

# 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  $\nu$  with properties

- $\text{domain}(\nu) := \text{domain of } \nu$
- $\text{boundaryIndex}(\nu) := \text{boundary index of points of } \nu$
- $\text{length}(\nu) := \text{length of } \nu$
- $\text{dim}(\nu) := \text{dimension of the space domain}(\nu) \text{ lies in}$

with methods

- $\text{points}(\nu) := \text{returns the points of } \nu$
- $\text{evalf}(\nu, x) := \text{evaluates the approximate for } \nu \text{ at } x.$
- $\text{sample}(\nu, f(x)) := \text{samples } f(x) \text{ at the leaves of } \nu$

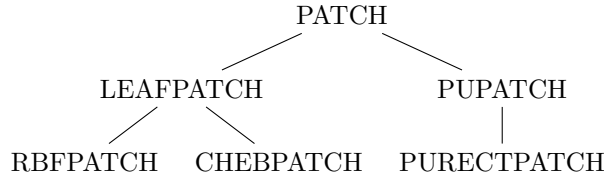


Figure 1: Object structure for the patches.

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

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

The methods would be defined, ignorant of the type of patches the children are. As an example, the  $\text{evalf}(\nu, x)$  method would be defined similarly as before where the leaves are assumed to be type PATCH (this objects would have there own  $\text{evalf}$  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

- $\text{PATCH } \nu_1 = \text{splitleaf}(\text{LEAFPATCH } \nu)$

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

---

**Algorithm 1**  $v = \text{eval}(\nu, x)$ 


---

```

if  $\nu$  is a leaf then
   $p := \text{interpolant}(\nu)$ 
   $v := p(x)$ 
else
   $v_0, v_1 := 0$ 
   $w_0 := w_0(\nu)$ 
   $w_1 := w_1(\nu)$ 
  for  $k = 0, 1$  do
    if  $x \in \text{domain}(c_k(\nu))$  then
       $v_k := \text{eval}(c_k(\nu), x)$ 
    end if
  end for
   $v := w_0(x)v_0 + w_1(x)v_1$ 
end if

```

---



---

**Algorithm 2**  $T = \text{refine}(f(x))$ 


---

```

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

```

---



---

**Algorithm 3**  $\text{PUsplit}(\nu)$ 


---

```

for  $k = 0, 1$  do
  if  $c_k(\nu)$  is a leaf and unresolved then
     $c_k(\nu) := \text{splitleaf}(c_k(\nu))$ 
  else
     $\text{PUsplit}(c_k(\nu))$ 
  end if
end for
 $\text{merge}(\nu)$ 

```

---

## 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  $\text{evalf}(\nu, x, \text{dim}, j)$  which evaluates the derivative of order  $j$  along dimension  $\text{dim}$  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**  $v = \text{evalf}(\nu, x, \text{dim}, j)$

---

```

 $w_0 := w_0(\nu)$ 
 $w_1 := w_1(\nu)$ 
 $v = 0$ 
for  $k = 0, 1$  do
   $v = v + \sum_{i=1}^j \binom{j}{i} \partial_{\text{dim}}^{j-1} w_k(x) \text{evalf}(c_k(\nu), x, \text{dim}, j)$ 
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  $\{l_{x_i}(x)\}, \{l_{y_j}(y)\}$  be the Lagrange polynomials for points  $\{x_i\}, \{y_j\}$ . We then have the interpolating polynomial for a function  $f(x, y)$  is

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

which can be expressed as

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

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  $\{l_{x_i}(x)\}, \{l_{y_j}(y)\}, \{l_{z_k}(z)\}$  be the Lagrange polynomials for points  $\{x_i\}, \{y_j\}, \{z_k\}$ . Then in-order to evaluate

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

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

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

Let suppose instead that we calculate the  $p(x, y)$  along the  $x$ -dimension for  $\{\hat{x}_i\}$ , i.e. call  $G = \text{bary}(x, F)$ , 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  $G = \text{shiftdim}(G, 1)$  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  $G = \text{bary}(y, G)$ , where  $y$  is the vector of points  $\{\hat{y}_j\}$ . This is repeated for the  $z$ -dimension. Putting it altogether, we have

```

1 G=bary(x,F);
2 G = shiftdim(G,1);
3 G=bary(y,G);
4 G = shiftdim(G,1);
5 G=bary(z,G);
6 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

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

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.

```

1 clear;
2
3 domain = [-1 1];
4
5 standard.degs = [3 5 9 17 33 65];
6
7 deg_ind = [5 5 5];
8 degs = standard.degs(deg_ind);
9
10 x=chebpts(degs(1),domain);
11 y=chebpts(degs(2),domain);
12 z=chebpts(degs(3),domain);
13
14
15 M = 6;
16 N=3;
17 chebpoints = cell(M,1);
18
19 chebmatrices = cell(M,2);
20
21 chebweights = cell(M,1);
22
23
24
25 for i=1:M
26     chebpoints{i} = chebpts(N);
27     chebmatrices{i,1} = diffmat(N,1);
28     chebmatrices{i,2} = diffmat(N,2);
29     chebweights{i} = chebtech2.barywts(N);
30     N = N+(N-1);
31 end
32
33 numb = 65;
34
35 %Simulate a splitting
36 xc = linspace(-1,1,65)';
37 yc = linspace(-1,1,65)';
38 yc = yc(yc>1-0.75);
39 zc = linspace(-1,1,65)';
40
41 gridpoints = {xc,yc,zc,wc};
42
43 [X3C,Y3C,Z3C] = ndgrid(xc,yc,zc);
44

```

```

45
46 XP3 = [X3C(:) Y3C(:) Z3C(:)];
47
48
49 [X3,Y3,Z3] = ndgrid(x,y,z);
50
51 tic;
52 G = F3;
53
54 h = @(x) ...
      2/(domain(2)-domain(1))*x-(domain(2)+domain(1))/(domain(2)-domain(1));
55
56 for k=1:ndims(X3C)
57     G = ...
          bary(h(gridpoints{k}),G,chebpoints{deg_ind(k)},chebweights{deg_ind(k)});
58     G = shiftdim(G,1);
59 end
60 toc
61
62 F3C = X3C.^2+Y3C.*X3C+Z3C.^3;
63
64 max(abs(F3C(:)-G(:)))
65
66 FUNS = zeros(length(XP3),1);
67
68 tic;
69 for i=1:size(XP3,1)
70     G = F3;
71     for k=1:size(XP3,2)
72         G = ...
            bary(XP3(i,k),G,chebpoints{deg_ind(k)},chebweights{deg_ind(k)});
73     end
74     FUNS(i) = G;
75 end
76 toc
77
78 max(abs(FUNS(:)-F3C(:)))
79 ;

```

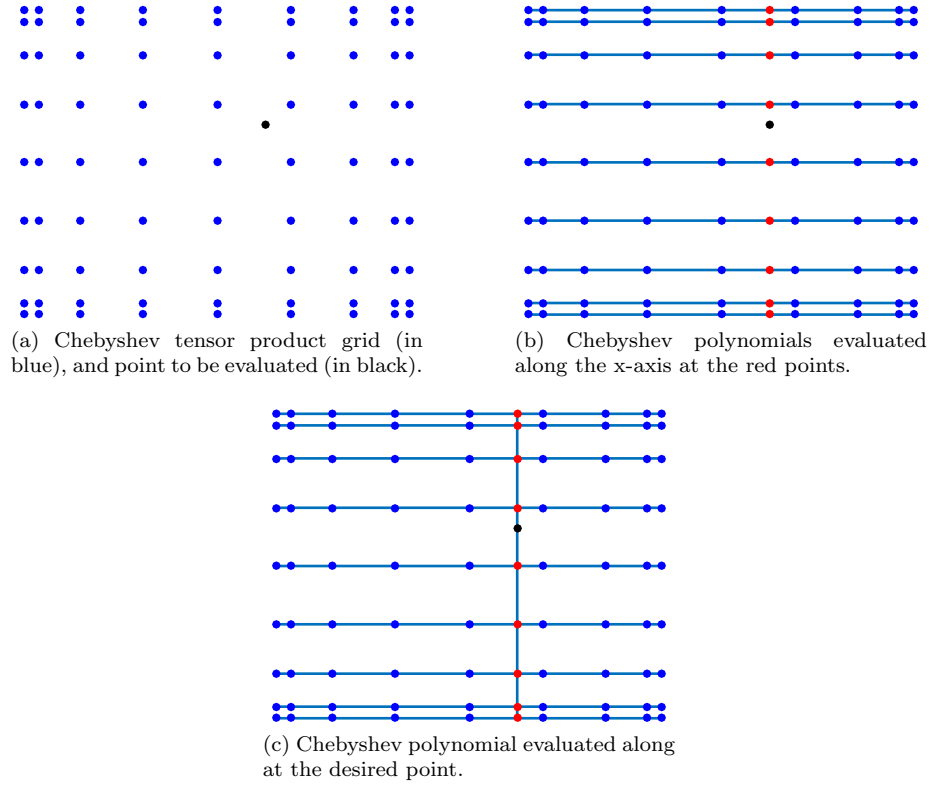


Figure 2: These pictures illustrate how we can computed multivariate Chebyshev interpolants using a one dimensional interpolation method.

## 4 Some Initial Results

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

$$\arctan((x + y)/0.05), \quad (6)$$

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

For the function

$$\arctan(x/0.05) + \arctan(y/0.05), \quad (7)$$

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,  $b_0(\nu), b_1(\nu)$  refer to the left and right interval points of  $\text{interval}(\nu)$  as in our paper. The  $\text{localsolve}(\nu, \text{lbc}, \text{rbc})$  method solves the BVP locally on  $\nu$  with the given boundary conditions; this is a method for leaves. The  $\text{AlternatingSchwartz}$  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**  $v = \text{AlternatingSchwartz}(\nu, \text{lbc}, \text{rbc})$

---

```

if  $c_0(\nu)$  is not a leaf then
   $\hat{\text{rbc}} = \text{evalf}(c_1(\nu), b_1(c_0(\nu)))$ 
   $v_0 = \text{AlternatingSchwartz}(c_0(\nu), \text{lbc}, \hat{\text{rbc}})$ 
else
   $v_0 = \text{localsolve}(c_0(\nu), \text{lbc}, \text{rbc})$ 
end if
if  $c_1(\nu)$  is not a leaf then
   $\hat{\text{lbc}} = \text{evalf}(c_0(\nu), b_0(c_1(\nu)))$ 
   $v_1 = \text{AlternatingSchwartz}(c_1(\nu), \hat{\text{lbc}}, \text{rbc})$ 
else
   $v_1 = \text{localsolve}(c_1(\nu), \text{lbc}, \text{rbc})$ 
end if
 $v = [v_0; v_1];$ 

```

---

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

---

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

---

```

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

```

---

We finally have the RASPEN iteration in 7.



---

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

---

```

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

```

---

## 6 Computing on a Grid

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  $\text{grid}(\nu_i)$  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  $\text{leafpoints}(\nu) = \text{points}(T) \cap \text{domain}(\nu)$ . In this case,  $\text{leafpoints}(\nu)$  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

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

In this case we would pre-calculate

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

For a general tree, we can write a recursive function to do this as seen in Algorithm 8. For a node  $\nu$ , let  $\text{weight}(\nu)$  be the weight multiplied by the approximate for the node (i.e., in Figure 3  $\text{weight}(\nu_1) = w_{\ell_2}(x)$ ). 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  $\nu_i$  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  $\text{leafpointindex}(\nu_i)$  be the indices of the points in  $\text{leafpoints}(\text{root}(T))$ . I describe in Algorithm 9 how to evaluate the approximation on the grids of the tree. This can easily be implemented using a parfor loop if needed.

---

**Algorithm 8**  $w = \text{CalculateWeights}(\nu, \text{dom}, X)$

---

```

if  $\nu$  is a leaf then
   $w \leftarrow \text{weights}(\nu)|_X$ 
else if  $\text{dom} \subseteq \text{domain}(c_0(\nu))$  then
   $w \leftarrow \text{weights}(\nu)|_X \cdot * \text{CalculateWeights}(c_0(\nu), \text{dom}, X)$ 
else
   $w \leftarrow \text{weights}(\nu)|_X \cdot * \text{CalculateWeights}(c_1(\nu), \text{dom}, X)$ 
end if

```

---



---

**Algorithm 9**  $F = \text{evalfTreeGrid}(T)$

---

```

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

```

---

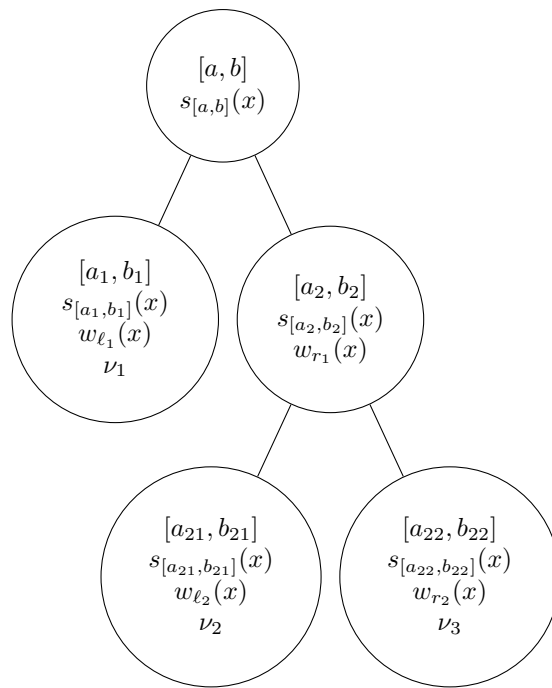


Figure 3: An example of a simple tree with nodes  $\nu_1, \nu_2, \nu_3$ , where each node is labeled with its domain, PU approximate and weight (in that order).

## 7 Over determined least squares method

Suppose we have an orthogonal set of tensor product chebyshev polynomials  $\{T_k(x, y)\}_k^{\text{inf}}$  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  $f : \Omega \rightarrow \mathbb{R}$ , with  $\Omega \subset [-1, 1] \times [-1, 1]$ . Let

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

and

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

The first question we might answer is: Does  $g_N \rightarrow f$  in the  $L^2(\Omega)$  norm? I would think so. If there exists an extension  $\varepsilon f(x, y) : [-1, 1] \times [-1, 1] \rightarrow \mathbb{R}$  such that  $\varepsilon f(x, y) \equiv f(x, y)$  on  $\Omega$ , 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  $\Omega$ ),
- 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

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

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