An acceleration method for moving least squares based on a generalized octree for massive data

Sanpeng Zheng

[a]School of Mathematicial Sciences, Guizhou Normal University, Huaxi University Town, Guiyang, 550025, China

Abstract

Moving least squares (MLS) method is one of the most classical methods in scattered data fitting. To improve computational efficiency and approximation accuracy, the compact support weight functions are used. However, due to "moving" in MLS, the approximations at different points are obtained by solving different weighted least squares problems, which significantly reduces the computational efficiency of MLS for massive data, especially in problems that require multiple approximations. To improve the computational efficiency, this paper proposes an acceleration method based on a generalized octree. This method reduces the time of the neighborhood search in MLS based on the specific data structure, improves the computational efficiency and preserves the approximation results of MLS. Numerical experiments are presented to demonstrate the efficiency and accuracy of the proposed method in comparison with the acceleration methods based on kd-tree and piecewise computation under data sets with various sizes and dimensions.

Keywords: Scattered Data Fitting, Massive Data, Moving Least Squares Method, Octree, Neighborhood search

1. Introduction

The study of data fitting methods has consistently been a classical problem in approximation. The continuous development of computer software and hardware has significantly improved the capabilities of obtaining and storing data. This has led to the emergence of scattered data, which refers to data lacking prior knowledge about their distribution and additional structures, in diverse fields, thereby rendering the fitting problem associated with

such data increasingly significant. Moving least squares (MLS) method, as one of the most classical methods in scattered data approximation, is not sensitive to the dimension and distribution of data, and can approximate the sampled function with high accuracy and continuity [1–5]. The generalized moving least squares method can directly approximate the derivatives of sampled functions [6–8]. These advantages make MLS widely used in curve and surface fitting [9, 10], mesh-free methods in numerical solution of PDEs [11, 12], outliers detection [13, 14] and so on.

The various advantages of MLS are due to the weight function which depends on the point to be approximated, which makes the result of MLS a local approximation with greater emphasis on nearby data. In each MLS approximation, the different weighted least squares (WLS) problems need to be formed and solved. It is obvious that if the size of data increases, the cost of computation to form and solve the WLS would significantly increase, making the single approximation time greatly increased. In the case, in problems such as curve and surface fitting that require multiple approximations, the computational time is easy to become unacceptable.

The compact support weight function (CSW) plays an important role in MLS. It helps derive theoretical results [1, 2] and reduces computational time [3, 4] by decreasing the scale of WLS solved each time. However, with the sharp increase in the size of data that needs to be processed in recent years, the neighborhood search that detects data points in the support of CSW requires a lot of time.

In many applications for massive data, there is such a neighborhood search problem. The current popular method is based on the kd-tree [15]. The kd-tree is a binary tree composed of k-dimensional data points. The original intention of building it is to accelerate the nearest neighbor search of data, and by the ability, it can be helpful to the neighborhood search. Fasshauer [4] uses kd-tree to accelerate MLS in curve fitting and Wendland also introduces kd-tree in his book [3]. The neighborhood search based on the kd-tree not only has top-down judgment, but also needs to backtrack to give an accurate result. Because of the binary tree structure, the depth of kd-trees would be very large for the massive data, and the time of the neighborhood search would be raised significantly. Lu et al. [16] believe that the search algorithms based on the kd-tree is only dominant in the size of hundreds of thousands of data, which is obviously not enough for applications and studies. In addition to the kd-tree, researchers also construct corresponding structures to accelerate specific methods. For instance, for RBF interpola-

tion based on partition of unity, Cavoretto and De Rossi [17, 18] partition the function's domain and dataset into cells (for bivariate functions) or cubes (for trivariate functions) to increase the efficiency of interpolation on each subdomain and the final combination.

In order to efficiently fit massive data, Li et al. [19] modify the framework of MLS and propose a piece-wise moving least squares method (PMLS). In PMLS, the sampling space is divided into some subspaces, and at each subspace a MLS approximation function at an anchor point is used as the approximation function for the whole subspace. Since the anchor points are given, the approximation functions can be calculated and stored in advance. When using PMLS, only the approximation function corresponding to the subspace which the approximated point belongs to is used, which saves the time of forming and solving WLS. Li et al. prove that PMLS is a specific optimal design, and use numerical experiments to show its slightly inferior approximation accuracy to MLS. However, we think that there are at least two defects of PMLS: 1. The continuity of the PMLS approximation function at the subspace boundaries is challenging to demonstrate; 2. The rough division of space can not ensure accuracy, and a fine division causes too many coefficients needed to be calculated and stored in advance, which can not ensure the advantage of computational time.

Different from the idea of modifying the framework of MLS following Li et al., this paper aims to improve the computational efficiency of MLS while retaining its various advantages, so that we uses a generalized octree to accelerate the neighborhood search in massive data and reduce the single approximation time.

The octree is a very important data structure for three-dimensional data [20, 21]. It not only stores data by region but also divides the space in the same time, providing convenience for the data storage and search [16]. Because of these advantages, octrees are used in the multilevel models [22, 23], the spatial discretization of three-dimensional space in solving PDEs [24, 25], the management of point cloud data [16] and so on. The idea of the octree is easy to be generalized into d-dimensional data and is introduced briefly in some papers [16, 26]. However, the applications of the octree are mainly concentrated on 2 and 3-dimensional data currently. Aiming to apply the generalized octree to d-dimensional data, there are many details that need to be completed.

In theory, MLS is not sensitive to the dimension of data, and it is applied to high-dimensional data in some papers [13, 14]. Therefore this paper con-

struct a generalized octree to accelerate MLS instead of using a traditional octree for 2 and 3-dimensional data. To raise the efficiency of MLS fully, the generalized octree proposed in this paper would not only be used to accelerate the neighborhood search, and the process of its building need also be fast and memory-saving. In order to avoid confusion, the tree constructed in this paper is denoted as G-Octree. Based on G-Octree, this paper proposes a fast neighborhood search method and combine the method with MLS to reduce the single approximation time. Numerical experiments show that under massive data, the preparation time and the single approximation time of the proposed method are both significantly lower than those of the acceleration method based on the popular kd-tree if the search radius is not very large for the data set. In experiments, the proposed method do not alter the results of MLS, and its approximation errors are lower than those of PMLS.

The rest of this paper is organized as follows. Section 2 briefly introduces the important preliminary knowledge: the moving least squares method and the traditional octree. Section 3 details the construction of G-Octree, the fast neighborhood search method based on G-Octree, and the process of MLS based on G-Octree. Section 4 compares the results of the acceleration method based on G-Octree with those of the method based on kd-tree and PMLS under data sets with various sizes and dimensions. Section 5 concludes this paper.

2. Preliminaries

In this section, we would briefly introduce the important preliminary knowledge: the moving least squares method and the traditional octree.

2.1. Moving least squares method

Generally, the moving least squares (MLS) method is considered to originate from the studies about curve fitting by Shepard [27] and McLain [28]. After being summarized by many scholars [3, 4], it can now be described in the following way. For an unknown function $f : \Omega \to \mathbb{R}$, where $\Omega \subset \mathbb{R}^d$, and Ω is bounded and non-empty. $X = \{\boldsymbol{x}_i\}_{i=1}^N$ is a set of N distinct points in Ω, the vector obtained by sampling on X is $\boldsymbol{F} = (f_1, \cdots, f_N)^T$. By a given weight function $\boldsymbol{\theta}$, the following functional can be constructed:

$$E_{2,\boldsymbol{\theta},\boldsymbol{x}}(p) = \sum_{i=1}^{N} (f_i - p(\boldsymbol{x}_i))^2 \boldsymbol{\theta}(\boldsymbol{x}, \boldsymbol{x}_i). \tag{2.1}$$

Let $\mathbb{P}_m^d$ denote the polynomial space containing all polynomials whose degrees are less than or equal to m on $\mathbb{R}^d$ and its dimension is $Q = C_{m+d}^d$. Suppose a basis for $\mathbb{P}_m^d$ is $\{p_i(\boldsymbol{x})\}_{i=1}^Q$. For an given point $\boldsymbol{x}$ in Ω, MLS approximation in $\mathbb{P}_m^d$ is obtained by minimizing (2.1), and the approximation value is

$$s_{f,X} = \sum_{i=1}^{Q} c_i(\boldsymbol{x}) p_i(\boldsymbol{x}).$$

Here, $\boldsymbol{c}(\boldsymbol{x})$ is a solution of

$$\boldsymbol{P}^\top \boldsymbol{W}(\boldsymbol{x}) \boldsymbol{P} \boldsymbol{c}(\boldsymbol{x}) = \boldsymbol{P}^\top \boldsymbol{W}(\boldsymbol{x}) \boldsymbol{F}, \tag{2.2}$$

where $\boldsymbol{P} = \big(p_i(\boldsymbol{x}_j)\big)_{Q\times N}$ and $\boldsymbol{W}(\boldsymbol{x}) = \mathrm{diag}\left((\theta(\boldsymbol{x},\boldsymbol{x}_1),\cdots,\theta(\boldsymbol{x},\boldsymbol{x}_N))\right)$.

Remark 2.1. For two different points $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$, $\boldsymbol{c}(\boldsymbol{x}_a)$ and $\boldsymbol{c}(\boldsymbol{x}_b)$ need to be solved by (2.2), respectively. Therefore, although the approximation function space is $\mathbb{P}_m^d$, $s_{f,X}(\boldsymbol{x}) \notin \mathbb{P}_m^d$, which is one of the meanings of "moving".

The compactly supported weight function (CSW) can be applied to improve computational efficiency. Generally, let $\theta(\boldsymbol{x},\boldsymbol{y}) = \Phi_\delta(\boldsymbol{x}-\boldsymbol{y}) = \phi((\boldsymbol{x}-\boldsymbol{y})/\delta)$, where ϕ is a univariate and nonnegative function and supported in $[0,1]$. Obviously, the support radius of the weight function is δ.

Remark 2.2. The use of CSW can significantly reduce the number of nonzero columns in matrix $\boldsymbol{P}^T\boldsymbol{W}$, thereby greatly reducing the computational time for forming the systems (2.2). However, it should be noted that for each approximation, it is necessary to detect which points in X belong to the support of CSW, which is called the neighborhood search in this paper and can be time-consuming for massive or high-dimensional data. In problems require multiple approximations, the total time can be unacceptable.

The fill distance of X in Ω is defined as $h = h_{X,\Omega} := \sup\limits_{\boldsymbol{x}\in\Omega} \min\limits_{1\le i\le N} \|\boldsymbol{x}-\boldsymbol{x}_i\|_2$ which denotes the radius of the largest ball which is completely contained in Ω and does not contain points in X. Then, using local polynomial reproduction theory, the following results can be obtained.

Lemma 2.1. Suppose that $\Omega \subseteq \mathbb{R}^d$ is compacted and satisfies an interior cone condition with angle $\theta \in (0,\ \pi/2)$ and radius $r > 0$. Fix $m \in \mathbb{N}$. There exists constants C_1, C_2 that can be calculated exactly. Let $h_0 = r/C_2$. Define Ω^* to

be the closure of $\bigcup_{\boldsymbol{x}\in\Omega} B(\boldsymbol{x}, 2C_2h_0)$. For any quasi-uniform $X \subseteq \Omega$ with $h \leq h_0$, let $\delta = 2C_2h$. Then for all $f \in C^{m+1}(\Omega^*)$, the system (2.2) is well-posed and there is

$$\|f - s_{f,X}\|_{L_\infty(\Omega)} \leq C_1 h^{m+1} |f|_{C^{m+1}(\Omega^*)},$$

where $s_{f,X}$ is MLS approximation in $\mathbb{P}_m^d$ employed a CSW whose radius of support is δ, and $|f|_{C^{m+1}(\Omega^*)} := \max_{|\alpha|=m+1} \|D^\alpha f\|_{L_\infty(\Omega^*)}$.

Constant C_1 depends on the weight function ϕ and some constants such as the dimension d, and $C_2 = \frac{16(1+\sin\theta)^2 m^2}{3\sin^2\theta}$. The results of Lemma 2.1 come from Corollary 4.8 and some conclusions in Section 4 in Wendland's book [3]. Interested readers can refer to the book for details.

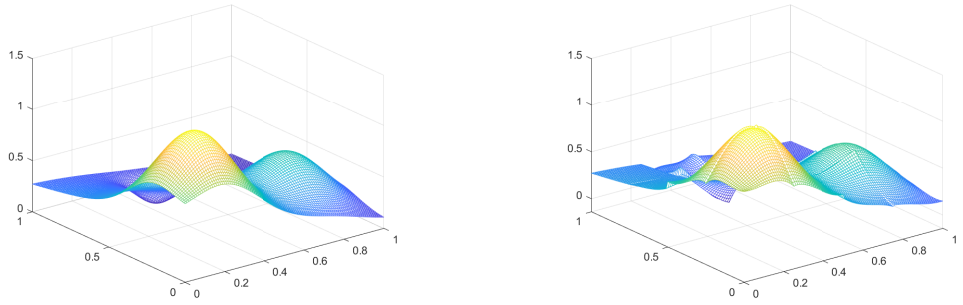
Remark 2.3. The famous approximation error bound depends on the use of CSW with the support radius of $\delta = 2C_2h$, which is crucial in the proof. Any CSW with a fixed-radius (including global weight functions) can not cause obvious convergence in experiments. Since C_2 depends on the cone condition satisfied by Ω, a constant multiple of h is recommended as the support radius in practice.

Since Lemma 2.1 ensures that the system (2.2) is well-posed, it theoretically provides a guarantee of the continuity of $s_{f,X}(\boldsymbol{x})$.

Corollary 2.1. If $\phi(\cdot)$ possesses k continuous derivatives then $s_{f,X}(\boldsymbol{x}) \in C^k(\mathbb{R}^d)$.

Therefore, the use of Wendland's compact support radial basis function [29] with certain continuity in the entire space as the weight function of MLS can give a continuous approximation function on Ω. In Section 1, we introduce a piece-wise moving least squares (PMLS) method proposed by Li et al. [19]. In PMLS, the "point-wise" in MLS is weakened to "piece-wise", so its continuity is difficult to be guaranteed. Fig. 1 shows the results of MLS and PMLS with 4×4 subspaces to approximate the famous Franke's test function [30]. It is obvious that the PMLS approximations are discontinuous among those subspaces.

Although increasing the number of the subspaces in PMLS can effectively improve the continuity of the surface visually, the time of calculating more coefficients and the memory for storing them would be increased at the same time. What is more, no matter how the number of subspaces is increased, there is no guarantee of the continuity of PMLS approximation on the whole space. The continuity of MLS approximation is very important in problems



(a) MLS approximation function to Franke's function

(b) PMLS approximation function with 4×4 subspaces to Franke's function

Fig. 1: Curve fitting results of MLS and PMLS.

such as numerical solution of PDEs [11, 12] and generalized moving least squares method [6–8]. Therefore, different from PMLS, this paper accelerate MLS by reducing the time to form system (2.2), which can not alter the approximation result.

2.2. Traditional octree

The concept of octree is originated by Hunter [20]. The traditional octree has a similar structure to the quadtree for two-dimensional data and can be regarded as a generalization of the quadtree for three-dimensional data. Sometimes, for easy understanding, the schematic diagram of a quadtree is used to represent the octree. In this subsection, we introduce the process of building a traditional octree and some key details.

Suppose that the data set $X_0 = \{\boldsymbol{x}_i\}_{i=1}^N,\ \forall i,\ \boldsymbol{x}_i \in \mathbb{R}^3$ is contained in a certain cuboid C_0, then we let $N_0 = \{X_0, C_0\}$ and call it the root node of the octree. As the tree grows from the root node to level 1, the cuboid C_0 is divided into 8 cuboids with a same size, $C_1, \cdots, C_8$, and then the data set X_0 is divided into $X_1, \cdots, X_8$ according to the cuboids, thus forming eight nodes $N_1, \cdots, N_8$ which are called the child nodes of N_0. Recursively divide the new nodes according to the same method to generate their child nodes until the new nodes satisfy a certain termination condition, and then mark them as leaf nodes. Once all the lowest-level nodes are leaf nodes, the algorithm stops, and the tree is built. We show the process of a parent node dividing into eight child nodes and the whole traditional octree structure in Fig. 2 and 3, respectively.

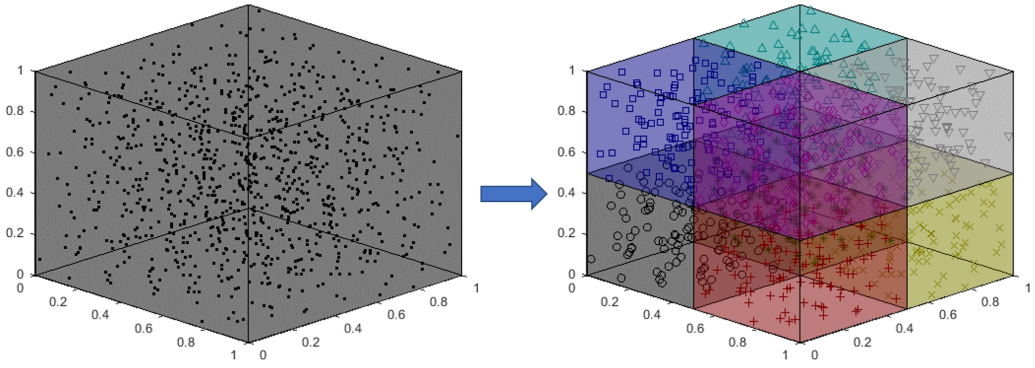


Fig. 2: Schematic diagram of the process from a parent node to eight child nodes.

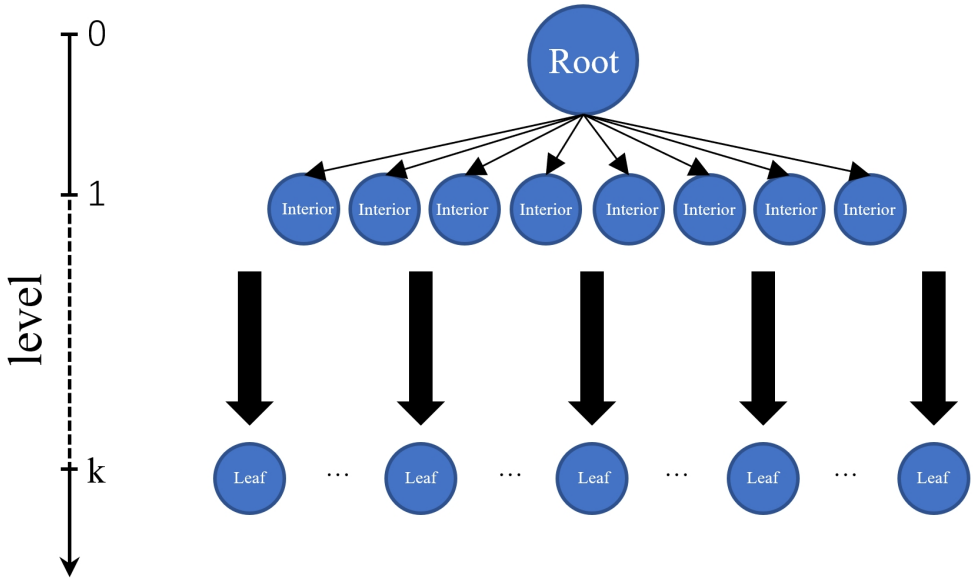


Fig. 3: Schematic Diagram of a traditional octree structure.

The division method of the octree is very direct, and the main factors determining its specific structure are the termination conditions and whether refinement is needed. Generally, the termination condition is reaching a specific depth D_{max} or keeping the number of points in the set X_i of each node N_i bigger than a lower bound. In order to minimize empty nodes in an octree as much as possible, the two termination conditions can also be combined [22, 23].

Since there may be still some empty nodes or leaf nodes distributed on multiple levels in the octree stopped by the above two conditions, the refinement is needed for some problems. At current, in many applications related with PDE, the octree need to satisfy the 1:2 balance between the levels of the adjacent nodes [24–26, 31] to make the spatial discretization relatively uniform.

In addition to the termination conditions, another key detail of the octree is the implementation and storage. Because of its tree-like structure, for each non-leaf node, in addition to the necessary information, it is seem that eight pointers need to be stored to point to its child nodes, which is considered to be memory-consuming for massive data. In order to reduce memory consumption, the octree is not stored as a tree in many problems [16, 26]. Generally, the specific form of the octree should be selected based on the characteristics of the problem.

Compared with the commonly used binary tree, the strategy to divide one node into eight makes the octree manage the data with a lower depth, and the process of dividing the data set also divides the space, which makes the octree commonly used in the following two problems.

- Search: For a given point $\boldsymbol{x}$, by relying on this tree structure, it can rapidly search the leaf node to which it belongs, which can then accelerate the search of the nearest point and the neighborhood search in X.

- Spatial decomposition: Since the spatial decomposition of the octree is based on the data point set X_0 and C_0, the cuboids to nodes can give multilevel decompositions [22, 23] of the space, and some of them can be regard as the mesh for C_0.

In this paper, our goal is to reduce the total computational time of MLS for massive and d-dimensional data by accelerating the neighborhood search.

Therefore, we construct a generalized octree suitable for d-dimensional data by replacing cuboids and the number of child nodes eight with hyper-cuboids and 2^d, respectively. The process can also refer to Fig. 3. In fact, the generalization is direct and introduced briefly in Sundar's and Lu et al.'s papers [16, 26]. However, the applications of the octree are currently focused on 2 and 3-dimensional data, and the detail problems, such as the termination condition and implementation, need to be solved to apply the generalized octree specifically to MLS.

Remark 2.4. The neighborhood search based on an octree involves the problem of searching the adjacent nodes of leaf nodes. Since traditional octrees only store parent-child relationships between nodes, searching for the adjacent nodes necessitates both down-top to higher-level nodes and then top-down to leaf nodes, which is complicated. Therefore, in order to improve the efficiency of the neighborhood search, it is necessary to design a method to quickly search the adjacent nodes of each node when constructing the specific generalized octree.

3. The acceleration method for moving least squares based on the generalized octree

In this section, we introduce the construction and the implementation details of the generalized octree which is named G-Octree and used to accelerate MLS in this paper. Then, based on G-Octree, we propose a fast neighborhood search method. Finally, we propose an acceleration method based on G-Octree and introduce the calculation process in detail.

3.1. Construction of the generalized octree

The aim of constructing the generalized octree, named G-Octree, in this paper is to efficiently complete the neighborhood search for a given point $\boldsymbol{x}$ in d-dimensional data set $X = \{\boldsymbol{x}_i\}_{i=1}^N$. For the cases of $d = 2$ and $d = 3$, G-Octree can automatically degrade into a specific quadtree and an octree.

Before constructing G-Octree, we suppose that the radius of the neighborhood search at $\boldsymbol{x}$ is $\boldsymbol{\delta}$, i.e. the search result is $X \bigcap B(\boldsymbol{x}, \boldsymbol{\delta})$ where $B(\boldsymbol{x}, \boldsymbol{\delta})$ is the d-dimensional open ball with the center at $\boldsymbol{x}$ and the radius of $\boldsymbol{\delta}$. The consumption of directly completing the search depends on N and d, which is not recommended for massive data.

Assuming that all nodes of G-Octree are $\{N_i\}_{i=1}^K$, and C_i is the corresponding hyper-cuboids of N_i which contains the data X_i. After building the

tree, one idea for quickly completing the neighborhood search is to find a hyper-cuboid covering $C(\boldsymbol{x}) = \bigcup_{i \in I(\boldsymbol{x})} C_i$ of $B(\boldsymbol{x}, \boldsymbol{\delta})$, i.e. $B(\boldsymbol{x}, \boldsymbol{\delta}) \subset C(\boldsymbol{x})$, and then determining each of the points stored in C_i belonging to $C(\boldsymbol{x})$ is inside $B(\boldsymbol{x}, \boldsymbol{\delta})$ or not, thereby reducing the number of points that need to be determined.

In Section 2.2, we point out that based on the traditional octree the node where $\boldsymbol{x}$ is located can be obtain quickly, so a hyper-cuboid covering $C(\boldsymbol{x})$ can be obtained by expanding the node by the adjacent relationship. However, in the traditional octree, only the parent-child relationships are stored. Moreover, for high-dimensional data, the adjacent relationships among nodes can be very complex, which implies that the expansion would be complex, too. Therefore, the first key point in constructing G-Octree is to enable the adjacent nodes to be efficiently searched to obtain a suitable hyper-cuboid covering.

When looking for a hyper-cuboid covering $C(\boldsymbol{x})$, considering nodes at different levels would increase the difficulty of expansion, thereby increasing the amount of calculation, so G-Octree constructed in this paper uses a specified depth D_{max} as the termination condition. Then the tree constructed in this way has the following characteristics:

(1) If a certain interior node N_j with $X_j = \emptyset$, then it still has 2^d child nodes $\{N_{j_i}\}_{i=1}^{2^d}$, and for all N_{j_i} there is $X_{j_i} = \emptyset$.

(2) In level 1, there are 2^d nodes $\{N_i\}_{i=1}^{2^d}$. In level 2, there are 2^{2d} nodes $\{N_{2^d+i}\}_{i=1}^{2^{2d}}$. Analogically, in level l, there are 2^{ld} nodes $\{N_{\left(\sum_{j=1}^{l-1} 2^{jd}\right)+i}\}_{i=1}^{2^{ld}}$.

(3) For any $l \leq D_{max}$, for all nodes in level l, $\{N_{\left(\sum_{j=1}^{l-1} 2^{jd}\right)+i}\}_{i=1}^{2^{ld}}$, there are $\bigcup_{1 \leq i \leq 2^{ld}} X_{\left(\sum_{j=1}^{l-1} 2^{jd}\right)+i} = X_0 = X$ and $\bigcup_{1 \leq i \leq 2^{ld}} C_{\left(\sum_{j=1}^{l-1} 2^{jd}\right)+i} = C_0$.

In other words, G-Octree constructed in this paper is a complete generalized octree with consistent the leaf nodes in a same level.

Characteristics (1) seems to imply that such the tree waste some memory, but if the data is a little uniform and D_{max} is seriously selected, the problem can be avoided. We discuss how to set D_{max} in Section 3.2. From Characteristics (2), it can be found out that although the octree is a tree-like structure in thought, the information contained in the tree can be stored linearly. In particular, in the traditional octree, sometimes it is necessary to

store the level and category (root, interior and leaf) of each node, but in the constructed G-Octree, those can be obtained according to the index of node, thus saving the corresponding memory. Combined with Characteristics (3), we can point out that the initial data set X and the hyper-cuboid C_0 can be accurately deduced by the information in each level's nodes. Because the division from farther node to child nodes is uniform in every dimension, if the each level's nodes are sorted in a specific order, the corresponding hyper-cuboid C_i can be calculated with no error by C_0. That is to say, for the generalized octree constructed with a specific depth D_{max} as the termination, only the initial hyper-cuboid C_0, the order of storage of nodes and the corresponding X_i (in fact, only the indexes of $\boldsymbol{x}_j$ are needed) are necessary. The all information about the tree can be deduced with no error by them.

Remark 3.1. In fact, the information that must to be stored to build G-Octree includes X ($d \times N$ double-precision floating-point numbers), C_0 ($d \times 2$ double-precision floating-point numbers), D_{max} (one integer) and $2^{dD_{max}}$ index vectors $\{I_i\}$ (the total are N integers), so that the total memory can be estimated easily, which means that G-Octree can avoid the problem of excessive memory in some traditional octrees.

Next, we begin to explain the details of building G-Octree. First, start from the root node being divided into 2^d child nodes. To facilitate understanding and programming implementation, we define a d-dimensional tensor T (or d-dimensional array) pointing to the index vector set $\{I_i\}$.

Remark 3.2. In C language, the tensor T can be used to store pointers, and in other languages, it can be used to store the index of the index vector set $\{I_i\}$. The use of T is to correspond the index of N_i to its spatial position. In fact, if $\{I_i\}$ is arranged in a specific order, T can be omitted, so it is not necessary.

In our mind, we store each child node in the corresponding position of the tensor T according to its spatial position, and in fact only store the index of the index vector corresponding to this child node. Therefore, when storing the nodes in level 1, we use a d-dimensional tensor T_1 whose size is $2 \times 2 \times 2 \times \cdots \times 2$. Due to the use of the tensor, we introduce a multi-index $\boldsymbol{\alpha} \in \mathbb{N}_0^d$, then the nodes in level 1 can be represented as $\{N_{\boldsymbol{\alpha}}^{(1)}\}_{\boldsymbol{\alpha} \in \{1,2\}^d}$. Next, we explain the hyper-cuboids and stored data corresponding to these nodes.

Suppose that $C_0 = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) \leq \boldsymbol{x}(i) \leq \boldsymbol{x}_{max}(i),\ 1 \leq i \leq d\}$, where $\boldsymbol{x}(i)$ is the i-th component of the vector $\boldsymbol{x}$. The size of the hyper-cuboid

can be represented by a d-dimensional edge vector $\boldsymbol{h}_0 = \boldsymbol{x}_{max} - \boldsymbol{x}_{min}$. Since each dimension is evenly divided, the edge vector of the hyper-cuboids corresponding to the child nodes is $\boldsymbol{h}_1 = \boldsymbol{h}_0/2$. There is a unique one among all child nodes whose lower limit coincides with the lower limit of $\boldsymbol{C}_0$, we denote it as $N^{(1)}_{(1,\cdots,1)}$, and the corresponding hyper-cuboid is $\boldsymbol{C}^{(1)}_{(1,\cdots,1)} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) \leq \boldsymbol{x}(i) \leq (\boldsymbol{x}_{min} + \boldsymbol{h}_1)(i),\ 1 \leq i \leq d\}$. Considering the distribution of those nodes in space, it can be induced that $\forall \boldsymbol{\alpha} \in \{1,2\}^d$, the hyper-cuboid corresponding to $N^{(1)}_{\boldsymbol{\alpha}}$ is

$$\boldsymbol{C}^{(1)}_{\boldsymbol{\alpha}} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i) - 1)\,\boldsymbol{h}_1(i) \leq \boldsymbol{x}(i) \leq \boldsymbol{x}_{min}(i) + \boldsymbol{\alpha}(i)\boldsymbol{h}_1(i),\ 1 \leq i \leq d\}. \tag{3.1}$$

According to the hyper-cuboids, the data set X is divided into $\{X^{(1)}_i\}_{i=1}^{2^d}$, and the corresponding index vectors are $\{I^{(1)}_i\}_{i=1}^{2^d}$. Suppose that $\forall \boldsymbol{\alpha} \in \{1,2\}^d$, if the index vector corresponding to $\boldsymbol{C}^{(1)}_{\boldsymbol{\alpha}}$ is $I^{(1)}_i$, then let

$$T_1(\boldsymbol{\alpha}) = i (\text{or Use pointer } T_1(\boldsymbol{\alpha}) \to I^{(1)}_i). \tag{3.2}$$

Through the above, it can be found that $\forall \boldsymbol{\alpha} \in \{1,2\}^d$, the hyper-cuboid $\boldsymbol{C}^{(1)}_{\boldsymbol{\alpha}}$ can be deduced according to $\boldsymbol{C}_0$, and the points stored in it can be obtained through $I^{(1)}_{T_1(\boldsymbol{\alpha})}$, which indicates that the information of level 1 is contained in X, $\boldsymbol{C}_0$, $\{I^{(1)}_i\}_{i=1}^{2^d}$ and T_1.

Next, we describe the growth process of the tree from level $l = k$ to level $l = k+1$. At the beginning of the growth, T_k and $\{I^{(k)}_i\}_{i=1}^{2^{kd}}$ are available, where T_k is a d-dimensional tensor whose size is $2^k \times 2^k \times 2^k \times \cdots \times 2^k$. The growth is completed by calculating the d-dimensional tensor T_{k+1} and the corresponding $\{I^{(k+1)}_i\}_{i=1}^{2^{(k+1)d}}$. The process can be done by going through $\boldsymbol{\alpha} \in \{1,2,\cdots,2^k\}^d$ and dividing the node $N^{(k)}_{\boldsymbol{\alpha}}$ into 2^d child nodes.

$\forall \boldsymbol{\alpha} \in \{1,2,\cdots,2^k\}^d$, the hyper-cuboid corresponding to the node $N^{(k)}_{\boldsymbol{\alpha}}$ is

$$\boldsymbol{C}^{(k)}_{\boldsymbol{\alpha}} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i) - 1)\,\boldsymbol{h}_k(i) \leq \boldsymbol{x}(i) \leq \boldsymbol{x}_{min}(i) + \boldsymbol{\alpha}(i)\boldsymbol{h}_k(i),\ 1 \leq i \leq d\}, \tag{3.3}$$

where $\boldsymbol{h}_k = \boldsymbol{h}_0/2^k$. Divide the hyper-cuboid into two parts in each dimension to obtain the hyper-cuboids corresponding to 2^d child nodes. Refer to formula (3.1) and use the multi-index $\boldsymbol{\beta} \in \{1,2\}^d$ to correspond to these child nodes. $\forall \boldsymbol{\beta} \in \{1,2\}^d$, let the corresponding hyper-cuboid $\boldsymbol{C}^{(k+1)}_{\boldsymbol{\alpha}_{\boldsymbol{\beta}}}$, then its specific

expression is

$$C_{\boldsymbol{\alpha}\boldsymbol{\beta}}^{(k+1)} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_k(i) + (\boldsymbol{\beta}(i)-1)\,\boldsymbol{h}_{k+1}(i) \le \boldsymbol{x}(i) \le \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_k(i) + \boldsymbol{\beta}(i)\boldsymbol{h}_{k+1}(i),\ 1 \le i \le d\}. \quad (3.4)$$

Since $\boldsymbol{h}_k = 2\boldsymbol{h}_{k+1}$, it can be calculated that

$$\begin{aligned} C_{\boldsymbol{\alpha}\boldsymbol{\beta}}^{(k+1)} =& \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (2\,(\boldsymbol{\alpha}(i)-1) + \boldsymbol{\beta}(i) - 1)\,\boldsymbol{h}_{k+1}(i) \le \boldsymbol{x}(i) \le \boldsymbol{x}_{min}(i) \\ & + (2\,(\boldsymbol{\alpha}(i)-1) + \boldsymbol{\beta}(i))\,\boldsymbol{h}_{k+1}(i),\ 1 \le i \le d\}, \\ =& C_{2(\boldsymbol{\alpha}-1)+\boldsymbol{\beta}}^{(k+1)}. \end{aligned}$$

Obviously, let

$$\boldsymbol{\gamma} = \boldsymbol{\alpha}_{\boldsymbol{\beta}} = 2(\boldsymbol{\alpha}-1) + \boldsymbol{\beta} \quad (3.5)$$

to get the index in T_{k+1} for the $\boldsymbol{\beta}$-th child node of $\boldsymbol{\alpha}$ in T_k and the corresponding hyper-cuboid $C_{\boldsymbol{\gamma}}^{(k+1)}$. Go through $\boldsymbol{\beta} \in \{1,2\}^d$, divide the data points corresponding to the index vector $I_{T_k(\boldsymbol{\alpha})}^{(k)}$ according to $C_{\boldsymbol{\gamma}}^{(k+1)}$ into $I_i^{(k+1)}$, and let $T_{k+1}(\boldsymbol{\gamma}) = i$ to finish the division of the node with index $\boldsymbol{\alpha}$ in level k.

Next, Go through $\boldsymbol{\alpha} \in \{1,2,\cdots,2^k\}^d$, repeat the above child node division process to finish the growth of the tree from level $l = k$ to level $l = k+1$.

Repeat the above growth process until $l = \boldsymbol{D}_{max}$, then the corresponding d-dimensional tensor $T_{D_{max}}$ and $\{I_i^{(D_{max})}\}_{i=1}^{2^{D_{max}d}}$ are obtained and G-Octree is built. The overall process is shown in Algorithm 1. In specific implementation, techniques such as variable overwrites and special storage order can also be used to save some memory and slightly improve computational efficiency. Interested readers can refer to the constructor 'MyOctree(X,box,delta0)' in the class definition file 'MyOctree.m' in the supplement and my GitHub repository[1].

3.2. Neighborhood search method based on G-Octree

In the previous section, we construct a special generalized octree, G-Octree, with a regular structure and low memory consumption. In this section, we detail how to perform the neighborhood search in MLS efficiently based on G-Octree.

[1] https://github.com/SanpengZheng/G-Octree-By-ZSP

Algorithm 1 Build G-Octree

Input: data set X, initial hyper-cuboid C_0, maximum depth D_{max};
Initialization: $l=0$, $T_0=1$, $I_1^0=\{1,2,\cdots,N\}$;

```
while k < D_max do
  Define the d-dimensional tensor T_{l+1} whose size is 2^{l+1} × 2^{l+1} × 2^{l+1} × ⋯ × 2^{l+1};
  Define 2^{(l+1)d} empty index vectors {I_i^{(l+1)}}_{i=1}^{2^{(l+1)d}};
  Set i = 1;
  for all α ∈ {1,2,⋯,2^l}^d do
    for all β ∈ {1,2}^d do
      Set γ = 2(α − 1) + β;
      Calculate C_γ^{(l+1)} referring to (3.4);
      for all j ∈ I_{T_l(α)}^{(l)} do
        if x_j ∈ C_γ^{(l+1)} then
          I_i^{(l+1)} = I_i^{(l+1)} ∪ {j};
        end if
      end for
      Set T_{l+1}(γ) = i;
      Set i = i + 1;
    end for
  end for
end while
```

Output: tensor $T_{D_{max}}$, index vector set $\{I_i^{(D_{max})}\}_{i=1}^{2^{D_{max}d}}$;

The neighborhood search at a given point $\boldsymbol{x}$ in X is detecting which points within set X belong to the support of the compact support weight functions in MLS, i.e. calculating $X \bigcap B(\boldsymbol{x}, \boldsymbol{\delta})$. The main process in this paper is as follows:

1. For the given $\boldsymbol{x}$, obtain the hyper-cuboid $C_{\boldsymbol{\alpha}}$ which contains $\boldsymbol{x}$ relied on G-Octree.

2. Obtain a suitable hyper-cuboid covering $C(\boldsymbol{x})$ of the d-dimensional ball $B(\boldsymbol{x}, \boldsymbol{\delta})$ by expanding $C_{\boldsymbol{\alpha}}$ relied on the adjacent relationship.

3. Go though the points in the hyper-cuboids belonging to $C(\boldsymbol{x})$, determine whether the points are in $B(\boldsymbol{x}, \boldsymbol{\delta})$, and obtain indexes of the inside points.

As mentioned in Section 3.1, D_{max} should be set seriously. This is a trade-off problem: the larger the D_{max} is, the more leaf nodes are, which means the tree-building takes longer time. But at this time, the smaller hyper-cuboid covering $C(\boldsymbol{x})$ of $B(\boldsymbol{x}, \boldsymbol{\delta})$ can be obtained, so that there are fewer points to be determined, which means the neighborhood search takes shorter time. Considering that a large D_{max} would cause the hyper-cuboid to be smaller and the composition of $C(\boldsymbol{x})$ to be very complex, which means that it takes a lot of time to obtain $C(\boldsymbol{x})$, so this paper recommends a small D_{max}, which is defined as follows:

Definition 3.1. D_{max} is the maximum depth that makes all the elements in the edge vector of the hyper-cuboids corresponding to leaf nodes greater than $\boldsymbol{\delta}_0$. In other words, D_{max} is such that $\forall\ 1 \leq i \leq d$, $\boldsymbol{h}_{D_{max}}(i) > \boldsymbol{\delta}_0$ but $\exists\ 1 \leq i \leq d$, $\boldsymbol{h}_{D_{max}+1}(i) \leq \boldsymbol{\delta}_0$.

Remark 3.3. It should be pointed out that $\boldsymbol{\delta}_0$ is used to define D_{max} instead of the radius $\boldsymbol{\delta}$ of the neighborhood search. This is because in MLS, the support radius $\boldsymbol{\delta}$ of weight function may be increased or decreased to make the normal equation well-posed or to reduce approximation errors. When $\boldsymbol{\delta}$ changes, rebuilding the G-Octree is undoubtedly inefficient. In the following, a neighborhood search method for $\boldsymbol{\delta}$ would be proposed which is only based on the G-Octree built by $\boldsymbol{\delta}_0$. Particularly, considering the well-posedness of MLS, $\boldsymbol{\delta}$ should not be set too small, so $\boldsymbol{\delta}_0$ should not be set to be very small, which means that D_{max} would not be large and there would not be many empty nodes in G-Octree. In the numerical experiment section, an empirical value for $\boldsymbol{\delta}_0$ would be proposed.

To locate the nodes to which $\boldsymbol{x}$ belongs, we give the following proposition.

Proposition 3.1. In G-Octree with D_{max} defined as Definition 3.1, $\forall\ \boldsymbol{x} \in C_0$ and $\forall 1 \le l \le D_{max}$, there exists $\boldsymbol{\alpha} \in \{1,2,\cdots,2^l\}^d$, such that $\boldsymbol{x} \in C_{\boldsymbol{\alpha}}^{(l)}$ and

$$\boldsymbol{\alpha}(i) = \left\lceil \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)} \right\rceil,\ \forall 1 \le i \le d. \tag{3.6}$$

Proof. According to formula (3.3), there is

$$C_{\boldsymbol{\alpha}}^{(l)} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i) - 1)\,\boldsymbol{h}_l(i) \le \boldsymbol{x}(i) \le \boldsymbol{x}_{min}(i) + \boldsymbol{\alpha}(i)\boldsymbol{h}_l(i),\ 1 \le i \le d\}.$$

For any $1 \le i \le d$, it is easy to verify that

$$\left\lceil \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)} \right\rceil - 1 \le \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)}. \tag{3.7}$$

Let the lower bound of the i-th component of $C_{\boldsymbol{\alpha}}^{(l)}$ be $LB(i)$, then according to formula (3.6) and inequality (3.7), there is

$$\begin{aligned} LB(i) &= \boldsymbol{x}_{min}(i) + \left(\left\lceil \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)} \right\rceil - 1 \right) \boldsymbol{h}_l(i) \\ &\le \boldsymbol{x}_{min}(i) + \boldsymbol{x}(i) - \boldsymbol{x}_{min}(i) \\ &= \boldsymbol{x}(i). \end{aligned}$$

According to the definition of the ceiling, we have

$$\left\lceil \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)} \right\rceil \ge \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)}. \tag{3.8}$$

Then

$$\begin{aligned} UB(i) &= \boldsymbol{x}_{min}(i) + \left\lceil \frac{(\boldsymbol{x} - \boldsymbol{x}_{min})(i)}{\boldsymbol{h}_l(i)} \right\rceil \boldsymbol{h}_l(i) \\ &\ge \boldsymbol{x}_{min}(i) + \boldsymbol{x}(i) - \boldsymbol{x}_{min}(i) \\ &= \boldsymbol{x}(i). \end{aligned}$$

Therefore, the i-th component of $\boldsymbol{x}$ is in the i-th range of $C_{\boldsymbol{\alpha}}^{(l)}$, which means $\boldsymbol{x} \in C_{\boldsymbol{\alpha}}^{(l)}$. □

Remark 3.4. According to Proposition 3.1, we can not only locate the leaf node to which $\boldsymbol{x}$ belongs but locate the interior nodes, which provides convenience for the neighborhood search and some multilevel methods. In traditional octrees, locating the node to which $\boldsymbol{x}$ belongs by the parent-child relationship needs d comparisons at each level, which amounts to a total of $d \cdot D_{max}$ comparisons. However, the method in Proposition 3.1 only needs d divisions and ceilings to obtain the corresponding node, and is independent of the number of levels.

Next, consider the adjacent nodes of $C_{\boldsymbol{\alpha}}^{(l)}$. Since the hyper-cuboids at the same level are the same size and fill C_0, the following conclusion is easy to obtain.

Proposition 3.2. In G-Octree, $\forall 1 \leq l \leq D_{max}$, $\forall \boldsymbol{\alpha} \in \{1,2,\cdots,2^l\}^d$ and $\boldsymbol{\beta} \in \{-1,0,1\}^d$ that are not all zero, if $\boldsymbol{\alpha}+\boldsymbol{\beta} \in \{1,2,\cdots,2^l\}^d$, then $C_{\boldsymbol{\alpha}}^{(l)}$ and $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ are adjacent.

This result in Proposition 3.2 is very common in 2 and 3-dimensional spaces, and here it is only extended to d-dimensional space. It indicates that at any level, there are 3^d-1 adjacent nodes for the inside nodes (8 in 2 dimensions, 26 in 3 dimensions), and the number for the nodes near the boundary of C_0 would be decreased.

Let

$$\overline{C_{\boldsymbol{\alpha}}^{(l)}} := \bigcup_{\boldsymbol{\beta} \in \{-1,0,1\}^d} C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}.$$

For inside nodes, it represents the union of $C_{\boldsymbol{\alpha}}^{(l)}$ and its adjacent hyper-cuboids. For nodes near the boundary, in addition to adjacent hyper-cuboids, some virtual hyper-cuboids outside C_0 are also added.

It is easy to know that $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ is also a hyper-cuboid, and it can be expressed as

$$\begin{aligned}\overline{C_{\boldsymbol{\alpha}}^{(l)}} = \{\boldsymbol{x} \in \mathbb{R}^d | \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i)-2)\,\boldsymbol{h}_l(i) \leq \boldsymbol{x}(i) \leq \boldsymbol{x}_{min}(i) \\ + (\boldsymbol{\alpha}(i)+1)\,\boldsymbol{h}_l(i),\ 1 \leq i \leq d\}. \quad (3.9)\end{aligned}$$

Define

$$R_l := \min_{i \in \{1,\cdots,d\}} \boldsymbol{h}_l(i), \quad \forall 0 \leq l \leq D_{max}.$$

Based on the G-Octree construction method in Section 3.1, there is

$$R_l = 2^l \times R_0 = 2^l \times \min_{i\in\{1,\cdots,d\}} \boldsymbol{h}_0(i).$$

Based on the above symbols, we can obtain the following proposition.

Proposition 3.3. In G-Octree, for every $B(\boldsymbol{x},\boldsymbol{\delta})$ with $\boldsymbol{x}\in C_0$ and $\boldsymbol{\delta}\le R_1$, suppose $\boldsymbol{x}\in C_{\boldsymbol{\alpha}}^{(l)}$, $\forall\ 1\le l\le D_{max}$. Then there exist a positive integer $l_{\boldsymbol{\delta}}\le D_{max}$ such that

$$B(\boldsymbol{x},\boldsymbol{\delta})\subset\overline{C_{\boldsymbol{\alpha}}^{(l_\delta)}}$$

is true, and if $l_{\boldsymbol{\delta}} < D_{max}$,

$$B(\boldsymbol{x},\boldsymbol{\delta})\subset\overline{C_{\boldsymbol{\alpha}}^{(l_\delta+1)}}$$

may be false.

Proof. Since the sequence $\{R_l\}_{l=1}^{D_{max}}$ is monotonically decreasing, the following optimization problem must have a solution.

$$\begin{aligned}&\min_{l\in\{1,\cdots,D_{max}\}} R_l\\ &\text{subject to}: \boldsymbol{\delta}\le R_l.\end{aligned}\tag{3.10}$$

We let $l_{\boldsymbol{\delta}}$ be the solution of (3.10). It is obvious that $\boldsymbol{\delta}\le R_{l_\delta}$.

For every $\boldsymbol{y}\in B(\boldsymbol{x},\boldsymbol{\delta})$, there is $\|\boldsymbol{y}-\boldsymbol{x}\|_2<\boldsymbol{\delta}$. So $\forall 1\le i\le d$, there is

$$\boldsymbol{x}(i)-\boldsymbol{\delta}\le\boldsymbol{y}(i)\le\boldsymbol{x}(i)+\boldsymbol{\delta}.$$

Because $\boldsymbol{x}\in C_{\boldsymbol{\alpha}}^{(l_\delta)}$, then it can be found from (3.3) that, $\forall 1\le i\le d$, there is

$$\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_{l_\delta}(i)\le\boldsymbol{x}(i)\le\boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_{l_\delta}(i).$$

Based on the two formulas and $\boldsymbol{\delta}\le R_{l_\delta}$, it is not difficult to obtain

$$\begin{aligned}\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_{l_\delta}(i)-\boldsymbol{\delta}\le&\boldsymbol{y}(i)\le\boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_{l_\delta}(i)+\boldsymbol{\delta},\\ \boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_{l_\delta}(i)-R_{l_\delta}\le&\boldsymbol{y}(i)\le\boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_{l_\delta}(i)+R_{l_\delta},\\ \boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-2)\,\boldsymbol{h}_{l_\delta}(i)\le&\boldsymbol{y}(i)\le\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)+1)\boldsymbol{h}_{l_\delta}(i).\end{aligned}$$

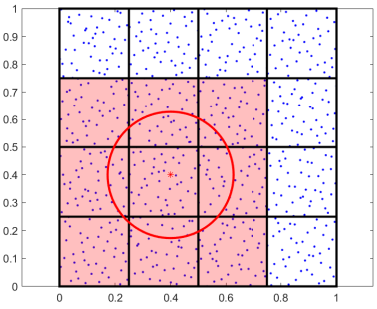
It can be found from the formula (3.9) that $\boldsymbol{y}$ is contained in $\overline{C_{\boldsymbol{\alpha}}^{(l_\delta)}}$, which means that $B(\boldsymbol{x},\boldsymbol{\delta})\subset\overline{C_{\boldsymbol{\alpha}}^{(l_\delta)}}$ is true.

If $l_\delta < D_{max}$, then it can be known from the optimization problem (3.10) that $\boldsymbol{\delta} > R_{l_\delta+1}$. We suppose j makes $\boldsymbol{h}_{l_\delta+1}(j) = R_{l_\delta+1}$. Then for $\boldsymbol{x}^* \in \boldsymbol{C}_{\boldsymbol{\alpha}}^{(l_\delta+1)}$ with $\boldsymbol{x}^*(j) = \boldsymbol{x}_{min}(j) + \boldsymbol{\alpha}(j)\boldsymbol{h}_{l_\delta+1}(j)$, let

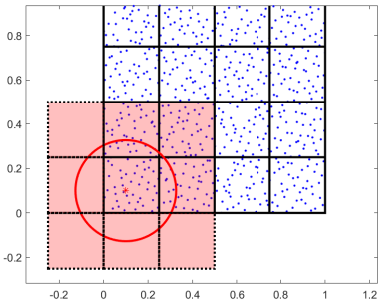
$$\boldsymbol{y}^*(i) = \begin{cases} \boldsymbol{x}^*(i) + \frac{\delta + R_{l_\delta+1}}{2}, & \text{if } i = j; \\ \boldsymbol{x}^*(i), & \text{if } i \neq j. \end{cases}$$

It is easy to verify that $\boldsymbol{y}^*$ is contained in $B(\boldsymbol{x}^*, \boldsymbol{\delta})$ but $\boldsymbol{y}^*$ is not contained in $\overline{\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l_\delta+1)}}$, which means that $B(\boldsymbol{x}, \boldsymbol{\delta}) \subset \overline{\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l_\delta+1)}}$ may be false. □

We take the division of $d = 2$ (quadtree) as an example of $\overline{\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l_\delta)}}$, and show the cases in Fig. 4 that $\boldsymbol{x}$ is inside and nearby the boundary of $\boldsymbol{C}_0$, respectively. The dashed rectangle in Fig. 4(b) represents the virtual hyper-cuboid outside $\boldsymbol{C}_0$, whose multi-index $\boldsymbol{\alpha} + \boldsymbol{\beta}$ is not in $\{1, 2, \cdots, 2^l\}^d$, and thus can be directly skipped in actual computations.



(a) $\boldsymbol{x}$ belongs to an inside node



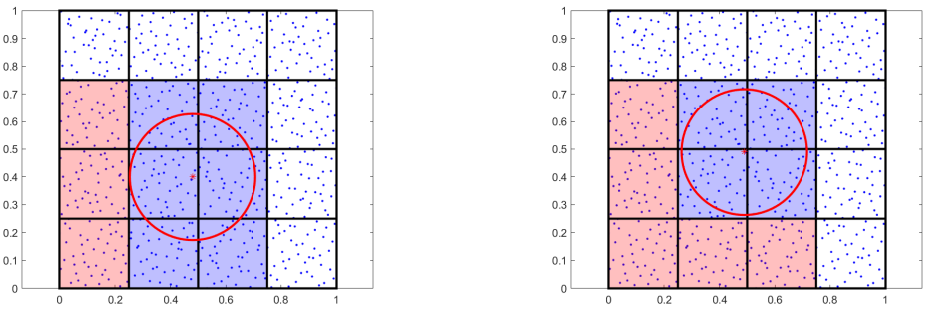
(b) $\boldsymbol{x}$ belongs to a node nearby the boundary of $\boldsymbol{C}_0$

Fig. 4: The diagrams of the relationship between $B(\boldsymbol{x}, \boldsymbol{\delta})$ and $\overline{\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l_\delta)}}$ in two cases.

As mentioned at the beginning of Section 3.1, the complexity of the adjacent relationships among nodes at different levels would significantly increase the time to obtain $\boldsymbol{C}(\boldsymbol{x})$. Therefore, we only use the nodes at a certain level l to form $\boldsymbol{C}(\boldsymbol{x})$, and the smallest covering among them is denoted as $\boldsymbol{C}_{opt}^{(l)}(\boldsymbol{x})$.

Based on the construction of G-Octree and Proposition 3.3, we can know that for all $1 \leq l \leq l_\delta$, $\overline{\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l)}}$ is a hyper-cuboid covering of $B(\boldsymbol{x}, \boldsymbol{\delta})$ which is composed only of $\boldsymbol{C}_{\boldsymbol{\alpha}}^{(l)}$ and its adjacent hyper-cuboids, but obviously, the

covering is not always $C_{opt}^{(l)}(\boldsymbol{x})$. We continue to use the quadtree as an example and show the relationship between $C_{opt}^{(l)}(\boldsymbol{x})$ with $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ in Fig. 5, where the blue rectangles form $C_{opt}^{(l)}(\boldsymbol{x})$ and would form $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ by adding the red rectangles.



(a) $\boldsymbol{x}$ is around the edge of a node

(b) $\boldsymbol{x}$ is around the corner of a node

Fig. 5: The diagrams of the relationship between $C_{opt}^{(l)}(\boldsymbol{x})$ and $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ in two cases.

It is obvious that the red rectangle is a subset of $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$. Taking the case in Fig. 5(b) as an example, $C_{opt}^{(l)}(\boldsymbol{x})$ only includes 4 rectangles, while $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ includes 9 rectangles! This means that using $C_{opt}^{(l)}(\boldsymbol{x})$ for neighborhood search is much more efficient than using $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$.

Because $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ is composed of 3^d hyper-cuboids, we can obtain $C_{opt}^{(l)}(\boldsymbol{x})$ with $l \leq l_{\delta}$. Therefore, the following proposition is proposed.

Proposition 3.4. In G-Octree, if $\boldsymbol{x} \in C_{\boldsymbol{\alpha}}^{(l)}$, then for any non-zero $\boldsymbol{\beta} \in \{-1,0,1\}^d$, there exists a points $\boldsymbol{x}_{\boldsymbol{\beta}} \in C_{\boldsymbol{\alpha}}^{(l)}$, s.t. the distance from $\boldsymbol{x}$ to $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ is $\|\boldsymbol{x}-\boldsymbol{x}_{\boldsymbol{\beta}}\|_2$, and $\forall 1 \leq i \leq d$ there is

$$\boldsymbol{x}_{\boldsymbol{\beta}}(i) = \begin{cases} \boldsymbol{x}_{min}(i) + (\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i), & \text{if } \boldsymbol{\beta}(i) = -1; \\ \boldsymbol{x}(i), & \text{if } \boldsymbol{\beta}(i) = 0; \\ \boldsymbol{x}_{min}(i) + \boldsymbol{\alpha}(i)\boldsymbol{h}_l(i), & \text{if } \boldsymbol{\beta}(i) = 1. \end{cases} \tag{3.11}$$

Proof. The distance from $\boldsymbol{x}$ to $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ can be represented by the following optimization problem.

$$\min_{\boldsymbol{y} \in C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}} \|\boldsymbol{x}-\boldsymbol{y}\|_2.$$

Because $C_{\boldsymbol{\alpha}}^{(l)}$ and $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ are adjacent closed hyper-cuboids, their intersection must be non-empty, which means that the solution to the above problem must be in the intersection. Therefore, the above problem is equivalent to

$$\min_{\boldsymbol{y}\in C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}} \|\boldsymbol{x}-\boldsymbol{y}\|_2 = \min_{\boldsymbol{y}\in C_{\boldsymbol{\alpha}}^{(l)}\bigcap C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}} \|\boldsymbol{x}-\boldsymbol{y}\|_2. \tag{3.12}$$

We know that

$$\begin{aligned} C_{\boldsymbol{\alpha}}^{(l)} =&\{\boldsymbol{x}\in\mathbb{R}^d | \boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i) \le \boldsymbol{x}(i) \le \boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_l(i), \\ &\qquad\qquad 1\le i\le d\}, \\ C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)} =&\{\boldsymbol{x}\in\mathbb{R}^d | \boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)+\boldsymbol{\beta}(i)-1)\,\boldsymbol{h}_l(i) \le \boldsymbol{x}(i) \le \boldsymbol{x}_{min}(i)+ \\ &\qquad\qquad (\boldsymbol{\alpha}(i)+\boldsymbol{\beta}(i))\,\boldsymbol{h}_l(i),\ 1\le i\le d\}. \end{aligned}$$

Obviously, $C_{\boldsymbol{\alpha}}^{(l)}\bigcap C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ can be described by the intersection of each component intervals.

$\forall 1\le i\le d$, if $\boldsymbol{\beta}(i)=-1$, then the i-component interval of $C_{\boldsymbol{\alpha}}^{(l)}$ and $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ is $[\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i), \boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_l(i)]$ and $[\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-2)\,\boldsymbol{h}_l(i), \boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i)]$, respectively. Obviously, the intersection of the two intervals is $\boldsymbol{x}(i)=\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i)$. Similarly, if $\boldsymbol{\beta}(i)=1$, the intersection is $\boldsymbol{x}(i)=\boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_l(i)$, and if $\boldsymbol{\beta}(i)=0$, then the intersection is the interval $[\boldsymbol{x}_{min}(i)+(\boldsymbol{\alpha}(i)-1)\,\boldsymbol{h}_l(i), \boldsymbol{x}_{min}(i)+\boldsymbol{\alpha}(i)\boldsymbol{h}_l(i)]$.

In summary, in $C_{\boldsymbol{\alpha}}^{(l)}\bigcap C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$, when $\boldsymbol{\beta}(i)=-1$ or $\boldsymbol{\beta}(i)=1$, the i-component must be the constant shown in (3.11), only when $\boldsymbol{\beta}(i)=0$, the i-component is in a interval which is the same as $C_{\boldsymbol{\alpha}}^{(l)}$. Therefore, $\boldsymbol{x}_{\boldsymbol{\beta}}$ defined as (3.11) is in $C_{\boldsymbol{\alpha}}^{(l)}\bigcap C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$, and it is obviously a solution to problem (3.12). □

With the way of calculating the distance, the following corollary is easy to obtain.

Corollary 3.1. In G-Octree, suppose that $\boldsymbol{x}\in C_{\boldsymbol{\alpha}}^{(l)}$ and $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}$ is a adjacent hyper-cuboids. Then $C_{\boldsymbol{\alpha}+\boldsymbol{\beta}}^{(l)}\bigcap B(\boldsymbol{x},\boldsymbol{\delta})\neq\emptyset$ if and only if $\|\boldsymbol{x}-\boldsymbol{x}_{\boldsymbol{\beta}}\|_2<\boldsymbol{\delta}$, where $\boldsymbol{x}_{\boldsymbol{\beta}}$ is defined as (3.11).

Corollary 3.1 provides a method for determining which adjacent hyper-cuboids of $\boldsymbol{x}\in C_{\boldsymbol{\alpha}}^{(l)}$ have an intersection with $B(\boldsymbol{x},\boldsymbol{\delta})$ at any level $l\le l_{\boldsymbol{\delta}}$.

Therefore, we can traverse the hyper-cuboids in $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ to obtain $C_{opt}^{(l)}(\boldsymbol{x})$ with $l \leq l_{\delta}$.

In the case of $l_{\delta} < D_{max}$, it can be known from Proposition 3.3 that to obtain $C_{opt}^{(l)}(\boldsymbol{x})$ with $l_{\delta} < l \leq D_{max}$, it is necessary to further determine which adjacent hyper-cuboids of $\overline{C_{\boldsymbol{\alpha}}^{(l)}}$ in level l intersect with $B(\boldsymbol{x}, \boldsymbol{\delta})$. The number of hyper-cuboids to be traversed would be increased from 3^d to at least 5^d, which means more calculations and time are required.

Based on the above discussion, we think $C_{opt}^{(l_{\delta})}(\boldsymbol{x})$ is a good hyper-cuboid covering for $B(\boldsymbol{x}, \boldsymbol{\delta})$ in G-Octree, because there are at most 3^d in it and $C_{opt}^{(l_{\delta})}(\boldsymbol{x}) \subseteq C_{opt}^{l}(\boldsymbol{x})$ holds for all $l \leq l_{\delta}$, obviously.

Therefore, $C_{opt}^{(l_{\delta})}(\boldsymbol{x})$ is employed to do the neighborhood search at $\boldsymbol{x}$ by determining whether the points contained in $C_{opt}^{(l_{\delta})}(\boldsymbol{x})$ are in $B(\boldsymbol{x}, \boldsymbol{\delta})$. The process of this section is shown in Algorithm 2, and interested readers can refer to the class method 'RangeSearch(obj,point,r)' in the class definition file 'MyOctree.m' in the attachment and my GitHub repository[2].

Remark 3.5. If $\boldsymbol{\delta} > R_1$, $B(\boldsymbol{x}, \boldsymbol{\delta})$ can not be contained in C_0 and we can use all the data for neighborhood search based on G-Octree and define $C_{opt}^{0}(\boldsymbol{x}) = C_0$. When $C_{opt}^{(l_{\delta})}(\boldsymbol{x})$ is employed to do the neighborhood search, it seems to require the information of nodes in level l_{δ}. However, as mentioned in Remark 3.1, the information of interior nodes can be completely induced from that of leaf nodes. Therefore, for all $\boldsymbol{\delta}$, to complete the neighborhood search, no information other than the outputs of Algorithm 1 is required, nor is it necessary to rebuild the G-Octree by changing $\boldsymbol{\delta}_0$.

3.3. The calculation of moving least squares based on G-Octree

In the previous two sections, we construct a special generalized octree, G-Octree, and provided a fast neighborhood search method based on it. Readers who are familiar with the moving least squares (MLS) method may already speculate how to use these results to accelerate MLS. To avoid ambiguity and ensure that readers can obtain the same results in our numerical experiments, we provide all details about calculation of MLS on G-Octree in this section.

[2]https://github.com/SanpengZheng/G-Octree-By-ZSP

Algorithm 2 Neighborhood search based on G-Octree

Input: Information of G-Octree $\{X, C_0, T, \{I_i\}_{i=1}^{2^{D_{max}d}}\}$, point to be searched $\boldsymbol{x}$, search radius $\boldsymbol{\delta}$;

Initialization: $I_{\boldsymbol{x}} = \emptyset$;

```
Calculate l_δ (refer to Proposition 3.3).
Calculate the multi-index α of the node where point x is located in level
l_δ (refer to formula (3.6));
for all β ∈ {−1,0,1}^d do
  if α+β ∈ {1,2,···,2^{l_δ}}^d then
    Calculate x_β (refer to formula (3.11));
    if ‖x − x_β‖_2 < δ then
      Calculate the multi-index set A which contains the multi-indexes of
      all child nodes of N^{(l_δ)}_{α+β} (refer to formula (3.5));
      for all j ∈ ⋃_{γ∈A} I_{T(γ)} do
        if ‖x − x_j‖_2 < δ then
          Add j into I_x;
        end if
      end for
    end if
  end if
end for
```

Output: Index vector of points in the neighborhood $I_{\boldsymbol{x}}$;

First, it is necessary to explain the polynomial space $\mathbb{P}_m^d$ used in MLS, as the choice of basis functions significantly affects the stability and accuracy of the approximation. At the approximation in $\boldsymbol{x}^*$ by MLS, only the points in $\boldsymbol{B}(\boldsymbol{x}^*, \boldsymbol{\delta})$ are used, which would cause the condition number of equation (2.2) too large if the basis functions are the classical power functions. Therefore, in massive data, we select the shifted and scaled polynomial basis recommended by Mirzaei et al. [6], and the formula is as following.

$$\left\{ \frac{(\boldsymbol{x} - \boldsymbol{x}^*)^{\boldsymbol{\alpha}}}{\boldsymbol{\delta}^{|\boldsymbol{\alpha}|}} \right\}_{0 \le |\boldsymbol{\alpha}| \le m}. \tag{3.13}$$

We call the set of bases $(\boldsymbol{x}^*, \boldsymbol{\delta})$-SSPB. It can be found from (3.13) that each non-constant basis $p_i(\boldsymbol{x})$ is constructed such that it takes the value 0 at the center $\boldsymbol{x}^*$ and ± 1 on the sphere surface of $\boldsymbol{B}(\boldsymbol{x}^*, \boldsymbol{\delta})$. This construction helps differentiate the values at distinct points within the ball. Both theoretical analysis and practical experience have shown that using $(\boldsymbol{x}^*, \boldsymbol{\delta})$-SSPB can effectively reduce the condition number of the normal equation, thereby enhancing the stability of the calculations.

After selecting the bases of $\mathbb{P}_m^d$, the next step is to explain how to calculate the coefficients of MLS efficiently. In Remark 2.2, we point out that the use of CSW can significantly reduce the time to solve the normal equation (2.2). If we use the method in Section 3.2 to obtain the index vector $\boldsymbol{I}_{\boldsymbol{x}^*} = \left(i_1, i_2, \cdots, i_{N_{\boldsymbol{x}^*}}\right)$, then it can be known that the solution of (2.2) is the same as the following equation.

$$\boldsymbol{P}_{\boldsymbol{x}^*}^\top \boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}^*) \boldsymbol{P}_{\boldsymbol{x}^*} \boldsymbol{c}(\boldsymbol{x}^*) = \boldsymbol{P}_{\boldsymbol{x}^*}^\top \boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}^*) \boldsymbol{F}_{\boldsymbol{x}^*}, \tag{3.14}$$

where $\boldsymbol{P}_{\boldsymbol{x}^*} \in \mathbb{R}^{Q \times N_{\boldsymbol{x}^*}}$ is the sampling matrix by $(\boldsymbol{x}^*, \boldsymbol{\delta})$-SSPB at the points corresponding to $\boldsymbol{I}_{\boldsymbol{x}^*}$, $\boldsymbol{F}_{\boldsymbol{x}^*}$ is the corresponding $N_{\boldsymbol{x}^*}$-dimensional sampling vector, and $\boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}) = \operatorname{diag}\left(\left(\theta(\boldsymbol{x}, \boldsymbol{x}_{i_1}), \cdots, \theta(\boldsymbol{x}, \boldsymbol{x}_{i_{N_{\boldsymbol{x}^*}}})\right)\right)$. By setting an appropriate $\boldsymbol{\delta}$, $N_{\boldsymbol{x}^*}$ is much smaller than N, so by comparing the scale of the matrices and vectors in (2.2) and (3.14), it can be intuitively found that solving (3.14) is more efficient.

Remark 3.6. Theoretically, for a given f, using the same $\boldsymbol{\delta}$ at different $\boldsymbol{x}^*$ can ensure the continuity of MLS approximation function. A large $\boldsymbol{\delta}$ would result in poor approximation errors, but a tiny $\boldsymbol{\delta}$ would cause the normal equation (3.14) to be ill-conditioned or singular at certain points. For a relatively uniform X, there are some empirical values [4, 5, 13] that can enable MLS

to avoid these two problems. When the empirical value performs poorly, adjusting $\boldsymbol{\delta}$ through experiments is also an acceptable strategy. Therefore, it is recommended to build G-Octree with D_{max} defined by an empirical value $\boldsymbol{\delta}_0$. This means that if it is necessary to adjust $\boldsymbol{\delta}$ around $\boldsymbol{\delta}_0$, $l_{\boldsymbol{\delta}}$ is likely to be D_{max} or $D_{max}-1$, i.e., the constructed G-Octree is not too deep.

After discussing the above, there is an efficient method to calculate MLS approximation p^* at $\boldsymbol{x}^*$ based on G-Octree at the beginning of scattered point set $X=\{\boldsymbol{x}_i\}_{i=1}^N$ and the corresponding sampling vector $\boldsymbol{F}=(f_1,\cdots,f_N)^T$. The whole process is shown in Algorithm 3.

Algorithm 3 Moving least squares method based on G-Octree

Input: Scattered point set X, sampling vector $\boldsymbol{F}$, point to be approximated $\boldsymbol{x}^*$, the degree of polynomials m, radius $\boldsymbol{\delta}$;
Preparation:(1) Calculate D_{max} (refer to Definition 3.1); (2) Build G-Octree (Algorithm 1);

Establish $(\boldsymbol{x}^*,\boldsymbol{\delta})$-SSPB $\{p_i(\boldsymbol{x})\}_{i=1}^Q$ (refer to (3.13));
Let $I_{\boldsymbol{x}^*}$ is the indexes of points in $X\bigcap B(\boldsymbol{x}^*,\boldsymbol{\delta})$ (Algorithm 2);
Calculate $\boldsymbol{P}_{\boldsymbol{x}^*}$, $\boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}^*)$ and $\boldsymbol{F}_{\boldsymbol{x}^*}$;
Let $\boldsymbol{c}^*$ is the solution to $\boldsymbol{P}_{\boldsymbol{x}^*}^\top\boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}^*)\boldsymbol{P}_{\boldsymbol{x}^*}\boldsymbol{c}(\boldsymbol{x}^*)=\boldsymbol{P}_{\boldsymbol{x}^*}^\top\boldsymbol{W}_{\boldsymbol{x}^*}(\boldsymbol{x}^*)\boldsymbol{F}_{\boldsymbol{x}^*}$ (directly or by pseudo-inverse);
Let $p^*=\sum_{i=1}^Q c_i^* p_i(\boldsymbol{x}^*)$;

Output: MLS approximation p^*;

It is worth noting that, in Algorithm 3, the building of G-Octree belongs to the preparation. In problems such as curve and surface fitting that require multiple approximations at different points, the built G-Octree can be stored, and only the main part of the algorithm is needed for subsequent approximations. For the specific implementation of Algorithm 3, interested readers can refer to the 'Example.m' or 'ExampleForGOctree.mlx' files in my GitHub repository[3].

4. Numerical experiments

In this section, we compare the results of the moving least squares (MLS) approximation by the proposed acceleration method based on G-Octree with

[3]https://github.com/SanpengZheng/G-Octree-By-ZSP

those by the acceleration method based on k-tree and direct computation in computational efficiency and approximation errors under data sets with various sizes and dimensions. Here the kd-tree and the neighborhood search based on the kd-tree is implemented by 'KDTreeSearcher' and 'rangesearch' in the Statistics and Machine Learning Toolbox of Matlab, respectively.

In addition to the classical MLS, we give the computational time and approximation errors of the piece-wise moving least squares (PMLS) by Li et al. [19] to show the characteristics and advantages of acceleration methods based on different ideas.

The D_{max} for building the G-Octree is calculated by Definition 3.1, where δ_0 is set to the empirical value of the support radius of the weight function in MLS as

$$\delta = \begin{cases} 2mh, & \text{if } d = 2; \\ (dQ/V)^{1/d}h, & \text{if } d \geq 3, \end{cases} \tag{4.1}$$

where Q is the dimension of the polynomial space and V is the volume of the d-dimensional unit ball. When the data distribution is relatively uniform, δ_0 is also chosen as the support radius of the weight function. In this case, to avoid the discontinuities in PMLS shown in Fig. 1(b), the subspaces in PMLS are hyper-cuboids which are circumscribed by a d-dimensional ball whose radius is δ_0 and the anchor points are the centers of the hyper-cuboids. Those settings can ensure that there may be an overlap of data for adjacent subspaces to weak the discontinuity of PMLS between subspaces.

The compactly supported C^4 Wendland's function $\phi(r) = (1-r)_+^6(35r^2 + 18r + 3)$ is employed in MLS. The chosen polynomial space is $\mathbb{P}_3^d$ which contains all polynomials whose degrees are less than or equal to 3.

All experiments in this section are carried out on our PC, the CPU is Intel(R) Core(TM) i7-12700K 3.60 GHz, and the programming software is Matlab R2024a. All experiments in this section can be reproduced using 'ExperimentForSec4_1.m', 'ExperimentForSec4_2.m', and 'ExperimentForSec4_3.m' in my GitHub repository.

4.1. Two-dimensional data—Franke's test function Data

In this section, the sampled function is the classical Franke's test function

$$f(x,y) = \frac{3}{4}\exp(\frac{(9x-2)^2+(9y-2)^2}{4}) + \frac{3}{4}\exp(-\frac{(9x-2)^2}{49} - \frac{(9y+1)^2}{10}) - \frac{1}{5}\exp(-(9x-4)^2-(9y-7)^2) + \frac{1}{2}\exp(\frac{(9x-7)^2+(9y-3)^2}{4}),$$

which is define in $\mathbb{R}^2$. Sampling point set $X = \{\boldsymbol{x}_i\}_{i=1}^N$ is Halton point set [32], which is quasi-scattered in $[0,\ 1]^2$ and whose filling distance can be approximately estimated as $h \approx N^{-1/2}$ [1], and the initial hyper-cuboid C_0 which includes those points is $[0,\ 1]^2$.

In this subsection, X with d =2 and $N = 10000$, 200000, 400000, 800000, 1600000 would be used. Due to the uniformity of the Halton point, under different N, $\boldsymbol{\delta}$ is set to the corresponding $\boldsymbol{\delta}_0$ as (4.1). For each of these sets, we generate 100 random points $\{\boldsymbol{y}_i\}_{i=100}^{100}$ in $[0,\ 1]^2$ for approximation. We record the preparation time (t_p), the single approximation time (t_s), and the approximation error for each time.

Under a certain sampling point set X, in the approximation at the point $\boldsymbol{y}_i$, the preparation time t_p is denoted as $t_p(i)$, the approximation time t_s is denoted as $t_s(i)$, the approximation value is denoted as p_i, and the error is defined as $e_i = |f(\boldsymbol{y}_i) - p_i|$. Then, under the point set, the mean of t_p is

$$\text{mean}(t_p) = \frac{1}{100}\sum_{i=1}^{100} t_p(i),$$

and the standard deviation (std) is

$$\text{std}(t_p) = \sqrt{\frac{1}{99}\sum_{i=1}^{100}(t_p(i) - \text{mean}(t_p))^2}.$$

The mean and std of the t_s and the approximation errors are calculated in the same way.

To verify the correctness of the acceleration methods based on G-Octree and kd-tree, and to compare the approximation capabilities of the two acceleration methods and PMLS, we present the mean and std of the approximation errors for each set in Table 1. The column corresponding to "classical"

represents the results of MLS without using acceleration methods. The results in Table 1 show that the mean and std of the approximation errors of MLS accelerated by G-Octree and kd-tree are exactly the same as the results without acceleration, which means that the two methods reduce the computational time without altering the results of MLS. Comparing the approximation errors of MLS and PMLS reveals that, under different sizes, the mean and std of the errors of PMLS are greater than those of MLS. Such results can be expected. PMLS only uses the approximation functions of the anchor points of subspaces, which inevitably leads to a decrease in approximation accuracy. Additionally, there are significant differences in errors around the anchor points and the boundaries of the subspaces, which increases the std of the errors.

Table 1: The mean and std of the approximation errors of MLS by different acceleration methods and PMLS under different sizes of set X

N	MLS						PMLS	
	classical		kd-tree		G-Octree			
	mean	std	mean	std	mean	std	mean	std
100000	7.89E-08	1.35E-07	7.89E-08	1.35E-07	7.89E-08	1.35E-07	4.33E-07	7.82E-07
200000	2.88E-08	4.79E-08	2.88E-08	4.79E-08	2.88E-08	4.79E-08	2.16E-07	4.40E-07
400000	4.94E-09	9.75E-09	4.94E-09	9.75E-09	4.94E-09	9.75E-09	3.66E-08	5.99E-08
800000	1.25E-09	1.52E-09	1.25E-09	1.52E-09	1.25E-09	1.52E-09	5.59E-09	1.23E-08
1600000	3.34E-10	4.44E-10	3.34E-10	4.44E-10	3.34E-10	4.44E-10	1.36E-09	2.44E-09

Next, we compare the computational time of PMLS and MLS under two acceleration methods. Here, the computational time is divided into the preparation time t_p and the single approximation time t_s. In the acceleration methods based on G-Octree and kd-tree, t_p mainly refers to the tree-building time, while in PMLS, t_p is the time for calculating and storing the approximation coefficients of all subspaces. The results of t_p and t_s are presented in Table 2 and 3, respectively.

As can be seen from Table 2, the mean of PMLS's t_p is the largest among data sets of various sizes. The mean of t_p has been inflated by 113 times when the data size increases by 16 times. This is because as the size of X increases, the division of subspaces in PMLS becomes denser, and more coefficients need to be precomputed. Combined with the results of the approximation errors of PMLS in Table 1, it can be found that our division is not excessive, which means that the t_p required by PMLS for massive data would be unacceptable, and it can be anticipated that t_p would also increase sharply with the increase

Table 2: The mean and standard deviation of the preparation time t_p (Unit: seconds) of MLS by different acceleration methods and PMLS under different sizes of set X

N	MLS						PMLS	
	classical		kd-tree		G-Octree			
	mean	std	mean	std	mean	std	mean	std
100000	–	–	1.04E-01	1.70E-03	3.09E-02	3.16E-03	2.28E+00	5.22E-02
200000	–	–	2.29E-01	1.27E-02	1.04E-01	9.98E-03	5.82E+00	2.04E-01
400000	–	–	5.06E-01	4.71E-02	1.41E-01	1.65E-02	1.78E+01	4.48E-01
800000	–	–	1.11E+00	2.48E-02	4.55E-01	8.49E-03	6.92E+01	1.08E+00
1600000	–	–	2.56E+00	8.91E-03	6.58E-01	3.47E-03	2.59E+02	1.65E+00

Table 3: The mean and standard deviation of the single approximation time t_s (Unit: seconds) of MLS by different acceleration methods and PMLS under different sizes of set X

N	MLS						PMLS	
	classical		kd-tree		G-Octree			
	mean	std	mean	std	mean	std	mean	std
100000	2.07E-02	1.14E-03	4.83E-03	6.40E-04	3.43E-03	1.43E-03	5.95E-05	2.22E-04
200000	3.74E-02	2.92E-03	5.76E-03	3.56E-04	3.43E-03	2.98E-04	1.91E-05	2.74E-06
400000	7.36E-02	7.81E-03	7.51E-03	5.77E-04	3.83E-03	4.12E-04	2.13E-05	3.31E-06
800000	1.35E-01	6.74E-03	1.08E-02	2.55E-04	4.53E-03	1.49E-04	2.30E-05	5.00E-06
1600000	2.64E-01	8.86E-03	1.78E-02	4.54E-04	6.20E-03	1.65E-04	2.32E-05	4.49E-06

of the dimension d of data. It can also be found from Table 2 that the mean of t_p of the acceleration method based on G-Octree is the lowest, which is significantly better than the method based on kd-tree. Considering the magnitude of the corresponding std, we think that in general cases, the t_p of the proposed method based on G-Octree is the lowest among the compared methods, which means that the proposed method is the most efficient in preparation.

It can be found from Table 3 that, with the increase of data size, the single approximation time t_s of MLS without acceleration methods obviously increases. Such an increase would cause a sharp increase in the total calculation time in problems that require multiple approximations. The three acceleration methods compared in this paper have significant effects on t_s, and PMLS is the best because of the pre-stored coefficients in preparation. Similar to the results of t_p, the mean of t_s of the method based on G-Octree is lower than that of the method based on kd-tree under various data sets.

To better demonstrate the performance of each algorithm, we list in Table 4 the total time for each of the three algorithms to complete the approximation at 100 random points, excluding the loading time of the data, under each set X. Just as we explained in Section 3.3, each of the three algorithms here only requires a preparation to complete 100 approximations.

Table 4: The total time (Unit: seconds) of MLS by different acceleration methods and PMLS under different sizes of set X

N	MLS			PMLS
	classical	kd-tree	G-Octree	
100000	2.07E+00	5.87E-01	3.73E-01	2.28E+00
200000	3.74E+00	8.05E-01	4.47E-01	5.82E+00
400000	7.36E+00	1.26E+00	5.23E-01	1.78E+01
800000	1.35E+01	2.19E+00	9.08E-01	6.92E+01
1600000	2.64E+01	4.34E+00	1.28E+00	2.59E+02

It can be known from Table 4 that under these sets, the total time used by the proposed G-Octree is the lowest. However, it must be noted that considering that MLS without using acceleration methods do not require preparation time, when the number of approximations is small enough, the total time it requires would be lower than that required by the accelerated methods. Similarly, according to the t_s of PMLS in Table 3, if the num-

ber of approximated is high enough, the total time of PMLS would become advantageous.

Combined the results from the four tables, we believe that PMLS is more suitable for cases with less data, more approximation times, and lower requirements for approximation accuracy. Under the relatively uniform sets, in MLS with the radius as (4.1), the acceleration method based on G-Octree is more efficient than that without accelerating or that based on kd-tree while maintaining the approximation results.

4.2. High-dimensional data—the exponential test function data

In this section, we use the exponential test function on $\mathbb{R}^d$ [13]:

$$g_d(\boldsymbol{x}) = \exp\left(\sum_{i=1}^{d} x_i\right),$$

to sample on Halton point sets with different dimensions. To eliminate the influence N on the results, we fix $N = 12800000$ and set the dimension of the data to d =2, 4, 6, 8, 10, respectively. In this case, the fill distance of Halton point set X can be approximately estimated as $h \approx N^{-1/d}$, and initial hypercuboid C_0 which includes those points is $[0,\ 1]^d$. Similar to the previous experiment, under different d, we set $\delta = \delta_0$ and generate 100 random points $\{\boldsymbol{y}_i\}_{i=100}^{100}$ in $[0,\ 1]^d$ for approximation. Since the approximation capabilities of the compared methods have already been shown in the previous experiment, in this experiment, we only record the preparation time t_p and the single approximation time t_s.

Table 5: The mean and standard deviation of the preparation time t_p (Unit: seconds) of MLS by different acceleration methods and PMLS under different dimensions

d	MLS						PMLS
	classical		kd-tree		G-Octree		
	mean	std	mean	std	mean	std	
4	–	–	3.29E+01	2.23E-01	4.81E+00	3.36E-02	1.72E+05
6	–	–	4.48E+01	1.19E+00	3.37E+00	1.10E-01	–
8	–	–	5.32E+01	1.61E+00	2.29E+00	5.18E-02	–
10	–	–	6.35E+01	6.50E-01	2.81E+00	2.69E-02	–

Table 6: The mean and standard deviation of the single approximation time t_p (Unit: seconds) of MLS by different acceleration methods and PMLS under different dimensions

d	MLS						PMLS
	classical		kd-tree		G-Octree		
	mean	std	mean	std	mean	std	
4	1.25E+00	1.70E-02	9.61E-02	4.26E-03	1.35E-02	1.93E-03	1.66E-03
6	1.32E+00	2.38E-02	1.45E-01	1.03E-02	4.22E-02	6.64E-03	–
8	1.49E+00	8.43E-02	2.67E-01	4.16E-02	1.02E-01	2.85E-02	–
10	1.99E+00	1.86E-01	6.66E-01	9.91E-02	3.54E-01	8.94E-02	–

In Tables 5 and 6, we only show the results of a single calculation of PMLS when $d = 4$. This is because in the approximation, the preparation needs about 48 hours due to the excessive number of subspaces in high-dimensional problems. This also means that for larger d, the t_p required by PMLS would further increase, making it less meaningful in the comparison of efficiency.

In Tables 5, the mean of t_p of the proposed acceleration method based on G-Octree is always lower than that of the method based on kd-tree. In the case of $d = 10$, the method based on kd-tree spends about 1 minute to prepare, but the proposed method only needs 2-3 seconds. Specifically, as the dimension increases, although the size of data does not increase, the mean of t_p based on kd-tree shows an obvious upward trend, while the t_p based on G-Octree shows a downward trend. This is because for the fixed N, the δ_0 which is used for building G-Octree would increase as d increases, which reduces the depth of G-Octree and make the tree-building time less. Those results mean that, in terms of tree-building, G-Octree is more suitable for high-dimensional and relatively uniform data than kd-tree.

Based on the results in Table 6, we can point out that, for the fixed N, the increase of dimension d would also increase the t_s of MLS. This is partly due to the increase of the dimension of the polynomial space, and partly due to the increased cost of distance calculation in high-dimensional space. Although the t_s used by MLS accelerated by G-Octree and kd-tree also shows an upward trend as the increase of d, it is still significantly lower than the unaccelerated results, indicating that these two acceleration methods are still effective for high-dimensional data. The mean of t_s based on G-Octree is lower than that based on kd-tree at all dimensions. Like the previous experiment, we list in Table 7 the total time for each of the three algorithms to complete

the approximation at 100 random points.

Table 7: The total time (Unit: seconds) of MLS by different acceleration methods and PMLS under different dimensions

d	MLS			PMLS
	classical	kd-tree	G-Octree	
4	1.25E+02	4.25E+01	6.16E+00	1.72E+05
6	1.32E+02	5.93E+01	7.59E+00	–
8	1.49E+02	7.99E+01	1.25E+01	–
10	1.99E+02	1.30E+02	3.82E+01	–

By comparing the results of Table 4 and Table 7, it can be found that as the dimension increases, the results of the unaccelerated MLS and PMLS have a greater gap compared with the other two acceleration methods, which implies that they would only be recommended under in a few cases.

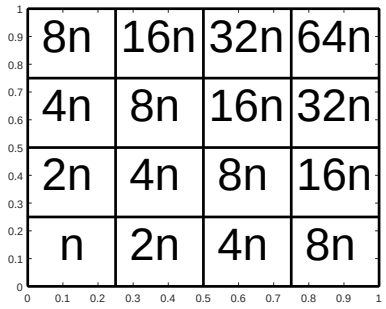
Based on the results of Section 4.1 and Section 4.2, we believe that for relatively uniform sampling point sets like the Halton point set, in MLS with the radius as (4.1), the acceleration method based on G-Octree proposed in this paper is efficient than the acceleration method based on kd-tree-based for massive data at all dimensions.

4.3. Non-uniform data and different search radii

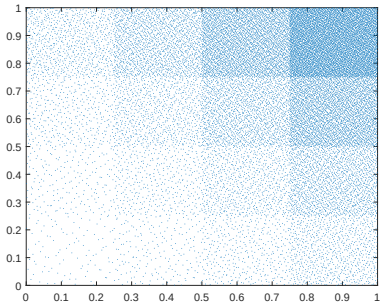
In the previous two experiments, since relatively uniform Halton point sets are used as sampling points, MLS with $\delta = \delta_0$ performs well in the calculation and there is no need to adjust δ. In some less regular data sets, in order to avoid making the normal equation (3.14) be ill-conditioned or singular, it is necessary to adjust δ.

In this section, we divide $[0,\ 1]^2$ into 16 subdomains and generate different numbers of Halton points in each subdomain, thereby obtaining a non-uniform point set X. The division and the amount of points in each subdomain are shown in Fig. 6(a). In Fig. 6(a), n, $2n$, $3n$, etc., represent the number of Halton points within corresponding subdomains. The total number of points in X is $N = 225n$. The non-uniform point set with $n = 100$ generated in this way is shown in Fig. 6(b).

Set $n = 10000$ to generate the non-uniform sampling point set X, which means that the size of X is 2250000. And the sampled function is the Franke's test function shown in Section 4.1. We still estimate its filling distance of



(a) Subdomain division and the number of points contained in each subdomain



(b) An example of X with $n = 100$

Fig. 6: The diagram and an example of the non-uniform point set X.

X as $h \approx N^{-1/2}$, which is obviously less than the exact, and then calculate δ_0 as (4.1) to build the G-Octree. Due to the inaccurate estimate, setting $\delta = \delta_0$ may cause MLS to perform poorly. Therefore, we set $\delta = \lambda \delta_0$, where λ ranges from 0.01 to 4 in intervals of 0.01. For each λ, we generate 100 random points $\{\boldsymbol{y}_i\}_{i=100}^{100}$ in $[0,\ 1]^2$ for approximation. In each approximation at $\boldsymbol{y}_i$, we record the 2-norm condition number of the matrix $\boldsymbol{P}_{\boldsymbol{y}_i}^{\top} \boldsymbol{W}_{\boldsymbol{y}_i}(\boldsymbol{y}_i) \boldsymbol{P}_{\boldsymbol{y}_i}$ in the normal equation (3.14), in addition to the preparation time (t_p), the single approximation time (t_s), and the approximation error. Given that the previous experiments have sufficiently demonstrated the performance of PMLS, in this experiment, we only compare the results of MLS without acceleration with MLS accelerated by G-Octree and kd-tree.

In our experimental results, when $\lambda \leq 0.4$, the normal equation becomes singular at certain points, which indicates that the minimum radius δ_{min} in MLS for the non-uniform point set X should be around $0.4\delta_0$. In Fig. 7, we show the mean of approximation errors at the 100 random points for each $\lambda > 0.4$, with the minimum point marked as a red star. Then, in Fig. 8, we display the mean of the condition numbers of the normal equation at 100 random points for each $\lambda > 0.4$. The λ corresponding to the minimum error point in Fig. 7 is also marked with a red star in Fig. 8.

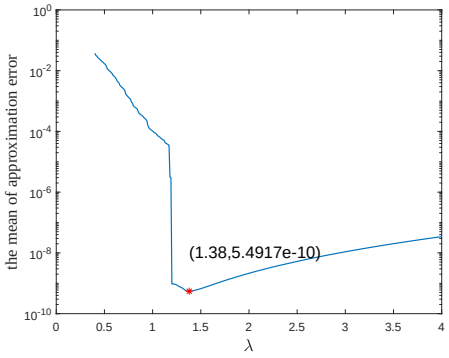


Fig. 7: The mean of approximation errors at the 100 random points for each $\lambda > 0.4$.

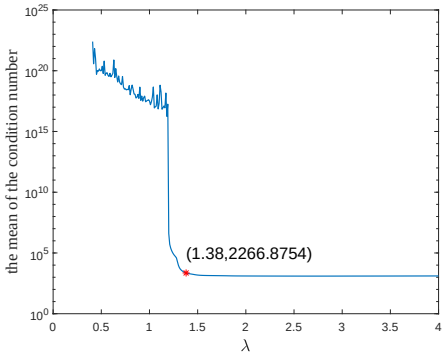


Fig. 8: The mean of the condition numbers of the normal equation at the 100 random points for each $\lambda > 0.4$.

Fig. 6 and 7 show that the optimal δ for the non-uniform point set X is around $1.38\delta_0$. When $\delta = 1.38\delta_0$, the mean of errors is lowest and the condition number is relatively small (the magnitude is 10^3) in this experiment. While increasing δ slightly reduces the condition number and improves calculation stability, it is not recommended due to the rise in the mean of errors.

For the non-uniform point set X, Fig. 9 shows the mean of single approximation time t_s at 100 points for MLS without acceleration, accelerated by G-Octree and kd-tree for each $\lambda > 0.4$.

As shown in Fig. 9, the mean of t_s obtained by the three methods shows an upward trend as λ increases. That of MLS without acceleration increases smallest, but it is always higher than that of the other two methods. This is because that the distances between all sampling points with the approximated point always need to be calculated in MLS without acceleration regardless of the radius δ of search. The mean of MLS accelerated by G-Octree is lowest but its increase is largest, which means that if δ is large enough, the mean may be lager than that of MLS accelerated by kd-tree or even be equal to that of MLS without acceleration. This is because that the proposed acceleration based on G-Octree would be invalid if $\delta > R_1$ as noted in Remark 3.5. It is worth noting that for MLS approximation, δ should not be set too small (e.g., $\delta_{min} > 0.4$ in this experiment) or too large. Around $\delta = 1.38\delta_0$ with excellent performance in the experiment, the mean of MLS accelerated by G-Octree is always significantly lower than that of the other two methods.

The means of time spent on 100 preparations by the two acceleration

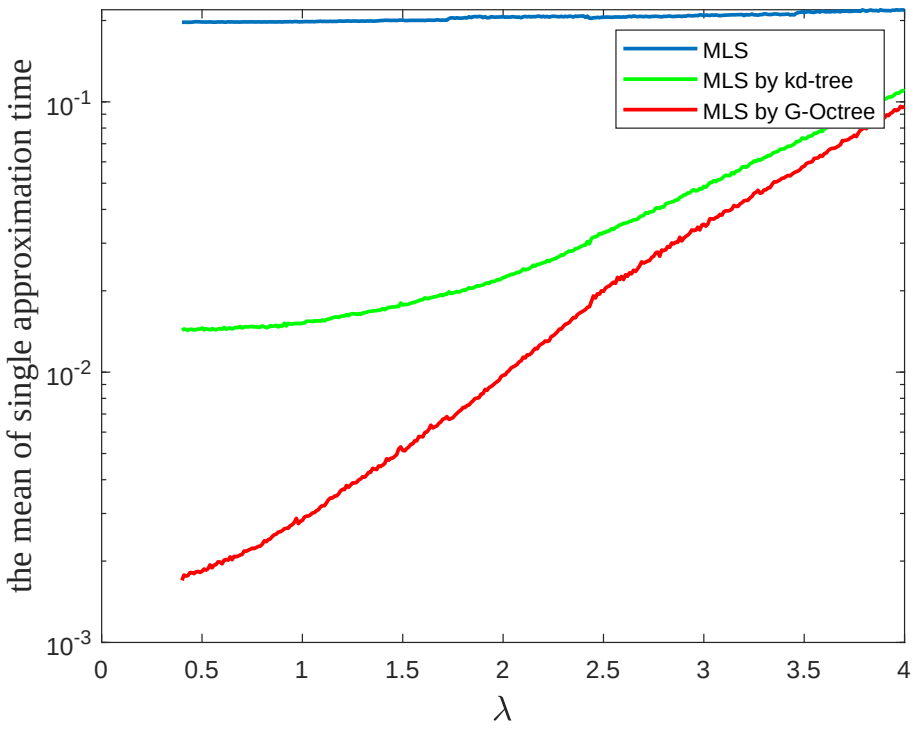


Fig. 9: The mean of single approximation time t_s at the 100 random points for each $\lambda > 0.4$.

methods under the non-uniform point set X and the total experimental time T_t, which includes one preparation and the time of 40000 approximations for λ from 0.01 to 4, of the three methods for the experiment to adjust δ are shown in Table 8.

Table 8: The mean of the preparation time t_p (Unit: seconds) and the total experimental time T_t (Unit: seconds) of MLS by different acceleration methods

	MLS		
	classical	kd-tree	G-Octree
mean(t_p)	–	1.93E+00	3.41E-01
T_t	8.19E+03	1.42E+03	8.93E+02

As shown in Table 8, for the non-uniform point set X, the proposed acceleration method based on G-Octree has a significant advantage over that based on kd-tree in preparation time. In the experiment to adjust δ, MLS accelerated by G-Octree takes much less time than the other two methods. This implies that the proposed acceleration method would also have the shortest total time if λ is selected in smaller intervals between 0.01 and 4, or the number of random points for each λ is adjusted.

Combining all results in this subsection, it is evident that for the constructed non-uniform point set X, the acceleration method in this paper is effective for not too large δ (theoretically $\delta < R_1$). Specifically, in this experiment where $\delta = \lambda \delta_0$ with $\lambda \in (0,4]$, the proposed method performs better than the acceleration based on kd-tree in both the preparation time and the single approximation time. Considering that too large δ is not recommended in MLS, we believe that the proposed acceleration method based on G-Octree is useful for non-uniform point sets like in the experiment and for the experiments to adjust δ.

5. Conclusion

This paper proposes an acceleration method based on a generalized octree for moving least squares (MLS) which can be particularly time-consuming when dealing with massive data and problems that require multiple approximations. By constructing a special generalized octree, an efficient neighborhood search algorithm is provided, thereby accelerating the computation of MLS using compactly supported weight functions in massive data of various

dimensions. In this paper, we detail the construction process of the special generalized octree and the neighborhood search algorithm, and present the complete procedure for accelerating MLS using the method.

Through numerical experiments, our acceleration method is compared with the acceleration method based on the popular kd-tree and the emerging piece-wise moving least squares method. The experimental results show that, under different dimensions and various size of sampling point sets, the proposed method preserves the approximation results of MLS. Moreover, if the radius in MLS is not too large, the preparation time of the method is shortest among the three methods, and the single approximation time is significantly less than that of the acceleration method based on kd-tree. These results indicate that in various applications based on MLS, it is worthwhile to consider using the proposed method to improve computational efficiency.

In addition to accelerating MLS, the acceleration method proposed in this paper can be attempted to be used in other high-dimensional algorithms by accelerating the neighborhood search. Moreover, combining the proposed generalized octree with the idea of the multilevel models in classical octrees and precomputing some models is a method worth considering for reducing the single approximation time while ensuring smoothness.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (No.12071019), Guizhou Provincial Education Department (Qianjiaoji[2024]57) and Guizhou Normal University (11904/0524102).

Data availability

The data of all images and codes involved in this paper are available from the corresponding.

References

[1] D. Levin, The approximation power of moving least-squares, Math. Comput. 67 (224) (1998) 1517–1531. doi:10.1090/s0025-5718-98-00974-0.

[2] H. Wendland, Local polynomial reproduction and moving least squares approximation, IMA J. Numer. Anal. 21 (1) (2001) 285–300. doi:10.1093/imanum/21.1.285.

[3] H. Wendland, Scattered Data Approximation, Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, Cambridge, 2004. doi:10.1017/CBO9780511617539.

[4] G. E. Fasshauer, Meshfree Approximation Methods with Matlab, WORLD SCIENTIFIC, 2007. doi:10.1142/6437.

[5] D. Mirzaei, Analysis of moving least squares approximation revisited, J. Comput. Appl. Math. 282 (2015) 237–250. doi:10.1016/j.cam.2015.01.007.

[6] D. Mirzaei, R. Schaback, M. Dehghan, On generalized moving least squares and diffuse derivatives, IMA J. Numer. Anal. 32 (3) (2011) 983–1000. arXiv:https://academic.oup.com/imajna/article-pdf/32/3/983/1966273/drr030.pdf, doi:10.1093/imanum/drr030.

[7] R. Salehi, M. Dehghan, A generalized moving least square reproducing kernel method, J. Comput. Appl. Math. 249 (2013) 120–132. doi:10.1016/j.cam.2013.02.005.

[8] D. Mirzaei, Error bounds for gmls derivatives approximations of sobolev functions, J. Comput. Appl. Math. 294 (2016) 93–101. doi:10.1016/j.cam.2015.08.003.

[9] L. Zhang, T. Gu, J. Zhao, S. Ji, M. Hu, X. Li, An improved moving least squares method for curve and surface fitting, Math. Probl. Eng. 2013 (2013) 1–6. doi:10.1155/2013/159694.

[10] M. Amirfakhrian, H. Mafikandi, Approximation of parametric curves by moving least squares method, Appl. Math. Comput. 283 (2016) 290–298. doi:10.1016/j.amc.2016.02.039.

[11] X. Li, S. Li, Analysis of the complex moving least squares approximation and the associated element-free galerkin method, Appl. Math. Model. 47 (2017) 45–62. doi:10.1016/j.apm.2017.03.019.

[12] D. Mirzaei, Direct approximation on spheres using generalized moving least squares, BIT 57 (4) (2017) 1041–1063. doi:10.1007/s10543-017-0659-8.

[13] A. Amir, D. Levin, Quasi-interpolation and outliers removal, Numer. Algorithms 78 (3) (2017) 805–825. doi:10.1007/s11075-017-0401-2.

[14] S. Zheng, R. Feng, A. Huang, An outlier detection and recovery method based on moving least squares quasi-interpolation scheme and l_0-minimization problem, Appl. Math. Model. 122 (2023) 127–150. doi:10.1016/j.apm.2023.05.032.

[15] J. L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (9) (1975) 509–517. doi:10.1145/361002.361007.

[16] B. Lu, Q. Wang, A. Li, Massive point cloud space management method based on octree-like encoding, Arab. J. Sci. Eng. 44 (11) (2019) 9397–9411. doi:10.1007/s13369-019-03968-7.

[17] R. Cavoretto, A. De Rossi, A meshless interpolation algorithm using a cell-based searching procedure, Computers and Mathematics with Applications 67 (5) (2014) 1024–1038. doi:10.1016/j.camwa.2014.01.007.

[18] R. Cavoretto, A. De Rossi, A trivariate interpolation algorithm using a cube-partition searching procedure, SIAM J. Sci. Comput. 37 (4) (2015) A1891–A1908. doi:10.1137/140989157.

[19] W. Li, G. Song, G. Yao, Piece-wise moving least squares approximation, Appl. Numer. Math. 115 (2017) 68–81. doi:10.1016/j.apnum.2017.01.001.

[20] G. Hunter, K. Steiglitz, Linear transformation of pictures represented by quad trees, Comput. Vision. Graph. 10 (3) (1979) 289–296. doi:10.1016/0146-664x(79)90008-x.

[21] D. Meagher, Geometric modeling using octree encoding, Comput. Vision. Graph. 19 (2) (1982) 129–147. doi:10.1016/0146-664x(82)90104-6.

[22] Y. Ohtake, A. Belyaev, H.-P. Seidel, 3d scattered data interpolation and approximation with multilevel compactly supported rbfs, Graph. Models 67 (3) (2005) 150–165. doi:10.1016/j.gmod.2004.06.003.

[23] A. Crivellaro, S. Perotto, S. Zonca, Reconstruction of 3d scattered data via radial basis functions by efficient and robust techniques, Appl. Numer. Math. 113 (2017) 93–108. doi:10.1016/j.apnum.2016.11.003.

[24] J. Teunissen, U. Ebert, Afivo: A framework for quadtree/octree amr with shared-memory parallelization and geometric multigrid methods, Comput. Phys. Commun. 233 (2018) 156–166. doi:10.1016/j.cpc.2018.06.018.

[25] A. A. Saputra, S. Eisenträger, H. Gravenkamp, C. Song, Three-dimensional image-based numerical homogenisation using octree meshes, Comput. Struct. 237 (2020) 106263. doi:10.1016/j.compstruc.2020.106263.

[26] H. Sundar, R. S. Sampath, G. Biros, Bottom-up construction and 2:1 balance refinement of linear octrees in parallel, SIAM J. Sci. Comput. 30 (5) (2008) 2675–2708. doi:10.1137/070681727.

[27] D. Shepard, A two-dimensional interpolation function for irregularly-spaced data, in: Proceedings of the 1968 23rd ACM national conference, ACM '68, Association for Computing Machinery, New York, NY, USA, 1968, pp. 517–524. doi:10.1145/800186.810616.
URL https://doi.org/10.1145/800186.810616

[28] D. H. McLain, Drawing contours from arbitrary data points, Comput. J. 17 (4) (1974) 318–324. doi:10.1093/comjnl/17.4.318.

[29] H. Wendland, Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree, Lect. Notes. Pure. Appl. 4 (1) (1995) 389–396. doi:10.1007/bf02123482.

[30] R. Franke, A critical comparison of some methods for interpolation of scattered data, Tech. rep., Naval Postgraduate School Monterey CA (1979).

[31] K. Saurabh, B. Gao, M. Fernando, S. Xu, M. A. Khanwale, B. Khara, M.-C. Hsu, A. Krishnamurthy, H. Sundar, B. Ganapathysubramanian, Industrial scale large eddy simulations with adaptive octree meshes using immersogeometric analysis, Comput. Math. Appl. 97 (2021) 28–44. doi:10.1016/j.camwa.2021.05.028.

[32] J. H. Halton, On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals, Numer. Math. 2 (1) (1960) 84–90.