A Hierarchical Voxel Hash for Fast 3D Nearest Neighbor Lookup

Bertram Drost[1] and Slobodan Ilic[2]

[1]MVTec Software GmbH, [2]Technische Universität München

Abstract. We propose a data structure for finding the exact nearest neighbors in 3D in approximately $O(\log(\log(N))$ time. In contrast to standard approaches such as k-d-trees, the query time is independent of the location of the query point and the distribution of the data set. The method uses a hierarchical voxel approximation of the data point's Voronoi cells. This avoids backtracking during the query phase, which is a typical action for tree-based methods such as k-d-trees. In addition, voxels are stored in a hash table and a bisection on the voxel level is used to find the leaf voxel containing the query point. This is asymptotically faster than letting the query point fall down the tree. The experiments show the method's high performance compared to state-of-the-art approaches even for large point sets, independent of data and query set distributions, and illustrates its advantage in real-world applications.

1 Introduction

Quickly finding the closest point from a large set of data points in 3D is crucial for alignment algorithms, such as ICP, as well as industrial inspection and robotic navigation tasks. Most state-of-the-art methods for solving the nearest neighbor problem in 3D are based on recursive subdivisions of the underlying space to form a tree of volumes. The various subdivision strategies include uniform subdivisions, such as octrees [14], as well as non-uniform subdivisions, such as k-d-trees [2] and Delaunay or Voronoi based subdivisions.

Tree-based methods require two steps to find the exact nearest neighbor. First, the query point falls down the tree to find its corresponding leaf node. Since the query point might be closer to the boundary of the node's volume than to the data points contained in the leaf node, tree backtracking is required as a second step to search neighboring volumes for the closest data point. The proposed method improves the time for finding the leaf node and removes the need for potentially expensive backtracking by using voxels to recursively subdivide space. The leaf voxel that contains the query point is found by bisecting the voxel size. For trees of depth L, this approach requires only $O(\log(L))$ operations, instead of $O(L)$ operations when letting the query point fall down the tree. In addition, each voxel contains a list of all data points whose Voronoi cells intersect that voxel, such that no backtracking is necessary. By storing the voxels in a hash table and enforcing a limit on the number of Voronoi intersections per voxel, the total query time is independent of the position of the query point and the distribution

of data points. The theoretical query time is of magnitude $O(\log(\log(N))$, where N is the size of the target data point set.

The amount of backtracking that is required in tree-based methods depends on the position of the query point. Methods based on backtracking therefore have non-constant query times even when using the same dataset, making them difficult to use in real-time applications. Since the proposed method does not require backtracking, the query time becomes almost independent of the position of the query point. Further, the method is largely parameter free, does not require an a-priori definition of a maximum query range, and is straightforward and easy to implement.

We evaluate the proposed method on different synthetic datasets that show different distributions of the data and query point sets, and compare it to two state of the art methods: a self-implemented k-d-tree and the Approximate Nearest Neighbour (ANN) library [15], which, contrary to its name, allows also to search for exact nearest neighbors. The experiments show that the proposed method is significantly faster for larger data sets and shows an improved asymptotic behaviour. As a trade-off, the proposed method uses a more expensive preprocessing step. Finally, we demonstrate the performance of the proposed method within two applications on real-world datasets, pose refinement and surface inspection. The runtime of both applications is dominated by the nearest neighbor lookups, which is why both greatly benefit from the proposed method.

2 Related Work

An extensive overview over different nearest neighbor search strategies can be found in [17]. Nearest-neighbor search strategies can roughly be divided into tree-based and hash-based approaches. Concerning tree-based methods, variants of the k-d-tree [2] are state-of-the-art for applications such as ICP, navigation and surface inspection [8]. For high-dimensional datasets, such as images or iamge descriptors, embeddings into lower-dimensional spaces are sometimes used to reduce the complexity of the problem [13].

Many methods were proposed for improving the nearest neighbor query time by allowing small errors in the computed closest point, i.e., by solving the approximate nearest neighbor problem [1, 11, 6]. While faster, using approximations changes the nature of the lookup and is only applicable for methods such as ICP, where a small number of incorrect correspondences can be dealt with statistically. The iterative nature of ICP can be used to accelerate subsequent nearest neighbor lookups through caching [16, 10]. Such approaches are, however, only usable for ICP and not for defect detection or other tasks.

Yan and Bowyer [18] proposed a regular 3D grid of voxels that allow constant-time lookup for a closest point, by storing a single closest point per voxel. However, such fixed-size voxel grids use excessive amounts of memory and require a tradeoff between memory consumption and lookup speed. The proposed multi-level adaptive voxel grid overcomes this problem, since more and smaller voxels are created only at the 'interesting' parts of the data point cloud, while the speed advantage of hashing is mostly preserved. Glassner [9] proposed to use a hash-table for accessing octrees, which is the basis for the proposed approach.

Using Voronoi cells is a natural way to approach the nearest neighbor problem, since a query point is always contained in the Voronoi cell of its nearest neighbor. Boada *et al.* [5] proposed an octree that approximates generalized Voronoi cells and that can be used to approximately solve the nearest neighbor problem [4]. Their work also gives insight into the construction costs of such an octree. Contrary to the proposed work, their work concentrates on the construction of the data structure and solves the nearest neighbor problem only approximately. Additionally, their proposed octree still requires $O(depth)$ operations for a query. However, their work indicates how the proposed method can be generalized to other metrics and to shapes other than points. Similar, [12] proposed an octree-like approximation of the Voronoi tesselation. Birn *et al.* [3] proposed a full hierarchy of Delaunay triangulations for 2D nearest neighbor lookups. However, the authors state that their approach is unlikely to work well in 3D and beyond.

3 Method

Notation and Overview We denote points from the original data set as $\mathbf{x} \in D$ and points of the query set $\mathbf{q} \in Q$. Given a query point $\mathbf{q}$, the objective is to find the closest point $\mathrm{NN}(\mathbf{q}, D) = \mathrm{argmin}_{x \in D} |\mathbf{q} - \mathbf{x}|_2$. The individual Voronoi cells of the Voronoi diagram of D are denoted $voro(\mathbf{x})$, which we see as closed set.

The proposed method requires a pre-processing step where the voxel hash structure for the data set D is created. Once this data structure is precomputed, it remains unchanged and can be used for subsequent queries. The creation of the data structure is done in three steps: The computation of the Voronoi cells for the data set D, the creation of the octree and the transformation of the octree into a hash table.

Octree Creation Using Voronoi cells is a natural way to approach the nearest neighbor problem. A query point $\mathbf{q}$ is always contained within the Voronoi cell of its closest point, i.e., $\mathbf{q} \in voro(\mathrm{NN}(\mathbf{q}, D))$. Thus, finding a Voronoi cell that contains $\mathbf{q}$ is equivalent to finding $\mathrm{NN}(\mathbf{q}, D)$. However, the irregular and data-dependent structure of the Voronoi tessellation does not allow a direct lookup. We thus use the octree to create a more regular structure on top of the Voronoi diagram, which allows to quickly find the corresponding Voronoi cell.

After computing the Voronoi cells for the data set D, an octree is created, whose root voxel contains the expected query range. Note that the root voxel can be several thousand times larger than the extend of the data set without significant performance implications.

Contrary to traditional octrees, where voxels are split based on the number of contained data points, we split each voxel based on the number of intersecting Voronoi cells: Each voxel that intersects more than M_{max} Voronoi cells is split into eight sub-voxels, which are processed recursively. Fig. 1 shows a 2D example for this splitting. The set of data points whose Voronoi cells intersect a voxel v is denoted

$$L(D, v) = \{\mathbf{x} \in D : voro(\mathbf{x}) \cap v \neq \emptyset\}. \quad (1)$$

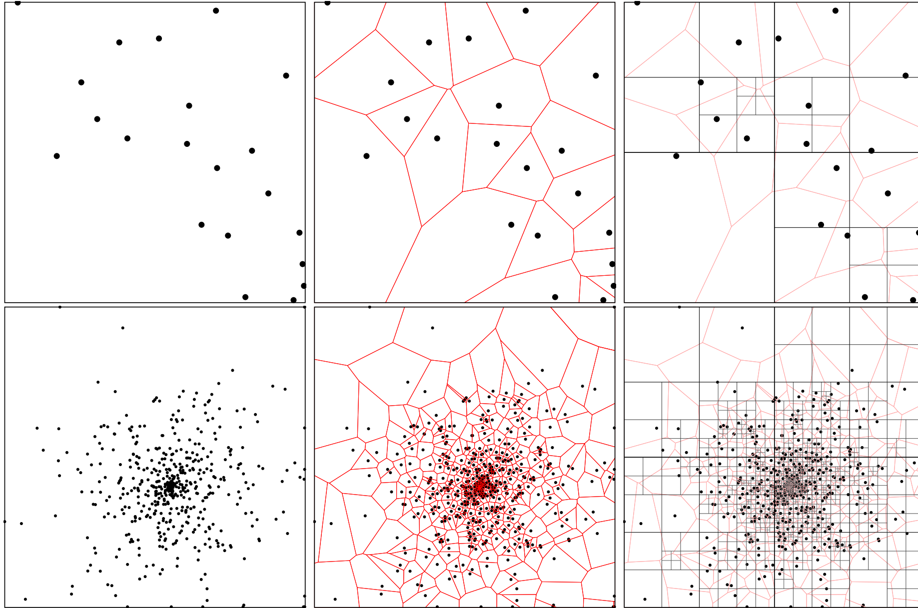


Fig. 1. Toy example in 2D of the creation of the hierarchical voxel structure. For the data point set (left), the Voronoi cells are computed (center). Starting with the root voxel that encloses all points, voxels are recursively split if the number of intersecting Voronoi cells exceeds M_{max}. In this example, the root voxel is split until each voxel intersects at most $M_{max} = 5$ Voronoi cells (right).

This splitting criterion allows a constant processing time during the query phase: For any query point $\mathbf{q}$ contained in a leaf voxel v_{leaf}, the Voronoi cell of the closest point $\text{NN}(\mathbf{q}, D)$ must intersect v_{leaf}. Therefore, once the leaf node voxel that contains $\mathbf{q}$ is found, at most M_{max} data points must be searched for the closest point. The given splitting criterion thus removes the requirement for backtracking.

The cost for this is a deeper tree, since a voxel typically intersects more Voronoi cells than it contains data points. The irregularity of the Voronoi tessellation and possible degenerated cases, as discussed below, make it difficult to give theoretical bounds on the depth of the octree. However, experimental validation shows that the number of created voxels scales linearly with the number of data points $|D|$ (see Fig. 4(a)).

Hash Table The result of the recursive subdivision is an octree, as depicted in Fig. 1. To find the closest point of a given query point $\mathbf{q}$, two steps are required: Find the leaf voxel $v_{leaf}(\mathbf{q})$ which contains $\mathbf{q}$ and search all points in $L(D, v_{leaf}(q))$ for the closest point of q. The computation costs for finding the leaf node are on average $O(depth) \approx O(\log(|D|))$ when letting $\mathbf{q}$ fall down the tree. We propose to use the regularity of the octree to reduce those costs to $O(\log(depth)) \approx O(\log(\log(|D|)))$. For this, all voxels of the octree are stored in a hash table which is indexed by the voxel's level $l(v)$ and its index $idx(v) \in Z^d$

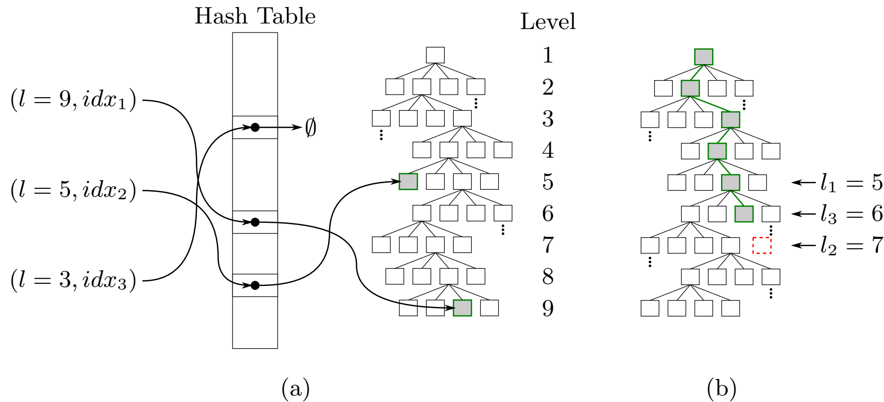


Fig. 2. (a) The hash table stores all voxels v, which are indexed through their level l and their index idx. The hash table allows to check for the existence of a voxel in constant time. (b) Toy example in 2D of how to find the leaf voxel by bisecting its level. Finding the leaf node by letting the query point fall down the tree would require $O(depth)$ operations on average (green path). Instead, the leaf node is found through bisection of its level. In each step, the hash table is used to check for the presence of the corresponding voxel. The search starts with the center level l_1 and, since the voxel exists, proceeds with l_2. Since the voxel at level l_2 does not exist, level l_3 is checked and the leaf node is found.

(Fig. 2(a)). $idx(v)$ is the integer-valued position of the voxel within the voxel grid of its level $l(v)$.

The leaf voxel $v_{leaf}(\mathbf{q})$ is then found through bisection of its level. The minimum and maximum voxel level is initialized as $l_{min} = 1$ and $l_{max} = depth$. The existence of the voxel with the 'center' level $l_c = \lfloor (l_{min} + l_{max})/2 \rfloor$ is tested using the hash table. If the voxel exists, the search proceeds with the interval $[l_c, l_{max}]$. Otherwise, it proceeds to search the interval $[l_{min}, l_c - 1]$. The search continues until the interval contains only one level, which is the level of the leaf voxel $v_{leaf}(\mathbf{q})$. Fig. 2 illustrates this bisection on a toy example.

Note that in our experiments, tree depths were in the order of 20-40 such that the expected speedup over the traditional method was around 5. Additionally, each voxel in the hash table contains the minimum and maximum depth of its subtree to speedup the bisection. Additionally, the lists $L(D, v)$ are stored only for the leaf nodes. The primary cost during the bisection are cache misses when accessing the hash table. Therefore, an inlined hash table is used to reduce the average amount of cache misses.

Degenerated Cases For some degenerated cases, the proposed method for splitting voxels based on the number of intersecting Voronoi cells might not terminate. This happens when more than M_{max} Voronoi cells meet at a single point, as depicted in Fig. 3. To avoid infinite recursion, a limit $L_{\max}$ on the depth of the octree is enforced. In such cases, the query time for points that fall within such an unsplit leaf voxel is larger than for other query points. However, we

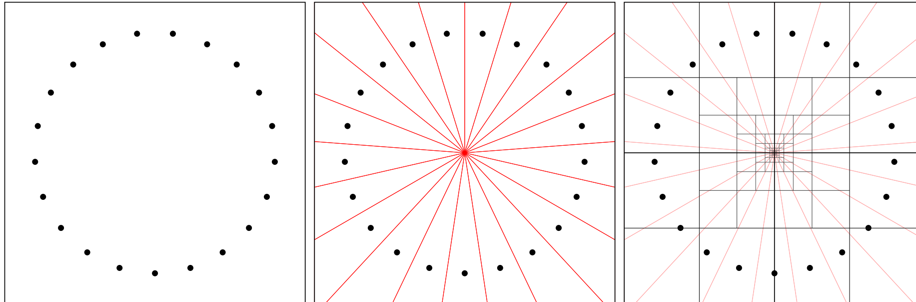


Fig. 3. Example of a degenerated point set (left) where many Voronoi cells meet at one point (center). In this case, the problem of finding the nearest neighbor is ill-posed for query points close to the center of the circle. To capture such degenerated cases, voxel splitting is stopped after L_{max} subdivisions (right). See the text for more comments on why such situations are not of practical interest.

Table 1. Performance in the real-world scenarios. $|D|$ is the number of data points, $|Q|$ the number of query points. The proposed voxel hash structure is up to one order of magnitude faster than k-d-trees, even for large values of M_{max}

			Voxel Hash, M_{max} =				
Dataset	$\|D\|$	$\|Q\|$	30	60	90	k-d-tree	ANN
ICP Matching	990,998	1,685,639	0.74 s	1.04 s	1.41 s	12.19 s	22.0 s
Comparison	990,998	2,633,591	0.85 s	1.29 s	1.87 s	10.62 s	232.1 s
ICP Room	260,595	916,873	0.26 s	0.37 s	0.41 s	0.97 s	2.5 s

found that in practice such cases appear only on synthetic datasets. Also, since the corresponding leaf voxels are very small, chances of a random query point to fall within the corresponding voxel are small. Additionally, note the problem of finding the closest point is ill-posed in situations where many Voronoi cells meet at a single point and the query point is close to that point: Small changes in the query point can lead to arbitrary changes of the nearest neighbor. The degradation in query time can be avoided by limiting the length of $L(D, v)$ of the corresponding leaf voxels. The maximum error made in this case is in bound by the diameter of the voxel of level $L_{\max}$. For example, $L_{\max} = 30$ reduces the error to 2^{-30} times the size of the root voxel, which is already smaller than the accuracy of single-precision floating point numbers. Summing up, the proposed method degrades only in artificial situations where the problem itself is ill-posed, but the method's performance guarantee can be restored at the cost of an arbitrary small error.

4 Results

Several experiments were conducted to evaluate the performance of the proposed method in different situations and to compare it to the k-d-tree and the ANN libary [15] as state-of-the-art methods. Both the k-d-tree and the voxel hash structure were implemented in C with similar optimization. The creation of the

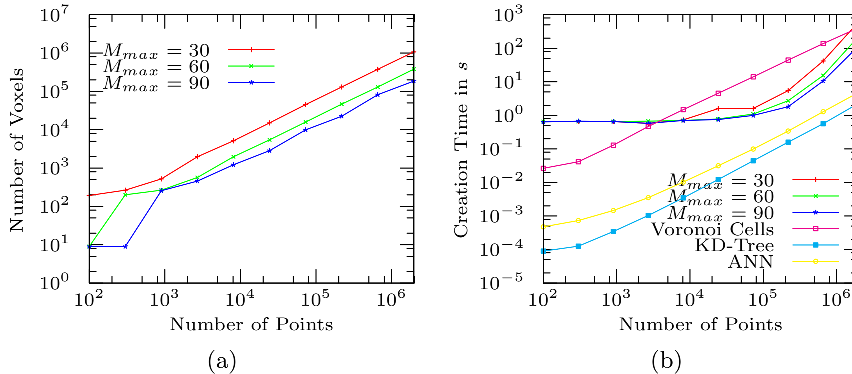
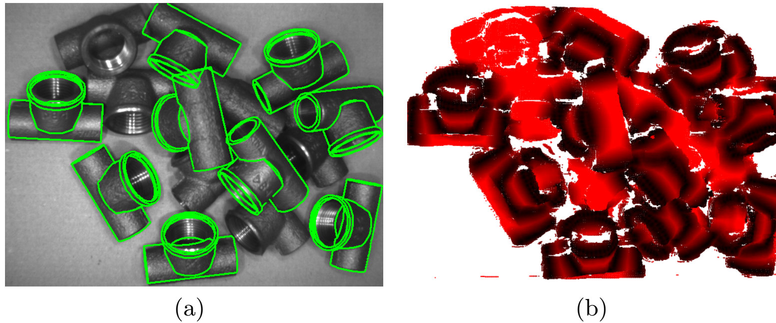


Fig. 4. Construction and memory costs of the proposed data structure for the CLUSTER dataset. (a) The number of created voxels depends linearly on the size of the data cloud. As a rule of thumb, one voxel is created per data point. Note that in practice, each voxel requires around 16-24 bytes of memory. (b) The creation time of the voxel data structure. The creation of the Voronoi cells is independent of the value of M_{max} and its creation time is plotted separately. Though the creation of the voxel data structure is significantly slower than for the k-d-tree and the ANN library, the creation times are still reasonable for off-line processing. Note that the constant performance of the proposed method for less than 10^5 data points is based on our particular implementation, which is optimized for large data sets and requires constant time for the creation of several caches. Overall, larger values of M_{max} lead to faster and less memory consuming data structure creation, at the expense of matching time (see Fig. 6).

voxel data structure was parallelized, queries were not. Times were measured on an Intel Xenon E5-2665 with 2.4 GHz.

Data Structure Creation Though the creation of the data structure is significantly more expensive than the creation of the k-d-tree and the ANN library, those costs are still within reasonable bounds. Fig. 4(b) compares the creation times for different values of M_{max}. The creation of the Voronoi cells is independent of the value of M_{max} and thus plotted separately. Fig. 4(a) shows the number of created voxels. They depend linearly on the number of data points, while the choice of M_{max} introduces an additional constant factor. Note that the constant performance of the proposed method for less than 10^5 data points is based on our particular implementation, which is optimized for large data sets and requires constant time for the creation of several caches.

Synthetic Datasets We evaluate the performance on different datasets with different characteristics. Three synthetic datasets were used and are illustrated in the top row of Fig. 6. For dataset RANDOM, the points are uniformly distributed in the unit cube $[0, 1]^3$. For CLUSTER, points are distributed using a Gaussian distribution. For SURFACE, points are taken from a 2D manifold and slightly disturbed. For each data set, two query sets with 1.000.000 points each were created. For the first set, points were distributed uniformly within the bounding cube surrounding the data point set. The corresponding times are shown in the



(a) (b)

Fig. 5. Example application for the proposed method. A 3D scan of the scene was acquired using a multi-camera stereo setup and approximate poses of the pipe joint were found using the method of Drost *et al*. [7]. (a) The poses were refined using ICP. The corresponding nearest neighbor lookups were logged and used for the evaluation show in Table 1. (b) For each scene point close to one of the detected objects, the distance to the object is computed and visualized. This allows the detection of defects on the surface of the objects. The lookups were again logged and used for the performance evaluation in Table 1.

center row of Fig. 6. The second query set has the same distribution as the data set, with the corresponding timings shown in the bottom row of Fig. 6.

The proposed data structure is significantly faster than the simple k-d-tree for all datasets with more than 10^5 points. The ANN library shows similar performance than the proposed method for $M_{max} = 30$ for the RANDOM and CLUSTER datasets. For the SURFACE dataset, our method clearly outperforms ANN even for smaller point clouds. Note that the SURFACE dataset represents a 2D manifold and thus shows the behaviour for ICP and other surface-based applications. Overall, the performance of the proposed method is less dependent on the distribution of data and query points. This advantage allows our method to be used in real-time environments.

Real-World Datasets Finally, real-world examples were used for evaluating the proposed method's performance. First, several instances of an industrial object were detected in a scene acquired with a multi-camera stereo setup. The original scene and the matches are shown in Fig. 5. We found approximate positions of the target object using the method of Drost *et al*. [7] and subsequently used ICP for each match for a precise alignment. The nearest neighbor lookups during ICP were logged and later evaluated with the available methods. The sizes of the data clouds and the lookup times are shown in Table 1.

Afterwards, we used the proposed method to find surface defects of the detected objects. For this, the distances of the scene points to the closest found model were computed. The distances are visualized in Fig. 5(b) and show a systematic error in the modeling of the object. We are, however, only interested in the required inspection time, which is shown in Table 1.

Finally, we used a Kinect sensor to acquire two slightly rotated scans of an office room and aligned both scans using ICP. For all three datasets, the

proposed method significantly outperforms both our k-d-tree implementation and the ANN library by up to one order of magnitude.

5 Conclusion

We proposed and evaluated a novel data structure for nearest-neighbor lookup in 3D, which can easily be extended to 2D. Compared to traditional tree-based methods, backtracking was made redundant by building an octree on top of the Voronoi diagram. In addition, a hash table was used, allowing a fast bisection search of the leaf voxel of a query point, which is faster than letting the query point fall down the tree. The proposed method combines the best of tree-based approaches and fixed voxel grids.

The evaluation on synthetic datasets shows that the proposed method is faster than traditional k-d-trees and the ANN library on larger datasets and has a query time which is almost independent of the data and query point distribution. Though the proposed structure takes significantly longer to be created, those times are still within reasonable bounds. The evaluation on real datasets shows that real-world scenarios such as ICP and surface defect detection greatly benefit from the performance of the method.

References

1. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. JACM 45(6), 891–923 (1998)
2. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
3. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: 11th Workshop on Algorithm Engineering and Experiments (2010)
4. Boada, I., Coll, N., Madern, N., Sellares, J.A.: Approximations of 3D generalized voronoi diagrams. 21th Europ. Workshop on Comp. Geometry (2005)
5. Boada, I., Coll, N., Madern, N., Sellares, J.A.: Approximations of 2D and 3D generalized voronoi diagrams. Int. Journal of Computer Mathematics 85(7) (2008)
6. Choi, W.S., Oh, S.Y.: Fast nearest neighbor search using approximate cached kd tree. In: IROS (2012)
7. Drost, B., Ulrich, M., Navab, N., Ilic, S.: Model globally, match locally: Efficient and robust 3D object recognition. In: CVPR (2010)
8. Elseberg, J., Magnenat, S., Siegwart, R., Nuechter, A.: Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. Journal of Software Engineering for Robotics 3(1), 2–12 (Mar 2012)
9. Glassner, A.S.: Space subdivision for fast ray tracing. Computer Graphics and Applications, IEEE 4(10), 15–24 (1984)
10. Greenspan, M., Godin, G.: A nearest neighbor method for efficient ICP. In: 3-D Digital Imaging and Modeling. IEEE (2001)
11. Greenspan, M., Yurick, M.: Approximate kd tree search for efficient ICP. In: 3DIM 2003. IEEE (2003)
12. Har-Peled, S.: A replacement for voronoi diagrams of near linear size. In: Proc. on Foundations of Computer Science. pp. 94–103 (2001)
13. Hwang, Y., Han, B., Ahn, H.K.: A fast nearest neighbor search algorithm by non-linear embedding. In: CVPR (2012)

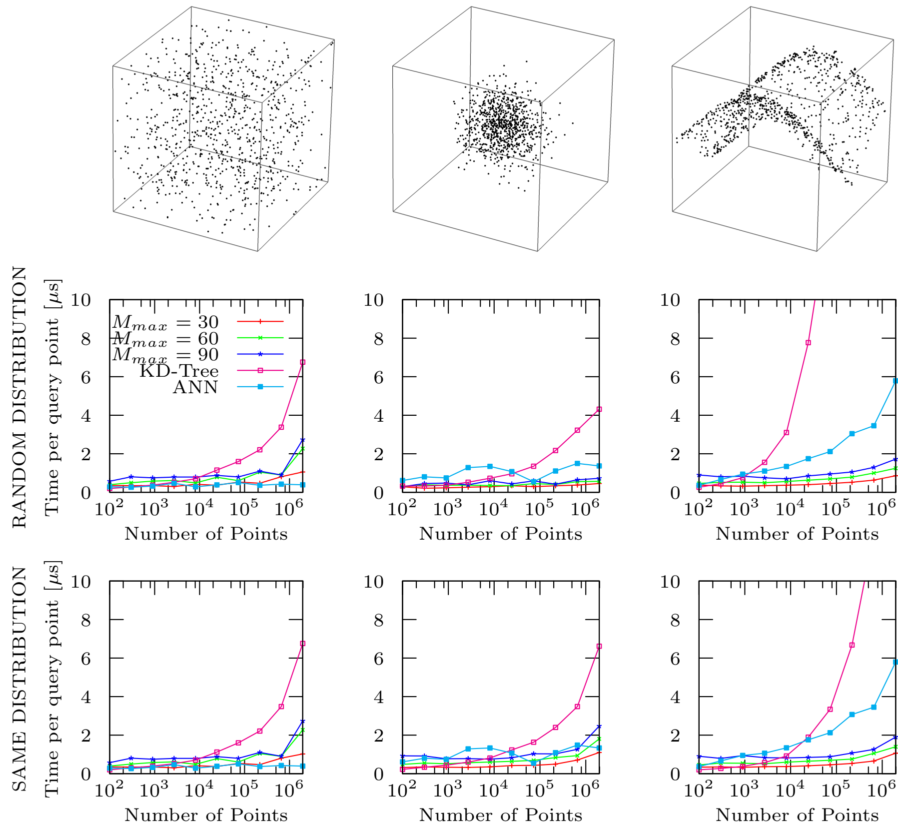


Fig. 6. Query time per query point for different synthetic datasets and methods. Each column represents a different dataset. From left to right: RANDOM, CLUSTER, and SURFACE dataset. The x-axis shows the number of data points, *i.e.*, $|D|$, the y-axis shows the average query time per query point. For the center row, query points were were randomly selected from the cuboid surrounding the data. For the bottom row, query points were taken from the same distribution as the data points. The query time for the proposed method is less dependent of the number of data points and almost independent of the distribution of the data and query points. It is especially of advantage for very large datasets, as well as for datasets representing 2D manifolds.

14. Meagher, D.: Geometric modeling using octree encoding. Computer graphics and image processing 19(2), 129–147 (1982)
15. Mount, D.M., Arya, S.: ANN: A library for approximate nearest neighbor searching, http://www.cs.umd.edu/~mount/ANN/
16. Nuchter, A., Lingemann, K., Hertzberg, J.: Cached kd tree search for ICP algorithms. In: 3DIM 2007. IEEE (2007)
17. Samet, H.: Foundations of Multidimensional And Metric Data Structures. Morgan Kaufmann (2006)
18. Yan, P., Bowyer, K.W.: A fast algorithm for ICP-based 3D shape biometrics. Computer Vision and Image Understanding 107(3) (2007)