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

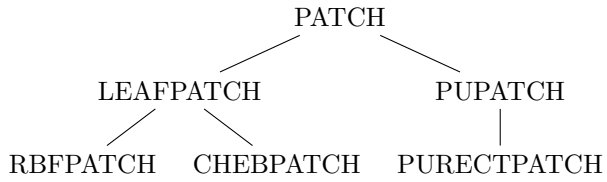


Figure 1: Object structure for the patches.

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 $N \times N \times M$. 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

```
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);
```

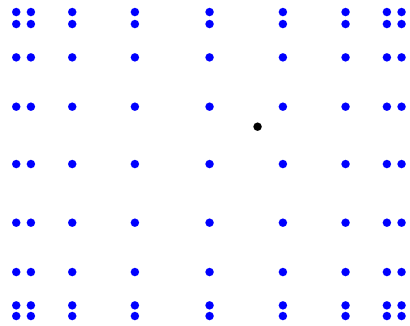
Our final result will be a $M \times M \times M$ grid evaluated at the interpolating tensor product grid. The work required is

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

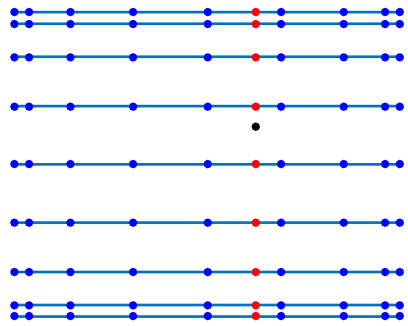
Thus if $N = \mathcal{O}(M)$ we have the first method requires $\mathcal{O}(M^6)$ work, while the second required $\mathcal{O}(M^4)$. This can be seen computationally. In my experiments, I simulate a possible splitting. Assuming a max degree 65 in all dimensions, the first method requires 15 seconds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

```

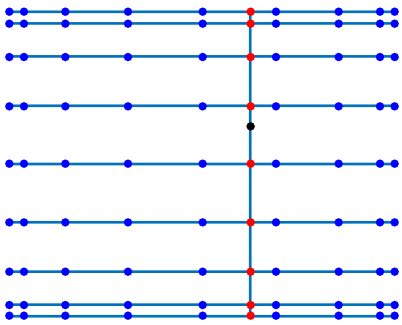
```
45
46  XP3 = [X3C(:) Y3C(:) Z3C(:)];
47
48
49  [X3,Y3,Z3] = ndgrid(x,y,z);
50
51  tic;
52  G = F3;
53
54  h = @(x) ...
        2/(domain(2)-domain(1))*x-(domain(2)+domain(1))/(domain(2)-domain(1));
55
56  for k=1:ndims(X3C)
57      G = ...
            bary(h(grid_points{k}),G,chebpoints{deg_ind(k)},chebweights{deg_ind(k)});
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  ;
```



(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, $b_0(\nu)$,$b_1(\nu)$ 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 $c_0(\nu)$ is not a leaf **then**
  $\hat{rbc}$ = evalf($c_1(\nu)$,$b_1(c_0(\nu))$)
  v0 = AlternatingSchwartz($c_0(\nu)$,lbc,$\hat{rbc}$)
else
  v0 = localsolve($c_0(\nu)$,lbc,rbc)
end if
if $c_1(\nu)$ is not a leaf **then**
  $\hat{lbc}$ = evalf($c_0(\nu)$,$b_0(c_1(\nu))$)
  v1 = AlternatingSchwartz($c_1(\nu)$,$\hat{lbc}$,rbc)
else
  v1 = localsolve($c_1(\nu)$,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)$\cap$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

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

Let leafpointindex(ν_i) be the indices of the points in leafpoints(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=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
$F = \sum_{i=1}^{N} F_i$.

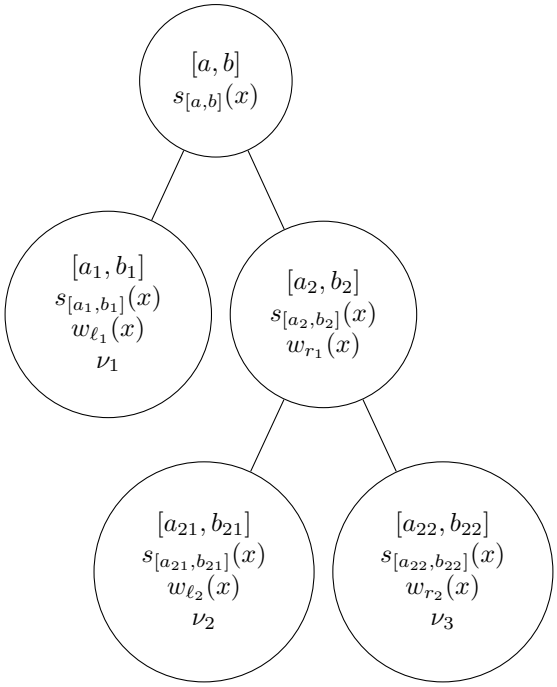


Figure 3: An example of a simple tree with nodes ν_1, ν_2, ν_3, where each node is labeled with its domain, PU approximate and weight (in that order).