TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator

HANG SI, Weierstrass Institute for Applied Analysis and Stochastics (WIAS)

TetGen is a C++ program for generating good quality tetrahedral meshes aimed to support numerical methods and scientific computing. The problem of quality tetrahedral mesh generation is challenged by many theoretical and practical issues. TetGen uses Delaunay-based algorithms which have theoretical guarantee of correctness. It can robustly handle arbitrary complex 3D geometries and is fast in practice. The source code of TetGen is freely available.

This article presents the essential algorithms and techniques used to develop TetGen. The intended audience are researchers or developers in mesh generation or other related areas. It describes the key software components of TetGen, including an efficient tetrahedral mesh data structure, a set of enhanced local mesh operations (combination of flips and edge removal), and filtered exact geometric predicates. The essential algorithms include incremental Delaunay algorithms for inserting vertices, constrained Delaunay algorithms for inserting constraints (edges and triangles), a new edge recovery algorithm for recovering constraints, and a new constrained Delaunay refinement algorithm for adaptive quality tetrahedral mesh generation. Experimental examples as well as comparisons with other softwares are presented.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *Algorithm design and analysis, efficiency, reliability and robustness*; I.3.5 [**Computational Geometry and Object Modeling**]

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Tetrahedral mesh generation, Delaunay, constrained Delaunay, Steiner points, flips, edge removal, boundary recovery, mesh quality, mesh refinement, mesh improvement

ACM Reference Format:
Hang Si. 2015. TetGen, a Delaunay-based quality tetrahedral mesh generator. ACM Trans. Math. Softw. 41, 2, Article 11 (January 2015), 36 pages.
DOI: http://dx.doi.org/10.1145/2629697

1. INTRODUCTION

A tetrahedral mesh is a 3D unstructured grid that partitions a 3D domain. This type of partitions has many favorable properties. For example, it well fits domains with arbitrarily complicated geometry, it can be easily locally refined and coarsened (with no hanging nodes), and it can be created fully automatically.

The development of TetGen is mainly motivated by the numerical solution of partial differential equations (PDEs), such as the finite element and finite volume methods. The quality of the mesh will tremendously affect the accuracy and convergence of the numerical solution. To generate a tetrahedral mesh having the desired mesh quality is a complicated problem, both in theory and practice.

Technologies for mesh generation have been greatly advanced in recent three decades. Early tetrahedral mesh generation methods are comprehensively surveyed

This work is supported by Weierstrass Institute for Applied Analysis and Stochastics (WIAS), Leibniz Institute in Forschungsverbund Berlin e.V.
Author's address: Research Group 3: Numerical Mathematics and Scientific Computing, Weierstrass Institute for Applied Analysis and Stochastics (WIAS), Mohrenstrasse 39, 10117, Berlin, Germany; email: si@wias-berlin.de.

DOI: http://dx.doi.org/10.1145/2629697

in the books [Thompson et al. 1999; Frey and George 2000]. Some of these methods have been successfully transferred into softwares. One of commercial softwares, `Tetmesh-GHS3D` [TetMesh-GHS3D 2010], has been integrated into many finite element softwares. Some open sources, like `NETGEN` [Schöberl 1997] and `GRUMMP` [Ollivier-Gooch 2005], are popularly used in research and academic areas. However, a major problem in most of the early methods is the lack of theoretical background. As the complexity of the 3D objects increases, these methods may either not be suitable or tend to fail often. In many engineering applications, in which a meshing software is used as a component (library) and will be called frequently, both of the robustness and efficiency of the meshing softwares are crucial.

The problem of tetrahedral mesh generation has been attracted theoretical studies since mid 1990s. Methods with certain theoretical guarantees on a successful termination and good mesh quality were developed, see Bern and Eppstein [1995] and Edelsbrunner [2001]. Some of these methods have been implemented in softwares, like `QualMesh` [Cheng et al. 2005] and `CGALmesh` [Jamin et al. 2013]. However, there are still many unsolved issues. These methods cannot handle arbitrary 3D objects, and their efficiency do not yet meet the requirements of engineering applications. Now it has been commonly acknowledged, the construction of robust and efficient mesh generation methods can only be achieved through algorithms which have solid theoretical justification and good practical heuristics. It is the principle that TetGen follows.

Ruppert [1995] proposed a Delaunay-based algorithm, extending an algorithm of Chew [1989b], provably generates an optimal Delaunay triangular mesh of a 2D polygonal domain with a bounded smallest angle. Shewchuk [1996b] provided a robust and efficient implementation of this algorithm in the freely available program `Triangle`. 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, Ruppert's algorithm in 3D has many limitations. The initial versions of TetGen (1.0 and 1.1) were neither reliable nor efficient.

Since 2002, TetGen has been supported by WIAS and has been continuously developing to date. The goal of developing TetGen is two-fold: First, it is a research project aimed to investigate the underlying mathematical problems and to develop innovative algorithms for mesh generation; Second, it provides a robust, efficient, and easy-to-use software to generate quality tetrahedral meshes for various engineering applications. TetGen is freely available through `http://www.tetgen.org`.

The way of combining research and applications has been proven very meaningful. During this period, the underlying meshing problems were better understood, and new algorithms with theoretical background were developed [Si 2008]. A milestone of TetGen is version 1.4.3 (Sept. 2009). The main algorithms in this version are theoretically correct and practically efficient [Si 2010]. It is able to handle arbitrary geometry and topology. Good mesh quality is guaranteed for certain class of geometries. The robustness of TetGen is very much improved. The current version, 1.5.0 (Nov. 2013), is a major improvement of version 1.4.3. It developed a more efficient mesh data structure and powerful local mesh operations. It developed good heuristic algorithms to improve mesh quality and to better support engineering applications.

However, the current version still has several limitations. One of the main limitations is that it only handles piecewise linear boundaries and constraints. Another issue is that the quality of the meshes produced by TetGen may not be as good as other softwares. It will be improved in the future.

This article focuses on the techniques and algorithms that used to develop the version 1.5.0 of TetGen. It is assumed that most of the readers of this article are meshing researchers or meshing software developers. The user's manual of TetGen [Si 2013] provides a detailed documentation about the usage of the program.

Outline. The rest of the article is organized as follows. After a brief review of background and related work (Section 2), a top-level workflow and an overview of the meshing problems dealt by TetGen and the algorithms used by TetGen are given (Section 3). The technical details of TetGen are presented from Section 4 to Section 9. They are arranged according to different topics in tetrahedral mesh generation. Each topic is presented as a self-contained section. It could be used to understand just that part of TetGen. References are provided therein for further understanding the details. The sections are outlined here to help the readers navigate and find information quickly. As an illustration of how these different ingredients work well together, we present experiment meshing results as well as comparisons with other softwares and methods in Section 10. Finally, we summarize a list of open issues and future works.

—Section 2 Related work
—Section 3 The Methodology of TetGen
—Section 4 Tetrahedral mesh data structure
—Section 5 Local mesh transformations
—Section 6 Filtered exact geometric predicates
—Section 7 Delaunay tetrahedralizations
—Section 8 Constrained tetrahedral mesh generation
—Section 9 Quality tetrahedral mesh generation
—Section 10 Experimental Results
—Section 11 Summary and Outlook

2. RELATED WORK

The methods established during the 1980s and 1990s can be roughly classified into three groups: octree-based, advancing-front, and Delaunay-based, see Thompson et al. [1999], Frey and George [2000], and Owen [1998] for comprehensive surveys of these methods. Only Delaunay-based methods are able to treat objects of essentially arbitrary complexity. Moreover, the structural properties of Delaunay triangulation offer some mathematically provable guarantees on mesh quality. Therefore, Delaunay-based methods are predominate and actively developed.

Delaunay-based tetrahedral mesh generation methods can be further characterized by the ways of handling domain boundaries and improving mesh quality. Keeping this classification in mind, we briefly review three groups of methods.

Boundary Constrained. A classical group of methods aims at preserving the input boundary mesh exactly [Baker 1989; George et al. 1991; Weatherill and Hassan 1994]. This is a common requirement by many engineering applications. For example, the input boundary mesh may be used to model special geometries, such as edge or corner singularities, or parameterized surfaces, or to apply numerical boundary conditions. Since these methods do not need to modify surface meshes, the mesh operations like flips and vertex insertion are simplified. Efficient implementations of these methods are available [Borouchaki et al. 1996; TetMesh-GHS3D 2010].

However, to preserve input domain boundary mesh is still a theoretically difficult problem. Most of the methods involved heuristic and tedious mesh modification techniques without guarantee of success. In practice, this task is the main cause of failure in these methods. A major limitation of these methods is that the mesh quality depends on the quality of the input surface mesh.

Delaunay Refinement. The simple idea of Delaunay refinement leads to a group of methods that aim at generating meshes with a theoretical guarantee on the output mesh quality [Chew 1989b; Ruppert 1995; Shewchuk 1998b]. For this purpose, the input boundary mesh needs to be modified. Different ways are proposed. One way is

to refine (subdivide) the edges and triangles in the original boundary mesh. This leads to conforming Delaunay tetrahedralizations [Murphy et al. 2000; Cohen-Steiner et al. 2004] or constrained Delaunay tetrahedralizations [Shewchuk 2002a; Si 2010]. This way is simple and can be implemented robustly. However, this way cannot remove some small angles in the input surface mesh, hence, the tetrahedral mesh quality is still restricted by the input surface mesh.

Another way is to re-mesh the domain boundary [Oudot et al. 2005; Cheng et al. 2007] such that a new boundary mesh is obtained. This leads to a restricted Delaunay triangulation [Edelsbrunner and Shah 1997] of the surface. By this way, the surface mesh and tetrahedral mesh can be generated simultaneously. It has several advantages – it does not depend on the quality of the input surface mesh; it may lead to a better approximation of the domain boundary and may result a better quality tetrahedral mesh. A key component of this approach is to efficiently identify triangles that belong to the restricted Delaunay triangulation. This requires to quickly find the intersection point of a ray and the input surface mesh. It is nontrivial for arbitrary complicated geometries.

However, the nice theoretical guarantees of Delaunay refinement are valid only for domains with no small (dihedral) angles. A main issue in Delaunay refinement methods is how to efficiently discretize domain boundary in order to preserve small angles. A major limitation of these methods in engineering applications is that they do not preserve the original input boundary mesh.

Variational Methods. A group of methods, so-called variational Voronoi-Delaunay methods [Alliez et al. 2005; Du and Wang 2003] aims at generating and improving good quality tetrahedral mesh simultaneously. The idea is to use a convex mesh-dependent energy so that a global (or local) minimum exists. Two such functionals are proposed in Du and Wang [2003] and Chen and Xu [2004]. The minimum of these two functionals are two well-known structures: a centroidal Voronoi tessellation (CVT) [Du et al. 1999] and a weighted Delaunay triangulation [Edelsbrunner 2001], respectively. Variational Voronoi-Delaunay methods tend to produce a very regularly distributed vertex set, which will lead to a high-quality tetrahedral mesh.

However, variational Voronoi-Delaunay methods suffer the same issues of Delaunay refinement methods. They do not preserve input boundary meshes. Since the energy functionals are evaluated over the whole mesh, they are much slower than boundary constrained and Delaunay refinement methods.

From this brief review, we have seen that the problem of generating quality tetrahedral mesh is very challenging. This is especially manifested in the difficulties of handling domain boundaries and generating good quality tetrahedral meshes at the same time. A practical mesh generator should be able to robustly handle arbitrary inputs, preserve the domain boundary mesh when it is needed, and efficiently generate good quality tetrahedral meshes even on low-quality input surface meshes.

3. THE METHODOLOGY OF TETGEN

The methodology of TetGen is a mixture of a classical boundary constrained methods [George et al. 1991] and a classical Delaunay refinement method [Ruppert 1995; Shewchuk 1998b]. This approach well combines engineering and theoretical ideas. It is efficient in generating a good quality tetrahedral mesh.

3.1. Piecewise Linear Complexes

TetGen uses a simple model of a 3D domain that may contain internal boundaries, that is, it may be a nonmanifold. A 3D *piecewise linear complex* (PLC) $\mathcal{X}$, first introduced by Miller et al. [1996], is a set of vertices, edges, polygons, and polyhedra, collectively

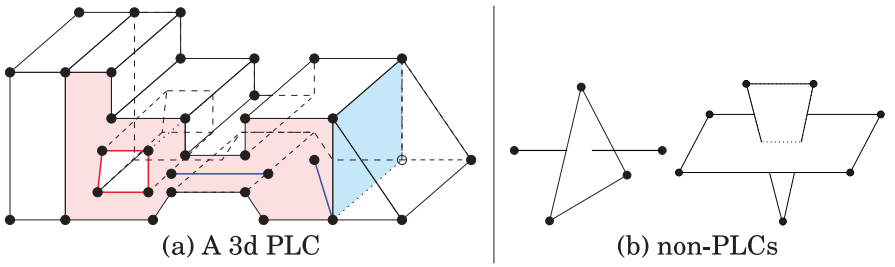


Fig. 1. (a) A 3D piecewise linear complex. The left shaded area shows a polygon, which is nonconvex and has a hole in it. It has also edges and vertices floating in it. The right shaded area shows an interior polygon separating two sub-domains. (b) Two configurations that are not PLCs.

called *cells*, that satisfies the following properties: (i) the boundary of each cell in $\mathcal{X}$ is a union of cells in $\mathcal{X}$; and (ii) 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 3D PLC. Some configurations that fail to satisfy the condition (ii) are shown in Figure 1(b).

The *underlying space* of a 3D PLC $\mathcal{X}$, denoted $|\mathcal{X}|$, is the union of all cells of $\mathcal{X}$. It is a topological space whose topology is given by the set $\mathcal{X}$ [Si 2008]. $|\mathcal{X}|$ is the *mesh domain* of TetGen. The *boundary complex* of $\mathcal{X}$ is the subset of cells of $\mathcal{X}$ whose dimensions are less than 3. It is a 2D PLC. The *boundary* of the mesh domain ($|\mathcal{X}|$) is the underlying space of the boundary complex of $\mathcal{X}$.

Tetrahedral Meshes of PLCs. A *tetrahedralization* $\mathcal{T}$ is a 3D simplicial complex. It decomposes its underlying space $|\mathcal{T}| \subseteq \mathbb{R}^3$ [Edelsbrunner 2001]. We say that $\mathcal{T}$ is a *tetrahedral mesh* of a 3D PLC $\mathcal{X}$ if: (i) every cell of $\mathcal{X}$ is represented by a union of simplices of $\mathcal{T}$, and (ii) $|\mathcal{X}| = |\mathcal{T}|$. Note that this definition implies that the vertex set of $\mathcal{X}$ is a subset of the vertex set of a tetrahedralization of $\mathcal{X}$. The extra vertices are called *Steiner points*. As we will see in the later sections, there will be various reasons to have Steiner points. We call edges and triangles in $\mathcal{T}$ *segments* and *subfaces*, respectively, if they are included in edges or polygons of the input PLC.

3.2. The Meshing Pipeline

Figure 2 shows the pipeline of the mesh generation process in TetGen. Given a 3D PLC $\mathcal{X}$, the actual input of TetGen is a surface triangular mesh, that is, a 2D simplicial complex that triangulates the boundary of $\mathcal{X}$. A quality tetrahedral mesh of $\mathcal{X}$ is generated in three steps.

(1) *Delaunay Tetrahedralization*. In this step, only the input vertices of $\mathcal{X}$ are considered. It creates a Delaunay tetrahedralization (DT) of the set of vertices.
(2) *Constrained Mesh Generation*. In this step, the edges and triangles, so-called constraints, of the input surface mesh of $\mathcal{X}$ are considered. Starting from the DT constructed in Step (1), an initial tetrahedral mesh that contains the constraints is constructed. Depending on the user-specified choice, the constraints either are entirely preserved or may be subdivided.
(3) *Quality Mesh Generation*. In this step, the mesh quality will be considered. Based on a set of user-specified mesh quality criteria. This Step is accomplished in two sub-phases. At first, the tetrahedral mesh constructed in Step (2) is refined by inserting new vertices. Second, the mesh quality is further improved by using a local mesh optimization scheme. Depending on the user choice, the constraints may either remain unmodified or be further subdivided for achieving better mesh quality.

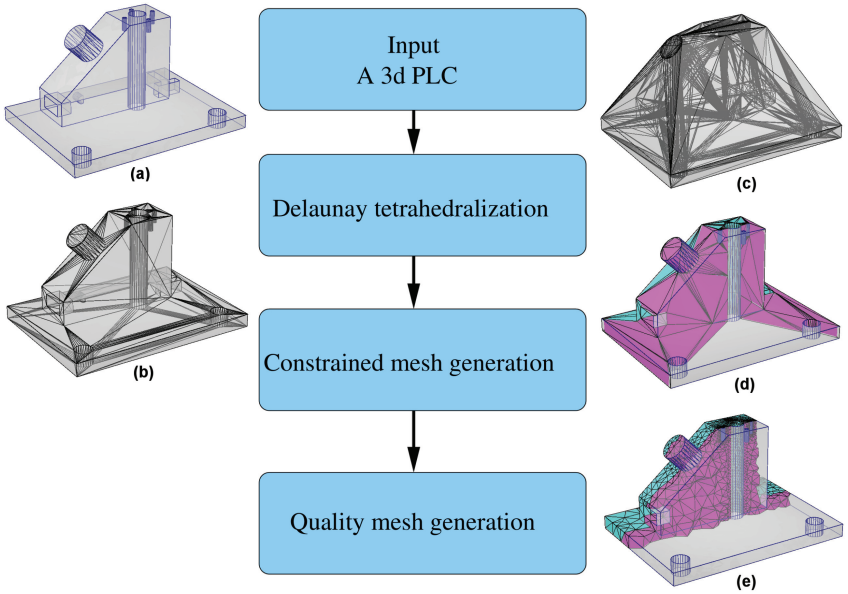


Fig. 2. The pipeline of the mesh generation process of TetGen. The main steps are shown in the flow chart. An example is shown in the labeled figures. (a) The mesh domain is modeled by a 3D PLC. (b) The actual input of TetGen is a surface triangular mesh of the boundary of the PLC. (c) The Delaunay tetrahedralization of the vertices of the PLC. (d) A constrained Delaunay tetrahedral mesh (CDT) of the PLC. (e) A quality tetrahedral mesh of the PLC.

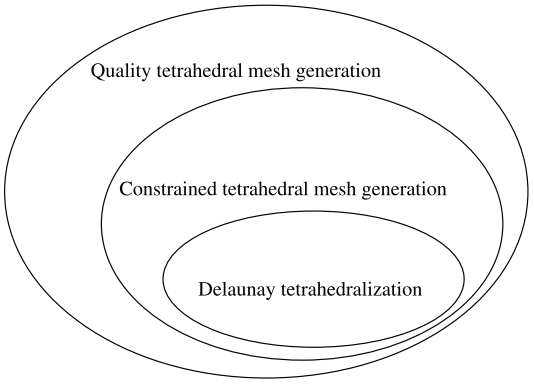


Fig. 3. The relationship of the meshing problems considered in TetGen.

3.3. An Overview of Meshing Problems and Algorithms

Each of the previous steps has its own problem and needs to be accomplished by corresponding algorithms. Moreover, these problems are related such that one is included in another (Figure 3). In this section, we give a brief description of the problems and solutions of TetGen. Detailed discussions are presented in the following sections.

Delaunay Tetrahedralization. The structure of Delaunay tetrahedralization is well studied. Optimal algorithms are proposed. The problem of Delaunay tetrahedralization is how to achieve a robust and efficient implementation.

TetGen implemented two incremental algorithms, the Bowyer-Watson [Bowyer 1987; Watson 1987] and the randomized flip [Edelsbrunner and Shah 1997] algorithms. The

key operations are point location and point insertion. TetGen implemented efficient vertex insertion algorithm based on the Bowyer-Watson algorithm (presented in Section 5.4). For large data sets, the efficiency of incremental algorithms depends on the order of point insertion. TetGen first sorts the points through a spatial point sorting scheme [Boissonnat et al. 2009]. Inserting points based on this order, each point can be located efficiently (in nearly constant time). The robustness of the implementation is achieved by using filtered exact geometric predicates. It is detailed in Section 6. A comparison with other Delaunay codes is presented in Section 10. By this comparison, the efficiency of the mesh data structure and the robust geometric predicates used in TetGen are demonstrated.

Constrained Tetrahedral Mesh Generation. The goal in this step is to preserve a set of constraints (edges and triangles) in a tetrahedral mesh. Such constraints, for examples, represent the (interior) boundaries of the domain, or special geometrical features (like flow directions, boundary layers) of the modeling equations. The problem of initial mesh generation is how to find a tetrahedral mesh with a small number of elements that represents the constraints efficiently.

Recall that in 2D, an input edge can always be enforced into a triangulation by a sequence of edge flips. Moreover, it does not need extra points. However, it is not always possible in 3D. It is well known that some additional vertices, so-called *Steiner points*, may be needed. However, many questions, like the optimal locations and the minimum number of Steiner points, are unsolved.

If the constraints are allowed to be subdivided, a good theoretical solution is to maintain a constrained Delaunay tetrahedralization (CDT). It is a variant of Delaunay tetrahedralization that is able to preserve triangles [Shewchuk 1998a, 2008]. Shewchuk proved a useful condition that guarantees the existence of a CDT without adding Steiner points when all input edges are Delaunay [Shewchuk 1998a]. Robust and efficient algorithms for constructing CDTs based on this condition are proposed [Shewchuk 2003; Si and Gärtner 2011; Si and Shewchuk 2014] (presented in Section 8.2). Most of the Steiner points are added in the input edges.

If the constraints are required to be preserved, it is not always possible to generate a CDT. The constraints needs to be recovered in a tetrahedral mesh with the help of inserting Steiner points at the interior of the mesh domain. A key operation in the recovery of constraints is to remove an edge from a tetrahedral mesh. Many other mesh operations, like edge recovery, face recovery, and vertex removal can be benefited by an enhanced edge removal operations. A new edge removal algorithm is developed for this purpose. It is presented in Section 5.3.

The next challenge is to design a robust and efficient algorithm that is able to process inputs of arbitrary complexity and place optimally interior Steiner points when they are necessary. TetGen develops an efficient algorithm to recover the constraints. it is presented in Section 8.4. Experiments and comparisons (presented in Section 10.2 and 10.3) show that this recovery algorithm introduces a relatively small number of Steiner points compared with those have been reported in the literatures.

Quality Tetrahedral Mesh Generation. The meaning of "mesh quality" depends on applications. In general, it can be interpreted as a set of geometric criteria on the desired shape, size, and orientation of the mesh elements. The problem is how to generate a good quality mesh that is best according to a set of given criteria.

A central question in quality mesh generation is how to efficiently place an appropriate number of Steiner points into the mesh domain such that they form a good quality tetrahedral mesh. Delaunay refinement [Chew 1989b; Ruppert 1995; Shewchuk 1998b] is one of few theoretical schemes that provides guarantees on mesh quality and mesh element size simultaneously. However, it may produce some very badly shaped

tetrahedra, so-called *slivers*, which contain nearly 0° or 180° dihedral angles. Moreover, Delaunay refinement may not terminate when there are *sharp features*, which are two constraints that form an acute (dihedral) angle. In practice, it is extremely hard to completely remove slivers located near sharp features.

TetGen applies the Delaunay refinement technique in boundary constrained methods for quality mesh generation. The constraints (and sharp features) are preserved and the termination is always guaranteed. The mesh quality is improved by inserting the Steiner points generated by the Delaunay refinement. If the constraints are allowed to be subdivided, TetGen maintains a CDT during the Delaunay refinement [Shewchuk and Si 2014]. The theoretical guarantees of Delaunay refinement hold on tetrahedra in places away from sharp features. Some badly shaped tetrahedra may remain. They are all located (within a bounded distance) near sharp features. The algorithm is detailed in Section 9. If the constraints need to be preserved, TetGen maintains a tetrahedral mesh as close as to a CDT. After the mesh refinement, TetGen uses a mesh improvement phase to remove slivers and improve mesh quality. Experiments show that this algorithm efficiently generates good quality meshes.

4. TETRAHEDRAL MESH DATA STRUCTURE

Many mesh data structures have been proposed (see, e.g., Dobkin and Laszlo [1989], Garimella [2002], Blandford et al. [2005], and Kremer et al. [2012]). For mesh generation purpose, two equally important considerations are the memory usage and computational efficiency, which are, however, contradicted to each other. The data structure of TetGen is specialized for tetrahedral mesh generation. It is a tradeoff between fast local navigation and modification (by storing extra incidence information) and a compact space usage (by reducing the additional information as small as possible).

4.1. Representation

A tetrahedralization $\mathcal{T}$ in TetGen is represented by a common tetrahedron-based data structure. Although there are more compact data structures, like Blandford et al. [2005], a tetrahedron-based data structure is very easy to maintain, and it requires less memory compared with other edge-based or the face-edge pair [Dobkin and Laszlo 1989; Mücke 1993] data structures.

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 4.2). 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 5.3). 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. 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 tetrahedralization belongs to two tetrahedra. This property simplifies the implementation of many mesh modification operations. This concept was already used in Guibas and Stolfi [1985] for modeling 2-manifolds and their dual.

When $\mathcal{T}$ is a tetrahedral mesh of a 3D PLC $\mathcal{X}$, TetGen stores additionally the set of segments and subfaces 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

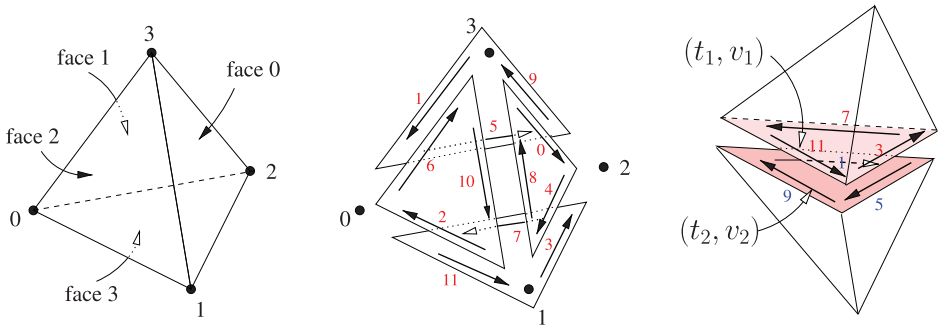


Fig. 4. Left: A numbering of the vertices and faces of a tetrahedron. Middle: A numbering of the 12 versions (directed edges) in a tetrahedron and the four edge rings. Right: A neighbor query on the handle (t_1, v_1) returns the handle (t_2, v_2). In this example, $v_1 = 11$, $v_2 = 9$, and $v_0 = 5$.

nonmanifold, the pointers to neighbors of the subfaces form a loop of singly linked list at their common edges.

For efficiency, TetGen allocates additional pointers in tetrahedra, surfaces, and segments to connect them together. For example, a subface contains two pointers to the two tetrahedra that contain it, and both tetrahedra contain a pointer to this subface. A segment contains a pointer to one of the tetrahedra that contains it, and all tetrahedra sharing at this segment contain a pointer to it.

4.2. Primitive Operations

Navigating and manipulating a tetrahedralization in TetGen are accomplished through a set of primitives. They are conceptually similar to those proposed in Guibas and Stolfi [1985] and Dobkin and Laszlo [1989], while the realization is different.

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 (Figure 4, Middle). Every three directed edges form an *edge ring* in a face of the tetrahedron, and there are four distinct edge rings. Each face of the tetrahedron can be uniquely determined by one of the edge ring. We can 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 edge ring, which is also the index of the face (Figure 4, Left), and the two upper bits encode the index (from 0 to 2) of this directed edge in this edge ring. Note that this numbering of versions is not unique, any of the directed edges of an edge ring can be chosen to have the upper index 0.

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

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 common face of t_1 and t_2 (Figure 4, Right). 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 in an edge ring of t_1 may be anyone of the three edges in the corresponding edge ring of t_2, the data structure stores the handle (t_2, v_0) in t_1, such

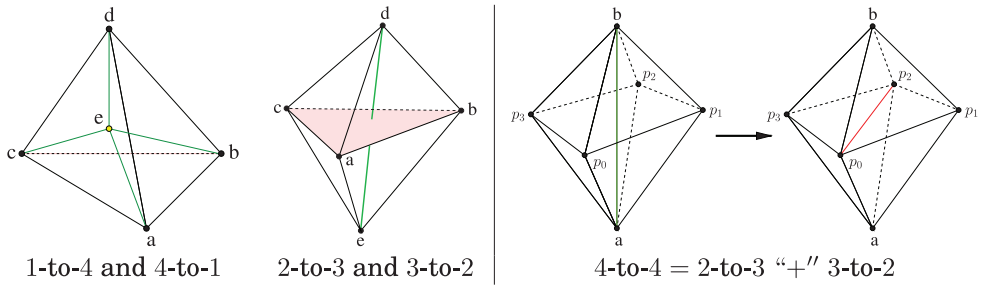


Fig. 5. Left: The four types of elementary flips in $\mathbb{R}^3$. Right: A 4-to-4 flip which interchanges two edges is a combination of a 2-to-3 and a 3-to-2 flips.

that (t_2, v_0) corresponds to the 0th edge in the edge ring of t_1. It is easy to verify, that

$$v_0 := (v_2 \ \& \ 3) + (((v_1 \ \& \ 12) + (v_2 \ \& \ 12)) \bmod 12),$$

where the symbol & means the bitwise AND operation. 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

$$v_2 := (v_0 + 12 - (v_1 \ \& \ 12)) \bmod 12.$$

Note that the values of v_0 and v_2 were previously calculated (using these formulae) and stored in two 12×12 tables, respectively.

5. LOCAL MESH TRANSFORMATIONS

A *local mesh transformation* (also referred to as *topological transformation*) replaces a set of tetrahedra with a set of different tetrahedra such that they occupy exactly the same space (referred to as *cavity*). It can be just an *elementary flip* that only interchanges two minimal sets of tetrahedra, or a combination of elementary flips, or a complicated algorithm like inserting and deleting a vertex.

Local mesh transformations are the key operations needed in almost all kinds of meshing algorithms. A comprehensive design and realization of these operations will result not only an efficient program, but also a succinct code.

5.1. Elementary Flips

According to Radon's theorem [Radon 1921], there are four elementary flips in $\mathbb{R}^3$, performed within the convex hull of five noncoplanar points. These are respectively: 1-to-4, 4-to-1, 2-to-3, and 3-to-2 flips (Figure 5, Left), where the numbers indicate the number of tetrahedra before and after each flip. 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, which removes a face, and the 3-to-2 flip, which reverses the 2-to-3 flip, are the simplest cases of inserting and deleting an edge, respectively.

Sometimes, the elementary flips are not possible. In particular, the 2-to-3 and the 3-to-2 flips are not possible when the face $[a, b, c]$ and the edge $[d, e]$ (Figure 5, Left) do not intersect in their relatively interiors. In this case, we say that the face $[a, b, c]$ as well as the edge $[d, e]$ are *not flippable*. Otherwise, they are *flippable*.

TetGen implemented all the elementary flips excluding the 1-to-4 flip, which is implemented inside the vertex insertion routine (in Section 5.4). Each routine, `flip`ij, takes an array of $i \in \{2, 3, 4\}$ tetrahedra as input and returns $j \in \{3, 2, 1\}$ new tetrahedra by the same array.

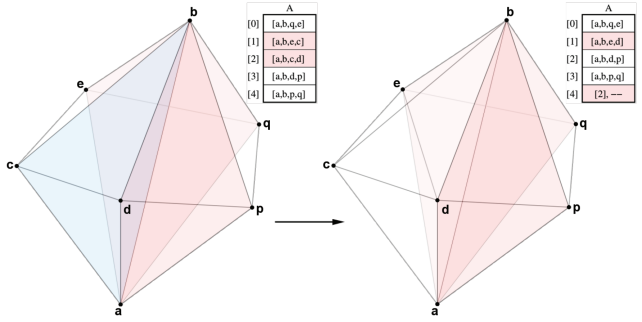


Fig. 6. Removing edge $e = [a, b]$ by flips. An illustration of the case when a face at e is removed by a 2-to-3 flip. (Left) Before the flip, there are 5 tetrahedra containing the edge $[a, b]$, and the content of array A is shown. The face $[a, b, c]$ (shown in light blue) is shared by two tetrahedra $A[1] = [a, b, e, c]$ and $A[2] = [a, b, c, d]$. (Right) After the flip, the face $[a, b, c]$ and the two old tetrahedra in $A[1]$ and $A[2]$ have been removed. The array A is shrunk by 1, and $A[1] = [a, b, e, d]$ stores the new tetrahedra.

5.2. Combinations of Flips

One can combine elementary flips to form more complex local transformations. For example, the 4-to-4 flip (Figure 5, Right), which interchanges two edges, can be regarded as the combination of a 2-to-3 flip (that inserts the desired new edge) and a 3-to-2 flip (that removes the old edge). Note that the first 2-to-3 flip may temporarily create a degenerate tetrahedron (whose volume is zero). It will be removed immediately by the followed 3-to-2 flip. Therefore, TetGen does not implement explicitly the 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 $m = 2n - 4$, that 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. In general, the sequence of 2-to-3 flips is not unique. This n-to-m flip was studied in Shewchuk [2002b] and George and Borouchaki [2003] and has been applied in various meshing purposes. However, an edge cannot be removed by an n-to-m flip when no face at this edge is flippable.

Some other types of local transformations by flips were reported. In particular, several (rather complicated) operations described in Joe [1995] are actually a combination of two n-to-m flips with the limitation that n is either 3 or 4.

In TetGen, an algorithm which attempts to remove an edge using flips is developed. It is able to combine an arbitrary number (as long as it is possible) of n-to-m flips with no limitation of n. A basic version of this algorithm is given in Section 5.3.

5.3. Edge Removal

Let e be the edge to be removed in a tetrahedralization $\mathcal{T}$. Let $A[0 \cdots n-1]$ be the array of n tetrahedra (handles) in $\mathcal{T}$ sharing at e, that is, the cardinality $|A| = n$, 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 $A[(i + 1) \bmod n]$ share a common face, refer to the left of Figure 6 for an example.

The basic idea of the edge removal algorithm is following: It first tries to reduce the cardinality $|A|$ of A by performing 2-to-3 flips to remove faces at the edge e, like an n-to-m flip, until either $|A| = 3$ or $|A| > 3$ and it cannot be reduced further. If $|A| = 3$, then either the edge is removed by a 3-to-2 flip or it cannot be flipped. In both case, the algorithm returns. If $|A| > 3$ and it cannot be reduced, the algorithm tries to remove an edge e_1 that contains one of the endpoints of e by an n-to-m flip (refer to Figure 7, Left). If e_1 is removed, then $|A|$ is reduced as well, and the algorithm continuous to reduce $|A|$

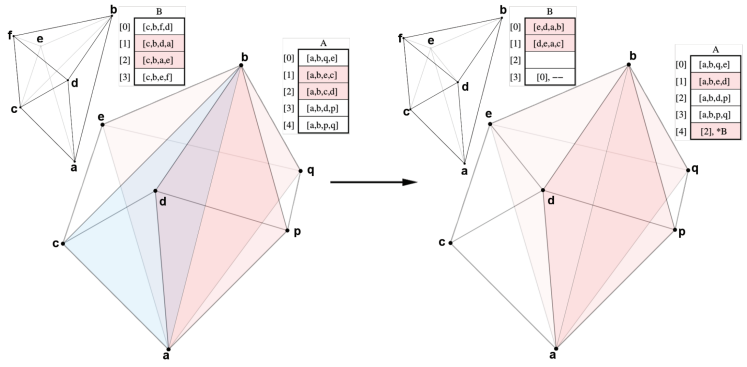


Fig. 7. Removing edge $e = [a, b]$ by flips. An illustration of the case when a neighboring edge is removed by an n-to-m flip. (Left) Before the flip, there are 5 tetrahedra containing the edge $[a, b]$, and the content of A is shown. The face $[a, b, c]$ (shown in light blue) is shared by two tetrahedra $A[1] = [a, b, e, c]$ and $A[2] = [a, b, c, d]$. In this case, the face $[a, b, c]$ is not flippable. The algorithm will remove the edge $[c, b]$. In this example, $[c, b]$ is shared by 4 tetrahedra (left-top). They are saved in the array B. Note that the two tetrahedra $[a, b, e, c]$ and $[a, b, c, d]$ are in both A and B. (Right) After the n-to-m flip, the edge $[c, b]$ is removed, and so the face $[a, b, c]$ is removed as well. The array A is shrunk by 1, and $A[1] = [a, b, e, d]$ is the new tetrahedron. The last entry of A, $A[4]$, is used to save the address of the array B.

by performing 2-to-3 flips. Otherwise, e cannot be removed and it returns. The removal of e_1 is done recursively so that more neighboring edges can be tried for removal. This increases the chance to remove e.

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 cardinality of A. It returns the final cardinality m of A, where m may be 2, which means e has been removed, otherwise, $m \geq 3$, and e is not removed. `flipnm_post` must be called immediately after `flipnm`. It takes the same array A and the two parameters, n and m. It releases the memory allocated inside `flipnm`.

Let F be the set of faces containing e, and E be the set of edges of the faces in F excluding e ($|F| = n$ and $|E| = 2n$). Normally, we can try to flip any face in F. Because if a face in F gets flipped, $|A|$ is reduced by 1. Since the routine `flipnm` will be called recursively, not every face in F should be flipped. When the `flipnm` is in a recursive call, in addition to the current array A (for removing the edge e), there might be another array B of tetrahedra (for removing an edge e'). If B does exist, then the purpose of removing edge e is for reducing $|B|$ (refer to Figure 7, Left, exchange A and B). However, to flip a face in F that belongs to a tetrahedron in $A \cap B$ may increase $|B|$. Hence, we skip flipping faces in F that belong to the tetrahedra in $A \cap B$. For the same reason, we also skip flipping edges in E that belong to the tetrahedra in $A \cap B$.

In summary, the subroutine `flipnm`$(A[0 \cdots n-1])$ 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 a face in F by `flip23`. If a face in F is successfully flipped, then $|A|$ is reduced by 1 (Figure 6, Right). It then (recursively) calls `flipnm`$(A[0 \cdots n-2])$. When no face in F can be removed, it goes to Step (3);

Step (3). ($n > 3$). It tries to remove an edge in E by a `flipnm`. Let $e_1 \in E$ be such an edge. It first initializes an array $B[0 \cdots n_1 - 1]$ of n_1 tetrahedra sharing at e_1, where $n_1 \geq 3$, then it calls $m_1 :=$ `flipnm`$(B[0 \cdots n_1 - 1])$. If e_1 has been removed, then $|A|$ is reduced by 1 (Figure 7, Right). The last entry, $A[n-1]$, is re-used to store the address of the array B (so that the memory it occupies can be released later). It then (recursively) calls `flipnm`$(A[0 \cdots n-2])$. Otherwise, e_1 is not removed, it calls

`flipnm_post`($B[0\cdots n_1-1], m_1$) to free the memory. If no edge in E can be removed, it returns $|A|$.

The subroutine `flipnm_post`($A[0\cdots n-1], m$) simply walks through the array A (from m to $n-1$). It checks if there is a saved `flipnm`. If it is found, the saved array address, for example, B, is extracted, and the memory of B is freed.

The complexity of this algorithm can be exponential with respect to the number of total flipped edges in the process. A "level" (>0) parameter is added into `flipnm` to limit the maximal number of recursions in `flipnm`.

5.4. Vertex Insertion

Vertex insertion is another key operation in generating meshes. It looks simple. However, it could be rather complicated when various requirements, like the Delaunay property, preserving constraints, quality measures, etc., are considered.

A vertex insertion can be implemented as a combination of elementary flips. A 1-to-4 flip inserts a vertex that lies in the interior of a tetrahedron. If a vertex lies on a face that is shared by two tetrahedra, t_1 and t_2, it can be inserted by combining a 1-to-4 and a 2-to-3 flips. The first 1-to-4 flip in t_1 will create a degenerated tetrahedron t_3 at the common face of t_1 and t_2. t_3 is removed immediately by the second 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 that 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.

These three cases can be unified into one process. Let $\mathcal{C}$ be the set of tetrahedra in a tetrahedralization $\mathcal{T}$ which intersect the new vertex $\mathbf{v} \notin \mathcal{T}$. Deleting all tetrahedra in $\mathcal{C}$ will create a cavity $C := \cup\,\mathcal{C}$ inside the tetrahedralization. C is a star-shaped simplicial polyhedron (whose faces are all triangles) with respect to $\mathbf{v}$, hence it can be filled by a set of new tetrahedra formed by the faces of C and $\mathbf{v}$. This process does not create any intermediate tetrahedra, hence it is general faster than performs a combination of flips. It is the basic point insertion routine implemented in TetGen.

If $\mathcal{T}$ is a Delaunay tetrahedralization, this cavity C of $\mathbf{v}$ can be enlarged such that it includes all tetrahedra in $\mathcal{T}$ whose circumspheres contain $\mathbf{v}$. C remains to be star-shaped with respect to $\mathbf{v}$, hence it can be re-tetrahedralized by the same way as previously shown. The updated tetrahedralization $\mathcal{T} := \mathcal{T} \cup \{\mathbf{v}\}$ is again a Delaunay tetrahedralization. This is the well-known Bowyer-Watson algorithm [Bowyer 1987; Watson 1987]. Alternatively, starting from the initial cavity C, one can apply a sequence of elementary flips to update $\mathcal{T}$ into a Delaunay tetrahedralization $\mathcal{T} := \mathcal{T} \cup \{\mathbf{v}\}$. This is the incremental flip algorithm [Edelsbrunner and Shah 1996]. Both algorithms are available in TetGen for creating Delaunay tetrahedralizations.

Vertex insertion in a nonDelaunay tetrahedralization or a constraint tetrahedral mesh is much more complicated. Although the basic idea of Bowyer-Watson algorithm can still be used, but the formed cavity C may not be star-shaped anymore. A validation and correction process is needed. The simplest way is to remove tetrahedra in the set $\mathcal{C}$ incrementally until the reduced cavity C becomes star-shaped. Although it should succeed theoretically, but to realize this process robustly is nontrivial. On the other hand, the incremental flip algorithm for vertex insertion remains simple in this situation, and it can be made robust easily. After the initial cavity is formed (by a 1-to-4, or 2-to-6, or n-to-$2n$ flip), one can apply a sequence of flips to update the tetrahedralization. The flips can be trigged by various criteria, such as the Delaunay property or some mesh quality measures. This process of flips terminates when no further flip can be performed.

TetGen implemented a general-purpose vertex insertion algorithm that uses the Bowyer-Watson scheme. It can be used in either Delaunay or nonDelaunay tetrahedralizations, and constraint tetrahedral meshes. In all of the cases, TetGen ensures

that a valid cavity will be obtained. Constraints are preserved when it is used in constrained tetrahedral mesh generation and refinement. TetGen also implemented a flip algorithm for both vertex insertion and incremental mesh modification.

6. FILTERED EXACT GEOMETRIC PREDICATES

Geometric predicates are simple tests of spatial relations of a set of geometric objects, such as points, lines, planes, and spheres. Two important predicates are:

—the `orient3D` test, which decides the position of a point relative to a plane defined by three other noncollinear points, and
—the `in_sphere` test, which decides whether a point lies in the relative inside or outside of a sphere defined by four other noncoplanar points.

Note there are exactly two orientations of a plane (or a sphere), which correspond to an even and an odd permutation of the sequence of the given points, respectively. It is up to the user to decide which one is "positively" oriented (so the other one is "negatively" oriented). For example, one can use either the left-handed rule or the right-handed rule to define a positively oriented plane.

Assuming the given points that define the plane or sphere are positively oriented, each of these tests is performed by evaluating the sign of the determinant of a matrix whose entries are the coordinates of the involved points, that is,

$$\texttt{orient3D}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d}) = \text{sign}(\det(A)) \quad \text{and} \quad \texttt{in\_sphere}(\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{d},\mathbf{e}) = \text{sign}(\det(B)),$$

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, for example, 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 the program crash. Computing the predicates exactly will avoid the rounding error and make the program robust [Yap 1997].

6.1. Filtered Exact Predicates

Exact arithmetics are usually expensive to compute. For predicates like `orient3D` and `in_sphere`, which only the signs of the computations are needed, a reasonably fast approach is to use the so-called *arithmetic filters* [Fortune and Van Wyk 1996] 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 [Broennimann et al. 1998]. The static filters are previously computed. They can quickly answer many "easy cases". But they are less accurate and may fail often. The dynamic filters, which calculated within the program, are more accurate, but they require more computational time. Automatic code generation for dynamic filtered predicates have been proposed [Burnikel et al. 2001; Nanveski et al. 2003]. A study of the performances of various combinations of filters is conducted by Devillers and Pion [2003].

By default, TetGen uses Shewchuk's implementation of filtered exact predicates, referred as *Shewchuk's predicates* (`http://www.cs.cmu.edu/~quake/robust.html`) It is built upon the techniques of arbitrary precision floating-point arithmetics [Priest 1991] (based on the IEEE standard) and an adaptive scheme to automatically extend the precisions [Shewchuk 1996a]. These predicates are very efficient in practice.

Shewchuk's predicates only use dynamic filters since they have no information of the range of the input data. Since the data range is known at runtime, TetGen previously calculates a static filter for each of the predicates: `orient3D` and `in_sphere`. We refer to the reference [Devillers and Pion 2003] for how to compute a static filter. They are used before calling Shewchuk's predicates.

7. DELAUNAY TETRAHEDRALIZATIONS

Delaunay triangulations, and their dual Voronoi diagrams, are the most well studied structures in computational geometry. They have many nice properties and are found in numerous applications (see, e.g., Aurenhammer [1991] and Rajan [1994]).

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 [Delaunay 1934], that is, 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*, that is, no $d+2$ vertices of V share a common sphere, then the set of all Delaunay simplices uniquely forms the Delaunay triangulation of V.

A Delaunay tetrahedralization of a set of n points in $\mathbb{R}^3$ may have $O(n^2)$ tetrahedra. This is also the optimal worst-case runtime for any algorithm that constructs Delaunay tetrahedralizations. Nevertheless, many data sets appear in applications have linear-sized Delaunay tetrahedralizations. They can be constructed quickly. A number of efficient algorithms have been proposed [Bowyer 1987; Watson 1987; Clarkson and Shor 1989; Barber et al. 1996; Edelsbrunner and Shah 1996].

7.1. Incremental Construction

Two well-known algorithms for constructing Delaunay tetrahedralizations are the Bowyer-Watson algorithm [Bowyer 1987; Watson 1987] and the incremental flip algorithm [Edelsbrunner and Shah 1996]. Both are incremental, that is, inserting one point at a time. Each point is first located, then inserted. The only difference between these two algorithms is the insertion of a new vertex, as discussed in Section 5.4. TetGen implemented both of these two algorithms.

It has been shown that the point location can be the bottleneck for the performance of an incremental algorithm [Liu and Snoeyink 2005]. The speed of point location can be significantly improved if the points are previously sorted such that two near points in space are likely close in their insertion orders. Therefore, starting from a tetrahedron containing the last inserted point, the next point can be quickly located [Liu and Snoeyink 2005]. A good practical scheme for sorting points is proposed by Boissonnat et al. [2009]. It first groups the points using the Biased Randomized Insertion Order (BRIO) [Amenta et al. 2003]. It then orders the points within each group along a space-filling curve, like the Hilbert curve. The hope is to keep the geometric locality while ensures the randomized feature of the points in the point set.

TetGen presorts the points using the method in Boissonnat et al. [2009], then inserts the points in this order. The simple stochastic walk algorithm [Devillers et al. 2002] is used for point location. Robustness is fully guaranteed by using the two robust geometric predicates, that is, `orient3D` (for point location) and `in_sphere` (for updating the Delaunay tetrahedralization). TetGen used a simplified symbolic perturbation scheme [Edelsbrunner and Mücke 1990] to handle the *degenerate cases*, that is, 5

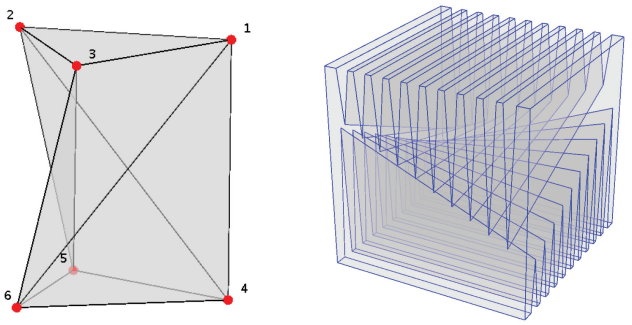


Fig. 8. Polyhedra which can not be tetrahedralized without Steiner points. Left: The Sch önhardt polyhedron [Schönhardt 1928]. Right: The Chazelle's polyhedron [Chazelle 1984].

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. All this together makes TetGen robust and efficient in computing Delaunay tetrahedralizations. A comparison with other public-domain Delaunay codes is reported in Section 10.

8. CONSTRAINED TETRAHEDRAL MESH GENERATION

A fundamental problem in mesh generation is how to generate a mesh that contains a set of constraints, such as edges or triangles. These constraints usually describe the special geometrical features like edge or corner singularities, domain boundaries, and the interface of a mesh partition. Depending on the applications, some constraints may be subdivided and some must be strictly preserved in the generated meshes.

The corresponding problem in 2D has been well solved. A triangulation $\mathcal{T}$ which contains a set $\mathcal{L}$ of noncrossing line segments can always be constructed by a modified-edge flip algorithm [Lawson 1977]. Moreover, this algorithm can create a so-called *constrained Delaunay triangulation* of $\mathcal{L}$, which has many properties as those of the Delaunay triangulation [Lee and Lin 1986]. An optimal $O(n \log n)$ algorithm to constructed a constrained Delaunay triangulation is proposed [Chew 1989a].

In three dimensions, however, this problem is far from solved. There exist (nonconvex) polyhedra which have no tetrahedralization with its own vertices, see Figure 8 for two examples. 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 to determine whether a simple polyhedron can be tetrahedralized without Steiner points [Ruppert and Seidel 1992]. On the other hand, there are polyhedra which may require a large number of Steiner points to be tetrahedralized [Chazelle 1984]. These facts make this problem difficult.

Constrained tetrahedral mesh generation has been long addressed in the literatures. There are various established methods based on the requirements of how the constraints should be represented, that is, either they must be strictly preserved or they are allowed to be subdivided. TetGen develops efficient methods that support both requirements. They are described in the following subsections.

8.1. Subdividing Constraints

When the constraints are allowed to be subdivided, theoretical solutions are available. The number of required Steiner points remains a critical issue.

Chazelle and Palios [1990] proposed an algorithm for tetrahedralizing a simple polyhedron with n vertices and r reflex edges into $O(n + r^2)$ tetrahedra. Besides the

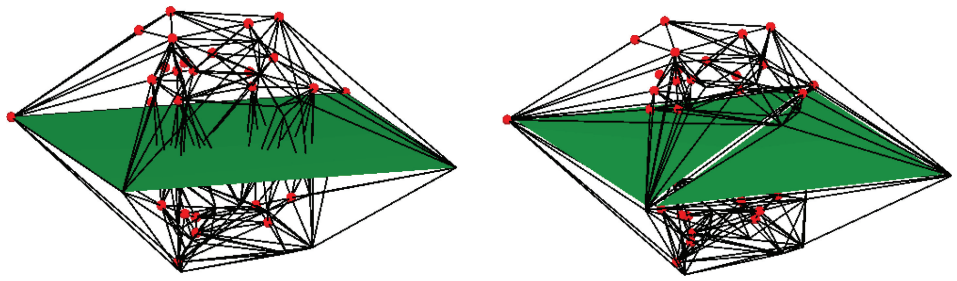


Fig. 9. Polygon insertion. Left: Inserting a rectangle (shaded) into the Delaunay tetrahedralization of a random vertex set. Right: The resulting CDT including the triangulated rectangle.

theoretical upper bound of the number of Steiner points, this algorithm is too complicated and may add Steiner points far more than necessary. Another approach constructs a *conforming Delaunay tetrahedralization* [Murphy et al. 2000; Cohen-Steiner et al. 2004]. NonDelaunay constraints are subdivided into sub-constraints, until they are represented by a union of Delaunay simplices. However, this approach may add too many Steiner points.

TetGen constructs a *constrained Delaunay tetrahedralization* (CDT) [Shewchuk 1998a, 2008], which is a generalization of the constrained Delaunay triangulation [Lee and Lin 1986] into three dimensions. CDTs have many nice properties similar to those of Delaunay tetrahedralizations [Shewchuk 2008]. A crucial difference between a CDT and a (conforming) Delaunay tetrahedralization is that some triangles are not required to be Delaunay, which frees the CDT to better respect the constraints, refer to Figure 9 for an example. Steiner points are needed in constructing CDTs. Compared with conforming Delaunay tetrahedralizations, CDTs usually contain much less Steiner points, therefore, they can also be created more efficiently.

8.2. Constrained Delaunay Tetrahedralizations

A triangle s in a tetrahedralization is said to be *locally Delaunay* if it only belongs to one tetrahedra in $\mathcal{T}$, or it is a face of two tetrahedra t_1 and t_2 and it has a circumscribed sphere that encloses no vertex of t_1 and t_2. A tetrahedralization $\mathcal{T}$ of a 3D PLC $\mathcal{X}$ is a *constrained Delaunay tetrahedralization* (CDT) of $\mathcal{X}$ if every triangle in $\mathcal{T}$ not included in a polygon in $\mathcal{X}$ is locally Delaunay.

A CDT of a 3D PLC may contain Steiner points. Shewchuk proved a condition [Shewchuk 1998a] which guarantees the existence of a CDT of a 3D PLC $\mathcal{X}$ 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 with no Steiner point [Shewchuk 1998a]. 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 the segments of $\mathcal{X}$, such that $\mathcal{Y}$ does have a CDT. It is called a *Steiner CDT* of $\mathcal{X}$. Note that if the point set of $\mathcal{X}$ is in general position, that is, no five vertices of $\mathcal{X}$ share a common sphere, Shewchuk's condition simplifies to require that every segment of $\mathcal{X}$ to be Delaunay.

Existing theoretical CDT algorithms [Shewchuk 2002a, 2003; Si and Gärtner 2005, 2011] all make use of Shewchuk's 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) insert the segments of $\mathcal{X}$; and (3) insert the polygons of $\mathcal{X}$. Theoretically, Steiner points are only needed in step (2) to ensure the existence of a CDT.

The segment insertion algorithm in TetGen [Si and Gärtner 2005] uses three simple rules to choose Steiner points to split non-Delaunay segments. This algorithm avoids

creating unnecessarily short edges. It is guaranteed to terminate. Unfortunately, the theoretical bound on resulting edge length is a constant which may be arbitrarily small. In practice, this algorithm is very efficient and creates no small edges as those predicated by theory. TetGen uses symbolic perturbation in the `in_sphere` test to simplify Shewchuk's condition. The running time of this algorithm is proportional to the number of Steiner points.

Polygon insertion is the key step in creating CDTs. TetGen implemented two algorithms, a flip algorithm [Shewchuk 2003] and a cavity retetrahedralization algorithm [Si and Gärtner 2011] to insert the polygons incrementally, see Figure 9 for an example. A comparison of these two algorithms is given in Si and Shewchuk [2014]. It shows that both algorithms are equally efficient. The cavity-retetrahedralization algorithm is more robust in practice.

Although it is theoretically guaranteed that no Steiner point is needed in the polygon insertion, it is based on the assumption that all vertices of the polygon are exactly coplanar. 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 these two polygon insertion algorithms to fail. This failure can be avoided by adding Steiner points in polygons [Si and Shewchuk 2014].

8.3. Preserving Constraints

The requirement that no constraint is allowed to be subdivided imposes a stronger restriction to the freedom of placing Steiner points, that is, 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 right structure for this problem, and which condition can certify the existence of such tetrahedralizations. More importantly, the question, "what is the optimal number of Steiner points", is still widely open.

Beside the theoretical questions, this problem has been long addressed in literatures. The most popular methods are those proposed in George et al. [1991, 2003] and Weatherill and Hassan [1994]. These methods have been shown successful in applications. However, the main used technologies, such as edge/face swaps (a synonym for flips), vertex deletions and suppressions are all not well-solved problems. More sophisticated methods to deal with these issues are proposed [Liu and Baida 2000; Du and Wang 2004; Chen et al. 2011]. However, these methods may be prohibitively slow and may have severe robustness issues.

The method implemented in TetGen resembles to those presented in George et al. [1991, 2003] and Weatherill and Hassan [1994], while new algorithms are developed for recovering the constraints. It works in three steps: (1) recover edges, (2) recover triangles, and (3) vertex suppression. In steps (1) and (2), the constraints are recovered by combining flips and Steiner points insertions. 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.

In the following, we describe the edge recovery algorithm of TetGen. The face recovery algorithm is basically done in the same spirit. Vertex suppression is done in the similar way as that was described in George et al. [2003].

8.4. Edge Recovery

The edge recovery algorithm initializes an array of all edges to be recovered, then it recovers them one by one. Let e be an edge to be recovered. Let F_e be the set of faces in current tetrahedralization $\mathcal{T}$ whose interior intersect e, and $F_e \neq \emptyset$. The algorithm tries to reduce the cardinality $|F_e|$ of F_e by removing a face in F_e one at a time. If $|F_e| = 0$, e must be recovered.

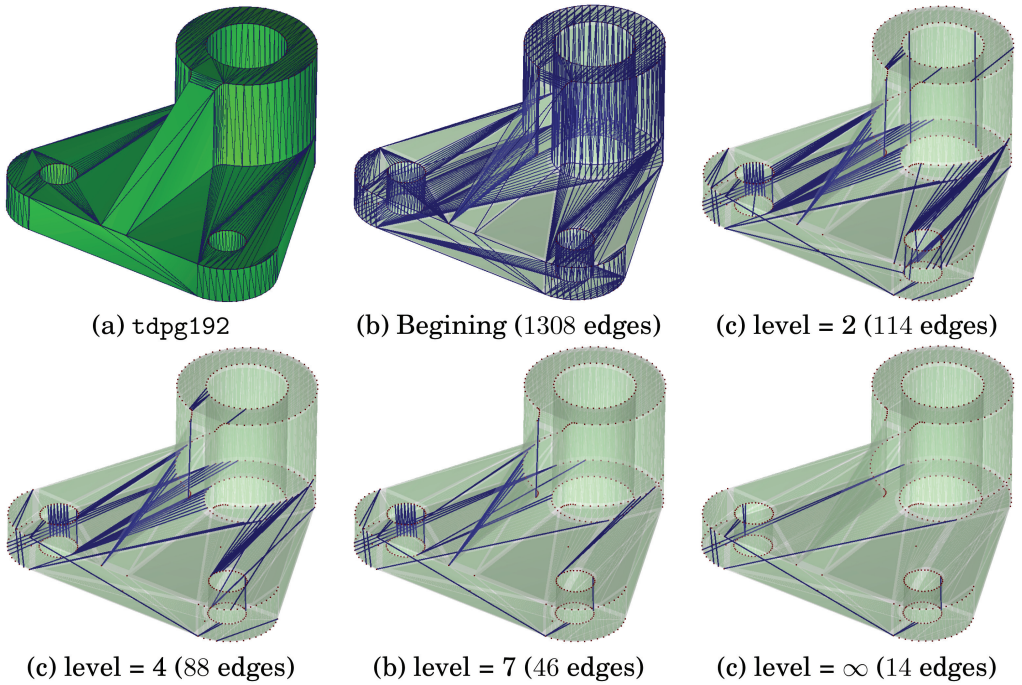


Fig. 10. An example of the edge recovery algorithm. The PLC model (432 vertices, 872 triangles) is shown in (a). All edges (1308) to be recovered are highlighted in (b). From (c) - (e), the missing edges after different levels are highlighted. 14 unrecovered edges are shown in (f).

Let $f \in F_e$. If f is flippable, that is, by a 2-to-3 flip, it is simply removed by `flip23`. 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. In both cases, $|F_e|$ is reduced by 1. Once an edge is recovered in $\mathcal{T}$, it is "locked" in $\mathcal{T}$ and will never be flipped away. If e cannot be recovered by this process, we split e by adding a Steiner point in it. A simple choice of a Steiner point is just its midpoint. The two resulting edges will be recovered by the same procedure.

While recovering an edge e, the `flipnm` operation will automatically search those flippable edges surrounding e and flip them. To ensure that the algorithm will terminate, we avoid creating a new face in $\mathcal{T}$ whose relative interior intersects e, hence $|F_e|$ never increases. A callback function is fed to the routine `flipnm` to reject flips which violate this condition. (Note that this condition might be relaxed.)

Since the `flipnm` operation will be called recursively, this edge recovery algorithm may become very slow when the number of recursions becomes large. We introduced a "level" parameter (> 0) to the routine `flipnm`. It limits the number of recursions inside a `flipnm` call. The edge and face recovery procedures are all iterated on the increasing number of "level"s starting at "level $= 1$". This gives a very good performance in practice. Typically, a large number of constraints are recovered in the iterations of first few levels. An example is shown in Figure 10.

Extensive experiments show that this method, which uses the new edge removal algorithm in Section 5.3, results a very low number of missing edges and faces, thus it adds a very small number of Steiner points. More experiments of this algorithm are reported in Section 10. A comparison of both the number of Steiner points and performances with other methods are also reported therein.

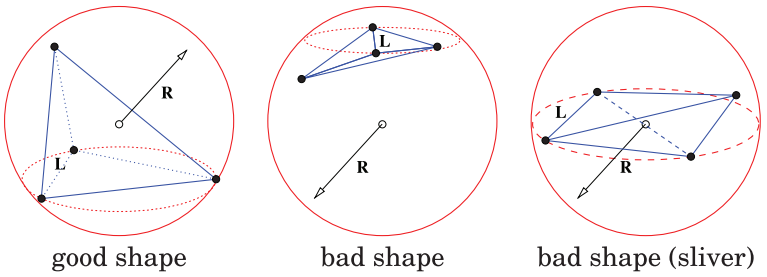


Fig. 11. The radius-edge ratio ($\frac{R}{L}$) of tetrahedra. Most of the badly shaped tetrahedra will have a large radius-edge-ratio except slivers (Right).

9. QUALITY TETRAHEDRAL MESH GENERATION

The meaning of "mesh quality" varies in different applications. In the context of numerical simulation, the mesh quality could be judged by the error on the interested numerical quantity or the efficiency on the solution time [Frey and George 2000]. Numerical PDE analysis showed that the mesh quality is a complex function that can be interpreted as a combination of several geometric measures on the shape, size, and orientation of the mesh elements [Shewchuk 2002c; Huang 2005; Mirebeau 2012]. Some numerical schemes showed that special geometric structure, such as the Delaunay-Voronoi structure, can carry important qualitative properties from the continuous to the discrete level [Si et al. 2010].

Suppose a set of desired criteria on mesh quality is given, the *quality mesh generation problem* studies how to generate the "best mesh" that has a small number of elements with the appropriated shape, size, and orientation. It is in general a very complex problem even for some very simple geometric criteria, such as the minimum or maximum dihedral angles in tetrahedra. Moreover, the requirement of Delaunay-Voronoi structure of the mesh in 3D is very challenging due to the complexity of the domain boundaries. This makes the general quality tetrahedral mesh generation problem difficult, both in theory and practice.

9.1. Tetrahedron Shape Measures

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. In these cases, it is necessary to have some small angles in the tetrahedra, but no small and large dihedral angles, 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° and 180° (Figure 11, Right).

A commonly used shape measure for a simplex is the aspect ratio. The *aspect ratio*, $\eta(\tau)$, of a tetrahedron τ can be defined as the ratio between the longest edge length l_{max} and the shortest height h_{min}, that is, $\eta(\tau) = l_{max}/h_{min}$. It measures the "roundness" of a tetrahedron in terms of a value between $\sqrt{2}/\sqrt{3}$ and $+\infty$. A low-aspect ratio implies a better shape. The *radius-edge ratio*, $\rho(\tau)$ of a tetrahedron τ is the ratio between the radius R of its circumscribed ball and the length L of its shortest edge, that is, $\rho(\tau) = \frac{R}{L} \geq \frac{1}{2\sin\theta_{min}}$, where θ_{min} is the smallest face angle of the tetrahedron τ (Figure 11). The radius-edge ratio $\rho(\tau)$ is at least $\sqrt{6}/4 \approx 0.612$, achieved by the regular tetrahedron. Most of the badly shaped tetrahedra will have a big radius-edge ratio. However, a sliver can have a relatively small radius-edge ratio, hence it is a flawed

tetrahedron shape measure. However, the radius-edge ratio is useful in theoretical analysis of Delaunay refinement algorithms.

TetGen currently uses several geometric shape measures, including the smallest face angle (i.e., the radius-edge ratio), the minimum and maximum dihedral angles.

9.2. Tetrahedral Mesh Refinement

TetGen considers the following mesh refinement problem: given a 3D PLC $\mathcal{X}$ and an initial tetrahedral mesh of $\mathcal{T}$ of $\mathcal{X}$, a set of tetrahedron shape measures, how to generate a tetrahedral mesh of $\mathcal{X}$ with good-shaped tetrahedra.

A central question in this problem is how to place Steiner points efficiently so that a tetrahedral mesh containing these points simultaneously satisfies the desired properties. Various approaches have been developed for this purpose, such as, Octree [Mitchell and Vavasis 2000], sphere packing [Miller et al. 1996], Longest-Edge Propagation Path (LEPP) [Rivara 1997], and Delaunay refinement [Chew 1989b; Ruppert 1995; Shewchuk 1998b; Hudson 2007].

Delaunay refinement is one of the few methods which provide theoretical guarantees on mesh quality and mesh size. Its main idea is relatively simple, that is, 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 from below. Moreover, it results locally adapted mesh elements with respect to the local feature size function [Ruppert 1995].

However, Delaunay refinement may produce slivers, that is, it has no guarantee on the smallest dihedral angle. A number of approaches have been proposed to remove slivers [Cheng et al. 2000; Li and Teng 2001].

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. The algorithm of Shewhcuk [1998b] requires that no acute dihedral angle in the input. Si [2009] proved that it can be reduced to $\arccos\frac{1}{3} \approx 70.53^\circ$. However, it is still a very strong limitation in practice. Handling sharp features remains a challenging problem in Delaunay refinement. Several methods have been proposed [Cheng et al. 2005; Pav and Walkington 2004; Rand and Walkington 2009]. However, they are usually very complicated and may over-refine the sharp features – that is, to produce too many tetrahedra, making them too small.

TetGen uses a new mesh refinement algorithm [Shewchuk and Si 2014]. Instead of updating a conforming Delaunay mesh, it updates a constrained Delaunay mesh (CDT). The boundary conformity and Steiner points insertion are well combined during the mesh refinement. Sharp features are recovered and maintained simultaneously by the CDT. This algorithm generates a tetrahedral mesh of the domain with well-shaped tetrahedra. It is described in the next section.

9.3. Constrained Delaunay Refinement

The input of this algorithm is a CDT of a PLC $\mathcal{X}$ and a positive constant B that specifies the maximum permitted radius-edge ratio for tetrahedra in the output mesh. A tetrahedron is called *skinny* if its radius-edge ratio exceeds B. The algorithm is guaranteed to terminate and produce a mesh if $B \geq 2$. If $\mathcal{X}$ contains no small angles, the mesh has no skinny tetrahedra. Otherwise, the mesh may have some, but a skinny tetrahedron can exist only if it adjoins (has at least one vertex lying on) the relative interior of a segment or polygon that forms a small angle. Every other tetrahedron is guaranteed not to be skinny.

The refinement algorithm maintains the CDT of an augmented PLC $\mathcal{Y}$ as it adds new vertices to $\mathcal{Y}$. The PLC $\mathcal{Y}$ is $\mathcal{X}$ with additional vertices, so the CDT of $\mathcal{Y}$ is a Steiner CDT

of $\mathcal{X}$. This algorithm has refinement rules typical of tetrahedral Delaunay refinement: skinny tetrahedra are "split" by new vertices inserted at their circumcenters, and "encroached" subsegments and subpolygons are split likewise. However, the rules are modified so that small domain angles do not cause havoc.

For each vertex v in the mesh, its *insertion radius* r_v is the distance to the closest distinct vertex visible from v at the moment when v is first inserted into the PLC $\mathcal{Y}$. If $\mathcal{Y}$ has a CDT, r_v is, equivalently, the length of the shortest edge that initially adjoins v. This definition differs from that of most Delaunay refinement algorithms by considering visibility; CDTs do not connect vertices that cannot see each other.

Unfortunately, the usual relationships in standard Delaunay refinement algorithms between insertion radii do not hold where PLC polygons or segments meet at small angles. Sometimes, it is necessary to insert a new vertex that creates a CDT edge that is much shorter than any prior edge. The central idea of our algorithm is to deprive those unreasonably short edges of the power to cause further refinement. Specifically, if a tetrahedron is skinny because it has an unreasonably short edge, we may decline to try to split the tetrahedron. This breaks an endless cycle wherein ever-shorter edges drive the creation of yet shorter edges. The cost is that some skinny tetrahedra survive in the final mesh, but only near small domain angles.

This policy is implemented by storing for each vertex v a *relaxed insertion radius* rr_v, which always satisfies the constraint $rr_v \geq r_v$. For most vertices, including all input vertices, $rr_v = r_v$. However, when the algorithm is forced to create a new edge that it considers to be unreasonably short, the newly inserted vertex v has rr_v greater than the length of that edge. This "tells" the algorithm that skinny tetrahedra having that edge should not be split if the splitting would create edges shorter than rr_v.

Specifically, when the radius-edge ratio of a tetrahedron t is computed, t's shortest edge is pretended that it is not shorter than rr_v for the minimizing vertex v of t. The *relaxed shortest edge length* ℓ_t of t is either the length of t's shortest edge or the smallest relaxed insertion radius among t's vertices—whichever is *greater*. The *relaxed radius-edge ratio* of t is t's circumradius divided by ℓ_t. We say that t is *splittable* if its relaxed radius-edge ratio exceeds B. Every splittable tetrahedron is skinny (has a radius-edge ratio greater than B), but not every skinny tetrahedron is splittable. Our algorithm eliminates all tetrahedra that are splittable but not *fenced in*, which is a tetrahedron whose circumcenter is not visible from its interior. The visibility is blocked by a segment or a polygon of the PLC.

As an experimental test, we created a PLC with 64 irregular "fan blades" adjoining at a common segment separated by very small dihedral angles, ensuring a great deal of mutual encroachment. Figure 12 shows the PLC and the mesh generated by this algorithm with a radius-edge ratio bound of $B = 2$. The main observations are that this algorithm successfully produces a mesh, the surviving skinny tetrahedra are all nested within the fan blades.

Removing Slivers by Constrained Delaunay Refinement. Delaunay refinement only bounds the face angles, but not the dihedral angles. Therefore, slivers may not be removed. However, it is shown in Shewchuk [1998b] and Si [2008] that Delaunay refinement is able to remove slivers despite there is no guarantee of termination. 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 using the constrained Delaunay refinement algorithm. 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°) to distinguish slivers. In practice, we observed that this constrained Delaunay refinement algorithm may terminate on a smallest dihedral angle bound as larger as 20° (Figure 13).

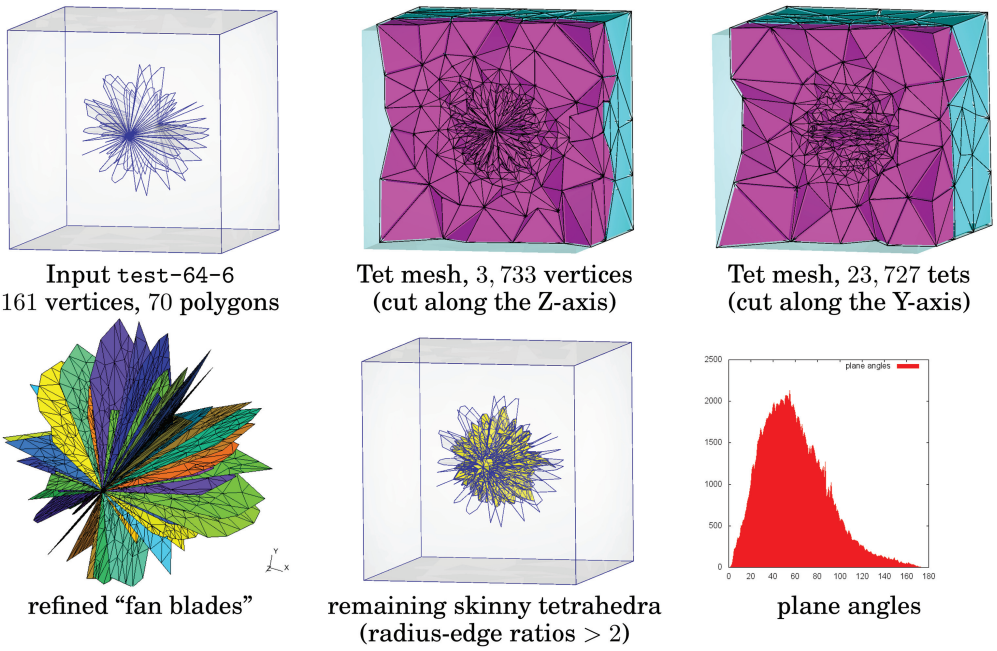


Fig. 12. An example (test-64-6) of constrained Delaunay refinement.

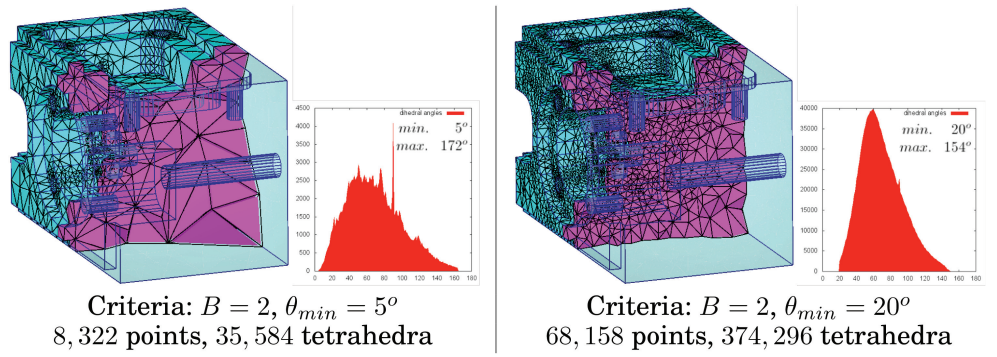


Fig. 13. Removing slivers by constrained Delaunay refinement. Two experiments on using a different minimum dihedral angle bound, 5° and 20°, are shown.

Adaptive Mesh Refinement. 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, for example, through a priori or a posteriori error estimators. It specifies the desired element size (such as the edge lengths) on the domain.

TetGen takes an optional isotropic mesh sizing function H as input. When it is available, in addition to checking the radius-edge ratio and minimum dihedral angle of a tetrahedron t, TetGen checks whether t satisfies the mesh size requirement. For simplicity, it checks if $r < H(\mathbf{c})$, where r, $\mathbf{c}$ are the circumradius and circumcenter of t, respectively. This algorithm then can generate an adaptive mesh whose mesh size is conforming to H. Well-conformity can be achieved for smoothed sizing functions. An example is shown in Figure 14.

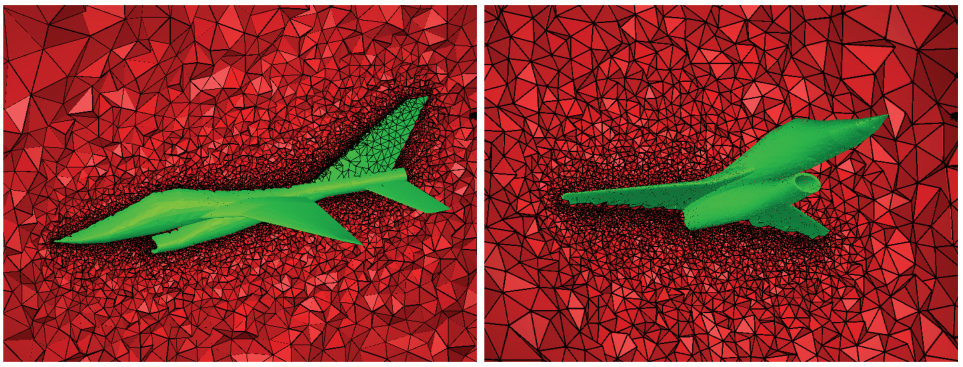


Fig. 14. An example of adaptive constrained Delaunay refinement. The input is an aircraft egads (courtesy of R. Haimes) placed inside a large sphere. The applied mesh sizing function is defined directly on the input vertices, such that all surface points of egads have a size 0.05, and all sphere points have a size 8.0. Two views of the generated tetrahedral meshes (387, 511 points, 2, 429, 987 tetrahedra) are shown.

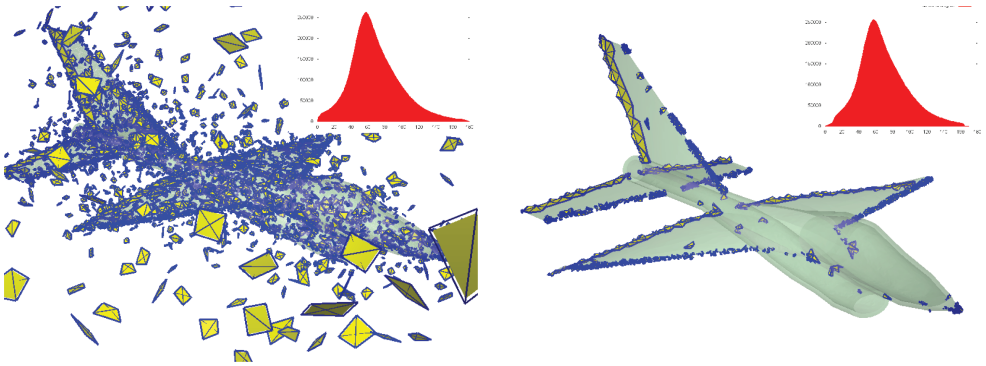


Fig. 15. Left: Before the mesh improvement, a highlight of the bad quality tetrahedra of the mesh in Figure 14. Each of these tetrahedra has either its minimum dihedral angle $< 3^\circ$ or its maximum dihedral angle $> 179^\circ$. Right: After the mesh improvement, a highlight of remaining bad-quality tetrahedra. They are all clustered near the sharp features of the input.

9.4. 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 in the adaptive mesh generation, some badly shaped tetrahedra may not be removed due to the mesh sizing limitation. For an example, a highlight of remaining bad-quality tetrahedra is shown in the left of the Figure 15.

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. 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, that is, face/edge flips and vertex insertion/deletion, and (2) vertex smoothing, that is, 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., Freitag and Ollivier-Gooch [1997] and Klinger and Shewchuk [2007]).

TetGen uses a simple "hill climbing" scheme to improve the mesh quality, that is, 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 replace bad-quality tetrahedra by improved ones. New low-quality tetrahedra are added back into the list. This process stops either when the list is empty or the maximum number of iterations is reached.

This simple scheme is effective, that is, it never degrades the mesh quality. However, it may easily get stuck in some "local optima". To overcome local optima is still a research topic. The edge removal algorithm (in Section 5.3) is applied in the mesh improvement process. It applies only after the simple edge/face swap fails. The flip sequences are reversed if the removal of an edge does not improve the mesh quality. The hope is to have more chances to overcome the local optima and thus the mesh quality can be further improved.

Figure 15 illustrates an example of TetGen's mesh improvement on the tetrahedral mesh shown in Figure 14. 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.

10. EXPERIMENTAL RESULTS

In this section, some experiments regarding the behavior and efficiency of the implemented algorithms in TetGen are reported. All experiments were performed on a 2.2-GHz Intel Cire i7 with 8 GB of 1333-MHz DDR3 memory (MacOSX 10.7.5). TetGen version 1.5.0 (November 4, 2013) was used. It was compiled using `GCC/G++` version 4.2.1 with optimization level `-O3`. Runtimes were reported in seconds. Moreover, every reported runtime was the average of at least three runs.

10.1. Comparison with Other Delaunay Codes

To experiment the efficiency of our implementation, we compared TetGen with two public codes: `qhull` (`http://www.qhull.org`) and `CGAL` (`http://www.cgal.org`).

`qhull` is a `C` program to compute convex hulls of point sets in general dimensions using the `Quickhull` algorithm [Barber et al. 1996], which computes Delaunay tetrahedralizations through computing convex hulls in 4D. We used the 2003 version of `qhull`. It was compiled using `GCC` version 4.2.1 with the compiler option `-O3 -fPIC -ansi`. We used the `qdelaunay` program provided by `qhull`, and the command "`cat file.node | qdelaunay`" to create the Delaunay tetrahedralization, where "`file.node`" contains the coordinates of a 3D point set.

`CGAL` is a `C++` geometric algorithm library. It includes an efficient implementation of the Bowyer-Watson algorithm [Boissonnat et al. 2009]. It uses spatial sorting for preprocessing the points, and it implements its own filtered exact predicates. In our tests, we used CGAL version 4.3, released in October 17, 2013. The libraries of CGAL, `libCGAL`, were compiled using the default `cmake` options (`CMAKE_BUILD_TYPE=Release`). We used the example `delaunay_3.cpp` provided by CGAL. It uses the filtered kernel `CGAL/Exact_predicates_inexact_constructions_kernel.h`, and the class `CGAL::Delaunay_triangulation_3<K>`. We modified the input part of this example so that it can read the data format of `qhull`. This example was compiled using `GCC/G++` version 4.2.1 with the compiler option `-O3 -DCGAL_NDEBUG`.

The Bowyer-Watson algorithm of TetGen was used in this comparison. TetGen can directly reads `qhull`'s data format.

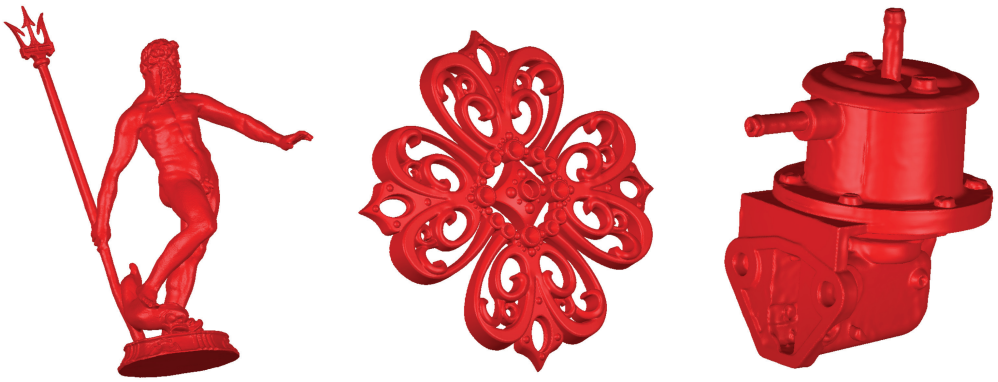


Fig. 16. Three point sets (scanned data sets from surfaces) used for experiments.

Table I. Runtime Comparison between qhull, CGAL, and TetGen

	500k	2m	q10k	Neptune	Filigree	Oilpump
# points	500, 000	2, 000, 000	10, 000	499, 417	514, 302	570, 018
# tetrahedra	3, 371, 591	13, 503, 890	24, 995, 000	3, 445, 298	3, 540, 956	4, 052, 388
qhull (2003)	20.54	107.50	N.A.	12.60	20.31	25.49
CGAL (4.3)	3.61	14.52	10.97	3.81	7.79	4.43
TetGen (1.5)	3.47	14.28	11.67	3.90	4.98	4.63

Comment: These runtimes did not include the times for reading and writing files. In the example q10k, we stopped running qhull after waiting about 20 minutes.

Six data sets were used for comparison, 500k and 2m are two randomly distributed point sets in the unit cube, q10k is a line-and-circle data set with 10,000 points that appeared in Hudson [2007]. In particular, its Delaunay tetrahedralization has a quadratic complexity. Neptune, Filigree, and Oilpump are three scanned data sets (Figure 16) obtained from the AIM@SHAPE Repository http://shapes.aim-at-shape.net.

Table I reports the running times (excluding times of file I/O) of these programs. qhull was less efficient and used much more memory than CGAL and TetGen. This is due to that the Quickhull algorithm [Barber et al. 1996] uses a special partition method to sort points. It is general in arbitrary dimensions. However, it is not efficient in constructing Delaunay tetrahedralizations.

TetGen and CGAL behaved similarly. This was expected – both of their algorithms and spatial point sorting schemes are same. The minor difference in runtime were mainly from the different mesh data structures and the robust predicates used by them. Table II reports the detailed predicates calls of TetGen in these tests. For each of orient3d and in_sphere, it reports, respectively, (1) the total number of calls, (2) the number of failures of static filters, and (3) the number of calls of adaptive version of the predicates (which may use exact arithmetics). For (2) and (3), it also calculates their percentages with respect to (1). From this information, it is possible to understand better about the difference of the runtimes in Table I.

TetGen ran slightly faster in the first two cases, that is, 500k and $r2m$. One can see that the failures of the static filters are very low. This means that nearly no dynamic filter and no exact arithmetics are needed. The difference of runtimes largely depends on the efficiency of the mesh data structure. In this context, TetGen's mesh data structure behaved more efficiently than CGAL's.

Table II. Statistics of Detailed Calls of Predicates in TetGen

	500k	2m	q10k	Neptune	Filigree	Oilpump
(1) orient3d calls	11, 263, 764	47, 066, 174	11, 009, 256	13, 990, 936	16, 180, 156	16, 080, 948
(2) static filter fails	6, 635	84, 739	251, 055	372	465, 555	355
	(0.06%)	(0.18%)	(2.28%)	(0.00%)	(2.88%)	(0.00%)
(3) adaptive calls	477	4, 698	61, 968	0	439, 612	138
	(0.00%)	(0.01%)	(0.56%)	(0.00%)	(2.72%)	(0.00%)
(1) in_sphere calls	22, 352, 563	89, 548, 559	52, 438, 540	23, 570, 480	24, 722, 501	27, 698, 380
(2) static filter fails	14, 092	536, 675	29, 222, 329	13, 307, 865	6, 706, 049	7, 654, 743
	(0.06%)	(0.60%)	(55.73%)	(56.46%)	(27.13%)	(27.64%)
(3) adaptive calls	11	44	11	0	154, 038	1
	(0.00%)	(0.00%)	(0.00%)	(0.00%)	(0.62%)	(0.00%)

TetGen ran slightly slower in the three cases, that is, q10k, Neptune, and Oilpump. We observed that the failures of static filters largely increased in these cases. Especially in the in_sphere predicates, nearly half of the times, the static filters failed. While there were nearly no adaptive calls, that is, (3). Obviously, the calculation of dynamic filters increased the runtime of TetGen. CGAL not only uses static filters, but also uses filters based on the interval arithmetics [Broennimann et al. 1998], which give more accurate error bounds. This avoids expensive calculations.

The case Filigree was more interesting. The runtime of CGAL was much slower than that of TetGen. We observed that there were relatively high numbers of adaptive calls, that is, (3), both in the orient3d and in_sphere tests, for instance, 0.62% of in_sphere tests might require the expansive exact arithmetics. This implies that the exact arithmetic algorithms used in CGAL was less efficient than that of Shewchuk's predicates [Shewchuk 1996a] that are used by TetGen.

10.2. 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 5.3) with an increasing "level" has been shown very effective in recovering boundaries in practice. Here we show further experiments of the boundary recovery algorithm on some publicly available examples.

The inputs: cami1a (Figure 2), tdpg192 (Figure 10), thepart (Figure 13), anc101 (Figure 17 left), thru-mazewheel (Figure 17 middle), monster (Figure 17 right), and mohne (Figure 18) are freely available from INRIA's Mesh Repository (http://www-roc.inria.fr/gamma/gamma/gamma.php). They usually contain very badly shaped triangles (with very small angles), which are not suitable as inputs for quality mesh generation. However, they are fairly good cases for testing the robustness of the boundary recovery algorithms.

Table III reports the statistics of the number of Steiner points (located in the interior of the domain) and running times of TetGen on these examples. The results showed that TetGen only added very few Steiner points and in a very efficient way.

Table IV shows some detailed statistics of the edge recovery and face recovery algorithms. These experiments showed that the majority of the edges were recovered after the finish of the first search level ("level" = 1). As the search level increased, more and more edges were recovered. Only few edges remained unrecovered after the full search ("level" = ∞). Steiner points were only needed in these cases.

Once all edges were recovered (or split by Steiner points), the recovery of triangles was much easier. In these examples, after "level = 1", almost all triangles have

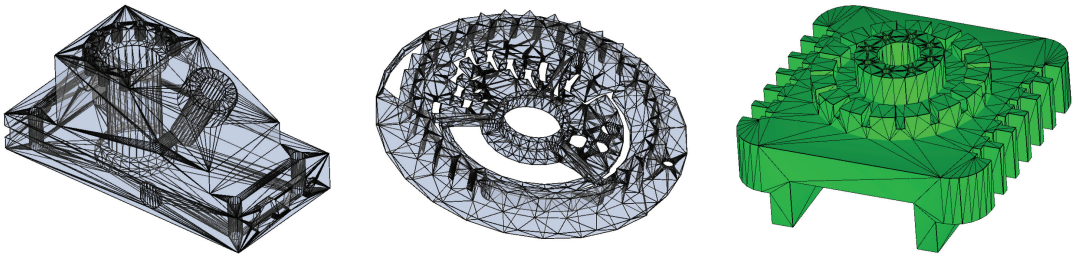


Fig. 17. Three inputs for the boundary recovery algorithm.

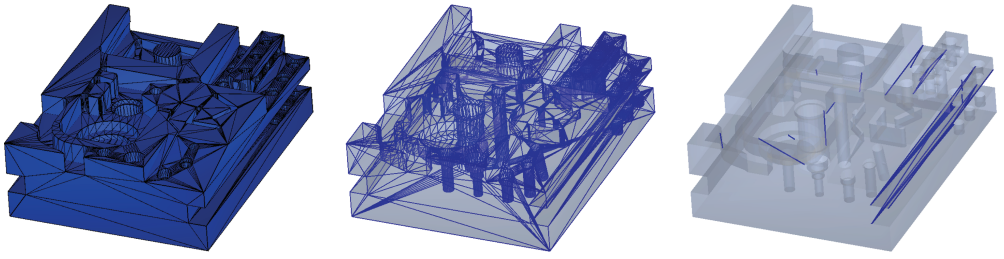


Fig. 18. An example of the edge recovery algorithm. From left to right: The input PLC (mohne), all the edges to be recovered, and the unrecovered edges after the algorithm.

Table III. Statistics of the Boundary Recovery Algorithm

	tdpg192	cami1a	thepart	anc101	monster4	mohne	thru-maze
# boundary points	432	460	994	1378	1392	2760	2781
# boundary edges	1308	1349	2988	4158	4176	8340	8433
# boundary triangles	872	884	1995	2772	2784	5560	5622
# mesh points	440	462	994	1378	1392	2767	2789
# mesh tetrahedra	1439	1529	3643	5293	4295	9078	6335
# Steiner points	8	2	0	0	13	7	8
running time (sec.)	0.052	0.043	0.050	0.083	0.128	0.184	0.113

been recovered. However, there were few cases, like those in monster4, mohne, and thru-mazewheel, still needed to add Steiner points.

Remark. Since we used a relatively strong condition (i.e., no creation of new intersecting faces of the recovering 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.

10.3. Comparison with Other Boundary Recovery Methods

A theoretical question is how many Steiner points are necessary in boundary recovery. Here, we compare our method with three other methods in terms of the used number of Steiner points. These methods are representative of most of the proposed methods. Moreover, statistics on public examples are available.

(1) [George et al. 1991]. This method is one of the most cited ones in the literature. It is implemented in the commercial program Tetmesh-GHS3D [TetMesh-GHS3D 2010]. This method only uses elementary flips.

Table IV. Detailed Statistics of the Edge and Face Recovery Algorithm

	tdpg192	cami1a	thepart	anc101	monster4	mohne	thru-maze
# edge recovery	1308	1349	2988	4158	4176	8340	8433
'level' = 1	193	81	168	295	275	279	182
'level' = 2	125	51	66	137	138	145	69
'level' = 3	103	41	45	51	101	96	40
'level' = 4	83	28	35	40	75	71	28
'level' = ...	...	...	...	...	...	...	...
'level' = ∞	13	13	6	21	53	31	16
# triangle recovery	872	954	1992	2772	2784	5566	5624
'level' = 1	0	4	1	3	2	4	2
'level' = 2	0	0	0	1	2	3	1
'level' = ...	...	...	...	...	...	...	...
'level' = ∞	0	0	0	0	2	1	1

Comment: The numbers in this table were unrecovered edges and triangles.

Table V. Comparison of the Numbers of Steiner Points

	cami1a	thepart	mohne	thru-maze
(1) [George et al. 1991]	N.A.	24	1605 (F)	18
(2) [Liu and Baida 2000]	N.A.	6	26	13
(3) [George et al. 2003]	14	N.A.	42	16
TetGen's method	2	0	7	8

Comment: Some results of the tested examples were not available. They are marked as "N.A." The statistics in (1) and (2) were both reported in Liu and Baida [2000]. Method (1) reportedly failed on the example mohne.

(2) [Liu and Baida 2000]. This method uses combination of flips. However, only very limited combinations of flips (described in Joe [1995]) were applied.
(3) [George et al. 2003]. This method is also implemented in the program Tetmesh-GHS3D as an alternative choice of the method [George et al. 1991].

Table V reports the comparison of the numbers of Steiner points used by these three methods and TetGen's method, respectively. In all available statistics, TetGen used the smallest number of Steiner points.

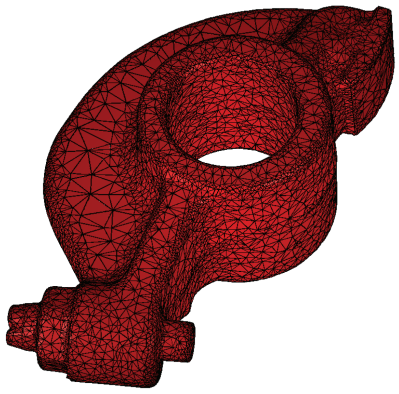
Remark. We comment that the comparison on the number of Steiner points made in this section was very rough. It is still far from getting any useful conclusion. The fact is that there is not enough published data available.

10.4. Comparison with Other Quality Tetrahedral Meshing Codes

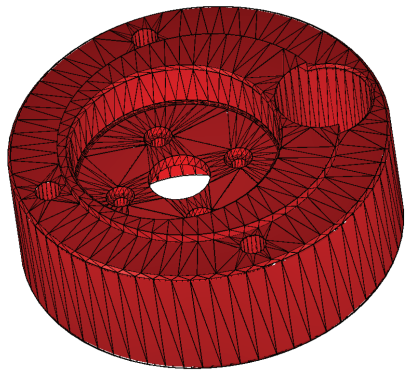
It is generally difficult to make a fair comparison between different codes. Because they usually focus on different applications, their methods are very different, and they prefer different input objects. In any case, it is interesting to compare their generated meshes for common 3D input objects.

In this section, we present some primitive experimental comparisons with two other quality tetrahedral meshing codes: CGALmesh [Jamin et al. 2013] and Tetmesh-GHS3D [TetMesh-GHS3D 2010]. We used the same inputs for these codes and TetGen, and compared the generated meshes regarding their mesh quality and speed.

CGALmesh is a mesh generation package of the CGAL library. It is based on the Delaunay refinement paradigm. It supports various types of input domain representations. By default, it performs boundary remeshing, that is, the input domain boundary mesh is disregarded and a new boundary mesh is returned. It also supports

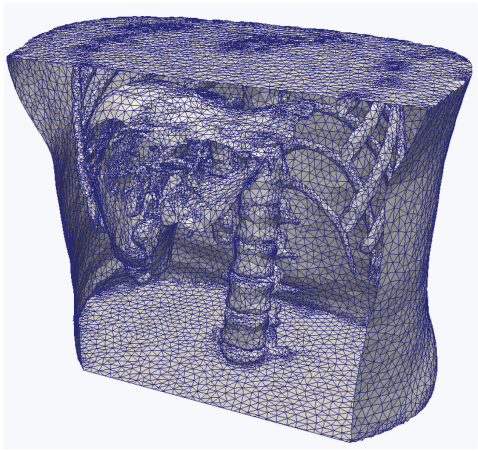


Input: rocker-arm,
10,044 vertices, 20,088 triangles

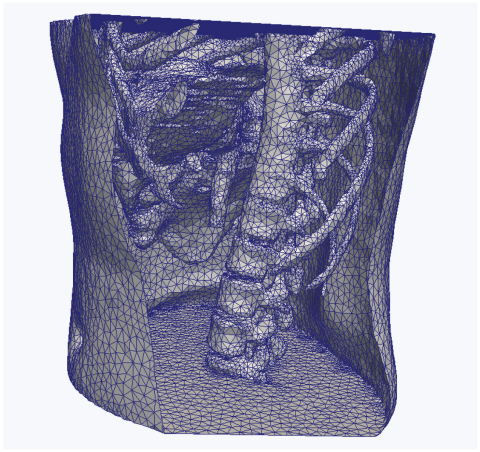


Input: couplingdown
1,841 vertices, 3714 triangles

Fig. 19. Two input surface meshes for comparison. The left (rocker-arm) approximates a C^1 smooth surface, and the smoothness of the right (couplingdown) is only C^0, that is, it contains sharp features.



Input: patient-organs02



51,124 vertices, 71,035 triangles

Fig. 20. The input surface mesh for comparison.

preserving features (line segments) of the input boundary mesh. We used the example, mesh_polyhedral_domain_with_features.cpp provided in CGAL version 4.3, released in October 2013. This example was compiled using GCC/G++ version 4.2.1 with the compiler option -O3 -DCGAL_NDEBUG.

Tetmesh-GHS3D is a commercial software mainly developed for finite element applications. It has been integrated in various finite element programs. This program is strictly boundary constrained, that is, it does not modify the input domain boundary mesh. It generates a quality tetrahedral mesh for the interior of the mesh domain. In our tests, we used the version 4.2, released in 2010.

We selected three inputs for comparison. They were obtained from the AIM@SHAPE Repository http://shapes.aim-at-shape.net. Two of them (Figure 19), one (rocker-arm) represents a smooth surface and one (couplingdown) contains sharp features, were used to comparing with CGALmesh. The other one (Figure 20), which has a relatively good quality surface mesh, was used for comparison with Tetmesh-GHS3D.

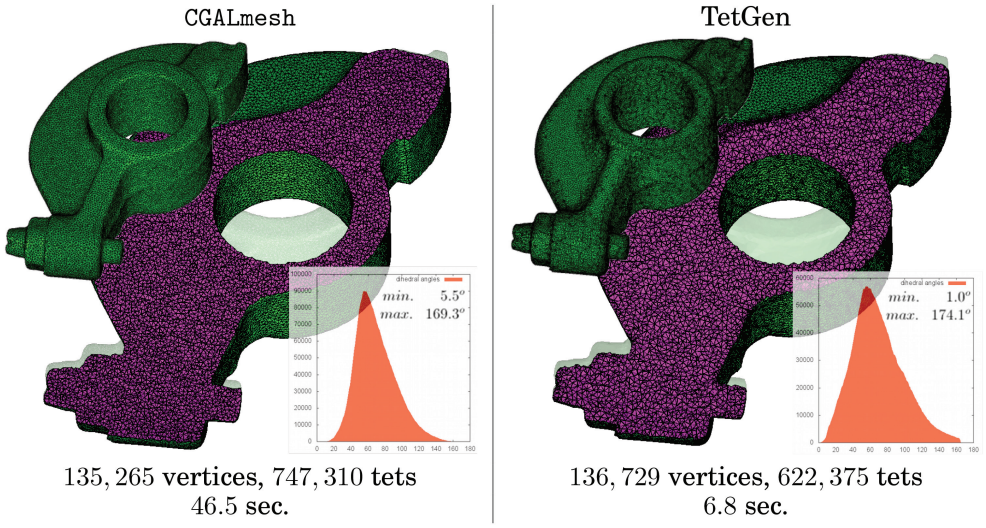


Fig. 21. The output meshes of `rocker-arm` generated by `CGALmeshg` (left) and TetGen (right). The command line of TetGen was: `-p178qVa2e-7`. The parameters and mesh criteria of *CGALmesh* were: `angle_bound = 120`, `cell_radius_edge_ratio = 3`, facet_angle = 25, `facet_distance = 0.005`, `cell_size = 0.00625`, `face_size = 0.00625`, `edge_size = 0.003125`.

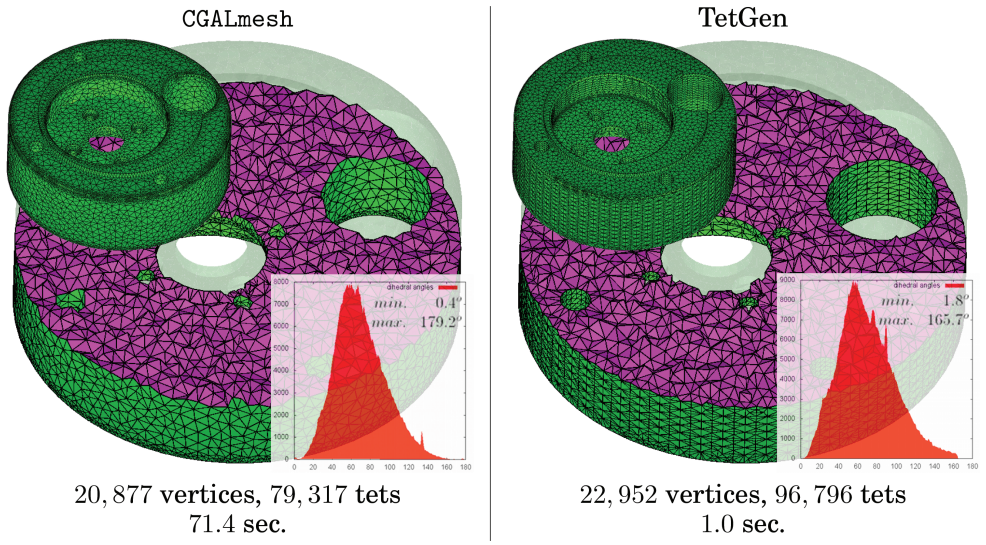


Fig. 22. The output meshes of `couplingdown` generated by `CGALmeshg` (left) and TetGen (right). The command line of TetGen was: `-p178qVa7e-6`. The parameters and mesh criteria of *CGALmesh* were: `angle_bound = 30`, `cell_radius_edge_ratio = 3`, facet_angle = 25, `facet_distance = 0.005`, `cell_size = 0.025`, `face_size = 0.025`, `edge_size = 0.0125`.

Recall that `CGALmesh` is based on Delaunay refinement, and it does surface remeshing. We used the combination of CDT boundary conformity and constrained Delaunay refinement algorithms of TetGen in this comparison. It is reported in Figure 21 and Figure 22. The results show that both `CGALmesh` and TetGen generated similar tetrahedral meshes in the interior of the domain. In case of a smooth surface (Figure 21), the advantage of surface remeshing is seen. The tetrahedral mesh quality generated

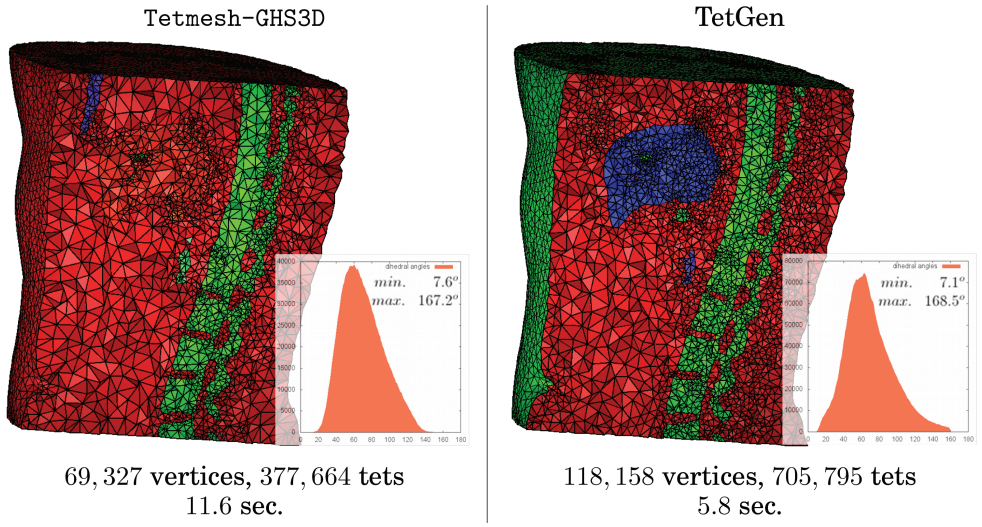


Fig. 23. The output meshes of `patient-organs02` generated by `Tetmesh-GHS3D` (left) and TetGen (right). Both meshes preserved the original input surface mesh. The command line of TetGen was: `-pq1.1VYA -o/160`. The default mesh quality options of `Tetmesh-GHS3D` were applied, and the "-c 0" option was used to let `GHS3D` mesh all components.

by `CGALmesh` was better, and the remeshed surface was also nicer than TetGen's. However, in case there are sharp features to be preserved, like the example `couplingdown` (Figure 22), `CGALmesh` did not result in a tetrahedral mesh with no skinny tetrahedra. The quality of TetGen's mesh was slightly better in this example. In both cases, the meshing times of TetGen were much faster than `CGALmesh`'s.

Figure 23 shows the comparison of TetGen with `Tetmesh-GHS3D`. In this example, we used the combination of preserving input boundary mesh and a modified version of constrained Delaunay refinement such that it only inserts interior Steiner points. We observed that the minimum and maximum dihedral angle bounds in their generated meshes were similar. TetGen was slightly faster in generating this mesh. However, the overall mesh quality in the mesh generated by `Tetmesh-GHS3D` was better. We believe that the mesh optimization process in `Tetmesh-GHS3D` is very effective.

Remark. The compared codes have a handful of options for quality mesh generation and optimization. We only tested some basic and default options of these codes, hence these comparisons are far from comprehensive.

11. SUMMARY AND OUTLOOK

In this article, the tetrahedral meshing problems encountered by TetGen are introduced and an overview of the algorithms and techniques used by TetGen is presented. Practical experiments showed that TetGen is robust and is able to generate isotropic tetrahedral meshes with high quality efficiently.

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 are worth investigating 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 show 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 nontrivial 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 the fundamental mathematical problems will lead to 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 wishes 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. The author sincerely thanks the anonymous referees for their corrections and useful comments that greatly improved this article.

REFERENCES

P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun. 2005. Variational tetrahedral meshing. *ACM Trans. Graph.* 24, 3, 617–625.

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

F. Aurenhammer. 1991. Voronoi diagrams – A study of fundamental geometric data structures. *ACM Comput. Surv.* 23, 345–405.

T. J. Baker. 1989. Automatic mesh generation for complex three-dimensional regions using a constrained delaunay triangulation. *Eng. Comput.* 5, 161–175.

C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. 1996. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.* 22, 4, 469–483.

M. W. Bern and D. Eppstein. 1995. Mesh generation and optimal triangulations. In *Computing in Euclidean Geometry* (2nd Ed.), D.-Z. Du and F. K.-M. Hwang (Eds.), Number 4 in Lecture Notes Series on Computing. World Scientific, 47–123.

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

J.-D. Boissonnat, O. Devillers, and S. Hornus. 2009. Incremental construction of the delaunay triangulation and the Delaunay graph in medium dimension. In *Proceedings of the 25th Annual Symposium on Computational Geometry*. ACM, 208–216.

H. Borouchaki, P. L. George, and S. H. Lo. 1996. Optimal Delaunay point insertion. *Int. J. Numer. Methods Engrg.* 39, 3407–3437.

A. Bowyer. 1987. Computing Dirichlet tessellations. *Comput. J.* 24, 2, 162–166.

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

C. Burnikel, S. Funke, and M. Seel. 2001. Exact geometric computations using cascading. *Internat. J. Comput. Geom. Appl.* 11, 245–266.

B. Chazelle. 1984. Convex partition of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. Comput.* 13, 3, 488–507.

B. Chazelle and L. Palios. 1990. Triangulating a non-convex polytope. *Disc. Computat. Geom.* 5, 3, 505–526.

J. Chen, D. Zhao, Z. Huang, Y. Zheng, and S. Gao. 2011. Three-dimensional constrained boundary recovery with an enhanced steiner point suppression procedure. *Comput. Struct.* 89, 5–6, 455–466.

L. Chen and J.-C. Xu. 2004. Optimal Delaunay triangulations. *J. Comput. Math.* 22, 2, 299–308.

S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. 2000. Sliver exudation. *J. ACM* 47, 883–904.

S.-W. Cheng, T. K. Dey, and J. A. Levine. 2007. A practical Delaunay meshing algorithm for a large class of domains. In *Proceedings of the 16th International Meshing Roundtable*. Sandia National Laboratories. Springer, 477–494.

S.-W. Cheng, T. K. Dey, E. A. Ramos, and T. Ray. 2005. Quality meshing for polyhedra with small angles. *Int. J. Comput. Geom. Appl.* 15, 421–461.

L. P. Chew. 1989a. Constrained Delaunay triangulation. *Algorithmica* 4, 97–108.

L. P. Chew. 1989b. Guaranteed-quality triangular meshes. Tech. Rep. TR 89-983. Department of Computer Science, Cornell University.

K. L. Clarkson and P. W. Shor. 1989. Applications of random sampling in computational geometry, II. *Disc. Computat. Geom.* 4, 387–421.

D. Cohen-Steiner, É. C. de Verdière, and M. Yvinec. 2004. Conforming Delaunay triangulations in 3D. *Comput. Geom. Theory Appl.* 28, 2–3, 217–233.

B. N. Delaunay. 1934. Sur la sphère vide. *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk* 7, 793–800.

O. Devillers and S. Pion. 2003. Efficient exact geometric predicates for Delaunay triangulations. In *Proceedings of the 5th Workshop on Algorithm Engineering and Experiments*. SIAM, 37–44.

O. Devillers, S. Pion, and M. Teillaud. 2002. Walking in triangulation. *Int. J. Found. Comput. Sci.* 13, 2, 181–199.

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

Q. Du, V. Faber, and M. Gunzburger. 1999. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Rev.* 41, 4, 637–676.

Q. Du and D. Wang. 2003. Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations. *Int. J. Numer. Meth. Eng.* 56, 1355–1373.

Q. Du and D. Wang. 2004. Constrained boundary recovery for the three dimensional Delaunay triangulations. *Int. J. Numer. Methods Eng.* 61, 1471–1500.

H. Edelsbrunner. 2001. *Geometry and Topology for Mesh Generation*. Cambridge University Press, Cambridge, UK.

H. Edelsbrunner and E. P. Mücke. 1990. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithm. *ACM Trans. Graph.* 9, 1, 66–104.

H. Edelsbrunner and N. R. Shah. 1996. Incremental topological flipping works for regular triangulations. *Algorithmica* 15, 223–241.

H. Edelsbrunner and N. R. Shah. 1997. Triangulating topological spaces. *Int. J. Computat. Geom. Appl.* 7, 4, 365–378.

S. Fortune and C. J. Van Wyk. 1996. Static analysis yield efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.* 15, 3, 223–248.

L. A. Freitag and C. Ollivier-Gooch. 1997. Tetrahedral mesh improvement using swapping and smoothing. *Int. J. Numer. Methods Eng.* 40, 21, 3979–4002.

P. J. Frey and P. L. George. 2000. *Mesh Generation - Application to Finite Elements* (1st Ed.). Hermes Science Publishing, Oxford, UK, 814 pages. ISBN 1-903398-00-2.

R. V. Garimella. 2002. Mesh data structure selection for mesh generation and FEA applications. *Int. J. Numer. Methods Eng.* 55, 4 (Oct. 2002), 451–478.

P. L. George and H. Borouchaki. 2003. Back to edge flips in 3 dimensions. In *Proceedings of the 12th International Meshing Roundtable*. Sandia National Laboratories, 393–402.

P. L. George, H. Borouchaki, and E. Saltel. 2003. Ultimate robustness in meshing an arbitrary polyhedron. *Int. J. Numer. Methods Eng.* 58, 1061–1089.

P. L. George, F. Hecht, and E. Saltel. 1991. Automatic mesh generator with specified boundary. *Comput. Methods Appl. Mech. Eng.* 92, 269–288.

L. Guibas and J. Stolfi. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.* 4, 4, 75–123.

W. Huang. 2005. Measuring mesh qualities and application to variational mesh adaption. *SIAM J. Sci. Comput.* 26, 1643–1666.

B. Hudson. 2007. Dynamic mesh refinement. Ph.D. dissertation, CMU-CS-07-162. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

C. Jamin, P. Alliez, M. Yvinec, and J.-D. Boissonnat. 2013. CGALmesh: A generic framework for Delaunay mesh generation. Tech. Rep. 8256. INRIA.

B. Joe. 1995. Construction of three-dimensional improved-quality triangulations using local transformations. *SIAM J. Sci. Comput.* 16, 6, 1292–1307.

B. M. Klinger and J. R. Shewchuk. 2007. Aggressive tetrahedral mesh improvement. In *Proceedings of the 16th International Meshing Roundtable*. Springer, 3–23.

M. Kremer, D. Bommes, and L. Kobbelt. 2012. OpenVolumeMesh - A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*. Springer, 531–548.

C. L. Lawson. 1977. Software for C^1 Surface Interpolation. In *Mathematical Software III*, Academic Press, New York, 164–191.

D. T. Lee and A. K. Lin. 1986. Generalized Delaunay triangulations for planar graphs. *Disc. Computat. Geom.* 1, 201–217.

X.-Y. Li and S.-H. Teng. 2001. Generating well-shaped Delaunay meshes in 3D. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*. SIAM, 28–37.

A. Liu and M. Baida. 2000. How far flipping can go towards 3D conforming/constrained triangulation. In *Proceedings of the 9th International Meshing Roundtable*. Sandia National Laboratories, 307–315.

Y. Liu and J. Snoeyink. 2005. A comparsion of five implementations of 3D Delaunay tessellation. In *Combinatorial and Computational Geometry*, vol. 52, J. E. Goodman, J. Pach, and E. Welzl (Eds.), MSRI Publications, New York, 439–458.

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

J.-M. Mirebeau. 2012. Optimally adapted meshes for finite elements of arbitrary order and $W^{1,p}$ norms. *Numer. Math.* 120, 271–305.

S. A. Mitchell and S. A. Vavasis. 2000. Quality mesh generation in higher dimensions. *SIAM J. Comput.* 29, 4, 1334–1370.

E. P. Mücke. 1993. Shapes and implementations in three-dimensions geometry. Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

M. Murphy, D. M. Mount, and C. W. Gable. 2000. A point-placement strategy for conforming Delaunay tetrahedralizations. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. Sandia National Laboratories, 69–93.

A. Nanveski, G. Blelloch, and R. Harper. 2003. Automatic Generation of staged geometric predicates. *High. Ord. Symb. Computat.* 16, 4, 379–400.

C. Ollivier-Gooch. 2005. GRUMMP – Generation and Refinement of Unstructed, Mixed-Element Meshes in Parallel. http://tetra.mech.ubc.ca/GRUMMP/. (2005). (Last accessed November 2009.)

S. Oudot, L. Rineau, and M. Yvinec. 2005. Meshing volumes bounded by smooth surfaces. In *Proceedings of the 14th International Meshing Roundtable*. Sandia National Laboratories, Springer, 203–220.

S. J. Owen. 1998. A survey of unstructed mesh generation technology. In *Proceedings of the 7th International Meshing Roundtable*. Sandia National Laboratories, 239–267.

S. E. Pav and N. Walkington. 2004. Robust three dimensional Delaunay refinement. In *Proceedings of the 13th International Meshing Roundtable*. Springer, 145–156.

D. M. Priest. 1991. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*. IEEE, 132–143.

J. Radon. 1921. Mengen Konvexer Körper, die Einen Gemeinschaftlichen Punkt Enthalten. *Math. Ann.* 83, 113–115.

V. T. Rajan. 1994. Optimality of the Delaunay triangulation in $\mathbb{R}^d$. *Disc. Computat. Geom.* 12, 189–202.

A. Rand and N. Walkington. 2009. Collars and intestines: Practical conforming Delaunay refinement. In *Proceedings of the 18th International Meshing Roundtable*. Springer, 481–497.

M.-C. Rivara. 1997. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *Int. J. Numer. Methods Eng.* 40, 3313–3324.

J. Ruppert. 1995. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algor.* 18, 3, 548–585.

J. Ruppert and R. Seidel. 1992. On the difficulty of triangulating three-dimensional nonconvex polyhedra. *Disc. Computat. Geom.* 7, 227–253.

J. Schöberl. 1997. NETGEN, An advancing front 2D/3D-mesh generator based on abstract rules. *Comput. Visual. Sci.* 1, 41–52.

E. Schönhardt. 1928. Über die Zerlegung von Dreieckspolyedern in Tetraeder. *Math. Ann.* 98, 309–312.

J. R. Shewchuk. 1996a. Robust adaptive floating-point geometric predicates. In *Proceedings of the 12th Annual Symposium on Computational Geometry*. ACM, 141–150.

J. R. Shewchuk. 1996b. `Triangle`: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha (Eds.). Lecture Notes in Computer Science, vol. 1148, Springer, Berlin Heidelberg, 203–222. http://www.cs.cmu.edu/~quake/triangle.html.

J. R. Shewchuk. 1998a. A condition guaranteeing the existence of higher-dimensional constrained Delaunay triangulations. In *Proceedings of the 14th Annual Symposium on Computational Geometry*. ACM, 76–85.

J. R. Shewchuk. 1998b. Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the 14th Annual Symposium on Computational Geometry*. ACM, 86–95.

J. R. Shewchuk. 2002a. Constrained Delaunay tetrahedralizations and provably good boundary recovery. In *Proceedings of the 11th International Meshing Roundtable*. Springer, 193–204.

J. R. Shewchuk. 2002b. Two discrete optimization algorithms for the topological improvement of tetrahedral meshes. www.cs.berkeley.edu/∼jrs/papers/edge.pdf.

J. R. Shewchuk. 2002c. What is a good linear element? Interpolation, Conditioning, and quality measures. In *Proceedings of 11th International Meshing Roundtable*. Springer, 115–126.

J. R. Shewchuk. 2003. Updating and constructing constrained Delaunay and constrained regular triangulations by flips. In *Proceedings of the 19th Annual Symposium on Computational Geometry*. ACM, 86–95.

J. R. Shewchuk. 2008. General-dimensional constrained Delaunay and constrained regular triangulations, I: Combinatorial properties. *Disc. Computat. Geom.* 39, 580–637.

J. R. Shewchuk and H. Si. 2014. Higher-quality tetrahedral mesh generation for domains with small angles by constrained Delaunay refinement. In *Proceedings of the 30th Annual Symposium on Computational Geometry*. ACM.

H. Si. 2008. Three dimensional boundary conforming Delaunay mesh generation. Ph.D. Dissertation, Institut für Mathematik, Technische Universität Berlin, Strasse des 17. Juni 136, D-10623, Berlin, Germany. http://opus.kobv.de/tuberlin/volltexte/2008/1966/.

H. Si. 2009. An analysis of Shewchuk's Delaunay refinement algorithm. In *Proceedings of the 18th International Meshing Roundtable*. Springer, UT, 499–517.

H. Si. 2010. Constrained Delaunay tetrahedral mesh generation and refinement. *Finite Elem. Anal. Des.* 46, 1, 33–46.

H. Si. 2013. TetGen, A quality tetrahedral mesh generator and a 3D Delaunay triangulator, version 1.5, user's manual. Tech. Rep. 13. Weierstrass Institute for Applied Analysis and Stochastics (WIAS).

H. Si, J. Fuhrmann, and K. Gärtner. 2010. Boundary conforming Delaunay mesh generation. *Comput. Math. Math. Phys.* 50, 1, 38–53.

H. Si and K. Gärtner. 2005. Meshing piecewise linear complexes by constrained Delaunay tetrahedralizations. In *Proceedings of the 14th International Meshing Roundtable*. Springer, CA, 147–163.

H. Si and K. Gärtner. 2011. 3D Boundary recovery by constrained Delaunay tetrahedralization. *Int. J. Numer. Methods Eng.* 85, 1341–1364.

H. Si and J. R. Shewchuk. 2014. Incrementally constructing and updating constrained Delaunay tetrahedralizations with finite-precision coordinates. *Eng. Comput.* 30, 2, 253–269.

TetMesh-GHS3D. 2010. A powerful isotropic tet-mesher, Version 4.2. http://www-roc.inria.fr/gamma/gamma/ghs3d/ghs.php.

J. F. Thompson, B. K. Soni, and N. P. Weatherill (Eds.). 1999. *Handbook of Grid Generation*. CRC Press, Boca Raton, FL.

D. F. Watson. 1987. Computing the *n*-dimensional Delaunay tessellations with application to Voronoi polytopes. *Comput. J.* 24, 2, 167–172.

N. P. Weatherill and O. Hassan. 1994. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *Internat. J. Numer. Methods Eng.* 37, 2005–2039.

C.-K. Yap. 1997. Towards exact geometric computation. *Comput. Geom.* 7, 1, 3–23.

Received March 2013; revised December 2013 and April 2014; accepted May 2014