AN ADAPTIVE PARTITION OF UNITY METHOD FOR MULTIVARIATE CHEBYSHEV POLYNOMIAL APPROXIMATIONS*

KEVIN W. AITON† AND TOBIN A. DRISCOLL†

Abstract. Spectral polynomial approximation of smooth functions allows real-time manipulation of and computation with them, as in the Chebfun system. Extension of the technique to two-dimensional and three-dimensional functions on hyperrectangles has mainly focused on low-rank approximation. While this method is very effective for some functions, it is highly anisotropic and unacceptably slow for many functions of potential interest. A method based on automatic recursive domain splitting, with a partition of unity to define the global approximation, is easy to construct and manipulate. Experiments show it to be as fast as existing software for many low-rank functions, and much faster on other examples, even in serial computation. It is also much less sensitive to alignment with coordinate axes. Some steps are also taken toward approximation of functions on nonrectangular domains, by using least-squares polynomial approximations in a manner similar to Fourier extension methods, with promising results.

Key words. partition of unity, polynomial interpolation, Chebfun, overlapping domain decomposition, Fourier extension

AMS subject classifications. 65L11, 65D05, 65D25

DOI. 10.1137/18M1184904

1. Introduction. A distinctive and powerful mode of scientific computation has emerged recently in which mathematical functions are represented by high-accuracy numerical analogs, which are then manipulated or analyzed numerically using a high-level toolset [22]. The most prominent example of this style of computing is the open-source Chebfun project [6, 10]. Chebfun, which is written in MATLAB, samples a given piecewise-smooth univariate function at scaled Chebyshev nodes and automatically determines a Chebyshev polynomial interpolant for the data, resulting in an approximation that is typically within a small multiple of double precision of the original function. This approximation can then be operated on and analyzed with algorithms that are fast in both the asymptotic and real-time senses. Notable operations include rootfinding, integration, optimization, solution of initial- and boundary-value problems, eigenvalues of differential and integral operators, and solution of time-dependent PDEs.

Townsend and Trefethen extended the one-dimensional Chebfun algorithms to two-dimensional (2D) functions over rectangles in Chebfun2 [20, 21], which uses low-rank approximations in an adaptive cross approximation. The construction and manipulation of 2D approximations is suitably fast for a wide range of smooth examples. Most recently, Hashemi and Trefethen created an extension of Chebfun called Chebfun3 for three-dimensional (3D) approximations on hyperrectangles using low-rank "slice-Tucker" decompositions [14]. The range of functions that Chebfun3 can cope with in a reasonable interactive computing time is somewhat narrower than for Chebfun2, as one would expect.

†Mathematical Sciences, University of Delaware, Newark, DE 19716 (kaiton@udel.edu, driscoll@udel.edu).

A distinctive aspect of the low-rank approximations used by Chebfun2 and Chebfun3 is that they are highly nonisotropic. That is, rotation of the coordinate axes can transform a rank-one or low-rank function into one with a much higher rank, greatly increasing the time required for function construction and manipulations.

An alternative to Chebfun and related projects ported to other languages is sparse grid interpolation. Here one uses linear or polynomial interpolants on hierarchical Smolyak grids. Notable examples of software based on this technique are the Sparse Grid Interpolation Toolbox [16] and the Sparse Grids Matlab Kit [5]. An advantage of these packages is that they are capable of at least medium-dimensional representations on hyperrectangles. However, they seem to be less focused on high-accuracy approximation for a wide range of functions, and they are less fully featured than the Chebfun family. These methods are also highly nonisotropic.

In this work we propose decomposing a hyperrectangular domain by adaptive, recursive bisections in one dimension at a time, generalizing earlier work in one dimension [3]. The resulting subdomains are defined to be overlapping, and on each we employ simple tensor-product Chebyshev polynomial interpolants. In order to define a global smooth approximation, we use a partition of unity to blend together the subdomains. This allows the approximation to capture highly localized function features while remaining computationally tractable. (For a different use of a partition of unity to blend high-order representations, see [8].)

The more general problem of approximation of a function with high pointwise accuracy over a nonrectangular domain $\Omega \subset \mathbb{R}^d$ allows more limited global options than in the hyperrectangular case. Neither low-rank nor sparse grid approximations have any clear global generalizations to this case. Two techniques that can achieve spectral convergence for at least some such domains are radial basis functions [11] and Fourier extension or continuation [1, 7, 9], but neither has been conclusively demonstrated to operate with high speed and reliability over a large collection of domains and functions.

Our use of an adaptive decomposition allows us to approximate on such domains with great flexibility. If a base subdomain is hyperrectangular, we proceed with a tensor-product interpolation for speed, but if its intersection with the global domain is nonrectangular, we can opt for a different representation. We need not be concerned with having a very large number of degrees of freedom in any local subproblem, since further subdivision is available, so the local algorithm need not be overly sophisticated.

The adaptive construction of function approximations is based on binary trees, as explained in section 2. In section 3 we describe fast algorithms for evaluation, arithmetic combination, differentiation, and integration of the resulting tree-based approximations. Numerical experiments over hyperrectangles in section 4 demonstrate that the tree-based approximations exhibit far less anisotropy than do Chebfun2 and Chebfun3. Our implementation is faster than Chebfun2 and Chebfun3 on all tested examples—in one case, by orders of magnitude—except for examples of very low rank, for which all the methods are acceptably fast. In section 5 we describe and demonstrate approximation on nonrectangular domains using a simple linear least-squares approximation by the tensor-product Chebyshev basis. While these results are preliminary, we think they show enough promise to merit further investigation.

2. Adaptive construction. Let $\Omega = \{\mathbf{x} \in \mathbb{R}^d : x_i \in [a_i, b_i], i = 1, \ldots, d\}$ be a hyperrectangle, and suppose we wish to approximate $f : \Omega \to \mathbb{R}$. Our strategy is to cover Ω with overlapping subdomains, on each of which f is well approximated by a multivariate polynomial, and use a partition of unity to construct a global approxi-

mation. We defer a description of the partition of unity scheme to section 3. In this section we describe an adaptive procedure for obtaining the overlapping domains and individual approximations over them.

The domains are constructed from recursive bisections of Ω into nonoverlapping hyperrectangular *zones*. Given a zone $\prod_{j=1}^{d}[\alpha_j, \beta_j]$, we extend it to a larger domain $\prod_{j=1}^{d}[\bar{\alpha}_j, \bar{\beta}_j]$ by fixing a parameter $t > 0$, defining

$$(2.1) \qquad \delta_j = \frac{\beta_j - \alpha_j}{2}(1 + t), \quad j = 1, \dots, d,$$

and then setting

$$(2.2) \qquad \bar{\alpha}_j = \max\{a_j, \beta_j - \delta_j\}, \quad \bar{\beta}_j = \min\{\alpha_j + \delta_j, b_j\}.$$

In words, the zone is extended on all sides by an amount proportional to its width in each dimension, up to the boundary of the global domain Ω.

We define a binary tree $\mathcal{T}$ with each node ν having the following properties:
- zone(ν): zone associated with ν;
- domain(ν): domain associated with ν;
- isdone(ν): n-vector of boolean values, where isdone$_j$ indicates whether the domain is determined to be sufficiently resolved in the jth dimension;
- child$_0(\nu)$,child$_1(\nu)$: left and right subtrees of ν (empty for a leaf);
- splitdim(ν): the dimension in which ν is split (empty for a leaf).

A leaf node has the following additional properties:
- grid(ν): tensor-product grid of Chebyshev second-kind points mapped to domain(ν),
- values(ν): function values at grid(ν),
- interpolant(ν): polynomial interpolant of values(ν) on grid(ν).

If ν is a leaf, its domain is constructed by extending zone(ν) as in (2.2). Otherwise, domain(ν) is the smallest hyperrectangle containing the domains of its children.

Let f be the scalar-valued function on Ω that we wish to approximate. A key task is to compute, for a given leaf node ν, the polynomial interpolant(ν) and determine whether f is sufficiently well approximated on domain(ν) by it. First we sample f at a Chebyshev grid of size N^d on domain(ν). This leads to the interpolating polynomial

$$(2.3) \qquad \tilde{p}(\mathbf{x}) = \sum_{i_1=0}^{N-1} \cdots \sum_{i_d=0}^{N-1} C_{i_1,\dots,i_d} T_{i_1}(x_1) \cdots T_{i_d}(x_d),$$

where the coefficient array C can be computed by FFT in $\mathcal{O}(N^d \log N)$ time [17]. Following the practice of Chebfun3t [14], for each $j = 1, \dots, d$, we define a scalar sequence $\gamma^{(j)}$ by summing $|C_{i_1,\dots,i_d}|$ over all dimensions except the jth. To each of these sequences we apply Chebfun's `StandardChop` algorithm, which attempts to measure decay in the coefficients in a suitably robust sense [4]. Let the output of `StandardChop` for sequence $\gamma^{(j)}$ be n_j; this is the degree that `StandardChop` deems to be sufficient for resolution at a user-set tolerance. If $n_j < N$ we say that the function is resolved in dimension j on ν. If f is resolved in all dimensions on ν, then we truncate the interpolant sums in (2.3) at the degrees n_j and store the samples of f on the corresponding smaller tensor-product grid.

Algorithm 2.1 describes a recursive adaptation procedure for building the binary tree $\mathcal{T}$, beginning with a root node whose zone and domain are both the original

Algorithm 2.1. refine(ν,f,N,t).

---

if ν is a leaf **then**
    Sample f on $\mathsf{grid}(\nu)$
    Determine chopping degrees $n_1, \ldots, n_d$
    for each j with $\mathsf{isdone}(\nu)_j = \text{FALSE}$ **do**
        if $n_j < N$ **then**
            $\mathsf{isdone}(\nu)_j := \text{TRUE}$
        else
            split(ν,j,t)
        end if
    end for
    if all $\mathsf{isdone}(\nu)$ are TRUE **then**
        Truncate (2.3) at degrees $n_1, \ldots, n_d$ to define $\mathsf{grid}(\nu)$, $\mathsf{values}(\nu)$, $\mathsf{interpolant}(\nu)$
    else
        refine(ν,f,N,t)
    end if
else
    refine($\mathsf{child}_0(\nu)$,f,N,t)
    refine($\mathsf{child}_1(\nu)$,f,N,t)
end if

---

hyperrectangle Ω. For a nonleaf input, the algorithm is simply called recursively on the children. For an input node that is currently a leaf of the tree, the function f is sampled, and chopping is used in each unfinished dimension to determine whether sufficient resolution has been achieved. Each dimension that is deemed to be resolved is marked as finished. If all dimensions are found to be finished, then the interpolant is chopped to the minimum necessary length in each dimension, and the node will remain a leaf. Otherwise, the node is split in all unfinished dimensions using Algorithm 2.2, and Algorithm 2.1 is applied recursively. Note that the descendants of a splitting

Algorithm 2.2. split(ν,j,t).

---

if ν is a leaf **then**
    $\mathsf{splitdim}(\nu)=j$
    Define new nodes ν_0, ν_1
    $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$ be the subintervals from $\mathsf{zone}(\nu)$
    Let $m := \frac{b_j + a_j}{2}$
    Let $\mathsf{zone}(\nu_0) := [a_1, b_1] \times \cdots \times [a_{j-1}, b_{j-1}] \times [a_j, m] \times [a_{j+1}, b_{j+1}] \times \cdots \times [a_d, b_d]$
    Let $\mathsf{zone}(\nu_1) := [a_1, b_1] \times \cdots \times [a_{j-1}, b_{j-1}] \times [m, b_j] \times [a_{j+1}, b_{j+1}] \times \cdots \times [a_d, b_d]$
    for $k = 0, 1$ **do**
        Define $\mathsf{domain}(\nu_k)$ from $\mathsf{zone}(\nu_k)$ with parameter t as in (2.2)
        Define $\mathsf{grid}(\nu_k)$ as Chebyshev tensor-product grid of size N^d in $\mathsf{domain}(\nu_k)$
        Let $\mathsf{isdone}(\nu_k) := \mathsf{isdone}(\nu)$
    end for
else
    split($\mathsf{child}_0(\nu)$,k,t)
    split($\mathsf{child}_1(\nu)$,k,t)
end if

inherit the isdone property that marks which dimensions have been finished, so no future splits are possible in such dimensions within this branch.

3. Computations with the tree representation. The procedure of the preceding section constructs a binary tree $\mathcal{T}$ whose leaves each hold an accurate representation of f over a subdomain. These subdomains overlap, and constructing a global partition of unity approximation from them is straightforward.

Define the C^∞ function

$$
(3.1) \qquad \psi_0(x) = \begin{cases} \exp\left(1 - \frac{1}{1-x^2}\right), & |x| \le 1, \\ 0, & |x| > 1, \end{cases}
$$

and let

$$
(3.2) \qquad \ell(x; a, b) = 2\frac{x - a}{b - a} - 1
$$

be the affine map from $[a, b]$ to $[-1, 1]$. Suppose ν is a leaf of $\mathcal{T}$ with domain $\Omega_\nu = \prod[\bar{\alpha}_j, \bar{\beta}_j]$. Then we can define the smoothed-indicator or bump function

$$
(3.3) \qquad \psi_\nu(\mathbf{x}) = \prod_{j=1}^{d} \psi_0\big(\ell(x_j; \bar{\alpha}_j, \bar{\beta}_j)\big).
$$

Next we use Shepard's method [23] to define a partition of unity $\{w_\nu(\mathbf{x})\}$, indexed by the leaves of $\mathcal{T}$:

$$
(3.4) \qquad w_\nu(\mathbf{x}) = \frac{\psi_\nu(\mathbf{x})}{\sum_{\mu \in \mathrm{leaves}(\mathcal{T})} \psi_\mu(\mathbf{x})}.
$$

Let s_ν be the polynomial interpolant of f over the domain of node ν. Then the global approximant is

$$
(3.5) \qquad s(\mathbf{x}) = \sum_{\nu \in \mathrm{leaves}(\mathcal{T})} w_\nu(\mathbf{x}) s_\nu(\mathbf{x}).
$$

Next we describe efficient algorithms using the tree representation of the global approximant to perform common numerical operations such as evaluation at points, basic binary arithmetic operations on functions, differentiation, and integration.

3.1. Evaluation. Note that (3.4) and (3.5) can be rearranged into

$$
(3.6) \qquad s(\mathbf{x}) = \sum_{\nu \in \mathrm{leaves}(\mathcal{T})} \frac{s_\nu(\mathbf{x})\psi_\nu(\mathbf{x})}{\sum_{\mu \in \mathrm{leaves}(\mathcal{T})} \psi_\mu(\mathbf{x})} = \frac{\sum_{\nu \in \mathrm{leaves}(\mathcal{T})} s_\nu(\mathbf{x})\psi_\nu(\mathbf{x})}{\sum_{\mu \in \mathrm{leaves}(\mathcal{T})} \psi_\mu(\mathbf{x})}.
$$

This formula suggests a recursive approach to evaluating the numerator and denominator, presented in Algorithm 3.1. Using it, only leaves containing $\mathbf{x}$ and their ancestors are ever visited. A similar approach was described in [19].

Algorithm 3.1 can easily be vectorized to evaluate $s(\mathbf{x})$ at multiple points, by recursively calling each leaf with all values of $\mathbf{x}$ that lie within its domain. In the

Algorithm 3.1. $[S,P]=\mathrm{numden}(\nu,\mathbf{x})$.

$S = 0,\ P = 0$
if ν is a leaf **then**
   $S = \psi_\nu(\mathbf{x})$
   $P = S \cdot \mathsf{interpolant}(\nu)(\mathbf{x})$
else
   for $k = 0, 1$ **do**
     if $\mathbf{x} \in \mathsf{domain}(\mathsf{child}_k(\nu))$ **then**
       $[S_k, P_k] = \mathrm{numden}(\mathsf{child}_k(\nu),\mathbf{x})$
       $S = S + S_k$
       $P = P + P_k$
     end if
   end for
end if

particular case when the evaluation is to be done at all points in a Cartesian grid, it is worth noting that the leaf-level interpolant in (2.3) can be evaluated by a process that yields significant speedup over a naive approach. As a notationally streamlined example, say that the desired values of $\mathbf{x}$ are $(\xi_{j_1},\ldots,\xi_{j_d})$, where each j_k is drawn from $\{1,\ldots,M\}$, and that the array of polynomial coefficients is of full size $O(N^d)$. Express (2.3) as

$$(3.7) \quad \sum_{i_1=0}^{N-1} \cdots \sum_{i_d=0}^{N-1} C_{i_1,\ldots,i_d} T_{i_1}(\xi_{j_1}) \cdots T_{i_d}(\xi_{j_d})$$

$$= \sum_{i_1=0}^{N-1} T_{i_1}(\xi_{j_1}) \sum_{i_2=0}^{N-1} T_{i_2}(\xi_{j_2}) \cdots \sum_{i_d=0}^{N-1} C_{i_1,\ldots,i_d} T_{i_d}(\xi_{j_d}).$$

The innermost sum yields $N^{d-1}M$ unique values, each taking $\mathcal{O}(N)$ time to compute. At the next level there are $N^{d-2}M^2$ values, and so on, finally leading to the computation of all M^d interpolant values. This takes $\mathcal{O}(MN(M+N)^{d-1})$ operations, as opposed to $\mathcal{O}(M^dN^d)$ when done naively.

3.2. Binary arithmetic operations. Suppose we have two approximations $s_1(\mathbf{x})$, $s_2(\mathbf{x})$, represented by trees $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and we want to construct a tree approximation for $s_1 \circ s_2$, where $\circ$ is one of the operators $+$, $-$, $\times$, or $\div$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ have identical tree structures, then it is straightforward to operate leafwise on the polynomial approximations. In the cases of multiplication and division, the resulting tree may have to be refined further using Algorithm 2.2, since these operations typically result in polynomials of degree greater than the operands.

If the trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are not structurally identical, we are free to use Algorithm 2.2 to construct an approximation by sampling values of $s_1 \circ s_2$. However, the tree of $s_1 \circ s_2$ likely shares refinement structure with both $\mathcal{T}_1$ and $\mathcal{T}_2$. For example, Figure 3.1 shows the refined zones of the trees for $\arctan(100(x^2+y))$, $\arctan(100(x+y^2))$, and their sum. Thus in practice we merge the trees $\mathcal{T}_1$ and $\mathcal{T}_2$ using Algorithm A.1, presented in Appendix A. The merged tree, whose leaves contain sampled values of the result, may then be refined further if chopping tests then reveal that the result is not fully resolved.
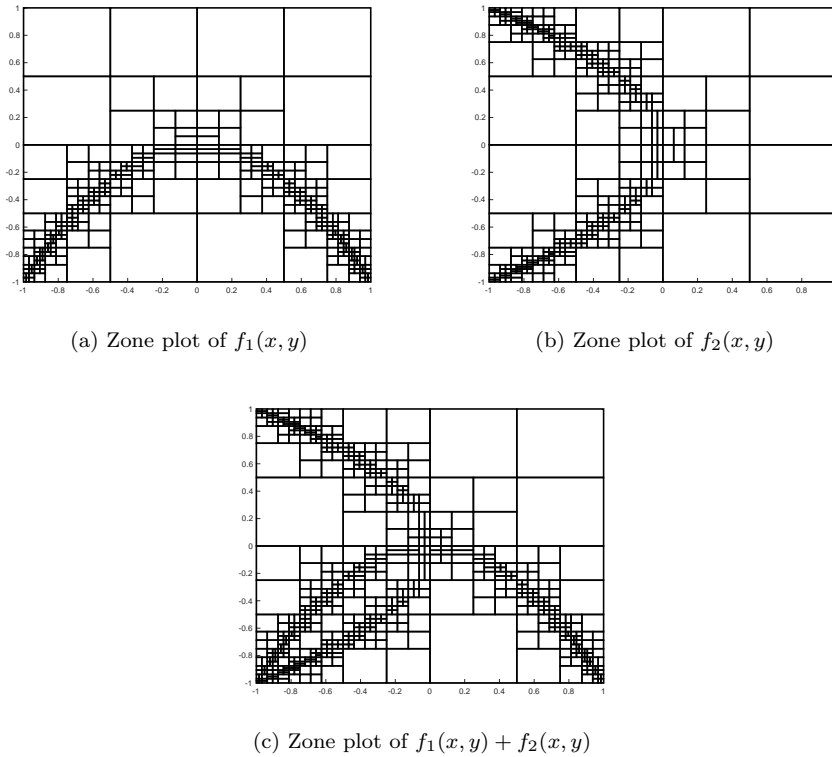
(a) Zone plot of $f_1(x, y)$

(b) Zone plot of $f_2(x, y)$

(c) Zone plot of $f_1(x, y) + f_2(x, y)$

FIG. 3.1. *Zone plots for* $f_1(x, y), f_2(x, y)$ *and* $f_1(x, y) + f_2(x, y)$.

3.3. Differentiation. Differentiation of the global approximant (3.5) results in two groups of terms:

$$\frac{\partial}{\partial x_j} s(\mathbf{x}) = \sum_{\nu \in \text{leaves}(\mathcal{T})} w_\nu(\mathbf{x}) \frac{\partial}{\partial x_j} s_\nu(\mathbf{x}) + \sum_{\nu \in \text{leaves}(\mathcal{T})} s_\nu(\mathbf{x}) \frac{\partial}{\partial x_j} w_\nu(\mathbf{x}).$$

The first sum is a partition of unity approximation of leafwise differentiated interpolants. That is, we simply apply standard spectral differentiation to the data stored in the leaves of $\mathcal{T}$. Although it may seem surprising at first, we can define the desired derivative approximation solely in terms of this first sum and neglect the second with little penalty.

THEOREM 3.1. *Define*

(3.8)
$$s^{(j)}(\mathbf{x}) = \sum_{\nu \in leaves(\mathcal{T})} w_\nu(\mathbf{x}) \frac{\partial}{\partial x_j} s_\nu(\mathbf{x}).$$

Then for all $\mathbf{x} \in \Omega$,

(3.9)
$$\left| s^{(j)}(\mathbf{x}) - \frac{\partial f}{\partial x_j}(\mathbf{x}) \right| \leq \sum_{\mathbf{x} \in domain(\nu)} w_\nu(\mathbf{x}) \left| \frac{\partial s_\nu}{\partial x_j}(\mathbf{x}) - \frac{\partial f}{\partial x_j}(\mathbf{x}) \right|.$$

Proof. By the partition of unity property,

$$s^{(j)}(\mathbf{x}) - \frac{\partial f}{\partial x_j}(\mathbf{x}) = \sum_{\nu \in \text{leaves}(\mathcal{T})} w_\nu(\mathbf{x}) \left[\frac{\partial}{\partial x_j} s_\nu(\mathbf{x}) - \frac{\partial f}{\partial x_j}(\mathbf{x}) \right].$$

□

The result follows because $w_\nu(\mathbf{x}) = 0$ if $\mathbf{x} \notin \text{domain}(\nu)$.

Hence, if $\mathbf{x}$ is not in an overlap region, the error in the global derivative approximation $s^{(j)}$ is the same as for the local approximant. Otherwise, it is bounded—pessimistically, since the weights are positive and sum to unity pointwise—by the sum of errors in all the contributing approximants. Since no point can be in more than 2^d subdomains (and then only near a meeting of hyperrectangle corners), we feel this error is acceptable in two and three dimensions.

3.4. Integration. The simplest and seemingly most efficient approach to integrating over the domain is to do so piecewise over the nonoverlapping zones,

$$(3.10) \qquad \int_\Omega f(\mathbf{x})d\mathbf{x} = \sum_{\nu \in \text{leaves}(\mathcal{T})} \int_{\text{zone}(\nu)} f(\mathbf{x})d\mathbf{x}.$$

Since the leaf interpolants are defined natively over the overlapping domains, they must be resampled at Chebyshev grids on the zones, after which Clenshaw–Curtis quadrature is applied.

4. Numerical experiments. All the following experiments were performed on a computer with a 2.6 GHz Intel Core i5 processor in version 2017a of MATLAB. Our code, which uses a serial object-oriented recursive implementation of the algorithms, is available for download.[1] Comparisons to Chebfun2 and Chebfun3 were done using Chebfun version 5.5.0. We also tried to use the Sparse Grid Interpolation Toolbox [16], but on all the examples we were unable to get it close to our desired error tolerances within its hard-coded limits on sparse grid depth.

4.1. 2D experiments. We first test the 2D functions $\log(1 + (x^2 + y^4)/10^{-5})$, $\arctan((x + y^2)/10^{-2})$, $\frac{10^{-4}}{(10^{-4}+x^2)(10^{-4}+y^2)}$, Franke's function [12], and the smooth functions from the Genz family test package [13]. For each function we record the time of construction, the time to evaluate on a 200×200 grid, and the max observed error on this grid. Table 4.1 shows the results for the new method. For the low-rank test cases, the methods are comparable, with neither showing a consistent advantage; most importantly, both methods are fast. In the tests of high-rank functions, the tree-based method enjoys a clear advantage in both construction and evaluation times. Moreover, the method remains fast enough for interactive computing even as the total number of nodes exceeds 1.6 million. We present plots of the functions and adaptively generated subdomains for the first three test functions in Figures 4.1 and 4.2.

One important aspect of low-rank approximation is that it is inherently non-isotropic. Consider the 2D "plane wave bump"

$$(4.1) \qquad f(x, y) = \arctan(250(\cos(t)x + \sin(t)y))$$

whose normal makes an angle t with the positive x-axis. We compare the construction times of our method to Chebfun2 for $t \in [0, \pi/4]$ in Figure 4.3. We observe the execution time of Chebfun2 varying over nearly three orders of magnitude. While our method is also responsive to the angle of the wave, the variation in time is about half an order of magnitude, and our codes are faster in all but the rank-one case $t = 0$ (for which both methods are fast).

---

[1] https://github.com/kevinwaiton/PUchebfun.

TABLE 4.1

Observed error and wall-clock times for the tree-based (T) and Chebfun2 (C) algorithms with target tolerance 10^{-12} and $N = 129$. Build time is for constructing the approximation object, and eval time for evaluating an approximant on a 200×200 uniform grid (all times in seconds). Also shown: for the tree-based method, the total number of stored sampled function values, and for Chebfun2, the numerically determined rank of the function. Here $u = [0.75, 0.25]$ and $a = [5, 10]$.

Function	Alg.	Error	Build time	Eval time	Points / Rank
$\log(1 + \frac{x_1^2 + x_2^4}{10^{-5}})$	T	1.05×10^{-13}	0.64	0.063	110496
	C	1.14×10^{-6}	2.30	0.10	30
$\arctan(\frac{x_1 + x_2^2}{10^{-2}})$	T	2.15×10^{-12}	2.97	0.31	1553816
	C	7.09×10^{-12}	150	5.0	816
$\frac{10^{-4}}{(10^{-4} + x_1^2)(10^{-4} + x_2^2)}$	T	1.01×10^{-11}	0.73	0.11	145280
	C	5.44×10^{-15}	0.049	0.0037	1
Franke	T	4.22×10^{-15}	0.012	0.0045	16641
	C	1.33×10^{-15}	0.020	0.0024	4
$\cos(u_1 \pi + \sum_{i=1}^{2} a_i x_i)$	T	2.65×10^{-14}	0.013	0.0025	1089
	C	4.47×10^{-14}	0.016	0.0020	2
$\prod_{i=1}^{2} (a_i^{-2} + (x_i - u_i)^2)^{-1}$	T	5.00×10^{-12}	0.056	0.016	29283
	C	1.59×10^{-12}	0.020	0.0022	1
$(1 + \sum_{i=1}^{2} a_i x_i)^{-3}$	T	2.27×10^{-12}	0.012	0.00090	25
	C	2.27×10^{-12}	0.012	0.0021	4
$\exp(-\sum_{i=1}^{2} a_i^2 (x_i - u_i)^2)$	T	1.65×10^{-14}	0.013	0.0026	2145
	C	4.44×10^{-16}	0.015	0.0022	1

(a) $\arctan\left((x + y^2)/0.01\right)$ (b) Overlapping subdomains
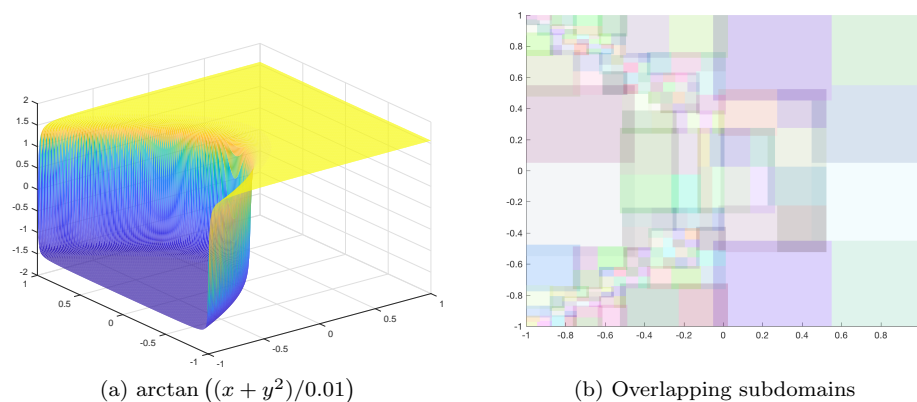
FIG. 4.1. *Overlapping subdomains constructed by the adaptive tree method for a function with a nonlinear "cliff."*

Our next experiment is to add and multiply the rank-one function $\arctan(250x)$ to the plane wave in (4.1). The construction time results are compared for $t \in [0, \pi/2]$ in Figure 4.4. Here the dependence of Chebfun2 on the angle is less severe than in the simple construction, though it is still more pronounced than for our method. More importantly, the absolute numbers for addition in particular with Chebfun2 would probably be considered unacceptable for interactive computation, while our method takes one second at most.

4.2. 3D experiments. We next test the 3D functions $1/(\cosh(5(x + y + z)))^2$, $\arctan(5(x + y) + z)$, and 3D versions of the smooth functions from the Genz family test package. Table 4.2 shows the construction time, the time taken to evaluate on a $200 \times 200 \times 200$ grid, and the max error on this grid. We see that Chebfun3 mostly outperforms the tree-based method for low-Tucker-rank functions, though the results

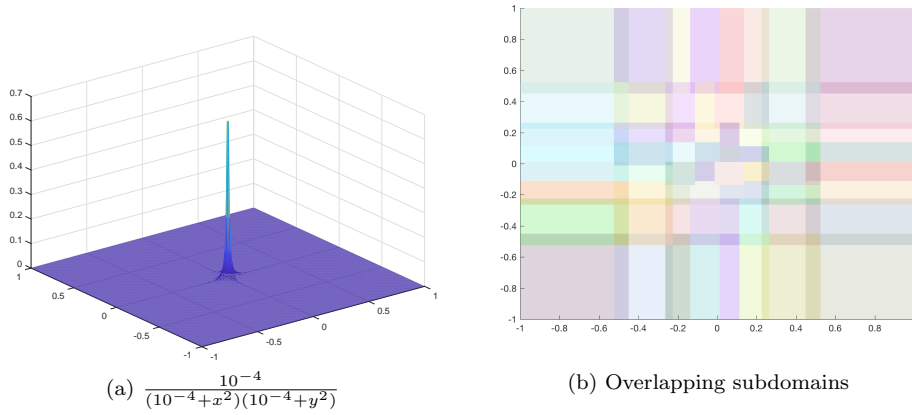(a) $\dfrac{10^{-4}}{(10^{-4}+x^2)(10^{-4}+y^2)}$

(b) Overlapping subdomains

Fig. 4.2. *Overlapping subdomains constructed by the adaptive tree method for a function with a sharp spike.*
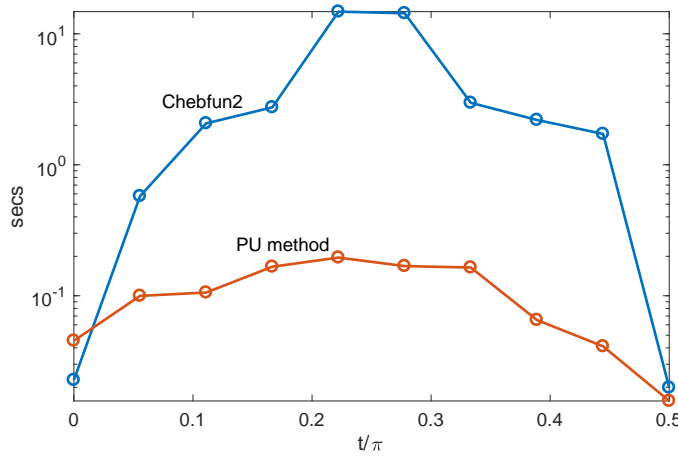

Fig. 4.3. *Comparison of construction times for* $\arctan(250(\cos(t)x + \sin(t)y))$ *for* $t \in [0, \pi/4]$.
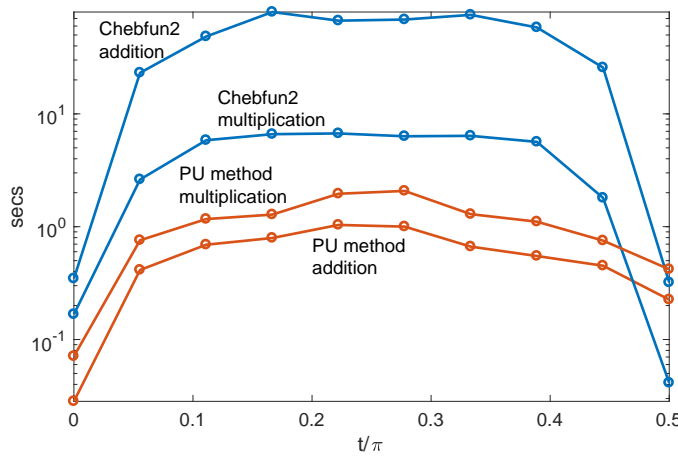

Fig. 4.4. *Comparison of execution times for multiplication and addition of* $\arctan(250x)$ *with* $\arctan(250(\cos(t)x + \sin(t)y))$ *for* $t \in [0, \pi/4]$.

TABLE 4.2

Observed error and wall-clock times for the tree-based (T) and Chebfun3 (C) algorithms with target tolerance 10^{-12} and $N = 65$. Build time is for constructing the approximation object, and eval time for evaluating an approximant on a 200^3 uniform grid (all times in seconds). Also shown: for the tree-based method, the total number of stored sampled function values, and for Chebfun3, the numerically determined rank of the function. Here $u = [0.75, 0.25, -0.75]$ and $a = [25, 25, 25]$.

Function	Alg.	Error	Build time	Eval time	Points / Rank
$\cos(u_1\pi + \sum_{i=1}^3 a_i x_i)$	T	2.27×10^{-13}	0.15	0.12	275000
	C	2.16×10^{-14}	0.061	0.15	2
$\prod_{i=1}^3 (a_i^{-2} + (x_i - u_i)^2)^{-1}$	T	1.52×10^{-5}	7.20	1.31	10400000
	C	5.66×10^{-7}	0.057	0.14	1
$(1 + \sum_{i=1}^3 a_i x_i)^{-3}$	T	4.66×10^{-10}	0.17	0.033	125
	C	4.07×10^{-10}	0.038	0.14	4
$\exp(-\sum_{i=1}^2 a_i^2 (x_i - u_i)^2)$	T	3.11×10^{-15}	0.082	0.078	275000
	C	7.77×10^{-16}	0.047	0.14	1
$1/(\cosh(5(x + y + z)))^2$	T	1.14×10^{-14}	0.71	0.55	2200000
	C	3.69×10^{-13}	81	0.40	93
$\arctan(5(x + y) + z)$	T	7.60×10^{-13}	0.75	0.030	549153
	C	4.69×10^{-13}	90	0.10	110

are mixed for the grid evaluations. In these cases, however, the tree-based method is still operating with acceptable interactive speed. (We also note that for the generalized Runge function, both methods failed to get close to the targeted accuracy.) For the two higher-rank examples, the story is very different. The tree-based method is able to perform all the tests in under a second, while Chebfun3 requires over 80 seconds to construct these cases. Once Chebfun3 finds the approximation, its evaluation time is also under a second.

We repeat our experiment testing the importance of axes alignment using the function

$$(4.2) \qquad \arctan(5(\sin(p)\cos(t)x + \sin(p)\sin(t)y + \cos(p)z))$$

for $p, t \in [0, \pi/4]$. Timing results can be seen in Figure 4.5. As in 2D, the Chebfun low-rank technique shows wide variation depending on the angles and a large region
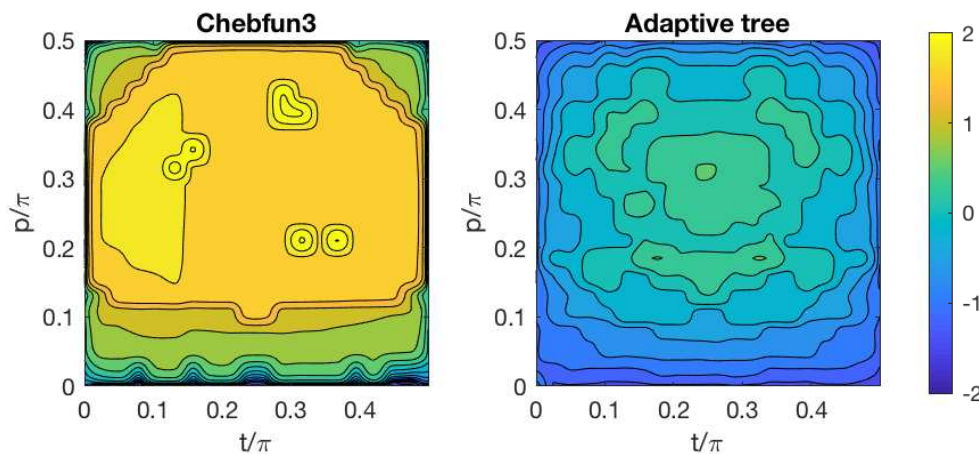

FIG. 4.5. *Construction time comparison for the 3D function* $\arctan(5(\sin(p)\cos(t)x + \sin(p)\sin(t)y + \cos(p)z))$, *with varying angles. Colors and contours correspond to the base-10 log of execution time in seconds.*

of long times. The tree-based method is much less sensitive and faster (by as much as two orders of magnitude) except for the purely axes-aligned cases.

5. Extension to nonrectangular domains. We now consider approximation over a nonrectangular domain $\Omega \subset \mathbb{R}^d$. In our construction, a leaf node ν whose domain Ω_ν lies entirely within Ω can be treated as before. However, if $\Omega_\nu \cap \Omega \subsetneq \Omega_\nu$, we use a different approximation technique on ν. The refinement criteria of Algorithm 2.1 are also modified for this situation.

5.1. Algorithm modifications. On a leaf whose domain extends outside of Ω, we again use a tensor-product Chebyshev polynomial as in (2.3), but choose its coefficient array C by satisfying a discrete least-squares criterion:

$$(5.1) \qquad \arg\min_C \sum_{i=1}^{P} \left(f(\mathbf{x}_i) - \tilde{p}(\mathbf{x}_i) \right)^2 ,$$

where $\Xi = \{\mathbf{x}_i\}_{i=1}^{P} \subset \Omega_\nu \cap \Omega$ is a point set in the "active" part of the leaf's domain, $\Omega_\nu \cap \Omega$. In practice we can form a matrix A whose columns are evaluations of each basis function at the points in Ξ, leading to a standard $P \times N^d$ linear least-squares problem. We choose Ξ as the part of the standard $(2N)^d$-sized Chebyshev grid lying inside Ω.

This technique resembles Fourier extension or continuation techniques [1, 15], so we refer to it as a *Chebyshev extension approximation*. Unlike the Fourier case, however, there is no real domain extension involved; rather one constrains the usual multivariate polynomial only over part of its usual tensor-product domain. The condition number of A in the Fourier extension case has been shown to increase exponentially with the degree of the approximation [2], because the collection of functions spanning the approximation space is a *frame* rather than a basis. We see the same phenomenon with Chebyshev extension; essentially, constraining the polynomial over only part of the hypercube leaves it underdetermined. To cope with the numerical rank deficiency of A, we rely on the basic least-squares solution computed by the MATLAB backslash. We found this to be as good as or better than the pseudoinverse with a truncated SVD.

We modify Algorithm 2.2 so that when a domain is split, the resulting zones of the children are shrunk if possible to just contact the boundary of Ω. (An exception is the shared interface between the newly created children, which is fixed.) This helps to keep a substantial proportion of a leaf's domain within Ω.

We also modify how refinement decisions are made and executed in Algorithm 2.1, for a subtle reason. The original algorithm is able to exploit the very different resolution requirements for a function such as, say, $xT_{60}(y)$, by testing for sufficient resolution in each dimension independently and splitting accordingly. We find experimentally that if the function is like this over Ω, the extension of it to the unconstrained part of the leaf node's domain has uniform resolution requirements in all variables. Therefore, we use a simpler refinement process: if the norm of the least-squares residual (normalized by $\sqrt{P}$) is not acceptably small, we split in all dimensions successively. In effect, the approximation becomes a quadtree or octree within those nodes that do not lie entirely within Ω.

5.2. Numerical experiments. We chose the test functions

$$g_1 = \exp(x + y), \qquad g_2 = \frac{1}{((x-1.1)^2) + (y-1.1)^2)^2},$$

(5.2)

$$g_3 = \cos(24x - 32y)\sin(21x - 28y), \quad g_4 = \arctan(3(x^2 + y)).$$

We approximated each function on each of three domains: the unit disk, the diamond $|x| + |y| \leq 1$, and the double astroid seen in Figure 5.1. The initial box (root of the approximation tree) was chosen to tightly enclose the given domain. For each test we set $N = 17$ and the target tolerance to 10^{-10}. We timed both the adaptive construction and the evaluation on a 200×200 grid and recorded the max error as in the previous section. In each case, we choose initial box to fit the domain as tightly as possible. These results can be seen in Table 5.1. The resulting approximation of g_4 on the double astroid is shown in Figure 5.1, along with the adaptively found subdomains.

When the function is smooth or contains localized features, we find that the method is both efficient and highly accurate; in the smoothest case of g_1, a global multivariate least-squares polynomial is sufficient. Only for g_3, which requires

(a) Plot of $\arctan(3(y^2 + x))$.
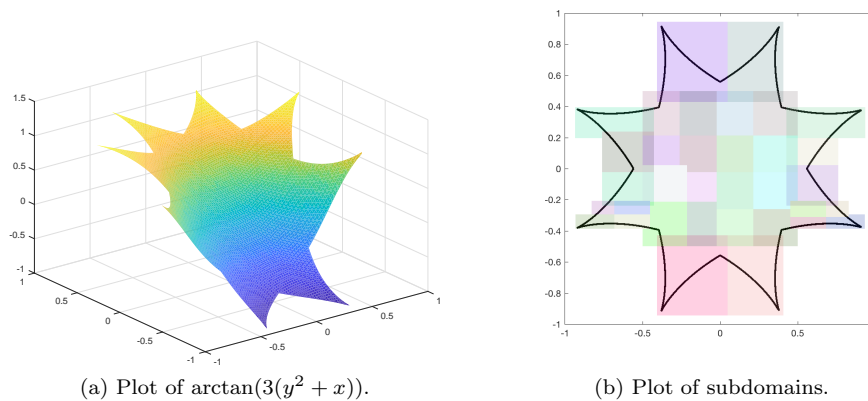

(b) Plot of subdomains.

FIG. 5.1. *Plot of* $\arctan(3(y^2 + x))$ *and the subdomains formed from the partition of unity method. The error in this approximation was found to be about* 10^{-11}.

TABLE 5.1
Observed error and wall-clock times for the adaptive tree method to approximate the functions given in (5.2) on three different 2D domains. Also shown is the total number of sampled function values stored over all the leaves of each final tree.

Function	Domain	Error	Construct time	Interp time	Points
	disk	5.44E-15	1.369	0.012	289
g_1	diamond	2.06E-11	0.040	0.002	289
	astroid	2.01E-08	0.071	0.001	289
	disk	2.40E-10	2.558	0.117	3757
g_2	diamond	2.40E-11	0.406	0.012	2023
	astroid	2.14E-10	1.511	0.023	4624
	disk	4.44E-11	11.305	1.500	245650
g_3	diamond	2.35E-11	10.894	0.854	178020
	astroid	1.67E-10	28.072	0.836	153780
	disk	7.49E-11	1.866	0.059	12138
g_4	diamond	1.45E-11	1.536	0.053	9826
	astroid	1.09E-11	3.221	0.049	9826

uniformly fine resolution throughout the domains, is there a construction time longer than a few seconds. The Fourier extension methods described in [18] are implemented in Julia, making a direct quantitative comparisons difficult, but based on the orders of magnitude of the results reported there, we feel confident that our results for these examples are superior.

6. Concluding remarks. For functions over hyperrectangles of uncorrelated variables or that otherwise are well aligned with coordinate axes, low-rank and sparse-grid approximations can be expected to be highly performant. We have demonstrated an alternative adaptive approach that, in two or three dimensions, still performs very well on such functions but is far less dependent on that property. Our method sacrifices the use of a single global representation that could achieve true spectral convergence, but in practice we are able to use a partition of unity to construct a smooth, global approximation of very high accuracy in a wide range of examples.

The adaptive domain decomposition offers some other potential advantages we have not yet exploited, but are studying. It offers a built-in parallelism for function construction and evaluation. It allows efficient updating of function values locally, rather than globally, over the domain. Finally, it has a built-in preconditioning strategy, based on additive Schwarz methods, for the solution of partial differential equations.

By replacing tensor-product interpolation on the leaves with a simple least-squares approximation using the same multivariate polynomials, we have been able to demonstrate at least reasonable performance in approximation over nonrectangular domains. Further investigation is required to better understand the least-squares approximation process, optimize adaptive strategies, and find efficient algorithms for merging trees and operations such as integration.

Appendix A. Merging trees. Algorithm A.1 describes a recursive method for merging two trees $\mathcal{T}_1$ and $\mathcal{T}_2$, representing functions f_1 and f_2, into a tree representation for $f_1 \circ f_2$, with $\circ$ as $+$, $-$, $\times$, or $\div$. The input arguments to the algorithm are the operation, corresponding nodes of $\mathcal{T}_1$, $\mathcal{T}_2$, and the merged tree, and the number r, which is the dimension that was most recently split in the merged tree. Initially the algorithm is called with root nodes representing the entire original domain, and $r = 0$.

We assume an important relationship among the input nodes. Suppose that $\mathsf{zone}(\nu_k) = \prod_{j=1}^{d} [\alpha_{kj}, \beta_{kj}]$ for $k = 1, 2$, and that $\mathsf{zone}(\nu_{\mathrm{merge}}) = \prod_{j=1}^{d} [A_j, B_j]$. Then we require for $k = 1, 2$ that

$$(A.1) \qquad [a_{kj}, b_{kj}] = [A_j, B_j] \quad \text{for all } j \text{ having} \quad \mathsf{isdone}(\nu_k)_j = \mathrm{FALSE}.$$

This is trivially true at the root level. The significance of this requirement is that it allows us to avoid ambiguity about what the zone of ν_{merge} should be after a new split in, say, dimension j. Since only an uncompleted dimension can be split, the zone of the children of ν_{merge} after splitting will be identical to that of whichever (or both) of the ν_k requires refinement in dimension j.

For example, suppose the zones of ν_1 and ν_2 are $[-1, 0] \times [-1, 1]$ and $[-1, 1] \times [0, 1]$, respectively, and $\mathsf{zone}(\nu_{\mathrm{merge}}) = [-1, 0] \times [0, 1]$. It is clear that we can interpolate from ν_1 and ν_2 onto ν_{merge}. It is also clear that we can further split in x in ν_1, and in y in ν_2. But if we were to split ν_2 in x, one of the children would have zone $[0, 1] \times [0, 1]$, which is inaccessible to ν_1.

Consider the general recursive call. If both ν_1 and ν_2 are leaves, then we simply evaluate the result of operating on their interpolants to get the values on ν_{merge}. If

Algorithm A.1. merge($\circ$,ν_1,ν_2,ν_{merge},r).

---

  if ν_1 and ν_2 are leaves **then**
    values(ν_{merge}):= interpolant(ν_1) $\circ$ interpolant(ν_2), evaluated on grid(T_{merge})
  else if ν_1 is a leaf **and** ν_2 is not a leaf **then**
    split(ν_{merge},splitdim(ν_1))
    merge($\circ$,ν_1,child$_0$(ν_2),child$_0$(ν_{merge}),splitdim(ν_2))
    merge($\circ$,ν_1,child$_1$(ν_2),child$_1$(ν_{merge}),splitdim(ν_2))
  else if ν_1 is not a leaf **and** ν_2 is a leaf **then**
    split(ν_{merge},splitdim(ν_1))
    merge(child$_0$(ν_1),ν_2,child$_0$(ν_{merge}),splitdim(ν_1))
    merge(child$_1$(ν_1),ν_2,child$_1$(ν_{merge}),splitdim(ν_1))
  else
    if splitdim(ν_1)=splitdim(ν_2) **then**
      split(ν_{merge},splitdim(ν_1))
      merge($\circ$,child$_0$(ν_1),child$_0$(ν_2),child$_0$(ν_{merge}),splitdim(ν_1))
      merge($\circ$,child$_1$(ν_1),child$_1$(ν_2),child$_1$(ν_{merge}),splitdim(ν_1))
    else
      $r_1 = ($splitdim(ν_1) $- r - 1) \mod d$
      $r_2 = ($splitdim(ν_2) $- r - 1) \mod d$
      If $r_1 > r_2$, swap ν_1 and ν_2
      split(ν_{merge},splitdim(ν_1))
      merge($\circ$,child$_0$(ν_1),ν_2,child$_0$(ν_{merge}),splitdim(ν_1))
      merge($\circ$,child$_1$(ν_1),ν_2,child$_1$(ν_{merge}), splitdim(ν_1))
    end if
  end if

---

exactly one of ν_1 and ν_2 is a leaf, then we split ν_{merge} the same way as the nonleaf and recurse into the resulting children; property (A.1) trivially remains true in these calls. If both ν_1 and ν_2 are nonleaves, and they both split in the same dimension, then we can split ν_{merge} in that dimension and recurse, and the zones will continue to match as in (A.1).

The only remaining case is that ν_1 and ν_2 are each split, but in different dimensions. In this case we have to use information about how the splittings are constructed in Algorithm 2.1. Recall that each unresolved dimension is split in order, while resolved dimensions are flagged as finished in all descendants. By inductive assumption, ν_{merge} was most recently split in dimension r. The algorithm determines which ν_k has splitting dimension j that comes the soonest after r (computed cyclically). Thus for all dimensions between r and j, neither of the given nodes splits, so it and its descendants all must have isdone set to TRUE in those dimensions, and property (A.1) makes no requirement. Furthermore, the dimension r_k does satisfy (A.1) for ν_k, and the same will be true for its children and the children of ν_{merge}. All other dimensions will inherit (A.1) from the parents.

REFERENCES

[1] B. ADCOCK AND D. HUYBRECHS, *On the resolution power of Fourier extensions for oscillatory functions*, J. Comput. Appl. Math., 260 (2014), pp. 312–336.
[2] B. ADCOCK, D. HUYBRECHS, AND J. MARTÍN-VAQUERO, *On the numerical stability of Fourier extensions*, Found. Comput. Math., 14 (2014), pp. 635–687.

[3] K. W. Aiton and T. A. Driscoll, *An adaptive partition of unity method for Chebyshev polynomial interpolation*, SIAM J. Sci. Comput., 40 (2018), pp. A251–A265, https://doi.org/10.1137/17m112052x.

[4] J. L. Aurentz and L. N. Trefethen, *Chopping a Chebyshev series*, ACM Trans. Math. Software, 43 (2017), pp. 33:1–33:21, https://doi.org/10.1145/2998442.

[5] J. Bäck, F. Nobile, L. Tamellini, and R. Tempone, *Stochastic spectral Galerkin and collocation methods for PDEs with random coefficients: A numerical comparison*, in Spectral and High Order Methods for Partial Differential Equations, J. Hesthaven and E. Ronquist, eds., Lect. Notes Comput. Sci. Eng. 76, Springer, Cham, 2011, pp. 43–62.

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

[7] J. P. Boyd, *A comparison of numerical algorithms for Fourier extension of the first, second, and third kinds*, J. Comput. Phys., 178 (2002), pp. 118–160, https://doi.org/10.1006/jcph.2002.7023.

[8] O. P. Bruno and L. A. Kunyansky, *A fast, high-order algorithm for the solution of surface scattering problems: Basic implementation, tests, and applications*, J. Comput. Phys., 169 (2001), pp. 80–110, https://doi.org/10.1006/jcph.2001.6714.

[9] O. P. Bruno and M. Lyon, *High-order unconditionally stable FC-AD solvers for general smooth domains i. basic elements*, J. Comput. Phys., 229 (2010), pp. 2009–2033, https://doi.org/10.1016/j.jcp.2009.11.020.

[10] T. A. Driscoll, N. Hale, and L. N. Trefethen, eds., *Chebfun Guide*, Pafnuty Publications, Oxford, UK, 2014.

[11] B. Fornberg and N. Flyer, *A Primer on Radial Basis Functions with Applications to the Geosciences*, SIAM, Philadelphia, PA, 2015.

[12] R. Franke, *A Critical Comparison of Some Methods for Interpolation of Scattered Data*, tech. report, Naval Postgraduate School, Monterey, CA, 1979.

[13] A. Genz, *A package for testing multiple integration subroutines*, in Numerical Integration, P. Keast and G. Fairweather, eds., Springer, Cham, 1987, pp. 337–340.

[14] B. Hashemi and L. N. Trefethen, *Chebfun in three dimensions*, SIAM J. Sci. Comput., 39 (2017), pp. C341–C363, https://doi.org/10.1137/16m1083803.

[15] D. Huybrechs, *On the Fourier extension of nonperiodic functions*, SIAM J. Numer. Anal., 47 (2010), pp. 4326–4355.

[16] A. Klimke and B. Wohlmuth, *Algorithm 847: Spinterp: Piecewise multilinear hierarchical sparse grid interpolation in MATLAB*, ACM Trans. Math. Software, 31 (2005), pp. 561–579, https://doi.org/10.1145/1114268.1114275.

[17] J. C. Mason and D. C. Handscomb, *Chebyshev Polynomials*, CRC Press, Boca Raton, FL, 2002.

[18] R. Matthysen and D. Huybrechs, *Function approximation on arbitrary domains using Fourier extension frames*, SIAM J. Numer. Anal., 56 (2018), pp. 1360–1385.

[19] I. Tobor, P. Reuter, and C. Schlick, *Reconstructing multi-scale variational partition of unity implicit surfaces with attributes*, Graph. Models, 68 (2006), pp. 25–41.

[20] A. Townsend and L. N. Trefethen, *An extension of Chebfun to two dimensions*, SIAM J. Sci. Comput., 35 (2013), pp. C495–C518.

[21] A. Townsend and L. N. Trefethen, *Continuous analogues of matrix factorizations*, Proc. A, 471 (2014), 20140585, https://doi.org/10.1098/rspa.2014.0585.

[22] L. N. Trefethen, *Computing numerically with functions instead of numbers*, Commun. Amer. Math. Soc., 58 (2015), pp. 91–97, https://doi.org/10.1145/2814847.

[23] H. Wendland, *Scattered Data Approximation*, Cambridge University Press, Cambridge, UK, 2004.