SIAM J. SCI. COMPUT.
Vol. 35, No. 6, pp. C495–C518

AN EXTENSION OF CHEBFUN TO TWO DIMENSIONS*

ALEX TOWNSEND† AND LLOYD N. TREFETHEN†

Abstract. An object-oriented MATLAB system is described that extends the capabilities of Chebfun to smooth functions of two variables defined on rectangles. Functions are approximated to essentially machine precision by using iterative Gaussian elimination with complete pivoting to form "chebfun2" objects representing low rank approximations. Operations such as integration, differentiation, function evaluation, and transforms are particularly efficient. Global optimization, the singular value decomposition, and rootfinding are also extended to chebfun2 objects. Numerical applications are presented.

Key words. MATLAB, Chebfun, Chebyshev polynomials, low rank approximation

AMS subject classifications. 41A10, 65D05

DOI. 10.1137/130908002

1. Introduction.

In scientific computing it is common that models are continuous even though discretizations are used in their solution. Abstraction, in the object-oriented programming sense of the term, is a powerful idea to bridge the gap between these models and the numerical algorithms that solve them. Chebfun2 is a software system written in MATLAB that exploits abstraction to compute with bivariate functions and is the first extension of Chebfun to two dimensions.

Chebfun was first released in 2004 [1] and has become a well-established software system for computing in one dimension [41]. For smooth functions $f : [-1, 1] \to \mathbb{R}$, a "chebfun" is a polynomial interpolant of $f(x)$ through $n+1$ Chebyshev points, given by

$$x_j = \cos\left(\frac{j\pi}{n}\right), \qquad 0 \le j \le n. \tag{1.1}$$

The polynomial degree is adaptively chosen so that the chebfun approximates f, on the whole interval, to machine precision. A chebfun object stores these $n + 1$ values $(f(x_j))_{0\le j\le n}$ as a vector, which are the coefficients in the Lagrange basis,

$$f(x) \approx \sum_{j=0}^{n} f(x_j)\ell_j(x), \qquad \ell_j(x) = \frac{\prod_{i=0,i\ne j}^{n}(x - x_i)}{\prod_{i=0,i\ne j}^{n}(x_j - x_i)}.$$

Chebfun allows numerical computing with functions, and commands such as `norm` and `diff` return the 2-norm and derivative, respectively, [39]. For convenience, we have summarized a selection of Chebfun commands in Table 1.1. A chebfun is, more specifically, a *column chebfun* when `size(f)` returns `inf` × `1`, where `inf` represents

*Submitted to the journal's Software and High-Performance Computing section January 31, 2013; accepted for publication (in revised form) August 20, 2013; published electronically December 5, 2013. Both authors were supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007–2013)/ERC grant agreement 291068. The views expressed in this paper are not those of the ERC or the European Commission, and the European Union is not liable for any use that may be made of the information contained here.

http://www.siam.org/journals/sisc/35-6/90800.html

†Mathematical Institute, Oxford University, Oxford, OX2 6GG, UK (townsend@maths.ox.ac.uk, trefethen@maths.ox.ac.uk). The first author's work was supported by EPSRC grant EP/P505666/1.

TABLE 1.1
A selection of Chebfun commands, their corresponding operations, and underlying algorithms. In addition to the references cited, most of these algorithms are discussed in [42].

Chebfun command	Operation	Algorithm
`chebpoly`	coefficients	fast Fourier transform
`feval`	evaluation	barycentric formula [5]
`sum`	integration	Clenshaw–Curtis quadrature [11, 44]
`diff`	differentiation	recurrence relation [26, p. 34]
`roots`	rootfinding	eigenvalues of colleague matrix [7]
`max`	maximization	roots of the derivative
`qr`	QR decomposition	Householder triangularization [40]

the continuous variable. Alternatively, if `size(f)` is `1` $\times$ `inf`, then it is a *row chebfun*. We can also horizontally concatenate column chebfuns together, as in `[f g h]`, to form a *column quasi-matrix* of size `inf` $\times$ `3`, and row quasi-matrices can be manipulated similarly. As we shall describe, a column quasi-matrix, a row quasi-matrix, and a diagonal matrix can be combined to approximate functions of two variables and are the main ingredients of a chebfun2.

Developing Chebfun2 has brought up many questions, and our answers have guided this process:

1. *What should we represent?* We have chosen to represent (a) scalar valued functions and (b) vector valued functions with two components, both on rectangles. The reason for giving (a) and (b) primary status is that they both have innumerable applications, but are mathematically quite distinct. For example, the notion of rootfinding has one natural meaning for scalar functions and quite a different one for vector functions.
2. *What type of approximations should we use?* We represent scalar valued functions by low rank approximants, i.e., sums of functions of the form $u(y)v(x)$, where $u(y)$ and $v(x)$ are univariate functions (which are represented as chebfuns). Low rank approximations allow us to build on one-dimensional (1D) Chebyshev technology in a powerful way. A vector valued function is represented by a low rank approximant for each component.
3. *How shall we construct them?* We use an algorithm that is mathematically equivalent to Gaussian elimination (GE) with complete pivoting to construct low rank approximations. This is an unusual application of GE in two respects: (1) We use it as an iterative, rather than a direct, algorithm. (2) In principle, we apply GE to functions rather than matrices. We decide when a low rank approximant achieves machine precision by employing an interesting mix of 1D and two-dimensional (2D) resolution tests (see section 2.1).
4. *What are the important operations?* We have taken MATLAB and Chebfun as our guide on this and have overloaded over 130 commands for chebfun2 and chebfun2v objects, which represent scalar and vector functions, respectively. For instance, if `f` is a chebfun2 on the domain $[-1, 1]$, then we have

$$\texttt{trace(f)} = \int_{-1}^{1} f(x,x)dx, \qquad \texttt{flipud(f)} = f(x,-y).$$

The meaning of some operations is obvious, like `diff` for computing derivatives in the x or y variable. For others, like `svd`, a clear choice is suggested by the mathematics, and Chebfun2 operations are designed whenever possible to be the

continuous analogues of their MATLAB or Chebfun predecessors.

5. *What reliable and efficient algorithms are there?* Often our algorithms are motivated by algorithms for discrete objects, like matrices, and we hop back and forth between a discrete and a continuous mode. The recurring theme is the use of low rank approximants, which consist of sums of products of univariate functions. Every algorithm attempts to exploit this low rank structure and build on well-established 1D technology. A full description of the 1D technology is given in [42].

Chebfun2 is publicly available under an open source license accompanied with examples and all the MATLAB code [36].

Throughout this paper we restrict our attention to scalar and vector valued functions defined on the default unit square, i.e., $[-1,1]^2$, though the software permits easy treatment of general rectangular domains.

In the next section we describe how GE with complete pivoting can be used as a practical algorithm for approximating a function of two variables. In section 3 we describe the Chebfun2 algorithms for integration, differentiation, function evaluation, and transforms, which are based on 1D technology. Section 4 presents the vector form of Chebfun2 for operations to implement "div, grad, curl, and all that" for vector valued functions. In section 5 we introduce an algorithm for global optimization and, as an example, use it to solve a problem from the SIAM 100-Digit Challenge [6, 38]. In section 6 we describe the singular value decomposition (SVD) of a chebfun2 and numerically demonstrate that GE with complete pivoting can compute near-optimal low rank approximations. In section 7 we discuss the current rootfinding capabilities of Chebfun2. Finally, in section 8 we describe two prospects for further work.

We first announced the release of Chebfun2 in an article in SIAM News [37].

2. Low rank function approximation. Given a continuous bivariate function $f : [-1,1]^2 \to \mathbb{R}$, the optimal rank k approximation in the L^2-norm is given by the SVD [33]. The SVD is defined, at least formally, as

$$f(x,y) = \sum_{j=1}^{\infty} \sigma_j \phi_j(y) \psi_j(x), \tag{2.1}$$

where $\sigma_1, \sigma_2, \ldots,$ is a nonincreasing real sequence of *singular values*, and the sets $\{\phi_1(x), \phi_2(x), \ldots,\}$ and $\{\psi_1(y), \psi_2(y), \ldots,\}$ are orthonormal functions in $L^2([-1,1])$. Each term in (2.1) is an "outer product" of two univariate functions, called a *rank* 1 *function*. The optimal rank k approximation to f in the L^2-norm can be found by truncating (2.1) after k terms,

$$f(x,y) \approx f_k(x,y) = \sum_{j=1}^{k} \sigma_j \phi_j(y) \psi_j(x), \qquad \|f - f_k\|_{L^2([-1,1]^2)} = \left(\sum_{j=k+1}^{\infty} \sigma_j^2 \right)^{\frac{1}{2}}.$$

Fortunately, the sequence $\sigma_1, \sigma_2, \ldots,$ decays rapidly for smooth functions. The convergence rates for the singular values have a similar flavor to univariate approximation theory: The smoother the function, the faster the singular values decay. Some results to this effect can be found in [18, 25, 46], and we hope to discuss the convergence question further in a separate publication.

Numerically, a rank k approximant to f can be computed by sampling it on an $n \times n$ Chebyshev tensor grid, taking the matrix of sampled values, and computing its

Algorithm: GE with complete pivoting for functions

Input: A function $f = f(x, y)$ on $[-1, 1]^2$ and a tolerance **tol**

Output: A low rank approximation $f_k(x, y)$ satisfying $|f - f_k| <$ **tol**

$e_0(x, y) = f(x, y)$, $f_0(x, y) = 0$

for $k = 0, 1, 2, \ldots,$

$|e_k(x_k, y_k)| = \max(|e_k(x, y)|), \qquad (x, y) \in [-1, 1]^2$

if $|e_k(x_k, y_k)| <$ **tol**, **stop**

$e_{k+1}(x, y) = e_k(x, y) - e_k(x_k, y)e_k(x, y_k)/e_k(x_k, y_k)$

$f_{k+1}(x, y) = f_k(x, y) + e_k(x_k, y)e_k(x, y_k)/e_k(x_k, y_k)$

end

FIG. 2.1. *Iterative GE with complete pivoting for approximation of functions of two variables. The first k steps construct a rank k approximation to f. Our numerical algorithm is based on a discretization of this continuous idealization.*

matrix SVD. The first k singular values and left and right singular vectors form the optimal rank k approximation to the sampled matrix in the discrete 2-norm. Functions can be constructed from each left (right) singular vector by polynomial interpolation, and provided n is sufficiently large, these will be good approximations to the first k singular functions. This process constructs an approximate SVD of a function and requires $\mathcal{O}(n^3)$ operations and n^2 evaluations of f. Instead, we use GE with complete pivoting to compute a near-optimal rank k approximation in $\mathcal{O}(k^2n + k^3)$ operations.

Here is our algorithm of GE with complete pivoting applied to a bivariate function f. First, we define $e_0 = f$ and find an approximation to the location of $\max |e_0(x, y)|$ over $[-1, 1]^2$, say (x_0, y_0). Then we construct a rank 1 function

$$f_1(x, y) = \frac{e_0(x_0, y)e_0(x, y_0)}{e_0(x_0, y_0)} = d_1 c_1(y) r_1(x), \qquad d_1 = \frac{1}{e_0(x_0, y_0)},$$

which interpolates f along the two lines $y = y_0$ and $x = x_0$. We calculate the residual $e_1 = f - f_1$ and repeat the same procedure to form a rank 2 function

$$f_2(x, y) = f_1(x, y) + \frac{e_1(x_1, y)e_1(x, y_1)}{e_1(x_1, y_1)} = f_1(x, y) + d_2 c_2(y) r_2(x),$$

where (x_1, y_1) is an approximate location to the maximum of $|e_1(x, y)|$. The function f_2 interpolates f along $x = x_0$, $x = x_1$, $y = y_0$, and $y = y_1$. We continue constructing successive approximations $f_1, f_2, \ldots, f_k$, where f_k interpolates f along $2k$ lines, until an approximate maximum of $|e_k| = |f - f_k|$ falls below relative machine precision. Figure 2.1 gives the pseudocode for this algorithm. Note that each rank 1 function is the product of two univariate functions, which are represented very efficiently by Chebfun.

The number of steps for convergence of this algorithm is function dependent, and in practice, we usually observe that the approximate maxima $d_1, \ldots, d_k$ decay at approximately the same rate as the singular values. Surprisingly often, only a few steps of GE are required to approximate a function to machine precision.

We call $(x_0, y_0), \ldots, (x_{k-1}, y_{k-1})$ the *pivot locations* and $d_1, \ldots, d_k$ the *pivot values*. We also refer to the functions $c_1(y), \ldots, c_k(y)$ and $r_1(x), \ldots, r_k(x)$ as the *column*

slices and *row slices*, respectively. After constructing k successive approximations $f_1, \ldots, f_k$, we say we have performed k steps of GE.

2.1. Algorithmic details. So far we have described GE at the continuous level, in terms of the manipulation of functions. However, to achieve machine precision with the speed of numerical linear algebra we actually work with a discretized version of the algorithm of Figure 2.1. The discretized algorithm splits into two stages, with the first designed to find candidate pivot locations, i.e., good approximate locations of the absolute maxima, and the second to ensure we have sufficiently sampled the column and row slices.

Stage 1: Finding candidate pivot locations and rank k. First we sample f on a 9×9 Chebyshev tensor grid and perform at most three steps of GE. If we find the sampled matrix can be approximated to machine precision by a rank 1, 2, or 3 matrix, then we move on to stage 2; otherwise, we sample on a 17×17 Chebyshev tensor grid and perform at most five steps of GE. We proceed to stage 2 if a matrix of rank 5, or less, is sufficient. We continue sampling on nested Chebyshev grids of size 9, 17, 33, 65, and so on until we discover the sampled matrix can be approximated to machine precision by a matrix of rank 3, 5, 9, 17, and so on.

Thus, stage 1 approximates a $\left(2^{j+2}+1\right) \times \left(2^{j+2}+1\right)$ matrix by a matrix of rank $2^j + 1$, or less, for $j \geq 1$. Generically, if $f : [-1,1]^2 \to \mathbb{R}$ can be approximated to essentially machine precision by a rank k function, then stage 1 samples f on a $\left(2^{j+2}+1\right) \times \left(2^{j+2}+1\right)$ tensor grid, where $j = \lceil \log_2(k-1) \rceil$, and k steps of GE are required. Since

$$2^{\lceil \log_2(k-1) \rceil + 2} + 1 \leq 8k + 1 = \mathcal{O}(k),$$

stage 1 requires $\mathcal{O}\left(k^3\right)$ operations. We store the pivot locations used in the k successful steps of GE and go to stage 2.

Stage 2: Resolving column and row slices. Stage 1 has determined a candidate set of pivot locations required to approximate f, and stage 2 is designed to ensure that the column and row slices are sufficiently sampled. For instance, $f(x,y) = x\cos(100y)$ is a rank 1 function, so stage 1 completes after sampling on a 9×9 Chebyshev tensor grid even though 147 Chebyshev samples are needed to resolve the oscillations in the y-direction. For efficiency we only sample f on a *k-skeleton* of a tensor grid, i.e., a subset consisting of k columns and rows of the grid, and perform GE on that skeleton. For example, Figure 2.2 shows the 4-skeleton used when approximating Franke's function [14],

$$\begin{aligned} f(x,y) = {} & \frac{3}{4} e^{-\left((9x-2)^2+(9y-2)^2\right)/4} + \frac{3}{4} e^{-\left((9x-1)^2/49-(9y+1)/10\right)} \\ & + \frac{1}{2} e^{-\left((9x-7)^2+(9y-3)^2\right)/4} - \frac{1}{5} e^{-\left((9x-4)^2+(9y-7)^2\right)}. \end{aligned} \tag{2.2}$$

After k steps of GE we have sampled the column and row slices at Chebyshev points. Following the procedure used by Chebfun, we convert each column and row slice to coefficients using the fast Fourier transform to ensure that their tails have decayed to essentially machine precision. Figure 2.2 shows the Chebyshev coefficients for the column slices used to approximate (2.2). For instance, if the column slices are not resolved with 33 points, then we sample f at 65 points along each column and repeat k steps of GE. We continue increasing the sampling along columns and rows until we have resolved them. Since the sets of 9, 17, 33, ..., Chebyshev points are nested, we

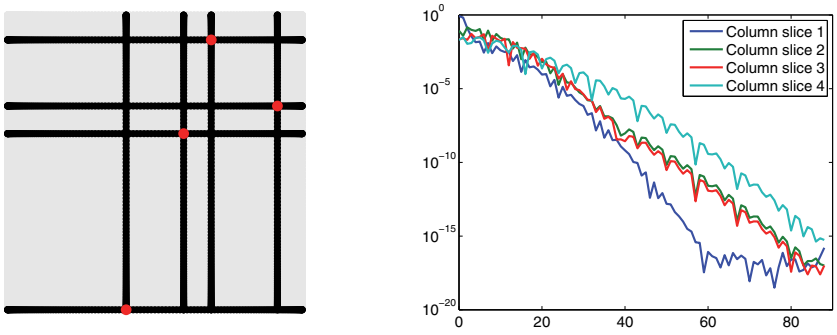

FIG. 2.2. *Left: The skeleton used in stage* 2 *of the construction algorithm for approximating Franke's function* (2.2). *Stage* 2 *only samples* f *on the skeleton, i.e., along the black lines. Right: The Chebyshev coefficients of the four column slices. The coefficients decay to machine precision, indicating that the column slices have been sufficiently sampled.*

can always pivot at the same locations as determined in stage 1. If the column slices require degree $m-1$ polynomial interpolants and the row slices require degree $n-1$ polynomial interpolants, then this stage samples f at, at most,

$$k\left(2^{\lceil \log_2(n) \rceil} + 2^{\lceil \log_2(m) \rceil}\right) \leq 2k(n+m)$$

points. We then perform k steps of GE on the selected rows and columns, requiring $\mathcal{O}\left(k^2(n+m)\right)$ operations.

At the end of the construction we have approximated a function f as

$$f(x,y) \approx f_k(x,y) = \sum_{j=1}^{k} d_j c_j(y) r_j(x).$$

We store the column slices in a column quasi-matrix $C(y) = [c_1(y), \ldots, c_k(y)]$, where $c_j(y)$ is a column chebfun. Similarly, the row slices are stored as chebfuns in a row quasi-matrix $R(x)$, and the pivot values $d_1, \ldots, d_k$ are stored as a vector. We can also write the approximant as

$$f_k(x,y) = C(y)DR(x), \quad D = \operatorname{diag}\left(d_1, \ldots, d_k\right). \tag{2.3}$$

This representation highlights how GE decouples the x and y variables, which will be very useful for tensor product operations (see section 3).

Figure 2.3 shows contour plots with pivot locations of six chebfun2 objects that approximate bivariate functions to machine precision. For example, the following Chebfun2 code produces the top-left plot:

```
f = chebfun2(@(x,y) 1./(1+100*(x.^2+y.^2).^2))
contour(f), hold on, plot(f,'.k')
```

Remark. We deliberately do not represent each column and row slice as an independent chebfun with the minimal degree. Instead we ensure all the column slices have the same degree. Similarly, all the row slices have the same degree (which can be different from the degree of the column slices). This is important for efficiency since subsequent operations can then be vectorized.

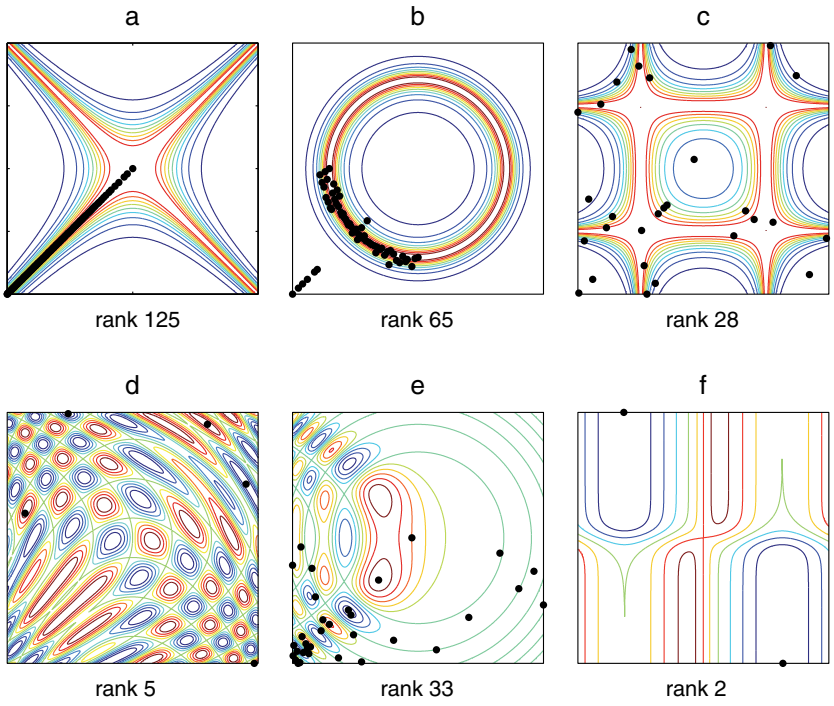

FIG. 2.3. *Contour plots for six functions f on $[-1,1]^2$, with the pivot locations marked by dots:* (a) $1/(1+100(x^2+y^2)^2)$; (b) $1/(1+100(\frac{1}{2}-x^2-y^2)^2)$; (c) $1/(1+1000((x-\frac{1}{2})^2(y+\frac{1}{2})^2(x+\frac{1}{2})^2(y-\frac{1}{2})^2))$; (d) $\cos(10(x^2+y))+\sin(10(x+y^2))$; (e) $Ai(5(x+y^2))Ai(-5(x^2+y^2))$; (f) $\tanh(10x)\tanh(10y)/\tanh(10)^2+\cos(5x)$.

The construction process is based on mathematical reasoning but has some heuristic features, as it must since it employs another heuristic algorithm, the Chebfun constructor. In practice, we find the Chebfun2 constructor relatively robust for smooth functions, and one of the many tricks that we employ is a sample test, where the final approximant is evaluated several times at arbitrary locations to ensure that the approximant is accurate. No doubt failures are possible in principle, but currently other algorithmic issues are more pressing.

2.2. Object composition. One or more chebfun2 objects can be combined together to make new ones; the object-oriented programming term for this is *object composition*. Table 2.1 summarizes some composition operations that are possible with chebfun2 objects. For example, if `f` is a chebfun2, then `cos(f)`, `exp(f)`, and `f.^2` return chebfun2 objects corresponding to the cosine, exponential, and square of `f`, respectively. To carry out these operations the Chebfun2 constructor is called, so that the chebfun2 for `exp(f)`, for example, is constructed by applying the iterative GE algorithm in the usual manner to samples of $\exp(f)$ on a sequence of grids.

An alternative to calling the Chebfun2 constructor for some composition operators might be to employ a compression algorithm based on a fast SVD of low rank functions. Specifically, the fast SVD as described in section 6 could be used to compute `f+g`, which would be analogous to the compression algorithm employed for the addition of two low rank matrices [3]. For other operations such as `f.*g`, `f.^n`, and

TABLE 2.1
A selection of object composition commands in Chebfun2. In each case the result is computed by calling the Chebfun2 constructor to generate a low rank approximation to machine precision.

Chebfun2 command	Operation
`+`, `-`	addition, subtraction
`.*`, `./`	multiplication, division
`cos`, `sin`, `tan`	trigonometric functions
`cosh`, `sinh`, `tanh`	hyperbolic functions
`exp`	exponential
`power`	integer powers

`exp(f)`, the mathematical bound on the rank of the resulting chebfun2 is large and compression algorithms bring less significant gains. However, we are not ruling out the possibility that such approaches may play a role in Chebfun2 for object composition operations in the future.

2.3. Related approaches. Ideas related to GE for functions have been developed by various authors under various names, though the connection with GE is usually not mentioned. We now briefly summarize some of the ideas of *pseudoskeleton approximation* [16], *adaptive cross approximation* (ACA) [3], interpolative decompositions [21], *Geddes–Newton series* [8], and *rank-revealing decompositions* [22, 31].

Pseudoskeletons, interpolative decompositions, and adaptive cross approximation. Pseudoskeletons were developed by Goreinov, Tyrtyshnikov, and Zamarashkin [16] and Tyrtyshnikov [43] and approximate a matrix $A \in \mathbb{R}^{n\times n}$ by a matrix of low rank by computing the CUR decomposition[1] $A \approx CUR$, where $C \in \mathbb{R}^{n\times k}$ and $R \in \mathbb{R}^{k\times n}$ are subsets of the columns and rows of A, respectively, and $U \in \mathbb{R}^{k\times k}$. Selecting *good* columns and rows of A is of paramount importance, and this can be achieved via maximizing volumes [15], randomized techniques [13, 21], or ACA. ACA was mainly developed by Bebendorf and constructs a skeleton approximation with columns and rows selected adaptively [2]. The selection of a column and row corresponds to choosing a pivot location in GE, where the pivoting entry is the element belonging to both the column and row. If the first k columns and rows are selected, then

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} - \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} A_{11}^{-1} \begin{pmatrix} A_{11} & A_{12} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & S \end{pmatrix}, \qquad A_{11} \in \mathbb{R}^{k\times k}, \tag{2.4}$$

where $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is the Schur complement of A_{22} in A. The relation (2.4) is also found in [3, p. 128] and can be compared with Theorem 1.4 of [35] to show that ACA and GE are equivalent. This connection remains even when any k columns and rows are selected [35, Theorem 1.8]. The use of the continuous analogue of GE with complete pivoting by Chebfun2 is equivalent to the continuous analogue of ACA with column and row selection via complete pivoting.

In practice, ACA is used to compute low rank approximations of matrices that are derived by sampling functions [3], and it has been used extensively by Hackbusch and others for constructing hierarchical representations of matrices [19, 20].

Closely related to the above developments are approximate matrix decompositions called *interpolative decompositions*, which involve selecting an index set J such that

[1] The pseudoskeleton literature writes $A \approx CGR$. More recently, it has become popular to follow the nomenclature of [13] and replace G by U.

$A = A(:, J)X$ for some X of size $|J| \times n$ [10, 21], where potentially conditions are imposed on X. A general theme employed in the literature related to interpolative decompositions is the use of randomness to select an appropriate subset of columns of A [21], whereas pseudoskeleton decompositions often select columns, at least in principle, based on maximizing a certain determinant.

Geddes–Newton series. Independently of ACA, Chapman in his Ph.D. thesis [9] in 2003 developed a theoretical framework for what he calls the *Geddes–Newton series*. For a function f and a *splitting point* $(a, b) \in [-1, 1]$ such that $f(a, b) \neq 0$, the *splitting operator* $\Upsilon_{(a,b)}$ is defined as

$$\Upsilon_{(a,b)} f(x, y) = \frac{f(x, b) f(a, y)}{f(a, b)},$$

and this is the rank 1 approximation to f obtained after one step of GE with pivot location (a, b). The splitting operator is now applied to the function

$$f(x, y) - \Upsilon_{(a,b)} f(x, y)$$

with a different splitting point. Repeating this process k times is equivalent to applying k steps of GE on f.

Carvajal, Chapman, and Geddes used GE for the quadrature of symmetric functions [8], and it was their work that initially led us to develop the algorithm for Chebfun2. Their integration algorithm first maps a function to $[0, 1]^2$ and then decomposes it into symmetric and antisymmetric parts, ignoring the antisymmetric part since it integrates to zero. However, Carvajal, Chapman, and Geddes employ a specialized pivoting strategy designed to preserve symmetry, whereas we use complete pivoting. Geddes–Newton series stand out for being the only case of an iterative GE type of algorithm that we know of in the literature introduced mainly to approximate functions rather than matrices [9].

Rank-revealing decomposition. The SVD computes the optimal rank k approximation to a matrix in the 2-norm, though it is relatively expensive. It is well known that the QR, UTV^T, and LU decompositions can also be used for constructing low rank matrix approximations that can be near-optimal [27, 23, 31, 34]. In particular, the LU decomposition computed via GE with complete pivoting is usually an excellent rank-revealing decomposition, and this effect for matrices is related to why the rank of a chebfun2 approximant to a function is usually close to the minimal rank required to achieve machine precision.

It would be interesting to investigate the relationship of the algorithm in the Chebfun2 constructor to rank-revealing matrix factorizations in greater depth. For example, a famous example by Kahan shows that failure of GE with complete pivoting is possible in rare cases (see Higham [22, p. 183]). We do not know if such examples can be adapted to produce examples of smooth functions that would cause failure of the Chebfun2 constructor.

3. Quadrature and other tensor product operations. If `f` is a matrix, then the MATLAB command `sum(f)`, or `sum(f,1)`, returns a row vector representing the sum of each column, and `sum(f,2)` returns a column vector after summing up the rows. If `f` is a chebfun2, then we have

$$\texttt{sum(f,1)} = \int_{-1}^{1} f(x, y)dy, \qquad \texttt{sum(f,2)} = \int_{-1}^{1} f(x, y)dx,$$

represented as row and column chebfuns, respectively. Thus in the language of probability, the sum command computes marginal distributions. The algorithm underlying these operations exemplifies how Chebfun2 takes advantage of its low rank representations, together with the established 1D Chebfun technology, to carry out substantial computations much faster than one might expect.

For example, consider the computation of `sum(f,2)`, whose result is a chebfun in the y variable. If f is of rank k, the necessary formula is

$$\int_{-1}^{1} f(x,y)dx = \int_{-1}^{1} \sum_{j=1}^{k} d_j c_j(y) r_j(x) dx = \sum_{j=1}^{k} d_j c_j(y) \left(\int_{-1}^{1} r_j(x)dx \right). \tag{3.1}$$

It follows that the evaluation of `sum(f,2)` reduces to the integration of k chebfuns, in parentheses in (3.1), and the addition of the corresponding multiples of $c_j(y)$ in Chebfun. The integration is done by calling Chebfun's `sum` command, which utilizes the Clenshaw–Curtis quadrature rule

$$\int_{-1}^{1} r_j(x)dx = \sum_{i=0}^{n} w_i r_j(x_i), \qquad 1 \le j \le k,$$

where x_i are the Chebyshev nodes (1.1) and w_i are the quadrature weights.

Following familiar MATLAB syntax, the command `sum(sum(f))` delivers the definite integral over $[-1,1]^2$ by integrating over the y variable, constructing an intermediary row chebfun, and then integrating this. The command `sum2(f)` computes the same quantity more efficiently because an intermediary chebfun is not constructed. The necessary result corresponds to the integration of $2k$ chebfuns in which all the column (row) slices are of the same degree. Thus, the Clenshaw–Curtis quadrature can be vectorized, and only two sets of quadrature weights are computed using Chebfun's algorithm based on [44].

Thus, integration is very efficient for a chebfun2, because it is carried out via 1D quadrature rules. A similar surprising efficiency extends to other tensor product operators, which represent a significant part of Chebfun2 functionality.

DEFINITION 3.1. *A tensor product operator $\mathcal{L}$ is a linear operator on functions of x and y with the property that if $f(x,y) = c(y)r(x)$, then $\mathcal{L} = \mathcal{L}_y(c)\mathcal{L}_x(r)$, for some operators $\mathcal{L}_y$ and $\mathcal{L}_x$. Thus, if f has rank k, then*

$$\mathcal{L}\left(\sum_{j=1}^{k} d_j c_j(y) r_j(x) \right) = \sum_{j=1}^{k} d_j \left(\mathcal{L}_y c_j(y) \right) \left(\mathcal{L}_x r_j(x) \right).$$

A tensor product operation can be computed with $\mathcal{O}(k)$ calls to well-established Chebfun algorithms since $\mathcal{L}_y$ and $\mathcal{L}_x$ act on functions of one variable. Four important examples of tensor product operators are integration (described above), differentiation, evaluation, and the computation of Chebyshev coefficients.

Differentiation. If `f` is an $n \times n$ matrix, then the MATLAB command `diff(f,N)`, or `diff(f,N,1)`, returns an $(n-N) \times n$ matrix of Nth order differences along the columns, while `diff(f,N,2)` calculates the differences along the rows. For a chebfun2 the same syntax represents differentiation:

$$\texttt{diff(f,N,1)} = \frac{\partial^N f}{\partial y^N}, \qquad \texttt{diff(f,N,2)} = \frac{\partial^N f}{\partial x^N}.$$

Differentiation is the tensor product of univariate differentiation since

$$\frac{\partial^N f}{\partial y^N} = \sum_{j=1}^{k} d_j \frac{\partial^N c_j(y)}{\partial y^N} r_j(x), \qquad \frac{\partial^N f}{\partial x^N} = \sum_{j=1}^{k} d_j c_j(y) \frac{\partial^N r_j(x)}{\partial y^N}.$$

These functions are computed with k calls to the Chebfun `diff` command.

These differentiation and integration algorithms mean that vector calculus on chebfun2 and chebfun2v objects is very efficient (see section 4).

Function evaluation. In MATLAB, indexing into a matrix selects a subset of its elements. In Chebfun2, the analogous operation is evaluation of `f` at various points, with `f(x,y)` returning the value of `f` at (x, y). Evaluation is a tensor product operation since

$$f(x, y) = \sum_{j=1}^{k} d_j c_j(y) r_j(x),$$

which involves the evaluation of $2k$ univariate functions, and this computation is carried out by using the barycentric formula [5, 32]. This formula has been vectorized so that multiple column and row slices of the same degree are evaluated efficiently.

Computation of Chebyshev coefficients. If `f` is a chebfun, `chebpoly(f)` computes its Chebyshev expansion coefficients, i.e., the coefficients in the expansion

$$f(x) = \sum_{j=0}^{n} \alpha_j T_j(x),$$

in $\mathcal{O}(n \log n)$ operations using the fast Fourier transform. This is a very important operation and the main reason why Chebfun uses a Chebyshev basis instead of, for example, a basis of Legendre polynomials.

Analogously, the Chebfun2 command `X=chebpoly2(f)` computes the expansion coefficients of a chebfun2, i.e., the coefficients in the expansion

$$f(x, y) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \alpha_{ij} T_i(y) T_j(x),$$

and the $m \times n$ matrix returned represents these coefficients. If `f` is of rank k, then the coefficient matrix is also of rank k, and the command `[A D B]=chebpoly2(f)` requires only $\mathcal{O}(k(n \log n + m \log m))$ operations since it returns the coefficient matrix in its low rank form, i.e., $X = ADB^T$, where $A, B \in \mathbb{R}^{n \times k}$ and $D \in \mathbb{R}^{k \times k}$ is diagonal. This operation is computed with $2k$ calls to the Chebfun command `chebpoly` since[2]

$$\texttt{chebpoly2(f)} = \sum_{j=1}^{k} d_j \texttt{chebpoly}(c_j)^T \texttt{chebpoly}(r_j),$$

where each c_j is evaluated at m Chebyshev points and each r_j at n Chebyshev points using the fast Fourier transform. For evaluation at points that are a subset of a

[2]The Chebfun command `chebpoly` always returns a row vector of coefficients, as of version 4.2.2194, though this may change in a future release.

TABLE 3.1

A selection of scalar Chebfun2 commands corresponding to tensor product operations. In each case the result is computed with much greater speed than one might expect because the algorithms can take advantage of the rank k structure.

Chebfun2 command	Operation
`sum`, `sum2`	integration
`cumsum`	cumulative integration
`prod`	product integration
`cumprod`	cumulative product integration
`norm`	L^2-norm
`diff`	differentiation
`chebpoly2`	fast Fourier transform
`f(x,y)`	evaluation
`plot`, `surf`, `contour`	plotting
`diag`	univariate function along a diagonal
`trace`	integration along $y = x$
`flipud`, `fliplr`	reverse direction of coordinates
`mean2`, `std2`	mean, standard deviation

Chebyshev grid, this approach can be significantly faster than using the barycentric formula.

Tensor product operations represent the ideal situation where well-established Chebfun technology can be exploited for computing in two dimensions. Table 3.1 shows a selection of Chebfun2 commands that are tensor product operations. If a prescribed function happens to be univariate, then $k = 1$, and these Chebfun2 commands are essentially the same as their univariate counterparts.

4. Chebfun2v objects and vector calculus. As mentioned in the introduction, Chebfun2 is also designed to work with vector valued functions defined on rectangles, as well as scalar valued ones. Our convention is to use a lowercase letter for a scalar function, f, and an uppercase letter for a vector function, $F = (f_1, f_2)^T$.

From the point of view of approximation, the vector aspect of Chebfun2 introduces no new complications. A vector function F is represented as two independent scalar functions, and such a representation is constructed by computing a low rank approximation to each component.

The more interesting side of vector Chebfun2 is the set of operations that can be implemented—algorithmically similar to scalar functions, but potentially very useful for applications.

Algebraic operations. The basic nondifferential operations of vector calculus are scalar multiplication fG, vector addition $F + G$, dot product $F \cdot G$, and cross product $F \times G$. Unfortunately, there are two notational inconveniences in MATLAB: (1) The inability to use the "`.`" symbol for the dot product, and (2) the lack of an "`x`" symbol for the cross product. The "`.`" symbol is already used as a special character for referencing, and "`x`" is usually used as a variable name, preventing it from being an operator. Accordingly, in Chebfun2 we have chosen `f.*G`, `F+G`, `dot(F,G)`, and `cross(F,G)` for the four operations. We explain these operations in turn.

Scalar multiplication is the product of a scalar function with a vector function, denoted by `f.*G`, and algorithmically, it is achieved by two scalar function multiplications, one for each component.

Vector addition, denoted by `F+G`, yields another chebfun2v and is computed by adding the two scalar components together. It satisfies the *parallelogram law* $2\|F\|_2^2 +$

$2\|G\|_2^2 = \|F+G\|_2^2 + \|F-G\|_2^2$, which can be verified numerically, as in this example:

```
F = chebfun2v(@(x,y)cos(x.*y),@(x,y)sin(x.*y));
G = chebfun2v(@(x,y)x+y,@(x,y)1+x+y);
abs((2*norm(F)^2 + 2*norm(G)^2) - (norm(F+G)^2 + norm(F-G)^2))
ans = 3.5527e-15
```

The dot product, denoted by `dot(F,G)`, takes two vector functions and returns a scalar function. Algebraically, it is the inner product, i.e., the sum of the products of the two components. If the dot product of two chebfun2v objects takes the value zero at (x_0, y_0), then the vectors $F(x_0, y_0)$ and $G(x_0, y_0)$ are orthogonal.

The cross product, denoted by `cross(F,G)`, is well known as an operation on vector valued functions with three components, and it can also be defined for 2D vector fields by

$$\texttt{cross(F,G)} = f_1 g_2 - f_2 g_1, \qquad \texttt{F} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}, \qquad \texttt{G} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix}.$$

Note that the cross product of two vector valued functions with two components is a scalar function, which can be represented by a chebfun2.

Differential operations. Vector calculus also involves various differential operators defined on scalar or vector valued functions such as gradient ∇f, curl $\nabla \times F$, divergence $\nabla \cdot F$, and Laplacian $\nabla^2 f$.

The gradient of a chebfun2 is represented by a chebfun2v such that

$$\texttt{grad(f)} = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \end{pmatrix},$$

which points in the direction of steepest ascent of `f`. The *gradient theorem* says that the integral of `grad(f)` over a curve depends only on the values of `f` at the endpoints of that curve. We can check this numerically by using the Chebfun2v command `integral`. This command computes the line integral of a vector valued function

$$\texttt{integral(F,C)} = \int_C F(\mathbf{r}) \cdot d\mathbf{r},$$

where C is a smooth curve represented by a complex valued chebfun (see Note below) and $\cdot$ is the dot product. For example,

```
f = chebfun2(@(x,y)sin(2*x)+x.*y.^2);        % chebfun2 object
F = grad(f);                                 % gradient (chebfun2v)
C = chebfun(@(t) t.*exp(100i*t),[0 pi/10]);  % spiral curve
v = integral(F,C); ends = f(pi/10,0)-f(0,0); % line integral
abs(v-ends)                                  % gradient theorem
ans = 5.5511e-16
```

The curl of a 2D vector function is a scalar function defined by

$$\texttt{curl(F)} = \frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial y}, \qquad \texttt{F} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

If the chebfun2v `F` describes a vector velocity field of fluid flow, for example, then `curl(F)` is a scalar function equal to twice the angular speed of a particle in the flow

TABLE 4.1
Vector calculus commands in Chebfun2. In each case the result is computed with greater speed than one might expect because of the exploitation of the tensor product structure (see section 3).

Command	Operation
`+, -`	addition, subtraction
`dot`	dot product
`cross`	cross product
`grad`	gradient
`curl`	curl
`div`	divergence
`lap`	Laplacian
`quiver`	phase portrait

at each point. A particle moving in a gradient field has zero angular speed and hence, `curl(grad(f))` $= 0$, a well-known identity that can also be checked numerically.

The divergence of a chebfun2v is defined as

$$\texttt{div(F)} = \frac{\partial f_1}{\partial x} + \frac{\partial f_2}{\partial y}, \qquad \texttt{F} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

This measures a vector field's distribution of sources or sinks. The Laplacian is closely related and is the divergence of the gradient `lap(f) = div(grad(f))`.

Table 4.1 summarizes the vector calculus commands available in Chebfun2.

Phase portraits. A phase portrait is a graphical representation of a system of trajectories for a two-variable autonomous dynamical system. In Chebfun2 we can plot phase portraits by using the `quiver` command, which has been overloaded to plot the vector field (see Figure 4.1).

In addition, Chebfun2 makes it easy to compute and plot individual trajectories of a vector field. If `F` is a chebfun2v, then `ode45(F,tspan,y0)` solves the autonomous system $dx/dt = f_1(x, y)$, $dy/dt = f_2(x, y)$, where f_1 and f_2 are the first and second components of F, respectively, with the prescribed time interval and initial conditions. This command returns a complex valued chebfun representing the trajectory in the form $x(t) + iy(t)$.

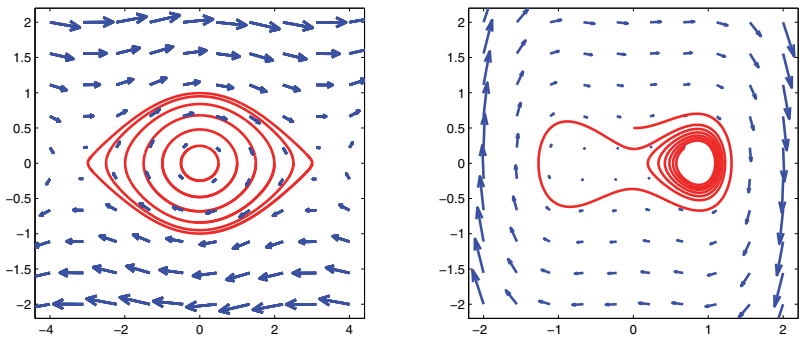

FIG. 4.1. *Phase plane diagrams and example trajectories for the nonlinear pendulum (left) and the Duffing oscillator (right). Red (in the electronic version) lines are trajectories in the phase plane computed using the Chebfun2 command* `ode45`.

Note. As is well known, a pair of real scalar functions f and g can be encoded as a complex function $f + ig$. In the design of Chebfun2 we have debated how much use to make of this trick, which can simplify many operations, but at the same time may be confusing. For example, every (real) chebfun2v might have been realized as a complex chebfun2. We decided not to go that far, and the complex output of `ode45` and `roots` (see section 7), as well as the second input argument to the command `integral` (above) are exceptions. The complex output of `ode45` allows the trajectory's length to be computed with the Chebfun command `arclength`.

The following code produces the phase portrait for a dynamical system corresponding to a nonlinear pendulum, $\dot{x} = y$, $\dot{y} = -\sin(x)/4$, together with some sample trajectories:

```
phasedom=[-4 4 -2 2];              % phase plane domain
F = chebfun2v(@(x,y)y,@(x,y)-sin(x)/4,phasedom);
for ic = .5:.5:3                   % initial conditions
   [t y]=ode45(F,[0 40],[ic,0]); % solve autonomous system
   plot(y,'r'), hold on,           % plot trajectory
end
quiver(F)                          % vector field
arclength(y)                       % length of a trajectory
ans = 14.0386
```

Figure 4.1 shows the resulting image, as well as a corresponding picture for the Duffing oscillator

$$\dot{x} = y, \qquad \dot{y} = \frac{4}{100}y - \frac{3}{4}x + x^3,$$

where the time interval is $[0, 50]$ and the initial condition is $x(0) = 0$, $y(0) = \frac{1}{2}$.

5. Global optimization. If `f` is a matrix, then the MATLAB command `max(f)`, or `max(f,[],1)`, returns a row vector of maximum values along each column of `f`. Similarly, `max(f,[],2)` returns a column vector of maximum values along each row. If `f` is a chebfun2, then the same syntax yields analogous results,

$$\texttt{max(f,[],1)} = \max_{y\in[-1,1]} f(x,y), \qquad \texttt{max(f,[],2)} = \max_{x\in[-1,1]} f(x,y)$$

represented as row and column chebfuns, respectively. Often these chebfuns are only piecewise smooth, and Chebfun's edge detection algorithm is used to decide the locations of the breakpoints [30]. If `f` and `g` are chebfun2 objects, then we do not currently allow `max(f,g)`, because this bivariate function will generally have discontinuities in its first derivative lying along curves, and Chebfun2 cannot represent such nonsmooth functions.

The global 2D maximum of a chebfun2 can be computed by `max(max(f))`, but the same result can be obtained more efficiently by avoiding an intermediary nonsmooth chebfun with `max2(f)`. If a chebfun2 is of rank 1, i.e., $f(x,y) = d_1 c_1(y) r_1(x)$, then the global maximum is computed by noting that

$$\texttt{max2(f)} = d_1 \max\left\{\texttt{max}c_1 \cdot \texttt{max}r_1, \texttt{min}c_1 \cdot \texttt{max}r_1, \texttt{max}c_1 \cdot \texttt{min}r_1, \texttt{min}c_1 \cdot \texttt{min}r_1\right\},$$

where each max and min involves only univariate functions and existing Chebfun algorithms are used [42, Chap. 18]. For $k \geq 2$, however, Chebfun2 uses other algorithms for global optimization, and Table 5.1 summarizes how the command and rank of a

TABLE 5.1

Optimization algorithms in Chebfun2. Various algorithms are used in different situations and the choice depends on the rank k of the chebfun2 and on the particular Chebfun2 command. For $k \geq 2$, the algorithms listed are used to provide initial guesses for a trust-region iteration.

	$k = 1$	$2 \leq k \leq 3$	$k \geq 4$
`minandmax2`, `max2`, or `min2`	Chebfun	convex hull algorithm	grid evaluation
`norm(·,inf)`	Chebfun	convex hull algorithm	power iteration

chebfun2 determine the algorithm employed. All the algorithms, for $k \geq 2$, are based on obtaining initial guesses by one method and then switching to a superlinearly convergent constrained trust-region method,[3] based on [12], in which the iterates are constrained to the domain of the chebfun2.

5.1. Convex hull algorithm. For low rank chebfun2 objects, currently those with rank $2 \leq k \leq 3$, if the global maximum or minimum is desired, then we employ the following algorithm based on the convex hull. Mathematically, this algorithm is not restricted to $k = 2, 3$, but it is inefficient if $k > 3$. For any fixed point $(x, y) \in [-1, 1]^2$ we have

$$f(x,y) = \sum_{j=1}^{k} d_j c_j(y) r_j(x) = \left(\sqrt{|d_1|}\, c_1(y), \ldots, \sqrt{|d_k|}\, c_k(y) \right) \begin{pmatrix} \pm\sqrt{|d_1|}\, r_1(x) \\ \vdots \\ \pm\sqrt{|d_k|}\, r_k(x) \end{pmatrix} = s(y)^T t(x),$$

where $s, t : [-1, 1] \to \mathbb{R}^k$ are continuous functions and the signs are chosen appropriately depending on d_j, $1 \leq j \leq k$. We can define the following parameterizable curves in $\mathbb{R}^k$:

$$S = \left\{ s(y) \in \mathbb{R}^k : y \in [-1, 1] \right\}, \qquad T = \left\{ t(x) \in \mathbb{R}^k : x \in [-1, 1] \right\},$$

and approximate them by discrete sets $\tilde{S} \in \mathbb{R}^{m \times k}, \tilde{T} \in \mathbb{R}^{n \times k}$ by evaluating $s(y)$ and $t(x)$ at m and n Chebyshev points, respectively. As always, m and n are the numbers of Chebyshev coefficients required to resolve the column and row slices (see section 2.1). We then make the following approximation to the global maximum:

$$\begin{aligned} \max\{ f(x,y) : x, y \in [-1,1] \} &= \max\left\{ s(y)^T t(x) : x, y \in [-1,1] \right\} \\ &= \max\left\{ a^T b : a \in S, b \in T \right\} \\ &\approx \max\left\{ a^T b : a \in \tilde{S}, b \in \tilde{T} \right\} \\ &= \max\left\{ a^T b : a \in \operatorname{conv}(\tilde{S}), b \in \operatorname{conv}(\tilde{T}) \right\}, \end{aligned}$$

[3] The current implementation employs the MATLAB command `fmincon`, which is only available in the optimization toolbox. Alternatives are currently under development.

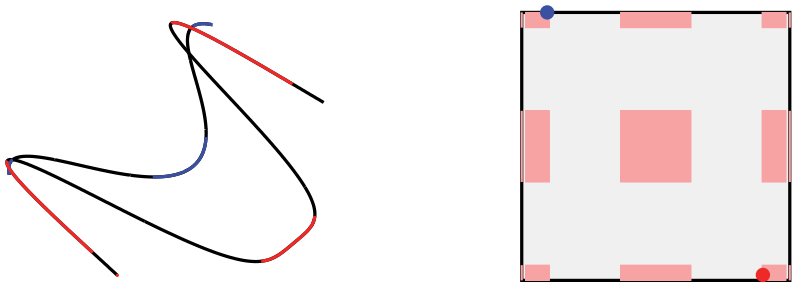

FIG. 5.1. *Left: Convex hull, or convex envelope, in* $\mathbb{R}^2$ *(red and blue in the electronic version) of the column and row slices (black) for the rank* 2 *function* $f(x,y) = 2y\cos(5x^2) + x\sin(2y^2)$. *Right: Gray regions are excluded from* $[-1,1]^2$ *because these points do not lie on the convex hull of the column and row slices. The global minimum and maximum lie in the pink (in the electronic version) region. The red (blue) (in the electronic version) dot is the location of the global maximum (minimum).*

where $\mathrm{conv}(X)$ is the convex hull of X. The convex hulls of $\tilde{S}$ and $\tilde{T}$ are computed using the MATLAB command `convhull`. We discard any point (x,y) if either $t(x) \notin \mathrm{conv}(\tilde{T})$ or $s(y) \notin \mathrm{conv}(\tilde{S})$.

Figure 5.1 shows an example of the convex hull algorithm. Here we take the rank 2 function $f(x,y) = 2y\cos(5x^2) + x\sin(2y^2)$ and show the convex hull of $\tilde{S}$ and $\tilde{T}$ and the corresponding excluded gray region in $[-1,1]^2$.

The convex hull algorithm allows us to obtain candidate regions in $[-1,1]^2$ containing the global extrema, and in these regions we sample the chebfun2 on the Chebyshev tensor grid and take the discrete maximum and minimum as initial guesses in a trust-region iteration.

5.2. Grid evaluation. For the commands `minandmax2`, `max2`, and `min2`, if $k > 3$, then we evaluate the chebfun2 on an $m \times n$ Chebyshev grid using the fast Fourier transform, where m and n are the numbers of Chebyshev coefficients required to represent the column and row slices. From this grid we take the discrete maximum and minimum as initial guesses for the trust-region iteration.

5.3. Power iteration. If the global absolute maximum of $|f(x,y)|$ is required with $k > 3$, such as for `norm(f,inf)`, then we use a fast power iteration as proposed in [4]. The advantage of this method is that it can be much faster when $k \ll \min(m,n)$.

For the power iteration we first sample the column and row slices of a chebfun2 at m and n Chebyshev points, respectively, and obtain a matrix $A \in \mathbb{R}^{m\times n}$ of rank k, though we do not form this matrix explicitly. Given a rank k matrix $A \in \mathbb{R}^{m\times n}$,

$$A = \sum_{j=0}^{k} d_j c_j r_j^T, \qquad c_j \in \mathbb{R}^{m\times 1}, \quad r_j \in \mathbb{R}^{n\times 1}, \quad d_j \in \mathbb{R},$$

the diagonal matrix $B \in \mathbb{R}^{mn\times mn}$ defined by

$$B = \sum_{j=1}^{k} d_j \mathrm{diag}(c_j) \otimes \mathrm{diag}(r_j)$$

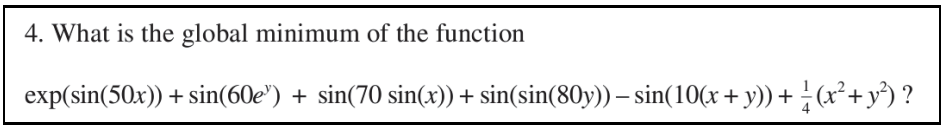

FIG. 5.2. *Problem* 4 *from A Hundred-Dollar, Hundred-Digit Challenge* [38].

has eigenvalues A_{ij} with eigenvectors $e_i \otimes e_j$, where e_i is the ith canonical vector. Hence, the maximum entry of $|A|$ can be computed by performing the power iteration on B. The low rank structure of A is exploited so that the mn diagonal entries of B are not formed explicitly. Moreover, the initial vector $x \in \mathbb{R}^{mn\times 1}$ is stored in Kronecker product form and kept in this form throughout the iteration by compression [4]. This power iteration requires $\mathcal{O}(k^2(m+n))$ operations per iteration.

Example: Global minimum of a complicated function. In February 2002, an article in SIAM News by the second author set a challenge to solve 10 problems each to 10 digits of precision (the solution of each problem was a real number) [38]. Figure 5.2 shows one of the problems, which involves finding the global minimum of a function. Since the term $(x^2+y^2)/4$ grows away from $(0,0)$ while the other terms remain bounded, it can be shown that the global minimum occurs in $[-1,1]^2$ [6].

The function is complicated and oscillatory, but of rank 4, as can be seen by rearranging its terms and using the identity $\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$,

$$f(x,y) = \left(\frac{x^2}{4} + e^{\sin(50x)} + \sin(70\sin(x))\right) + \left(\frac{y^2}{4} + \sin(60e^y) + \sin(\sin(80y))\right) - \cos(10x)\sin(10y) - \sin(10x)\cos(10y).$$

The Chebfun2 algorithm as implemented in the command `min2` finds the global minimum in 0.18 seconds[4] and achieves 12 correct digits.

```
f = @(x,y) exp(sin(50*x)) + sin(60*exp(y)) + sin(70*sin(x)) +...
    sin(sin(80*y)) - sin(10*(x+y)) + (x.^2+y.^2)./4;
g = chebfun2(f); Y = min2(g);           % Global minimum
abs(Y - (-3.306868647475237))           % Compare with exact answer
ans = 4.4098e-13
```

A surprising number of interesting functions of two variables are of low rank, and many that are mathematically of infinite rank are numerically of low rank.

6. Singular value decomposition. If `A` is an $n \times n$ matrix, the MATLAB command `[U,S,V]=svd(A,0)` returns the "economy-sized" SVD, where $U, V \in \mathbb{R}^{n\times k}$ and $S \in \mathbb{R}^{k\times k}$ is a diagonal matrix. If `f` is a chebfun2, then `[U,S,V]=svd(f)`, or `svd(f,0)`, returns the SVD of `f`, where `U`, `V` are column quasi-matrices with k orthonormal columns (in the L^2 sense) and `S` is a $k \times k$ diagonal matrix. Of course this is not the exact SVD, which, in general, would have infinite rank, but a finite rank approximation accurate to machine precision.

If each column of `U` and `V` is a chebfun of degree at most n, then computing the SVD of a rank k chebfun2 requires $\mathcal{O}\left(k^2 n\right)$ operations. Figure 6.1 shows the pseudocode used for the computation. The first step requires the Householder triangularization of column quasi-matrices and uses the Chebfun command `qr` described

[4]Timing was done on a 2012 1.6GHz Intel Core i5 MacBook Air with MATLAB 2012a.

Pseudocode: SVD of a chebfun2

Input: A chebfun2 $f(x,y) = C(y)DR(x)$ (see (2.3))

Output: Quasimatrices with orthonormal columns U_f and V_f, and a diagonal matrix S_f

1. Householder triangularization: $C(y) = Q_C(y)R_C$, $R(x)^T = Q_R(x)R_R$
2. Compute: $A = R_C D R_R^T$
3. Matrix SVD: $A = U_A S_A V_A^T$
4. Compute: $U_f = Q_C(y)U_A$, $V_f = Q_R(x)V_A$, and $S_f = S_A$

FIG. 6.1. *Pseudocode for computing the SVD of a chebfun2. This algorithm uses Householder triangularization of a quasi-matrix* [40]. *This algorithm is a continuous version of a fast SVD for low rank matrices* [3].

in [40]. An excellent way to construct the SVD of a smooth bivariate function is usually by first constructing a chebfun2 and then orthogonalizing the column and row slices.

Example: Gaussian bumps. As an example, we compute the singular values of a function composed by adding together 300 Gaussian bumps at arbitrary locations,

$$f(x,y) = \sum_{j=0}^{300} e^{-\gamma\left((x-s_j)^2+(y-t_j)^2\right)}, \qquad (s_j,t_j) \in [-1,1]^2, \tag{6.1}$$

where $\gamma = 10, 100, 1000$. A Gaussian bump is a rank 1 function, and hence, mathematically, f is of rank 300 (almost surely). However, it can be approximated to machine precision by a function of much lower rank. Figure 6.2 displays a contour plot for the case $\gamma = 1000$ and shows that the singular values of f decay supergeometrically. The figure is computed with the following code:

```
s = RandStream('mt19937ar','Seed',1);        % for reproducibility
gamma = 1000; f = chebfun2();
for n = 1:300
    x0 = 2*rand(s)-1; y0 = 2*rand(s)-1;
    df = chebfun2(@(x,y)exp(-((x-x0).^2+(y-y0).^2)/gamma));
    f = f + df;
end
subplot(1,2,1), contour(f)
subplot(1,2,2), semilogy(svd(f))
```

Example: Near-optimality. The SVD allows us to explore the near-optimality of GE with complete pivoting for constructing low rank approximations. In Figure 6.3 we take the 2D Runge function given by

$$f(x,y) = \frac{1}{1+\gamma\left(x^2+y^2\right)^2}, \qquad \gamma = 1, 10, 100,$$

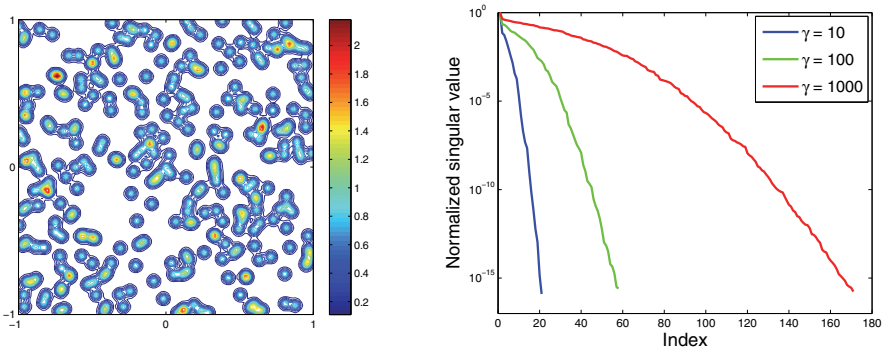

FIG. 6.2. *Left: Three hundred arbitrarily centered Gaussian bump functions added together, as in* (6.1)*, with* $\gamma = 1000$*. Right: Supergeometric decay of the normalized singular values (the first singular value is scaled to be* 1*) for* $\gamma = 10, 100, 1000$*. Mathematically,* (6.1) *is almost surely of rank* 300*, but it can be approximated to machine precision in these realizations by functions of rank* 21*,* 59*, and* 176*, respectively. The supergeometric decay of the singular values can be explained by theory related to the fast Gauss transform* [17].

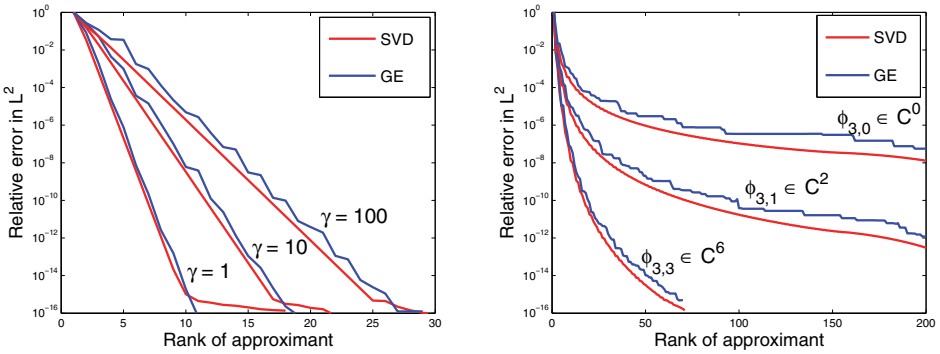

FIG. 6.3. *A comparison shows the near-optimality of GE approximations to smooth bivariate functions. Left: 2D Runge functions. Right: Wendland's compactly supported radial basis functions.*

which is analytic, and Wendland's compactly supported radial basis functions [45],

$$\phi_{3,k}(|x-y|) = \phi_{3,k}(r) = \begin{cases} (1-r)_+^8 \left(32r^3 + 25r^2 + 8r + 1\right), & k = 3, \\ (1-r)_+^4 \left(4r + 1\right), & k = 1, \\ (1-r)_+^2, & k = 0, \end{cases}$$

which have $2k$ continuous derivatives and thus belong to $\mathcal{C}^{2k}$. Figure 6.3 shows the L^2 errors of the optimal rank approximations computed via the expensive SVD, and the near-optimal rank approximations constructed via the Chebfun2 constructor (GE with complete pivoting). We observe that GE with complete pivoting usually computes near-optimal low rank approximations.

7. Rootfinding. There are two operations on chebfun2 objects that could be referred to as *rootfinding*: finding the zero level curves of $f(x, y)$ and solving for the common roots of $f(x, y)$ and $g(x, y)$. These operations are mathematically distinct

with the former, generically, having solutions along curves that do not end abruptly[5] [29] and the latter, generically, having a finite number of isolated solutions [24].

If `f` is a chebfun2, then `roots(f)` finds the zero level curves of `f` by using the MATLAB command `contourc`. We use this command to find points in $[-1,1]^2$ that interpolant the zero curves of `f` and to classify them into groups that belong to nonintersecting zero curves. From here, we construct a complex valued chebfun for each group that interpolates the points encoded as complex numbers. Accordingly, if `f` is a chebfun2, `roots(f)` returns a quasi-matrix, where each column is a complex valued chebfun interpolating a zero contour of `f` in $[-1,1]^2$.

If `F` is a chebfun2v, then `roots(F)` returns the isolated solutions of the system

$$F(x,y) = \begin{pmatrix} f_1(x,y) \\ f_2(x,y) \end{pmatrix} = 0, \qquad (x,y) \in [-1,1]^2. \tag{7.1}$$

To achieve this, we recursively subdivide $[-1,1]^2$ and approximate $f_1(x,y)$ and $f_2(x,y)$ by low degree piecewise polynomials. On each subdomain we find the solutions of the low degree polynomials by a hidden variable resultant method based on Bézout resultants. Usually, resultant methods are prone to numerical issues, but these have been overcome by various techniques such as representing polynomials in the Chebyshev basis, Bézoutian regularization, and local refinement. The details of this algorithm are rather involved and are discussed in a separate publication [28]. In many practical cases, this algorithm allows us to reliably compute the isolated solutions of (7.1) when f_1 and f_2 are of large numerical degree (≥ 1000).

If f_1 and f_2 are of very large numerical degree, then the resultant method can be computationally expensive. Instead, we approximate the zero contours of f_1 and f_2 using `contourc` and then find their intersections, which are used as initial guesses in a Newton iteration. We do not employ this approach when the degrees are small because it does not offer the same reliability as the resultant method in [28].

[5] A formal discussion is beyond the scope of this paper.

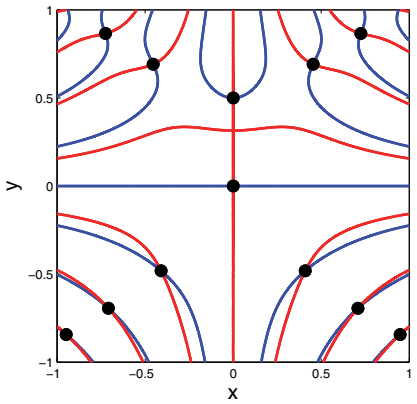

FIG. 7.1. *The critical points of* $f(x,y) = \left(x^2 - y^3 + 1/8\right)\sin(10xy)$ *(black dots) lie at the intersections of the zero contours of* f_x *(blue (in the electronic version) curves), and* f_y *(red (in the electronic version) curves). If* `f` *is a chebfun2, then* `roots(f)` *returns complex valued chebfuns interpolating the zero contours of* `f`*, and* `roots(gradient(f))` *returns the critical points of* `f`*.*

As a simple example of both kinds of rootfinding, Figure 7.1 plots the critical points and the zero contours of f_x and f_y for $f(x,y) = \left(x^2 - y^3 + 1/8\right)\sin(10xy)$ by using the following code:

```
f = chebfun2(@(x,y) (x.^2-y.^3+1/8).*sin(10*x.*y));
r = roots(gradient(f));                  % critical points
plot(roots(diff(f,1,2)),'b'), hold on    % plot zero contours of f_x
plot(roots(diff(f)),'r')                 % plot zero contours of f_y
plot(r(:,1),r(:,2),'k.','MarkerSize',30) % plot extrema
```

Rootfinding gives us another algorithm for the commands `minandmax2`, and `max2`, `min2` in section 5. For instance, for `max2` we could find all the critical points of a function on $[-1,1]^2$, via `roots(gradient(f))`, and then select a critical point that achieves the maximum value. Of course, the maximum may occur on the boundary of $[-1,1]^2$ and we also need to solve four one-dimensional maximization problems via the `max` command in Chebfun. Currently, this is a strong contender as the algorithm of choice for global optimization in Chebfun2, but more investigations are required.

8. Future work. We are entering into an exciting stage of development where computing with scalar and vector valued multivariate functions to machine precision is practical. Chebfun2 has shown that such computations can be performed in MATLAB in two dimensions by building on Chebfun technology. Challenges ahead include the extension of Chebfun2 to functions with singularities and higher dimensions. Another challenge is to extend Chebfun2 to functions defined on nonrectangular domains. One possibility is to perform a change of variables to map a nonrectangular domain onto a rectangle and do subsequent operations on the mapped function defined on the rectangle. The mapping can be conformal or nonconformal as long as the determinant of the Jacobian never numerically vanishes. Initial investigations have been made, but substantial issues remain to be addressed such as accurately representing a domain and providing a succinct user interface, as well as many algorithmic issues. Even on rectangular domains, a full understanding of the approximation properties of low rank approximations computed with the Chebfun2 constructor is not yet available.

Acknowledgments. We are grateful to Nick Hale for his investigation into 2D Chebfun-like computing, which provided an initial benchmark. We have had fruitful discussions with the Chebfun team: Anthony Austin, Ásgeir Birkisson, Toby Driscoll, Nick Hale, Mohsin Javed, Mark Richardson, and Kuan Xu. We thank Mario Bebendorf for clarifying the ACA literature and John Boyd for his suggestions for rootfinding. Discussions with Paul Constantine, Daan Huybrechs, Yuji Nakatsukasa, Sheehan Olver, Bor Plestenjak, Dmitry Savostyanov, and Jared Tanner have been valuable.

REFERENCES

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

[2] M. Bebendorf, *Approximation of boundary element matrices*, Numer. Math., 86 (2000), pp. 565–589.

[3] M. Bebendorf, *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*, Springer-Verlag, Berlin, 2008.

[4] M. Bebendorf, *Adaptive cross approximation of multivariate functions*, Constr. Approx., 34 (2011), pp. 149–179.

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

[6] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel, *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*, SIAM, Philadelphia, 1987.
[7] J. P. Boyd, *Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding*, SIAM J. Numer. Anal., 40 (2003), pp. 1666–1682.
[8] O. A. Carvajal, F. W. Chapman, and K. O. Geddes, *Hybrid symbolic-numeric integration in multiple dimensions via tensor-product series*, in Proceedings of ISSAC'05, M. Kauers, ed., ACM, New York, 2005, pp. 84–91.
[9] F. W. Chapman, *Generalized Orthogonal Series for Natural Tensor Product Interpolation*, Ph.D. dissertation, School of Computer Science, University of Waterloo, Waterloo, Ontario, 2003.
[10] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, *On the compression of low rank matrices*, SIAM J. Sci. Comput., 26 (2005), pp. 1389–1404.
[11] C. W. Clenshaw and A. R. Curtis, *A method for numerical integration on an automatic computer*, Numer. Math., 2 (1960), pp. 197–205.
[12] T. F. Coleman and Y. Li, *An interior trust region approach for nonlinear minimization subject to bounds*, SIAM J. Optim., 6 (1996), pp. 418–445.
[13] P. Drineas, R. Kannan, and M. W. Mahoney, *Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition*, SIAM J. Comput., 36 (2006), pp. 184–206.
[14] R. Franke, *A Critical Comparison of Some Methods for Interpolation of Scattered Data*, Technical Report, Naval Postgraduate School, Monterey, CA, 1979.
[15] S. A. Goreinov, I. V. Oseledets, D. V. Savostyanov, E. E. Tyrtyshnikov, and N. L. Zamarashkin, *How to find a good submatrix*, in Matrix Methods: Theory, Algorithms and Applications, V. Olshevsky and E. Tyrtyshnikov, eds., World Scientific, Hackensack, NJ, 2010, pp. 247–256.
[16] S. A. Goreinov, E. E. Tyrtyshnikov, and N. L. Zamarashkin, *A theory of pseudoskeleton approximations*, Linear Algebra Appl., 261 (1997), pp. 1–21.
[17] L. Greengard and J. Strain, *The fast Gauss transform*, SIAM J. Sci. Statist. Comput., 12 (1991), pp. 79–94.
[18] M. Griebel and H. Harbrecht, *Approximation of Two-variable Functions: Singular Value Decomposition Versus Regular Sparse Grids*, preprint 488, Berichtsreihe des SFB 611, Universität Bonn, Bonn, Germany, 2010.
[19] W. Hackbusch, *Hierarchische Matrizen: Algorithmen und Analysis*, Springer, Dordrecht, 2009.
[20] W. Hackbusch, *Tensor Spaces and Numerical Tensor Calculus*, Springer, Heidelberg, 2012.
[21] N. Halko, P.-G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.
[22] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.
[23] Y. P. Hong and C.-T. Pan, *Rank-revealing QR factorizations and the singular value decomposition*, Math. Comp., 58 (1992), pp. 213–232.
[24] F. Kirwan, *Complex Algebraic Curves*, Cambridge University Press, Cambridge, UK, 1992.
[25] G. Little and J. B. Reade, *Eigenvalues of analytic kernels*, SIAM J. Math. Anal., 15 (1984), pp. 133–136.
[26] J. C. Mason and D. C. Handscomb, *Chebyshev Polynomials*, Taylor & Francis, Boca Raton, FL, 2002.
[27] L. Miranian and M. Gu, *Strong rank revealing LU factorizations*, Linear Algebra Appl., 367 (2003), pp. 1–16.
[28] Y. Nakatsukasa, V. Noferini, and A. Townsend, *Computing the common zeros of two bivariate functions via Bezout resultants*, submitted, 2013.
[29] A. Ostrowski, *Über den ersten und vierten Gaußschen Beweis des Fundamental-Satzes der Algebra*, Carl Friedrich Gauss Werke Band X Abt., 2 (1920), pp. 1–18.
[30] R. Pachón, R. B. Platte, and L. N. Trefethen, *Piecewise-smooth chebfuns*, IMA J. Numer. Anal., 30 (2010), pp. 898–916.
[31] C.-T. Pan, *On the existence and computation of rank-revealing LU factorizations*, Linear Algebra Appl., 316 (2000), pp. 199–222.
[32] H. E. Salzer, *Lagrangian interpolation at the Chebyshev points $x_{n,v} = \cos(v\pi/n), v = o(1)n$; some unnoted advantages*, Comput. J., 15 (1972), pp. 156–159.
[33] E. Schmidt, *Zur Theorie der linearen und nichtlinearen Integralgleichungen*, Math. Ann., 63 (1907), pp. 433–476.

[34] G. W. Stewart, *Updating a rank-revealing ULV decomposition*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 494–499.
[35] G. W. Stewart, *Matrix Algorithms, Volume 1: Basic Decompositions*, SIAM, Philadelphia, 1998.
[36] A. Townsend, *Chebfun2 software*, http://www.maths.ox.ac.uk/chebfun, 2013.
[37] A. Townsend and L. N. Trefethen, *Gaussian elimination as an iterative algorithm*, SIAM News, 46, 2013.
[38] L. N. Trefethen, *A hundred-dollar, hundred-digit challenge*, SIAM News, 35, 2002.
[39] L. N. Trefethen, *Computing numerically with functions instead of numbers*, Math. Comput. Sci., 1 (2007), pp. 9–19.
[40] L. N. Trefethen, *Householder triangularization of a quasimatrix*, IMA J. Numer. Anal., 30 (2009), pp. 887–897.
[41] L. N. Trefethen et al., *Chebfun Version 4.2, The Chebfun Development Team*, http://www.maths.ox.ac.uk/chebfun/, 2011.
[42] L. N. Trefethen, *Approximation Theory and Approximation Practice*, SIAM, Philadelphia, 2013.
[43] E. E. Tyrtyshnikov, *Incomplete cross approximation in the mosaic-skeleton method*, Computing, 64 (2000), pp. 367–380.
[44] J. Waldvogel, *Fast construction of the Fejér and Clenshaw–Curtis quadrature rules*, BIT, 46 (2006), pp. 195–202.
[45] H. Wendland, *Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree*, Adv. Comput. Math., 4 (1995), pp. 389–396.
[46] H. Weyl, *Das asymptotische Verteilungsgesetz der Eigenwerte linearer partieller Differentialgleichungen (mit einer Anwendung auf die Theorie der Hohlraumstrahlung)*, Math. Ann., 71 (1912), pp. 441–479.