An efficient algorithm for the nearest neighbourhood search for point clouds

Luca Di Angelo[1] and Luigi Giaccari[2]

**[1] Department of Industrial Engineering, University of L'Aquila
L'Aquila, 67100, Italy**

**[2] ANSYS Germany Gmbh
Otterfing, 83624, Germany**

Abstract

This paper presents a high-performance method for the k-nearest neighbourhood search. Starting from a point cloud, first the method carries out the space division by the typical cubic grid partition of the bounding box; then a new data structure is constructed. Based on these two previous steps, an efficient implementation of the k-nearest neighbourhood is proposed. The performance of the method here presented is compared with that of the *kd-tree* and *bd-tree* algorithms taken from the ANN library [1] as regards the computing time for some benchmarking point clouds and artificially generated test cases. The results are analysed and critically discussed.

Keywords: *k-nearest neighbour, point cloud, space partition.*

1. Introduction

For the last few years the use of points as the representational primitives of geometric models has spread out in computer graphics and geometric modelling applications ([2], [3] and [4]). This is also due to the recent introduction on the market of 3D scanning systems offering high resolutions with a measuring accuracy as high as 10 µm, which make it possible to capture the smallest surface features. However, these devices generate very large data sets. Some point clouds, such as those obtained by means of 3D scanning, cannot be directly used in the previously defined applications; they need to be processed in order to reconstruct high-level information starting from the only Cartesian coordinates. Typically, point clouds are processed to remove any residual noise, change the sampling rate, estimate the points' normal and/or proceed to their tessellation. All these operations require the computation of the k-nearest neighbourhoods (*knn*) for each point in the cloud. As pointed out by Sankaranarayanan et al in [4], the correct computation of neighbourhoods is important both for algorithms that estimate properties that are common in the neighbourhood and for algorithms that analyse variations in these properties. It is evident that these neighbourhoods must be obtained at the lowest computational cost as possible, so that even clouds with several millions points can be easily managed. When analysing the related literature, it seems evident that the methods, used with the typical computing powers, show such a performance that they constitute the major bottleneck in the implementation of the above-mentioned applications.

In order to make a useful contribution to this field, this paper proposes a simple algorithm for the *knn* search. It is based on a new data structure applied to the typical space division approach, which makes possible a more efficient search for the nearest neighbourhoods. This method is tested for the *knn* search in some benchmarking point clouds and artificially generated test cases. The results derived from these experiments are critically discussed hereinafter.

2. Related works

The more recent exhaustive overview of the *knn* search methods are presented in [4]. In what follows we will be considering some of the most important papers which are related to the method here being proposed, leaving out, for example, algorithms which work with multiple processors CPU and GPU and for the approximation of the k–nearest neighbourhoods.

The simplest method to construct the k–nearest neighbours of datasets is based on the simple brute-force algorithm [5]: first the Euclidean distances between each point and all the other ones are calculated; then, the k-nearest neighbours are found as those k points with the shortest distances. This algorithm is computationally inefficient since, for each data point, its time complexity is O(n_p^2), where n_p is the number of points. In order to reduce this computational cost, many methods are proposed in literature; the most important ones can be divided into three main categories:

- Voronoi-point based;
- space division strategy based;
- pivot – based.

2.1 Voronoi point based approaches

The methods belonging to the first category are mainly used in two-dimensional datasets and are based on the consideration that the Voronoi diagram decomposes the plane into cells, each of which contains a point. For a point **p** contained in the cell *C*, the points located in the cells sharing edges with *C* are the nearest to **p**.
The first algorithm that uses this approach in a three-dimensional dataset is presented by Dey et al. in [6] which proposed a method that is based on the dual of the Voronoi graph: it determines the k–nearest neighbours in a three-dimensional dataset by constructing a Dirichlet triangulation. As pointed out by Li et al in [7], it takes $O(n_p^2 \log_2 n_p)$ time, making it impractical for use in reverse engineering where, more and more often, the clouds have over one million points.
In order to improve the efficiency of the search, recently Goodsell in [8] has proposed a new method for two–dimensional datasets, which is based on the Voronoi points; the results reported show that the timing of the algorithm is quadratic ($O(n_p^2)$).

2.2 Space division strategy based approaches

Typically, with the space division strategy based methods determining whether a point is a member of the k-nearest neighbours permits to work with a small subset of the data; this way, computational costs are strongly reduced. Some of the most widely used algorithms for the *knn* search belonging to this category are the *kd-tree* and *bd-tree*. The first one is based on a k – dimensional binary search tree ([9], [10]). By the *kd-tree* the space is hierarchically partitioned into hyper – rectangular regions (*buckets*) by using hyper – planes perpendicular to the coordinate axes to form a tree. Once this structure is constructed, the search for the nearest neighbour is done by descending the tree to find the bucket containing the query point. The search for the *knn* is limited to the points within that bucket or those contained in the near buckets. Optimally, the *kd-tree* requires $O(n_p log_2 n_p)$ operations for its construction and an $O(log_2 n_p)$ operation for the search ([10] and [12]). The *box – decomposition tree* (*bd-tree*) ([13]) is a variant of the *kd-tree* that was introduced to provide greater robustness for highly clustered datasets. Above all, the *bd-tree* differs from the *kd-tree* in the fact that, in addition to the *splitting* operation, there is another decomposition operation called *shrinking*. According to the *shrinking* rule, it is possible to further divide a box containing more points than the bucket size.
Piegl and Tiller in [14] proposed a much simple algorithm for computing all the k-nearest neighbours in 2-D. Firstly, the dataset is partitioned by a rectangular grid and the points are binned in appropriate cells. If several points fall under the same bin, they are stored in a linked list. The search is extended to the rings (in ascending order) around the cell containing the query point. The search stops when the k-th shortest distance is smaller than the distance between the query point and the closest wall of the outer cell ring. The empirical tests show that the algorithm is sub-linear for small k (around 1-5% of the data); it is linear for medium k (up to about 10-20% of data) and quadric for large k (over 20% of data). Furthermore, the algorithm seems to not be practically affected by the topology of the point cloud and by the grid size.
Li and Cripps in [7] proposed a method for which the bounding box containing the points is first partitioned by a cubic grid, whose grid size (ρ_2) is estimated by the following empirical formula:

$$\rho_2 = \alpha\left(\sqrt[3]{\frac{(x_{\max}-x_{\min})(y_{\max}-y_{\min})(z_{\max}-z_{\min})}{n_p}}\right) \qquad (1)$$

where:

- α is a user – defined scalar factor;
- n_p is the number of points.

The points are stored by using the following cube structure:

cube[i][j][k] with $i=1,\ldots,n_x$; $j=1,\ldots,n_y$; $k=1,\ldots,n_z$; (2)

where n_x, n_y and n_z are the number of divisions along **x**, **y** and **z** directions, respectively. For each point **p** of the cloud, the search for the k-nearest points is carried out among those (*candidate points*) which are inside the inner and intersecting cubes of a sphere with centre at **p** and a radius $r = \rho_2 \cdot \min(n_x, n_y, n_z) \cdot \sqrt[3]{\frac{k}{n_p}}$. If the number of *candidate points* is less than *k*, the search is carried out inside a sphere of larger radius. The experimental results show that the timings are not significantly affected by the structure of the point clouds and that they are approximately linear for $k<0.05n_p$.
An efficient and simple method is proposed by Franklin in [15]. Concerning the data structure, it essentially consists of a ragged array, containing the points belonging to each cubical cell, and of a dope vector pointing to the first point of the cell. The search of closest points of a query point **q** is carried out inside a rectangular blocks of cells around that containing **q**, by using a sorted cells list.
In order to improve the efficiency of the k-nearest neighbourhood search, Gejun et al in [16] put forth a new strategy for space division. After a preliminary division where the side-length grid is chosen by the user, a secondary division is done on the basis of an empirical formula. The experiments' results show, if we focus only on the *knn* search speed, that after the second division the search range has been reduced and the searching efficiency has improved.

Several techniques are used in order to transform the d-dimensional data points in 1-d values. Some solutions of this type are based on the *pyramid technique* ([17], [18], [19]). This technique, proposed by Berchtold et al. in [20], consists in the partitioning the d-dimensional space $[0,1]^d$ (called *unit hypercube*) into 2d pyramids with the tops at (0,5; 0,5; : : : ; 0,5) and bases on each of the 2d faces of the *unit hypercube*. At each point a hash value, that is the sum between the identification number of the pyramid to which the point belongs and the distance of the point from the pyramid vertex, is assigned. All the points are stored, according to hash values, in a B^+-tree for optimal querying. The reported results in [18] show that the proposed method has a speed-up factor over the *kd-tree* between 1.6 and 2.9.
The previously presented approaches compute the neighbourhood of each point of a cloud, one point at a time. Sankaranarayanan et al. in [4] presented a more sophisticated approach that reuses point neighbourhoods already calculated to determine neighbourhoods of adjacent points. Moreover, in order to manage a large amount of points, the authors use a disk-based data structure. The results reported show that the method's performance is promising, above all, in terms of capability to elaborate clouds with 50 millions points and not in terms of computational times.

2.3 Pivot – based approaches

Generally, the pivot based methods select some *pivots* from the database and classify all the other elements according to their distance from the *pivots*. The distances $d(\mathbf{s}_j, \mathbf{p}_i)$ between elements ($\mathbf{s}_j$) and pivots ($\mathbf{p}_i$) and between the query $\mathbf{q}_k$ and the pivots ($d(\mathbf{s}_j, \mathbf{q}_k)$) are used to filter out elements. Typical algorithms belonging to this group are the *AESA* ([16]), the *LAESA* ([22] and [23]) and its variants ([24] and [25]) and the *Fixed Queries Array* ([26]). These algorithms are based on the common idea that if for some pivots $\mathbf{p}_i$ $\left|d(\mathbf{s}_j, \mathbf{p}_i) - d(\mathbf{s}_j, \mathbf{q}_k)\right| > r$ then, by the triangular inequality, $d(\mathbf{q}_k, \mathbf{p}_i) > r$ without explicitly evaluating $d(\mathbf{q}_k, \mathbf{p}_i)$. All the points which do not verify the first previous inequality must be directly compared against the query point. By increasing the number of pivots, distance evaluations increase but so does the number of elements being filtered out. As pointed out by Chavez et al. in [27], the optimum value of the pivots cannot be normally reached because it is too high in terms of space requirements. Recently some improvements to these algorithms have been proposed by Chavez et al. in [27] and Fredriksson in [28].

3. Algorithm description

In this section we describe our algorithm in detail. Generally speaking, it consists of two main steps:
- the data structure construction;
- the nearest-neighbourhood search.

3.1 The data structure construction

Efficient k-nearest neighbourhood search requires an efficient data structure which prevents from searching the entire dataset for each candidate point. For this purpose, when using *SDS* (acronym of *Search Data Structure*), the points are first indexed as $\{1, ..., n_p\}$, where n_p is the number of points. Then, similarly to the methods based on the space division strategy, the axes-aligned bounding box of the points is partitioned by a cubic grid. The box edge size (*step*) is then evaluated by the density parameter ρ of the points ($\rho = \frac{n_p}{n_{box}}$, where n_p is the number of points and n_{box} is the number of boxes), by using the formula (1) proposed by Li and Cripps in [7]. Every point is assigned to its corresponding box by the following hashing function which performs a double to integer conversion:

$$box_{id} = \frac{x}{d} + \frac{y}{d} \cdot n_x + \frac{z}{d} \cdot n_x \cdot n_y \tag{3}$$

where:
- x, y and z are the point coordinates;
- n_x, n_y are the number of divisions along **x** and **y** directions, respectively.

The box_{id} is the attribute to each data point that performs the assignment of a point to a box. All the points in a box are implicitly sorted by the index (from *1* to n_p).
The new data structure consists of two different arrays (*First* and *Next*). *First* has dimension $n_{box}x1$, where n_{box} is the number of boxes partitioning the point cloud. In the *i-th* row of *First*, of all the points contained in the *i-th* box, the point having the lowest index is recorded. The flag -1 is used for an empty box. *Next* has dimension $n_p x1$, where n_p is the number of points. In the *j-th* row of *Next*, the point contained in the *i-th* box and having an index value immediately higher than *j* is recorded. The flag -1 is used for the last point in the box.
By using this structure, it is possible to access all the points in one box with the following simple and fast operation:

$$idPoint=Next[IdPoint] \tag{4}$$

Figure 1 shows an example of decomposition of a cubic box (figure 1a) and also reports the corresponding data structure (figure 1b) for six 3D points.
In the case of very large scanned point clouds, this data structure produces a great number of empty boxes which occupy a lot of memory space and may cause the search in many useless boxes. In order to overcome this problem, a modified data structure is also proposed (henceforth *SDS_m*). In particular, the *first_m* is an associative container in which the *key value* is the number of non –

empty box and the *mapped value* is the corresponding first point included within (figure 1c). In this case, the access to the first point of the box is obtained by a binary search:

idPoint=BynarySearch(first, boxid) *(5)*

This access is intrinsically slower than that in Eq. (4); *SDS_m* is convenient, as opposed to the previous version, in cases of a very small percentage of non–empty boxes.

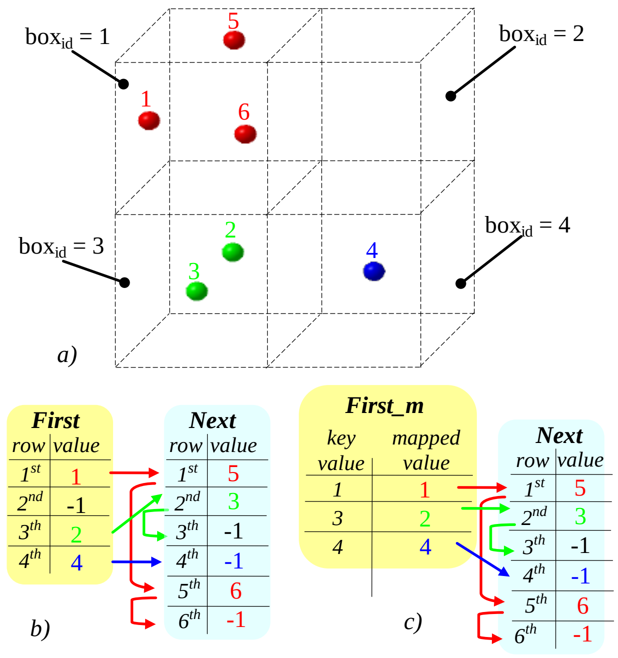

Figure 1. An example of decomposition of a cubic box (a) and the corresponding data structures (b) and (c) for six 3D points

3.2 The nearest-neighbourhood search

In order to better explain the algorithm for the nearest neighbourhood search, figure 2 proposes the pseudo – code for the *SDS* data structure and the 2D case. The reported considerations are easily extendible to the 3D space and the *SDS_m*.

The first box to be analysed is the one including the query point (**q**); for each point contained in that box the distance from **q** is evaluated. Then, the distances between **q** and the walls of the box are calculated (figure 3b). These distances are concerned with which boxes need to be further analysed (figure 3c). Contrary to the algorithm proposed by Piegl and Tiller in [14] and by Franklin in [15], this strategy prevents that all the cells belonging to the i-th ring of the cell containing the query point are explored. The search is performed until the k-th shortest distance is smaller than that existing between the query point and the closest wall of the outer box being considered. In a similar way to the typical methods presented in literature, *k* distances are stored in a priority queue with standard insertion, a structure suited for small values of *k*. The algorithm has also some controls to prevent any search attempts outside the axes-aligned bounding box of the points.

3.3 The input parameter of the method

The proposed method requires that the box size parameter should be set. The box size parameter affects the performance of the query operations of the SDS. Generally speaking, the ideal box size identifies boxes with just one point (no “overloaded boxes”). The number of boxes affects memory usage, and the number of points inside a box affects the number of distances which need to be computed during the nearest point search time. In practical cases, the optimal box size minimises empty and overloaded boxes. In what follows specific experimentations are carried out so as to investigate, in different typical applications, the values that best satisfy these conflicting constraints.

4. Performance of SDS

As pointed out by Hoppe et al. in [29], it is difficult to analyse analytically the time complexity of the search for the k-nearest neighbourhood since it strongly depends on the input data. For this reason, we have empirically analysed the performance of the *SDS* in the data structure construction and in the nearest neighbour search for some benchmarking scanned point clouds and artificially generated test cases Furthermore, in order to qualify how the proposed method compares with the state-of-art, its performances are compared with that of the *kd-tree* and the *bd-tree* algorithms taken from the ANN library [1]. Any other methods have not been analysed because their implementations were not available. Obviously, all the methods which have been analysed perform the same identification of the nearest points but they show different time complexity. All the tests have been run on a WorkStation with 3.0 GHz Intel Xeon processor and 16.0 Gb RAM.

4.1 Scanned point clouds

The test cases being considered are the typical benchmarks used in the related literature to evaluate the *knn* search methods (table 1). They consist of 16 point clouds acquired with different sampling rate from objects having different number of points and geometries ([30], [31] and [32]). Some of these point clouds are very large datasets (*Amphora*, *Neptune*, *Asian Dragon* and *Thai Statue*).

Let **q**($x_{\mathbf{q}}$, y) be the query point;
Let *k* be the number of the nearest points to **q** being needed;
Let *d*(**q**,**p**[i]) be the distance between **q** and i-th point of the dataset;
Let d_{min} *[1:k]* be the sorted list in ascending order containing the k-smallest distances between **q** and the data points;
Let *indices[1:k]* be pointers to the k-nearest points from **q**;
Let *dx_left* be the distance between **q** and the vertical left wall of the box containing **q**;
Let *dx_right* be the distance between **q** and the vertical right wall of the box containing **q**;
Let *dy_up* be the distance between **q** and the horizontal upper wall of the box containing **q**;
Let *dy_down* be the distance between **q** and the horizontal bottom wall of the box containing **q**;
Let *r* be the ring level;
Let *rx_max*, *rx_min*, *ry_max*, *ry_min* be the offset reached by the ring;
Let n_x and n_y be the number of divisions along **x** and **y** directions;
Let *step* be the cell dimensions;
Let *BoxSearch* be the function to calculate the distances between the points contained in a box and **q** and to upgrade the d_{min}*[1:k]*:

```
        void BoxSearch(ix ,iy)
        {
            id= ix+ iy*ny; /* get box id from coordinates */
            idPoint=First[id]; /* first point of the box */
            while (idPoint>-1) /* loop all the points in the box */
            {
                dist = d(q,p[idPoint]);
                if (dist < $d_{min}$[k]) /* compute squared distance between query points and reference point idp */
                {  insertion of dist in $d_{min}$; /* update the array of the smallest distances between q and the data points */
                   insertion of idPoint in indices; /* update nearest neighbour pointer*/}
                idPoint = Next[idPoint];} /* get the next reference point in the box */
        }
   initialise r=1, $d_{min}$[1:k] = big number;
   initialise rx_max, rx_min, ry_max, ry_min = 0;
   evaluation of dx_left, dx_right, dy_up, dy_down;
   evaluation of ix = ($x_q$ /d) and iy = ($y_q$ /d)
   BoxSearch(ix, iy);
   goon=true;
   while (goon=true)
   {  goon=false;

     if (($d_{min}$[k] > dx_left) AND (ix-r)>=0)
      {  // dx_left upgrade
         goon = true;
         rx_min=-r;
         for j=ry_min to ry_max
         {   BoxSearch(ix-r ,iy+j);}
         dx_left = dx_left + step;
      }

     if (($d_{min}$[k] > dx_right) AND (ix+r)< $n_x$)
      {  // dx_right upgrade
         goon = true;
         rx_max=r;
         for j=ry_min to ry_max
         {   BoxSearch(ix+r ,iy+j);}
         dx_right = dx_right + step;
      }

     if (($d_{min}$[k] > dy_up) AND (iy+r)< $n_y$)
      {  // dx_up upgrade
         goon = true;
         ry_max=r;
         for i=rx_min to rx_max
         {   BoxSearch(ix+i ,iy+r);}
         dy_up = dy_up + step;
      }

     if (($d_{min}$[k] > dy_down) AND (iy+r)< $n_y$)
      {  dy_ up upgrade
         goon = true;
         ry_min=-r;
         for i=rx_min to rx_max
         {   BoxSearch(ix+i ,iy-r);}
         dy_down = dy_down + step;
      }

     r=r+1; */ go to the next ring */
   } /* end while*/
```

Figure 2. The nearest-neighbourhood search algorithm

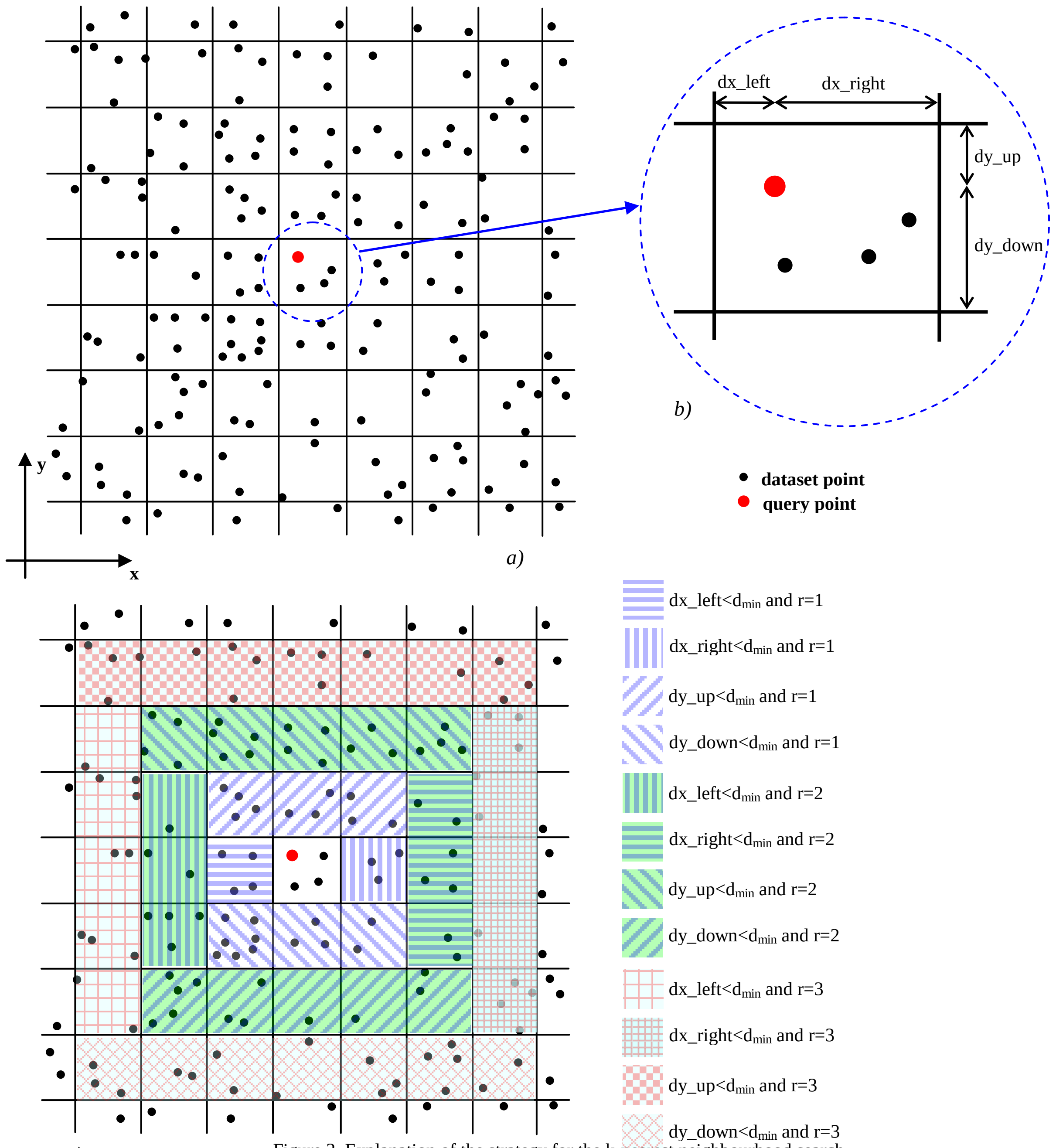
dx_left
dx_right
dy_up
dy_down
b)
dataset point
query point
y
x
a)
dx_left<d$_{min}$ and r=1
dx_right<d$_{min}$ and r=1
dy_up<d$_{min}$ and r=1
dy_down<d$_{min}$ and r=1
dx_left<d$_{min}$ and r=2
dx_right<d$_{min}$ and r=2
dy_up<d$_{min}$ and r=2
dy_down<d$_{min}$ and r=2
dx_left<d$_{min}$ and r=3
dx_right<d$_{min}$ and r=3
dy_up<d$_{min}$ and r=3
dy_down<d$_{min}$ and r=3
c)

Figure 3. Explanation of the strategy for the k-nearest neighbourhood search

A first experiment is carried out in order to verify the influence on *SDS* of the density parameter ρ by varying the number of nearest points (k) needed. As it was to be expected, as the value of ρ decreases, the computational time for the data structure construction and the amount of memory usage increase. But, on the other hand, when analysing the total computational time it is possible to verify that, for each k, the best results are obtained when the density value decreases as the number of points in the cloud increases. The obtained results show that ρ can be successfully approximated as a function of n_p, according to the following expression:

$$\rho = \max\left(-0.0618 \cdot \ln\left(n_p\right) + 0.969; 0.05\right) \qquad (6)$$

The *coefficient of determination* of the logarithmic regression approximating the 16 scanned point clouds is R^2 = 0.8618. No explicit dependence has been noticed on the k value. The choice to keep down the value of ρ to 0.05

comes from the necessity to contain memory usage in the case of very large datasets. Eq. (6) has proved to be effective for ρ estimation also in the case of point clouds whose results are not here reported since they have not been used to build the previous regression. Thus, in what follows, the experiment is carried out by using the value of ρ resulting from (6).

Table 1: Scanned point clouds used to evaluate the *knn* search methods

Name	*n. of points*
Rocker-arm	10,044
Stanford Bunny	35,947
Horse	48,485
Armadillo	172,975
Pulley	293,672
Dragon	435,545
Bimba	502,694
Happy Buddha	543,652
Rolling Stage	596,903
Chinese Dragon	655,980
Turbine Blade	882,954
Nicolò da Uzzano	946,760
Amphora	1,317,152
Neptune	2,003,933
Asian Dragon	3,609,601
Thai Statue	4,999,997

The table 2 reports the ratio between the number of empty boxes and the number of boxes ($n_{empty-boxes}/n_{boxes}$) and also the average computational time per query point for the scanned point clouds of table 1. The value of the ratio can be considered to increase with the number of points, reaching a maximum value of 0.9940. Except for the three largest datasets (namely, *Neptune*, *Asian Dragon* and *Thai Statue*), the value of the average computational time for each k remains almost constant for the different clouds. The last result verifies that the *SDS* is not practically affected by the topology of the point cloud. Furthermore, by analysing separately the two contributions of the computational time, it is found that the *SDS* structure provides a time complexity O(n_p) for the data structure construction and O(n_p•$log_2(n_p)$) for the nearest neighbour search, versus the time complexity O(n_p•$log_2(n_p)$) for both phases provided by the two other methods. The table 3 reports the mean values of the speed-ups obtained by comparing the total computational time (structure construction and nearest neighbour search) of the *SDS* with those of the *kd-tree* and the *bd-tree*, for the different *k* values. It is evident that the *SDS* is always computationally convenient, but the gain value decreases as the value of *k* increases. This is due to the fact that when the *k* value is increased the three methods tend toward the brute search.
If we use expression (6) in the case of very large datasets, the best value of ρ decreases to values which require a great amount of memory. Therefore, a further experimentation is carried out in order to verify the effectiveness of the *SD*S_m in these cases. The table 4 reports, by way of an example, the performance comparison between the *SDS* and the *SDS_m* in the cases of *Neptune*, *Asian Dragon* and *Thai Statue* with varying ρ and for k = 8. With equal ρ, an average decrease of 19% of speed-ups produces an average decrease of 86% in memory usage by the data structure. Very similar conclusions are reached when analysing the results for the other values of *k* considered.

4.2 Uniform point clouds

The test cases analysed in this section consist of different clouds whose n_p points are randomly generated with a uniform probability density and are contained in a cube. All the time values reported further down have been obtained as the mean time values of 30 cases analysed.
On analysing the influence of the density parameter ρ when varying the number of nearest points (k) needed, obtained results show that ρ can be approximated as a function of k/n_p, according to the following expressions:

$$\begin{cases} \rho = 0.1021 \cdot \log\left(\dfrac{k}{n_p}\right) + 1.99 & \text{for } \dfrac{k}{n_p} \le 0.04 \\ \rho = 0.1609 \cdot \log\left(\dfrac{k}{n_p}\right) + 1.51 & \text{for } \dfrac{k}{n_p} > 0.04 \end{cases} \qquad (7)$$

The *coefficient of determination* of the two logarithmic regressions are R^2 = 0.5312 and R^2 = 0.6842 respectively. In what follows, the experiment is performed by using the value of ρ resulting from Eq. (7). After analysing separately the two contributions of the computational time, it is found that, similarly to what happens with scanned point clouds, the *SDS* provides a time complexity O(n_p) for the data structure construction and O(n_p•$log_2(n_p)$) for the nearest neighbour search, versus the time complexity O(n_p•$log_2(n_p)$) for both phases provided by the two other methods. The table 5 reports the speed-ups mean value obtained by comparing the total computational time (structure construction and nearest neighbour search) of the *SDS* with those of the *kd-tree* and the *bd-tree*, for some *k* values. As it has been previously highlighted, the structure of *SDS* and the expression of ρ, as proposed in Eq. (7), make the method being proposed particularly efficient in the *knn* search for small values of *k*.
With a view to verifying the efficiency of the data structure and the *knn* search method which is here presented, a further experimentation is carried out so as to compare the time performance of *SDS*, *kd-tree* and *bd-tree* with that of the brute search in the case of a small number of points. The table 6 shows the mean value of the speed up for different *k*. It is evident that the use of SDS is convenient also for uniform point clouds with few points.

4.3 Very pathological cases

There are cases in which the *SDS*, like the rest of methods based on a cubic grid partition of the bounding box, cannot be used because it turns out to be strongly inefficient as opposed to the *kd-tree* and the *bd-tree*. A typical example is a scanned point cloud with one point that is very distant from the others. By using in this case the value of ρ obtained according to expression (6), both $n_{empty-boxes}/n_{boxes}$ and the maximum number of points contained in a box strongly increase. In the worst case, all the points, except for the outlier, are within a single box: *SDS* degenerates into the brute-search algorithm with a computational time complexity of O(n_p^3), as opposed to the O($n_p \bullet log_2(n_p)$) of the worst case with the *kd-tree* and *bd-tree*.

The final experiment is carried out in order to find a parameter that efficiently measures (without affecting the computational time) the dataset uniformity and its limit value for which *SDS* is not convenient as opposed to the other methods. The test cases here considered are artificially generated starting from the 16 scanned point clouds reported in table 1 by adding, to each cloud, a point at the minimum distance from the others for which the *SDS* is inconvenient, for each *k*, as opposed to the other two methods. The results show that the ratio $n_{empty-boxes}/n_{boxes}$ is a satisfactory parameter that does not affect the computational time; furthermore, the obtained value made it possible to define the limit values according to the following expressions:

$$if\ \left\{ \left[(n_p > 2*10^5)\ AND\ \left(\frac{n_{empty-boxes}}{n_{boxes}} \geq 0.996 \right) \right] \text{OR} \right.$$
$$\left. \left[(n_p \leq 2*10^5)\ AND\ \left(\frac{n_{empty-boxes}}{n_{boxes}} \geq 0,0021 \cdot \ln\left(n_p\right) + 0.97 \right) \right] \right\}$$
$$\Rightarrow \quad \text{switch to } kd\text{-}tree \text{ or } bd\text{-}tree \tag{8}$$

Since in this paper we have demonstrated that the construction of its data structure is very efficient, the *SDS* can be used to measure the uniformity of the dataset; if n_p and $n_{empty-boxes}/n_{boxes}$ satisfy any of the two conditions in (8), then the *knn* search needs to be done by the *kd-tree* or the *bd-tree.*

4. Conclusion

This paper has presented a high performance method for the *k*-nearest neighbourhood search. Said method is based on a data structure founded on the typical cubic grid partition of the bounding box. Like the principal methods presented in literature, also *SDS* requires that a parameter ($\rho = n_p/n_{box}$) should be empirically set; expressions which furnish its best value are proposed in typical applications. The performance of the *SDS* is verified by a comprehensive experiment which analyses both typical scanned and uniform point clouds. The results obtained show that the *SDS* structure provides, for both types of clouds, a time complexity O(n_p) for the data structure construction and O($n_p \bullet log_2(n_p)$) for the nearest neighbour search. Furthermore, for the scanned point clouds, the results verify that the *SDS* is not practically affected by the topology of point cloud.

When comparing the performance of the *SDS* with that of the *kd-tree* and the *bd-tree* algorithms taken from the ANN library [1], it is evident that the *SDS* is always computationally convenient; the gain value decreases as the value of *k* increases. This is due to the fact that when the *k* value is increased, the three methods tend toward the brute search.

Finally, the use of the *SDS* is also convenient, as opposed to the brute search algorithm, for uniform point clouds with few points.

References

[1] http://www.cs.umd.edu/~mount/ANN/

[2] M. Andersson, J. Giesen, M. Pauly, B. Speckmann, “Bounds on the k-neighbourhood for locally uniformly sampled surfaces”. In Proceedings of the Euro – graphics symposium on point-based graphics, June 2-4, 2004, Zurich, Switzerland, pp. 167–171.

[3] M. Pauly, R. Keiser, L. P. Kobbelt, M. Gross, “Shape modelling with point-sampled geometry”, ACM Transactions on Graphics, Vol. 22, No. 3, 2003, pp. 641–50.

[4] J. Sankaranarayanan, H. Samet, A. Varshney, “A fast all nearest neighbour algorithm for applications involving large point-clouds”, Comp. & Graphics, Vol. 31, No. 2, 2007, pp. 157–174.

[5] M. Sarkar and T. Leong, “Application of k-nearest neighbours algorithm on breast cancer diagnosis problem”, In Proceedings of the 2000 AMIA Annual Symposium, November 4–8, 2000, Los Angeles, California, USA.

[6] T. Dey, C. Bajaj, and K. Sugihara, “On good triangulation in three dimensions”, In Proceedings of the ACM Symposium on Solid modelling and application, 1991 (ACM Press, New York), pp. 431–441.

[7] X. Li and R. J. Cripps, “Algorithm for finding all k-nearest neighbours in three dimensional scattered data points and its application in reverse engineering”, Proceedings of the Institution of Mech. Engineers, Part B: Journal of Engineering Manufacture, Vol. 221, No. 9, 2007, 1467-1472.

[8] G. Goodsell, “On finding p-th nearest neighbours of scattered points in two dimensions for small p”, Computer Aided Geometric Design, Vol. 17, No. 4, 2000, pp. 387–392.

[9] J.L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching”, Communications of the ACM, Vol. 18, No. 9, 1975, pp. 509-517.

Table 2. Percentage of the number of empty boxes and average computational time per query point for the 16 scanned point clouds of Table 1

Name	$\frac{n_{empty-boxes}}{n_{boxes}}$	**Average computational time per query point [µs]**								
		k								
		1	*2*	*4*	*8*	*16*	*32*	*64*	*128*	*256*
Rocker-arm	0.8636	0.45	0.79	1.04	1.19	2.12	4.10	8.34	18.66	41.83
Stanford Bunny	0.9304	0.46	0.51	0.69	1.21	2.08	3.97	8.02	17.81	42.31
Horse	0.9551	0.52	0.54	0.76	1.32	2.30	4.20	8.30	18.75	40.70
Armadillo	0.9690	0.39	0.54	0.71	1.19	2.25	3.98	8.03	17.81	40.27
Pulley	0.9445	0.35	0.58	0.81	1.41	2.37	4.57	9.38	21.37	48.83
Dragon	0.9681	0.37	0.51	0.76	1.24	2.23	4.01	8.14	19.19	41.65
Bimba	0.9747	0.38	0.60	0.81	1.37	2.41	4.33	8.82	19.21	42.49
Happy Buddha	0.9687	0.37	0.53	0.78	1.28	2.27	4.24	8.68	19.72	45.83
Rolling Stage	0.9781	0.37	0.49	0.74	1.18	2.09	3.81	7.83	18.23	39.35
Chinese Dragon	0.9702	0.37	0.64	0.85	1.45	2.54	4.62	9.43	21.07	46.91
Turbine Blade	0.9597	0.33	0.46	0.72	1.15	1.98	4.04	8.53	19.84	45.48
Nicolò da Uzzano	0.9871	0.43	0.72	0.89	1.44	2.66	5.02	10.52	22.01	48.09
Amphora	0.9802	0.36	0.49	0.75	1.21	2.09	3.88	7.99	19.05	41.81
Neptune	0.9926	0.54	0.84	1.09	1.71	2.91	5.57	11.69	26.12	59.07
Asian Dragon	0.9940	0.45	0.74	0.93	1.57	2.86	5.40	11.81	24.83	114.04
Thai Statue	0.9922	0.45	0.73	1.05	1.76	3.22	6.47	13.87	38.81	173.23

Table 3. Speed-up mean value obtained for the scanned point clouds used to evaluate the *knn* search methods

		k								
		1	*2*	*4*	*8*	*16*	*32*	*64*	*128*	*256*
SDS	$time_{kd-tree}/time_{SDS}$	6.84	5.98	4.87	3.72	2.88	2.38	2.03	1.88	1.94
	$time_{bd-tree}/time_{SDS}$	11.28	9.69	7.89	5.90	4.55	3.72	3.12	2.71	2.66

Table 4. Performance comparison between the *SDS* and the *SDS_m* in the case of very large datasets.

Name	***ρ***	**Memory usage by the data structure [Mb]**		**Speed ups**			
		SDS	*SDS_m*	$time_{kd-tree}/time_{SDS}$	$time_{bd-tree}/time_{SDS}$	$time_{kd-tree}/time_{SDS\_m}$	$time_{bd-tree}/time_{SDS\_m}$
Neptune	0.05	160,53	8,64	4,05	6,56	3,26	5,29
	0.1	84,09	8,30	3,73	6,05	2,98	4,83
	0.15	58,61	8,15	3,50	5,68	2,78	4,52
	0.2	45,87	8,07	3,30	5,36	2,62	4,24
	0.25	38,22	8,01	3,13	5,07	2,45	3,97
Asian Dragon	0.05	289,16	15,43	4,26	6,85	3,53	5,68
	0.1	151,46	14,84	4,06	6,52	3,32	5,33
	0.15	105,57	14,60	3,90	6,27	3,16	5,07
	0.2	82,62	14,46	3,70	5,94	2,98	4,79
	0.25	68,85	14,37	3,52	5,66	2,85	4,59
Thai Statue	0.05	400,54	22,04	5,21	8,36	4,31	6,91
	0.1	209,81	21,02	4,91	7,87	4,01	6,43
	0.15	146,23	20,59	4,62	7,41	3,72	5,97
	0.2	114,44	20,34	4,34	6,96	3,57	5,72
	0.25	95,37	20,17	4,13	6,62	3,42	5,48

Table 5. Speed-up mean value obtained for the uniform point clouds used to evaluate the *knn* search methods

k	**Speed-ups**	$\mathbf{n_p}$						
		100	*1,000*	*10,000*	*100,000*	*250,000*	*500,000*	*1,000,000*
1	$time_{kd-tree}/time_{SDS}$	11.4	7.3	7.4	8.9	9.3	9.3	12.2
	$time_{bd-tree}/time_{SDS}$	14.8	12.6	13.9	14.8	14.8	14.7	19.1
16	$time_{kd-tree}/time_{SDS}$	3.6	2.4	2.3	5.5	3.3	2.8	2.8
	$time_{bd-tree}/time_{SDS}$	5.1	3.5	3.4	7.8	4.5	3.7	3.6

256	$time_{kd-tree}/time_{SDS}$	--	1.7	1.9	3.3	2.6	2.2	2.1
	$time_{bd-tree}/time_{SDS}$	--	2.1	2.2	4.4	3.2	2.7	2.5

Table 6. Speed-up mean value obtained for the scanned point clouds used to evaluate the *knn* search methods

	Speed ups		
n_p	$time_{brute-search}/time_{SDS}$	$time_{brute-search}/time_{kd-tree}$	$time_{brute-search}/time_{bd-tree}$
10	1.16	0.12	0.11
25	1.19	0.14	0.12
50	1.29	0.26	0.21
100	1.70	0.39	0.27

[10] J.L. Bentley, “Multidimensional Binary Search Trees in Database Applications”, IEEE Transactions on Software Engineering, Vol. 5, No. 4, 1979, pp. 333-340.
[11] J. L. Bentley, B.W. Weide, “Optimal Expected-Time Algorithms for Closest Point Problems”, ACM Transactions on. Mathematical Software, Vol. 6, No. 4, 1980, pp. 563-580.
[12] J. L. Bentley, “Multidimensional Divide-and-Conquer”, Communications of the ACM, Vol. 23, No. 4, 1980, pp. 214-229.
[13] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, A. Wu, “An optimal algorithm for approximate nearest neighbour searching”, Journal of the ACM, Vol. 45, No. 6, 1998, pp. 891–923.
[14] L. A. Piegl and W. Tiller, “Algorithm for finding all k nearest neighbours”, Computer Aided Design, Vol. 34, No. 2, 2002, pp. 167-172.
[15] W. R. Franklin, “NearPt3: Nearest Point Query on 184M Points in E3 with a Uniform Grid”, In Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG), 2005, pp. 239-242.
[16] Z. Gejun, M. Changsheng, X. Feng, “The K-nearest neighbour fast searching algorithm of scattered data”, In proceedings of the Intern. Conf. on Future Information Technology and Management Engineering (FITME), October 9-10, 2010, Changzhou, China, pp. 125 – 128.
[17] D. H .Lee and H. J .Kim, “An efficient technique for nearest-neighbour query processing on the SPY-TEC”, IEEE Transactions on Knowledge and Data Engineering, Vol. 15, No 6, 2003, pp. 1472-1486.
[18] B. G. Nickerson and Q. Shi, “K-nearest neighbour search using the pyramid technique”, In Proceedings of the 18th Canadian Conference on Computational Geometry (CCCG), 2006, pp. 155-158.
[19] R. Zhang, P. Kalnis, B. C. Ooi, and K. L. Tan, “Generalized multidimensional data mapping and query processing”, ACM Transactions on Database Systems, Vol. 30, No 3, 2005, pp. 661-697.
[20] S. Berchtold, C. Böhm, and H. P. Kriegel, “The pyramid-technique: towards breaking the curse of dimensionality”, In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, 1998, pp. 142-153.
[21] E. Vidal, “An algorithm for finding nearest neighbours in (approximately) constant average time”, Pattern Recognition Letters, Vol. 4, No. 3, 1986, pp. 145–157.
[22] L. Mico’, J. Oncina, E. Vidal, “A new version of the nearest-neighbour approximating and eliminating search (AESA) with linear pre-processing-time and memory requirements”, Pattern Recognition Letters, Vol. 15, No. 1, 1994, pp. 9–17.
[23] L. Mico’, J. Oncina, R. C. Carrasco, “A fast branch and bound nearest neighbour classifier in metric spaces”, Pattern Recognition Letters, Vol. 17, No. 7, 1996, pp. 731–739.
[24] S. Nene and S. Nayar, “A simple algorithm for nearest neighbour search in high dimensions”, IEEE Trans. on Pattern Analysis and Machine Intel., Vol. 19, No. 9, 1997, 989–1003.
[25] E. Chavez, J. Marroquìn, R. Baeza-Yates, “Spaghettis: an array based algorithm for similarity queries in metric spaces”, In Proceedings of String Processing and Information Retrieval Symposium, September 21 – 24, 1999, Cancun, Mexico.
[26] E. Chavez, J. L. Marroquìn, G. Navarro, “Fixed queries array: A fast and economical data structure for proximity searching”, Multimedia Tools and Applications, Vol. 14, No. 2, 2001, pp. 113– 135.
[27] E. Chávez, G. Navarro, “A compact space decomposition for effective metric indexing”, Pattern Recognition Letters, Vol. 26, No. 9, 2005, pp. 1363–1376.
[28] K. Fredriksson, “Engineering efficient metric indexes”, Pattern Recognition Letters, Vol. 28, No.1, 2007, pp. 75-84.
[29] H. Hoppe, T. Derose, T. Duchamp, J. McDonald, W. Stuetzle, “Surface reconstruction from unorganized point clouds”, In ACM SIGGRAPH, 1992, pp. 71-78.
[30] http://www.graphics.stanford.edu/data/3Dscanrep/
[31] http://shapes.aimatshape.net/
[32] http://www.lodbook.com/models/

Dr Luca Di Angelo obtained his degree in Mechanical Engineering in 1999 at the Faculty of Engineering of L’Aquila and his PhD in Mechanical Engineering in 2002 at the University of ‘Tor Vergata’ in Rome. Since the 2005, he has been a researcher at Faculty of Engineering of L’Aquila, Italy. His research interests include: computational geometry, geometric modelling of functional geometric shape, shape errors modelling and simulation and features based CAD technology. Dr. Luca Di Angelo is co-author over fifty papers in international journals and international conferences.

Luigi Giaccari received B.S degree and M.S degree in Mechanical Engineering at the Faculty of Engineering of L’Aquila, in 2007 and 2009. Since May 2011, he has been a software developer at ANSYS Germany Gmbh. His research interests include computational geometry and mesh generation. Giaccari is

co-author of two papers in international journals and international conferences.