Acta Mechanica Sinica (2013) 29(1):99–109
DOI 10.1007/s10409-013-0001-x

RESEARCH PAPER

A new insertion sequence for incremental Delaunay triangulation

Jian-Fei Liu · Jin-Hui Yan · S. H. Lo

Received: 11 June 2012 / Revised: 19 July 2012 / Accepted: 5 November 2012
©The Chinese Society of Theoretical and Applied Mechanics and Springer-Verlag Berlin Heidelberg 2013

Abstract Incremental algorithm is one of the most popular procedures for constructing Delaunay triangulations (DTs). However, the point insertion sequence has a great impact on the amount of work needed for the construction of DTs. It affects the time for both point location and structure update, and hence the overall computational time of the triangulation algorithm. In this paper, a simple deterministic insertion sequence is proposed based on the breadth-first-search on a Kd-tree with some minor modifications for better performance. Using parent nodes as search-hints, the proposed insertion sequence proves to be faster and more stable than the Hilbert curve order and biased randomized insertion order (BRIO), especially for non-uniform point distributions over a wide range of benchmark examples.

Keywords Incremental Delaunay triangulation algorithms · Insertion sequences · Kd-tree

1 Introduction

The Delaunay triangulations (DTs) and their dual Voronoi diagrams are often used in many applications, such as surface reconstruction, molecular modeling, geographical information systems and finite element mesh generation. They have been extensively studied and many different construction techniques have been devised. Among these construction schemes, incremental algorithm is most popular for its easy implementation and ability to be generalized over higher dimensions.

Throughout the construction process, incremental algorithm operates by maintaining a DT into which points are inserted one by one, and the triangulation is completed when all the points have been inserted. A formal procedure of the incremental DT algorithm is shown in Procedure 1.

Procedure 1 —— incremental DT
Input: A point set P.
Output: A DT of P.

0. Rearrange the order of input points.
for each point $p \in P$,
 1. Locate point p in the current DT, i.e., find the element containing p. It is also called base location and the element found is called the base.
 2. Create a cavity, starting from the base by adding all the elements conflicting with p. An element is conflicting (non-Delaunay) with p if its circumsphere contains p.
 3. Fill in the cavity, i.e., construct elements by connecting p with the facets on the boundary of the cavity.
 end for
end procedure

Although very simple, it has been studied in every aspect [1–11]. To create a cavity was once an error-prone step and now can be done robustly utilizing Shewchuck's precision predicates [1]. A lot of efforts focus on improving its computational efficiency [2]. Borouchaki and Lo [3] devised some techniques to speed up cavity filling. Yet more researchers paid attention to the base location and cavity creating.

Base location is usually a walking procedure: pick a starting element, walk step by step towards p, until the target element containing p is reached. The starting element often determines the walking time to the base. To decide on a starting element, one usually finds a starting point (also called a

The project was supported by the National Natural Science Foundation of China (10972006 and 11172005) and the National Basic Research Program of China (2010CB832701).

J.-F. Liu(✉) · J.-H. Yan
Department of Mechanics and Aerospace Engineering,
College of Engineering, Peking University, 100871 Beijing, China
e-mail: liujianfei@pku.edu.cn

S.-H. Lo
Department of Civil Engineering, University of Hong Kong,
Pokfulum Road, Hong Kong, China

search-hint) and then determines the starting element in the triangulation associated with the search-hint in $O(1)$ time. Hence, the base location time can be decomposed into two parts, the time for getting a search-hint and the time for the walk to the base. If one can get a search-hint in $O(1)$, which is usually the case for most of the recent schemes, then the base location time will be equal to the walking time. A walking procedure can be evaluated in terms of the number of orientation operations, and orientation is an operation to determine on which side of a triangle (or edge in a 2D case) a point is.

Upon inserting a new point, cavity creating time (as well as cavity filling time) depends on the size of a cavity and the number of existing elements conflicting with the new point. It is well known that the efficiency of the incremental DT algorithm would be affected by the sequence of insertion, as both the number of orientation operations and the number of conflicting elements are sensitive to the order that points are inserted. As a result, the sequence of insertion has been rigorously studied by researchers to improve the overall efficiency of incremental DT [4–11].

1.1 Four insertion sequences

In the earlier algorithms, points were taken by the natural order (the order as they were input) or sorted by a lexical axis-order [4]. This scheme had a tendency to produce intermediate triangulations of higher complexity than the final triangulation. The widely used random order was then devised to overcome the shortcomings of natural or lexical axis-order; it simply states: shuffle the points and then insert them one by one [5]. There are several robust and efficient implementations of the incremental DT construction based on random order point insertion and that contained in the α-shapes software.

Amenta et al. [6] presented a biased randomized insertion order (BRIO) in 2003. Their argument was: Since modern memory architecture is hierarchical and the paging policies favor programs that observe locality of reference, a major concern is cache coherence: a sequence of recent memory references should be clustered locally rather than randomly in the address space. A program implementing a randomized algorithm does not observe this rule and can be dramatically slowed down when its address space no longer fits in the main memory.

The BRIO preserves enough randomness in the input points so that the performance of a randomized incremental algorithm is unchanged but points are ordered by spatial locality to improve cache coherence. However, from the reports of other researchers [7, 8], the practical performance of BRIO is not promising. Indeed, Amenta et al. [6] considered BRIO a concept rather than a specific order, and BRIO could be implemented by merging with various insertion orders in a number of ways.

Recently, the pendulum seems swinging back to the deterministic order. Due to the work by Liu and Snoeyink [7], Zhou and Jones [8], Boissonnat et al. [9], Buchin [10,11], the space-filling curve orders are now widely used for constructing DTs. Among them, the Hilbert curve order is considered to be the most efficient order because of its locality-preserving behavior. In 2005, Liu and Snoeyink [7] used the Hilbert curve order in their program tess3 for 3D Delaunay tessellation and compared it with qhull, CGAL2.4, pyramid and hull. In their empirical comparisons, tess3 was the fastest for both uniform and non-uniform point distributions. Now the Hilbert curve order is also employed in the latest version of CGAL–CGAL4.0 [12], though mixed with the idea of BRIO in its implementation.

As a variant of the space-filling curve discovered by Peano [13], the Hilbert curve is a fractal continuous space-filling curve first described by Hilbert [14] in 1891. Its construction rule for a 2D case is shown in Fig. 1. A square with an arrow is subdivided into four sub-squares. The ordering of the sub-squares is indicated by a bold curve which connects the centers of neighboring sub-squares. For each square, there is an arrow indicating its orientation. Repeat subdividing each sub-square, the bold curve becomes longer and longer; it is called a Hilbert curve. Figure 2a shows a 2D Hilbert curve after 5 subdivision steps generated by MATLAB. Figure 2b shows a 3D Hilbert curve after 3 subdivision steps generated by MATLAB.

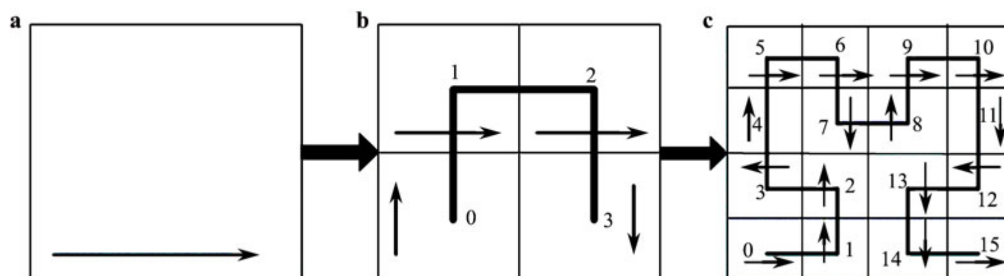

Fig. 1 The construction rule of the 2D Hilbert curve

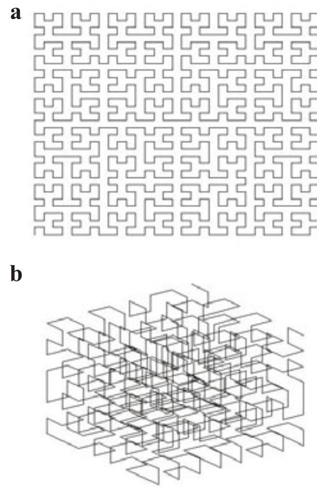

Fig. 2 The Hilbert curves in 2D and 3D

To order a set of points along a 3D Hilbert curve, one usually subdivides the bounding cube of the points into $(2^k)^3$ boxes and sorts the points using the index of the box on the Hilbert curve that contains each point. Parameter k is chosen large enough so that the number of points each box contains is small.

1.2 Two factors to evaluate the insertion sequence

The overall efficiency of the incremental algorithm is of course the ultimate judge of an insertion sequence. As mentioned before, there are two factors to be considered, namely, the efficiency of base location and the efficiency of triangulation update in cavity creating and filling. Base location is dominated by the orientation test which determinates on which side of a triangle a point is. Cavity creating and filling mainly involves identifying and deleting elements conflicting with the inserting point. To a great extent, the cost of triangulation may be measured more objectively by counting the number of orientation operations and conflicting elements rather than by the raw program running time [8].

Researchers are always interested in the theoretical complexity of an algorithm. The theoretical estimate of point location efficiency is possible and we refer the interesting readers to Sect. 2.1 where some existed 2D results are presented. However, the theoretical analysis for triangulation update is rather difficult as it depends also on the structural layout of the points, i.e., point distribution and patterns. The estimation techniques used in random order suggesting that the expected number of structural changes for n randomly inserted points is $O(n)$ simply could not be applied to deterministic orders [15]. Experimental tests thus have to be used to evaluate and to compare the performance of random and deterministic orders [8]. Following this idea, we will take note of the number of conflicting element to evaluate the structure update efficiency and the number of orientation tests to evaluate the base location efficiency.

1.3 Summary of our work and the organization of this paper

We will present a new deterministic nodal insertion sequence for the construction of incremental DT and then evaluate its constructing time and the associated base location time in Sect. 2. The experimental results for a wide range of examples are presented and compared with the sequence given by BRIO and Hilbert curve in Sect. 3. Results will be assessed by the computer time taken and the counts of orientation operations and conflicting elements. Finally, in Sect. 4, we will give some conclusions and discussions.

2 A new order from breadth-first travel of a Kd-tree

We propose a simple and intuitive insertion order without randomness. Our idea is quite simple, we build a Kd-tree to store points, then a new sequence is established by a breadth first travel across the Kd-tree of points. Kd-tree is a useful data structure for a lot of geometrical problems [16] for which various rules can be adopted for its construction. Our scheme has only a minor modification compared to the standard form. Hence, we will first show the standard way to build a Kd-tree, and then we will introduce the modifications and highlight the differences between the two schemes.

A standard Kd-tree is built following a so called alternative rule. Figure 3 shows the entire building process for a set of 15 points in the 2D case by means of a recursive procedure. Given a point set, we find its median, store the point

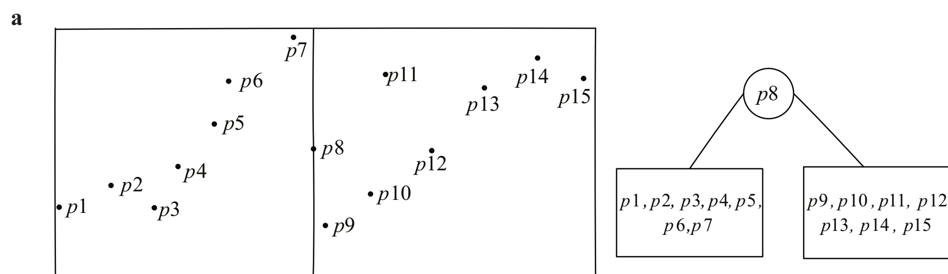

Fig. 3 Steps to build a Kd-tree following the alternative rule

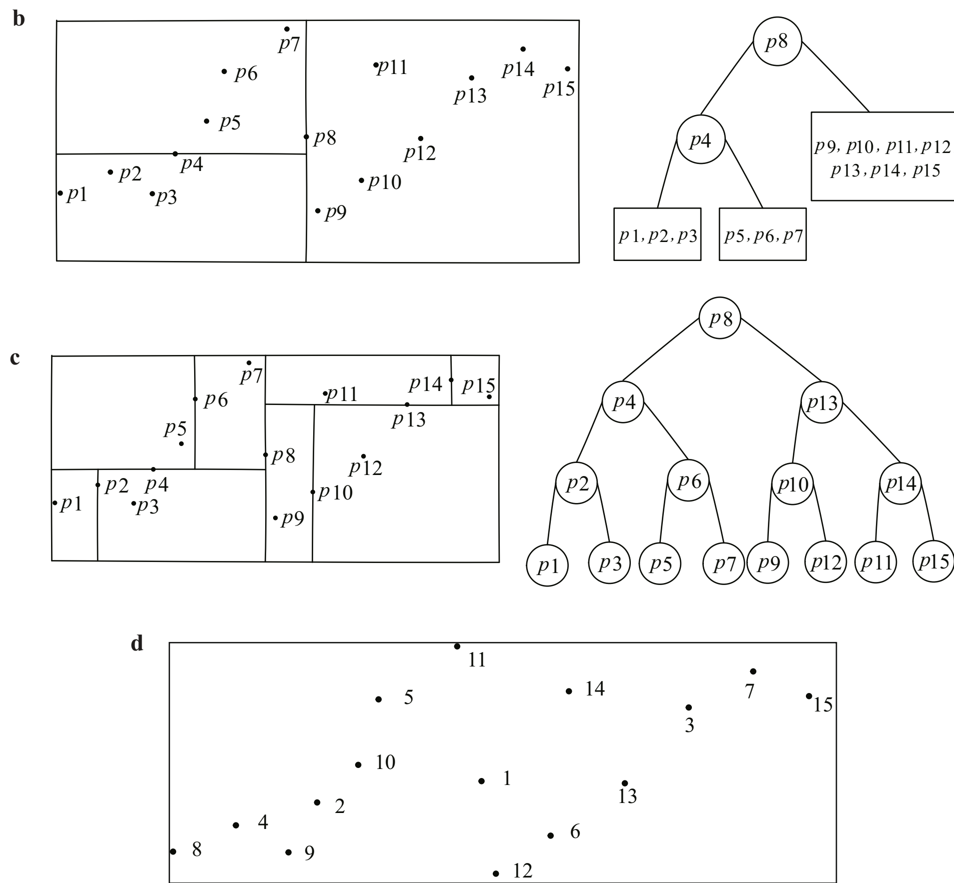

Fig. 3 Steps to build a Kd-tree following the alternative rule (continued)

on the median in the root of a tree and divide the remaining points into two sets which will contribute respectively to the left and the right subtrees of the root. By the alternative rule, we first divide points along the x-median, then the y-median, making alternative rotations about the three axes on the way of construction.

In Fig. 3d, a relabeling of the points is shown corresponding to the breadth-first travel of the Kd-tree in Fig. 3c. We apply a similar procedure to build a Kd-tree, with a minor modification. We use a slightly different rule, cutting-longest-edge rule, to divide the points as shown in Fig. 4. Given a set of points, its bounding box is determined. There are two ways in 2D (and 3 ways in 3D) to divide the set into two subsets. Our rule is to cut the longest edge of the bounding box so as to create regions of more homogeneous size along different dimensions. Using the cut-longest-edge rule to build the Kd-tree for the same set of points in Fig. 3 is shown in Fig. 5.

From the final result in Fig. 5c, we can get the new insertion sequence for this point set by the breadth first travel across the Kd-tree. That is: $p8$, $p4$, $p12$, $p2$, $p6$, $p10$, $p14$, $p1$, $p3$, $p5$, $p7$, $p9$, $p11$, $p13$, $p15$ and they are relabeled in a new insertion sequence as shown in Fig. 5d. The pseudo code for the above procedure is shown in Appendix I, which is almost the same as given by de Berg et al. [16].

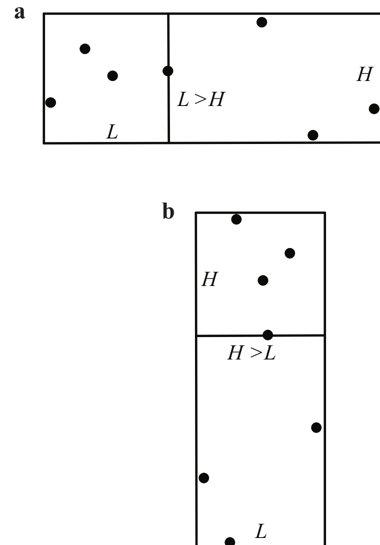

Fig. 4 Cut-longest-edge rule. **a** A vertical split line used as $L > H$; **b** A horizontal split line used as $H > L$

To get a median for a set of points is an operation needed for each recursive step, which, in our program, is done by invoking a C^{++} STL function named nth_element().

The nodes of the Kd-tree are stored in an array in the breadth-first manner, from top to bottom and from left to

right. When a new point is to be inserted, the parent node on the Kd-tree, which should have already been inserted in the triangulation, will serve as the starting point (or search-hint). The reason for using the median point as the search-hint is that the median point is usually a close neighbor of the inserting point. In general, the closer the inserting point is to its search-hint, the fewer the elements that will be walked through on the way to the base.

As we will show later, the proposed sequence from the modified Kd-tree is superior to the other orders currently used. A 2D version theoretical analysis on the base location efficiency is presented in Sect. 2.1. As for the conflicting elements, we have a conjecture: inserting points with an even separation (distance apart) in space would reduce the conflicting elements. From earlier experiences, it is known that adding points from one end towards the other end of the point space directly will result in relatively large numbers of conflicting elements, and this can be avoided by adding nodes randomly. We postulate that incremental DT algorithm prefers processing points with reasonable separation in space in a balanced manner. Random or not actually has no direct consequence on the efficiency.

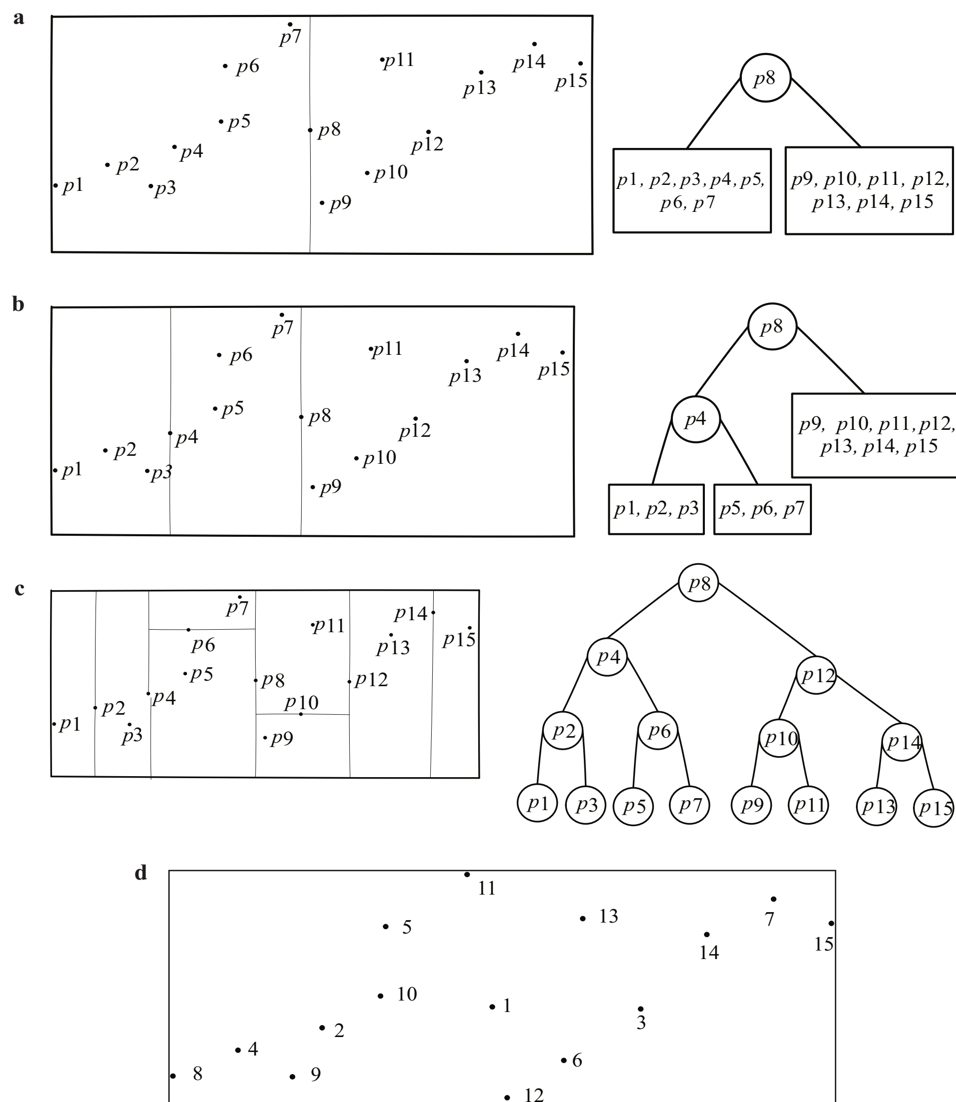

Fig. 5 Steps of building a Kd-tree following the cut-longest-edge rule

2.1 Some theoretical discussion about the time complexities of the new sequence

There are three parts about its time complexity, namely building time of the new order, locating time and structure updating time when a point is inserted. We can give an exact time complexity for building a new order. However, for the locating we have only some relevant results for 2D cases borrowed from others; for structure updating we can not give any theoretical results and can only refer our readers to the experiment tests in Sect. 3.2.

The construction of the new sequence is based on the building of a Kd-tree with a modification in the way the point set is divided. It is well known that this process takes $O(n \lg n)$ time [16]. Tests were conducted to verify this and experimental results are presented in Sect. 3.2.

Point location depends on the search-hint. Obviously if the search-hint is proximal to the target, fewer elements will be walked through and fewer orientation tests will be performed. Devroye et al. [17–19] considered point location in DTs in the 2D cases and analyzed several methods in which simple data structures were employed to first locate a point close to the query point. The following is their conclusion: For points uniformly distributed on a unit square, the expected point location complexities are $O(n^{1/2})$ for the Green–Sibson rectilinear search, $O(n^{1/3})$ for Jump and Walk, $O(n^{1/4})$ for BinSearch and Walk (1D search tree is used), $O(n^{0.056})$ for search based on a random 2D tree, and $O(\lg n)$ for search aided by a 2D median tree.

From their study, the point location time is $O(\lg n)$ aided by a two dimension Kd-tree. This is actually the time to search for a starting point from a Kd-tree. If, for an inserting point, its position on the 2D Kd-tree is already known in relation to its parent node, then walking from the parent point to the inserting point takes only $O(1)$ time [19]. Although Devroye analyzed the problem of point location and established the conclusion only in the 2D case, our tests showed that it is also true in the 3D case. Point location costs only $O(1)$ time taking the parent node as a search-hint.

3 Performance of BRIO, Hilbert curve order and the new sequence

Buchin [10, 11] proved that the incremental construction along space-filling curves (with no additional point data structure) computes the DT of uniformly distributed points in a bounded convex region in linear expected time when mixed with BRIO, apparently not including the $O(n \lg n)$ time for preparing the order. Amenta et al. [6] also gave a rigorous proof that with incremental construction using the BRIO, the expected running time is $O(n^2)$ in the worst case and $O(n \lg n)$ in the realistic cases. We do not know how to apply such an analysis to our new sequence because it is deterministic. Therefore in the next section, a number of experimental tests are conducted in three dimensions to see how the three orders BRIO, Hilbert curve order and the new sequence perform under the framework of CGAL.

3.1 Empirical comparison of the node sequencing schemes

The comparison is focused on the insertion sequences in such a way that other factors must be kept unchanged. There is a function called insert() in the DT in CGAL, this function can take as input either a single point or a set of points. If points are fed to the function one by one, the order is determined by the input in its natural order. As an option, an extra parameter of insert() could be used to take any point which has been inserted before as a search-hint for the inserting point.

If a point set is used as input, the points will be sorted along the Hilbert curve automatically and then inserted into the DT following the Hilbert curve order. In the comparison, we build the new sequence and BRIO first, then let insert() take one point at a time following the established orders. As for Hilbert curve order, we just feed insert() all the points as a single set. In this way, other factors are fixed and the only difference in the process is the insertion sequence. Different sequences are evaluated by the DT time (including the time for order building), the number of orientation tests and conflicting elements encountered.

We have inspected the related code of CGAL. It should be pointed out that, though CGAL is mainly a Hilbert curve order, its current implementation is mixed with some idea of BRIO.

Seven different point distributions for the three sequences are tested as shown in Fig. 6.

Points are all disturbed a little away from their geometry model. Times are recorded in seconds on a common PC. The results of the comparison are showed in Tables 1–14.

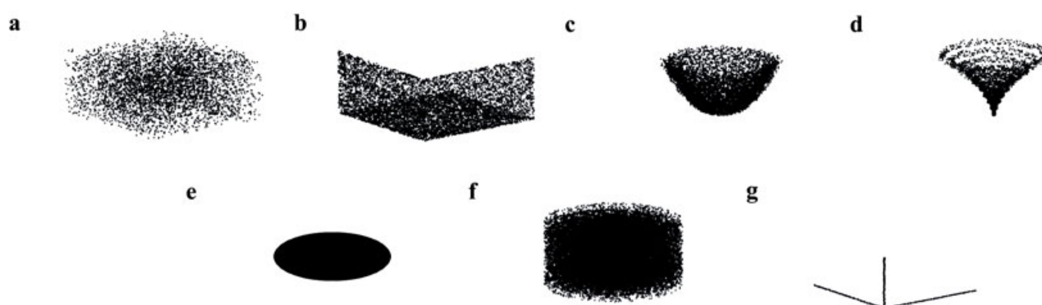

Fig. 6 Seven point distributions. **a** Points in a cube; **b** Points around three planes; **c** Points around a paraboloid; **d** Points around a spiral; **e** Around a disk; **f** On a cylinder; **g** Along three axes

3.2 The results for seven scholastic models

From Tables 1–7, computation times of the three sequences for the seven examples with various data size are presented. It is noted that the time for order building are included. The test platform is a Pentium®Dual-Core CPU E53000 @2.6GHz+Windows XP system with 2G EMS memory.

Table 1 Cube, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
100 000	5	5	5
200 000	10	12	10
300 000	14	16	16
400 000	19	21	21
500 000	25	29	27
600 000	31	32	32

Table 2 Plane, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
300 000	13	20	17
450 000	20	29	26
600 000	27	41	35
750 000	35	59	44
900 000	41	63	54
1 050 000	48	87	63

Table 3 Paraboloid, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
180 000	8	14	11
360 000	17	30	24
540 000	26	45	33
720 000	36	51	44
900 000	42	61	58
1 050 000	50	79	72

Table 4 Spiral, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
360 000	15	18	18
720 000	30	38	36
1 080 000	44	55	54
1 440 000	56	75	71
1 800 000	69	92	89
2 160 000	83	115	107

Table 5 Disc, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
360 000	17	30	23
720 000	31	58	51
1 080 000	48	88	81
1 440 000	63	125	113
1 800 000	83	191	147
2 160 000	100	248	182

Table 6 Cylinder, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
360 000	17	26	21
720 000	33	67	46
1 080 000	50	94	75
1 440 000	68	146	112
1 800 000	86	201	145
2 160 000	101	267	201

Table 7 Axes, CPU time in seconds

Datasize	Kd-tree	BRIO	Hilbert
300 000	14	215	148
450 000	22	317	259
600 000	25	496	397
750 000	32	786	557
900 000	40	1 079	743
1 050 000	45	1 385	938

Test results showed that the Hilbert curve order is faster than the BRIO, and this observation is in agreement with the experiments of Liu et al. [7] and Zhou et al. [8]. The new sequence is the fastest among the three for all the testing cases. While the improvement is marginal for uniform point distributions, it is significant for non-uniform distributions. A merit of the proposed sequence is that it is very stable while the others vary considerably with point distributions in a trend that the performance deteriorates drastically with non-uniform distributions. In the extreme case, when the points are distributed along coordinate axes, the program run with Hilbert order and BRIO were extremely slow; yet the program run with the proposed Kd-tree sequence was not slowed down compared to the uniform distribution cases.

From Tables 8–14, the orientation and conflict element counts are listed. We found two functions namely orientation() and find_conflicts() in CGAL and have put a counter in each function. The counts are divided by n, the data size, and the average numbers of orientations and conflicts for each point are thus obtained. In almost all the examples, the new

sequence is the best with the smallest counts. The only exception is that Hilbert order has a slightly smaller number of conflicting elements for the cylinder case. However, in this case, the orientation count for Hilbert order is much larger and the overall time is thus much longer than that of the Kd-tree sequence.

Table 8 Cube: the number of orientation and conflicting elements

Data size			100 000	200 000	300 000	400 000	500 000	600 000
Cube	Kd-tree	Orientation	11.787	11.675	11.776	11.7	11.672	11.765
		Conflicts	17.000	17.035	17.229	17.133	17.143	17.276
	BRIO	Orientation	16.383	15.735	17.776	16.977	16.592	20.106
		Conflicts	17.103	17.619	21.006	17.988	19.722	20.847
	Hilbert curve	Orientation	12.716	12.852	13.208	12.78	12.656	12.600
		Conflicts	18.686	18.930	19.023	18.725	18.798	18.821

Table 9 Plane: the number of orientation and conflicting elements

Data size			300 000	500 000	600 000	750 000	900 000	1 050 000
Plane	Kd-tree	Orientation	11.738	12.037	12.192	12.204	11.741	12.151
		Conflicts	15.738	15.807	15.942	15.902	15.782	15.978
	BRIO	Orientation	33.741	39.254	40.121	35.223	42.091	36.455
		Conflicts	18.547	17.222	19.043	17.947	23.589	17.954
	Hilbert curve	Orientation	22.966	23.031	23.337	23.626	23.931	24.412
		Conflicts	18.837	18.748	18.772	18.767	18.803	18.779

Table 10 Paraboloid: the number of orientation and conflicting elements

Data size			80 000	360 000	540 000	720 000	900 000	1 050 000
Paraboloid	Kd-tree	Orientation	12.024	12.029	12.015	12.026	11.910	12.002
		Conflicts	16.494	16.415	16.486	16.362	16.288	16.417
	BRIO	Orientation	34.920	41.203	40.449	41.612	49.853	47.911
		Conflicts	17.015	17.317	22.212	17.731	19.834	17.900
	Hilbert curve	Orientation	23.285	26.649	28.025	29.137	30.181	31.140
		Conflicts	17.754	17.678	17.666	17.692	17.668	17.675

Table 11 Spiral: the number of orientation and conflicting elements

Data size			360 000	720 000	1 080 000	1 440 000	1 800 000	2 160 000
Spiral	Kd-tree	Orientation	11.347	11.077	10.836	10.789	10.589	10.527
		Conflicts	14.587	13.978	13.506	13.156	12.782	12.562
	BRIO	Orientation	16.920	17.801	21.199	20.381	15.947	18.256
		Conflicts	16.696	16.667	17.758	16.579	18.102	16.316
	Hilbert curve	Orientation	14.703	14.450	14.498	14.641	14.824	15.007
		Conflicts	17.082	17.047	17.052	17.063	17.033	16.996

Table 12 Disc: the number of orientation and conflicting elements

Data size			360 000	720 000	1 080 000	1 440 000	1 800 000	2 160 000
Disc	Kd-tree	Orientation	12.376	12.593	12.388	12.695	12.491	12.426
		Conflicts	15.403	15.607	15.716	15.748	15.776	15.826
	BRIO	Orientation	43.863	75.440	78.536	79.689	91.278	87.655
		Conflicts	19.253	25.219	24.317	24.633	29.626	21.459
	Hilbert curve	Orientation	34.980	42.088	48.379	54.091	59.411	64.467
		Conflicts	18.886	19.244	19.422	19.460	19.561	19.572

Table 13 Cylinder: the number of orientation and conflicting elements

Data size			360 000	720 000	1 080 000	1 440 000	1 800 000	2 160 000
Cylinder	The new order	Orientation	11.892	11.984	11.907	12.063	11.958	11.976
		Conflicts	15.971	16.098	16.182	16.213	16.160	16.269
	BRIO	Orientation	47.798	75.891	89.370	111.29	100.86	92.376
		Conflicts	20.574	17.705	17.483	23.390	23.696	19.205
	Hilbert curve	Orientation	27.241	35.823	43.954	51.756	59.360	66.942
		Conflicts	11.892	11.984	11.907	12.063	11.958	11.976

Table 14 Axes: the number of orientation and conflicting elements

Data size			300 000	450 000	600 000	750 000	900 000	1 050 000
Axis	Kd-tree	Orientation	14.239	14.868	14.783	14.775	15.129	13.811
		Conflicts	15.109	15.302	15.348	15.361	15.517	15.318
	BRIO	Orientation	884.11	1 317.2	1 213.7	1 511.3	1 982.1	1 803.3
		Conflicts	17.650	22.323	18.884	18.259	22.504	26.999
	Hilbert curve	Orientation	667.35	826.76	954.80	1 077.4	1 189.8	1 288
		Conflicts	18.977	19.012	19.040	19.031	19.033	19.033

We have also tested the time to build a sequence. Since it is relatively small, only large data sets show meaningful building time statistics. As shown in Table 15, Hilbert order seems cost less time to build. Yet, this is not altogether a bad news for the Kd-tree sequence. Since it has been included in the CPU times of Tables 1–7, the overall performance of the Kd-tree sequence is still superior to Hilbert curve order.

Table 15 Sequence building time for Kd-tree and Hilbert orders in seconds

	Cube		Cylinder	
Datasize	Kd-tree	Hilbert order	Kd-tree	Hilbert order
500 000	1	1	1	1
1 000 000	2	1	1	1
1 500 000	2	2	1	2
2 000 000	3	2	2	2
2 500 000	4	2	3	2
3 000 000	5	2	4	3
3 500 000	6	3	5	3

3.3 The results for three practical models.

We have tested three practical examples, the happy Buddha, a constrained block, and a gear box.

The happy Buddha, Fig. 7, is downloaded from http://graphics.stanford.edu/data/3Dscanrep. There are two kinds of information in the downloaded data file, namely vertices and triangles; only its vertices are used as input points in our test.

The constrained block, Fig. 8, is in the form of a cube. Three of its six edges are constrained in a wireframe structure in its real environment and thus causing high stress along the edges. We are going to generate a finite element (FE) mesh for the analysis and a large number of nodes are placed near the constrained edges to capture regions of high stress variation.

The gear box, Fig. 9, is a rather complex mechanical model. We would like to generate a FE mesh for engineering analysis. We know, somehow, its failure zone is on two cross sections and around a line segment at the bottom, and these areas are refined in the mesh generation.

Fig. 7 The happy Buddha

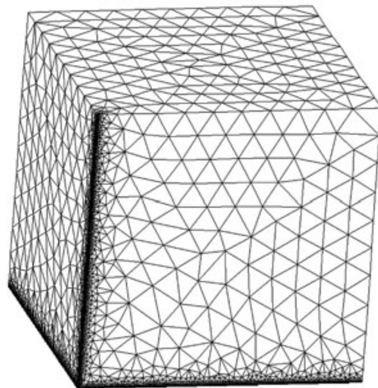

Fig. 8 A constrained cube meshed with element of different sizes

Fig. 9 A gear box meshed with elements of smaller size at specific locations

In the mesh generation, nodes are generated first and they are then inserted by the DT based method to create tetrahedral elements. The mesh generation procedure has been applied to the three models, cube, gear box and happy Buddha so as to compare with CGAL on some practical examples, and the results are listed in Table 16.

Since in random order quasi-uniformly distributed points we can generate a DT in expected linear time, one may question why we bother to use a different order scheme especially when building time for such a scheme is $O(n \lg n)$. The reason is that the building time is only very small for practical n values and the total cost is still much less than that of the random order. Indeed random order can be improved significantly if coupled with jump and walk. However, we have observed the updating time in random order alone is already longer than the total time needed for the new order.

Table 16 Statistics for the four orders (CPU time in seconds), abort means longer than 30 min

	New order	Hilbert	BRIO	Random
Happy Buddha (543 652 points)	30	33	36	223
Constrained cube (4 386 533 points)	237	384	467	abort.
Gear box (3 204 652 points)	175	211	295	abort.

The threshold is 30 min adopted for aborting triangulation of the examples shown in Table 16, in which the largest size tested is less than 5×10^6 points. We would like to compare the performances of the four methods for models of larger size, however, for $n > 5 \times 10^6$, CGAL gets slower in our machine because of the memory requirement. Hence, we could only test one more example with 5.5×10^6 points, which is built by adding some nodes on element edges of the tet-mesh of the constrained cube. The results of this example of 5.5×10^6 points are listed in Table 17. We noticed that the Hilbert order failed in this situation; it seems that Hilbert order needs more space to run than the other schemes. Nevertheless, the performance of the new order is still very stable and is much faster than the other schemes tested.

Table 17 Statistics for larger size of points, timing in seconds, abort means longer than 30 min

	New order	Hilbert	BRIO	Random
Refined constrained cube (5 500 000 points)	454	fail	1 139	abort.

4 Conclusions and discussions

In this paper, a deterministic nodal sequence for the construction of incremental DT is proposed. The modified Kd-tree sequence advocated is easy to implement in any dimensions and its performance is superior to Hilbert curve order and BRIO, especially for non-uniformly distributed points. However, the exact analysis of the time complexity is not trivial, which depends not only on the method used but also the pattern of the point distribution. As a result, the proposed insertion sequence was rigorously tested and compared with existing insertion orders through a wide range of point distributions of variable data size. The Kd-tree sequence is more efficient as it guarantees an almost uniform separation between insertion points, which seems to be effective in reducing the conflicting (non-Delaunay) elements during triangulation. Employing the parent node as the search hint would

also help a great deal in reducing the walking time of an insertion point towards its base.

While the complexity for node relabeling is $O(n \lg n)$, theoretical analysis for computing a DT following the Kd-tree insertion sequence is rather difficult. However, the tests showed that its expected time is of order $O(n)$ for random point sets. Moreover, we found that the performance deteriorated upon switching split rule for building Kd-tree from the cut-longest-edge rule to the commonly adopted alternative rule. Furthermore, the efficiency of the new sequence is almost the same for taking either parent or sibling points as a search hint, since the two points are pretty close indeed. Finally, it is interesting to extend the Kd-tree sequence to higher dimensions and we have confidence that it will still be more efficient than the existing insertion sequences, especially for non-uniform point distributions.

Acknowledgement The authors would like to thank the anonymous reviewers for their helpful suggestions to improve the paper.

Appendix I

Procedure 2 —— Build Kd-tree(P)
Input: A set of points P.
Output: The root of a Kd-tree storing P.

If P contains only one point then return a leaf storing this point
else

Choose a median plane, say h, which cuts the longest edge of the bounding box of P at median point v to split P into two subsets. Let P_1 be the set of points on the negative side of h and P_2 be the set of points on positive side of h.
T_{left} = buildkdTree(P_1);
T_{right} = buildkdTree(P_2);
Create a node T storing v, make T_{left} the left child of T and make T_{right} the right child of T
return T
end procedure

References

1 Shewchuk, J. R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete & Computational Geometry **18**, 305–363 (1997)
2 Borouchaki, H., George, P. L., Lo, S. H.: Optimal Delaunay point insertion. Int. J. for Num. Methods in Engg. **39**, 3407–3437 (1996)
3 Borouchaki, H., Lo, S. H.: Fast Delaunay triangulation in three dimensions. Computer Methods in Applied Mechanics and Engineering **128**, 153–167 (1995)
4 Joe, B.: Construction of three-dimensional Delaunay triangulations using local transformations. Computer Aided Geometric Design **10**, 123–142 (1989)
5 Edelsbrunner, H., Shah, N. R.: Incremental topological flipping works for regular triangulations. Algorithmica **15**, 223–241 (1996)
6 Amenta, N., Choi, S., Rote, G.: Incremental constructions con BRIO. In: Proc. 19th Annu. ACM Sympos. Comput. Geom. 211–219 (2003)
7 Liu, Y., Snoeyink, J.: A comparison of five implementations of 3rd Delaunay tesselation. Combinatorial and Computational Geometry **52**, 439–458 (2005)
8 Zhou, S., Jones, C. B.: HCPO: an efficient insertion order for incremental Delaunay triangulation. Inf. Process. Lett. **93**, 37–42 (2005)
9 Boissonnat, J. D., Devillers, O., Samuel, H.: Incremental construction of the Delaunay graph in medium dimension. In: Proc. 25th Annual Symposium on Computational Geometry, 208–216 (2009)
10 Buchin, K.: Organizing point sets: Space-filling curves, Delaunay tessellations of random point sets, and flow complexes. [Ph.D. Thesis], Free University, Berlin (2007)
11 Buchin, K.: Constructing Delaunay triangulations along space-filling curves. In: Proc. 2nd Internat. Sympos. Voronoi Diagrams in Science and Engineering, 184–195(2005)
12 CGAL Editorial Board: CGAL User and Reference Manual. Edition 4.0, 2167–2387 (2012)
13 Peano, G.: Sur une courbe, qui remplit toute une aire plane. Mathematische Annalen **36**, 157–160 (1890) (in German)
14 Hilbert, D.: Über die stetige Abbildung einer Linie auf ein Flächenstück. Mathematische Annalen **38**, 459–460 (1891) (in German)
15 Guibas, L. J., Knuth, D. E., Sharir, M.: Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica **7**, 381–413 (1992)
16 de Berg, M., van Kreveld, M., Overmars, M., et al.: Computational Geometry: Algorithms and Applications. 2nd ed. Springer-Verlag, Berlin, Germany (2000)
17 Devroye, L., Mücke, E., Zhu, B.: A note on point location in Delaunay triangulations of random points. Algorithmica **22**, 477–482(1998)
18 Devroye, L., Lemaire, C., Moreau, J. M.: Fast Delaunay point location with search structures. In: Eleventh Canadian Conference on Computational Geometry, 15–18 (1999)
19 Devroye, L., Lemaire, C., Moreau, J. M.: Expected time analysis for Delaunay point location. Computational Geometry **29**, 61–89 (2004)