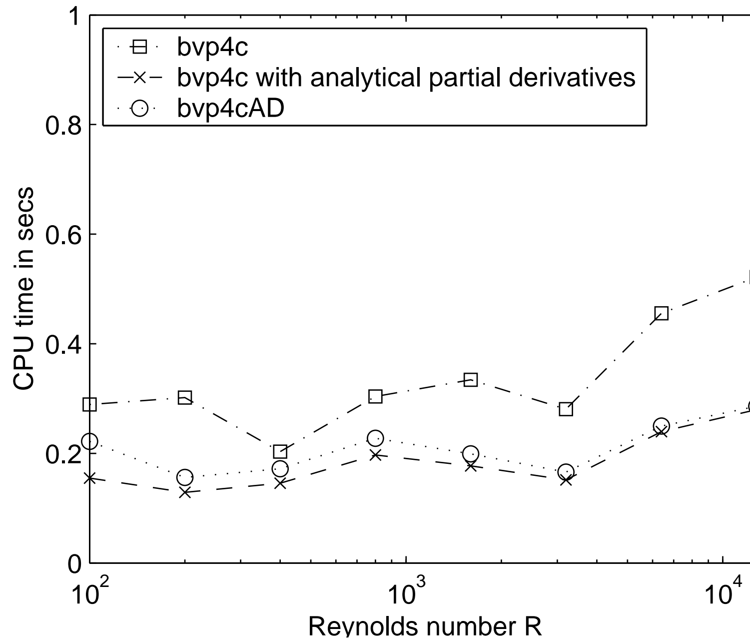

Fig. 3. Fluid Injection Problem—vectorized

The difference between `bvp4cAD` and `bvp4c` with analytical derivatives is seen to decrease with increasing Reynolds number. As the Reynolds number increases, the boundary layer becomes thinner. The BVP solver is of fixed order, so it deals with this by introducing more mesh points. Because the computations are vectorized, the overhead of using AD becomes relatively less important as the Reynolds number is increased. Indeed, as the mesh is refined, the efficiency of AD approaches that obtained with user-supplied partial derivatives.

5.2 A Collection of Problems

The text [Shampine et al. 2003] provides a tutorial on the use of `bvp4c`. Eight examples show what kinds of problems can be solved and how to prepare problems so as to deal with various kinds of difficulties. In aggregate they provide a test of `bvp4cAD` for all common situations and some that are not so common. To solve these problems, all we had to do was change the name of the function from `bvp4c` to `bvp4cAD`. The computations were successful and the results of the two codes were consistent, so the use of AD was transparent in all cases. In Table I we compare the run times of the two solvers on a Pentium IV 2.40GHz. We note that problems 5 and 6 have functions that are vectorized and problems 2,3,4, and 6 involve known parameters. Problem 7 stands out because AD is exceptionally inefficient. Investigation of this BVP led to the simplified problem that we used in §3.2 to contrast finite differences and AD.

For nearly all problems the run times increased from MATLAB version 6.1 to 6.5 for both `bvp4c` and `bvp4cAD`, although by not as much for `bvp4cAD`, thus improving

Table I. CPU times for Examples of [Shampine et al. 2003].

	MATLAB 6.1			MATLAB 6.5		
Problem	`bvp4c`	`bvp4cAD`	Ratio	`bvp4c`	`bvp4cAD`	Ratio
1	0.12	0.13	1.05	0.25	0.16	0.63
2	0.22	0.09	0.39	0.28	0.11	0.38
3	0.54	1.33	2.48	0.63	1.66	2.64
4	1.28	2.18	1.70	1.78	3.04	1.71
5	0.67	0.58	0.86	0.78	0.61	0.78
6	1.65	0.91	0.55	1.94	1.08	0.56
7	1.35	7.21	5.32	1.11	9.01	8.13
8	0.16	0.36	2.26	0.22	0.48	2.15

the ratio of times for some problems. The main difference between versions 6.1 and 6.5 is the introduction of the JIT Accelerator [MathWorks 2002]. However, due to use of vectorization within `bvp4c` this does not seem to improve run times for the calculations we perform. We have also run the test cases on a Laptop Intel Pentium M processor, a Pentium IV running Linux and an Ultra 10 Solaris machine. On all these platforms the results were similar to those of Table I, although the differences in absolute run time between the two versions of MATLAB were not as large.

6. CONCLUSIONS

Our goal was to obtain the advantages of analytical partial derivatives in `bvp4c`, namely improved robustness and efficiency, without giving up any of the capabilities and convenience of the solver. Though not entirely straightforward, we have achieved this using the MAD tool. The tool has minor limitations discussed in §3.1.3, but when it cannot provide the requested partial derivatives, `bvp4cAD` recognizes this and automatically turns to the finite difference approximation of partial derivatives that are the default in `bvp4c`. The partial derivatives are generally evaluated quite efficiently by MAD, but users do have the option of supplying functions for the analytical evaluation of partial derivatives should this be desirable. Vectorization plays several roles in `bvp4c`, one of which is to speed up the approximation of partial derivatives by finite differences. This role falls away in `bvp4cAD`, but the other, and more important, role of speeding up the computation of residuals is still played in `bvp4cAD`. Interestingly, we discovered that we could use vectorization to speed up the evaluation of partial derivatives by AD in `bvp4cAD`.

REFERENCES

ASCHER, U., MATTHEIJ, R., AND RUSSELL, R. 1995. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. SIAM, Philadelphia, PA, USA.

BISCHOF, C., LANG, B., AND VEHRESCHILD, A. 2003. Automatic differentiation for MATLAB programs. *Proc. Appl. Math. Mech 2,* 1, 50–53.

BISCHOF, C. H., CARLE, A., HOVLAND, P. D., KHADEMI, P., AND MAUER, A. 1998. ADIFOR 2.0 user's guide (Revision D). Tech. rep., Mathematics and Computer Science Division Technical Memorandum no. 192 and Center for Research on Parallel Computation Technical Report CRPC-95516-S. See `www.mcs.anl.gov/adifor`.

BISCHOF, C. H., ROH, L., AND MAUER, A. 1997. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software – Practice and Experience 27,* 12, 1427–1456. See `www-fp.mcs.anl.gov/division/software`.

COLEMAN, T. F., SANTOSA, F., AND VERMA, A. 2000. Efficient calculation of Jacobian and adjoint

vector products in the wave propagational inverse problem using automatic differentiation. *Journal of Computational Physics 157*, 234–255.

Coleman, T. F. and Verma, A. 1998a. ADMAT: An automatic differentiation toolbox for MATLAB. Tech. rep., Computer Science Department, Cornell University.

Coleman, T. F. and Verma, A. 1998b. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput. 19,* 4, 1210–1233.

Coleman, T. F. and Verma, A. 2000. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Trans. Math. Softw. 26,* 1 (Mar.), 150–175.

FastOpt 2003. *Transformation of Algorithms in Fortran, Manual, Draft Version, TAF Version 1.6*. FastOpt. See http://www.FastOpt.com/taf.

Forth, S. A. 2001. User guide for MAD - a Matlab automatic differentiation toolbox. Applied Mathematics and Operational Research Report AMOR 2001/5, Cranfield University (RMCS Shrivenham), Swindon, SN6 8LA, UK. June.

Griewank, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Penn.

Griewank, A., Juedes, D., and Utke, J. 1996. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software 22,* 2, 131–167.

Kierzenka, J. and Shampine, L. 2001. A BVP solver based on residual control and the Matlab PSE. *ACM Trans. Math. Softw. 27*, 299–316.

MathWorks. 2002. Matlab 6.5 release notes.

Matlab 2000. Matlab *6*. The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760.

Pryce, J. D. and Reid, J. K. 1998. ADO1, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England. See ftp://matisa.cc.rl.ac.uk/pub/reports/prRAL98057.ps.gz.

Rich, L. C. and Hill, D. R. 1992. Automatic differentiation in MATLAB. *App. Num. Math. 9*, 33–43.

Shampine, L., Gladwell, I., and Thompson, S. 2003. *Solving ODEs in Matlab*. Cambridge University Press, Cambridge, UK.

Shampine, L. and Reichelt, M. 1997. The Matlab ODE Suite. *SIAM J. Sci. Comput. 18*, 1–22.

Shampine, L., Kierzenka, J. and Reichelt, M. Solving boundary value problems for ordinary differential equations in Matlab with bvp4c. See ftp://ftp.mathworks.com/pub/doc/papers/bvp/.

Tapenade 2003. The TAPENADE tutorial http://www-sop.inria.fr/tropics/tapenade/tutorial.html. Web Site.

Vehreschild, A. 2001. Semantic augmentation of MATLAB programs to compute derivatives. M.S. thesis, Institute for Scientific Computing, Aachen University, Germany.

Verma, A. 1998. Structured automatic differentiation. Ph.D. thesis, Cornell University Department of Computer Science, Ithaca, NY.

Received Month Year; revised Month Year; accepted Month Year