Report no. 10/15

Automatic Fréchet differentiation for the numerical solution of boundary-value problems

Asgeir Birkisson
Oxford University Mathematical Institute
24–29 St Giles, Oxford, OX1 3LB, UK

Tobin A. Driscoll
Department of Mathematical Sciences, University of Delaware
505 Ewing Hall, Newark, DE 19716, USA

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 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.

Key words and phrases: Chebfun, functional Newton iteration, linearization of boundary-value problems, object-oriented MATLAB.

Oxford University Mathematical Institute
Numerical Analysis Group
24-29 St Giles'
Oxford, England OX1 3LB
E-mail: asgeir.birkisson@maths.ox.ac.uk

November, 2010

1 Introduction

In scientific computing, one typically views functions and operators as subjects for analysis. When numerical computation is to be applied, these objects become vectors and matrices through an approximation process that incurs a truncation error that is usually understood to be much larger than the rounding error.

Automatic differentiation (or algorithmic differentiation — AD in both cases), on the other hand, provides an alternative to truncated differentiation. AD starts from numerical values and produces numerical derivatives, gradients, and Jacobian matrices that are approximate only in the sense of roundoff error. While the premise of AD is simple, its efficient general implementation is far from trivial [9]. Arguably the most straightforward approach is the use of object-oriented programming methods to manipulate derivatives in *forward mode* — basically, by composing the algorithmic rules taught in elementary calculus.

A typical use of AD is to remove the need for handwritten code or difference-based methods to produce the Jacobian matrices in a Newton iteration for a nonlinear system of equations. When such a system is derived by discretization of a nonlinear boundary-value problem (BVP), however, we produce accurate numerical Jacobians for a system that is itself a truncation! More troublingly, the computational expense of AD, at least in forward mode, becomes unacceptable as the number of system variables, i.e. the discretization size, becomes large.

It is possible to describe the numerical solution of a BVP without 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* [2, 6]; we will simply use the name Newton. In this iteration, the iterates are functions updated through the solution of a succession of linear boundary-value problems. Each linear update problem is described by the *Fréchet derivatives* of the operators of the original problem, linear operators analogous to Jacobian matrices. Under suitable conditions and with a sufficiently good initial guess, 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 very different, Fréchet differentiation and differentiation of a function with respect to a real variable both rely on similar differentiation rules from calculus (e.g. the chain rule). Thus, Fréchet derivatives can in principle be produced through AD techniques. The idea is appealing because it produces a linearization for the original problem, in which the number of unknown functions is orders of magnitude less than the number of degrees of freedom in a discretization. Hence problems with large-scale inefficiency of AD may be avoided.

Of course, if AD is to produce a numerical Fréchet derivative in the form of an operator, we first require a way to express and work with such objects. The Chebfun software system, which is a free downloadable add-on for MATLAB, provides just such an environment [14]. Chebfun uses the fast convergence and fast algorithms associated with the (piecewise) Chebyshev series representation of functions to provide the illusion of symbolic facility with numerical representations. While each function is ultimately

represented by a discretization of finite length, the length is chosen automatically and dynamically so that the system can represent and manipulate most functions essentially to roundoff error with great speed. This includes the operations of rootfinding, integration, and differentiation.

Within Chebfun is an object class called *linop* (formerly called *chebop*) which allows one to create and manipulate linear operators on functions. Each linop 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 linop class has methods `\`, `eigs` and `expm` for the solution of boundary-value and eigenvalue problems and time evolution of differential, integral, and integro-differential operators using spectral collocation techniques [7]. Section 2 shows a sample of how a functional Newton iteration looks in the Chebfun system.

We have extended Chebfun to apply forward-mode AD to find Fréchet derivatives of nonlinear operators. We have used this facility to create a new *chebop* class with which the system performs a damped (or pure) Newton iteration to solve nonlinear boundary-value problems. So, for example, solving the problem

$$u'' + 2u\sin u = 0, \quad 0 < x < 5, \quad u'(0) = 0, \quad u(5) = 1, \tag{1.1}$$

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

```
phi = @(u) diff(u,2)+2*u.*sin(u);
alpha = @(u) diff(u);  beta = @(u) u-1;
N = chebop(domain(0,5),phi,alpha,beta);
u = N\0;
```

This solution takes about 4.0 seconds on a quad-core 2.33 GHz workstation[1], delivering the converged solution as a polynomial of degree 177 with a residual 2-norm less than 6×10^{-10}.

Section 3 explains how forward-mode AD techniques were 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 all uses. To keep with the highly interactive nature of Chebfun, we also desired a syntax as unobtrusive and natural as possible. These goals led us to make new contributions to AD techniques: the use of 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`. Furthermore, the independent and dependent variables are specified simultaneously; there is no need to designate `u` or `v` specially before the construction of `f` or calls to `diff`. We believe this innovation to be unique in all published AD software.

Section 4 describes the damped Newton iteration and other details behind the new chebop class and the "nonlinear backslash" capability. It also provides numerous examples illustrating various aspects of flexibility in the system. Finally, Section 5 mentions

[1] All running times cited in this paper are based on the same machine.

some of the limitations still imposed by the system and suggests avenues of future investigation and application.

Before continuing, we mention that this is not the first time AD has been used in solving BVPs. A predecessor we are aware of is [13], where the authors introduce the MATLAB method `bvp4cAD`, an overloaded version MATLAB's well known boundary-value problem solver BVP4C. Instead of using finite difference schemes to calculate Jacobians as the default version of `bvp4c` does, `bvp4cAD` uses AD to supply high accuracy partial derivatives, increasing robustness and efficiency of the solver.

2 Functional 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 $f(x^*) = 0$ [2]. Members of the sequence are defined successively via

$$x^{(k+1)} - x^{(k)} = -\Big[f'\big(x^{(k)}\big)\Big]^{-1} f\big(x^{(k)}\big), \tag{2.1}$$

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 (2.1) as $y^{(k)}$, the *Newton update*. The motivation for the update formula is the linearization

$$f(x) \approx f\big(x^{(k)}\big) + f'\big(x^{(k)}\big)\big(x - x^{(k)}\big)$$

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 (2.1) 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 on the form

$$\phi(u) = 0, \tag{2.2}$$

$$\alpha(u)\big|_{x=a} = 0, \quad \beta(u)\big|_{x=b} = 0, \tag{2.3}$$

where u is a function (i.e. $u = u(x)$) and ϕ, α, and β are operators between suitable normed function spaces; presumably, ϕ is a differential operator, and α and β may also be differential operators of lower order. When convenient, we shall refer to (2.2)–(2.3) simply as $\mathcal{N}(u) = 0$, i.e. we let $\mathcal{N}$ denote a BVP operator. We generalize the Newton iteration to produce a sequence of functions $u_1, u_2, \ldots$ such that if $u_k \to u^*$ as $k \to \infty$, then $\mathcal{N}(u^*) = 0$.

The crucial component of this 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: 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 [10]

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

Note in this definition that v is a function, and we require that the limit exists as $v \to 0$ in any manner.[2]

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

$$\phi'(u_k)v = -\phi(u_k), \tag{2.4}$$

$$\big[\alpha'(u_k)v\big]_{x=a} = -\alpha(u_k)\big|_{x=a}, \quad \big[\beta'(u_k)v\big]_{x=b} = -\beta(u_k)\big|_{x=b}, \tag{2.5}$$

or using the notation introduced above,

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

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 Kantorovich Theorem guarantees quadratic convergence for suitably close initial guesses [6].

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

$$u'' + 2u\sin u = 0, \tag{2.6}$$

$$u'(0) = 0, \tag{2.7}$$

$$u(5)u'(5) = 2, \tag{2.8}$$

which is identical to (1.1) 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

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

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

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

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 (2.5), 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

$$v''(x) + \big[2\sin(u_k(x)) + 2u_k(x)\cos(u_k(x))\big]v(x) = -\phi(u_k), \tag{2.9}$$

$$v'(0) = -u_k'(0), \tag{2.10}$$

$$u_k(5)v'(5) + u_k'(5)v(5) = -\big(u_k(5)u_k'(5) - 2\big). \tag{2.11}$$

The linearity of α implies that the iterates u_k will all satisfy the left boundary condition after the first Newton update, and (2.10) 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 (2.6)–(2.8) is shown below. The linop class described in the introduction is used to represent the linear operators involved. The iteration stops when the norm of the Newton update falls below 10^{-10}, which takes ten iterations with the initial guess $u_0(x) = x$. The code runs in 1.4 seconds. 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.

```
[d,x] = domain(0,5);                % Problem domain, and the chebfun "x"
phi = @(u) diff(u,2)+2*u.*sin(u);   % ODE part of the nonlinear BVP
D = diff(d);                        % Differentiation operator on the domain
u = x;                              % Initial guess
nrmv = 1;  y = 0;
plot3(x,chebfun(y,d),u), hold on
while nrmv > 1e-10                          % Newton iterations
  A = D^2 + diag(2*sin(u)+2*u.*cos(u));  % Frechet derivative at u
  Du = D*u;                                 % Needed to compute the BCs
```

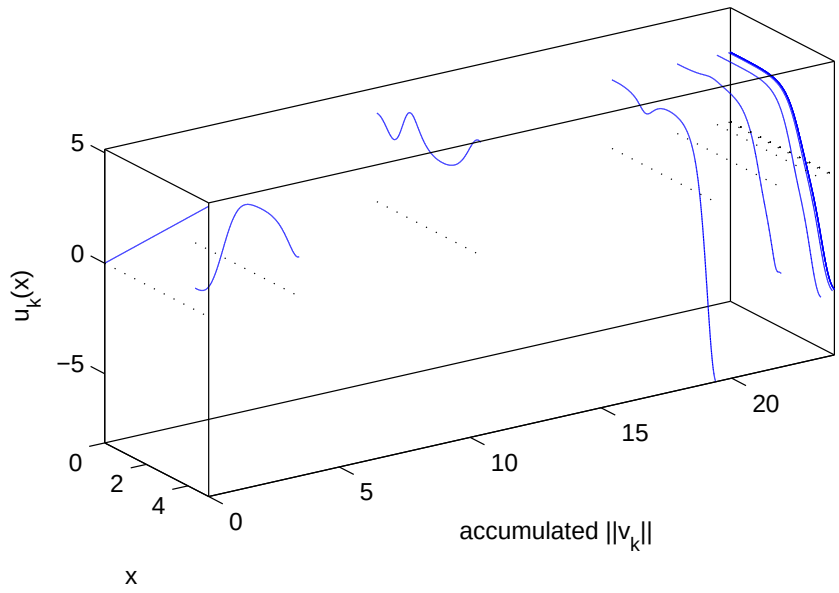

Figure 1: The solution of the BVP (2.6)–(2.8) by the Newton iteration using Fréchet derivatives in Chebfun. The y axis shows the cumulative sum of $\|v_k\|_2$ for the Newton update functions v_k. Note that each dotted line corresponds to one iteration, and as we get closer to a solution, the lines cluster (indicating that the updates are getting smaller). After some initial large updates, the iteration eventually converges quadratically to the solution.

```
  A.lbc = {D,-Du(0)};                      % Neumann condition at x=0
  A.rbc = {u(5)*D + Du(5),2-u(5)*Du(5)}; % 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,d),u)
end
```

A closer inspection of the results of running the program above reveals an apparent 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 (one more than the degree of the polynomial interpolant in Chebyshev points) used to represent the solution jumps from 2 (the linear initial guess) to 44, grows quickly to 147, stagnates, then jumps to 211 at the last iteration. This last iteration is particularly suspect: an update of size less than 10^{-10} is added to a function that is $O(1)$, yet the length of the numerical representation grows by almost 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 last iteration by 30%. By simulating operations on continuous functions without advance discretization, our approach needs to solve each linear subproblem at a discretization size tailored to the characteristics of the correction function. By contrast, a traditional numerical approach sets a fixed discretization size at the start, even when some of the correction functions could be expressed more efficiently.

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 [9]. 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, i.e. in how the chain rule is treated [4]. 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 [9], 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.

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 [4]. We note that both methods have previously been implemented for MATLAB. A well known AD package for MATLAB is MAD — Matlab Automatic Differentiation, see [8]. MAD implements the operator overloaded forward mode of AD. In [11], the authors present the MSAD package, a source transformation implementation of forward mode automatic differentiation for MATLAB based on MAD. Finally, we mention that the INTLAB (INTerval LABoratory) package also offers an operator overloaded forward mode of AD. For details of INTLAB, see [12].

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. In [9], the authors 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.[3] 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.
- There is no need to declare variables to be "AD variables", or to initalize derivative fields explicitly before calculations are performed.
- We can obtain derivatives of any chebfun with respect to any other chebfun.

As stated in the introduction, to avoid unwanted computational overhead on tasks that do not require AD, all calculations involved in adding AD capabilities to the Chebfun system are done with delayed evaluation. This idea is similar to the one used in the implementation of linops. As is described in [7], 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:

```
function fout = sin(fin)
fout = comp(fin, @(x) sin(x));
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`) the MATLAB anonymous function handle

```
@(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 meaning of the `diag` method here, and the process of evaluating the Jacobian field to obtain a linop, is described in Section 3.1.

[3]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.

In the fourth line of the code above, 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.[4] 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 above 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.[5] 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 three 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.

(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 below:

```
function fout = sin(fin)
fout = comp(fin, @(x) sin(x));
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

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

[5] In short, the MATLAB constructor of anonymous functions does not make an efficient usage of objects already created.

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".

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), i.e. 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 series 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 above, 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 [7], if `h` is a chebfun, `diag(cos(h))` is a linop that corresponds to the multiplication operator

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

The second part of the anon is `diff(fin,u)`. As explained above, 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),
```

`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 domain.

Comparing the anon above 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 syntax described above 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.

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

3.2 Examples of Fréchet derivatives in the Chebfun system

We now show examples of automatic Fréchet derivatives in the Chebfun system. We begin by defining a domain and the linear function x on that domain. Other functions on the same domain can then be defined:

```
>> [dom,x] = domain(-1,1);
>> 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

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

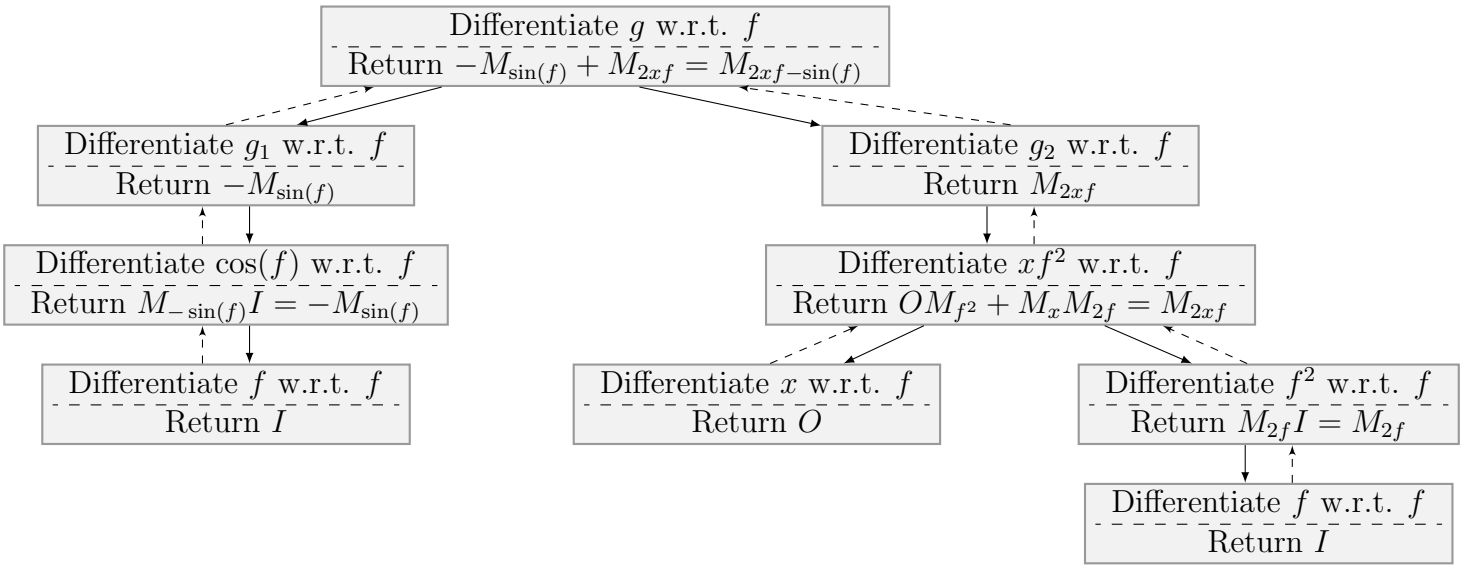

Figure 2: Recursive evaluation of the function g with respect to the function f, where $f = \sin(x)$, $g_1 = \cos(f)$, $g_2 = xf^2$ and $g = g_1 + g_2$. In the figure, I stands for the identity operator, M for a multiplication operator (such that $M_{h(x)} : k(x) \mapsto h(x)k(x)$) and O for the zero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

In general, Jacobian operators map functions on $[a, b]$ to functions on $[a, b]$. In many cases, though not always, the operators are multiplier operators (the continuous analogue of diagonal matrices). Throughout the rest of this section, we use partial 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

$$\frac{\partial g}{\partial f} = \frac{\partial}{\partial f} x + \frac{\partial}{\partial f} f^2 = 0 + 2f = 2x^2.$$

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

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

With chebfuns, we obtain this operator by calling `diff` with two arguments:

```
>> dgdf = diff(g,f)
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 domain. This is done in the overloaded function `diag`, which lets us extract the function ψ_{gf}:

```
>> psi_gf = diag(dgdf);
```

We verify that the AD derivative matches the analytical one to machine precision:

```
>> norm(psi_gf - 2*x.^2)
ans =
     0
```

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

$$h = \sin(f) + \frac{dg}{dx},$$

we have

$$h + \delta h = \sin(f) + \frac{d(g + \delta g)}{dx},$$

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

```
>> dhdg = diff(h,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 $\frac{\partial h}{\partial g}$ operate on the function x on the domain and measure the norm between the resulting chebfun and the analytical derivative:

```
>> norm(dhdg*x-1)
ans =
     0
```

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

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

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

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

Again, the answer obtained using AD is accurate to machine precision.

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)

$$\begin{aligned}\phi(u) &= 0, \\ \alpha(u)\big|_{x=a} = 0, \qquad \beta(u)\big|_{x=b} &= 0,\end{aligned} \tag{4.1}$$

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 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

$$u_{k+1} = u_k + v_k. \tag{4.2}$$

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 or diverge. In order to increase the chance of an initial guess converging to a solution, Equation (4.2) is often modified to include a damping parameter λ_k, so that the solution is updated via

$$u_{k+1} = u_k + \lambda_k v_k. \tag{4.3}$$

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 $\|\mathcal{N}(u_{k+1})\|$, i.e. for a given guess u_k and an update v_k, we want to come as close as possible to a true solution u 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 [2], we define our natural criterion function as

$$\gamma(w_k^\lambda) = \frac{1}{2}\|w_k^\lambda\|_{\mathrm{F}}, \tag{4.4}$$

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. For the rest of this paper, we let $\|\cdot\|$ denote the Frobenius norm.

The reason why we use (4.4) as our objective function rather than simply

$$\|\mathcal{N}(u_k + \lambda v_k)\| \tag{4.5}$$

is summarized in [2, P. 333]. The simple objective function given by (4.5) 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 (4.4) is known as an *affine invarant objective function*. The importance of working in an affine invariant framework is further described in [6].

To help visualizing the objective function γ, assume that one value of λ gives a smaller value of $\gamma(w_k^\lambda)$ than another value of λ. We then expect the norm of the next Newton update to be smaller if we select the first value, implying that we should be closer to a solution. Using an algorithm which details we give below, we find the optimal value for λ in each iteration.

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 a kick, 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:

- The norm of the latest update is less than a specified tolerance ρ_v.
- The norm of the residual is less than a specified tolerance ρ_{Res}. 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 the Chebfun system is to be scale-invariant, all tolerances involved are relative tolerances with respect to the norm of the current guess of the solution.

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 [6, P. 365] and "Damped Newton method" algorithm described in [2, P. 335]. Unlike the former algorithm, the mesh is chosen globally and automatically by the Chebfun internals, so we don't 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 well known 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

The computational specification of the BVP (4.1) from Section 4.1 requires the following elements:

- The domain on which the problem is defined, $[a, b]$.
- The differential equation operator, ϕ.
- The boundary condition operators, α and β.
- An initial guess of the solution, u_0.

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

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 ρ_{v} and ρ_{Res} and an iteration limit *maxiter*.

Output: A solution u of the boundary value problem or a flag stating no convergence.

while $\eta_v > \rho_{\text{v}}$, $\eta_{\text{Res}} > \rho_{\text{Res}}$ 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

$$\Theta_{k-1} = \frac{\|v_k\|}{\|v_{k-1}\|}.$$

end
if $k = 0$ **or** ($k \geq 1$ **and** $\Theta_{k-1} \leq 1$) **then**
Set $\lambda_k = 1$ to take the full Newton step.
else
Calculate the Newton step size λ_k (as described in Appendix A).
end
Update the solution:

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

Evaluate quantities used for convergence checks:

$$\eta_v = \frac{\|v_k\|^2}{\|u_{k+1}\|^2},$$

$$\eta_{\text{Res}} = \frac{\|\phi(u_{k+1})\|^2 + \|\alpha(u)\big|_{x=a}\|^2 + \|\beta(u)\big|_{x=b}\|^2}{\|u_{k+1}\|^2}.$$

end
if $k = maxiter$ **then**
return a flag stating no convergence.
else
return the solution u_{k+1}.
end

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 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 simply point evaluations (e.g., Neumann or Robin conditions), 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 [3]:

$$\epsilon u'' + 2(1 - x^2)u + u^2 = 1, \quad u(-1) = 0, \quad u(1) = 0 \tag{4.6}$$

with $\epsilon = 0.01$. We can create a chebop by calling the `domain` function with three arguments:

```
>> [d,x,N] = domain(-1,1);
```

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

```
>> N.op = @(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 [14]. In the case of homogenous Dirichlet or Neumann boundary conditions, one can simply use a string:

```
>> N.bc = 'dirichlet';
```

If we want to use the initial guess

$$u_0 = 2(x^2 - 1)\left(1 - \frac{2}{1 + 20x^2}\right), \tag{4.7}$$

we can assign it to the `guess` field of the chebfun object:

```
>> N.guess = 2*(x.^2-1).*(1-2./(1+20*x.^2));
```

To solve the BVP (4.6) 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 nonlinear backslash with two arguments, we also get a vector with the norms of the corrections at each iteration:

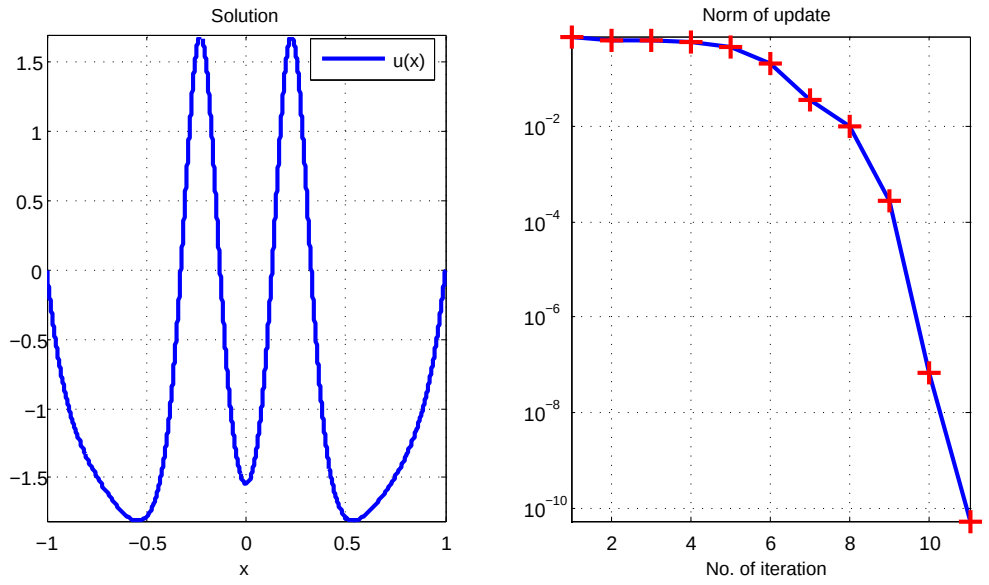

Figure 3: A solution of the Carrier BVP (4.6) and the norm of the updates.

```
>> [u, delta_norms] = N\1;
```

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 retrieves 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)
ans =
    1.042484141631300e-011
```

The code above runs in 3.3 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.

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

```
>> chebopprefʻdampedʻ,ʻoffʻ)
```

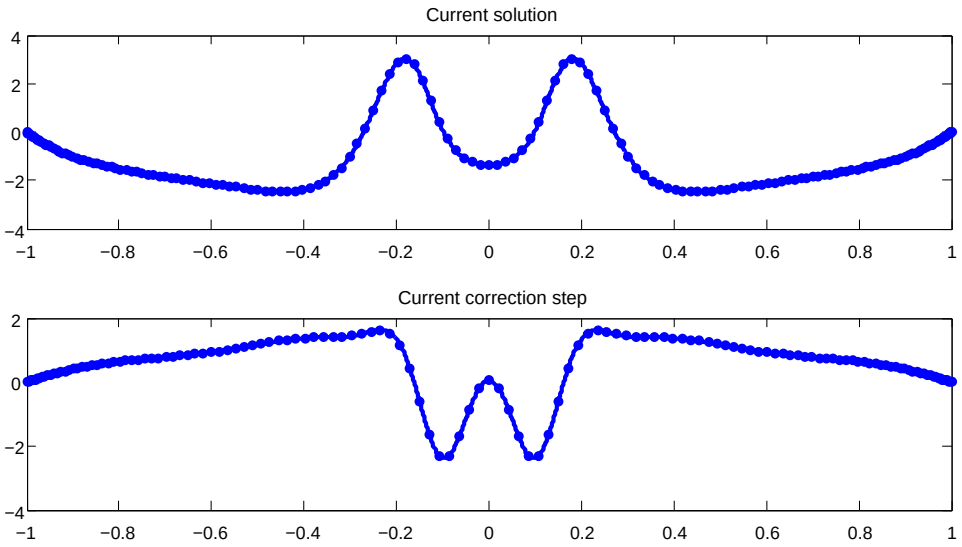

Figure 4: A plot shown after the second iteration of solving the BVP (4.6). The top plot shows the current guess of the solution and the bottom plot shows the latest Newton update. The filled circles on the lines are located at the Chebyshev interpolation points (for the respective degrees of polynomials of each function).

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 as is the case in this example.

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

```
>> chebopprefʼplottingʼ,ʼonʼ)
```

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 above (using a pure Newton iteration and starting from the initial guess (4.7)).

If we want to change the BVP (4.6) so that we impose different boundary conditions, we can reuse the chebop created above and only change the relevant fields. If we now want to solve

$$\epsilon u'' + 2(1-x^2)u + u^2 = 1, \quad u(-1) = 1, \quad u'(1) + u(1) = 0 \tag{4.8}$$

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

```
>> N.lbc = 1;
>> N.rbc = @(u) diff(u)+u;
>> [u delta_norms] = N\1;
```

(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)
ans =
    3.056740713605460e-011
```

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

$$u'' + 2u\sin u = 0, \tag{4.9}$$
$$u'(0) = 0, \tag{4.10}$$
$$u(5)u'(5) = 2, \tag{4.11}$$

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

```
>> chebopprefʼ(ʼdampedʼ,ʼoffʼ)
```

The problem is then solved as follows:

```
>> [d,x,N] = domain(0,5);
>> N.op = @(u) diff(u,2)+2*u.*sin(u);
>> N.lbc = ʼneumannʼ;
>> N.rbc = @(u) diff(u).*u-2;
>> N.guess = x;
>> [u normv] = N\0;
```

This code runs in 2.3 seconds, which is somewhat longer than the 1.4 seconds the code in Section 2 took to solve the problem. The main difference originates from the fact that now we are calculating the Fréchet 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 nonlinear backslash 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.

Many more examples are available online in the Chebfun software package.

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. In order to avoid runaway memory usage, we check the size of anons in the

class constructor. If it exceeds a certain size (the default is 100 MB), a dummy anon is returned and AD calculations for further operations on the associated chebfun are disabled.

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

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

This implies that everytime 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.

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. 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 [5].

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. Such developments may improve the robustness of convergence for BVPs in which suitable initial guesses are difficult to come by, and we foresee to offer the possibility of better initial guesses created by automated continuation methods (for discussion on numerical continuation methods, see [1]).

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 [2, 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 λ_k.

Define the objective function

$$\gamma(w_k^\lambda) = \frac{1}{2}\|w_k^\lambda\|$$

Initialize control parameters; $\lambda_{\min} = 0.1$ (mininum allowed step size), $\sigma = 0.01$ (ensures sufficient decrease of γ), $\tau = 0.01$ (ensures validity of quadratic model of γ).
Set the initial value of λ to be $\lambda = 1$.
Store $\gamma_0 = \frac{1}{2}\|v\|$ (this holds since $w_k^0 = v_k$, see definition of w_k^λ below).
while λ has not been accepted **do**
  Find a solution w_k^λ to the linear BVP

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

  Set $\gamma_\lambda = \gamma(w_k^\lambda)$.
  [Test acceptance of λ. If value is not accepted, search for the next value of λ with the weak line search.]
  if $\gamma_\lambda \le (1 - 2\lambda\sigma)\gamma_0$ **then**
    Accept λ, set $\lambda_k = \lambda$.
  else
    $\lambda = \max\left(\tau\lambda, \dfrac{\lambda^2\gamma_0}{(2\lambda - 1)\gamma_0 + \gamma_\lambda}\right)$
    if $\lambda < \lambda_{\min}$ **then**
      Accept the value $\lambda_{\min}$ for λ, set $\lambda_k = \lambda_{\min}$.
    end
    if $\lambda_{k-3} = \lambda_{k-2} = \lambda_{k-1} = \lambda_k = \lambda_{\min}$ **then**
      Set $\lambda_k = 1$ to take the full Newton step and give the solution process a kick.
    end
  end
end
return λ_k.

Acknowledgements. We wish to thank Nick Trefethen for the many comments, suggestions and revisions while working on this paper. We are also grateful to Shaun Forth at Cranfield University and Peter Deuflhard and Martin Weiser at the Zuse Institute for discussions on automatic differentiation and solving methods for boundary-value problems respectively. Finally, we would like to acknowledge Pedro Gonnet, Oxford University, for pointing out that our AD implementation can result in unnecesary linop calculations. Both authors were supported for this work by UK EPSRC Grant EP/E045847. The first author was also supported by Sloane Robinson Foundation Graduate Award associated with Lincoln College, Oxford.

References

[1] E. L. Allgower and K. Georg, *Introduction to Numerical Continuation Methods*, SIAM, Philadelphia, 2003.

[2] U. M. Ascher, R. M. M. Mattheij, and R. D. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM, Philadelphia, 1995.

[3] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, McGraw–Hill, New York, 1978.

[4] C. H. Bischof and H. M. Bücker, *Computing derivatives of computer programs*, in Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition, J. Grotendorst, ed., vol. 3 of NIC Series, NIC-Directors, Jülich, 2000, pp. 315–327.

[5] H. G. Bock, E. Kostina, and J. P. Schlöder, *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 Modelling and Optimization, M. J. Powell and S. Scholtes, eds., Deventer, The Netherlands, 2000, Kluwer, B.V., pp. 51–74.

[6] P. Deuflhard, *Newton Methods for Nonlinear Problems*, Springer-Verlag, Berlin/Heidelberg, 2006.

[7] T. A. Driscoll, F. Bornemann, and L. N. Trefethen, *The chebop system for automatic solution of differential equations*, BIT Numerical Mathematics, 48 (2008), pp. 701–723.

[8] S. A. Forth, *An efficient overloaded implementation of forward mode automatic differentiation in* Matlab, ACM Trans. Math. Soft., 32 (2006), pp. 195–222.

[9] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2nd ed., 2008.

[10] V. Hutson, J. S. Pym, and M. J. Cloud, *Applications of Functional Analysis and Operator Theory*, Elsevier, Amsterdam, 2nd ed., 2005.

[11] R. Kharche and S. Forth, *Source transformation for* Matlab *automatic differentiation*, in Computational Science - ICCS 2006, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, eds., vol. 3994 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2006, pp. 558–565.

[12] S. Rump, *INTLAB – INTerval LABoratory*, in Developments in Reliable Computing, T. Csendes, ed., Kluwer Academic Publishers, Dordrecht, 1999, pp. 77–104.

[13] L. F. Shampine, R. Ketzscher, and S. A. Forth, *Using AD to solve BVPs in* Matlab, ACM Trans. Math. Soft., 31 (2006), pp. 79–94.

[14] L. N. Trefethen, N. Hale, R. B. Platte, T. A. Driscoll, and R. Pachón, *Chebfun version 3*. Oxford University, 2009. http://www.maths.ox.ac.uk/chebfun/.