Automatic Fréchet Differentiation for the Numerical Solution of Boundary-Value Problems

ASGEIR BIRKISSON, University of Oxford
TOBIN A. DRISCOLL, University of Delaware

A new solver for nonlinear boundary-value problems (BVPs) in MATLAB is presented, based on the Chebfun software system for representing functions and operators automatically as numerical objects. The solver implements Newton's method in function space, where instead of the usual Jacobian matrices, the derivatives involved are Fréchet derivatives. A major novelty of this approach is the application of automatic differentiation (AD) techniques to compute the operator-valued Fréchet derivatives in the continuous context. Other novelties include the use of anonymous functions and numbering of each variable to enable a recursive, delayed evaluation of derivatives with forward mode AD. The AD techniques are applied within a new Chebfun class called `chebop` which allows users to set up and solve nonlinear BVPs, both scalar and systems of coupled equations, in a few lines of code, using the "nonlinear backslash" operator (\). This framework enables one to study the behaviour of Newton's method in function space.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*Automatic differentiation*; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations—*Boundary value problems*.

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Chebfun, Newton's method in function space, linearization of boundary-value problems, object-oriented MATLAB.

ACM Reference Format:
Birkisson, A. and Driscoll, T. A. 2012. Automatic Fréchet diffentation for the numerical solution of boundary-value problems. ACM Trans. Math. Softw. 38, 4, Article 26 (August 2012), 29 pages.
DOI = 10.1145/2331130.2331134 http://doi.acm.org/10.1145/2331130.2331134

1. INTRODUCTION

Automatic differentiation (or algorithmic differentiation—AD in both cases) provides the ability to differentiate functions without truncation error. AD starts from numerical values and produces numerical derivatives, gradients, and Jacobian matrices that are approximate only in the sense of roundoff. The efficient implementation of AD is described in detail in the excellent book by Griewank and Walther [2008], and we give a brief overview for our purposes in Section 3.

A typical use of AD is to remove the need for handwritten code or difference-based approximation methods to produce the Jacobian matrices within a Newton iteration

A. Birkisson is supported in this work by UK EPSRC Grant EP/E045847 and a Sloane Robinson Foundation Graduate Award associated with Lincoln College. T. A. Driscoll is supported in this work by UK EPSRC Grant EP/E045847.
Authors' addresses: A. Birkisson, University of Oxford Mathematical Institute, 24-29 St Giles, Oxford, OX1 3LB England; email: birkisson@maths.ox.ac.uk; T. A. Driscoll, Department of Mathematical Sciences, University of Delaware, Newark, DE 19716; email: driscoll@udel.edu.

DOI 10.1145/2331130.2331134 http://doi.acm.org/10.1145/2331130.2331134

for a nonlinear system of equations. In the special case of a nonlinear system arising from discretization of a nonlinear boundary-value problem this is the approach taken, for example, by the MATLAB function `bvp4cAD` [Shampine et al. 2006], which overloads MATLAB's built-in boundary-value problem solver `bvp4c` [Kierzenka and Shampine 2001]. Instead of using finite difference schemes to calculate Jacobian matrices, `bvp4cAD` uses AD to supply high accuracy partial derivatives, increasing the robustness and (at least in some cases) the efficiency of the solver. Interestingly, `bvp4cAD` finds a very accurate Jacobian for a nonlinear system of algebraic equations that is itself a discretization of the original boundary value problem and thus has incurred a truncation error. Moreover, the computational expense of AD becomes very significant as the number of system variables, that is, the total discretization size, becomes large.

We can switch the order of steps and linearize before discretization of the derivatives. The Newton iteration has a clear generalization to this context, variously referred to as *Newton's method in function space* or *quasilinearization* [Ascher et al. 1995; Deuflhard 2006]. The iterates of the function space version are updated through the solution of successive linear boundary-value problems. Each linear update problem is described by the *Fréchet derivatives* of the original BVP, linear operators analogous to Jacobian matrices [Hutson et al. 2005]. Under suitable conditions and with a sufficiently good initial guess u_0, the Newton iterates can be expected to converge quadratically to a solution of the original BVP. This process is described in more detail in Section 2.

Although conceptually quite different, Fréchet differentiation and differentiation of a function with respect to a real variable both rely on familiar algorithmic rules from calculus (e.g., the chain rule). Thus, Fréchet derivatives can in principle be produced through AD techniques. The idea is appealing in part because it produces a linearization in which the relevant number of variables is that of the original continuous system, not its discretization, and therefore the technique is not susceptible to the large-scale inefficiency of AD.

In this work we describe introducing the Fréchet AD idea into the Chebfun software system, which is a free downloadable add-on for MATLAB [Trefethen et al. 2011]. Chebfun uses the fast convergence and fast algorithms associated with the (piecewise) Chebyshev series representation of functions to provide the illusion of symbolic manipulation of their numerical representations. While each function is ultimately represented by a discretization of finite length by interpolants in suitably rescaled Chebyshev points

$$\cos(j\pi/n),\ 0 \le j \le n,$$

the length (n) is chosen automatically and dynamically so that the system can represent and manipulate most functions essentially to roundoff error with great speed. The chebfun method `length` returns the number of Chebyshev points used to represent a function, which is one more than the degree of the polynomial interpolant. Major operations available include rootfinding, integration, and differentiation.

Within Chebfun is an object class that allows one to create and manipulate linear operators on functions [Driscoll et al. 2008]. Each linear operator maintains both a functional form and the ability to instantiate itself as a matrix of arbitrary size in order to act on discretizations. In addition to standard arithmetic operators, the object class has methods `\`, `eigs` and `expm` for the solution of linear boundary-value, eigenvalue, and time evolution problems for differential, integral, and integro-differential operators.

We have extended Chebfun to apply forward-mode AD to find Fréchet derivatives of nonlinear operators. The AD techniques are applied only to the variables of the original operator, not the discretized variables. The result of the differentiation is a linear operator object, which can represent a linearized BVP in the Newton iteration. By pairing the Fréchet differentiation with a damped Newton iteration, we have created a new object class in Chebfun for nonlinear operators, capable of fully automatic solution of a wide range of nonlinear BVPs of the form

$$\phi(u) = 0,$$
$$\alpha(u)\big|_{x=a} = 0, \quad \beta(u)\big|_{x=b} = 0.$$

Here, ϕ, α, and β are (differential) operators; ϕ represents the differential equation, and α and β the left and right boundary conditions, respectively. They are further discussed in Section 2. Problems possible to solve include both scalar and systems of coupled equations of any order, although beyond fourth order, poor matrix conditioning becomes an overriding concern unless special steps are taken [Don and Solomonoff 1997]. The solutions returned are in many cases obtained at accuracies and speeds not available from well-known MATLAB methods, most notably `bvp4c` and `bvp5c` [Kierzenka and Shampine 2007] (and the existing Chebfun overloads for them[1]). Furthermore, the problem specification syntax for the new methods is simpler and more direct than that of the `bvp4c/5c` family; for instance, there is no need to write a high-order problem as a first-order system.

The linear operator class in Chebfun was originally given the name `chebop`. In recognition of the greater generality available from the work described here, the `chebop` class name applies to the new class for nonlinear operators as of Chebfun version 4. The syntax and capabilities for linear operators from version 3 have been migrated to work with a newly named `linop` class. For most users and problems, however, the chebop syntax is now the most appropriate one.

From the given description of BVPs, a computational object such as a chebop which may be used to solve BVPs requires the following components, here given with their mathematical symbols and name of chebop fields used to represent them.

Component	Mathematical symbol	chebop field
Interval on which problem is defined	$[a, b]$	`dom`
Differential equation operator	ϕ	`op`
Boundary condition operators	α and β	`lbc` and `rbc`
Initial guess of the solution	u_0	`init`

The fields of a chebop can be set either in the chebop constructor when it is created, or later by accessing them directly (e.g., with `N.init = 0`). The chebop class is further discussed in Section 4.2.

As a prototype example of chebop use, solving the problem

$$u'' + 2u \sin u = 0, \quad 0 < x < 5, \quad u'(0) = 0, \quad u(5) = 1, \tag{1}$$

for the function $u = u(x)$, becomes simply

```
% Create an anonymous function for the differential equation
phi = @(u) diff(u,2)+2*u.*sin(u);
% Anonymous functions for left and right boundary conditions
alpha = @(u) diff(u);  beta = @(u) u-1;
% Create a chebop object representing the BVP on the interval [0,5].
N = chebop(phi,[0 5],alpha,beta);
```

[1]Chebfun offers overloaded `bvp4c/5c` methods, which call the original methods for computing solutions. They differ from the original methods mainly in that their outputs are chebfuns, rather than vectors.

```
% Use the initial guess of the solution u0 = pi
N.init = pi;
% Obtain a solution to the problem N(u)=0 using nonlinear backslash
u = N\0;
```

With the default tolerance (as described in Section 4.1) of 10^{-10}, this solution takes about 0.62 seconds on a tri-core 2.5 GHz workstation[2], delivering the converged solution u as a polynomial of degree 58 with a residual 2-norm, $\|\phi(u)\|$, less than $4 \cdot 10^{-11}$. (We acknowledge that that small residuals do not necessarily indicate that solutions are close to the true ones; in Section 4.4 we compare the chebop solutions to the analytical ones for problems where the analytical solutions are known.)

Section 3 explains how forward-mode AD techniques are implemented to find Fréchet derivatives automatically. Because Chebfun has many uses besides the solution of nonlinear BVPs, we placed a high priority on not creating a large computational overhead for other uses. To keep with the highly interactive nature of Chebfun, we also desired a syntax as unobtrusive and automatic as possible. These goals led us to make what we believe to be two new contributions to AD software techniques. First, we exploit delayed evaluation, implemented using recursion and the assignment of identification tags to Chebfun objects. Thus, for example, if `f` is a chebfun created though expressions with chebfuns `u` and `v`, then calls of `diff(f,u)` and `diff(f,v)` initiate AD computations using information cached and encapsulated in `f`. Second, the independent and dependent variables are specified simultaneously; in the previous example, there is no need to designate `u` or `v` specially before the construction of `f` or calls to `diff`.

Section 4 describes the damped Newton iteration and other details behind the new chebop class and the "nonlinear backslash" capability, and provides numerous examples illustrating various aspects of flexibility in the system. It includes experimental comparison of this new solver with other BVP solvers in MATLAB. We find that if only low accuracy is requested, our solver is slower, but at higher requested accuracy, our solver is comparable to or faster then the others in execution time. Since the human time used for setting up many problems is in fact much longer than the actual running time for the computations, and the chebop syntax can be considered to be more convenient for the user than the one used for the other solvers, we believe this new method has significant benefits to offer. The section also includes a short introduction to *chebgui*, a graphical user interface to Chebfun. Finally, Section 5 mentions some of the limitations still imposed by the system and suggests avenues of future investigation and application.

2. NEWTON ITERATIONS IN CHEBFUN

The Newton iteration for a multivariate function $f : \mathbf{R}^n \mapsto \mathbf{R}^n$ generates a sequence of iterates $x^{(1)}, x^{(2)}, \ldots$, starting from an initial guess $x^{(0)}$, such that if $x^{(k)} \to x^*$ as $k \to \infty$, then $f(x^*) = 0$ [Ascher et al. 1995]. Members of the sequence are defined successively via

$$x^{(k+1)} - x^{(k)} = -\left[f'\left(x^{(k)}\right)\right]^{-1} f\left(x^{(k)}\right), \tag{2}$$

where $f'(x)$ is the Jacobian matrix of partial derivatives $\partial f_i/\partial x_j$. We shall refer to the quantity on the right-hand side of (2) as $y^{(k)}$, the *Newton update*. The motivation for the update formula is the linearization

$$f(x) \approx f\left(x^{(k)}\right) + f'\left(x^{(k)}\right)\left(x - x^{(k)}\right),$$

[2]All running times cited in this article are based on the same machine.

which is rigorously justifiable using Taylor series. The update is chosen so that $x^{(k)}+y^{(k)}$ is a root of the linearization rather than of f itself. Recall that the use of the inverse matrix in (2) is a mathematical formality standing for the solution of a square linear system of algebraic equations using standard efficient algorithms. In practice, one can approximate the Jacobian or its inverse and obtain an approximate solution using a quasi-Newton method.

Now suppose we have a boundary-value problem of the form described in the introduction,

$$\begin{aligned} \phi(u) &= 0, \\ \alpha(u)\big|_{x=a} = 0, &\qquad \beta(u)\big|_{x=b} = 0, \end{aligned} \tag{3}$$

where u is a function (i.e., $u = u(x)$) and ϕ, α, and β are operators between suitable normed function spaces; typically, ϕ is a differential operator, and α and β may also be differential operators of lower order. When convenient, we shall refer to (3) simply as $\mathcal{N}(u) = 0$, that is, we let $\mathcal{N}$ denote a BVP operator. The Newton iteration can be generalized for operator equations to produce a sequence of functions $u_1, u_2, \ldots$ such that if $u_k \to u^*$ as $k \to \infty$, then $\mathcal{N}(u^*) = 0$. Such infinite-dimensional Newton methods are often known as Newton methods in function space or quasilinearization [Ascher et al. 1995; Deuflhard 2006], but we will not distinguish between the finite- and infinite-dimensional versions in our terminology.

The crucial component of the Newton iteration is the linearization of $\mathcal{N}$ about each u_k. This process requires the Fréchet derivatives of ϕ, α, and β, which are defined as follows [Hutson et al. 2005]: Let V and W be Banach spaces and let U be an open subset of V. Then for $\phi : U \to W$, the Fréchet derivative of ϕ at $u \in U$ is defined (when possible) as the unique linear operator $A = \phi'(u)$, $A : V \to W$, such that

$$\lim_{v\to 0} \frac{\|\phi(u+v) - \phi(u) - Av\|_W}{\|v\|_V} = 0.$$

Note in this definition that v is a function, and we require that the limit exists as $v \to 0$ in any manner.[3]

Conceptually, given the current iterate u_k, we replace the operator $\mathcal{N}(u_k+v)$ for a perturbation v by the linearization $\mathcal{N}(u_k)+\mathcal{N}'(u_k)v$, and find a root of this approximation to determine the *continuous Newton update*. Hence we obtain the linear boundary-value problem

$$\phi'(u_k)v = -\phi(u_k), \tag{4}$$

$$\big[\alpha'(u_k)v\big]_{x=a} = -\alpha(u_k)\big|_{x=a}, \quad \big[\beta'(u_k)v\big]_{x=b} = -\beta(u_k)\big|_{x=b}, \tag{5}$$

or using the notation introduced previously,

$$\mathcal{N}'(u_k)v = -\mathcal{N}(u_k).$$

Solving this problem for the function v to obtain the update v_k, we define $u_{k+1} = u_k + v_k$ and iterate. Under suitable conditions (essentially, the invertibility of the linearization at a solution of $\mathcal{N}(u) = 0$), the theorem of Kantorovich guarantees quadratic convergence for suitably close initial guesses [Deuflhard 2006].

[3]That is, given $\varepsilon > 0$, there exists $\delta > 0$ such that for $\|v\|_V < \delta$

$$\|\phi(u+v) - \phi(u) - Av\|_W \le \varepsilon\|v\|_V.$$

In order to help us set ideas and also introduce details regarding the Chebfun software, let us be explicit about the process for the nonlinear boundary-value problem

$$u'' + 2u \sin u = 0, \tag{6}$$
$$u'(0) = 0, \tag{7}$$
$$u(5)u'(5) = 2, \tag{8}$$

which is identical to (1) except with a nonlinear condition at the right boundary. Thus $\phi(u) = u'' + 2u \sin u$, $\alpha(u) = u'$, and $\beta(u) = uu' - 2$. We can find the relevant linearizations in standard perturbation fashion:

$$\begin{aligned}\phi(u + v) - \phi(u) &= v'' + 2v \sin(u + v) + 2u(\sin(u + v) - \sin(u)) \\ &= v'' + 2v \sin(u) + 2uv \cos(u) + O\big(\|v\|^2\big).\end{aligned}$$

Upon dropping high-order terms, this leads to the Fréchet derivative

$$\phi'(u) = \frac{d^2}{dx^2} + \xi(u),$$

where $\xi(u) = 2 \sin(u) + 2u \cos(u)$. This linear operator has one part that performs differentiation and another that performs pointwise multiplication by $\xi(u)$, such that

$$\phi'(u) : f(x) \mapsto f''(x) + \xi(u(x)) f(x).$$

Proceeding similarly for the boundary conditions, we find

$$\begin{aligned}0 &= u'(0) + v'(0), \\ 2 &= u(5)u'(5) + u(5)v'(5) + u'(5)v(5) + O\big(\|v\|^2\big).\end{aligned}$$

By comparison with (5), we identify

$$\alpha'(u) = \frac{d}{dx}, \quad \beta'(u) = u\frac{d}{dx} + u'.$$

Note that since α is itself linear, $\alpha'(u)$ is independent of u, and that $\beta'(u)$, like $\phi'(u)$, consists of a differentiation term and a multiplication term. Altogether, the Newton update v_k is defined as the solution of

$$v''(x) + \big[2 \sin(u_k(x)) + 2u_k(x) \cos(u_k(x))\big]v(x) = -\phi(u_k), \tag{9}$$
$$v'(0) = -u_k'(0), \tag{10}$$
$$u_k(5)v'(5) + u_k'(5)v(5) = -\big(u_k(5)u_k'(5) - 2\big). \tag{11}$$

The linearity of α implies that the iterates u_k will all satisfy the left boundary condition after the first Newton update, and (10) becomes a homogeneous condition. In practice this means that in the common case in which the boundary conditions are linear but nonhomogeneous, we can choose an initial guess for the Newton iteration without regard to these conditions and correct them after just one Newton step.

A Chebfun implementation of the Newton iteration for the BVP (6)–(8) is shown in the following code. We use the general chebop class to represent the linear operators involved, the system internals will take care of automatically converting them to linops which were discussed in the introduction. The iteration stops when the norm of the Newton update falls below 10^{-11}, which takes 11 iterations with the initial guess $u_0(x) = x$. In Figure 1 we plot the sequence of solution estimates, with an additional axis showing the cumulative 2-norms of the update functions. After the first six or so iterations, the quadratic convergence of the iteration can be readily confirmed from the numerical values of the residuals.

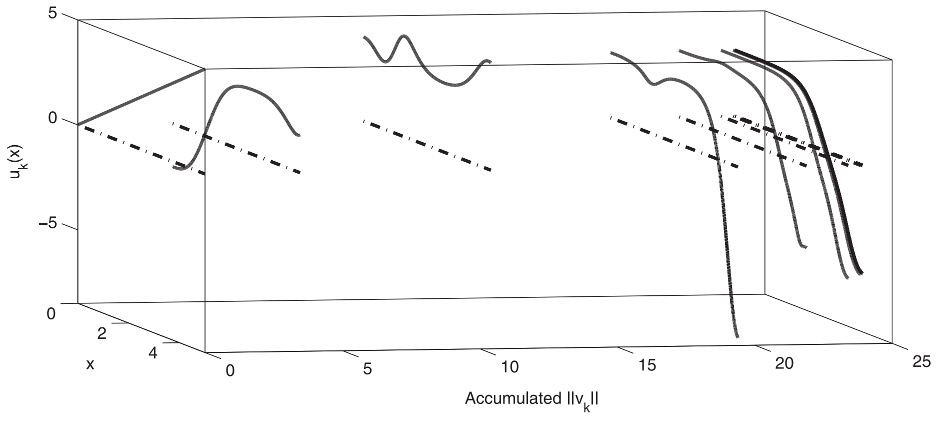


Fig. 1. The solution of the BVP (6)–(8) by the Newton iteration using Fréchet derivatives in Chebfun. The y axis shows the cumulative sum of $\|v_k\|_2$ for the Newton update functions v_k. Note that each dash-dotted line corresponds to one iteration, and as we get closer to a solution, the lines cluster (indicating that the updates are getting smaller). After some initial large updates, the iteration eventually converges quadratically to the solution.

```
A = chebop(0,5);                   % Operator on the interval [0,5]
x = chebfun('x',[0 5]);            % The chebfun "x" on the problem interval
phi = @(u) diff(u,2)+2*u.*sin(u);  % ODE part of the nonlinear BVP
u = x;                             % Initial guess
nrmv = 1;  y = 0;                  % Initialize variables
plot3(x,chebfun(y,[0 5]),u), hold on
while nrmv > 1e-11                                  % Newton iterations
  A.op =  @(v) diff(v,2)+(2*sin(u)+2*u.*cos(u)).*v; % Frechet derivative at u
  Du = diff(u);                                     % Needed to compute the BCs
  A.lbc = @(v) diff(v)+Du(0);                       % Neumann condition at x=0
  A.rbc = @(v) u(5).*diff(v)+Du(5)*v+u(5)*Du(5)-2;  % Robin condition at x=5
  v = A\(-phi(u));                                  % Solve the linearized BVP
  nrmv = norm(v);  y = y+norm(v);                   % 2-norm of Newton update
  u = u+v;
  plot3(x,chebfun(y,[0 5]),u)
end
```

A closer inspection of the results of running this program reveals a subtle anomaly. If we add the statement `length(u)` to the end of the Newton loop, we discover that the length of the vector of collocated function values (i.e., the value of the solution at Chebyshev points) used to represent the solution jumps from 2 (the linear initial guess) to 54, grows quickly to 150, stagnates, then jumps to 232 at the last iteration. This last iteration is particularly suspect: an update of size less than 10^{-11} is added to a function that is $O(1)$, yet the length of the numerical representation grows by more than 50%. The explanation boils down to the difference between absolute and relative accuracy. Each chebfun object is nominally resolved to high accuracy relative to its own scale. When that object is actually a Newton update v_k, however, it is clearly more appropriate to resolve it to the scale of the solution estimate u_k. To do this, we can add the line

```
  A.scale = norm(u);
```

just before the update BVP is solved via \. Making this change may have an impact on execution time, because many fewer Chebyshev series coefficients may then be needed to resolve v_k when its scale is much smaller than that of u_k. For this particular example, this change speeds up the solution of the linear problem in the last iteration by almost 20%, and the solution returned will be of length 150 rather than 232.

3. FRÉCHET DERIVATIVES BY AUTOMATIC DIFFERENTIATION

Automatic, or algorithmic, differentiation (AD) introduces techniques that allow a reliable and accurate way to obtain derivatives in scientific computing. For the standard reference on the subject, see [Griewank and Walther 2008]. Many ideas of how to accumulate derivatives have been described in the literature, of which the two best known strategies are forward and reverse mode. The two approaches differ in the way the derivatives are accumulated, that is, in how the chain rule is treated [Bischof and Bücker 2000]. In forward mode, the derivatives are computed at the same time as the function is evaluated, but in reverse mode, the function is first evaluated while information is stored that enables the computation of the derivative afterwards. As argued in Griewank and Walther [2008], forward mode should generally be used when the number of output variables is much greater than the number of input variables, whereas reverse mode should be used when the number of output variables is much less than the number of input variables.

An important concept in AD is that of the *evaluation trace*, defined as follows.

> An evaluation trace is basically a record of a particular run of a particular program, with particular specified values for the input variables, showing the sequence of floating point values calculated by a (slightly idealized) processor and the operators that computed them. [Griewank and Walther 2008, Page 4]

Each member of the sequence of operations is associated with an intermediate variables, and it is by storing information about and working with these intermediate variables that AD is able to compute derivatives.

Generally there are two methods used to implement AD: source transformation and operator overloading. Both methods have their advantages and disadvantages as described thoroughly in [Bischof and Bücker 2000]. We note that both methods have previously been implemented for MATLAB. A well known AD package for MATLAB is MAD—MATLAB Automatic Differentiation, on which `bvp4cAD` is based on; see Forth [2006] and Shampine et al. [2006]. MAD implements the operator overloaded forward mode of AD. Kharche and Forth [2006] present the MSAD package, a source transformation implementation of forward mode automatic differentiation for MATLAB based on MAD. Another established AD software tool for MATLAB is ADIMAT [Bischof et al. 2002]. Finally, we mention that the INTLAB (INTerval LABoratory) package also offers an operator overloaded forward mode of AD. For details of INTLAB, see Rump [1999].

Since the Chebfun project involves overloading MATLAB functions, it was logical to use overloading for our implementation of AD. Furthermore, since we can expect the number of variables involved in solving practical BVPs to be small, we chose to implement the more straightforward forward mode. Griewank and Walther [2008] suggest defining a new data structure (named *adouble*) for implementing this approach to AD. An adouble contains two floating point values, corresponding to the calculated values of the function and the derivative of that function. One can think of this as an act of associating a new field to the usual double precision numbers to enable calculations of derivatives. Similarly, to enable AD for chebfuns, we now introduce a new field in the chebfun object named `jac`, as an abbreviation for Jacobian.[4] When chebfuns are created using the constructor, we initialize the `jac` field to be empty, which will allow the recursive evaluation of derivatives later on.

Our implementation of forward-mode operator overloaded AD has several features which we believe to be novel and unique.

— The derivatives are linear operators, not matrices or vectors. The resolutions needed for the derivatives will be determined automatically by the Chebfun system.
— We use anonymous functions and number each chebfun to enable a recursive, delayed evaluation of derivatives with forward mode AD. In standard forward mode AD, on the other hand, the derivatives are computed at the same time as the functions themselves are evaluated.
— There is no need to declare variables to be "AD variables" (known as *active variables* in the AD literature), or to initialize derivative fields explicitly before calculations are performed. This is in contrast to standard AD implementations, where the user has to decide at the start of calculations what variables he or she wants to find derivatives with respect to (the "base variables"), and initialize derivative values accordingly.
— We can obtain derivatives of any chebfun with respect to any other chebfun.

Delayed evaluation is particularly helpful to avoiding computational overhead for Chebfun tasks that do not require AD. This idea is similar to the one used in the implementation of linops for linear Chebfun operators. As is described in Driscoll et al. [2008], one way to achieve this delayed evaluation in MATLAB is to use anonymous functions.

As an example, take the overloaded `sin` method. The essential lines are the following.

```
function fout = sin(fin)
% Compose the input chebfun with the sin function
fout = comp(fin, @(x) sin(x));
% Assign a Jacobian (anon. func.) and ID (integers) to the chebfun returned
fout.jac = @(u) diag(cos(fin))*diff(fin,u);
fout.ID = newIDnum;
```

The call to the `comp` method in the second line is Chebfun's facility to compose the input chebfun with the sin function. In line three, we assign to the `jac` field of the chebfun returned (`fout`) a MATLAB anonymous function handle via

```
fout.jac = @(u) diag(cos(fin))*diff(fin,u);
```

Here, the argument `u` has the meaning of the independent variable we will be differentiating with respect to. The use of an anonymous function allows delayed evaluation; the value of `fin.jac` does not need to be known at construction time (in the sense that we only need to know the anonymous function in the `jac` field of `fin`, not the actual derivative). Without the anonymous function wrapper, we would have a typical forward-mode implementation in which the `jac` field of some chebfun would need to be initialized as the identity operator to signal it as a basis variable, before other chebfuns could be derived from it. All subsequent chebfun constructions would then use computational time to keep Jacobians up to date. The `diag` method here, which corresponds to creating a multiplication operator, will be further explained in Section 3.1.

[4]The term *Jacobian* technically only describes finite-dimensional operators, not our continuous setting, but we prefer its familiarity and the close connection with discretization that it suggests.

The process of evaluating the Jacobian field to obtain a linop is also described in that same section.

In the fourth line of the code given previously, we assign to the private field `ID` the output from the private method `newIDnum`. The function `newIDnum` returns to every chebfun created a unique ID which consists of two integers. One of the integers is a timestamp initialized the first time the `newIDnum` method is called in a MATLAB session, and the other is a counter which starts at 1 in each session and is incremented each time `newIDnum` is called.[5] We emphasize that each intermediate chebfun created when a function evaluation takes place gets a unique ID. This will be important when we eventually evaluate derivatives.

While the straightforward use of anonymous functions here is effective, it leads to a serious inefficiency for certain client codes. When operations are applied to a chebfun repeatedly, the anonymous functions effectively create a stack of those operations, in particular the variable workspaces in context at the creation of the `jac` fields. Unfortunately, MATLAB internally creates new copies of the entire stack of workspaces upon each anonymous function creation, leading to exponential growth in memory use. Thus, we found it necessary to create a new service class named *anon* to store these workspaces more efficiently while offering similar functionality to anonymous functions.

An anon object has the following four fields.

— `function`. A string that defines the function the anon represents.
— `variablesName`. A cell array of strings with the names of the variables in the `function` string.
— `workspace`. A cell array of variables (doubles and chebfuns) with the values of these variables.
— `depth`. A double used for memory control, further discussed in Section 3.3.

(The reader who has studied the `functions` method of normal anonymous functions in MATLAB will spot the similarity.)

By overloading the `feval` and `subsref` methods for the anon class, one can work with anons in a similar way as one would with anonymous functions. Using the new anon class, the method `sin.m` now appears as shown here.

```
function fout = sin(fin)
% Compose the input chebfun with the sin function
fout = comp(fin, @(x) sin(x));
% Assign a Jacobian (anon) and ID (integers) to the chebfun returned
fout.jac = anon('@(u) diag(cos(fin))*diff(fin,u)',{'fin'},{fin});
fout.ID = newIDnum;
```

3.1. Evaluation of Derivatives

We now describe how recursion and ID fields are used to evaluate Jacobians. The main result is that the evaluation is performed in a recursive way, which is made possible by giving each chebfun a unique ID. To the reader already familiar with AD, our implementation might seem like "forward-mode AD with a bit of reverse feel to it".

[5]It is necessary to make the ID consist of two integers to avoid problems with saving and reloading MATLAB workspaces.

To explain the evaluation process, assume we have two functions f and g, represented by the chebfuns `f` and `g`. Furthermore, assume that g depends on f (and possibly other functions), that is, there exists an operator G,

$$G : f \mapsto G(f, \cdot),$$

such that $g = G(f, \cdot)$. Then, when we speak of the "derivative of g with respect to f" or the "the Jacobian of g with respect to f", mathematically, we mean the Fréchet derivative of the operator G at the point (in function space) f.

To obtain `g`, we need to start with `f` and perform a sequence of calculations. In that series of calculations, we will be creating temporary variables known as *intermediate variables* in the AD literature. Each intermediate variable is obtained by performing elementary operations, such as +, *, sin and exp on variables which already exist. Every intermediate variable has a `jac` field similar to the one shown for the `sin.m` method, which contains an anon that depends on variables that have been previously created.

To obtain the derivative of g with respect to f, we call `diff(g,f)`. The `diff` method starts by using ID fields to see whether `f` and `g` are the same chebfun. If so, the Jacobian is an identity linop. Otherwise, the anon object in the jac field of `g` is evaluated with `f` as its argument. This will continue recursively through all intermediate variables in the evaluation trace between `f` and `g`, until `f` is reached and the identity operator forms the bottom of the recursion.

To put this information into context with the example shown previously of the overloaded sin method, the third line of the method read

```
fout.jac = anon('@(u) diag(cos(fin))*diff(fin,u)',{'fin'},{fin});
```

As described in Driscoll et al. [2008], if `h` is a chebfun, `diag(cos(h))` is a linop that corresponds to the multiplication operator

$$M_{\cos(h(x))} : k(x) \mapsto \cos(h(x))k(x).$$

Such multiplication operators are the continuous analogues of diagonal matrices. The second part of the anon is `diff(fin,u)`. As explained previously, this corresponds to the linop which is the Jacobian of f_{in} with respect to u. However, since `diff(fin,u)` is a part of the anon, no actual function call is made at the time the `jac` field gets assigned a value. If we make the assignment

```
g = sin(f);
```

`g` corresponds to `fout` and `f` corresponds to `fin`. Then, when we call

```
dgdf = diff(g,f);   % Differentiate g with respect to f
```

`u` gets assigned the value `f`, and the result will be the linear operator

$$M_{\cos(f(x))}I,$$

where I is the identity operator on the interval f is defined on.

Comparing this anon with the standard rules of calculus, we see that it represents the chain rule. The evaluation process is further explained in the listing of Algorithm 1 and Figure 2 where we show an example of the evaluation of a Fréchet derivative.

Note that the syntax described here is an extension to the `diff` method already in the Chebfun system. If `diff` is called with one chebfun argument, the result is another chebfun which corresponds to the derivative of the polynomial representation of the input chebfun. Thus, when `diff` is called with one argument, the derivative returned is calculated with traditional numerical differentiation, and no information from the evaluation trace is used.

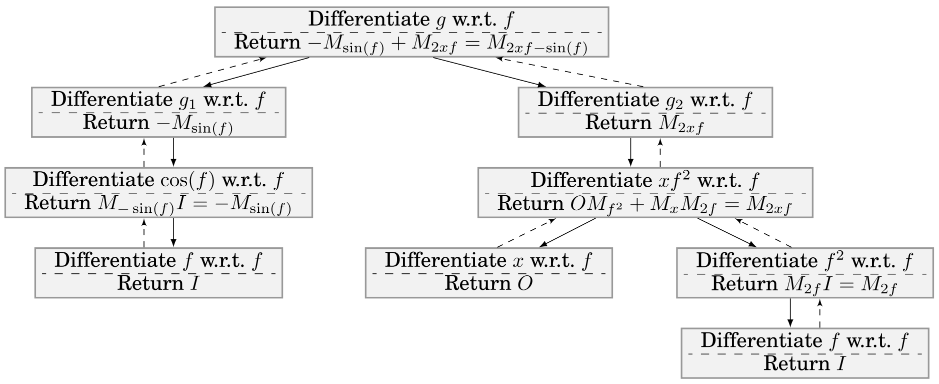

Differentiate g w.r.t. f
Return $-M_{\sin(f)} + M_{2xf} = M_{2xf-\sin(f)}$
Differentiate g_1 w.r.t. f
Return $-M_{\sin(f)}$
Differentiate g_2 w.r.t. f
Return M_{2xf}
Differentiate $\cos(f)$ w.r.t. f
Return $M_{-\sin(f)}I = -M_{\sin(f)}$
Differentiate xf^2 w.r.t. f
Return $OM_{f^2} + M_x M_{2f} = M_{2xf}$
Differentiate f w.r.t. f
Return I
Differentiate x w.r.t. f
Return O
Differentiate f^2 w.r.t. f
Return $M_{2f}I = M_{2f}$
Differentiate f w.r.t. f
Return I

Fig. 2. Recursive evaluation of the function g with respect to the function f, where $f = \sin(x)$, $g_1 = \cos(f)$, $g_2 = xf^2$ and $g = g_1 + g_2$. In the figure, I stands for the identity operator, M for a multiplication operator (such that $M_{h(x)} : k(x) \mapsto h(x)k(x)$) and O for the zero operator on the domain of the functions. Note that I, M and O are all linear operators.

Algorithm 1 Recursive evaluation of Fréchet derivatives.

```
 Input:   Chebfuns f and g.
Output:   The derivative of $g$ with respect to $f$.

Extract IDs of g and f.
if jac field of g is empty then
    return $O$, the zero operator on the domain of $f$ and $g$.
else if IDs match then
    return $I$, the identity operator on the domain of $f$ and $g$.
else
    Evaluate the derivatives of the intermediate variables g is composed of
    with respect to f (recursively), and combine the resulting derivatives
    according to the chain rule.
end if
```

3.2. Examples of Fréchet Derivatives in the Chebfun System

We begin some examples of automatic Fréchet differentiation by defining the linear function x on the interval of interest. Other functions on the same interval can then be defined as follows.

```
>> % Create a linear chebfun on [-1,1]
>> x = chebfun('x',[-1,1]);
>> % Create further chebfuns by composing chebfuns already created
>> f = x.^2;
>> g = x+f.^2;
>> h = sin(f) + diff(g);
```

This corresponds to defining the functions f, g and h on the interval $[-1, 1]$ by

$$\begin{aligned} f &= && x^2 \\ g &= && x + f^2 && = x + x^4 \\ h &= && \sin(f) + g' && = \sin(x^2) + 1 + 4x^3, \end{aligned}$$

where $'$ denotes the derivative of a function with respect to the independent function x. Throughout the rest of this section, we use total derivatives to denote Jacobian operators.

Suppose for example that we perturb the function f by an infinitesimal function δf. What effect will this have on g? The answer is that the first order variation will take the form

$$\frac{\mathrm{d}g}{\mathrm{d}f} = \frac{\partial g}{\partial f} + \frac{\partial g}{\partial x}\cancelto{0}{\frac{\mathrm{d}x}{\mathrm{d}f}} = \frac{\partial}{\partial f}\left(x + f^2\right) = 0 + 2f = 2x^2.$$

In other words, $\frac{\mathrm{d}g}{\mathrm{d}f}$ is the linear operator that multiplies a function on $[-1, 1]$ by $2x^2$, that is,

$$\frac{\mathrm{d}g}{\mathrm{d}f} : k(x) \mapsto 2x^2 k(x).$$

With chebfuns, we obtain this operator by calling `diff` with two arguments — `diff(g,f)`. In order to see the same representations that are used internally, here we request the output in the form of a linop.

```
>> % Differentiate the chebfun g with respect to the chebfun f
>> dgdf = diff(g,f,'linop')
dgdf = linop
   operating on chebfuns defined on:
     interval [-1,1]
   with n=6 realization:
     2.0000    0.0000    0.0000    0.0000    0.0000    0.0000
     0.0000    1.3090    0.0000    0.0000    0.0000    0.0000
     0.0000    0.0000    0.1910    0.0000    0.0000    0.0000
     0.0000    0.0000    0.0000    0.1910    0.0000    0.0000
     0.0000    0.0000    0.0000    0.0000    1.3090    0.0000
     0.0000    0.0000    0.0000    0.0000    0.0000    2.0000
   with functional representation:
     @(u) innersum(u,i,j)
```

In the case where Jacobian operators are diagonal, a function is defined by the information on the diagonal. We can extract this function from the diagonal by letting it operate on the function **1** on the interval. This is done in the overloaded function `diag`.

```
>> diag_dgdf = diag(dgdf); % Extract the diagonal of the linear operator
```

We verify that the AD derivative matches the analytical exactly.

```
>> % Measure the difference between the two functions in the 2-norm
>> norm(diag_dgdf - 2*x.^2)
ans =
     0
```

However, not all Jacobian operators are diagonal. For example, suppose we want to obtain $\frac{\mathrm{d}h}{\mathrm{d}g}$, the derivative of h with respect to g. Since

$$h = \sin(f) + g',$$

we have

$$h + \delta h = \sin(f) + (g + \delta g)',$$

so $\frac{\mathrm{d}h}{\mathrm{d}g}$ must be the differentiation operator on the interval with respect to the variable x:

```
>> dhdg = diff(h,g,'linop') % Differentiate h w.r.t. g
dhdg = linop
   operating on chebfuns defined on:
     interval [-1,1]
   with n=6 realization:
     -8.5000   10.4721   -2.8944    1.5279   -1.1056    0.5000
     -2.6180    1.1708    2.0000   -0.8944    0.6180   -0.2764
      0.7236   -2.0000    0.1708    1.6180   -0.8944    0.3820
     -0.3820    0.8944   -1.6180   -0.1708    2.0000   -0.7236
      0.2764   -0.6180    0.8944   -2.0000   -1.1708    2.6180
     -0.5000    1.1056   -1.5279    2.8944  -10.4721    8.5000
   with functional representation:
     @(u) innersum(u,i,j)
   and differential order 1
```

We confirm that the AD result is the correct operator by letting $\frac{\mathrm{d}h}{\mathrm{d}g}$ operate on the function x on the interval and measure the norm between the resulting chebfun and the analytical derivative.

```
>> % Measure the difference between the computed and analytical answer
>> norm(dhdg*x-1)   % Default call to norm uses the 2-norm
ans =
     0
```

We can also obtain the derivative of h with respect to f. We now have that

$$\begin{aligned}\frac{\mathrm{d}h}{\mathrm{d}f} &= \frac{\partial h}{\partial f} + \frac{\partial h}{\partial g}\frac{\mathrm{d}g}{\mathrm{d}f} + \frac{\partial h}{\partial x}\cancelto{0}{\frac{\mathrm{d}x}{\mathrm{d}f}} \\ &= \cos(f) + \mathcal{D}\cdot 2f = \cos(f) + \mathcal{D}\cdot 2x^2,\end{aligned}$$

where $\mathcal{D}$ is the differentiation operator on the interval $[-1, 1]$. If we now let $\frac{\mathrm{d}h}{\mathrm{d}f}$ operate on the function x, the result will be $\cos(f)x + (2x^2x)' = \cos(f)x + 6x^2$:

```
>> dhdf = diff(h,f);
>> norm(dhdf*x - (cos(f).*x+6*x.^2))
ans =
     0
```

Again, the answer obtained using AD is exact.

3.3. Controlling Memory Usage

Whereas the anon class described here removes certain inefficiencies of implementing AD for Chebfun, another issue arises for certain client codes, especially ones based on iterative processes (such as GMRES [Saad and Schultz 1986]). As the number of iterations grows, the calculation tree (whose root in this notation is the chebfun x) becomes huge, and the risk of running out of memory becomes high. In fact, memory control is a very important issue for AD in general [Griewank and Walther 2008].

In order to manage the use of memory, we monitor the `depth` property of the anon object. The depth of an anon indicates how far from the initial chebfun variable another chebfun is (i.e., how high in the calculation tree it is). By definition, the depth of the linear chebfun variable `x` is 0, and the depth of other variables is given by the maximum of the depth of the variables it is created from plus 1. Once the depth exceeds a maximum value (the default value is 25), a dummy anon with an empty workspace is returned and future AD calculations with the resulting variables are disabled.[6]

4. AUTOMATIC ITERATIONS FOR BOUNDARY-VALUE PROBLEMS

4.1. Damped Newton Iteration

As described in Section 2, we wish to solve a nonlinear boundary-value problem (BVP)

$$\begin{aligned} \phi(u) &= 0, \\ \alpha(u)\big|_{x=a} = 0, &\qquad \beta(u)\big|_{x=b} = 0, \end{aligned} \tag{12}$$

which we sometimes write as

$$\mathcal{N}(u) = 0,$$

where $\mathcal{N}$ is a nonlinear BVP operator. If we have a guess of the solution, u_k, we can use a Newton iteration to solve the problem by calculating the update v_k, where v_k satisfies the linearized boundary value problem

$$\mathcal{N}'(u_k)v_k = -\mathcal{N}(u_k).$$

We use automatic differentiation to obtain $\mathcal{N}'$, the derivative of the operator $\mathcal{N}$, which incorporates linearized forms of the boundary conditions as well as the differential equation. Formally, we solve for the update:

$$v_k = -(\mathcal{N}'(u_k))^{-1}\mathcal{N}(u_k).$$

In the software, this equation implies the automatic solution of a linear BVP using the backslash operator of the linop class.

Once the update v_k has been calculated, we update our current guess of the solution to obtain the next guess of the solution, u_{k+1}, via

$$u_{k+1} = u_k + v_k. \tag{13}$$

However, this update formula is only guaranteed to converge to a solution for a sufficiently close initial guess, and in practice, the solution process could take too many iterations, stagnate, diverge or even fail if $\mathcal{N}'$ is singular. In order to increase the chance of an initial guess converging to a solution, Equation (13) is often modified to include a damping parameter λ_k, so that the solution is updated via

$$u_{k+1} = u_k + \lambda_k v_k. \tag{14}$$

[6]The maximum AD depth can be set using `cheboppref('adddepth',newvalue)`. This allows users to switch AD off completely by setting the new value to be 0.

An obvious question now is how to choose the damping parameter (also known as the *step size*). Ideally, we would want to find a λ_k that minimizes $\|\mathcal{N}(u_{k+1})\|$, that is, for a given guess u_k and an update v_k, we want to minimize the residual of our next guess in some norm $\|\cdot\|$. The problem of finding the value for λ_k thus becomes an optimization problem, which in turn depends on the choice of an objective related to the residual $\mathcal{N}(u)$. Similarly to the *natural criterion function* described in Ascher et al. [1995], we define our natural criterion function as

$$\gamma(w_k^\lambda) = \frac{1}{2}\|w_k^\lambda\|_{\mathrm{F}}, \tag{15}$$

where w_k^λ is a solution to the linear BVP

$$\mathcal{N}'(u_k)w = -\mathcal{N}(u_k + \lambda v_k),$$

and $\|\cdot\|_{\mathrm{F}}$ is the Frobenius norm (or since we are working in the continuous context, the Hilbert-Schmidt norm). For the rest of this article, we let $\|\cdot\|$ denote the Frobenius norm. Using an algorithm whose details we give in the following, we find the optimal value for λ in each iteration.

The reason why we use (15) as our objective function rather than simply

$$\|\mathcal{N}(u_k + \lambda v_k)\| \tag{16}$$

is summarized in Ascher et al. [1995, p. 333]. The simple objective function given by (16) is sensitive to scaling of the variables as well as rescaling of the equations in BVPs. Furthermore, it runs into trouble when the Jacobians are ill-conditioned, as it loses sensitivity to improvements with respect to the BVPs. The objective function (15) is known as an *affine invariant objective function*. The importance of working in an affine invariant framework is further described in [Deuflhard 2006].

In practice, we have found that even though the value we find for λ is the optimal one in a given iteration, sometimes we benefit more in the long run by giving the solution process a "kick", by which we mean that if our linesearch algorithm suggests taking the smallest allowed step size, we try instead to take the full Newton step. In our code, we take the full Newton step if the line search has predicted the minimum allowed value of λ for three iterations in a row. This approach is generally not taken in the literature, where all algorithms we have found return a "no convergence flag" if the recommended step size is smaller than the minimum allowed step size. However, our experience suggests that reverting to the Newton step as a last resort rather than abandoning the iteration can lead to successful convergence in some difficult problems.

We use three criteria to check whether the solution process has converged, and if any criterion is satisfied, we stop the Newton iteration. These criteria are as follows.

— The norm of the latest update is less than a specified tolerance ρ_v (default value 10^{-10}).

— The norm of the residual is less than a specified tolerance ρ_{Res} (default value 10^{-10}). We define the norm of the residual at the k-th iteration as

$$R_k = \sqrt{\|\phi(u_k)\|^2 + \|\alpha(u)\big|_{x=a}\|^2 + \|\beta(u)\big|_{x=b}\|^2}.$$

— The number of iterations is greater than a specified maximum.

Note that since one of the design aims of Chebfun is to be scale-invariant, all tolerances involved are relative tolerances with respect to the norm of the current guess of the solution.

Algorithm 2 Newton iteration for solving boundary value problems.

```
Input:  A boundary value problem of the form $\mathcal{N}(u) = 0$, equivalent to $\phi(u) = 0$, $\alpha(u)\big|_{x=a} = 0$, $\beta(u)\big|_{x=b} = 0$. An initial guess of the solution, $u_0$, convergence tolerances $\rho_v$ and $\rho_{\text{Res}}$ and an iteration limit *maxiter*.
Output: A solution $u$ of the boundary value problem or a flag stating no convergence.
Initialize: Set $\eta_v \leftarrow \infty$, $\eta_{\text{Res}} \leftarrow \infty$.
while $\eta_v > \rho_v$, $\eta_{\text{Res}} > \rho_{\text{Res}}$ and $k < maxiter$ do
    Linearize the operator $\mathcal{N}$ around the latest guess of the solution $u_k$ to
    obtain $\mathcal{N}'(u_k)$.

    Find a solution $v_k$ to the linear BVP
        $$\mathcal{N}'(u_k)v = -\mathcal{N}(u_k).$$
    if $k \geq 1$ then
        Calculate the contraction factor
            $$\Theta_{k-1} \leftarrow \frac{\|v_k\|}{\|v_{k-1}\|}.$$
    end if
    if $k = 0$ or ($k \geq 1$ and $\Theta_{k-1} \leq 1$) then
        Set $\lambda_k \leftarrow 1$ to take the full Newton step.
    else
        Calculate the Newton step size $\lambda_k$ (as described in Appendix A).
    end if
    Update the solution:
        $$u_{k+1} \leftarrow u_k + \lambda_k v_k.$$
    Evaluate quantities used for convergence checks:
        $$\eta_v \leftarrow \frac{\|v_k\|^2}{\|u_{k+1}\|^2},$$
        $$\eta_{\text{Res}} \leftarrow \frac{\|\phi(u_{k+1})\|^2 + \|\alpha(u)\big|_{x=a}\|^2 + \|\beta(u)\big|_{x=b}\|^2}{\|u_{k+1}\|^2}.$$
end while
if $k = maxiter$ then
    return a flag stating no convergence.
else
    return the solution $u_{k+1}$.
end if
```

The details of the damped Newton iteration are summarized in the listing of Algorithm 2. The Newton algorithm we implemented is inspired by the "Error matching algorithm" found in Deuflhard [2006, p. 365] and the "Damped Newton method" algorithm described in Ascher et al. [1995, p. 335]. Unlike the former algorithm, the mesh is chosen globally and automatically by the Chebfun internals, so we do not have to refine the mesh explicitly when we solve BVPs. The step size search we implemented is taken from the latter algorithm, and is the same as the one used in MATLAB's BVP routine `bvp4c`. The step size search is performed using weak line search; we give the algorithm in Appendix A.

4.2. The Chebop Class

As stated in the introduction, the computational specification of the BVP (12) requires the following components.

Component	Mathematical symbol	chebop field
Interval on which problem is defined	$[a, b]$	`dom`
Differential equation operator	ϕ	`op`
Boundary condition operators	α and β	`lbc` and `rbc`
Initial guess of the solution	u_0	`init`

We have created a new class called *chebop* in order to store and work with all this information. Note that when solving BVPs, the user will pass information about the right-hand side of the differential equation to the solver directly (i.e., not using the chebop class), while the right-hand sides of the boundary conditions are always assumed to be 0 (so α and β must be on the form such that $\alpha(u)\big|_{x=a} = 0$ and $\beta(u)\big|_{x=b} = 0$ where u is a solution of the BVP). The fields of a chebop can be set either in the chebop constructor when it is created, or later by accessing them directly.

The user may supply the initial solution guess u_0, but if none is given, the chebop constructor will start with the lowest-degree polynomial for each solution variable that interpolates all numerical boundary values given. When the boundary operators are not of Dirichlet type, that is, $u(a) = A$, $u(b) = B$, where A and B are constants, the initial guess will usually not satisfy the boundary conditions. If the boundary conditions are linear, then one damped Newton iteration will correct them; otherwise, convergence to satisfaction of the conditions is not assured.

4.3. Examples

To illustrate how we work with chebops, assume that we want to find a solution to this nonlinear boundary value problem due to Carrier [Bender and Orszag 1978]:

$$\epsilon u'' + 2(1 - x^2)u + u^2 = 1, \quad u(-1) = 0, \quad u(1) = 0, \tag{17}$$

with $\epsilon = 0.01$. We can create a chebop by calling the chebop constructor with arguments corresponding to the endpoints of the interval of interest as follows.

```
>> N = chebop(-1,1);     % Create a chebop on [-1,1]
```

We use anonymous functions to define the differential equation of the operator.

```
>> % Assign the DE of the chebop
>> N.op = @(x,u) 0.01*diff(u,2) + 2*(1-x.^2).*u + u.^2;
```

Boundary conditions can be set up in a number of ways, as explained in the online guide [Trefethen et al. 2011]. In the case of homogeneous Dirichlet or Neumann boundary conditions, one can simply use a string as follows.

```
>> % Assign both LBC and RBC using the keyword 'dirichlet'
>> N.lbc = 'dirichlet'; N.rbc = 'dirichlet';
```

If we want to use the initial guess

$$u_0 = 2(x^2 - 1)\left(1 - \frac{2}{1 + 20x^2}\right), \tag{18}$$

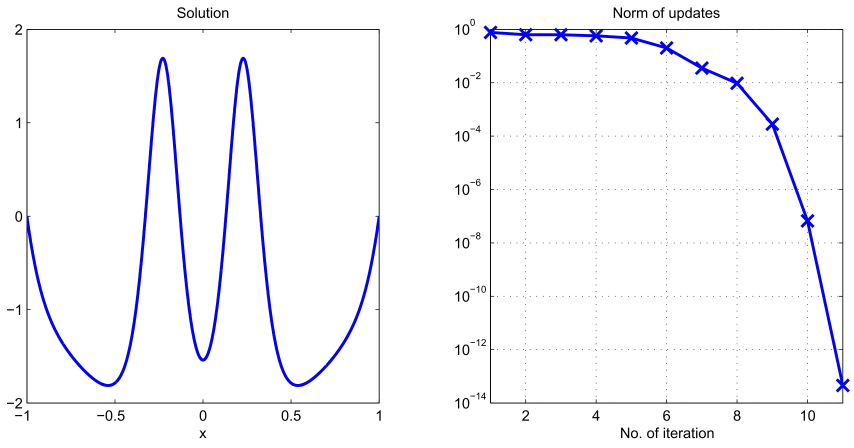


Fig. 3. A solution of the Carrier BVP (17) and the norm of the updates.

we can assign it to the `init` field of the chebfun object.

```
>> x = chebfun('x')  % Create the function "x" on [-1,1]
>> % Assign initial guess of the solution to the chebop
>> N.init = 2*(x.^2-1).*(1-2./(1+20*x.^2));
```

To solve the BVP (17) using a damped Newton iteration, we now only need to execute the overloaded backslash (\) operator on the chebop `N`. The right-hand side of the backslash will be the right-hand side of the differential equation of the problem, in this case, 1. If we call the backslash with two output arguments, we also get a vector with the norms of the corrections at each iteration.

```
>> [u, delta_norms] = N\1; % Solve the nonlinear BVP
```

In Figure 3, we plot the solution obtained as well as the norms of the updates. In the right part of the figure, we see that the convergence starts off slowly, but achieves its quadratic behaviour once we get nearer to the solution. Note that if we had started with a different initial guess, we could have converged to another solution. The following computation shows that the solution is obtained to a high accuracy.

```
>> norm(N(u)-1) % The 2-norm of the residual of the BVP
ans =
    1.042484141631300e-011
```

This code runs in 1.9 seconds. While there certainly exists software that is able to solve this problem faster (for example MATLAB's `bvp4c` and `bvp5c`), our approach offers two important benefits compared to other software.

— A very accurate solution can be produced.
— An extremely simple problem setup, requiring none of the usual problem manipulation (such as casting as a first-order system) and separate file programming common to other numerical software.

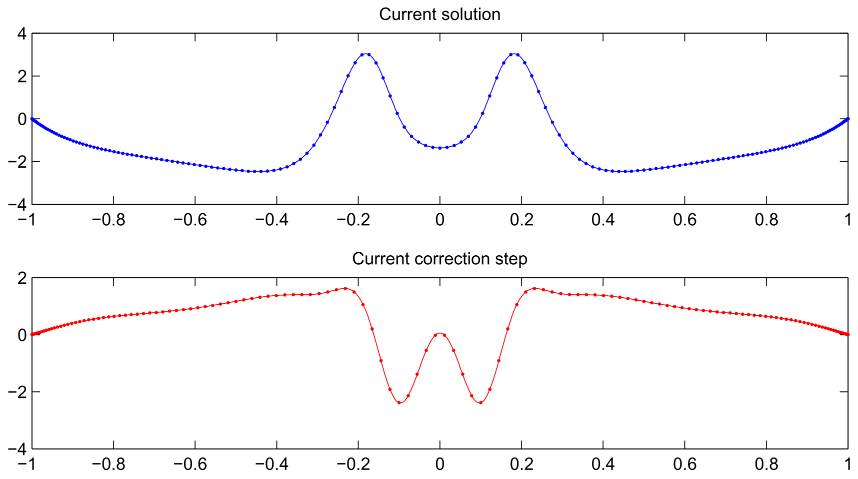


Fig. 4. A plot shown after the second iteration of solving the BVP (17). The top plot shows the current guess of the solution and the bottom plot shows the latest Newton update. The filled circles on the lines are located at the Chebyshev interpolation points (for the respective degrees of polynomials of each function).

For research or instructional purposes, we have the option of solving the problem using a pure Newton iteration (i.e., no damping) by executing the command

```
>> cheboppref('damped','off')   % Turn damped Newton iteration off
```

before the backslash command. For this example, the solver will still converge when using pure Newton iteration. However, we should only expect that to happen when we have a good initial guess.

The option of the pure Newton iteration enables our system to offer the unique possibility of visualizing it in the functional setting. By executing the command

```
>> cheboppref('plotting','on')  % Turn plotting for each iteration on
```

before calling the backslash operator, our solver plots in each iteration the current guess of the solution as well as the latest Newton update. In Figure 4 we show such a plot, taken after the second iteration when solving the Carrier problem (using a pure Newton iteration and starting from the initial guess (18)).

If we want to change the BVP (17) so that we impose different boundary conditions, we can reuse the chebop created before and change only the relevant fields. If we now want to solve

$$\epsilon u'' + 2(1 - x^2)u + u^2 = 1, \quad u(-1) = 1, \quad u'(1) + u(1) = 0, \tag{19}$$

with $\epsilon = 0.01$ as before, we execute the following commands.

```
>> N.lbc = 1;               % Change the LBC so that u(-1) = 1
>> N.rbc = @(u) diff(u)+u; % Change the RBC so that u'(1)+u(1) = 0
>> [u delta_norms] = N\1;  % Find a solution to the BVP
```

(When boundary conditions are assigned in the way `N.lbc` is here, the system automatically creates anonymous functions to represent the boundary condition operators.) Again, we obtain the solution to a high accuracy.

```
>> norm(N(u)-1) % The 2-norm of the residual of the BVP
ans =
    3.056740713605460e-011
```

We now show how to solve the BVP from Section 2, which we state again for convenience:

$$u'' + 2u\sin u = 0, \quad u'(0) = 0, \quad u(5)u'(5) = 2. \tag{20}$$

As in Section 2, we use a pure Newton iteration, so we begin by executing the following statement.

```
>> cheboppref('damped','off') % Turn damped Newton iteration off
```

The problem is then solved as follows.

```
>> N = chebop(0,5);                    % Create a chebop on [0,5]
>> N.op = @(u) diff(u,2)+2*u.*sin(u); % Assign the DE part
>> N.lbc = 'neumann';                  % Homogeneous Neumann LBC
>> N.rbc = @(u) diff(u).*u-2;          % Set RBC, u'(5)*u(5) = 2
>> N.init = chebfun('x',[0 5]);        % Initial guess of the solution
>> [u normv] = N\0;                    % Solve using overloaded \
```

This code runs in 1.8 seconds, whereas if we perform the Newton iteration by using advance knowledge of the Fréchet derivative of $\mathcal{N}$ to set up and solve linear boundary value problems directly using Chebfun's linop objects for linear operators, the problem is solved in 1.1 seconds. The main difference originates from the fact that now we are calculating the derivatives automatically, whereas previously the user supplied them explicitly. Our estimate shows that for this example, around 20% of the total solution time is spent in AD calculations. Furthermore, the code underlying the chebop solution is more general and robust than the simple Newton iteration shown in Section 2. We believe that this moderate penalty is a reasonable price to pay for the increased convenience in solving BVPs.

The problem discussed in the introduction section (which is identical to the BVP (20) except for the right boundary-condition $u(5) = 1$) has multiple solutions. To obtain a second solution, we can start by solving the initial-value problem

$$u'' + 2u\sin u = 0, \quad u(0) = 3, \quad u'(0) = 0, \tag{21}$$

and use the solution we obtain as an initial guess for the original BVP. The code for these computations is shown in the following; in Figure 5 we plot the two solutions. Many more examples are available online, see Trefethen et al. [2011].

```
>> N = chebop(0,5);                    % Create a chebop on [0,5]
>> N.op = @(u) diff(u,2)+2*u.*sin(u); % Assign the DE part
>> N.lbc = @(u) [u-3,diff(u)];         % Two initial conditions
>> ivpSol = N\0;                       % Solve the IVP using overloaded \
>> N.lbc = 'neumann'; N.rbc = 1;       % Assign original BCs to N
>> N.init = ivpSol;                    % Assign IVP soln. as initial guess
>> bvpSol = N\0;                       % Solve the original BVP
```

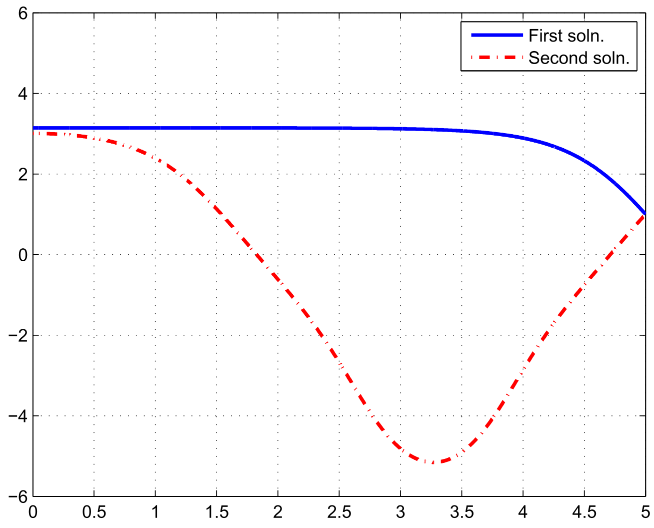


Fig. 5. Multiple solutions of the BVP $u'' + 2u\sin u = 0,\ u'(0) = 0,\ u(5) = 1.$

4.4. Comparison with Other BVP Software

For comparison with other BVP software in MATLAB, we consider the six following nonlinear BVPs:

$$
\begin{aligned}
u'' + 3.5e^{u} &= 0, & u(0) &= u(1) = 0\\
xu'' - \sqrt{\frac{u'}{x}} &= u', & u(1) &= 1,\ u(2) = 2\\
u'' - \cos(x)u' + u\log(u) &= 0, & u(0) &= 1,\ u(1) = e\\
u'' - u' + e^{2x}u + u^2 &= \sin\left(e^{x}\right)^2, & u(0) &= \sin(1),\ u\left(\tfrac{5}{2}\right) = \sin\left(e^{\frac{5}{2}}\right)\\
u'' + 18(u - u^3) &= 0, & -u(-1) &= u(1) = \tanh(3)\\
u''u' - 48xu&, & u(0) &= 0,\ u(1) = 1.
\end{aligned}
$$

They can be shown to have the exact solutions

$$
\begin{aligned}
&u(x) = -2\ln\left[\frac{\cosh\left(\left(x-\frac{1}{2}\right)\frac{\theta}{2}\right)}{\cosh\left(\frac{\theta}{4}\right)}\right], \text{ where } \theta \text{ is a sol. of } \theta = \sqrt{7}\cosh\left(\frac{\theta}{4}\right)\\
&u(x) = \frac{A^2}{8}(x^2-1) - \frac{A}{2}(x-1) + \frac{1}{4}\log(x) + 1,\ \text{where } A = \frac{2\left(1+\sqrt{7-\frac{3}{2}\log(2)}\right)}{3}\\
&u(x) = e^{\sin(x)}\\
&u(x) = \sin(e^{x})\\
&u(x) = \tanh(3x)\\
&u(x) = x^4,
\end{aligned}
$$

respectively, which allows us to compare the true error, rather than residuals. The first problem is known as the Bratu problem [Ascher et al. 1995], and is often written on

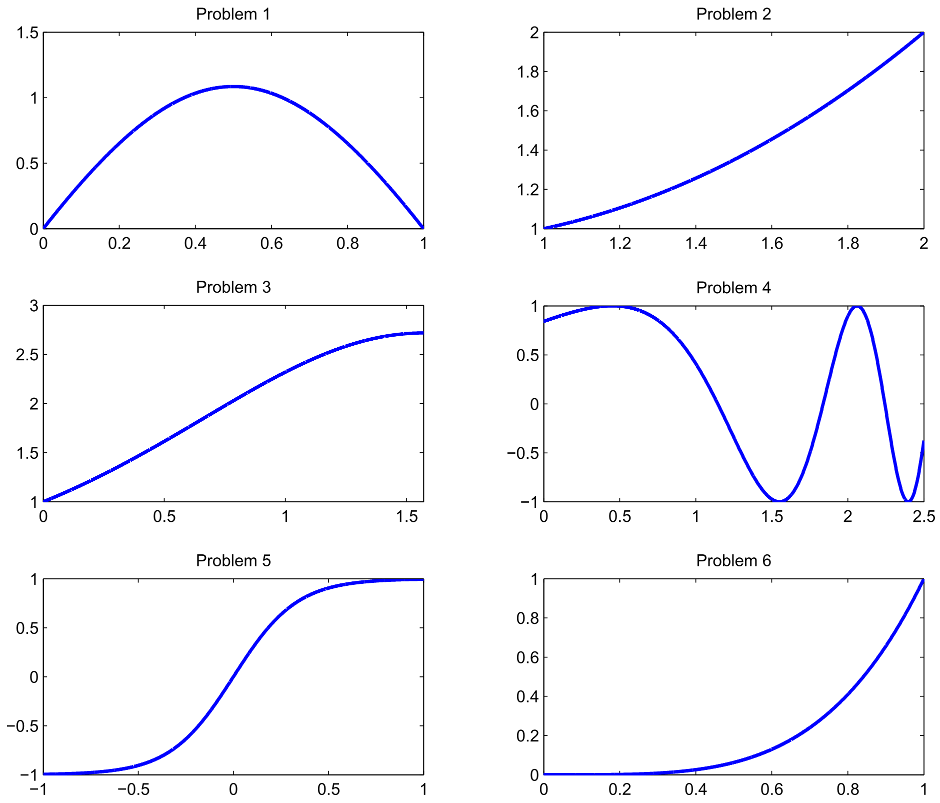


Fig. 6. Plot of the solutions of the six nonlinear BVPs we consider for comparing solvers.

the form $u'' + \lambda e^u = 0$. For our choice of $\lambda = 3.5$, it is known to have two solutions, that is, there are two values of θ which satisfy $\theta = \sqrt{7}\cosh\left(\frac{\theta}{4}\right)$. We find one such value of θ numerically using MAPLE with 25 digits of accuracy, it can then be used to construct the exact solution. The solution corresponding to the value of θ we found turns out to be the same one as the solution found by all the solvers tested. We constructed the other five problems to have nonlinear problems with the exact solution known, in order to be able to compare the true errors rather than the residuals. In Figure 6, we plot the solutions of these six problems.

As an example of the different syntaxes required, we solve the first problem using Chebfun with the following three lines of code.

```
% Create a chebop describing the problem on the interval [0,1]
N = chebop(@(u) diff(u,2)+3.5*exp(u),[0 1]);
N.lbc = 'dirichlet'; N.rbc = 'dirichlet';   % Assign homogeneous Dirichlet BCs
u = N\0;                                    % Solve using nonlinear \
```

With the other solvers, we need to rewrite the problem as a first-order system. Using bvp4c, for example, the problem can then be solved it with the following lines of code.

```
dydx = @(x,y) [y(2), -3.5*exp(y(1))];       % First order system
bcres = @(ya,yb) [ya(1),yb(1)];             % BCs residual
solinit = bvpinit(linspace(0,1,5)',[0 1]);  % Initial guess
sol = bvp4c(dydx,bcres,solinit);            % Obtain solution
```

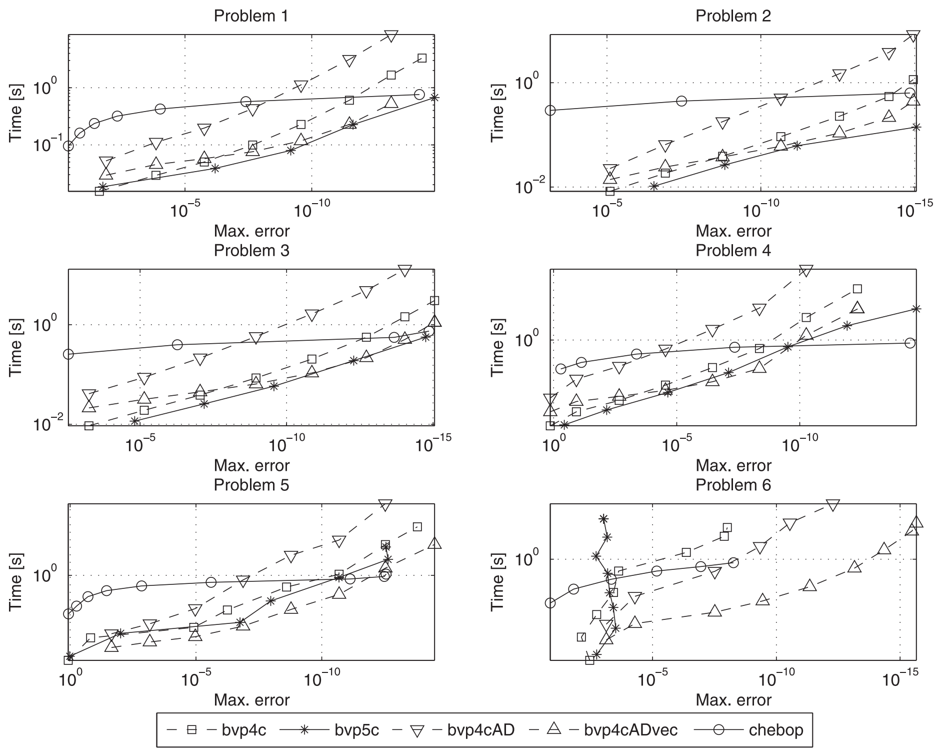


Fig. 7. Running time vs. accuracy obtained with different solvers for six nonlinear BVPs.

By changing the last line, it is possible to obtain solutions from different solvers, that is, `bvp5c` and `bvp4cAD`.

By changing the tolerances of the solvers, as well as limiting the number of Newton steps allowed, it is possible to create work-precision diagrams.[7] These show how much time it takes to get a specific accuracy, by which we mean the maximum error of the numerical solution compared to the analytical one. Such plots are shown in Figure 7. The running times are found by solving each problem 101 times at a given level of accuracy, subtracting the first running time (to avoid distortion from time taken loading the programs into the computer cache) and taking the average of the remaining ones. For all the problems, we used simple initial guesses, either constants or the guesses constructed automatically by Chebfun.

From Figure 7, it is clear that for low accuracy, Chebfun is outperformed by the other methods. However, if we require higher accuracy, Chebfun usually has clear benefits, taking equivalent or shorter times to return solutions closer to the exact ones than the other solvers. Importantly, as the problems only take a couple of seconds to solve, the human time used to set up problems, for instance, the error-prone process of writing them as first order systems, will greatly outweigh the computer time used to

[7]The limiting of Newton steps forces the solvers, both the Chebfun one and the others, to return their current guess of the solution even though the other convergence criteria of the solvers haven't been met at that point. This way, we can obtain solutions after various running times which we then use to compare with the true solution. Informally, we can think of this as giving the solvers a fixed time to solve a problem, and see how good that solution is once that time is up.

solve the problems. In that aspect, chebops offer many benefits compared with existing software.

To comment further on the results of solving the test problems, we note that generally, the regular `bvp4cAD` method is found to be slower than all the other methods for intermediate and high accuracies. However, if we rewrite the problems in vectorized form, and turn on the vectorized option for `bvp4cAD`, the performance is drastically improved. We also found that using vectorization did not speed up the `bvp4c/5c` solvers. These observations are in agreement with Forth and Shampine [2005]. As the requested accuracy is increased, the number of gridpoints `bvp4c/5c` and `bvp4cAD` use grows very quickly, in most cases up to many thousands, whereas the Chebfun solutions require much more modest numbers of gridpoints. Finally, we mention that an interesting phenomenon happens in the last problem, where `bvp5c` completely fails to converge to the solution, and `bvp4c` and chebops struggle to obtain high accuracy.

4.5. A Graphical User Interface to Chebfun

To further add to the convenient way chebops can be used to solve nonlinear BVPs, we have designed a graphical user interface (GUI) to the chebop solver, called *chebgui*. The GUI accepts a "natural syntax" for problems, removing the need for users to define anonymous functions and construct chebop objects, and instead it is possible to input problems in a form such as

$$u'' = -\sin(u)$$

(which corresponds to a differential equation describing the motion of a nonlinear pendulum). By setting up boundary conditions in a similarly intuitive way, the user can press the solve button and the system takes care of all the work required to convert the BVP to a form chebops can work with. The GUI offers many other features, including a collection of demos. We however consider the most important feature to be the capability of exporting problems from the GUI to a MATLAB script. This enables users to start with a problem in the GUI, and automatically generate code for some more serious explorations. We show a screenshot of the GUI in Figure 8.

5. LIMITATIONS AND FUTURE DIRECTIONS

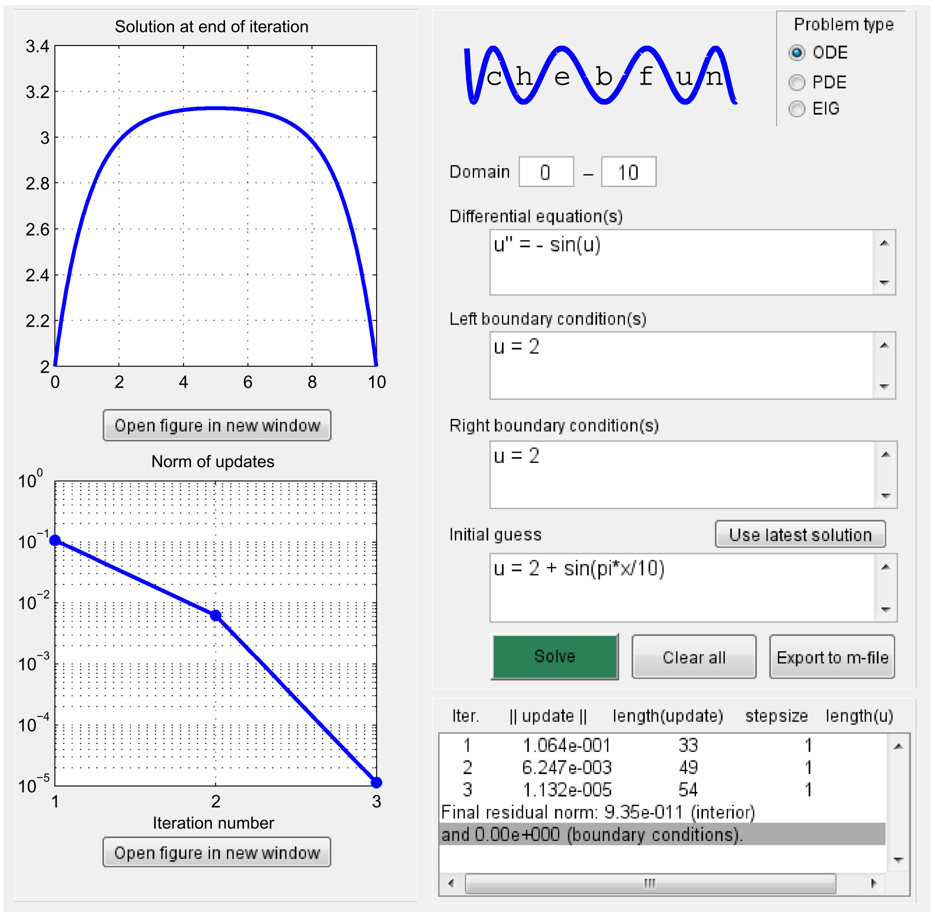
The AD implementation we have chosen to use in Chebfun creates some limitations. For example, our use of delayed evaluation to avoid AD computational overhead until a Jacobian is demanded requires the attachment to each chebfun of a memory stack (in the form of an anon object) for all the operations leading to the construction of that chebfun. It is evident from Figure 2 that our AD implementation can give rise to unnecessary calculations being performed. For example, in the overloaded `sin.m` method, the `jac` field of the chebfun returned is given by the following.

```
anon('@(u) diag(cos(fin))*diff(fin,u)',{'fin'},{fin});
```

This implies that every time we differentiate the chebfun returned, we calculate the linop corresponding to the multiplication operator

$$M_{\cos(f_{\text{in}}(x))} : k(x) \mapsto \cos(f_{\text{in}}(x))k(x),$$

even though `fin` might not depend on the function we are differentiating with respect to (in which case, `diff(fin,u)` will be the zero linop, and the derivative returned will thus be the zero linop as well). We believe that it might be possible to improve the anon class to prevent such unnecessary calculations. Given Chebfun's symbolic feel, there might even be valuable ideas available from work done on AD with computer algebra systems (CAS). One such software tool is the CODEGEN package for MAPLE discussed



Problem type
[✓ ODE] PDE EIG

Domain 0 – 10

Differential equation(s)
u'' = - sin(u)

Left boundary condition(s)
u = 2

Right boundary condition(s)
u = 2

Initial guess
u = 2 + sin(pi*x/10)

```
Iter.   || update ||   length(update)   stepsize   length(u)
  1      1.064e-001        33              1
  2      6.247e-003        49              1
  3      1.132e-005        54              1
Final residual norm: 9.35e-011 (interior)
and 0.00e+000 (boundary conditions).
```

Fig. 8. A screenshot of the Chebfun GUI.

in Monagan and Monagan [1997], where the authors discuss common subexpression optimization which is achieved by hashing of formulae. With the planned move of Chebfun to MATLAB's new object-oriented programming model, which introduces handle classes, we might be able to make improvements to address these issues.

A number of possible improvements of the current code readily suggest themselves. There is no reason in principle that the code cannot be extended to solve more general boundary-value problems, such as ones with nonseparable boundary conditions or auxiliary parameters. For example, the nonseparated condition $u(1) - u(0) = 0$ is associated with the linear functional $E_1 - E_0$, where E_x represents evaluation at the point x. This functional is its own Fréchet derivative, and there is no difficulty with representing it in Chebfun. However, automatic differentiation of a scalar result presents a problem in our system, because the result of Matlab type `double` cannot encode the information needed to construct the derivative. We are exploring the option of temporarily representing a scalar result as a constant chebfun in order to permit differentiation. The ability to differentiate functionals may also lead to interesting applications in control and optimization.

We do not regard our step size selection in the damped Newton iteration as a settled matter, and plan to improve our strategy, for example using the idea of restrictive monotonicity test discussed in Bock et al. [2000]. The idea of giving the solution process a kick by taking the full Newton step rather than the smallest allowed step size if the Newton iteration appears to be failing to converge, described in Section 4.1, seems to work well in practice. It would be interesting to see whether coming up with other heuristic might help to increase the probability of the iteration to converge. Efforts are already underway to apply the nonlinear BVP code to partial differential equations. Another effort is being undertaken to apply Fréchet AD techniques to nonlinear eigenvalue and other types of continuation problems [Allgower and Georg 2003], and to find robust ways to converge to multiple solutions without explicit initial guesses.

APPENDIX A. WEAK LINE SEARCH ALGORITHM

The following weak line search algorithm is used for the damping parameter/step size search in the damped Newton iteration. It is based on the algorithm shown in Ascher et al. [1995, p. 335] to which we refer for further discussion on the control parameters in the algorithm. $\|\cdot\|$ denotes the Frobenius norm.

Algorithm 3 Step size search at the kth iteration of Newton iteration.

```
Input:  A boundary value problem of the form $\mathcal{N}(u) = 0$, a current guess of the solution $u_k$,
        the latest Newton update $v_k$, the inverse of the Fréchet derivative of $\mathcal{N}$ at $u_k$.
Output: A damping parameter $\lambda_k$.
Define the objective function
    $\gamma(w_k^\lambda) = \frac{1}{2}\|w_k^\lambda\|$
Set control parameters; $\lambda_{\min} \leftarrow 0.1$ (minimum allowed step size), $\sigma \leftarrow 0.01$ (ensures sufficient
decrease of $\gamma$), $\tau \leftarrow 0.01$ (ensures quadratic model of $\gamma$ is valid).
Initialize: Set $\lambda \leftarrow 1$. Store $\gamma_0 \leftarrow \frac{1}{2}\|v\|$ (this holds since $w_k^0 = v_k$, see definition of $w_k^\lambda$ as
follows).
while $\lambda$ has not been accepted do
    Find a solution $w_k^\lambda$ to the linear BVP
        $\mathcal{N}'(u_k)w_k = -\mathcal{N}(u_k + \lambda v_k)$.
    Set $\gamma_\lambda \leftarrow \gamma(w_k^\lambda)$.
    [Test acceptance of $\lambda$. If value is not accepted, search for the next value
    of $\lambda$ with the weak line search.]
    if $\gamma_\lambda \le (1 - 2\lambda\sigma)\gamma_0$ then
        Accept $\lambda$, set $\lambda_k \leftarrow \lambda$.
    else
        $\lambda \leftarrow \max\left(\tau\lambda, \frac{\lambda^2\gamma_0}{(2\lambda - 1)\gamma_0 + \gamma_\lambda}\right)$
        if $\lambda < \lambda_{\min}$ then
            Accept the value $\lambda_{\min}$ for $\lambda$, set $\lambda_k \leftarrow \lambda_{\min}$.
        end if
```

```
            if $\lambda_{k-3} = \lambda_{k-2} = \lambda_{k-1} = \lambda_k = \lambda_{\min}$ then
                Set $\lambda_k \leftarrow 1$ (take a full Newton step and give the solution process
                a kick).
            end if
        end if
    end while
    return$\lambda_k$.
```

ACKNOWLEDGMENTS

We wish to thank Nick Trefethen for many comments, suggestions and revisions while working on this article. We are also grateful to Shaun Forth at Cranfield University for discussions on automatic differentiation and for giving us a working copy of MAD and `bvp4cAD`, as well as Peter Deuflhard and Martin Weiser at the Zuse Institute, Berlin, for discussions on methods for boundary-value problems. We thank the referees for their comments, which greatly improved this article. In particular, the second referee's observation that the BVP discussed in Section 4.3 has multiple solutions was useful for demonstrating some capabilities of Chebfun. Finally, we would like to acknowledge Pedro Gonnet, Oxford University, for pointing out flaws with our previous memory control approach.

REFERENCES

ALLGOWER, E. L. AND GEORG, K. 2003. *Introduction to Numerical Continuation Methods*. SIAM, Philadelphia, PA.

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

BENDER, C. M. AND ORSZAG, S. A. 1978. *Advanced Mathematical Methods for Scientists and Engineers*. McGraw-Hill, New York.

BISCHOF, C. H. AND BÜCKER, H. M. 2000. Computing derivatives of computer programs. In *Modern Methods and Algorithms of Quantum Chemistry: Proceedings* 2nd Ed. J. Grotendorst Ed., NIC Series, vol. 3, NIC-Directors, Jülich, 315–327.

BISCHOF, C. H., BÜCKER, H. M., LANG, B., RASCH, A., AND VEHRESCHILD, A. 2002. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*. IEEE, 65–72.

BOCK, H. G., KOSTINA, E., AND SCHLÖDER, J. P. 2000. On the role of natural level functions to achieve global convergence for damped Newton methods. In *Proceedings of the 19th IFIP TC7 Conference on System Modeling and Optimization*. M. J. Powell and S. Scholtes Eds., Kluwer, B.V., Deventer, The Netherlands, 51–74.

DEUFLHARD, P. 2006. *Newton Methods for Nonlinear Problems*. Springer.

DON, W. S. AND SOLOMONOFF, A. 1997. Accuracy enhancement for higher derivatives using Chebyshev collocation and a mapping technique. *SIAM J. Sci. Comp. 18*, 4, 1040–1055.

DRISCOLL, T. A., BORNEMANN, F., AND TREFETHEN, L. N. 2008. The chebop system for automatic solution of differential equations. *BIT Numer. Math. 48*, 701–723.

FORTH, S. A. 2006. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Trans. Math. Softw. 32*, 195–222.

FORTH, S. A. AND SHAMPINE, L. F. 2005. bvp4cAD: An automatic differentiation enabled boundary value solver. http://www.amorg.co.uk/AD/ADODE/bvp4cAD/index.html.

GRIEWANK, A. AND WALTHER, A. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* 2nd Ed. SIAM, Philadelphia, PA.

HUTSON, V., PYM, J. S., AND CLOUD, M. J. 2005. *Applications of Functional Analysis and Operator Theory* 2nd Ed. Elsevier.

KHARCHE, R. AND FORTH, S. 2006. Source transformation for MATLAB automatic differentiation. In *Proceedings of the 6th International Conference on Computational Science*. Part IV. V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra Eds., Lecture Notes in Computer Science, vol. 3994, 558–565.

KIERZENKA, J. AND SHAMPINE, L. F. 2001. A BVP solver based on residual control and the MATLAB PSE. *ACM Trans. Math. Softw. 27*, 299–316.

KIERZENKA, J. AND SHAMPINE, L. F. 2007. A BVP solver that controls residual and error. http://faculty.smu.edu/shampine/finalbvp5c.pdf.

MONAGAN, M. B. AND MONAGAN, G. 1997. A toolbox for program manipulation and efficient code generation with an application to a problem in computer vision. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. 257–264.

RUMP, S. 1999. INTLAB – INTerval LABoratory. In *Developments in Reliable Computing*, T. Csendes Ed., Kluwer Academic Publishers, Dordrecht, The Netherlands, 77–104.

SAAD, Y. AND SCHULTZ, M. H. 1986. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Statist. Comput. 7*, 3, 856–869.

SHAMPINE, L. F., KETZSCHER, R., AND FORTH, S. A. 2006. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Softw. 31*, 79–94.

TREFETHEN, L. N., ET AL. 2011. Chebfun Version 4.0. The Chebfun Development Team. http://www.maths.ox.ac.uk/chebfun/.

Received August 2010; revised May 2011, December 2011; accepted December 2011