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

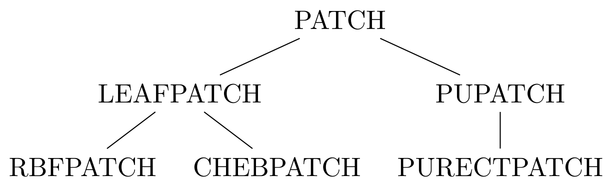


Figure 1: Object structure for the patches.

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

- c$_0$(ν), c$_1$(ν):= two PATCH children
- w$_0$(ν), w$_1$(ν):= 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 $\nu$ is a leaf then
    $p$:=interpolant($\nu$)
    v:= $p(x)$
else
    $v_0, v_1$:=0
    $w_0$:=$\mathrm{w}_0(\nu)$
    $w_1$:=$\mathrm{w}_1(\nu)$
    for $k = 0, 1$ do
        if $x \in$ domain($\mathrm{c}_\mathrm{k}(\nu)$) then
            $v_k$:=eval($\mathrm{c}_\mathrm{k}(\nu)$,$x$)
        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 $\mathrm{c}_\mathrm{k}(\nu)$ is a leaf and unresolved then
        $\mathrm{c}_\mathrm{k}(\nu)$:=splitleaf($\mathrm{c}_\mathrm{k}(\nu)$)
    else
        PUsplit($\mathrm{c}_\mathrm{k}(\nu)$)
    end if
end for
merge($\nu$)
```

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 $N \times N \times M$. 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);
```

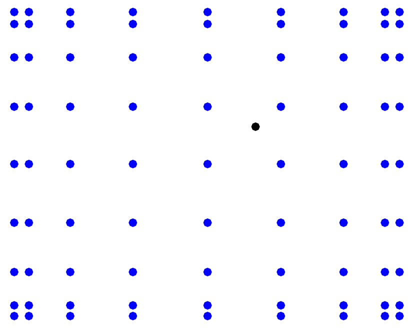
Our final result will be a $M \times M \times M$ grid evaluated at the interpolating tensor product grid. The work required is

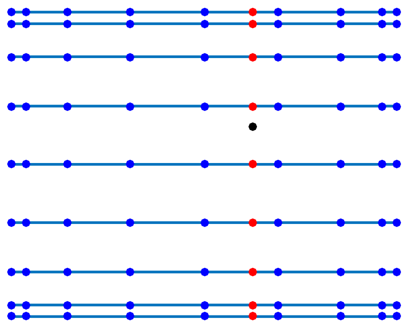
$$\mathcal{O}(MN^3 + M^2N^2 + M^3N). \tag{5}$$

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.

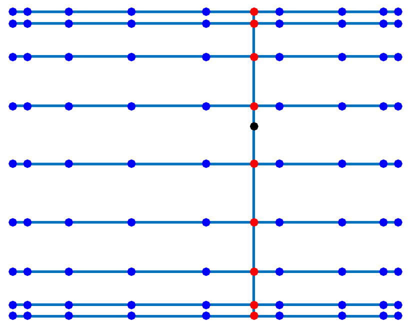

(c) Chebyshev polynomial evaluated along at the desired point.

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
    $\mathrm{r\hat{b}c}$ = evalf($\mathrm{c}_1(\nu)$,$\mathrm{b}_1(\mathrm{c}_0(\nu))$)
    v0 = AlternatingSchwartz($\mathrm{c}_0(\nu)$,lbc,$\mathrm{r\hat{b}c}$)
else
    v0 = localsolve($\mathrm{c}_0(\nu)$,lbc,rbc)
end if
if $\mathrm{c}_1(\nu)$ is not a leaf then
    $\mathrm{l\hat{b}c}$ = evalf($\mathrm{c}_0(\nu)$,$\mathrm{b}_0(\mathrm{c}_1(\nu))$)
    v1 = AlternatingSchwartz($\mathrm{c}_1(\nu)$,$\mathrm{l\hat{b}c}$,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 non-linear 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 leafs 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)∩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

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

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.

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

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

Algorithm 9 F=evalfTreeGrid(T)

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

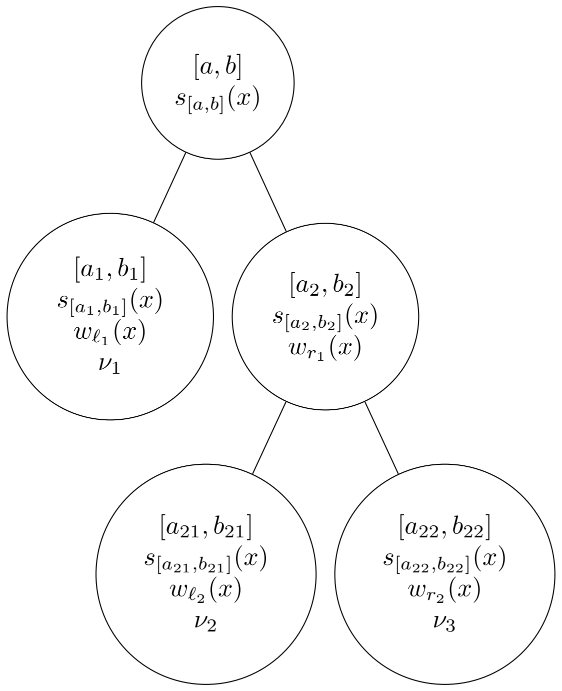


Figure 3: An example of a simple tree with nodes ν_1, ν_2, ν_3, where each node is labeled with its domain, PU approximate and weight (in that order).

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

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

and

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

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

$$\hat{g}_N = \operatorname*{argmin}_{g\in G_N} \| \left.(g-f)\right|_{X_M} \|_2 \tag{13}$$

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 $C=1.2$.

For this problem, I let

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

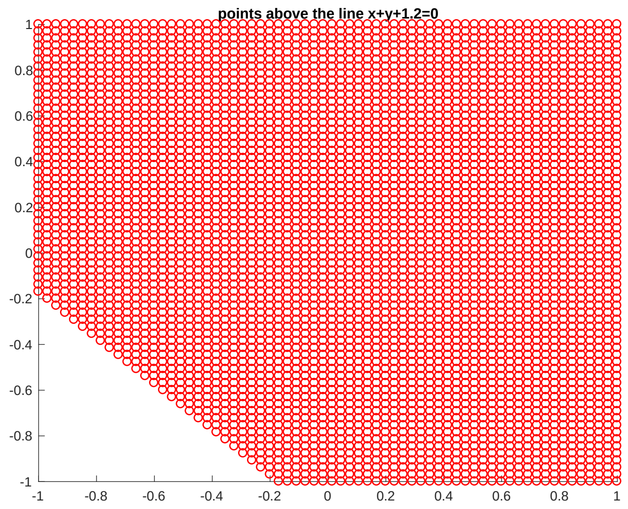


Figure 4: Region with $A = B = 1$, $C = 1.2$.

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

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

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.

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.

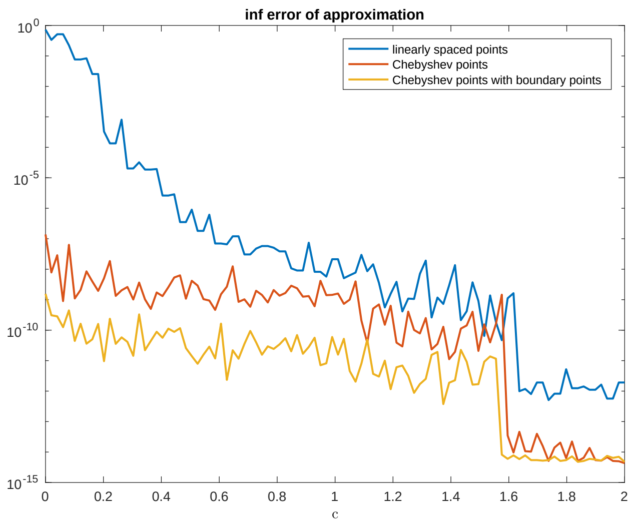


Figure 5: Semi-log plot of inf error for $C \in [0, 2]$.

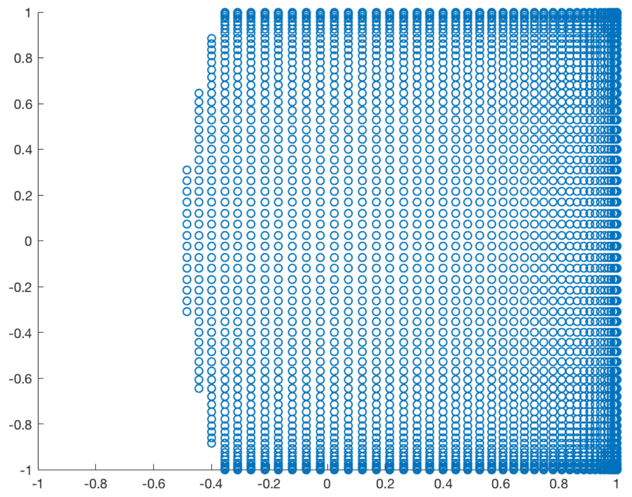


Figure 6: Region with $c = 3.5$.

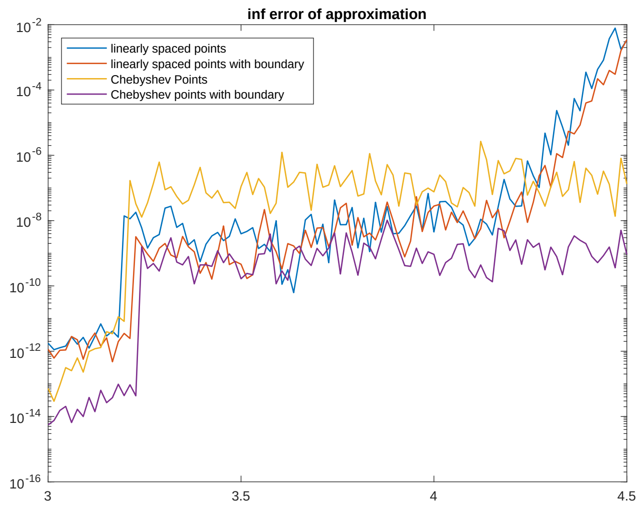


Figure 7: Semi-log plot of inf error for $C \in [0, 2]$.

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

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

will have multiple solutions. Let

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

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

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

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

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

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:

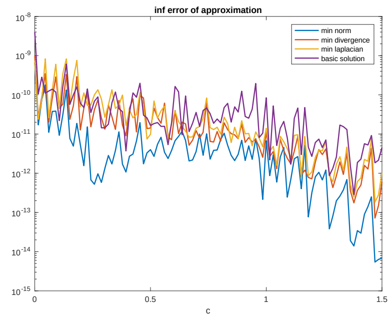
- $B = I$ (in this case we minimize the norm of the coefficients),

- $B = D_x + D_y$, 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.

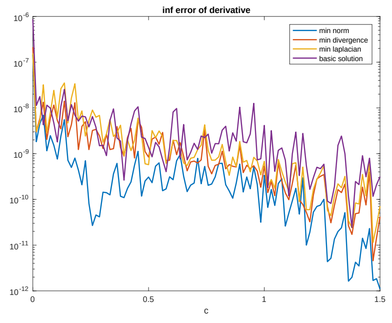
For the region with the triangle cut off defined in (15), we record the errors for the function

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

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 $C \in [3.3, 4.5]$.

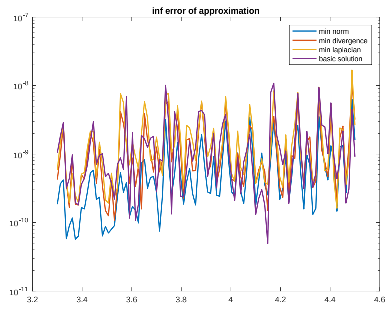


(a)

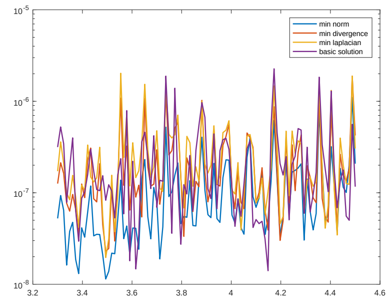


(b)

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

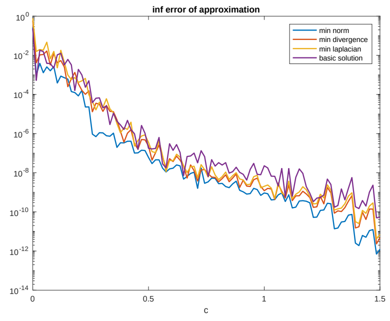


(a)

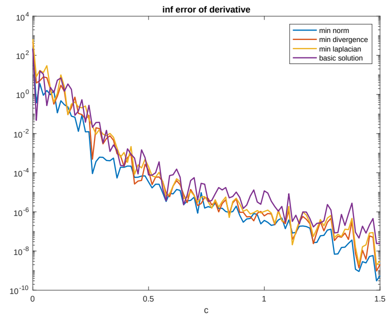


(b)

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

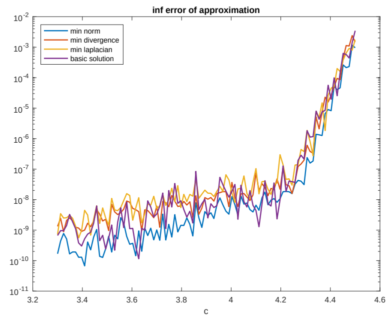


(a)

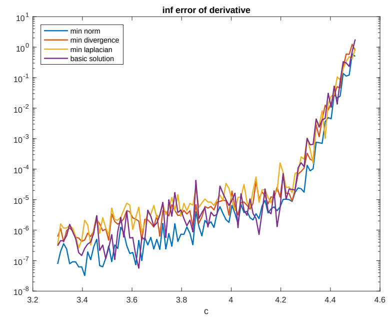


(b)

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($\nu$) are less than MaxLengths($\nu$) and at least one point from InteriorPoints($\nu$) lies in domain($\nu$) then
    $\nu$ is geometrically refined.
else
    $\nu$ is not geometrically refined.
end if
```

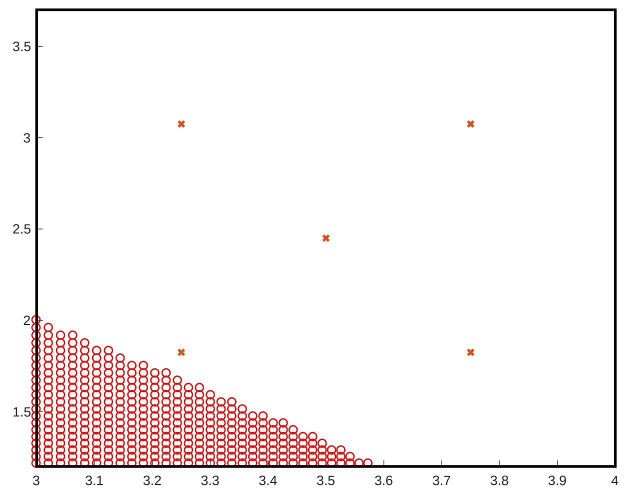


Figure 12: An example of a splitting where a patch ν would be geometrically refined. Here the red circles indicate the domain Ω, and the x's are the points InteriorPoints(ν).

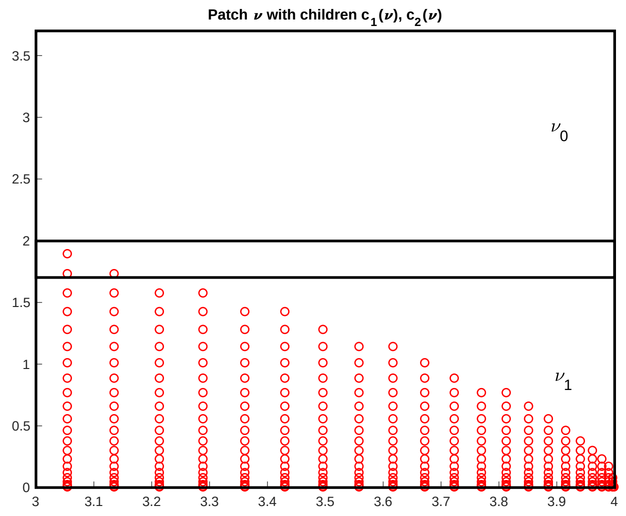


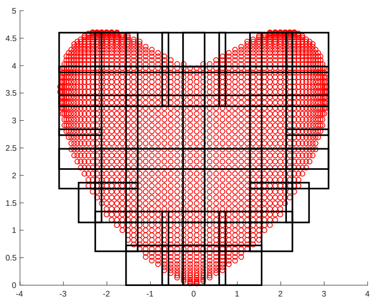
Figure 13: An example of a patch ν and domain Ω where the InteriorPoints(ν) do not lie in Ω.

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.

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

```
if $\nu$ is geometrically refined and $\nu$ can refine $f(x)$ on domain($\nu$) then
  Child = $\nu$
else
  Define domain0 and domain1 as the domains of the rectangular patches
  split along the dimension of greatest length of BoxDomain($\nu$).
  for k=0,1 do
    if If domaink $\cap \Omega$ = domaink then
      $\nu_k$:= ChebPatch with domaink
    else
      $\nu_k$:= LSPATCH2D with domaink
    end if
  end for
  if If domain($\nu$) $\subseteq$ domain($c_0(\nu)$) then
    Child = $\nu_0$
  else if If domain($\nu$) $\subseteq$ domain($c_1(\nu)$) then
    Child = $\nu_1$
  else
    Child = A PUPatch with children $c_0$(Child)=$\nu_0$,$c_1$(Child):=$\nu_1$
  end if
end if
```


(a)

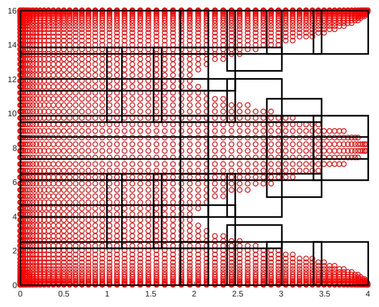

(b)

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.

- 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.

10 Kronecker products and vec in 3D

For two dimensional arrays, we have the well known result

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

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

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

giving us

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

References