

A MATLAB Differentiation Matrix Suite

J. A. C. WEIDEMAN

University of Stellenbosch

and

S. C. REDDY

Oregon State University

A software suite consisting of 17 MATLAB functions for solving differential equations by the spectral collocation (i.e., pseudospectral) method is presented. It includes functions for computing derivatives of arbitrary order corresponding to Chebyshev, Hermite, Laguerre, Fourier, and sinc interpolants. Auxiliary functions are included for incorporating boundary conditions, performing interpolation using barycentric formulas, and computing roots of orthogonal polynomials. It is demonstrated how to use the package for solving eigenvalue, boundary value, and initial value problems arising in the fields of special functions, quantum mechanics, nonlinear waves, and hydrodynamic stability.

Categories and Subject Descriptors: G.1.7 [Numerical Analysis]: Ordinary Differential Equations—*Boundary value problems*; G.1.8 [Numerical Analysis]: Partial Differential Equations—*Spectral methods*

General Terms: Algorithms

Additional Key Words and Phrases: MATLAB, spectral collocation methods, pseudospectral methods, differentiation matrices

1. INTRODUCTION

This paper is about the confluence of two powerful ideas, both developed in the last two or three decades. The first is the concept of a differentiation matrix that has proven to be a very useful tool in the numerical solution of differential equations [Canuto et al. 1988; Fornberg 1996]. The second is the matrix-based approach to scientific computing that was introduced in the MATLAB (Matrix Laboratory) software package [The MathWorks

Work by J.A.C. Weideman was supported in part by NSF grant DMS-9404599.

Authors' addresses: J. A. C. Weideman, Department of Applied Mathematics, University of Stellenbosch, Private Bag XI, Matieland, 7602, South Africa; email: weideman@na-net.ornl.gov; S. C. Reddy, Department of Mathematics, Oregon State University, Corvallis, OR 97331; email: reddy@math.orst.edu.

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

© 2001 ACM 0098-3500/00/1200-0465 \$5.00

Table I. Examples

Example	Domain	Problem Type	Solution Procedure	Application
Error Function	$[0, \infty)$	Boundary value	Chebyshev	Probability
Mathieu	periodic	Eigenvalue	Fourier	Dynamical Systems
Schrödinger	$[0, \infty)$	Eigenvalue	Laguerre	Quantum Mechanics
Sine-Gordon	$(-\infty, \infty)$	Evolution	Fourier, Hermite, sinc	Nonlinear Waves
Orr-Sommerfeld	$[-1, 1]$	Eigenvalue	Chebyshev	Fluid Mechanics

1998]. The basic unit in the MATLAB programming language is the matrix, and this makes MATLAB the ideal tool for working with differentiation matrices.

Differentiation matrices are derived from the spectral collocation (also known as pseudospectral) method for solving differential equations of boundary value type. This method is discussed in some detail below but for more complete descriptions we refer to Canuto et al. [1988], Fornberg [1996], Funaro [1992], and Gottlieb et al. [1984]. In the spectral collocation method the unknown solution to the differential equation is expanded as a global interpolant, such as a trigonometric or polynomial interpolant. In other methods, such as finite elements or finite differences, the underlying expansion involves local interpolants such as piecewise polynomials. In practice this means that the accuracy of the spectral method is superior: for problems with smooth solutions convergence rates of $O(e^{-cN})$ or $O(e^{-c\sqrt{N}})$ are routinely achieved, where N is the number of degrees of freedom in the expansion [Canuto et al. 1988; Stenger 1993; Tadmor 1986]. In contrast, finite elements or finite differences yield convergence rates that are only algebraic in N , typically $O(N^{-2})$ or $O(N^{-4})$.

There is, however, a price to be paid for using a spectral method instead of a finite element or a finite difference method: full matrices replace sparse matrices; stability restrictions may become more severe; and computer implementations, particularly for problems posed on irregular domains, may not be straightforward. Nevertheless, provided the solution is smooth the rapid convergence of the spectral method often compensates for these shortcomings.

The Differentiation Matrix Suite introduced here consists of 17 MATLAB functions, summarized in the Appendix, that enable the user to generate spectral differentiation matrices based on Chebyshev, Fourier, Hermite, and other interpolants. These functions enable one to solve, with just a few lines of additional code, a variety of problems in scientific computation. We picked the five important differential equations listed in Table I as examples. Each of these problems is solved in tutorial-style to illustrate the usage of our codes.

To introduce the idea of a differentiation matrix we recall that the spectral collocation method for solving differential equations is based on weighted interpolants of the form [Canuto et al. 1988; Fornberg 1996; Welfert 1997]

$$f(x) \approx p_{N-1}(x) = \sum_{j=1}^N \frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) f_j. \quad (1)$$

Here $\{x_j\}_{j=1}^N$ is a set of distinct interpolation nodes; $\alpha(x)$ is a weight function; $f_j = f(x_j)$; and the set of interpolating functions $\{\phi_j(x)\}_{j=1}^N$ satisfies $\phi_j(x_k) = \delta_{jk}$ (the Kronecker delta). This means that $p_{N-1}(x)$ defined by (1) is an interpolant of the function $f(x)$ in the sense that

$$f(x_k) = p_{N-1}(x_k), \quad k = 1, \dots, N.$$

A list of commonly used nodes, weights, and interpolating functions are tabulated in Section 3 below. These include the Chebyshev, Hermite, and Laguerre expansions, in which case the interpolating functions $\{\phi_j(x)\}$ are polynomials of degree $N - 1$.¹ Two well-known nonpolynomial cases are also included in our list, namely the trigonometric (Fourier) and sinc (cardinal) interpolants.

Associated with an interpolant such as (1) is the concept of a collocation derivative operator. This operator is generated by taking ℓ derivatives of (1) and evaluating the result at the nodes $\{x_k\}$:

$$f^{(\ell)}(x_k) \approx \sum_{j=1}^N \frac{d^\ell}{dx^\ell} \left[\frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) \right]_{x=x_k} f_j, \quad k = 1, \dots, N.$$

The derivative operator may be represented by a matrix $D^{(\ell)}$, the differentiation matrix, with entries

$$D_{k,j}^{(\ell)} = \frac{d^\ell}{dx^\ell} \left[\frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) \right]_{x=x_k}. \quad (2)$$

The numerical differentiation process may therefore be performed as the matrix-vector product

$$\mathbf{f}^{(\ell)} = D^{(\ell)} \mathbf{f}, \quad (3)$$

where \mathbf{f} (resp. $\mathbf{f}^{(\ell)}$) is the vector of function values (resp. approximate derivative values) at the nodes $\{x_k\}$.

When solving differential equations, the derivatives are approximated by the discrete derivative operators (3). A linear two-point boundary value problem may thus be converted to a linear system. A differential eigenvalue problem may likewise be converted to a matrix eigenvalue problem. Solving

¹In the standard notation, one considers interpolating polynomials of degree N and sums, as in (1), to have lower limit $j = 0$ and upper limit N . Since MATLAB does not have a zero index we begin sums with $j = 1$, and consequently our notation will involve polynomials of degree $N - 1$.

linear systems and computing the eigenvalues of matrices are one-line commands in MATLAB, involving the backslash operator and the function `eig` respectively. Examples of the procedure are given in Sections 4 and 5.

After solving the matrix problem, approximations to the function values at the nodes become available. It is often necessary to compute approximations at arbitrary points in the domain, however, so some form of interpolation is required. MATLAB has a built-in function `polyfit.m` for polynomial interpolation, but this function is intended for data fitting and may not be the most efficient for the calculations we have in mind. For the same reason the direct calculation of the interpolant (1) is not recommended. In most cases the interpolant may be expressed in the so-called barycentric form, which allows a more efficient implementation, and will therefore be used here. For a discussion of barycentric interpolation formulas, and in particular their superior stability properties, we refer to Henrici [1982, Sect. 5.4] (polynomial) and to Henrici [1986, Sect. 13.6] (trigonometric).

The software suite introduced here consists of a set of MATLAB functions that enable the user to compute differentiation matrices $D^{(\ell)}$, plus associated nodes $\{x_k\}$, for (a) all the important special cases (Chebyshev, Legendre, Laguerre, Hermite, Fourier, sinc), and (b) an arbitrary number of derivatives $\ell = 1, \dots, M$. Auxiliary codes include functions for computing the roots of some orthogonal polynomials (Legendre, Laguerre, Hermite), as well as barycentric interpolation formulas (Chebyshev, Fourier), plus functions for implementing special boundary conditions. A summary of all the functions in the suite is given in the Appendix.

This paper emphasizes the matrix-based implementation of the spectral collocation method. It is well known, however, that certain methods—Fourier, Chebyshev, sinc—can also be implemented by using the Fast Fourier Transform (FFT). By applying the FFT technique the matrix-vector product (3) can be computed in $O(N \log N)$ operations rather than the $O(N^2)$ operations that the direct computation of such a product requires. There are, however, situations where one might prefer the matrix approach in spite of its inferior asymptotic operation count.

First, for small values of N the matrix approach is in fact faster than the FFT implementation. Also, for the FFT to be optimally efficient N has to be a power of 2; otherwise the matrix approach may not be much slower in practice even for large N . Second, the FFT approach places a limitation on the type of algorithm that can be used for solving the linear system or eigenvalue problem that arises after discretization of the differential equation. Only iterative algorithms based on matrix-vector products, such as conjugate gradients or GMRES for linear systems and the Lanczos or Arnoldi iterations for eigenvalues, can be used. These are not built-in codes in MATLAB, so users will have to supply their own. (For these reasons the solution procedures sketched in Section 5 apply only to the matrix approach.)

In spite of the advantages of the matrix-based approach in the MATLAB setting, we have included three transform-based functions in our suite for

educational purposes and for the sake of completeness. These functions correspond to the Fourier, Chebyshev, and sinc methods, which are all based on the FFT. (We point out that asymptotically fast algorithms for polynomial interpolation and differentiation at arbitrary points have been proposed in Dutt et al. [1996]. These algorithms are not based on the FFT, and have not been included in our suite.)

A few comments about our MATLAB coding style are in order. It is well known that efficient coding in MATLAB means vectorization, and the use of built-in (compiled) functions wherever possible. Conditionals and loops, particularly nested loops, are to be avoided. We have tried to adhere to this guideline, even if it meant a more cryptic code. To compensate for this, we made an effort to elucidate the logic of our codes in the text of this paper.

The execution times of MATLAB functions need not be proportional to the number of floating-point operations performed. A code with an optimal operation count may be slow in practice due to the reasons given above. When faced with this situation, we chose the implementation which executes quicker even if it meant a higher operation count.

Many of the functions in the suite will not run in versions of MATLAB older than Version 5. This is primarily because of two reasons. First, Version 5 is the first version of MATLAB that allows arrays with more than two indices. Our functions typically generate arrays of dimension $N \times N \times M$ where the third index is reserved for the order of the derivative. Second, MATLAB 5 has a variable of type `logical`, not found in earlier versions, which is utilized in several of the codes.

We are aware of two other general software packages for spectral computations, both in Fortran. The first is Funaro [1993], and the second is PseudoPack 2000 [Costa and Don 1999]. Funaro's package computes first and second derivatives and has support for general Jacobi polynomials, many quadrature formulas, and routines for computing expansion coefficients. PseudoPack can compute up to fourth-order Fourier, Chebyshev, and Legendre collocation derivatives. Additional features include routines for filtering, coordinate mapping, and differentiation of functions of two and three variables. The listings of various Fortran programs may be found in Canuto et al. [1988] and Fornberg [1996]. Examples of spectral computations using MATLAB are also given in Trefethen [2000], where the reader will find many applications that complement the ones listed in Table I.

Concerning higher derivatives, we remark that often the second- and higher-derivative matrices are equal to the first-derivative matrix raised to the appropriate power. But this is not always the case—for a sufficient condition, involving the weight function $\alpha(x)$, we refer to Welfert [1997]. Even when this is the case, it is not a good idea to compute higher-derivative matrices by computing powers of the first-derivative matrix. The computation of powers of a full matrix requires $O(N^3)$ operations, compared to the $O(N^2)$ for the recursive algorithm described in the next section. Not only is this recursion faster, it also introduces less roundoff error compared to the computation of matrix powers.

We should point out that we are not advocating the use of the spectral collocation method as a panacea for solving all differential equations. First, we note that there are alternatives to the collocation approach, most notably the tau and Galerkin methods. But the collocation method is typically easier to implement, particularly for nonconstant coefficient or nonlinear problems. (For a comparison of the collocation, Galerkin, and tau methods we refer to Fornberg [1996].) Second, in some applications it is advantageous to convert differential equations into integral equations which are then solved numerically; for example see Greengard [1991] and Greengard and Rokhlin [1991]. The suite presented here does not make provision for this approach. Third, all spectral methods (be it collocation, tau, or Galerkin) are known to suffer from certain stability problems [Weideman and Trefethen 1988], and they are not easily adaptable to irregular domains [Canuto et al. 1988]. Our suite does not circumvent any of these problems. But as was pointed out above, provided the solution is sufficiently smooth the superior convergence rate of the spectral method compensates for its defects.

The primary attraction of the software suite introduced here is the ease of use of the MATLAB functions. The modular nature of these functions enables the user to combine them in a plug-and-play manner to solve a variety of problems. Moreover, the MATLAB environment allows interactive access to linear algebra routines, ODE solvers, and graphics. Taking into account the time required to write and verify computer code, we believe that the methodology presented here is very attractive.

This software suite should also be useful for educational purposes. It could be used for a course on spectral methods or any other course where differential equations are to be solved.

The outline of the paper is as follows. In Section 2 we review the main algorithm on which most of the functions for computing differentiation matrices are based. Section 3 is a summary of the formulas used in our codes. Section 4 incorporates boundary conditions. The examples are presented in Section 5.

The codes are available at <http://ucs.orst.edu/~weidemaj/differ.html> and <http://www.mathworks.com/support/ftp/diffeqv5.shtml>. The second site is the *Differential Equations* category of the Mathworks user contributed (MATLAB 5) M-file repository.

2. AN ALGORITHM FOR POLYNOMIAL DIFFERENTIATION

In this section we consider the important special case in which the set of interpolating functions $\{\phi_j(x)\}$ consists of polynomials of degree $N - 1$. The two main functions in our suite, `poldif.m` and `chebdif.m`, deal with this situation. The former function computes differentiation matrices for arbitrary sets of points and weights; the latter function is restricted to Chebyshev nodes and constant weights.

The computation of spectral collocation differentiation matrices for derivatives of arbitrary order has been considered by Huang and Sloan [1994]

(constant weights) and Welfert [1997] (arbitrary $\alpha(x)$). The algorithm implemented in `poldif.m` and `chebdif.m` follows these references closely. At a certain point we had to adopt a different approach, however, which was necessitated by the rules of efficient programming in MATLAB.

Until further notice $\{x_j\}$ is a set of N distinct but otherwise arbitrary nodes. The interpolant (1) is given by

$$p_{N-1}(x) = \sum_{j=1}^N \frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) f_j,$$

where $\{\phi_j(x)\}$ are the Lagrangian interpolating polynomials defined by

$$\phi_j(x) = \prod_{\substack{m=1 \\ m \neq j}}^N \left(\frac{x - x_m}{x_j - x_m} \right), \quad j = 1, \dots, N.$$

The function $\alpha(x)$ is an arbitrary, positive weight, with at least M continuous derivatives. The differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$, that we wish to compute are defined by (2).

We distinguish between the computation of the diagonal and the off-diagonal entries of $D^{(\ell)}$, starting with the latter. The following recursion for $k \neq j$ was derived in Welfert [1997]:

$$D_{k,j}^{(\ell)} = \frac{\ell}{x_k - x_j} \left(\frac{c_k}{c_j} D_{k,k}^{(\ell-1)} - D_{k,j}^{(\ell-1)} \right), \quad \ell = 1, \dots, M. \quad (4)$$

Here $D^{(0)}$ is the identity matrix, and the constants c_j are defined by

$$c_j = \alpha(x_j) \prod_{\substack{m=1 \\ m \neq j}}^N (x_j - x_m), \quad j = 1, \dots, N. \quad (5)$$

There are two approaches to computing the diagonal entries, one being the recursive procedure suggested in Welfert [1997]. The alternative is to note that the diagonal entries are uniquely determined by the off-diagonal entries. In particular, the differentiation matrices should at least differentiate the weight function $\alpha(x)$ perfectly (in exact arithmetic, that is). Therefore

$$D^{(\ell)} \alpha = \alpha^{(\ell)},$$

where α is the vector of function values $\alpha(x_k)$, and $\alpha^{(\ell)}$ is the vector of values $\alpha^{(\ell)}(x_k)$. Once the off-diagonal entries have been computed by (4), everything in this equation is known but the main diagonal of $D^{(\ell)}$. Hence the diagonal entries may be solved for.

This direct approach to computing the diagonals of Chebyshev differentiation matrices was recommended in Baltensperger and Berrut [1999] and Bayliss et al. [1994], and therefore we have used it in our function `chebdif.m`. This approach is ill-conditioned, however, when $\alpha(x)$ is a function for which $\max_k \alpha(x_k) / \min_k \alpha(x_k)$ is large. This occurs, for example, in the Hermite case ($\alpha(x) = e^{-x^2/2}$), as well as the Laguerre case ($\alpha(x) = e^{-x/2}$). We have therefore used the recursive procedure of Welfert [1997] in our more general code `poldif.m`.

The details of the recursion are presented here for the first diagonal entry $D_{1,1}^{(\ell)}$. (Below we shall generalize to arbitrary $D_{j,j}^{(\ell)}$.) Define $\psi_1(x)$, \dots , $\psi_N(x)$ by

$$\psi_1(x) = \frac{\alpha(x)}{\alpha(x_1)}$$

$$\psi_n(x) = \left(\frac{x - x_n}{x_1 - x_n} \right) \psi_{n-1}(x), \quad n = 2, \dots, N. \quad (6)$$

One observes that $\psi_N(x)$ is the interpolating function $(\alpha(x)/\alpha(x_1))\phi_1(x)$. In general $\psi_n(x)$ is the interpolant for the nodes $\{x_k\}_{k=1}^n$, since $\psi_n(x_1) = 1$ and $\psi_n(x_k) = 0$ for $k = 2, \dots, n$.

By taking ℓ derivatives of (6), and using Leibniz's rule for the higher derivatives of products, one obtains

$$\psi_n^{(\ell)}(x_1) = \frac{\ell}{x_1 - x_n} \psi_{n-1}^{(\ell-1)}(x_1) + \psi_{n-1}^{(\ell)}(x_1), \quad n = 2, \dots, N, \quad \ell = 1, \dots, M. \quad (7)$$

The Fortran code given in Welfert [1997] implements this recursion for computing the diagonal entries $\psi_N^{(\ell)}(x_1)$, $\ell = 1, \dots, M$. This code, with its three nested loops, cannot be efficiently implemented in MATLAB, and we considered it necessary to vectorize it.

To this end, apply (7) to the term $\psi_{n-1}^{(\ell)}(x)$ in (7), and continue this procedure. Noting that $\psi_1^{(\ell)}(x_1) = \alpha^{(\ell)}(x_1)/\alpha(x_1)$, one obtains

$$\psi_n^{(\ell)}(x_1) = \frac{\alpha^{(\ell)}(x_1)}{\alpha(x_1)} + \ell \sum_{m=2}^n \frac{\psi_{m-1}^{(\ell-1)}(x_1)}{x_1 - x_m}.$$

For an arbitrary diagonal entry (j, j) , this generalizes to

$$\psi_{n,j}^{(\ell)}(x_j) = \beta_j^{(\ell)} + \ell \sum_{\substack{m=1 \\ m \neq j}}^n \frac{\psi_{m-1,j}^{(\ell-1)}(x_j)}{x_j - x_m}, \quad j, n = 1, \dots, N, \quad \ell = 1, \dots, M, \quad (8)$$

where we have defined

$$\beta_j^{(\ell)} = \frac{\alpha^{(\ell)}(x_j)}{\alpha(x_j)}, \quad j = 1, \dots, N, \quad \ell = 1, \dots, M. \quad (9)$$

The quantities $\psi_{N,j}^{(\ell)}(x_j)$ are the (j, j) diagonal entries of $D^{(\ell)}$, $\ell = 1, \dots, M$, that we wish to compute.

We are now in a position to describe the function `poldif.m` that implements the two recursions (4) and (8). Define the $N \times N$ matrix Z by

$$Z_{k,j} = \begin{cases} \frac{1}{x_k - x_j} & k \neq j \\ 0 & k = j, \quad k, j = 1, \dots, N, \end{cases}$$

and the two $(N - 1) \times N$ matrices $Y^{(\ell)}$ and X by

$$Y_{k,j}^{(\ell)} = \psi_{k,j}^{(\ell)}(x_j), \quad k = 1, \dots, N - 1, \quad j = 1, \dots, N,$$

and

$$X_{k,j} = \begin{cases} \frac{1}{x_j - x_{k+1}} & k \geq j \\ \frac{1}{x_j - x_k} & k < j, \quad k = 1, \dots, N - 1, \quad j = 1, \dots, N. \end{cases}$$

(Observe that X is the same as Z^T , except that the zero diagonal entry of each column of Z^T has been removed.) Also define the $M \times N$ matrix B by

$$B_{\ell,j} = \beta_j^{(\ell)}, \quad j = 1, \dots, N, \quad \ell = 1, \dots, M,$$

where the latter quantity is given by (9).

With Y initialized to an $(N - 1) \times N$ matrix of 1's, the following MATLAB recursion implements the formula (8) for the diagonal entries:

```
Y = cumsum([B(ell,:), ell*Y(1:N-1,:).*X])
```

A built-in function in MATLAB, `cumsum`, computes cumulative sums over the columns of the (matrix) argument. The dot multiplication `.*` is the symbol for componentwise multiplication of two matrices in MATLAB.

The recursion for the off-diagonal entries, Eq. (4), may be implemented as

```
D = ell*Z.*(C.*(repmat(diag(D),1,N)) - D)
```

where D is initialized to the $N \times N$ identity matrix, and C is the $N \times N$ matrix with entries

$$C_{k,j} = \frac{c_k}{c_j}, \quad k, j = 1, \dots, N.$$

Table II. Main Loop of `poldif.m`. The variables are defined in the text.

```

for ell = 1:M
    Y = cumsum([B(ell,:); ell*Y(1:N-1,:).*X]); % Recursion for diagonals
    D = ell*Z.*(C.*(repmat(diag(D),1,N) - D)); % Recursion for off-diagonals
    D(L) = Y(N,:); % Correct the diagonal of D
    DM(:,ell) = D; % Store current D in DM
end

```

In the above code `repmat` is a MATLAB function used for replicating a matrix. The command `repmat(diag(D),1,N)` creates an $N \times N$ matrix with the first diagonal entry of D on the first row, the second diagonal entry of D on the next row, etc.

The main loop of `poldif.m` is reproduced in Table II. Observe that in deriving (8), and by exploiting MATLAB's vector capabilities, we have reduced the generation of the differentiation matrices to a single for loop. The variable `DM` is an $N \times N \times M$ matrix that is used to store the differentiation matrices $D^{(1)}, \dots, D^{(M)}$. `L` is the logical identity matrix of order $N \times N$.

The usage of the function `poldif.m` is described in Section 3.1. Its computational efficiency, which involves questions of operation count and numerical stability, may be summarized as follows.

We follow the convention that each floating-point addition and each floating-point multiplication count as one flop. Each call to `cumsum` involves N^2 flops, and thus each update for the diagonal entries requires about $3N^2$ flops. Each update for the off-diagonal entries involves precisely $4N^2$ flops. Thus the loop shown in Table II executes a total of about $7MN^2$ flops. There is also an overhead cost for computing the matrices Z and C . Since the matrix Z is skew-symmetric, it suffices to compute the upper (or lower) triangular part for a total operation count of about N^2 . It turns out, however, to be faster in MATLAB to compute the full matrix, for an operation count of about $2N^2$. The computation of the quantities c_j requires about N^2 multiplications, and the computation of the matrix C requires an additional N^2 divisions. The overall operation count of `poldif.m` is therefore roughly $(7M + 4)N^2$.

As for the stability of `poldif.m` and `chebdif.m`, we have conducted the following investigation. We used the function `chebdif.m` to construct differentiation matrices $D^{(\ell)}$ on Chebyshev points. This was done in MATLAB's double-precision arithmetic (with machine epsilon $\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$). We then computed the same matrices in quadruple precision in Fortran using the algorithm of Welfert [1997]. The function `poldif.m` was tested in a similar manner, by computing Hermite and Laguerre differentiation matrices as explained below in Sections 3.3 and 3.4.

Table III. The table shows the values of d , rounded to the nearest integer, where Rel. Err. = 10^{-d} and the relative error is defined by (10).

Chebyshev					Hermite				
N	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$	N	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
8	16	15	15	14	8	14	15	14	15
16	16	15	15	15	16	14	14	14	14
32	16	16	15	14	32	14	14	14	14
64	16	15	15	14	64	13	14	13	14

To measure the error, we computed the relative error in the Frobenius norm,

$$\text{Rel. Err.} = \frac{\|\tilde{D}^{(\ell)} - D^{(\ell)}\|_F}{\|\tilde{D}^{(\ell)}\|_F}, \quad \ell = 1, \dots, 4, \quad (10)$$

where the tilde refers to the matrices computed in quadruple precision. The results are summarized in Table III for the Chebyshev and Hermite differentiation matrices. (The Laguerre results were similar to the Hermite case and are not displayed here.)

The results of Table III confirm that `chebdif.m` and `poldif.m` represent stable algorithms for computing differentiation matrices. It should be kept in mind, however, that the application of these matrices to data vectors can be numerically unstable, particularly for higher derivatives [Breuer and Everson 1992; Don and Solomonoff 1994; Fornberg 1996].

We conclude this section by discussing the computation of the nodes $\{x_k\}$. In some cases (such as the set of Chebyshev points defined in Eq. (13) below) explicit formulas are available, but this is the exception rather than the rule. We have included three MATLAB functions in our suite for computing the zeros of the Legendre, Laguerre, and Hermite polynomials (called `legroots.m`, `lagroots.m`, and `herroots.m` respectively).

The basis of these three functions is the three-term recurrence relation

$$q_{n+1}(x) = (x - \alpha_n)q_n(x) - \beta_n^2 q_{n-1}(x), \quad n = 0, 1, 2, \dots \quad (11)$$

with

$$q_0(x) = 1, \quad q_{-1}(x) = 0.$$

It is well known that the roots of the orthogonal polynomial $q_N(x)$ are given by the eigenvalues of the $N \times N$ tridiagonal Jacobi matrix

$$J = \begin{pmatrix} \alpha_0 & \beta_1 & & & \\ \beta_1 & \alpha_1 & \beta_2 & & \\ & & \ddots & \beta_{N-1} & \\ & & & \beta_{N-1} & \alpha_{N-1} \end{pmatrix}.$$

The coefficients (α_n, β_n) are given in the table.

	Legendre	Laguerre	Hermite
α_n	0	$2n + 1$	0
β_n	$n / \sqrt{4n^2 - 1}$	n^2	$1/2n$

Using MATLAB's convenient syntax the Jacobi matrix can easily be generated. For example, in the Legendre case this requires no more than three lines of code:

```
>>n = [1:N-1];
>>b = n./sqrt(4*n.^2-1);
>>J = diag(b,1) + diag(b,-1);
```

Once J has been created MATLAB's built-in eig routine can be used to compute its eigenvalues:

```
>>r = eig(J);
```

The eig function employs the QR algorithm, as good a method as any for computing these roots. Our function legroots.m (and similarly lagroots.m and herroots.m) consists essentially of the above instructions. But it utilizes the fact that J is sparse, and it also sorts the roots in increasing order of magnitude.

The functions legroots.m, lagroots.m, and herroots.m may be used in conjunction with poldif.m to generate the corresponding differentiation matrices. The Laguerre and Hermite cases will be discussed in more detail in the next section. As for the Legendre case, assuming a constant weight the following two lines of code will generate first- and second-derivative matrices of order $N \times N$ on Legendre points

```
>> x = legroots(N);
>> D = poldif(x,2);
```

3. SUMMARY OF FORMULAS AND CODES

In this section we summarize the main formulas implemented in our MATLAB functions. We also discuss the calling command for each function.

We remark that many of the formulas listed below differ from the conventional notation seen in the literature. This is a consequence of the fact that array indices in MATLAB start at 1 and not 0 (see footnote 1). For example, the Chebyshev points listed in Eq. (13) below are more commonly denoted by $x_k = \cos(k\pi/N)$, $k = 0, \dots, N$.

3.1 Arbitrary Interval

Arbitrary Polynomial

MATLAB files: poldif.m, polint.m²

²Note that polint stands for polynomial *interpolation*, not polynomial *integration*. Likewise chebint and fourint discussed below refer to Chebyshev and Fourier interpolation.

Interval: Arbitrary

$$[a, b].$$

Nodes: Arbitrary, but distinct

$$x_1, x_2, \dots, x_N.$$

Weight Function: Arbitrary positive function, M times continuously differentiable

$$\alpha(x).$$

Interpolant:

$$p_{N-1}(x) = \sum_{j=1}^N \frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) f_j.$$

The $\{\phi_j(x)\}$ are given by Lagrange's formula

$$\phi_j(x) = \prod_{\substack{m=1 \\ m \neq j}}^N \left(\frac{x - x_m}{x_j - x_m} \right), \quad j = 1, \dots, N,$$

or equivalently,

$$\phi_j(x) = \frac{\omega_N(x)}{\omega'_N(x_j)(x - x_j)},$$

where

$$\omega_N(x) = \prod_{m=1}^N (x - x_m).$$

Barycentric Form of Interpolant [Henrici 1982, Sect. 5.4]: (Computed by `polint.m`.)

$$p_{N-1}(x) = \frac{\alpha(x) \sum_{j=1}^N \frac{w_j}{x - x_j} \frac{f_j}{\alpha(x_j)}}{\sum_{j=1}^N \frac{w_j}{x - x_j}}, \quad (12)$$

where

$$w_j^{-1} = \prod_{\substack{m=1 \\ m \neq j}}^N (x_j - x_m).$$

Differentiation Matrices: (Computed by `poldif.m`.)

$$D_{k,j}^{(\ell)} = \frac{d^\ell}{dx^\ell} \left[\frac{\alpha(x)}{\alpha(x_j)} \phi_j(x) \right]_{x=x_k}.$$

In general

$$D^{(\ell)} \neq (D^{(1)})^\ell.$$

Transform Formulas: See note (d) below.

Accuracy: No general error analysis applicable to arbitrary $\{x_k\}$ has been undertaken.

Notes:

- (a) Although `poldif.m` computes the spectral differentiation matrix for arbitrary nodes $\{x_k\}$, approximation theory dictates that the $\{x_k\}$ cannot be just any set of nodes. The best choices are the roots of orthogonal polynomials such as the Chebyshev, Laguerre, and Hermite polynomials discussed below.
- (b) For efficient execution in MATLAB, the barycentric interpolation formulas (12) and (15) below were coded as matrix-vector multiplications.
- (c) For each set of nodes $\{x_k\}$ the weights $\{w_j\}$ that appear in (12) may be computed once and for all. Our code `polint.m` does not make provision for this, however, and it is up to the user to incorporate this (trivial) modification if the interpolation is to be performed on many different occasions.
- (d) Asymptotically fast algorithms for polynomial interpolation, differentiation, and integration have been suggested in Dutt et al. [1996]. Based on the fast multipole method, these algorithms evaluate a polynomial of degree N at N arbitrary points in $O(N \log \epsilon)$ operations. Here ϵ is a user-specified tolerance, for unlike the Fast Chebyshev Transform discussed below, the multipole-based algorithms do not represent the underlying polynomial exactly (disregarding round-off error) but only to within an error ϵ . These algorithms have not been included in our suite.

Calling Commands:

- (a) The code `poldif.m` implements the algorithm of Section 2. Its calling command is

```
>>D = poldif(x, malpha, beta);
```

The input parameter `x` is a vector of length N containing the set of distinct nodes. The parameter `malpha` could be:

- (i) An integer M , which is the highest derivative required. In this case a constant weight function $\alpha(x)$ is assumed, and the input parameter `beta` is omitted.

- (ii) A vector of length N , containing the values of the weight function sampled at the nodes, i.e., $\alpha_k = \alpha(x_k)$. The parameter `beta` is then an $M \times N$ array containing the quantities $\beta_k^{(\ell)}$ defined by (9).

In both cases (i) and (ii) it is assumed that $0 < M < N - 1$. On output `D` is an $N \times N \times M$ array containing the differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$.

- (b) The code `polint.m` implements the barycentric formula (12). Its calling command is either

```
>>p = polint(xk, fk, x);
```

when a constant weight is assumed, or

```
>>p = polint(xk, fk, x, alphaxk, alphax);
```

in the case of a nonconstant $\alpha(x)$. In both cases the input vectors `xk`, `fk` are the coordinates $(x_k, f(x_k))$, $k = 1, \dots, N$. The vector `x`, of arbitrary length, contains the ordinates where the interpolant is to be evaluated. The vectors `alphaxk` and `alphax` are the values of the weight function sampled at `xk` and `x` respectively. On output the vector `p` contains the corresponding values of the interpolant $p_{N-1}(x)$ as computed by formula (12).

3.2 Bounded Interval

Chebyshev

MATLAB files: `chebdif.m`, `chebint.m`

Interval:

$$[-1, 1].$$

Nodes: (Computed by `chebdif.m`)

$$x_k = \cos\left(\frac{(k-1)\pi}{N-1}\right), \quad k = 1, \dots, N. \quad (13)$$

(These are the Chebyshev points of the second kind, or equivalently, the extreme points on $[-1, 1]$ of $T_{N-1}(x)$, the Chebyshev polynomial of degree $N - 1$.)

Weight Function:

$$\alpha(x) = 1.$$

Interpolant [Canuto et al. 1988, p. 69]:

$$p_{N-1}(x) = \sum_{j=1}^N \phi_j(x) f_j,$$

where

$$\phi_j(x) = \frac{(-1)^j}{c_j} \frac{1 - x^2}{(N - 1)^2} \frac{T'_{N-1}(x)}{x - x_j}. \quad (14)$$

Here $c_1 = c_N = 2$ and $c_2 = \dots = c_{N-1} = 1$. (These constants are not the same as the c_j defined in Eq. (5); they differ by a factor $(-1)^j(N - 1)/2^{N-2}$.)

Barycentric Form of Interpolant [Henrici 1982, p. 252]: (Computed by `chebint.m`.)

$$p_{N-1}(x) = \frac{\sum_{j=1}^N \frac{(-1)^j f_j}{c_j(x - x_j)}}{\sum_{j=1}^N \frac{(-1)^j}{c_j(x - x_j)}}. \quad (15)$$

Differentiation Matrices [Canuto et al. 1988, p. 69]: (Computed by `cheb-dif.m`.)

$$D_{k,j}^{(1)} = \begin{cases} \frac{c_k(-1)^{j+k}}{c_j(x_k - x_j)} & j \neq k \\ -\frac{1}{2} \frac{x_k}{(1 - x_k^2)} & j = k \neq 1, N \\ \frac{2(N-1)^2 + 1}{6} & j = k = 1 \\ -\frac{2(N-1)^2 + 1}{6} & j = k = N. \end{cases}$$

$$D^{(\ell)} = (D^{(1)})^\ell, \quad \ell = 1, 2, \dots$$

Transform Formulas [Canuto et al. 1988, p. 68]: (Implemented by `cheb-difft.m`.)

$$p_{N-1}(x_k) = \sum_{j=0}^{N-1} a_j T_j(x_k) \Rightarrow p'_{N-1}(x_k) = \sum_{j=0}^{N-1} b_j T_j(x_k), \quad (16)$$

where

$$b_{N-1} = 0, \quad b_{N-2} = 2(N-1)a_{N-1}, \quad b_0 = \frac{1}{2}b_2 + a_1,$$

and

$$b_j = b_{j+2} + 2(j+1)a_{j+1}, \quad j = N-3, \dots, 1.$$

(Apply repeatedly for derivatives of higher order.)

Accuracy: For an error analysis, see Tadmor [1986].

Notes:

- (a) The canonical interval is $[-1, 1]$. If the differential equation is posed on $[a, b]$ it should first be converted to $[-1, 1]$ through the change of variables $x \longleftrightarrow (1/2)((b - a)x + (b + a))$.
- (b) Our suite contains two functions for Chebyshev differencing: `cheb-dif.m` for computing differentiation matrices, and `chebdifft.m` for computing derivatives using the FFT. Some implementation details of these two codes are as follows.
- (c) Making use of the identity $\cos\theta = \sin((\pi/2) - \theta)$ the Chebyshev nodes (13) may be expressed as

$$x_k = \sin\left(\frac{\pi(N + 1 - 2k)}{2(N - 1)}\right), \quad k = 1, \dots, N,$$

which is the formula implemented in `chebdif.m`. This formula has the advantage that in floating-point arithmetic it yields nodes that are perfectly symmetric about the origin, which is not the case for (13).

- (d) The differences $x_k - x_j$ that appear in the differentiation matrices may be subject to floating-point cancellation errors for large N . The computation of these differences may be avoided by the use of the trigonometric identity [Don and Solomonoff 1994]:

$$\cos\left(\frac{(k - 1)\pi}{N - 1}\right) - \cos\left(\frac{(j - 1)\pi}{N - 1}\right) = 2\sin\left(\frac{\pi(k + j)}{2(N - 1)}\right)\sin\left(\frac{\pi(k - j)}{2(N - 1)}\right).$$

An additional complication is the fact that $\sin\theta$ can be computed to high relative accuracy when $\theta \approx 0$, but $\sin(\pi - \theta)$ cannot. The recommended remedy, referred to as the “flipping trick” in our codes, is to compute only the top half of the differentiation matrix and to obtain the lower half using symmetry relations; see Don and Solomonoff [1994].

- (e) The function `chebdif.m` implements the algorithm discussed in Section 2. The modifications mentioned under points (c) and (d) above have been incorporated into the code.
- (f) The function `chebdifft.m` implements the transform formulas (16), by using the FFT. The discrete Chebyshev coefficients $\{a_j\}$ are given by

$$a_j = \frac{2}{(N - 1)c_{j+1}} \sum_{k=1}^N \frac{f_k}{c_k} \cos \frac{j\pi(k-1)}{N-1}, \quad j = 0, \dots, N - 1,$$

where the c_j have been defined below (14). The values $\{f_k\}$ can be recovered using the discrete inverse Chebyshev transform:

$$f_k = \sum_{j=0}^{N-1} a_j \cos \frac{\pi j(k-1)}{N-1}, \quad k = 1, \dots, N.$$

It can be shown that

$$a_j = \frac{1}{(N-1)c_{j+1}} \sum_{k=1}^{2(N-1)} \tilde{f}_k e^{2i\pi j(k-1)/2(N-1)}, \quad j = 0, \dots, N-1,$$

where

$$\{\tilde{f}_k\}_{k=1}^{2(N-1)} = \{f_1, f_2, \dots, f_{N-1}, f_N, f_{N-1}, \dots, f_2\},$$

and

$$f_k = \frac{1}{2} \sum_{j=0}^{2N-3} \tilde{a}_j e^{2i\pi j(k-1)/2(N-1)}, \quad k = 1, \dots, N,$$

where

$$\{\tilde{a}_j\}_{j=0}^{2N-3} = \{2a_0, a_1, \dots, a_{N-2}, 2a_{N-1}, a_{N-2}, \dots, a_1\}.$$

Both of these sums can be computed in $O(N \log N)$ operations using the FFT.

Calling Commands:

- (a) The calling command for `chebdif.m` is

```
>>[x, D] = chebdif(N, M);
```

On input the integer N is the size of the required differentiation matrices, and the integer M is the highest derivative needed. On output the vector x , of length N , contains the Chebyshev points (13), and D is an $N \times N \times M$ array containing differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$. It is assumed that $0 < M \leq N - 1$.

- (b) The calling command for `chebint.m` is

```
>>p = chebint(f, x);
```

On input the vector f , of length N , contains the values of the function $f(x)$ at the Chebyshev points (13). The vector x , of arbitrary length, contains the ordinates where the interpolant is to be evaluated. On output the vector p contains the corresponding values of the interpolant $p_{N-1}(x)$ as computed by the formula (15).

- (c) The calling command for `chebdifft.m` is

```
>>Dmf = chebdifft(f, M);
```

On input the vector f , of length N , contains the values of the function $f(x)$ at the Chebyshev points (13). M is the order of the required derivative. On output the vector Dmf contains the values of the M th derivative of $f(x)$ at the corresponding points.

3.3 Real Line

Hermite

MATLAB files: `herdif.m`, `herroots.m`

Interval:

$$(-\infty, \infty).$$

Nodes: (Computed by `herroots.m`.)

x_1, \dots, x_N are the roots of $H_N(x)$, the Hermite polynomial of degree N , indexed in ascending order. *Note:* $-x_1 = x_N = O(\sqrt{N})$ as $N \rightarrow \infty$; see Abramowitz and Stegun [1964, Ch. 22].

Weight Function:

$$\alpha(x) = e^{-x^2/2}.$$

Interpolant:

$$p_{N-1}(x) = \sum_{j=1}^N \frac{e^{-x^2/2}}{e^{-x_j^2/2}} \phi_j(x) f_j,$$

where

$$\phi_j(x) = \frac{H_N(x)}{H'_N(x_j)(x - x_j)}.$$

Barycentric Form of Interpolant: Similar to (12).

Differentiation Matrices: (Computed by `herdif.m`.) For a formula we refer to Funaro [1992, Ch. 7]. Note that

$$D^{(\ell)} \neq (D^{(1)})^\ell.$$

Transform Formulas: There exists no fast algorithm for Hermite expansions similar to the Fast Chebyshev Algorithm described in Section 3.2 above. The algorithm for arbitrary polynomials alluded to in Section 3.1 could be considered, but this has not been done here.

Accuracy: Has not been analyzed in general; for special cases see Boyd [1984; 1989] and Tang [1993].

Implementation Notes:

- (a) The real line $(-\infty, \infty)$ may be mapped to itself by the change of variable $x = b\tilde{x}$, where b is any positive real number. By the chain rule

$$\frac{df}{d\tilde{x}} = b \frac{df}{dx}, \quad \frac{d^2f}{d\tilde{x}^2} = b^2 \frac{d^2f}{dx^2}, \quad \text{etc.} \quad (17)$$

One observes that the first-derivative matrix corresponding to $b = 1$ should be multiplied by b , the second-derivative matrix by b^2 , etc. At the same time the nodes are rescaled to x_k/b . It means that the Hermite differentiation process is exact for functions of the form

$$e^{-\frac{1}{2}b^2x^2}p(x)$$

where $p(x)$ is any polynomial of degree $N - 1$ or less (assuming exact arithmetic of course). The freedom offered by the parameter b may be exploited to optimize the accuracy of the Hermite differencing process; see Tang [1993].

- (b) To apply the algorithm of Section 2 to compute Hermite differentiation matrices, it is necessary to compute the quantities $\beta_j^{(\ell)}$ defined by (9). Using the three-term recurrence relation for the Hermite polynomials (11), plus the Rodriguez formula

$$H_n(x) = \frac{(-1)^n}{2^n} e^{x^2} \frac{d^n}{dx^n} e^{-x^2},$$

it is possible to show that for $j = 1, \dots, N$

$$\beta_j^{(\ell)} = -x_j \beta_j^{(\ell-1)} - (\ell - 1) \beta_j^{(\ell-2)}, \quad \ell = 1, \dots, M, \quad (18)$$

where $\beta_j^{(-1)} = 0$, $\beta_j^{(0)} = 1$. The factor $(\ell - 1)$ on the right is a sign that the recurrence is unstable, but in most applications M is no larger than 2, perhaps 4, so this need not be a concern.

- (c) `herdif.m` computes the Hermite points by calling `herroots.m`. It then computes the quantities $\beta_j^{(\ell)}$ via the recurrence (18). A call to `poldif.m` completes the computation of the differentiation matrices.
- (d) Our suite does not include a function `herint.m` for weighted barycentric interpolation at Hermite points. The function `polint.m` should be used for this purpose. (The same remarks apply to the Laguerre case below.)

Calling Commands:

- (a) The calling command for `herdif.m` is

```
>>[x, D] = herdif(N, M, b);
```

On input the integer N is the size of the required differentiation matrices, and the integer M is the highest derivative needed. The scalar b is the scaling parameter b defined by (17). On output the vector x , of length N , contains the Hermite points scaled by b . D is an $N \times N \times M$ array containing the differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$.

- (b) The calling command for `herroots.m` is

```
>>r = herroots(N);
```

The input integer N is the degree of the Hermite polynomial, and the output vector r contains its N roots.

Sinc

MATLAB files: `sincdif.m`

Interval:

$$(-\infty, \infty).$$

Nodes: (Computed by `sincdif.m`) Equidistant points with spacing h , symmetric with respect to the origin

$$x_k = \left(k - \frac{N+1}{2}\right)h, \quad k = 1, \dots, N. \quad (19)$$

Weight Function:

$$\alpha(x) = 1.$$

Interpolant [Stenger 1993]:

$$s_N(x) = \sum_{j=1}^N \phi_j(x) f_j,$$

where

$$\phi_j(x) = \frac{\sin(\pi(x - x_j)/h)}{\pi(x - x_j)/h}.$$

Barycentric Form of Interpolant: See Berrut [1989].

Differentiation Matrices [Stenger 1993]: (Computed by `sincdif.m`).

$$D^{(1)} = \frac{1}{h} \begin{pmatrix} 0 & 1 & -\frac{1}{2} & \cdots & \frac{(-1)^N}{N-1} \\ -1 & 0 & 1 & & \\ \frac{1}{2} & -1 & 0 & & -\frac{1}{2} \\ & & & \ddots & 1 \\ \frac{(-1)^{N-1}}{N-1} & \cdots & \frac{1}{2} & -1 & 0 \end{pmatrix}$$

$$D^{(2)} = \frac{1}{h^2} \begin{pmatrix} \frac{-\pi^2}{3} & 2 & -\frac{1}{2} & \cdots & \frac{2(-1)^N}{(N-1)^2} \\ 2 & \frac{-\pi^2}{3} & 2 & & \\ -\frac{1}{2} & 2 & \frac{-\pi^2}{3} & & -\frac{1}{2} \\ & & & \ddots & 2 \\ \frac{2(-1)^N}{(N-1)^2} & \cdots & -\frac{1}{2} & 2 & \frac{-\pi^2}{3} \end{pmatrix} \quad (20)$$

Note that all sinc derivative matrices are Toeplitz, i.e., constant along diagonals. In general

$$D^{(\ell)} \neq (D^{(1)})^\ell.$$

Transform Formulas: See note (d) below.

Accuracy: See Stenger [1993].

Notes:

- (a) Like the Hermite method and the Laguerre method discussed below, the sinc method contains a free parameter, namely the step size h . For optimal estimates of this parameter for certain classes of functions, see Stenger [1993]. The typical estimate is $h = C/\sqrt{N}$ for some C .
- (b) The sinc method, in its original form, is intended for solving problems on the real line $(-\infty, \infty)$. With the aid of the mapping functions introduced in Stenger [1993] the method may also be applied to the intervals $[0, \infty)$ and $[a, b]$. It should not be difficult to extend the present set of codes to do this.
- (c) Our suite contains two functions for sinc differencing: `sincdif.m` for computing differentiation matrices, and `sincdiff.ft.m` for computing derivatives using the FFT. Some implementation details of these two codes are as follows.
- (d) Since all sinc differentiation matrices are Toeplitz, they are determined uniquely by their first rows and columns. Moreover, matrices representing an even (resp. odd) derivative are symmetric (resp. skew-symmetric). It therefore suffices to generate the first columns of the differentiation matrices only. This can be done recursively, as follows. By making a change of variable $t = \pi x/h$ one gets

$$\frac{d^\ell}{dx^\ell} \frac{\sin(\pi x/h)}{\pi x/h} = \left(\frac{\pi}{h}\right)^\ell \sigma_\ell(t)$$

where

$$\sigma_\ell(t) = \frac{d^\ell}{dt^\ell} \frac{\sin t}{t}.$$

A recurrence relation for computing $\sigma_\ell(t)$ is given by Wimp [1984, p. 16]:

$$\sigma_\ell(t) = t^{-1}(-\ell \sigma_{\ell-1}(t) + \text{Im}(i^{\ell-1} e^{it})), \quad \ell = 1, \dots, M. \quad (21)$$

The first column of each differentiation matrix $D^{(\ell)}$, $\ell = 1, \dots, M$, is therefore given by

$$D_{k,1}^{(\ell)} = \left(\frac{\pi}{h}\right)^\ell \sigma_\ell((k-1)\pi), \quad k = 1, \dots, N.$$

Like the recurrence (18), the recurrence (21) is unstable for large M , but for the same reason given in the Hermite case this does not cause problems in practice.

The code `sincdif.m` implements the recurrence (21), and uses MATLAB's built-in Toeplitz function to generate the differentiation matrices.

- (e) Since sinc derivative matrices are Toeplitz, it is possible to compute the matrix-vector product (3) asymptotically fast, in $O(N \log N)$ operations. A circulant matrix of dimension at least $2N \times 2N$ is constructed, which contains the Toeplitz matrix as its first $N \times N$ block. The data vector is padded with zeros so that it has the same row dimension as the circulant matrix. The product of any circulant matrix and a vector can be computed rapidly using the FFT. The product of the Toeplitz matrix and the original data vector is then easily recovered. The function `sincdift.m` implements this idea. For more details, see for example Strang [1986].

Calling Commands:

- (a) The calling command for `sincdif.m` is

```
>>[x, D] = sincdif(N, M, h);
```

On input the integer N is the size of the required differentiation matrices, and the integer M is the highest derivative needed. The scalar h is the mesh spacing h . On output the vector x contains the N sinc points (19). D is an $N \times N \times M$ array containing differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$.

- (b) The calling command for `sincdift.m` is

```
>>Dmf = sincdift(f, M, h);
```

On input the vector f , of length N , contains the values of the function $f(x)$ at the sinc points (19). M is the order of the required derivative.

On output the vector Dmf contains the values of the M th derivative of $f(x)$ at the corresponding points.

3.4 Half Line

Laguerre

MATLAB files: `lagdif.m`, `lagroots.m`

Interval:

$$[0, \infty).$$

Nodes: (Computed by `lagroots.m`)

$x_1 = 0$, and x_2, \dots, x_N are the roots of $L_{N-1}(x)$, the Laguerre polynomial of degree $N - 1$, indexed in increasing order of magnitude. *Note:* $x_N = O(N)$ as $N \rightarrow \infty$; see Abramowitz and Stegun [1964, Ch. 22].

Weight Function:

$$\alpha(x) = e^{-x/2}.$$

Interpolant:

$$p_{N-1}(x) = \sum_{j=1}^N \frac{e^{-x/2}}{e^{-x_j/2}} \phi_j(x) f_j,$$

where

$$\phi_j(x) = \frac{x L_{N-1}(x)}{(x L_{N-1})'(x_j)(x - x_j)}.$$

Barycentric Form of Interpolant: Similar to (12).

Differentiation Matrices: (Computed by `lagdif.m`.) For a formula we refer to Funaro [1992, Ch. 7]. Note that

$$D^{(\ell)} = (D^{(1)})^\ell.$$

Transform Formulas: The comments related to the Hermite method (see Section 3.3) are also applicable to the Laguerre method.

Accuracy: Has not been analyzed in any detail.

Notes:

- (a) The interval $[0, \infty)$ can be mapped to itself by the change of variable $x = b\tilde{x}$, where b is any positive real number. Like the Hermite method the Laguerre method therefore contains a free parameter—cf. point (a) in the notes of the Hermite method. It means that the Laguerre differentiation process is exact for functions of the form

$$e^{-\frac{1}{2}bx} p(x),$$

where $p(x)$ is any polynomial of degree $N - 1$ or less.

- (b) The quantities $\beta_j^{(\ell)}$ defined by (9) are given by

$$\beta_j^{(\ell)} = \left(-\frac{1}{2}\right)^\ell, \quad \ell = 1, \dots, M, \quad (22)$$

independent of j .

- (c) The code `lagdif.m` computes the Laguerre points by calling `lagroots.m`. It adds a node at $x = 0$ to facilitate the incorporation of boundary conditions; see Section 5.3. It then computes the quantities (9) defined by (22). A call to `poldif.m` completes the computation of the differentiation matrices.
- (d) It should be straightforward to generalize `lagdif.m` and `lagroots.m` to include the associated Laguerre polynomials $L_n^{(l)}(x)$; see Abramowitz and Stegun [1964, Ch. 22]. To the best of our knowledge it has not been investigated whether there is a significant advantage to be gained by considering this more general basis set.

Calling Commands:

(a) The calling command for `lagdif.m` is

```
>>[x, D] = lagdif(N, M, b);
```

On input the integer N is the size of the required differentiation matrices, and the integer M is the highest derivative needed. The scalar b is the scaling parameter b discussed above. On output the vector x , of length N , contains the Laguerre points scaled by b , plus a node at $x = 0$. D is an $N \times N \times M$ array containing differentiation matrices $D^{(\ell)}$, $\ell = 1, \dots, M$.

(b) The calling command for `lagroots.m` is

```
>>r = lagroots(N);
```

The input integer N is the degree of the Laguerre polynomial, and the output vector r contains its N roots.

3.5 Periodic Domain

Fourier

MATLAB files: `fourdif.m`, `fourint.m`

Interval:

$[0, 2\pi]$ (periodicity assumed).

Nodes: (Computed by `fourdif.m`)

$$x_k = (k - 1)h, \quad h = \frac{2\pi}{N}, \quad k = 1, \dots, N. \quad (23)$$

Weight Function:

$$\alpha(x) = 1.$$

Interpolant [Gottlieb et al. 1984; Henrici 1986, Sect. 13.6]:

$$t_N(x) = \sum_{j=1}^N \phi_j(x) f_j$$

where

$$\phi_j(x) = \frac{1}{N} \sin \frac{N}{2}(x - x_j) \cot \frac{1}{2}(x - x_j), \quad N \text{ even},$$

$$\phi_j(x) = \frac{1}{N} \sin \frac{N}{2}(x - x_j) \csc \frac{1}{2}(x - x_j), \quad N \text{ odd}.$$

Barycentric Form of Interpolant [Henrici 1986, Sect. 13.6]: (Computed by `fourint.m`)

$$t_N(x) = \frac{\sum_{j=1}^N (-1)^j f_j \cot \frac{1}{2}(x - x_j)}{\sum_{j=1}^N (-1)^j \cot \frac{1}{2}(x - x_j)}, \quad N \text{ even}, \quad (24)$$

$$t_N(x) = \frac{\sum_{j=1}^N (-1)^j f_j \csc \frac{1}{2}(x - x_j)}{\sum_{j=1}^N (-1)^j \csc \frac{1}{2}(x - x_j)}, \quad N \text{ odd}. \quad (25)$$

Differentiation Matrices [Gottlieb et al. 1984]: (Computed by fourdif.m.)

N even, $k, j = 1, \dots, N$:

$$D_{kj}^{(1)} = \begin{cases} 0 & k = j \\ \frac{1}{2}(-1)^{k-j} \cot \frac{(k-j)h}{2} & k \neq j \end{cases}$$

$$D_{kj}^{(2)} = \begin{cases} -\frac{\pi^2}{3h^2} - \frac{1}{6} & k = j \\ -(-1)^{k-j} \frac{1}{2} \csc^2 \frac{(k-j)h}{2} & k \neq j. \end{cases} \quad (26)$$

N odd, $k, j = 1, \dots, N$:

$$D_{kj}^{(1)} = \begin{cases} 0 & k = j \\ \frac{1}{2}(-1)^{k-j} \csc \frac{(k-j)h}{2} & k \neq j \end{cases}$$

$$D_{kj}^{(2)} = \begin{cases} -\frac{\pi^2}{3h^2} - \frac{1}{12} & k = j \\ -(-1)^{k-j} \frac{1}{2} \csc \frac{(k-j)h}{2} \cot \frac{(k-j)h}{2} & k \neq j. \end{cases} \quad (27)$$

If N is odd, then

$$D^{(\ell)} = (D^{(1)})^\ell.$$

If N is even, this last formula only holds for odd ℓ .

Transform Formulas [Gottlieb et al. 1984; Henrici 1986, Sect. 13.6]:

N even, $k = 1, \dots, N$:

$$t_N(x_k) = \sum_{j=-N/2}^{N/2-1} a_j e^{ijx_k} \Rightarrow t_N^{(\ell)}(x_k) = \begin{cases} \sum_{j=-N/2}^{N/2-1} (ij)^\ell a_j e^{ijx_k}, & \ell \text{ even} \\ \sum_{j=-N/2+1}^{N/2-1} (ij)^\ell a_j e^{ijx_k}, & \ell \text{ odd.} \end{cases} \quad (28)$$

N odd, $k = 1, \dots, N$:

$$t_N(x_k) = \sum_{j=-(N-1)/2}^{(N-1)/2} a_j e^{ijx_k} \Rightarrow t_N^{(\ell)}(x_k) = \sum_{j=-(N-1)/2}^{(N-1)/2} (ij)^\ell a_j e^{ijx_k}. \quad (29)$$

Accuracy: For an error analysis, see Tadmor [1986].

Notes:

- (a) The canonical interval is $[0, 2\pi]$. If the differential equation is posed on $[a, b]$ it should first be converted to $[0, 2\pi]$ through the linear transformation $x \longleftrightarrow a + (1/(2\pi))(b - a)x$.
- (b) Our suite contains two functions for Fourier differencing: `fourdif.m` for computing differentiation matrices, and `fourdiff.m` for computing derivatives using the FFT. The latter is based on a straightforward implementation of (28)–(29) using MATLAB's built-in FFT routine.
- (c) The function `fourdif.m` differs from the previous functions for constructing differentiation matrices in one important respect: the other functions compute the differentiation matrices $D^{(1)}, \dots, D^{(M)}$ recursively. In the Fourier case we could not find such a recursion in the literature, and we follow a different approach:

In the first- and second-derivative cases formulas (26)–(27) are used to compute the matrices explicitly. The first row and column of these matrices are computed with the aid of a “flipping trick” analogous to that discussed in point (d) in the notes of the Chebyshev method. The `toeplitz` command is then used to create the matrices.

To compute higher derivatives we note that all Fourier differentiation matrices are circulant. Therefore it suffices to construct the first column of each. The first column of $D^{(m)}$ is simply $D^{(m)}\mathbf{v}$, where \mathbf{v} is the column vector with 1 in the first position and zeros elsewhere. $D^{(m)}\mathbf{v}$ may be computed by applying the FFT-based method to the vector \mathbf{v} .

Calling Commands:

- (a) The calling command of `fourdif.m` is

```
>>[x, DM] = fourdif(N, M);
```

On input the integer N is the size of the required differentiation matrix, and the integer M is the derivative needed. On output, the vector \mathbf{x} , of length N , contains the equispaced nodes given by (23), and \mathbf{DM} is the $N \times N$ containing the differentiation matrix $D^{(M)}$.

Unlike the other functions in the suite, `fourdif.m` computes only the single matrix $D^{(M)}$, not the sequence $D^{(1)}, \dots, D^{(M)}$.

- (b) The calling command of `fourint.m` is

```
>>t = fourint(f, x)
```

On input the vector f , of length N , contains the function values at the equispaced nodes (23). The entries of the vector x , of arbitrary length, are the ordinates where the interpolant is to be evaluated. On output the vector t contains the corresponding values of the interpolant $t_N(x)$ as computed by the formula (24) or (25).

- (c) The calling command for `fourdifft.m` is

```
>>Dmf = fourdifft(f, M);
```

On input the vector f , of length N , contains the values of the function $f(x)$ at the equispaced points (23). M is the order of the required derivative. On output the vector Dmf contains the values of the M th derivative of $f(x)$ at the corresponding points.

4. BOUNDARY CONDITIONS

We now turn our attention to implementing boundary conditions. In the case of homogeneous Dirichlet boundary conditions of the form $u(1) = 0$ or $u(-1) = 0$, this amounts to nothing more than the deletion of appropriate rows and columns of the differentiation matrix. Examples of this strategy may be seen in Sections 5.1 and 5.3.

The treatment of boundary conditions that involve derivatives, such as Neumann or more general Robin conditions, is more complicated. There does not appear to be a unified approach in the literature, and we discuss two approaches here. The first involves Hermite interpolation, which is an extension of Lagrange interpolation that enables one to incorporate derivative values in addition to function values [Huang and Sloan 1992]. In the second approach the boundary conditions are enforced explicitly by adding additional equations to the main system [Canuto et al. 1988; Fornberg 1996].

The canonical interval is taken to be $[-1, 1]$; as remarked above, an arbitrary finite interval $[a, b]$ can always be rescaled to $[-1, 1]$ via a linear transformation. On the interval $[-1, 1]$ we use the Chebyshev points (13) as nodes. The Lagrangian interpolation polynomials associated with these points are denoted by $\{\phi_j(x)\}$; cf. Eq. (14).

4.1 Second Derivatives

In this section we discuss the details of `cheb2bc.m`, a function that enables one to solve the general two-point boundary value problem

$$u''(x) + q(x)u'(x) + r(x)u(x) = f(x), \quad -1 < x < 1, \quad (30)$$

subject to the boundary conditions

$$a_+u(1) + b_+u'(1) = c_+, \quad a_-u(-1) + b_-u'(-1) = c_-. \quad (31)$$

We assume, of course, that a_+ and b_+ are not both 0, and likewise for a_- and b_- .

The function `cheb2bc.m` generates a set of nodes $\{x_k\}$, which are essentially the Chebyshev points with perhaps one or both boundary points omitted. (When a Dirichlet condition is enforced at a boundary, that particular node is omitted, since the function value is explicitly known there.) The function also returns differentiation matrices $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ which are the first- and second-derivative matrices with the boundary conditions (31) incorporated. The matrices $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ may be computed from the Chebyshev differentiation matrices $D^{(1)}$ and $D^{(2)}$, which are computed by `chebdif.m`. Details are given below.

Our approach is based on Hermite interpolation; we refer to Huang and Sloan [1992, p. 52] for the general formulas regarding this type of interpolation. The following steps are taken to solve (30)–(31):

- (a) approximate $u(x)$ by the Hermite polynomial interpolant $p(x)$ that satisfies the boundary conditions (31);
- (b) require $p(x)$ to satisfy the Eq. (30) at the interpolation points, thereby converting the differential equation to a linear system;
- (c) solve the linear system for the unknown function values.

The form of the Hermite interpolant in step (a) depends on the type of boundary conditions. There are three cases to consider: Dirichlet/Dirichlet, Dirichlet/Robin, and Robin/Robin.

Dirichlet/Dirichlet Conditions. Occurs when

$$b_+ = b_- = 0.$$

Nodes: Since function values are specified at both endpoints the nodes are the interior Chebyshev points:

$$x_k = \cos\left(\frac{k\pi}{N-1}\right), \quad k = 1, \dots, N-2.$$

Interpolant: The interpolant is a polynomial of degree $N-1$ that satisfies the interpolation conditions

$$p_{N-1}(x_k) = u_k, \quad k = 1, \dots, N-2,$$

as well as the boundary conditions

$$a_+p_{N-1}(1) = c_+, \quad a_-p_{N-1}(-1) = c_-.$$

It is given explicitly by

$$p_{N-1}(x) = \tilde{\phi}_+(x) + \tilde{\phi}_-(x) + \sum_{j=1}^{N-2} u_j \tilde{\phi}_j(x), \quad (32)$$

where

$$\tilde{\phi}_+(x) = \left(\frac{c_+}{a_+} \right) \phi_1(x),$$

$$\tilde{\phi}_-(x) = \left(\frac{c_-}{a_-} \right) \phi_N(x),$$

$$\tilde{\phi}_j(x) = \phi_{j+1}(x), \quad j = 1, \dots, N-2.$$

(This interpolant is actually not of Hermite type, since derivative values do not appear.)

Differentiation Matrices: Define

$$\tilde{D}_{k,j}^{(1)} = \tilde{\phi}'_j(x_k), \quad \tilde{D}_{k,j}^{(2)} = \tilde{\phi}'_j(x_k) k, \quad j = 1, \dots, N-2.$$

Since $\tilde{\phi}_j(x) = \phi_{j+1}(x)$, the matrices $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ are submatrices of $D^{(1)}$ and $D^{(2)}$. The function `cheb2bc.m` computes $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ by calling `chebdif.m` to compute $D^{(1)}$ and $D^{(2)}$ and then extracting the submatrices corresponding to rows and columns $2, \dots, N-1$.

Dirichlet/Robin Conditions. Occurs when

$$b_+ \neq 0 \text{ and } b_- = 0.$$

In this case there is a Dirichlet condition at $x = -1$ and a Robin condition at $x = 1$.

Nodes: Since a function value is specified at $x = -1$ the Chebyshev node corresponding to $k = N$ is dropped:

$$x_k = \cos\left(\frac{(k-1)\pi}{N-1}\right), \quad k = 1, \dots, N-1.$$

Interpolant: The interpolant is a polynomial of degree N that satisfies the interpolation conditions

$$p_N(x_k) = u_k, \quad k = 1, \dots, N-1,$$

as well as the boundary conditions

$$a_+ p_N(1) + b_+ p'_N(1) = c_+, \quad a_- p_N(-1) = c_-.$$

It is given explicitly by

$$p_N(x) = \tilde{\phi}_+(x) + \tilde{\phi}_-(x) + \sum_{j=1}^{N-1} u_j \tilde{\phi}_j(x), \quad (33)$$

where

$$\begin{aligned} \tilde{\phi}_+(x) &= \left(\frac{c_+}{b_+} \right) (x - 1) \phi_1(x), \\ \tilde{\phi}_-(x) &= \left(\frac{c_-}{a_-} \right) \left(\frac{1 - x}{2} \right) \phi_N(x), \\ \tilde{\phi}_1(x) &= \left(1 - \left(\phi'_1(1) + \frac{a_+}{b_+} \right) (x - 1) \right) \phi_1(x), \\ \tilde{\phi}_j(x) &= \left(\frac{1 - x}{1 - x_j} \right) \phi_j(x), \quad j = 2, \dots, N - 1. \end{aligned}$$

Differentiation Matrices: Define

$$\tilde{D}_{k,j}^{(1)} = \tilde{\phi}'_j(x_k), \quad \tilde{D}_{k,j}^{(2)} = \tilde{\phi}''_j(x_k), \quad k, j = 1, \dots, N - 1.$$

The quantities $\tilde{\phi}'_j(x_k)$ and $\tilde{\phi}''_j(x_k)$ may be expressed explicitly in terms of the quantities $\phi'_j(x_k)$ and $\phi''_j(x_k)$, which are the entries of the standard Chebyshev differentiation matrices $D^{(1)}$ and $D^{(2)}$ computed by `chebdif.m`. The function `cheb2bc.m` computes $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ by calling `chebdif.m` to compute $D^{(1)}$ and $D^{(2)}$ and then taking appropriate combinations of these two matrices.

When the Dirichlet and Robin conditions are reversed, i.e., $b_+ = 0$ and $b_- \neq 0$, formulas similar to the above are obtained. The function `cheb2bc.m` also handles this case.

Robin/Robin Conditions. Occurs when

$$b_+ \neq 0 \text{ and } b_- \neq 0.$$

Nodes: Since function values are specified at neither endpoint the nodes are the full set of Chebyshev points:

$$x_k = \cos\left(\frac{(k-1)\pi}{N-1}\right), \quad k = 1, \dots, N.$$

Interpolant: The interpolant is a polynomial of degree $N + 1$ that satisfies the interpolation conditions

$$p_{N+1}(x_k) = u_k, \quad k = 1, \dots, N,$$

as well as the boundary conditions

$$a_+p_{N+1}(1) + b_+p'_{N+1}(1) = c_+, \quad a_-p_{N+1}(-1) + b_-p'_{N+1}(-1) = c_-.$$

It is given explicitly by

$$p_{N+1}(x) = \tilde{\phi}_+(x) + \tilde{\phi}_-(x) + \sum_{j=1}^N u_j \tilde{\phi}_j(x), \quad (34)$$

where

$$\tilde{\phi}_+(x) = \left(\frac{c_+}{b_+} \right) \left(\frac{x^2 - 1}{2} \right) \phi_1(x),$$

$$\tilde{\phi}_-(x) = \left(\frac{c_-}{b_-} \right) \left(\frac{1 - x^2}{2} \right) \phi_N(x),$$

$$\tilde{\phi}_1(x) = \left[\frac{1+x}{2} + \left(\frac{1}{2} + \phi'_1(1) + \frac{a_+}{b_+} \right) \left(\frac{1-x^2}{2} \right) \right] \phi_1(x),$$

$$\tilde{\phi}_j(x) = \left(\frac{1-x^2}{1-x_j^2} \right) \phi_j(x), \quad j = 2, \dots, N-1,$$

$$\tilde{\phi}_N(x) = \left[\frac{1-x}{2} + \left(\frac{1}{2} - \phi'_N(-1) - \frac{a_-}{b_-} \right) \left(\frac{1-x^2}{2} \right) \right] \phi_N(x).$$

Differentiation Matrices: Define

$$\tilde{D}_{k,j}^{(1)} = \tilde{\phi}'_j(x_k), \quad \tilde{D}_{k,j}^{(2)} = \tilde{\phi}''_j(x_k), \quad k, j = 1, \dots, N.$$

As before the quantities $\tilde{\phi}'_j(x_k)$ and $\tilde{\phi}''_j(x_k)$ may be expressed explicitly in terms of the quantities $\phi'_j(x_k)$ and $\phi''_j(x_k)$. Hence $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$ are computed by combining $D^{(1)}$ and $D^{(2)}$, both of which are computed by `chebdif.m`.

The function `cheb2bc.m` computes the various matrices and boundary condition vectors described above. The calling command is

```
>>[x, D2t, D1t, phip, phim] = cheb2bc(N, g);
```

On input N is the integer N defined above in (32)–(34). The array $g = [a_+ \ b_+ \ c_+; a_- \ b_- \ c_-]$ contains the boundary condition coefficients, with a_+ , b_+ , and c_+ on the first row and a_- , b_- , and c_- on the second. On output x is the node vector x . The matrices $D1t$ and $D2t$ contain $\tilde{D}^{(1)}$ and $\tilde{D}^{(2)}$, respectively. The first and second columns of `phip` contain $\tilde{\phi}'_+(x)$ and $\tilde{\phi}''_+(x)$, evaluated at the points in the node vector. Similarly, the first and

Table IV. Solving the Boundary Value Problem (35)

```

>>N = 16;
>>g = [2 -1 1; 2 1 -1]; % Boundary condition array
>>[x, D2t, D1t, phip, phim] = cheb2bc(N, g); % Get nodes, matrices, and
% vectors
>>f = 4*exp(x.^2);
>>p = phip(:,2)-2*x.*phip(:,1); % psi+
>>m = phim(:,2)-2*x.*phim(:,1); % psi-
>>D = D2t-diag(2*x)*D1t+2*eye(size(D1t)); % Discretization matrix
>>u = D\(f-p-m); % Solve system

```

second columns of phim contain $\tilde{\phi}'_-(x)$ and $\tilde{\phi}''_-(x)$, evaluated at points in the node vector. Since $\tilde{\phi}_+(x)$ and $\tilde{\phi}_-(x)$ are both 0 at points in the node vector, these function values are not returned by `cheb2bc.m`.

Using `cheb2bc.m`, it becomes a straightforward matter to solve the two-point boundary value problem (30)–(31). Consider, for example,

$$u'' - 2xu' + 2u = 4e^{x^2}, \quad 2u(1) - u'(1) = 1, \quad 2u(-1) + u'(-1) = -1. \quad (35)$$

Since Robin conditions are specified at each end point, we consider the interpolating polynomial $p_{N+1}(x)$ in (34). Requiring that this polynomial satisfies the differential equation at each point of the node vector implies

$$\psi_+(x_k) + \psi_-(x_k) + \sum_{j=1}^N u_j(\tilde{\phi}_j''(x_k) - 2x_k \tilde{\phi}_j'(x_k) + 2\tilde{\phi}_j(x_k)) = 4e^{x_k^2}, \quad (36)$$

for $k = 1, \dots, N$, where

$$\psi_+(x_k) = \tilde{\phi}_+''(x_k) - 2x_k \tilde{\phi}_+'(x_k) + 2\tilde{\phi}_+(x_k),$$

and

$$\psi_-(x_k) = \tilde{\phi}_-''(x_k) - 2x_k \tilde{\phi}_-'(x_k) + 2\tilde{\phi}_-(x_k).$$

(Observe that the last term in each expression is in fact 0.)

Putting (36) in matrix form, we have

$$(\tilde{D}^{(2)} + Q\tilde{D}^{(1)} + R)\mathbf{u} + \mathbf{p} + \mathbf{m} = \mathbf{f}.$$

Q and R are $N \times N$ diagonal matrices with $-2x_k$ and 2 on the diagonal, respectively, and

$$\mathbf{u}_k = u_k, \quad \mathbf{p}_k = \psi_+(x_k), \quad \mathbf{m}_k = \psi_-(x_k), \quad \mathbf{f}_k = 4e^{x_k^2}, \quad k = 1, \dots, N.$$

The MATLAB code for solving (35) is given in Table IV.

Table V. Solving the Eigenvalue Problem (37)

```

>>N = 16;
>>g = [1 1 0; 1 0 0];           % Boundary condition array
>>[x, D2t] = cheb2bc(N, g);      % Get nodes and 2nd derivative matrix
>>e = eg(D2t);                   % Compute eigenvalues

```

The function `cheb2bc.m` can also be employed to solve differential eigenvalue problems. Consider, for example, the model problem:

$$u'' = \lambda u, \quad u(1) + u'(1) = 0, \quad u(-1) = 0. \quad (37)$$

Since Dirichlet/Robin conditions are applicable, we approximate $u(x)$ by the interpolating polynomial $p_N(x)$ in (33), noting that $c_+ = c_- = 0$. Requiring $p_N(x)$ to satisfy the differential equation at points in the node vector, we have

$$\sum_{j=1}^{N-1} u_j \tilde{\phi}_j''(x_k) = \lambda u_k, \quad k = 1, \dots, N-1.$$

In matrix form, this is

$$\tilde{D}^{(2)} \mathbf{u} = \lambda \mathbf{u},$$

which may be solved by the MATLAB code in Table V.

There are other approaches to the solution of two-point boundary value problems by spectral methods. Perhaps the most noteworthy is the integral equation approach of Greengard [1991] and Greengard and Rokhlin [1991]. Huang and Sloan [1993] advocate the use of different interpolants for approximating the first- and second-derivatives, an approach that seems to yield higher accuracy for singularly perturbed problems. Another technique for dealing with such problems has been proposed in Tang and Trummer [1996].

4.2 Fourth Derivatives

We shall make no attempt to cover all possible boundary conditions that appear in conjunction with fourth-order problems such as

$$u''''(x) = f(x). \quad (38)$$

We focus instead on the two sets of conditions that appear to be the most relevant in physical situations, namely the clamped conditions

$$u(\pm 1) = u'(\pm 1) = 0, \quad (39)$$

and the hinged conditions

$$u(\pm 1) = u''(\pm 1) = 0. \quad (40)$$

Starting with the clamped boundary conditions, our task is to construct a polynomial of degree $N + 1$ that satisfies $N - 2$ interpolation conditions

$$p_{N+1}(x_k) = u_k, \quad (41)$$

as well as the four boundary conditions

$$p_{N+1}(\pm 1) = p'_{N+1}(\pm 1) = 0. \quad (42)$$

To this end, consider the set of Chebyshev nodes $\{x_k\}$ with the endpoints $x = \pm 1$ deleted, i.e.,

$$x_k = \cos\left(\frac{k\pi}{N-1}\right), \quad k = 1, \dots, N-2. \quad (43)$$

Let $\{\phi_j\}$ be the corresponding set of Lagrangian interpolating polynomials of degree $N - 3$, i.e.,

$$\phi_j(x) = (-1)^j \frac{1 - x_j^2}{(N-1)^2} \frac{T'_{N-1}(x)}{x - x_j}, \quad j = 1, \dots, N-2.$$

Define $\{\tilde{\phi}_j\}$ by

$$\tilde{\phi}_j(x) = \left(\frac{1 - x^2}{1 - x_j^2}\right)^2 \phi_j(x), \quad j = 1, \dots, N-2.$$

It is now readily checked that

$$p_{N+1}(x) = \sum_{j=1}^{N-2} u_j \tilde{\phi}_j(x)$$

is a polynomial of degree $N + 1$ that satisfies the interpolating conditions (41), as well as the boundary conditions (42). The approximation to the fourth-derivative operator is therefore defined by

$$p'''_{N+1}(x_k) = \sum_{j=1}^{N-2} u_j \tilde{\phi}_j'''(x_k),$$

i.e., the differentiation matrix has entries

$$\tilde{D}_{k,j}^{(4)} = \tilde{\phi}_j'''(x_k), \quad k, j = 1, \dots, N-2. \quad (44)$$

It is possible to generate $\tilde{D}^{(4)}$ using the algorithm of Section 2. The weight function is taken as $\alpha(x) = (1 - x^2)^2$, and the nodes are the interior Chebyshev points (43). This algorithm has been implemented in a function `cheb4c.m`, the calling command of which is

```
>>[x, D4] = cheb4c(N);
```

The function returns the nodes in (43) and the matrix $\tilde{D}^{(4)}$ in (44). The modifications for enhanced accuracy mentioned in point (c) of the notes of Section 3.2 have been incorporated into `cheb4c.m`. We postpone an application of this function to Section 5.5, where it will be used to solve a problem in hydrodynamic stability.

Turning to the hinged conditions (40), we observe that it may not be possible to construct a unique Hermite polynomial interpolant that satisfies these boundary conditions. (*Example:* The polynomial $p(x) = 0$ and all multiples of the polynomial $p(x) = 3x^5 - 10x^3 + 7x$ satisfy the interpolating condition $p(0) = 0$ as well as the hinged boundary conditions $p(\pm 1) = p''(\pm 1) = 0$.) This nonuniqueness does not occur with clamped boundary conditions. For a discussion of the connection between the uniqueness of the solution to the interpolation problem and the question of well-posed boundary conditions of Eq. (38) we refer to Sharma [1972].

For this reason we shall solve (38) subject to (40) with the method of explicit enforcement of boundary conditions. The interpolating polynomial is taken to be

$$p_{N-1}(x) = \sum_{j=2}^{N-1} u_j \phi_j(x), \quad (45)$$

where $\{\phi_j(x)\}$ is again the Lagrangian basis set (14) corresponding to the full set of Chebyshev nodes $\{x_j\}$ as defined by (13). This interpolant satisfies $p_{N-1}(\pm 1) = 0$. We require the Eq. (38) to be satisfied at the interior $N - 4$ gridpoints:

$$\sum_{j=2}^{N-1} u_j \phi_j''''(x_k) = f(x_k), \quad k = 3, \dots, N - 2. \quad (46)$$

The hinged boundary conditions imply

$$\sum_{j=2}^{N-1} u_j \phi_j''(x_1) = 0, \quad \sum_{j=2}^{N-1} u_j \phi_j''(x_N) = 0. \quad (47)$$

Equations (46)–(47) form a linear system of $N - 2$ equations. In matrix form, we have

$$\tilde{D}\mathbf{u} = \mathbf{f},$$

Table VI. Solving the Boundary Value Problem (38) Subject to (40)

```

>>N = 16;
>>[x, Dm] = chebdf(N,4);           % Get derivative matrices
>>D2 = Dm(:, :, 2);                 % 2nd derivative
>>D4 = Dm(:, :, 4);                 % 4th derivative
>>D = [D2(1,2:N-1); D4(3:N-2,2:N-1); D2(N,2:N-1)]; % Create D
>>f = [0; exp(x(3:N-2).^2); 0];    % Create rhs vector
>>u = D\f;                          % Solve system

```

where $\mathbf{u} = [u_2 \cdot \cdot \cdot u_{N-1}]^T$, $\mathbf{f} = [0 \ f(x_3) \cdot \cdot \cdot f(x_{N-2}) \ 0]^T$, and

$$\tilde{D}_{1,j} = \phi_j'(x_1), \quad j = 2, \dots, N-1,$$

$$\tilde{D}_{k-1,j} = \phi_j''''(x_k), \quad j = 2, \dots, N-1, \quad k = 3, \dots, N-2,$$

$$\tilde{D}_{N-2,j} = \phi_j''(x_N), \quad j = 2, \dots, N-1.$$

The MATLAB code for solving Eq. (38) subject to the hinged conditions (40) with right side $f(x) = e^{x^2}$ is given in Table VI.

In the case of an eigenvalue problem the procedure in Table VI must be modified slightly. Consider, for example, the model problem

$$u'''' = \lambda u, \quad u(\pm 1) = u''(\pm 1) = 0. \quad (48)$$

As before we use the interpolating polynomial (45), and require the differential equation to be satisfied at the interior grid points, yielding

$$\sum_{j=2}^{N-1} u_j \phi_j''''(x_k) = \lambda u_k, \quad k = 3, \dots, N-2. \quad (49)$$

To put the discrete Eqs. (47) and (49) in the form of an algebraic eigenvalue problem we eliminate u_2 and u_{N-1} . Define

$$M_1 = - \begin{bmatrix} \phi_3''(x_1) & \cdots & \phi_{N-2}''(x_1) \\ \phi_3''(x_N) & \cdots & \phi_{N-2}''(x_N) \end{bmatrix}, \quad M_2 = \begin{bmatrix} \phi_2''(x_1) & \phi_{N-1}''(x_1) \\ \phi_2''(x_N) & \phi_{N-1}''(x_N) \end{bmatrix},$$

and

$$M_3 = \begin{bmatrix} \phi_2''''(x_3) & \phi_{N-1}''''(x_3) \\ \phi_2''''(x_4) & \phi_{N-1}''''(x_4) \\ \vdots & \vdots \\ \phi_2''''(x_{N-2}) & \phi_{N-1}''''(x_{N-2}) \end{bmatrix}.$$

The differential eigenvalue problem now becomes the algebraic eigenvalue problem

Table VII. Solving the Eigenvalue Problem (48)

```

>>N = 16;
>>[x, DM] = chebdif(N, 4); % Get derivative matrices
>>D2 = DM(:, :, 2); % 2nd derivative
>>D4 = DM(:, :, 4); % 4th derivative
>>M1 = -[D2(1, 3:N-2); D2(N, 3:N-2)];
>>M2 = [D2(1, 2) D2(1, N-1); D2(N, 2) D2(N, N-1)];
>>M3 = [D4(3:N-2, 2) D4(3:N-2, N-1)];
>>D4t = D4(3:N-2, 3:N-2) + M3*(M2\M1); % Create D4(tilde)
>>e = eig(D4t); % Compute eigenvalues

```

$$\tilde{D}^{(4)}\mathbf{u} = \lambda\mathbf{u},$$

where $\mathbf{u} = [u_3 \dots u_{N-2}]^T$ and

$$\tilde{D}^{(4)} = \bar{D}^{(4)} + M_3 M_2^{-1} M_1.$$

Here $\bar{D}^{(4)}$ is the interior $(N - 4) \times (N - 4)$ submatrix, corresponding to rows and columns 3 to $N - 2$, of the standard fourth-derivative Chebyshev matrix $D^{(4)}$, which is computed by `chebdif.m`.

The MATLAB code for solving the eigenvalue problem (48) is given in Table VII.

5. APPLICATIONS

In this section we provide templates for using the functions in our suite to solve a variety of problems. We hope that many users may find here an example that is sufficiently similar to theirs that it would require only a few lines of additional code to solve their particular problem. With this aim in mind we attempt to cover a wide a range of problems that utilize nearly all the functions in our suite. The problems are summarized in Table I.

The examples discussed here are all well-known problems in physics and engineering. Many efficient numerical methods have been proposed for their solution, but few of them have been solved by the differentiation matrix approach followed here. It is beyond the scope of this paper to undertake a detailed comparison of how this approach compares to the more established numerical procedures with regards to accuracy, stability, and efficiency. Nevertheless, we are confident that our methods are competitive. And as our examples will show, in terms of coding effort their efficiency can hardly be surpassed.

Each of the examples below involves the use of at least one of the differentiation, interpolation, or rootfinding functions in our suite. All additional MATLAB coding required to solve the problem is displayed in Tables VIII–XVII. The user should be able to duplicate these results by simply retyping the lines following the MATLAB prompt `>>`. Only the

instructions that control the appearance of the graphics output, such as the linewidth, symbol size, and headings of the plots, have been omitted (to avoid clutter).

When a code is going to be used several times it is efficient to turn it into a MATLAB script or function file. Our suite contains all the MATLAB codes of Tables VIII–XVII as *M*-files; see the Appendix for a summary.

5.1 The Complementary Error Function

The accurate computation of the complementary error function, defined by

$$\operatorname{erfc}(t) = \frac{2}{\sqrt{\pi}} \int_t^{\infty} e^{-x^2} dx, \quad t > 0, \quad (50)$$

is required in applications in physics, engineering, and statistics. Since this function decays superexponentially as $t \rightarrow \infty$, it is practical to approximate instead

$$y(t) = e^{t^2} \operatorname{erfc}(t),$$

a function that decays like $1/t$.

Our suite contains two function files, `cerfa.m` and `cerfb.m`, for computing $y(t)$. They implement different boundary conditions as explained below. Both of these codes are based on a method in Schonfelder [1978], which we have adapted for the differentiation matrix approach.

We start by converting the integral to a differential equation, by multiplying (50) by e^{t^2} and differentiating the result:

$$y'(t) - 2ty = -\frac{2}{\sqrt{\pi}}, \quad 0 \leq t < \infty.$$

This is accompanied by one of the side conditions $y(0) = 1$ or $y(\infty) = 0$. Since the domain of interest is the half-line, the Laguerre approach seems suitable. The function $y(t)$ decays slowly, however, and therefore this method is not particularly accurate. Following Schonfelder [1978], we consider instead the Chebyshev method in combination with the change of variables

$$x = \frac{t - c}{t + c} \Leftrightarrow t = c \frac{1 + x}{1 - x}. \quad (51)$$

This maps $t \in [0, \infty)$ to the canonical Chebyshev interval $x \in [-1, 1]$ for each positive value of the parameter c . This parameter is free to be tuned for optimum accuracy.

Having made the change of variables the differential equation is converted to

$$(1-x)^3 y' - 4c^2(1+x)y = \frac{4c}{\sqrt{\pi}}(x-1), \quad -1 \leq x \leq 1.$$

One could use either of the two boundary conditions

$$(a) y = 0 \text{ at } x = 1, \text{ or } (b) y = 1 \text{ at } x = -1. \quad (52)$$

The first condition was used in Schonfelder [1978]; we consider both conditions here.

At this point our approach starts to deviate from Schonfelder's. In Schonfelder [1978] the function y is approximated by the Chebyshev series $\sum a_n T_n(x)$ which leads to a linear system for the expansion coefficients a_n . Computing the coefficients of the linear system involves the manipulation of Chebyshev identities which is a complicated procedure compared to what we propose here.

Let D be the Chebyshev first-derivative matrix of order $N+1$, and let $\{x_k\}$ be the corresponding Chebyshev nodes as defined in Section 3.2. To incorporate the first boundary condition in (52), the first row and column of D are deleted. Then the differential equation may be approximated by the $N \times N$ linear system

$$A\mathbf{y} = \mathbf{b},$$

where

$$A = \text{diag}((1-x_k)^3)D - 4c^2 \text{diag}(1+x_k), \quad \mathbf{b}_k = \frac{4c}{\sqrt{\pi}}(x_k-1), \quad k = 1, \dots, N.$$

The solution of the linear system provides approximations to $y(t_k)$, where the t_k are the images of the Chebyshev points $x_k = \cos(k\pi/N)$ under the transformation (51). To compute $y(t)$ at arbitrary points Chebyshev barycentric interpolation is used.

The MATLAB computation shown in Table VIII is essentially the code of the script file `cerfa.m`. It computes the values of $y(t) = e^{t^2} \text{erfc}(t)$ at the (arbitrarily chosen) set of ordinates $t = 10^m$, $m = -2, -1, 0, 1$. As in Schonfelder [1978] we have used $c = 3.75$, which was empirically found to be near optimal.

As N is increased, the method yields the rapid convergence seen in Table IX. The accuracy is comparable to that of Schonfelder's method. The condition number of A grows relatively slowly, namely $\text{cond}(A) \approx 29, 63, 110$ for $N = 10, 15, 20$, indicating that this is a reasonably well-conditioned procedure for computing the complementary error function.

Table VIII. Computing the Function $y(t) = e^{t^2} \operatorname{erfc}(t)$

```

>>c = 3.75; N = 20;           % Initialize parameters
>>t = 10.^[-2:1:2];          % Points at which y(t) are required

>>[x, D] = chebdif(N+1,1);    % Compute Chebyshev points,
>>D = D(2:N+1,2:N+1);        % assemble differentiation matrix,
>>x = x(2:N+1);              % and incorporate boundary condition

>>A = diag((1-x).^3)*D-diag(4*c.^2*(1+x)); % Coefficient matrix
>>b = 4*c/sqrt(pi)*(x-1);     % Right-hand side
>>y = A\b;                    % Solve linear system

>>p = chebint([0; y], (t-c)./(t+c));      % Interpolate

```

Table IX. Chebyshev Approximations to $y(t) = e^{t^2} \operatorname{erfc}(t)$

N	$t = 0.01$	$t = 0.1$	$t = 1$	$t = 10$
10	0.98881546	0.89645698	0.427584	0.0561409
15	0.9888154610463	0.896456979969	0.427583576156	0.0561409927
20	0.9888154610463	0.89645697996912	0.42758357615581	0.056140992743823
$y(t)$	0.988815461046343	0.896456979969126	0.427583576155807	0.0561409927438226

We remark that with boundary condition (b) in (52) the implementation is slightly more complicated due to the incorporation of the inhomogeneous term; see `cerfb.m` for details. The accuracy appears to be higher, however, and the condition number is also smaller. If the reader wishes to use this algorithm for practical computations `cerfb.m` is recommended.

Another bit of practical advice is that if the code is to be used many times, it will be efficient to compute the function values $y(t_k)$ and save them for later use. Then each subsequent evaluation of $y(t)$ would skip the linear system solver and go directly to the interpolation routine. This is akin to Schonfelder's approach in which the Chebyshev expansion coefficients are computed once and for all.

5.2 The Mathieu Equation

Mathieu's equation,

$$y''(x) + (a - 2q \cos 2x)y = 0, \quad (53)$$

arises when solving the wave equation in elliptical coordinates, in the stability analysis of Duffing's equation, and as the first step in the inverse scattering procedure for solving the Korteweg–de Vries equation with periodic boundary conditions. The basic question in these applications is which sets of parameter values (q, a) , called characteristic values, give rise to bounded nontrivial solutions of period π (resp. 2π).

Table X. Plotting the Characteristic Values of Mathieu's Equation

```

>>[x, D] = fourdif(32,2);
>>for q = 0.1:0.1:12                                % For loop over q values
>>a = eig(2*q*diag(cos(2*x))-D);                    % Compute eigenvalues (period 2 pi)
>>plot(q+i*a,'o');
>>hold on;
>>end
>>axis([0 12 -10 30])                                % Zoom in

```

We first consider the case of solutions of period 2π . The classical approach to solving the problem is to assume the Fourier series representation $y(x) = \sum c_k e^{ikx}$. Inserting this series into (53) yields a recurrence relation for the expansion coefficients c_k in terms of a and q . The eigenvalues are typically found by computing determinants, or by computing continued fractions. For a survey on the numerical solution of Mathieu's equation we refer to Alhargan [1996].

We propose an alternative to the classical methods here, namely the use of the Fourier differentiation matrix. Let D be the $N \times N$ Fourier second-derivative matrix defined in Section 3.5, based on the equidistant nodes $x_k = 2\pi(k-1)/N$, $k = 1, \dots, N$. Let

$$C = \text{diag}(\cos 2x_k).$$

Then the Mathieu equation (53) may be approximated by

$$(2qC - D)\mathbf{y} = a\mathbf{y} \tag{54}$$

where \mathbf{y} is the vector of approximate eigenfunction values $y(x_k)$. To compute solutions of period π it is necessary to replace D by $4D$, and to modify the definition of C to $C = \text{diag}(\cos x_k)$. This follows from the change of variables $x \longleftrightarrow 2x$.

Equation (54) defines an algebraic eigenvalue problem that may be solved by MATLAB's `eig` routine. It yields N values of a for each q . Not all of these values should be accepted, however, since only the values of a nearest the origin can be assumed to be accurate, as these eigenvalues correspond to the least rapidly oscillating eigenfunctions $y(x)$.

The strategy is as follows: solve the eigenvalue problem (54), for q ranging from 0 to some maximum value. This defines a set of curves in the (q, a) parameter plane which may be plotted using MATLAB's graphing routines. The code in Table X, which is the script file `matplot.m` in our suite, implements this strategy. Note that by restricting the axes of the figure, attention is focussed on the eigenvalues of smallest magnitude. The resulting plot, shown in Figure 1, is indistinguishable from the plot given in Abramowitz and Stegun [1964, p. 724].

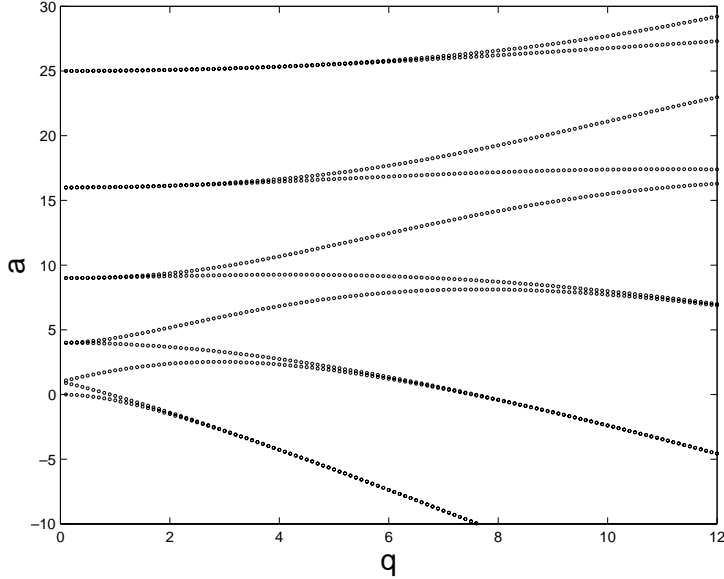


Fig. 1. Characteristic values of Mathieu's equation.

The procedure suggested here may also be used to compute the Mathieu functions. For example, the Mathieu cosine-elliptic function $ce_0(x, q)$ is the eigenfunction of period π , associated with the smallest characteristic number a (corresponding to the lower curve in Figure 1). There are various normalizations used in the literature—we shall use the one consistent with Abramowitz and Stegun [1964, Ch. 20], which is

$$\int_0^{2\pi} ce_0^2(x, q) dx = \pi.$$

The strategy for computing $ce_0(x, q)$ is as follows: solve the eigenproblem (54), rescaled to have period π , and determine the smallest eigenvalue. The corresponding eigenvector, after normalization, represents the values $ce_0(x_k, q)$ on the evenly spaced nodes x_k in $[0, \pi]$. To compute the function at arbitrary values of x , barycentric trigonometric interpolation is used. Our MATLAB function `ce0.m` implements this strategy; see Table XI.

For $q = 25$ the function `ce0.m` yields the rapidly converging approximations seen in Table XII. For smaller values of q the convergence is even faster. (The actual values of the Mathieu function were taken from the tables in Abramowitz and Stegun [1964, p. 748].)

Table XI. Computing the Mathieu Functions $ce_0(x, q)$

```

function c = ce0(x, q, N);
% The function c = ce0(x, q, N) computes the value of the
% Mathieu cosine-elliptic function ce0(x, q) using N Fourier modes.

[t, D] = fourdif(N, 2);           % Assemble Differentiation Matrix
[V, E] = eig((q/2)*diag(cos(t))-D); % Solve Eigenproblem-rescaled to
                                   % period pi

[m, l] = min(diag(E));           % Determine smallest characteristic
                                   % number
v = abs(V(:, l))*sqrt(N/2);      % Normalize the corresponding
                                   % eigenfunction

c = fourint(v, 2*x);             % Compute function values with
                                   % barycentric trigonometric
                                   % interpolation

```

Table XII. Fourier approximations to $ce_0(x, 25)$

N	$x = 0$	$x = \pi/2$
15	2.17(-4)	1.65749
20	2.158625(-4)	1.6575103
25	2.15863018(-4)	1.6575103
$ce_0(x, 25)$	2.15863018(-4)	1.6575103

5.3 The Schrödinger Equation

The Schrödinger equation,

$$-y''(x) + y(x) = \lambda q(x)y(x),$$

plays a central role in the theory of quantum mechanics. Different potentials $q(x)$ describe various phenomena—among these is the interaction of a neutron with a heavy nucleus which is modeled by the Woods-Saxon potential [Flügge 1971, p. 162]

$$q(x) = \frac{1}{1 + e^{(x-r)/\epsilon}}.$$

Physically meaningful values of the constants r and ϵ are given in the MATLAB code in Table XIII, and the boundary conditions are

$$y(0) = 0, \quad \lim_{x \rightarrow \infty} y(x) = 0.$$

Since the domain is $[0, \infty)$, solving the Schrödinger equation by the Laguerre spectral collocation method is a natural idea. We could not,

Table XIII. Computing the Eigenvalues of the Schrödinger Equation

```

>>b = 4; N = 20; % Initialize parameters.
>>r = 5.08685476; epsi = 0.929852862;

>>[x,D] = lagdif(N+1,2,b); % Compute Laguerre derivative matrix.
>>D2 = D(2:N+1,2:N+1,2); % Delete first row and column,
>>x = x(2:N+1); % to enforce boundary condition.

>>Q = diag(1./(1+exp((x-r)/epsi))); % Woods-Saxon potential.
>>I = eye(size(D2)); % Identity matrix.

>>e = min(eig(-D2+I,Q)); % Compute smallest eigenvalue.

```

however, find any references to this approach—numerical work reported in the literature involved the sinc method [Eggert et al. 1987], various shooting methods [Pryce 1993], and the finite-element method [Schoombie and Botha 1981].

Let D be the second-derivative Laguerre matrix of order $N + 1$, as computed by `lagdif.m`. Let the scaling parameter be b , as defined in Section 3.4. This means that the nodes are $x_j = r_j/b$, where the r_j are the roots of $L_N(x)$. There is an additional boundary node $x = 0$; incorporation of the boundary condition at this node means the first row and column of D are to be deleted. The boundary condition at $x = \infty$ is automatically taken care of by the Laguerre expansion. The Schrödinger equation is therefore approximated by the $N \times N$ matrix eigenvalue problem

$$(-D + I)\mathbf{y} = \lambda Q\mathbf{y},$$

where \mathbf{y} represents the approximate eigenfunction values at the nodes, where I is the identity matrix, and where

$$Q = \text{diag}\left(\frac{1}{1 + e^{(x_j - r)/\epsilon}}\right).$$

The MATLAB function `schrod.m` in our suite implements this method; it is reproduced in Table XIII.

The physically interesting eigenvalue (associated with the $1s$ ' state of the neutron) is the one of smallest magnitude. It has been computed to seven-digit accuracy as $\lambda = 1.424333$ independently by Eggert et al. [1987] and Schoombie and Botha [1981]. The Laguerre method shown in Table XIII computed this eigenvalue to full accuracy with $N = 20$ (resp. $N = 30$) and all scaling parameters roughly in the range $b \in [3, 6]$ (resp. $b \in [2, 9]$).

5.4 The Sine-Gordon Equation

The sine-Gordon equation,

$$u_{tt} = u_{xx} - \sin u,$$

is related to the Korteweg–de Vries and cubic Schrödinger equations in the sense that all these equations admit soliton solutions. The equation describes nonlinear waves in elastic media, and it also has applications in relativistic field theory [Drazin and Johnson 1989]. Being completely integrable, the sine-Gordon equation is solvable, at least in principle, by the method of inverse scattering [Drazin and Johnson 1989]. In practice, however, this method of solution is cumbersome to execute when arbitrary initial data $u(x, 0)$ and $u_t(x, 0)$ are prescribed. A more practical approach to solving the sine-Gordon equation is direct numerical simulation.

To prepare the equation for numerical solution we introduce the auxiliary variable $v = u_t$. This reduces the second-order equation to the first-order system

$$u_t = v \tag{55}$$

$$v_t = u_{xx} - \sin u.$$

The derivative with respect to x may be approximated by any second-derivative matrix D appropriate for the domain under consideration, which leads to

$$\mathbf{u}_t = \mathbf{v} \tag{56}$$

$$\mathbf{v}_t = D\mathbf{u} - \sin \mathbf{u}.$$

The unknown function values at the gridpoints are represented by the vectors \mathbf{u} and \mathbf{v} , and the operation $\sin \mathbf{u}$ should be interpreted as applying the sine function componentwise to the entries of \mathbf{u} .

The domain of interest for the sine-Gordon equation is typically the real line $x \in (-\infty, \infty)$. We will therefore use the Hermite and sinc second-derivative matrices for D , as well as the Fourier matrix rescaled to a truncated domain $[-L, L]$. We suspect that this might be the first application of the Hermite method (and perhaps also the sinc method) to simulate solitons. The traditional approaches to solving these equations involve the method of finite differences and the Fourier method; see Fornberg [1996, p. 130].

Having selected an appropriate differentiation matrix D , it remains to solve the nonlinear system of ordinary differential equations (56). We shall use MATLAB's built-in function `ode45.m` for this purpose [Shampine and Reichelt 1997]. It is based on a Runge-Kutta fourth- and fifth-order pair, combined with Fehlberg's time-step selection scheme.

Table XIV. Function for Computing the Right-Hand Side of the Sine-Gordon System (56)

```

function dw = sgrhs(t,w,flag,D);

% Function for computing the right-hand side of the SG equation

N = length(w)/2;
u = w(1:N); v = w(N+1:2*N); % Extract the u, v variables from w

du = v; % Compute the right-hand side
dv = D*u-sin(u);

dw = [du; dv]; % Recombine the du/dt, dv/dt vectors into dw/dt
dt

```

The first step in using `ode45.m` is to write a function that computes the right side of the system (56). This function is shown in Table XIV. (The `flag` input parameter is used by the integration routine; type `help ode45` in MATLAB for more details.)

It remains to select initial conditions. We consider an example with a known solution,

$$u(x, t) = 4 \tan^{-1} \left(\frac{\sin(t/\sqrt{2})}{\cosh(x/\sqrt{2})} \right), \quad (57)$$

and extract the corresponding initial conditions

$$u(x, 0) = 0 \quad (58)$$

$$u_t(x, 0) = 2\sqrt{2} \operatorname{sech}(x/\sqrt{2}).$$

The solution (57)—called a “breather soliton” for its oscillatory temporal evolution—is represented in Figure 2.

We can now assemble these components in a main program, shown in Table XV. It generates the differentiation matrix, computes the initial conditions (58), and passes all this information to the function `ode45.m`. On output it yields approximations to $u(x_k, t_j)$, where the x_k are the nodes corresponding to the differentiation matrix D , and t_j are time levels distributed uniformly over $[0, t_{\text{final}}]$. The final step is to display the solution as a mesh-plot using MATLAB’s `mesh` function.

In the code, we have specified the relative error tolerance as 10^{-6} , and the same value for the absolute error tolerance on each component. (See the vector options in the code, and refer to the description of `ode45.m` in MATLAB’s help menu.)

The code in Table XV shows the computation for the Hermite method. It is straightforward to adapt it for other methods, and this is done in the

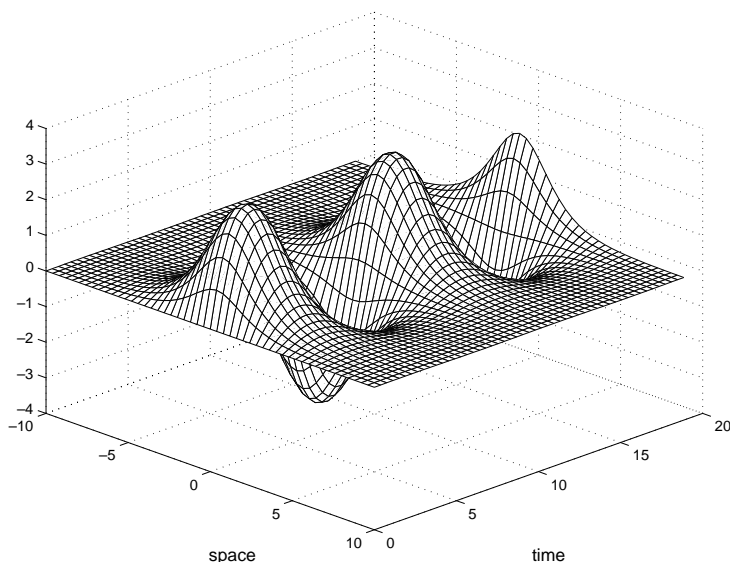


Fig. 2. Breather solution of the sine-Gordon equation.

Table XV. Solving the Sine-Gordon Equation

```

>>b = 0.545; N = 32; tfinal = 6*pi;           % Initialize
                                           parameters

>>[x,D] = herdif(N,2,b);           % Compute Hermite differentiation matrices
>>D = D(:, :, 2);                   % Extract second derivative

>>u0 = zeros(size(x));              % Compute initial conditions
>>v0 = 2*sqrt(2)*sech(x/sqrt(2));
>>w0 = [u0; v0];

>>tspan = [0:tfinal/40:tfinal];
>>options = odeset('RelTol',1e-6,'AbsTol',1e-6); % Set tolerance
>>[t,w] = ode45('sgrhs', tspan, w0, options, D); % Solve ODEs

>>u = w(:,1:N);                     % Extract u variable from solution array

>>mesh(x,t,u);                      % Generate a mesh plot of u

```

script file `sineg.m`. To use the sinc method only the line `[x,D] = herdif(N,2,b)` needs to be modified to `[x,D] = sincdif(N,2,h)`. To implement the Fourier method on a truncated domain one would compute the Fourier second-derivative matrix with `[x,D] = fourdif(N,2)`. This would assume a periodic domain $x \in [0, 2\pi]$, which should be rescaled to a large interval $[-L, L]$ by the change of variables $x \longleftrightarrow L(x - \pi)/\pi$; for details, we refer to `sineg.m`. The half-period L , like the scaling parameter

Table XVI. Errors in the Numerical Solution of the Sine-Gordon Equation at Time $t = 6\pi$

Method	Hermite	Sinc	Fourier
Parameter (near optimal)	$b = 0.545$	$h = 0.795$	$L = 12.4$
Error	4.9(-5)	2.5(-4)	5.1(-4)

b in the Hermite method and the step size h in the sinc method, may be adjusted to optimize accuracy.

Table XVI shows the maximum absolute error at the gridpoints with each of the Hermite, sinc, and Fourier methods, with $N = 32$. The time level was arbitrarily chosen to be $t = 6\pi$ (which is the final time level displayed in Figure 2). The free parameters were determined empirically, by varying their values over a certain range and comparing the numerical solution with the exact solution (57). The three methods give results that are accurate to within an order of magnitude of each another. The mesh plots of the three numerical solutions were indistinguishable from the actual solution shown in Figure 2.

In these computations stiffness was not a concern. For each of the Hermite, sinc, and Fourier methods the Runge-Kutta-Fehlberg algorithm executed between 610 and 620 function evaluations as part of about 100 successful steps and 3 or 4 failed attempts on the interval $0 \leq t \leq 6\pi$. If stiffness becomes a problem for larger N , the use of `ode15s.m` instead of `ode45.m` should be considered; see Shampine and Reichelt [1997].

The code of Table XV may be used to solve more challenging problems, perhaps involving moving or interacting solitons, or it may be modified to solve other nonlinear evolution equations. It may also serve as basis for the comparison of various numerical methods for solving PDEs such as these.

5.5 The Orr-Sommerfeld Equation

The Orr-Sommerfeld equation, derived in 1907, governs the linear stability of a two-dimensional shear flow [Drazin and Reid 1981]. For the case of flow between two plates (Poiseuille flow), it can be put in the form

$$R^{-1}(y'''' - 2y'' + y) - 2iy - i(1 - x^2)(y'' - y) = c(y'' - y), \quad (59)$$

subject to the boundary conditions

$$y(\pm 1) = y'(\pm 1) = 0. \quad (60)$$

This is an eigenvalue problem for the unknown complex wave speed, c , and the normal perturbation velocity, $y(x)$. If there is an eigenvalue with positive real part, then the flow is linearly unstable. The constant R is the Reynolds number, which is inversely proportional to the viscosity.

An accurate solution to the Orr-Sommerfeld equation was first obtained in 1971 by S. Orszag, who used a Chebyshev tau method [Orszag 1971].

Table XVII. Computing the Eigenvalues of the Orr-Sommerfeld Equation

```

>>N = 64; R = 1e4; i = sqrt(-1);           % Initialize parameters

>>[x,DM] = chebdif(N,2);                     % Compute second derivative
>>D2 = D(2:N-1,2:N-1,2);                     % Enforce boundary conditions

>>[x,D4] = cheb4c(N);                         % Compute fourth derivative
>>I = eye(size(D4));                          % Identity matrix

>>A = (D4-2*D2+I)/R-2*i*I-i*diag(1-x.^2)*(D2-I); % Set up A and
                                                    % B matrices
>>B = D2-I;

>>e = eig(A,B);                               % Compute eigenvalues

```

Here we use the differentiation matrix approach considered in Huang and Sloan [1994].

Let $D^{(4)}$ be the fourth-derivative Chebyshev matrix that implements the clamped boundary conditions (60), and let $D^{(2)}$ be the second-derivative Chebyshev matrix with boundary conditions $y(\pm 1) = 0$. Note that different boundary conditions are employed for these matrices—this approach eliminates spurious eigenvalues [Huang and Sloan 1994].

The discretized Orr-Sommerfeld equation has the form

$$A\mathbf{y} = cB\mathbf{y}, \quad (61)$$

where

$$A = R^{-1}(D^{(4)} - 2D^{(2)} + I) - 2iI - i \operatorname{diag}(1 - x_k^2)(D^{(2)} - I), \quad B = D^{(2)} - I.$$

The MATLAB program for solving the eigenvalue problem is given in Table XVII.

Orszag computed the eigenvalue with greatest real part for $R = 10^4$ to be $c_1 = 0.00373967 - 0.2375265i$. The code in Table XVII computes this eigenvalue to about two significant places when $N = 32$ and to full precision when $N \geq 50$.

The Chebyshev method has also been applied to investigate stability of streamwise streaks [Reddy et al. 1998; Waleffe 1995]. In this case a system of differential eigenvalue problems must be solved.

5.6 Extensions

The applications outlined above can be extended in various directions, two of which we outline here. The first is the construction of spectral methods based on rational interpolants; the second is the extension to two-dimensional problems.

Consider the rational weight function

$$\alpha(x) = \frac{1}{(x - p_1)(x - p_2) \dots (x - p_M)},$$

where the poles p_j are chosen outside the interval $[a, b]$. (If they are complex they should appear as conjugate pairs to ensure that $\alpha(x)$ is real.) Then the interpolant (1) is nothing but a rational interpolant with pre-assigned poles. It is a straightforward matter to use `poldif.m` to create the corresponding differentiation matrices.

The question is how to pick the parameters p_j optimally. If the function $f(x)$ that one is trying to approximate has poles in the neighborhood of $[a, b]$, then the parameters p_j should be selected to coincide with these poles. If the function has no other singularities in this neighborhood, the rational interpolation process converges rapidly. The problem is that when solving differential equations the solution is unknown, and so too are the poles. To estimate the location of the poles a priori, asymptotic techniques such as boundary layer analysis or WKB analysis may be used. This strategy has been used successfully in Weideman [1999] to solve a boundary layer problem as well as a Sturm-Liouville problem.

Another possible application of our MATLAB suite is the solution of problems in two space dimensions. Consider the case of a function $f(x, y)$ defined on $[-1, 1] \times [-1, 1]$. Suppose there are four collocation points (x_1, y_1) , (x_1, y_2) , (x_2, y_1) , (x_2, y_2) , with $x_1 \neq x_2$, $y_1 \neq y_2$. The polynomial interpolant is

$$f(x, y) \approx p(x, y) = \sum_{i=1}^2 \sum_{j=1}^2 \phi_i(x) \phi_j(y) f_{ij},$$

where $f_{ij} = f(x_i, y_j)$, and where the $\{\phi_i\}$ are the usual Lagrange interpolating polynomials for one dimension with nodes $\{x_1, x_2\}$ (or equivalently $\{y_1, y_2\}$).

Suppose we wish to compute the first derivative in x . Differentiating the above expression and evaluating at the gridpoints, we have

$$f_x(x_\ell, y_m) \approx \sum_{i=1}^2 \sum_{j=1}^2 \phi_i^{(1)}(x_\ell) \phi_j(y_m) f_{ij}.$$

The derivative operator can be represented by a matrix. Let us enumerate the discretization points by row in the x direction, and then create the data vector:

$$\mathbf{f} = [f_{11} \ f_{21} \ f_{12} \ f_{22}]^T.$$

Let $\mathbf{f}^{(1,0)}$ denote the vector of approximate first-derivative values with the same ordering. The differentiation process can then be written as

$$\mathbf{f}^{(1,0)} = D^{(1,0)}\mathbf{f},$$

where

$$D^{(1,0)} = \begin{bmatrix} \phi_1^{(1)}(x_1) & \phi_2^{(1)}(x_1) & 0 & 0 \\ \phi_1^{(1)}(x_2) & \phi_2^{(1)}(x_2) & 0 & 0 \\ 0 & 0 & \phi_1^{(1)}(x_1) & \phi_2^{(1)}(x_1) \\ 0 & 0 & \phi_1^{(1)}(x_2) & \phi_2^{(1)}(x_2) \end{bmatrix}.$$

Here we have used the fact that $\phi_j(y_m) = \delta_{jm}$ to simplify the matrix entries. Note that $D^{(1,0)}$ is a block diagonal matrix with the first-derivative matrix for one dimension on the diagonal. In other words, it is a Kronecker tensor product (for which MATLAB has a built-in function `kron`) of the identity matrix and $D^{(1)}$. This obviously generalizes to the case of more than four collocation points. The matrix $D^{(1,0)}$ corresponding to N^2 points can therefore be computed as follows

```
>>[x,D]=chebdif(N,1); % First derivative matrix in one dimension
>>Dl0=kron(eye(N,N),D); % Partial derivative with respect to x
```

For the matrix corresponding to the partial derivative with respect to y , we need only interchange the order of the entries in the Kronecker product, i.e., `D01=kron(D,eye(N,N))`. One can generalize these ideas to compute the matrix corresponding to $(\partial^{\ell+k})/(\partial x^\ell \partial y^k)$, namely `Dlk=kron(Dk,Dl)`, where `Dl` and `Dk` are the matrices for the ℓ th and k th derivatives in one dimension.

It is not difficult to extend these ideas to incorporate boundary conditions and to mix different discretizations in the x and y directions, such as Chebyshev and Fourier.

A disadvantage of this approach for solving two-dimensional problems is that the order of the matrices are $N^2 \times N^2$, which becomes large quickly. It might be more appropriate to create routines for applying the differentiation matrix without explicitly creating it, and then use iterative methods to solve the linear systems or eigenvalue problems.

APPENDIX

SUMMARY OF MATLAB FUNCTIONS IN THE SUITE

The Differentiation Matrix Suite is available at <http://ucs.orst.edu/~weidemaj/differ.html> and at <http://www.mathworks.com/support/ftp/diffeqv5.shtml> in the *Differential Equations* category of the Mathworks user-contributed (MATLAB 5) M-file repository.

(I) Differentiation Matrices (Polynomial Based)

- (1) `poldif.m`: General differentiation matrices
- (2) `chebdif.m`: Chebyshev differentiation matrices
- (3) `herdif.m`: Hermite differentiation matrices
- (4) `lagdif.m`: Laguerre differentiation matrices

(II) Differentiation Matrices (Nonpolynomial)

- (1) `fourdif.m`: Fourier differentiation matrices
- (2) `sincdif.m`: Sinc differentiation matrices

(III) Boundary Conditions

- (1) `cheb2bc.m`: Chebyshev second-derivative matrix incorporating Robin conditions
- (2) `cheb4c.m`: Chebyshev fourth-derivative matrix incorporating clamped conditions

(IV) Interpolation

- (1) `polint.m`: Barycentric polynomial interpolation at arbitrary distinct nodes
- (2) `chebint.m`: Barycentric polynomial interpolation at Chebyshev nodes
- (3) `fourint.m`: Barycentric trigonometric interpolation at equidistant nodes

(V) Transform-Based Derivatives

- (1) `chebdifft.m`: FFT-based Chebyshev derivative
- (2) `fourdifft.m`: FFT-based Fourier derivative
- (3) `sincdifft.m`: FFT-based sinc derivative

(VI) Roots of Orthogonal Polynomials

- (1) `legroots.m`: Roots of Legendre polynomials
- (2) `lagroots.m`: Roots of Laguerre polynomials
- (3) `herroots.m`: Roots of Hermite polynomials

(VII) Examples

- (1) `cerfa.m`: Function file for computing the complementary error function—boundary condition (a) in (52) is used
- (2) `cerfb.m`: Same as `cerfa.m`, but boundary condition (b) in (52) is used
- (3) `matplot.m`: Script file for plotting the characteristic curves of Mathieu's equation
- (4) `ce0.m`: Function file for computing the Mathieu cosine-elliptic function
- (5) `sineg.m`: Script file for solving the sine-Gordon equation
- (6) `sgrhs.m`: Function file for computing the right-hand side of the sine-Gordon system
- (7) `schrod.m`: Script file for computing the eigenvalues of the Schrödinger equation
- (8) `orrsom.m`: Script file for computing the eigenvalues of the Orr-Sommerfeld equation

ACKNOWLEDGMENTS

We acknowledge the Department of Mathematics, Oregon State University, where a large part of J. A. C. Weideman's work was completed with financial support from an NSF grant. We would also like to thank the

Department of Computer Science, University of Utah, for their hospitality during a sabbatical visit 1996–97, and in particular Frank Stenger for many discussions on the sinc method. The Fortran computations mentioned in Section 2 were performed on a Cray T90; time was provided through a grant to SCR from the National Partnership for Advanced Computational Infrastructure. Correspondence with Peter Acklam concerning the subtleties of efficient coding in MATLAB was very helpful, and two anonymous referees provided input that lead to substantial improvements in the original paper.

REFERENCES

- ABRAMOWITZ, M. AND STEGUN, I. E. 1964. *Handbook of Mathematical Functions*. National Bureau of Standards, Washington, DC.
- ALHARGAN, F. A. 1996. A complete method for the computations of Mathieu characteristic numbers of integer orders. *SIAM Rev.* 38, 2, 239–255.
- BALTENSPERGER, R. AND BERRUT, J. P. 1999. The errors in calculating the pseudospectral differentiation matrices for Chebyshev-Gauss-Lobatto points. *Comput. Math. Appl.* 37, 1, 41–48.
- BAYLISS, A., CLASS, A., AND MATKOWSKY, B. J. 1995. Roundoff error in computing derivatives using the Chebyshev differentiation matrix. *J. Comput. Phys.* 116, 2 (Feb.), 380–383.
- BERRUT, J. P. 1989. Barycentric formulae for cardinal (sinc-) interpolants. *Numer. Math.* 54, 703–718.
- BOYD, J. P. 1984. The asymptotic coefficients of Hermite function series. *J. Comput. Phys.* 54, 382–410.
- BOYD, J. P. 1989. *Chebyshev and Fourier Spectral Methods*. Springer-Verlag, Berlin, Germany.
- BREUER, K. S. AND EVERSON, R. M. 1992. On the errors incurred calculating derivatives using Chebyshev polynomials. *J. Comput. Phys.* 99, 1 (Mar.), 56–67.
- CANUTO, C., HUSSAINI, M. Y., QUARTERONI, A., AND ZANG, T. A. 1988. *Spectral Methods in Fluid Dynamics*. Springer-Verlag, Berlin, Germany.
- COSTA, B. AND DON, W. S. 1999. Pseudopack 2000. See <http://www.labma.ufrj.br/~bcosta/PseudoPack2000/Main.html>.
- DON, W. S. AND SOLOMONOFF, A. 1995. Accuracy and speed in computing the Chebyshev collocation derivative. *SIAM J. Sci. Comput.* 16, 6 (Nov.), 1253–1268.
- DRAZIN, P. G. AND JOHNSON, R. S. 1989. *Solitons: An Introduction*. Cambridge University Press, New York, NY.
- DRAZIN, P. G. AND REID, W. H. 1981. *Hydrodynamic Stability*. Cambridge University Press, New York, NY.
- DUTT, A., GU, M., AND ROKHLIN, V. 1996. Fast algorithms for polynomial interpolation, integration, and differentiation. *SIAM J. Numer. Anal.* 33, 5, 1689–1711.
- EGGERT, N., JARRATT, M., AND LUND, J. 1987. Sinc function computation of the eigenvalues of Sturm-Liouville problems. *J. Comput. Phys.* 69, 1 (Mar. 1), 209–229.
- FLÜGGE, S. 1971. *Practical Quantum Mechanics I*. Springer-Verlag, Berlin, Germany.
- FORNBERG, B. 1996. *A Practical Guide to Pseudospectral Methods*. Cambridge University Press, New York, NY.
- FUNARO, D. 1992. *Polynomial Approximation of Differential Equations*. Springer-Verlag, Berlin, Germany.
- FUNARO, D. 1993. Fortran routines for spectral methods. (available via anonymous FTP at <ftp.ian.pv.cnr.it> in pub/splib)
- GOTTLIEB, D., HUSSAINI, M. Y., AND ORSZAG, S. A. 1984. Theory and applications of spectral methods. In *Spectral Methods for Partial Differential Equations*, R. Voigt, D. Gottlieb, and M. Hussaini, Eds. 1–54.
- GREENGARD, L. 1991. Spectral integration and two-point boundary value problems. *SIAM J. Numer. Anal.* 28, 4 (Aug.), 1071–1080.

- GREENGARD, L. AND ROKHLIN, V. 1991. On the numerical solution of two-point boundary value problems. *Comm. Pure Appl. Math.* 44, 419–452.
- HENRICI, P. 1982. *Essentials of Numerical Analysis with Pocket Calculator Demonstrations*. John Wiley and Sons, Inc., New York, NY.
- HENRICI, P. 1986. *Applied and Computational Complex Analysis: Discrete Fourier Analysis—Cauchy Integrals—Construction of Conformal Maps—Univalent Functions*. Vol. 3. John Wiley and Sons, Inc., New York, NY.
- HUANG, W. AND SLOAN, D. M. 1992. The pseudospectral method for third-order differential equations. *SIAM J. Numer. Anal.* 29, 6 (Dec.), 1626–1647.
- HUANG, W. AND SLOAN, D. M. 1993. A new pseudospectral method with upwind features. *IMA J. Num. Anal.* 13, 413–430.
- HUANG, W. AND SLOAN, D. M. 1994. The pseudospectral method for solving differential eigenvalue problems. *J. Comput. Phys.* 111, 2 (Apr.), 399–409.
- ORSZAG, S. A. 1971. An accurate solution of the Orr-Sommerfeld equation. *J. Fluid Mech.* 50, 689–703.
- PRYCE, J. D. 1993. *Numerical Solution of Sturm-Liouville Problems*. Monographs on Numerical Analysis. Oxford University Press, Oxford, UK.
- REDDY, S. C., SCHMID, P. J., BAGGETT, J. S., AND HENNINGSON, D. S. 1998. On stability of streamwise streaks and transition thresholds in plane channel flows. *J. Fluid Mech.* 365, 269–303.
- SCHONFELDER, J. L. 1978. Chebyshev expansions for the error and related functions. *Math. Comput.* 32, 1232–1240.
- SCHOOMBIE, S. W. AND BOTHA, J. F. 1981. Error estimates for the solution of the radial Schrödinger equation by the Rayleigh-Ritz finite element method. *IMA J. Num. Anal.* 1, 47–63.
- SHAMPINE, L. F. AND REICHEL, M. W. 1997. The MATLAB ODE suite. *SIAM J. Sci. Comput.* 18, 1, 1–22.
- SHARMA, A. 1972. Some poised and nonpoised problems of interpolation. *SIAM Rev.* 14, 129–151.
- STENGER, F. 1993. *Numerical Methods Based on Sinc and Analytic Functions*. Springer-Verlag, New York, NY.
- STRANG, G. 1986. A proposal for Toeplitz matrix calculations. *Stud. Appl. Math.* 74, 2 (Apr.), 171–176.
- TADMOR, E. 1986. The exponential accuracy of Fourier and Chebyshev differencing methods. *SIAM J. Numer. Anal.* 23, 1 (Feb.), 1–10.
- TANG, T. 1993. The Hermite spectral method for Gaussian-type functions. *SIAM J. Sci. Comput.* 14, 3 (May), 594–606.
- TANG, T. AND TRUMMER, M. R. 1996. Boundary layer resolving pseudospectral methods for singular perturbation problems. *SIAM J. Sci. Comput.* 17, 2, 430–438.
- THE MATHWORKS, INC. 1998. MATLAB 5.2.
- TREFETHEN, L. N. 2000. *Spectral Methods in MATLAB*. SIAM, Philadelphia, PA.
- WALEFFE, F. 1995. Hydrodynamic stability and turbulence: Beyond transients to a self-sustaining process. *Stud. Appl. Math.* 95, 319–343.
- WEIDEMAN, J. A. C. 1999. Spectral methods based on nonclassical orthogonal polynomials. In *Approximations and Computation of Orthogonal Polynomials*, W. Gautschi, G. Golub, and G. Opfer, Eds. Birkhäuser, Basel, 239–251.
- WEIDEMAN, J. A. C. AND TREFETHEN, L. N. 1988. The eigenvalues of second-order spectral differentiation matrices. *SIAM J. Numer. Anal.* 25, 1279–1298.
- WELFERT, B. D. 1997. Generation of pseudospectral differentiation matrices I. *SIAM J. Numer. Anal.* 34, 4, 1640–1657.
- WIMP, J. 1984. *Computation with Recurrence Relations*. Pitman Publishing, Inc., Marshfield, MA.

Received: August 1998; revised: March 1999 and February 2000; accepted: March 2000