BIT Numerical Mathematics (2008)46:000-000
DOI:10.1007/s10543-000-0000-x
© Springer 2008.

THE CHEBOP SYSTEM FOR AUTOMATIC SOLUTION OF DIFFERENTIAL EQUATIONS *

TOBIN A. DRISCOLL[1], FOLKMAR BORNEMANN[2] and LLOYD N. TREFETHEN[3] †

[1] *Department of Mathematical Sciences, University of Delaware Newark, DE 19716, USA. email: driscoll@udel.edu*

[2] *Zentrum Mathematik - M3, Technical University of Munich, 85747 Garching bei München, Germany. email: bornemann@ma.tum.de*

[3] *Computing Laboratory, University of Oxford, Parks Rd., Oxford OX1 3QD, UK. email: LNT@comlab.ox.ac.uk*

Abstract.

In MATLAB, it would be good to be able to solve a linear differential equation by typing `u = L\f`, where `f`, `u`, and `L` are representations of the right-hand side, the solution, and the differential operator with boundary conditions. Similarly it would be good to be able to exponentiate an operator with `expm(L)` or determine eigenvalues and eigenfunctions with `eigs(L)`. A system is described in which such calculations are indeed possible, based on the previously developed chebfun system in object-oriented MATLAB. The algorithms involved amount to spectral collocation methods on Chebyshev grids of automatically determined resolution.

AMS subject classification (2000): 65L10, 65M70, 65N35.

Key words: chebfun, chebop, spectral method, Chebyshev points, object-oriented MATLAB, differential equations

1 Introduction.

Thousands of scientists, engineers, and numerical analysts use MATLAB for linear algebra computations. To solve a system of equations or a least-squares problem, or to find eigenvalues or singular values, one need only type "\" or "`eig`" or "`svd`". Most of the time we don't need to trouble ourselves with the details of the underlying algorithms such as Gaussian elimination or QR iteration. This article describes a step toward achieving the same kind of easy operation for differential equations.

The system we shall introduce is built on the chebfun system, whose properties we now briefly summarize [3, 4, 16, 20]. In MATLAB, one may start with a vector `v` and apply operations such as `sum(v)` (sum of the components), `diff(v)` (finite differences), or `norm(v)` (square root of sum of squares). In the chebfun system vectors are replaced by functions defined on an interval $[a, b]$, and commands like these are overloaded by their continuous analogues such as integral, derivative,

*Received ?. Revised ?. Communicated by ?.

†The work of Driscoll and Trefethen was supported by UK EPSRC Grant EP/E045847.

or L^2-norm. The functions are represented, invisibly to the user, by interpolants in suitably rescaled Chebyshev points $\cos(j\pi/n)$, $0 \le j \le n$, or equivalently by expansions in rescaled Chebyshev polynomials, either globally or piecewise. For example, the following commands construct a chebfun `f` corresponding to $f(x) = \sin(x) + \sin(x^2)$ on the interval $[0, 10]$ and plot the image shown on the left in Figure 1.1.

```
>> f = chebfun('sin(x)+sin(x.^2)',[0,10]);
>> plot(f)
```

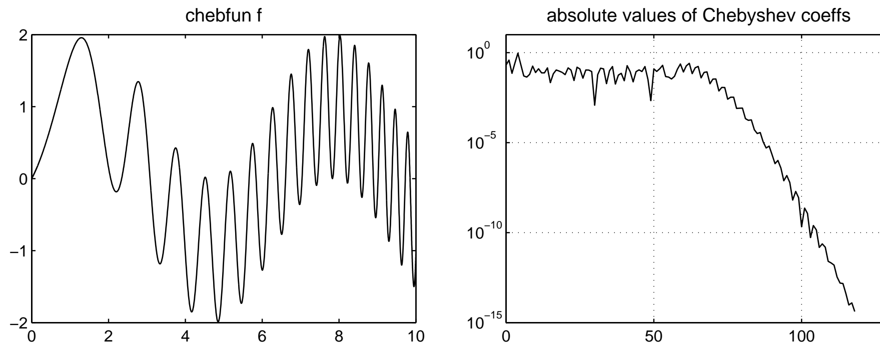

Figure 1.1: On the left, a chebfun, realized in this case by a polynomial interpolant through 119 scaled Chebyshev points. On the right, the absolute values of the corresponding Chebyshev coefficients, showing the automatically determined truncation at the level of machine precision.

Here are the integral from 0 to 10, the norm, and the global maximum:

```
>> sum(f)
ans =  2.422742429006074
>> norm(f)
ans =  3.254782212326119
>> max(f)
ans =  1.985446580874098
```

To construct the chebfun, the function is sampled at $9, 17, 33, \ldots$ appropriately scaled Chebyshev points, and in each case the polynomial interpolant through these data is found and converted by FFT to coefficients of a Chebyshev expansion. The process is terminated when the Chebyshev coefficients fall to a relative magnitude of about 10^{-16}. We can verify this truncation with the following commands, which produce the plot on the right of Figure 1.1.

```
>> a = chebpoly(f);
>> n = length(f)
   n = 119
>> semilogy(0:n-1,abs(a(n:-1:1)))
```

Evidently it only takes a polynomial interpolant through 119 Chebyshev points, that is, a polynomial of degree 118, to represent this function f to machine precision. The integral, norm, and maximum are calculated by algorithms described in [4].

Thus the central principle of the chebfun system is to evaluate functions in sufficiently many Chebyshev points for a polynomial interpolant to be accurate to machine precision. But what if the function of interest is only defined implicitly as the solution of a differential equation? Can one develop a similar way of solving problems with MATLAB ease? In this paper we propose a method of doing this based on collocation in the Chebyshev points and lazy evaluation of the associated spectral discretization matrices, all implemented in object-oriented MATLAB on top of the chebfun system.

Chebops solve linear ordinary differential equations. One may then use them as tools for more complicated computations that may be nonlinear and may involve partial differential equations. This is analogous to the situation in MATLAB itself, and indeed in computational science generally, where the basic tools are linear and vector-oriented but they are exploited all the time to solve nonlinear and multidimensional problems.

2 chebop syntax: eye, diff, cumsum, diag, expm.

This section introduces the grammar of chebops and illustrates what they can do. The following two sections explain how they do it.

We begin by specifying a domain such as $[0, 1]$, optionally generating at the same time a chebfun `x` corresponding to the linear variable on that domain:

```
[d,x] = domain(0,1);
```

In a chebfun computation, one could now construct a function on this domain and perform operations like differentiation:

```
f = cos(x);             % make another chebfun
fp = diff(f);           % 1st derivative
fpp = diff(f,2);        % 2nd derivative
```

In the chebop system, we can construct differential and integral operators that encapsulate such calculations. For example, here we construct a chebop corresponding to the operator $L : u \mapsto 0.0025u'' + u$ on $[0, 1]$:

```
D2 = diff(d,2);
I = eye(d);
L = 0.0025*D2 + I;
```

Syntactically speaking, what is going on in this sequence of operations is as follows. The chebop system is implemented by means of four MATLAB classes: domain, chebop, oparray, and varmat. A domain object is an interval $[a, b]$ on which chebfuns and chebops may be defined. A chebop object is an operator that can be applied to chebfuns. Oparray and varmat objects, which are not

normally relevant at user level, will be discussed in the next section. In the first line above, the command `diff` has been overloaded so that when its first input argument is a domain, the output is a chebop defined for that domain, in this case a chebop corresponding to the second derivative operator. In the second line, the command `eye` has been overloaded so that when its argument is a domain, the output is a chebop corresponding to the identity operator for that domain. (It is not actually necessary here to define `I`; the command `L = 0.0025*D2 + 1` would have the same effect.) In the third line, the `*` and `+` operators have been overloaded to apply to chebops, producing further chebops as output.

Via a further overload of the `*` operator, one can now apply `L` to a chebfun defined on `d`. Here are two examples.

```
>> norm(L*f)
ans =  0.850701052485606
>> norm(L*sin(20*x))
ans =  4.979316639832139e-14
```

The second result reveals that $\sin(20x)$ is annihilated by `L`. (The fact that the computed norm is a good deal larger than machine precision reflects standard ill-conditioning in spectral discretizations, not inaccuracies of the chebop system per se. For comments on how this effect can be avoided, see §7.)

To solve differential equations, we must impose boundary conditions and then invert the operator. For example, this command augments `L` with the boundary conditions $u(0) = u(1) = 0$:

```
L.bc = 'dirichlet';
```

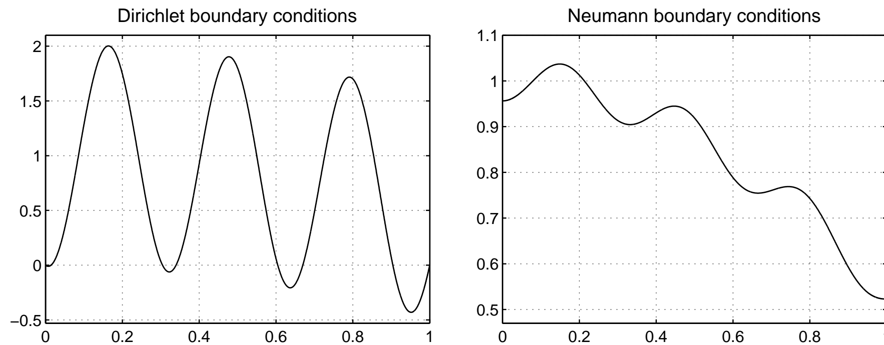

Figure 2.1: Solutions to two boundary value problems $0.0025u'' + u = f$ via `u = L\f`.

We can now solve a boundary value problem using the backslash operator; the solution is plotted in Figure 2.1.

```
u = L\f; plot(u)
```

Here is confirmation that the solution has been achieved to high accuracy.

```
>> length(u)
ans = 35
>> norm(L*u-f)
ans =  4.407101448104595e-14
```

Since the output is a chebfun, we can apply further operations to it:

```
>> mean(u)                  % mean value of solution
ans =  0.793521203939255
>> u(0.5)                   % value at x=0.5
ans =  1.799943564035490
>> roots(u)                 % zeros of u
ans =
                  0
  0.014418616029463
  0.304557992965981
  0.339742830044090
  0.604543335555234
  0.669638743984168
  0.904061151243311
  1.000000000000000
```

Different boundary conditions can also be imposed:

```
L.lbc = pi; L.rbc = sqrt(2);          % inhomogeneous Dirichlet
L.lbc = {D,pi}; L.rbc = {D,sqrt(2)};  % inhomogeneous Neumann
L.bc = 'neumann';                     % homogeneous Neumann
L.bc = 'periodic';                    % periodic
```

The right side of Figure 2.1 shows the result obtained in the case of Neumann boundary conditions at both ends. Here are the largest six eigenvalues of that operator, calculated by a command `eigs` overloaded to apply to chebops:

```
>> eigs(L)
ans =
  0.999999999999990
  0.975325988997260
  0.901303955989076
  0.777933900975464
  0.605215823956393
  0.383149724931890
```

These numbers match the exact eigenvalues $1 - \pi^2 k^2/400$ $(k = 0, 1, \ldots, 5)$ in all but the final two digits.

Another way to construct a chebop is by applying the overloaded "`diag`" command to a chebfun `g`. The resulting chebop corresponds to the multiplication operator $L : u(x) \mapsto g(x)u(x)$. In the following example we determine the first six eigenvalues of the harmonic oscillator operator $L : u \mapsto -u'' + x^2 u$ on the real axis, here truncated to $[-10, 10]$:

```
>> [d,x] = domain(-10,10);
>> L = -diff(d,2) + diag(x.^2);
>> eigs(L)
ans =
  1.000000000000143
  3.000000000000008
  5.000000000000036
  7.000000000000032
  8.999999999999998
 10.999999999999966
```

The computation takes about 0.1 secs. on a 2008 workstation, and the results closely match the exact eigenvalues 1, 3, 5, 7, 9, 11.

The chebop system also permits exponentiation. In MATLAB, `expm(A)` computes the exponential of a matrix A, and this command has been overloaded to compute the exponential of a chebop. One can use `expm(t*L)` to compute the solution at time t of a PDE $u_t = Lu$ with appropriate initial and boundary conditions (see Example 5 of §5).

In summary, chebops can be constructed from the following operations:

```
eye(domain)             % identity operator
zeros(domain)           % zero operator
diff(domain)            % differentiation operator
cumsum(domain)          % indefinite integration operator
diag(chebfun)           % multiplication operator
expm(chebop)            % exponential of operator
scalar op chebop        % op is +, -, *
chebop op scalar        % op is +, -, *, /
chebop op chebop        % op is +, -, *
chebop^posint           % nonnegative integer power
```

When it comes to applying a chebop, there are two possibilities, both of which produce chebfuns:

FORWARD MODE: `chebfun = chebop*chebfun, e.g. f = L*u`
INVERSE MODE: `chebfun = chebop\chebfun, e.g. u = L\f`

The chebfuns produced in all cases satisfy the chebfun constructor criteria, as in Figure 1.1, so one may expect accuracy not too far from machine precision. In addition, eigenvalues and optionally eigenfunctions of a chebop, or of a pair of chebops defining a generalized eigenvalue problem $Au = \lambda Bu$, can be computed with `eigs`.

Here is another example. We can solve the Airy equation $u'' - xu = 1$ on $[-30, 30]$, with boundary conditions $u(-30) = 0$ and $u(30) = 4$, by the following sequence:

```
[d,x] = domain(-30,30);
L = diff(d,2) - diag(x);
f = chebfun(1,d);
L.lbc = 0; L.rbc = 4;
u = L\f;
```

The result is shown on the left in Figure 2.2, and the right side of the same figure shows what happens if the equation is altered to $u'' + 0.1x^2u = 1$. The lengths of the two chebfuns are 172 and 229, and their integrals are 9.52882658199204 and −36.0659308455985.

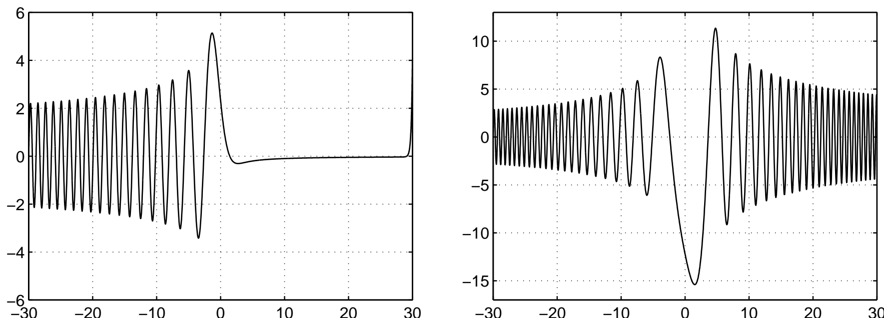

Figure 2.2: Left: solution to the Airy equation $u'' - xu = 1$ on $[-30, 30]$ with boundary conditions $u(-30) = 0$, $u(30) = 4$. Right: same but with $-x$ replaced by $+0.1x^2$.

3 chebop implementation: lazy evaluation of spectral discretizations.

The implementation of chebops combines the numerical analysis idea of spectral collocation with the computer science idea of lazy or delayed evaluation.

Let `L` be a chebop. There are various fields that define `L` (currently ten; see §4), of which two are the crucial ones conceptually: `L.oparray` and `L.varmat`. `L.oparray` is an oparray, a function handle or array of function handles that normally determines what happens when `L` is applied in forward mode. (An exception is the command `expm`; see §4.) For example, consider the following sequence of three commands. On the right is shown the anonymous function created as the `oparray` field in each case, which can subsequently be applied to a chebfun `u`.

```
I = eye(d)    ⟶   I.oparray = @(u) u
D = diff(d)   ⟶   D.oparray = @(u) diff(u)
L = I + D     ⟶   L.oparray = @(u) I(u)+D(u)
```

Whenever chebops are combined to make new chebops, `L.oparray` is updated by operations such as these. The chebop system does not need to keep track of the details; the bookkeeping resides in the anonymous functions. The reason "array" is part of the name is that for systems of equations, there is more going on: an oparray is not just a single operator but a square array of operators, and rules of array composition and addition have been implemented so that oparrays can be combined algebraically.

`L.varmat` is a varmat, another new class that normally determines what happens when `L` is applied in inverse mode. A varmat is a matrix of undetermined

dimension, or rather a prescription for how to construct a matrix of arbitrary dimension. For example:

```
>> I = varmat(@eye);
>> I(3)
ans =
     1     0     0
     0     1     0
     0     0     1
>> J = I + 2*I;
>> J(5)
ans =
     3     0     0     0     0
     0     3     0     0     0
     0     0     3     0     0
     0     0     0     3     0
     0     0     0     0     3
```

One can combine varmats with the usual operations of algebra, and again the bookkeeping is done by MATLAB anonymous functions.

Here is what the chebop system does when it executes a command of the form `u = L\f`, where `L` is a chebop and `f` is a chebfun. The chebfun constructor is called to construct a chebfun `u` following the usual procedure involving sampling at $n = 9, 17, 33, \ldots$ points, as outlined in §1. The new feature is what it means to "sample" at n points. Given a value of n, the varmat associated with `L` is instantiated as a matrix L_n and `f` is sampled at appropriate Chebyshev points to yield a vector f_n. MATLAB's ordinary backslash command is then invoked to compute a vector $u_n = L_n \backslash f_n$. The process continues on successively finer grids, as usual, until the Chebyshev coefficients of u_n fall to a relative magnitude on the order of machine precision. (The actual termination condition has some special features, discussed in the next section.)

If chebops were just multiples of the identity as in the example above, all this would not be very interesting. The mathematical substance comes from the fact that the matrices involved may be spectral discretizations of differential and integral operators. For example, in the computation `u = L\f` whose result is plotted on the left in Figure 2.1, we have `length(f) = 13` (since `f` is just a cosine) and `length(u) = 35` (since the small coefficient on the second derivative makes `u` oscillate). The length of `u` has been determined automatically, with no explicit connection to the length of `f`.

The spectral discretizations we employ are standard ones based on polynomial interpolants in Chebyshev points as described in [6, 9, 19]. For example, to differentiate a function spectrally, one interpolates it by a polynomial in Chebyshev points, differentiates the interpolant, and evaluates the result at the same points. This is equivalent to a matrix-vector multiplication, and the process can be inverted (or could be if this were not a singular example) by solving a linear system of equations. The requisite matrix is constructed from standard formulas based on the code `cheb` of [19]. The precise matrix formulations for

the inverse mode inevitably depend on boundary conditions, and it is these that make the matrices nonsingular. Chebop boundary conditions are imposed in standard ways as described in Chapters 7 and 13 of [19] and Chapter 6 of [6]. Each boundary condition at the left end of a domain involves the modification of one of the initial rows of the spectral matrix, and each boundary condition on the right modifies one of the final rows.

For example, here is the second-order spectral differentiation matrix on a grid of 5 points in $[-1, 1]$, with no boundary conditions:

```
>> d = domain(-1,1);
>> D2 = diff(d,2);
>> D2(5)
ans =
   17.0000  -28.4853   18.0000  -11.5147    5.0000
    9.2426  -14.0000    6.0000   -2.0000    0.7574
   -1.0000    4.0000   -6.0000    4.0000   -1.0000
    0.7574   -2.0000    6.0000  -14.0000    9.2426
    5.0000  -11.5147   18.0000  -28.4853   17.0000
```

If we apply Dirichlet boundary conditions, the first and last rows are replaced by corresponding rows of the identity:

```
>> D2.bc = 'dirichlet';
>> D2(5)
ans =
    1.0000         0         0         0         0
    9.2426  -14.0000    6.0000   -2.0000    0.7574
   -1.0000    4.0000   -6.0000    4.0000   -1.0000
    0.7574   -2.0000    6.0000  -14.0000    9.2426
         0         0         0         0    1.0000
```

With Neumann boundary conditions, the first and last rows are replaced by corresponding rows of the first derivative operator:

```
>> D2.bc = 'neumann';
>> D2(5)
ans =
   -5.5000    6.8284   -2.0000    1.1716   -0.5000
    9.2426  -14.0000    6.0000   -2.0000    0.7574
   -1.0000    4.0000   -6.0000    4.0000   -1.0000
    0.7574   -2.0000    6.0000  -14.0000    9.2426
    0.5000   -1.1716    2.0000   -6.8284    5.5000
```

4 Further details.

We now give more details about some of the features of the chebop system.

Sparse matrices. Where possible, our algorithms make use of MATLAB sparse matrices [10], and in particular, these are the formats produced by the `eye` and

`diag` operations. For example, the program `eye.m` in the `@domain` directory—i.e., the overloaded `eye` command that is invoked when the argument is a domain—essentially consists of one line: `I = chebop(@speye,@(u)u,d)`. This constructs the chebop `I` that generates matrices from MATLAB's `speye` command in inverse mode and acts like `@(u)u` in forward mode.

Piecewise smooth chebfuns. The chebfun system is designed to work with piecewise as well as globally smooth functions. For example, the command `f = chebfun('abs(x)')` produces a chebfun on $[-1,1]$ with two linear pieces. For the most part, however, the chebop system is not currently able to operate with piecewise smooth chebfuns. An exception is the `expm` command, which makes it possible in some cases to calculate chebfuns of the form `expm(L)*f` even when `f` is only piecewise smooth (because the result is smooth even though `f` is not, and no matrix inverse is involved). Such a calculation is illustrated in Example 5 of the next section.

More about boundary conditions. We have mentioned the basic method of imposing boundary conditions in the chebop system: replacement of rows at the beginning and/or end of the matrices defining the chebop by corresponding rows of other matrices. These normally correspond to the identity or a derivative, but arbitrary matrices are allowed. The entries of the right-hand-side vector get adjusted in the same positions according to what boundary values have been specified. When the boundary condition is specified as `'periodic'`, we find ourselves in the nonstandard situation of solving problems by Chebyshev methods that would normally get a Fourier treatment. For a differential operator of order d, d rows of the matrix are replaced by differences of rows of appropriate spectral differentiation matrices to enforce the conditions $u^{(\nu)}(a) = u^{(\nu)}(b)$, $0 \le \nu \le d-1$, where a and b are the endpoints of the domain.

Accuracy and filtering. When a chebfun is constructed, the aim is always accuracy close to machine epsilon, although safeguards are built into the constructor to minimize the risk of failure of convergence in difficult cases. For chebops, difficult cases are almost universal because of the ill-conditioning associated with spectral discretizations. As the examples in this article show, it is common to lose three or four digits of accuracy in such computations, and this figure can be much worse, for example in the case of a fourth-order operator on a fine grid. Accordingly, a crucial feature of the chebop system is a procedure to weaken the convergence criterion so as to stop when a "noise plateau" appears to be reached. This is achieved through the use of a function called `filter`.

Adjustable scaling. Ordinarily in the chebfun and chebop systems, as in IEEE arithmetic, all calculations and convergence criteria are relative: if you multiply a function by 10^{-100}, its chebfun will be multiplied by 10^{-100} too. In particular, when a chebop is constructed, normally the constructor will aim to achieve a precision about 15 orders of magnitude below the scale of the function, or as close to this as the filtering process just described allows. However, there are situations where it is wasteful to seek so much accuracy. For example, in a Newton iteration for a nonlinear problem, one might at some step add a correction of size 10^{-10} to a function whose accuracy will ultimately be 10^{-14}. In such a case there is no

point in trying to resolve the correction to more than four relative digits. The chebop system provides a field `L.scale` which can be used to achieve this effect (see Example 8 of §6). In the calculation just mentioned, the user could set `L.scale` to 1 to force all chebops to be constructed relative to that fixed scale.

Storage of LU factors. A calculation of the form `u = L\f` generates matrices L_n of various dimensions and solves corresponding linear systems of equations using MATLAB's backslash command, based on LU factorization. The amount of work involved is $O(n^3)$ for each matrix, and this becomes a problem if n is larger than in the hundreds. Even for n in the hundreds, the operation count is a problem in contexts where chebops are to be applied over and over again, for example in time-stepping for solving time-dependent PDEs. To speed up such calculations, the chebop system includes an option whereby, whenever a linear system is to be solved involving a matrix L_n, a check is made as to whether this matrix has appeared before. If not, its LU factors are computed and stored in a MATLAB persistent variable. If so, previously stored LU factors are recalled from storage so that the system can be solved with $O(n^2)$ work. All this happens invisibly to the user and makes some PDE calculations feasible that would otherwise be very slow. See Example 7 of §6.

Chebop fields. Section 3 described two of the fields defining a chebop, namely `oparray` (for forward mode) and `varmat` (for inverse mode, describing matrices with boundary conditions), and we have also mentioned the field `L.scale`. In addition there are seven other fields. `fundomain` is the domain on which the chebop is defined. `difforder` and `numbc` keep track of the order of a differential operator and of the number of boundary conditions, respectively; a warning message is printed if one attempts a computation `L\f` where the two do not match. `difforder` also stores information needed to implement the mnemonics `'dirichlet'`, `'neumann'`, and `'periodic'`. `lbc` and `rbc` store information related to the left and right boundary conditions. `ID` stores an identification number to enable the system to know whether a chebop is new or old for the storage and retrieval of LU factors. `blocksize` stores information related to the solution of systems of equations.

How `expm` *works.* If `L` is a chebop and `f` is a chebfun, then in most cases `L*f` is computed from an operator representation `L.oparray` and `L\f` from a matrix representation `L.varmat`. However, for some chebops an operator representation is not available, and here both `L*f` and `L\f` are computed from matrices. In particular, this is the situation with `expm`: both `expm(L)*f` and `expm(L)\f` are computed from matrices constructed from MATLAB's `expm` command.

How `eigs` *works.* In MATLAB, `eig` is the command for dense matrix eigenvalue and generalized eigenvalue problems, where all eigenvalues are computed by the QR algorithm and related methods as in LAPACK [2], and `eigs` is the command for sparse problems, where a small number of eigenvalues are computed by the implicitly restarted Arnoldi iteration of ARPACK [12]. In the chebop system the functionality required is that of `eigs`: an operator will generally have infinitely many eigenvalues, and we want to find a finite number of them. The chebop eigenvalue command is accordingly called `eigs`, but its implementation is actu-

ally based on applying MATLAB `eig` to dense matrices, retaining whatever types of boundary conditions have been specified but taking them to be homogeneous.

It is a tricky matter to decide which modes ought to be returned by `eigs`, and how to use the chebfun constructor to determine when they have been found accurately. Our approach is as follows. As with the standard MATLAB `eigs`, the user may explicitly request any number of eigenvalues of largest or smallest magnitude, or of smallest or largest real part, or those closest to a given complex number λ_0. If the user does not give this information, the function computes eigenvalues of matrices at sizes 33 and 65. If all of the eigenvalues change significantly during this refinement, the one that changes the least is taken to be λ_0. If some eigenvalues change very little, the changes could be largely due to rounding error, so their eigenvectors are analyzed and the eigenvector with the smallest 1-norm of Chebyshev expansion coefficients is used to select λ_0.

Once λ_0 is determined, the chebfun constructor is called. It samples a function that finds a discrete eigendecomposition, sorts eigenvectors according to the given eigenvalue criterion, and returns a fixed, randomly selected linear combination of the number of eigenvectors requested by the user. In this way, the constructor adapts to information from all of the eigenfunctions without risking an accidental cancellation that could result from, say, a simple sum of them. Both the automatic selection of λ_0 and the convergence criterion work well in many applications, but these methods are undoubtedly susceptible to failure, particularly for highly non-normal operators. As a practical safeguard, `eigs` exits with an error message if the adaptively selected discretization grows larger than size 1025.

Systems of equations. The chebop system can also solve problems involving $k > 1$ coupled variables. In such a case the variables are stored not just as chebfuns but as “quasimatrices” whose “columns” are chebfuns. The chebops that operate on such structures have an appropriate $k \times k$ block form. We shall not give details apart from Example 6 below.

5 Examples and applications.

In this section we present six further examples of linear ODE calculations, illustrating a number of variations:

- initial-value problems (Examples 1,6)
- variable coefficients (Examples 2,3,4)
- periodic boundary conditions (Example 3)
- eigenvalue and generalized eigenvalue problems (Examples 3,4)
- complex variables (Example 4)
- fourth-order differentiation (Example 4)
- exponentiation of an operator (Example 5)
- piecewise smooth functions (Example 5)
- systems of equations (Example 6)

In each case we report one or more numerical quantities as points of comparison for others who may wish to perform tests on the same problems.

Example 1. Initial-value problem. The first example shows the use of chebops to solve a 2nd-order initial-value problem:

$$u'' + \pi^2 u = 0, \quad u(0) = 1, \; u'(0) = 0. \tag{5.1}$$

The following code performs the calculation, producing the chebfun plotted in Figure 5.1, a good approximation to the exact solution $\cos(\pi x)$.

```
d = domain(0,40); D = diff(d);
L = D^2 + pi^2;
L.lbc(1) = 1; L.lbc(2) = D;
u = L\0;
```

Notice that the first boundary condition is implicitly taken to be Dirichlet, since the type is not specified, and the second is implicitly taken to have value 0, since the value is not specified. An inspection of the solution reveals `length(u) = 119` and $u(40) = 1.000000000017879$.

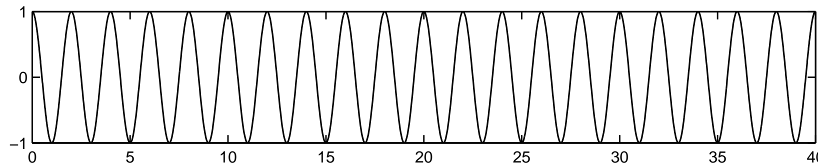

Figure 5.1: The function $u(x) = \cos(\pi x)$ computed as the solution to the initial-value problem (5.1). The chebfun has length 119 and the accuracy is about 12 digits.

Example 2. Bessel equation. Next we consider a boundary-value problem for the Bessel equation,

$$x^2 u'' + x u' + (x^2 - \nu^2) u = 0, \quad x \in [0, 60], \quad x(0) = 0, \; x(60) = 1, \tag{5.2}$$

with $\nu = 1$. The solution is a multiple of the Bessel function $J_1(x)$, namely $u(x) = J_1(x)/J_1(60)$. The left side of Figure 5.2 shows the solution computed with the following code:

```
[d,x] = domain(0,60);
I = eye(d); D = diff(d); X = diag(x);
L = X^2*D^2 + X*D + (X^2-I);
L.lbc = 0; L.rbc = 1;
u = L\0;
```

We can further examine the solution with these commands:

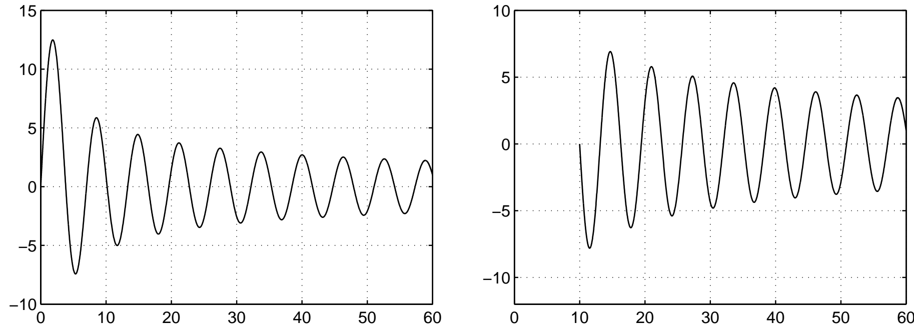

Figure 5.2: On the left, solution to the Bessel boundary-value problem (5.2) for $\nu = 1$, with length 77. On the right, the same with the left boundary moved from $x = 0$ to $x = 10$.

```
>> exact = chebfun('besselj(1,x)',d)/besselj(1,60);
>> error = norm(u-exact,inf)
   error = 8.929810672665855e-12
>> tv = norm(diff(u),1)
   tv = 150.4756276607209
```

The first shows that the overall accuracy is about 11 digits. The second reports the total variation of the solution.

The right side of Figure 5.2 shows the result of a repetition of the same computation, but now with the left-hand boundary point moved from $x = 0$ to $x = 10$. As a result of this change, the solution is now a linear combination of $J_1(x)$ and $Y_1(x)$, the Neumann function of the same order. (Of course the chebop system does not know anything about Bessel and Neumann functions; it just computes solutions numerically.) The computed total variation changes to 154.2015444041971.

Example 3. Mathieu equation. Our third example is an eigenvalue problem with periodic boundary conditions, the Mathieu equation:

$$Lu = \lambda u, \quad Lu = -u'' + 2q\cos(2x)u, \quad u \in [-\pi, \pi], \tag{5.3}$$

where q is a real parameter. A chebop code looks like this:

```
q = 10;
[d,x] = domain(-pi,pi);
L = -diff(d,2) + 2*q*diag(cos(2*x));
eigs(L & 'periodic')
```

This program produces the following values of the first six eigenvalues, which match values reported on pp. 748–749 of [1]. A check in Mathematica shows that all but the last three digits are correct in each case.

```
-13.936979956658631  -13.936552479250203   -2.399142400035606
 -2.382158235956632    7.717369849779567    7.986069144681781
```

Example 4. Orr–Sommerfeld operator. The fourth example is an Orr–Sommerfeld eigenvalue problem. This is a generalized eigenvalue problem of fourth order involving complex variables and two parameters R, the Reynolds number, and α, a wave number:

$$Au = \lambda Bu, \quad B = (D^2 - \alpha^2 I), \quad A = B^2/R - i\alpha(2 + (1 - x^2)B). \tag{5.4}$$

The domain is $[-1, 1]$ and the boundary conditions are $u(-1) = u'(-1) = u(1) = u'(1) = 0$. This problem is associated with the eigenvalue instability of plane Poiseuille fluid flow, and in the code below, R and α are set to their critical values $R = 5772.22$ and $\alpha = 1.02056$, first determined by Orszag in 1971 [15].

```
[d,x] = domain(-1,1);
I = eye(d); D = diff(d);
R = 5772.22; alpha = 1.02056;
B = D^2 - alpha^2;
A = B^2/R - 1i*alpha*(2+diag(1-x.^2)*B);
A.lbc(1) = I; A.lbc(2) = D;
A.rbc(1) = I; A.rbc(2) = D;
e = eigs(A,B,50,'LR');
```

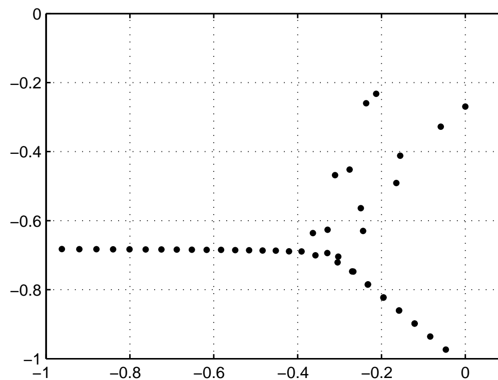

Figure 5.3: Eigenvalues of Orr–Sommerfeld operator (5.4) with critical parameters R and α. The seven dots on the lower-right branch each correspond to two eigenvalues very close to one another.

The computed eigenvalues are plotted in Figure 5.3. The choices of parameters are such that, to six or more digits, the rightmost eigenvalue in the complex plane should have real part equal to zero. This expectation is met nicely by the numerical value of that eigenvalue, $0.0000000505 - 0.2694296366i$.

Example 5. Heat and advection-diffusion equations. Our fifth example shows the exponentiation of an operator, in this case the second derivative operator on the interval $[-4, 2]$ with boundary conditions $u(-4) = u(2) = 0$. If L is this

operator and f is a function defined on $[-4,2]$, then the solution to the heat equation

$$u_t = u_{xx}, \quad x \in [-4,2], \quad u(x,0) = f(x), \quad u(-4,t) = u(2,t) = 0 \tag{5.5}$$

at time $t > 0$ is given by

$$u(t) = e^{tL} f. \tag{5.6}$$

(Properly speaking, the notation e^{tL} refers to a semigroup with generator L, but we will not concern ourselves with a rigorous formulation.) In the following code, the initial function is taken as a hat function, represented in the chebfun system through the piecewise smooth feature [16].

```
[d,x] = domain(-4,2);
L = diff(d,2); f = max(0,1-abs(x));
hold off, plot(f), hold on
for t = .1:.1:.5
  expLt = expm(t*L & 'dirichlet');
  v = expLt*f;
  plot(v)
end
```

The left side of Figure 5.4 shows a successful computation; the maximum value at the end is 0.368610340795447. For any $t > 0$, the solution is smooth and in principle this construction will work, but for very small t the length of the chebfun will be large and the computation based on exponentials of matrices very slow. Experiments show that the lengths of the solution chebfuns for $t = 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}$ are 68, 157, 291, 751.

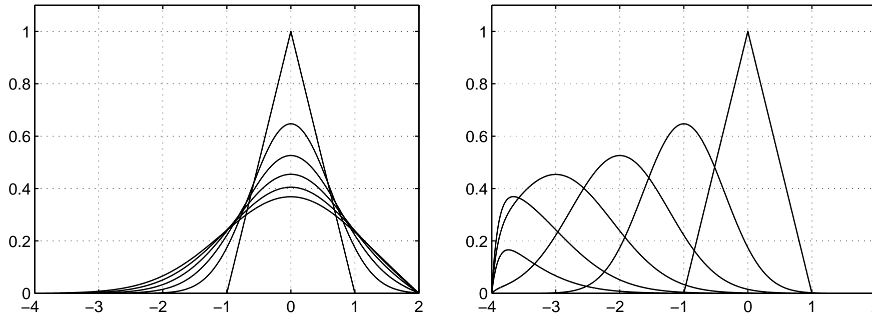

Figure 5.4: On the left, solution to the time-dependent PDE (5.5) at $t = 0, 0.1, \ldots, 0.5$ computed by the operator exponential `expm`. On the right, the same except with an advection term $20u_x$ added to the equation. Note: although `expm` can work with non-smooth inputs like these when the output is smooth, most chebop operations are restricted at present to globally smooth chebfuns.

The right side of Figure 5.4 shows the effect of adding an advection term to the equation, specifically, replacing `L = diff(d,2)` by `L = diff(d,2) + 20*diff(d)`. Now the maximum value at the end is 0.165845979585510.

Example 6. A system of equations. Consider two masses m with rest positions x_1 and x_2 coupled together by a spring of constant k, with the first mass coupled to a wall by another spring of the same constant. If friction terms of constant c are included, the free motions $x_1(t)$, $x_2(t)$ are governed by the equations

$$mx_1'' + cx_1' + 2kx_1 - kx_2 = 0, \quad mx_2'' + cx_2' + kx_2 - kx_1 = 0. \tag{5.7}$$

We choose to solve this equation as an initial value problem on the time interval $0 \le t \le 20$ with initial conditions

$$x_1(0) = -1, \quad x_2(0) = 1, \quad x_1'(0) = x_2'(0) = 0. \tag{5.8}$$

This problem has the form of a 2×2 system: a block 2×2 second-order operator times a block 2×1 vector of unknown functions $x_1(t)$ and $x_2(t)$ equals a block 2×1 vector of zero functions. As the code below shows, in the chebop system one sets up the problem in just this form, apart from one alteration: for implementation reasons the 2×1 vectors are actually stored as quasimatrices, which one would normally think of as corresponding to shape 1×2 rather than 2×1. Note that the boundary conditions, all at the initial point $t = 0$, are also expressed in terms of block operators. With `Z` defined as a zero operator, the chebops `[I Z]` and `[Z I]` apply to the values of first and second variables of the system, and similarly, `[D Z]` and `[Z D]` apply to x_1' and x_2'. The plot is shown in Figure 5.5. The computed energy of the system at $t = 20$ is 0.024320893389668.

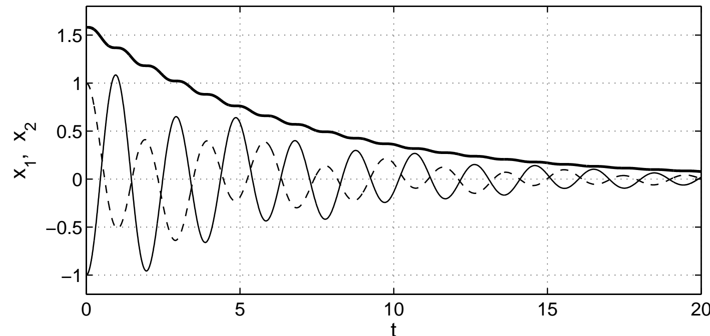

Figure 5.5: Solution of the coupled mass-spring system (5.7)–(5.8). The thin solid and dashed curves show the positions of the two masses as functions of time t, and the thick solid curve represents a pointwise upper bound derived from consideration of potential and kinetic energy. Note that the energy bound is flat at points of maximum amplitude, since the velocity is zero there and thus there is no damping.

```
[d,t] = domain(0,20);
D = diff(d); I = eye(d); Z = zeros(d);
m = 1; k = 4; c = 0.3;
A = [m*D^2+c*D+2*k*I -k*I; -k*I m*D^2+c*D+k*I];
A.lbc(1) = {[I Z],-1};      % pull x1 left
A.lbc(2) = {[Z I], 1};      % pull x2 right
```

```
A.lbc(3) =  [D Z];          % x1 at rest
A.lbc(4) =  [Z D];          % x2 at rest
u = A\[0*t 0*t];
x1 = x(:,1); x2 = x(:,2);
energy = m*(diff(x1).^2+diff(x2).^2)/2 + k*(x1.^2+(x2-x1).^2)/2;
energybound = sqrt(energy/k);
plot(u), hold on, plot(energybound)
```

6 Nonlinear problems and PDEs.

The previous sections have explored numerical methods and software for basic problems related to linear ordinary differential equations. As mentioned earlier, these tools provide a foundation for more complicated computations involving nonlinearity and/or partial differential equations. This is a large and open-ended area, and we illustrate just a few of the possibilities by presenting two examples involving first the solution of a nonlinear time-dependent PDE by time-stepping, then the solution of a nonlinear ODE by Newton iteration.

Example 7. Nonlinear Schrödinger equation. The cubic nonlinear Schrödinger equation

$$u_t = \frac{i}{2}u_{xx} + i|u|^2 u = Lu + G(u) \tag{6.1}$$

admits on $-\infty < x < \infty$ the soliton solution $4e^{i(2x+6t)}\mathrm{sech}(4(x-2t))$, a complex pulse traveling at speed 2. We solve this equation on the finite interval $[-4,4]$ with homogeneous Dirichlet boundary values by using chebops for the space discretization and standard ODE formulas for the time discretization. Since the time discretization is of a low order, the overall accuracy will be less than machine precision.

The computation begins by setting up the domain, the operators L and G, and the initial condition:

```
[d,x] = domain(-4,4);
L = 0.5i*diff(d,2);
G = @(u) 1i*u.*(u.*conj(u));
u = 4*exp(2i*x).*sech(4*x); u = u-u(4);
```

Next we initiate the time-stepping with two steps of a first-order method: backward Euler on the linear term and forward Euler on the nonlinear term.

```
tmax = 6; dt = 0.01;
A = (1-dt*L) & 'dirichlet';
Gu = zeros(d,1);
for n = 1:2
   Gu(:,n) = G(u(:,n));
   f = u(:,n) + dt*Gu(:,n);
   u(:,n+1) = A\f;
end
```

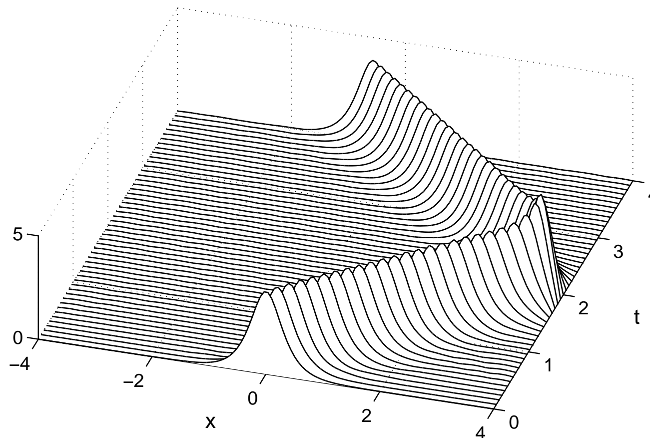

Figure 6.1: Solution of the cubic Schrödinger equation (6.1) by chebop discretization in x and finite differencing in t.

There are now enough data levels present to start up a higher-order time-stepping scheme. Our choice is the implicit/explicit formula

$$\frac{3}{2}u_{n+1} - 2u_n + \frac{1}{2}u_{n-1} = \Delta t\left(Lu_{n+1} + \frac{8}{3}G(u_n) - \frac{7}{3}G(u_{n-1}) + \frac{2}{3}G(u_{n-2})\right), \tag{6.2}$$

a combination of the second-order backward difference formula for L and the third-order Adams–Bashforth formula for G, chosen for its stability on a part of the imaginary axis. For each new time level, (6.2) together with the Dirichlet boundary conditions defines a linear boundary-value problem for the unknown u_{n+1}.

```
A = (1.5-dt*L) & 'dirichlet';
for n = 3:tmax/dt
   Gu(:,3) = G(u(:,n));
   f = u(:,n-1:n)*[-0.5;2] + Gu*([2;-7;8]*dt/3);
   u(:,n+1) = A\f;
   Gu(:,1) = Gu(:,2); Gu(:,2) = Gu(:,3);
end
```

In this application, since the time step is fixed, the linear operator defining the boundary-value problem never changes. This is a great advantage, for it means that the LU factors of discrete instances of the operator are computed just once and retrieved from memory thereafter, all invisibly to the user, as described in §4.

The following lines produce the plot shown in Figure 6.1.

```
xx = linspace(d,120); tt = dt*(0:8:400); uu = u(xx,1:8:401);
waterfall(xx,tt,abs(uu)')
```

Example 8. Tracy–Widom distribution for random matrices. Our final example stems from a problem arising in random matrix theory. Mathematically, the main computation involves a nonlinear ODE boundary-value problem with boundary conditions derived from truncating an infinite domain.

The boundary-value problem concerns the Hastings–McLeod connecting orbit of the Painlevé II equation,

$$\begin{aligned} u''(x) &= 2u(x)^3 + xu(x), \\ u(x) &= \sqrt{-x/2}\,\big(1 + \tfrac{1}{8}x^{-3} - \tfrac{73}{128}x^{-6} + \tfrac{10657}{1024}x^{-9} + O(x^{-12})\big) \quad (x \to -\infty), \\ u(x) &\sim \mathtt{Ai}(x) \quad (x \to \infty). \end{aligned}$$

We solve this by a Newton iteration using a function `newton` that also works for finite dimensional systems:

```
function x = newton(f,df,x,tol)
while true
   dx = -(df(x)\f(x));
   x = x + dx;
   if norm(dx) <= tol, break; end
end
```

We solve the problem with the code

```
function u = hml
[d,x] = domain(-30,8);
D = diff(d);

function f = residual(u)
   f = D^2*u - (2*u.^3 + x.*u);
end

function J = jacobian(u)
   J = D^2 - diag(6*u.^2 + x) & 'dirichlet';
   J.scale = norm(u);
end

lbc = @(s) sqrt(-s/2)*(1+1/(8*s^3)-73/(128*s^6)+10657/(1024*s^9));
rbc = @(s) airy(s);
u0  = chebfun([lbc(d(1)),rbc(d(2))],d);
u = newton(@residual,@jacobian,u0,1e-14);
end
```

The truncation of the infinite domain to the finite interval $[-30, 8]$ turns out to be accurate to an absolute error of about 10^{-14}. The initial guess `u0` is just the linear function satisfying the boundary conditions, so the Newton corrections only have to satisfy a homogeneous form of them. As noted in §4, it is important to construct the chebfuns for the Newton corrections with an accuracy relative to the scale of the iterates. This is achieved by including the line `J.scale`

= norm(u) in the definition of the Jacobian. Eight iterations are sufficient to converge to a correction size below the required tolerance of 10^{-14}, taking about 1 second on a workstation. The solution is correct to about 13 digits, which is considerably more accurate than the solution calculated by Dieng [8] using MATLAB's bvp4c with a far more sophisticated initial iterate.

Once the boundary-value problem is solved, the chebfun system can easily post-process u to yield the Tracy–Widom distribution $F_2(s)$ of random matrix theory. This gives, in the large matrix limit, the probability that the rescaled maximum eigenvalue of the Gaussian unitary ensemble is below s [18]:

$$F_2(s) = \exp\left(-\int_s^\infty \int_\xi^\infty u(\eta)^2 \, d\eta d\xi\right).$$

```
>> v = sum(u.^2)-cumsum(u.^2);
>> F2 = exp(cumsum(v)-sum(v));
```

The mean value $\int s \, dF_2(s)$ can now be calculated by

```
>> [d,s] = domain(F2.ends);
>> sum(s.*diff(F2))
ans = -1.77108680741657
```

and this result is correct to 12 digits. For information on this problem, see [5].

7 Discussion.

The algorithms and software system we have described are extraordinarily convenient, at least in their basic setting of linear problems with smooth solutions. They give users interactive access to one of the most powerful technologies for high-accuracy solutions of differential equations, namely spectral collocation. On the other hand, one must be aware that as a rule, these methods come without mathematical guarantees. The algorithms described in this article often converge to the correct solutions as the grid is refined, but this need not happen always, and so far as we know, there is no theory that readily identifies the dangerous cases.

Quite different from this convergence issue is the matter of loss of digits in spectral methods related to ill-conditioning of the associated matrices. A strategy for avoiding this problem, put forward by Greengard in 1991, is to take the highest order derivative of a variable as the basic unknown [7, 11, 13, 14]. For example, in a second-order differential equation involving the variable u, one can regard $v = u''$ as the unknown and reformulate the differential equation in integral form. This is a powerful idea, and in the chebop system such computations can be carried out with the use of the cumsum operator. Examples will be considered in another publication.

The methods we have described are automatic but not adaptive. By this we mean that grids are refined automatically to achieve a certain accuracy, but the refinement is global. There are applications where one would wish for local adaptivity, and we hope to investigate extensions of this kind in the future,

making use for example of the ideas of [17]. The current state of the art in adaptive spectral methods, however, is not very advanced. Adaptivity has been carried much further for finite difference and finite element methods, but these are usually of lower accuracy.

Finally we mention a fundamental algorithmic and conceptual question. Following the custom in the field of spectral methods, the algorithms we have described are based on square matrices. However, one might argue that since differentiation lowers the degree of a polynomial, a spectral differentiation matrix should have more columns than rows, and a spectral integration matrix should have more rows than columns. This way of thinking opens up a path towards making integration and differentiation more truly inverses of one another, and the matter is more than philosophical because of the crucial matter of boundary conditions and how they can be effectively imposed at the discrete level. We think it is possible that a system like the one we have described should best be realized via rectangular matrices, and we plan to investigate this idea in the future.

Chebfuns and chebops are freely available, together with users guides and other materials, at `http://www.comlab.ox.ac.uk/chebfun`.

Acknowledgements.

The chebop system was developed in the first half of 2008 during an exciting period of intense discussion and code development at Oxford with our chebfun coauthors and companions Ricardo Pachón and Rodrigo Platte. We also thank Lotti Ekert for support of many kinds, including initial development of the chebfun/chebop web site.

REFERENCES

1. M. Abramowitz and I. A. Stegun, eds., *Handbook of Mathematical Functions*, Dover, 1965.
2. E. Anderson, et al., *LAPACK User's Guide,* SIAM, 1999.
3. Z. Battles, *Numerical Linear Algebra for Continuous Functions,* DPhil thesis, Oxford University Computing Laboratory, 2006.
4. Z. Battles and L. N. Trefethen, *An extension of Matlab to continuous functions and operators,* SIAM J. Sci. Comp., 25 (2004), pp. 1743–1770.
5. F. Bornemann, *On the numerical evaluation of Fredholm determinants,* manuscript, 2008.
6. J. P. Boyd, *Chebyshev and Fourier Spectral Methods,* 2nd ed., Dover, 2001.
7. E. A. Coutsias, T. Hagstrom, and D. Torres, *An efficient spectral method for ordinary differential equations with rational function coefficients,* Math. Comp., 65 (1996), 611–635.
8. M. Dieng, *Distribution Functions for Edge Eigenvalues in Orthogonal and Symplectic Ensembles: Painlevé Representations*, PhD thesis, University of California, Davis, 2005.
9. B. Fornberg, *A Practical Guide to Pseudospectral Methods*, Cambridge, 1996.

10. J. R. Gilbert, C. Moler and R. Schreiber, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Matrix Anal. Applics., 13 (1992), pp. 333–356.
11. L. Greengard, *Spectral integration and two-point boundary value problems,* SIAM J. Numer. Anal., 28 (1991), 1071–1080.
12. R. B. Lehoucq, D. C. Sorensen and C. Yang, *ARPACK User's Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods,* SIAM, 1998.
13. N. Mai-Duy and R. I. Tanner, *A spectral collocation method based on integrated Chebyshev polynomials for two-dimensional biharmonic boundary-value problems,* J. Comp. Appl. Math., 201 (2007), pp. 30–47.
14. B. Muite, *A comparison of Chebyshev methods for solving fourth-order semilinear initial boundary value problems,* manuscript, 2007.
15. S. A. Orszag, *Accurate solution of the Orr–Sommerfeld equation,* J. Fluid Mech., 50 (1971), pp. 689–703.
16. R. Pachón, R. Platte and L. N. Trefethen, *Piecewise smooth chebfuns,* SIAM J. Sci. Comp., submitted.
17. T. W. Tee and L. N. Trefethen, *A rational spectral collocation method with adaptively determined grid points,* SIAM J. Sci. Comp., 28 (2006), pp. 1798–1811.
18. C. A. Tracy and H. Widom, *Level-spacing distributions and the Airy kernel,* Comm. Math. Phys., 159 (1994), 151–174.
19. L. N. Trefethen, *Spectral Methods in* MATLAB, SIAM, Philadelphia, 2000.
20. L. N. Trefethen, *Computing numerically with functions instead of numbers,* Math. in Comp. Sci., 1 (2007), pp. 9–19.