CHEBFUN IN THREE DIMENSIONS

BEHNAM HASHEMI AND LLOYD N. TREFETHEN†

Abstract. We present an algorithm for numerical computations involving trivariate functions in a 3D rectangular parallelepiped in the context of Chebfun. Our scheme is based on low-rank representation through multivariate adaptive cross approximation (MACA). The component 1D functions are represented by finite Chebyshev expansions, or trigonometric expansions in the periodic case. Numerical experiments show the power and convenience of Chebfun3 for problems such as function manipulation, differentiation, optimization, and integration, as well as for exploration of fundamental issues of multivariate approximation and low-rank compression.

Key words. Chebfun, slice-Tucker decomposition, low-rank approximation, multivariate adaptive cross approximation (MACA)

AMS subject classifications. 15A69, 41A10

1. Introduction. Chebfun is a MATLAB toolbox for numerical computing with functions to approximately 15-digit precision [12]. It began in 2003 [2] with univariate functions, and the 2D extension Chebfun2, due to Alex Townsend, was released in 2013 [30, 31, 32, 33]. Given a smooth bivariate function $f(x, y)$ defined on a rectangle $[a, b] \times [c, d]$, Chebfun2 attempts to represent f in a low-rank format computed by adaptive cross approximation (ACA) [3, 4], i.e., iterative application of an approximation of Gaussian elimination with complete pivoting. This means that f is represented by a sum of products of univariate functions,

$$f(x, y) \approx \sum_{k=1}^{r} \alpha_k u_k(x) v_k(y). \tag{1.1}$$

Our aim has been to extend Chebfun and Chebfun2 to trivariate functions defined on a parallelepiped $[a, b] \times [c, d] \times [e, g]$. For simplicity of exposition, we will take the domain to be the unit cube $[-1, 1]^3$. In developing an effective representation, we have explored different techniques. The reference point from which they all spring is a full tensor product, in which a function is represented by a triple sum of Chebyshev polynomials,

$$f(x, y, z) \approx \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \sum_{k=0}^{p-1} c_{ijk} T_i(x) T_j(y) T_k(z), \tag{1.2}$$

where $C = \{c_{ijk}\}$ is an $m \times n \times p$ tensor of coefficients. Mathematically, one choice of coefficients c_{ijk} would be that corresponding to truncation of an infinite 3-variable Chebyshev series of f (weighted L^2 projection), and another would correspond to 3D *Chebyshev interpolation,* i.e., interpolation of f in a 3D tensor product grid obtained from Chebyshev points in the x, y, and z directions. The two are related by aliasing, and the difference in accuracy will normally be small [37, chap. 5]. See also [22]. For ready comparison of the tensor product representation with others, Chebfun3

†Mathematical Institute, University of Oxford, Oxford OX2 6GG, UK, hashemi@maths.ox.ac.uk and trefethen@maths.ox.ac.uk. Supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/20072013)/ERC grant agreement no. 291068. The views expressed in this article 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.

includes a class `chebfun3t` implementing this tensor product Chebyshev interpolation idea. Chebfun3t computes the necessary coefficients using the 3D discrete cosine transform (DCT), choosing m, n, p automatically to achieve 15-digit precision, and requires $\mathcal{O}\big(mnp\log(mnp)\big)\big)$ operations. See Section 5 for further discussion and comparisons.

The problem with full tensor representations is the product mnp: they require a great deal of work and storage when m, n, p are not small. For some functions, this is unavoidable, but for a good proportion of functions that arise in practice, one can do better by low-rank representations [38]. Like Chebfun2, Chebfun3 makes use of this technique. A difference from the 2D case is that in 3D, there are many different possible low-rank representations to consider.

The general idea of low-rank approximation algorithms is as follows. Beginning with a function f, we approximate it by a function with a simple structure. Subtracting off this first approximation, we then approximate the difference by another simple function. We proceed in such a manner until the error is 15 or 16 digits below the scale of the original function.

Within this general framework, many methods are possible. Our choice for Chebfun3 is a variant of multivariate adaptive cross approximation *(MACA)* [4] that we call *slice-Tucker decomposition*. Here, the simple functions used in the first stage of constructing a low rank approximation of f are products of univariate and bivariate functions. Once this first stage is complete, a second stage is executed in which the bivariate functions are themselves approximated, as in Chebfun2, by sums of products of univariate functions. The details of our method are presented in Section 2, and comments on related algorithms and mathematics are given in Section 6.

Broadly speaking, the vision of the Chebfun enterprise is the development of mathematical ideas and computational algorithms that are continuous analogues of familiar discrete ideas and algorithms of scientific computing. Here, our computations with trivariate functions based on low-rank representations build on the well-established literature of low-rank representations for order-3 tensors. Specifically, we make use of a tensor F that approximates f on a tensor product grid of points,

$$F(i, j, k) \approx f(x_i, y_j, z_k). \tag{1.3}$$

To conclude this section we review a few concepts from tensor analysis [14] and mention some of the software available for tensor computations in MATLAB.

A discrete order-3 tensor $F \in \mathbb{C}^{m\times n\times p}$ is a real or complex 3-dimensional array $F(1{:}m, 1{:}n, 1{:}p)$. In general, throughout the paper, tensors are of size $m\times n\times p$, though occasionally we consider $m \times m \times m$. A *fiber* of F is a vector obtained by fixing all but one of the indices of F; the directions x, y, and z are conventionally called *modes* 1, 2, and 3. Mode-1 fibers are *columns*, mode-2 fibers are *rows,* and mode-3 fibers are *tubes.* A *slice* is a matrix obtained by fixing all but two of the indices of F. All these notions such as fibers, columns, rows, and tubes, as well as further objects of tensor analysis, have continuous analogues for functions, and Chebfun3 enables users to work with them.

DEFINITION 1.1. *Let $a \in \mathbb{C}^m$, $b \in \mathbb{C}^n$, and $c \in \mathbb{C}^p$ be vectors and let $A \in \mathbb{C}^{m\times n}$ be a matrix. The* outer product *of a, b, c is the tensor $F := a \circ b \circ c \in \mathbb{C}^{m\times n\times p}$ defined by $F(i, j, k) = a(i)b(j)c(k)$, and the* outer product *of A and c is the tensor $G := A \circ c \in \mathbb{C}^{m\times n\times p}$ defined by $G(i, j, k) = A(i, j)c(k)$.*

DEFINITION 1.2. *[14] Let $T \in \mathbb{C}^{n_1\times n_2\times n_3}$ be a tensor. For $k = 1, 2, 3$, the* mode-k unfolding *of T, denoted by $T_{(k)}$, is the matrix of size $n_k \times (n_1n_2n_3/n_k)$ whose*

columns are the mode-k fibers of T. Moreover, if A is matrix of size $m_k \times n_k$, then the mode-k contraction *of T with A, denoted by $S := T \times_k A$, is the tensor such that $S_{(k)} := A\, T_{(k)}$.*

Given a discrete tensor $F \in \mathbb{C}^{m\times n\times p}$, a *Tucker representation* of F is a representation of the form

$$F = T \times_1 A \times_2 B \times_3 C,$$

where $T \in \mathbb{C}^{r_1\times r_2\times r_3}$ is the *core tensor,* $A \in \mathbb{C}^{m\times r_1}$, $B \in \mathbb{C}^{n\times r_2}$, and $C \in \mathbb{C}^{p\times r_3}$ are *factor matrices,* and (r_1, r_2, r_3), or occasionally $r := \max\{r_1, r_2, r_3\}$, is the *Tucker rank* of (this representation of) F. Instead of mnp entries, Tucker format allows the representation of F by $mr_1 + nr_2 + pr_3 + r_1r_2r_3$ entries. See [19] for a review of tensor decompositions and [17] for a more recent review of low rank approximation of tensors.

The Tensor Toolbox by Bader and Kolda [1] contains, among other things, a class of tensors in Tucker format called `ttensor`. The latest version of Tensorlab by Vervliet, et al. [39] has several features including different tensor decompositions. The `htucker` toolbox by Kressner and Tobler allows the construction and manipulation of tensors in the Hierarchical Tucker format [21]. TT-Toolbox by Oseledets [25] allows fast multidimensional array operations in the tensor train format.

The remainder of this paper is organized as follows. In Section 2 we present our algorithm for constructing representations of 3D functions in Chebfun3. Section 3 describes algorithms for numerical computing with chebfun3 objects. Section 4 presents examples of computations in Chebfun3, and Section 5 shows how Chebfun3 and Chebfun3t can be used as a laboratory for exploration of issues of low-rank approximation. Section 6 discusses some related tensor factorizations, and Section 7 finishes with some concluding remarks.

2. The Chebfun3 constructor.

2.1. Overview. Our low-rank approximation technique consists of three phases, which are summarized in Table 2.1. The algorithm aims to take advantage both of situations where dependence on one variable is simpler than dependence on the other two, and of situations where the rank is smaller than the complexity of a tensor product representation (1.2) would lead one to expect. We first outline these phases at a high level, then give details in the subsections.

TABLE 2.1
Outline of the Chebfun3 constructor. BTD stands for Block Term Decomposition.

Phase	Action	Key point
1	slice decomposition and discrete BTD	computations involving full tensors of (hopefully) small size
2	resolve univariate skeletons	sizes of skeletons may increase
3	convert to Tucker form	final non-adaptive compression phase

Phase 1 begins by separating one of the variables from the other two. We can decompose f into a sum of products of bivariate and univariate functions

$$f(x, y, z) \approx \sum_{k=1}^{r_z} g_k(x, y)\, h_k(z), \tag{2.1}$$

where r_z is called the (xy, z) *numerical separation rank* of f, or the z*-rank* for short. Alternatively, we can represent f as

$$f(x,y,z) \approx \sum_{k=1}^{r_y} g_k(z,x)\, h_k(y) \approx \sum_{k=1}^{r_x} g_k(y,z)\, h_k(x),$$

where r_y and r_x are the y*-rank* and x*-rank*. Each bivariate function g_k corresponds to a slice, and each univariate function h_k to a fiber. We call any of these formulas a *slice decomposition* of f. See Figure 2.1. The numbers r_x, r_y, and r_z may differ greatly.

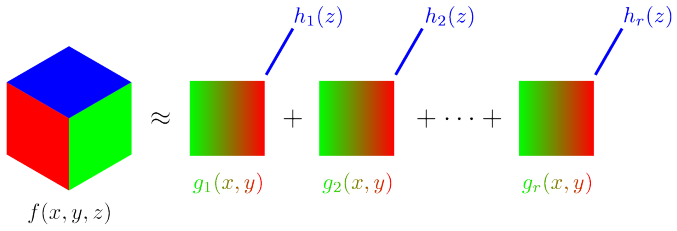

FIG. 2.1. *Phase* 1 *of the Chebfun3 constructor: Slice decomposition, eq.* (2.1). r *stands for* r_z.

Which of the three decompositions should be used? The first step of construction attempts to make a good choice, i.e., one that allows the approximation of f with low complexity. This is the problem known as *dimension clustering*. We use the following heuristic: sample f on a $10 \times 10 \times 10$ tensor product grid, use the SVD to compute the numerical ranks of the three modal unfoldings of the sample tensor with tolerance machine epsilon, and find the dimension that corresponds to the minimum rank.[1] In the case of a tie, we choose the variable whose resolution requires the bigger number of Chebyshev coefficients.[2] For simplicity, in what follows we assume that the dimension clustering step has estimated that the decomposition (2.1) is the best of the three. Thus Figure 2.1 and equation (2.1) can be kept in mind for the remaining discussion.

The main computations of Phase 1 now involve, first, constructing the bivariate functions $g_k(x,y)$ and tubes $h_k(z)$ of the slice decomposition. Then the functions $g_k(x,y)$ are reduced to low-rank 2D representations by a process modeled on Chebfun2, the goal being to identify what we call (following more or less familiar terminology) *skeleton columns* and *skeleton rows* of each of these slices, to be combined with the *skeleton tubes* $h_k(z)$ already determined. The final result of Phase 1 is a representation like what is called a block term decomposition (BTD) by De Lathauwer [10] (though he does not use the same algorithm). See Figure 2.2.

Phase 2 then resolves the skeleton columns and rows of each slice and the skeleton tubes identified in Phase 1.

[1] Recall that encountering distinct ranks for different variables would be impossible for a bivariate function $f(x,y)$: for any matrix, row rank equals column rank.

[2] A different approach to dimension clustering has been proposed by Bebendorf and Kuske [6]. For a 3D function, their scheme forms a 3×3 covariance matrix from a probability density function assigned to f, which is then inspected to find variables with smallest correlation. This requires the approximate computation of seven triple integrals, which they carry out by a quasi Monte-Carlo method with a Halton sequence. Our experiments with this technique show that in our 3D context, the SVD-based dimension clustering gets satisfactory results with considerably fewer function evaluations.

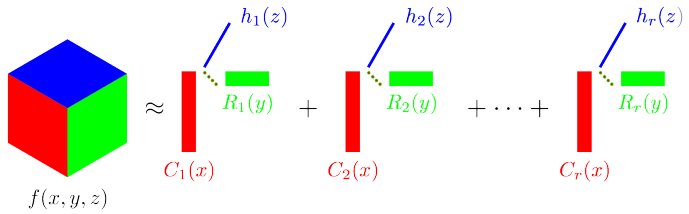

FIG. 2.2. *Phase 1, continued: Block-term decomposition. Phase 2 then resolves the univariate functions in the three directions.*

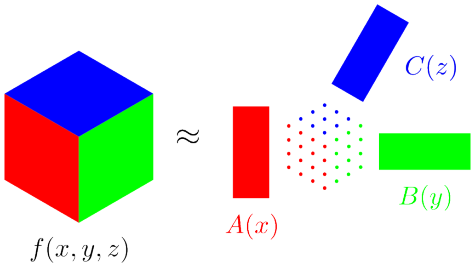

FIG. 2.3. *Phase 3: Compression to Tucker representation.*

Phase 3 compresses the block term representation of Phase 2 to get a Tucker decomposition of f, as depicted in Figure 2.3. In this final representation of f as a chebfun3, there is no longer any asymmetry in the treatment of the three variables x, y, and z.

Considering $f(x,y,z)$ as an order 3 infinite dimensional *continuous tensor* defined on the domain $[a,b]\times[c,d]\times[e,g]$, we get the following representation of f:

$$f(x,y,z) \approx T \times_1 A(x) \times_2 B(y) \times_3 C(z), \tag{2.2}$$

where T is a discrete tensor of size $r_x \times r_y \times r_z$, $A(x)$ is a quasimatrix[3] of size $\infty \times r_x$ defined over $[a,b]$, $B(y)$ is a quasimatrix of size $\infty \times r_y$ over $[c,d]$ and $C(z)$ is a quasimatrix of size $\infty \times r_z$ defined over $[e,g]$. The above formula involves modal contractions of the core tensor T and the quasimatrices $A(x)$, $B(y)$ and $C(z)$ and can be rewritten as follows

$$f(x,y,z) \approx \sum_{i=1}^{r_x}\sum_{j=1}^{r_y}\sum_{k=1}^{r_z} t_{i,j,k}\ a_i(x)\ b_j(y)\ c_k(z).$$

We call either (r_x, r_y, r_z) or $r := \max\{r_x, r_y, r_z\}$ the *numerical* or *approximate Tucker rank* of f (i.e., the Tucker rank of the approximation of f on the right-hand side of (2.2).)

Our algorithm has a number of similarities with the CROSS3D algorithm of [13, 27], and some significant differences. Both algorithms aim at computing a Tucker

[3] A *quasimatrix* is a 'matrix' whose 'columns' are functions rather than vectors, a term coined by Stewart [29].

decomposition and have complexity linear in the size of the underlying tensors. At the same time we are interested in a continuous analogue of the Tucker decomposition and use complete pivoting partly to take advantage of the nesting property of Chebyshev or Fourier points, while a greedy pivoting strategy is used in [27] together with maximum subvolume matrices. Another difference is that we do not aim at getting orthogonal factors.

2.2. Phase 1: Determination of the separation rank and skeletons. Algorithm 1 presents Phase 1 of the Chebfun3 constructor, which involves grids in each dimension potentially of sizes $17, 23, 33, 46, \ldots$ (one more than a power of $\sqrt{2}$) and z-ranks potentially as large as $6, 8, 11, 16, \ldots$ (approximately the same numbers divided by the rather arbitrary constant $2\sqrt{2}$). In words, we begin by sampling f on a $q \times q \times q$ tensor product Chebyshev grid with $q = 17$. We perform up to 6 steps of MACA to determine whether the discretely sampled f can be approximated to the specified tolerance by a function of z-rank at most 6. Specifically, in Step 4 of Algorithm 1 we find the value of the entry with maximum magnitude over the whole tensor together with the corresponding optimal indices (i, j, k), which are then used to form the 2D slices and 1D tubes. If MACA was not successful with $q = 17$, we increase the grid size to $q = 23$ and perform at most 8 steps of MACA to determine whether the z-rank is at most 8. The process continues on Chebyshev grids with $q = 33, 46, \ldots, 363$, with checks for z-rank of at most $11, 16, \ldots, 128$. Carefully tracking these numbers leads to the conclusion that, if it is successful, Phase 1 approximates a tensor of size

$$\lfloor \sqrt{2}^{t+7} + 1 \rfloor \times \lfloor \sqrt{2}^{t+7} + 1 \rfloor \times \lfloor \sqrt{2}^{t+7} + 1 \rfloor \tag{2.3}$$

by a tensor of z-rank at most $\lfloor \sqrt{2}^{t+4} + \sqrt{2}^{-3} \rfloor$ with $t \geq 1$. To put it another way, if f can be relatively approximated to the tolerance by a function of z-rank k, then k steps of MACA are needed and Phase 1 samples f on a tensor grid of size (2.3), where $t = \lceil -4.5 + 2 \log_2 k \rceil$. Note that

$$q = \lfloor \sqrt{2}^{t+7} + 1 \rfloor \leq \sqrt{2}^{\lceil -4.5 + 2 \log_2 k \rceil + 7} + 1 \leq 4k + 1 = \mathcal{O}(k).$$

If we simplify parameters, each iteration involves $\mathcal{O}(k^3)$ operations in the MACA level and $\mathcal{O}(k^3)$ operations in the 2D ACA level. This means that overall, Phase 1 needs $\mathcal{O}(k^4)$ arithmetic operations.

We have oversimplified slightly. In fact, Phase 1 permits the grid in the x-y plane to rise to a higher grid parameter $\hat{q} > q$ than in the z direction. See the algorithm as displayed. The first two conditional statements in Step 5 basically check the rank $\hat{r}_1$ of the first slice. This is used to examine the success of internal 2D ACA steps. The third conditional statement, on the other hand, checks the success of the external MACA.

At the end of Phase 1, we have an approximation of the form

$$f(x, y, z) \approx \sum_{l=1}^{r_z} \sum_{s=1}^{\hat{r}_l} c_{l,s}(x)\ r_{l,s}(y)\ h_l(z), \tag{2.4}$$

as depicted in Figure 2.2. Similar representations appear in the literature; see e.g. [4, 5, 27]. De Lathauwer [10, Fig. 2.1] uses the name *rank* $(r, r, 1)$ *decomposition* in the more general context of block term decompositions. See also [24, eq. (13)] and [9, p. 65]. We note in passing that the *tensor train* format [26], though different in the way

Algorithm 1 Chebfun3 constructor: Phase 1.

1: Apply dimension clustering to a $10 \times 10 \times 10$ sample of f, as described in §§2, to choose which of x, y, and z is to be separated. Hereafter we assume this is z.
2: Set $\hat{q} := q := 17$ and a tolerance *tol*, e.g. 10^{-15}.
3: Sample f at $\hat{q} \times \hat{q} \times q$ tensor product Chebyshev points to get a discrete tensor F.
4: Set $R_1 := F$, and then for $l = 1, 2, 3, \ldots$.
- Find $\varepsilon = \max_{i,j,k} |R_l(:,:,:)|$ and associated indices i, j, k. Now consider the 2D slice $S_l = R_l(:,:,k)$ and the 1D tube $R_l(i,j,:)$.
- Identify skeleton columns and rows via 2D ACA for this 2D slice. Write $S_l = C_l D_l E_l^T$, with C_l, D_l, E_l of sizes $\hat{q} \times \hat{r}_l$, $\hat{r}_l \times \hat{r}_l$, and $\hat{q} \times \hat{r}_l$.
- Set $R_{l+1} := R_l - S_l \circ R_l(i,j,:)/R_l(i,j,k)$. See e.g. eq. (26) of [4].

5: Step 4 may terminate in various ways.
- If $\varepsilon < tol$ and $\hat{r}_1 < \lfloor q/2 \rfloor$, Phase 1 is finished; go to Phase 2.
- If $\varepsilon < tol$ and $\hat{r}_1 \geq \lfloor q/2 \rfloor$, increase $\hat{q}$ to $\lceil \sqrt{2}\hat{q} \rceil$ and return to Step 3.
- If $\varepsilon \geq tol$ and $l = l_{\max} = \lceil 2^{-3/2} q \rceil$, increase q and $\hat{q}$ by $\sqrt{2}$ and return to Step 3, unless q would exceed $q_{\max} = 363$, in which case the construction exits with a failure.

it is computed, employs a similar representation for approximating order 3 discrete tensors. The same is true of *Tucker2* representation of order-3 discrete tensors.

If f is triply periodic and the flag `'trig'` (or `'periodic'`) is invoked, then instead of Chebyshev points, tensor product grids of equally spaced points are used and a different refinement sequence is employed to exploit the grid nesting.

2.3. Phase 2: Resolving skeleton columns, rows, and tubes. At the end of Phase 1 we have the block term decomposition (2.4). This means that we have identified not only the locations of pivot values and skeleton slices and tubes, but also skeleton columns and rows of each slice, as depicted in Figure 2.2. Phase 2 aims to fully resolve these slices and tubes. For example, $f(x,y,z) = (x+y)\sin(200z)$ is a function of z-rank 1. Consequently, Phase 1 terminates after sampling on a $17 \times 17 \times 17$ tensor product Chebyshev grid even though a Chebyshev interpolant of degree 261 is required to resolve the oscillations in the variable z. In Phase 2, we sample f only on a subset of a finer grid consisting of k columns, rows and tubes and perform MACA on those skeletons.

Suppose that after Phase 1 is finished, there are r skeleton columns and rows of size q in each slice and each skeleton tube is also of size q. From Phase 2 on, we work only with the skeletons, not forming explicitly any $q \times q$ slices. Instead we follow a Chebyshev grid refinement procedure modeled on Chebfun to update the skeleton columns and rows based on 3D MACA and 2D ACA. At the end of Phase 2, the skeleton columns, rows, and tubes are of dimensions m, n, p, respectively, with $m, n, p \geq q$.

Each refinement in Phase 2 doubles the grid size, and the number of samples needed to compute a degree $n-1$ interpolant is $n \leq 2^{\lceil \log_2 n \rceil}$. Therefore, if the pivot tubes require degree $p-1$ Chebyshev interpolants in z and the pivot columns and rows require degree $m-1$ and $n-1$ Chebyshev interpolants in x and y, respectively, then this stage samples f at at most

$$k\big(2^{\lceil \log_2 p \rceil} + k(2^{\lceil \log_2 m \rceil} + 2^{\lceil \log_2 n \rceil})\big) \leq 2kp + 2k^2 m + 2k^2 n$$

points. Another way to see this is that Phase 2 starts by resampling f on the points

corresponding to skeletons in the block term decomposition. There are k skeleton tubes, each a $p \times 1$ vector, and k skeleton columns and rows, matrices of size $m \times k$ and $n \times k$, respectively.

We then perform k steps of MACA on the specified columns, rows and tubes, and at the end of each MACA step we need to apply a bivariate ACA to the skeleton columns and rows corresponding to a single slice. The overall cost of MACA is $\mathcal{O}(mk^4)$ for the columns, $\mathcal{O}(nk^4)$ for rows, and $\mathcal{O}(pk^2)$ for the tubes.

On the other hand, in order to fully update the skeleton columns of size $m \times k$ corresponding to each slice, a single 2D ACA is used, involving $\mathcal{O}(mk^2)$ operations. Similarly, $\mathcal{O}(nk^2)$ operations are needed to fully update the skeleton rows corresponding to each slice. Since we have k skeleton columns and k skeleton rows, the total cost of all bivariate ACA steps is $\mathcal{O}\big((m+n)k^3\big)$. In total, Phase 2 requires

$$\mathcal{O}\big((m+n)k^4 + (m+n)k^3 + pk^2\big)$$

arithmetic operations.

Note that Phase 1 also needed $\mathcal{O}(k^3)$ evaluations of f. So, in the simplified case where $m = n = p$ and f has Tucker rank r, the total number of evaluations of f that our algorithm needs is $\mathcal{O}(mr^2 + r^3)$. Phase 3 does not involve any evaluations of f.

Note that one of the three factor matrices of the Tucker format can now be formed as a $p \times k$ matrix having skeleton tubes as its columns. The other two factor matrices and the core tensor will be computed in Phase 3.

2.4. Phase 3: From block term decomposition to Tucker format. Let us consider the slice decomposition of the first part of Phase 1. This contains two types of dependency: between variables x and y in each slice $g_k(x, y)$, and between different slices forming the whole decomposition. Phase 1 has already removed the first type of dependency by block term decomposition. Phase 3 is designed to remove the second, which is still present in the more compact representation of the block term decomposition (2.4). See Steps 7–9 of Algorithm 2. We have reserved Step 6 for the whole Phase 2.

Algorithm 2 Phase 3: BTD2Tucker compression.

7: Put the skeleton columns in a matrix $U := [\ C_1 \mid C_2 \mid \cdots \mid C_{r_z}\ ]$ of dimension $m \times \check{r}$ and the skeleton rows in a matrix $V := [\ E_1 \mid E_2 \mid \cdots \mid E_{r_z}\ ]$ of dimension $n \times \check{r}$. Here, $\check{r} := \sum_{l=1}^{r_z} \hat{r}_l$.

8: Apply two-dimensional ACA to U and V to get their low rank decompositions, i.e., $C_U D_U R_U^T = U$ and $C_V D_V R_V^T = V$.

9: Form the core tensor using the matrices D_U, R_U, D_V and R_V and pivot values from the 2D ACA and MACA steps according to size of the low rank representation of each slice. C_U and C_V are factor matrices of the Tucker representation.

10: Chop the tails and run a "sampleTest", to check the approximation accuracy. If passed, then construct the chebfun3 object, otherwise if $m, n, p \le m_{\max}$, refine them and go to Step 3.

Step 7 involves $\mathcal{O}\big((m+n)k^2+k^4\big)$ operations, assuming that it applies 2D ACA to two matrices of size $m \times k^2$ and $n \times k^2$ each having rank k. The two factor matrices C_U and C_V computed in Step 8 are $m \times k$ and $n \times k$, respectively. Therefore, Step 10 needs $\mathcal{O}\big(k(m \log m + n \log n + p \log p)\big)$ operations for k FFTs. Thus we have established the following result.

PROPOSITION 2.1. *The Chebfun3 constructor requires*

$$\mathcal{O}\big((m+n)r^4 + pr^2 + (m\log m + n\log n + p\log p)r\big)$$

arithmetic operations and

$$\mathcal{O}\big((m+n)r^2 + pr\big)$$

function evaluations, where r is the numerical Tucker rank of f.

2.5. A toy example. Consider the function

$$f(x,y,z) = 3x^7z + yz + yz^2 + \log(2+y)z^3 - 2z^5, \tag{2.5}$$

which has the following three slice decompositions obtained by separating x, y, and z, respectively:

$$f(x,y,z) = 1(yz + yz^2 + \log(2+y)z^3 - 2z^5) + 3x^7z \tag{2.6}$$

$$= 1(3x^7z - 2z^5) + y(z+z^2) + \log(2+y)z^3 \tag{2.7}$$

$$= z(3x^7 + y) + z^2y + z^3\log(2+y) - 2z^5. \tag{2.8}$$

This means that if we start Phase 1 by separating in the x dimension, we get the representation (2.6), i.e. (modulo constant factors),

$$h_1(x) = 1, \quad g_1(y,z) = yz + yz^2 + \log(2+y)z^3 - 2z^5,$$
$$h_2(x) = x^7, \quad g_2(y,z) = 3z,$$

and the x-rank of f is 2. See Figure 2.1. Note that at the end of this slice decomposition step we have already formed a basis for the "x space" of f, but not necessarily for its "y space" and "z space". On the other hand, (2.7) indicates that the y-rank of f is 3, and (2.8) indicates that the z-rank of f is 4.

Going back to (2.6), we can form the bivariate separable representations

$$g_1(y,z) = 1(-2z^5) + y(z+z^2) + \log(2+y)z^3, \quad g_2(y,z) = 1(3z)$$

which then make up the block-term decomposition of f as in Figure 2.2. Then, Step 7 of the algorithm corresponds to forming the quasimatrices

$$U(y) = [\,1 \;\; y \;\; \log(2+y) \;\; 1\,], \quad V(z) = [\,-2z^5 \;\; z+z^2 \;\; z^3 \;\; 3z\,].$$

In contrast to $V(z)$, which has full rank 4, $U(y)$ has rank 3 because of the obvious linear dependence in its columns. To form a basis for the "y space" of f, we proceed by removing the last column of $U(y)$. This explains why Tucker decomposition is considered a *subspace representation* [18]. The factor quasimatrices of the Tucker decomposition of f are

$$A(x) = [\,1 \;\; x^7\,], \quad B(y) = [\,1 \;\; y \;\; \log(2+y)\,], \quad C(z) = [\,z^5 \;\; z+z^2 \;\; z^3 \;\; z\,].$$

We thus see that the trilinear rank of f is $(2,3,4)$: these three numbers are the x-rank, the y-rank, and the z-rank. The $2\times3\times4$ discrete core tensor T has the following mode-1 unfolding:

$$T_{(1)} = \left[\begin{array}{ccc|ccc|ccc|ccc} -2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \end{array}\right].$$

As our first illustration of Chebfun3 in action, here is the display that results if we construct a chebfun3 for (2.5) numerically.

```
>> f = chebfun3(@(x,y,z) 3*x.^7.*z+y.*z+y.*z.^2+log(2+y).*z.^3-2*z.^5)
f =
   chebfun3 object
   cols: Inf x 2 chebfun
   rows: Inf x 3 chebfun
  tubes: Inf x 4 chebfun
   core: 2 x 3 x 4
 length: 8, 25, 6
 domain: [-1, 1] x [-1, 1] x [-1, 1]
 vertical scale = 7
```

The lines labeled `cols, rows, tubes,` and `core` indicate that a $2 \times 3 \times 4$ representation has been found. The `length` parameters 8, 25, and 6 indicate that the maximum polynomial degrees needed to represent the univariate fibers in the x, y, and z directions are 7, 24, and 5. The degrees 7 and 5 result from the powers x^7 and z^5 in (2.5), and the degree 24 is what is needed to resolve $\log(2+y)$ to machine precision on the interval $[-1, 1]$, as one can see in Chebfun by executing `f = chebfun(@(y) log(2+y)); plotcoeffs(f)` or `explain('log(2+y)')`. The univariate functions constructed by Chebfun are available through `f.cols, f.rows,` and `f.tubes` and will span the spaces indicated above, though the precise choices of functions and scalings will be different.

As explained in the last section, the representation of a chebfun3 is symmetric with respect to x, y, and z, even though the construction process begins by separating one of these dimensions from the other two. In keeping with this symmetry, the display above gives no indication of whether it was x, y, or z that was separated in the constuction of this chebfun3. As it happens, the separated variable was x, matching the formulas given above—not that it would make much difference for such a simple function. A user can force the constructor to choose a particular separation dimension with a command like `f = chebfun3(...,'fiberDim',i)`, where `i` is 1, 2, or 3.

3. Numerical computation with chebfun3 objects. The purpose of Chebfun and Chebfun3 is not just to represent functions, but to compute with them. Once chebfun3 objects have been constructed, many subsequent operations can exploit the Tucker representation (2.2), reducing most of the work to one-dimensional operations available in 1D Chebfun. Chebfun3 does this for operations involving differentiation and integration, but not for addition, multiplication or evaluation of functions of a chebfun3. For use of Tucker format for addition of tensors, see for example [21].

Throughout the Chebfun3 code, speedups would be possible if various operations made use of MEX files from compiled programming languages. However, the Chebfun project has a policy of avoiding MEX files in the interests of portability and durability over time. Therefore, Chebfun3 is entirely written in MATLAB. Also following the Chebfun convention, everything runs if the user has only core MATLAB, without toolboxes, though a speedup is possible if the Optimization Toolbox is installed as described below under Minima and maxima.

Evaluation. To evaluate a chebfun3 f at an argument x, y, z, we first evaluate the factor quasimatrices $A(x)$, $B(y)$, and $C(z)$ with 1D Chebfun (which uses the Clenshaw recurrence) and then apply (2.2). This computation scales linearly with the number of evaluation points.

Addition and subtraction. If f and g are chebfun3 objects, then a chebfun3 for $f + g$ is computed by calling the constructor, i.e., based on sampling $f + g$ at various

points. The convergence tolerance is scaled not to the samples but to the inputs f and g, so that if $g \approx -f$, for example, the result returned will be zero rather than an attempt to resolve rounding error noise. Differences $f - g$ are computed in the same fashion.

Multiplication and division. The pointwise product fg, invoked by the syntax `f.*g,` is also computed by calling the constructor, as is the quotient f/g, invoked by `f./g.` If g takes the value zero somewhere in the domain, then f/g will usually result in a construction failure. If an exactly infinite value is detected at some point along the way, the result is an immediate error message and exit.

Functions of a chebfun3. If f is a chebfun3, then chebfun3 objects representing functions such as $\exp(f)$ and $\sin(f)$ are computed by calling the constructor. If singularities are encountered, as may happen for example with `log, tan, sqrt`, or `abs`, the construction will usually fail.

Trivial functions of a chebfun3. A few functions can be evaluated without the need for the constructor, notably unary plus (which makes no change) and unary minus (which negates the core tensor T).

Differentiation. Partial derivatives with respect to x, y, or z are computed by differentiating (2.2) with respect to the appropriate variable. For example, if f is a chebfun3, then a chebfun3 for $\partial f/\partial x$ is formed from the equation

$$\frac{\partial f(x,y,z)}{\partial x} \approx T \times_1 \frac{\partial A(x)}{\partial x} \times_2 B(y) \times_3 C(z)$$

making use of the `diff` command of 1D Chebfun to compute $\partial A/\partial x$. Differentiation is invoked in Chebfun3 by the commands `diffx, diffy,` and `diffz,` or by a general command `diff` that combines these capabilities.

Laplacian and biharmonic operators. Commands `lap` and `biharm` (or equivalently `laplacian` and `biharmonic`) are available for computing Δf and $\Delta^2 f$, again mapping a chebfun3 to a chebfun3. The implementation consists of adding up the appropriate partial derivatives.

Definite integrals. The `sum3` command computes the triple definite integral of f. This is based on the integral of (2.2),

$$\int_e^g \int_c^d \int_a^b f(x,y,z)\,dx\,dy\,dz \approx T \times_1 \int_a^b A(x)\,dx \times_2 \int_c^d B(y)\,dy \times_3 \int_e^g C(z)\,dz,$$

with each 1D integral computed by 1D Chebfun (Clenshaw–Curtis quadrature). Following the analogous MATLAB syntax, there is also a `sum` command for integration with respect to just one of the variables, by default x, which maps a chebfun3 to a chebfun2. Thus `sum3(f)` is mathematically equivalent to `sum(sum(sum(f)))`. Similarly the command `sum2` integrates with respect to two variables, by default x and y, mapping a chebfun3 to a chebfun.

Mean and standard deviation. The commands `mean3` and `std3` follow directly from `sum3`, each mapping a chebfun3 to a scalar. Again following the MATLAB pattern, there are commands `mean` and `std` that operate along one dimension, producing chebfun2 objects, and `mean2` and `std2` operate along two dimensions and produce chebfuns.

Indefinite integrals. Like Chebfun and Chebfun2, Chebfun3 has a `cumsum` command that computes an indefinite integral with respect to one variable, by default x. The algorithm is analogous to the above, using (2.2) to reduce the problem to 1D pieces. Similarly `cumsum2` integrates with respect to two variables, by default x and y,

and `cumsum3` integrates with respect to all three variables. All of these operations map a chebfun3 to a chebfun3.

Norms. If f is a chebfun3, then `norm(f)` and `norm(f,'fro')` both return the trivariate 2-norm, i.e., the square root of the integral of $|f^2|$. This could be computed by constructing $\overline{f}f$ and then evaluating its integral as described above, but for greater speed, following a proposal of De Lathauwer [11], we instead make use of the higher-order SVD, described below. The command `norm(f,inf)` returns the maximum of $|f|$, computed as described below. The 1-norm of a chebfun3 also makes sense mathematically, but this operation is not implemented in Chebfun3 since its computation would involve a non-smooth integral. Note that in all of the operations just described, no operator norms are in play, just norms of multivariate functions. This follows the same convention as in Chebfun2 that by default, Chebfun objects are interpreted as functions, not operators, although functionality for certain operator interpretations may also be available when explicitly requested.

Minima and maxima. The commands `min3` and `max3` compute global minima and maxima of chebfun3 objects. At present, our algorithm is a rather simple one that works in most cases but does not fully exploit Chebfun3's rank-compressed structures. The algorithm begins by computing an initial guess for the optimum using sample of f on a discrete tensor of appropriate size. It then improves the initial guess using MATLAB `fmincon` (an interior point algorithm) if the user has the Optimization Toolbox, otherwise `fminsearch` (Nelder–Mead simplex algorithm). Commands `max2, min2, max` and `min` for optimization along just one or two variables are at present implemented only in crude low-accuracy versions.

Higher-order SVD (HOSVD). The higher-order SVD (HOSVD) of a discrete tensor was introduced in [11]. For an order-3 tensor, this uses SVDs of the three modal unfolding matrices to compute a factorization in Tucker format for which the resulting three factor matrices have orthogonal columns and the core tensor is *all orthogonal*, i.e., the horizontal, lateral and frontal slices have orthogonal columns. Chebfun3 has a continuous analogue of this discrete HOSVD. The underlying algorithm starts by calling the Chebfun `qr` method to compute QR factorizations of the three factor quasimatrices $A(x)$, $B(y)$, and $C(z)$ [36]. It then computes the discrete HOSVD of the core tensor of the chebfun3 object contracted with the triangular factors computed in the QR factorizations.

Vector-valued functions and operations. A chebfun3 is a scalar function, but many 3D applications need three-dimensional vectors, and for this we have also implemented a class `chebfun3v`, modeled on the class `chebfun2v` in Chebfun2. A chebfun3v consists of a triple of three chebfun3 objects,

$$\boldsymbol{f}(x,y,z) = \begin{bmatrix} f(x,y,z) \\ g(x,y,z) \\ h(x,y,z) \end{bmatrix}.$$

The implementation is a straightforward extension of Chebfun3, and we shall not give details. The method `grad` (or `gradient`) computes ∇f, mapping a chebfun3 to a chebfun3v. Conversely, `div` computes $\nabla \cdot \boldsymbol{f}$, mapping a chebfun3v to a chebfun3. The method `curl` computes the curl of a vector field $\boldsymbol{f}$, mapping a chebfun3v to a chebfun3v, and `cross` computes the cross product $\boldsymbol{f} \times \boldsymbol{g}$ of two vector fields.

Curves, surfaces, and vector calculus. One of the major themes of the Chebfun project in recent years has been capabilities for various kinds of computing on curves and surfaces. One aspect of this has been the Spherefun and Diskfun classes for

functions defined on spheres and disks [35, 34]. Another has been various capabilities in both Chebfun2 and now Chebfun3 for performing operations on general curves and surfaces defined by parametrizations. In particular, Chebfun3 includes a method `integral` for computing the line integral of a chebfun3 over a parametrized curve in 3D, and `integral2` for computing the surface integral over a parameterized surface. There are also analogous vector commands by the same names applicable to chebfun3v objects. These capabilities combine to enable a broad range of computations related to vector calculus, involving Green's theorems and Stokes' theorem for example, and some examples are posted at `www.chebfun/org/examples/approx3.`

Rootfinding. Chebfun2 has algorithms for computing the "roots" of a chebfun2, which consist generically of curves that are represented as chebfuns, and the roots of of a chebfun2v, which consist generically of points. In three dimensions the analogous operations are much more challenging, and at present, Chebfun3 has only a primitive command `root` that attempt to return a single root of a chebfun3v. This is an area of ongoing research.

4. Numerical examples. We now give four examples to illustrate Chebfun3 computing. In a few pages we can not show very much, and for more examples, see the *Guide* and the examples collection at `www.chebfun.org.` Our emphasis here is on general capabilities, whereas the next section focusses on algorithmic investigations.

We set the stage by mentioning a Chebfun3 command that users find very convenient, analogous to the commands `cheb.x` and `cheb.xy` for Chebfun and Chebfun2. If one types `cheb.xyz,` then three chebfun3 objects are put into the workspace: `x` representing the function x in the default domain $[-1,1]^3$, `y` representing y, and `z` representing z. In what follows we assume that `cheb.xyz` has been executed.

Example 1. *An arbitrary sequence of operations.* As an illustration entirely without scientific content, consider this sequence of operations:

```
f = exp(x.*y.*z); g = cos(sin(f)); h = log(x+y.*z+f+g)
```

Chebfun produces this output in about 2 sections on a 2015 desktop machine:

```
h =
   chebfun3 object
   cols: Inf x 27 chebfun
   rows: Inf x 36 chebfun
  tubes: Inf x 36 chebfun
   core: 27 x 36 x 36
 length: 55, 54, 54
 domain: [-1, 1] x [-1, 1] x [-1, 1]
 vertical scale = 1.7
```

Here are illustrations of typical ongoing computations:

```
>> sum3(h)
ans = 3.518814806858063

>> [val,pos] = min3(h)
val =
  -0.497265559894641
pos =
```

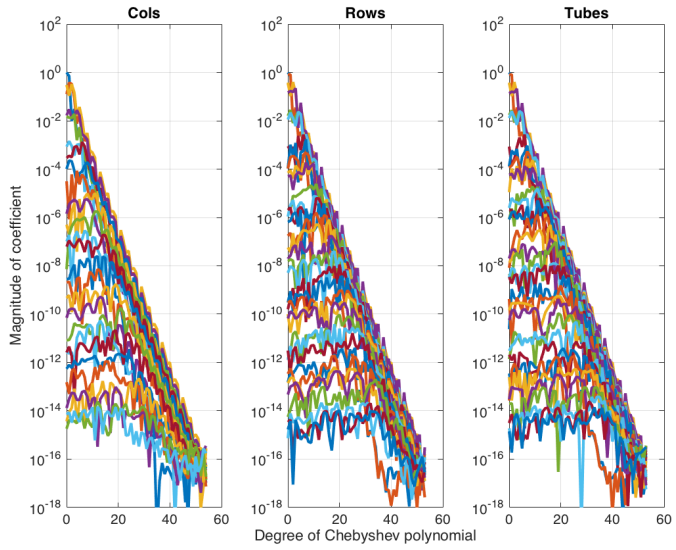

FIG. 4.1. `plotcoeffs(h)` *plot, showing the Chebyshev series involved in representing the columns, rows, and tubes of Example* 1 *to machine precision.*

```
    -1.000000000000000   -0.263375189750789    0.971965306269114

>> h(0,0,0)
ans = 0.510645654808261
```

The command

```
plotcoeffs(h)
```

produces the plot shown in Fig. 4.1, showing the Chebyshev series from which this chebfun3 is constructed.

Example 2. *A periodic lattice function.* The function

$$f(x,y,z) = \cos^2(2\pi x) + \cos^2(2\pi y) + \cos^2(2\pi z) \tag{4.1}$$

is available in the Chebfun3 Gallery collection (`cheb.gallery3('lattice')`). We can construct f and produce a plot like that of Fig. 4.2 with the commands

```
f = cos(2*pi*x).^2+cos(2*pi*y).^2+cos(2*pi*z).^2
isosurface(f,0.3)
```

The image depicts the surface in the unit cube where $f(x,y,z) = 0.3$. This function has a core of size $2 \times 2 \times 2$, with chebfuns in the three directions of lengths $39, 39, 39$. Since it is periodic, it can also be contructed with a trigonometric representation,

```
f = chebfun3(@(x,y,z) cos(2*pi*x).^2+cos(2*pi*y).^2+cos(2*pi*z).^2,'trig')
```

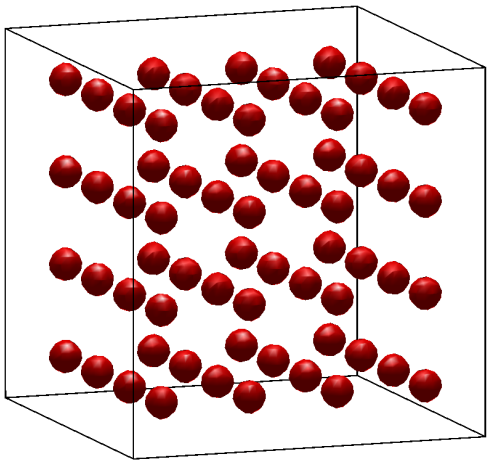

FIG. 4.2. `isosurface(f,0.3)` *plot, showing a level surface of the function* (4.1).

in which case the lengths in the three directions (now of Fourier rather than Chebyshev series) shrink to $9, 9, 9$. Both the maximum and the integral of this function are readily computed in one's head:

```
>> max3(f)
ans = 3

>> sum3(f)
ans = 12.000000000000007
```

Example 3. *Global minimum of a function of Wagon.* One of the problems of the SIAM 100-Dollar, 100-Digit Challenge [8] was to compute the global minimum of the bivariate function

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

This function has rank 4 (exact, not just numerical), and in about 1 second on a desktop computer, the Chebfun2 command `min2(cheb.gallery2('challenge'))` produces the value -3.306868647475238, which is correct in all but the last digit [31]. On p. 99 of the book about the challenge [8], Stan Wagon proposed a generalization of the function to 3D whose optimization should be "a lot harder":

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

In Chebfun3, we find the following in about 2 seconds:

```
>>  f = cheb.gallery3('wagon');
>>  min3(f)
    ans = -3.328338345663266
```

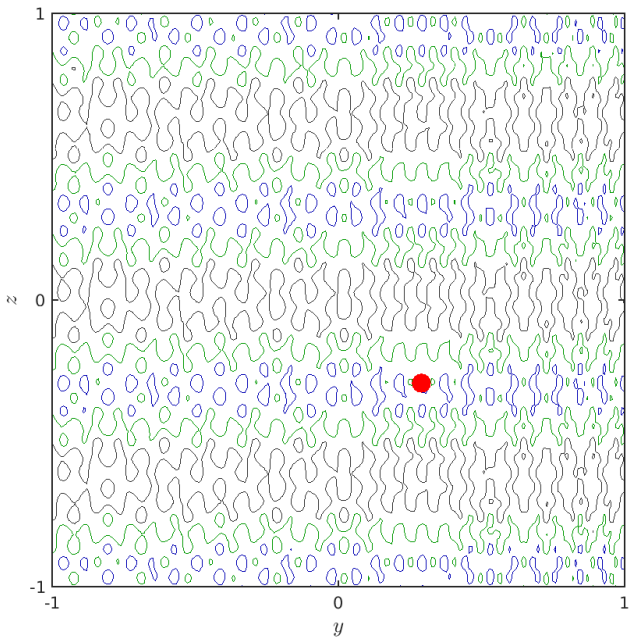

FIG. 4.3. *Contour plot of a slice through Wagon's function* (4.2) *at the critical level* $x = -0.1580368$*, showing its considerable complexity (level curves blue at* -2*, green at* 0*, grey at* 2*). Nevertheless, Chebfun3 finds the 3D global minimum in two seconds, marked by the red dot.*

All but at worst the final two digits of this number must be correct, since it lies in the interval $[-3.328338345663281, -3.328338345663262]$ reported in [8], computed by interval arithmetic. Figure 4.3, showing just one 2D slice, indicates the great complexity of this function, while a look at its structure indicates how Chebfun has nevertheless been able to deal with it efficiently:

```
>> f
f =
   chebfun3 object
   cols: Inf x 4 chebfun
   rows: Inf x 3 chebfun
  tubes: Inf x 5 chebfun
   core: 4 x 3 x 5
 length: 666, 1058, 102
 domain: [-1, 1] x [-1, 1] x [-1, 1]
 vertical scale = 6.9
```

The rank is very low, with a core tensor of size just $4 \times 3 \times 5$, but the lengths in the three dimensions are 666, 1058, 102. Thus Wagon's function turns out to be one for which low-rank compression is extremely effective.

It is interesting to consider, how has the Chebfun3 setting helped us in solving Wagon's minimization problem? — which could of course be solved in other ways too, since an explicit formula for Wagon's function is provided. There are perhaps two

main aspects to this. One is that, by a bit of luck, the function has turned out to have very low rank, making the representation of Chebfun3 very economical. The other, more fundamental observation has to do with the global nature of all optimization (and rootfinding) calculations in Chebfun, Chebfun2, and Chebfun3. These systems make use of global representations of functions, which include implicit information about the space scales on which the functions vary. Thus right at the start of the computation of a minimum, Chebfun3 has available a sample on a grid which is known to be of about the right resolution for the function at hand. These samples on the grid are exploited to find initial guesses for minimization, providing exceptionally fast global optimization in many cases, although without a guarantee of optimality.

Still more fundamentally, Chebfun3 is a system that enables us to perform computations such as minimization on functions that do not, like Wagon's, have explicit representations.

Example 4. *Higher-order singular values of a 3D function.* On p. 233 of [18], Hackbusch reports the modal singular values of the function

$$f(x,y,z) = xz + x^2y, \quad (x,y,z) \in [0,1]^3. \tag{4.2}$$

We can compute these numbers with Chebfun3 as follows:

```
>> f = chebfun3(@(x,y,z) x.*z + x.^2.*y, [0, 1, 0, 1, 0, 1]);
>> s = hosvd(f)
>> s{1}
   0.549642914043599
   0.025892949222491
>> s{2}
   0.548590017185186
   0.042740739611470
>> s{3}
   0.548590017185186
   0.042740739611470
```

These results match those of Hackbusch to 16 digits. Note the decay of each set of modal singular values, analogous to the decay of singular values of bivariate functions.

5. Exploring low rank approximation issues. Hundreds of authors have employed low-rank compression in hundreds of applications. What distinguishes Chebfun3 is its application of such methods to functions rather than discrete tensors, and its ambition to be a general computational tool, providing results to machine precision for all kinds of 3D functions of unpredictable nature. The generality of the tool also makes it a good laboratory for exploration of issues of function representation and low-rank compression, with the pure tensor-product alternative Chebfun3t providing a useful comparator.

Our first example illustrates that some functions have low rank for algebraic reasons. Consider $f(x,y,z) = \cos(k\pi(x+y+z))$, where k is a parameter. For any $k \neq 0$, the mathematical rank is 2, and Chebfun3 exploits this property. For example, here we take $k = 500$.

```
>> tic, f = chebfun3(@(x,y,z) cos(500*pi*(x+y+z))), toc
f =
   chebfun3 object
   cols: Inf x 2 chebfun
```

```
    rows: Inf x 2 chebfun
   tubes: Inf x 2 chebfun
    core: 2 x 2 x 2
 length: 1684, 1684, 1684
 domain: [-1, 1] x [-1, 1] x [-1, 1]
vertical scale = 1
Elapsed time is 0.206855 seconds.
```

Note that the degrees in each direction are large – approximately $k\pi$. This means that polynomials of high degree are involved in representing f, but only univariate ones. The mean of f^2 comes out correctly as ≈ 0.5:

```
>> tic, mean3(f.^2), toc
ans = 0.499999999999952
Elapsed time is 3.385625 seconds.
```

Attempting to represent this function in a purely tensor manner would be completely impractical. Even if k is reduced to 50, the expense is great:

```
>> tic, f = chebfun3t(@(x,y,z) cos(50*pi*(x+y+z))), toc
f =
   chebfun3t object
   coeffs: 214 x 214 x 214
   domain: [-1, 1] x [-1, 1] x [-1, 1]
   vertical scale = 1
Elapsed time is 33.066422 seconds.
```

Not many functions have exactly low rank for reasons of algebra, but quite a few have numerically low rank for reasons of analysis. In [38] it is argued that there are two main properties that enable functions to be well approximated with low rank: localization of (near-) singularities, and alignment with axes. As an example of the first kind, here is a trivariate Runge function. In this and the subsequent displays of chebfun3 objects, we have deleted the "cols", "rows", and "tubes" lines, which can be inferred from the core dimensions, and also the relatively uninteresting display of vertical scale.

```
>> tic, f = chebfun3(@(x,y,z) 1./(1+25*(x.^2+y.^2+z.^2))), toc
f =
   chebfun3 object
   core: 19 x 19 x 19
 length: 191, 191, 191
 domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 1.140339 seconds.
```

The low-rank nature of f is reflected in the fact that the lengths of each univariate piece are ten times the dimension of the core tensor in each direction. This is caused by the fact that the behavior of f is dominated by a localized region near $(0, 0, 0)$. A similar construction with Chebfun3t cannot take advantage of this structure, and takes much longer:

```
>> tic, f = chebfun3t(@(x,y,z) 1./(1+25*(x.^2+y.^2+z.^2))), toc
f =
```

```
   chebfun3t object
   coeffs: 177 x 179 x 181
   domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 11.127352 seconds.
```

The complexity of the Chebfun3t construction is reflected in its needing many function evaluations along the way: 26,328,437 as compared with 903,380 for Chebfun3.

Another feature of Chebfun3 is its dimension clustering step, which attempts to split one dimension from the other two in an advantageous way. For example, here is an entirely satisfactory construction process involving a function that is complicated in two directions and relatively simple in the third. This kind of difference between coordinates arises very commonly in applications.

```
>> tic, f = chebfun3(@(x,y,z) tanh(5*(x+z)).*exp(y)), toc
f =
   chebfun3 object
   core: 73 x 1 x 73
 length: 126, 15, 124
 domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 0.324121 seconds.
```

The algorithm has successfully determined that the variable y should be separated from x and z in Phase 1. If we override this decision and force x to be separated from y and z, the computation becomes very slow, even though the end result is approximately the same:

```
>> tic, f = chebfun3(@(x,y,z) tanh(5*(x+z)).*exp(y),'fiberDim',1), toc
f =
   chebfun3 object
   core: 72 x 1 x 72
 length: 120, 15, 121
 domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 28.486261 seconds.
```

Some functions, unfortunately, lack the kind of structure that can be readily compressed to low rank. For example, the function $f(x,y,z) = 1/\cosh^2(k(x+y+z))$, which is not well aligned with the x, y, or z axes, remains highly complicated in our representations when k is large. Even for $k = 3$, Chebfun3 is slow:

```
>> tic, f = chebfun3(@(x,y,z) 1./cosh(3*(x+y+z)).^2), toc
f =
   chebfun3 object
   core: 59 x 59 x 58
 length: 84, 82, 82
 domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 10.136970 seconds.
```

Essentially no compression is taking place here; the ratio of dimensions about 82 in the univariate lengths to 59 in the core tensor reflects not low-rank compression but the $\pi/2$ factor associated with the irregular spacing of Chebyshev grids. The timing will get rapidly worse if k is increased. By contrast the pure tensor approach of Chebfun3t wastes no time exploring compressed representations.

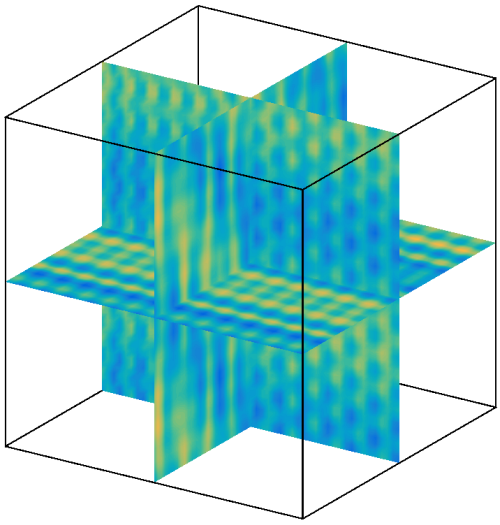

FIG. 5.1. *A "*`slice`*" plot of the chebfun3 for Wagon's function in the box* $[-0.5, 0.5]^3$.

```
>> tic, f = chebfun3t(@(x,y,z) 1./cosh(3*(x+y+z)).^2), toc
   chebfun3t object
   coeffs: 81 x 82 x 83
   domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 1.539751 seconds.
```

As before, the difference in timings is reflected in counts of the function evaluations. For this example Chebfun3 requires 9,354,852 evaluations and Chebfun3t just 3,325,276.

We finish with another illustration of the power of low-rank compression for functions aligned with the axes. In Example 3 of the last section we considered the complicated 3D function devised by Stan Wagon for global optimization. The function is of such low rank that Chebfun3 represents it quickly,

```
>> [~,ff] = cheb.gallery3('wagon');
>> tic, f = chebfun3(ff), toc
f =
   chebfun3 object
   core: 4 x 3 x 5
 length: 666, 1058, 102
 domain: [-1, 1] x [-1, 1] x [-1, 1]
Elapsed time is 0.172304 seconds.
```

Attempting to represent this function in Chebfun3t, however, would be utterly impractical. To bring the time under a minute we can shrink the domain by a factor of 2 in each dimension:

```
>> tic, chebfun3t(ff,.5*[-1 1 -1 1 -1 1]), toc
f =
   chebfun3t object
   coeffs: 307 x 464 x 62
   domain: [-0.5, 0.5] x [-0.5, 0.5] x [-0.5, 0.5]
Elapsed time is 42.131969 seconds.
```

Figure 5.1 shows the plot produced with the Chebfun3 command `slice(f,0,0,0)`. To bring the time down to that of the Chebfun3 construction, it would be necessary to shrink the cube by a factor of 25 in each dimension, that is, to reduce it to $[-.04, .04]^3$.

6. Related tensor approximations. As mentioned at the outset, many representations of multivariate functions are available. We do not claim that the choices we have made in designing Chebfun3 are the only reasonable ones. For important alternatives see, for example, [16, 20, 28]. However, to further explain some of the thinking behind these choices, we shall now comment on three other methods in the tensor field.

Polyadic decomposition. Our representation (2.2) involves a dense core tensor, T. An alternative approach is effectively to require this object to be diagonal, giving the *polyadic decomposition,* where f is decomposed into a sum of outer products of univariate functions [7]:

$$f(x,y,z) \approx \sum_{l=1}^{r} \alpha_l a_l(x) b_l(y) c_l(z).$$

Given a discrete tensor F of size $m \times n \times p$, the r-term polyadic decomposition of F has the attractive property that the number of parameters needed to represent F is only $r(m+n+p)$. The *smallest* possible r is called the *tensor rank* or the *canonical rank*. If r turns out to be the tensor rank, then the above representation is called the *canonical polyadic (CP) decomposition*[4]. Unfortunately, it is not easy to determine the optimal value for r in the course of computation, so in practice an estimate of r is often simply fixed beforehand. Another point worth mentioning is the lack of the so-called *interpolation property* in current implementations of polyadic decomposition, as noted in [5]. Specifically, one of the reasons why we choose a slice and a fiber at each step of Phase 1 of our algorithm rather than three fibers as in one variant of MACA which forms a polyadic decomposition is that zeros introduced at one step might be lost at subsequent steps of the latter approach. See [18, p. 455] for further discussion. Even if one fixes a value of r, most algorithms for the polyadic format involve steps requiring optimization, whereas our approach based on a Tucker format is carried out in a purely algebraic fashion.

Tensor train. Another method of low rank approximation is the tensor train (TT) format [26]. The great power of the TT approach emerges in high dimensions, as has recently been shown by Gorodetsky, Karaman and Marzouk [15]. For applications like ours in three dimensions, it is not so advantageous. The TT decomposition of a discrete tensor of size $m \times n \times p$ consists of a 'train' with just three 'carriages': a matrix of size $m \times r_1$, a tensor of size $r_1 \times n \times r_3$, and a matrix of size $r_3 \times p$. This representation is similar to the Tucker2 format [19], except with one of the three factor matrices of the Tucker format set to be the identity matrix. In the continuous

[4] In theory the tensor rank might be as great as $\min\{mn, mp, np\}$.

framework, the TT representation of a 3D function is

$$f(x,y,z) \approx \sum_{i=1}^{r_1} \sum_{j=1}^{r_2} a_i(x) b_{ij}(y) c_j(z).$$

In contrast, our slice-Tucker approach corresponds to the Tucker (i.e., Tucker3) decomposition. The difference is obvious as Tucker3 allows the function to be compressed three times, i.e., in all three variables, while Tucker2 (or tensor train) does this twice.

Higher-order SVD. In principle one could use the higher order SVD to represent a discrete tensor in the Tucker format, offering a certain kind of optimality. Similarly, one could use the SVD to compute low rank approximation of bivariate functions in Chebfun2. However, there is a substantial cost in computing these optimal approximations which is unlikely to be amortized in the context of numerical computing with functions, where new objects are constantly being created from existing ones. Thus in 2D, Chebfun2 is based on ACA, a much faster alternative to SVD that usually delivers approximations that are reasonably close to optimal. Specifically, assume that $f(x,y)$ is a bivariate function of numerical rank r and the degree of Chebyshev interpolants needed to resolve f to a specified accuracy (machine epsilon by default) is $m-1$ in x and $n-1$ in y. Then, the number of arithmetic operations needed in the ACA-based Chebfun2 constructor is $\mathcal{O}\big(r^3+r^2(m+n)+r(m\log m+n\log n)\big)$, i.e., the cost is loglinear in the number of Chebyshev coefficients. The Chebfun2 constructor also involves $\mathcal{O}\big(r^2+r(m+n)\big)$ evaluations of f. As we have shown in Proposition 2.1, the chebfun3 constructor also has log-linear complexity in the length of the function.

7. Conclusion. Chebfun3 was written by the first author during 2014–16 and has been a part of Chebfun since the V5.5.0 release on 1 July 2016, freely available from `www.chebfun.org` or `http://github/chebfun/chebfun`. This toolbox is the first of its kind, and we believe it has great potential for all kinds of computations to near-machine precision involving smooth scalar or vector functions in a box. Extensions to functions with singularities, solution of PDEs, and non-rectangular domains are mostly challenges for the future, but one is available already: the "spin3" code for fast solution of stiff PDE in periodic 3D domains by exponential integrators [23].

Acknowledgements. We would like to thank all the members of the Chebfun team for their help and many insights along the way with Chebfun3, especially Jared Aurentz, Anthony Austin and Alex Townsend. We are also grateful for valuable comments and stimulating discussions to Sergey Dolgov, Mike Espig, Tamara Kolda, and Daniel Kressner.

REFERENCES

[1] B. W. Bader, T. G. Kolda, et al., *MATLAB Tensor Toolbox Version 2.6*. Available online, February 2015. `http://www.sandia.gov/~tgkolda/TensorToolbox/`.

[2] Z. Battles and L. N. Trefethen, *An extension of MATLAB to continuous functions and operators*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1743–1770.

[3] M. Bebendorf, *Hierarchical Matrices*, Springer, 2008.

[4] ———, *Adaptive cross approximation of multivariate functions*, Constructive Approximation, 34 (2011), pp. 149–179.

[5] M. Bebendorf, A. Kühnemund, and S. Rjasanow, *An equi-directional generalization of adaptive cross approximation for higher-order tensors*, Applied Numerical Mathematics, 74 (2013), pp. 1–16.

[6] M. Bebendorf and C. Kuske, *Separation of variables for function generated high-order tensors*, Journal of Scientific Computing, 61 (2014), pp. 145–165.

[7] G. Beylkin and M. J. Mohlenkamp, *Algorithms for numerical analysis in high dimensions*, SIAM Journal on Scientific Computing, 26 (2005), pp. 2133–2159.
[8] F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel, *The SIAM 100-Digit Challenge: a Study in High-Accuracy Numerical Computing*, SIAM, 2004.
[9] A. Cichocki, R. Zdunek, A. H. Phan, and S.-i. Amari, *Nonnegative Matrix and Tensor Factorizations*, John Wiley & Sons, 2009.
[10] L. De Lathauwer, *Decompositions of a higher-order tensor in block terms-Part II: Definitions and uniqueness*, SIAM Journal on Matrix Analysis and Applications, 30 (2008), pp. 1033–1066.
[11] L. De Lathauwer, B. De Moor, and J. Vandewalle, *A multilinear singular value decomposition*, SIAM Journal on Matrix Analysis and Applications, 21 (2000), pp. 1253–1278.
[12] T. A. Driscoll, N. Hale, and L. N. Trefethen, *Chebfun Guide*, Pafnuty Publications, Oxford, available online, `http://www.chebfun.org/docs/guide/`, 2014.
[13] H. J. Flad, B. N. Khoromskij, D. V. Savostyanov, and E. E. Tyrtyshnikov, *Verification of the cross 3D algorithm on quantum chemistry data*, Russian Journal of Numerical Analysis and Mathematical Modelling, 23 (2008), pp. 329–344.
[14] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013.
[15] A. A. Gorodetsky, S. Karaman, and Y. M. Marzouk, *Function-Train: A continuous analogue of the tensor-train decomposition*, arXiv:1510.09088v1, (2015).
[16] L. Grasedyck, M. Kluge, and S. Krämer, *Alternating least squares tensor completion in the TT-format*, arXiv:1509.00311, (2015).
[17] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitt., 36 (2013), pp. 53–78.
[18] W. Hackbusch, *Tensor Spaces and Numerical Tensor Calculus*, Springer, Heidelberg, 2012.
[19] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.
[20] D. Kressner, M. Steinlechner, and B. Vandereycken, *Low-rank tensor completion by Riemannian optimization*, BIT Numerical Mathematics, 54 (2014), pp. 447–468.
[21] D. Kressner and C. Tobler, *Algorithm 941: htucker—a MATLAB toolbox for tensors in hierarchical tucker format*, ACM Transactions on Mathematical Software, 40 (2014), p. 22.
[22] J. C. Mason, *Near-best multivariate approximation by Fourier series, Chebyshev series and Chebyshev interpolation*, Journal of Approximation Theory, 28 (1980), pp. 349–358.
[23] H. Montanelli and N. Bootland, *Solving periodic semilinear stiff PDEs in 1D, 2D and 3D with exponential integrators*, arXiv:1604.08900, (2016).
[24] K. K. Naraparaju and J. Schneider, *Generalized cross approximation for 3d-tensors*, Computing and Visualization in Science, 14 (2011), pp. 105–115.
[25] I. Oseledets, *TT-Toolbox 2.2.*, available online, `http://spring.inm.ras.ru/osel/?page\_id=24`, (2016).
[26] I. V. Oseledets, *Tensor-train decomposition*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2295–2317.
[27] I. V. Oseledets, D. Savostianov, and E. E. Tyrtyshnikov, *Tucker dimensionality reduction of three-dimensional arrays in linear time*, SIAM Journal on Matrix Analysis and Applications, 30 (2008), pp. 939–956.
[28] M. Steinlechner, *Riemannian optimization for high-dimensional tensor completion*, SIAM Journal on Scientific Computing, 38 (2016), pp. S461–S484.
[29] G. W. Stewart, *Afternotes Goes to Graduate School: Lectures on Advanced Numerical Analysis*, SIAM, 1998.
[30] A. Townsend, *Computing with Functions in Two Dimensions*, PhD thesis, University of Oxford, 2014.
[31] A. Townsend and L. N. Trefethen, *An extension of Chebfun to two dimensions*, SIAM Journal on Scientific Computing, 35 (2013), pp. C495–C518.
[32] ———, *Gaussian elimination as an iterative algorithm*, SIAM News, (2013).
[33] ———, *Continuous analogues of matrix factorizations*, Proc. Roy. Soc. A, 471 (2015), p. 20140585.
[34] A. Townsend, H. Wilber, and G. Wright, *Computing with functions in spherical and polar geometries II. The disk*, arXiv:1604.03061, (2016).
[35] A. Townsend, H. Wilber, and G. B. Wright, *Computing with functions in spherical and polar geometries I. The sphere*, to appear in SIAM Journal on Scientific Computing, (2016).
[36] L. N. Trefethen, *Householder triangularization of a quasimatrix*, IMA Journal of Numerical Analysis, 30 (2010), pp. 887–897.
[37] ———, *Approximation Theory and Approximation Practice*, SIAM, 2013.

[38] ———, *Cubature, approximation, and isotropy in the hypercube.* to appear in SIAM Review, 2016.

[39] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, *Tensorlab 3.0*, available online, `www.tensorlab.net`, (2016).