Weierstraß-Institut
für Angewandte Analysis und Stochastik
Leibniz-Institut im Forschungsverbund Berlin e. V.

Preprint ISSN 0946 – 8633

TetGen, towards a quality tetrahedral mesh generator

Hang Si[1]

submitted: March 5, 2013

[1] Weierstrass Institute
Mohrenstr. 39
10117 Berlin
Germany
E-Mail: hang.si@wias-berlin.de

No. 1762

Berlin 2013

2010 *Mathematics Subject Classification.* 65M50, 65N50, 65D18, 68U05, 68N99.

Key words and phrases. tetrahedral mesh generation, Delaunay tetrahedralization, constrained Delaunay, boundary recovery, mesh refinement, mesh quality, flips, edge removal.

This work was supported by the Weierstrass Institute for Applied Analysis and Stochastics (WIAS), Berlin, Germany.

Edited by
Weierstraß-Institut für Angewandte Analysis und Stochastik (WIAS)
Leibniz-Institut im Forschungsverbund Berlin e. V.
Mohrenstraße 39
10117 Berlin
Germany

Fax: +49 30 20372-303
E-Mail: preprint@wias-berlin.de
World Wide Web: http://www.wias-berlin.de/

Abstract

TetGen is a C++ program for generating quality tetrahedral meshes aimed to support numerical methods and scientific computing. It is also a research project for studying the underlying mathematical problems and evaluating algorithms. This paper presents the essential meshing components developed in TetGen for robust and efficient software implementation. And it highlights the state-of-the-art algorithms and technologies currently implemented and developed in TetGen for automatic quality tetrahedral mesh generation.

1 Introduction

A tetrahedral mesh is a partition of a three-dimensional geometric domain into a set of non-overlapping tetrahedra. This type of partitions has many favorable properties. For instances, it is able to easily represent domain boundaries with arbitrarily complicated shape, it can be refined and coarsened locally, and it can be created fully automatically.

Meshes find use in many areas. The development of TetGen is mainly motivated by the numerical solution of partial differential equations, such as the finite element and finite volume methods. The first step of these methods is to obtain an appropriate mesh of the simulation domain. The quality of the mesh will tremendously affect the accuracy and convergence of these methods.

The term “mesh quality"depends on the physical problems and the applied numerical methods. Frey and George [31] gave a nice exposition on various meanings of mesh quality. Numerically, the mesh quality can be assessed by taking into account several measures on element shape, size, and orientation, see e.g. [37]. Following the definition of Bern and Eppstein [6], an *optimal mesh* is a partition of the domain, that is best according to some criterion that measures the size, shape, or number of elements. The corresponding problem,

referred as the *quality mesh generation* problem, can be generally formulated as: *how to create an optimal mesh with respect to a given mesh quality measure?* It is a complex problem which requires an interdisciplinary research combing topics in mathematics, computer science, and engineering [26].

Technologies for mesh generation (or synonymously called grid generation) have been greatly advanced in recent decades. Relatively comprehensive sources are found in the book edited by Thompson, Soni, and Weatherill [78] and the book of Frey and George [31]. There are established methods for tetrahedral mesh generation, such as Octree [82] , Advancing-Front [47, 46] , Delaunay [4, 35, 80], and combination of them [48]. Great achievements have been made by engineering techniques in automatically tetrahedralizing complex three-dimensional configurations. However, most of the early methods are heuristic and lack of theoretical background. Robust software implementations are rare and may not be freely available. Now it is commonly acknowledged that the construction of efficient and robust mesh generation techniques can only be achieved through algorithms with solid theoretical justification. Provable algorithms for mesh generation are evolving in the field of computational geometry [6] and related mathematical areas [26].

In two dimensions, Ruppert's algorithm [62], extending an algorithm of Chew [19], provably generates a triangular mesh with a bounded smallest angle, and the total number of triangles is optimal. Shewchuk provided a robust and efficient implementation of Ruppert's algorithm in the freely available program `Triangle` [66]. The original development of TetGen (from 2000 to 2001) was largely inspired by the program `Triangle` and aimed to extend it into three dimensions. However, many theoretical questions as well as algorithmic issues in tetrahedral mesh generation are not settled yet, an extending into three dimensions is far from straightforward.

It turns out that one of the fundamental difficulties is to preserve an arbitrary edge or face in a tetrahedral mesh. This issue ubiquitously appears in almost all tetrahedral meshing problems, regardless of the method used. Although a quality tetrahedral mesh can be generated through various mesh refinement and mesh improvement techniques, there is no theoretically sound guarantee on some useful mesh quality measures, such as the minimum and maximum dihedral angles.

TetGen has been a research project of WIAS since 2002. The goal of developing TetGen is two-fold: First, it provides a research tool for evaluating state-of-the-art algorithms and technologies for quality tetrahedral mesh generation; Second, it provides a robust, efficient, and easy-to-used software for various applications. It is one of the meaningful ways to combine researches and applications. TetGen's source code is freely available through the website `http://www.tetgen.org`.

The current library of TetGen includes efficient algorithms for generating Delaunay tetrahedralizations, constrained Delaunay tetrahedralizations, and good quality isotropic tetrahedral meshes. These algorithms can tetrahedralize three-dimensional objects with arbitrary complexity. Specific input constraints, such as edges and triangles, can be preserved in the generated tetrahedral meshes. TetGen implements extensive local mesh operations to efficiently refine and improve the mesh quality and the mesh adaptivity. TetGen uses advanced techniques in computational geometry to improve its robustness. Figure 1 illustrates the workflow of TetGen to generate a quality tetrahedral mesh of a

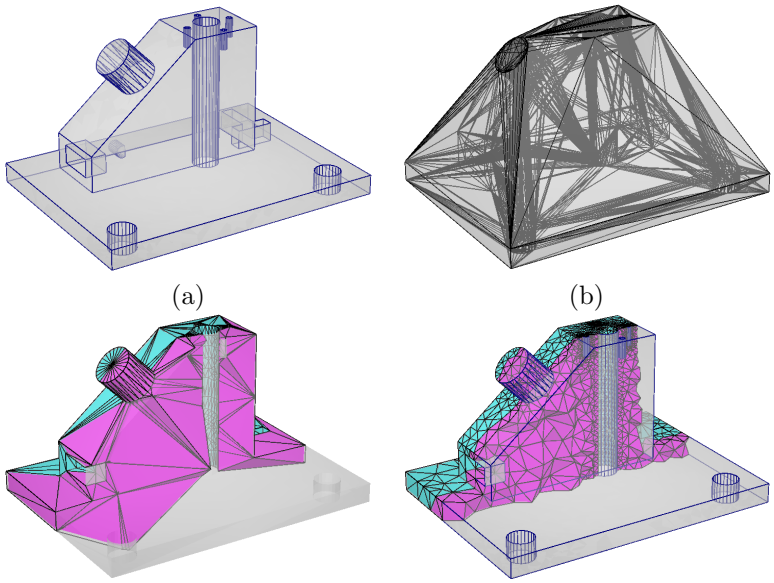


Figure 1: The workflow of generating a quality tetrahedral mesh in TetGen: (a) the mesh domain (`cami1a`) – a three-dimensional piecewise linear complex (PLC), (b) the Delaunay tetrahedralization of the input vertices, (c) a constrained Delaunay tetrahedral mesh, and (d) a quality tetrahedral mesh. A cut view is made in (c) and (d) to visualize the internal tetrahedra.

three-dimensional domain.

Outline

The rest of this paper is organized as follows. Section 2 briefly describes the input objects of TetGen. The mesh data structure of TetGen is explained in Section 3. Section 4 describes the important local mesh operations (flips) including a new edge removal algorithm that is first implemented in TetGen. Section 5 describes the techniques for robustly implementing geometric algorithms. The algorithms and techniques used by TetGen are described in the following sections: Delaunay tetrahedralizations (in Section 6), boundary conformity (in Section 7), and quality tetrahedral mesh generation (in Section 8). Finally, a summary and outlook of future works are given in Section 9.

2 Piecewise Linear Complexes

The input of TetGen is a three-dimensional *piecewise linear complex* (PLC) $\mathcal{X}$, introduced by Miller et al. [51]. It is relatively simple and is able to model a domain with internal boundaries (non-manifolds). A PLC is a set of vertices, edges, polygons, and polyhedra, collectively called *cells*, that satisfies the following properties: (1) The boundary of each cell in $\mathcal{X}$ is a union of cells in $\mathcal{X}$. (2) If two distinct cells $F, G \in \mathcal{X}$ intersect, their intersection is a union of cells

in $\mathcal{X}$, all having lower dimension than at least one of F or G. Figure 1 (a) illustrates a three-dimensional PLC.

The *underlying space* of a three-dimensional PLC $\mathcal{X}$, denoted $|\mathcal{X}|$, is the union of all cells of $\mathcal{X}$. The *boundary complex* of $\mathcal{X}$, denoted $\partial\mathcal{X}$, is the subset of cells of $\mathcal{X}$ whose dimensions are less than 3. It is a two-dimensional PLC. Any geometric domain $\Omega \subset \mathbb{R}^3$ can be modeled by a PLC $\mathcal{X}$ such that $|\mathcal{X}|$ is homeomorphic to Ω, and the geometric shape of Ω is approximated by $|\partial\mathcal{X}|$.

A PLC is eligibly represented by a Boundary Representation (B-Rep) solid with only linear facets. That is, the input only contains a list of vertices, edges, and polygons of the PLC and their incidence relations, such as a surface mesh of the boundary complex of the PLC. B-Rep solids are generally supported by most of CAD softwares. The definition of PLC requires that a valid B-Rep input must contain no self-intersections between its cells and it must be water-tight.

3 Tetrahedral Mesh Data Structure

A *tetrahedralization* $\mathcal{T}$ is a three-dimensional simplicial complex. It decomposes its underlying space $|\mathcal{T}| \subseteq \mathbb{R}^3$, where $|\mathcal{T}|$ is not necessarily convex, and it may have an arbitrary topology [26]. We say that $\mathcal{T}$ is a *tetrahedral mesh* of a three-dimensional PLC $\mathcal{X}$ if: (1) every cell of $\mathcal{X}$ is represented by a union of simplices of $\mathcal{T}$, and (2) $|\mathcal{X}| = |\mathcal{T}|$.

There are many mesh data structures can be used for representing tetrahedral meshes, see e.g., [25, 32, 7]. For mesh generation purpose, two equally important considerations are the memory usage and computational efficiency, which are however contradict to each other. The data structure used in TetGen is specialized for tetrahedral mesh generation. It is a trade-off between fast local navigations and modifications (by storing extra incidence informations) and a compact space usage (by reducing the additional informations as small as possible).

3.1 Representation

A tetrahedralization $\mathcal{T}$ in TetGen is represented by a common tetrahedron-based data structure. It stores the set of tetrahedra and vertices of $\mathcal{T}$. The basic structure of a tetrahedron contains four pointers to its neighbors and four to its vertices. To distinguish the common faces/edges in the neighbors of a tetrahedron, each neighbor contains an extra 4-bit integer (explained in Section 3.2). An additional 16-bit integer is included in the structure for setting flags on faces, edges, and the tetrahedron itself. These flags are used to speed up various algorithms like the vertex insertion and flips (in Section 4.3). In total, a tetrahedron basically uses 8 pointers and 32 bits. Each vertex contains its x-, y-, and z-coordinates, and a pointer to a tetrahedron to which it belongs. TetGen stores the input index of this vertex. It is used by a symbolic perturbation algorithm (in Section 6.1). Both structures for tetrahedra and vertices can include user data.

TetGen always maintains an extended tetrahedralization which includes *fictitious tetrahedra* formed by joining exterior boundary faces of the original tetrahedralization to a dummy"point at infinity". Hence every face in the extended

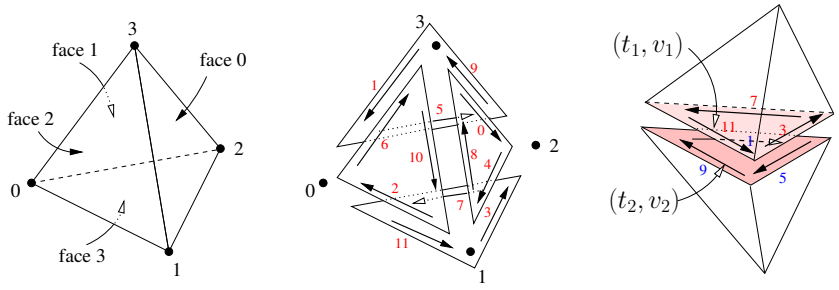


Figure 2: Left: A numbering of the vertices and faces of a tetrahedron. Middle: A numbering of the 12 versions (directed edges) in a tetrahedron. Right: A neighbor query on the handle (t_1, v_1) returns the handle (t_2, v_2). In this example, $v_1 = 11$, $v_2 = 9$, and $v_0 = 5$.

tetrahedralization belongs to two tetrahedra. This property simplifies the implementation of tetrahedralization algorithms. This concept was used by Guibas and Stolfi [36] for modeling 2-manifolds and their duals.

When $\mathcal{T}$ is a tetrahedral mesh of a three-dimensional PLC $\mathcal{X}$, TetGen stores additionally the set of subfaces and subsegments of $\mathcal{T}$. A triangle-based data structure is used to represent the subfaces and their connections. The structure of a subface basically contains nine pointers to its neighbors, vertices, and subsegments. Since the boundary of $\mathcal{X}$ may be non-manifold, the pointers to neighbors of the subfaces form a loop of singly linked list at their common edges. Extra pointers are allocated in tetrahedra, surfaces, and segments to point each others.

3.2 Primitives Operations

Navigating and manipulating a tetrahedralization in TetGen are accomplished through a set of primitives. They are conceptually similar to those proposed in [36, 25]. While the implementation of these primitives is specialized for tetrahedron-based data structure.

The atomic unit on which these primitives are operated is a structure, called a "handle", which is a pair (t, v), where t is a pointer to a tetrahedron and v is an integer, called *version*, which refers to a specific face of the tetrahedron, and a specific edge of this face. There are 12 versions in a tetrahedron, corresponding to the 12 even permutations of its 4 vertices. They can be uniquely represented by the 12 directed edges shown in Figure. 2. Every three edges form an *edge ring* in an *oriented face* of the tetrahedron, and there are four distinct edge rings (oriented faces). We number the 12 versions of a tetrahedron from 0 to 11. They can be encoded into a 4-bit integer such that the two lower bits encode the index (from 0 to 3) of the oriented face to which this edge belongs, and the two upper bits encode the index (from 0 to 2) of this edge in this oriented face. Figure 2 shows such a numbering.

Moving within the same edge ring of a handle (t, v) is a simple arithmetic operation on its version, i.e., $(t, (v + i)$ **modulo** $12)$, where $i \in \{4, 8\}$. Moving between two edge rings can be done by using a global lookup table, $L[0...11]$, such that (t, v) and $(t, L[v])$ refer to the same edge of t, while they are in two

oriented faces of t.

When traveling from one tetrahedron to one of its adjacent tetrahedra, it is desired that the neighbor query on a handle (t_1, v_1) will return the handle (t_2, v_2), such that they both refer to the same edge of the same face shared by t_1 and t_2, see Figure 2. The data structure already stores the pointer to t_2 in t_1. It remains to determine v_2 from v_1. Since an edge of an oriented face in t_1 may be anyone of the three edges of the oriented face in t_2, the data structure stores the handle (t_2, v_0) in t_1, such that (t_2, v_0) corresponds to the 0-th edge in the oriented face of t_1, see Figure 2 for an example. It is easy to verify, that

$$v_0 := (v_2 \textbf{ and } 3) + (((v_1 \textbf{ and } 12) + (v_2 \textbf{ and } 12)) \textbf{ modulo } 12).$$

Then the neighbor query of (t_1, v_1) is accomplished in two steps: first the handle (t_2, v_0) is obtained, then the wanted version v_2 is calculated, where

$$v_2 := (v_0 + 12 - (v_1 \textbf{ and } 12)) \textbf{ modulo } 12.$$

Note that the values of v_0 and v_2 in the above formulae can all be previously calculated and stored by using two-dimensional tables of size 12×12.

In order to quickly access the versions during the neighbor queries, TetGen uses the same trick of `Triangle` [66], i.e., a version is encoded in the pointer (in the low 4 bits) to each neighbor. This requires that the addresses of the memory allocated to tetrahedra records are 16-byte-aligned.

4 Local Mesh Transformations

A *local mesh transformation* (also referred as *topological transformation*) replaces a set of tetrahedra with a set of different tetrahedra such that they occupy the same space (referred as *cavity*). It can be just an *elementary flip* which only interchanges two minimal sets of tetrahedra, or a combination of elementary flips, or a complicated algorithm like inserting a vertex. It is questionable whether or not the deletion of a vertex is a local mesh transformation. It is not always possible to re-fill the cavity resulted by removing a vertex with a set of tetrahedra only using the vertices of the cavity.

Local mesh transformations are essential functionalities needed in almost all kinds of meshing problems. A comprehensive design and realization of these operations will produce not only an efficient program but also a succinct code.

4.1 Elementary Flips

According to Radon's theorem [57], there are four elementary flips in $\mathbb{R}^3$, performed within the convex hull of five non-coplanar points. These are respectively: 1-to-4, 4-to-1, 2-to-3, and 3-to-2 flips, where the numbers indicate the number of tetrahedra before and after each flip, see Figure 3. The 1-to-4 as well as its reverse 4-to-1 flips are the simplest cases of inserting and deleting a vertex, respectively. The 2-to-3 flip removes a face and creates an edge. It is also referred as a *face-to-edge* flip. The 3-to-2 flip, which reverses the 2-to-3 flip, is also referred as the *edge-to-face* flip. It is the simplest case of removing an edge.

TetGen implemented all the elementary flips except the 1-to-4 flip which is done within the vertex insertion routine (in Section 4.4). Each routine,

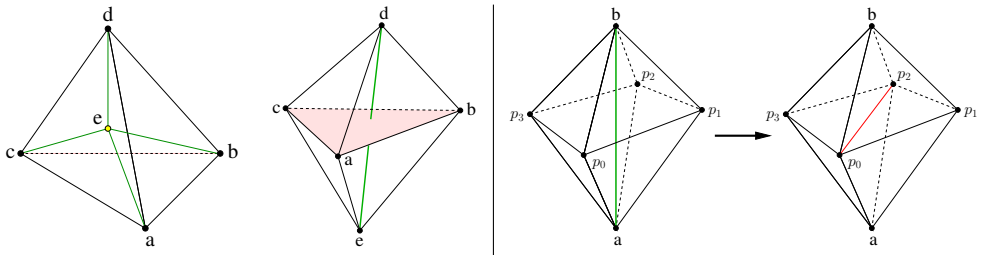


Figure 3: Left: the four types of elementary flips in $\mathbb{R}^3$. Right: a 4-to-4 flip which interchanges two coplanar edges is a combination of a 2-to-3 and a 3-to-2 flips.

`flip`ij, takes an array of $i \in \{2, 3, 4\}$ tetrahedra (handles) as input and returns $j \in \{3, 2, 1\}$ new tetrahedra (handles) by the same array. Fictitious tetrahedra may be involved in each flip. The implementation must accordingly create new fictitious tetrahedra. For example, if one of the two input tetrahedra of `flip23` is fictitious, then all three resulting tetrahedra are fictitious, i.e., they all contain the “point at infinity”.

4.2 Combinations of Flips

One can combine elementary flips to form more complex local transformations. For an instance, the 4-to-4 flip which interchanges two coplanar edges can be regarded as the combination of a 2-to-3 flip and a 3-to-2 flip, see Figure 3. Note that the first 2-to-3 flip will temporarily create a degenerate tetrahedron (whose volume is zero). It is removed immediately by the followed 3-to-2 flip. For this reason, TetGen does not have a routine for doing 4-to-4 flip. Also, by maintaining an extended tetrahedralization, the 2-to-2 flip which occurs on the exterior mesh boundaries becomes a 4-to-4 flip.

Indeed, the 4-to-4 flip is a special case of a more general *n-to-m flip*, where $n \geq 3$ and $m = 2n - 4$, which removes an edge in tetrahedralization. It is a combination of flips of a sequence of $(n-3)$ 2-to-3 flips followed by a final 3-to-2 flip. The orders of the 2-to-3 flips determine the set of resulting tetrahedra. This n-to-m flip was studied in [33] and has been applied in various meshing purposes.

Some other types of combinations of flips were reported. In particular, several operations described in [38] can be viewed as a combination of two n-to-m flips with the restriction that n is either 3 or 4. Experimental evidences were shown that these special operations performed effectively in improving mesh quality [38] and in recovery of specific boundaries in tetrahedral meshes [44].

An edge in a tetrahedralization may belong to an arbitrary number of tetrahedra. Unless the number of tetrahedra is 3, it is not obvious whether or not this edge can be removed by flips. In particular, an edge belongs to $n \geq 4$ tetrahedra can not be removed by the n-to-m flip when no face at this edge can be flipped, i.e., none of the required 2-to-3 flip is possible. However, it may be removed if some edges near it are removed first. This may be a combination of two or more n-to-m flips.

In TetGen, an algorithm which attempts to remove an edge by flips is developed. It is able to combine an arbitrary number (as long as it is possible) of n-to-m flips. Moreover, the sequence of performed flips is remembered. It can be used to restore the original tetrahedralization as if no flip has been ever performed. Here only the basic version of this algorithm is described.

4.3 Edge Removal

Let e be the edge to be removed in a tetrahedralization $\mathcal{T}$. Let $A[0...n-1]$ be the array of n tetrahedra (handles) in $\mathcal{T}$ sharing at e, where $n \geq 3$. These handles (tetrahedra) in A are all aligned at the edge e with the same direction, and they are ordered cyclicly such that the two tetrahedra $A[i]$ and $A[(i+1)$ **modulo** $n]$ share a common face. The idea of the algorithm is similar to the n-to-m flip. It tries to reduce the size of A by performing elementary flips until either the size is 3 or it can not be reduced. While not only tetrahedra containing e but also tetrahedra which do not contain e may be flipped.

The algorithm consists of two subroutines, denoted as `flipnm` and `flipnm_post`, respectively. `flipnm` takes the array A (whose size is n) as input. It does the "forward" flips to reduce the size of A. It returns the current size m of A, where m may be 2 which means e has been removed, otherwise, $m \geq 3$, e is not removed. `flipnm_post` must be called immediately after `flipnm`. It basically frees the memory used by `flipnm`. In addition, it is able to do the "backward" flips to restore the original tetrahedralization before doing `flipnm`.

Let F be the set of faces containing e, and E be the set of edges of the faces in F except e ($|F| = n$ and $|E| = 2n$). We say that a face in F (or an edge in E) is *A-reducible* if the removal of this face (or edge) will reduce the size of A by 1. Not all faces in F (or edges in E) are A-reducible (explained later).

The subroutine `flipnm`$(A[0...n-1])$ does the following three steps:

Step *(1)*: If $n = 3$, then either e is removed by a `flip32`, or it is not flippable. It returns the current size m of A. Otherwise, it goes to step (2);

Step *(2)*: $(n > 3)$ It tries to remove an A-reducible face in F. Let $f \in F$ be such a face. If f is flippable, it is removed by a `flip23`. The array A is shrunk by 1. The last entry, $A[n-1]$, is re-used to store the information of `flip23`, refer to Figure 4. It then recursively calls `flipnm`$(A[0...n-2])$. If no A-reducible face in F can be removed, it goes to step (3);

Step *(3)*: $(n > 3)$ It tries to remove an A-reducible edge in E. Let $e_1 \in E$ be such an edge. It first initializes an array $B[0...n_1-1]$ of n_1 tetrahedra sharing at e_1, where $n_1 \geq 3$, then it calls `flipnm`$(B[0...n_1-1])$. If e_1 has been removed, then the array A is shrunk by 1. The last entry, $A[n-1]$, is re-used to store the information of `flipnm`$(B[0...n_1-1])$, refer to Figure 4. It then recursively calls `flipnm`$(A[0...n-2])$. Otherwise, e_1 is not removed, it calls `flipnm_post`$(B[0...n_1-1], m_1)$ to free the memory. If no edge in E can be removed, it returns the current size m of A.

Which faces or edges are not A-reducible? Note that `flipnm` may have two arrays of tetrahedra at the same time, such as A and B in above. Let $F_{A\cap B}$ be the set of faces shared by A and B, then the removal of any face in $F_{A\cap B}$ does not reduce the size of A. Hence faces in $F_{A\cap B}$ are not A-reducible. Similarly, the set $E_{A\cap B}$ of edges of faces in $F_{A\cap B}$ except e are not A-reducible.

Each saved flip (in array A and B) contains informations of that flip which enable us to undo this flip. It turns out that the only information we need to

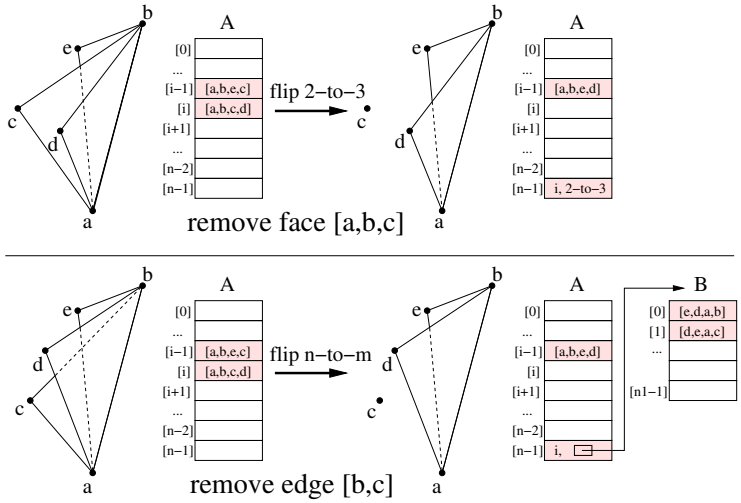


Figure 4: The contents of array A before and after flips. $e = [a, b]$ is the edge to be removed. Before the flip (Left), the face $[a, b, c]$ is shared by two tetrahedra $A[i-1] = [a, b, e, c]$ and $A[i] = [a, b, c, d]$. After the flip (Right), the face $[a, b, c]$ is removed by either a 2-to-3 flip (Top) or by a n-to-m flip on one of its edges (here is [b,c]) (Bottom). In both cases, the array A is shrunk by 1, and $A[i-1] = [a, b, c, d]$ is the new created tetrahedron containing $[a, b]$. The last entry of A is used to save the informations of the flip which removes the face $[a, b, c]$.

save a `flip23` is the index i in the array A, such that $A[i-1]$ and $A[i]$ were the two flipped tetrahedra. For saving a `flipnm` (to remove an edge shared by $A[i-1]$ and $A[i]$), in addition to the index i, the address of the array (e.g., B) need to be saved. It is assumed that the handle structure has enough space to save these informations.

The complexity of this algorithm can be exponential with respect to the number of total flipped edges in the process. TetGen adds a 'level' (> 0) parameter into `flipnm` to limit the maximal number of recursions in `flipnm`.

4.4 Vertex Insertion

In principle, vertex insertion is just a combination of elementary flips. A 1-to-4 flip inserts a vertex lies in the interior of a tetrahedron. If a vertex lies on a face shared by two tetrahedra t_1 and t_2, it can be inserted by first doing a 1-to-4 flip in t_1, this creates a degenerated tetrahedron t_3 which is removed immediately by second doing a 2-to-3 flip on t_3 and t_2. This is equivalent to a 2-to-6 flip. If a vertex lies on an edge which is shared by n tetrahedra, where $n \geq 3$, then it can be inserted by doing a combination of a first 1-to-4 flip, followed by $(n-2)$ 2-to-3 flip(s), and a final 3-to-2 flip. Several degenerated tetrahedra are temporarily involved in these flips. This is equivalent to an n-to-$2n$ flip.

TetGen unifies the above special cases into one routine, `insertpoint`. Let $\mathcal{C}$ be the set of tetrahedra which intersect the new vertex $\mathbf{v}$. $|\mathcal{C}| \in \{1, 2, n\}$. Delete all tetrahedra in $\mathcal{C}$ creates a cavity inside the tetrahedralization. The cavity is

star-shaped with respect to $\mathbf{v}$. Hence it can be filled by a set of new tetrahedra formed by the boundary faces of the cavity and $\mathbf{v}$. This process does not create any intermediate tetrahedra and is general faster than performs a combination of flips.

5 Robust Geometric Predicates

Geometric predicates are simple tests of spatial relations of a set of geometric objects, such as points, lines, planes, and spheres. Two predicates used in generating Delaunay tetrahedralizations are the `orient3d` which decides the orientation of four points and the `in_sphere` which decides if a point lies inside the circumscribed sphere of a tetrahedron given by its four vertices. Each of these tests is performed by evaluating the sign of a determinant of a matrix whose entries are the coordinates of the involved points, i.e.,

$$\texttt{orient3d}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) = \text{sign}(\det(A)) \quad \text{and} \quad \texttt{in\_sphere}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{e}) = \text{sign}(\det(B)),$$

where

$$A = \begin{bmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} a_x & a_y & a_z, & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z, & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z, & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z, & d_x^2+d_y^2+d_z^2 & 1 \\ e_x & e_y & e_z, & e_x^2+e_y^2+e_z^2 & 1 \end{bmatrix}.$$

If the computation is performed by using the floating-point numbers, e.g., the `float` or `double` numbers in C/C++, which only have finite precisions, roundoff errors may occur and may be accumulated. Geometric algorithms are very sensitive to numerical rounding errors. They may either lead to a wrong result or eventually cause a crash of the program. Computing the predicates exactly will avoid the rounding error and make the program robust [81].

5.1 Filtered Predicates

Exact arithmetics are usually expansive to compute. For predicates like `orient3d` and `in_sphere`, where only the sign of the computation is of interested, a reasonably fast approach is to use the so-called *arithmetic filters* [29, 11] which are upper bounds of the rounding errors. One can evaluate the polynomial in floating-point first, together with some estimation of the rounding error, and fall back to exact arithmetic only if the error is too big to determine the sign.

Depending on how the error bounds are computed, filters can be classified into *static*, *semi-static*, and *dynamic* filters [11]. The static filters are the most efficient. They can safely and quickly answer many "easy cases", while they are less accurate and may fail often. The dynamic filters are more accurate but require more computational time. Devillers and Pion [23] studied the performances of various combinations of filters for the two predicates.

TetGen used the robust implementation of the two predicates by Shewchuk [65]. It is built upon the techniques of arbitrary precision floating-point arithmetics [56] (based on the IEEE standard) and an adaptive scheme to automatically extend the precisions. Shewchuk's predicates use dynamic filters. TetGen takes the

advantage of knowing the range of the input data at runtime. It adds a static filter in each of Shewchuk's predicates computed from the bounding box of the input point set [23]. This further improves the performances of TetGen.

TetGen can alternatively use other available robust implementations of these predicates, such as those in the CGAL library [12], which include efficient dynamic filters based on interval arithmetics [11].

6 Delaunay Tetrahedralizations

Delaunay triangulations, and their dual Voronoi diagrams, are the most well studied structures in computational geometry. A *Delaunay triangulation* of a set of d-dimensional point set V is a d-dimensional simplicial complex $\mathcal{D}$ such that every simplex in $\mathcal{D}$ has the *empty sphere* property [22], i.e., it has a circumscribed sphere whose inside contains no other vertex of V, and the underlying space $|\mathcal{D}|$ is the convex hull of V. Any simplex whose vertices are in V and it satisfies the empty sphere property is called a *Delaunay* simplex. If V is in *general position*, i.e., no $d+2$ vertices of V share a common sphere, then all Delaunay simplices form the unique Delaunay triangulation of V.

According to McMullen's Upper Bound Theorem [49], a Delaunay tetrahedralization of a set of n points in $\mathbb{R}^3$ may have $O(n^2)$ tetrahedra. This bounds the optimal worst-case runtime for any algorithm which constructs Delaunay tetrahedralizations. While many data sets appear in applications have linear-sized Delaunay tetrahedralizations, which can be constructed efficiently. A number of different algorithms have been proposed [10, 79, 20, 5, 28].

6.1 Incremental Construction

Two well-known algorithms for constructing Delaunay tetrahedralizations are the Bowyer-Watson algorithm [10, 79] and the incremental flip algorithm [28]. Both are incremental, i.e., inserting one point at a time. Each point is first located, then inserted. The only difference between these two algorithms is the way to recover the Delaunay tetrahedralization which includes the new vertex, either by growing a cavity and re-tetrahedralizing it, or by doing a sequence of elementary flips. Both algorithms guarantee the correctness.

The Bowyer-Watson algorithm appears simpler and empirically it behaves more efficiently than the incremental flip algorithm does. Therefore it is commonly used. An efficient implementation of the Bowyer-Watson algorithm is given in detail by Borouchaki et al [9]. On the other hand, Mücke [53] presented a detailed implementation of the incremental flip algorithm.

Liu and Snoeyink [45] provided a comparison of several implementations of incremental algorithms. They pointed out that the performance of an incremental algorithm is very much affected by the speed of point location. They showed that the speed of point location can be significantly improved if the points are previously sorted. Their method first sorts the points into subgroups (using bit-leveling), then orders points within each subgroup along a space-filling curve, such as the Hilbert curve. By this ordering, two near points in space are likely close in their insertion orders. This allows the simple stochastic walk algorithm [24] quickly locates the point by starting its search from a tetrahedron containing the previous inserted point.

Boissonnat et al [8] presented an efficient implementation of incremental construction which is available in the library CGAL [12]. They pre-sorted the points similarly as Liu and Snoeyink [45] while the Biased Randomized Insertion Order (BRIO) of Amenta et al [1] is used to group the points. The hope is to keep the geometric locality while ensures the randomized feature of the points in the point set.

TetGen implemented both of the Bowyer-Watson and the incremental flip algorithms in the routine `incrementaldelaunay`. The points are pre-sorted into an array as described by Boissonnat et al [8], then inserts the points in this order. The simple stochastic walk algorithm [24] is used for point location. Robustness is fully guaranteed by using the two robust geometric predicates, i.e., `orient3d` (for point location) and `in_sphere` (for updating the Delaunay tetrahedralization).

TetGen used a simplified symbolic perturbation scheme [27] to handle the *degenerate cases*, i.e., 5 or more points share a common sphere, so that the `in_sphere` test never returns a zero. Thus there is always one canonical Delaunay tetrahedralization of any point set. It perturbs the weights of the weighted points in the fourth dimension. Let $\mathbf{a}' = (a_x, a_y, a_z, a_x^2 + a_y^2 + a_z^2 - a_w) \in \mathbb{R}^4$ be the *weighted point* of $\mathbf{a} = (a_x, a_y, a_z) \in \mathbb{R}^3$, where $a_w \in \mathbb{R}^+$ is the *weight* of $\mathbf{a}'$. The wights are chosen in such a way that the weights of any two weighted points $\mathbf{a}'$ and $\mathbf{b}'$ are in different magnitudes, e.g., $a_w >> b_w$. Then if five points $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e} \in \mathbb{R}^3$ are co-spherical, i.e., $\texttt{in\_sphere}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}) = 0$, the test $\texttt{in\_sphere}(\mathbf{a}', \mathbf{b}', \mathbf{c}', \mathbf{d}', \mathbf{e}') \neq 0$ and the sign of this test is used. It is easy to verify that,

$$\begin{aligned} \texttt{in\_sphere}(\mathbf{a}', \mathbf{b}', \mathbf{c}', \mathbf{d}', \mathbf{e}') &= a_w\ \texttt{orient3d}(\mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}) - b_w\ \texttt{orient3d}(\mathbf{a}, \mathbf{c}, \mathbf{d}, \mathbf{e}) \\ &+ c_w\ \texttt{orient3d}(\mathbf{a}, \mathbf{b}, \mathbf{d}, \mathbf{e}) - d_w\ \texttt{orient3d}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}) \\ &+ e_w\ \texttt{orient3d}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}). \end{aligned}$$

Then the sign of the right hand side of the above equation is solely determined by one of the `orient3d` tests whose preceding weight has the largest magnitude. TetGen used the indices of the points to determine which term to be evaluated.

7 Boundary Conformity

A fundamental problem in mesh generation is how to enforce (or recover) a set of non-existing constraints, such as edges or triangles, in a given triangulation or tetrahedralization. These constraints usually describe the special features in the domain boundaries, such as the boundary complex of a PLC, and they are required to be represented in the generated meshes. It is generally referred as the *boundary conformity* or *boundary recovery* problem.

The two-dimensional problem has been well solved. A triangulation $\mathcal{T}$ which contains a set $\mathcal{L}$ of line segments can always be transformed from the Delaunay triangulation of its vertices by a modified edge flip algorithm [41]. Moreover, $\mathcal{T}$ is called a *constrained Delaunay triangulation* of $\mathcal{L}$ if it is as close as possible to the Delaunay triangulation [42]. Chew proposed an algorithm [18] which can construct a constrained Delaunay triangulation in optimal $O(n \log n)$ time.

In three dimensions, however, boundary recovery is far from solved. There exist (non-convex) polyhedra which have no tetrahedralization with its own vertices [64, 58]. Any algorithm for tetrahedralizing polyhedra must be able to

judiciously create additional points, so-called *Steiner points*, at locations where they are needed. However, it is NP-complete [63] to determine whether a simple polyhedron can be tetrahedralized without Steiner points. On the other hand, there are polyhedra which may require a large number of Steiner points to be tetrahedralized [13]. These facts make the design of theoretical and efficient algorithms difficult.

In many engineering problems, a pre-discretized surface mesh is used as input, and it is required that this surface mesh be exactly preserved in the generated tetrahedral mesh, i.e., no subdivision of the surface mesh is allowed. This requirement makes boundary recovery much harder.

Nevertheless, boundary recovery has been long addressed in the literatures. There are various established methods based on the requirements of how the constraints should be represented in the tetrahedralizations, i.e., either they must be strictly preserved or they are allowed to be subdivided. TetGen has developed efficient methods which support both options for boundary recovery.

7.1 Subdividing Constraints

When the constraints are allowed to be subdivided, theoretical algorithms are available.

Chazelle and Palios [14] proposed an algorithm for tetrahedralizing a simple polyhedron with n vertices and r reflex edges into $O(n+r^2)$ tetrahedra. Besides the theoretical upper bound for the number of Steiner points, this algorithm is too complicated and generally adds Steiner points far more than necessary.

Another approach constructs a *conforming Delaunay tetrahedralization* for a set of constraints [54, 21]. Edges (or triangles) which are non-Delaunay are subdivided into smaller ones by placing Steiner points directly in them, until they are represented by a union of Delaunay edges (or triangles). However, this approach may require an unnecessarily large number of Steiner points. Especially, when there are sharp features (small angles) in the domain boundary.

TetGen constructs a *constrained Delaunay tetrahedralization* (CDT) [67, 72], which is a generalization of the constrained Delaunay triangulation [42, 18] into three dimensions. CDTs have many optimal properties similar to those of Delaunay tetrahedralizations [72]. A crucial difference between a CDT and a (conforming) Delaunay tetrahedralization is that triangles in the constraints are not required to be Delaunay, which frees the CDT to better respect the constraints, refer to Figure 5 for an example. A CDT usually uses much less Steiner points than a conforming Delaunay tetrahedralization does. Hence it can be created more efficiently. Moreover, sharps features are well preserved by CDTs.

7.2 CDT Construction

Shewchuk proved a condition [67] which guarantees that a three-dimensional PLC $\mathcal{X}$ has a CDT (with no Steiner point). A segment $e \in \mathcal{X}$ is *strongly Delaunay* if there exists a circumscribed sphere of e, such that no other vertex of $\mathcal{X}$ lies in and on that sphere. If every segment of $\mathcal{X}$ is strongly Delaunay, then it has a CDT [67]. If a PLC $\mathcal{X}$ does not satisfy this condition, it can always be transformed into another PLC $\mathcal{Y}$ by adding a number of Steiner points on segments of $\mathcal{X}$, such that $\mathcal{Y}$ does have a CDT which is called a *Steiner CDT* of

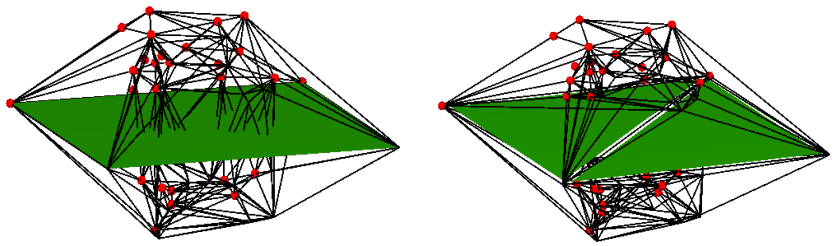


Figure 5: Polygon recovery. Left: inserting a rectangle (shaded) into the Delaunay tetrahedralization of a random vertex set. Right: the resulting CDT including the triangulated rectangle.

$\mathcal{X}$. Note that if the point set of $\mathcal{X}$ is in general position, Shewchuk's condition is simplified to require that every segment of $\mathcal{X}$ is Delaunay.

Existing CDT algorithms [69, 71, 75, 76] all make use of this condition. A (Steiner) CDT of a PLC $\mathcal{X}$ is generated in three steps: (1) create the Delaunay tetrahedralization of the vertices of $\mathcal{X}$; (2) recover the segments of $\mathcal{X}$; and (3) recover the polygons of $\mathcal{X}$. Steiner points are only inserted in the step of segment recovery to ensure that Shewchuk's condition is satisfied.

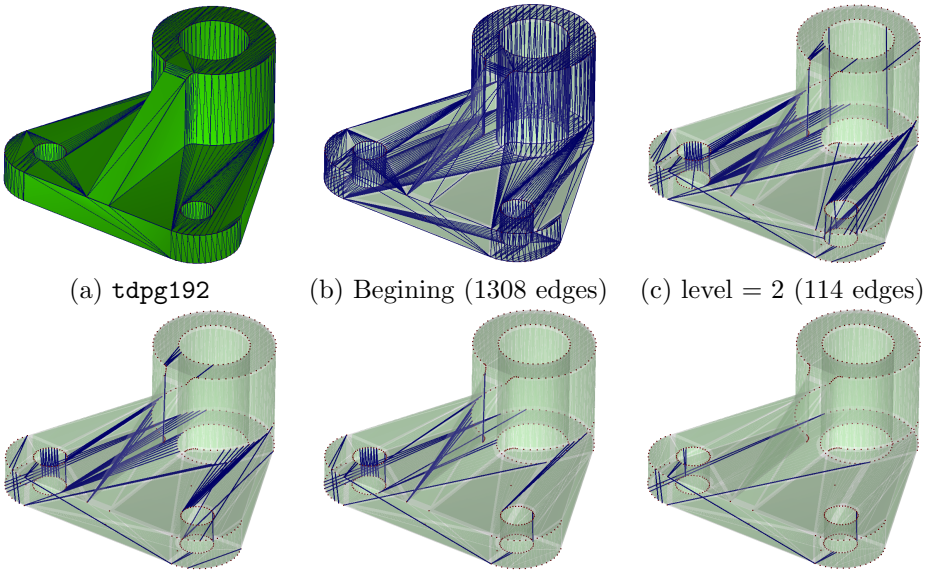
Polygon recovery is the key step in creating CDTs. TetGen implemented two algorithms, a flip algorithm [71] and a cavity re-tetrahedralization algorithm [76] to incrementally recover the polygons, see Figure 5 for an example. A comparison of this two algorithms is given in [77]. It shows that both algorithms behave efficiently. The cavity-retetrahedralization algorithm is more robust in practice.

Although it is theoretically guaranteed that no Steiner point is needed in the polygon recovery, it is based on the PLC assumption: the vertices of a polygon must be co-planar. When the vertex coordinates are represented by the finite precision floating point numbers, the vertices of a polygon are usually not exactly co-planar. This deviation may cause both algorithms [71, 76] fail to recover a polygon. It is shown that this failure can be remedied by adding Steiner points [77]. TetGen's implementation of these two algorithms is able to cope with a polygon that is not perfectly flat. It is described in detail in [77].

7.3 Preserving Constraints

The requirement that no constraint is allowed to be subdivided imposes a stronger restriction to the freedom of placing Steiner points, i.e., they can only be added in the interior of the domain. Neither conforming Delaunay tetrahedralizations nor CDTs are suitable objects for this problem. It is not clear which "Delaunay-like" tetrahedralization is a suitable structure for this problem, and which condition can certify the existence of such tetrahedralization.

Beside the theoretical questions, this problem has been long addressed in mesh generation literatures. The most popular methods are those proposed in [35, 80, 34]. These methods have been shown successful in applications. However, the main technologies used in these methods, such as edge/face swaps (a synonyms for flips), vertex deletions and suppressions are not well-addressed. More sophisticated and heuristics methods to deal with these issues are proposed [44, 34, **?**, **?**]. In practice, these methods are usually hard to implement



(a) `tdpg192` (b) Begining (1308 edges) (c) level = 2 (114 edges)

(c) level = 4 (88 edges) (b) level = 7 (46 edges) (c) level = ∞ (14 edges)

Figure 6: An example of the edge recovery algorithm. The PLC model (432 vertices, 872 triangles) is shown in (a). All edges (1308) to be recovered are highlighted in (b). From (c) - (e), the missing edges after different levels are highlighted. 14 unrecovered edges are shown in (f).

robustly.

TetGen implemented a constructive method for this purpose. It combines the simple ideas in the papers of George et al [35], Weatherill et al [80], and George et al [34]. This method works in three steps: (1) recover edges, (2) recover triangles, and (3) vertex suppression. In (1) and (2), constraints are recovered by an iterative process combined by flips and Steiner points insertion. Constraints may be subdivided. In step (3), all those Steiner points added in constraints are either deleted or repositioned into the interior of the mesh (as it is proposed in [34]).

TetGen developed a simple algorithm to recover edges and triangles. It first uses the edge removal algorithm (described in Section 4.3). This typically leads to a fast recovery and less number of Steiner points. The edge recovery algorithm is described below. The face recovery algorithm is basically done in the same spirit.

7.4 Edge Recovery

The edge recovery algorithm initializes an array of all edges to be recovered, then recovers them one by one. Let e be an edge to be recovered. Let F be the set of faces in current tetrahedralization $\mathcal{T}$ whose interior intersect e, and $F \neq \emptyset$. The algorithm tries to reduce the size of F by removing a face in F one at a time. If all faces in F are removed, e is recovered.

Let f be a face in F. If it is flippable, i.e., by a 2-to-3 flip, it is removed by `flip23`, and the size of F is reduced by 1. Otherwise, there must be an edge

e' of f which causes f not flippable. The algorithm then tries to remove e' by the routine `flipnm`. If e' is removed, then$f$ is also removed, and the size of F is reduced by 1. To ensure that the size of F does not increase during the removal of edge e', we make sure that no new face in $\mathcal{T}$ whose interior intersects e is created. A call back function is fed to the routine `flipnm` to reject flips which violate this condition. (Note that this requirement might be too strong. It might be relaxed.) If e can not be recovered by the above process, we split e by adding a Steiner point in it. A simple choice is just its midpoint. The two resulting edges will be recovered by the same process. Once an edge is recovered in $\mathcal{T}$, it is "locked" in $\mathcal{T}$ and will never be flipped away. This process must terminate with possibly Steiner points added in edges.

The use of `flipnm` is indeed very effective to recover constraints. However, it may run very slow if the number of recursions in `flipnm` becomes large. We introduced a 'level' parameter (> 0) to the routine `flipnm`. It limits the number of recursions in `flipnm`. The edge and face recovery processes are all iterated on the increasing number of 'level's starting at 'level = 1'. An example of edge recovery by increasing the levels in the routine `flipnm` is shown in Fig. 6.

8 Quality Tetrahedral Mesh Generation

Once the boundary can be properly represented by a tetrahedral mesh, the next problem considered by TetGen is how to generate a tetrahedral mesh with good quality.

8.1 Mesh quality

The actual meaning of mesh quality depends on the potential applications. In the context of numerical solution for partial differential equations, it generally means a combination of several measures on the element shape, size, and orientation, see e.g. [37, 70].

Geometrically, a well-shaped tetrahedron should have no very small and very large angles and dihedral angles. By this criterion, the *regular tetrahedron* (whose edge lengths are all equal) is ideal. There are applications which are best solved by *anisotropic elements* whose shapes are elongated and oriented, see e.g. [2, 60]. In these cases, it is necessary to have some small angles in the tetrahedra, but no small dihedral angles.Nevertheless, large angles and dihedral angles should always be avoided [3, 40], such as the *sliver*, which is a type of very flat tetrahedron, it may have good face angles but have dihedral angles arbitrarily close to 0^o and 180^o. Slivers should always be avoided in finite element methods.

On the other hand, it is shown that a *regular triangulation* [83] (or equivalently a weighted Delaunay triangulation [26]) minimizes the linear interpolation error in $L^p (1 < p < \infty)$ norm for a given convex function among all triangulations with the same set of vertices [15]. For example, the Delaunay triangulation is optimal for piecewise linear interpolation to the quadratic function $\|\mathbf{x}\|^2$. However, it is well known that in a Delaunay tetrahedralization may contain slivers. Interestingly, it has been shown that the dual Voronoi diagram of a Delaunay tetrahedralization is the most appropriate partition for a finite volume method [50, 74].

TetGen currently supports several common geometric shape measures, including the smallest face angle, the minimum and maximum dihedral angles. In addition, it is intended to have the Delaunay-Voronoi property for the resulting meshes.

Besides the shape of elements, a control on their sizes is also important. It is essential in adaptive numerical methods whose aim is to seek the best approximated solution at a low computational cost. Typically, a *mesh sizing function* is provided, e.g., through a priori or a posteriori error estimators. It specifies the desired element size (such as the edge lengths) on the domain.

8.2 Mesh Refinement

TetGen considers the following mesh refinement problem: given a three-dimensional PLC $\mathcal{X}$ and an initial tetrahedral mesh of $\mathcal{T}$ of $\mathcal{X}$, a mesh quality measure, and a mesh sizing function H defined on $|\mathcal{X}|$, how to generate a tetrahedral mesh of $\mathcal{X}$ with good mesh quality and the mesh size conforms to H? Currently only isotropic mesh sizing functions are considered.

A central question in this problem is how to efficiently generate and distribute Steiner points so that a tetrahedral mesh of these points simultaneously satisfies the desired properties. Various approaches have been developed for this purpose, such as, Octree [52], sphere packing [51], Longest-Edge Propagation Path (LEPP) [61], and Delaunay refinement [19, 62, 68].

Delaunay refinement is one of the few methods which provide theoretical guarantees on mesh quality and mesh size. It is developed by Chew [19] , Ruppert [62], and Shewchuk [68]. Its main idea is relatively simple, i.e., it updates a conforming Delaunay mesh by inserting the circumcenters of bad-quality triangles or tetrahedra. Delaunay refinement guarantees that the smallest angle of the mesh elements is bounded. Moreover, it also guarantees that the resulting mesh size well-conforms to the local feature size [62]. However, Delaunay refinement may produce slivers, i.e., it has no guarantee on the smallest dihedral angle. A number of approaches have been proposed to remove slivers [16, 43].

The main limitation of Delaunay refinement is that it may not terminate if the input contains *sharp features*, which are small angles and dihedral angles formed by input cells of the PLC. Handling sharp features remains a challenging problem in Delaunay refinement. Several methods have been proposed [17, 55, 59]. However, they are usually very complicated and may introduce a large number of Steiner points.

TetGen's mesh refinement algorithm [73] is a simple variant of Shewchuk's algorithm [68]. Instead of updating a conforming Delaunay mesh, it updates a constrained Delaunay mesh (CDT). A distinguished advantage of maintaining a CDT is that the sharp features are always presented and can be easily protected.

8.3 Constrained Delaunay Refinement

TetGen estimates for each vertex $\mathbf{p}$ in the initial CDT a real value p such that no Steiner point is allowed to be placed inside the ball $B(\mathbf{p}, p)$ which is centered at $\mathbf{p}$ with radius p. $B(\mathbf{p}, p)$ is called the *protecting ball* of the vertex $\mathbf{p}$. Default, only vertices on sharp features have protecting balls. Other vertices are not protected (by setting their radii be zero). This is sufficient for termination.

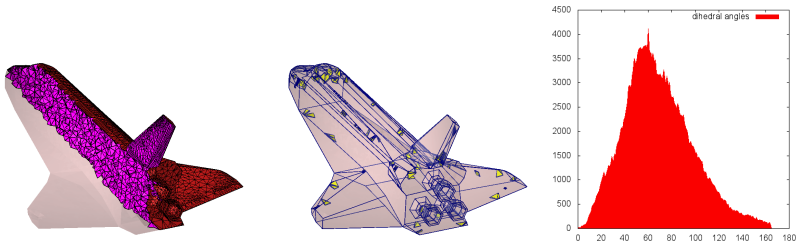
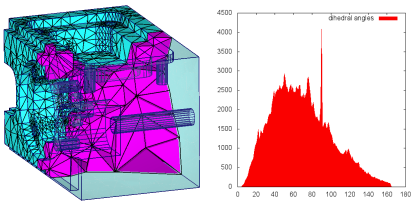
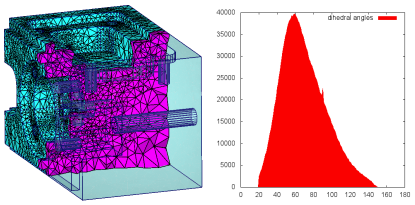


Figure 7: From left to right: A refined tetrahedral mesh (`m1249`), a highlight of the remaining bad quality tetrahedra, and the distribution of the dihedral angles of the mesh. In this example, min. dihedral angle = 0.27^o, max. dihedral angle = 168.7^o.



min. dihedral angle = 5^o
max. dihedral angle = 165^o
$8,322$ points, $35,584$ tetrahedra



min. dihedral angle = 20^o
max. dihedral angle = 154^o
$68,158$ points, $374,296$ tetrahedra

Figure 8: The distributions of dihedral angles in two refined tetrahedral meshes using different minimum dihedral angle measures.

Badly quality tetrahedra may exist on termination, they are all located around the sharp features, see an example shown in Figure 7.

The question is how to efficiently calculate the most appropriate radii for the vertices. Delaunay refinement uses implicitly the local feature sizes [62] as the mesh sizing function. They are the most appropriate values for the radii of protecting balls. However, to exactly calculate the local feature size is difficult and may be very slow. The initial CDT provides a fairly good structure for calculating an approximated local feature size at each vertex. It is only calculated locally and thus is very efficient. TetGen calculates the approximated local feature sizes at all vertices of the initial CDT. We thus obtain a discrete mesh sizing function G on the initial CDT.

Delaunay refinement only bounds the face angles but not the dihedral angles. Hence slivers are not removed. TetGen provides an option to impose a smallest dihedral angle bound. When it is specified, TetGen will remove those tetrahedra which have dihedral angles lower than this bound. The actual value of this bound is purely empirical. Since the algorithm always checks the face angle bound first, it is sufficient to use a small value (say 5^o) to distinguish slivers. In practice, we observed that Delaunay refinement algorithm may terminate on a smallest dihedral angle bound as larger as 20^o, see Figure 8 for examples.

TetGen takes an optional isotropic mesh sizing function H as input. When it is available, TetGen first unifies H and G, such that $H'(\mathbf{p}) := \min\{H(\mathbf{p}), G(\mathbf{p})\}$ for all vertices of the initial CDT (here we assume $H(\mathbf{p}) > 0$). This algorithm

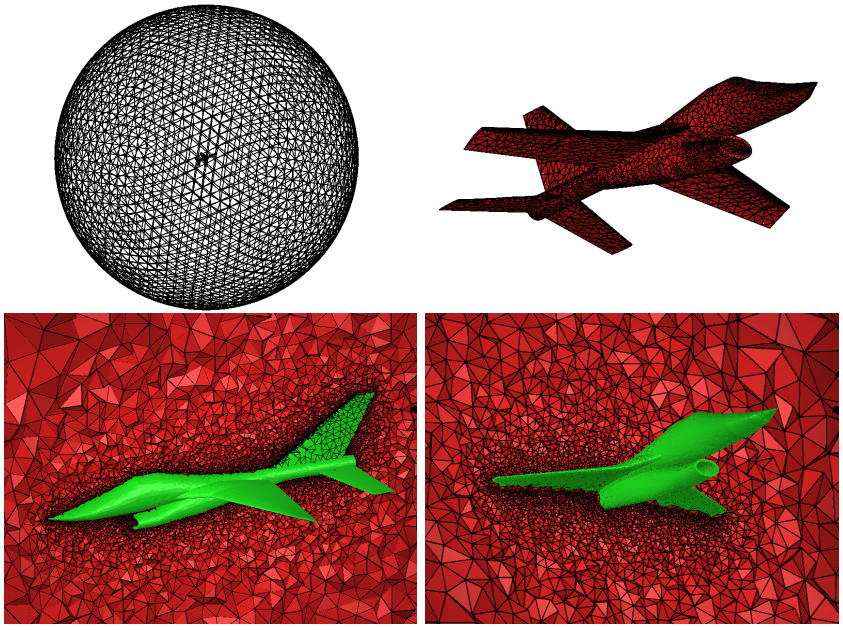


Figure 9: Adaptive mesh generation. The input (Top) is an aircraft `egads` (courtesy of Bob Haimes) placed inside a large sphere. The applied mesh sizing function is defined directly on the input vertices, such that a small mesh size (0.05) is given to the surface points of `egads`, and a big size (8.0) is given to the points on the sphere. Two views of the generated tetrahedral meshes (387, 511 points, 2, 429, 987 tetrahedra) are shown (Bottom).

then can generate an adaptive mesh whose mesh size is conformed to H'. Well conformity can be achieved for smoothed sizing functions, see Figure 9 for an example.

8.4 Mesh Improvement

It is necessary to further improve the mesh quality after the constrained Delaunay refinement. There are mainly two reasons: The first is the possible existence of slivers. The second is due to the use of protecting balls, bad-quality tetrahedra may not be removed if their circumcenters lie inside some protecting balls. An example is shown in the left of the Figure 10.

Mesh improvement (also referred as mesh optimization) is an important subject in mesh generation. There exists a whole branch of works on this topic. TetGen only focuses on the removal of tetrahedra which have the worst quality, such as slivers. It does not intend to improve the average quality of the mesh. For this purpose, only the local mesh operations are needed. It can then be done efficiently.

There are two local operations to improve the mesh quality, (1) topological transformation, i.e., face/edge flips and vertex insertion/deletion, and (2) vertex smoothing, i.e., relocating vertices without changing the mesh topology. Previous works have shown that the combination of these two operations is very effective to improve the mesh quality, see e.g. [30, 39].

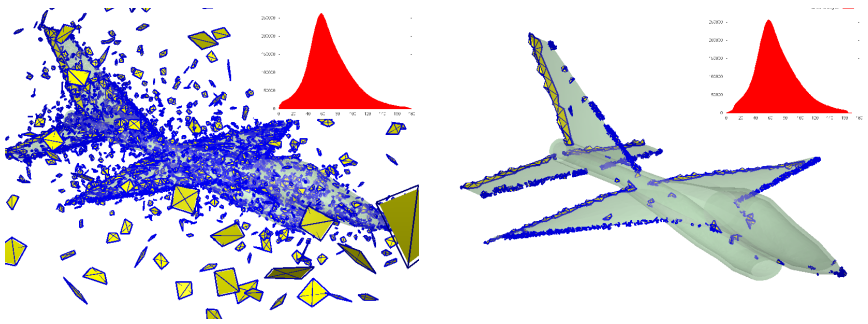


Figure 10: Left: Before the mesh improvement, a highlight of the bad quality tetrahedra of the mesh in Figure 9. Each plot tetrahedron has either its minimum dihedral angle $< 3^o$ or its maximum dihedral angle $> 179^o$. Right: after the mesh improvement, a highlight of remaining bad quality tetrahedra. They are all clustered near the sharp features of the input.

TetGen uses a simple "hill climbing" scheme to improve the mesh quality, i.e., a local operation is only performed if the resulting tetrahedra all have better quality than the worst quality of current tetrahedra.

TetGen initializes a list of bad quality tetrahedra whose qualities are less than a given objective value (currently the maximal dihedral angle is used). It then uses the local operations: edge/face flips and vertex smoothing to remove them. These operations are combined to iteratively to replace bad quality tetrahedra by improved ones. New low quality tetrahedra are added back into the list. This process stops either the list is empty or the maximum iteration number is reached.

Figure 10 illustrates an exmaple of TetGen's mesh improvement on the tetrahedral mesh shown in Figure 9. It is shown that most of the bad quality tetrahedra are successfully removed. The remaining bad quality tetrahedra are all clustered near the sharp features of the input.

9 Summary and Outlook

In this paper, the fundamental tetrahedral meshing problems treated by TetGen, i.e., local mesh transformations (flips), Delaunay tetrahedralizations, boundary recovery, and quality mesh refinement and improvement, are introduced, and the state-of-the-art algorithms and technologies used by TetGen are reported. Practice experiments show that TetGen behaves robustly and is able to efficiently generate isotropic tetrahedral meshes with high quality.

Tetrahedral mesh generation is an active ongoing research topic. It still faces many challenges in designing provable and efficient algorithms and in robust software implementation. Two fundamental issues which worth to be deeply investigated are the edge recovery and the vertex deletion. So far, no condition which can certify these operations is known. In the problem of CDT construction, a lower bound on the minimum number of Steiner points is not known yet. Our experiments shows that the number is almost linear with respect to the input vertices and segments. Can it be quadratic? Furthermore, what is the optimal number of Steiner points for constructing a CDT? In quality mesh

generation, a proof of a non-trivial minimum or maximum dihedral angle bound for Delaunay refinement is still missing. Progress in any of these problems may greatly improve the performance of TetGen.

We firmly believe that understanding of the fundamental mathematical problems will lead to a fruitful contributions in algorithms and applications. On the other hand, engineering practice is equally important. It provides opportunities to validate algorithms and to discover issues not yet covered by our current knowledge. The future development of TetGen will persistently follow these two guidelines.

Acknowledgments

The author wish to thank WIAS for the long-term support of the research and development of TetGen. In particular, thanks to Jürgen Fuhrmann, Klaus Gärtner, Eberhard Bänsch for their perspective in the subject of numerical mesh generation and their numerous help in developing TetGen. Thanks to Volker John for his support during the writing of this article.

References

[1] N. Amenta, S. Choi, and G. Rote, *Incremental construction con BRIO*, in Proceedings of the 19th ACM Symposium on Computational Geometry, 2003, pp. 211–219.

[2] T. Apel, *Anisotropic Finite Elements: Local Estimates and Applications*, Teubner, Stuttgart, 1999.

[3] I. Babuška and A. K. Aziz, *On the angle condition in the finite element method*, SIAM J. Numer. Anal., 13 (1976), pp. 214–226.

[4] T. J. Baker, *Automatic mesh generation for complex three-dimensional regions using a constrained Delaunay triangulation*, Engineering with Computers, 5 (1989), pp. 161–175.

[5] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa, *The Quickhull algorithm for convex hulls*, ACM Trans. Math. Software, 22 (1996), pp. 469–483.

[6] M. W. Bern and D. Eppstein, *Mesh generation and optimal triangulation*, in Computing in Euclidean Geometry, D.-Z. Du and F. K.-M. Hwang, eds., no. 4 in Lecture Notes Series on Computing, World Scientific, second ed., 1995, pp. 47–123.

[7] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow, *Compact representations of simplical meshes in 2 and 3 dimensions*, Internat. J. Comput. Geom. Appl., 15 (2005), pp. 3–24.

[8] J.-D. Boissonnat, O. Devillers, and S. Hornus, *Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension*, in Proceedings of the 25th Annual Symposium on Computational Geometry, 2009.

[9] H. Borouchaki, P. L. George, and S. H. Lo, *Optimal Delaunay point insertion*, Internat. J. Numer. Methods Engrg., 39 (1996), pp. 3407–3437.

[10] A. Bowyer, *Computing Dirichlet tessellations*, Comp. Journal, 24 (1987), pp. 162–166.

[11] H. Broennimann, C. Burnikel, and S. Pion, *Interval arithmetic yields efficient dynamic filters for computational geometry*, in Proceedings of the 14th Annual Symposium on Computational Geometry, 1998, pp. 165–174.

[12] CGAL, *User and Reference Manual, release 4.1*, October 2012.

[13] B. Chazelle, *Convex partition of polyhedra: a lower bound and worst-case optimal algorithm*, SIAM J. Comput., 13 (1984), pp. 488–507.

[14] B. Chazelle and L. Palios, *Triangulating a non-convex polytope*, Discrete Comput. Geom., 5 (1990), pp. 505–526.

[15] L. Chen and J.-C. Xu, *Optimal Delaunay triangulations*, J. Comput. Math., 22 (2004), pp. 299–308.

[16] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng, *Sliver exudation*, J. Assoc. Comput. Mach., 47 (2000), pp. 883–904.

[17] S.-W. Cheng, T. K. Dey, E. A. Ramos, and T. Ray, *Quality meshing for polyhedra with small angles*, Internat. J. Comput. Geom. Appl., 15 (2005), pp. 421–461.

[18] L. P. Chew, *Constrained Delaunay triangulations*, Algorithmica, 4 (1989), pp. 97–108.

[19] ——, *Guaranteed-quality triangular meshes*, Tech. Report TR 89-983, Dept. of Comp. Sci., Cornell University, 1989.

[20] K. L. Clarkson and P. W. Shor, *Applications of random sampling in computational geometry, II*, Discrete Comput. Geom., 4 (1989), pp. 387–421.

[21] D. Cohen-Steiner, É. C. De Verdière, and M. Yvinec, *Conforming Delaunay triangulation in 3D*, in Proceedings of the 18th Annual Symposium on Computational Geometry, 2002.

[22] B. N. Delaunay, *Sur la sphère vide*, Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk, 7 (1934), pp. 793–800.

[23] O. Devillers and S. Pion, *Efficient exact geometric predicates for Delaunay triangulations*, in 5th Workshop on Algorithm Engineering and Experiments, 2003, pp. 37–44.

[24] O. Devillers, S. Pion, and M. Teillaud, *Walking in triangulation*, Internat. J. Found. Comput. Sci., 13 (2002), pp. 181–199. INRIA Tec. Report No. 4120, 2001.

[25] D. P. Dobkin and M. J. Laszlo, *Primitives for the manipulation of three-dimensional subdivisions*, Algorithmica, 4 (1989), pp. 3–32.

[26] H. Edelsbrunner, *Geometry and topology for mesh generation*, Cambridge University Press, Cambridge, England, 2001.

[27] H. Edelsbrunner and E. P. Mücke, *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithm*, ACM Transactions on Graphics, 9 (1990), pp. 66–104.

[28] H. Edelsbrunner and N. R. Shah, *Incremental topological flipping works for regular triangulations*, Algorithmica, 15 (1996), pp. 223–241.

[29] S. Fortune and C. J. Van Wyk, *Static analysis yield efficient exact integer arithmetic for computational geometry*, ACM Transactions on Graphics, 15 (1996), pp. 223–248.

[30] L. A. Freitag and C. Ollivier-Gooch, *Tetrahedral mesh improvement using swapping and smoothing*, Internat. J. Numer. Methods Engrg., 40 (1997), pp. 3979–4002.

[31] P. J. Frey and P. L. George, *Mesh Generation - Application to Finite Elements*, Hermes Science Publishing, Oxford, UK, 1st ed., 2000. ISBN 1-903398-00-2.

[32] R. V. Garimella, *Mesh data structure selection for mesh generation and FEA applications*, Internat. J. Numer. Methods Engrg., 55 (2002), pp. 451–478.

[33] P. L. George and H. Borouchaki, *Back to edge flips in* 3 *dimensions*, in Proceedings of the 12th International Meshing Roundtable, 2003.

[34] P. L. George, H. Borouchaki, and E. Saltel, *Ultimate robustness in meshing an arbitrary polyhedron*, Internat. J. Numer. Methods Engrg., 58 (2003), pp. 1061–1089.

[35] P. L. George, F. Hecht, and E. Saltel, *Automatic mesh generator with specified boundary*, Comput. Methods Appl. Mech. Engrg., 92 (1991), pp. 269–288.

[36] L. Guibas and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Transactions on Graphics, 4 (1985), pp. 75–123.

[37] W. Huang, *Measuring mesh qualities and application to variational mesh adaption*, SIAM J. Sci. Comput., 26 (2005), pp. 1643–1666.

[38] B. Joe, *Construction of three-dimensional improved-quality triangulations using local transformations*, SIAM J. Sci. Comput., 16 (1995), pp. 1292–1307.

[39] B. M. Klinger and J. R. Shewchuk, *Aggressive tetrahedral mesh improvement*, in Proceedings of the 16th International Meshing Roundtable, 2007, pp. 3–23.

[40] M. Křížek, *On the maximum angle condition for linear tetrahedral elements*, SIAM J. Numer. Anal., 29 (1992), pp. 513–520.

[41] C. L. Lawson, *Software for c^1 surface interpolation*, Mathematical Software III, Academic Press, (1977), pp. 164–191.

[42] D. T. Lee and A. K. Lin, *Generalized Delaunay triangulations for planar graphs*, Discrete Comput. Geom., 1 (1986), pp. 201–217.

[43] X.-Y. Li and S.-H. Teng, *Generating well-shaped Delaunay meshes in 3D*, in Proc. 12th ann. ACM-SIAM Symp. on Disc. Algo., 2001, pp. 28–37.

[44] A. Liu and M. Baida, *How far flipping can go towards 3D conforming/constrained triangulation*, in Proceedings of the 9th International Meshing Roundtable, 2000, pp. 307–315.

[45] Y. Liu and J. Snoeyink, *A comparsion of five implementations of 3D Delaunay tessellation*, in Combinatorial and Computational Geometry, J. E. Goodman, J. Pach, and E. Welzl, eds., vol. 52, MSRI publications, 2005, pp. 439–458.

[46] S. H. Lo, *Volume discretization into tetrahedra - II. 3D triangulation by advancing front approach*, Computers & Structures, 39 (1991), pp. 501–511.

[47] R. Löhner and P. Parikh, *Three-dimensional grid generation by the advancing-front method*, Internat. J. Numer. Methods Fluids, 8 (1988), pp. 1135–1149.

[48] D. L. Marcum and N. P. Weatherill, *Unstructured grid generation using iterative point insertion and local reconnection*, AIAA Journal, 33 (1995), pp. 1619–1625.

[49] P. McMullen, *The maximum number of faces of a convex polytope*, Mathematika, 17 (1970), pp. 179–184.

[50] G. Miller, D. Talmor, S.-H. Teng, and N. Walkington, *On the radius-edge condition in the control volume method*, SIAM J. Numen. Anal., 36 (1999), pp. 1690–1708.

[51] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang, *Control volume meshes using sphere packing: Generation, refinement and coarsening*, in Proceedings of the 5th International Meshing Roundtable, 1996, pp. 47–61.

[52] S. A. Mitchell and S. A. Vavasis, *Quality mesh generation in higher dimensions*, SIAM J. Comput., 29 (2000), pp. 1334–1370.

[53] E. P. Mücke, *Shapes and Implementations in Three-Dimensions Geometry*, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1993.

[54] M. Murphy, D. M. Mount, and C. W. Gable, *A point-placement strategy for conforming Delaunay tetrahedralizations*, in Proceedings of the 11th annual ACM-SIAM Symposium on Discrete Algorithms, 2000, pp. 69–93.

[55] S. E. Pav and N. Walkington, *Robust three dimensional Delaunay refinement*, in Proceedings of the 13th International Meshing Roundtable, 2004.

[56] D. M. Priest, *Algorithms for arbitrary precision floating point arithmetic*, in 10th Symposium on Computer Arithmetic, 1991, pp. 132–143.

[57] J. Radon, *Mengen konvexer Körper, die einen gemeinschaftlichen Punkt enthalten*, Math. Ann., 83 (1921), pp. 113–115.

[58] J. Rambau, *On a generalization of Schönhardt's polyhedron*, in Combinatorial and Computational Geometry, J. E. Goodman, J. Pach, and E. Welzl, eds., vol. 52, MSRI publications, 2005, pp. 501–516.

[59] A. Rand and N. Walkington, *Collars and intestines: practical conforming Delaunay refinement*, in Proceedings of the 18th International Meshing Roundtable, 2009, pp. 481–497.

[60] S. Rippa, *Long and thin triangles can be good for linear interpolation*, SIAM J. Numer. Anal., 29 (1992), pp. 257–270.

[61] M.-C. Rivara, *New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations*, Internat. J. Numer. Methods Engrg., 40 (1997), pp. 3313–3324.

[62] J. Ruppert, *A Delaunay refinement algorithm for quality 2-dimensional mesh generation*, Journal of Algorithms, 18 (1995), pp. 548–585.

[63] J. Ruppert and R. Seidel, *On the difficulty of triangulating three-dimensional nonconvex polyhedra*, Discrete Comput. Geom., 7 (1992), pp. 227–253.

[64] E. Schönhardt, *Über die zerlegung von dreieckspolyedern in tetraeder*, Math. Ann., 98 (1928), pp. 309–312.

[65] J. R. Shewchuk, *Robust adaptive floating-point geometric predicates*, in Proceedings of the 12th Annual Symposium on Computational Geometry, 1996.

[66] ——, `Triangle`*: Engineering a 2D quality mesh generator and Delaunay triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., vol. 1148 of Lect. Notes in Comput. Sci., Springer, 1996, pp. 203–222. `http://www.cs.cmu.edu/ quake/triangle.html`.

[67] ——, *A condition guaranteeing the existence of higher-dimensional constrained Delaunay triangulations*, in Proceedings of the 14th Annual Symposium on Computational Geometry, 1998, pp. 76–85.

[68] ——, *Tetrahedral mesh generation by Delaunay refinement*, in Proceedings of the 14th Annual Symposium on Computational Geometry, 1998, pp. 86–95.

[69] ——, *Constrained Delaunay tetrahedralizations and provably good boundary recovery*, in Proceedings of the 11th International Meshing Roundtable, 2002, pp. 193–204.

[70] ——, *What is a good linear element? interpolation, conditioning, and quality measures*, in Proceedings of 11th International Meshing Roundtable, 2002, pp. 115–126.

[71] ——, *Updating and constructing constrained Delaunay and constrained regular triangulations by flips*, in Proceedings of the 19th Annual Symposium on Computational Geometry, 2003, pp. 86–95.

[72] ——, *General-dimensional constrained Delaunay and constrained regular triangulations, I: combinatorial properties*, Discrete Comput. Geom., 39 (2008), pp. 580–637.

[73] H. Si, *Adaptive tetrahedral mesh generation by constrained delaunay refinement*, Internat. J. Numer. Methods Engrg., 75 (2008), pp. 856–880.

[74] H. Si, J. Fuhrmann, and K. Gärtner, *Boundary conforming Delaunay mesh generation*, Comput. Math. Math. Phys., 50 (2010), pp. 38–53.

[75] H. Si and K. Gärtner, *Meshing piecewise linear complexes by constrained Delaunay tetrahedralizations*, in Proceedings of the 14th International Meshing Roundtable, 2005, pp. 147–163.

[76] ——, *3D boundary recovery by constrained Delaunay tetrahedralization*, Internat. J. Numer. Methods Engrg., 85 (2011), pp. 1341–1364.

[77] H. Si and J. R. Shewchuk, *Incrementally constructing and updating constrained Delaunay tetrahedralizations with finite precision coordinates*, in Proceedings of the 21th International Meshing Roundtable, 2012.

[78] J. F. Thompson, B. K. Soni, and N. P. Weatherill, eds., *Handbook of Grid Generation*, CRC Press, 1998.

[79] D. F. Watson, *Computing the n-dimensional Delaunay tessellations with application to Voronoi polytopes*, Comput. Journal, 24 (1987), pp. 167–172.

[80] N. P. Weatherill and O. Hassan, *Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints*, Internat. J. Numer. Methods Engrg., 37 (1994), pp. 2005–2039.

[81] C.-K. Yap, *Towards exact geometric computation*, Comput. Geom., 7 (1997), pp. 3–23.

[82] M. A. Yerry and M. S. Shephard, *Automatic 3D mesh generation by the modified-octree technique*, Internat. J. Numer. Methods Engrg., 20 (1984), pp. 1965–1990.

[83] G. M. Ziegler, *Lectures on Polytopes*, vol. 152 of Graduate Texts in Mathematics, Springer-Verlag, New York, second edition ed., 1997.

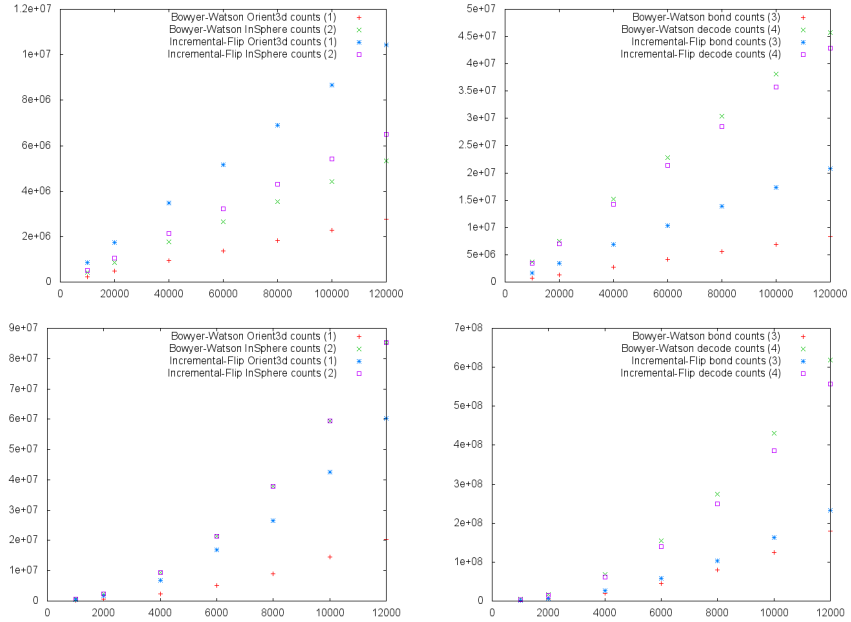


Figure 11: Bowyer-Watson vs Incremental Flip. Counts of the four operations performed on the `random` data sets (Top) from sizes 10,000 to 120,000 and on the `line-and-circle` data sets (Bottom) from sizes 1,000 to 12,000.

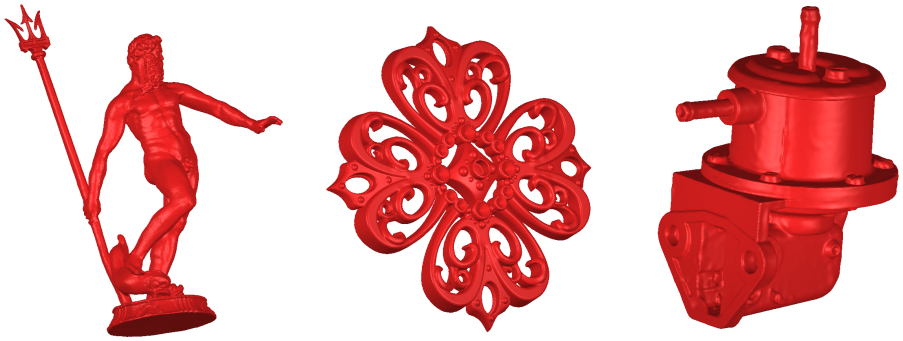
10 Experimental Results

In this section, some experiments regarding the behavior and efficiency of the implemented algorithms in TetGen are reported. All experiments have been performed on a 2.2 GHz Intel Cire i7 with 8 GB of 1333 MHz DDR3 memory (MacOSX 10.7.5). TetGen version 1.5 (pre-lease version, 2011) was used. It was compiled using `GCC/G++` version 4.2.1 with optimization level `-O3`.

10.1 Bowyer-Watson vs Incremental Flip

Although these two algorithms have the same optimal complexity, it is observed that the Bowyer-Watson algorithm behaves more efficiently in practice. It is obvious that the incremental flip algorithm may create and delete some temporary tetrahedra. While the Bowyer-Watson algorithm needs an extra process to search the tetrahedra in order to form the cavity. We compared these two algorithms by counting four key operations they have performed, which are (1) the `orient3d` test, (2) the `insphere` test, (3) the `encode` primitive, and (4) the `decode` primitive. Where an `encode` connects two adjacent tetrahedra, and a `decode` performs a neighbor query. The results are shown in Figure. 11.

Two types of data sets are used: `random`, points are randomly distributed inside a unit cube; and `line-and-circle`, points are evenly distributed on a line and a circle where the line passes through the center of the circle. The Delaunay tetrahedralization of a `line-and-circle` has a quadratic complexity. Our comparisons (in Figure 11) showed that the Incremental flip algorithm



`Neptune` `Filigree` `Oilpump`

Figure 12: Point sets used for experiments. Available from `http://shapes.aim-at-shape.net`

performs `orient3d` and `encode` about 2 to 3 times more than the Bowyer-Watson algorithm does. This is due to the amount of temporary tetrahedra created by the incremental flip algorithm. The numbers of `insphere` tests are similar in the two algorithms. While the Bowyer-Watson algorithm performs slightly more neighbor queries than the incremental algorithm due to the need of searching tetrahedra to form the cavity.

10.2 Comparison with Other Delaunay Codes

To experiment the efficiency of our implementation, we compared TetGen with two public Delaunay codes: `qhull` and `CGAL`.

`qhull` is a `C` program to compute convex hulls of point sets in general dimensions using the `Quickhull` algorithm [5]. It computes Delaunay tetrahedralizations through 4-dimensional convex hulls. We used the 2003 version of `qhull`. It is compiled using `GCC` version 4.2.1 with the default optimization level `-O2 -fPIC -ansi`.

`CGAL` is a `C++` geometric algorithm library. It includes an efficient implementation of the Bowyer-Watson algorithm for incremental construction of Delaunay tetrahedralizations [8]. It uses spatial sorting for preprocessing the points, and it implements its own filtered predicates which is more efficient than Shewchuk's predicates. We tested CGAL version 4.1, released 2011. The libraries of CGAL was compiled using the default `cmake` options (`CMAKE_BUILD_TYPE=Release`). We used the example `delaunay_3.cpp` provided by CGAL. It uses the kernel

`CGAL/Exact_predicates_inexact_constructions_kernel.h`.

It is compiled using the options `-O3 -DCGAL_NDEBUG`.

Five data sets are used for comparison, `500k` and `2m` are two randomly distributed point sets in the unit cube, `Neptune`, `Filigree`, and `Oilpump` are three data sets obtained from the AIM@SHAPE Repository, see Figure 12.

Table 1 reports the running times of these programs. `TetGen` (using the Bowyer-Watson algorithm) ran at lease as efficient as `CGAL`. Both ran fast than `qhull`.

Table 1: Comparing `TetGen` with `qhull` and `CGAL`.

	500k	2m	N	F	O
# points	500,000	2,000,000	499,417	514,302	570,018
# tetrahedra	3,371,591	13,503,890	3,445,298	3,540,956	4,052,388
`qhull` (2003)	20.54	107.50	12.60	20.31	25.49
`CGAL` (4.1)	3.54	14.46	3.83	14.79	4.39
`TetGen` (1.5)	3.39	13.91	3.81	4.86	4.54

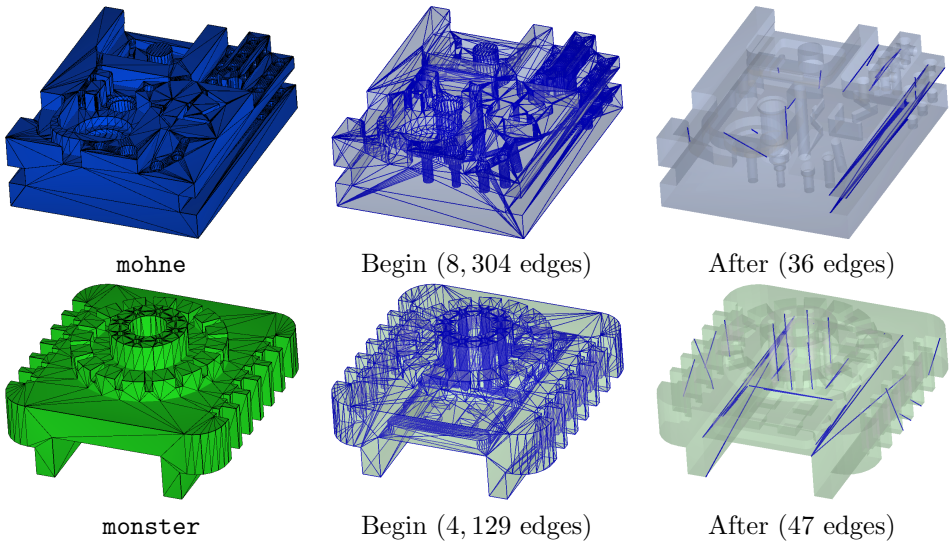


Figure 13: Two examples of the edge recovery algorithm.

10.3 Experiments with Boundary Recovery

Recovering constraints (edges and triangles) in tetrahedralization is still a research problem. There are many questions to be investigated. The use of the edge removal algorithm (developed in Section **??**) with an increasing 'level' has been shown very effective in recovering boundaries in practice. Table 2 further reports the experiments of the edge recovery algorithm on various inputs.

The inputs: `cami1a` (in Figure 1), `mohne`, and `monster` (in Figure 13) are freely available from INRIA's Mesh Repository. These experiments showed that the majority of the edges can be recovered after the search of first level ('level' = 1). As the search level increases, more and more edges have been recovered. Only few edges remain unrecovered after the full search ('level' = ∞).

Remark: Since we used a relatively strong condition (i.e., no creation of new intersecting faces of the removing edge) in the current edge recovery algorithm, it is not clear whether or not these remaining edges can be recovered by flips. Further investigations are to be done in the future.

Table 2: Experiments with the edge recovery algorithm. The inputs are freely available, `cami1a` (in Figure 1), `mohne`, and `monster` are from INRIA's GAMMA group (`http://www-roc.inria.fr/gamma/gamma/gamma.php`), `Neptune` and `Filigree` are from the AIM@SHAPE repository (`http://shapes.aim-at-shape.net`).

	`cami1a`	`anc101`	`monster`	`mohne`	`Neptune`	`Filigree`
# points	460	1,378	1,392	2,760	499,416	514,300
# triangles	884	2,772	2,784	5,560	998,840	1,028,856
# edges	1,349	4,158	4,176	8,340	1,498,260	1,543,284
'level' = 1	90	273	299	266	123	138
'level' = 2	40	139	135	134	33	23
'level' = 3	34	67	98	102	17	8
...	...	...	...	...	...	...
'level' = ∞	13	21	49	36	10	6