1 ND PU Structure

Since we are thinking of mixing methods for our PU approximation, it seems we should think more about the modularity of the method. My first thought is to have an abstract PATCH ν with properties

- domain(ν):= domain of ν
- boundaryIndex(ν):= boundary index of points of ν
- length(ν):= length of ν
- dim(ν):= dimension of the space domain(ν) lies in

with methods

- points(ν):=returns the points of ν
- evalf(ν,x):=evaluates the approximate for ν at x.
- sample(ν,f(x)):= samples $f(x)$ at the leaves of ν

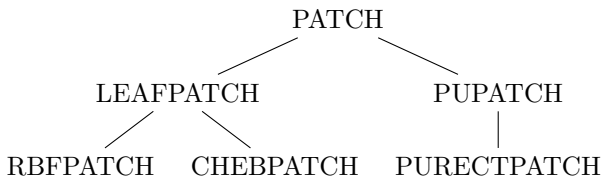


Figure 1: Object structure for the patches.

Here RBFPATCH and CHEBPATCH would be used as leaves where RECTPUPATCH would be used for nonleaves. A PUPATCH has additional properties

- $c_0(\nu)$, $c_1(\nu)$:= two PATCH children
- $w_0(\nu)$, $w_1(\nu)$:= weights used for children

The methods would be defined, ignorant of the type of patches the children are. As an example, the evalf(ν,x) method would be defined similarly as before where the leaves are assumed to be type PATCH (this objects would have there own evalf method). By defining different objects for different methods (RBF,CHEB), we can use this generic method for our different cases (a box, arbitrary boundary).

The leaf patches would be defined for the specific method at the leaves. We next define a method

- PATCH ν_1 = splitleaf(LEAFPATCH ν)

that returns a PUPATCH with two children if ν needs to be split, and ν if it does not to be split. We can then define a refine method for a tree T and its nodes ν.

Algorithm 1 v=eval(ν,x)

```
if ν is a leaf then
  p:=interpolant(ν)
  v:= p(x)
else
  v_0, v_1:=0
  w_0:=w_0(ν)
  w_1:=w_1(ν)
  for k = 0, 1 do
    if x ∈ domain(c_k(ν)) then
      v_k:=eval(c_k(ν),x)
    end if
  end for
  v := w_0(x)v_0 + w_1(x)v_1
end if
```

Algorithm 2 T = refine($f(x)$)

```
define T has a tree with single node
while T has unrefined leaves do
  sample(T,f(x))
  if root(T) is a leaf then
    root(T) = splitleaf(root(T))
  else
    PUsplit(root(T))
  end if
end while
```

Algorithm 3 PUsplit(ν)

```
for k = 0, 1 do
  if c_k(ν) is a leaf and unresolved then
    c_k(ν):=splitleaf(c_k(ν))
  else
    PUsplit(c_k(ν))
  end if
end for
merge(ν)
```

2 Calculating Derivatives and Partition of Unity Weights

For the ND problem, we intend to evaluate the derivatives instead of constructing differentiation matrices. Suppose we have a method evalf(ν,x,dim,j) which evaluates the derivative of order j along dimension dim at x. We can compute this simply using the product rule as seen in Algorithm 4. While this is formulation is simple, it is inefficient. In this formulation we end up repeating calculations for the derivatives of the children of the node. This can be avoided by passing up the evaluations of the approximation and its derivatives up to the desired order.

Algorithm 4 v = evalf(ν,x,dim,j)

w_0:=w$_0(\nu)$
w_1:=w$_1(\nu)$
v = 0
for $k = 0, 1$ **do**
  v = v+$\sum_{i=1}^{j} \binom{j}{i} \partial_{dim}^{j-1} w_k(x)$ evalf(c$_k(\nu)$,x,dim,j)
end for

For our method we intend to split only along one dimension. In this case our partition of unity weights need only to depend on the dimension that we are splitting along. Some care must be used when calculating the derivatives; if the dimension we are differentiation along is different than the splitting dimension, the derivative of the weights are zero.

3 Barycentric Interpolation on ND Chebyshev grids

Suppose we have a tensor product of Chebyshev grids with points $\{x_i\}, \{y_j\}$. Let $\{l_{x_i}(x)\}$, $\{l_{y_i}(x)\}$ be the Lagrange polynomials for points $\{x_i\}, \{y_j\}$. We then have the interpolating polynomial for a function $f(x, y)$ is

$$p(x, y) = \sum_j \sum_i l_{y_j}(y) l_{x_i}(x) f(x_i, y_j) \tag{1}$$

which can be expressed as

$$p(x, y) = \sum_j l_{y_j}(y) \sum_i l_{x_i}(x) f(x_i, y_j). \tag{2}$$

This implies that we can evaluate p(x,y) by:

- Evaluating $c_j = \sum_i l_{x_i}(x) f(x_i, y_j)$ for all points $\{y_j\}$,
- and then evaluating $\sum_j l_{y_j}(y) c_j$

We can thus evaluate $p(x, y)$ with interpolation methods in one dimension. This is illustrated in Figure 2.

Chebfun includes a `bary(x,F)` method which will evaluate the Chebyshev interpolants at `x` given a sampling `F`; if `F` is matrix, the method will compute the Chebyshev interpolants for each of the columns of `F`. The method can thus be used as is for 2D interpolation. But lets suppose `F` is $f(x, y, z)$ sampled on a 3D Chebyshev tensor product grid of dimension $n_1 \times n_2 \times n_3$ (i.e., F is a multidimensional array). Lets now suppose we have $\{x_i\}, \{y_j\}, \{z_k\}$. Let $\{l_{x_i}(x)\}$, $\{l_{y_i}(x)\}, \{l_{z_k}(x)\}$ be the Lagrange polynomials for points $\{x_i\}, \{y_j\}, \{z_k\}$. Then in-order to evaluate

$$p(x, y, z) = \sum_k l_{z_k}(z) \sum_j l_{y_j}(y) \sum_i l_{x_i}(x) f(x_i, y_j, z_k) \tag{3}$$

by

- Evaluating $c_{kj} = \sum_i l_{x_i}(x) f(x_i, y_j, z_k)$ for points $\{y_j\}, \{z_k\}$,

- evaluating $b_k = \sum_j l_{y_j}(y) c_{kj}$ for points $\{z_k\}$
- and then evaluating $\sum_k l_{z_k}(z) b_k$.

To do this first step, I call `bary(x(1),reshape(F,[n1 n2*n3]))`; by reshaping `F`, I can evaluate the Chebyshev polynomials that run along the x-dimension. This can be generalized to higher dimensions. I rewrote the `bary` method to accept multidimensional arrays for `F` and interpolate along the first dimension.

Lets suppose we want to interpolate onto a tensor product grid with points $\{\hat{x}_i\}$, $\{\hat{y}_j\}$, $\{\hat{z}_k\}$, and lets suppose for simplicity Chebyshev grid has N points in each direction while the interpolating set of points has M points in each direction.

There are two ways to evaluate $p(x, y, z)$ on the grid. The first is to use method above for each of the points. This results in a cost of

$$\mathcal{O}(M^3N^3 + M^3N^2 + M^3N). \tag{4}$$

Let suppose instead that we calculate the p(x,y) along the x-dimension for $\{\hat{x}_i\}$, i.e. call `G=bary(x,F)`, where `x` is the vector of points $\{\hat{x}_i\}$ (with my rewritten code). Here `G` will be a multidimensional array of dimension $M \times N \times N$. In this case, `G(i,:,:)` is the set of Chebyshev interpolant evaluations for points $\{y_j\}, \{z_k\}$ calculated at $\hat{x}_i$.

Let's shift the dimensions of `G` with `G = shiftdim(G,1)` i.e. now the dimensions of G are $N \times N \times N$. Since we are interpolating on a grid, for each $\hat{y}_j$ we need to evaluate $p(\hat{x}_i, y, z)$ for each i. This can be achieved by calling `G=bary(y,G)`, where `y` is the vector of points $\{\hat{y}_j\}$. This is repeated for the z-dimension. Putting it altogether, we have

```
G=bary(x,F);
G = shiftdim(G,1);
G=bary(y,G);
G = shiftdim(G,1);
G=bary(z,G);
G = shiftdim(G,1);
```

Our final result will be a $M \times M \times M$ grid evaluated at the interpolating tensor product grid. The work required is

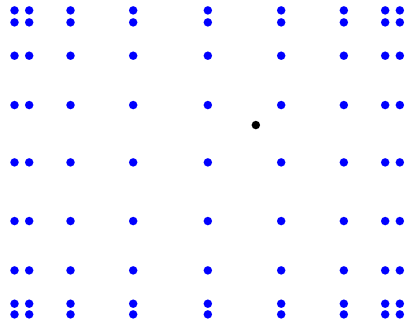
$$\mathcal{O}(MN^3 + M^2N^2 + M^3N). \tag{5}$$

Thus if $N = \mathcal{O}(M)$ we have the first method requires $\mathcal{O}(M^6)$ work, while the second required $\mathcal{O}(M^4)$. This can be seen computationally. In my experiments, I simulate a possible splitting. Assuming a max degree 65 in all dimensions, the first method requires 15 seconds to interpolate onto the grid. The second require 0.18 seconds. The code I used is below.

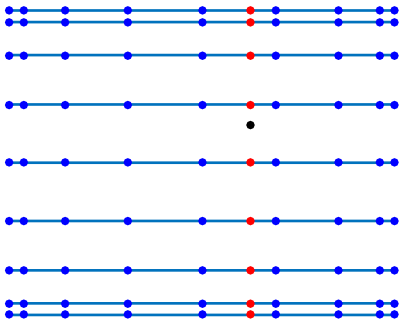
```
clear;

domain = [-1 1];

standard_degs = [3 5 9 17 33 65];

deg_ind = [5 5 5];
degs = standard_degs(deg_ind);

x=chebpts(degs(1),domain);
```

```
y=chebpts(degs(2),domain);
z=chebpts(degs(3),domain);


M = 6;
N=3;
chebpoints = cell(M,1);

chebmatrices = cell(M,2);

chebweights = cell(M,1);



for i=1:M
    chebpoints{i} = chebpts(N);
    chebmatrices{i,1} = diffmat(N,1);
    chebmatrices{i,2} = diffmat(N,2);
    chebweights{i} = chebtech2.barywts(N);
    N = N+(N-1);
end

numb = 65;

%Simulate a splitting
xc = linspace(-1,1,65)';
yc = linspace(-1,1,65)';
yc = yc(yc>1-0.75);
zc = linspace(-1,1,65)';

grid_points = {xc,yc,zc,wc};

[X3C,Y3C,Z3C] = ndgrid(xc,yc,zc);


XP3 = [X3C(:) Y3C(:) Z3C(:)];


[X3,Y3,Z3] = ndgrid(x,y,z);

tic;
G = F3;

h = @(x) 2/(domain(2)-domain(1))*x-(domain(2)+domain(1))/(domain(2)-domain(1));

for k=1:ndims(X3C)
    G = ...
        bary(h(grid_points{k}),G,chebpoints{deg_ind(k)},chebweights{deg_ind(k)});
    G = shiftdim(G,1);
end
toc

F3C = X3C.^2+Y3C.*X3C+Z3C.^3;

max(abs(F3C(:)-G(:)))

FUNS = zeros(length(XP3),1);

tic;
for i=1:size(XP3,1)
```

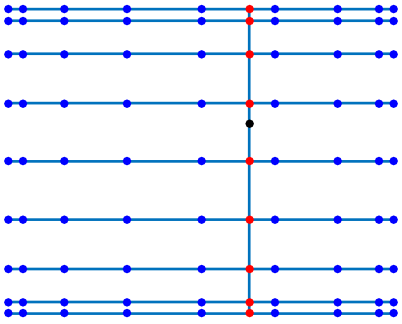
```
    G = F3;
    for k=1:size(XP3,2)
        G = bary(XP3(i,k),G,chebpoints{deg_ind(k)},chebweights{deg_ind(k)});
    end
    FUNS(i) = G;
end
toc

max(abs(FUNS(:)-F3C(:)))
;
```



(a) Chebyshev tensor product grid (in blue), and point to be evaluated (in black).



(b) Chebyshev polynomials evaluated along the x-axis at the red points.



(c) Chebyshev polynomial evaluated along at the desired point.

Figure 2: These pictures illustrate how we can computed multivariate Chebyshev interpolants using a one dimensional interpolation method.

4 Some Initial Results

I have tested the our method on a 2D box with domain $[-1,1]x[-1,1]$. For the function

$$\arctan((x+y)/0.05), \tag{6}$$

we have that our method takes 2.135446 seconds (with a splitting tolerance of 1e-14) while Chebfun2 takes 18 seconds.

For the function

$$\arctan(x/0.05) + \arctan(y/0.05), \tag{7}$$

our method takes 1.4 seconds while Chebfun2 takes 0.1 seconds. Here is my initial though: Our method's complexity depends on how sharp the features of the function are while Chebfun2's complexity depends on the rank of the function.

5 RASPEN Solver

Here I lay out a plan for our to approach the Raspin Solver for a 1D BVP. Here, $\mathrm{b}_0(\nu)$,$\mathrm{b}_1(\nu)$ refer to the left and right interval points of interval(ν) as in our paper. The localsolve(ν,lbc,rbc) method solves the BVP locally on ν with the given boundary conditions; this is a method for leaves. The AlternatingSchwartz method would be called on non-leaves. This algorithm can be generalized to higher dimensions, but some care must be given to the boundaries. I will leave this for later.

Algorithm 5 v=AlternatingSchwartz(ν,lbc,rbc)

if $\mathrm{c}_0(\nu)$ is not a leaf **then**
  $\hat{\mathrm{rbc}}$ = evalf($\mathrm{c}_1(\nu)$,$\mathrm{b}_1(\mathrm{c}_0(\nu))$)
  v0 = AlternatingSchwartz($\mathrm{c}_0(\nu)$,lbc,$\hat{\mathrm{rbc}}$)
else
  v0 = localsolve($\mathrm{c}_0(\nu)$,lbc,rbc)
end if
if $\mathrm{c}_1(\nu)$ is not a leaf **then**
  $\hat{\mathrm{lbc}}$ = evalf($\mathrm{c}_0(\nu)$,$\mathrm{b}_0(\mathrm{c}_1(\nu))$)
  v1 = AlternatingSchwartz($\mathrm{c}_1(\nu)$,$\hat{\mathrm{lbc}}$,rbc)
else
  v1 = localsolve($\mathrm{c}_1(\nu)$,lbc,rbc)
end if
v = [v0;v1];

For Algorithm 6, the residuals of the BVP'S should be included in the nonlinear solve.

Algorithm 6 c=CourseSolve(T,x_c,D_x,D_{xx},u)

sample(T,u)
[ubglob,udglobdx,uglobd2x] = evalf(T,points(T))
rglob = ODE(ubglob,udglobdx,uglobd2x)
u_c = evalf(T,x_c)
r_c = ODE(u_c,$D_x u_c$, $D_{xx} u_c$)
sample(T,rglob)
g = rc + evalf(T,x_c)
c = fsolve(@(s) ODE(s,$D_x s$,$D_{xx} s$)-g)-u_c

We finally have the RASPEN iteration in 7.

Algorithm 7 F=Raspen(u,T,x_c,D_x,D_{xx},lbc,rbc)

$u_{\text{init}} = u$
cf=Chebfun(CourseSolve(T,x_c,D_x,D_{xx},u))
sample(T,u+cf(points(T)))
Alt = AlternatingSchwartz(T,lbc,rbc)
sample(T,Alt)
F = unit-evalf(T,points(T))

6 Computing on a Gird

I completed the code for evaluating the approximation on the grid; this works very fast. The dominate costs for the approximation evaluation on a grid X are:

- Determining the sub-grids of X are in the nodes of the tree,
- and calculating the weights.

For a tree T with leaves $\{\nu_i\}_{i=1}^N$, let grid(ν_i) be the grid of the leaf (in Matlab, this would be stored as a cell array of coordinate vectors). For a forward solver, we would need to evaluate the approximation on $\{\text{grid}(\nu_i)\}_{i=1}^N$. From what I have seen, a single iteration in the RASPEN solve will require hundreds of evaluations. For the approximation of $tan(x+y)$, the method took 6 seconds to evaluate on the grid.

Looking at the profiler though, almost %75 to %80 of the work is determining which points belong to which patches and calculating the weights. These can be pre-calculated. Suppose we have a cell array of the leaves as well as the tree (since I am using the handle class, changes in one data structure will change the other since Matlab uses references).

First, we could use the tree to determine leafpoints(ν) = points(T) $\cap$ domain(ν). In this case, leafpoints(ν) could be stored as a cell array of grids. Let's look at an example of a tree in Figure 3 to see how we might precalculate the weights. In this case, the approximate in terms of the leaves is

$$s_{[a,b]}(x) = w_{\ell_1}(x)s_{[a_1,b_1]}(x) + w_{r_1}(x)w_{\ell_2}(x)s_{[a_{21},b_{21}]}(x) + w_{r_1}(x)w_{r_2}(x)s_{[a_{22},b_{22}]}(x) \tag{8}$$

In this case we would pre-calculate

$$\begin{aligned} w_{\ell_1}(x) &\text{ at leafpoints}(\nu_1), \\ w_{r_1}(x)w_{\ell_2}(x) &\text{ at leafpoints}(\nu_2), \\ \text{and } w_{r_1}(x)w_{r_2}(x) &\text{ at leafpoints}(\nu_3). \end{aligned} \tag{9}$$

For a general tree, we can write a recursive function to do this as seen in Algorithm 8. For a node ν, let weight(ν) be the weight multiplied by the approximate for the node (i.e., in Figure 3 weight(ν_1) = $w_{\ell_2}(x)$). For the root of the tree, we set the weight to the constant function 1. In my code I use a standard weight with parameters to shift and scale; this implies that we can just store the parameters for the weight used at the node of the tree. For each leaf ν_i of the tree T, we set

$$\text{weightvals}(\nu_i)=\text{CalculateWeights}(\text{root}(T),\text{domain}(\nu_i),\text{leafpoints}(\nu_i)). \tag{10}$$

Let leafpointindex(ν_i) be the indices of the points in leafpoints(root(T)). I describe in Algorithm 10 how to evaluate the approximation on the grids of the tree. This can easily be implemented using a parfor loop if needed.

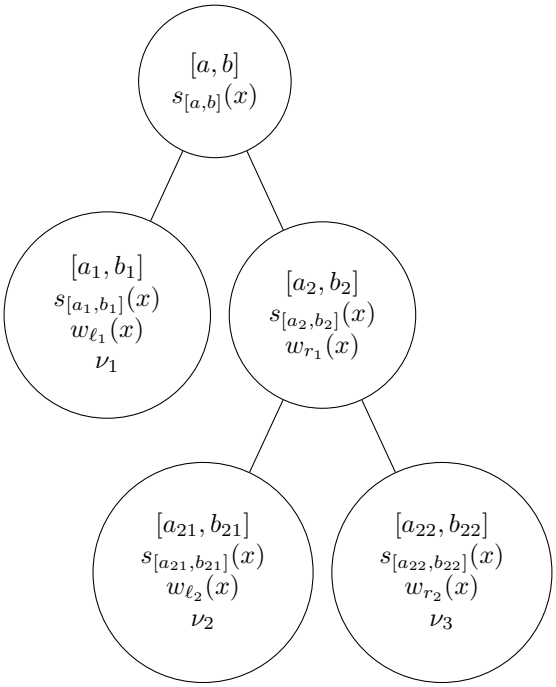


Figure 3: An example of a simple tree with nodes ν_1, ν_2, ν_3, where each node is labeled with its domain, PU approximate and weight (in that order).

Algorithm 8 w=CalculateWeights(ν,dom,X)

if ν is a leaf **then**
  w $\leftarrow$ weights(ν)$|_X$
else if dom $\subseteq$ domain(c$_0$(ν)) **then**
  w $\leftarrow$ weights(ν)$|_X$. $*$ CalculateWeights(c$_0$(ν),dom,X)
else
  w $\leftarrow$ weights(ν)$|_X$. $*$ CalculateWeights(c$_1$(ν),dom,X)
end if

Algorithm 9 F=evalfTreeGrid(T)

for each leaf ν_i of T **do**
  F_i = zeros(length(T), 1)
  F_i(leafpointindex(ν_i)) $\leftarrow$ weightvals(ν_i). $*$ evalf(ν_i, leafpoints(ν_i))
end for
$F = \sum_{i=1}^{N} F_i$.

7 Over determined least squares method

Suppose we have an orthogonal set of tensor product chebyshev polynomials $\{T_k(x,y)\}_{k=1}^{\infty}$ for $[-1,1]\times[-1,1]$ (i.e. the set of polynomials is formed from the tensor product of the polynomials in x and y). Our goal is to approximate the function $f:\Omega\to\mathbb{R}$, with $\Omega\subset[-1,1]\times[-1,1]$. Let

$$G_N = \text{span}\{T_k(x,y) : k < N\} \tag{11}$$

and

$$g_N = \underset{g\in G_N}{\text{argmin}} \|g(x,y) - f(x,y)\|_{L^2(\Omega)}. \tag{12}$$

The first question we might answer is: Does $g_N \to f$ in the $L^2(\Omega)$ norm? I would think so. If there exists an extension $\varepsilon f(x,y) : [-1,1]\times[-1,1]\to\mathbb{R}$ such that $\varepsilon f(x,y) \equiv f(x,y)$ on Ω, then a straight forward convergence argument can be made. I would need to read more about this though.

Though we might could find an exact representation for g_N, it will either be

- cumbersome to compute (we would likely need to compute integrals over Ω),
- unstable to compute (as seen in the Fourier extensions).

Let $X_M\subset\Omega$ be a discrete set of M points. We instead solve for

$$\hat{g}_N = \underset{g\in G_N}{\text{argmin}} \| \ (g-f)|_{X_M} \ \|_2 \tag{13}$$

The first question that comes to mind is: as $M\to\infty$, does $\hat{g}_N \to g_n$ (at least in exact arithmetic)? I would suspect so.

7.1 Simple 2D experiment

As a simple experiment, I try to approximate

$$f(x, y) = \cos((x-1)^2 + (y-1)^2) \tag{14}$$

In the region

$$\Omega = \{(x, y) \in [-1, 1] \times [-1, 1] : (Ax + By + C)/B \geq 0\} \tag{15}$$

i.e. the region in $[-1, 1] \times [-1, 1]$ above the line $Ax + By + C = 0$. As a first test, I set $A = B = 1$, and look at regions for $C \in [0, 2]$. For $C = 0$, this gives the half the square along its diagonal and $C = 2$ gives the whole square. Figure 4 shows the region with $C = 1.2$.

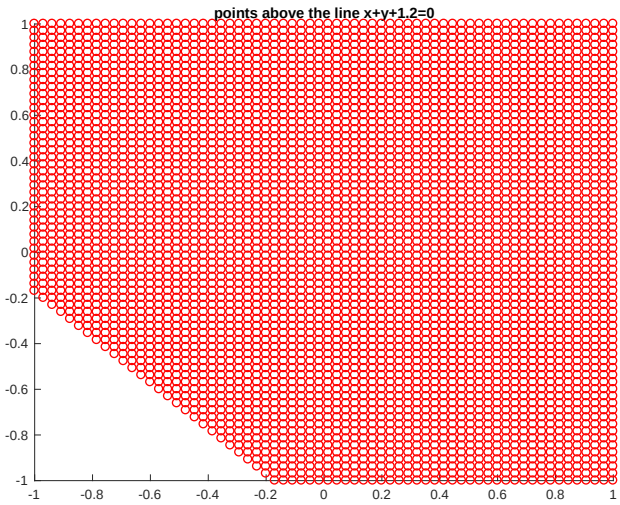


Figure 4: Region with $A = B = 1$, $C = 1.2$.

For this problem, I let

$$G_N = \text{span}\{T_i(x)T_j(y) : i, j \leq N\} \tag{16}$$

where $T_i(x)$ is the i_{th} Chebyshev polynomial and

$$\hat{g}_N = \underset{g \in G_N}{\text{argmin}} \parallel (g - f)|_X \parallel_2 \tag{17}$$

for a discrete set $X \in \Omega$. For my test, I set $N = 33$, and find $\hat{g}_N$ when

- X is the subset of the 66×66 equidistant points in Ω,
- X is the subset of the 66×66 Chebyshev points (i.e. a tensor product) in Ω,
- and X is the subset of the 66×66 Chebyshev points in Ω with 66 equidistant points added to the diagonal boundary.

To measure the accuracy of $\hat{g}_N$ as approximation, we compute the inf norm for the 132×132 equidistant points in Ω. The results can be seen in Figure 5. It would seem:

- If there is less empty space the error is lower,
- we are better off using Chebyshev points instead of equidistant points,
- and there is a benefit to adding points along the boundary.

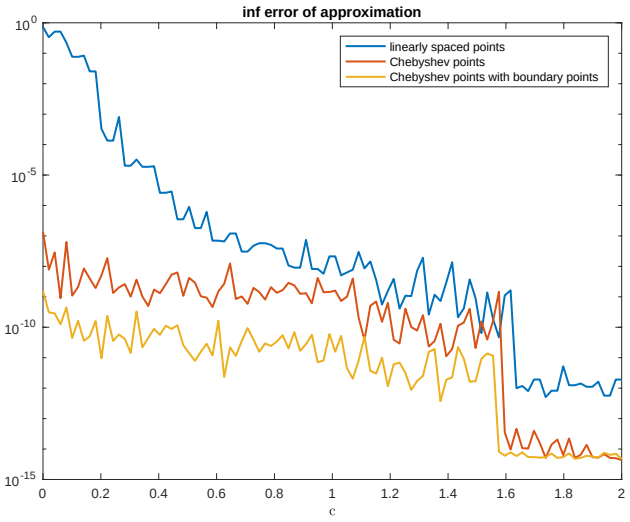


Figure 5: Semi-log plot of inf error for $C \in [0, 2]$.

For a second test, we set

$$\Omega = [-1, 1] \times [-1, 1] \cap B_4([c, 0]) \tag{18}$$

for $c \in [3, 4.5]$. Figure 6 shows the region with $c = 3.5$. We test with for similar collocation points as before, but included linearly spaced points with points added to the boundary. The results can be seen in Figure 7. Here we see that Chebyshev points with the boundary gives the best results.

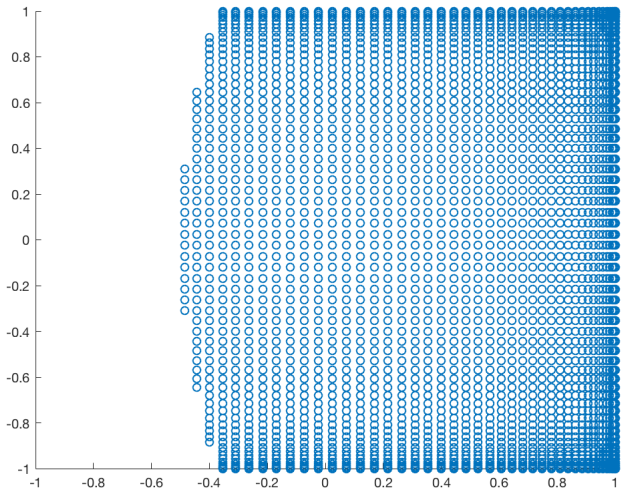


Figure 6: Region with $c = 3.5$.

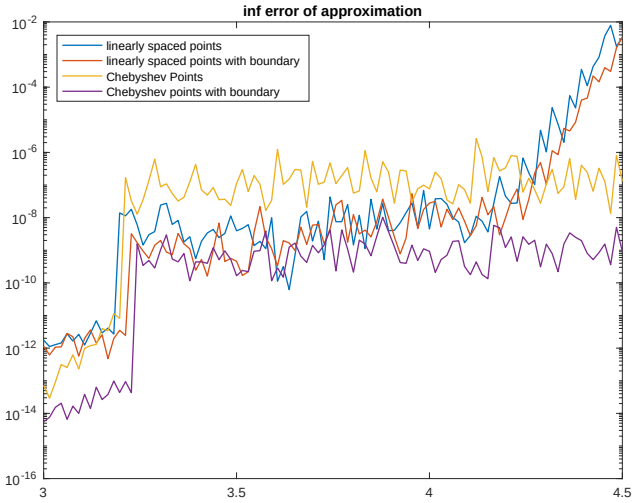


Figure 7: Semi-log plot of inf error for $C \in [0, 2]$.

8 Over determined least squares method with rank deficient matrices

When use a collocation matrix A for a subset Ω on the square $[-1, 1] \times [-1, 1]$ we often find that A is rank deficient. This is unsurprising since we are not specifying function values outside of Ω. Suppose that A has rank r. Then A has a QR decomposition of the form

where

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix} \begin{matrix} \}r \\ \}m-r \end{matrix} \tag{19}$$

with R_{11} nonsingular. In this case, the least squares solution to

$$\underset{x \in \mathbb{C}^n}{\operatorname{argmin}} \|Ax - b\|_2 \tag{20}$$

will have multiple solutions. Let

$$M = \left\{ x | \underset{x \in \mathbb{C}^n}{\operatorname{argmin}} \|Ax - b\|_2 \right\} \tag{21}$$

i.e. the set of x that minimizes the least squares problem. We can find a unique solution if try to find x that satisfies

$$\underset{x \in M}{\operatorname{argmin}} \|Bx\|_2 \tag{22}$$

where B has linearly independent rows. For this problem x has the form

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{23}$$

where x_2 is a free parameter and

$$x_1 = R_{11}^{-1}(Q_1^T b - R_{12} x_2). \tag{24}$$

Let S be the matrix such that

$$R_{11} S = R_{12}. \tag{25}$$

In order to solve (22), we set

$$x_2 = \underset{\hat{x}_2}{\operatorname{argmin}} \left\| B \begin{pmatrix} S \\ -I_{n-r} \end{pmatrix} \hat{x}_2 - B \begin{pmatrix} R_{11}^{-1} Q_1^T b \\ 0 \end{pmatrix} \right\|_2. \tag{26}$$

The solution return by $x = A \backslash b$ in Matlab is the **basic solution**, where x_1 is found by setting $x_2 = 0$.

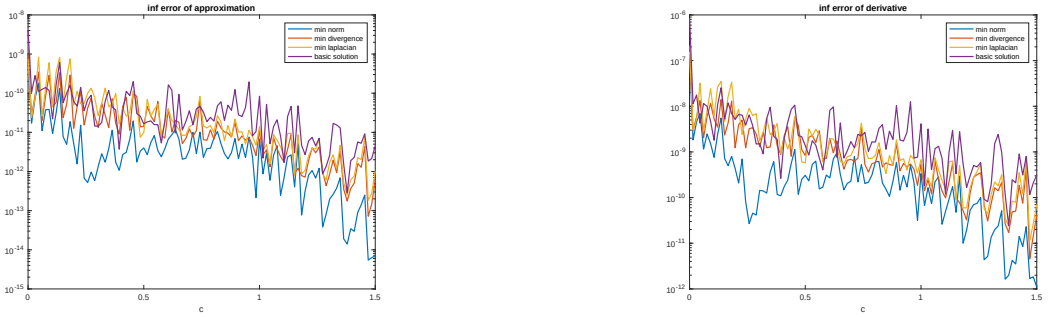
We repeat the experiments we did before on both sets of domains, where we record the errors for:

- $B = I$ (in this case we minimize the norm of the coefficients),
- $B = D_x + D_y$, where D_x, D_y are the differentiation matrices for the coefficients (in this case we minimize the norm of the coefficients of the divergence)
- $B = D_x^2 + D_y^2$ (in this case we minimize the norm of the coefficients of the laplacian),
- and the coefficients returned by the basic solution.

For the region with the triangle cut off defined in (15), we record the errors for the function

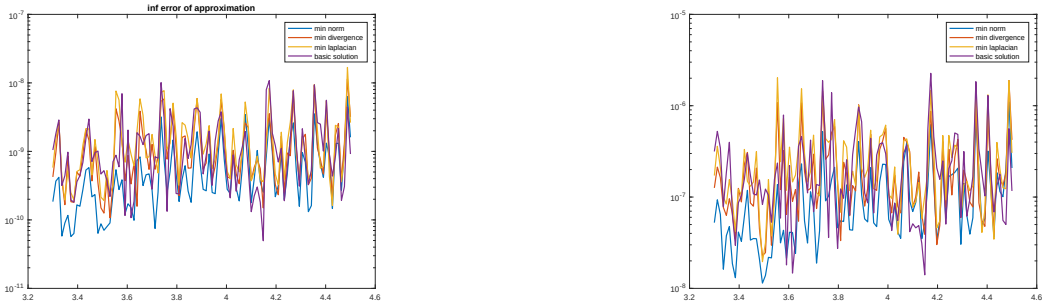
$$f(x, y) = \cos((x-1)^2 + (y-1)^2) \tag{27}$$

and its derivative for $C \in [0, 1.5]$; for C higher than 1.5 the collocation matrix has full rank, making each of the different cases the same. For this experiment we used Chebyshev points with points added at the boundary. I repeat this experiment for the circular defined in (18) for $C \in [3.3, 4.5]$.



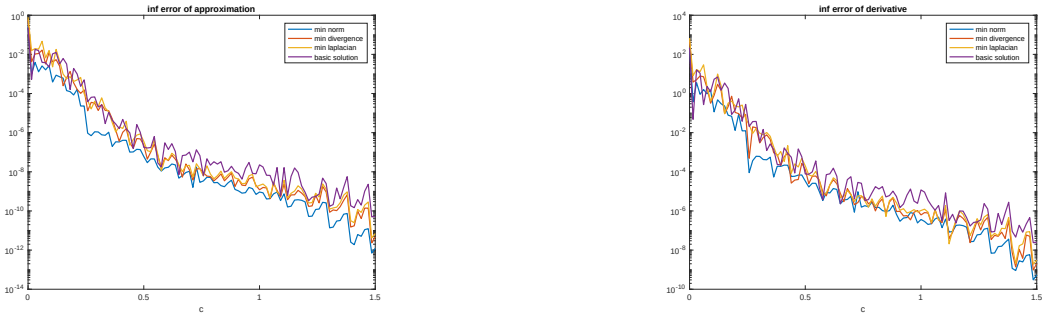
(a) (b)

Figure 8: Infinity errors for the region defined in defined in (15) for $C \in [0, 1.5]$.



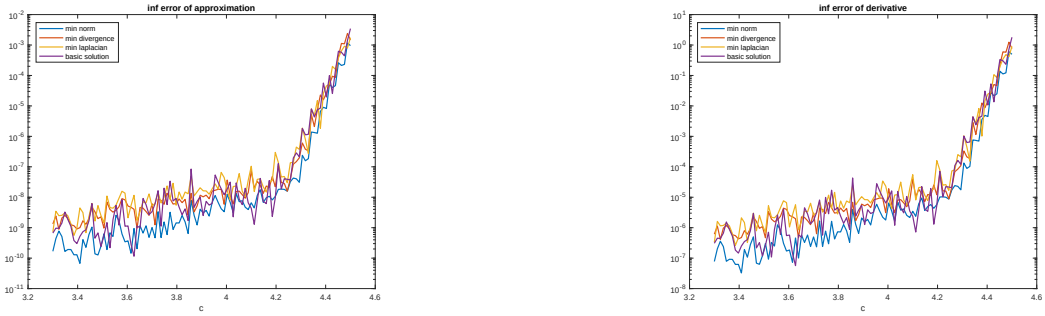
(a) (b)

Figure 9: Infinity errors for the region defined in defined in (18) for $C \in [3.3, 4.5]$.



(a) (b)

Figure 10: Infinity errors for the region defined in defined in (15) for $C \in [0, 1.5]$ using linearly spaced points.



(a) (b)

Figure 11: Infinity errors for the region defined in defined in (18) for $C \in [3.3, 4.5]$ using linearly spaced points.

9 Partition of Unity geometric refinement

Suppose we have a domain $\Omega \in \mathbb{R}^2$ which is not a rectangle, and function $f(x) : \Omega \to \mathbb{R}$. Our goal is to create an adaptive method by overlaying Ω with overlapping squares. We first start by finding a square $[a, b] \times [c, d]$ such that $\Omega \subseteq [a, b] \times [c, d]$. For the adaptive refinement, we use the method we have for the square. We refine by splitting the square along a certain dimension, and recursively repeat this until we have an accurate PU approximation. The PU weights have support on the squares, while the approximations are only defined in the domain itself.

In this case each leaf of the tree is a rectangle. For a leaf ν if $\Omega \cap \text{domain}(\nu) \neq \text{domain}(\nu)$ (i.e. $\Omega \cap \text{domain}(\nu)$ is not square) we use the least square method; otherwise we use the standard Chebyshev method. We define a LSPATCH2D object which is a leafPatch object in Section 1 with the following added properties:

- BoxDomain(ν):the domain of the outer box domain.
- MaxLengths(ν):array of maximum lengths of for BoxDomain(ν).
- InteriorPoints(ν): the set of test points as seen in Figure 12.

For these leaves, $\text{domain}(\nu) = \Omega \cap \text{BoxDomain}(\nu)$. We add an additional method:

- IsGeometricallyRefined(ν):determines if refinement is needed based on the geometry of Ω.

The method can be seen in Algorithm 10. The goal is to have a patch structure that follows the geometry of Ω.

The first challenge for this method is to determine when to drop leaves based on the geometry of Ω. For instance, suppose we split the leaf ν as seen in Figure 13. In this case, the child ν_0 becomes unnecessary since the domain sits entirely in BoxDomain(ν_1). For a tree T, we drop ν_0 be replacing ν with ν_1. The splitting algorithm can be seen in Algorithm 11. Examples of the geometric refinement can be seen in Figure 14.

Algorithm 10 T = IsGeometricallyRefined(ν)

if The lengths of BoxDomain(ν) are less than MaxLengths(ν) **and** at least one point from InteriorPoints(ν) lies in domain(ν) **then**
  ν is geometrically refined.
else
  ν is not geometrically refined.
end if

9.1 Some comments and questions

- I am not attached to the way this algorithm is implemented right now, and believe it can be improved upon. I am thinking that if at least three of the interior points of ν are in Ω, we mark the patch as geometrically refined regardless of the size of the outer box.

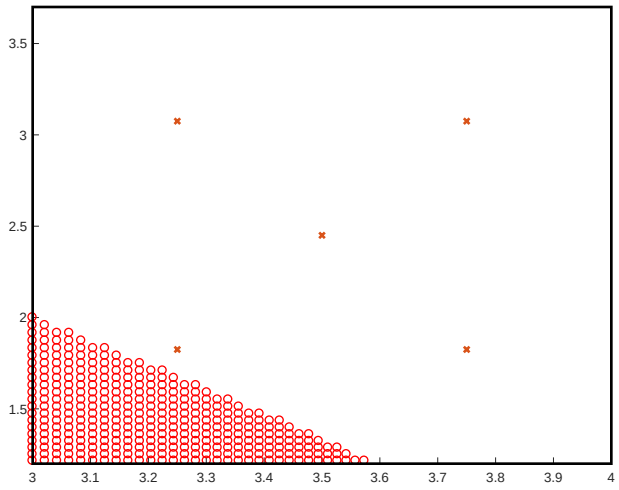


Figure 12: An example of a splitting where a patch ν would be geometrically refined. Here the red circles indicate the domain Ω, and the x's are the points InteriorPoints(ν).

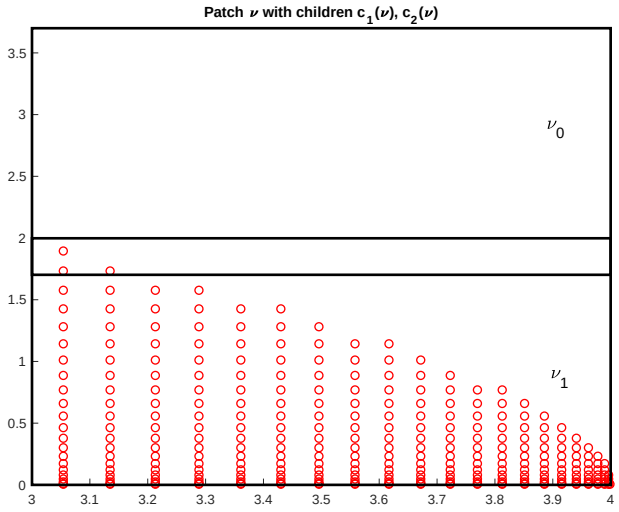


Figure 13: An example of a patch ν and domain Ω where the InteriorPoints(ν) do not lie in Ω.

- We could resize the outerbox of the leaves before we split. While this does avoid the problem of dropping leaves, I don't think implementation will be less complicated than now. The only way I know how to resize the outerbox is to sample the domain and fit a box around that.

Algorithm 11 Child = SplitLeaf(ν,$f(x)$) (method for LSPATCH2D)

if ν is geometrically refined **and** ν can refine $f(x)$ on domain(ν) **then**
  Child = ν
else
  Define domain0 and domain1 as the domains of the rectangular patches split along the dimension of greatest length of BoxDomain(ν).
  for k=0,1 **do**
    if If domaink $\cap\, \Omega$ = domaink **then**
      ν_k:= ChebPatch with domaink
    else
      ν_k:= LSPATCH2D with domaink
    end if
  end for
  if If domain(ν) $\subseteq$ domain($c_0(\nu)$) **then**
    Child = ν_0
  else if If domain(ν) $\subseteq$ domain($c_1(\nu)$) **then**
    Child = ν_1
  else
    Child = A PUPatch with children c_0(Child)=ν_0,c_1(Child):=ν_1
  end if
end if

(a) (b)

Figure 14: An example of a geometric refinement for (a) a heart and (b) a domain whose right boundary is dictated by a cosine curve.

10 Kronecker products and vec in 3D

For two dimensional arrays, we have the well known result

$$\mathrm{vec}(ACB) = (B^T \otimes A)\,\mathrm{vec}(C). \tag{28}$$

This can be useful for forming matrices for 2D tensor product approximations. For instance, suppose that we have a Chebyshev tensor product approximation

$$s(x,y) = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y} c_{ij} T_i(x) T_j(y), \tag{29}$$

along with points $\{x_n\}_{n=1}^{S_x}$, $\{y_n\}_{n=1}^{S_y}$ that we wish to approximate $s(x,y)$ given any set of coefficients $(c_{i,j})$. Define a matrix M^x such that

$$M^x_{i,j} = T_j(x_i). \tag{30}$$

Given a vector of coefficients $d = [d_1, \ldots, d_{N_x}]^T \in \mathbb{C}^{N_x}$ we have

$$(M^x d)_n = \sum_{i=1}^{N_x} d_n T_j(x_n), \tag{31}$$

i.e. M^x will evaluate a Chebyshev approximation at $\{x_n\}_{n=1}^{S_x}$ given a vector of coefficients. We can similarly define a matrix M^y for points $\{y_n\}_{n=1}^{S_y}$. Define $A \in \mathbb{C}^{S_x S_y}$ such that

$$A_{nm} = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y} c_{ij} T_i(x_n) T_j(y_m). \tag{32}$$

Using our matrices M^x and M^y we can deduce that

$$\begin{aligned} A_{nm} &= \sum_{i=1}^{N_x}\sum_{j=1}^{N_y} M^x_{ni} c_{ij} M^y_{mj} \\ &= \sum_{i=1}^{N_x} M^x_{ni} \sum_{j=1}^{N_y} c_{ij} (M^y)^T_{jm} \\ &= \sum_{i=1}^{N_x} M^x_{ni} ((c_{ij})(M^y)^T)_{im} \\ &= (M^x (c_{ij})(M^y)^T)_{nm}. \end{aligned} \tag{33}$$

We thus have from (28) that

$$\operatorname{vec}(A) = (M^y \otimes M^x) \operatorname{vec}((c_{ij})). \tag{34}$$

Suppose now we want to do the same but for the approximation

$$s(x,y,z) = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z} c_{ijk} T_i(x) T_j(y) T_k(z), \tag{35}$$

where we also have points $\{z_n\}_{n=1}^{S_z}$ and interpolation matrix M^z. Suppose we have a set of coefficients $C \in \mathbb{C}^{N_x N_y N_z}$. In MATLAB,

$$\operatorname{vec}(C) = \begin{bmatrix} \operatorname{vec}(C(:,:,1)) \\ \operatorname{vec}(C(:,:,2)) \\ \vdots \\ \operatorname{vec}(C(:,:,N_z)) \end{bmatrix}. \tag{36}$$

Now define $A \in \mathbb{C}^{S_x S_y S_z}$ such that

$$A_{nmp} = \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z} C_{ijk} T_i(x_n) T_j(y_m) T_k(y_p). \tag{37}$$

We first have

$$\begin{aligned} A(:,:,p)_{nm} &= \sum_{i=1}^{N_x}\sum_{j=1}^{N_y}\sum_{k=1}^{N_z} C_{ijk}M^x_{ni}M^y_{mj}M^z_{pk} \\ &= \sum_{k=1}^{N_z} M^z_{pk}\sum_{i=1}^{N_x}\sum_{j=1}^{N_y} C_{ijk}M^x_{ni}M^y_{mj} \\ &= \sum_{k=1}^{N_z} M^z_{pk}(M^x C(:,:,k)(M^y)^T)_{nm}. \end{aligned} \tag{38}$$

Thus from (34) we can infer that

$$\operatorname{vec}(A(:,:,p)) = \sum_{k=1}^{N_z} M^z_{pk}(M^y \otimes M^x)\operatorname{vec}(C(:,:,k)) \tag{39}$$

and can further argure that

$$\begin{aligned} &\sum_{k=1}^{N_z} M^z_{pk}(M^y \otimes M^x)\operatorname{vec}(C(:,:,k)) \\ &= [M^z_{p1}(M^y \otimes M^x)\dots M^z_{pN_z}(M^y \otimes M^x)] \begin{bmatrix} \operatorname{vec}(C(:,:,1)) \\ \vdots \\ \operatorname{vec}(C(:,:,N_z)) \end{bmatrix} \\ &= (M^z(p,:) \otimes (M^y \otimes M^x))\operatorname{vec}(C), \end{aligned} \tag{40}$$

given how MATLAB vectorizes multidimensional arrays. We thus have

$$\operatorname{vec}(A) = \begin{bmatrix} M^z(1,:) \otimes (M^y \otimes M^x) \\ \vdots \\ M^z(N_z,:) \otimes (M^y \otimes M^x) \end{bmatrix} \operatorname{vec}(C) \tag{41}$$

giving us

$$\operatorname{vec}(A) = (M^z \otimes M^y \otimes M^x)\operatorname{vec}(C). \tag{42}$$

11 PU derivatives again

Suppose we have intervals $[0,t]$, $[-t,0]$ and we approximate the function $f(x)$ with

$$s(x) = w_\ell(x)s_\ell(x) + w_r(x)s_r(x) \tag{43}$$

where $\{w_\ell(x), w_r(x)\}$ forms a PU with $\{[0,t],[-t,0]\}$ and $s_\ell(x), s_r(x)$ approximate $f(x)$ on the respective intervals. In our first PU paper in (15) we found

$$\begin{aligned} |f'(x) - s'(x)| \le{}& |w_\ell(x)\left(f'(x) - s'_\ell(x)\right)| + |w_r(x)\left(f'(x) - s'_r(x)\right)| \\ &+ |w'_\ell(x)\left(f(x) - s_\ell(x)\right)| + |w'_r(x)\left(f(x) - s_r(x)\right)|, \end{aligned} \tag{44}$$

implying

$$\begin{aligned} \|f'(x) - s'(x)\|_{L_\infty[-1,1]} \le{}& \max\left(\|f'(x) - s'_\ell(x)\|_{L_\infty[-1,t]}, \|f'(x) - s'_r(x)\|_{L_\infty[-t,1]}\right) \\ &+ \|w'_\ell(x)\|_{L_\infty[-t,t]} \max\left(\|f(x) - s_\ell(x)\|_{L_\infty[-t,t]}, \|f(x) - s_r(x)\|_{L_\infty[-t,t]}\right). \end{aligned} \tag{45}$$

and chose the overlap by balancing the two terms in (45). While examining the Shepard's weights, I found that the derivatives can get quite complicated. My hypothesis is that if we have sufficient overlap that

$$w_\ell(x)s'_\ell(x) + w_r(x)s'_r(x) \tag{46}$$

can sufficiently approximate $f'(x)$.

We have

$$s'(x) = w_\ell(x)s'_\ell(x) + w_r(x)s'_r(x) + w'_\ell(x)s_\ell(x) + w_r(x)s_r(x). \tag{47}$$

Since $w_\ell + w_r = 1$, we have $w'_\ell + w'_r = 0$. Thus

$$s'(x) = w_\ell(x)s'_\ell(x) + w_r(x)s'_r(x) + w'_\ell(x)(s_\ell(x) - f(x)) + w_r(x)(s_r(x) - f(x)). \tag{48}$$

We can thus infer

$$\begin{aligned} &s'(x) - f'(x) \\ &- w'_\ell(x)(s_\ell(x) - f(x)) - w_r(x)(s_r(x) - f(x)) \\ &= w_\ell(x)s'_\ell(x) + w_r(x)s'_r(x) - f'(x). \end{aligned} \tag{49}$$

Letting

$$\hat{s}(x) = w_\ell(x)s'_\ell(x) + w_r(x)s'_r(x), \tag{50}$$

we get

$$\begin{aligned} \|\hat{s}(x) - f(x)\| \leq &\|s'(x) - f'(x)\| + \\ &\|w'_\ell(x)\|_{L_\infty[-t,t]} \max\left(\|f(x) - s_\ell(x)\|_{L_\infty[-t,t]}, \|f(x) - s_r(x)\|_{L_\infty[-t,t]}\right). \end{aligned} \tag{51}$$

We can infer from here that $\hat{s}(x)$ can approximate $f'(x)$.

12 Chebyshev tensor product approximation

12.1 Chebyshev interpolation

We use Chebyshev interpolants for our partition of unity method because they enjoy spectral convergence. Suppose that $f(x)$ is analytic inside a Bernstein ellipse E_ρ (an ellipse with foci ± 1 and semi-major axis $\rho > 1$). We then have Theorem 6 from [5]:

Theorem 12.1. *Suppose $f(z)$ is analytic on and inside the Bernstein ellipse E_ρ. Let p_n be the polynomial that interpolates $f(z)$ at $n+1$ Chebyshev points of the second kind. Then there exists a constant $C > 0$ such that for all $n > 0$,*

$$\|f(x) - p_n(x)\|_\infty \leq C\rho^{-n}.$$

If $f(x)$ is Lipschitz continuous on $[-1, 1]$ then

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x), \quad a_k = \frac{2}{\pi}\int_{-1}^{1} \frac{f(x)T_k(x)}{\sqrt{1-x^2}}dx, \tag{52}$$

where T_k denotes the degree k Chebyshev polynomial (and for a_0, we multiply by $\frac{1}{\pi}$ instead of $\frac{2}{\pi}$). Furthermore if $p_n(x)$ is the nth degree Chebyshev interpolant then

$$f(x) - p_n(x) = \sum_{k=n+1}^{\infty} a_k \left(T_k(x) - T_m(x)\right), \tag{53}$$

where

$$m = [(k + n - 1)(\text{mod } 2n) - (n - 1)], \tag{54}$$

implying we can determine the accuracy of the interpolant $p_n(x)$ by inspecting the Chebyshev coefficients [6]. Chebfun's standardChop method determines the minimum required degree by searching for a plateau of low magnitude coefficients [1]. For example, Figure 15 shows the first 128 coefficients of $f(x) = \exp(\sin(\pi x))$. We see that all coefficients after the first 46 have magnitude less than 10^{-15}. In this case, Chebfun determines the ideal degree to be 50.

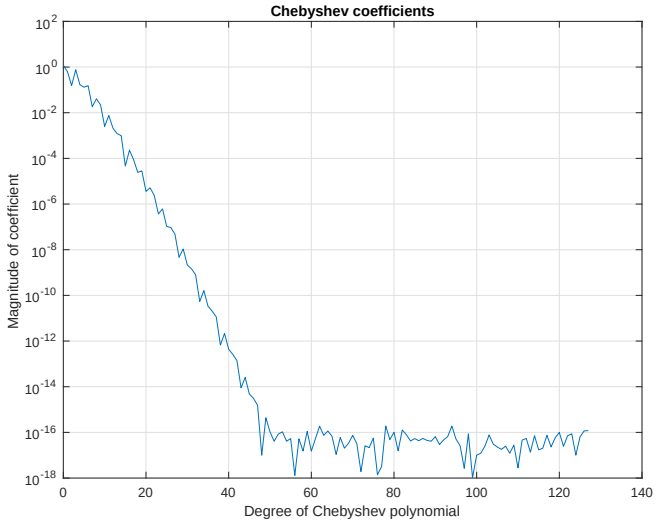


Figure 15: Chebyshev coefficients for $f(x) = \exp(\sin(\pi x))$.

Suppose now we are approximation $f(\mathbf{x}) \in [-1, 1]^n \to \mathbb{C}$. For a multivariate monomial $m = x_1^{k_1} x_2^{k_2} \dots x_n^{k_n}$ we define the $d_{\max}(m) = \max\{k_i\}_{i=1}^n$. Then for a polynomial p the max degree is the max degree of its monomials. We can then similarly argue spectral convergence, as seen in theorem from [4]:

Theorem 12.2. *Let N_{n,h^2} be the open region of the complex plain bounded by the ellipse with foci 0 and n and leftmost point $-h^2$. Then if $f(\mathbf{x})$ is analytic for all $\vec{x}$ with $x_1^2 + x_2^2 + \cdots + x_n^n \in N_{n,h^2}$, then for Chebyshev series p we have*

$$\inf_{d_{max}(p) \le s} \|f - p\|_{[-1,1]^n} = O_\varepsilon(\rho^{-s})$$

where $\rho = h + \sqrt{1 + h^2}$.

12.2 Refinement with hypercubes

Here we will present an argument for how to determine if a multidimensional Chebyshev approximation is refined. For simplicity we will present the argument in 2D. Suppose for $f(\mathbf{x}) \in [-1, 1]^2 \to \mathbb{C}$, we have a Chebyshev tensor product projection

$$p(x, y) = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} T_i(x) T_j(y) a_{ij}. \tag{55}$$

Our goal is to see how we can use Chebfun's `happinessCheck` to determine if $p(x, y)$ can approximate $f(x, y)$. The `happinessCheck` method uses the coefficients of a polynomial to search for a tail end below some tolerance. Throughout the rest of this section, I will assume Assumption 1.

Assumption 1. *If $p(x)$ is the Chebyshev projection of $f(x)$ and $p(x)$ is determined to be happy with tolerance ε using Chebfun's* `happinessCheck`*, then it is.*

Let P_{N_x} be the map that takes a function $f(x)$ to its Chebyshev projection $p_{N_x}(x)$, and let P_{N_y} map $f(y)$ to $p_{N_y}(y)$. Define $< \cdot, \cdot >_x$ be the Chebyshev L^2 inner product with respect to x; similarly define $< \cdot, \cdot >_y$. Here I will show that by examining (a_{ij}), we can determine the accuracy of $p(x, y)$.

Corollary 12.2.1. *If the set of coefficients*

$$d_i = \sum_{j=1}^{N_y} |a_{ij}|$$

is determined to be happy, then $\forall c \in [-1, 1]$ $p(x, c)$ is happy. A similar result holds true if

$$d_j = \sum_{i=1}^{N_x} |a_{ij}|.$$

Proof. We have that

$$p(x, c) = \sum_{i=0}^{N_x} T_i(x) \sum_{j=0}^{N_y} T_j(c) a_{ij} \tag{56}$$

so that the Chebyshev coefficients $\{c_i\}$ of $p(x, c)$ are

$$c_i = \sum_{j=0}^{N_y} T_j(c) a_{ij}. \tag{57}$$

Since $|T_j(x)| \leq 1$ for all $x \in [-1, 1]$ we have

$$|c_i| \leq \sum_{j=0}^{N_y} |T_j(c)||a_{ij}| \leq \sum_{j=0}^{N_y} |a_{ij}| = d_i. \tag{58}$$

Thus if $\{d_i\}$ passes the happiness check, then $p(x, c)$ will as well. □

It should noted that if a polynomial $q(x, c)$ is happy along $y = c$, that does not necessarily mean $q(x, c)$ accurately approximates $f(x, c)$. A simple example would be with

$$f(x, y) = g(y), \tag{59}$$

where $g(y)$ requires a high resolution. If we say sampled on a ten-by-ten grid and produced a Chebyshev polynomial $q(x, y)$, $q(x, c)$ is constant and hence happy even though it is unlikely to approximate $f(x, c)$ well. I take some care to avoid this in the (rough) theorem below.

Rough Theorem 12.1. *Suppose that the set of coefficients $\{d_i\}$ and $\{d_j\}$ from Corollary 12.2.1 are happy. Then $p(x, y)$ approximates $f(x, y)$.*

Proof. Suppose we want to evaluate (x', y'), and we have a Chebyshev tensor grid $\{x_i\}_{i=0}^{N_x} \times \{y_j\}_{j=0}^{N_y}$. The discrete coefficients (a_{ij}) give $p(x_i, y_j) = f(x_i, y_j)$ (i.e. `chebfun2.vals2coeffs(chebfun2.vals2coeffs = vals)`). For the rest of the proof, I will use the discrete Chebyshev projection (where the coefficients are evaluated with FFT).

We have that

$$p(x_i, y) = \sum_{j=0}^{N_y} T_j(y_j) \left\langle T_j(y), \sum_{i=0}^{N_x} T_i(x_i) \left\langle T_i(x), f(x, y) \right\rangle_x \right\rangle_y . \tag{60}$$

We again assume that the projection at the Chebyshev points is equal to the function we are approximating. That is,

$$\sum_{i=0}^{N_x} T_i(x_i) \left\langle T_i(x), f(x, y) \right\rangle_x = f(x_i, y) \tag{61}$$

so we get

$$p(x_i, y) = \sum_{j=0}^{N_y} T_j(y_j) \left\langle T_j(y), f(x_i, y) \right\rangle_y , \tag{62}$$

i.e. the Chebyshev projection of $f(x_i, y)$. Since $p(x_i, y)$ is assumed to be happy, we have that $p(x_i, y)$ approximates $f(x_i, y)$. We can thus use $p(x_1, y)$,$p(x_2, y)$,...,$p(x_{N_x}, y)$ to approximate $f(x, y)$ at (x_1, y'), (x_2, y'),...,(x_{N_x}, y') respectively. Since $p(x, y')$ is presumed to be happy and approximates $f(x, y')$ on the Chebyshev points along $y = y'$, we have that $p(x, y')$ approximates $f(x, y)$. Thus we can assume $p(x', y')$ approximates $f(x', y')$. □

12.3 Refinement take two

Here I attempt to give a formal arguement for the sum method. Suppose that

$$f(x, y) = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} T_i(x) T_j(y) a_{ij}$$

where

$$a_{ij} = \frac{2\hat{\delta}_i}{\pi} \int_{-1}^{1} \frac{1}{\sqrt{1 - x^2}} T_i(x) \frac{2\hat{\delta}_j}{\pi} \int_{-1}^{1} \frac{1}{\sqrt{1 - y^2}} T_j(y) f(x, y) dy dx,$$

such that $\hat{\delta}_i = 1 - \delta_{k0}/2$. Define

$$p(x, y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} T_i(x) T_j(y) a_{ij}. \tag{63}$$

Here we demonstrate how we can look at the coefficients to determine if refinement is needed. Define $c_j(x)$ such that

$$c_j(x) = \frac{2\hat{\delta}_j}{\pi} \int_{-1}^{1} \frac{1}{\sqrt{1 - y^2}} T_j(y) f(x, y) dy. \tag{64}$$

We then have that the Chebyshev approximation of $f(\hat{x}, y)$ along y is $\sum_{j=1}^{N_y} T_j(y)c_j(\hat{x})$. We also have

$$p(x,y) = \sum_{i=0}^{N_x} T_i(x) \int_{-1}^{1} \frac{2\hat{\delta}_i}{\pi} \sum_{j=0}^{N_y} T_j(y)c_j(x)dx \tag{65}$$

implying $p(x, \hat{y})$ is the Chebyshev approximation of $\sum_{j=1}^{N_y} T_j(\hat{y})c_j(x)$. Here we take an idea from a theorem in [3].

Theorem 12.3. *Let $f : [-1, 1]^2 \to \mathbb{R}$ be continuous such that*

$$\log(n)\,\mathrm{osc}(f; [-1, 1]^2; 1/n) \to 0 \text{ as } n \to \infty$$

where

$$\mathrm{osc}(f; [-1, 1]^2; 1/n) := \max_{|\mathbf{x}-\mathbf{y}|<1/n} (|f(\mathbf{x}) - f(\mathbf{y})|; \mathbf{x}, \mathbf{y} \in [-1, 1]^2).$$

Then for $\varepsilon > 0$ there exists N_x, N_y such that

$$p(x,y) = \sum_{i=0}^{N_x} \sum_{j=0}^{N_y} T_i(x)T_j(y)a_{ij}$$

and

$$\|f(x,y) - p(x,y)\|_\infty \leq \varepsilon.$$

There are essentially two main arguments to the theorem:

1. We can find N_y such that $\|f(\cdot, y) - \sum_{i=0}^{N_y} c_j(\cdot)T_j(y)\| \leq \varepsilon/2$,
2. and then find N_x such that $\|p(x, \cdot) - \sum_{i=0}^{N_y} c_j(x)T_j(\cdot)\| < \varepsilon/2$.

This argument of course works by looking at x first and then y. We seek to develop an algorithm testing for these conditions with the coefficients (a_{ij}) produced by the discrete Chebyshev transform with a sampling from the Chebyshev tensor product grid $\{x_i\}_{i=0}^{N_x} \times \{y_j\}_{j=0}^{N_y}$. We have that

$$\|p(x, \cdot) - \sum_{i=0}^{N_y} c_j(x)T_j(\cdot)\| \leq \sum_{i=N_x+1}^{\infty} |\sum_{j=0}^{N_y} T_j(\cdot)a_{ij}| \leq \sum_{i=N_x+1}^{\infty} \sum_{j=0}^{N_y} |a_{ij}|. \tag{66}$$

We can use Chebfun's `standardChop` on $\{\sum_{j=0}^{N_y} |a_{ij}|\}_{i=0}^{N_x}$ to test the second condition. We test the first condition by looking at the series of $f(x_i, y)$ for each Chebyshev point x_i. The coefficients given by the discrete Chebyshev transform will force $p(x, y)$ interpolant $f(x, y)$ on the Chebyshev tensor grid [2]. This means with that for each x_i, for all y_j that $p(x_i, y_j) = f(x_i, y_j)$. We thus have that $p(x_i, y)$ is the discrete Chebyshev transform of $f(x_i, y)$. This implies that we can `standardChop` on $\{\sum_{i=0}^{N_x} |a_{ij}|\}_{j=0}^{N_y}$ to test if each $f(x_i, y)$ is resolved. This argument works if the conditions of x and y are reversed. We thus determine $p(x, y)$ refined $f(x, y)$ if both $\{\sum_{j=0}^{N_y} |a_{ij}|\}_{i=0}^{N_x}$, $\{\sum_{i=0}^{N_x} |a_{ij}|\}_{j=0}^{N_y}$ are determined to be adequate with `standardChop`.

13 Adding and multiplying PU approximations

Suppose we have PU approximations represented by trees T_1 and T_2. My first idea for addition was to add leaf by leaf if they have the same structure, and create a new tree by sampling T_1.evaluate$(x) + T_2$.evaluate(x) otherwise. I first looked at examples in 2D. As a test example, I added

$$f_1(x, y) = \arctan(100(x^2 + y)), \quad f_2(x, y) = \arctan(100(x + y^2)). \tag{67}$$

Suppose T_1,T_2 are the trees representing the PU approximations of $f_1(x, y)$ and $f_2(x, y)$ respectively. I first tried just sampling T_1.evaluate$(x) + T_2$.evaluate(x); this took 60 seconds to do.

The main reason for the time increase is the time needed for barycentric interpolation (this can be seen in the MATLAB profile). To evaluate a n^2 grid with a Chebyshev polynomial of max degree n, $O(n^3)$ work is needed. It becomes clear that we can speed up the addition by anticipating the splitting. My second idea was to take the tree with the most patches, and refine further on that tree. By starting with the tree at T_1 and refining down, we shorten the time to 30 seconds. It appears that the new tree of the sum is a merging of T_1 and T_2, as seen in Figure 16. This is where Algorithm 12 comes from. Using this algorithm, the time of adding T_1 and T_2 is reduced to 5 seconds.

In order to motivate the algorithm, suppose that T_1 is an arbitrary tree and T_2 is a single leaf, with respective approximations $\hat{s}_1(x), \hat{s}_2(x)$. Suppose with PU $\{w_k(x)\}_{k=1}^m$ that

$$\hat{s}_1(x) = \sum_{k=1}^{m} w_k(x) s_k(x). \tag{68}$$

Since $\sum_{k=1}^m w_k(x) = 1$ we have that

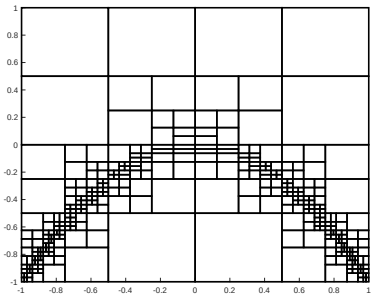
$$\hat{s}_1(x) + \hat{s}_2(x) = \sum_{k=1}^{m} w_k(x)(s_k(x) + \hat{s}_2(x)). \tag{69}$$

We are thus free to add $\hat{s}_2(x)$ to each leaf of T_1. This is the essential idea of the first two cases of Algorithm 12. By tracing down the trees T_1, T_2 and seeing which nodes are enclosed in the leaves of others, we can easily predetermine the splitting that is needed; this is what the third case of Algorithm 12 is doing. This only works though if the trees are split similarly; a key point is that the leaf of one tree contains each leaf of the other tree. For example, in Figure 17, we have an example of a splitting where the leaves of the trees don't contain each other. In this case, we sample the sum via T_1.evaluate$(x) + T_2$.evaluate(x).

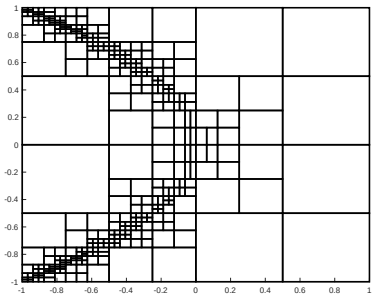
For multiplication, we have Algorithm 13 which is similar to addition. Again, suppose we have $\hat{s}_1(x) = \sum_{k=1}^m w_k(x) s_k(x)$, and a single approximation $\hat{s}_2(x)$. We then have that the product is

$$\hat{s}_2(x) \sum_{k=1}^{m} w_k(x) s_k(x) = \sum_{k=1}^{m} w_k(x) \hat{s}_2(x) s_k(x). \tag{70}$$

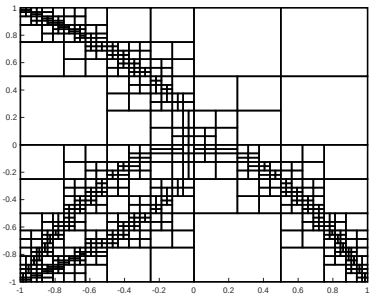
In this case, we could resample the leaves $\hat{s}_2(x) s_k(x)$ for each patch, splitting if necessary.



(a) Zone plot of $f_1(x, y)$

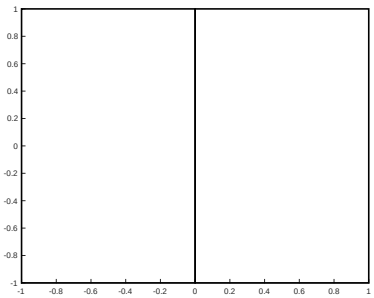


(b) Zone plot of $f_2(x, y)$

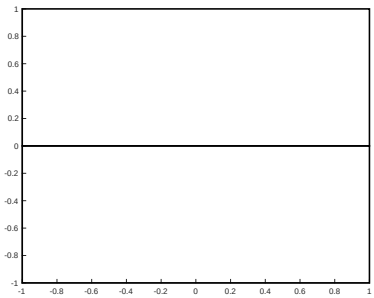


(c) Zone plot of $f_1(x, y) + f_2(x, y)$

Figure 16: Zone plots for $f_1(x, y)$,$f_2(x, y)$ and $f_1(x, y) + f_2(x, y)$.



(a)



(b)

Figure 17: Example of two trees where recursion can not be used.

Algorithm 12 T = add(T_1,T_2)

if T_1 is a leaf **then**
  $T := T_2$
  for each leaf ν of T, add interpolant(T_1) to interpolant(ν)
else if T_2 is a leaf **then**
  $T := T_1$
  for each leaf ν of T, add interpolant(T_2) to interpolant(ν)
else if splittingDim(T_1)=splittingDim(T_2) **then**
  $c_0(T)$ = add($c_0(T_1)$,$c_0(T_2)$)
  $c_1(T)$ = add($c_1(T_1)$,$c_1(T_2)$)
else
  T=refine(max($T_1.n_{\max}, T_2.n_{\max}$),$T_1.t$,$T_1$.root.evalute($x$) + T_2.root.evalute(x))
end if

Algorithm 13 T = multiply(T_1,T_2)

if T_1 is a leaf **then**
  $T := T_2$
  for each leaf ν of T, refine ν with $f(x)$ = interpolant(ν)(x)interpolant(T_1)(x)
else if T_2 is a leaf **then**
  $T := T_1$
  $T := T_2$
  for each leaf ν of T, refine ν with $f(x)$ = interpolant(ν)(x)interpolant(T_2)(x)
else if splittingDim(T_1)=splittingDim(T_2) **then**
  $c_0(T)$ = multiply($c_0(T_1)$,$c_0(T_2)$)
  $c_1(T)$ = multiply($c_1(T_1)$,$c_1(T_2)$)
else
  T=refine(max($T_1.n_{\max}, T_2.n_{\max}$),$T_1.t$,$T_1$.root.evalute($x$)$T_2$.root.evalute($x$))
end if

14 Faster methods for adding and multiplying PU approximations

Here I attempt to increase the speed of the addition and multiplication methods from before. There are two key differences here:

1. Here the new tree is created with recursive splits rather than by copying the subtrees,
2. and we add a case for dealing with trees where the splitting dimension does not match.

If we have a tree with uniform splitting (such as a 1D tree, or a 2D tree where leaves are always split in quarters), then the first four cases of Algorithm 15 will suffice. Suppose that we have trees T_1, T_2 that are first split in x and y respectively, and none of the children of T_2 are split in x. A simple example can be seen in Figure 18. In our method, we would add $\mathrm{c}_0(T_1)$ with T_2 and $\mathrm{c}_1(T_1)$ with T_2. Since T_2 has no splits in x, we can assume that any subsequent splits of T_{add} in Algorithm 15 will match that of T_1 and T_2 (i.e., the dividing hyperplanes will be the same).

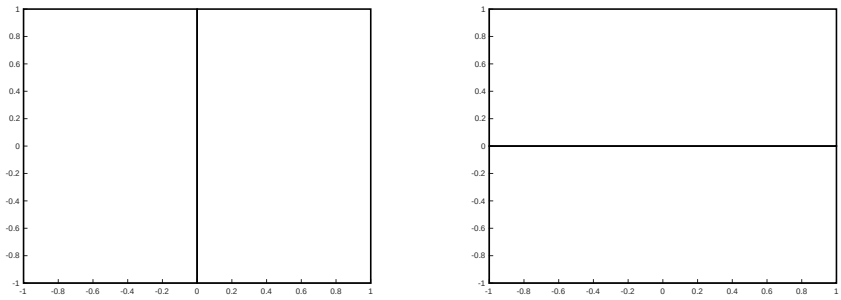
For our trees,

1. we always split in order (x-y in 2D, x-y-z in 3D),
2. and if a tree stops splitting in a dimension at any node, then no child of the node will split in that dimension.

In 2D we can then see that the first 5 cases in Algorithm 15 will suffice. Suppose that we have two trees T_1, T_2 where the splitting dimensions don't match, and the previous step in the recursion the trees split in y. In this case *nextSplit* in Algorithm 15 will be in the x direction. If T_1, T_2 split in x, y respectively, then we know that T_2 does not split in x direction anymore since the x split was skipped (there were two splits in y in a row). Since the T_2 only splits in y, at any other point of the recursion where *nextSplit* is in the x direction we know the fifth case of Algorithm 15 will hold true. I have found this to work well in 3D, and have not found a case where the first 5 cases do not suffice. For example, adding the numerically approximations for

$$f_1(x, y, z) = \arctan(10(x + y) + z), \quad f_2(x, y, z) = \arctan(10(x + z) + y) \tag{71}$$

only took 3 seconds; it takes 15 seconds to build an approximation for $f_1 + f_2$ from scratch. I've included a failsafe step to ensure the algorithm does not fail for good measure.



(a) (b)

Figure 18: Example of two trees with different splittings.

Algorithm 14 T_{add} = add(T_1,T_2,T_{add},*split*)

if T_1 and T_2 are leaves **then**
  interpolant(T_{add})=interpolant(T_1)+interpolant(T_2) on domain(T_{add}).
else if T_1 is a leaf while T_2 is not **then**
  split(T_{add},T_1.splitDim,T_2.overlap)
  $c_0(T_{\text{add}})$=add(T_1,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_2.splitDim)
  $c_1(T_{\text{add}})$=add(T_1,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_2.splitDim)
else if T_1 is not a leaf while T_2 is **then**
  split(T_{add},T_1.splitDim,T_1.overlap)
  $c_0(T_{\text{add}})$=add($c_0(T_1)$,T_2,$c_0(T_{\text{add}})$,T_1.splitDim)
  $c_1(T_{\text{add}})$=add($c_1(T_1)$,T_2,$c_1(T_{\text{add}})$,T_1.splitDim)
else if T_1.splitDim=T_2.splitDim **then**
  split(T_{add},T_1.splitDim,T_1.overlap)
  $c_0(T_{\text{add}})$=add($c_0(T_1)$,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_1.splitDim)
  $c_1(T_{\text{add}})$=add($c_1(T_1)$,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_1.splitDim)
else
  Let *nextSplit* be the next ordered split after *split* from T_1.splitDim, T_2.splitDim
  if *nextSplit*=T_1.splitDim and T_2 does not split in T_1.splitDim anymore **then**
    split(T_{add},T_1.splitDim,T_1.overlap)
    $c_0(T_{\text{add}})$=add($c_0(T_1)$,T_2,$c_0(T_{\text{add}})$,T_1.splitDim)
    $c_1(T_{\text{add}})$=add($c_1(T_1)$,T_2,$c_1(T_{\text{add}})$,T_1.splitDim)
  else if *nextSplit*=T_2.splitDim and T_1 does not split in T_2.splitDim anymore **then**
    split(T_{add},T_1.splitDim,T_1.overlap)
    $c_0(T_{\text{add}})$=add(T_1,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_1.splitDim)
    $c_1(T_{\text{add}})$=add(T_1,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_1.splitDim)
  else
    T_{add}=refine(max($T_1.n_{\max}, T_2.n_{\max}$),$T_1$.overlap,$T_1(x) + T_2(x)$)
  end if
end if

Algorithm 15 T_{add} = multiply(T_1,T_2,T_{add},*split*)

if T_1 and T_2 are leaves **then**
  T_{add}=refine(max($T_1.n_{\max}, T_2.n_{\max}$),$T_1$.overlap,$T_1(x)T_2(x)$) on domain($T_{\text{add}}$).
else if T_1 is a leaf while T_2 is not **then**
  split(T_{add},T_1.splitDim,T_2.overlap)
  $c_0(T_{\text{add}})$=multiply(T_1,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_2.splitDim)
  $c_1(T_{\text{add}})$=multiply(T_1,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_2.splitDim)
else if T_1 is not a leaf while T_2 is **then**
  split(T_{add},T_1.splitDim,T_1.overlap)
  $c_0(T_{\text{add}})$=multiply($c_0(T_1)$,T_2,$c_0(T_{\text{add}})$,T_1.splitDim)
  $c_1(T_{\text{add}})$=multiply($c_1(T_1)$,T_2,$c_1(T_{\text{add}})$,T_1.splitDim)
else if T_1.splitDim=T_2.splitDim **then**
  split(T_{add},T_1.splitDim,T_1.overlap)
  $c_0(T_{\text{add}})$=multiply($c_0(T_1)$,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_1.splitDim)
  $c_1(T_{\text{add}})$=multiply($c_1(T_1)$,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_1.splitDim)
else
  Let *nextSplit* be the next ordered split after *split* from T_1.splitDim, T_2.splitDim
  if *nextSplit*=T_1.splitDim and T_2 does not split in T_1.splitDim anymore **then**
    split(T_{add},T_1.splitDim,T_1.overlap)
    $c_0(T_{\text{add}})$=multiply($c_0(T_1)$,T_2,$c_0(T_{\text{add}})$,T_1.splitDim)
    $c_1(T_{\text{add}})$=multiply($c_1(T_1)$,T_2,$c_1(T_{\text{add}})$,T_1.splitDim)
  else if *nextSplit*=T_2.splitDim and T_1 does not split in T_2.splitDim anymore **then**
    split(T_{add},T_1.splitDim,T_1.overlap)
    $c_0(T_{\text{add}})$=multiply(T_1,$c_0(T_2)$,$c_0(T_{\text{add}})$,T_1.splitDim)
    $c_1(T_{\text{add}})$=multiply(T_1,$c_1(T_2)$,$c_1(T_{\text{add}})$,T_1.splitDim)
  else
    T_{add}=refine(max($T_1.n_{\max}, T_2.n_{\max}$),$T_1$.overlap,$T_1(x)T_2(x)$)
  end if
end if

References

[1] Jared L. Aurentz and Lloyd N. Trefethen. Chopping a Chebyshev series. *ACM Trans. Math. Softw.*, 43(4):33:1–33:21, January 2017.

[2] John C Mason and David C Handscomb. *Chebyshev polynomials*. CRC Press, 2002.

[3] Alvise Sommariva, Marco Vianello, and Renato Zanovello. Adaptive bivariate chebyshev approximation. *Numerical Algorithms*, 38(1-3):79–94, 2005.

[4] L. N. Trefethen. Multivariate polynomial approximation in the hypercube. *ArXiv e-prints*, August 2016.

[5] Lloyd N Trefethen. *Spectral Methods in MATLAB*, volume 10. SIAM, 2000.

[6] Lyod N. Trefethen. *Approximation Theory and Approximation Practice*. Society for Industrial and Applied Mathematics, 2013.