Software Implementation of the Partition of Unity Method

Dolomites Research Notes on Approximation

Special Issue SA2022 – *Software for Approximation 2022*, Volume 15 · 2022 · Pages 35–46

Software Implementation of the Partition of Unity Method*

Roberto Cavoretto [a,c] · Alessandra De Rossi [a,c] · Sandro Lancellotti [a,c] · Emma Perracchione [b,c]

Abstract

We present a software for efficiently interpolating via kernel bases large datasets in a multivariate setting. It is based on the partition of unity method, and hence on splitting the given (potentially huge) interpolation problem into many small ones. To reach the widest audience among the potential users, we propose both a MATLAB and a Python implementation. We discuss the algorithm details, and hence the paper results in a step-by-step user-friendly tutorial.

1 Introduction

Kernel-based schemes are popular methods used in many applied fields, such as scattered data interpolation, regression and machine learning. Their success both in approximation theory [27] and artificial intelligence [24] is due to the fact that they are meshfree and easy to implement in *any* dimension. For a complete review on the topic, we refer the reader to e.g. [2].

One of the main disadvantages of kernel-based interpolation schemes is that the collocation matrices, generated by imposing the interpolation conditions, are typically full and hence, their complexity cost is not affordable when a *large* number of data is available. In this setting the so-called Partition of Unity (PU) scheme is nowadays a well-established and efficient kernel-based interpolation method. First introduced in the mid 1990s in [1], the PU scheme produces a global approximant by gluing together, via the use of compactly supported weights, *many* local fits [28]. Such a scheme is also rather popular for researchers working on local collocation schemes for PDEs; refer e.g. to [3, 16, 20, 23].

The PU method organizes the initial set of scattered data, that lay on a multivariate domain, into several subdomains, also known as patches. Then, for each of those patches it solves a *small* interpolation problem. A key step in its implementation is thus the one of efficiently distributing the scattered data into the different patches. The reader can find a MATLAB implementation of the PU scheme, based on the so-called kd-tree partitioning data structures, in [9]. Unfortunately, such an algorithm makes use of files with dll extension (also known as mex-files) that cannot be executed in recent MATLAB releases. With this motivation, we propose an effective implementation of the PU scheme based on what we call the integer-based routines. Although the theory behind such partitioning data procedures is not new and can be thought as a multivariate extension of the algorithms proposed in [4], we here discuss for the first time its detailed implementation. Moreover, motivated by the growing interest of the kernel community towards Python packages for machine learning, we also develop a Python implementation of the PU scheme. Both the MATLAB and Python codes are available at `https://github.com/sandro-lancellotti/PU`

The paper is organized as follows. In Section 2, we briefly review the basics of kernel-based schemes. Sections 3 and 4 provide the details on the PU scheme and its MATLAB and Python implementation, respectively. In Section 5 some experiments with the two implemented algorithms are proposed, and our conclusions are offered in Section 6.

2 Preliminaries

Given $X = \{\boldsymbol{x}_i, i = 1, \dots, n\} \subseteq \Omega$ a set of distinct data, arbitrarily distributed on a domain $\Omega \subseteq \mathbb{R}^d$, with an associated set $F = \{f_i = f(\boldsymbol{x}_i), i = 1, \dots, n\}$ of data values, which are obtained by sampling some (unknown) function $f : \Omega \longrightarrow \mathbb{R}$ at the nodes $\boldsymbol{x}_i$, the scattered data interpolation problem consists in finding a function $P_f : \Omega \longrightarrow \mathbb{R}$ such that it matches the measurements at the corresponding locations, i.e.:

$$P_f(\boldsymbol{x}_i) = f_i, \quad i = 1, \dots, n.$$

We now suppose to have a univariate function $\phi : [0, \infty) \to \mathbb{R}$ (known as Radial Basis Function (RBF), which might depend on the so-called shape parameter ε) that provides, for $\boldsymbol{x}, \boldsymbol{z} \in \Omega$, the real symmetric strictly positive definite kernel (see e.g. [27])

$$\kappa(\boldsymbol{x}, \boldsymbol{z}) = \phi(||\boldsymbol{x} - \boldsymbol{z}||_2) := \phi(r).$$

*The preface of this special issue to which the article belongs is given in [5].

[a]Dipartimento di Matematica "Giuseppe Peano", Università di Torino, Via Carlo Alberto, 10, 10123 Torino, Italy
[b]Dipartimento di Scienze Matematiche "Giuseppe Luigi Lagrange", Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129 Torino, Italy
[c]Member of the INdAM Research group GNCS

The selection of the shape parameter strongly affects the accuracy of the fit. It is then a critical issue that will not be discussed in this paper; we refer the reader to e.g. [11, 17, 15, 22] for discussions on the topic. The kernel-based interpolant P_f can be written as

$$P_f(\boldsymbol{x}) = \sum_{k=1}^{n} c_k \kappa(\boldsymbol{x}, \boldsymbol{x}_k), \quad \boldsymbol{x} \in \Omega,$$

whose coefficients are the solution of [10, 14]

$$\mathsf{K}\boldsymbol{c} = \boldsymbol{f},$$

where $\boldsymbol{c} = (c_1, \ldots, c_n)^\intercal$, $\boldsymbol{f} = (f_1, \ldots, f_n)^\intercal$, and

$$\mathsf{K}_{ik} = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_k), \quad i, k = 1, \ldots, n.$$

We suppose that κ is a symmetric and strictly positive definite kernel, and hence the system has exactly one solution.

Since the scope of the manuscript is computationally-oriented, we already refer the reader to the MATLAB script 1, where in the lines 2–7, an example of an interpolation problem in two dimensions is given. Precisely, we consider $n = 4225$ Halton scattered data [13] (defined via the function `haltonset.m` [19]) as samples of the Franke's function [12]. The kernel that we will use in the experiments and which is defined at line 4 of the MATLAB script 1 is the Matérn C^2 function

$$\phi(r) = (1 + \varepsilon r)e^{-\varepsilon r},$$

whose shape parameter ε is set to 1.

3 The partition of unity method

When a huge number of data is involved, inverting the kernel collocation matrix might be computationally prohibitive. To avoid this drawback, the PU method turns out to be particularly effective [28]. Indeed, it splits the problem via a partition of the open and bounded domain Ω into m *subdomains or patches* Ω_j, such that $\Omega \subseteq \cup_{j=1}^{m} \Omega_j$, with some mild overlap among them. To simplify the notation, in what follows, we fix $\Omega = [0, 1]^d$.

3.1 MATLAB implementation of the PU method

To practically explain how the PU method works, we now also refer to the example shown in the MATLAB script 1 and to the function `PU.m` reported in the MATLAB function 2. The PU covering will be made of overlapping balls of a fixed radius whose centres are grid data $P = \{\tilde{\boldsymbol{x}}_k,\ k = 1, \ldots, m\}$; see line 16 of the MATLAB function 2, where the input `m_d` denotes the number of patches in one direction*. Precisely, we remark that, assuming to have a nearly uniform node distribution, m is a suitable number of PU subdomains on Ω if [9]

$$\frac{N}{m} \approx 2^d.$$

Then, the covering property is satisfied by taking the radius δ so that

$$\delta \geq \frac{1}{m^{1/d}}.$$

Referring to the MATLAB script 1 (line 8), we fix the number of patches in one direction as $\lfloor n^{1/d}/2 \rfloor$, while the radius is set as $\delta = \sqrt{2}/m^{1/d}$; see line 17 of the MATLAB function 2.

Then, the PU method solves on each subdomain a local interpolation problem and constructs the global approximant by gluing together the local contributions thanks to the use of some weights. Precisely, such weights form a partition of unity, i.e. a family of compactly supported, non-negative, continuous functions w_j, with $\operatorname{supp}(w_j) \subseteq \Omega_j$ and such that

$$\sum_{j=1}^{m} w_j(\boldsymbol{x}) = 1, \quad \boldsymbol{x} \in \Omega.$$

In the software, we take the Shepard's weights [25], i.e.

$$w_j(\boldsymbol{x}) := \frac{\bar{w}_j(\boldsymbol{x})}{\sum_{k=1}^{m} \bar{w}_k(\boldsymbol{x})}, \quad j = 1, \ldots, m,$$

where $\bar{w}_j$ are compactly supported functions, with support on Ω_j; see line 19 of MATLAB function 2, where `w`, defined at line 9 of the MATLAB script 1, denotes here the Wendland C^2 function [27].

*The function `MakeSDGrid.m` is provided by [9].

Once we choose the partition of unity $\{w_j\}_{j=1}^m$, the global interpolant is formed by the weighted sum of m local approximants P_f^j, i.e.

$$P_f(\boldsymbol{x}) = \sum_{j=1}^{m} P_f^j(\boldsymbol{x})\, w_j(\boldsymbol{x}) = \sum_{j=1}^{m} \left(\sum_{k=1}^{n_j} c_k^j \kappa(\boldsymbol{x}, \boldsymbol{x}_k^j) \right) w_j(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega,$$

where n_j indicates the number of points on Ω_j and $\boldsymbol{x}_k^j \in X_j = X \cap \Omega_j$, with $k = 1, \ldots, n_j$. The MATLAB functions `IntegerBasedContainingQuery.m`, `IntegerBasedStructure.m`, `IntegerBasedNeighbourhood.m` and `IntegerBasedRangeSearch.m`, respectively reported in the MATLAB functions 3, 4, 5 and 6, organize the points among the sudomains and will be discussed later. Supposing that the data are already properly distributed among the patches, we are now able to determine the coefficients $\{c_k^j\}_{k=1}^{n_j}$ by enforcing the n_j local interpolation conditions, i.e.

$$P_f^j(\boldsymbol{x}_i^j) = f_i^j, \quad i = 1, \ldots, n_j,$$

that leads to solving m linear systems of the form $\mathsf{K}_j \boldsymbol{c}_j = \boldsymbol{f}_j$ (see lines 32–33 of MATLAB function 2)

$$\mathsf{K}_j \boldsymbol{c}_j = \boldsymbol{f}_j,$$

where $\boldsymbol{c}_j = (c_1^j, \ldots, c_{n_j}^j)^T$, $\boldsymbol{f}_j = (f_1^j, \ldots, f_{n_j}^j)^T$ and $\mathsf{K}_j \in \mathbb{R}^{n_j \times n_j}$ is[†]

$$\mathsf{K}_j = \begin{pmatrix} \kappa(\boldsymbol{x}_1^j, \boldsymbol{x}_1^j) & \cdots & \kappa(\boldsymbol{x}_1^j, \boldsymbol{x}_{n_j}^j) \\ \vdots & \ddots & \vdots \\ \kappa(\boldsymbol{x}_{n_j}^j, \boldsymbol{x}_1^j) & \cdots & \kappa(\boldsymbol{x}_{n_j}^j, \boldsymbol{x}_{n_j}^j) \end{pmatrix} = \underbrace{\begin{pmatrix} \phi\left(||\boldsymbol{x}_1^j - \boldsymbol{x}_1^j||_2\right) & \cdots & \phi\left(||\boldsymbol{x}_1^j - \boldsymbol{x}_{n_j}^j||_2\right) \\ \vdots & \ddots & \vdots \\ \phi\left(||\boldsymbol{x}_{n_j}^j - \boldsymbol{x}_1^j||_2\right) & \cdots & \phi\left(||\boldsymbol{x}_{n_j}^j - \boldsymbol{x}_{n_j}^j||_2\right) \end{pmatrix}}_{\texttt{K\_j=phi(ep,DM\_data)}}.$$

The global PU interpolant is then evaluated at a grid of test or evaluation data, $E = \{\bar{\boldsymbol{x}}_\ell,\ \ell = 1, \ldots, s\}$. Precisely, once we determine $\boldsymbol{c}^j$ by solving the collocation system on Ω_j, we can evaluate the local interpolant at the set $E_j = E \cap \Omega_j$ of cardinality s_j as (see lines 34–35 of MATLAB function 2)

$$\begin{pmatrix} P_f^j\left(\bar{\boldsymbol{x}}_1^j\right) \\ \vdots \\ P_f^j\left(\bar{\boldsymbol{x}}_{s_j}^j\right) \end{pmatrix} = \begin{pmatrix} \phi\left(||\bar{\boldsymbol{x}}_1^j - \boldsymbol{x}_1^j||_2\right) & \cdots & \phi\left(||\bar{\boldsymbol{x}}_1^j - \boldsymbol{x}_{n_j}^j||_2\right) \\ \vdots & \ddots & \vdots \\ \phi\left(||\bar{\boldsymbol{x}}_{s_j}^j - \boldsymbol{x}_1^j||_2\right) & \cdots & \phi\left(||\bar{\boldsymbol{x}}_{s_j}^j - \boldsymbol{x}_{n_j}^j||_2\right) \end{pmatrix} \begin{pmatrix} c_1^j \\ \vdots \\ c_{n_j}^j \end{pmatrix}.$$

The final evaluation on the whole grid E is carried out by summing up all the local contributes weighted by the matrix $W \in \mathbb{R}^{s \times m}$ of compactly supported weights as in line 36 of the MATLAB function 2. To give an example, we report the surface interpolating the Franke's function (see Figure 1) at the 4225 data, as the MATLAB script 1 does. As error indicator, we take the Maximum Absolute Error (MAE) defined as:

$$\text{MAE} := \max_{i=1,\ldots,s} |P_f(\bar{x}_i) - f(\bar{x}_i)|.$$

In the considered example, we obtain $\text{MAE} = 6.67\text{E}-04$.

```
                        % Example 1
d = 2; n = [65].^d; % Define the space dimension and the number of data
p = haltonset(d); x = net(p,n); % Define the interpolation data (Halton points)
phi = @(epsilon,r) (1+epsilon*r).*exp(-epsilon*r); epsilon = 1; % Define the kernel
franke = @(x1,x2) 0.75 * exp(-(9*x1-2).^2/4 - (9*x2-2).^2/4)+ 0.75 * exp(-(9*x1+1).^2/49 - ...
    (9*x2+1)./10)+0.5 * exp(-(9*x1-7).^2/4 - (9*x2-3).^2/4)-0.2 * exp(-(9*x1-4).^2 - (9*x2-7).^2);
f = franke(x(:,1),x(:,2)); % Define the test function and the function values
m_d = floor(n^(1/d)/2); s_d = 60; % Define the number of patches and evaluation data in one direction
w = @(supp,r) (max(1-(supp*r),0).^4).*(4*(supp*r)+1); % Define the PU weights (Wendland C^2)
bar_x = MakeSDGrid(d,s_d); % Create s_d^d equally spaced test data
Pf = PU(d,x,bar_x,m_d,phi,w,f,epsilon); % Compute the PU interpolant
```

MATLAB script 1: Template script for computing the PU interpolant.

```
function [Pf] = PU(d,x,bar_x,m_d,phi,w,f,epsilon)
%
% Goal: script that performs partition of unity
%
% Inputs:  d: space dimension
%          x: nXd matrix representing a set of n interpolation data
%          bar_x: sXd matrix representing a set of s evaluation data
```

[†]The function `DistanceMatrix.m` is provided by [9].

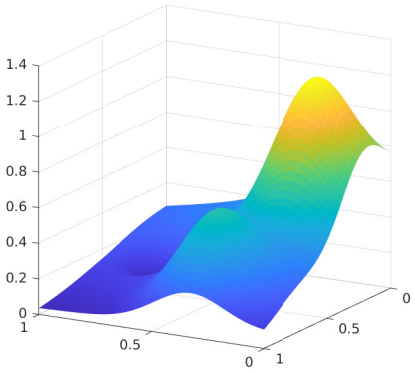

Figure 1: Graphical representation of the PU interpolant returned by the MATLAB script 1.

```
%          m_d: number of PU subdomains in one direction
%          phi: radial basis function
%          w: weight function
%          f: the function values
%          epsilon: the shape parameter
%
% Outputs:  Pf: sXd matrix representing the PU fit
%
tilde_x = MakeSDGrid(d,m_d); % Create m_d^d equally spaced PU centres
delta = sqrt(2)/m_d;  supp = 1/delta; % Define the PU radius and the parameter for the weight functions
m = size(tilde_x,1); s = size(bar_x,1); Pf = zeros(s,1);  % Initialize and compute the Shepard matrix
DM_eval = DistanceMatrix(bar_x,tilde_x); W = w(supp,DM_eval); W = spdiags(1./(W*ones(m,1)),0,s,s)*W;
q = ceil(1./delta); % Parameter for the integer-based partitioning structure
% Build the partitioning structure for interpolation and evaluation data
X_block = IntegerBasedStructure(x,q,delta,d); bar_X_block = IntegerBasedStructure(bar_x,q,delta,d);
for j = 1:m % Loop over subdomains
    k = IntegerBasedContainingQuery(tilde_x(j,:),q,delta,d); % Find the box with the j-th PU centre
    % Find the interpolation data located in the j-th subdomain
    [X_NeigBlock, idx_X_NeigBlock] = IntegerBasedNeighbourhood(x,X_block,k,q,d);
    n_j = IntegerBasedRangeSearch(tilde_x(j,:),delta,X_NeigBlock,idx_X_NeigBlock);
    % Find the evaluation data located in the j-th subdomain
    [bar_X_NeigBlock, idx_bar_X_NeigBlock] = IntegerBasedNeighbourhood(bar_x,bar_X_block,k,q,d);
    s_j = IntegerBasedRangeSearch(tilde_x(j,:),delta,bar_X_NeigBlock,idx_bar_X_NeigBlock);
    if  (~isempty(s_j)) &&  (~isempty(n_j))
        DM_data = DistanceMatrix(x(n_j,:),x(n_j,:)); K_j = phi(epsilon,DM_data); % Interpolation matrix
        c_j = K_j\f(n_j);  % Compute the interpolation coefficients
        DM_eval = DistanceMatrix(bar_x(s_j,:),x(n_j,:)); % Compute the evaluation matrix
        K_eval = phi (epsilon, DM_eval); P_fj = K_eval*c_j; % Compute the local fit
        Pf(s_j) = Pf(s_j) + P_fj.*W(s_j,j); % Accumulate the global fit
    end
end
```

MATLAB function 2: Implementation of the PU method.

3.2 Partitioning data structures

In this subsection, we comment about the procedure used to organize the points among the different subdomains. The so-called integer based routines have been introduced in [7] and can be thought as a multidimensional improvement of the fast procedures, known as cross-strip algorithms, studied in [4] and further developed in [6].

To organize the data into the different subdomains, we make use of an additional structure, which is obtained from the subdivision of Ω into several boxes. The number q of boxes (also called blocks) along one side of $[0,1]^d$ is linked to the PU radius δ and is given by (see line 20 the MATLAB function 2)

$$q = \left\lceil \frac{1}{\delta} \right\rceil. \tag{1}$$

We thus number blocks from 1 to q^d, starting from the subspace of dimension $d-1$, obtained projecting along the first coordinate and thus parallel to the remaining ones. Then to organize the points we use the so-called integer-based routines discussed in what follows.

The first query we have to answer is: given a PU centre find the index of the block it belongs to. We observe that, given a PU centre $\bar{\boldsymbol{x}}_j$, the index of the k-th block containing the subdomain centre is given by (see line 13 of the Matlab function 3)

$$k = \sum_{\ell=1}^{d-1} (k_\ell - 1)\, q^{d-\ell} + k_d. \tag{2}$$

To find the indices k_ℓ, $\ell = 1, \ldots, d$, in (2), we use an integer-based procedure consisting in rounding off to an integer value. Specifically, for each PU centre $\bar{\boldsymbol{x}}_j = (\bar{x}_{j1}, \ldots, \bar{x}_{jd})$, we have that

$$k_\ell = \left\lceil \frac{\bar{x}_{j\ell}}{\delta} \right\rceil.$$

```
1  function [k] = IntegerBasedContainingQuery(tilde_x,q,delta,d)
2  %
3  % Goal: script that given a subdomain centre returns the index of the square block containing the
4  %       subdomain centre
5  %
6  % Inputs: tilde_x: subdomain centre
7  %         q: number of blocks in one direction
8  %         delta: radius of the PU subdomains
9  %         d: space dimension
10 %
11 % Outputs: k: the index of the block containing the subdomain centre
12 %
13 k_l = ceil(tilde_x/delta); l = 1:d-1; k_l(k_l == 0) = 1; k = sum((k_l(l)-1)*q.^(d-l)) + k_l(end);
```

MATLAB function 3: Implementation of the integer-based containing query procedure.

The same rounding off strategy is adopted in order to store both the scattered data and the evaluation points into the different blocks. We create a partitioning data structure in which we save all the indices of the points lying on the different subdomains. This is carried out with the routine reported in the MATLAB function 4.

```
1  function [X_block] = IntegerBasedStructure(x,q,delta,d)
2  %
3  % Goal: find the data sites located in each of the q^d blocks
4  %
5  % Inputs: x: nXd matrix representing a set of data
6  %         q: number of blocks in one direction
7  %         delta: radius of PU subdomains
8  %         d: space dimension
9  %
10 % Outputs: X_block: multiarray containing the indices of the data points located in k-th block
11 %
12 n = size(x,1); X_block = cell(q^d,1); k = 1:d-1; % Initialize
13 for i = 1:n % Find the indices of the data points located in k-th block
14     idx = ceil(x(i,:)/delta); idx(idx == 0) = 1; index = sum((idx(k)-1)*q.^(d-k)) + idx(end);
15     X_block{index} = [X_block{index}; i];
16 end
```

MATLAB function 4: Implementation of the integer-based data structure.

After the implementation of the containing query procedure and after distributing the data among the boxes, we now have to address the following computational issues:

- given a set of interpolation data X and a subdomain Ω_j, find all the interpolation points located in that patch, i.e. $\boldsymbol{x}_i \in X_j$, $i = 1, \ldots, n_j$.
- given a set of evaluation data E and a subdomain Ω_j, find all evaluation points located in that patch, i.e. $\bar{\boldsymbol{x}}_i \in E_j$, $i = 1, \ldots, s_j$.

By setting q as in (1), we know that, given a centre $\tilde{\boldsymbol{x}}_j$ that belongs to the k-th block, we have to search for the neighboring points in the k-th block and in its $3^d - 1$ neighboring blocks (defined via the MATLAB function 5).

```
function [X_NeigBlock, idx_X_NeigBlock] = IntegerBasedNeighbourhood(x,X_block,k,q,d)
%
% Goal: script that finds the neighbouring blocks
%
% Inputs:  x: nXd matrix representing a set of n data sites
%          X_block: the integer-based data structure
%          k: the k-th block cointaining the subdomain centre
%          q: number of blocks in one direction
%          d: space dimension
%
% Outputs:  X_NeigBlock, dx_X_NeigBlock: points (and indices) lying in the k-th neighbourhood
%
neigh = []; l = d-1; index = k; % Initialize
while l  > 0 % Find neighbouring blocks
     neigh = [k+q.^l,k-q.^l];
    if l - 1 > 0
        neigh = [neigh,neigh+q.^(l-1),neigh-q.^(l-1)];
    end
    l = l - 1;
end
k2 = 1; neighplus = []; neighminus = []; % Initialize
for i = 1:length(neigh)
    neighplus(k2) = neigh(i) + 1; neighminus(k2) = neigh(i) - 1; k2 = k2 + 1;
end
neigh = [neigh,k+1,k-1,neighplus,neighminus]; % Reduce the number of blocks for border blocks
j = find(neigh > 0 & neigh <= q^d); index = [index; neigh(j)']; X_NeigBlock = []; idx_X_NeigBlock = [];
for p = 1:length(index)
    X_NeigBlock = [X_NeigBlock;x(X_block{index(p)},:)];
    idx_X_NeigBlock = [idx_X_NeigBlock;X_block{index(p)}];
end
```

MATLAB function 5: Implementation of the integer-based neighbourhood routine.

In particular, the partitioning structure based on blocks enables us to examine in the searching process at most $3^d - 1$ blocks. In fact, when a block lies on the boundary of the bounding box, we reduce the number of neighboring blocks to be considered. Finally, the points lying on the j-th patch are found via the so-called range search procedure, where a sorting routine is used to speed up the process, see line 18 of the MATLAB function 6.

```
function [n_j] = IntegerBasedRangeSearch(tilde_x,delta,X_NeigBlock,idx_X_NeigBlock)
%
% Goal: find the data sites located in a given subdomain and the distances between the
%       subdomain centre and data sites
%
% Inputs:  tilde_x: subdomain centre; delta: radius of PU subdomains
%          X_NeigBlock: nXd matrix representing a set of n points
%          idx_X_NeigBlock: vector containing the indices of the data points located in the k-th block
%                           and in the neighbouring blocks
%
% Outputs: n_j:  vector containing the indices of the data points located in a given PU subdomain
%
N = size(X_NeigBlock,1); n_j = []; % Initialize
for i = 1:N % Compute distances between the data sites and the centre
    dist1(i) = norm(tilde_x - X_NeigBlock(i,:));
end
if N > 0 % Use a sort procedure to order distances
    [sort_dist,IX] = sort(dist1); N1 = size(sort_dist,2); j1 = 1; j2 = 1; %Initialize
    while (j2 <= N1) && (sort_dist(j2) <= delta) % Find the data sites located in the given subdomain
        n_j(j1) = idx_X_NeigBlock(IX(j2)); j1 = j1 + 1; j2 = j2 + 1;
    end
end
```

MATLAB function 6: Implementation of the range search routine.

We conclude this section with a few remarks on the complexity costs of the PU method. The integer-based partitioning structure, after organizing the scattered data into the different blocks, given a subdomain Ω_j searches for all the points lying on Ω_j in a reduced number of blocks. Specifically, in order to store the scattered data among the different blocks, it assigns to each node $\boldsymbol{x}_i$, $i = 1, \ldots, n$, the corresponding block. This step requires $\mathcal{O}(N)$ time. Then,

we apply the optimized searching routine that, supposing to have quasi-uniform nodes, is performed in a constant time. The final step consists in solving m linear systems of size $n_j \times n_j$, with $n_j \ll n$, thus requiring a running time $\mathcal{O}(n_j^3)$, $j = 1, \ldots, m$, for each patch.

4 Python implementation of the PU method

The Python implementation follows broadly that one of Matlab except for the use of some inherent structures of the language. We tried to remain as faithful as possible to the names of variables and functions of the Matlab implementation to facilitate the comparison of the codes. Furthermore we use some other libraries as NumPy [8] and SciPy [26]. For details about the method see Section 3.

The Python function 7 works as the Matlab function 2, except for the fact that we use a *list comprehension* to generate the grid of equally spaced centers.

```
def PU(x, f, bar_x, m_d, w,  phi, epsilon):
    """
    Goal: build the partition of unity interpolant and approximate
          the values on a given set of points

    Inputs: x: nxd numpy array  representing a set of n data sites
            f: the function values
            bar_x: sxd numpy array  representing a set of s evaluation data
            m_d: number of PU subdomains in one direction
            w: weight function
            phi: radial basis function
            epsilon: the shape parameter

    Outputs: Pf: sXd numpy array representing the PU fit
    """

    m, d = x.shape
    s, _ = bar_x.shape
    Pf = np.zeros(s)

    # Create m_d^d equally spaced PU centres
    tilde_x = np.array([i for i in itertools.product(np.linspace(0, 1, int(m_d)), repeat=d)])
    # Define the PU radius and the parameter for the weight functions
    radius_par = 2 ** (1 / 2)
    delta = radius_par / m_d
    supp = 1 / delta

    # Parameter for the integer-based partitioning structure
    q = np.ceil(1 / delta)

    # Build the partitioning structure for interpolation and evaluation data
    X_block = integer_based_structure(x, q, delta, d)
    bar_X_block = integer_based_structure(bar_x, q, delta, d)

    # Initialize and compute the Shepard matrix
    sem = w(supp, distance_matrix(bar_x, tilde_x))
    sem = spdiags(1/(sem@np.ones(int(m_d)**d)), 0, s, s)@sem

    # Loop over subdomains
    for j, center in enumerate(tilde_x):

        # Find the box with the j-th PU centre
        k = integer_based_containing_query(center, q, delta, d)
        # Find the interpolation data located in the j-th subdomain
        X_neig_block = integer_based_neighbourhood(k, q, d)
        n_j = integer_based_range_search(center, delta, x, X_block, X_neig_block)

        if n_j.size != 0:
            # Interpolation matrix
            c_j = np.linalg.solve(phi(epsilon, distance_matrix(x[n_j, :], x[n_j, :])), f[n_j])
            bar_X_neig_block = integer_based_neighbourhood(k, q, d)
```

```
            s_j = integer_based_range_search(center, delta, bar_x, bar_X_block, bar_X_neig_block)

            if s_j.size != 0:
                # Compute the local fit
                p_fj = np.dot(phi(epsilon, distance_matrix(bar_x[s_j, :], x[n_j, :])), c_j)
                # Accumulate the global fit
                Pf[s_j] += p_fj * np.array(sem[s_j, j])
    return Pf
```

Python function 7: Implementation of the PU method.

The Python function 8 represents the Python arrangement of the Matlab function 5. By applying an integer-based procedure it finds the block to which a given subdomain center belongs.

```
def integer_based_containing_query(tilde_x, q, delta, d):
    """
    Goal:   given a subdomain centre returns the index of the
            square block containing the subdomain centre

    Inputs: tilde_x: subdomain centre
            q: number of blocks in one dimension
            delta: radius of PU subdomains
            d: space dimension

    Outputs: k: the block containing the subdomain centre
    """
    k = np.ceil(tilde_x/delta)
    k[k == 0] = 1
    k = np.sum((k[:-1] - 1) * q ** np.arange(d - 1, 0, -1)) + k[-1]
    return k
```

Python function 8: Implementation of the integer-based containing query procedure.

The same computations made to find to which block a given subdomain belongs are used in the Python function 9 to distribute the data among the blocks. This procedures turns out in a structure in which for each block we have an array that contains all the indices of the points it contains. The difference to underline with respect to the Matlab function 4 is the use of the dictionary structure, i.e. in place of multi-arrays, keys are utilized for blocks and values for arrays of points.

```
def integer_based_structure(x, q, delta, d):
    """
    Goal: find the data sites located in each of the q^d blocks

    Inputs: x: nxd numpy array  representing a set of n data sites
            q: number of blocks in one dimension
            delta: radius of PU subdomains
            d: space dimension

    Outputs: X_block: dictionary {key: values} key represent the index
             of the block  and values is a ndarray that contain indexes
             of points located in k-th block
    """

    X_block = {}
    n, _ = x.shape
    for ind in range(n):
        index = np.ceil(x[ind]/delta)
        index[index == 0] = 1
        index = int(np.sum((index[:-1] - 1)*q**np.arange(d-1, 0, -1)) + index[-1])
        if index in X_block.keys():
            X_block[index] = np.append(X_block[index], [ind], axis=0)
        else:
            X_block[index] = np.array([ind])
```

```
    return X_block
```

Python function 9: Implementation of the integer-based data structure.

As the Matlab function 5, its corresponding Python version, i.e. function 10, given a block, finds the indexes of its neighbouring blocks. The construction of such a neighborhood is useful to avoid applying the searching procedure on the whole dataset reducing the searching process to such neighbourhood.

```
def integer_based_neighbourhood(k, q, d):
    """
    Goal: given a block finds neighbouring blocks

    Inputs:  k: index of the block
             q: number of blocks in one direction
             d: space dimension

    Outputs: X_NeigBlock: indices of the neighbouring blocks
    """

    neigh = np.array([])
    ld = d - 1
    while ld > 0:
        neigh = np.append(neigh, [k + q ** ld, k - q ** ld])
        if ld - 1 > 0:
            neigh = np.append(neigh, np.array([neigh + q ** (ld - 1), neigh - q ** (ld - 1)]))
        ld -= 1

    neigh = np.append(neigh, k)
    neigh = np.append(neigh, [neigh + 1, neigh - 1])
    X_neig_block = neigh[np.logical_and(neigh > 0, neigh <= q ** d)]
    return X_neig_block
```

Python function 10: Implementation of the integer-based neighbourhood routine.

The searching process is implemented in the function 11 in which, given a subdomain center and the structures built via the functions 9 and 10, all the points belonging to the subdomain are provided as an array.

```
def integer_based_range_search(tilde_x, delta, x,  X_block, idx_X_neigh_block):
    """
    Goal:   find the data sites located in a given subdomain

    Inputs: tilde_x: subdomain centre
            delta: radius of PU subdomain
            x: nxd numpy array  representing a set of n data sites
            idx_X_neigh_block:  dictionary {key: value} key represent
                                the index of the block  and value
                                is a list that contain indexes of
                                points located in k-th block  and in
                                the neighbouring blocks

    Outputs: n_j:  list of the indexes of the points belonging to a
                   given subdomain
    """

    if idx_X_neigh_block.size != 0:
        n_j = []
        for key in idx_X_neigh_block:
            try:
                for ind in X_block[key]:
                    if np.linalg.norm(tilde_x - x[ind]) <= delta:
                        n_j.append(ind)
            except KeyError:
                pass
        n_j = np.array(n_j)
```

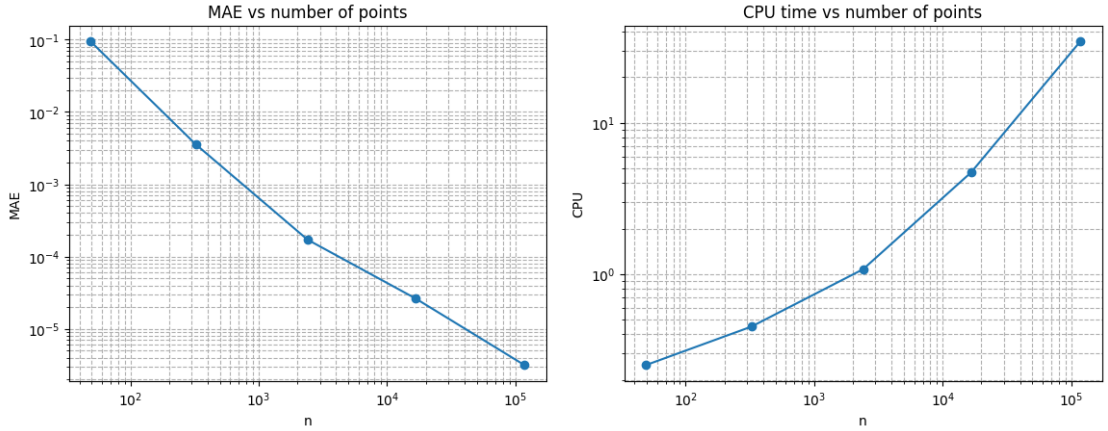

Figure 2: Log-Log representation of the maximum absolute error and execution time as the number of interpolation points increases.

```
    else:
        n_j = np.arange(x.shape[0])
    return n_j
```

Python function 11: Implementation of the range search routine.

5 Numerical examples

Tests are carried out on a MacBook Air (2020), 1.2 GHz Quad-Core Intel Core i7 processor, 16 GB 3733 MHz LPDDR4X RAM, via Python 3.9.12. To run the Python code the user needs to have at least the Python 3.8 release and install the *partitionunity* package from the Python Package Index. The command line to install the package differs from the OS. Unix/macOS:

```
python3 -m pip install partitionunity
```

Windows:

```
py -m pip install partitionunity
```

After the installation the user needs to import the package, as shown in the Python script 12, with the instruction

```
import partitionunity
```

the alias *pu* is used to not repeat the full package name in the code.

To point out the efficacy of the PU implementation, we take $n = \lfloor 7^{p/2} \rfloor^2$, $p = 2, 3, 4, 5, 6$, equispaced interpolation data and we sample the Franke function. For each of the five datasets, we report in Figure 2 the MAE and the CPU times needed for computing the PU interpolant. The experiments are carried out by taking the Matérn C^2 kernel. For completeness and for the reproducibility of the results, we report the Python script 12 that returns the outcomes depicted in Figure 2.

```
import partitionunity as pu
import numpy as np
import time
import matplotlib.pyplot as plt

# Define the test function
def franke(X):
    return 0.75 * np.exp(-(9*X[:, 0]-2)**2/4 - (9*X[:, 1]-2)**2/4) + \
           0.75 * np.exp(-(9*X[:, 0]+1)**2/49 - (9*X[:, 1]+1) / 10) + 0.5 * \
           np.exp(-(9*X[:, 0]-7)**2/4 - (9*X[:, 1]-3)**2/4)-0.2 * \
           np.exp(-(9*X[:, 0]-4)**2 - (9*X[:, 1]-7)**2)


# Define the space dimension and the number of interpolation data
d = 2
p = np.append(2, np.arange(2, 7))
n = np.floor(np.sqrt(7**p)).astype(int)

```

```

# Define the kernel and parameter
def Phi(eps, r):
    return (1 + eps * r) * np.exp(-eps * r)


epsilon = 1

# Define the weights for PU
def weight(e, r):
    return np.multiply(np.power(np.fmax(1-(e*r), 0*(e*r)), 4), (4*(e*r)+1))


# Define s_d^d equally spaced test data
s_d = 60
bar_x = np.array(np.meshgrid(np.linspace(0, 1, s_d), np.linspace(0, 1, s_d))).T.reshape(-1, d)
t = []
MAE = []
for i in range(len(p)):

    # Define the interpolation data
    x = np.array(np.meshgrid(np.linspace(0, 1, n[i]), np.linspace(0, 1, n[i]))).T.reshape(-1, d)

    # Define the function values
    y = franke(x)

    m_d = np.floor((n[i] ** 2) ** (1 / d) / 2)
    tm = time.time()

    # Compute the PU interpolant
    Pf = pu.PU(x, y, bar_x, m_d, weight, Phi, epsilon)

    t.append(time.time() - tm)
    MAE.append(np.max(abs(franke(bar_x) - Pf)))

# Display the results
plt.loglog(n[1:]**2, MAE[1:], 'o-', )
plt.title("MAE vs number of points")
plt.xlabel("n")
plt.ylabel("MAE")
plt.grid(True, which="both", linestyle='--')
plt.show()

plt.loglog(n[1:]**2, t[1:], 'o-')
plt.title("CPU time vs number of points")
plt.xlabel("n")
plt.ylabel("CPU(s)")
plt.grid(True, which="both", linestyle='--')
plt.show()
```

Python script 12: Usage example of the Python package.

6 Conclusions and work in progress

We discussed the MATLAB and Python routines needed for interpolating large datasets via the PU method. Its efficacy is numerically shown in Section 5. Indeed, we are able to interpolate more than 100000 data in a reasonable time. Work in progress consists in investigating its extension to other settings, e.g. to regression and support vector machines [18], and hence use the PU structure together with local fits provided by regression routines that belong to the Python Scikit Learn package [21] or the MATLAB Statistics and Machine Learning Toolbox.

Acknowledgements

This work was supported by the Gruppo Nazionale per il Calcolo Scientifico (INdAM – GNCS) and by the 2020 projects "Models and numerical methods in approximation, in applied sciences and in life sciences"and "Mathematical methods

in computational sciences" funded by the Dipartimento di Matematica "Giuseppe Peano" of the Università di Torino. This research has been accomplished within the RITA "Research ITalian network on Approximation" and the UMI Group TAA "Approximation Theory and Applications". The authors sincerely thank the anonymous referees for the valuable comments and suggestions, which enabled to improve the paper.

References

[1] I. Babuška, J.M. Melenk. The partition of unity method. *Int. J. Numer. Meth. Eng.*, 40:727–758, 1997.

[2] M.D. Buhmann. Radial Basis Functions: Theory and Implementation. Cambridge Monogr. Appl. Comput. Math., vol. 12, Cambridge Univ. Press, Cambridge, 2003.

[3] R. Cavoretto, A. De Rossi. Error indicators and refinement strategies for solving Poisson problems through a RBF partition of unity collocation scheme. *Appl. Math. Comput.*, 369:124824, 2020.

[4] R. Cavoretto, A. De Rossi. A trivariate interpolation algorithm using a cube-partition searching procedure. *SIAM J. Sci. Comput.*, 37:A1891–A1908, 2015.

[5] R. Cavoretto, A. De Rossi. Software for Approximation 2022 (SA2022). *Dolomites Res. Notes Approx.*, Special Issue SA2022, 15:i–ii, 2022.

[6] R. Cavoretto, A. De Rossi, F. Dell'Accio, F. Di Tommaso. An efficient trivariate algorithm for tetrahedral Shepard interpolation. *J. Sci. Comput.*, 82:57, 2020.

[7] R. Cavoretto, A. De Rossi, E. Perracchione. Optimal selection of local approximants in RBF-PU interpolation. *J. Sci. Comput.*, 74:1–22, 2018.

[8] R. Charles, K. Harris, J. Millman, J.S. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant. Array programming with NumPy. *Nature* 585:357–362, 2020.

[9] G.E. Fasshauer. Meshfree Approximations Methods with MATLAB. World Scientific, Singapore, 2007.

[10] G.E. Fasshauer, M.J. McCourt. Kernel-based Approximation Methods Using MATLAB. World Scientific, Singapore, 2015.

[11] B. Fornberg, G. Wright. Stable computation of multiquadrics interpolants for all values of the shape parameter. *Comput. Math. Appl.*, 47:497–523, 2004.

[12] R. Franke, H. Hagen. Least squares surface approximation using multiquadrics and parametric domain distorsion. *Comput. Aided Geom. Design*, 16:177–196, 1999.

[13] J.H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math.*, 2:84–90, 1960.

[14] A. Iske. Scattered data approximation by positive definite kernel functions. *Rend. Sem. Mat. Univ. Pol. Torino*, 69:217–246, 2011.

[15] E. Larsson, B. Fornberg. Theoretical and computational aspects of multivariate interpolation with increasingly flat radial basis functions. *Comput. Math. Appl.*, 49:103–130, 2005.

[16] E. Larsson, V. Shcherbakov, A. Heryudono. A least squares radial basis function partition of unity method for solving PDEs. *SIAM J. Sci. Comput.*, 39:A2538–A2563, 2017.

[17] F. Marchetti. The extension of Rippa's algorithm beyond LOOCV. *Appl. Math. Letters*, 120:107262, 2021.

[18] F. Marchetti, E. Perracchione. Local-to-Global Support Vector Machines (LGSVMs). *Pattern Recognit.*, 132:108920, 2022.

[19] MATLAB, Natick, Massachusetts, The Mathworks, Inc. R2019b.

[20] D. Mirzaei. The direct radial basis function partition of unity (D-RBF-PU) method for solving PDEs. *SIAM J. Sci. Comput.* 43:A54–A83, 2021.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. Scikit-learn: Machine Learning in Python, *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[22] S. Rippa. An algorithm for selecting a good value for the parameter c in radial basis function interpolation. *Adv. Comput. Math.* 11:193–210, 1999.

[23] A. Safdari-Vaighani, A. Heryudono, E. Larsson. A radial basis function partition of unity collocation method for convection-diffusion equations arising in financial applications. *J. Sci. Comput.*, 64:341–367, 2015.

[24] B. Schölkopf, A.J. Smola. Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA, USA, 2002.

[25] D. Shepard. A two-dimensional interpolation function for irregularly spaced data. In: Proceedings of 23-rd National Conference, Brandon/Systems Press, Princeton, 517–524, 1968.

[26] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K.J. Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C.J. Carey, I. Polat, Y. Feng, E.W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I Henriksen, E.A. Quintero, C. R Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. ciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17(3):261–272, 2020.

[27] H. Wendland. Scattered Data Approximation. Cambridge Monogr. Appl. Comput. Math., vol. 17, Cambridge Univ. Press, Cambridge, 2005.

[28] H. Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In: C.K. Chui et al. (Eds.), Approximation Theory X: Wavelets, Splines, and Applications, Vanderbilt Univ. Press, Nashville, 473–483, 2002.