PIECEWISE SMOOTH CHEBFUNS

RICARDO PACHÓN*, RODRIGO B. PLATTE†, AND LLOYD N. TREFETHEN‡

Abstract. Algorithms are described that make it possible to manipulate piecewise-smooth functions on real intervals numerically with close to machine precision. Breakpoints are introduced in some such calculations at points determined by numerical rootfinding, and in others by recursive subdivision or automatic edge detection. Functions are represented on each smooth subinterval by Chebyshev series or interpolants. The algorithms are implemented in object-oriented MATLAB in an extension of the chebfun system, which was previously limited to smooth functions on $[-1, 1]$.

Key words. edge detection, chebfun system, Chebyshev series, barycentric interpolation

AMS subject classifications. 33F05, 65D05

1. Chebyshev calculations on $[-1, 1]$. The mathematics of approximation of smooth functions on $[-1, 1]$ by Chebyshev series and interpolants goes back about a century, and landmarks in the algorithmic side of this subject include Salzer's barycentric interpolation formula [17] and the determination of Chebyshev coefficients via FFT [10]. Recently a software system in object-oriented MATLAB was developed by Zachary Battles and the third author, the *chebfun* system, that aims to exploit these tools to compute with functions in a manner combining the speed of floating-point numerics with the "feel" of symbolic computing [1, 2]. A chebfun is a MATLAB representation of a function of a continuous variable following MATLAB syntax for vectors, with MATLAB's vector operations overloaded by appropriate analogues. The vision of the project is that if f is one chebfun defined on the interval $[-1, 1]$, for example, and g is another, then the MATLAB command `h = f.*g` should compute a new chebfun h with the property that for each $x \in [-1, 1]$, $h(x)$ is equal to $f(x)g(x)$ up to a relative error no greater than about machine precision (relative to the maximum of $|fg|$ on $[-1, 1]$, not its value at the point x) [18]. To achieve this, f and g are represented by polynomial interpolants through data at sufficiently many Chebyshev points, with this number determined automatically, or equivalently, by polynomials in the form of finite Chebyshev series. The number of data points can be anything from 1 for a constant function to tens or hundreds of thousands; function evaluation is rapidly and stably carried out by barycentric interpolation [3, 14, 17].

For example, the commands

```
>> x = chebfun(@(x) x);
>> f = sin(10*x);
>> g = 1./sqrt(2-x);
>> [length(x) length(f) length(g)]
ans =  2    36    27
```

construct chebfuns `x`, `f`, and `g` corresponding to x and two other simple functions on $[-1, 1]$, and the "lengths" reported incidate that 2, 36 and 27 points, respectively, are

*Oxford Computing Laboratory, Wolfson Bldg., Parks Rd., Oxford OX1 3QD, UK (ricp@comlab.ox.ac.uk, http://www.comlab.ox.ac.uk/ricardo.pachon/).

†Oxford Computing Laboratory, Wolfson Bldg., Parks Rd., Oxford OX1 3QD, UK (rodp@comlab.ox.ac.uk, http://www.comlab.ox.ac.uk/rodrigo.platte/).

‡Oxford Computing Laboratory, Wolfson Bldg., Parks Rd., Oxford OX1 3QD, UK (LNT@comlab.ox.ac.uk, http://www.comlab.ox.ac.uk/nick.trefethen/).

needed to represent these functions to machine precision. That is, `f` and `g` amount to polynomials of degrees 35 and 26. Algebraically speaking, the product fg is of degree 61, but after truncating to machine precision the system finds only degree 34:

```
>> h = f.*g;
>> length(h)
ans = 35
```

The sequence

```
>> x = rand
x = 0.814723686393179
>> h(x)
ans = 0.879309706420458
>> sin(10*x)/sqrt(2-x)
ans = 0.879309706420459
```

confirms (at least for one value of x) that the accuracy is nevertheless close to the level of machine precision. Similarly f/g is not a polynomial at all, mathematically speaking, but in the chebfun system it is a polynomial of degree 35:

```
>> length(f./g)
ans = 36
```

Other calculations with similar precision can also be carried out with these functions. For example, the following sequence involving overloads of MATLAB's `sum` and `roots` commands determines the integral of fg from -1 to 1 and the roots of $f+g$ in $[-1,1]$, yielding results in each case correct except sometimes in the final digit.

```
>> sum(h)
ans = 0.031767660431063
>> roots(f+g)
ans =
  -0.879457197419040
  -0.693833354191292
  -0.241007073210693
  -0.076692881584451
   0.405558247388803
   0.531272924965241
```

All this works excellently so long as the functions of interest are smooth. In applications, however, many functions that arise are not smooth, though they are often piecewise smooth. In this article we propose numerical algorithms for calculating with piecewise smooth functions, which have been implemented in an extension of the chebfun system. For example, the command

```
>> h = max(f,g);
>> plot(h)
```

produces the result shown in Figure 1.1. In the original chebfun system, such a computation was not possible, because $\max(f,g)$ cannot be represented to machine precision by a single polynomial unless it is of degree on the order of 10^{15}.

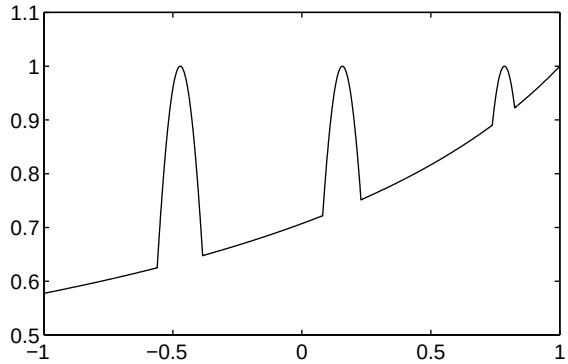

FIG. 1.1. *A piecewise smooth chebfun constructed by the* `max` *operator from breakpoints determined by zerofinding.*

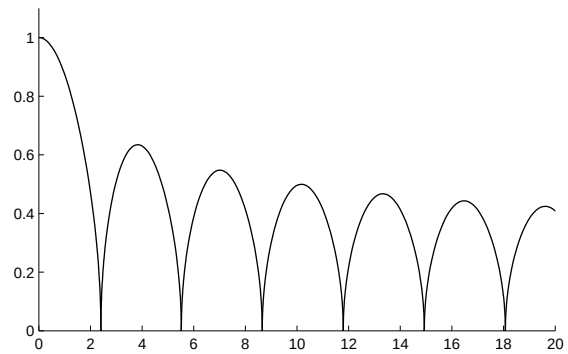

FIG. 1.2. *An example in which the breakpoints have been located by automatic edge detection.*

Figure 1.2 shows another example, in which six points of discontinuity of a function defined on the interval $[0, 20]$ have been located adaptively by execution of the command `f = chebfun(@(x) sqrt(abs(besselj(0,x))),[0 20])`.

Capabilities for dealing with piecewise functions defined symbolically have existed for a number of years in the symbolic computing systems Maple and Mathematica, which in fact both have commands with the name "piecewise" [15, 20]. As always, the symbolic approach brings great power in certain cases but a great cost in others, when the quantities of interest cannot be determined symbolically or when expressions grow combinatorially in length [18]. Concerning numerical computation with piecewise functions, we do not know of any other projects closely related to the present one. A more distant relative is the fascinating work of Curtis and Powell from forty years ago, in which the aim was to determine spline fits to given functions automatically to a specified accuracy [8, 16]. (Curtis and Powell insist on C^2 continuity but point out that this condition could be relaxed.) It is marvelous to see in this work how fundamental mathematical ideas have lasted through the generations while the computational environment has changed beyond recognition. The numerical example discussed at some length in [16], for example, is the function $f(x) = 1/(0.01+(x-0.3)^2)$, which the chebfun system approximates to 15-digit accuracy in less than 0.1 secs. on a desktop computer.

2. Piecewise smooth chebfuns. "Classic" smooth chebfuns still exist—they have been renamed `funs` and are now a separate MATLAB class used to represent each of the pieces of a chebfun. A chebfun is now an object with six fields:

`funs`, `nfuns`, `scl`, `trans`, `ends`, `imps`.

The field `funs` is a vector containing n funs for some positive integer n, which is stored as `nfuns`. The field `scl` is a scalar equal to the largest of the absolute values of all the data defining a chebfun. The field `trans` is a flag set to 0 for a column chebfun, 1 for a row chebfun; this makes it possible to compute an inner product, for example, with the syntax `f'*g`. The field `ends` is a vector of $n+1$ floating point numbers in monotonically increasing order indicating subintervals:

$$a = \texttt{ends}(1) < \texttt{ends}(2) < \cdots < \texttt{ends}(n+1) = b.$$

The system scales $\texttt{funs}(i)$ to the interval $[\texttt{ends}(i), \texttt{ends}(i+1)]$. The field `imps` contains the function values at the breakpoints themselves, which if the function is discontinuous may match the fun on the left, the fun on the right, or neither. Thus in mathematical terms we may think of a chebfun at a discontinuity as (to floating-point approximation) lower-semicontinuous, upper-semicontinuous, or neither. In general `imps` may be a matrix rather than just a vector, including data associated with Dirac delta functions at the breakpoints, which are introduced for example if one differentiates a step function. In this article we do not mention delta functions further, for their proper treatment is a substantial topic in its own right and still under investigation.

The chebfun constructor can make a chebfun in various ways, and we give a partial list here. The simplest is based on an explicit list of functions and endpoints, with the functions specified by any combination of constants, strings, anonymous functions, or other chebfuns. For example, the following sequence constructs a chebfun with a single piece corresponding to the Bessel function $J_0(x)$ on the interval $[0, 1000]$, computes its zeros, and reports their number. This executes in a few hundredths of a second on a laptop, and additional checks show that the chebfun differs from the exact Bessel function on $[0, 1000]$ by no more than 1.5×10^{-14}.

```
>> f = chebfun(@(x) besselj(0,x),[0,1000]);
>> length(f)
ans = 581
>> length(roots(f))
ans = 318
```

For another example, consider the command

```
>> f = chebfun('x.*cos(8*pi*x)',1,'4-1.5*x','abs(0.15./(t-4+.1i))',[0:3 5])
```

This generates a chebfun defined on the interval $[0, 5]$ with four explicitly defined pieces on the intervals $[0, 1]$, $[1, 2]$, $[2, 3]$ and $[3, 5]$, and `plot(f)` produces the image shown in Figure 2.1. The default behavior is that at each breakpoint, the chebfun takes the value of the piece on the right. The command `sum(f)` reveals that the integral of f from 0 to 5 is 2.149466885089391, and `length(f)` shows that the total number of data points on all four intervals is 195. To find the integral of f from 1 to 3, we can construct the indefinite integral $\int_0^x f(s)\,ds$ with `g = cumsum(f)` and then evaluate `g(3)-g(1)`. The result is reported as 1.250000000000000.

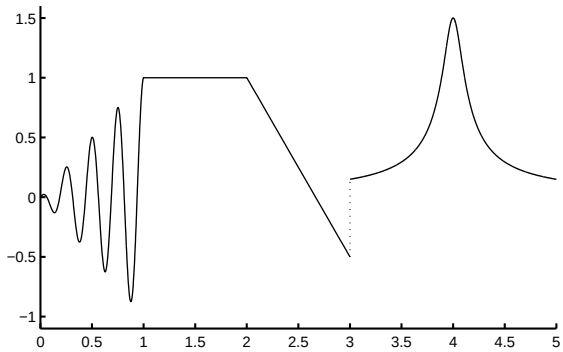

FIG. 2.1. *A chebfun with four pieces.*

This example introduces points of discontinuity explicitly. More interesting are cases where chebfuns are constructed from other chebfuns in such a way that new breakpoints appear implicitly. This may happen when one of the commands

```
abs, sign, floor, ceil, round, fix, mod, rem
```

is applied to a chebfun or one of the commands

```
min, max, conv
```

is applied to a pair of chebfuns. In the case of `abs(f)` or `sign(f)`, for example, the zeros of f on its interval of definition are first determined by the `roots` command, which uses the method described in [2], in which the chebfun is recursively reduced to problems of length not greater than about 100 that are solved by finding eigenvalues of suitable colleague (or "Chebyshev companion") matrices [2, 5, 6, 12]. At each such zero, a new breakpoint in the chebfun is introduced, in addition to any breakpoints that may already be there. Similarly for `min(f,g)` or `max(f,g)`, where `f` and `g` are chebfuns defined on the same interval, the system first finds the roots of $f - g$ and introduces new breakpoints accordingly. This is how the chebfun of Figure 1.1 was constructed.

Once a piecewise smooth chebfun has been constructed, about 100 overloaded MATLAB commands can be applied to it. In each case these commands are executed by performing the appropriate operations on the component funs. For example, if a function like `sin`, `exp`, or `tanh` is applied to a chebfun `f`, the system applies the indicated operation to each component fun, adjusts the length adaptively to achieve the usual precision, and concatenates the resulting funs into the desired chebfun output. The command `sum` returns an integral computed by Clenshaw–Curtis quadrature on each piece, and `cumsum` returns an indefinite integral chebfun constructed from the indefinite integral funs of each piece. The command `roots` returns a vector of roots of each fun as well as roots at breakpoints, e.g. if a real function discontinuously passes from negative to positive. The command `norm(f)` depends only on integration, since it returns a 2-norm, whereas `norm(f,inf)` and `norm(f,1)` utilize rootfinding to find local extrema and zeros of f, respectively. The command `diff` performs differentiation piecewise, with delta functions introduced at points of discontinuity as mentioned earlier. The command `length(f)` reports the total number of data points defining the chebfun f, and one can extract its ends and funs with `f.ends` and `f.funs`.

3. Automatic subdivision. In addition to these processes, the chebfun constructor itself may automatically introduce breakpoints if it is presented with a difficult function. It does this by an algorithm combining two features:

- *Edge detection* for functions with discontinuous values or derivatives,
- *Recursive subdivision* for functions without clear discontinuities but nonetheless having complicated behavior.

The user has an option of enabling or disabling these processes via the commands

```
splitting on, splitting off
```

The former choice is necessary when dealing with functions that are far from smooth. The latter, which was used in the examples given so far, except that of Figure 2.1, is useful for exploratory work, for example if one wants to examine the properties of Chebyshev interpolants of high orders, and it also has advantages for applications in differential equations, where the introduction of breakpoints may lead to difficulties.

Before presenting the algorithm, we give illustrative examples. First are three examples of the edge detection kind.[1] The command `chebfun(@(x)abs(x-0.1))` produces a chebfun consisting of two funs, each linear, with a breakpoint at $x = 0.1$ (up to an error of $2^{-56} \approx 1.4 \times 10^{-17}$). Since the construction process only samples the function at various points x (in contrast to the command `abs(chebfun(@(x)x)-0.1)`, which would take advantage of the rootfinder), the result of this computation illustrates the edge detector successfully at work. Similarly the command `chebfun(@(t) sign(sin(t)), [0 10*pi])`, while again only sampling the function at various points in $[0, 10\pi]$, succeeds in producing a chebfun with 10 pieces, each constant. The breakpoints are now at $\pi, 2\pi, \ldots, 9\pi$, or more precisely, at their correctly rounded approximations in the floating point number system. A third example is this:

```
>> nodes = 0:8;
>> vals = sin(nodes);
>> f = chebfun(@(x) spline(nodes,vals,x),[0 8]);
>> f.ends
ans =       0  1.999948  2.999928  3.999992  5.000029  5.999874  8.000000
```

In this sequence, the chebfun constructor is asked to produce a chebfun by sampling the function `spline(nodes,vals,x)`. Now MATLAB's `spline` command produces a cubic spline interpolant through the given nodes and values, so the function being sampled, while appearing smooth to the eye, is actually just C^2, with jumps in the third derivative at the integers. The chebfun constructor has duly located these discontinuities. It is interesting to note that the located breakpoints match the true ones only to an accuracy on the order of the cube root of machine precision, a result that is reasonable since this is all that is needed to capture the function to machine precision. We were puzzled when we first saw this output—why were there no discontinuities at 1 and 7? A review of the documentation for `spline` informed us that by default it imposes "not-a-knot" boundary conditions, in which the first and last interior points in the list of nodes are not actually taken as discontinuities.

[1] The subject of edge detection is a big one, with connections to many mathematical and engineering problems, and we shall make no attempt to review the literature. One reference we have found useful is [11]. Our situation is unlike the usual edge detection problem in engineering, as we are aiming for machine precision and are able to sample the function as necessary to achieve this.

Next we give two examples of recursive subdivision in the absence of clear point discontinuities. First, consider the result of `chebfun(@(x)sin(x),[0 1e4])`. With `splitting off`, the output is a single global chebfun of length 5165. With `splitting on`, the interval is broken into 128 equal pieces, on each of which the function is represented by between 71 and 89 points, for a total of 9834 data values. Second, consider the function

$$f(x) = \sqrt{x}, \quad x \in [0, 1].$$

Though mathematically f has "just one piece," being analytic throughout $(0, 1]$, it has a singularity at $x = 0$ and cannot be represented to machine precision by a polynomial of reasonable order. In this case the constructor returns a chebfun consisting of seven funs on exponentially graded subintervals, as revealed by the following sequence:

```
>> f = chebfun(@(x) sqrt(x),[0 1]);
>> f.ends'
ans =
                    0
   0.000000000100000
   0.000000010000000
   0.000001000000000
   0.000100000000000
   0.010000000000000
   0.505000000000000
   1.000000000000000
>> length(f)
ans = 566
```

Each subinterval is 100 times smaller than the last, and the lengths of the corresponding funs are between 18 and 126. With 566 data points all together, one could regard this as an absurdly complicated representation of $\sqrt{x}$, but on the other hand the process is a general one that will be equally effective for much more complicated functions. For example if we change `sqrt(x)` to `sqrt(x).*cos(100*x)`, the length of the chebfun actually shrinks to 546; even with `sqrt(x).*cos(1000*x)` it increases only to 1285. And though the underlying representation is complicated, the command `sum(f)` applied to this chebfun for $\sqrt{x}$ gives an answer that differs from the correctly rounded value of $2/3$ by just $2^{-53} \approx 1.1 \times 10^{-16}$.

We now describe the algorithm that achieves these effects. For this we introduce the term `naf`, which stands for "not-a-`fun`". In this algorithm, a `fun` is a subinterval with a successful representation of the given function by a polynomial of degree <128. By a `naf` we mean a subinterval on which this polynomial degree has been found insufficient for the required accuracy. During the construction process the chebfun F consists of a sequence of `fun`s and `naf`s, and the process is finished when these are all `fun`s.

ALGORITHM TO CONSTRUCT A CHEBFUN F FROM A FUNCTION f

Try to construct F as a single `fun`*. The result may be a* `fun` *or a* `naf`*.*
while *F has some* `naf`*s in it*
  for $i = 1$ **to** *the number of* `naf`*s*
    Let $[a, b]$ be the interval associated with the ith `naf`*.*

Call **detectedge**(a, b, f) *(see below) to find an edge* $c \in [a, b]$*, if any.*
if *an edge* c *is found at distance* $\geq 10^{-14}(b-a)$ *from endpoints* a *and* b
Mark c *as a genuine edge.*
elseif *an edge* c *is found at distance* $< 10^{-14}(b-a)$ *from endpoint* a *or* b
Set c *equal to the number at distance* $0.01(b-a)$ *from that endpoint and mark* c *as a removable edge.*
else *(i.e. if no edge is found)*
Set $c = (a+b)/2$ *and mark it as a removable edge.*
end if
Split this `naf` *at* c*, and try to construct a* `fun` *on each side.*
The result may be `fun+fun`*,* `naf+fun`*,* `fun+naf`*, or* `naf+naf`*.*
end for
end while
for $i = 1$ **to** *the number of removable edges introduced above*
Merge the funs adjacent to edge(i) into a single fun if possible.
end for

It remains to describe **detectedge**. The purpose of this procedure is to speed up the calculation by looking quickly for discontinuities in f, f', f'', or f''' in the interval of interest, using a simple finite difference scheme rather than the more general Chebyshev interpolation. If **detectedge** finds no discontinuity, little harm is done, since the algorithm then falls back upon the general procedure without having wasted too much effort. If it does find a discontinuity, the improvement in function evaluations per edge may be as great as (very roughly)

$$\text{from} \quad 129 \log_2(2^{-52}) \approx 7000 \quad \text{to} \quad 15 \log_7(2^{-52}) \approx 300.$$

The first number corresponds to sampling on 129-point Chebyshev grids at each of 52 levels of binary recursion down to machine precision, and the second to sampling on 15-point equispaced grids at each of 19 levels of recursion.

Specifically, **detectedge** works by calculating estimates of $|f'|, |f''|, |f'''|, |f''''|$ from finite differences of sampled values of f of orders 1, 2, 3, 4 on 15-point equally-spaced grids. If any of these estimates grows as the grid is refined, then a singularity appears to be present, and the grid is repeatedly shortened by a factor of 7, all the way down to machine precision. If it is the estimate of $|f'|$ that blows up, this case is treated specially by a program **findjump** which uses bisection to locate the discontinuity, normally down to the very last bit in floating-point arithmetic.

function edge = **detectedge***(f,a,c)* *% Find singularity of* f *in* $[a, c]$
edge = `NaN`
On a 50*-point equispaced grid in* $[a, c]$*, compute estimates of* $|f'|, |f''|, |f'''|, |f''''|$*.*
Set b *= the gridpoint associated with the maximum estimate of* $|f''''|$*.*
Set $d_{\max} = 4$*.*
while *the current interval* $[a, c]$ *is larger than machine precision*
Set a *and* c *to the gridpoints left and right of* b*, respectively.*
Refine 7*-fold to a* 15*-point grid in* $[a, c]$*, and find the gridpoints associated there with the maximum estimates of* $|f'|, \ldots, |f^{(d_{\max})}|$*.*
if *refinement has increased none of the estimates by a factor of* 1.5 *or more*
return *(i.e., no edge has been detected)*
end if

Set $d_{\max}$ = order of lowest derivative that has increased by such a factor.
Set b = gridpoint associated with maximum value of $|f^{(d_{\max})}|$.
if $d_{\max} = 1$
b = **findjump***(f,a,c)*
return
end if
end while
edge = b

The above descriptions are slightly simplified accounts of what goes on in our actual program. In particular, our treatment of vertical and horizontal scales is more complicated than is suggested above by the term "machine precision." In fact, all convergence criteria are relative, making chebfun construction scale-invariant in the usual manner associated with IEEE floating-point arithmetic. Thus for example if `f` is the chebfun constructed from $f(x)$ on $[0,1]$ and `f2` is the chebfun constructed from $sf(tx)$ on $[0,t^{-1}]$, then so long as underflow and overflow limits are not reached, `f2(x)` will match s`f(`t`x)` closely, and the two will be identical if s and t are powers of 2.

Here is one more example of automatic edge detection in action. The following code sequence constructs chebfuns for $e^x + \cos(7x) + 0.1\mathrm{sign}(x - x_0)$, where x_0 takes three random values in $[0,1]$, and compares x_0 with the automatically determined breakpoint.

```
for j = 1:3
  x0 = rand;
  f = chebfun(@(x) exp(x)+cos(7*x)+0.1*sign(x-x0));
  fends = f.ends;
  disp([x0 fends(2) x0-fends(2)])
end
```

The results, delivered in 0.1 secs. on a workstation, are as follows:

```
   0.594896074008614   0.594896074008614              0
   0.262211747780845   0.262211747780845              0
   0.602843089382083   0.602843089382083              0
```

The final column shows that the differences are exactly zero, confirming that the edge detector locates jump discontinuities down to the last bit.

4. Applications. The algorithms and software we have presented may be useful in education, for example in courses on calculus, statistics, numerical analysis or approximation theory, and also in practical computing. We shall not attempt to distinguish between educational and practical applications, but just present eight examples that may be of some interest.

Example 1. *Riemann approximation to an integral.* In calculus we learn to approximate a continuous function by a Riemann sum. In the chebfun system, the Riemann sum can be realized as a function. For example, here is a sequence that plots a smooth function f on $[0,1]$ and a piecewise-constant approximation f_h. The corresponding plot appears in Figure 4.1. Here and in our other examples, most details associated with labels, line widths, and other formatting matters are omitted from the MATLAB listing.

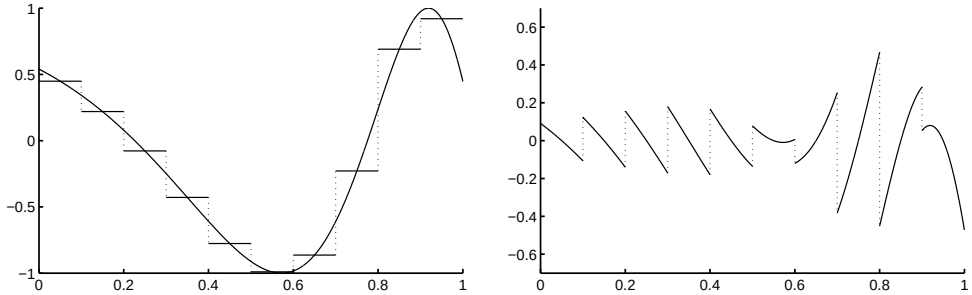

FIG. 4.1. *On the left, a smooth function and its Riemann approximation realized as a chebfun. On the right, the chebfun obtained by taking the difference between the two.*

```
f = chebfun(@(x) cos(exp(2*x)),[0 1]);
h = 0.1; ends = 0:h:1; midpts = h/2:h:1;
fh = chebfun(num2cell(f(midpts)),ends);
plot(f), hold on, plot(fh)
```

Once f and f_h have been constructed, we can compute with them. The right side of Figure 4.1 plots their difference, and the accuracy of the approximation can be explored numerically with commands like these:

```
>> sum(f)
ans = -0.113851287074054
>> sum(fh)
ans = -0.108779592055534
>> norm(f-fh,inf)
ans = 0.471646386553596
```

One could extend the experiment in a number of ways, for example, to investigate convergence as $h \to 0$.

Example 2. *Convolution, B-splines, and the Central Limit Theorem.* MATLAB's `conv` command has been overloaded for chebfuns, performing convolution of continuous or piecewise continuous functions. By repeatedly convolving a step function with itself, we can use this command to illustrate the mathematics of B-splines. For our purposes the nth B-spline B_n is a piecewise polynomial of degree n with support $[-n-1, n+1]$, breakpoints at $-n+1, -n+3, \ldots, n-1$, and $n-1$ continuous derivatives [7]. The following command sequence constructs such functions by convolution, using `B.ends` to track the breakpoints. The resulting plot is shown in Figure 4.2.

```
step = chebfun(0.5); B = step;
for n = 1:4
  B = conv(B,step);
  subplot(2,2,n)
  plot(B), hold on, plot(B.ends,B(B.ends),'.')
end
```

As n increases, these curves converge to Gaussians. This follows from the Central Limit Theorem, for the nth B-spline is the probability distribution for the sum of $n+1$ random numbers uniformly distributed in $[-1, 1]$. The Central Limit Theorem

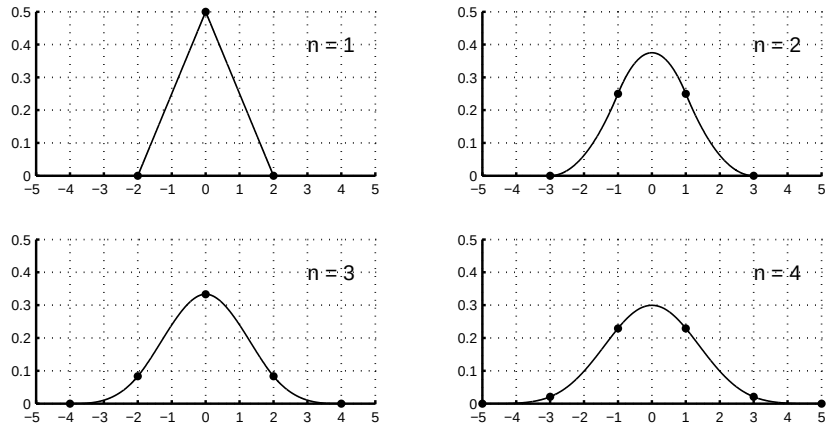

FIG. 4.2. *The first four B-splines, constructed by convolution.*

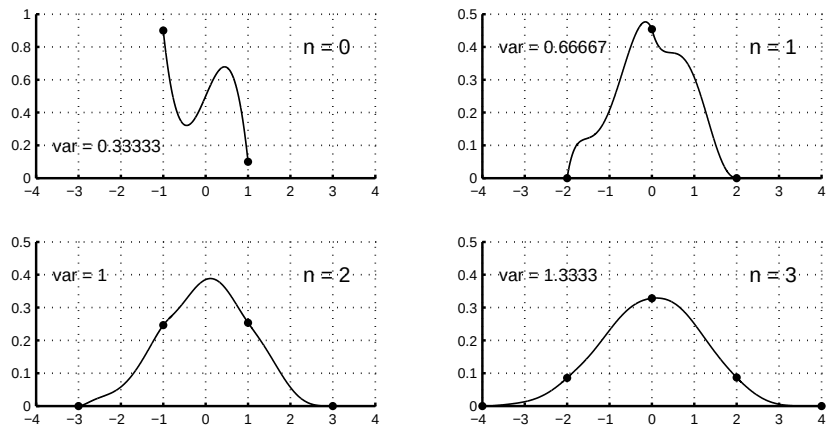

FIG. 4.3. *The same construction but starting from a different initial function. According to the Central Limit Theorem, the limit is still Gaussian.*

also implies that the limit will be Gaussian if we start with some other initial curve. Figure 4.3 shows the example in which `step` has been replaced by the initial function $0.5 + 0.6x - x^3$, these coefficients having been chosen so that the function again has integral 1 and first moment 0. The figure also displays the second moment or variance, computed from `sum(chebfun(@(x) B(x).*x.^2,[-n-1 n+1]))`.

The Gaussian of integral 1 and variance σ^2 is $(2\pi\sigma^2)^{-1/2}\exp(-(x/\sigma)^2/2)$. Setting $\sigma^2 = 4/3$ at the end of the computation above, we find the following deviation from a Gaussian:

```
>> x = chebfun(@(x) x,[-n-1 n+1]);
>> sigma = sqrt(4/3);
>> gaussian = exp(-(x/sigma).^2/2)/(sigma*sqrt(2*pi));
>> norm(B-gaussian,inf)
ans = 0.018338990457137
```

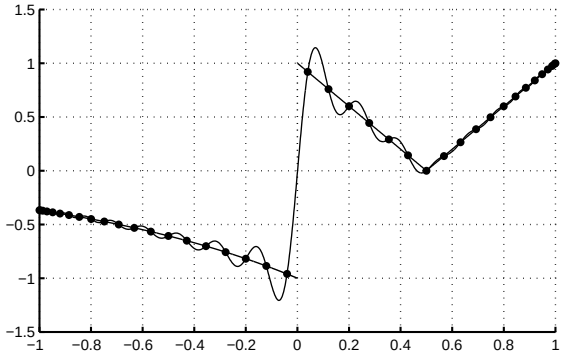

FIG. 4.4. *The Gibbs phenomenon for polynomial interpolation in Chebyshev points.*

One can readily modify these commands to investigate the (slow) convergence to Gaussian form as n increases further.

Example 3. *The Gibbs phenomenon.* The Gibbs phenomenon refers to the fact that Fourier or Chebyshev interpolants or truncated series oscillate near (and also not so near!) points of discontinuity. With piecewise smooth chebfuns, we can conveniently examine this effect not only for the usual step function but also for more general functions with discontinuities. Here is an example, with the resulting plot shown in Figure 4.4. The second line explicitly constructs an interpolant of degree 39 rather than attempting to determine the degree adaptively.

```
f = chebfun('-exp(x)','2*abs(x-0.5)',-1:1);
fn = chebfun(@(x) f(x),40);
plot(f), hold on, plot(fn,'.-')
```

How big is the overshoot? We can measure it by using `max`. The following code computes the overshoot and compares it to the limiting value as $n \to \infty$ reported in [13].

```
exact = 0.282283455775;
for n = 2.^(1:9)
  fn = chebfun(@(x) f(x),n);
  overshoot = max(abs(fn)-abs(f));
  err = overshoot - exact;
  fprintf('%5d %14.8f %12.6f\n', n, overshoot, err)
end

    2     0.65803014     0.375747
    4     0.05097072    -0.227424
    8     0.40693660     0.124653
   16     0.31523215     0.032949
   32     0.28221031    -0.000073
   64     0.28468454     0.002401
  128     0.28271534     0.000432
  256     0.28265007     0.000367
  512     0.28242243     0.000139
```

Example 4. *Analytic functions.* Piecewise smooth chebfuns offer a convenient

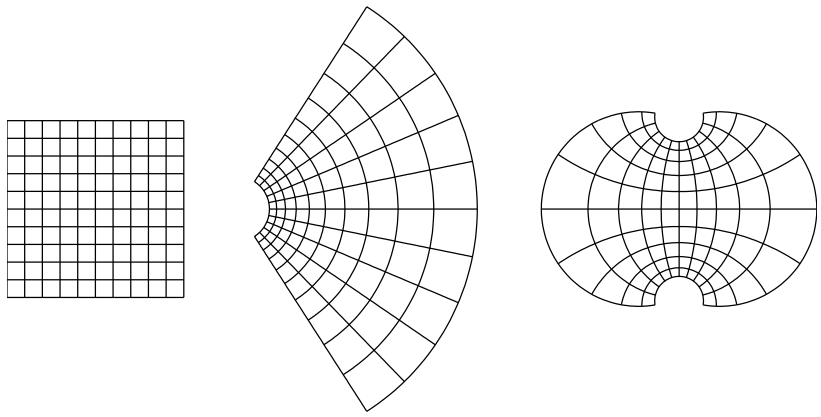

FIG. 4.5. *A grid corresponding to the unit square* $[-1,1] \times [-1,1]$ *in the complex plane represented as a piecewise linear chebfun* S*, and its images* $\exp(S)$ *and* $\tan(S)$.

means to visualize analytic functions in the complex plane. (Another approach, which we shall not discuss, would be to use quasimatrices, i.e., chebfuns containing multiple columns [19].) For example, the following sequence constructs a chebfun S with 22 pieces consisting of various horizontal and vertical lines in the unit square $[-1,1] \times [-1,1]$. It then plots S and its images $\exp(S)$ and $\tan(S)$ (Figure 4.5). The manner in which this chebfun is constructed, by successively appending new intervals to an initial interval $[-1,1]$, leads to it being defined on the interval $[-1,43]$, but for this application, we care only about its range, not its domain. The data plotted in these curves will be accurate to the usual 14 or 15 decimal places.

```
x = chebfun(@(x) x);
S = chebfun;                        % make an empty chebfun
for d = -1:.2:1
  S = [S; d+1i*x; 1i*d+x];          % add 2 more lines to the collection
end
plot(S), plot(exp(S)), plot(tan(S))
```

Example 5. *Complex contour integrals.* Complex analysis is full of contour integrals of analytic functions, as in the Cauchy integral formula. Sometimes the contours are circles, and by using the chain rule, we can reduce a circular contour to an interval by a change of variables. Other times, more complicated contours are useful, but almost always, these are piecewise smooth. These can be reduced to intervals too, and now the integrands are continuous and piecewise analytic.

For example, this code constructs a chebfun on $[0,4]$ corresponding to a contour Γ of a familiar "keyhole" form (Figure 4.6).

```
s = chebfun(@(s) s,[0 1]);
c = [-2+.05i -.2+.05i -.2-.05i -2-.05i];
z = [c(1)+s*(c(2)-c(1))
     c(2)*c(3).^s./c(2).^s
     c(3)+s*(c(4)-c(3))
     c(4)*c(1).^s./c(4).^s]
plot(z)
```

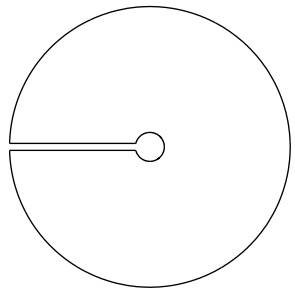

FIG. 4.6. *A keyhole contour for a complex integral.*

Integrals over such contours usually appear in theoretical arguments, not computer programs, but in the chebfun system we can work with them numerically. The chain rule gives

$$\int_\Gamma f(z)\,dz = \int_a^b f(z(s))\,z'(s)\,ds, \tag{4.1}$$

and the integral becomes `sum` in the chebfun system. Notice that $z'(s)$ will be a piecewise smooth function with jumps corresponding to corners in the contour, but this poses no difficulties for the integral. For example, the following code segment integrates $f(z) = \log(z)\tanh(z)$ around the contour:

```
>>  f = log(z).*tanh(z);
>>  I = sum(f.*diff(z))
I = 0.000000000000003 + 5.674755637702227i
```

The exact solution, which is $2\pi i$ times sum of the residues at the poles of f at $\pm\pi i/2$, is $4\pi i \log(\pi/2) \approx 5.674755637702224i$.

Example 6. *Green functions.* Green functions are solutions to linear differential equations with homogeneous boundary conditions and inhomogeneous forcing data consisting of a Dirac function. For a one-dimensional problem, they typically take the form of a continuous function that is smooth except for a jump in the derivative at a point. In the chebfun system, such a function can be realized numerically.

For example, the operator $Lu = u'' + u$ on $[0, \pi/2]$ has the Green function

$$g(x,y) = \begin{cases} -\sin(x)\cos(y) & \text{if } x \le y, \\ -\cos(x)\sin(y) & \text{if } x \ge y. \end{cases} \tag{4.2}$$

In MATLAB, the following anonymous function takes as input a value of y and produces as output the chebfun function of x corresponding to $g(x,y)$:

```
green = @(y) chebfun(@(x)-sin(x)*cos(y), @(x)-cos(x)*sin(y), [0 y pi/2]);
```

The left side of Figure 4.7 shows some of these functions as plotted by the sequence `for y = .2:.2:1, plot(green(y)), hold on, end`.

Now suppose we want to solve $Lu = f$ using these Green functions for some function f defined on $[0, \pi/2]$. The solution is given by the formula

$$u(y) = \int_0^{\pi/2} f(x)\,g(x,y)\,dx, \tag{4.3}$$

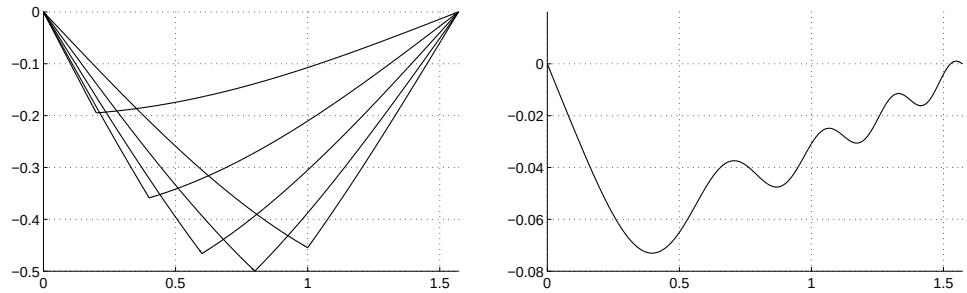

FIG. 4.7. *On the left, Green functions $g(x, y)$ for $y = 0, 0.2, \ldots, 1$ for the operator $Lu = u'' + u$ with homogeneous Dirichlet boundary conditions on $[0, \pi/2]$. On the right, the solution to a problem $Ly = f$ computed via these Green functions.*

which we can realize in the chebfun system. For example, the following sequence produces the plot shown on the right side of Figure 4.7.

```
f = chebfun(@(x) exp(x).*sin(10*x.^2),[0 pi/2]);
u = chebfun(@(y) sum(f.*green(y)),[0 pi/2]);
plot(u)
```

Here is a verification that the required equation has been solved:

```
>> err = diff(u,2) + u - f;
>> norm(err)
ans = 1.228647822329242e-12
```

Notice the extraordinary compactness of this solution: one line to define the Green function, one to define the right-hand side, and one to compute the integral of the product of the two.

Although the use of Green functions to solve boundary value problems is a fundamental mathematical idea, it is not usually the best approach numerically. Within the chebfun system, there are better ways to solve an equation like $u'' + u = f$, discussed in another work with Bornemann and Driscoll [9].

Example 7. *Global optimization in two dimensions.* In the chebfun system the global minimization of a function of one variable, whether globally smooth or just piecewise smooth, is routine. Now suppose we have a function of two variables defined on a rectangle, like this one:

$$f(x, y) = \sin(6x) + \sin(5y) + \sin(8x + 3y), \quad -1 \le x, y \le 1. \tag{4.4}$$

The left side of Figure 4.8 plots contours of the function, and it is clear that there are several local minima. One way to find the global minimum is to minimize in one direction at a time. The following code segment defines an anonymous function `fx(x)` whose value for each x is a chebfun in y corresponding to that value of x. The function g is then constructed as the chebfun in x of the minima of these functions `fx(x)` (right side of Figure 4.8). It is continuous but only piecewise differentiable, and the breakpoints apparent in the right-hand plot of Figure 4.8 have been found by automatic edge detection. By minimizing this piecewise smooth univariate function g, one obtains the global minimum of the original bivariate function, all with just three

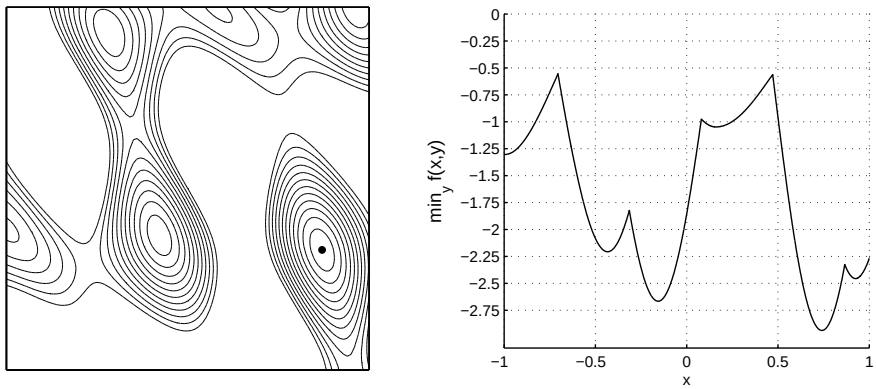

FIG. 4.8. *On the left, contours of the function f of* (4.4) *at levels* $-2.75, -2.5, \ldots, 0$. *On the right, the piecewise smooth chebfun $g(x)$ whose value for each x is* $\min_y f(x,y)$. *The global minimum is marked on the left.*

lines of MATLAB code. The position (`xmin`,`ymin`) of the minimum is also calculated and is marked by a dot in the figure.

```
>> fx = @(x) chebfun(@(y) sin(6*x)+sin(5*y)+sin(8*x+3*y));
>> g = chebfun(@(x) min(fx(x)));
>> [minval,xmin] = min(g)
   minval = -2.937379284008478
   xmin = 0.740775338910084
>> [minval2,ymin] = min(fx(xmin))
   minval2 = -2.937379284008477
   ymin = 0.338025338316668
```

The computation is successful, but this is not the best example of an application of piecewise smooth chebfuns. The difficulty is that a good deal of time is spent locating breakpoints so that g can be represented globally to machine precision; but all that care with breakpoints is not so relevant to the problem of interest, for the minimum of g lies away from the breakpoints in the middle of one of its smooth pieces. For this relatively simple function the computation takes a few seconds, but for the more complicated function given as Problem 4 of the SIAM 100-Dollar, 100-Digit Challenge [4], getting the answer takes about ten minutes on our workstations. (That answer is correct to 14 digits, more than the 10 digits needed to win full points in the Challenge.)

Example 8. *Approximation of the Runge function.* Our final example can be regarded as a use of the piecewise smooth chebfun system to illustrate principles of approximation theory; or alternatively, as an exploration of some of the mathematical principles relevant to the numerical representation of complicated functions. Let f be the familiar Runge function,

$$f(x) = \frac{1}{1+25x^2}, \quad x \in [-1,1], \tag{4.5}$$

with poles at $\pm i/5$. Since f is analytic on $[-1,1]$, its polynomial interpolants in Chebyshev points will converge geometrically (at the rate $O(\rho^{-n})$ with $\rho = 1/5 + \sqrt{26/25} \approx 1.22$). On the other hand, what if we approximate f separately on the

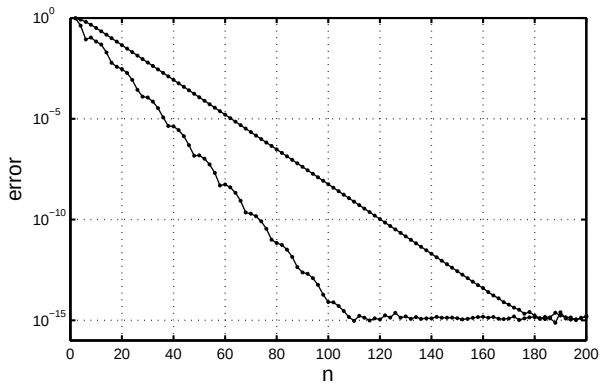

FIG. 4.9. *The upper curve shows convergence of n-point global Chebyshev interpolants to the Runge function* (4.5)*, and the lower curve shows convergence for piecewise approximations involving $n/2$ points each on $[-1,0]$ and $[0,1]$. Both approximations converge geometrically, but the splitting into subintervals improves the efficiency by a constant factor.*

two intervals $[-1,0]$ and $[0,1]$ with the same total number of points? The following code sequence explores this idea, producing the plot of Figure 4.9 that shows that the convergence rate improves by a constant factor. One could pursue the mathematics of such effects to attempt to derive more nearly optimal splitting strategies for piecewise smooth chebfuns, but we have not done this.

```
runge = @(x) 1./(1+25*x.^2);
exact = chebfun(runge);
errp = []; errq = []; nn = 2:2:200;
for n = nn
  p = chebfun(runge,n);
  errp = [errp; norm(p-exact,inf)];
  q = chebfun(runge,runge,[-1 0 1],[n/2 n/2]);
  errq = [errq; norm(q-exact,inf)];
end
semilogy(nn,errp,'.-'), hold on, semilogy(nn,errq,'.-')
hold on, grid on
semilogy(nn,errq,'.-r','linewidth',.9,'markersize',8)
```

5. Discussion. The computations presented in this paper were carried out with chebfun Version 2, released in 2008; most recently they have been confirmed in chebfun v2.0399, released in January 2009. The code is freely available under a BSD-type software license and can be found together with a user's guide and other information at `www.comlab.ox.ac.uk/chebfun`. This is an evolving software system, not yet stable enough for backward compatibility of successive versions to be fully achievable. We have attempted in this paper to discuss a number of algorithmic issues of long-term importance for any system like this, while at the same time illustrating the power of these methods numerically.

One of the referees of the original version of this paper commented that it might be advantageous for users to have the option of constructing chebfuns to an accuracy weaker than machine precision. In fact, such an option has been introduced in recent releases of chebfun (via the parameter `chebfunpref('eps')`), though we do not

currently recommend its use in most applications.

Acknowledgments. All of the chebfun project builds on the original version of the system developed by the third author with Zachary Battles during 2002–2005. In addition we have benefitted from discussions with Folkmar Bornemann, Anne Gelb, Michael Overton, Simon Scheuring, and Jared Tanner. Most importantly, Toby Driscoll is also one of the authors of the chebfun system and the principal author of the related "chebop" system for chebfun solution of differential equations [9].

REFERENCES

[1] Z. Battles, *Numerical Linear Algebra for Continuous Functions,* DPhil thesis, Oxford University Computing Laboratory, 2006.

[2] Z. Battles and L. N. Trefethen, An extension of MATLAB to continuous functions and operators, *SIAM J. Sci. Comp.* 25 (2004), 1743–1770.

[3] J.-P. Berrut and L. N. Trefethen, Barycentric Lagrange interpolation, *SIAM Review* 46 (2004) 501–517.

[4] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel, *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*, SIAM, 2005.

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

[6] D. Day and L. Romero, Roots of polynomials expressed in terms of orthogonal polynomials, *SIAM J. Numer. Anal.* 43 (2005), 1969–1987.

[7] C. de Boor, *A Practical Guide to Splines,* Springer, 1978.

[8] A. R. Curtis and M. J. D. Powell, Using cubic splines to approximate functions of one variable to prescribed accuracy, AERE Report No. 5602, Harwell Laboratory, England, 1967.

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

[10] K. O. Geddes, Near-minimax polynomial approximation in an elliptical region, *SIAM Journal on Numerical Analysis* 15 (1978), 1225–1233.

[11] A. Gelb and E. Tadmor, Adaptive edge detectors for piecewise smooth data based on the *minmod* limiter, J. Sci. Comput. 28 (2006), 279–306.

[12] I. J. Good, The colleague matrix, a Chebyshev analogue of the companion matrix, *Quart. J. Math.* 12 (1961), 61–68.

[13] G. Helmberg and P. Wagner, Manipulating Gibbs' phenomenon for Fourier interpolation, *J. Approx. Th.* 89 (1997), 308–320.

[14] N. J. Higham, The numerical stability of barycentric Lagrange interpolation, *IMA J. Numer. Anal.* 24 (2004), 547–556.

[15] Maplesoft, a division of Waterloo Maple Inc., *Maple user Manual,* Toronto, 2005–2008.

[16] M. J. D. Powell, Curve fitting by splines in one variable, in J. G. Hayes, ed., *Numerical Approximation of Functions and Data,* The Athlone Press, U. of London, 1970, pp. 65–83.

[17] 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), 156–159.

[18] L. N. Trefethen, Computing numerically with functions instead of numbers, *Math. in Comp. Sci.* 1 (2007), 9–19.

[19] L. N. Trefethen, Householder triangularization of a quasimatrix, *IMA J. Numer. Anal.,* to appear.

[20] S. Wolfram, *The Mathematica Book,* 5th ed., Wolfram Media, 2003.