SIAM J. Sci. Comput.
Vol. 25, No. 5, pp. 1743–1770
© 2004 Society for Industrial and Applied Mathematics

AN EXTENSION OF MATLAB TO CONTINUOUS FUNCTIONS AND OPERATORS*

ZACHARY BATTLES† AND LLOYD N. TREFETHEN†

Abstract. An object-oriented MATLAB system is described for performing numerical linear algebra on continuous functions and operators rather than the usual discrete vectors and matrices. About eighty MATLAB functions from `plot` and `sum` to `svd` and `cond` have been overloaded so that one can work with our "chebfun" objects using almost exactly the usual MATLAB syntax. All functions live on $[-1, 1]$ and are represented by values at sufficiently many Chebyshev points for the polynomial interpolant to be accurate to close to machine precision. Each of our overloaded operations raises questions about the proper generalization of familiar notions to the continuous context and about appropriate methods of interpolation, differentiation, integration, zerofinding, or transforms. Applications in approximation theory and numerical analysis are explored, and possible extensions for more substantial problems of scientific computing are mentioned.

Key words. MATLAB, Chebyshev points, interpolation, barycentric formula, spectral methods, FFT

AMS subject classifications. 41-04, 65D05

DOI. 10.1137/S1064827503430126

1. Introduction.

Numerical linear algebra and functional analysis are two faces of the same subject, the study of linear mappings from one vector space to another. But it could not be said that mathematicians have settled on a language and notation that blend the discrete and continuous worlds gracefully. Numerical analysts favor a concrete, basis-dependent matrix-vector notation that may be quite foreign to the functional analysts. Sometimes the difference may seem very minor between, say, expressing an inner product as (u, v) or as $u^T v$. At other times it seems more substantial, as, for example, in the case of Gram–Schmidt orthogonalization, which a numerical analyst would interpret as an algorithm, and not necessarily the best one, for computing a matrix factorization $A = QR$. Though experts see the links, the discrete and continuous worlds have remained superficially quite separate; and, of course, sometimes there are good mathematical reasons for this, such as the distinction between spectrum and eigenvalues that arises for operators but not matrices.

The purpose of this article is to explore some bridges that may be built between discrete and continuous linear algebra. In particular we describe the "chebfun" software system in object-oriented MATLAB, which extends many MATLAB operations on vectors and matrices to functions and operators. This system consists of about eighty M-files taking up about 100KB of storage. It can be downloaded from http://www.comlab.ox.ac.uk/oucl/work/nick.trefethen/, and we assure the reader that going through this paper with a computer at hand is much more fun.

Core MATLAB contains hundreds of functions. We have found that this collection has an extraordinary power to focus the imagination. We simply asked ourselves, for one MATLAB operation after another, what is the "right" analogue of this operation in the continuous case? The question comes in two parts, conceptual and algorithmic. What should each operation mean? And how should one compute it?

*Received by the editors June 18, 2003; accepted for publication (in revised form) November 16, 2003; published electronically May 20, 2004.
http://www.siam.org/journals/sisc/25-5/43012.html

†Computing Laboratory, Oxford University, Wolfson Building, Parks Road, Oxford OX13QD, England (battles@comlab.ox.ac.uk, LNT@comlab.ox.ac.uk).

On the conceptual side, we have made a basic restriction for simplicity. We decided that our universe will consist of functions defined on the interval $[-1, 1]$. Thus a vector `v` in MATLAB becomes a function $v(x)$ on $[-1, 1]$, and it is immediately clear what meaning certain MATLAB operations must now take on, such as

$$\texttt{norm(v)} = \left(\int_{-1}^{1} v^2 dx \right)^{1/2}. \tag{1.1}$$

(Throughout this article we assume that the reader knows MATLAB, and we take all quantities to be real except where otherwise specified, although our software system realizes at least partially the generalization to the complex case.) Of course, if our system is to evolve into one for practical scientific computing, it will eventually have to be generalized beyond $[-1, 1]$.

And how might one implement such operations? Here again we have made a very specific choice. *Every function is represented by the polynomial interpolant through its values in sufficiently many Chebyshev points for accuracy close to machine precision.* By Chebyshev points we mean the numbers

$$x_j = \cos \frac{\pi j}{N}, \qquad 0 \le j \le N, \tag{1.2}$$

for some $N \ge 0$. (For $N = 0$ we take $x_0 = 1$.) We evaluate the polynomial interpolant of data in these points by the fast, stable barycentric formula first published by Salzer [1, 11, 23]. Implementing numerical operations involving such interpolants raises fundamental questions of numerical analysis. The right way to evaluate (1.1), for example, is by Clenshaw–Curtis quadrature implemented with a fast Fourier transform (FFT) [4], and indeed many of our methods utilize the FFT to move back and forth between Chebyshev grid functions on $[-1, 1]$ and sets of coefficients of expansions in Chebyshev polynomials. But these matters are all in principle invisible to the user, who sees only that familiar operations like `+` and `norm` and `sin` have been overloaded to operations that give the right answers, usually to nearly machine precision, for functions instead of vectors.

From functions, the next step is matrices whose columns are functions. For these "column maps" [6] or "matrices with continuous columns" [28] or "quasi matrices" [25], we define and implement operations such as matrix-vector product, QR factorization, singular value decomposition (SVD), and least-squares solution of overdetermined systems of equations (section 9). Thus our overloaded "`qr`" function, for example, can be used to generate orthogonal polynomials. At the end we mention preliminary steps to treat the case of "matrices" that are continuous in both directions, which can be regarded as bivariate functions on $[-1, 1] \times [-1, 1]$ or as integral operators.

2. Chebfuns and barycentric interpolation. Our fundamental objects are MATLAB structures called *chebfuns*, which are manipulated by overloaded variants of the usual MATLAB functions for vectors. "Under the hood," the data defining a chebfun consist of a set of numbers $f_0, \ldots, f_N$ for some $N \ge 0$, and each operation is defined via polynomial interpolation of the values $\{f_j\}$ at the Chebyshev points $\{x_j\}$ defined by (1.2). The interpolation is carried out numerically by the fast, stable barycentric formula developed by Salzer [23] for these interpolation points:

$$p(x) = \sum_{j=0}^{N} \frac{w_j}{x - x_j} f_j \Bigg/ \sum_{j=0}^{N} \frac{w_j}{x - x_j} \tag{2.1}$$

with

$$w_j = \begin{cases} (-1)^j/2, & j = 0 \text{ or } j = N, \\ (-1)^j & \text{otherwise.} \end{cases} \tag{2.2}$$

For an introduction to barycentric interpolation, see [1], and for a proof of its numerical stability, see [11]. Polynomial interpolation has a spotty reputation, but this is a result of difficulties if one uses inappropriate sets of points (e.g., equispaced) or unstable interpolation formulas (e.g., Newton with improper ordering). For Chebyshev points and the barycentric formula, polynomial interpolants have almost ideal properties, at least for approximating functions that are smooth. We summarize some of the key facts in the following theorem.

THEOREM 2.1. *Let f be a continuous function on $[-1,1]$, p_N its degree N polynomial interpolant in the Chebyshev points* (1.2)*, and p_N^* its best approximation on* $[-1,1]$ *in the norm $\|\cdot\| = \|\cdot\|_\infty$. Then*

(i) $\|f - p_N\| \le (2 + \frac{2}{\pi}\log N)\|f - p_N^*\|$;

(ii) *if f has a kth derivative in $[-1,1]$ of bounded variation for some $k \ge 1$, $\|f - p_N\| = O(N^{-k})$ as $N \to \infty$;*

(iii) *if f is analytic in a neighborhood of $[-1,1]$, $\|f - p_N\| = O(C^N)$ as $N \to \infty$ for some $C < 1$; in particular we may take $C = 1/(M+m)$ if f is analytic in the closed ellipse with foci ± 1 and semimajor and semiminor axis lengths $M \ge 1$ and $m \ge 0$.*

Proof. It is a standard result of approximation theory that a bound of the form (i) holds with a constant $1 + \Lambda_N$, where Λ_N is the *Lebesgue constant* for the given set of interpolation points, i.e., the ∞-norm of the mapping from data in these points to their degree N polynomial interpolant on $[-1,1]$ [18]. The proof of (i) is completed by noting that for the set of points (1.2), Λ_N is bounded by $1 + (2/\pi)\log N$ [3]. Result (ii) can be proved by transplanting the interpolation problem to one of Fourier (= trigonometric) interpolation on an equispaced grid and using the Poisson (= aliasing) formula together with the fact that a function of bounded variation has a Fourier transform that decreases at least inverse-linearly; see Theorem 4(a) of [27]. One might think that this result would be a standard one in the literature, but its only appearance that we know of is as Corollary 2 in [16]. Condition (iii) is a standard result of approximation theory, due originally to Bernstein (see, e.g., [14, Thm. 5.7] or [27, Thm. 5.6]). It can be proved, for example, by the Hermite integral formula of complex analysis [7, 30]. ∎

It follows from condition (i) of Theorem 2.1 that the Chebyshev interpolant of a function f on $[-1,1]$ is within a factor 10 of the best approximation if $N < 10^5$, a factor 100 if $N < 10^{66}$. Thus Chebyshev interpolants are *near-best*. Following familiar terminology in certain circles, we may say that conditions (ii) and (iii) establish that they are also *spectrally accurate*.

In our object-oriented MATLAB system, chebfuns are implemented as a class of objects with directory `@chebfun`. The most fundamental operation one may carry out is to create a chebfun by calling the constructor program `chebfun.m`. (From now on, we omit the "`.m`" extensions.) For example, we might write

```
>> f = chebfun('x.^3')
```

By evaluating x^3 in various points, the MATLAB code then determines automatically, with high, though inevitably not perfect, reliability, how large N must be to represent this function to a normwise relative accuracy of about 13 digits: in this case, $N = 3$. The output returned by MATLAB from the above command,

generated by the `@chebfun` code `display`, is a list of the function values at the points $x_0 = 1, \ldots, x_N = -1$:

```
ans = column chebfun
    1.0000
    0.1250
   -0.1250
   -1.0000
```

If we type `whos`, we get

```
Name      Size             Bytes  Class
f        -3x1                538  chebfun object
```

Note that we have adopted the convention of taking the "row dimension" of a column vector chebfun to be the negative of its grid number N. This may seem gimmicky at first, but one quickly comes to appreciate the convenience of being able to spot continuous dimensions at a glance. Similarly, we find

```
>> size(f)
ans =
    -3     1

>> length(f)
ans = -3
```

and thus, for example,

```
>> length(chebfun('x.^7 -3*x + 5'))
ans = -7
```

These extensions of familiar MATLAB functions are implemented by `@chebfun` functions `size` and `length`.

The function used to generate a chebfun need not be a polynomial, and indeed our chebfun constructor has no knowledge of what form the function may have. It simply evaluates the function at various points and determines a grid parameter N that is sufficiently large. Thus, for example, we may write

```
>> g = chebfun('sin(5*pi*x)');
>> length(g)
ans = -43
```

Evidently 44 Chebyshev points suffice to represent $\sin(5\pi x)$ to close to machine precision. Our constructor algorithm proceeds essentially as follows (some details omitted). On a given grid with parameter N, the Chebyshev expansion coefficients of the polynomial interpolant through the given function values are computed by means of the FFT (see section 5). A coefficient is considered to be negligible if it is smaller in magnitude than twice machine precision times the largest Chebyshev coefficient computed from the current grid. If the last two coefficients are negligible, then N is reduced to the index associated with the last nonnegligible coefficient. Otherwise, N is doubled and we start again. If convergence is not reached before $N = 2^{16}$, then the iteration stops with this value of N and a warning message is printed.

It is clear from Theorem 2.1 that if a function is not smooth, the associated chebfun may require a large value of N. For example, we find $N = 248$, 856, and 11750 for $|x|^7$, $|x|^5$, and $|x|^3$, respectively; each of these computations takes much less than 1 sec. on our workstation. For the function $|x|$ the system quits with $N = 2^{16}$

after about 1 sec. For any function, one has the option of forcing the system to use a fixed value of N by a command such as `f = chebfun('abs(x)',1000)`.

If f were a column vector in MATLAB, we could evaluate it at various indices by commands like `f(1)` or `f([1 2 3])`. For a chebfun, the appropriate analogue is that f should be evaluated at the corresponding arguments, not indices. For example, if `f` is the x^3 chebfun defined above, we get

```
>> f(5)
ans = 125.0000

>> f(-0.5)
ans = -0.1250
```

If `g` is the $\sin(5\pi x)$ chebfun, we get (after executing `format long`)

```
>> g(0:.05:.2)
ans =
   0.00000000000000
   0.70710678118655
   1.00000000000000
   0.70710678118655
   0.00000000000000
```

The chebfun system does not "know" that `g` is a sine function or that $\sin(\pi/4) = 1/\sqrt{2} \approx 0.70710678118655$, but the barycentric evaluation of the polynomial interpolant has computed the right results nonetheless.

If `f` and `g` are chebfuns of degrees N_f and N_g, then `f+g` is a chebfun of degree $N = \max\{N_f, N_g\}$. We compute `f+g` by evaluating the sum in each of the points of the Chebyshev N grid, taking advantage of the FFT as described in section 5, without automatically checking for fortuitous cancellation that might lower the degree. Differences `f-g` are handled in the same way. Following standard conventions for object-oriented programming in MATLAB, such operations are overloaded on the usual symbols `+` and `-` by programs `plus` and `minus`. For unary operations of the form `+f` and `-f`, we have codes `uplus` and `uminus`. The former is the identity operation, and the latter just negates the data defining the chebfun. The functions `real`, `imag`, and `conj` also have the obvious meanings and implementations.

Since chebfuns are analogous to vectors, the product `f*g` of chebfuns `f` and `g` is dimensionally incorrect; it yields an error message. Instead, a correct notion is pointwise multiplication `f.*g`, implemented by the function `times`. To compute `f.*g` we evaluate the product of `f` and `g` on various Chebyshev grids until a sufficiently large parameter N is found; this might be $N = N_f + N_g$, but usually it is smaller. For example, we have

```
>> f = chebfun('sin(x)');
>> length(f)
ans = -13
```

but

```
>> length(f.*f)
ans = -16
```

The chebfun system, like MATLAB, also allows for adding or multiplying a scalar with commands such as `3*f` or `3+f`, implemented by `mtimes` and `plus`, respectively.

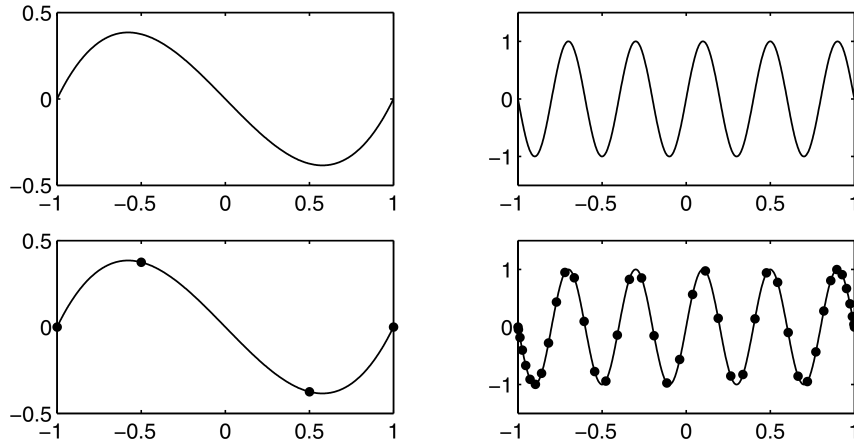

FIG. 2.1. *Results of* `plot(f)` *and* `plot(f,'.-')` *for* `f = chebfun('x.^3-x')` (*left*) *and* `f = chebfun('sin(5*pi*x)')` (*right*). *The first plot in each pair is the basic one, showing the underlying function. The second reveals the implementation by displaying the Chebyshev points at which this function is defined.*

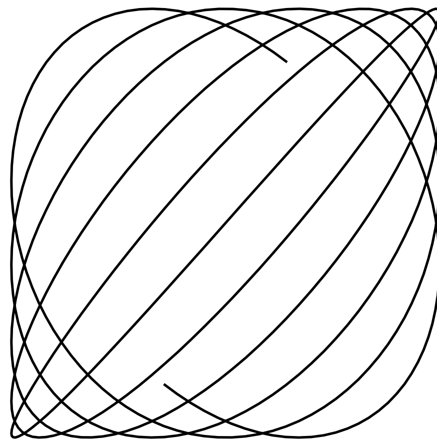

FIG. 2.2. *Result of* `plot(sin(16*x),sin(18*x))`, *assuming* `x = chebfun('x')`.

If `v` is a MATLAB vector of length k, `plot(v)` yields a broken-line plot of the k entries of v against the numbers $1, \ldots, k$. If `f` is a chebfun, the appropriate output should be a smooth curve plotted against $[-1, 1]$. This is achieved in the function `plot` by evaluating f in 1000 Chebyshev points. Additional MATLAB plotting instructions such as line types and colors and widths are passed along to the underlying MATLAB plotting engine. As a special case, what should one do with the line type `.-`, which in MATLAB normally produces dots connected by line segments? The right answer for chebfuns is surely to produce the usual curve together with dots marking the points of the underlying Chebyshev grid; see Figure 2.1.

As in MATLAB, one can also plot one chebfun against another with a command like `plot(f,g)`. An example is shown in Figure 2.2. The chebfun commands we have described are summarized in Table 2.1.

Table 2.1
Basic chebfun commands.

Typical command	M-file	Function
`f = chebfun('x')`	`chebfun.m`	Create a chebfun
`whos`	`size.m`	List variables
`size(f)`	`size.m`	Row and column dimensions
`length(f)`	`length.m`	Grid parameter N
`plot(f)`	`plot.m`	Plot against $[-1, 1]$
`plot(f,'.-')`	`plot.m`	Plot and show grid points
`semilogy(f)`	`semilogy.m`	Semilog plot
`f([.5 .6])`	`subsref.m`	Evaluate at specified points
`f+g`	`plus.m`	Sum
`+f`	`uplus.m`	Identity operator
`f-g`	`minus.m`	Difference
`-f`	`uminus.m`	Negation
`f.*g`	`times.m`	Pointwise product
`f./g`	`rdivide.m`	Pointwise division
`f.\g`	`ldivide.m`	Pointwise left division
`3+f`	`plus.m`	Scalar sum
`3*f`	`mtimes.m`	Scalar product
`f/3`	`mrdivide.m`	Scalar division
`f.^3, 3.^f`	`power.m`	Pointwise power
`real(f)`	`real.m`	Real part
`imag(f)`	`imag.m`	Imaginary part
`conj(f)`	`conj.m`	Complex conjugate

3. Elementary functions. A command like `chebfun('exp(sin(x))')` constructs a chebfun by evaluating the indicated function in various points. However, often it is more convenient to compute elementary functions of chebfuns directly, as in

```
h = exp(sin(x))
```

assuming that `x` has previously been defined by `x = chebfun('x')`. We have accordingly overloaded a number of the elementary functions to make such computations possible (Table 3.1). The underlying algorithm is the same adaptive procedure employed by the chebfun constructor, but there is a difference: `chebfun('exp(sin(x))')` evaluates the exponential at certain points determined by the sine function itself, whereas `exp(sin(x))` is constructed by evaluating the exponential at values determined by the chebfun approximation to the sine function. The procedure should be the same up to close to rounding errors, and indeed we usually find that the chebfuns are almost identical:

```
>> f = chebfun('exp(sin(x))');
>> g = exp(sin(chebfun('x')));

>> [length(f) length(g)]
ans =
   -21   -21

>> norm(f-g)
ans = 5.2011e-16
```

There are a number of MATLAB elementary functions which we have not implemented for chebfuns because they return discontinuous results: `fix`, `floor`, `ceil`, `round`, `mod`, and `rem`. We have, on the other hand, implemented certain other functions for which discontinuities are sometimes an issue: `sign`, `abs`, `sqrt`, `log`, `angle`, and division carried out by `/` or `\`. If these functions are applied to functions that

Table 3.1

Elementary functions of chebfuns. The commands marked with daggers will be unsuccessful if `f` *passes through zero.*

Typical command	M-file	Function
`exp(f)`	`exp.m`	Exponential
†`log(f)`	`log.m`	Logarithm
†`sqrt(f)`	`sqrt.m`	Square root
†`abs(f)`	`abs.m`	Absolute value
†`sign(f)`	`sign.m`	Sign
†`angle(f)`	`angle.m`	Argument
`cos(f)`	`cos.m`	Cosine
`cosh(f)`	`cosh.m`	Hyperbolic cosine
`sin(f)`	`sin.m`	Sine
`sinh(f)`	`sinh.m`	Hyperbolic sine
`tan(f)`	`tan.m`	Tangent
`tanh(f)`	`tanh.m`	Hyperbolic tangent
`erf(f)`	`erf.m`	Error function
`erfc(f)`	`erfc.m`	Complementary error function
`erfcx(f)`	`erfcx.m`	Scaled complementary error function
`erfinv(f)`	`erfinv.m`	Inverse error function

pass through zero, the results should be discontinuous (or in the case of `sqrt`, have a discontinuous derivative). Such results are not representable within the chebfun system, and, as before, the system quits with a warning message at $N = 2^{16}$.

4. Applications in approximation theory. We have already described enough of the chebfun system to enable some interesting explorations in approximation theory, which give a hint of how this system might be used in the classroom.

Gibbs phenomenon. A famous effect in approximation theory is the Gibbs phenomenon, the tendency of interpolants and approximants to oscillate near points of discontinuity [12]. We can illustrate this effect for interpolation in Chebyshev points by the command

```
f = chebfun('sign(x)',25); plot(f,'.-')
```

This command constructs the interpolant in 26 points to a function that is -1 for $x < 0$ and 1 for $x > 0$. Figure 4.1 shows the result, with the expected oscillations. (The command `max(f)` returns the number `1.2808`, showing that the overshoot is about 28%; see section 7.) The oscillations diminish away from the discontinuity, but only algebraically; the contamination is global. This is the principal drawback of global polynomial representations of functions: local irregularities have nonnegligible global consequences, in contrast to other forms of approximations such as splines with irregular meshes.

Smoothness and rate of convergence. According to Theorem 2.1(ii), the Chebyshev interpolant to the function $f(x) = |x|^5$ will have errors of order $O(N^{-5})$. We can illustrate this with the sequence

```
s = 'abs(x).^5'
exact = chebfun(s);
for N = 1:60
  e(N) = norm(chebfun(s,N)-exact);
end
loglog(e), grid on, hold on
loglog([1 60].^(-5),'--')
```

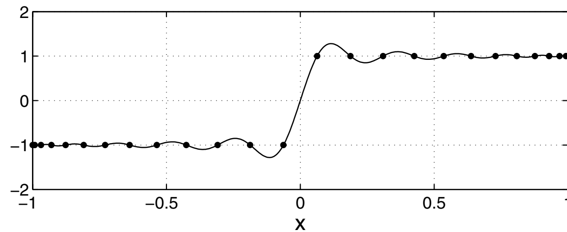

FIG. 4.1. *The Gibbs phenomenon.*

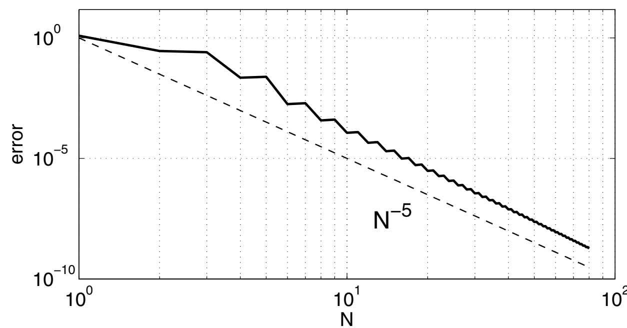

FIG. 4.2. *Fifth-order convergence for* $|x|^5$ *(loglog scale).*

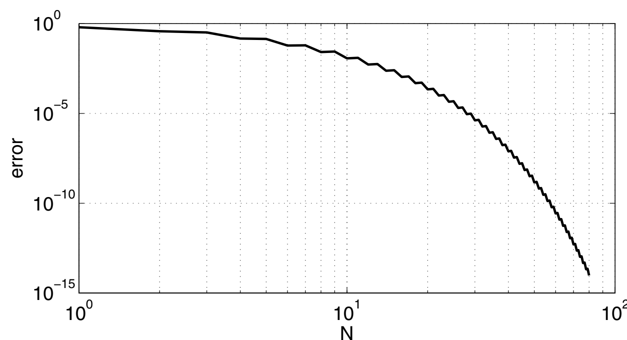

FIG. 4.3. *Geometric convergence for* $1/(1+6x^2)$.

(The function `norm`, which implements (1.1), is described in section 6.) Figure 4.2 shows the result. On the other hand if f is analytic in a neighborhood of $[-1,1]$, then by Theorem 2.1(iii) the convergence will be geometric. Figure 4.3 shows the result if in the code above, `s` is changed to `'1./(1+6^x.^2)'`. With `semilogy` instead of `loglog`, this would be a straight line, whose slope is readily derived from Theorem 2.1(iii).

Interpolation of random data. Though the chebfun system is designed for smooth functions, we can use it to illustrate the robustness of interpolation in Chebyshev points by considering random data instead of smooth. The command

```
plot(chebfun('rand(51,1)',50),'.-')
```

constructs and plots the degree 50 interpolant through data consisting of uniformly distributed random numbers in $[0,1]$ located at 51 Chebyshev points. The result,

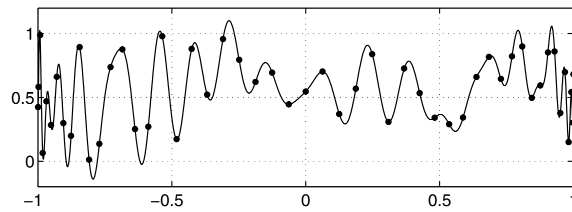

FIG. 4.4. *Degree* 50 *Chebyshev interpolation of* 51 *uniform random numbers.*

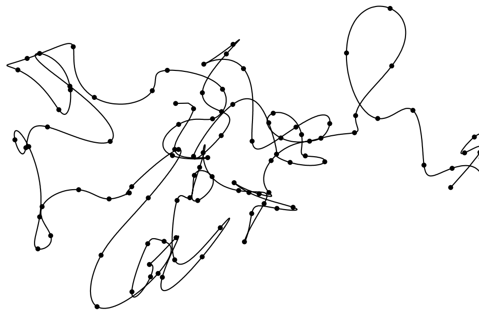

FIG. 4.5. *A two-dimensional analogue of the previous plot: the chebfun interpolant through a* 101*-point random walk.*

shown in Figure 4.4, emphasizes how well behaved the interpolation process is; any "wild oscillations" one might have expected to see are features of equispaced grids, not Chebyshev, as indeed must be the case in view of Theorem 2.1(i). The same robustness is illustrated in two dimensions in Figure 4.5, which shows the output of the commands

```
x = chebfun('cumsum(randn(101,1))',100);
y = chebfun('cumsum(randn(101,1))',100);
plot(x,y,'.-'), axis equal
```

The points are those of a random walk in the plane with x and y increments normally distributed, and the curve has x and y coordinates given by the Chebyshev interpolants through these data.

Extrapolation outside $[-1, 1]$. Polynomial interpolation in Chebyshev points is beautifully well behaved, but extrapolation from these or any other points is another matter. Suppose we again form the chebfun corresponding to $\sin(5\pi x)$ and then evaluate this function not just in $[-1, 1]$ but in a larger interval, say, $[-1.4, 1.4]$. A suitable code is

```
g = chebfun('sin(5*pi*x)');
xx = linspace(-1.4,1.4)';
plot(xx,g(xx),'.')
```

The upper half of Figure 4.6 shows that the result is a clean sine wave in $[-1, 1]$ and a bit beyond, but has large errors for $|x| > 1.3$. The lower half of the figure quantifies these errors with the command

```
error = abs(g(xx)-sin(5*pi*xx));
semilogy(xx,error,'.')
```

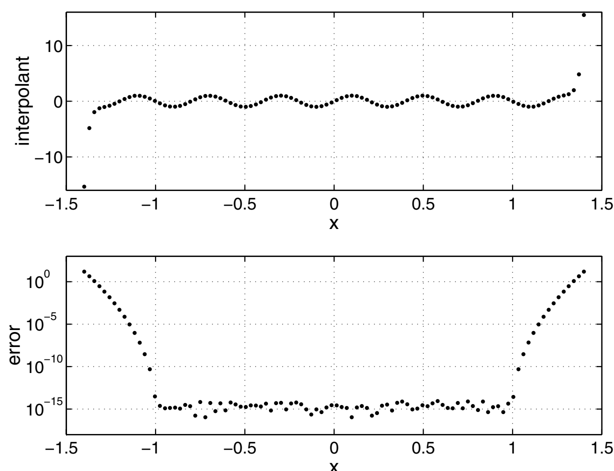

FIG. 4.6. *Extrapolation of the Chebyshev interpolant of* $\sin(5\pi x)$ *on* $[-1, 1]$ *to* $[-1.4, 1.4]$. *The errors grow rapidly for* $|x| > 1$.

Arguments related to Theorem 2.1(iii) [27, Chap. 5] show that outside $[-1, 1]$, the errors will be on the order of machine precision times $(x+\sqrt{x^2-1})^N$, a formula which matches the dots in the lower half of Figure 4.6 well. We emphasize that instability in the sense of sensitivity to rounding errors is not the essential issue here. In theory, if all the computations were performed exactly, these extrapolants would converge to $\sin(5\pi x)$ as $N \to \infty$ in any compact region of the complex plane, but this is only because $\sin(5\pi x)$ is analytic throughout the complex plane. For a function analytic just in a neighborhood of $[-1, 1]$, according to the theory of "overconvergence" [30], convergence in exact arithmetic will occur inside the largest ellipse of analyticity with foci ± 1. The problem we are up against is the ill-posed nature of analytic continuation, which, although highlighted by rounding errors, cannot be blamed upon them.

5. Chebyshev expansions and FFT.

For reasons of simplicity, we have chosen to base the chebfun system on interpolation in Chebyshev points, not expansion in Chebyshev polynomials [15, 22]. These two formulations are close cousins, however, with the same excellent approximation properties [3], and one can travel back and forth between the two at a cost of $O(N \log N)$ operations by means of the FFT. Indeed, it is a crucial feature of our package that it takes advantage of the FFT in implementing many operations, enabling us easily to handle vectors of lengths in the tens of thousands.

Suppose we are given a chebfun `p` of grid number N and wish to determine its Chebyshev expansion coefficients $\{a_k\}$:

$$p(x) = \sum_{k=0}^{N} a_k T_k(x). \tag{5.1}$$

Another way to describe the $\{a_k\}$ would be to say that they are the coefficients of the unique polynomial interpolant of degree $\leq N$ through the data values $p(x_j)$, $0 \leq j \leq N$. The chebfun system includes a function `chebpoly(f)` that computes these numbers. For example, we find

```
>> chebpoly(x.^3)
ans =
    0.2500         0    0.7500         0
```

since $x^3 = \frac{1}{4}T_3(x) + \frac{3}{4}T_1(x)$, assuming again that `x` has been previously defined by `x = chebfun('x')`. Note that, following the usual MATLAB convention, the high-order coefficients are listed first. Similarly, we have

```
>> chebpoly(exp(x))
ans =
   0.00000000000004   0.00000000000104   0.00000000002498
   0.00000000055059   0.00000001103677   0.00000019921248
   0.00000319843646   0.00004497732295   0.00054292631191
   0.00547424044209   0.04433684984866   0.27149533953408
   1.13031820798497   1.26606587775201
```

The fact that the first (leading) coefficient is just above the level of machine precision illustrates that the chebfun constructor has selected the smallest possible value of N. The `chebpoly` function is one of only three in the chebfun system that are not extensions of functions in standard MATLAB, and accordingly we distinguish it in Table 5.1 by an asterisk. The existing function in MATLAB is `poly`, which calculates the coefficients in the monomial basis of the characteristic polynomial of a matrix. We have overloaded `poly` to compute the coefficients in the monomial basis of a chebfun:

```
>> poly(x.^3)
ans =
     1     0     0     0

>> poly((1+x).^9)
ans =
     1     9    36    84   126   126    84    36     9     1

>> poly(exp(x))
ans =
   0.00000000016343   0.00000000213102   0.00000002504800
   0.00000027550882   0.00000275573484   0.00002480163504
   0.00019841269739   0.00138888887054   0.00833333333348
   0.04166666667007   0.16666666666667   0.49999999999976
   1.00000000000000   1.00000000000000
```

Note that these last coefficients, unlike their Chebyshev counterparts, do not fall as low as machine precision, reflecting the $O(2^N)$ gap between the monomial and Chebyshev bases. Polynomial coefficients in the monomial basis have the virtue of simplicity, but few numerical virtues for the manipulation of functions on $[-1, 1]$.[1] The chebfun system computes them by calling `chebpoly` first and then converting from one basis to the other with the recursion $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. We regard the outputs of `poly` as interesting for small explorations but not normally useful in serious computations.

The implementation of `chebpoly` is based on the standard three-way identification of a real variable $x \in [-1, 1]$, an angular variable $\theta = \cos^{-1} x \in [0, 2\pi]$, and a complex variable $z = e^{i\theta}$ on the unit circle in the complex plane. As described, for example, in Chapter 8 of [27], we have

$$T_k(x) = \mathrm{Re}\, z^k = \tfrac{1}{2}(z^k + z^{-k}) = \cos k\theta$$

[1] In terms of complex analysis, the number 2 comes from the fact that the logarithmic capacity of the unit disk (1) is twice that of the unit interval ($\frac{1}{2}$) [21]. The monomials are the Faber polynomials for the unit disk, and naturally adapted to approximations there, whereas the Chebyshev polynomials are the Faber polynomials for the unit interval. See [7, 30] and also [29].

Table 5.1

Operations based on FFT. The asterisks mark functions that are not overloaded variants of existing MATLAB functions. The function `chebpoly` *is called internally by many chebfun functions to speed up various computations by means of the FFT.*

Typical command	M-file	Function
`poly(f)`	`poly.m`	Polynomial coeffs. of chebfun
*`chebpoly(f)`	`chebpoly.m`	Chebyshev coeffs. of chebfun
*`chebfun([1 2 3])`	`chebfun.m`	Chebfun with given Chebyshev coeffs.

for each $n \geq 0$. Thus (5.1) can be extended to

$$p(x) \,=\, \sum_{k=0}^{N} a_k T_k(x) \,=\, \tfrac{1}{2} \sum_{k=0}^{N} a_k (z^k + z^{-k}) \,=\, \sum_{k=0}^{N} a_k \cos k\theta. \tag{5.2}$$

In the x variable, $p(x)$ is an algebraic polynomial determined by its values in the $N+1$ Chebyshev points $x_0, \ldots, x_N$. In the z variable, it is a Laurent polynomial determined by its values in the corresponding $2N$ roots of unity $z_0, \ldots, z_{2N-1}$; this function of z takes equal values at z and $\overline{z}$ and thus comprises $N+1$ independent numbers. In the θ variable, the same function is a trigonometric polynomial determined by its values in the $2N$ equally spaced points $\theta_0, \ldots, \theta_{2N+1}$ with $\theta_j = \pi j/N$; in this case the function takes equal values at θ and $2\pi - \theta$.

We use these identifications to transplant the Chebyshev coefficient problem to the more familiar problem of computing trigonometric polynomial coefficients on a regular grid in $[0, 2\pi]$—a discrete Fourier transform. This leads to the following method by which we compute `chebpoly(p)`: extend the data vector $(p(x_0), \ldots, p(x_N))$ of length $N+1$ to a vector

$$(p(x_0), \ldots, p(x_{N-1}), p(x_N), p(x_{N-1}), \ldots, p(x_1))$$

of length $2N$; apply MATLAB's `fft` function; extract the first $N+1$ entries in reverse order; and divide the first and last of these numbers by $2N$ and the others by N.[2]

The inverse of the `chebpoly` operation should be a map from Chebyshev coefficients to chebfuns. According to the principles of object-oriented programming, this process of construction of a chebfun object should be carried out by the standard constructor program `chebfun`. Accordingly, our program `chebfun` is designed to produce chebfuns from two alternative kinds of input. If the input is a string, as in `chebfun('exp(x)')`, then the adaptive process described in section 2 is executed. If the input is a numerical vector, as in `chebfun([1 2 3])`, then the input is interpreted as a set of Chebyshev coefficients ordered from highest degree down, and the appropriate chebfun is constructed by means of the FFT, inverting the process described above with the aid of MATLAB's `ifft` function.

For example, we have

```
>> chebfun([1 2 3])
ans = column chebfun
     6
     2
     2
```

[2]The ideas behind this algorithm are discussed in Chapter 8 of [27]. As mentioned there, this algorithm could be speeded up by the use of the discrete cosine transform (DCT) rather than the general complex FFT. However, although a DCT code is included in MATLAB's Signal Processing Toolbox, there is no DCT in MATLAB itself.

The three numbers displayed are the values of $T_2(x) + 2T_1(x) + 3T_0(x)$ at $x_0 = 1$, $x_1 = 0$, $x_2 = -1$. Similarly,

```
plot(chebfun([1 zeros(1,10)]))
```

constructs a plot of $T_{10}(x)$ on $[-1, 1]$ (not shown). The commands

```
r = randn(1,100000);
f = chebfun(r);
```

produce a chebfun with $N = 100000$ in a fraction of a second, and in a few more seconds we compute

```
>> norm(r)
ans = 3.157618089537492e+02

>> norm(f)
ans = 3.160389611083427e+02

>> norm(r-chebpoly(f))
ans = 1.7236e-13
```

The first and third numbers are ordinary MATLAB norms of vectors, and the middle one is the norm of a chebfun. The approximate agreement of the first two reflects the fact that for large n, $\int_{-1}^{1}(T_n(x))^2 dx \approx 1$: for example, `norm(chebfun([1 zeros(1,20)]))` yields 0.9997.

We have shown how a user may go back and forth between function values and Chebyshev coefficients with the FFT-based functions `chebpoly` and `chebfun` (see Table 5.1). Most often, however, the FFT is used by various chebfun functions (which call `chebpoly` internally) to speed up certain computations from $O(N^2)$ to $O(N \log N)$. The guiding principle is this: to determine the values of a chebfun at an arbitrary vector of points, use barycentric interpolation, but to determine values on a grid of Chebyshev points (presumably with a different parameter N' from the parameter N of f), use the FFT. An electrical engineer would call this a process of "downsampling" or "upsampling" [20]. For upsampling, with $N' \geq N$, we determine the Chebyshev coefficients of `f`, pad them with $N' - N$ zeros, and then inverse transform to the N' grid. For downsampling, with $N' < N$, we determine the Chebyshev coefficients of `f` and then fold $N - N'$ of them back into the N' coefficients to be retained by a process of aliasing. Thus, for example, if we downsample from `f` with $N = 20$ to `g` with $N' = 15$, then $a_5(\texttt{g}) = a_5(\texttt{f}) + a_{20}(\texttt{f})$. This operation is carried out in the chebfun system by the private function `prolong`, not accessible to the user.

6. Integration and differentiation. Many MATLAB operations on vectors involve sums, and their chebfun analogues involve integrals (Table 6.1). The starting point is the function `sum`. If f is a vector, `sum(f)` returns the sum of its components, whereas if f is a chebfun, we want

$$\texttt{sum(f)} = \int_{-1}^{1} f(x)dx.$$

The mathematical problem implicit in this command is that of determining the integral of the polynomial interpolant through data in a set of Chebyshev points (1.2). This is the problem solved by the method known as Clenshaw–Curtis quadrature [4], devised in 1960 and described in various references such as [5] and [13].

TABLE 6.1
Chebfun functions involving integration or differentiation.

Typical command	M-file	Function
`sum(f)`	`sum.m`	Definite integral
`cumsum(f)`	`cumsum.m`	Indefinite integral
`prod(f)`	`prod.m`	Integral of product
`cumprod(f)`	`cumprod.m`	Indefinite integral of product
`mean(f)`	`mean.m`	Mean
`norm(f)`	`norm.m`	2-norm
`var(f)`	`var.m`	Variance
`std(f)`	`std.m`	Standard deviation
`x'*y`	`mtimes.m`	Inner product
`diff(f)`	`diff.m`	Derivative

Clenshaw–Curtis quadrature becomes quite straightforward with the aid of the FFT, and that is how we implement it.[3] Given a chebfun f, we construct its Chebyshev coefficients $a_0, \ldots, a_N$ by a call to `chebpoly` and then integrate termwise using the identity

$$\int_{-1}^{1} T_k(x)dx = \begin{cases} 0, & k \text{ odd}, \\ \dfrac{2}{1-k^2}, & k \text{ even}. \end{cases} \tag{6.1}$$

The result returned from a command `sum(f)` will be exactly correct apart from the effects of rounding errors. For example, here are some integrals the reader may or may not know:

```
>> sum(sin(pi*x).^2)
ans = 1

>> sum(chebfun('1./(5+3*cos(pi*x))'))
ans = 0.50000000000000

>> sum(chebfun('abs(x).^9.*log(abs(x)+1e-100)'))
ans = -0.02000000000000
```

These three chebfuns have $N = 26$, 56, and 194. The addition of `1e-100` is needed in the last example so that $0 \log(0)$ comes out as 0 (almost) rather than NaN.

From the ability to compute definite integrals we readily obtain chebfun functions for the three statistical operations of mean, variance, and standard deviation, with functionality as defined by these expressions:

```
mean(f) = sum(f)/2
var(f) = mean((f-mean(f)).^2)
std(f) = sqrt(var(f))
```

Similarly we have

```
norm(f) = norm(f,2) = norm(f,'fro') = sqrt(sum(f.^2))
```

since in MATLAB, the default norm is the 2-norm. Thus, for example, here is an

[3]The MATLAB code `clencurt` of [27] works well for smaller values of N, but as it is based on explicit formulas rather than FFT, the work is $O(N^2)$ rather than $O(N \log N)$.

unusual computation of $\sqrt{2/5}$:

```
>> norm(x.^2)
ans = 0.63245553203368
```

The evaluation of the 1- and ∞-norm variants `norm(f,1)` and `norm(f,inf)` is more complicated and is discussed in section 7.

Inner products of functions are also defined by integrals. If `f` and `g` are chebfuns, then the chebfun system produces a result for `f'*g` equivalent to

```
f'*g = sum(f.*g)
```

All these operations are very similar in that they come down to definite integrals from -1 to 1. In addition, one may ask about indefinite integrals, and the appropriate function for this is MATLAB's cumulative sum operation `cumsum`. If `f` is a chebfun and we execute `g = cumsum(f)`, the result `g` is a chebfun equal to the indefinite integral of `f` with the particular value $g(-1) = 0$. If `f` has grid number N (a polynomial of degree N), `g` has grid number $N+1$ (degree $N+1$). The implementation here is essentially the same as that of `sum`: we compute Chebyshev coefficients by the FFT and then integrate termwise. The crucial formula is

$$\int^x T_k(x)dx \;=\; \frac{T_{k+1}(x)}{2(k+1)} - \frac{T_{k-1}(x)}{2(k-1)} + C, \quad k \geq 2 \tag{6.2}$$

(see [5, p. 195] or [15, p. 32]); for $k = 0$ or 1, the nonconstant terms on the right-hand side become $T_1(x)$ or $T_2(x)/4$, respectively. For an example of indefinite integration, recall that the error function is defined by

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

We can compare a chebfun indefinite integral to MATLAB's built-in function `erf` as follows:

```
>> f = chebfun('(2/sqrt(pi))*exp(-t.^2)');
>> erf2 = cumsum(f); erf2 = erf2 - erf2(0);
>> norm(erf2-erf(x))
ans = 4.1022e-16
```

The grid parameters for `erf(x)` and `erf2` are 21 and 23, respectively.

MATLAB also includes functions `prod` and `cumprod` for products and cumulative products. These are overloaded in the chebfun system by functions

```
prod(f) = exp(sum(log(f)))
cumprod(f) = exp(cumsum(log(f)))
```

If `f` is of one sign in $[-1, 1]$, these will generally compute accurate results with no difficulty:

```
>> prod(exp(x))
ans = 1

>> prod(exp(exp(x)))
ans = 10.48978983369024
```

If `f` changes sign, the computation will usually fail because of the discontinuity involving the logarithm of 0, as discussed in section 3.

The inverse of indefinite integration is differentiation, and in MATLAB, the inverse of `cumsum` (approximately) is `diff`: for example, `diff([1 4 4]) = [3 0]`. The overloaded function `diff` in the chebfun system converts a chebfun with grid value N to a chebfun with grid value $N-1$ corresponding to the derivative of the underlying polynomial. Again, for efficiency the computation is based on the FFT,[4] making use of the fact that if $p(x)$ has the Chebyshev expansion $p(x) = \sum_{k=0}^{N} a_k T_k(x)$ as in (5.1), then the coefficients $\{b_k\}$ of the expansion $p'(x) = \sum_{k=0}^{N-1} b_k T_k(x)$ are

$$b_{k-1} = b_{k+1} + 2ka_k, \quad 2 \le k \le N, \tag{6.3}$$

and $b_0 = b_2/2 + a_1$, with $b_N = b_{N+1} = 0$. (See (4.15)–(41.6) of [9] or [15, sect. 2.4.5].) If a second argument is supplied to the `diff` function, the chebfun is differentiated the corresponding number of times. Thus, for example,

```
>> f = sin(5*x);
>> g = diff(f,4);
>> norm(g)/norm(f)
ans = 625.0000000170987
```

The reader may enjoy comparing the following two results with their analytic values, assuming as usual that `x` is defined by `x = chebfun('x')`:

```
>> f = diff(sin(exp(x.^2)));
>> f(1)
ans = -4.95669946591073

>> g = chebfun('1./(2+x.^2)');
>> h = diff(g);
>> h(1)
ans = -0.22222222222299
```

It is not hard to figure out that if `f` is a MATLAB vector, we have

```
cumsum(diff(f)) = f(2:end)-f(1),  diff(cumsum(f)) = f(2:end)
```

(apart from rounding errors, of course), whereas if `f` is a chebfun,

```
cumsum(diff(f)) = f-f(-1),  diff(cumsum(f)) = f.
```

7. Operations based on rootfinding.

MATLAB's `roots` function finds all the roots of a polynomial given by its coefficients in the monomial basis [19]. In the chebfun system, the analogue is an overloaded function `roots` that finds all the roots of a chebfun. As we shall see, this is not an optional extra but a crucial tool for our implementations of `min`, `max`, `norm(f,1)`, and `norm(f,inf)` (Table 7.1).

What do we mean by the roots of a chebfun with grid number N? The natural answer would seem to be, all the roots of the underlying degree N polynomial. Thus we have a *global* rootfinding problem: it would not be enough, say, to find a single root in $[-1,1]$. Now the problem of finding roots of polynomials is notoriously ill-conditioned [31], but the difficulty arises when the polynomials are specified by their

[4] As with integration, one could do this in a more direct fashion by means of the differentiation matrices used in the field of spectral methods, as implemented, for example, in the program `cheb` of [27]. Again, however, this approach involves $O(N^2)$ operations.

Table 7.1
Operations based on rootfinding.

Typical command	M-file	Function
`roots(f)`	`roots.m`	roots
*`introots(f)`	`introots.m`	Roots in $[-1,1]$
`max(f)`	`max.m`	Maximum
`min(f)`	`min.m`	Minimum
`norm(f,inf)`	`norm.m`	∞-norm
`norm(f,1)`	`norm.m`	1-norm

coefficients. Here, they are specified by data in points well distributed in the interval of primary interest, a different and better-behaved situation.

What is the "right" way to find these roots? This is one of the most interesting mathematical problems we have encountered in designing the chebfun system. Other than the paper [10], we are aware of no literature at all on this problem,[5] a curious situation in view of the fact that Chebyshev expansions are abundantly useful in practice, whereas there are thousands of papers on the problem of roots of polynomials in the relatively impractical monomial basis [17]. The method we have adopted makes use of a transplantation from $[-1,1]$ to the unit circle, achieved as always with the FFT. Given a chebfun `f`, we compute its Chebyshev coefficients $\{a_k\}$ as in section 5, and then, following (5.2), we interpret these as coefficients of a Laurent polynomial $q(z) = \sum_{k=0}^{N} a_k(z^k + z^{-k})$. The roots of $q(z)$ in the z-plane are the same as those of $z^N q(z)$, apart from some bookkeeping at $z = 0$, and this is a polynomial of degree $2N$. We find the roots of $z^N q(z)$ by calling MATLAB `roots`, which makes use of the QR algorithm for eigenvalues applied to a balanced companion matrix. It has been observed that although this approach has an operation count of $O(N^3)$ rather than $O(N^2)$, it is one of the most stable methods available for computing polynomial roots [8, 26]. Because of the special structure of our coefficients, the roots we obtain come in pairs $\{z, z^{-1}\}$, which we transplant back to the x variable by the formula $x = (z+z^{-1})/2$. This gives us as output from the `roots` function a total of N numbers. In general, many of these will be spurious in the sense that they have nothing to do with the behavior of `f` on or near $[-1,1]$. Accordingly, we also provide a variant function

```
introots(f)
```

which returns only those numbers from `roots` that are contained in $[-1,1]$, in sorted order.

For example, here is a solution to a problem sometimes found as an exercise in numerical analysis textbooks:

```
>> introots(x-cos(x))
ans = 0.73908513321516
```

Since this function has just a single root, however, it does not showcase the global nature of chebfun rootfinding. We can do that with

```
>> introots(x-cos(4*x))
ans =
   0.31308830850065
  -0.53333306291483
  -0.89882621679039
```

[5] See the note added in proof.

And here are the zeros in $[0, 20]$ of the Bessel function $J_0(x)$:

```
>> f = chebfun('besselj(0,10*(1+x))');
>> 10*(1+introots(f))
ans =
   2.40482555769577
   5.52007811028631
   8.65372791291101
  11.79153443901428
  14.93091770848778
  18.07106396791093
```

These numbers are correct in every digit, except that the fifth one should end with `79` instead of `78` and the sixth should end with `92` instead of `93`.

With the ability to find roots in $[-1, 1]$, we can now perform some other more important operations. For example, if `f` is a chebfun, what is `max(f)`? The answer should be the global maximum on $[-1, 1]$, and to compute this, we compute the derivative with `diff`, find its roots, and then check the value of `f` at these points and at ± 1. The functions `min(f)` and `norm(f,inf)` are easy variants of this idea. Thus, for example,

```
>> f = x-x.^2;
>> min(f)
ans = -2

>> max(f)
ans = 0.25000000000000

>> [y,x] = max(f)
y = 0.25000000000000
x = 0.50000000000000

>> norm(f,inf)
ans = 2
```

The computation of a 1-norm by `norm(f,1)` is carried out by similar methods. The definition is

$$\|f\|_1 = \int_{-1}^{1} |f(x)| dx,$$

and since f may pass through 0, the integrand may have singularities in the form of points of discontinuity of the derivative. To evaluate the integral we use `introots` to find any roots of f in $[-1, 1]$, divide into subintervals accordingly, and integrate. Thus `norm(f,1)` reduces to `introots(f)` and `cumsum(f)`. For example, with `f` as in the example above, we find

```
>> norm(f,1)
ans = 1
```

One can combine our functions rather nicely into a little program for computing the *total variation* of a function on $[-1, 1]$:

```
function tv = tv(f)
tv = norm(diff(f),1);
```

With tv defined in this fashion we find, for example,

```
>> tv(x)
ans = 2

>> tv(sin(5*pi*x))
ans = 20.00000000000005
```

The $O(N^3)$ operation count of our roots, min, and max computations is out of line with the $O(N \log N)$ figure for most other chebfun calculations, and it makes the performance unsatisfactory for $N \gg 100$. There is not much literature on rootfinding for polynomials of high degrees, though an algorithm of Fox and Lindsey has been applied to polynomials of degrees in the hundreds of thousands [24]. We are investigating the possible use of this or other rootfinding algorithms in the chebfun system.

8. Applications in numerical analysis. Having described a major part of the chebfun system, we can now consider more applications. To begin with, let us determine some numbers of the kind that students learn how to compute in an introductory numerical analysis course. For illustration we will take the function

$$f(x) = \tan(x + \tfrac{1}{4}) + \cos(10x^2 + e^{e^x}). \tag{8.1}$$

The chebfun for this function (which turns out to have grid number $N = 77$) is constructed in less than 0.1 sec. by

```
s = 'tan(x+1/4) + cos(10*x.^2+exp(exp(x)))';
f = chebfun(s);
```

and the result of plot(f) is shown in Figure 8.1.

One topic in a numerical analysis course is quadrature. A standard approach using the built-in adaptive MATLAB code gives

```
>> quad(s,-1,1,1e-14)
ans = 0.29547767624377
```

in about 7 secs. on our workstation; MATLAB's alternative code quadl reduces this figure to 1.6 secs. In the chebfun system, the adaptation has already taken place in constructing the chebfun, and it takes just 0.02 secs. more to compute

```
>> sum(f)
ans = 0.29547767624377
```

Another basic topic is minimization. In standard MATLAB we find, for example,

```
>> opts = optimset('tolx',1e-14);
>> [x0,f0] = fminbnd(s,-1,1,opts)
x0 = -0.36185484293847
f0 = -1.09717538514564
```

From the figure it is evident that this result is a local minimum rather than the global one. We can get the latter by refining the search interval:

```
>> [x0,f0] = fminbnd(s,-1,-.5,opts)
x0 = -0.89503073635821
f0 = -1.74828014625170
```

These calculations take around 0.05 secs. In the chebfun system the min function

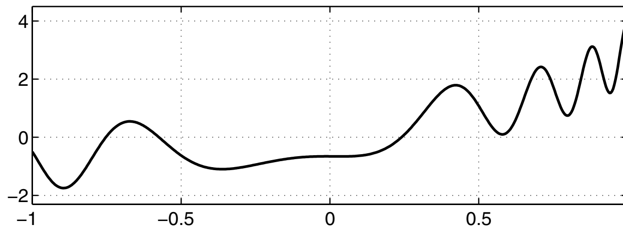

FIG. 8.1. *A function to illustrate quadrature, minimization, and rootfinding.*

gives the global minimum directly:

```
>> [f0,x0] = min(f)
f0 = -1.74828014625170
x0 = -0.89503073653153
```

The timing is somewhat worse, 0.25 secs., because of the $O(N^3)$ rootfinding operation.

Turning to rootfinding itself, in standard MATLAB we have

```
>> fzero(s,0)
ans = 0.24078098023501
```

and again it is clear from Figure 8.1 that this result is only local, since this function has three zeros in $[-1,1]$. In the chebfun system we get them all at once:

```
>> introots(f)
ans =
   0.24078098023501
  -0.57439914100933
  -0.75298521313936
```

We shall now consider two applications from the field of ordinary differential equations (ODEs). The first is in the area known as *waveform relaxation*. Suppose we are given the nonlinear, variable coefficient ODE

$$u' = e^{-2.75xu}, \quad -1 < x < 1, \qquad u(-1) = 0; \tag{8.2}$$

we have picked the constant -2.75 to make the solution interesting. The problem can be rewritten in integral form as

$$u(x) = \int_{-1}^{x} e^{-2.75su} ds, \tag{8.3}$$

which suggests that we might try to solve (8.3) by Picard iteration, i.e., successive substitution. The following code carries out this process with chebfuns:

```
x = chebfun('x');
uold = chebfun('0');
du = 1
while du > 1e-13
  u = cumsum(exp(-2.75*x.*uold));
  du = norm(u-uold);
  uold = u;
end
u(1)
```

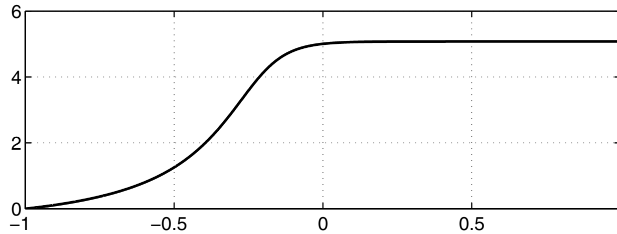

FIG. 8.2. *Solution of the ODE* (8.2) *by waveform relaxation.*

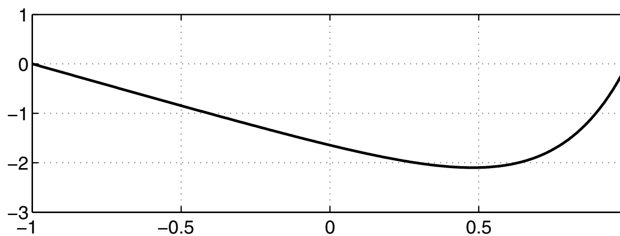

FIG. 8.3. *Solution of the boundary-value problem* (8.4) *using* `cumsum`.

It takes around 7 secs. for the solution to be found, with $u(1) = 5.07818302388064$. The resulting chebfun, with $N = 181$, is plotted in Figure 8.2. For comparison, here is a solution using standard ODE software:

```
opts = odeset('abstol',1e-14,'reltol',1e-14);
[x,u] = ode45(@f,[-1,1],0,opts);
u(end)

function f = f(x,u)
f = exp(-2.75*x*u);
```

This code prints the number 5.07818302388075 in about 2 secs., and we do not claim that the chebfun approach is in any way superior, merely that it is interesting. (We are in the process of considering extensions of our system to tackle ODEs more directly.)

A variation on this theme of waveform relaxation would be to use chebfuns iteratively to find solutions to functional differential equations. We have used this idea successfully for simple problems, such as finding a function $f(x)$ on $[-1, 1]$ such that $f(f(x)) = \tanh(x)$, and are considering possible extensions to more challenging problems such as the approximation of Feigenbaum's constant or Daubechies wavelets.

Our next example, also from the field of ODEs, concerns the linear boundary-value problem

$$u'' = e^{4x}, \quad -1 < x < 1, \quad u(-1) = u(1) = 0. \tag{8.4}$$

This problem is solved by a Chebyshev spectral method in Program 13 of [27]; the exact solution is $u(x) = [e^{4x} - x\sinh(4) - \cosh(4)]/16$. We can solve it in the chebfun system with the sequence

```
f = exp(4*x);
u = cumsum(cumsum(f));
u = u - u(1).*(1+x)/2;
```

and Figure 8.3 shows the result, which is accurate to 14 digits.

9. Chebfun quasi matrices. We are finally ready to move from vectors to matrices (Table 9.1). Specifically, in this section we consider "matrices" that consist of a discrete set of columns, each of which is a chebfun of the kind we have been discussing. Thus our matrices are continuous in the row dimension, discrete in the column dimension. The idea of such objects is surely an old one, and they are mentioned explicitly by de Boor [6] ("column maps"), Trefethen and Bau [28, p. 52] ("matrices with continuous columns"), and Stewart [25, p. 33] ("quasi matrices"). To be definite we shall use the latter term.

We can explain the idea with an example built on `x = chebfun('x')` as usual:

```
>> A = [1 x x.^2 x.^3 x.^4]
A = column chebfun
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0000    0.7071    0.5000    0.3536    0.2500
    1.0000         0         0         0         0
    1.0000   -0.7071    0.5000   -0.3536    0.2500
    1.0000   -1.0000    1.0000   -1.0000    1.0000

>> size(A)
ans =
    -4     5
```

We see that `A` is a matrix of five columns, each of which is a chebfun. In principle, each might have a different grid value N, in which case the Chebyshev point data defining `A` could be stored in a MATLAB cell array, but our implementation instead forces all columns to share the same maximal value of N and stores the data in an ordinary matrix. We can evaluate `A` at various points by commands like these:

```
>> A(3,5)
ans = 81.0000

>> A(0.5,:)
ans =
    1.0000    0.5000    0.2500    0.1250    0.0625

>> A(:,3)
ans = column chebfun
    1.0000
    0.5000
         0
    0.5000
    1.0000
```

Note that in each case the first argument is a function argument, and the second is an index.

If `c` is a column vector of the appropriate dimension (namely, `size(A,2)`), the product `A*c` should be a chebfun equal to the linear combination of the columns of `A` with the given coefficients. Thus the sequence

```
>> c = [1 0 -1/2 0 1/24]';
>> f = A*c;
>> norm(f-cos(x),inf)
ans = 0.0014
```

reveals that the maximum deviation of $\cos x$ on $[-1,1]$ from its fourth-order Taylor

Table 9.1
Operations involving quasi matrices.

Typical command	M-file	Function
`A = [f g h]`	`horzcat.m`	Construction of quasi matrix
`A([.5 .6],[2 3])`	`subsref.m`	Evaluate at specified points
`A*x`	`mtimes.m`	Quasi matrix times vector
`[Q,R] = qr(A,0)`	`qr.m`	QR factorization
`[U,S,V] = svd(A,0)`	`svd.m`	Singular value decomposition
`cond(A)`	`cond.m`	Condition number
`rank(A)`	`rank.m`	Rank
`null(A)`	`null.m`	Basis of nullspace
`pinv(A)`	`pinv.m`	Pseudoinverse
`norm(A)`	`norm.m`	Norm
`A\b`	`mldivide.m`	Least-squares solution
`A'*B`	`mtimes.m`	Inner product of quasi matrices

series approximation is about 0.0014. The figure becomes twenty times smaller for the degree 4 Chebyshev interpolant, whose coefficients are only very slightly different:

```
>> c = poly(chebfun('cos(x)',4))
c =
    0.0396         0   -0.4993         0    1.0000

>> f = A*c(end:-1:1)';
>> norm(f-cos(x),inf)
ans = 6.4809e-05
```

Many familiar matrix operations make sense for quasi matrices. For example, if A is a quasi matrix with n chebfun columns, then in the (reduced) QR decomposition $A = QR$, Q will be another quasi matrix of the same dimension whose columns are orthonormal, and R will be an $n \times n$ upper-triangular matrix. The chebfun system performs the computation by a modified Gram–Schmidt method:

```
[Q,R] = qr(A,0)
Q = column chebfun
    0.7071    1.2247    1.5811    1.8708    2.1213
    0.7071    0.8660    0.3953   -0.3307   -0.8618
    0.7071         0   -0.7906   -0.0000    0.7955
    0.7071   -0.8660    0.3953    0.3307   -0.8618
    0.7071   -1.2247    1.5811   -1.8708    2.1213
R =
    1.4142         0    0.4714         0    0.2828
         0    0.8165         0    0.4899   -0.0000
         0         0    0.4216   -0.0000    0.3614
         0         0         0    0.2138   -0.0000
         0         0         0         0    0.1077
```

Similarly, the (reduced) SVD of A should be a factorization $A = USV^T$, where U has the same type as A, S is $n \times n$ and diagonal with nonincreasing nonnegative diagonal entries, and V is $n \times n$ and orthogonal. We compute this by computing first the reduced QR decomposition $A = QR$ as above and then the ordinary SVD $R = U_1 SV^T$; we then have $A = USV^T$ with $U = QU_1$:

```
[U,S,V] = svd(A,0)
U = column chebfun
    0.9633   -1.4388    1.7244   -1.7117   -1.8969
    0.7440   -0.8205    0.1705    0.4315    0.9035
    0.5960    0.0000   -0.7747    0.0000   -0.8958
    0.7440    0.8205    0.1705   -0.4315    0.9035
    0.9633    1.4388    1.7244    1.7117   -1.8969
S =
    1.5321         0         0         0         0
         0    0.9588         0         0         0
         0         0    0.5181         0         0
         0         0         0    0.1821         0
         0         0         0         0    0.0809
V =
    0.9130    0.0000   -0.4014   -0.0000   -0.0725
   -0.0000   -0.8456   -0.0000    0.5339    0.0000
    0.3446   -0.0000    0.6640   -0.0000    0.6636
   -0.0000   -0.5339   -0.0000   -0.8456   -0.0000
    0.2182   -0.0000    0.6308    0.0000   -0.7446
```

The function `svd` by itself gives just the singular values, i.e., the diagonal entries of S, which can be interpreted as the semiaxis lengths of the hyperellipsoid that is the image of the unit ball in n-space under A. The function `cond` gives the ratio of the maximum and minimum singular values, and `rank` counts the number of singular values above a prescribed tolerance, which defaults to a number on the order of machine precision. Thus after the above calculations we have

```
>> cond(A)
ans = 18.9286

>> cond(Q)
ans = 1.0000

>> rank(A)
ans = 5

>> rank([1 x.^2 x.^2])
ans = 2

>> rank([sin(x) sin(2*x) sin(3*x) sin(4*x)])
ans = 4

>> rank([sin(2*x) sin(x).*cos(x)])
ans = 1
```

A moment ago we computed the QR factorization of the quasi matrix with columns 1, x, x^2, x^3, x^4. The resulting Q is a quasi matrix whose columns are orthogonal polynomials, each normalized to have 2-norm equal to 1. Thus the columns of Q are the *Legendre polynomials*, except that the latter are conventionally normalized to take the value 1 at $x = 1$. We can achieve that normalization like this:

```
>> for i = 1:5;
     P(:,i) = Q(:,i)/Q(1,i);
   end
   plot(P)
```

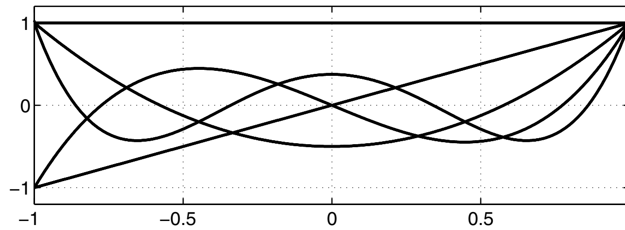

FIG. 9.1. *The first five Legendre polynomials, computed by QR decomposition of the quasi matrix whose columns are chebfuns for* 1, x, x^2, x^3, x^4.

which produces the familiar picture of Figure 9.1. For example, here are the coefficients of $P_4(x)$:

```
>> poly(P(:,5))
ans =
    4.3750   -0.0000   -3.7500    0.0000    0.3750
```

From the QR factorization comes a ready ability to solve least-squares problems, and this is implemented in an overloaded \ operator:

```
>> c = A\cos(x)
c =
    1.0000
   -0.0000
   -0.4994
    0.0000
    0.0398

>> norm(A*c-cos(x),inf)
ans = 9.2561e-05
```

Note that this is comparable to the result found earlier by interpolation in Chebyshev points.

Stewart has a half-serious remark in a footnote on p. 34 of his *Afternotes Goes to Grad School* [25]:

> If you introduce a formal mathematical object, people are likely to start writing papers about it. Like "On the generalized inverse of quasi-matrices." Ugh!

Well, Pete, here we are, sooner than you could have imagined:

```
>> pinv(A)
ans = row chebfun
    0.9375   -0.4980    1.7578   -0.4980    0.9375
   -3.7500    1.9887   -0.0000   -1.9887    3.7500
  -13.1250    7.7930   -8.2031    7.7930  -13.1250
    8.7500   -1.5468    0.0000    1.5468   -8.7500
   19.6875   -7.9980    7.3828   -7.9980   19.6875
```

The implementation is `[U,S,V] = svd(A,0)`, `pinv = V*inv(S)*U'`.

10. Conclusion. The chebfun system is under development. One of our goals is to find ways to make it more effective at solving problems by spectral collocation

methods such as those embodied in the MATLAB programs of [27]. Another, already partly achieved, is to extend the system to the case of "matrices" that are continuous in both the column and row dimensions rather than just the latter. Such objects can be regarded as bivariate functions on $[-1,1]\times[-1,1]$ or as integral operators. We have already implemented a good deal of this functionality and hinted at the possibility of continuity in the column dimension in the discussions of `f'*g` and `pinv`, but we will not give details here as they have not yet settled down.

Computation in the chebfun system has a hybrid flavor: the feel is symbolic, but the implementation is numerical. Having become used to this style of programming, we find it powerful and appealing. Of course, true symbolic computation would generally be better for those problems where it is applicable, but these are a minority.[6] The class of problems whose solutions are merely smooth, as we have targeted here, is much wider.

Many topics of numerical analysis can be illustrated in the chebfun system, as well as some fundamental ideas of approximation theory, functional analysis, and object-oriented programming; one can imagine many classrooms to which this system might be able to contribute. MATLAB itself was originally introduced as an educational tool, but eventually, it proved to be much more than that. Perhaps the chebfun system too will mature in interesting ways.

Note added in proof. Since this paper was accepted for publication we have become aware of a paper by Boyd that presents some of the same ideas [2]. Boyd is not concerned with general operations on functions, nor with overloading MATLAB commands, but with the specific subject of finding zeros of a function on an interval. The algorithm he proposes is the same as ours in two crucial respects: he approximates the function by a Chebyshev series of adjustable degree, and he finds the roots of this approximation by a change of variables that transplants the unit interval to the unit circle. Boyd then introduces a third powerful idea: recursive subdivision of the rootfinding interval to reduce the cubic operation count that is otherwise entailed in rootfinding. Following his lead, we have subsequently introduced recursion in the chebfun rootfinding operations too and thereby reduced the computing time for such operations, for large degrees, by an order of magnitude.

Acknowledgments. We are grateful to Jean-Paul Berrut, Folkmar Bornemann, Kevin Burrage, Carl de Boor, David Handscomb, Des Higham, Nick Higham, Cleve Moler, Pete Stewart, Peter Vertesi, and Andy Wathen for commenting on drafts of this article, suggesting applications, and pointing us to relevant literature.

REFERENCES

[1] J.-P. Berrut and L. N. Trefethen, *Barycentric Lagrange interpolation*, SIAM Rev., 46 (3) (2004), to appear.

[2] J. A. Boyd, *Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding*, SIAM J. Numer. Anal., 40 (2002), pp. 1666–1682.

[3] L. Brutman, *Lebesgue functions for polynomial interpolation—a survey*, Ann. Numer. Math., 4 (1997), pp. 111–128.

[4] C. W. Clenshaw and A. R. Curtis, *A method for numerical integration on an automatic computer*, Numer. Math., 2 (1960), pp. 197–205.

[5] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration,* 2nd ed., Academic Press, New York, 1984.

[6] See the appendix "The definition of numerical analysis" in [28].

[6] C. de Boor, *An alternative approach to (the teaching of) rank, basis, and dimension*, Linear Algebra Appl., 146 (1991), pp. 221–229.

[7] D. Gaier, *Lectures on Complex Approximation*, Birkhäuser, Boston, 1987.

[8] S. Goedecker, *Remark on algorithms to find roots of polynomials*, SIAM J. Sci. Comput., 15 (1994), pp. 1059–1063.

[9] D. Gottlieb, M. Y. Hussaini, and S. A. Orszag, *Theory and applications of spectral methods*, in Spectral Methods for Partial Differential Equations, R. G. Voigt, D. Gottlieb, and M. Y. Hussaini, eds., SIAM, Philadelphia, 1984, pp. 1–54.

[10] J. A. Grant and A. Ghiatis, *Determination of the zeros of a linear combination of Chebyshev polynomials*, IMA J. Numer. Anal., 3 (1983), pp. 193–206.

[11] N. J. Higham, *The numerical stability of barycentric Lagrange interpolation*, IMA J. Numer. Anal., to appear.

[12] T. W. Körner, *Fourier Analysis*, Cambridge University Press, Cambridge, UK, 1988.

[13] A. R. Krommer and C. W. Ueberhuber, *Computational Integration*, SIAM, Philadelphia, 1998.

[14] G. G. Lorentz, *Approximation of Functions*, 2nd ed., Chelsea, New York, 1986.

[15] J. C. Mason and D. C. Handscomb, *Chebyshev Polynomials*, Chapman & Hall/CRC, Boca Raton, FL, 2003.

[16] G. Mastroianni and J. Szabados, *Jackson order of approximation by Lagrange interpolation.* II, Acta Math. Hungar., 69 (1995), pp. 73–82.

[17] J. M. McNamee, *A bibliography on roots of polynomials*, J. Comput. Appl. Math., 47 (1993), pp. 391–392; also available online from www.elsevier.com/homepage/sac/cam/mcnamee/.

[18] G. Meinardus, *Approximation of Functions: Theory and Numerical Methods*, Springer-Verlag, Berlin, 1967.

[19] C. B. Moler, *ROOTS—Of polynomials, that is*, MathWorks Newsletter, 5 (1991), pp. 8–9.

[20] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[21] T. Ransford, *Potential Theory in the Complex Plane*, Cambridge University Press, Cambridge, UK, 1995.

[22] T. J. Rivlin, *Chebyshev Polynomials: From Approximation Theory to Algebra and Number Theory*, Wiley, New York, 1990.

[23] H. E. Salzer, *Lagrangian interpolation at the Chebyshev points* $x_{n,\nu} = \cos(\nu\pi/n)$*,* $\nu = 0(1)n$*; some unnoted advantages*, Computer J., 15 (1972), pp. 156–159.

[24] G. A. Sitton, C. S. Burrus, J. W. Fox, and S. Treitel, *Factorization of very high degree polynomials in signal processing,* IEEE Signal Process. Mag., 20 (2003), pp. 27–42.

[25] G. W. Stewart, *Afternotes Goes to Graduate School: Lectures on Advanced Numerical Analysis*, SIAM, Philadelphia, 1998.

[26] K.-C. Toh and L. N. Trefethen, *Pseudozeros of polynomials and pseudospectra of companion matrices*, Numer. Math., 68 (1994), pp. 403–425.

[27] L. N. Trefethen, *Spectral Methods in MATLAB*, SIAM, Philadelphia, 2000.

[28] L. N. Trefethen and David Bau III, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

[29] L. N. Trefethen and J. A. C. Weideman, *Two results on polynomial interpolation in equally spaced points*, J. Approx. Theory, 65 (1991), pp. 247–260.

[30] J. L. Walsh, *Interpolation and Approximation by Rational Functions in the Complex Domain*, 5th ed., AMS, Providence, RI, 1969.

[31] J. H. Wilkinson, *The perfidious polynomial*, in Studies in Numerical Analysis, G. H. Golub, ed., Math. Assoc. Amer., Washington, D.C., 1984, pp. 1–28.