A. CREATING OF THE TREE AND SEARCHING USING THE TREE STRUCTURE

Naturally, non-compactly supported basic functions are better suited to reconstruction of irregular measured data. Also, using compactly supported basic functions leads not only to some technical problems of sorting data, but not proper selection of radius of support can produce undesirable artifacts. Nevertheless, the implementation of this technique can be useful for some practical applications. The interpolation matrices based on compact support are sparse and for the interpolants few terms have to be considered.

Space recursive subdivision is an elegant and popular way for sorting the scattered 3D data. We propose an efficient approach, which allows us to get the resulting matrix as a band diagonal matrix that reduces computational complexity and permits the explorations of large data sets. Our first goal is to build an octal tree data structure from original point data. The structure of octal tree is very similar to binary trees, very well studied in literature. The difference is that each node can contain up to 8 sub-nodes or leaves, each one representing one eights of the original volume. Aside from auxiliary members we store following data in each node of the tree:

- Pointer to parent node,
- 8 pointers to child nodes,
- Pointer to the list of points (empty, if this node is not a leaf).

All the memory we need to store such a tree is (memory for each node)$\times$(number of nodes) + (memory to store point) $\times N$.

Our algorithm of building of the tree from initial N three-dimensional points requires additional parametric value K – the maximum number of points in the leaf. For the sake of convenience we scale all N points in 3D space that performs scaling them to the desired unit cube. Afterwards we divide this cube into 8 equal sub-cubes. All points are shared between these sub-cubes. Root of the tree corresponds to the initial cube, 8 sub-cubes correspond to 8 nodes from the root of the tree. This procedure is applied to each cube, until each K points would be in their own small cubes. If one of the 8 cubes does not contain points on the current step we will not build a sub-tree in that direction. Handling the tree uses the following algorithms.

Algorithm I. Octal tree creation
- *Create root of the tree.* - *Put the first point in the root (up to now it is a leaf).* - *Put the next point in the current node and so on, until the number of points in the root is less than K. When this number becomes greater than K, additional sub-nodes are created and points are stored in those sub-nodes. The tree grows and we continue to add points to the root, until we will add all N points. They go through nodes to the leaves, which are created accordingly.*

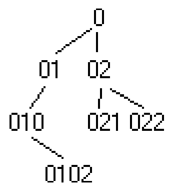
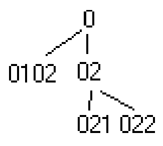
Afterwards we can use this tree to search for neighbors of any given point for the given N points. The neighbors are the points which belong to a ball (located in the given point) with the radius r. We call this ball r-sphere.

Algorithm II. Searching using the tree
- *For each node in the tree we can state that the node is either:* *a. Entirely inside the given r-sphere, or* *b. Entirely outside the given r-sphere, or* *c. Partly inside the given r-sphere, partly outside.* - *For the case (a), we return all list of points for current node leafs recursively without any checks. In the case (b) we do not check this node and any it's sub-nodes anymore. In the case (c) we follow the same procedure for its child sub-nodes. If we get the leaf on this step we make a check by enumeration of all its K points and return one, which belongs the r-sphere.*

B. SIMPLIFACTION OF THE TREE AND THE LIBRARY DESCRIPTION

To accelerate the search we produce simplification of the tree after tree creation. Simplification is done by removing unnecessary nodes. If the node *A* has only one sub-node *B*, we can remove *A* node and put *B* instead of it.

For example, consider the following octal tree (numerals represent node numbers):

Initial octal tree	Octal tree after simplification
	

This procedure is needed only if *K* is small. For instance, for data in the example 1 (see Table.1) if *K* is set to 1 this procedure is removing 113 nodes from the initially created 2427 nodes.

Maximum complexity of the described two algorithms strongly depends on the initial data and parameter *K*. The depth of the tree depends on the length of the cube edge corresponding to the leaf. This length equals to $(1/2)^{M}$, where *M* is the depth of the tree. *M* depends on the original data. If initial points are distributed more-or-less uniformly, then the tree will have sufficiently uniform filling and will be symmetric. If *K*=1, at that time the tree will be close to a full octal tree with *N* leaves. The maximum complexity of searching the tree will be proportional to the depth of this tree that is $[\log_8 N]$. The case that the depth of the tree would be *N* is also possible, but it is very improbable (all points should belong to the one selected r-sphere).

An application-programming library was developed and contains 7 main C++ classes:

1. The **CDataList** class defines stack, which we use in our implementation, instead of list. It stores the pointer to the first element of the list and defines operations with the first element.
2. The **CDataListItem** class represents an element of the list. The element stores the pointer to the next element of the list and the number of the data elements in some abstract array (or list). The technology of storing data could be arbitrary; all functions depending on the realization are in the inherited classes of the **CGetFunctions** class.
3. The **CGetFunctions** class defines the interface for functions depending on the data storing technology. An exemplar of the inherited class is stored in the tree and defines the interaction with the data array and the conditions for the recursive searching.
4. The **CPointFunctions** class inherited from **CGetFunctions** class contains functions connecting a **COctupleTree** class with the array of points, and also condition that points in space belong to the same r-sphere.
5. The **CNodeNumber** class serves for describing the node's number in the tree. Each node in the tree has its number. Actually, it is a sequence of numerals 0-7 defining which child branch should be selected on the path from the root to this node. This class represents an array for storing the number fields and allows adding more fields to the number.
6. The **COctupleTree** class defining the octal tree contains main functions. The function **add_data** of this class represents algorithm I; the function **simplify** - algorithm of simplification; the function **get_neighbors** - algorithm II. The function **add_data** appends information about one new point added to the tree without any checks for "uniqueness", that is, if the coordinates happen to be exactly the same we can add very close points. Thus it is possible to add more than *K* the same points. This function will form a correct tree with new data, even if the **simplify** function is applied. It leads to a small complication of the algorithm I, because after the simplification the number of fields in parent and child nodes can differ more than one. In the function **get_neighbors,** a leaf node is tested using the **COctupleTree::Func**, and if it fully satisfies this condition the **get_neighbors** function fills out resulting list for all points belonging this node (case a. in the algorithm II); if the leaf node partly complies with this condition (case c.), then the **CGetFunction::CheckDataItem** function is used to test each element of this node.
7. The **COctupleTreeNode** class serves to represent the node in octal tree.

C. SORTING USING OCTAL TREE

Algorithm III. Sorting data using octal tree

- Take a point from the initial data and put it in the list,
- Go through the list, for each point in the list search for the neighbors from initial data. Once we found new points, we add them to the list,
- Remove points putted in the list from the initial array,
- *If there is no more points in the list (all points were appended in the second step), than we take first point remaining from the initial array and repeat the steps.*

For example, if there are 6 points and neighbors of point 1 are 1, 3, 6; neighbors 2 - 2, 5; neighbors 3 - 3, 1, 4; neighbors 4 – 3,4; neighbors 5 - 2, 5; neighbors 6 - 1, 6. Then algorithm will make the following:

Step	List	Initial array
1	-	1,2,3,4,5,6
2	1*,3,6	2,4,5
3	1,3*,6,4	2,5
4	1,3,6*,4	2,5
5	1,3,6,4*	2,5
6	1,3,6,4,2*	5
7	1,3,6,4,2,5*	-

We will get the following matrix from it:

	1	*3*	*6*	*4*	*2*	*5*
1	1	X	X	0	0	0
3	X	1	0	X	0	0
6	X	0	1	0	0	0
4	0	X	0	1	0	0
2	0	0	0	0	1	X
5	0	0	0	0	X	1

Here "X" denote nonzero elements. On all sides of the diagonal we will get the band, with the maximum dimension α_1, which would be maximum number of the neighbors of the point.

After that we will get a band with maximum size α_1 of the neighbors of a point. The maximum complexity of this algorithm is the complexity of searching for neighbors through the octal tree for each point, i.e. $N\times$(maximum complexity of the searching algorithm II). We can reduce our computational outlays by calculating the matrix and ordering of the points simultaneously. On the first step of algorithm III we have the first point and the list of its neighbors. That means that we can calculate one row (and one column) of the band diagonal sub-matrix. On the next step, take the second point and calculate the next row of the matrix and so on, until we calculate total matrix.