Weierstraß-Institut
für Angewandte Analysis und Stochastik

im Forschungsverbund Berlin e.V.

Technical Report ISSN 1618 – 7776

TetGen
A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator
Version 1.3 User's Manual

Hang Si[1]

submitted: August 31th 2004

[1] Weierstrass Institute for Applied Analysis and Stochastics
Mohrenstrasse 39, D - 10117 Berlin
email: si@wias-berlin.de

No. 9
Berlin 2004

2000 *Mathematics Subject Classification.* 65N50.

Key words and phrases. tetrahedral mesh, quality mesh generation, boundary mesh generation, Delaunay tetrahedralization, constrained Delaunay tetrahedralization.

Edited by
Weierstraß-Institut für Angewandte Analysis und Stochastik (WIAS)
Mohrenstraße 39
10117 Berlin
Germany

Fax: + 49 30 2044975
E-Mail: preprint@wias-berlin.de
World Wide Web: http://www.wias-berlin.de/

Abstract

TetGen is a quality tetrahedral mesh generator and 3-dimensional Delaunay triangulator. Based on the the-state-of-the-art algorithms for Delaunay tetrahedralization and mesh generation, this program has been specifically designed to fulfill the task of automatically generating high quality tetrahedral meshes, which are suitable for scientific computing using numerical methods such as finite element and finite volume methods.

The purpose of this document is to give a brief explanation of the problems solved by TetGen and to provide a detailed user's documentation. In this document, the user will learn how to create tetrahedral meshes using TetGen's input files and command line switches. In the following section, the programming interface of TetGen for calling TetGen from another program is explained. Various examples are given to simplify the "getting started".

keywords: tetrahedral mesh, quality mesh generation, boundary mesh generation, Delaunay tetrahedralization, constrained Delaunay tetrahedralization

Contents

1 Introduction

TetGen is a program for generating tetrahedral meshes for arbitrary three-dimensional domains. The main purpose of TetGen is to create high-quality tetrahedral meshes for solving partial differential equations using finite element and finite volume methods. This program, adopting Delaunay methods, currently generates meshes including exact constrained Delaunay tetrahedralizations and quality (conforming Delaunay) meshes. The latter are nicely graded and the tetrahedra have circumradius-to-shortest-edge ratio bounded, hence are satisfying the requirements known from numerical analysis. For a three-dimensional point set, it generates its exact Delaunay tetrahedralization and convex hull as well.

This program, written in C++, includes a suit of state-of-the-art algorithms for Delaunay tetrahedralization and quality mesh generation. It can be compiled into an executable program or a library which can be integrated into other applications. The code is highly portable and has been successfully compiled and tested on all major operating systems, e.g. Unix/Linux, MacOS, Windows, etc.

The remainder of this section is to give a brief description of the problems that TetGen solves and the specific algorithms implemented in TetGen. References are given for people who are particularly interesting in these approaches. However, this information is not really crucial for all users. Most of the sections can be skipped but Section 1.2.1 and Section 1.3.1, which contain some important points to get a solvable problem.

1.1 Delaunay Tetrahedralization and Convex Hull

The Delaunay triangulation of a vertex set, introduced by Delaunay [1] in 1934, has many favorite properties which make it be a very useful geometric structure. It has been used extensively both in the design of efficient algorithms and in practical applications.

The language of combinatorial topology will be used to describe Delaunay triangulations. Understanding of some basic notions is necessary, such as convex hull, simplex, simplical complex and so on. A brief explanation of these notions can be found in Appendix A.

Let V be a set of n-dimensional vertices. s be a k-simplex ($0 \leq k \leq n$) formed from vertices of V. The *circumsphere* of s is a full dimensional sphere that passes through all vertices of s, hence it is a circle when $n = 2$ and a sphere when $n = 3$. If $k = d$, s has a unique circumsphere, otherwise, there are infinitely many circumspheres of s. The simplex s is *Delaunay* if there exists a circumsphere of s such that no vertex of V lies inside it. Figure 1 (a) shows Delaunay simplices in a set of two-dimensional vertices.

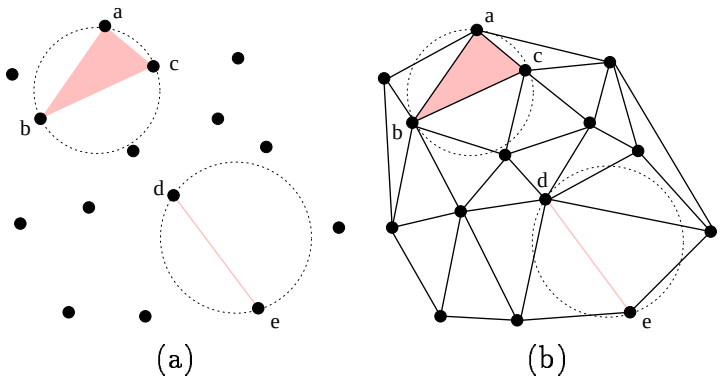

Figure 1: The Delaunay criterion and Delaunay triangulation in two dimensions. **(a)** Both the 2-simplex *abc* and the 1-simplex *de* are Delaunay. **(b)** The corresponding Delaunay triangulation of the point set shown in (a).

The *Delaunay triangulation* $\mathcal{D}$ of V is a simplical complex consisting of Delaunay simplices, and the set of all simplices of $\mathcal{D}$ covers the convex hull of V. A two-dimensional Delaunay triangulation is illustrated in Figure 1 (b). In three dimensions, it is also called *Delaunay tetrahedralization*.

The Delaunay triangulation has many optimal properties. For example, among all triangulations of a set of points in $\mathbf{R}^2$, it maximizes the minimum angle, and also minimizes the maximum circumradii; optimal $O(n\ log\ n)$ time divide-and-conquer and plane-sweep algorithms are known to efficient construct two-dimensional Delaunay triangulations. A discussion on some optimal properties of the Delaunay triangulation in three or higher dimensions can be found in [9]. In the following, we introduce two properties of the Delaunay triangulations which are both useful in numerical methods and the design of efficient algorithms.

A favorite property of the Delaunay triangulation is its relation with another useful geometric structure: the Voronoi diagram defined on the same vertex set. For any vertex $a \in V$, the *Voronoi cell* of a is the set of points with distance to a not greater than to any other vertex of V, i.e. it is the set

$$\{x \in \mathbf{R}^n \mid |x - a| \leq |x - b|, \forall b \in V\}.$$

where $|\cdot|$ stands for the Euclidean distance. The *Voronoi diagram* of V is a subdivision of space $\mathbf{R}^n$ into Voronoi cells (some of which may be infinite). Delaunay triangulation and Voronoi diagram are geometrical dual. For example, in two dimensions, Voronoi polygons are corresponding to Delaunay vertices, Voronoi edges are corresponding to Delaunay edges, and Voronoi vertices are corresponding to Delaunay triangles, as illustrated in Figure 2 (a).

Another useful property is the localization of the Delaunay property. Let $\mathcal{T}$ be an arbitrary triangulation of V, and s be a simplex of $\mathcal{T}$. Let $\mathcal{K}$ be a subcomplex of $\mathcal{T}$ formed by simplices containing s. s is *locally Delaunay* if there exists a circumsphere of s encloses no vertices from the vertex set of $\mathcal{K}$ in its interior. Figure 2 (b) illustrates

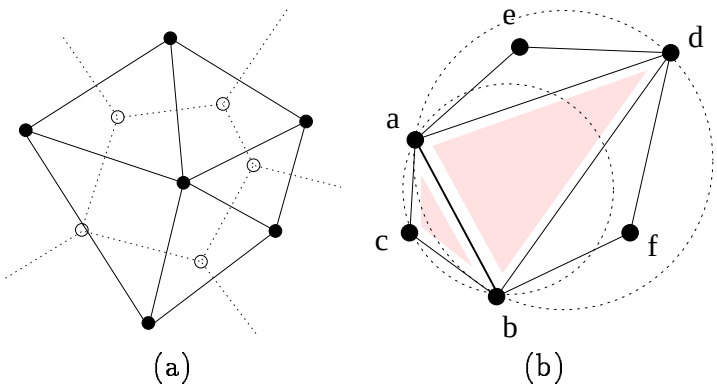

Figure 2: Properties of Delaunay triangulations. **(a)** The relation between Delaunay triangulation and Voronoi diagram. **(b)** Locally Delaunay property. Edge ab is locally Delaunay. Here only c and d affect the property because there are triangles abc and abd sharing edge ab. e and f are excluded from the definition of the property.

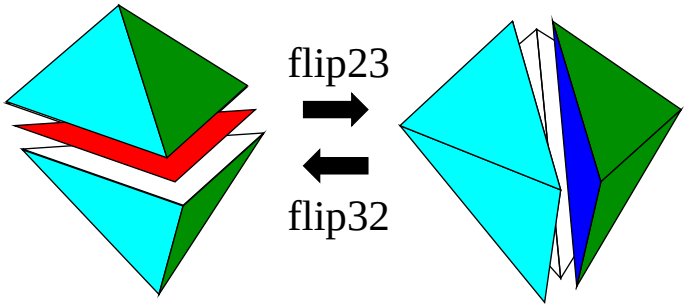

Figure 3: Two flip operations in three dimensions. A 2-to-3 flip replaces a non-locally Delaunay face (the red face on the left figure) into three locally Delaunay faces (the blue and white faces on the right figure); while a 3-to-2 flip does the inverse.

the property in two dimensions. Evidently, if every simplex of $\mathcal{T}$ is locally Delaunay, then $\mathcal{T}$ is Delaunay triangulation.

Many algorithms for constructing Delaunay triangulations use "flips". A *flip* is an operation to transform a set of non-locally Delaunay simplices into another set of simplices which are locally Delaunay. Figure 3 illustrates two types of flips in three dimensions. The basic idea of these algorithms is relatively simple: start from an arbitrary triangulation, do flips on non-locally Delaunay simplices, stop when all simplices are locally Delaunay, the result is a Delaunay triangulation due to the fact mentioned above. Lawson [2] first used a flip algorithm to construct two-dimensional Delaunay triangulations. However, in dimension higher than two, such simple scheme may not terminate (see the discussion in [3]). Nevertheless, it has been proved [3] that Delaunay tetrahedralizations can be constructed by incremental insertion of points and flips.

For a three-dimensional point set (number of points ≥ 4), TetGen generates its exact Delaunay tetrahedralization and convex hull. The convex hull is represented in a set of triangular faces. The algorithm TetGen uses is from Edelsbrunner and Shah [10]. It is incremental and adds points in a sequence of flips. The algorithm is simple and easy to implement, and performs well in practice.

Our implementation is robust and fast. On a 3.06GHz, Intel Computer, TetGen computes the Delaunay tetrahedralization of 40,000 randomly distributed points in 4.8 seconds. It used 3 minutes to compute the Delaunay tetrahedralization for 1 million points. The robustness is achieved by the use of the adaptive exact arithmetic [13] code for performing two geometric predications, i.e., orientation and insphere tests.

1.2 Constrained Delaunay Tetrahedralization (CDT)

The problem here is to decompose a three-dimensional geometric object Ω into tetrahedra, where Ω may be arbitrarily complicated in shape, may contain internal boundaries (for separating different regions), and holes. This problem is also referred as *boundary mesh generation* or *boundary conformity*.

A *constrained Delaunay tetrahedralization* (CDT) is a variation of a Delaunay tetrahedralization that is constrained to respect the boundary of Ω. CDTs maintain many properties of Delaunay tetrahedralizations [4, 16]. They are nice structures for solving this problem.

For simplifying the design of an algorithm, one can assume the boundary Γ of Ω is a three-dimensional polyhedron, i.e., Γ is the underlying space of a two-dimensional simplical complex. More specifically, Γ is a set $V \subset \mathbf{R}^3$ of vertices, and a set of polygons. Each polygon is a segment-bounded constraining facet. TetGen uses a more general representation of Γ which will be introduced separatly in Section 1.2.1.

The visibility between two vertices p and q of V is *occluded* if there is a constraining polygon f such that p and q lie on opposite sides of the plane that includes f, and the line segment pq intersects this polygon (see Figure 4). A tetrahedron t formed by vertices of V is *constrained Delaunay* if its circumsphere encloses no vertex of V, which is visible from any point in the relative interior of t (see Figure 4).

A tetrahedralization T of Γ is called *constrained tetrahedralization* if T and Γ have exactly the same vertex set, and each polygon of Γ is completely represented by a union of triangular faces of T.

T is a *constrained Delaunay tetrahedralization* of Γ if it is a constrained tetrahedralization and every tetrahedron of T is constrained Delaunay.

Intuitively, the definitions of Delaunay tetrahedralization and constrained Delaunay tetrahedralization are the same except that, for the CDT, we ignore the volume of a sphere whenever the sphere passes through a constraining polygon. Note that simplices (tetrahedra, triangles, and edges) in a CDT are not always Delaunay.

However to construct CDTs is not so straightforward. One obstacle is: there are

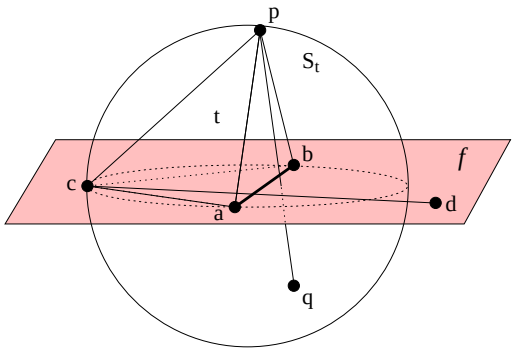

Figure 4: Visibility and constrained Delaunay tetrahedron. The shaded region represents a constraining polygon f of Γ including vertices a, b, c, and d. Vertices p and q lie on opposite sides of f, they can not see each other. While c and d can see each other even if ab is a segment of f. Here t is constrained Delaunay. S_t is the circumsphere of tetrahedron t ($abcp$) and it encloses q but not d.

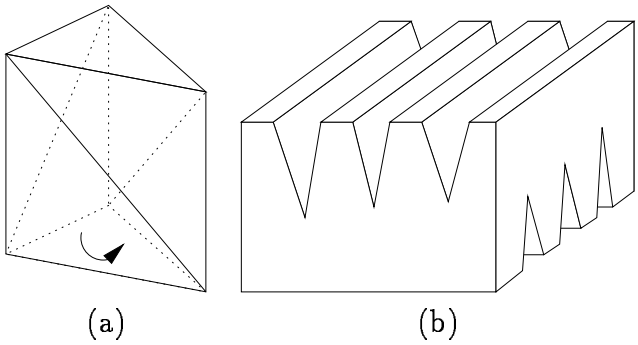

Figure 5: Two polyhedrons which can not be tetrahedralized. **(a)** The Schönhardt polyhedron, which is formed by rotating one end of a triangular prism, resulting three non-convex diagonal edges. **(b)** The Chazelle's polyhedron, which can be built out of a cube by cutting deep wedges.

polyhedra having no tetrahedralization at all. Two examples are shown in Figure 5. Furthermore, Ruppert and Seidel [7] showed that it is NP-complete to decide whether a simple polyhedron can be tetrahedralized or not. Nevertheless, every polyhedron is tetrahedralizable with inserting additional points into it. Due to these facts, algorithms for constructing CDTs have to resort to inserting additional vertices. A key question, when such additional points are necessary to ensure the existence, is to decide what is the optimal (minimal) number of additional points. Another concern, mainly for mesh refinement algorithms, is to avoid very short edges, which engender very small tetrahedra, hence the number of additional points can be undesirably large. Various approaches [16, 17, 18, 19] based on different point insertion schemes in order to construct CDTs have been proposed in literatures.

Shewchuk gave a condition [14] that guarantees the existence of CDTs, which is

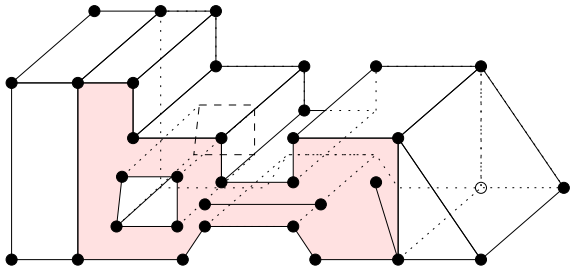

Figure 6: A piecewise linear complex, which is a set of vertices, and a set of segments and facets. The shaded area shows a facet which is non-convex, has a hole, segments and vertices in it.

useful for designing algorithms to construct CDTs. Because each polygon of Γ is segment-bounded. One considers the set of all segments of Γ. A segment s is Delaunay if there exists a circumsphere S of s, such that no vertices of Γ lies inside S. Furthermore, s is *strongly Delaunay* if no vertices of Γ lies inside or on S. The condition is if all segments of Γ are strongly Delaunay, then the CDT of Γ exists.

The algorithm that TetGen implemented is described in Si and Gärtner [20]. For a given Γ, this algorithm follows the usual routes of the CDT construction: at first form the Delaunay tetrahedralization of the vertices of Γ, recover the missing segments and facets in the following phase. Specifically, it uses the local geometric information and only inserts additional points on some segments of Γ, updates Γ into Γ' for which the existence of a CDT can be guaranteed. The result is a CDT of the updated Γ. This algorithm usually requires fewer additional points than other similar algorithms and it always keeps the resulting subsegments as long as possible.

TetGen's implementation of this algorithm is both robust and efficient. It not only handles reasonably defined geometric objects but also inputs which are rather badly discretized.

1.2.1 Piecewise Linear Complex (PLC)

Three-dimensional geometric objects are often more complicated than polyhedra, TetGen uses a more general input called *piecewise linear complex* (PLC), defined by Miller, Talmor, Teng, Walkington, and Wang [12]. A PLC X is a set of vertices, segments and *facets* (see Figure 6). Each facet is a polygonal region, it may have any number of sides and may be non-convex, possibly with holes, segments and vertices in it. A facet can represent any *planar straight line graph* (PSLG), which is a popular input model used by many two-dimensional mesh algorithms. A facet is actually a PSLG embedded in three dimensions. An example is given in Figure 6, the shaded area highlights a facet.

PLCs have restrictions like any other complex. For a PLC X, the elements of X must be closed under intersection. For example, two segments only can intersect

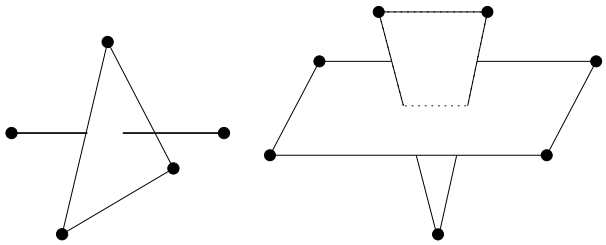

Figure 7: Some invalid PLCs. Left: one vertex is missing; right: two vertices and one segment are missing.

at a common vertex which is also in X. Two facets of X may intersect only at a shared segment or vertex or a union of shared segments and vertices (because facets are non-convex). Figure 7 shows non-closed configurations for examples.

Another restriction of the facets of PLCs is that the point set which used to define a facet must be coplanar.

Any polyhedron is a PLC. Furthermore, PLCs are more flexible than polyhedra to represent three-dimensional geometric objects. For example, the shaded facet in Figure 6 can not be represented by any polygon. For domains having curved surfaces (which are not piecewise linear), the surface triangulations are previously required, hence, PLCs can approximately represent any three-dimensional domain.

1.3 Quality Tetrahedral Mesh

The main goal of TetGen is to create tetrahedral meshes that have special properties for solving *partial differential equations* (PDEs) by *finite element methods* (FEM) and *finite volume methods* (FVM). These methods have been widely applied in contemporary engineering applications for simulating physical phenomena, such as mechanical deformation, heat transfer, electromagnetic problems, etc.

The problem is to generate a tetrahedral mesh conforming to a given (polyhedral or piecewise linear) domain in three dimensions together with certain constraints for the size and shape of the mesh elements. It is a typical problem of *provably good mesh generation* or *quality mesh generation*. The techniques of quality mesh generation provide the "shape" and "size" guarantees on the meshes:

- all elements have a certain quality measure bounded, and
- the number of elements is limited by a constant factor with respect to the number of optimal.

Meshes having the above the two properties are called *quality meshes*. They are desired as meshes for the FEM and FVM, in which the running time generally

increases with the number of elements, and where the convergence and stability may be hurt by very badly-shaped elements.

One approach for solving this problem, referred as *Delaunay refinement*. It is maintaining a Delaunay tetrahedralization, which is refined by insertion of additional vertices. The placement of these vertices is chosen to enforce boundary conformity and to improve the quality of the mesh. Delaunay refinement was successfully applied to the corresponding two-dimensional problem. Such algorithms can be found in the work of Chew [5, 6], and Ruppert [8]. However, these algorithms do not generalize to three dimensions because of the problem of slivers, which are very flat and nearly degenerate tetrahedra.

The algorithm TetGen currently uses to tackle this problem is a Delaunay refinement algorithm from Shewchuk [15]. It is a smooth generalization of Ruppert's algorithm to three dimensions. Given a complex of vertices, constraining segments and facets in three dimensions, with no input angle less than 90°, this algorithm can generate a quality mesh of Delaunay tetrahedra with radius-edge ratios (see section 1.3.1) not greater than 2.0. The size of the tetrahedra can grade from small to large over a short distance. The advantages of this algorithm are its simplicity. The implementation of TetGen shows that the algorithm generates meshes generally surpassing the theoretical bounds and is effective in eliminating tetrahedra with small or large dihedral angles.

1.3.1 A Quality Measure: Radius-Edge Ratio

There are several quality measures available in literature. This section explains the quality measure used in TetGen due to the algorithm [15] it implements.

For good accuracy bounds in FEM, it is necessary that the shapes of elements have bounded aspect ratio. The *aspect ratio* of an element is defined as its maximum side length divided by its minimum altitude. For a good quality mesh, this value should as small as possible. For example, "thin and flat" tetrahedra tend to have large aspect ratio.

A similar but weaker quality measure is radius-edge ratio, proposed by Miller, Talmor, Teng, Walkington, and Wang [11]. A tetrahedron t has a unique circumsphere. Let $R = R(t)$ be that radius and $L = L(t)$ the length of the shortest edge. The *radius-edge ratio* $Q = Q(t)$ of the tetrahedron is measured by taking the ratio, that is:

$$Q = \frac{R}{L}$$

The radius-edge ratio is effective for measuring the quality of a tetrahedron. For all well-shaped tetrahedra, this value is small (see Figure 8), while for most of badly-shaped tetrahedra, this value is large (see Figure 9). Hence, for a good quality mesh, this value should be bounded as small as possible. However, the ratio can not be arbitrarily small, it is minimized by the regular tetrahedron (in which case the

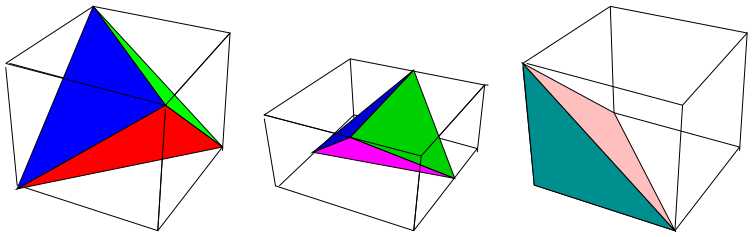

Figure 8: The radius-edge ratios of some well-shaped tetrahedra.

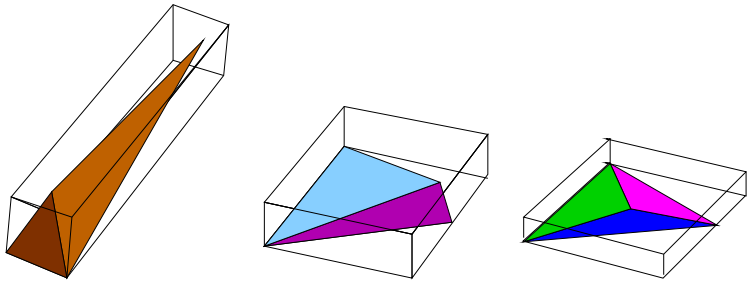

Figure 9: The radius-edge ratios of some badly-shaped tetrahedra.

lengths of the six edges are equal, and the circumcenter is the barycenter), that is:

$$Q \geq \sqrt{6}/4 \approx 0.612.$$

A special type of badly-shaped tetrahedron is called *sliver* (see Figure 10), which is very flat and nearly degenerate. Slivers can have very small radius-edge ratio, as small as $\sqrt{2}/2 \approx 0.707$. The radius-edge ratio is not a proper measure for slivers. However, Miller, Talmor, Teng, Walkington, and Wang [11] have pointed out that it is the most natural and elegant measure for analyzing Delaunay refinement algorithms.

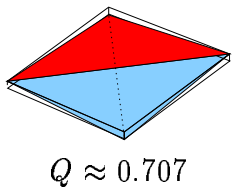

Figure 10: The radius-edge ratios of a sliver.

2 Getting Started

The distributed version of TetGen (available at `http://tetgen.berlios.de`) should include the following files:

README	General information.
LICENSE	Copyright notices.
tetgen.h	Header file of the TetGen library.
tetgen.cxx	C++ source code of the TetGen library.
predicates.cxx	C++ source code of the geometric predicates.
makefile	Makefile for compiling TetGen.
manual.pdf	User's manual (this file).
example.poly	A sample data file.

TetGen is written by using the standard C++ library only. The code is highly portable, it should run on all 32-bit and 64-bit computers. File predicates.cxx is incorporated from the work of Jonathan R. Shewchuk [13]. It includes C++ versions of the C codes to perform exact geometric predicates.

2.1 Compiling

At first one has to compile TetGen. What you need is just a C++ compiler available on the system you want to run TetGen, such as g++ on Unix/Linux, or Microsoft C++ or Borland C++ on Windows, and so on. Depending on your purpose to use TetGen, you can compile TetGen into an executable program, or a library which can be embedded into another program, or both of them. The compilation should be relatively easy.

2.1.1 On Unix/Linux

The easiest way to compile TetGen is to edit and use the included makefile. Before compiling, put all source files (tetgen.h, tetgen.cxx, and predicates.cxx) and the makefile in one directory (usually they are), read the makefile, which describes your options, and edit it accordingly. You should specify the C++ compiler and the level of optimization.

Once you've done this, type "make" to compile TetGen into an executable program or type "make tetlib" to compile TetGen into a library. The executable file "tetgen" or the library "libtet.a" appears in the same directory as the makefile.

Alternatively, the files are usually easy to compile without a makefile. Assume you're using g++, first compile the file predicates.cxx to get an object file:

```
g++ -c predicates.cxx
```

To compile TetGen into an executable file, use the following command:

```
g++ -O -o tetgen tetgen.cxx predicates.o -lm
```

To compile TetGen into a library, the symbol TETLIBRARY is needed:

```
g++ -O -DTETLIBRARY -c tetgen.cxx
ar r libtet.a tetgen.o predicates.o
```

2.1.2 On Windows 9x/NT/2000/XP

TetGen compiles as a console program "tetgen.exe" or a library "tetgen.lib" on Win32 systems. Test have been done with Microsoft Visual C++ 6.0 (VC++). The easiest way to compile TetGen is to use the integrated development environment (IDE) of VC++. The minimum steps to create "tetgen.exe" are:

- create a "Win32 console application" called "tetgen",
- add all source files into this project,
  i.e., tetgen.h, tetgen.cxx, and predicates.cxx
- build the project.

To create a library do the following minimum steps:

- create a "Win32 static library" called "library",
- add all source files into this project,
- add the symbol "TETLIBRARY" to compile switches.
- build the project.

2.2 Testing

TetGen gives a short list of command line options if it is invoked without arguments (that is, just type "tetgen"). A brief description of the usage is printed by invoking TetGen with the -h switch:

```
tetgen -h
```

The enclosed sample file, example.poly, is a simple three-dimensional piecewise linear complex (PLC). Try out TetGen using:

```
tetgen -p example
```

TetGen will read this PLC, and write its constrained Delaunay tetrahedralization to files example.1.node, example.1.ele, and example.1.face, which are a list of mesh nodes, a list of tetrahedra, and a list of boundary faces (triangles), respectively. A typical output of TetGen looks like this:

```
Opening example.poly.
Constructing Delaunay tetrahedrization.
Delaunay seconds:  0.05
Creating surface mesh.
Recovering non-Delaunay segments.
Inserting subfaces into Delaunay tetrahedralization.
Segment and facet seconds:  0.04
Removing unwanted tetrahedra.
Hole seconds:  0
Removing illegal tetrahedra.
Repair seconds:  0

Writing example.1.node.
Writing example.1.ele.
Writing example.1.face.

Output seconds:  0
Total running seconds:  0.09

Statistics:

  Input points: 54
  Input facets: 29
  Input holes: 0
  Input regions: 0

  Mesh points: 55
  Mesh tetrahedra: 119
  Mesh faces: 291
  Mesh subfaces: 106
  Mesh subsegments: 82
```

To get the quality tetrahedral mesh of this PLC, try:

```
tetgen -pq example
```

The result mesh is contained in the same three files as before. But this time, it is a quality tetrahedral mesh, whose tetrahedra have circumradius-to-shortest-edge ratio bounded by 2.0. Now try to run:

```
tetgen -pq1.414V example
```

TetGen will again generate a quality mesh, which contains more points than previous one, and all tetrahedra have radius-edge ratio bounded by 1.414. In addition, TetGen prints a rough mesh quality report (due to the -V switch). It looks as below:

```
Mesh quality statistics:

  Smallest volume:       0.0053785   |  Largest volume:          34.295
```

```
Shortest edge:            0.30902   |  Longest edge:             8.9271
Smallest dihedral:         1.7613   |  Largest dihedral:       176.3442

Radius-edge ratio histogram:
       < 0.707     :        38      |    1.6 - 1.8        :         0
 0.707 - 1         :      1049      |    1.8 - 2          :         0
     1 - 1.1       :       374      |      2 - 2.5        :         0
   1.1 - 1.2       :       268      |    2.5 - 3          :         0
   1.2 - 1.4       :       285      |      3 - 10         :         0
   1.4 - 1.6       :         5      |     10 -            :         0
(A tetrahedron's radius-edge ratio is its radius of circumsphere divided
  by its shortest edge length)

Aspect ratio histogram:
1.1547 - 1.5       :         0      |     15 - 25         :         9
   1.5 - 2         :         0      |     25 - 50         :         3
     2 - 2.5       :         0      |     50 - 100        :         2
   2.5 - 3         :        73      |    100 - 300        :         0
     3 - 4         :       890      |    300 - 1000       :         0
     4 - 6         :       927      |   1000 - 10000      :         0
     6 - 10        :        95      |  10000 - 100000     :         0
    10 - 15        :        20      | 100000 -            :         0
(A tetrahedron's aspect ratio is its longest edge length divided by the
  diameter of its inscribed sphere)

Dihedral Angle histogram:
     0 - 10 degrees:        16     |     90 - 100 degrees:       644
    10 - 20 degrees:        61     |    100 - 110 degrees:       403
    20 - 30 degrees:       224     |    110 - 120 degrees:       328
    30 - 40 degrees:       634     |    120 - 130 degrees:       169
    40 - 50 degrees:       738     |    130 - 140 degrees:        99
    50 - 60 degrees:       303     |    140 - 150 degrees:        48
    60 - 70 degrees:        43     |    150 - 160 degrees:        25
    70 - 80 degrees:        33     |    160 - 170 degrees:        14
    80 - 90 degrees:       250     |    170 - 180 degrees:         6
```

To compute the Delaunay tetrahedralization and convex hull of the point set of this PLC, try this:

```
cp example.poly example.node
tetgen example
```

The Delaunay tetrahedralization is saved in example.1.node and example.1.ele. The convex hull is represented by a list of triangles in file example.1.face.

All these meshes and Delaunay tetrahedralizations can be visualized by the programs introduced in the next section.

Detailed descriptions of the command line switches and file formats are found in Section 3.

2.3 Visualizing

Geometric objects and meshes are best understood by visualization. This section helps to use the viewing tools.

2.3.1 Viewing by TetView

TetView is a graphic interface for viewing piecewise linear complexes and meshes. It is specially created for TetGen. It can read the input and output files and display the objects like piecewise linear complexes, tetrahedral meshes and triangular meshes. It also shows other informations as well, such as boundary types and materials. The interactive GUI allows the user to manipulate (i.e., rotate, translate, zoom in/out, cut, shrink, etc.) the viewing objects easily through either mouse or keyboard. TetView also saves the current window contents into high quality encapsulated postscript files.

TetView is freely available from the website `http://tetgen.berlios.de/tetview.html`. You will find a list of precompiled executable versions on different platforms. You only need to download the one for your system.

The basic usage of TetView is very easy. For example, to show the PLC in example.poly, first copy the executable file (tetview) to the directory where you have this file, it is loaded by running:

```
tetview example.poly
```

And the following command will display the mesh (in files example.1.node, example.1.ele, and example.1.face):

```
tetview example.1
```

The instruction for using TetView can be found in the website mentioned above.

2.3.2 Viewing by Other Tools

Two programs can be alternatively used to view the mesh created by TetGen, both are publicly available and run on most computer systems. They are:

- *Medit*, a user-friendly mesh viewer, available at `http://www-rocq.inria.fr/gamma/medit`.
- *Gid*, the personal pre- and post-processor, available at `http://gid.cimne.upc.es`.

For viewing mesh under Medit, add a '-g' switch in the command line. TetGen will additionally output a file example.1.mesh, which can be read and rendered directly by Medit. Try running:

```
tetgen -pg example.poly
medit example.1
```

By invoking '-G' switch in command line, two additional files example.1.ele.msh and example.1.face.msh will be output by TetGen, which are tetrahedral mesh and surface mesh, respectively. These files can be loaded into Gid (version 7.0) using menu "Files" $\rightarrow$ "Import" $\rightarrow$ "Gid Mesh...".

3 Using TetGen

This section describes the use of TetGen as a stand alone program. It is invoked from the command line with a set of switches and an input file name. Switches are used to control the behavior of TetGen and to specify the output files. In correspondence to the different switches, TetGen will generate the Delaunay tetrahedrization, the constrained Delaunay tetrahedrization or a quality conforming Delaunay mesh. The command syntax is:

```
tetgen [-pq__a__Ars__iMT__tzo_fengGBNEFIOCQVvh] input_file
```

Underscores indicate that numbers may optionally follow certain switches. Do not leave any space between a switch and its numeric parameter. Command line switches are described in Section 3.1. "input_file" must be a file with extension .node, or extension .poly or .smesh if the -p switch is used. If -r is used, you must supply .node and .ele files, and possibly a .face file, and a .vol file as well. File formats are described in Section 3.2.

3.1 Command Line Switches

Table 1 is an overview of all command line switches and a short description follows each switch. This information is also available by invoking TetGen without any switch and input file (i.e., type "tetgen").

3.1.1 -p Generates a CDT

The -p switch reads a piecewise linear complex (PLC) stored in file .poly or .smesh, which can specify vertices, facets, holes, regional attributes, and volume constraints. The result is a constrained Delaunay tetrahedralization (CDT) fitting the input.

If the file extension is not specified, TetGen will look for a file which has extension .poly or .smesh and use whichever one is available. For example, "tetgen -p xxx" opens the file named xxx.poly (if it doesn't exist, open the file xxx.smesh instead) and possibly also opens the file xxx.node; reads in the whole PLC; and generates a CDT resulting in three files: xxx.1.node, xxx.1.ele, and xxx.1.face.

In combination with the -q or -a switch, TetGen will generate a quality tetrahedral mesh of the PLC. -p is not compatible with -r, and should not be used them together.

3.1.2 -q Quality mesh generation

The -q switch performs quality mesh generation by Shewchuk's Delaunay refinement algorithm [15]. It adds vertices to the CDT (used together with -p) or a previously generated mesh (used together with -r) to ensure that no tetrahedra have radius-edge

-p	Tetrahedralizes a piecewise linear complex (.poly or .smesh file).
-q	Quality mesh generation. A minimum radius-edge ratio may be specified (default 2.0).
-a	Applies a maximum tetrahedron volume constraint.
-A	Assigns attributes to identify tetrahedra in certain regions.
-r	Reconstructs/Refines a previously generated mesh.
-s	Attempts to remove slivers. A maximum dihedral angle (in degree) may be specified (default 175).
-i	Inserts a list of additional points into mesh.
-M	Does not merge coplanar facets.
-T	Set a tolerance for coplanar test (default 1e-8).
-t	Refines a surface mesh.
-z	Numbers all output items starting from zero.
-o2	Generates second-order subparametric elements.
-f	Outputs faces (including non-boundary faces) to .face file.
-e	Outputs subsegments to .edge file.
-n	Outputs tetrahedra neighbors to .neigh file.
-g	Outputs mesh to .mesh file for viewing by Medit.
-G	Outputs mesh to .msh file for viewing by Gid.
-B	Suppresses output of boundary information.
-N	Suppresses output of .node file.
-E	Suppresses output of .ele file.
-F	Suppresses output of .face file.
-I	Suppresses mesh iteration numbers.
-O	Ignores holes in .poly or .smesh file.
-C	Checks the consistency of the final mesh.
-Q	Quiet: No terminal output except errors.
-V	Verbose: Detailed information, more terminal output.
-v	Prints the version information.
-h	Help: A brief instruction for using TetGen.

Table 1: Overview of TetGen's command line switches.

ratio greater than 2.0. An alternative minimum radius-edge ratio may be specified after the 'q'. For a too small ratio, e.g., smaller than 1.0, TetGen may not terminate.

If no input angle or input dihedral angle (of the PLC) is smaller than 60°, this algorithm is guaranteed to terminate and no tetrahedron has radius-edge ratio greater than 2.0. In practice, one can observe successful termination for radius-edge ratios down to $\sqrt{2} \approx 1.414$ or even smaller.

Here are some examples of using the -q switch. Tests can be executed and compared with the enclosed example file (example.poly):

```
tetgen -pq example
tetgen -rq1.414 example.1
tetgen -ra0.5 example.2
```

3.1.3 -a Imposes volume constraints

The -a switch imposes a maximum volume constraint on all tetrahedra. If a number follows the 'a', no tetrahedra is generated whose volume is larger than that number.

If no number is specified and the -r switch is not used, TetGen will read the region part in file .poly or .smesh. A .poly file or .smesh file can optionally contain a volume constraint for each facet-bounded region, thereby controlling tetrahedra densities in a tetrahedralization of a PLC in a more specific way.

One can impose both a fixed volume constraint and a varying volume constraint for some regions by invoking the -a switch twice, once with and once without a number following. Each volume specified may include a decimal point.

If no number is specified and the -r switch is used, a .vol file is expected, which contains a separate volume constraint for each tetrahedron. It is useful for refining a finite element or finite volume mesh based on a posteriori error estimates.

3.1.4 -A Assigns region attributes

The -A switch assigns an additional attribute to each tetrahedron that identifies what facet-bounded region each tetrahedron belongs to. Attributes are assigned to regions by the .poly file or .smesh file. If a region is not explicitly marked by the .poly file or .smesh file, tetrahedra in that region are assigned an attribute of zero. The -A switch has an effect only when the -p switch is used and the -r switch is not.

3.1.5 -r Reconstructs/refines a mesh

The -r switch reconstructs a previously generated mesh. The mesh is read from a .node and an .ele file, and possibly a .face file. If a .face file exists, it is read and used to constrain subfaces in the mesh, else, TetGen will automatically identify the

subfaces, internal subfaces are also identified by comparing the attributes of two adjacent tetrahedra. Subsegments will be automatically identified from the existing subfaces. The reconstructed mesh is distinguished from its origin with a different iteration number.

For example, "tetgen -r xxx.1" reads the mesh in files xxx.1.node, xxx.1.ele and possibly xxx.1.face and xxx.1.edge if they exist; reconstructs the mesh; outputs it into three files xxx.2.node, xxx.2.ele and xxx.2.face. Now, xxx.2 can be used as input in the above command, the result is another mesh saved files xxx.3.node, and so on. Mesh iteration numbers allow you to create a sequence of successively finer meshes.

One can refine a mesh by combining -r with the -q, -a, and -i switches. Several ways are possible:

- One can impose tighter quality constraints by using the -q with a smaller number, or the -a followed by a smaller volume than the one used to generate the mesh currently refining.

- One can create a .vol file, which specifies a maximum volume for each tetrahedra, and use the -a switch (without a number following). Each tetrahedron's volume constraint is applied to that tetrahedra.

- One can create a .node file, which contains a list of additional nodes that you want them to be in the mesh. Use the -i switch to inform TetGen that an additional node list needs to be inserted.

-r should not be used with the -p and -I together.

3.1.6 -i Inserts additional points

The -i switch indicates to insert a list of additional points into a CDT (when -p switch is used) or a previously generated mesh (when -r switch is used). The list of additional file is read from file xxx-a.node, which xxx stands for the input file name (i.e., xxx.poly or xxx.smesh, or xxx.ele, ...). This switch is useful for refining a finite element or finite volume mesh using a list of user-defined points. Following are some pointers that you may need be careful:

- Points lie out of the mesh domain are ignored by TetGen.

- The mesh may not be constrained Delaunay or conforming Delaunay any more after the insertion of additional points. However, in combination with the -q switch, TetGen will automatically add additional points to ensure the conforming Delaunay property of the mesh.

3.1.7 -T Sets a tolerance

The -T switch sets a user-defined tolerance used by many computations of TetGen, default is $1e-8$.

The main reason to use the tolerance is that, in principle, the vertices which are used to define a facet of a PLC should be exactly coplanar. But this is very hard to achieve in practice due to the inconsistent nature of the floating-point format used in computers. TetGen accepts facets which vertices are not exactly but approximately coplanar. In TetGen, four points a, b, c and d are assumed be coplanar if the ratio v/l^3 is smaller than the given tolerance, where v is the volume of the tetrahedron $abcd$, and l is the average edge length of tetrahedron $abcd$.

3.1.8 Other Switches

-I The -I switch does not use the iteration numbers, it suppresses the output of .node file, so your input file won't be overwritten. It cannot be used with the -r switch, because that would overwrite your input .ele file. It shouldn't be used with the -q, -a, -s, or -t switch if one is using a .node file for input, because no .node file is written, so there is no record of any added Steiner points.

-z The -z switch numbers all output items starting from zero. This switch is normally overridden by the value used to number the first vertex of the input .node or .poly file or .smesh file. However, this switch is useful in case of calling TetGen from another program.

-o2 TetGen generates meshes with quadratic elements if the -o2 switch is specified. Quadratic elements have ten nodes per element, rather than four. The six extra nodes of a tetrahedron fall at the midpoints of its six edges.

-C The -C switch indicates TetGen to check the consistency of the mesh on finish. If it is specified twice, i.e., '-CC', TetGen also checks constrained Delaunay (for the -p switch) or conforming Delaunay (for -q, -a, or -i) property for the mesh.

-V The -V switch gives detailed information about what TetGen is doing. More 'V's are increasing the amount of detail.

Specifically, '-V' gives information on algorithmic progress and more detailed statistics including a rough mesh quality report. To get the statistics for an existing mesh, run TetGen on that mesh with the '-rNEP' switches to read the mesh and print the statistics without writing any files.

'-VV' gives more details on the algorithms, and slow down the execution. While '-VVV' isx only useful for debugging.

.node	input/output	a list of nodes.
.poly	input	a PLC.
.smesh	input/output	a simple PLC.
.ele	input/output	a list of tetrahedra.
.face	input/output	a list of triangular faces.
.edge	output	a list of boundary edges.
.vol	input	a list of maximum volumes.
.neigh	output	a list of neighbors.

Table 2: Overview of TetGen's file formats.

3.2 File Formats

Table 2 gives an overview over all file formats of TetGen. All files are of ASCII form and may contain comments prefixed by the character '#'. Points, tetrahedra, facets, holes, and maximum volume constraints must be numbered consecutively, starting from either 1 or 0. Whichever you choose, all input files must be consistent; if the vertices are numbered from 1, so must be all other objects. TetGen automatically detects your choice while reading the .node (or .poly or .smesh) file. (When calling TetGen from another program, use the -z switch if you wish to number objects from zero.)

Remark: in the following description '#' stands for 'number' – it should not cause confusion with the comment prefix.

3.2.1 .node files

```
First line:  <# of points> <dimension (3)> <# of attributes>
             <boundary markers (0 or 1)>
Remaining lines list # of points:
  <point #> <x> <y> <z> [attributes] [boundary marker]
  ...
```

A .node file contains a list of three-dimensional points. Each point has three coordinates (x, y and z), probably has one or several attributes, and a boundary marker as well. The .node files used as both input and output files to represent the point set of a PLC, or the point set of a mesh, or a set of additional points (for the -i switch) which need to be inserted into a mesh.

The attributes, which are typically floating-point values of physical quantities (such as mass or conductivity) associated with the points, are copied unchanged to the output mesh. If -p switch is used, each new Steiner point inserted on segments of the mesh has attributes assigned to it by linear interpolation. Furthermore, if -q, -a, or -i is selected, each new point added to the mesh to improve mesh quality has attributes zero.

If the fourth entry of the first line is '1', the last column of the remainder of the file is assumed to contain boundary markers. Boundary markers are used to identify boundary points (points resting on PLC facets). The .node file produced by TetGen contains boundary markers in the last column unless they are suppressed by the -B switch. The boundary marker associated with each point in an output .node file is chosen as follows:

- If a point is assigned a nonzero boundary marker in the input file, then it is assigned the same marker in the output .node file.
- Otherwise, if the point occurs on a boundary of the mesh, then the point is assigned the marker one (1).
- Otherwise, the point is assigned the marker zero (0).

TetGen can determine which points are on the boundary, input with the boundary marker zero (or use no markers at all) will result in output with boundary marker 1 for all points on the boundary.

3.2.2 .poly files

A .poly file represents a piecewise linear complex (PLC) as well as some additional information. Although there is no restriction on facets of PLCs, TetGen requires that the mesh region represented by a PLC should be completely facet-bounded, i.e. it is waterproof.

The .poly file format consists of four parts, which are a list of points, a list of facets, a list of (volume) hole points, and a list of region attributes, respectively. The first three parts are mandatory, but the fourth part is optional.

Part 1 - node list

```
First line:  <# of points> <dimension (3)> <# of attributes>
             <boundary markers (0 or 1)>
Remaining lines list # of points:
  <point #> <x> <y> <z> [attributes] [boundary marker]
  ...
```

Part 1 lists all the points, and is identical to the format of .node files. <# of points> may be set to zero to indicate that the points are listed in a separate .node file.

Part 2 - facet list

```
One line: <# of facets> <boundary markers (0 or 1)>
Following lines list # of facets:
   <facet #>
   ...
```

To understand the format of a facet is crucial. Each facet is a planar straight line graph (PSLG), i.e., it is a polygonal region which may contain segments, single points and holes in it. A list of *polygons* is used to represent a facet. Each polygon has n corners, $n \geq 1$. The polygon can be degenerate, i.e., $n = 1$ or $n = 2$ represents a single point or a segment, respectively. The format of a facet is:

```
One line: <# of polygons> [# of holes] [boundary marker]
Following lines list # of polygons:
  <# of corners> <corner 1> <corner 2> ... <corner #>
  ...
Following lines list # of holes:
  <hole #> <x> <y> <z>
  ...
```

Each polygon is specified by giving the number of corners, followed by the list of ordered point indices. Each line of corners should not be arbitrarily long because the maximum characters per line read by TetGen is 1024. The corner list can be given by more than one line.

A hole in a facet (don't confuse with the volume hole below) is sepcified by identifying a point inside the hole. However, this point need not be exactly in the hole, as long as its orthogonal projection onto this facet is in the hole. The list of hole points (consecutively) follows the list of polygons.

If the boundary markers is '1', TetGen will produce an additional boundary marker for each face in .face file (in the last column of each record). You can prevent boundary markers from being written into .face file by using the -B switch.

Boundary markers of facets are tags used mainly to identify which faces of the tetrahedralization are associated with which PLC facet, hence identify which faces occur on a boundary of the tetrahedralization. A common use is to determine where different boundary condition types should be applied to a mesh.

Part 3 - (volume) hole list

```
One line: <# of holes>
Following lines list # of holes:
  <hole #> <x> <y> <z>
  ...
```

Holes in the volume are specified by identifying a point inside each hole. After the constrained Delaunay tetrahedrization is formed, TetGen creates holes by removing

tetrahedra. Thus exactly is the reason why TetGen requires a closed boundary of the PLCs. In case of non closed PLC facets the whole tetrahedrization will be 'eaten' away. If two tetrahedra abutting a subface are removed, the subface itself is also vanished. Hole points have to be placed inside a region, else the rounding error determines which side of the facet is being transformed into the hole.

Part 4 - region attributes list

```
One line: <# of region>
Following lines list # of region attributes:
  <region #> <x> <y> <z> <region number> <region attribute>
  ...
```

The optional fourth section lists regional attributes (to be assigned to all tetrahedra in a region) and regional constraints on the maximum tetrahedron volume. TetGen will read this section only if the -A switch is used or the -a switch without a number is invoked. Regional attributes and volume constraints are propagated in the same manner as holes.

If two values are written on a line after the x, y and z coordinate, the former is assumed to be a regional attribute (but will only be applied if the -A switch is selected), and the latter is assumed to be a regional volume constraint (but will only be applied if the -a switch is selected). It is possible to specify just one value after the coordinates. It can serve as both an attribute and an volume constraint, depending on the choice of switches. A negative maximum volume constraint allows to use the -A and the -a switches without imposing a volume constraint in this specific region.

Depending on the command line switches -p and -q, the constrained Delaunay or conforming Delaunay property at the internal boundary between two regions are automatically preserved.

3.2.3 .smesh files

A .smesh file represents a PLC of special type - surface meshes. The .smesh file format is a simplified version of the .poly format, that each facet only has exactly one polygon, no holes, no segment and point inside. It is less flexible than the .poly file format but is much simpler and useful when the surface mesh is created by other programs.

The same as .poly file format, the .smesh file format consists of four parts, which are points, facets, holes and regions, respectively. Only the part describing facets is different from the .poly format. Hence only this part is described in this section. Please refer to Section 3.2.2 for other parts.

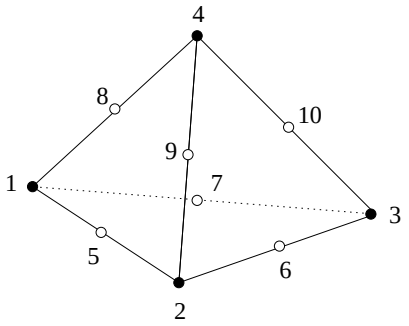

Figure 11: The numbering of nodes of a 10-node tetrahedron.

Part 2 - facet list

```
One line: <# of facets> <boundary markers (0 or 1)>
Following lines list # of facets:
  <# of corners> <corner 1> ... <corner #> [boundary marker]
  ...
```

Part 2 of a .smesh file lists all facets. It is different from the facet format of .poly file. Each facet consists of exactly one polygon. The corner list of each polygon can be distributed over a number of lines. The optional boundary marker of each facet is given at the end of the corner list.

3.2.4 .ele files

```
First line: <# of tetrahedra> <nodes per tet. (4 or 10)>
            <region attribute (0 or 1)>
Remaining lines list # of tetrahedra:
  <tetrahedron #> <node> <node> ... <node> [attribute]
  ...
```

A .ele file contains a list of tetrahedra. Each tetrahedron has four corners or ten corners (if -o2 switch is used). Nodes are indices into the corresponding .node file. The first four nodes are the corner vertices. If -o2 is used, the remaining six nodes are generated on the midpoints of the edges of the tetrahedron. Figure 11 shows how these nodes are numbered.

If the region attribute in the first line is '1', each tetrahedra has an additional region attribute (an integer) in the last column. Region attributes of tetrahedra are tags used mainly to identify which tetrahedra of the tetrahedralization are associated with which facet-bounded region of the PLC, set in the part 4 of a .poly or a .smesh file. Region attributes do not diffuse across facets, all tetrahedra in the same region have exactly the same region attribute. A common use of the region attribute is to determine which material a tetrahedron has.

The .ele file is the default output file of TetGen if it generated a mesh or tetrahedralization. However, it can be omitted by -E switch. If -r switch is used, TetGen reads a .ele file and reconstructs a tetrahedral mesh from it.

3.2.5 .face files

```
First line: <# of faces> <boundary marker (0 or 1)>
Remaining lines list # of faces:
  <face #> <node> <node> <node> [boundary marker]
  ...
```

A .face file contains a list of triangular faces, which may be boundary faces (if -p or -r switch is used), or convex hull faces. Each face has three corners and possibly has a boundary marker. Nodes are indices into the corresponding .node file.

After generating a mesh or Delaunay tetrahedralization, TetGen default outputs the boundary faces or the convex hull into a .face file. However, this file can be omitted by -F switch. If -r switch is used, TetGen can also read the .face file for identifying boundary faces in a reconstructed mesh. The optional column of Boundary markers is suppressed by the -B switch.

If the boundary marker in the first line is '1', each face has an additional boundary marker (an integer) in the last column. Boundary markers of facets are defined in the .poly or the .smesh files. They are tags used mainly to identify which faces of the tetrahedralization are associated with which PLC facet, and to identify which faces occur on a boundary of the tetrahedralization. A common use is to determine where different boundary condition types should be applied on boundary faces.

3.2.6 .edge files

```
First line: <# of edges>
Remaining lines list # of edges:
  <edge #> <endpoint> <endpoint>
  ...
```

A .edge file contains a list of edges, which are (sub)segments of a PLC. Each edge has two endpoints which are indices into the corresponding .node file. It is part of TetGen's output when -e switch is used.

3.2.7 .vol files

```
First line: <# of tetrahedra>
Remaining lines list # of maximum volumes:
  <tetrahedron #> <maximum volume>
  ...
```

A .vol file associates with each tetrahedron a maximum volume which is used for mesh refinement. It is read by TetGen in case the -r switch is used.

As with other file formats, every tetrahedron must be represented, and they must be numbered consecutively. A tetrahedron may be left unconstrained by assigning it a negative maximum volume.

3.2.8 .neigh files

```
First line:  <# of tetrahedra> <# of nei. per tet. (always 4)>
Following lines list # of neighbors:
  <tetrahedra #> <neighbor> <neighbor> <neighbor> <neighbor>
  ...
```

A .neigh file associates with each tetrahedron its neighbors (adjacent tetrahedra), which are indices into the corresponding .ele file. An index of -1 indicates no neighbor (because the tetrahedron is on a boundary of mesh domain). The first neighbor of tetrahedron i is opposite the first corner of tetrahedron i, and so on. It is output by TetGen when -n switch is used.

3.3 Examples

This section provides some simple examples and is designed to support learning by trying. The topics are description of the geometry using TetGen's .poly file format and constructing different quality meshes through the command line switches of TetGen.

3.3.1 A Bar with Two Boundary Markers

Figure 12 shows the geometry of a rectangular bar. This bar consists of eight nodes and six facets (which are all rectangles). In addition, there are two boundary markers (-1 and -2) associated to the leftmost facet and the rightmost facet, respectively. This simple model has its physical meaning. It can be seen as a typical heat transfer problem. The task is to compute the temperature diffusion in the bar, in which the flow of heat moves from hot side to cold side. The two boundary markers can represent two different boundary conditions, one has high temperature and the other has low temperature. Here is the input file "bar.poly" describing the bar:

```
# Part 1 - the node list.
# A bar with 8 nodes in 3D, no attributes, no boundary marker.
8  3  0  0
  # The 4 leftmost nodes:
  1  0  0  0
  2  2  0  0
  3  2  2  0
```

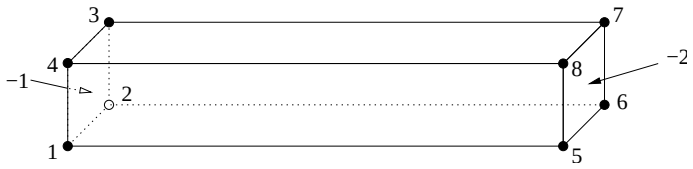

Figure 12: A rectangular bar with two boundary markers (−1 and −2) defined.

```
  4  0  2  0
  # The 4 rightmost nodes:
  5  0  0  12
  6  2  0  12
  7  2  2  12
  8  0  2  12
# Part 2 - the facet list.
# Six facets with boundary markers.
6  1
  # The leftmost facet.
  1  0  -1   # 1 polygon, no hole, boundary marker (-1)
  4  1 2 3 4
  # The rightmost facet.
  1  0  -2   # 1 polygon, no hole, boundary marker (-2)
  4  5 6 7 8
  # Other facets.
  1
  4  1 5 6 2 # bottom side
  1
  4  2 6 7 3 # back side
  1
  4  3 7 8 4 # top side
  1
  4  4 8 5 1 # front side
# Part 3 - the hole list.
# There is no hole in bar.
0
# Part 4 - the region list.
# There is no region defined.
0
```

The command line is chosen as follows: first mesh the PLC (-p), then impose the quality constraint (-q). This will result a quality mesh in three files: bar.1.node, bar.1.ele, and bar.1.face.

```
  > tetgen -pq bar
```

Here is the output file "bar.1.node". It contains twelve nodes. The last four points were added by TetGen automatically to meet the quality measure.

```
12  3  0  0
```

```
   1    0  0  0
   2    2  0  0
   3    2  2  0
   4    0  2  0
   5    0  0  12
   6    2  0  12
   7    2  2  12
   8    0  2  12
   9    2  2  6
  10    2  0  6
  11    0  0  6
  12    0  2  6
# Generated by tetgen -pq bar
```

Here is the output file "bar.1.ele", which contains thirteen tetrahedra.

```
13  4  0
    1      10    12     4    11
    2       5    11     9    10
    3       5     6    10     9
    4       9     5     6     7
    5       5     9     8     7
    6      10    12    11     9
    7      10    11     4     1
    8       4    10     1     2
    9       4     3    10     2
   10       3    12    10     9
   11       9    11     8    12
   12      11     5     9     8
   13      10     4    12     3
# Generated by tetgen -pq bar
```

Here is the output file "bar.1.face" with twenty boundary faces. Faces 1 and 2 are on the leftmost facet thus have markers -1; faces 13 and 14 have markers -2 indicating they are on the rightmost facet. Other faces have the default markers zero.

```
20  1
    1      1     2     4    -1
    2      4     2     3    -1
    3     11     1     4    0
    4     11     8     5    0
    5     10     1    11    0
    6      5     6    10    0
    7      6     9    10    0
    8      6     7     9    0
    9      9     7     8    0
   10      8    12     9    0
   11      3     2    10    0
   12      3     9    12    0
   13      5     7     6    -2
   14      8     7     5    -2
```

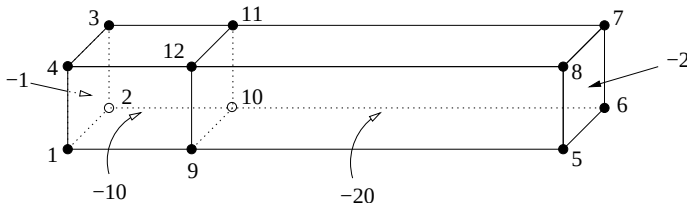

Figure 13: A rectangular bar with two regions (with region attributes -10 and -20, respectively) and two boundary markers (-1 and -2) defined.

```
   15     10      9      3     0
   16     10      2      1     0
   17      5     10     11     0
   18     11     12      8     0
   19     12     11      4     0
   20      4      3     12     0
# Generated by tetgen -pq bar
```

However, the mesh above may be too coarse for numerical simulation using finite element method or finite volume method. Using either -q or -a switch or both of them will results a more dense quality mesh:

```
> tetgen -pq1.414a0.1 bar
```

TetGen generates a mesh with 318 points and 1058 tetrahedra. The added points are due to both the -q and -a switches we've applied. To see a quality report of the mesh, type:

```
> tetgen -rNEFV bar.1
```

3.3.2 A Bar with Two Regions

In this example, we add an internal boundary facet into the bar (in Figure 12), so that create two regions (separated by the newly added facet) in the bar. Figure 13 shows the modified geometry. This bar consists of twelve nodes (which I numbered) and seven facets (Note, some of them are not a single polygon any more). In addition, there are two regions defined, which have region attributes -10 and -20, respectively. Physically, you can associate two different materials to each of these two regions. And the two boundary markers (-1 and -2) in last example still remain. Here is the input file "bar2.poly" describing the modified bar:

```
# Part 1 - the node list.
# The model has 12 nodes in 3D, no attributes, no boundary marker.
12  3  0  0
  # The 4 leftmost nodes:
```

```
  1  0  0  0
  2  2  0  0
  3  2  2  0
  4  0  2  0
  # The 4 rightmost nodes:
  5  0  0  12
  6  2  0  12
  7  2  2  12
  8  0  2  12
  # The 4 added nodes:
  9  0  0  3
  10 2  0  3
  11 2  2  3
  12 0  2  3
# Part 2 - the facet list.
# Seven facets with boundary markers.
7  1
  # The leftmost facet.
  1  0  -1   # 1 polygon, no hole, boundary marker (-1)
  4  1 2 3 4
  # The rightmost facet.
  1  0  -2   # 1 polygon, no hole, boundary marker (-2)
  4  5 6 7 8
  # Each of following facets has two polygons, which are
  # one rectangle (6 corners) and one segment.
  2
  6  1 9 5 6 10 2 # bottom side
  2  9 10
  2
  6  2 10 6 7 11 3 # back side
  2  10 11
  2
  6  3 11 7 8 12 4 # top side
  2  11 12
  2
  6  4 12 8 5 9 1 # front side
  2  12 9
  # The internal facet separates two regions.
  1
  4  9 10 11 12
# Part 3 - the hole list.
# There is no hole in bar.
0
# Part 4 - the region list.
# There are two regions (-10 and -20) defined.
2
  1  1.0 1.0 1.5  -10  0.1
  2  1.0 1.0 5.0  -20  -1
```

After "tetgen -pqaA bar2", here is the part of output file "bar2.1.ele". It is different from file "bar.1.ele" in last example that each record follows an additional region attribute indicating which region it belongs to.

```
306  4  1
    1       85    91    33    86    -10
    2       76    95    46    85    -10
    3       16    26    29    27    -20
    4        7     5    13     6    -20
    5       49    80    55    48    -10
    6       27   104    28    26    -20
    7       66    93    42    83    -10
    ...
# Generated by tetgen -pqaA bar2
```

Visualization of the resulting meshes (by TetView or other tools) shows the refinement in the region with attribute -10 is denser than the other. This is due to the volume constraints (0.1) defined in the file "bar2.poly" and the '-aA' switches.

4 Calling TetGen from Another Program

In addition to being used as a stand alone program, TetGen can be called from another program. The TetGen library provides functions and data structures for calling TetGen. One of the advantages of using the TetGen library is that it can be repeatedly called by other programs without the overhead of reading and writing files. The feature is very useful for applications like adaptive FEM and FVM methods.

This section gives the necessary instructions for using the TetGen library. Examples are included for better understanding. Users are supposed to be able to use TetGen, i.e., know its command line switches and the input and output file formats. Furthermore, Section 2 contains the instruction of how to compile TetGen into a library.

4.1 The Header File

All programs calling TetGen must include the header file "tetgen.h".

```
#include "tetgen.h"
```

It includes all data types and function declarations of the TetGen library. More specifically, it defines the function "tetrahedralize()" and the data type "tetgenio", which are provided for users to call TetGen with all its functionalities. They are described in Section 4.2 and Section 4.3, respectively.

4.2 The Calling Convention

The function "tetrahedralize()" is declared as follows:

```
void tetrahedralize(char *switches, tetgenio *in, tetgenio *out)
```

The parameter "switches" is a string containing the command line switches for this call. In this string, no initial dash '-' is required. The 'Q' (quiet) switch is recommended in the final code. The 'I' (no iteration numbers), 'g' (.mesh file output), and 'G' (.msh file output) switches have no effect.

The parameters "in" and "out", which are two pointers pointing to objects of "tetgenio", describing the input and the output. "in" and "out" may never be NULL.

4.3 The "tetgenio" Data Type

The "tetgenio" structure is used to pass data into and out of the tetrahedralize() procedure. It replaces the input and output files of TetGen by a collection of arrays,

which are used to store points, tetrahedra, markers, and so forth. It is declared as a C++ class including data fields and functions. The data fields of "tetgenio":

```
int firstnumber; // 0 or 1, default 0.
int mesh_dim;    // must be 3.

REAL *pointlist;
REAL *pointattributelist;
REAL *addpointlist;
int *pointmarkerlist;
int numberofpoints;
int numberofpointattributes;
int numberofaddpoints;

int *tetrahedronlist;
REAL *tetrahedronattributelist;
REAL *tetrahedronvolumelist;
int *neighborlist;
int numberoftetrahedra;
int numberofcorners;
int numberoftetrahedronattributes;

facet *facetlist;
int *facetmarkerlist;
int numberoffacets;

REAL *holelist;
int numberofholes;

REAL *regionlist;
int numberofregions;

int *trifacelist;
int *trifacemarkerlist;
int numberoftrifaces;

int *edgelist;
int *edgemarkerlist;
int numberofedges;
```

4.3.1 Description of Arrays

In all cases, the first item in any array is stored starting at index [0]. However, that item is item number "firstnumber" (0 or 1) unless the 'z' switch is used, in which case it is item number '0'. Following is the description of arrays.

pointlist An array of point coordinates. The first point's x coordinate is at index [0], its y coordinate at index [1], and its z coordinate at index [2], followed by the coordinates of the remaining points. Each point occupies three REALs.

pointattributelist An array of point attributes. Each point's attributes occupy "numberofpointattributes" REALs.

pointmarkerlist An array of point markers; one int per point.

addpointlist An array of additional point coordinates, which will be read and inserted into the mesh when the -i switch is used.. The same structure as "pointlist", each point occupies three REALs.

tetrahedronlist An array of tetrahedron corners. The first tetrahedron's first corner is at index [0], followed by its other three corners, followed by any other nodes if the '-o2' switch is used. Each tetrahedron occupies "numberofcorners" (4 or 10) ints.

tetrahedronattributelist An array of tetrahedron attributes. Each tetrahedron's attributes occupy "numberoftetrahedronattributes" REALs.

tetrahedronvolumelist An array of tetrahedron volume constraints; one REAL per tetrahedron. Input only.

neighborlist An array of tetrahedron neighbors; four ints per tetrahedron. Output only.

facetlist An array of PLC facets. Each facet is an object of type "facet" (see Section 4.3.3).

facetmarkerlist An array of facet markers; one int per facet.

holelist An array of holes. The first hole's x, y and z coordinates are at indices [0], [1] and [2], followed by the remaining holes. Three REALs per hole.

regionlist An array of regional attributes and volume constraints. The first constraints' x, y and z coordinates are at indices [0], [1] and [2], followed by the regional attribute at index [3], followed by the maximum volume at index [4], followed by the remaining volume constraints. Five REALs per volume constraint. Each regional

attribute is used only if the 'A' switch is used, and each volume constraint is used only if the 'a' switch (with no number following) is used, but omitting one of these switches does not change the memory layout.

trifacelist An array of triangular faces. The first face's corners are at indices [0], [1] and [2], followed by the remaining faces. Three ints per face.

trifacemarkerlist An array of face markers; one int per face.

edgelist An array of segment endpoints. The first segment's endpoints are at indices [0] and [1], followed by the remaining segments. Two ints per segment.

edgemarkerlist An array of segment markers; one int per segment.

4.3.2 Memory Management

The "initialize()" and "deinitialize()" routines defined in tetgenio are used for memory initialization and cleaning. They are:

```
void initialize();
void deinitialize();
```

"initialize()" initializes all fields, that is, all pointers to arrays are initialized to NULL, and other variables are initialized to zero except the variable 'numberofcorners', which is 4 (a tetrahedron has 4 nodes). Initialization is implicitly called by the constructor of tetgenio. For an example, the following line creates an object of tetgenio named "io", all fields of "io" are initialized:

```
tetgenio io;
```

The next step is to allocate memory for each array which will be used. In C++ the memory allocation and deletion can be done by the "new" and "delete" operators. Another pair of functions (preferred by C programmers) are "malloc()" and "free()". Whatever you use, you must stick with one of these two pairs, e.g., 'new'/'delete' and 'malloc'/'free' can not be mixed. For example, the following line allocates memory for "io.pointlist":

```
io.pointlist = new REAL[io.numberofpoints * 3];
```

"deinitialize" frees the memory allocated in objects of tetgenio by using 'delete'. It is automatically called on deletion of the tetgenio objects. If the memory was allocated by using the function "malloc()", the user is responsible to free it. After having freed all memory, one call of "initialize()" disables the automatic memory deletion.

To reuse an object is possible: first call "deinitialize()", then call "initialize()" before the next use.

4.3.3 The "facet" Data Structure

The "facet" data structure defined in tetgenio can be used to represent any facet of a PLC. However, to preserve the flexibility, to use is not straightforward.

The structure of facet shown below consists of a list of polygons and a list of hole points.

```
typedef struct {
  polygon *polygonlist;
  int numberofpolygons;
  REAL *holelist;
  int numberofholes;
} facet;
```

A polygon is again an object of type "polygon"'. It consists of a list of corner points ("vertexlist"). The structure is shown below.

```
typedef struct {
  int *vertexlist;
  int numberofvertices;
} polygon;
```

In fact, the structure of facet is a direct translation of the facet format in a .poly file, described in Section 3.2.2. The front facet of Figure 13 serves an example for setting a PLC facet into an object of facet. It has two polygons, one has six vertices, and the other is a segment, no holes, the ASCII data is:

```
2
6  4 12 8 5 9 1 # front side
2  12 9
```

The following C++ code does the translation. Assume the object of tetgenio is "io" and has already be created.

```
tetgenio::facet *f;   // Define a pointer of facet.
tetgenio::polygon *p; // Define a pointer of polygon.

// All indices start from 1.
io.firstnumber = 1;

...

// Use 'f' to point to a facet of 'facetlist'.
f = &io.facetlist[i];
// Initialize the fields of this facet.
//   There are two polygons, no holes.
f->numberofpolygons = 2;
// Allocate memory for polygons.
f->polygonlist = new tetgenio::polygon[2];
f->numberofholes = 0;
f->holelist = NULL;

// Set the data of the first polygon into facet.
p = &f->polygonlist[0];
p->numberofvertices = 6;
// Allocate memory for vertices.
p->vertexlist = new int[6];
p->vertexlist[0] = 4;
p->vertexlist[1] = 12;
p->vertexlist[2] = 8;
p->vertexlist[3] = 5;
p->vertexlist[4] = 9;
p->vertexlist[5] = 1;

// Set the data of the second polygon into facet.
p = &f->polygonlist[1];
p->numberofvertices = 2;
p->vertexlist = new int[2]; // Alloc. memory for vertices.
p->vertexlist[0] = 12;
p->vertexlist[1] = 9;
```

4.4 An Example

This section gives an example of how to call TetGen from another program by using the "tetgenio" data structure and the function "tetrahedralize()". The example in Section 3.3.1 (Figure 12) is used again.

The complete C++ source code is given below. It is also available in TetGen's website:`http://tetgen.berlios.de/files/tetcall.cxx`. The code illustrates the following basic steps:

- at first creates an input object "in" of tetgenio containing the data of the bar;
- then it calls function "tetrahedralize()" to create a quality mesh of the bar with output in "out".

In addition, It outputs the PLC in the object "in" into two files (barin.node and barin.poly), and outputs the mesh in the object "out" into three files (barout.node, barout.ele, and barout.face). These files can be visualized by TetView(Section2.3.1).

This example can be compiled into an executable program with the library of TetGen. To compile it you need to do the following steps:

- Compile TetGen into a library named "libtet.a" (see Section 2.1 for compiling);
- Save the file "tetcall.cxx" into the same directory in which you have the files "tetgen.h" and "libtet.a";
- Compile it using the following command:

  ```
  g++ -o test tetcall.cxx -L./ -ltet
  ```

  which will result an executable file "test".

Following are the complete source code:

```
//
// main()  Create and refine a mesh using TetGen library.
//

#include "tetgen.h" // Defined tetgenio, tetrahedralize().

int main(int argc, char *argv[])
{
  tetgenio in, out;
  tetgenio::facet *f;
  tetgenio::polygon *p;
  int i;

  // All indices start from 1.
  in.firstnumber = 1;

  in.numberofpoints = 8;
  in.pointlist = new REAL[in.numberofpoints * 3];
  in.pointlist[0]  = 0;  // node 1.
  in.pointlist[1]  = 0;
  in.pointlist[2]  = 0;
  in.pointlist[3]  = 2;  // node 2.
  in.pointlist[4]  = 0;
  in.pointlist[5]  = 0;
  in.pointlist[6]  = 2;  // node 3.
  in.pointlist[7]  = 2;
  in.pointlist[8]  = 0;
  in.pointlist[9]  = 0;  // node 4.
  in.pointlist[10] = 2;
  in.pointlist[11] = 0;
```

```
// Set node 5, 6, 7, 8.
for (i = 4; i < 8; i++) {
  in.pointlist[i * 3]     = in.pointlist[(i - 4) * 3];
  in.pointlist[i * 3 + 1] = in.pointlist[(i - 4) * 3 + 1];
  in.pointlist[i * 3 + 2] = 12;
}

in.numberoffacets = 6;
in.facetlist = new tetgenio::facet[in.numberoffacets];
in.facetmarkerlist = new int[in.numberoffacets];

// Facet 1. The leftmost facet.
f = &in.facetlist[0];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 1;
p->vertexlist[1] = 2;
p->vertexlist[2] = 3;
p->vertexlist[3] = 4;

// Facet 2. The rightmost facet.
f = &in.facetlist[1];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 5;
p->vertexlist[1] = 6;
p->vertexlist[2] = 7;
p->vertexlist[3] = 8;

// Facet 3. The bottom facet.
f = &in.facetlist[2];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 1;
p->vertexlist[1] = 5;
p->vertexlist[2] = 6;
p->vertexlist[3] = 2;

// Facet 4. The back facet.
```

```
f = &in.facetlist[3];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 2;
p->vertexlist[1] = 6;
p->vertexlist[2] = 7;
p->vertexlist[3] = 3;

// Facet 5. The top facet.
f = &in.facetlist[4];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 3;
p->vertexlist[1] = 7;
p->vertexlist[2] = 8;
p->vertexlist[3] = 4;

// Facet 6. The front facet.
f = &in.facetlist[5];
f->numberofpolygons = 1;
f->polygonlist = new tetgenio::polygon[f->numberofpolygons];
f->numberofholes = 0;
f->holelist = NULL;
p = &f->polygonlist[0];
p->numberofvertices = 4;
p->vertexlist = new int[p->numberofvertices];
p->vertexlist[0] = 4;
p->vertexlist[1] = 8;
p->vertexlist[2] = 5;
p->vertexlist[3] = 1;

// Set 'in.facetmarkerlist'

in.facetmarkerlist[0] = -1;
in.facetmarkerlist[1] = -2;
in.facetmarkerlist[2] = 0;
in.facetmarkerlist[3] = 0;
in.facetmarkerlist[4] = 0;
in.facetmarkerlist[5] = 0;

// Output the PLC to files 'barin.node' and 'barin.poly'.
in.save_nodes("barin");
in.save_poly("barin");
```

```
  // Tetrahedralize the PLC. Switches are chosen to read a PLC (p),
  //   do quality mesh generation (q) with a specified quality bound
  //   (1.414), and apply a maximum volume constraint (a0.1).

  tetrahedralize("pq1.414a0.1", &in, &out);

  // Output mesh to files 'barout.node', 'barout.ele' and 'barout.face'.
  out.save_nodes("barout");
  out.save_elements("barout");
  out.save_faces("barout");

  return 0;
}
```

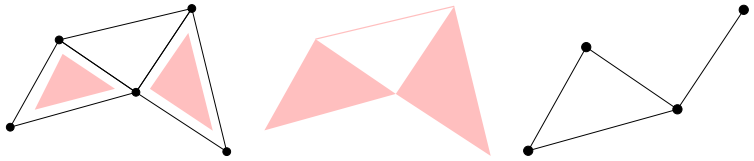

Figure 14: **(a)** A two-dimensional simplical complex K consists of 2 triangles(which are shaded), 7 edges, and 5 vertices. **(b)** The underlying space. **(c)** A subcomplex, which is a 1-dimensional simplical complex, consists of 4 edges, and 4 vertices.

A Some Notions of Combinatorial Topology

This section gives simplified explanations of some basic notions of combinatorial topology as a quick reference.

Convex Hull The *convex hull* of a set $V \subset \mathbf{R}^n$ of points, denoted conv$V$, is the smallest convex set that contains V.

Simplex A *simplex* σ is the convex hull of an affinely independent set S of points. The *dimension* of σ is one less than the number of points of S. Specifically, in $\mathbf{R}^3$ the maximum number of affinely independent points is 4, so we have non-empty simplices of dimensions 0, 1, 2 and 3 referred to as *vertices*, *edges*, *triangles*, and *tetrahedra*, respectively. For any subset $T \subseteq S$, the simplex $\tau =$ conv$T$ is a *face* of σ and we write $\tau \leq \sigma$. τ is a *proper face* of σ if T is a proper subset of S.

Simplical Complex A *simplical complex* K is a finite set of simplices such that (i) any face of a simplex in K is also in K, and (ii) the intersection of any two simplices in K is a face of both. Condition (ii) allows for the case in which two simplices are disjoint because the empty set is the unique (-1)-dimensional simplex, which is a face of any simplex. Figure 14 (a) illustrates a two-dimensional simplical complex.

Underlying Space The *underlying space* of a set of simplices L, denoted $|L|$, is the union of interiors, $\bigcup_{\sigma \in L}$intσ (see an illustration in Figure 14 (b)). $|L|$ is a topologically closed set if and only if L is a simplical complex.

Subcomplex A *subcomplex* of K is a subset of simplices of K that is also a simplical complex. For an example see Figure 14 (c).

References

[1] Boris N. Delaunay, *Sur la Sphère Vide.* Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk **7**:793-800, 1934.

[2] C. L. Lawson, *Software for C^1 surface interpolation.* In Mathematical Software III, Academic Press, New York, 161-194, 1977.

[3] B. Joe, *Construction of Three-dimensional Delaunay Triangulations from local transformations.* Computer Aided Geometric Design **8**: 123-142, 1991.

[4] L. P. Chew, *Constrained Delaunay Triangulation.* Algorithmica **4**(1):97-108, 1989.

[5] L. P. Chew, *Guaranteed-Quality Triangular Meshes.* Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[6] L. P. Chew, *Guaranteed-Quality Mesh Generation for Curved Surfaces.* Proceedings of the 9th Annual Symposium on Computational Geometry, pages 274-280. ACM May, 1993.

[7] J. Ruppert and R. Seidel, *On the difficulty of triangulating three-dimensional non-convex polyhedra.* Discrete & Computational Geometry **7**: 227-254, 1992.

[8] J. Ruppert *A Delaunay Refinement Algorithm for Quality Mesh Generation.* Journal of Algorithms **18**(3):548-585, May 1995.

[9] V. T. Rajan, *Optimality of the Delaunay Triangulation in* $\mathbf{R}^d$. Discrete Comput. Geom. **12**: 189-202, 1994.

[10] H. Edelsbrunner and N. R. Shah, *Incremental Topological Flipping Works for Regular Triangulations.* Algorithmica **15**: 223-241, 1996.

[11] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang, *A Delaunay Based Numerical Method for Three Dimensions: Generation, Formulation, and Partition.* Proceeding of 27th Annual ACM Symposium on the Theory of Computing, pages 683-692, May 1995.

[12] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang, *Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening.* Proceeding of 5th International Meshing Roundtable, pages 47-61, 1996.

[13] J. R. Shewchuk, *Adaptive Precision Floating-Point Arithmetic and Robust Geometric Predicates.* Discrete & Computational Geometry **18**(3):305-363, October 1997. C source code is available at `http://www.cs.cmu.edu/~quake/robust.html`.

[14] J. R. Shewchuk, *A Condition Guaranteeing the Existence of Higher-Dimensional Constrained Delaunay Triangulations.* Proceedings of the Fourteenth Annual Symposium on Computational Geometry (Minneapolis, Minnesota), pages 76-85, 1998.

[15] J. R. Shewchuk, *Tetrahedral Mesh Generation by Delaunay Refinement.* Proceedings of the Fourteenth Annual Symposium on Computational Geometry (Minneapolis, Minnesota), pages 86-95, 1998.

[16] J. R. Shewchuk, *Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery.* Eleventh International Meshing Roundtable (Ithaca, New York), pages 193-204. Sandia National Laboratories, September 2002.

[17] P. Pébay, *A Priori Delaunay-Conformity*, Proceedings of 7th International Meshing Roundtable, SANDIA, 1998.

[18] M. Murphy, D. M. Mount and C. W. Gable, *A Point-Placement Strategy for Conforming Delaunay Tetrahedralization*, Proceeding of the Eleventh Annual Symposium on Discrete Algorithms, pages 67-74. Association for Computing Machinery, January 2000.

[19] D. Cohen-Steiner, E. de Verdière, and M. Yvinec, *Conforming Delaunay Triangulations in 3D*, Proceedings of Eighteenth Annual Symposium on Computational Geometry (Barcelona, Spain). Association for Computing Machinery, June 2002.

[20] H. Si and K. Gärtner, *An Algorithm for Three-Dimensional Constrained Delaunay Tetrahedralizations*, Appears in the Proceeding of the Fourth International Conference on Engineering Computational Technology, Lisbon, Portugal, September 2004.