Delaunay Refinement Mesh Generation

Jonathan Richard Shewchuk

May 18, 1997

CMU-CS-97-137

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Gary L. Miller, *co-chair*
David R. O'Hallaron, *co-chair*
Thomas R. Gross
Omar Ghattas, Department of Civil and Environmental Engineering
Jim Ruppert, Arris Pharmaceutical

Copyright 1997 Jonathan Richard Shewchuk

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, in part by the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship, and in part by a grant from the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Abstract

Delaunay refinement is a technique for generating unstructured meshes of triangles or tetrahedra suitable for use in the finite element method or other numerical methods for solving partial differential equations. Popularized by the engineering community in the mid-1980s, Delaunay refinement operates by maintaining a Delaunay triangulation or Delaunay tetrahedralization, which is *refined* by the insertion of additional vertices. The placement of these vertices is chosen to enforce boundary conformity and to improve the quality of the mesh. Pioneering papers by L. Paul Chew and Jim Ruppert have placed Delaunay refinement on firm theoretical ground. The purpose of this thesis is to further this progress by cementing the foundations of two-dimensional Delaunay refinement, and by extending the technique and its analysis to three dimensions.

In two dimensions, I unify the algorithms of Chew and Ruppert in a common theoretical framework. Using Ruppert's analysis technique, I prove that one of Chew's algorithms can produce triangular meshes that are nicely graded, are size-optimal, and have no angle smaller than $26.5°$. (Chew proved a $30°$ bound without guarantees on grading or size.) I show that there are inputs with small angles that cannot be meshed by any algorithm without introducing new small angles; hence, all provably good mesh generation algorithms, including those not yet discovered, suffer from a fundamental limitation. I introduce techniques for handling small input angles that minimize the impact of this limitation on two-dimensional Delaunay refinement algorithms.

In three dimensions, I introduce a Delaunay refinement algorithm that can produce tetrahedral meshes that are nicely graded and whose tetrahedra have circumradius-to-shortest edge ratios bounded below 1.63. By sacrificing good grading in theory (but not in practice), one can improve the bound to 1.15. This theoretical guarantee ensures that all poor quality tetrahedra except *slivers* (a particular type of poor tetrahedron) are removed. The slivers that remain are easily removed in practice, although there is no theoretical guarantee. These results assume that all input angles are large; the removal of this restriction remains the most important open problem in three-dimensional Delaunay refinement. Nevertheless, Delaunay refinement methods for tetrahedral mesh generation have the rare distinction that they offer strong theoretical bounds and frequently perform well in practice.

I describe my implementations of the triangular and tetrahedral Delaunay refinement algorithms. The robustness of these mesh generators against floating-point roundoff error is strengthened by fast correct floating-point implementations of four geometric predicates: the two-dimensional and three-dimensional orientation and incircle tests. These predicates owe their speed to two features. First, they employ new fast algorithms for arbitrary precision arithmetic on standard floating-point units. Second, they are adaptive; their running time depends on the degree of uncertainty of the result, and is usually small. Hence, these predicates cost little more than ordinary nonrobust predicates, but never sacrifice correctness for speed.

Keywords: tetrahedral mesh generation, Delaunay triangulation, arbitrary precision floating-point arithmetic, computational geometry, geometric robustness

Contents

About this Thesis

The triangular mesh generator Triangle, described in Chapter 5, can be obtained through the Web page `http://www.cs.cmu.edu/~quake/triangle.html`.

The robust floating-point predicates described in Chapter 6 can be obtained through the Web page `http://www.cs.cmu.edu/~quake/robust.html`.

This thesis is copyright 1997 by Jonathan Richard Shewchuk. Please mail me (`jrs@cs.cmu.edu`) comments and corrections.

My thanks go to Omar Ghattas, Thomas Gross, Gary Miller, David O'Hallaron, Jim Ruppert, James Stichnoth, Dafna Talmor, and Daniel Tunkelang for comments and conversations that significantly improved this document. Of these, Jim Ruppert must also be singled out for writing the paper on Delaunay refinement that lit a fire under me from the day I first read it.

For Chapter 6, which forms a sort of mini-dissertation within a dissertation, my thanks go to Steven Fortune, Douglas Priest, and Christopher Van Wyk for comments on my draft and for writing the papers that provided the foundations of my research in floating-point arithmetic. Steven Fortune unwittingly sparked my research with a few brief email responses. He also provided LN-generated predicates for timing comparisons.

Double thanks to David O'Hallaron for the many ways he has supported me during my seven years at Carnegie Mellon.

Most of this work was carried out at the 61c Café in Pittsburgh.

Chapter 1

Introduction

Meshes composed of triangles or tetrahedra are used in applications such as computer graphics, interpolation, surveying, and terrain databases. Although the algorithms described in this document have been used successfully to generate meshes for these and other purposes, the central focus of this research is the generation of meshes for use in numerical methods for the solution of partial differential equations. These numerical methods are an irreplaceable means of simulating a wide variety of physical phenomena in scientific computing. Furthermore, they place particularly difficult demands on mesh generation. If one can generate meshes that are completely satisfying for numerical techniques like the finite element method, the other applications fall easily in line.

Delaunay refinement, the topic of this thesis, is a mesh generation technique that has theoretical guarantees to back up its good performance in practice. The center of this thesis is an extensive exploration of the theory of Delaunay refinement in two and three dimensions, found in Chapters 3 and 4. Implementation concerns are addressed in Chapter 5. Delaunay refinement is based upon a well-known geometric structure called the *Delaunay triangulation*, reviewed in Chapter 2.

This introductory chapter is devoted to explaining the problem that the remaining chapters undertake to solve. Unfortunately, the problem is not entirely well-defined. In a nutshell, however, one wishes to create a mesh that conforms to the geometry of the physical problem one wishes to model. This mesh must be composed of triangles or tetrahedra of appropriate sizes—possibly varying throughout the mesh—and these triangles or tetrahedra must be nicely shaped. Reconciling these constraints is not easy. Historically, the automation of mesh generation has proven to be more challenging than the entire remainder of the simulation process.

A detailed preview of the main results of the thesis concludes the chapter.

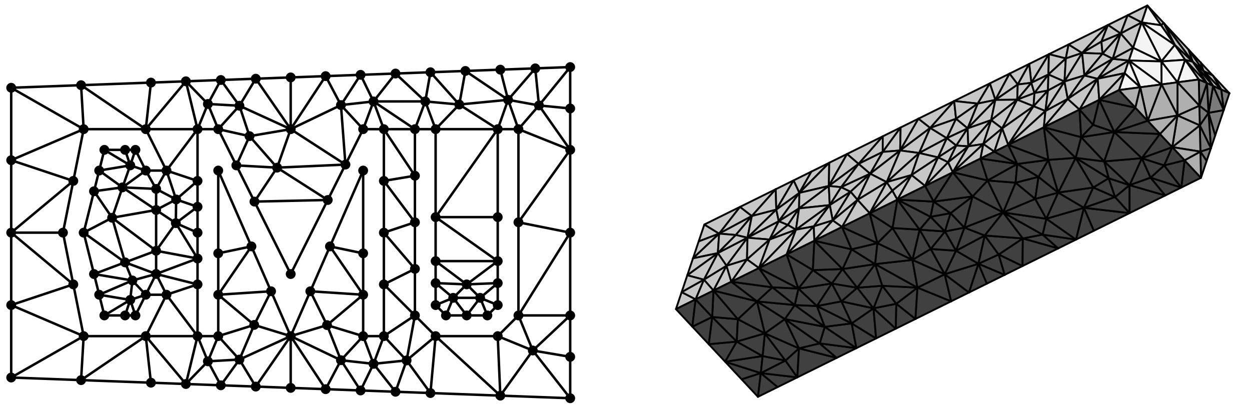


Figure 1.1: Two and three-dimensional finite element meshes. At left, each triangle is an element. At right, each tetrahedron is an element.

1.1 Meshes and Numerical Methods

Many physical phenomena in science and engineering can be modeled by partial differential equations (PDEs). When these equations have complicated boundary conditions or are posed on irregularly shaped objects or domains, they usually do not admit closed-form solutions. A numerical approximation of the solution is thus necessary.

Numerical methods for solving PDEs include the *finite element method* (FEM), the *finite volume method* (FVM, also known as the *control volume method*), and the *boundary element method* (BEM). They are used to model disparate phenomena such as mechanical deformation, heat transfer, fluid flow, electromagnetic wave propagation, and quantum mechanics. These methods numerically approximate the solution of a linear or nonlinear PDE by replacing the continuous system with a finite number of coupled linear or nonlinear algebraic equations. This process of *discretization* associates a variable with each of a finite number of points in the problem domain. For instance, to simulate heat conduction through an electrical component, the temperature is recorded at a number of points, called *nodes*, on the surface and in the interior of the component.

It is not enough to choose a set of points to act as nodes; the problem domain (or in the BEM, the boundary of the problem domain) must be partitioned into small pieces of simple shape. In the FEM, these pieces are called *elements*, and are usually triangles or quadrilaterals (in two dimensions), or tetrahedra or hexahedral bricks (in three dimensions). The FEM employs a node at every element vertex (and sometimes at other locations); each node is typically shared among several elements. The collection of nodes and elements is called a *finite element mesh*. Two and three-dimensional finite element meshes are illustrated in Figure 1.1. Because elements have simple shapes, it is easy to approximate the behavior of a PDE, such as the heat equation, on each element. By accumulating these effects over all the elements, one derives a system of equations whose solution approximates a set of physical quantities such as the temperature at each node.

The FVM and the BEM also use meshes, albeit with differences in terminology and differences in the meshes themselves. Finite volume meshes are composed of *control volumes*, which sometimes are clusters of triangles or tetrahedra, and sometimes are the cells of a geometric structure known as the *Voronoi diagram*. In either case, an underlying simplicial mesh is typically used to interpolate the nodal values and to generate the control volumes. Boundary element meshes do not partition an object; only its boundaries are partitioned. Hence, a two-dimensional domain would have boundaries divided into straight-line elements,

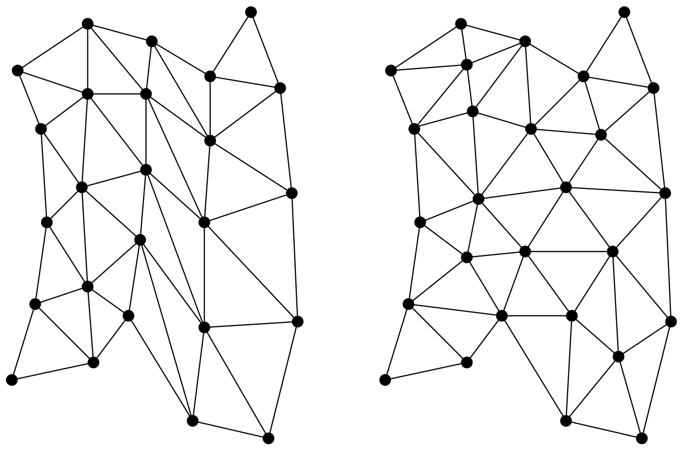


Figure 1.2: Structured (left) and unstructured (right) meshes. The structured mesh has the same topology as a square grid of triangles, although it is deformed enough that one might fail to notice its structure.

and a three-dimensional domain would have boundaries partitioned into polygonal (typically triangular) elements.

Meshes can (usually) be categorized as structured or unstructured. Figure 1.2 illustrates an example of each. Structured meshes exhibit a uniform topological structure that unstructured meshes lack. A functional definition is that in a structured mesh, the indices of the neighbors of any node can be calculated using simple addition, whereas an unstructured mesh necessitates the storage of a list of each node's neighbors.

The generation of both structured and unstructured meshes can be surprisingly difficult, each posing challenges of their own. This document considers only the task of generating unstructured meshes, and furthermore considers only simplicial meshes, composed of triangles or tetrahedra. Meshes with quadrilateral, hexahedral, or other non-simplicial elements are passed over, although they comprise an interesting field of study in their own right.

1.2 Desirable Properties of Meshes and Mesh Generators

Unfortunately, discretizing one's object of simulation is a more difficult problem than it appears at first glance. A useful mesh satisfies constraints that sometimes seem almost contradictory. A mesh must conform to the object or domain being modeled, and ideally should meet constraints on both the size and shape of its elements.

Consider first the goal of correctly modeling the shape of a problem domain. Scientists and engineers often wish to model objects or domains with complex shapes, and possibly with curved surfaces. Boundaries may appear in the interior of a region as well as on its exterior surfaces. *Exterior boundaries* separate meshed and unmeshed portions of space, and are found on the outer surface and in internal holes of a mesh. *Interior boundaries* appear within meshed portions of space, and enforce the constraint that elements may not pierce them. These boundaries are typically used to separate regions that have different physical properties; for example, at the contact plane between two materials of different conductivities in a heat propagation problem. An interior boundary is represented by a collection of edges (in two dimensions) or faces (in three dimensions) of the mesh.

In practice, curved boundaries can often be approximated by piecewise linear boundaries, so theoretical mesh generation algorithms are often based upon the idealized assumption that the input geometry is piecewise linear—composed without curves. This assumption is maintained throughout this document, and curved surfaces will not be given further consideration. This is not to say that the problem of handling curves is so easily waved aside; it surely deserves study. However, the simplified problem is difficult enough to provide ample gristle for the grinder.

Given an arbitrary straight-line two-dimensional region, it is not difficult to generate a triangulation that conforms to the shape of the region. It is trickier to find a tetrahedralization that conforms to an arbitrary linear three-dimensional region; some of the fundamental difficulties of doing so are described in Section 2.1.3. Nevertheless, the problem is reasonably well understood, and a thorough survey of the pertinent techniques, in both two and three dimensions, is offered by Bern and Eppstein [10].

A second goal of mesh generation is to offer as much control as possible over the sizes of elements in the mesh. Ideally, this control includes the ability to grade from small to large elements over a relatively short distance. The reason for this requirement is that element size has two effects on a finite element simulation. Small, densely packed elements offer more accuracy than larger, sparsely packed elements; but the computation time required to solve a problem is proportional to the number of elements. Hence, choosing an element size entails trading off speed and accuracy. Furthermore, the element size required to attain a given amount of accuracy depends upon the behavior of the physical phenomena being modeled, and may vary throughout the problem domain. For instance, a fluid flow simulation requires smaller elements amid turbulence than in areas of relative quiescence; in three dimensions, the ideal element in one part of the mesh may vary in volume by a factor of a million or more from the ideal element in another part of the mesh. If elements of uniform size are used throughout the mesh, one must choose a size small enough to guarantee sufficient accuracy in the most demanding portion of the problem domain, and thereby possibly incur excessively large computational demands. To avoid this pitfall, a mesh generator should offer rapid gradation from small to large sizes.

Given a *coarse* mesh—one with relatively few elements—it is not difficult to *refine* it to produce another mesh having a larger number of smaller elements. The reverse process is not so easy. Hence, mesh generation algorithms often set themselves the goal of being able, in principle, to generate as small a mesh as possible. (By "small", I mean one with as few elements as possible.) They typically offer the option to refine portions of the mesh whose elements are not small enough to yield the required accuracy.

A third goal of mesh generation, and the real difficulty, is that the elements should be relatively "round" in shape, because elements with large or small angles can degrade the quality of the numerical solution.

Elements with large angles can cause a large *discretization error*; the solution yielded by a numerical method such as the finite element method may be far less accurate than the method would normally promise. In principle, the computed discrete solution should approach the exact solution of the PDE as the element size approaches zero. However, Babuška and Aziz [3] show that if mesh angles approach 180° as the element size decreases, convergence to the exact solution may fail to occur.

Another problem caused by large angles is large errors in derivatives of the solution, which arise as an artifact of interpolation over the mesh. Figure 1.3 demonstrates the problem. The element illustrated has values associated with its nodes that represent an approximation of some physical quantity. If linear interpolation is used to estimate the solution at non-nodal points, the interpolated value at the center of the bottom edge is 51, as illustrated. This interpolated value depends only on the values associated with the bottom two nodes, and is independent of the value associated with the upper node. As the angle at the upper node approaches 180°, the interpolated point (with value 51) becomes arbitrarily close to the upper node (with value 48). Hence, the directional derivative of the estimated solution in the vertical direction may become arbitrarily large, and is clearly specious, even though the nodal values may themselves be perfectly accurate. This effect occurs because a linearly interpolated value is necessarily in error if the true solution

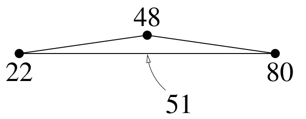


Figure 1.3: The nodal values depicted may represent an accurate estimate of the correct solution. Nevertheless, as the large angle of this element approaches $180°$, the vertical directional derivative, estimated via linear interpolation, becomes arbitrarily large.

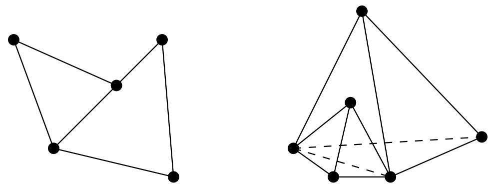


Figure 1.4: Elements are not permitted to meet in the manner depicted here.

is not linear, and any error is magnified in the derivative computation because of the large angle. This problem can afflict any application that uses meshes for interpolation, and not just PDE solvers. However, the problem is of particular concern in simulations of mechanical deformation, in which the derivatives of a solution (the strains) are of interest, and not the solution itself (the displacements).

Small angles are also feared, because they can cause the coupled systems of algebraic equations that numerical methods yield to be ill-conditioned [16]. If a system of equations is ill-conditioned, roundoff error degrades the accuracy of the solution if the system is solved by direct methods, and convergence is slow if the system is solved by iterative methods.

By placing a lower bound on the smallest angle of a triangulation, one is also bounding the largest angle; for instance, in two dimensions, if no angle is smaller than θ, then no angle is larger than $180° - 2\theta$. Hence, many mesh generation algorithms take the approach of attempting to bound the smallest angle.

Despite this discussion, the effects of element shape on numerical methods such as the finite element method are still being investigated. Our understanding of the relative merit of different metrics for measuring element quality, or the effects of small numbers of poor quality elements on numerical solutions, is based as much on engineering experience and rumor as it is on mathematical foundations. Furthermore, the notion of a nicely shaped element varies depending on the numerical method, the type of problem being solved, and the polynomial degree of the piecewise functions used to interpolate the solution over the mesh. For physical phenomena that have anisotropic behavior, the ideal element may be long and thin, despite the claim that small angles are usually bad. Hence, the designer of algorithms for mesh generation is shooting at an ill-defined target.

The constraints of element size and element shape are difficult to reconcile because elements must meet squarely along the full extent of their shared edges or faces. Figure 1.4 illustrates illegal meetings between adjacent elements. For instance, at left, the edge of one triangular element is a portion of an edge of an adjoining element. There are variants of methods like the finite element method that permit such *nonconforming elements*. However, such elements are not preferred, as they may degrade or ruin the convergence of the method. Although nonconforming elements make it easier to create a mesh with seemingly nicely shaped elements, the problems of numerical error may still persist.

For an example of how element quality and mesh size are traded off, look ahead to Figure 3.19 on Page 61.

1.3 Why Unstructured Meshes?

Is it really worth the trouble to use unstructured meshes? The process of solving the linear or nonlinear systems of equations yielded by the finite element method and its brethren is simpler and faster on structured meshes, because of the ease of determining each node's neighbors. Because unstructured meshes necessitate the storage of pointers to each node's neighbors, their demands on storage space and memory traffic are greater. Furthermore, the regularity of structured meshes makes it straightforward to parallelize computations upon them, whereas unstructured meshes engender the need for sophisticated partitioning algorithms and parallel unstructured solvers.

Nonetheless, there are cases in which unstructured meshes are preferable or even indispensable. Many problems are defined on irregularly shaped domains, and resist structured discretization. Several more subtle advantages of unstructured meshes are visible in Figures 1.6 and 1.7, which depict meshes used to model a cross-section of the Los Angeles Basin, itself illustrated in Figure 1.5.

A numerical method is used to predict the surface ground motion due to a strong earthquake. The mesh of Figure 1.7 is finer in the top layers of the valley, reflecting the much smaller wavelength of seismic waves in the softer upper soil, and becomes coarser with increasing depth, as the soil becomes stiffer and the corresponding seismic wavelength increases by a factor of twenty. Whereas an unstructured mesh can be flexibly tailored to the physics of this problem, the structured mesh must employ a uniform horizontal distribution of nodes, the density being dictated by the uppermost layer. As a result, it has five times as many nodes as the unstructured mesh, and the solution time and memory requirements of the simulation are correspondingly larger. The disparity is even more pronounced in three-dimensional domains and in simulations where the scales of the physical phenomena vary more.

Another important difference is that the mesh of Figure 1.7 conforms to the interior boundaries of the basin in a way that the mesh of Figure 1.6 cannot, and hence may better model reflections of waves from the interfaces between layers of soil with differing densities. This difference in accuracy only manifests itself if the unstructured and structured meshes under comparison are relatively coarse.

Unstructured meshes, far better than structured meshes, can provide multiscale resolution and conformity to complex geometries.

1.4 Outline of the Thesis

The central topic of this thesis is the study of a technique, called *Delaunay refinement*, for the generation of triangular and tetrahedral meshes. Delaunay refinement methods are based upon a well-known geometric construction called the *Delaunay triangulation*, which is discussed extensively in Chapter 2.

Chapter 2 also briefly surveys some of the previous research on simplicial mesh generation. Algorithms based upon the Delaunay triangulation are discussed. So are several fundamentally different algorithms, some of which are distinguished by having provably good bounds on the quality of the meshes they produce. There are several types of bounds an algorithm might have; for instance, quite a few mesh generation algorithms produce provably good elements. In other words, some quality measure—usually the smallest or largest angle—of every element is constrained by some minimum or maximum bound. Some of these algorithms also offer bounds on the sizes of the meshes they generate. For some, it is possible to prove that the meshes are nicely graded, in a mathematically well-defined sense that is explained in Chapter 3. Roughly speaking, the presence of small elements in one portion of the mesh does not have an unduly strong effect on the sizes of elements in another nearby portion of the mesh. One should be aware that the theoretical

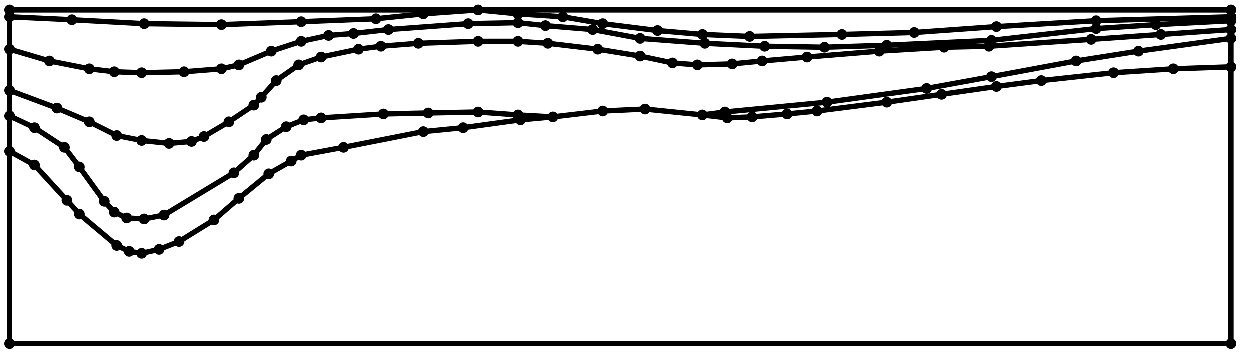


Figure 1.5: Los Angeles Basin.

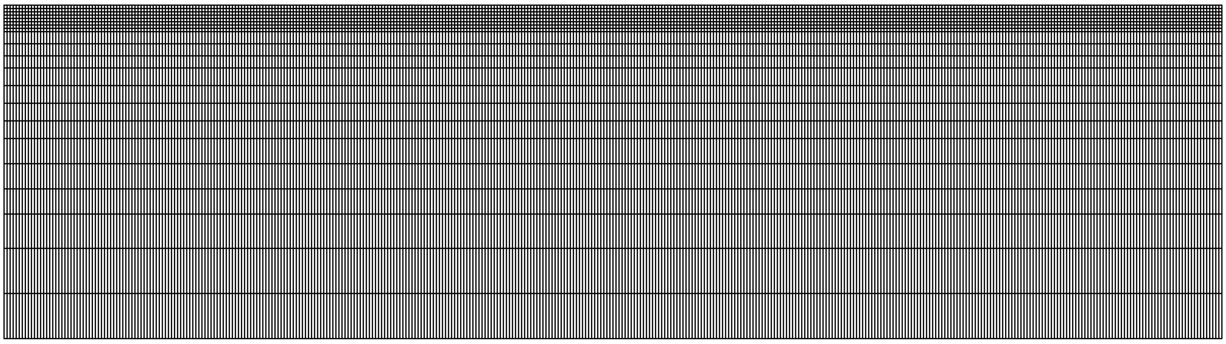


Figure 1.6: Structured mesh of Los Angeles Basin.

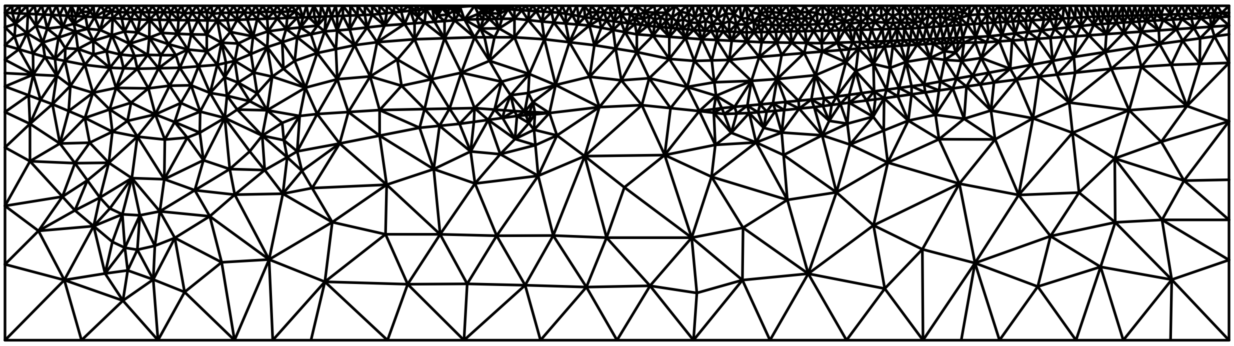


Figure 1.7: Unstructured mesh of Los Angeles Basin.

bounds promised by mesh generation algorithms are not in every case strong enough to be useful guarantees in practice, but some of these algorithms do much better in practice than their theoretical bounds suggest.

Jim Ruppert and L. Paul Chew have developed two-dimensional Delaunay refinement algorithms that exhibit provable bounds on element quality, mesh grading, and mesh size; these algorithms are effective in practice as well. In Chapter 3, I review these algorithms, unify them, and solve an outstanding problem related to inputs with small angles.

To clarify the relationship between these algorithms (including my own modifications), I list here the provable bounds on each of these algorithms prior and subsequent to the present research. Chew's first Delaunay refinement algorithm [19], published as a technical report in 1989, was the first Delaunay refinement algorithm to offer a guarantee: it produces meshes with no angle smaller than 30°. The elements of these meshes are of uniform size, however; grading of element sizes is not offered. Ruppert's Delaunay refinement algorithm [82], first published as a technical report in 1992 [80], offers different guarantees. Although it promises only a minimum angle of roughly 20.7°, it also offers a guarantee of good grading, which in turn can be used to prove that the algorithm is *size-optimal*: the number of elements in the final mesh is at most a constant factor larger than the number in the best possible mesh that meets the same bound on minimum angle. Chew published a second Delaunay refinement algorithm [21] in 1993, which offers the same 30° lower bound as his first algorithm. Chew's second algorithm produces nicely graded meshes in practice, although Chew provides no theoretical guarantee of this behavior.

Ruppert's algorithm and Chew's second algorithm can take a minimum angle as a parameter, and produce a mesh with no angle smaller than that minimum. In Ruppert's algorithm, this parameter may be chosen between 0° and 20.7°. The bounds on grading and size-optimality are stronger for smaller minimum angles. As the minimum angle increases to 20.7°, the other bounds become progressively weaker. In practice, both Ruppert's algorithm and Chew's second algorithm exhibit a tradeoff between element quality and mesh size, but allow better angle bounds than the theory predicts. (Again, see Figure 3.19 for an example of the tradeoff in Ruppert's algorithm.)

My new results in two-dimensional mesh generation, also detailed in Chapter 3, are as follows. I show that Ruppert's analysis technique can be applied to Chew's second algorithm, and I thereby prove that Chew's second algorithm produces nicely graded meshes for minimum angles of up to roughly 26.5°. Hence, if a user specifies a minimum angle no greater than 26.5°, good grading and size-optimality are guaranteed. (Observe that this improves upon the 20.7° bound of Ruppert's algorithm.) If a minimum angle between 26.5° and 30° is specified, termination is still guaranteed (by Chew's own result), but good grading and size-optimality are not theoretically guaranteed (although they are exhibited in practice). I also introduce the notion of *range-restricted segment splitting*, which extends an idea of Chew. Ruppert's algorithm, modified to use range-restricted segment splitting, is guaranteed to terminate for minimum angles up to 30°, like Chew's algorithm.

Ruppert's and Chew's algorithms are not entirely satisfying because their theoretical guarantees do not apply when the problem domain has small angles. In this circumstance, their behavior is poor in practice as well; they may even fail to terminate. This problem reflects not merely a deficiency of the algorithms, but a fundamental difficulty in triangular mesh generation. Although small angles inherent in the input geometry cannot be removed, one would like to find a way to triangulate a problem domain without creating any *new* small angles. I prove that this problem is not always soluble. For instance, I can exhibit an input that bears an angle of half a degree, and cannot be triangulated without adding a new angle smaller than 30°. Similarly, for any angle θ, however small, I can exhibit an input that cannot be triangulated without creating a new angle smaller than θ. (The input I exhibit has a small angle which itself is much smaller than θ.)

This negative result implies that Ruppert's algorithm will never terminate on such an input; it will ceaselessly try to rid itself of removable small angles, only to find the culprits replaced by others. I propose a modification to the algorithm that prevents this cycle of endless refinement; termination is guaranteed. A few bad angles must necessarily remain in the mesh, but these appear only near small input angles. The modification does not affect the behavior of the algorithm on inputs with no small angles.

Based on these foundations, I design a three-dimensional Delaunay refinement algorithm in Chapter 4. This chapter is the climax of the thesis, although its results are the simplest to outline. I first extend Ruppert's algorithm to three dimensions, and show that the extension generates nicely graded tetrahedral meshes whose circumradius-to-shortest edge ratios are nearly bounded below two. By adopting two modifications to the algorithm, *equatorial lenses* and *range-restricted segment splitting*, the bound on each element's circumradius-to-shortest edge ratio can be improved to 1.63 with a guarantee of good grading, or to 1.15 without. (Meshes generated with a bound of 1.15 exhibit good grading in practice, even if there is no theoretical guarantee.)

A bound on the circumradius-to-shortest edge ratio of a tetrahedron is helpful, but does not imply any bound on the minimum or maximum dihedral angle. However, some numerical methods, including the finite element method, require such bounds to ensure numerical accuracy. The Delaunay refinement algorithm is easily modified to generate meshes wherein all tetrahedra meet some bound on their minimum angle. Termination can no longer be guaranteed in theory, but is obtained in practice for reasonable angle bounds.

The main shortcoming of my three-dimensional Delaunay refinement algorithm is that severe restrictions are made that outlaw small angles in the input geometry. One would like to have methods for handling small input angles similar to those I have developed for the two-dimensional case. I am optimistic that such methods will be found, but I do not discuss the problem in any depth herein.

I have implemented both the two-dimensional and three-dimensional Delaunay refinement algorithms. A great deal of care is necessary to turn these algorithms into practical mesh generators. My thoughts on the choice of data structures, triangulation algorithms, and other implementation details are found in Chapter 5.

Although nearly all numerical algorithms are affected by floating-point roundoff error, there are fundamental reasons why geometric algorithms are particularly susceptible. In ordinary numerical algorithms, the most common problem due to roundoff error is inaccurate results, whereas in computational geometry, a common result is outright failure to produce any results at all. In many numerical algorithms, problems due to roundoff error can be eliminated by careful numerical analysis and algorithm design. Geometric algorithms yield to such an approach with greater difficulty, and the only easy way to ensure geometric robustness is through the use of exact arithmetic.

Unfortunately, exact arithmetic is expensive, and can slow geometric algorithms considerably. Chapter 6 details my contributions to the solution of this problem. My approach is based firstly upon a new fast technique for performing exact floating-point arithmetic using standard floating-point units, and secondly upon a method for performing these computations adaptively, spending only as much time as is necessary to ensure the integrity of the result. Using these two techniques, I have written several geometric predicates that greatly improve the robustness of my mesh generators, and are useful in other geometric applications as well.

Chapter 2

The Delaunay Triangulation and Mesh Generation

The Delaunay triangulation is a geometric structure that has enjoyed great popularity in mesh generation since mesh generation was in its infancy. In two dimensions, it is not hard to understand why: the Delaunay triangulation of a vertex set maximizes the minimum angle among all possible triangulations of that vertex set. If one is concerned with element quality, it seems almost silly to consider using a triangulation that is not Delaunay.

This chapter surveys Delaunay triangulations, their properties, and several algorithms for constructing them. I focus only on details relevant to mesh generation; for more general surveys, Aurenhammer [1], Bern and Eppstein [10], and Fortune [33] are recommended. I also discuss two generalizations of the Delaunay triangulation: the constrained Delaunay triangulation, which ensures that input segments are present in the mesh, and the Delaunay tetrahedralization, which generalizes the Delaunay triangulation to three dimensions. The Delaunay tetrahedralization is not quite so effective as the Delaunay triangulation at producing elements of good quality, but it has nevertheless enjoyed nearly as much popularity in the mesh generation literature as its two-dimensional cousin.

Also found in this chapter is a brief survey of research in mesh generation, with special attention given to methods based on Delaunay triangulations and tetrahedralizations, and methods that generate meshes that are guaranteed to have favorable qualities. These algorithms are part of the history that led to the discovery of the provably good Delaunay refinement algorithms studied in Chapters 3 and 4.

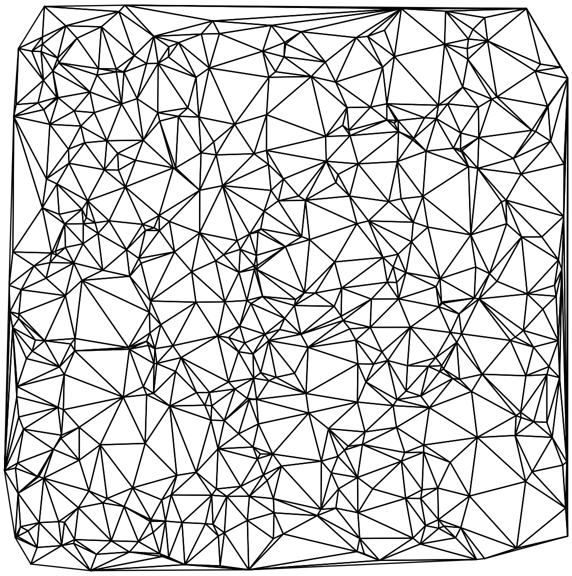


Figure 2.1: A Delaunay triangulation.

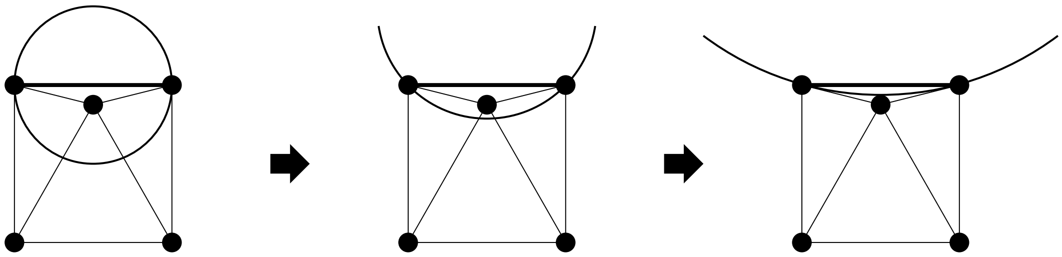


Figure 2.2: Each edge on the convex hull is Delaunay, because it is always possible to find an empty circle that passes through its endpoints.

2.1 Delaunay Triangulations and Tetrahedralizations

2.1.1 The Delaunay Triangulation

In two dimensions, a *triangulation* of a set V of vertices is a set T of triangles whose vertices collectively form V, whose interiors do not intersect each other, and whose union completely fills the convex hull of V.

The *Delaunay triangulation* D of V, introduced by Delaunay [27] in 1934, is the graph defined as follows. Any circle in the plane is said to be *empty* if it contains no vertex of V in its interior. (Vertices are permitted on the circle.) Let u and v be any two vertices of V. The edge uv is in D if and only if there exists an empty circle that passes through u and v. An edge satisfying this property is said to be *Delaunay*. Figure 2.1 illustrates a Delaunay triangulation.

The Delaunay triangulation of a vertex set is clearly unique, because the definition given above specifies an unambiguous test for the presence or absence of an edge in the triangulation. Every edge of the convex hull of a vertex set is Delaunay. Figure 2.2 illustrates the reason why. For any convex hull edge e, it is always possible to find an empty circle that contains e by starting with the smallest containing circle of e and "growing" it away from the triangulation.

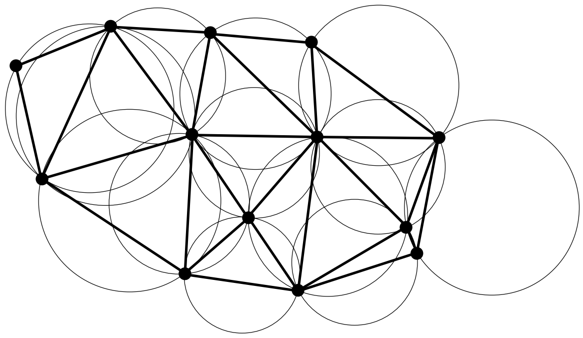


Figure 2.3: Every triangle of a Delaunay triangulation has an empty circumcircle.

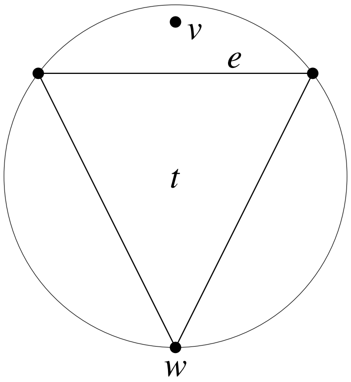


Figure 2.4: If the triangle t is not Delaunay, then at least one of its edges (in this case, e) is not Delaunay.

Every edge connecting a vertex to its nearest neighbor is Delaunay. If w is the vertex nearest v, the smallest circle passing through v and w does not contain any vertices.

It's not at all obvious that the set of Delaunay edges of a vertex set collectively forms a triangulation. For the definition I have given above, the Delaunay triangulation is guaranteed to be a triangulation only if the vertices of V are in *general position*, here meaning that no four vertices of V lie on a common circle. As a first step to proving this guarantee, I describe the notion of a Delaunay triangle. The *circumcircle* of a triangle is the unique circle that passes through all three of its vertices. A triangle is said to be *Delaunay* if and only if its circumcircle is empty. This defining characteristic of Delaunay triangles, illustrated in Figure 2.3, is called the *empty circumcircle property*.

Lemma 1 *Let T be a triangulation. If all the triangles of T are Delaunay, then all the edges of T are Delaunay, and vice versa.*

Proof: If all the triangles of T are Delaunay, then the circumcircle of every triangle is empty. Because every edge of T belongs to a triangle of T, every edge is contained in an empty circle, and is thus Delaunay.

If all the edges of T are Delaunay, suppose for the sake of contradiction that some triangle t of T is not Delaunay. Because T is a triangulation, t cannot contain any vertices (except its corners), so some vertex v of T lies inside the circumcircle of t, but outside t itself. Let e be the edge of t that separates v from the interior of t, and let w be the vertex of t opposite e, as illustrated in Figure 2.4. One cannot draw a containing circle of e that contains neither v nor w, so e is not Delaunay. The result follows by contradiction. ■

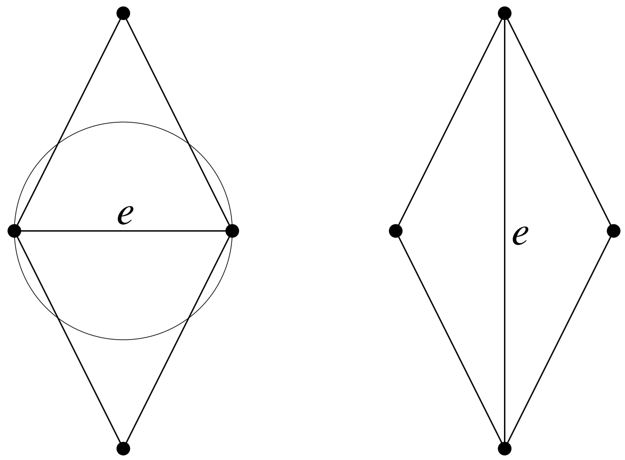


Figure 2.5: Two triangulations of a vertex set. At left, e is locally Delaunay; at right, e is not.

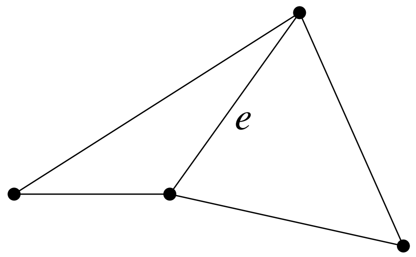


Figure 2.6: In this concave quadrilateral, e cannot be flipped.

The method by which I prove that the Delaunay triangulation is a triangulation is somewhat nonintuitive. I will describe a well-known algorithm called the *flip algorithm*, and show that all the edges of the triangulation produced by the flip algorithm are Delaunay. Then I will show that no other edges are Delaunay.

The flip algorithm begins with an arbitrary triangulation, and searches for an edge that is not *locally Delaunay*. All edges on the boundary (convex hull) of the triangulation are considered to be locally Delaunay. For any edge e not on the boundary, the condition of being locally Delaunay is similar to the condition of being Delaunay, but only the two triangles that contain e are considered. For instance, Figure 2.5 demonstrates two different ways to triangulate a subset of four vertices. In the triangulation at left, the edge e is locally Delaunay, because the depicted containing circle of e does not contain either of the vertices opposite e in the two triangles that contain e. In the triangulation at right, e is not locally Delaunay, because the two vertices opposite e preclude the possibility that e has an empty containing circle. Observe that if the triangles at left are part of a larger triangulation, e might not be Delaunay, because vertices may lie in the containing circle, although they lie in neither triangle. However, such vertices have no bearing on whether or not e is locally Delaunay.

Whenever the flip algorithm identifies an edge that is not locally Delaunay, the edge is *flipped*. To flip an edge is to delete it, thereby combining the two containing triangles into a single *containing quadrilateral*, and then to insert the crossing edge of the quadrilateral. Hence, an edge flip could convert the triangulation at left in Figure 2.5 into the triangulation at right, or vice versa. (The flip algorithm would perform only the latter flip.) Not all triangulation edges are flippable, as Figure 2.6 shows, because the containing quadrilateral of an edge might not be convex.

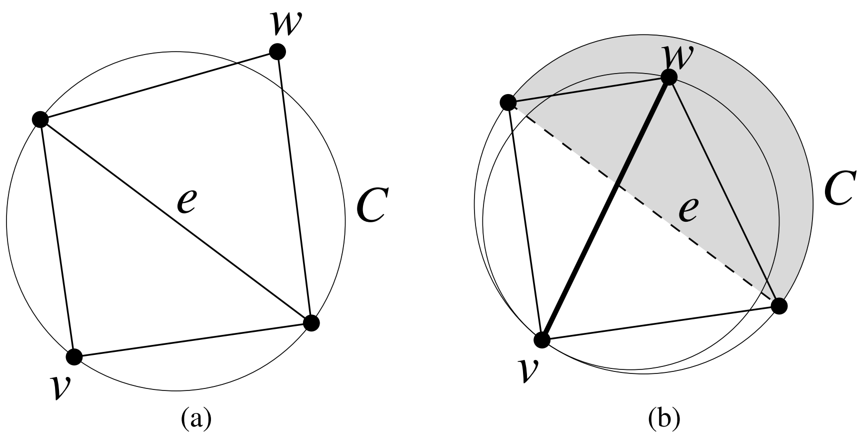


Figure 2.7: (a) Case where e is locally Delaunay. (b) Case where e is not locally Delaunay. The edge created if e is flipped is locally Delaunay.

Lemma 2 *Let* e *be an edge of a triangulation of* V*. Either* e *is locally Delaunay, or* e *is flippable and the edge created by flipping* e *is locally Delaunay.*

Proof: Let v and w be the vertices opposite e, which together with e define the containing quadrilateral of e, illustrated in Figure 2.7. Let C be the circle that passes through v and the endpoints of e. Either w is strictly inside C, or w lies on or outside C.

If w is on or outside C, as in Figure 2.7(a), then the empty circle C demonstrates that e is locally Delaunay.

If w is inside C, then w is contained in the section of C defined by e and opposite v; this section is shaded in Figure 2.7(b). The containing quadrilateral of e is thus constrained to be strictly convex, and the edge e is flippable. Furthermore, the circle that passes through v and w, and is tangent to C at v, does not contain the endpoints of e, as Figure 2.7(b) demonstrates; hence the edge vw is locally Delaunay. ■

The success of the flip algorithm relies on the fact, proven below, that if any edge of the triangulation is not Delaunay, then there is an edge that is not locally Delaunay, and can thus be flipped.

Lemma 3 *Let* T *be a triangulation whose edges are all locally Delaunay. Then every edge of* T *is (globally) Delaunay.*

Proof: Suppose for the sake of contradiction that all edges of T are locally Delaunay, but some edge of T is not Delaunay. By Lemma 1, the latter assertion implies that some triangle t of T is not Delaunay. Let v be a vertex inside the circumcircle of t, and let e_1 be the edge of t that separates v from the interior of t, as illustrated in Figure 2.8(a). Without loss of generality, assume that e_1 is oriented horizontally, with t below e_1.

Draw a line segment from the midpoint of e_1 to v (see the dashed line in Figure 2.8(a)). Let e_1, e_2, e_3, $\ldots$, e_m be the sequence of triangulation edges (from bottom to top) whose interiors this line segment intersects. (If the line segment intersects some vertex other than v, replace v with the first such vertex.) Let w_i be the vertex above e_i that forms a triangle t_i in conjunction with e_i. Because T is a triangulation, $w_m = v$.

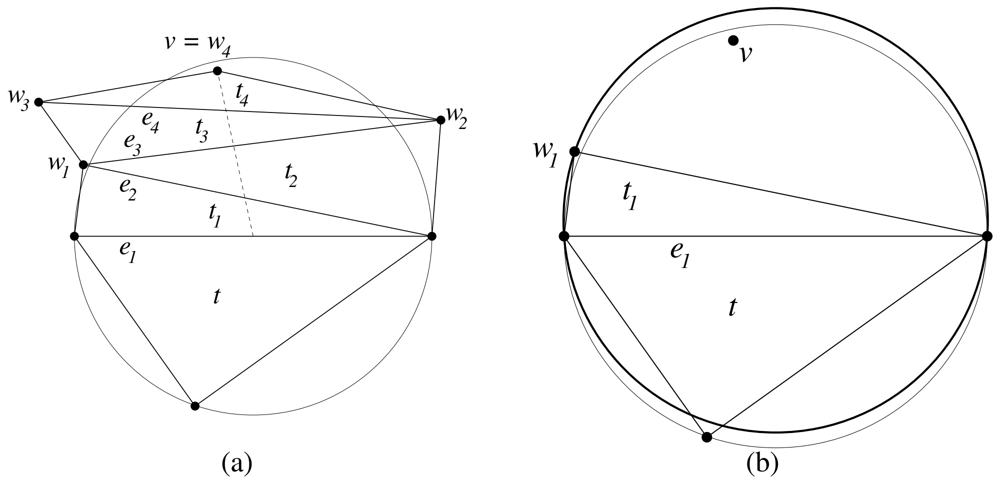


Figure 2.8: (a) If v lies inside the circumcircle of t, there must be an edge between v and t that is not locally Delaunay. (b) Because v lies above e_1 and inside the circumcircle of t, and because w_1 lies outside the circumcircle of t, v must lie inside the circumcircle of t_1.

By assumption, e_1 is locally Delaunay, so w_1 lies outside the circumcircle of t. As Figure 2.8(b) shows, it follows that the circumcircle of t_1 contains every point above e_1 in the circumcircle of t, and hence contains v. Repeating this argument inductively, one finds that the circumcircle of t_m contains v in its interior. But $w_m = v$ is a vertex of t_m, which contradicts the claim that v is in the interior of the circumcircle of t_m. ∎

An immediate consequence of Lemma 3 is that if a triangulation contains an edge that is not Delaunay, then it contains an edge that is not locally Delaunay, and thus the flip algorithm may proceed. The following lemma shows that the flip algorithm cannot become trapped in an endless loop.

Lemma 4 *Given a triangulation of n vertices, the flip algorithm terminates after $\mathcal{O}(n^2)$ edge flips, yielding a triangulation whose edges are all Delaunay.*

Proof: Let $\Phi(T)$ be a function defined over all triangulations, equal to the number of vertex-triangle pairs (v, t) such that v is a vertex of T, t is a triangle of T, and v lies inside the circumcircle of t. Because T has n vertices and $\mathcal{O}(n)$ triangles, $\Phi(T) \in \mathcal{O}(n^2)$.

Suppose an edge e of T is flipped, forming a new triangulation T'. Let t_1 and t_2 be the triangles containing e, and let v_1 and v_2 be the apices of t_1 and t_2. Because e is not locally Delaunay, v_1 is contained in the circumcircle of t_2, and v_2 is contained in the circumcircle of t_1. Let t_1' and t_2' be the triangles that replace t_1 and t_2 after the edge flip. Let C_1, C_2, C_1', and C_2' be the circumcircles of t_1, t_2, t_1', and t_2' respectively, as illustrated in Figure 2.9(a).

It is not difficult to show that $C_1 \cup C_2 \supset C_1' \cup C_2'$ (Figure 2.9(b)) and $C_1 \cap C_2 \supset C_1' \cap C_2'$ (Figure 2.9(c)). Therefore, if a vertex v lies inside n_v circumcircles of triangles of T, and hence contributes n_v to the count $\Phi(T)$, then v lies inside no more than n_v circumcircles of triangles of T', and contributes at most n_v to the count $\Phi(T')$. If, after the edge flip, a vertex is counted because it lies in C_1' or C_2', then it must have lain in C_1 or C_2 before the edge flip; and if it lies in both C_1' and C_2', then it must have lain in both C_1 and C_2.

However, the vertices v_1 and v_2 each lie in one less circumcircle than before the edge flip. For instance, v_1 lay in C_2, but lies in neither C_1' nor C_2'. Hence, $\Phi(T') \leq \Phi(T) - 2$.

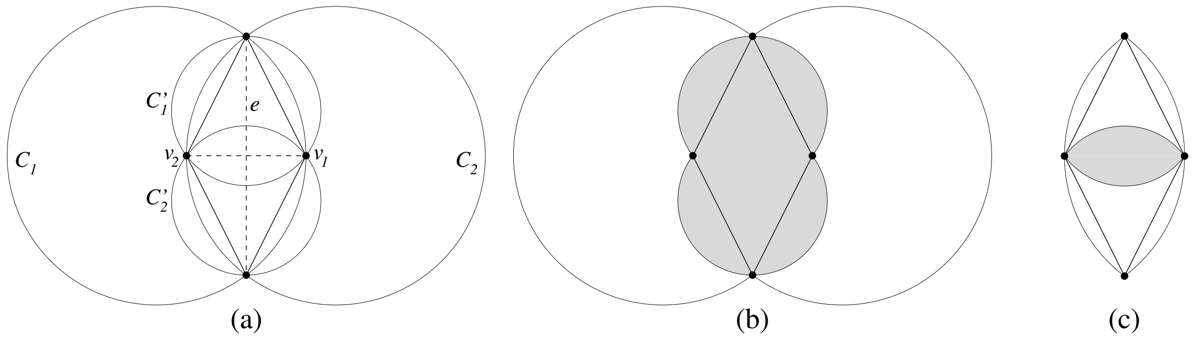


Figure 2.9: (a) Circumcircles before and after an edge flip. (b) The union of the circumcircles afterward (shaded) is contained in the union of the prior circumcircles. (c) The intersection of the circumcircles afterward (shaded) is contained in the intersection of the prior circumcircles.

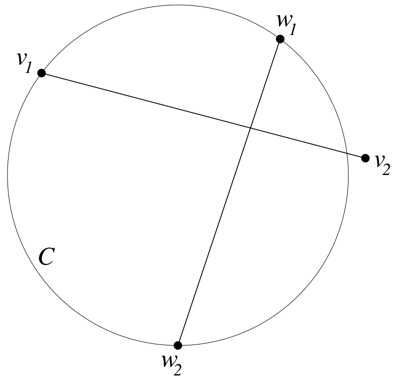


Figure 2.10: If no four vertices are cocircular, two crossing edges cannot both be Delaunay.

The flip algorithm terminates after $\mathcal{O}(n^2)$ edge flips because $\Phi \in \mathcal{O}(n^2)$, every edge flip reduces Φ by at least two, and Φ cannot fall below zero. The flip algorithm terminates only when every edge is locally Delaunay; thus, by Lemma 3, every edge is Delaunay. ■

Theorem 5 *Let V be a set of three or more vertices in the plane that are not all collinear. If no four vertices of V are cocircular, the Delaunay triangulation of V is a triangulation, and is produced by the flip algorithm.*

Proof: Because the vertices of V are not all collinear, there exists a triangulation of V. By Lemma 4, the application of the flip algorithm to any triangulation of V produces a triangulation D whose edges are all Delaunay.

I shall show that no other edge is Delaunay. Consider any edge $v_1v_2 \notin D$, with $v_1, v_2 \in V$. Because D is a triangulation, v_1v_2 must cross some edge $w_1w_2 \in D$. Because w_1w_2 is in D, it is Delaunay, and there is a circle C passing through w_1 and w_2 whose interior contains neither v_1 nor v_2. Because no four vertices are cocircular, at least one of v_1 and v_2 lies strictly outside C. It follows that no empty circle passes through v_1 and v_2, hence v_1v_2 is not Delaunay (see Figure 2.10).

Therefore, D is the Delaunay triangulation of V. ■

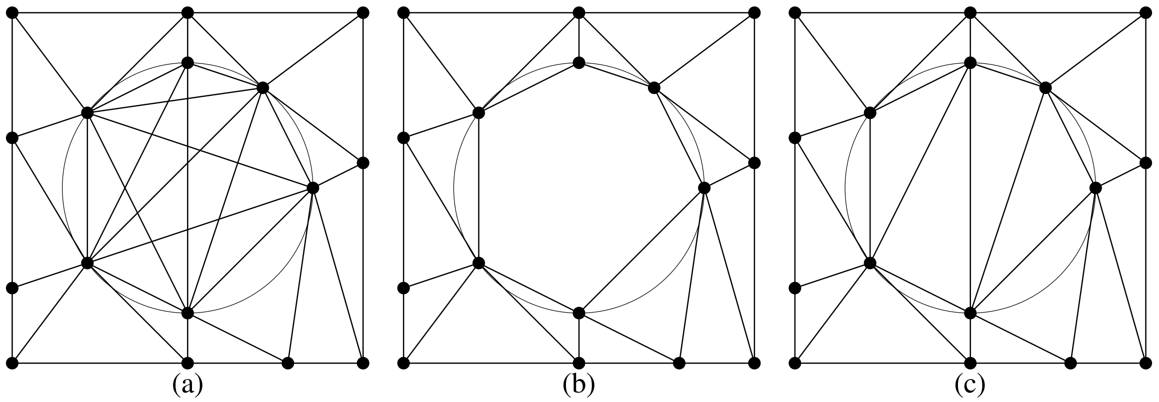


Figure 2.11: Three ways to define the Delaunay diagram in the presence of cocircular vertices. (a) Include all Delaunay edges, even if they cross. (b) Exclude all crossing Delaunay edges. (c) Choose a subset of Delaunay edges that forms a triangulation.

What if V contains cocircular vertices? In this circumstance, the Delaunay triangulation may have crossing edges, as illustrated in Figure 2.11(a). Because an arbitrarily small perturbation of the input vertices can change the topology of the triangulation, V and its Delaunay triangulation are said to be *degenerate*.

The definition of "Delaunay triangulation" is usually modified to prevent edges from crossing. Occasionally, one sees in the literature a definition wherein all such crossing edges are omitted; polygons with more than three sides may appear in the Delaunay diagram, as Figure 2.11(b) shows. (The usefulness of this definition follows in part because the graph thus defined is the geometric dual of the well-known Voronoi diagram.) For most applications, however, it is desirable to have a true triangulation, and some of the Delaunay edges (and thus, some of the Delaunay triangles) are omitted to achieve this, as in Figure 2.11(c). In this case, the Delaunay triangulation is no longer unique. The flip algorithm will find one of the Delaunay triangulations; the choice of omitted Delaunay edges depends upon the starting triangulation. Because numerical methods like the finite element method generally require a true triangulation, I will use this latter definition of "Delaunay triangulation" throughout the rest of this document.

Delaunay triangulations are valuable in part because they have the following optimality properties.

Theorem 6 *Among all triangulations of a vertex set, the Delaunay triangulation maximizes the minimum angle in the triangulation, minimizes the largest circumcircle, and minimizes the largest min-containment circle, where the* min-containment circle *of a triangle is the smallest circle that contains it.*

Proof: It can be shown that each of these properties is locally improved when an edge that is not locally Delaunay is flipped. The optimal triangulation cannot be improved, and thus has no locally Delaunay edges. By Theorem 5, a triangulation with no locally Delaunay edges is the Delaunay triangulation. ■

The property of max-min angle optimality was first noted by Lawson [59], and helps to account for the popularity of Delaunay triangulations in mesh generation. Unfortunately, neither this property nor the min-max circumcircle property generalizes to Delaunay triangulations in dimensions higher than two. The property of minimizing the largest min-containment circle was first noted by D'Azevedo and Simpson [25], and has been shown to hold for higher-dimensional Delaunay triangulations by Rajan [78].

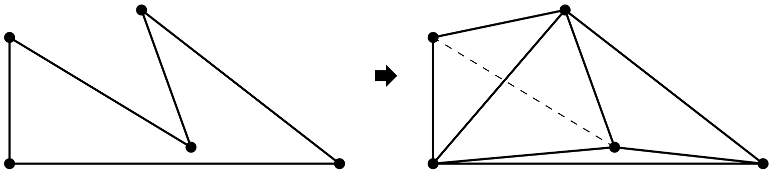


Figure 2.12: The Delaunay triangulation of a set of vertices does not usually solve the mesh generation problem, because it may contain poor quality triangles and omit some domain boundaries.

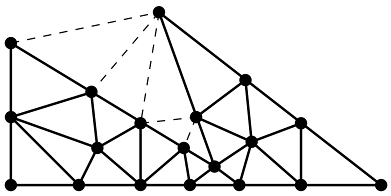


Figure 2.13: By inserting additional vertices into the triangulation, boundaries can be recovered and poor quality elements can be eliminated.

2.1.2 Planar Straight Line Graphs and Constrained Delaunay Triangulations

Given that the Delaunay triangulation of a set of vertices maximizes the minimum angle (in two dimensions), why isn't the problem of mesh generation solved? There are two reasons, both illustrated in Figure 2.12, which depicts an input object and a Delaunay triangulation of the object's vertices. The first reason is that Delaunay triangulations are oblivious to the boundaries that define an object or problem domain, and these boundaries may or may not appear in a triangulation. The second reason is that maximizing the minimum angle usually isn't good enough; for instance, the bottommost triangle of the triangulation of Figure 2.12 is quite poor.

Both of these problems can be solved by inserting additional vertices into the triangulation, as illustrated in Figure 2.13. Chapters 3 and 4 will discuss this solution in detail. Here, however, I review a different solution to the first problem that requires no additional vertices. Unfortunately, it is only applicable in two dimensions.

The usual input for two-dimensional mesh generation is not merely a set of vertices. Most theoretical treatments of meshing take as their input a *planar straight line graph* (PSLG). A PSLG is a set of vertices and segments that satisfies two constraints. First, for each segment contained in a PSLG, the PSLG must also contain the two vertices that serve as endpoints for that segment. Second, segments are permitted to intersect only at their endpoints. (A set of segments that does not satisfy this condition can be converted into a set of segments that does. Run a segment intersection algorithm [24, 85], then divide each segment into smaller segments at the points where it intersects other segments.)

The *constrained Delaunay triangulation* (CDT) of a PSLG X is similar to the Delaunay triangulation, but every input segment appears as an edge of the triangulation. An edge or triangle is said to be *constrained Delaunay* if it satisfies the following two conditions. First, its vertices are *visible* to each other. Here, visibility is deemed to be obstructed if a segment of X lies between two vertices. Second, there exists a circle that passes through the vertices of the edge or triangle in question, and the circle contains no vertices of X that are visible from the interior of the edge or triangle.

Segments of X are also considered to be constrained Delaunay.

Figure 2.14 demonstrates examples of a constrained Delaunay edge e and a constrained Delaunay trian-

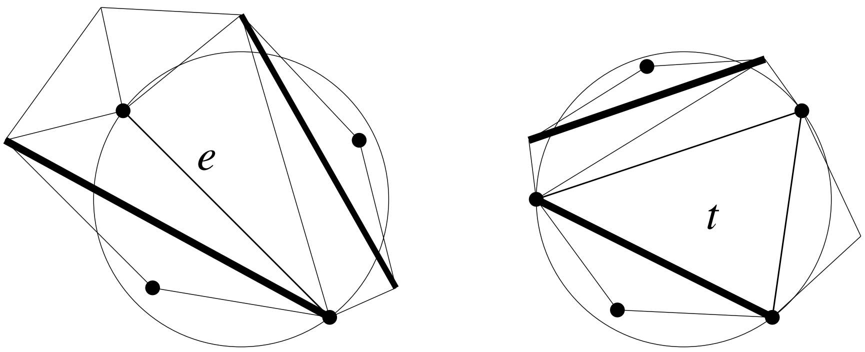


Figure 2.14: The edge e and triangle t are each constrained Delaunay. Bold lines represent segments.

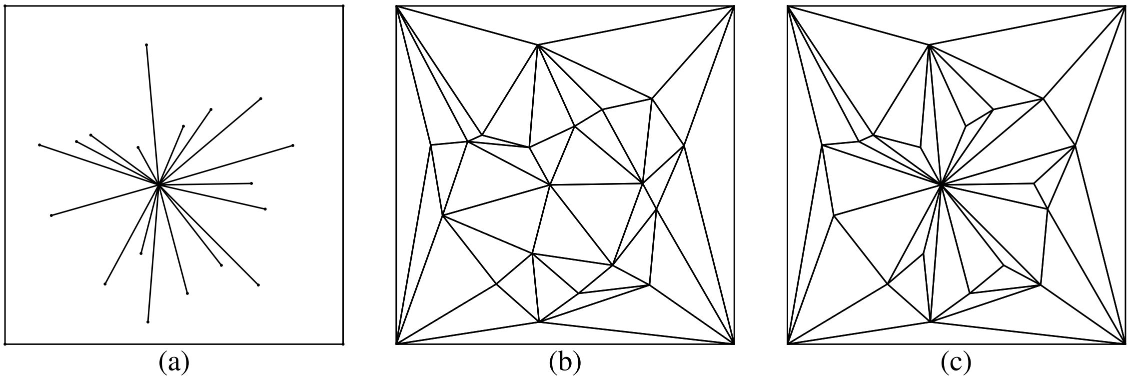


Figure 2.15: (a) A planar straight line graph. (b) Delaunay triangulation of the vertices of the PSLG. (c) Constrained Delaunay triangulation of the PSLG.

gle t. Input segments appear as bold lines. Although there is no empty circle that contains e, the depicted containing circle of e contains no vertices that are visible from the interior of e. There are two vertices inside the circle, but both are hidden behind segments. Hence, e is constrained Delaunay. Similarly, the circumcircle of t contains two vertices, but both are hidden from the interior of t by segments, so t is constrained Delaunay.

Is this notion of visibility ambiguous? For instance, what if a triangle t has a vertex v in its circumcircle, and a segment s only partly obstructs the view, so that v is visible from some points in t but not others? In this case, one of the endpoints of s also lies in the circumcircle of t, so t is unambiguously not constrained Delaunay. (This argument does not extend to three dimensions, unfortunately, which largely explains why no consistent definition of constrained Delaunay tetrahedralization has been put forth.)

Figure 2.15 illustrates a PSLG, a Delaunay triangulation of its vertices, and a constrained Delaunay triangulation of the PSLG. Some of the edges of the CDT are constrained Delaunay but not Delaunay. Take note: constrained Delaunay triangulations are not necessarily Delaunay triangulations.

Like Delaunay triangulations, constrained Delaunay triangulations can be constructed by the flip algorithm. However, the flip algorithm should begin with a triangulation whose edges include all the segments of the input PSLG. To show that such a triangulation always exists (assuming the input vertices are not all collinear), begin with an arbitrary triangulation of the vertices of the PSLG. Examine each input segment in turn to see if it is missing from the triangulation. Each missing segment is forced into the triangulation by deleting all the edges it crosses, inserting the new segment, and retriangulating the two resulting polygons (one on each side of the segment), as illustrated in Figure 2.16. (For a proof that any polygon can be

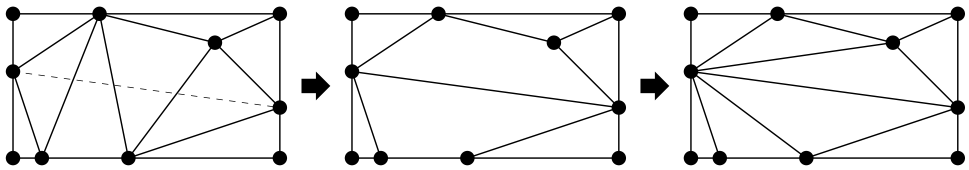


Figure 2.16: Inserting a segment into a triangulation.

triangulated, see Bern and Eppstein [10].)

Once a triangulation containing all the input segments is found, the flip algorithm may be applied, with the provision that segments cannot be flipped. The following results may be proven analogously to the proofs in Section 2.1.1. The only changes that need be made in the proofs is to ignore the presence of vertices that are hidden behind input segments.

Lemma 7 *Let T be a triangulation. If all the triangles of T are constrained Delaunay, then all the edges of T are constrained Delaunay, and vice versa.* ■

Lemma 8 *Let T be a triangulation whose unconstrained edges (those that do not represent input segments) are all locally Delaunay. Then every edge of T is (globally) constrained Delaunay.* ■

Lemma 9 *Given a triangulation of n vertices in which all input segments are represented as edges, the flip algorithm terminates after $\mathcal{O}(n^2)$ edge flips, yielding a triangulation whose edges are all constrained Delaunay.* ■

Theorem 10 *Let X be a PSLG containing three or more vertices that are not all collinear. If no four vertices of X are cocircular, the constrained Delaunay triangulation of X is a triangulation, and is produced by the flip algorithm.* ■

Theorem 11 *Among all constrained triangulations of a PSLG, the constrained Delaunay triangulation maximizes the minimum angle, minimizes the largest circumcircle, and minimizes the largest min-containment circle.* ■

In the case where an input PSLG has no segments, the constrained Delaunay triangulation reduces to the Delaunay triangulation. Hence, by proving these results for the CDT, they are also proven for the Delaunay triangulation. However, I instead presented the simpler proofs for the Delaunay triangulation to aid clarity.

2.1.3 The Delaunay Tetrahedralization

The Delaunay tetrahedralization of a vertex set V is a straightforward generalization of the Delaunay triangulation to three dimensions. An edge, triangular face, or tetrahedron whose vertices are members of V is said to be *Delaunay* if there exists an empty sphere that passes through all its vertices. If no five vertices are cospherical, the Delaunay tetrahedralization is a tetrahedralization and is unique. If cospherical vertices are present, it is customary to define the Delaunay tetrahedralization to be a true tetrahedralization. As with degenerate Delaunay triangulations, a subset of the Delaunay edges, faces, and tetrahedra may have to be

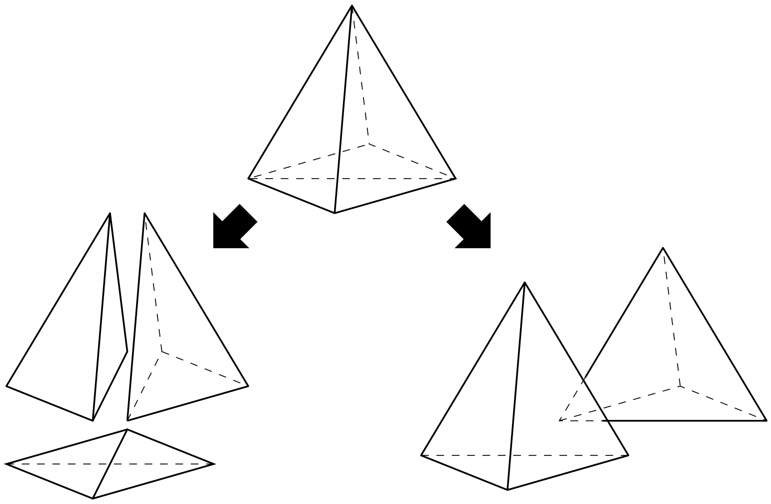


Figure 2.17: This hexahedron can be tetrahedralized in two ways. The Delaunay tetrahedralization (left) includes an arbitrarily thin tetrahedron known as a *sliver*, which could compromise the accuracy of a finite element simulation. The non-Delaunay tetrahedralization on the right consists of two nicely shaped elements.

omitted to achieve this, thus sacrificing uniqueness. The definition of Delaunay triangulation generalizes to dimensions higher than three as well.

I have mentioned that the max-min angle optimality of the two-dimensional Delaunay triangulation, first shown by Lawson [59], does not generalize to higher dimensions. Figure 2.17 illustrates this unfortunate fact with a three-dimensional counterexample. A hexahedron is illustrated at top. Its Delaunay tetrahedralization, which appears at lower left, includes a thin tetrahedron known as a *sliver* or *kite*, which may have dihedral angles arbitrarily close to 0° and 180°. A better quality tetrahedralization of the hexahedron appears at lower right.

Edge flips, discussed in Section 2.1.1, have a three-dimensional analogue, which toggles between these two tetrahedralizations. There are two types of flips in three dimensions, both illustrated in Figure 2.18. A *2-3 flip* transforms the two-tetrahedron configuration into the three-tetrahedron configuration, eliminating the face $\triangle cde$ and inserting the edge ab and three triangular faces connecting ab to c, d, and e. A *3-2 flip* is the reverse transformation, which deletes the edge ab and inserts the face $\triangle cde$.

Recall from Figure 2.6 that a two-dimensional edge flip is not possible if the containing quadrilateral of an edge is not strictly convex. Similarly, a three-dimensional flip is not possible if the containing hexahedron of the edge or face being considered for elimination is not strictly convex. A 2-3 flip is prevented if the line ab does not pass through the interior of the face $\triangle cde$. A 3-2 flip is prevented if $\triangle cde$ does not pass through the interior of the edge ab (Figure 2.18, bottom).

Although the idea of a flip generalizes to three or more dimensions, the flip algorithm in its simplest form does not. Joe [52] gives an example that demonstrates that if the flip algorithm starts from an arbitrary tetrahedralization, it may become stuck in a local optimum, producing a tetrahedralization that is not Delaunay. The tetrahedralization may contain a locally non-Delaunay face that cannot be flipped because its containing hexahedron is not convex, or a locally non-Delaunay edge that cannot be flipped because it is contained in more than three tetrahedra.

It is not known whether an arbitrary tetrahedralization can always be transformed into another arbitrary

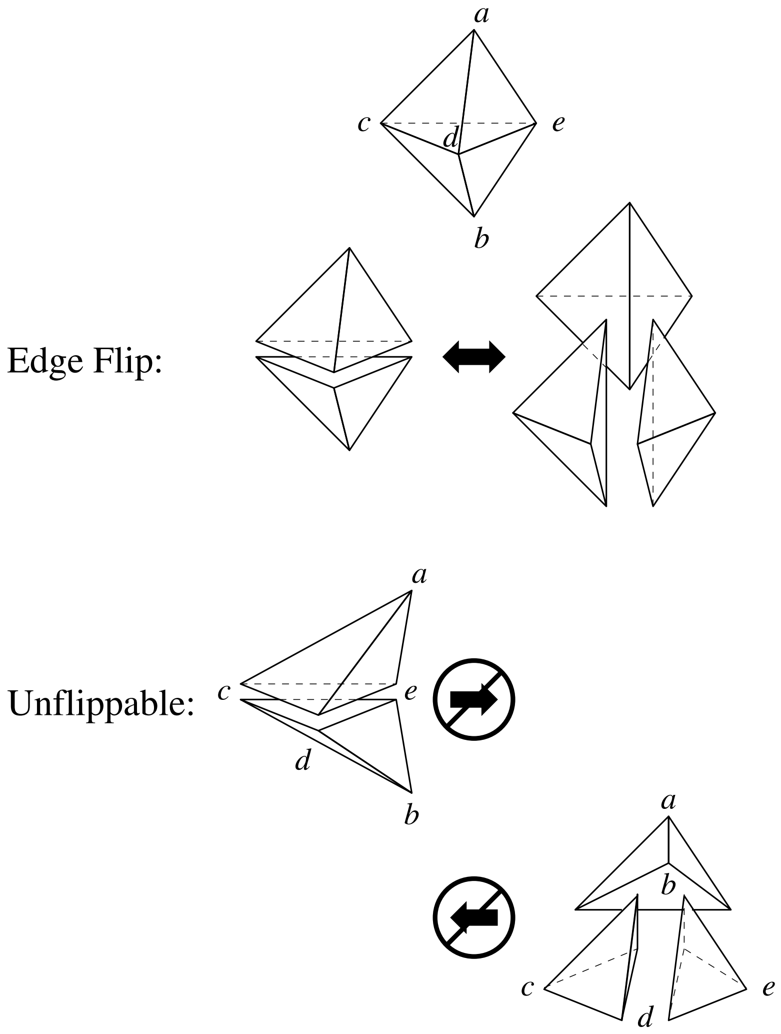


Figure 2.18: Flips in three dimensions. The two-tetrahedron configuration (left) can be transformed into the three-tetrahedron configuration (right) only if the line ab passes through the interior of the triangular face $\triangle cde$. The three-tetrahedron configuration can be transformed into the two-tetrahedron configuration only if the plane containing $\triangle cde$ passes through the interior of the edge ab.

tetrahedralization of the same vertex set through a sequence of flips. Nevertheless, Delaunay tetrahedralizations can be constructed by an incremental insertion algorithm based on flips, discussed in Section 2.1.4.

Any algorithm based on flips in dimensions greater than two must give some consideration to the possibility of coplanar vertices. For instance, a three-dimensional flip-based incremental Delaunay tetrahedralization algorithm must be able to explicitly or implicitly perform the *4-4 flip* demonstrated in Figure 2.19. This transformation is handy when the vertices c, d, e, and f are coplanar. This flip is directly analogous to the two-dimensional edge flip, wherein the edge df is replaced by the edge ce. 4-4 flips are used often in cases where c, d, e, and f lie on an interior boundary facet of an object being meshed. One should be aware of the special case where c, d, e, and f lie on an exterior boundary, and the top two tetrahedra, as well as the vertex a, are missing. One might refer to this case as a 2-2 flip.

A programmer does not need to implement the 4-4 flip directly, because its effect can be duplicated by performing a 2-3 flip (for instance, on tetrahedra $acdf$ and $adef$) followed by a 3-2 flip. However, this

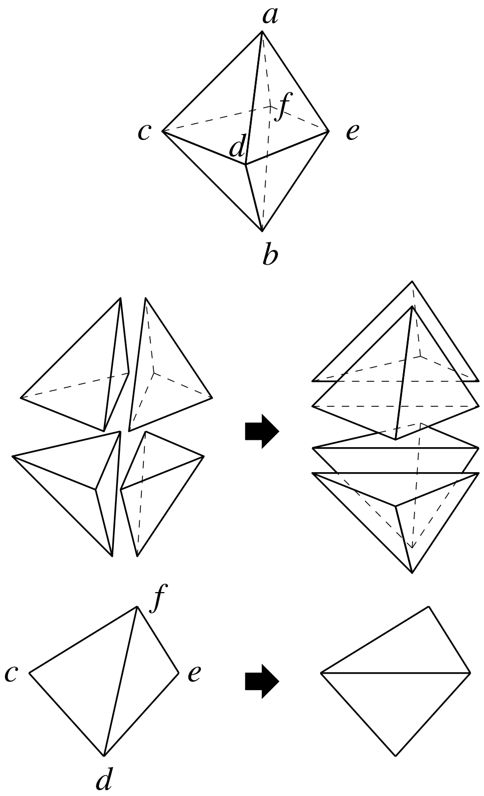


Figure 2.19: A 4-4 flip. The vertices c, d, e, and f are coplanar. This transformation is analogous to the two-dimensional edge flip (bottom).

sequence transiently creates a sliver tetrahedron $cdef$ (created by the first flip and eliminated by the second) with zero volume, which may be considered undesirable. It is up to the individual programmer to decide how best to address this issue.

Although Delaunay tetrahedralizations are invaluable for three-dimensional mesh generation, they are in many ways more limited than their two-dimensional brethren. The first difficulty is that, whereas every polygon can be triangulated (without creating additional vertices), there are polyhedra that cannot be tetrahedralized. Schönhardt furnishes an example depicted in Figure 2.20 (right). The easiest way to envision this polyhedron is to begin with a triangular prism. Imagine grasping the prism so that one of its two triangular faces cannot move, while the opposite triangular face is rotated slightly about its center without moving out of its plane. As a result, each of the three square faces is broken along a diagonal *reflex edge* (an edge at which the polyhedron is locally concave) into two triangular faces. After this transformation, the upper left corner and lower right corner of each (former) square face are separated by a reflex edge and are no longer visible to each other through the interior of the polyhedron. Hence, no vertex of the top face can see all three vertices of the bottom face. It is not possible to choose four vertices of the polyhedron that do not include two separated by a reflex edge; thus, any tetrahedron whose vertices are vertices of the polyhedron will not lie entirely within the polyhedron. Schönhardt's polyhedron cannot be tetrahedralized without inserting new vertices.

Nevertheless, any convex polyhedron can be tetrahedralized. However, it is not always possible to tetrahedralize a convex polyhedron in a manner that conforms to interior boundaries, because those interior boundaries could be the facets of Schönhardt's polyhedron. Hence, constrained tetrahedralizations do not always exist. What if we forbid constrained facets, but permit constrained segments? Figure 2.21 illustrates

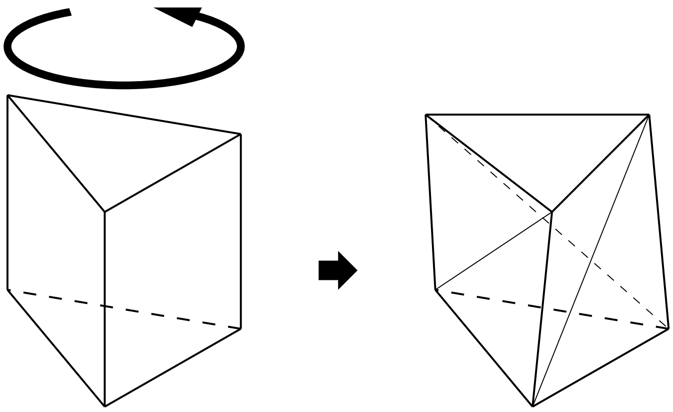


Figure 2.20: Schönhardt's untetrahedralizable polyhedron (right) is formed by rotating one end of a triangular prism (left), thereby creating three diagonal reflex edges.

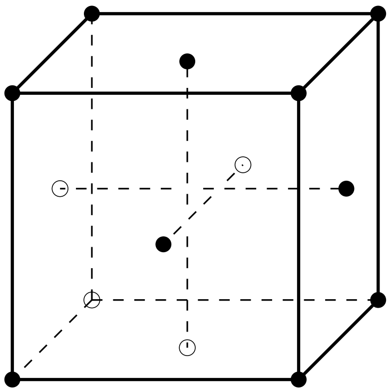


Figure 2.21: A set of vertices and segments for which there is no constrained tetrahedralization.

a set of vertices and segments for which a constrained tetrahedralization does not exist. (The convex hull, a cube, is illustrated for clarity, but no constrained facets are present in the input.) Three orthogonal segments pass by each other near the center of the cube, but do not intersect. If any one of these segments is omitted, a tetrahedralization is possible. Hence, unlike the two-dimensional case, it is not always possible to insert a new segment into a tetrahedralization.

Even in cases where a constrained tetrahedralization does exist, nobody has yet put forth a convincing definition of *constrained Delaunay tetrahedralization*. It seems unlikely that there exists a definition that has the desired qualities of uniqueness, symmetry, and rotational invariance (in nondegenerate cases). This difficulty arises because, whereas a segment cleanly partitions a circumcircle in two dimensions, except when an endpoint of the segment lies in the circle, segments and facets do not necessarily partition circumspheres in three dimensions.

Another nail in the coffin of constrained tetrahedralizations comes from Ruppert and Seidel [83], who show that the problem of determining whether or not a polyhedron can be tetrahedralized without additional vertices is NP-complete. Hence, the prospects for developing constrained tetrahedralization algorithms that consistently recover boundaries are pessimistic.

The mesh generation algorithm discussed in Chapter 4 recovers boundaries by strategically inserting additional vertices. Unfortunately, Ruppert and Seidel also show that the problem of determining whether a

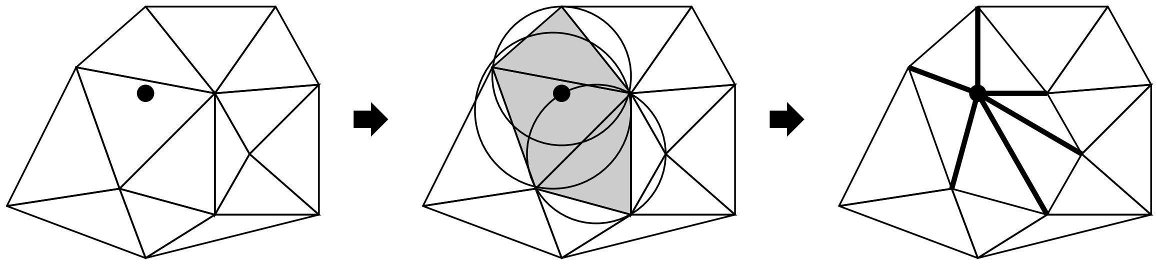


Figure 2.22: The Bowyer/Watson algorithm in two dimensions. When a new vertex is inserted into a triangulation (left), all triangles whose circumcircles contain the new vertex are deleted (center; deleted triangles are shaded). Edges are created connecting the new vertex to the vertices of the insertion polyhedron (right).

polyhedron can be tetrahedralized with only k additional vertices is NP-complete. On the bright side, Bern and Eppstein [10] show that any polyhedron can be tetrahedralized with the insertion of $\mathcal{O}(n^2)$ additional vertices, so the demands of tetrahedralization are not limitless.

2.1.4 Algorithms for Constructing Delaunay Triangulations

Three types of algorithms are in common use for constructing Delaunay triangulations. The simplest are incremental insertion algorithms, which have the advantage of generalizing to arbitrary dimensionality, and will be discussed in some depth here. In two dimensions, there are faster algorithms based upon divide-and-conquer and sweepline techniques, which will be discussed here only briefly. Refer to Su and Drysdale [91, 90] for an informative overview of these and other two-dimensional Delaunay triangulation algorithms. The discussion below is centered on abstract features of the algorithms; see Section 5.1 for further details on implementation.

Incremental insertion algorithms operate by maintaining a Delaunay triangulation, into which vertices are inserted one at a time. The earliest such algorithm, introduced by Lawson [59], is based upon edge flips. An incremental algorithm that does not use edge flips, and has the advantage of generalizing to arbitrary dimensionality, was introduced simultaneously by Bowyer [12] and Watson [93]. These two articles appear side-by-side in a single issue of the Computer Journal[1]. I will examine the Bowyer/Watson algorithm first, and then return to the algorithm of Lawson.

In the Bowyer/Watson algorithm, when a new vertex is inserted, each triangle whose circumcircle contains the new vertex is no longer Delaunay, and is thus deleted. All other triangles remain Delaunay, and are left undisturbed. The set of deleted triangles collectively form an *insertion polyhedron*, which is left vacant by the deletion of these triangles, as illustrated in Figure 2.22. The Bowyer/Watson algorithm connects each vertex of the insertion polyhedron to the new vertex with a new edge. These new edges are Delaunay due to the following simple lemma.

Lemma 12 *Let v be a newly inserted vertex, and let w be a vertex of a triangle t that is deleted because its circumcircle contains v. Then vw is Delaunay.*

Proof: See Figure 2.23. The circumcircle of t contains no vertex but v. Let C be the circle that passes through v and w, and is tangent to the circumcircle of t at w. C is empty, so vw is Delaunay. ∎

[1]The two algorithms are similar in all essential details, but Bowyer reports a better asymptotic running time than Watson, which on close inspection turns out to be nothing more than an artifact of his more optimistic assumptions about the speed of point location.

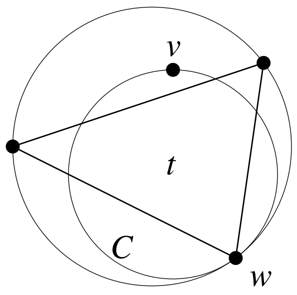


Figure 2.23: If v is a newly inserted vertex, and w is a vertex of a triangle t whose circumcircle contains only v, then vw is Delaunay.

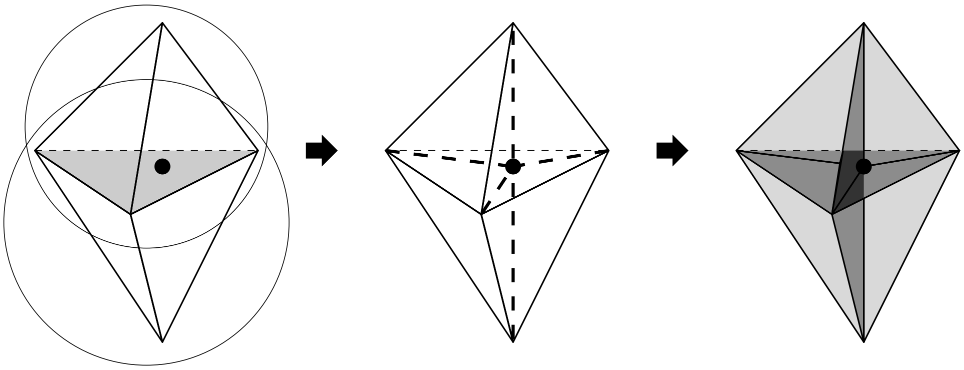


Figure 2.24: The Bowyer/Watson algorithm in three dimensions. At left, a new vertex falls inside the circumspheres of the two tetrahedra illustrated. (These tetrahedra may be surrounded by other tetrahedra, which for clarity are not shown.) These tetrahedra are deleted, along with the face (shaded) between them. At center, the five new Delaunay edges (bold dashed lines). At right, the nine new Delaunay faces (one for each edge of the insertion polyhedron) are drawn translucent. Six new tetrahedra are formed.

All new edges created by the insertion of a vertex v have v as an endpoint. This must be true of any correct incremental insertion algorithm, because if an edge (not having v as an endpoint) is not Delaunay before v is inserted, it will not be Delaunay after v is inserted.

The Bowyer/Watson algorithm extends in a straightforward way to three (or more) dimensions. When a new vertex is inserted, every tetrahedron whose circumsphere contains the new vertex is deleted, as illustrated in Figure 2.24. The new vertex then floats inside a hollow *insertion polyhedron*, which is the union of the deleted tetrahedra. Each vertex of the insertion polyhedron is connected to the new vertex with a new edge. Each edge of the insertion polyhedron is connected to the new vertex with a new triangular face.

In its simplest form, the Bowyer/Watson algorithm is not robust against floating-point roundoff error. Figure 2.25 illustrates a degenerate example in which two triangles have the same circumcircle, but due to roundoff error only one of them is deleted, and the triangle that remains stands between the new vertex and the other triangle. The insertion polyhedron is not simple, and the triangulation that results after the new triangles are added is nonsensical.

In two dimensions, this problem may be avoided by returning to Lawson's algorithm [59], which is based upon edge flips. Lawson's algorithm is illustrated in Figure 2.26.

When a vertex is inserted, the triangle that contains it is found, and three new edges are inserted to attach the new vertex to the vertices of the containing triangle. (If the new vertex falls upon an edge of the triangulation, that edge is deleted, and four new edges are inserted to attach the new vertex to the vertices

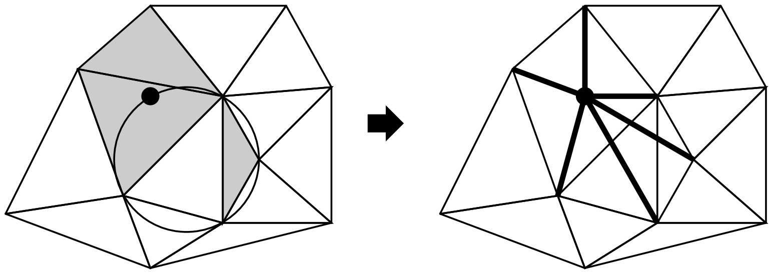


Figure 2.25: The Bowyer/Watson algorithm may behave nonsensically under the influence of floating-point roundoff error.

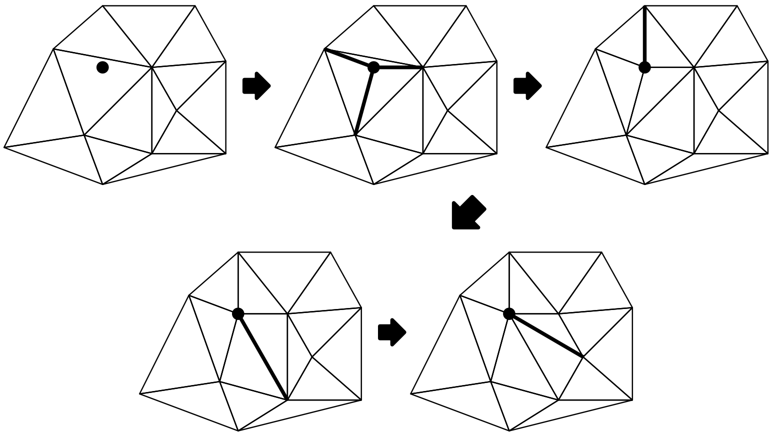


Figure 2.26: Lawson's incremental insertion algorithm uses edge flipping to achieve the same result as the Bowyer/Watson algorithm.

of the containing quadrilateral.) Next, a recursive procedure tests whether the new vertex lies within the circumcircles of any neighboring triangles; each affirmative test triggers an edge flip that removes a locally non-Delaunay edge. Each edge flip reveals two additional edges that must be tested. When there are no longer any locally non-Delaunay edges opposite the new vertex, the triangulation is globally Delaunay.

Disregarding roundoff error, Lawson's algorithm achieves exactly the same result as the Bowyer/Watson algorithm. In the presence of roundoff error, Lawson's algorithm avoids the catastrophic circumstance illustrated in Figure 2.25. Lawson's algorithm is not absolutely robust against roundoff error, but failures are rare compared to the most naïve form of the Bowyer/Watson algorithm. However, the Bowyer/Watson algorithm can be implemented to behave equally robustly; for instance, the insertion polygon may be found by depth-first search from the initial triangle.

A better reason for noting Lawson's algorithm is that it is slightly easier to implement, in part because the topological structure maintained by the algorithm remains a triangulation at all times. Guibas and Stolfi [47] provide a particularly elegant implementation.

Joe [53, 54] and Rajan [78] have generalized Lawson's flip-based algorithm to arbitrary dimensionality. Of course, these algorithms have the same effect as the Bowyer/Watson algorithm, but may present the same advantages for implementation that Lawson's algorithm offers in two dimensions.

I do not review the mathematics underpinning three-dimensional incremental insertion based on flips, but I shall try to convey some of the intuition behind it. Returning first to the two-dimensional algorithm, imagine yourself as an observer standing at the newly inserted vertex. From your vantage point, suppose

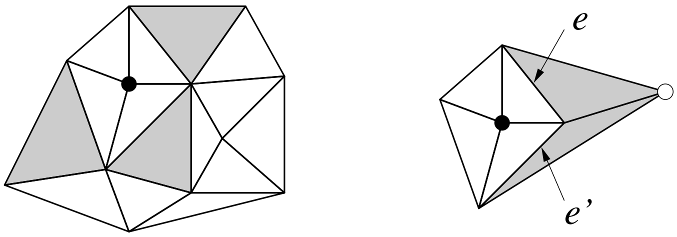


Figure 2.27: Left: The shaded triangles are considered to be visible from the new vertex, and are considered for removal (by edge flip). Right: Triangles under consideration for removal fall into two categories. The upper right triangle has an apex (open circle) visible through its base edge e from the new vertex. Only one of this triangle's sides faces the new vertex. The lower right triangle has an apex (the same open circle) that is not visible through its base edge e', and thus the base edge cannot be flipped. Two of this triangle's sides face the new vertex.

that any triangle (not adjoining the new vertex) is visible to you if it might be eligible for removal by the next edge flip. These triangles are shaded in Figure 2.27.

For each such triangle, there are two cases. The apex of the triangle (the vertex hidden from your view) may or may not fall within the sector of your vision subtended by the base edge of the triangle. If the apex falls within this sector, then only the base edge of the triangle faces toward you; the other two sides face away (see the upper right triangle of Figure 2.27). If the apex falls outside this sector, then two sides of the triangle face toward you (see the lower right triangle of Figure 2.27). In the latter case, the base edge cannot be flipped, because its containing quadrilateral is not strictly convex.

Returning to the three-dimensional case, imagine yourself as a vertex that has just been inserted inside a tetrahedron, splitting it into four tetrahedra. As you look around, you see the four faces of the original tetrahedron, and the neighbor tetrahedra behind these faces (which are analogous to the shaded triangles in Figure 2.27).

For each neighbor tetrahedron, there are three possibilities. The tetrahedron might have one face directed toward you and three away (Figure 2.28, left), in which case a 2-3 flip is possible. If performed, this flip deletes the visible face, revealing the three back faces, and creates a new edge extending from the new vertex (your viewpoint) to the newly revealed vertex in the back. The flip also creates three new faces, extending from the new vertex to the three newly revealed edges.

If the tetrahedron has two faces directed toward you (Figure 2.28, center), and neither face is obscured by an interposing tetrahedron, a 3-2 flip is possible. If performed, this flip deletes both visible faces, revealing the two back faces. A new face is created, extending from the new vertex to the newly revealed edge.

If the tetrahedron has three faces directed toward you (Figure 2.28, right), no flip is possible.

I have omitted the degenerate case in which you find yourself precisely coplanar with one face of the tetrahedron, with one other face directed toward you and two directed away. This circumstance would appear similar to the upper left image of Figure 2.28, but with d directly behind the edge ab. If the new vertex falls within the circumcircle of the face $\triangle abd$, then $abcd$ is no longer Delaunay, and the aforementioned 4-4 flip may be used, thus eliminating both tetrahedra adjoining $\triangle abd$.

Each flip uncovers two to four new faces, possibly leading to additional flips.

This discussion of incremental insertion algorithms in two and three dimensions has assumed that all new vertices fall within the existing triangulation. What if a vertex falls outside the convex hull of the previous vertices? One solution is to handle this circumstance as a special case. New triangles or tetrahedra

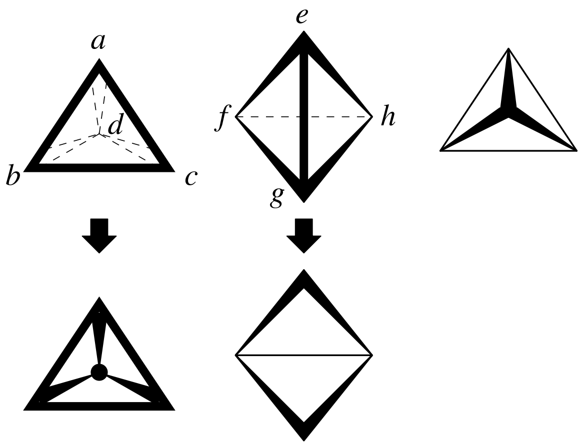


Figure 2.28: Three orientations of a tetrahedron as viewed from a newly inserted vertex p. Left: One face of tetrahedron $abcd$ is directed toward p. If $abcd$ is no longer Delaunay, a 2-3 flip deletes the face $\triangle abc$, replacing the tetrahedra $abcd$ and $abcp$ with $abdp$, $bcdp$, and $cadp$. Center: Two faces of tetrahedron $efgh$ are directed toward p. If neither face is obscured by another tetrahedron, and $efgh$ is no longer Delaunay, a 3-2 flip deletes the edge eg and faces $\triangle egf$, $\triangle egh$, and $\triangle egp$, replacing the tetrahedra $efgh$, $efgp$, and $ghep$ with $fghp$ and $hefp$. Right: Three faces of a tetrahedron are directed toward p. No flip is possible.

are created to connect the new vertex to all the edges or faces of the convex hull visible from that vertex. Then, flipping may proceed as usual. An alternative solution that simplifies programming is to bootstrap incremental insertion with a very large triangular or tetrahedral bounding box that contains all the input vertices. After all vertices have been inserted, the bounding box is removed as a postprocessing step. The problem with this approach is that one must be careful to choose the vertices of the bounding box so that they do not cause triangles or tetrahedra to be missing from the final Delaunay triangulation.

Assuming that one has found the triangle or tetrahedron in which a new vertex is to be inserted, the amount of work required to insert the vertex is proportional to the number of flips, which is typically small. Pathological cases can occur in which a single vertex insertion causes $\mathcal{O}(n)$ flips in two dimensions, or $\mathcal{O}(n^2)$ in three; but such cases arise rarely in mesh generation, and it is common to observe that the average number of flips per insertion is a small constant.

In two dimensions, this observation is given some support by a simple theoretical result. Suppose one wishes to construct, using Lawson's algorithm, the Delaunay triangulation of a set of vertices that is entirely known at the outset. If the input vertices are inserted in a random order, chosen uniformly from all possible permutations, then the expected number of edge flips per vertex insertion is bounded below three.

This elegant result seems to originate with Chew [20], albeit in the slightly simpler context of Delaunay triangulations of convex polygons. This result was proven more generally by Guibas, Knuth, and Sharir [46], albeit with a much more complicated proof than Chew's. The result is based on the observation that when a vertex is inserted, each edge flip increases by one the degree of the new vertex. Hence, if the insertion of a vertex causes four edge flips, there will be seven edges incident to that vertex. (The first three edges connect the new vertex to the vertices of the triangle in which it falls, and the latter four are created through edge flips.)

Here, the technique of *backward analysis* is applied. The main principle of backward analysis is that after an algorithm terminates, one imagines reversing time and examining the algorithm's behavior as it runs backward to its starting state. In the case of Lawson's algorithm, one begins with a complete Delaunay triangulation of all the input vertices, and removes one vertex at a time.

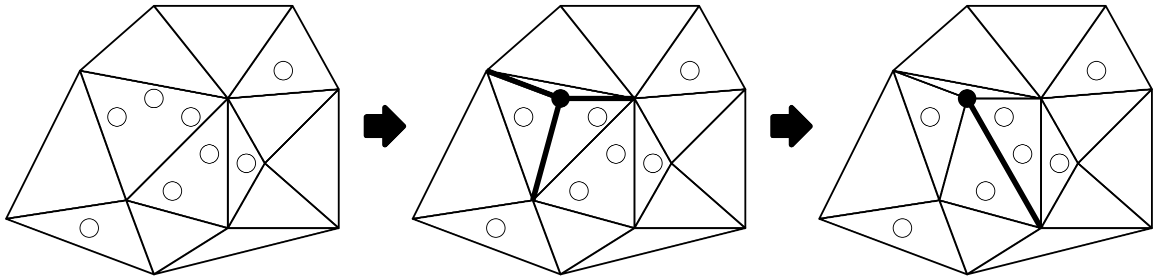


Figure 2.29: The algorithm of Guibas, Knuth, and Sharir maintains a mapping between uninserted vertices (open circles) and triangles. The bounding box vertices and the edges incident to them are not shown.

The power of backward analysis stems from the fact that a uniformly chosen random permutation read backward is still a uniformly chosen random permutation. Hence, one may imagine that triangulation vertices are being randomly selected, one at a time from a uniform distribution, for removal from the triangulation. With time running backward, the amount of time spent removing a vertex from the triangulation is proportional to the degree of the vertex. Because the average degree of vertices in a planar graph is bounded below six, the expected number of edge flips observed when a randomly chosen vertex is removed is bounded below three.

Hence, when Lawson's algorithm is running forward in time, the expected number of edge flips required to insert a vertex is at most three. Unfortunately, this result is not strictly applicable to most Delaunay-based mesh generation algorithms, because the entire set of vertices is not known in advance, and thus the vertex insertion order cannot be randomized. Nevertheless, the result gives useful intuition for why constant-time vertex insertion is so commonly observed in mesh generation.

Unfortunately, when finding the Delaunay triangulation of an arbitrary set of vertices, edge flips are not the only cost. In many circumstances, the dominant cost is the time required for *point location*: finding the triangle or tetrahedron in which a vertex lies, so that the vertex may be inserted. Fortunately, most Delaunay-based mesh generation algorithms insert most of their vertices in places that have already been identified as needing refinement, and thus the location of each new vertex is already known. However, in a general-purpose Delaunay triangulator, point location is expensive.

In two dimensions, point location can be performed in expected amortized $\mathcal{O}(\log n)$ time per vertex, where n is the number of vertices in the mesh. Clarkson and Shor [24] were the first to achieve this bound, again by inserting the vertices in random order. Clarkson and Shor perform point location by maintaining a *conflict graph*, which is a bipartite graph that associates edges of the triangulation with vertices that have not yet been inserted. Specifically, the conflict graph associates each uninserted vertex with the edges of that vertex's insertion polygon (including triangulation edges in the interior of the insertion polygon). The conflict graph is updated with each vertex insertion.

Rather than explain the Clarkson and Shor algorithm in detail, I present a simpler variant due to Guibas, Knuth, and Sharir [46]. For simplicity, assume that a large bounding box is used to contain the input vertices. One version of the algorithm of Guibas et al. maintains a simpler conflict graph in which each uninserted vertex is associated with the triangle that contains it (Figure 2.29, left). If a vertex lies on an edge, either containing triangle is chosen arbitrarily.

When a triangle is divided into three triangles (Figure 2.29, center) or an edge is flipped (Figure 2.29, right), the vertices in the deleted triangle(s) are redistributed among the new triangles as dictated by their positions. When a vertex is chosen for insertion, its containing triangle is identified by using the conflict graph. The dominant cost of the algorithm is the cost of redistributing uninserted vertices to their new containing triangles each time a vertex is inserted.

Although Clarkson and Shor [24] and Guibas et al. [46] both provide ways to analyze this algorithm, the simplest analysis originates with Kenneth Clarkson and is published in a report by Seidel [85]. Here I give a rough sketch of the proof, which relies on backward analysis. Suppose the Delaunay triangulation of n vertices is being constructed. Consider the step wherein a $(p-1)$-vertex triangulation is converted into a p-vertex triangulation by inserting a randomly chosen vertex; but consider running the step in reverse. In the backward step, a random vertex v of the p-vertex triangulation is chosen for deletion. What is the expected number of vertices that are redistributed? Each triangle of the triangulation has three vertices, so the probability that any given triangle is deleted when v is deleted is $\frac{3}{p}$. (The probability is actually slightly smaller, because some triangles have vertices of the bounding box, but $\frac{3}{p}$ is an upper bound.) If a triangle is deleted, all vertices assigned to that triangle are redistributed. Each of the $n-p$ uninserted vertices is assigned to exactly one triangle; so by linearity of expectation, the expected number of vertices redistributed when v is deleted (or, if time is running forward, inserted) is $\frac{3(n-p)}{p}$. Hence, the running time of the algorithm is $\sum_{p=1}^{n} \frac{3(n-p)}{p} \in \mathcal{O}(n \log n)$.

The same analysis technique can be used, albeit with complications, to show that incremental Delaunay triangulation in higher dimensions can run in randomized $\mathcal{O}(n^{\lceil d/2 \rceil})$ time. Consult Seidel [85] for details.

The first $\mathcal{O}(n \log n)$ algorithm for two-dimensional Delaunay triangulation was not an incremental algorithm, but a divide-and-conquer algorithm. Shamos and Hoey [86] developed an algorithm for the construction of a Voronoi diagram, which may be easily dualized to form a Delaunay triangulation. In programming practice, this is an unnecessarily difficult procedure, because forming a Delaunay triangulation directly is much easier, and is in fact the easiest way to construct a Voronoi diagram. Lee and Schachter [60] were the first to publish a divide-and-conquer algorithm that follows this easier path. The algorithm is nonetheless intricate, and Guibas and Stolfi [47] provide an important aid to programmers by filling out many tricky implementation details. Dwyer [30] offers an interesting modification to divide-and-conquer Delaunay triangulation that achieves better asymptotic performance on some vertex sets, and offers improved speed in practice as well. There is also an $\mathcal{O}(n \log n)$ algorithm for constrained Delaunay triangulations due to Chew [18]. Divide-and-conquer Delaunay triangulation is discussed further in Section 5.1.

Another well-known $\mathcal{O}(n \log n)$ two-dimensional Delaunay triangulation algorithm is Fortune's sweep-line algorithm [31].

2.2 Research in Mesh Generation

The discussion in this chapter has heretofore been concerned with triangulations of complete vertex sets. Of course, a mesh generator rarely knows all the vertices of the final mesh prior to triangulation, and the real problem of meshing is deciding where to place vertices to ensure that the mesh has elements of good quality and proper sizes.

I attempt here only the briefest of surveys of mesh generation algorithms. Detailed surveys of the mesh generation literature have been supplied by Thompson and Weatherill [92] and Bern and Eppstein [10]. I focus my attention on algorithms that make use of Delaunay triangulations, and on algorithms that achieve provable bounds. I postpone three algorithms, due to L. Paul Chew and Jim Ruppert, that share both these characteristics. They are described in detail in Chapter 3.

Only simplicial mesh generation algorithms are discussed here; algorithms for generating quadrilateral, hexahedral, or other non-simplicial meshes are omitted. The most popular approaches to triangular and tetrahedral mesh generation can be divided into three classes: Delaunay triangulation methods, advancing front methods, and methods based on grids, quadtrees, or octrees.

2.2.1 Delaunay Mesh Generation

It is difficult to trace who first used Delaunay triangulations for finite element meshing, and equally difficult to tell where the suggestion arose to use the triangulation to guide vertex creation. These ideas have been intensively studied in the engineering community since the mid-1980s, and began to attract interest from the computational geometry community in the early 1990s.

I will name only a few scattered references from the voluminous literature. Many of the earliest papers suggest performing vertex placement as a separate step, typically using structured grid techniques, prior to Delaunay triangulation. For instance, Cavendish, Field, and Frey [17] generate grids of vertices from cross-sections of a three-dimensional object, then form their Delaunay tetrahedralization. The idea of using the triangulation itself as a guide for vertex placement followed quickly; for instance, Frey [41] removes poor quality elements from a triangulation by inserting new vertices at their *circumcenters*—the centers of their circumcircles—while maintaining the Delaunay property of the triangulation. This idea went on to bear vital theoretical fruit, as Chapters 3 and 4 will demonstrate.

I have mentioned that the Delaunay triangulation of a vertex set may be unsatisfactory for two reasons: elements of poor quality may appear, and input boundaries may fail to appear. Both these problems have been treated in the literature. The former problem is typically treated by inserting new vertices at the circumcenters [41] or centroids [94] of poor quality elements. It is sometimes also treated with an advancing front approach, discussed briefly in Section 2.2.2.

The problem of the recovery of missing boundaries may be treated in several ways. These approaches have in common that boundaries may have to be broken up into smaller pieces. For instance, each input segment is divided into a sequence of triangulation edges which I call *subsegments*, with a vertex inserted at each division point. In three dimensions, each facet of an object to be meshed is divided into triangular faces which I call *subfacets*. Vertices of the tetrahedralization lie at the corners of these subfacets.

In the earliest publications, boundary integrity was assured simply by spacing vertices sufficiently closely together on the boundary prior to forming a triangulation [41]—surely an error-prone approach. A better way to ensure the presence of input segments is to first form the triangulation, and then check whether any input segments are missing.

Missing segments can be recovered by one of several methods, which work in two or three dimensions. One method inserts a new vertex (while maintaining the Delaunay property of the mesh) at the midpoint of any missing segment, splitting it into two subsegments [94]. Sometimes, the two subsegments appear as edges of the resulting triangulation. If not, the subsegments are recursively split in turn. This method, sometimes called *stitching*, is described in more detail in Section 3.3.1. Although it is not obvious how this method might generalize to three-dimensional facet recovery, I will demonstrate in Section 4.2.1 that this generalization is possible and has some advantages over the next method I describe.

Another method, usually only used in three dimensions, can be applied to recover both missing segments and missing facets. This method inserts a new vertex wherever a face or edge of the triangulation intersects a missing segment or facet [95, 48, 96, 79]. The method is often coupled with flips [43, 95], which are used to reduce the number of vertices that must be inserted. The pessimistic results on constrained tetrahedralizations in Section 2.1.3 imply that, in three dimensions, flips cannot always achieve boundary recovery on their own; in some cases, new vertices must inevitably be inserted to fully recover a boundary.

Boundary recovery methods will be discussed further in Sections 3.3.1, 4.2.1, and 5.3.1.

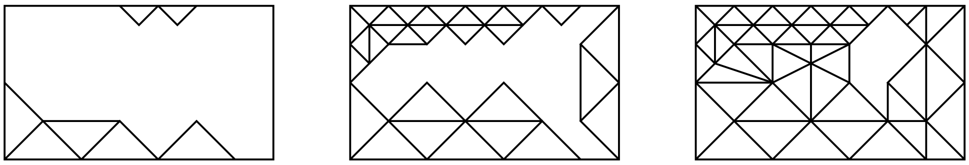


Figure 2.30: Several stages in the progression of an advancing front algorithm.

2.2.2 Advancing Front Methods

Advancing front methods [62, 6, 51, 64] begin by dividing the boundaries of the mesh into edges (in two dimensions) or triangular faces (in three). These discretized boundaries form the initial *front*. Triangles or tetrahedra are generated one-by-one, starting from the boundary edges or faces, and work toward the center of the region being meshed, as illustrated in Figure 2.30. The inner surface of these elements collectively form an *advancing front*.

Advancing front methods require a good deal of second-guessing, first to ensure that the initial division of the boundaries is prudent, and second to ensure that when the advancing walls of elements collide at the center of the mesh, they are merged together in a manner that does not compromise the quality of the elements. In both cases, a poor choice of element sizes may result in disaster, as when a front of small elements collides with a front of large elements, making it impossible to fill the space between with nicely shaped elements. These problems are sufficiently difficult that there are, to my knowledge, no provably good advancing front algorithms. Advancing front methods typically create astonishingly good triangles or tetrahedra near the boundaries of the mesh, but are much less effective where fronts collide.

In three dimensions, generating the surface mesh may be a difficult problem itself. Ironically, the mesh generator described by Marcum and Weatherill [63] uses a Delaunay-based mesh generator to create a complete tetrahedralization, then throws away the tetrahedralization except for the surface mesh, which is used to seed their advancing front algorithm.

Mavriplis [64] combines the Delaunay triangulation and advancing front methods. The combination makes a good deal of sense, because a Delaunay triangulation in the interior of the mesh is a useful search structure for determining how close different fronts are to each other. (Some researchers use background grids for this task.) Conversely, the advancing front method may be used as a vertex placement method for Delaunay meshing. A sensible strategy might be to abandon the advancing front shortly before fronts collide, and use a different vertex placement strategy (such as inserting vertices at circumcenters or centroids of poor quality elements) in the center of the mesh, where such strategies tend to be most effective.

Figure 2.31 depicts one of the world's most famous meshes, generated by an advancing front method of Barth and Jesperson [9]. The mesh is the Delaunay triangulation of vertices placed by a procedure moving outward from this airfoil. Of course, the problems associated with colliding fronts are reduced in circumstances like this, where one is meshing the exterior, rather than the interior, of an object.

2.2.3 Grid, Quadtree, and Octree Methods

The last decade has seen the emergence of mesh generation algorithms with provably good bounds.

Baker, Grosse, and Rafferty [5] gave the first algorithm to triangulate PSLGs with guaranteed upper and lower bounds on element angle. By placing a fine uniform grid over a PSLG, warping the edges of the grid to fit the input segments and vertices, and triangulating the warped grid, they are able to construct a

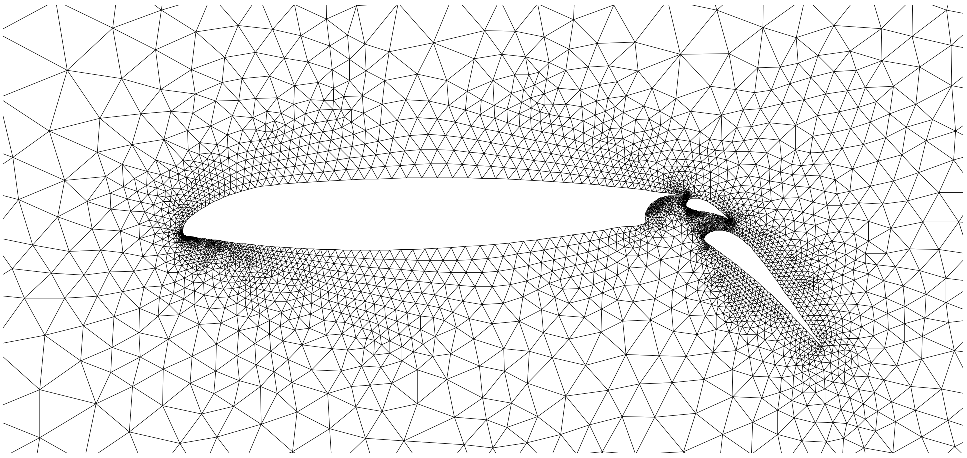
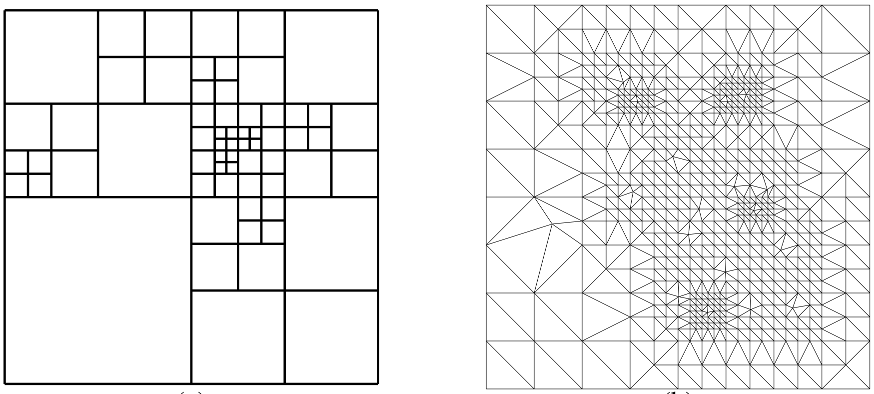


Figure 2.31: Mesh produced by an advancing front, moving outward from an airfoil.



(a) (b)

Figure 2.32: (a) A quadtree. (b) A quadtree-based triangulation of a vertex set, with no angle smaller than $20°$ (courtesy Marshall Bern).

triangular mesh whose angles are bounded between $13°$ and $90°$ (except where the input PSLG has angles smaller than $13°$; these cannot be improved). The elements of the mesh are of uniform size.

To produce graded meshes, some researchers have turned to *quadtrees*. A quadtree is a recursive data structure used to efficiently manipulate multiscale geometric objects in the plane. Quadtrees recursively partition a region into axis-aligned squares. A top-level square called the *root* encloses the entire input PSLG. Each quadtree square can be divided into four *child* squares, which can be divided in turn, as illustrated in Figure 2.32(a). *Octrees* are the generalization of quadtrees to three dimensions; each cube in an octree can be subdivided into eight cubes. See Samet [84] for a survey of quadtree data structures.

Meshing algorithms based on quadtrees and octrees have been used extensively in the engineering community for over a decade [98, 99, 87]. Their first role in mesh generation with provable bounds appears in a paper by Bern, Eppstein, and Gilbert [11]. The Bern et al. algorithm triangulates a polygon with guaranteed bounds on both element quality and the number of elements produced. All angles (except small input angles) are greater than roughly $18.4°$, and the mesh is size-optimal (as defined in Section 1.4). The angle bound applies to triangulations of polygons with polygonal holes, but cannot be extended to general PSLGs, as Section 3.6 will show. Figure 2.32(b) depicts a mesh generated by one variant of the Bern et al. algorithm. For this illustration, a set of input vertices was specified (with no constraining segments), and a mesh was

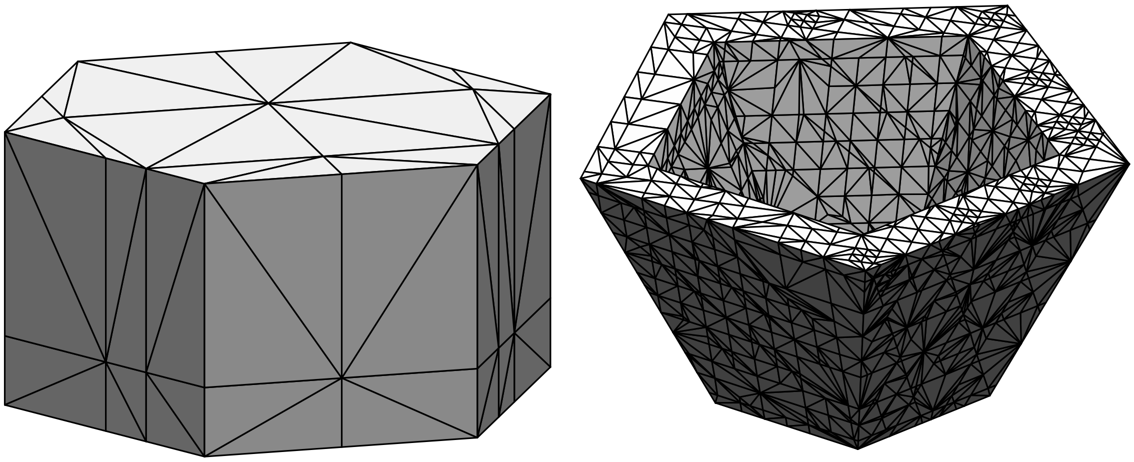


Figure 2.33: Two meshes generated by Stephen Vavasis' QMG package, an octree-based mesh generator with provable bounds. (Meshes courtesy Stephen Vavasis.)

generated (adding a great many additional vertices) that accommodates the input vertices and has no angle smaller than 20°. Figure 3.7 (top) on Page 47 depicts a mesh of a polygon with holes.

The algorithm of Bern et al. works by constructing a quadtree that is dense enough to isolate each input feature (vertex or segment) from other features. Next, the quadtree is warped to coincide with input vertices and segments. (Warping changes the shape of the quadtree, but not its topology.) Finally, the squares are triangulated.

Neugebauer and Diekmann [73] have improved the results of Bern et al., replacing the square quadtree with a rhomboid quadtree so that the triangles of the final mesh tend to be nearly equilateral. Assuming there are no small input angles, polygonal domains with polygonal holes and isolated interior points can be triangulated with all angles between 30° and 90°.

Remarkably, provably good quadtree meshing has been extended to polyhedra of arbitrary dimensionality. Mitchell and Vavasis [69, 70] have developed an algorithm based on octrees (and their higher-dimensional brethren) that triangulates polyhedra, producing size-optimal meshes with guaranteed bounds on element aspect ratios. The generalization to more than two dimensions is quite intricate, and the theoretical bounds on element quality are not strong enough to be entirely satisfying in practice. Figure 2.33 depicts two meshes generated by Vavasis' QMG mesh generator. The mesh at left is quite good, whereas the mesh at right contains some tetrahedra of marginal quality, with many small angles visible on the surface.

In practice, the theoretically good mesh generation algorithms of Bern, Eppstein, and Gilbert [11] and Mitchell and Vavasis [69] often create an undesirably large number of elements. Although both algorithms are size-optimal, the constant hidden in the definition of size-optimality is large, and although both algorithms rarely create as many elements as their theoretical worst-case bounds suggest, they typically create too many nonetheless. In contrast, the Finite Octree mesh generator of Shephard and Georges [87] generates fewer tetrahedra, but offers no guarantee. Shephard and Georges eliminate poor elements, wherever possible, through mesh smoothing, described below.

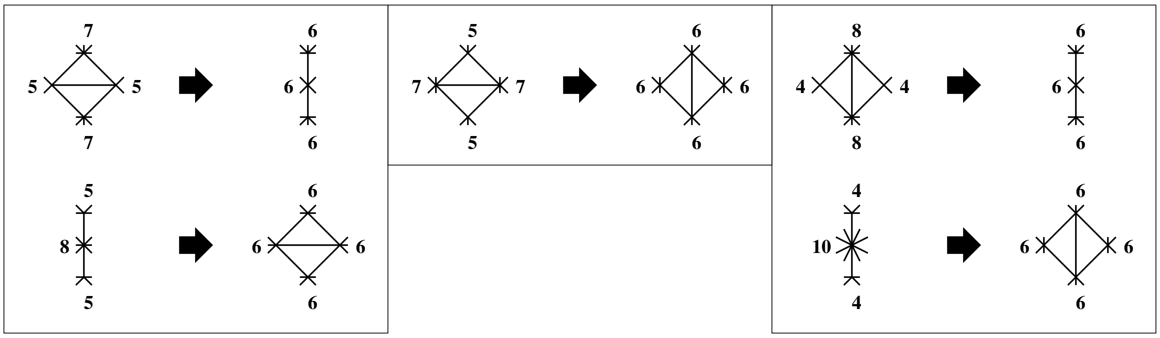


Figure 2.34: A selection of topological local transformations. Each node is labeled with its degree. These labels represent ideal cases, and are not the only cases in which these transformations would occur.

2.2.4 Smoothing and Topological Transformations

All the algorithms discussed thus far have the property that once they have decided to insert a vertex, the vertex is rooted permanently in place. In this section, I discuss techniques that violate this permanence. These are not mesh generation methods; rather, they are mesh improvement procedures, which may be applied to a mesh generated by any of the methods discussed heretofore.

Smoothing is a technique wherein mesh vertices are moved to improve the quality of the adjoining elements. No changes are made to the topology of the mesh. Of course, vertices that lie in mesh boundaries may be constrained so that they can only move within a segment or facet, or they may be unable to move at all.

The most famous smoothing technique is *Laplacian smoothing*, in which a vertex is moved to the centroid of the vertices to which it is connected [49], if such a move does not create collapsed or inverted elements. Typically, a smoothing algorithm will run through the entire set of mesh vertices several times, smoothing each vertex in turn. Laplacian smoothing is reasonably effective in two dimensions, but performs poorly in three.

Were Laplacian smoothing not so easy to implement and so fast to execute, it would be completely obsolete. Much better smoothing algorithms are available, based on constrained optimization techniques [75]. The current state of the art is probably the nonsmooth optimization algorithm discussed by Freitag, Jones, and Plassman [38] and Freitag and Ollivier-Gooch [39, 40]. The latter authors report considerable success with a procedure that maximizes the minimum sine of the dihedral angles of the tetrahedra adjoining the vertex being smoothed.

Another approach to mesh improvement is to use the *topological transformations* outlined by Canann [14], which are similar to ideas proposed by Frey and Field [42]. Examples of some transformations are illustrated in Figure 2.34. The familiar edge flip is included, but the other transformations have the effect of inserting or deleting a vertex. An unusual aspect of Canann's approach is that he applies transformations based on the topology, rather than the geometry, of a mesh. In two dimensions, the ideal degree of a vertex is presumed to be six (to echo the structure of a lattice of equilateral triangles), and transformations are applied in an attempt to bring the vertices of the mesh as close to this ideal as possible. Canann claims that his method is fast because it avoids geometric calculations and makes decisions based on simple topological measures. The method relies upon smoothing to iron out any geometric irregularities after the transformations are complete. The research is notable because of the unusually large number of transformations under consideration.

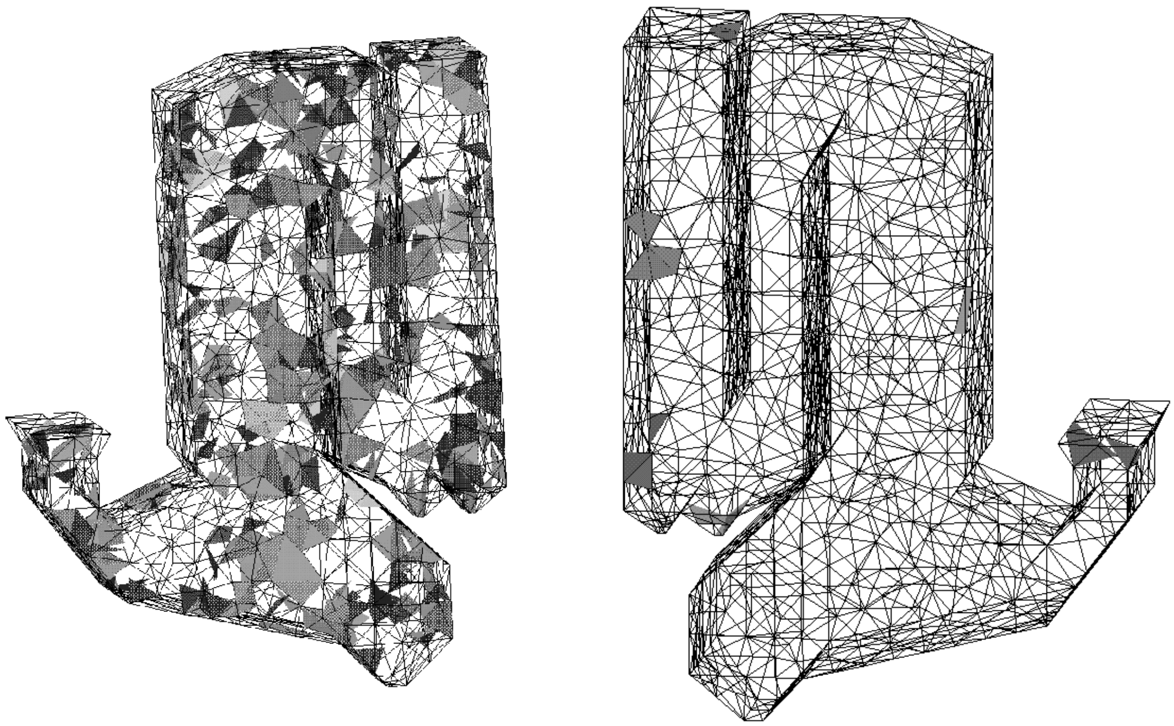


Figure 2.35: Tire incinerator mesh before and after mesh improvement. Shaded tetrahedra have dihedral angles smaller than 18° or greater than 162°. (Courtesy Lori Freitag and Carl Ollivier-Gooch.)

Other researchers have considered mixing smoothing with topological transformations, but typically consider only a limited set of transformations, often restricted to 2-3 and 3-2 flips. For instance, Golias and Tsiboukis [45] report obtaining good results in tetrahedral mesh improvement by alternating between Laplacian smoothing and flipping.

A more sophisticated approach is taken by Freitag and Ollivier-Gooch [39, 40], who combine optimization-based smoothing with several transformations, including 2-3 and 3-2 flips, as well as another set of transformations they refer to as "edge swapping". Figure 2.35 demonstrates the results obtained by these techniques. In these before-and-after images, tetrahedra with poor dihedral angles are shaded. Before mesh improvement, the dihedral angles range from 0.66° to 178.88°. Afterward, they range from 13.67° to 159.82°.

As Delaunay tetrahedralizations lack the optimality properties of their two-dimensional counterparts, it is natural to ask whether one should forgo the Delaunay criterion, and instead use flips to directly maximize the minimum solid angle. Joe [52] studies this question experimentally, and concludes that a procedure that performs local flips to locally optimize the minimum solid angle is notably inferior to the Delaunay tetrahedralization. However, if one first constructs the Delaunay tetrahedralization, and then applies additional flips to locally improve the minimum solid angle, one does better than the Delaunay tetrahedralization alone. Joe's results indicate that a tetrahedralization that is locally optimal with respect to solid angle may be far from globally optimal. Although the Delaunay tetrahedralization does not maximize the minimum solid angle, it certainly seems to optimize something useful for mesh generation. Marcum and Weatherill [63] suggest that alternating between the Delaunay criterion and a min-max criterion (minimize the maximum dihedral angle) works even better.

Later research by Joe [55] indicates that local improvements can often be made by considering the effect of two consecutive flips (even though the first of the two flips may worsen element quality). Joe identifies several dual transformations that are frequently effective in practice, and several that rarely prove to be useful.

All of these mesh improvement techniques are applicable to meshes generated by the algorithms described in Chapters 3 and 4. However, I will not explore them further in this document.

Chapter 3

Two-Dimensional Delaunay Refinement Algorithms for Quality Mesh Generation

Delaunay refinement algorithms for mesh generation operate by maintaining a Delaunay or constrained Delaunay triangulation, which is refined by inserting carefully placed vertices until the mesh meets constraints on element quality and size.

These algorithms are successful because they exploit several favorable characteristics of Delaunay triangulations. One such characteristic, already mentioned in Chapter 2, is Lawson's result that a Delaunay triangulation maximizes the minimum angle among all possible triangulations of a point set. Another feature is that inserting a vertex is a local operation, and hence is inexpensive except in unusual cases. The act of inserting a vertex to improve poor quality elements in one part of a mesh will not unnecessarily perturb a distant part of the mesh that has no bad elements. Furthermore, Delaunay triangulations have been extensively studied, and good algorithms are available.

The greatest advantage of Delaunay triangulations is less obvious. The central question of any Delaunay refinement algorithm is "where should the next vertex be inserted?" As this chapter will demonstrate, a reasonable answer is "as far from other vertices as possible." If a new vertex is inserted too close to another vertex, the resulting small edge will engender thin triangles.

Because a Delaunay triangle has no vertices in its circumcircle, a Delaunay triangulation is an ideal search structure for finding points that are far from other vertices. (It's no coincidence that the circumcenter of each triangle of a Delaunay triangulation is a vertex of the corresponding Voronoi diagram.)

This chapter begins with a review of Delaunay refinement algorithms introduced by L. Paul Chew and Jim Ruppert. Ruppert [81] proves that his algorithm produces nicely graded, size-optimal meshes with no angles smaller than about 20.7°. I show that Ruppert's analysis technique can be used to prove that Chew's second published Delaunay refinement algorithm [21] can produce nicely graded size-optimal meshes with no angles smaller than about 26.5°. Chew proves that his algorithm can produce meshes with no angles smaller than 30°, albeit without any guarantees of grading or size-optimality. I generalize Chew's idea so that it can be applied to Ruppert's algorithm (and later to three-dimensional Delaunay refinement). I also discuss theoretical and practical issues in triangulating regions with small angles. The foundations built here undergird the three-dimensional Delaunay refinement algorithms examined in the next chapter.

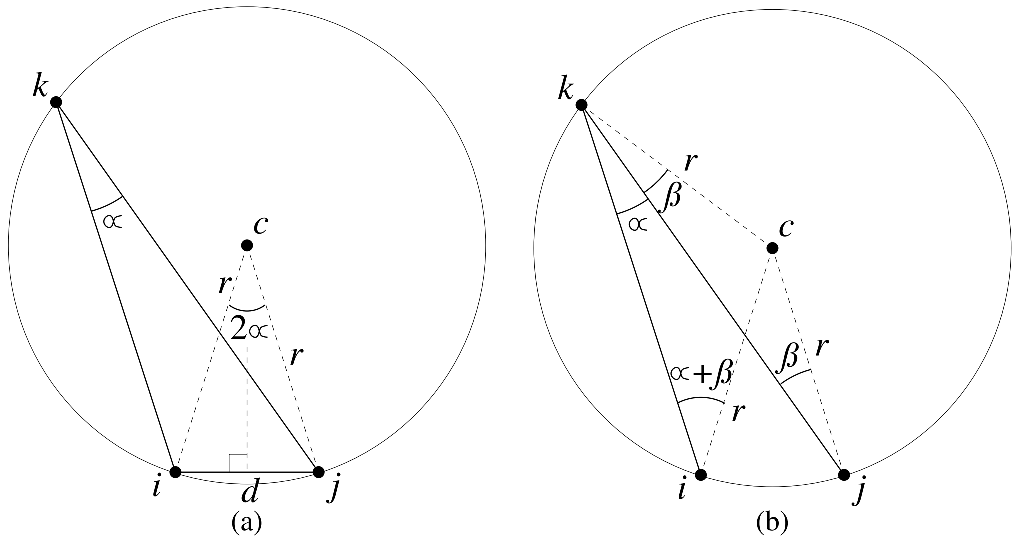


Figure 3.1: (a) Diagram for proof that $d = 2r \sin \alpha$. (b) Diagram for proof that $\angle icj = 2\angle ikj$.

3.1 A Quality Measure for Simplices

In the finite element community, there are a wide variety of measures in use for the quality of an element, the most obvious being the smallest and largest angles of each simplex. Miller, Talmor, Teng, and Walkington [66] have pointed out that the most natural and elegant measure for analyzing Delaunay refinement algorithms is the *circumradius-to-shortest edge ratio* of a simplex: the radius of the circumsphere of the simplex divided by the length of the shortest edge of the simplex. For brevity, I will occasionally refer to this ratio as the *quality* of a simplex. One would like this ratio to be as small as possible.

In two dimensions, a triangle's circumradius-to-shortest edge ratio is a function of its smallest angle. Let $\triangle ijk$ have circumcenter c and circumradius r, as illustrated in Figure 3.1(a). Suppose the length of edge ij is d, and the angle opposite this edge is $\alpha = \angle ikj$.

It is a well-known geometric fact that $\angle icj = 2\alpha$. See Figure 3.1(b) for a derivation. Let $\beta = \angle jkc$. Because $\triangle kci$ and $\triangle kcj$ are isosceles, $\angle kci = 180^\circ - 2(\alpha + \beta)$ and $\angle kcj = 180^\circ - 2\beta$. Subtracting the former from the latter, $\angle icj = 2\alpha$. (This derivation holds even if β is negative.)

Returning to Figure 3.1(a), it is apparent that $\sin \alpha = d/(2r)$. It follows that if the triangle's shortest edge has length d, then α is its smallest angle. Hence, if B is an upper bound on the circumradius-to-shortest edge ratio of all triangles in a mesh, then there is no angle smaller than $\arcsin \frac{1}{2B}$ (and vice versa). A triangular mesh generator is wise to make B as small as possible.

Unfortunately, a bound on circumradius-to-shortest edge ratio does not imply an angle bound in dimensions higher than two. Nevertheless, the ratio is a useful measure for understanding Delaunay refinement in higher dimensions.

With these facts in mind, I shall describe two-dimensional Delaunay refinement algorithms due to Paul Chew and Jim Ruppert that act to bound the maximum circumradius-to-shortest edge ratio, and hence bound the minimum angle of a triangular mesh.

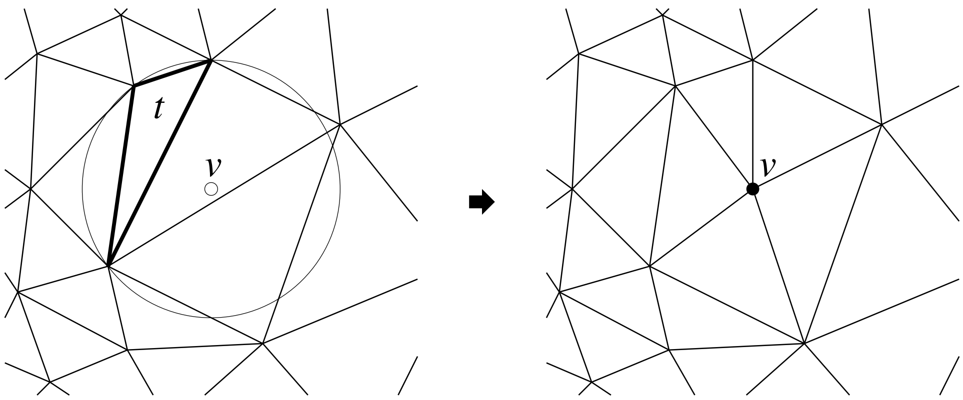


Figure 3.2: Any triangle whose circumradius-to-shortest edge ratio is larger than some bound B is split by inserting a vertex at its circumcenter. The Delaunay property is maintained, and the triangle is thus eliminated. Every new edge has length at least B times that of shortest edge of the poor triangle.

3.2 Chew's First Delaunay Refinement Algorithm

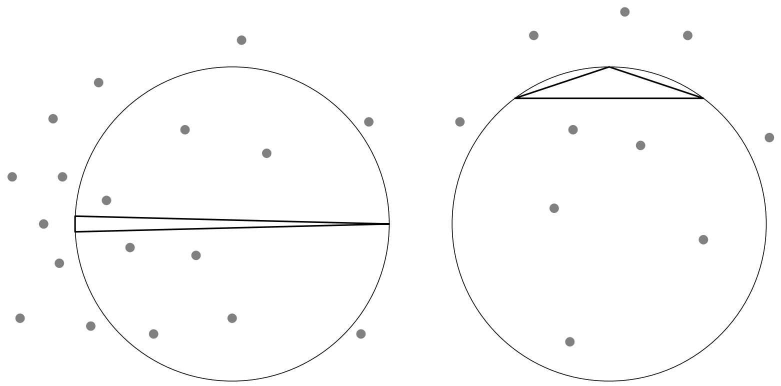
Paul Chew has published at least two Delaunay refinement algorithms of great interest. The first, described here, produces triangulations of uniform density [19]. The second, which can produce graded meshes [21], will be discussed in Section 3.4.

3.2.1 The Key Ideas Behind Delaunay Refinement

The central operation of Chew's, Ruppert's, and my own Delaunay refinement algorithms is the insertion of a vertex at the circumcenter of a triangle of poor quality. The Delaunay property is maintained, preferably by Lawson's algorithm or the Bowyer/Watson algorithm for the incremental update of Delaunay triangulations. The poor triangle cannot survive, because its circumcircle is no longer empty. For brevity, I refer to the act of inserting a vertex at a triangle's circumcenter as *splitting* a triangle. The idea dates back at least to the engineering literature of the mid-1980s [41].

The main insight of all the Delaunay refinement algorithms is that Delaunay refinement is guaranteed to terminate if the notion of "poor quality" includes only triangles that have a circumradius-to-shortest edge ratio larger than some appropriate bound B. Recall that the only new edges created by the Delaunay insertion of a vertex v are edges connected to v (see Figure 3.2). Because v is the circumcenter of some triangle t, and there were no vertices inside the circumcircle of t before v was inserted, no new edge can be shorter than the circumradius of t. Because t has a circumradius-to-shortest edge ratio larger than B, every new edge has length at least B times that of the shortest edge of t.

Henceforth, a triangle whose circumradius-to-shortest edge ratio is greater than B is said to be *skinny*. Figure 3.3 provides an intuitive illustration of why all skinny triangles are eventually eliminated by Delaunay refinement. The new vertices that are inserted into a triangulation (grey dots) are spaced roughly according to the length of the shortest nearby edge. Because skinny triangles have relatively large circumradii, their circumcircles are inevitably popped. When enough vertices are introduced that the spacing of vertices is somewhat uniform, large empty circumcircles cannot adjoin small edges, and no skinny triangles can remain in the Delaunay triangulation. Fortunately, the spacing of vertices does not need to be so uniform that the mesh is poorly graded; this fact is formalized in Section 3.3.4.



Needles **Caps**

Figure 3.3: Skinny triangles have circumcircles larger than their smallest edges. Each skinny triangle may be classified as a needle, whose longest edge is much longer than its shortest edge, or a cap, which has an angle close to $180°$. (The classifications are not mutually exclusive.)

Chew's algorithms both employ a bound of $B = 1$ (though, as we shall see, the early algorithm is stricter). With this bound, every new edge created is at least as long as some other edge already in the mesh. This fact is sufficient to prove that Delaunay refinement terminates. Suppose that Delaunay refinement is applied to improve the angles of a triangulation T whose shortest edge has length $h_{\min}$. Delaunay refinement never introduces a shorter edge, so any two vertices are separated by a distance of at least $h_{\min}$. Hence, if each vertex is the center of a disk whose radius is $h_{\min}/2$, all such disks have disjoint interiors. Let $\mathcal{B}(\mathcal{T})$ be a bounding box of T that is everywhere a distance of at least $h_{\min}/2$ from T; all the discs defined above are inside $\mathcal{B}(\mathcal{T})$. Hence, the number of vertices times $\pi h_{\min}^2/4$ cannot exceed the total area of $\mathcal{B}(\mathcal{T})$, and termination is inevitable.

The implication is that the augmented triangulation will eventually run out of places to put vertices, because vertices may only be placed at least a distance of $h_{\min}$ away from all other vertices. At this time (if not sooner), all triangles have a quality of one or smaller, and Delaunay refinement terminates. Upon termination, because no triangle has a circumradius-to-shortest edge ratio larger than one, the mesh contains no angle smaller than $30°$.

Chew's first algorithm splits any triangle whose circumradius is greater than $h_{\min}$, and hence creates a uniform mesh. Chew's second Delaunay refinement algorithm relaxes this stricture, splitting only triangles whose circumradius-to-shortest edge ratios are greater than one, and hence produces graded meshes in practice, although Chew supplies no theoretical guarantee of good grading. In Section 3.4.2, I will show that by slightly relaxing the quality bound, a guarantee of good grading can be obtained.

When the early algorithm terminates, all edge lengths are bounded between $h_{\min}$ and $2h_{\min}$. The upper bound follows because if the length of a Delaunay edge is greater than $2h_{\min}$, then at least one of the two Delaunay triangles that contain it has a circumradius larger than $h_{\min}$ and is eligible for splitting.

My description of Delaunay refinement thus far has a gaping hole: mesh boundaries have not been accounted for. The flaw in the procedure I have presented above is that the circumcenter of a skinny triangle might not lie in the triangulation at all. Figure 3.4 illustrates an example in which there is a skinny triangle, but no vertex can be placed inside its circumcircle without creating an edge smaller than $h_{\min}$, which would compromise the termination guarantee.

The remainder of this chapter, and the entirety of the next chapter, are devoted to the problem of mod-

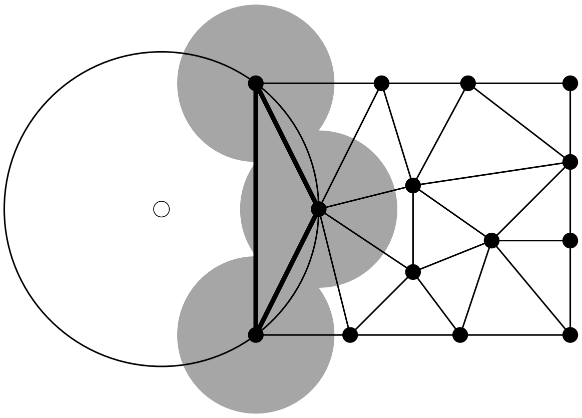


Figure 3.4: The bold triangle could be eliminated by inserting a vertex in its circumcircle. However, a vertex cannot be placed outside the triangulation, and it is forbidden to place a vertex within a distance of $h_{\min}$ from any other vertex. The forbidden region includes the shaded disks, which entirely cover the bold triangle.

ifying Delaunay refinement so that it respects mesh boundaries. Before commencing that quest, I want to emphasize that the central idea of Delaunay refinement generalizes without change to higher dimensions. (For instance, Dey, Bajaj, and Sugihara [28] describe a straightforward generalization of Chew's first algorithm to three dimensions.) Imagine a triangulation that has no boundaries—perhaps it has infinite extent, or perhaps it is mapped onto a manifold that is topologically equivalent to a torus (or higher-dimensional generalization thereof). Regardless of the dimensionality, Delaunay refinement can eliminate all simplices having a circumradius-to-shortest edge ratio greater than one, without creating any edge smaller than the smallest edge already present. Unfortunately, boundaries complicate mesh generation immensely, and the difficulty of coping with boundaries increases in higher dimensions.

3.2.2 Mesh Boundaries in Chew's First Algorithm

The input to Chew's algorithm is a PSLG that is presumed to be *segment-bounded*, meaning that the region to be triangulated is entirely enclosed within segments. (Any PSLG may be converted to a segment-bounded PSLG by any two-dimensional convex hull algorithm, if a convex triangulation is desired.) Untriangulated holes in the PSLG are permitted, but these must also be bounded by segments. A segment must lie anywhere a triangulated region of the plane meets an untriangulated region.

For some parameter h chosen by the user, all segments are divided into subsegments whose lengths are in the range $[h, \sqrt{3}h]$. New vertices are placed at the division points. The parameter h must be chosen small enough that some such partition is possible. Furthermore, h may be no larger than the smallest distance between any two vertices of the resulting partition. (If a vertex is close to a segment, this latter restriction may necessitate a smaller value of h than would be indicated by the input vertices alone.)

The constrained Delaunay triangulation of this modified PSLG is computed. Next, Delaunay refinement is applied. Circumcenters of triangles whose circumradii are larger than h are inserted, one at a time. When no such triangle remains, the algorithm terminates.

Because no subsegment has length greater than $\sqrt{3}h$, and specifically because no boundary subsegment has such length, the circumcenter of any triangle whose circumradius exceeds h falls within the mesh, at a distance of at least $h/2$ from any subsegment. Why? If a circumcenter is a distance less than $h/2$ from a subsegment whose length is no greater than $\sqrt{3}h$, then the circumcenter is a distance less than h from one of the subsegment's endpoints.

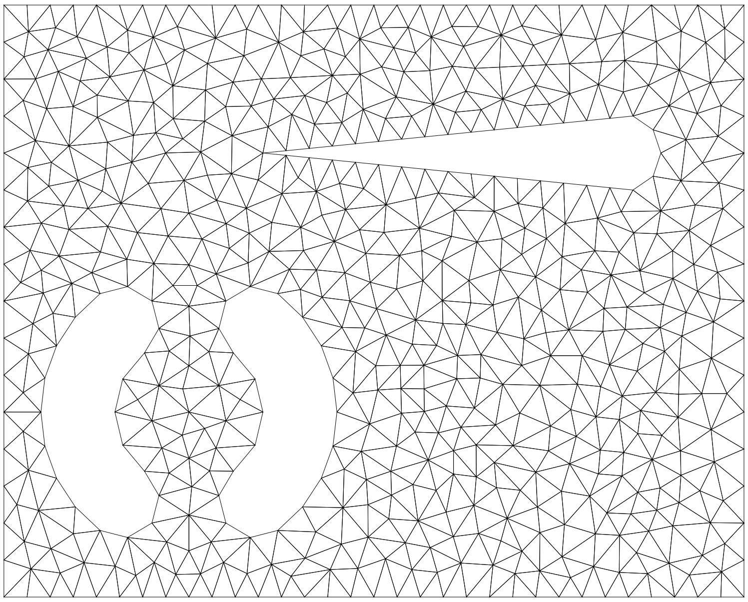


Figure 3.5: A mesh generated by Chew's first Delaunay refinement algorithm. (Courtesy Paul Chew).

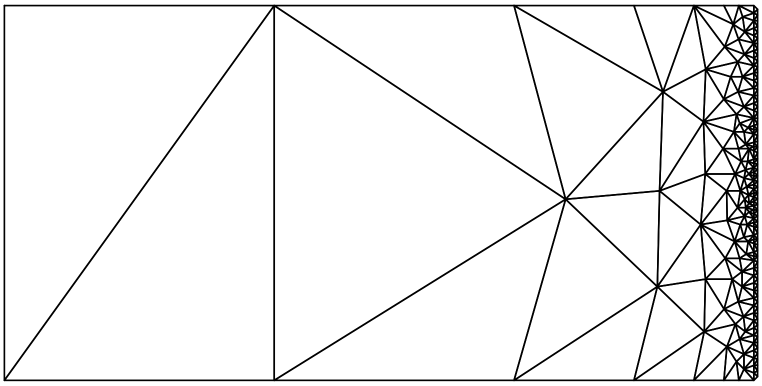


Figure 3.6: A demonstration of the ability of the Delaunay refinement algorithm to achieve large gradations in triangle size while constraining angles. No angles are smaller than 24°.

Chew's early algorithm handles boundaries in a simple and elegant manner, at the cost that it only produces meshes of uniform density, as illustrated in Figure 3.5. The remainder of this thesis examines Delaunay refinement algorithms that generate graded meshes.

3.3 Ruppert's Delaunay Refinement Algorithm

Jim Ruppert's algorithm for two-dimensional quality mesh generation [82] is perhaps the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice. It extends Chew's early algorithm by allowing the density of triangles to vary quickly over short distances, as illustrated in Figure 3.6. The number of triangles produced is typically smaller than the number produced either by Chew's algorithm or the Bern-Eppstein-Gilbert quadtree algorithm [11] (discussed in Section 2.2.3), as Figure 3.7 shows.

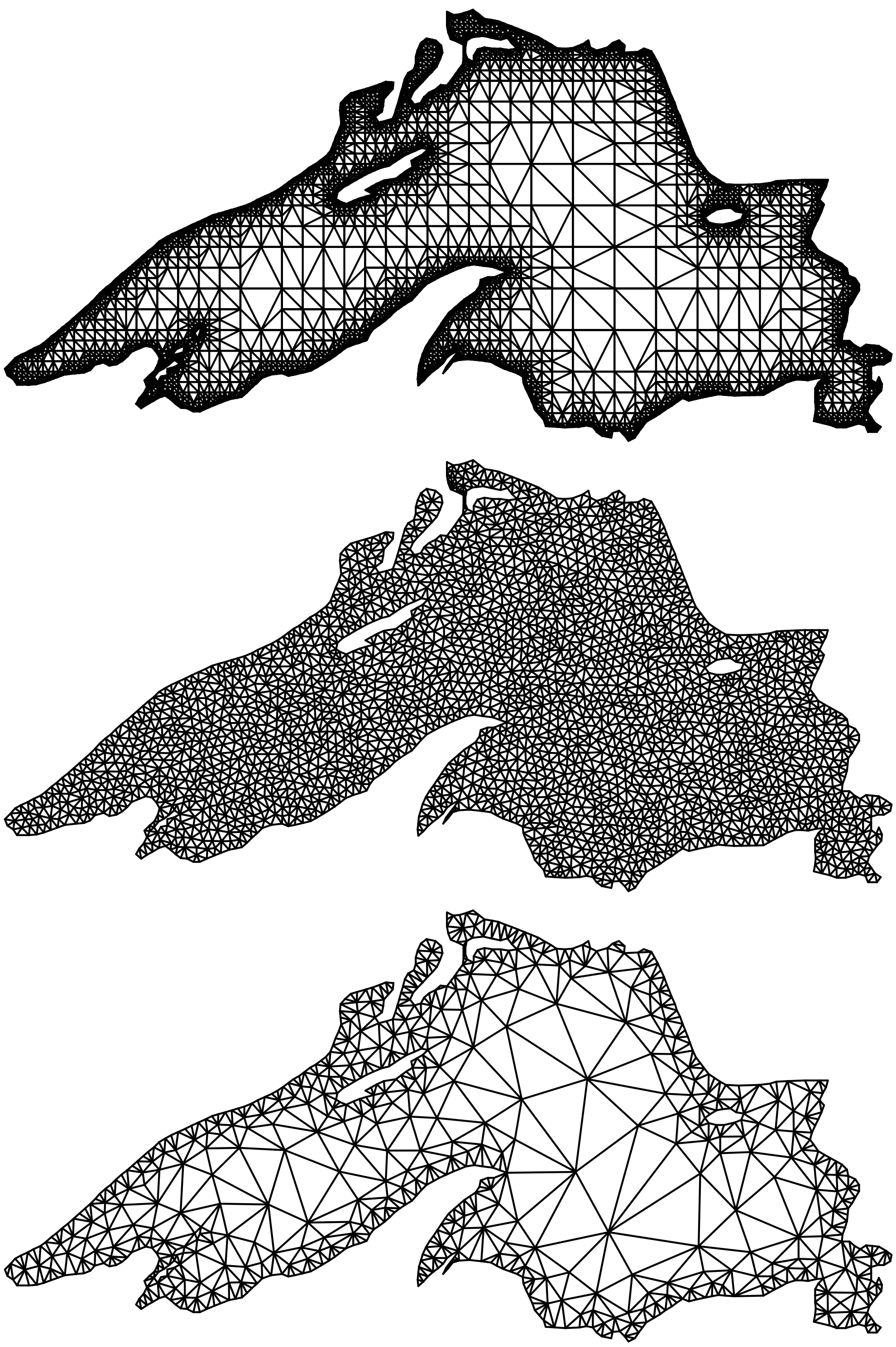


Figure 3.7: Meshes generated by the Bern-Eppstein-Gilbert quadtree-based algorithm (top), Chew's first Delaunay refinement algorithm (center), and Ruppert's Delaunay refinement algorithm (bottom). (The first mesh was produced by the program `tripoint`, courtesy Scott Mitchell.)

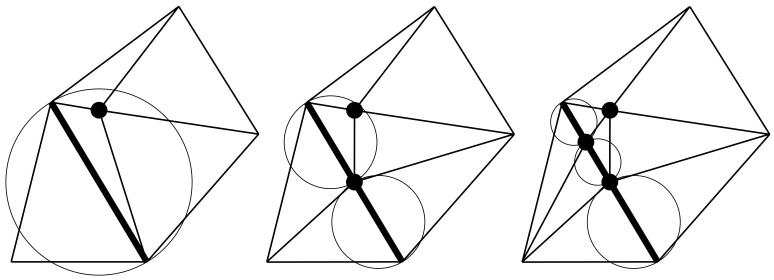


Figure 3.8: Segments are split recursively (while maintaining the Delaunay property) until no subsegments are encroached.

I have already mentioned that Chew independently developed a similar algorithm [21]. It may be worth noting that Ruppert's earliest publications of his results [80, 81] slightly predate Chew's. I present Ruppert's algorithm first because it is accompanied by a theoretical framework with which he proves its ability to produce meshes that are both nicely graded and *size-optimal*. Size optimality means that, for a given bound on minimum angle, the number of elements composing any mesh produced by the algorithm is at most a constant factor larger than the number in the smallest possible mesh that meets the same angle bound. (The constant depends only upon the minimum allowable angle, and is too large to be useful as a practical bound.) In Section 3.4.2, I will discuss how to apply Ruppert's framework to Chew's algorithm, for which better bounds can be derived.

3.3.1 Description of the Algorithm

Like Chew's algorithms, Ruppert's algorithm takes a segment-bounded PSLG as its input. Unlike Chew's algorithms, Ruppert's algorithm may start with either a constrained or unconstrained Delaunay triangulation. Ruppert's presentation of the algorithm is based on unconstrained triangulations, and it is interesting to see how the algorithm responds to missing segments, so assume that we start with the Delaunay triangulation of the input vertices, ignoring the input segments. Input segments that are missing from the triangulation will be inserted as a natural consequence of the algorithm.

Again like Chew's algorithms, Ruppert's refines the mesh by inserting additional vertices (using Lawson's algorithm to maintain the Delaunay property) until all triangles satisfy the quality constraint. Vertex insertion is governed by two rules.

- The *diametral circle* of a subsegment is the (unique) smallest circle that contains the subsegment. A subsegment is said to be *encroached* if a vertex lies strictly inside its diametral circle, or if the subsegment does not appear in the triangulation. (Recall that the latter case generally implies the former, the only exceptions being degenerate examples where several vertices lie precisely on the diametral circle.) Any encroached subsegment that arises is immediately bisected by inserting a vertex at its midpoint, as illustrated in Figure 3.8. The two subsegments that result have smaller diametral circles, and may or may not be encroached themselves.
- As with Chew's algorithm, each skinny triangle (having a circumradius-to-shortest edge ratio larger than some bound B) is normally split by inserting a vertex at its circumcenter. The Delaunay property guarantees that the triangle is eliminated, as illustrated in Figure 3.9. However, if the new vertex would encroach upon any subsegment, then it is not inserted; instead, all the subsegments it would encroach upon are split.

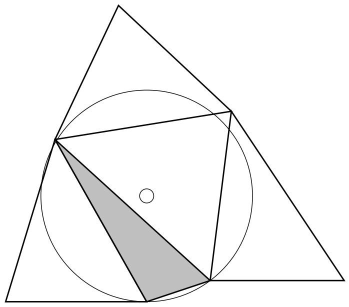


Figure 3.9: Each skinny triangle is split by inserting a vertex at its circumcenter and maintaining the Delaunay property.

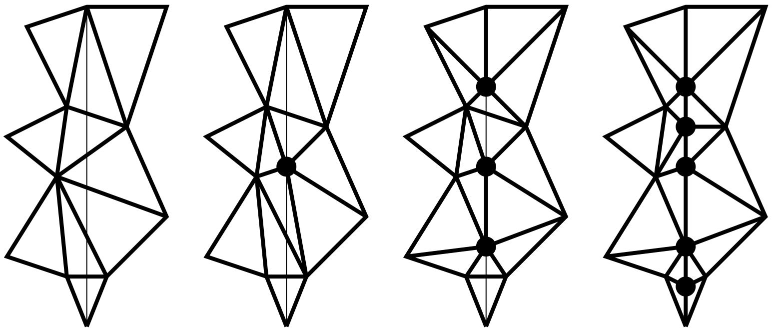


Figure 3.10: Missing segments are forced into the mesh by the same recursive splitting procedure used for encroached subsegments that are in the mesh. In this sequence of illustrations, the thin line represents a segment missing from the triangulation.

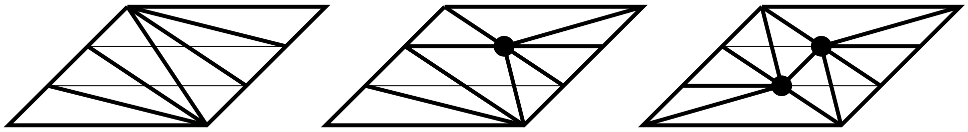


Figure 3.11: In this example, two segments (thin lines) must be forced into a triangulation. The first is successfully forced in with a single vertex insertion, but the attempt to force in the second eliminates a subsegment of the first.

Encroached subsegments are given priority over skinny triangles.

An implementation may give encroached subsegments that are not present in the mesh priority over encroached subsegments that are present (though it isn't necessary). If this option is chosen, the algorithm's first act is to force all missing segments into the mesh. Each missing segment is bisected by inserting a vertex into the mesh at the midpoint of the segment (more accurately, at the midpoint of the place where the segment should be). After the mesh is adjusted to maintain the Delaunay property, the two resulting subsegments may appear in the mesh. If not, the procedure is repeated recursively for each missing subsegment until the original segment is represented by a linear sequence of edges of the mesh, as illustrated in Figure 3.10. We are assured of eventual success because the Delaunay triangulation always connects a vertex to its nearest neighbor; once the spacing of vertices along a segment is sufficiently small, its entire length will be represented. In the engineering literature, this process is sometimes called *stitching*.

Unfortunately, the insertion of a vertex to force a segment into the triangulation may eliminate a subseg-

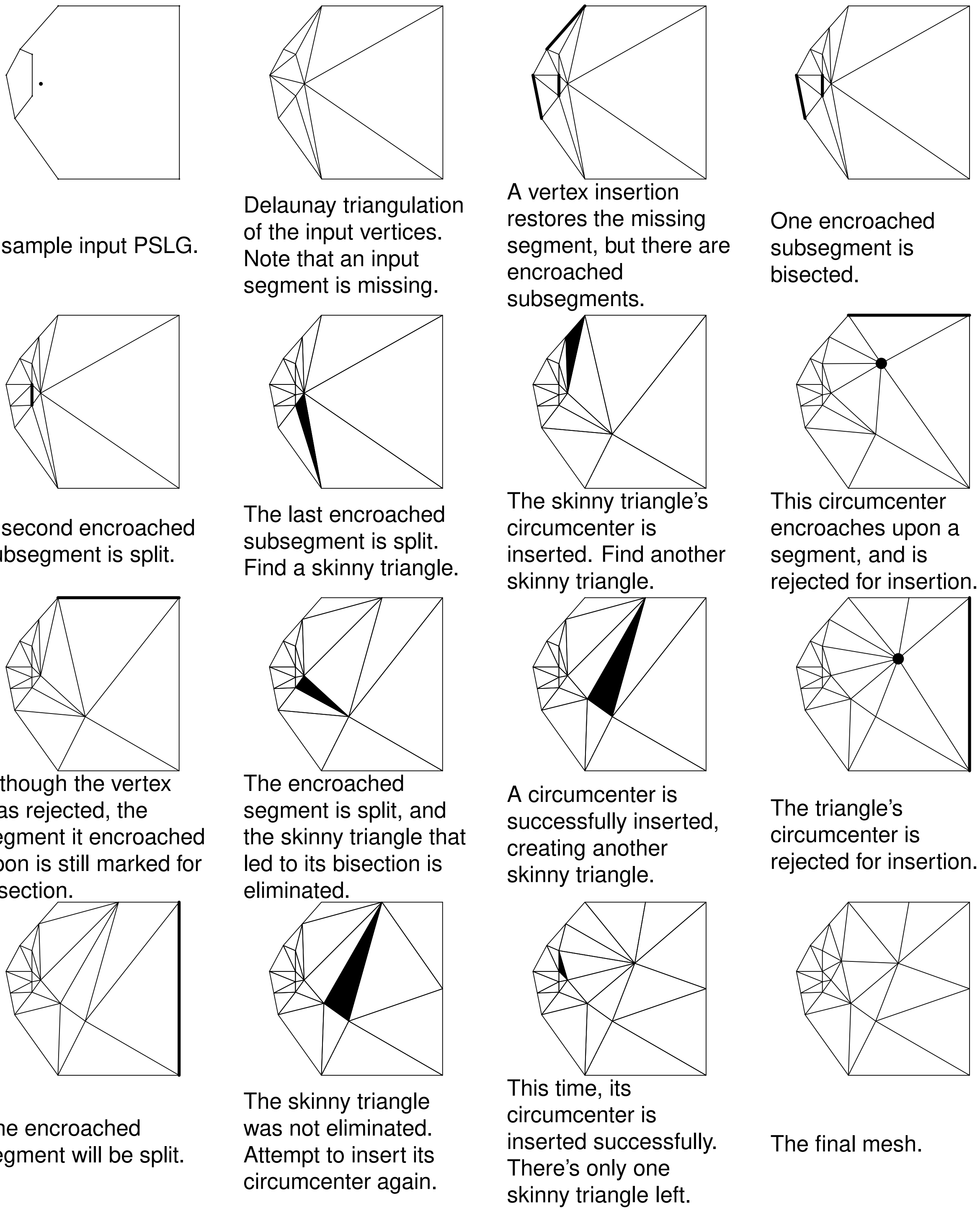


Figure 3.12: A complete run of Ruppert's algorithm with the quality bound $B = \sqrt{2}$. The first two images are the input PSLG and the (unconstrained) Delaunay triangulation of its vertices. In each image, highlighted subsegments or triangles are about to be split, and highlighted vertices are rejected for insertion because they encroach upon a subsegment.

ment of some other segment (Figure 3.11). The subsegment thus eliminated is hence encroached, and must be split further. To avoid eliminating subsegments, one could *lock* subsegments of the mesh by marking the edges that represent them to indicate that they are constrained. Flipping of such constrained edges is forbidden. However, subsegments whose diametral circles are nonempty are still considered encroached, and will still be split eventually; hence, it makes little material difference to the algorithm whether one chooses to lock subsegments. Nevertheless, locked subsegments yield faster implementations and will be necessary for Chew's second algorithm. The reader may wish to assume that all subsegments become permanent as soon as they appear, although it was not part of Ruppert's original specification.

If a subsegment is missing from a Delaunay triangulation, then the subsegment is not Delaunay, and there must be a vertex is its diametral circle. (There is a degenerate exception to this rule, wherein several vertices fall on the diametral circle, but this exception is not theoretically problematic.) This observation is important because it unifies the theoretical treatment of missing subsegments and subsegments that are present in the mesh but encroached.

After all encroached subsegments have been recursively bisected, and no subsegments are encroached, all edges (including subsegments) of the triangulation are Delaunay. A mesh produced by Ruppert's algorithm is truly Delaunay, and not merely constrained Delaunay.

Figure 3.12 illustrates the generation of a mesh by Ruppert's algorithm from start to finish. Several characteristics of the algorithm are worth noting. First, if the circumcenter of a skinny triangle is rejected for insertion, it may still be successfully inserted later, after the subsegments it encroaches upon have been split. On the other hand, the act of splitting those subsegments is sometimes enough to eliminate the skinny triangle. Second, the smaller features at the left end of the mesh lead to the insertion of some vertices toward the right, but the size of the elements to the right remains larger than the size of the elements to the left. The smallest angle in the final mesh is $21.8°$.

There is a loose end to tie up. One might ask what should happen if the circumcenter of a skinny triangle falls outside the triangulation. Fortunately, the following lemma shows that the question is moot.

Lemma 13 *Let T be a segment-bounded Delaunay triangulation (hence, any edge of T that belongs to only one triangle is a subsegment). Suppose that T has no encroached subsegments. Let v be the circumcenter of some triangle t of T. Then v lies in T.*

Proof: Suppose for the sake of contradiction that v lies outside T. Let c be the centroid of t; c clearly lies inside T. Because the triangulation is segment-bounded, the line segment cv must cross some subsegment s, as Figure 3.13 illustrates. Because cv is entirely contained in the interior of the circumcircle of t, the circumcircle must contain a portion of s; but the Delaunay property requires that the circumcircle be empty, so the circumcircle cannot contain the endpoints of s.

Say that a point is *inside* s if it is on the same side of s as c, and *outside* s if it is on the same side of s as v. Because the center v of the circumcircle of t is outside s, the portion of the circumcircle that lies strictly inside s (the bold arc in the illustration) is entirely enclosed by the diametral circle of s. The vertices of t lie upon t's circumcircle and are (not strictly) inside s. Up to two of the vertices of t may be the endpoints of s, but at least one vertex of t must lie strictly inside the diametral circle of s. But by assumption T has no encroached subsegments; the result follows by contradiction. ■

Lemma 13 offers another reason why encroached subsegments are given priority over skinny triangles. Because a circumcenter is inserted only when there are no encroached subsegments, one is assured that the

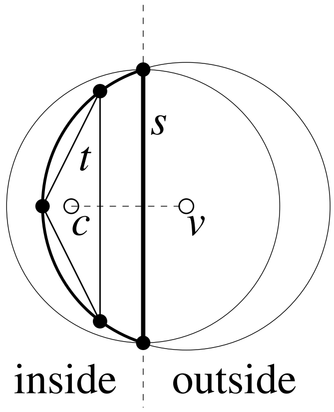


Figure 3.13: If the circumcenter v of a triangle t lies outside the triangulation, then some subsegment s is encroached.

circumcenter will be within the triangulation. Conversely, the act of splitting encroached subsegments rids the mesh of triangles whose circumcircles lie outside it. The lemma also explains why the triangulation should be completely bounded by segments before applying the refinement algorithm.

In addition to being required to satisfy a quality criterion, triangles can also be required to satisfy a maximum size criterion. If a finite element simulation requires that elements be small enough to model a phenomenon within some error bound, one may specify an upper bound on allowable triangle areas or edge lengths as a function of location in the mesh. Triangles that exceed the local upper bound are split, whether they are skinny or not. So long as the function bounding the sizes of triangles is itself bounded everywhere above some positive constant, there is no threat to the algorithm's termination guarantee.

3.3.2 Local Feature Size

The claim that Ruppert's algorithm produces nicely graded meshes is based on the fact that the spacing of vertices at any location in the mesh is within a constant factor of the sparsest possible spacing. To formalize the idea of "sparsest possible spacing," Ruppert introduces a function called the *local feature size*, which is defined over the domain of the input PSLG.

Given a PSLG X, the local feature size $\text{lfs}(p)$ at any point p is the radius of the smallest disk centered at p that intersects two nonincident vertices or segments of X. Figure 3.14 gives examples of such disks for a variety of points.

The local feature size of a point is proportional to the sparsest possible spacing of vertices in the neighborhood of that point. The function $\text{lfs}(\cdot)$ is continuous and has the property that its directional derivatives are bounded in the range $[-1, 1]$. The latter property, proven by the following lemma, sets a bound on the fastest possible grading of element sizes in a mesh.

Lemma 14 (Ruppert [82]) *For any PSLG X, and any two points u and v in the plane,*

$$\text{lfs}(v) \leq \text{lfs}(u) + |uv|.$$

Proof: The disk having radius $\text{lfs}(u)$ centered at u intersects two nonincident features of X. The disk having radius $\text{lfs}(u) + |uv|$ centered at v contains the prior disk, and thus also intersects the same two nonincident

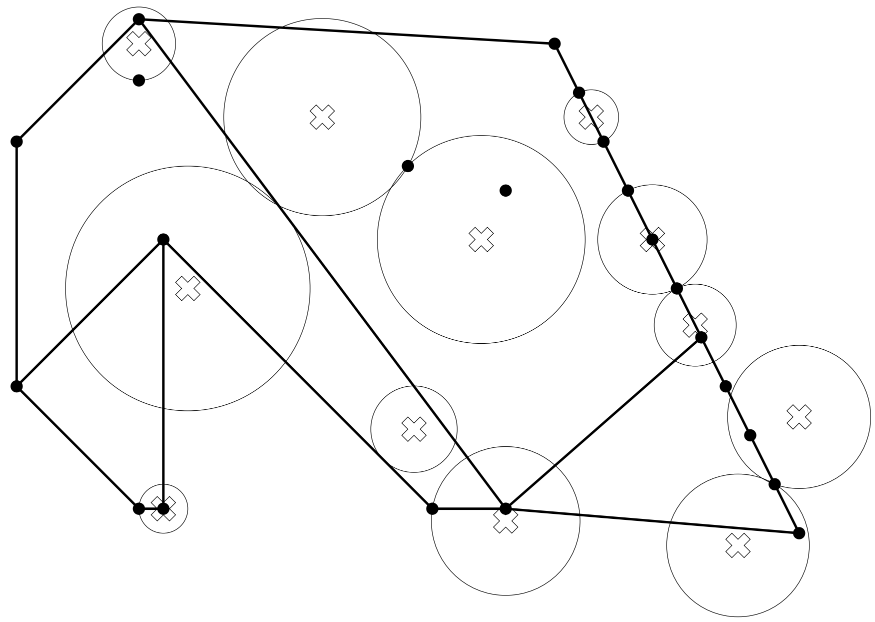


Figure 3.14: The radius of each disk illustrated is the local feature size of the point at its center.

features of X. Hence, the largest disk centered at v that contains two nonincident features of X has radius no larger than $\mathrm{lfs}(u) + |uv|$. ■

This lemma generalizes without change to higher dimensions, so long as the question "Which pairs of points are said to lie on nonincident features?" has a consistent answer that is independent of u and v. In essence, the proof relies only on the triangle inequality: if u is within a distance of $\mathrm{lfs}(u)$ of each of two nonincident features, then v is within a distance of $\mathrm{lfs}(u) + |uv|$ of each of those same two features.

3.3.3 Proof of Termination

In this section and the next, I present two proofs of the termination of Ruppert's algorithm. The first is similar to the proof that Chew's early algorithm terminates, and is included for its intuitive value. The second is taken from Ruppert, but is rewritten in a somewhat different form to bring out features that will figure prominently in my own extensions. The second proof shows that the algorithm produces meshes that are nicely graded and size-optimal.

Both proofs require that $B \geq \sqrt{2}$, and any two incident segments (segments that share an endpoint) in the input PSLG must be separated by an angle of 60° or greater. (Ruppert asks for angles of at least 90°, but an improvement to the original proof is made here.) For the second proof, these inequalities must be strict.

With each vertex v, associate an *insertion radius* r_v, equal to the length of the shortest edge connected to v immediately after v is introduced into the triangulation. Consider what this means in three different cases.

- If v is an input vertex, then r_v is the Euclidean distance between v and the input vertex nearest v; see Figure 3.15(a).
- If v is a vertex inserted at the midpoint of an encroached subsegment, then r_v is the distance between v and the nearest encroaching mesh vertex; see Figure 3.15(b). If there is no encroaching vertex in the

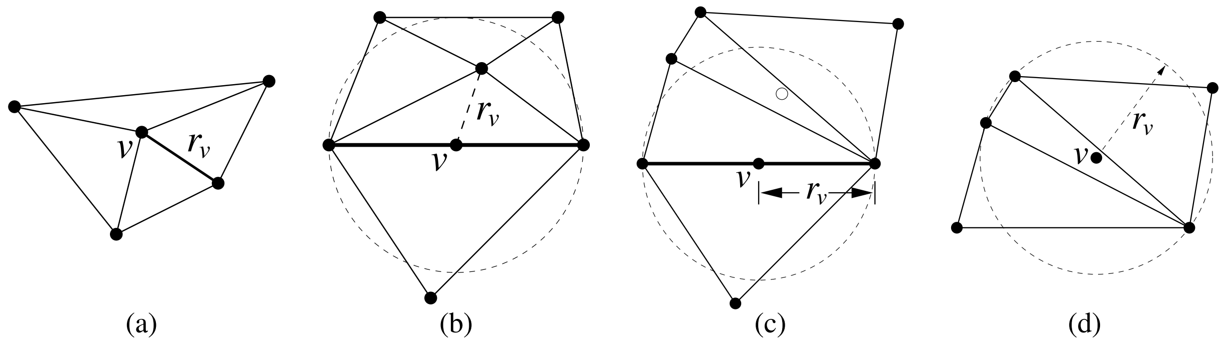


Figure 3.15: The insertion radius of a vertex v is the distance to the nearest vertex when v first appears in the mesh. (a) If v is an input vertex, r_v is the distance to the nearest other input vertex. (b) If v is the midpoint of a subsegment encroached upon by a vertex of the mesh, r_v is the distance to that vertex. (c) If v is the midpoint of a subsegment encroached upon only by a vertex that was rejected for insertion, r_v is the radius of the subsegment's diametral circle. (d) If v is the circumcenter of a skinny triangle, r_v is the radius of the circumcircle.

mesh (some triangle's circumcenter was considered for insertion but rejected as encroaching), then r_v is the radius of the diametral circle of the encroached subsegment, and hence the length of each of the two subsegments thus produced; see Figure 3.15(c).

- If v is a vertex inserted at the circumcenter of a skinny triangle, then r_v is the circumradius of the triangle; see Figure 3.15(d).

Each vertex v, including any vertex that is considered for insertion but not actually inserted because it encroaches upon a subsegment, has a *parent* vertex $p(v)$, unless v is an input vertex. Intuitively, for any non-input vertex v, $p(v)$ is the vertex that is "responsible" for the insertion of v. The parent $p(v)$ is defined as follows.

- If v is an input vertex, it has no parent.
- If v is a vertex inserted at the midpoint of an encroached subsegment, then $p(v)$ is the vertex that encroaches upon that subsegment. (Note that $p(v)$ might not be inserted into the mesh as a result.) If there are several such vertices, choose the one nearest v.
- If v is a vertex inserted (or rejected for insertion) at the circumcenter of a skinny triangle, then $p(v)$ is the most recently inserted endpoint of the shortest edge of that triangle. If both endpoints of the shortest edge are input vertices, choose one arbitrarily.

Each input vertex is the root of a tree of vertices. However, we are not interested in these trees as a whole; only in the ancestors of any given vertex, which form a sort of history of the events leading to the insertion of that vertex. Figure 3.16 illustrates the parents of all vertices inserted or considered for insertion during the sample execution of Ruppert's algorithm in Figure 3.12.

Working with these definitions, one can show why Ruppert's algorithm terminates. The key insight is that no descendant of a mesh vertex has an insertion radius smaller than the vertex's own insertion radius. Certainly, no edge will ever appear that is shorter than the smallest feature in the input PSLG. To prove these facts, consider the relationship between a vertex's insertion radius and the insertion radius of its parent.

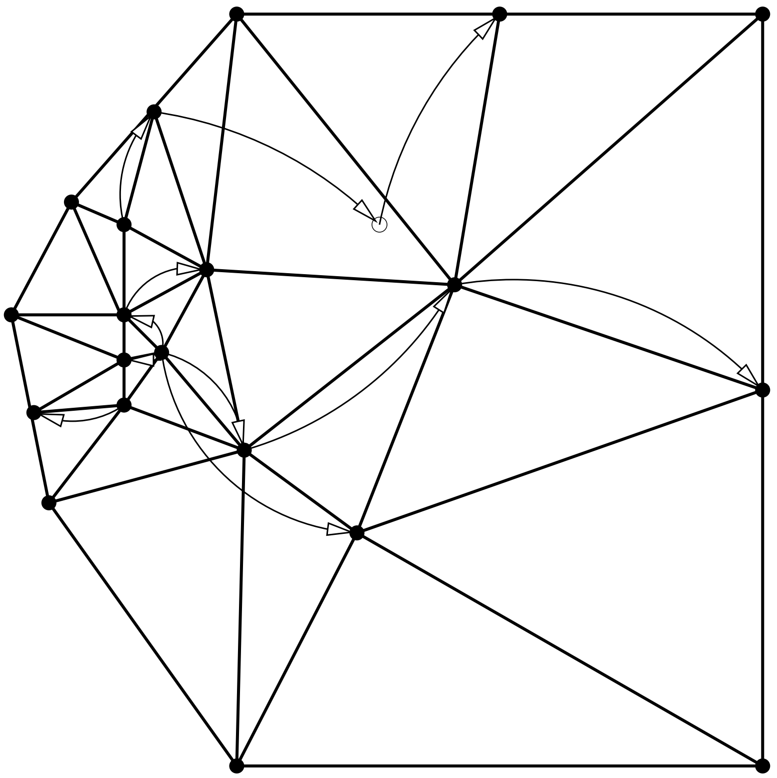


Figure 3.16: Trees of vertices for the example of Figure 3.12. Arrows are directed from parents to their children. Children include all inserted vertices and one rejected vertex.

Lemma 15 *Let v be a vertex of the mesh, and let $p = p(v)$ be its parent, if one exists. Then either $r_v \geq \mathrm{lfs}(v)$, or $r_v \geq Cr_p$, where*

- *$C = B$ if v is the circumcenter of a skinny triangle,*
- *$C = \frac{1}{\sqrt{2}}$ if v is the midpoint of an encroached subsegment and p is the circumcenter of a skinny triangle,*
- *$C = \frac{1}{2\cos\alpha}$ if v and p lie on incident segments separated by an angle of α (with p encroaching upon the subsegment whose midpoint is v), where $45^\circ \leq \alpha < 90^\circ$, and*
- *$C = \sin\alpha$ if v and p lie on incident segments separated by an angle of $\alpha \leq 45^\circ$.*

Proof: If v is an input vertex, there is another input vertex a distance of r_v from v, so $\mathrm{lfs}(v) \leq r_v$, and the theorem holds.

If v is inserted at the circumcenter of a skinny triangle, then its parent $p = p(v)$ is the most recently inserted endpoint of the shortest edge of the triangle; see Figure 3.17(a). Hence, the length of the shortest edge of the triangle is at least r_p. Because the triangle is skinny, its circumradius-to-shortest edge ratio is at least B, so its circumradius is $r_v \geq Br_p$.

If v is inserted at the midpoint of an encroached subsegment s, there are four cases to consider. The first two are all that is needed to prove termination of Ruppert's algorithm if no angles smaller than 90° are present in the input. The last two cases consider the effects of acute angles.

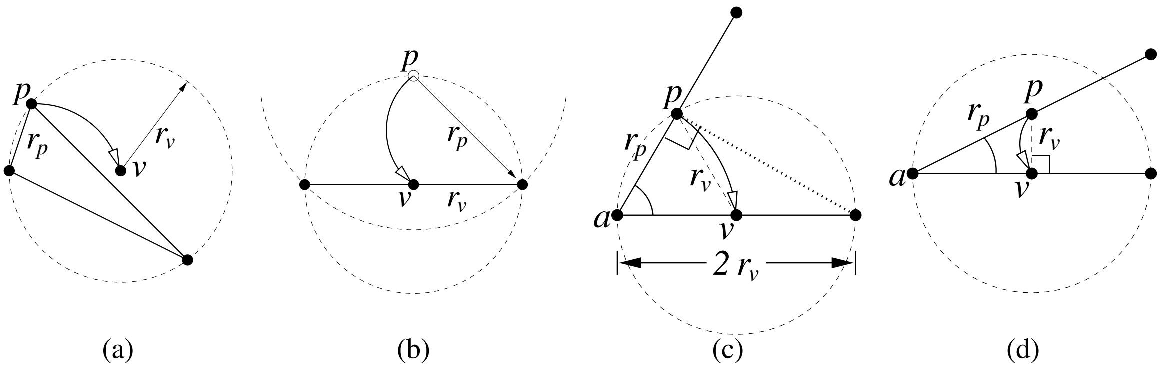


Figure 3.17: The relationship between the insertion radii of a child and its parent. (a) When a skinny triangle is split, the child's insertion radius is at least B times larger than that of its parent. (b) When a subsegment is encroached upon by the circumcenter of a skinny triangle, the child's insertion radius may be (arbitrarily close to) a factor of $\sqrt{2}$ smaller than the parent's, as this worst-case example shows. (c, d) When a subsegment is encroached upon by the midpoint of an incident subsegment, the relationship depends upon the angle α separating the two segments.

- If the parent p is an input vertex, or was inserted on a segment not incident to the segment containing s, then $\text{lfs}(v) \leq r_v$.

- If p is a circumcenter that was considered for insertion but rejected because it encroaches upon s, then p lies strictly inside the diametral circle of s. By the Delaunay property, the circumcircle centered at p contains no vertices, so its radius is limited by the nearest endpoint of s. Hence, $r_v > \frac{r_p}{\sqrt{2}}$; see Figure 3.17(b) for an example where the relation is nearly equality.

- If v and p lie on incident segments separated by an angle α where $45^\circ \leq \alpha < 90^\circ$, the vertex a (for "apex") where the two segments meet obviously cannot lie inside the diametral circle of s; see Figure 3.17(c). Because s is encroached upon by p, p lies inside its diametral circle. (If s is not present in the triangulation, p might lie on its diametral circle in a degenerate case.) To find the worst-case value of $\frac{r_v}{r_p}$, imagine that r_p and α are fixed; then $r_v = |vp|$ is minimized by making the subsegment s as short as possible, subject to the constraint that p cannot fall outside its diametral circle. The minimum is achieved when $|s| = 2r_v$; if s were shorter, its diametral circle would not contain p. Basic trigonometry shows that $|s| = 2r_v \geq \frac{r_p}{\cos \alpha}$.

- If v and p lie on incident segments separated by an angle α where $\alpha \leq 45^\circ$, then $\frac{r_v}{r_p}$ is minimized not when p lies on the diametral circle, but when v is the orthogonal projection of p onto s, as illustrated in Figure 3.17(d). Hence, $r_v \geq r_p \sin \alpha$. ■

The lemma just proven places limits on how quickly the insertion radius can decrease as one walks down a tree from an input vertex to a descendant. If the insertion radius cannot decrease at all, Ruppert's method is easily guaranteed to terminate. Figure 3.18 expresses this notion as a dataflow graph: labeled arrows indicate how a vertex can lead to the insertion of a new vertex whose insertion radius is some factor times that of its parent. If this graph contains no cycle whose product is less than one, termination is guaranteed. If some cycle has a product smaller than one, then a sequence of ever-smaller edges might be produced. The graph makes clear why the quality bound B must be at least $\sqrt{2}$, and why the minimum angle between input segments must be at least 60°. The following theorem formalizes this idea.

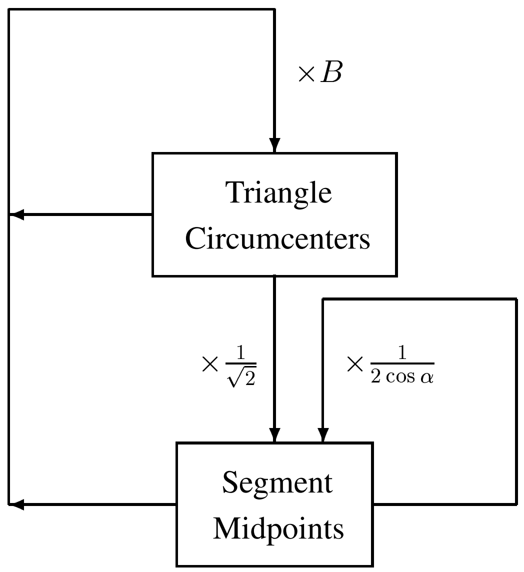


Figure 3.18: Dataflow diagram illustrating the worst-case relation between a vertex's insertion radius and the insertion radii of the children it begets. If no cycles have a product smaller than one, Ruppert's Delaunay refinement algorithm will terminate. Input vertices are omitted from the diagram because they cannot contribute to cycles.

Theorem 16 *Let* $\mathrm{lfs}_{\min}$ *be the shortest distance between two nonincident entities (vertices or segments) of the input PSLG*[1].

Suppose that any two incident segments are separated by an angle of at least $60°$*, and a triangle is considered to be skinny if its circumradius-to-shortest edge ratio is larger than* $B \geq \sqrt{2}$*. Ruppert's algorithm will terminate, with no triangulation edge shorter than* $\mathrm{lfs}_{\min}$.

Proof: Suppose for the sake of contradiction that the algorithm introduces an edge shorter than $\mathrm{lfs}_{\min}$ into the mesh. Let e be the first such edge introduced. Clearly, the endpoints of e cannot both be input vertices, nor can they lie on nonincident segments. Let v be the most recently inserted endpoint of e.

By assumption, no edge shorter than $\mathrm{lfs}_{\min}$ existed before v was inserted. Hence, for any ancestor a of v, $r_a \geq \mathrm{lfs}_{\min}$. Let $p = p(v)$ be the parent of v, and let $g = p(p)$ be the grandparent of v (if one exists). Consider the following cases.

- If v is the circumcenter of a skinny triangle, then by Lemma 15, $r_v \geq Br_p \geq \sqrt{2}r_p$.
- If v is the midpoint of an encroached subsegment and p is the circumcenter of a skinny triangle, then by Lemma 15, $r_v \geq \frac{1}{\sqrt{2}}r_p \geq \frac{B}{\sqrt{2}}r_g \geq r_g$. (Recall that p is not inserted into the mesh.)
- If v and p lie on incident segments, then by Lemma 15, $r_v \geq \frac{r_p}{2\cos\alpha}$. Because $\alpha \geq 60°$, $r_v \geq r_p$.

[1]Equivalently, $\mathrm{lfs}_{\min} = \min_u \mathrm{lfs}(u)$, where u is chosen from among the input vertices. The proof that both definitions are equivalent is omitted, but it relies on the recognition that if two points lying on nonincident segments are separated by a distance d, then at least one of the endpoints of one of the two segments is separated from the other segment by a distance of d or less. Note that $\mathrm{lfs}_{\min}$ is not a lower bound for $\mathrm{lfs}(\cdot)$ over the entire domain; for instance, a segment may have length $\mathrm{lfs}_{\min}$, in which case the local feature size at its midpoint is $\mathrm{lfs}_{\min}/2$.

In all three cases, $r_p \geq r_a$ for some ancestor a of p in the mesh. It follows that $r_p \geq \mathrm{lfs}_{\min}$, contradicting the assumption that e has length less than $\mathrm{lfs}_{\min}$. It also follows that no edge shorter than $\mathrm{lfs}_{\min}$ is ever introduced, and the algorithm must terminate. ■

Ruppert's algorithm terminates only when all triangles in the mesh have a circumradius-to-shortest edge ratio of B or better; hence, at termination, there are no angles smaller than $\arcsin \frac{1}{2B}$. If $B = \sqrt{2}$, the smallest value for which termination is guaranteed, no angle is smaller than 20.7°. Later, I will describe several modifications to the algorithm that improve this bound.

3.3.4 Proof of Good Grading and Size-Optimality

Theorem 16 guarantees that no edge of the final mesh is smaller than $\mathrm{lfs}_{\min}$. This guarantee may be satisfying for a user who desires a uniform mesh, but is not satisfying for a user who requires a spatially graded mesh. What follows is a proof that each edge of the output mesh has length proportional to the local feature sizes of its endpoints. Hence, a small local feature size in one part of a mesh does not unreasonably reduce the edge lengths at other, distant parts of the mesh. Triangle sizes vary quickly over short distances where such variation is desirable to help reduce the number of triangles in the mesh.

Lemma 15 was concerned with the relationship between the insertion radii of a child and its parent; the next lemma is concerned with the relationship between $\frac{\mathrm{lfs}(v)}{r_v}$ and $\frac{\mathrm{lfs}(p)}{r_p}$. For any vertex v, define $D_v = \frac{\mathrm{lfs}(v)}{r_v}$. Think of D_v as the one-dimensional density of vertices near v when v is inserted, weighted by the local feature size. Ideally, one would like this ratio to be as small as possible. Note that $D_v \leq 1$ for any input vertex, but D_v tends to be larger for a vertex inserted later.

Lemma 17 *Let v be a vertex with parent $p = p(v)$. Suppose that $r_v \geq C r_p$ (following Lemma 15). Then $D_v \leq 1 + \frac{D_p}{C}$.*

Proof: By Lemma 14, $\mathrm{lfs}(v) \leq \mathrm{lfs}(p) + |vp|$. The insertion radius r_v is usually $|vp|$ by definition, except in the case where p is rejected for insertion, in which case $r_v > |vp|$. Hence, we have

$$\begin{aligned} \mathrm{lfs}(v) &\leq \mathrm{lfs}(p) + r_v \\ &= D_p r_p + r_v \\ &\leq \frac{D_p}{C} r_v + r_v. \end{aligned}$$

The result follows by dividing both sides by r_v. ■

Lemma 17 generalizes to any dimension (assuming that some value for C can be proven), because it relies only upon Lemma 14. Ruppert's first main result follows.

Lemma 18 (Ruppert [82]) *Suppose the quality bound B is strictly larger than $\sqrt{2}$, and the smallest angle between two incident segments in the input PSLG is strictly greater than 60°. There exist fixed constants $D_T \geq 1$ and $D_S \geq 1$ such that, for any vertex v inserted (or considered for insertion and rejected) at the circumcenter of a skinny triangle, $D_v \leq D_T$, and for any vertex v inserted at the midpoint of an encroached subsegment, $D_v \leq D_S$. Hence, the insertion radius of every vertex is proportional to its local feature size.*

Proof: Consider any non-input vertex v with parent $p = p(v)$. If p is an input vertex, then $D_p = \frac{\text{lfs}(p)}{r_p} \leq 1$. Otherwise, assume for the sake of induction that the lemma is true for p, so that $D_p \leq D_T$ if p is a circumcenter, and $D_p \leq D_S$ if p is a midpoint. Hence, $D_p \leq \max\{D_T, D_S\}$.

First, suppose v is inserted or considered for insertion at the circumcenter of a skinny triangle. By Lemma 15, $r_v \geq Br_p$. Thus, by Lemma 17, $D_v \leq 1 + \frac{\max\{D_T, D_S\}}{B}$. It follows that one can prove that $D_v \leq D_T$ if D_T is chosen so that

$$D_T \geq 1 + \frac{\max\{D_T, D_S\}}{B}. \tag{3.1}$$

Second, suppose v is inserted at the midpoint of a subsegment s. If its parent p is an input vertex or lies on a segment not incident to s, then $\text{lfs}(v) \leq r_v$, and the theorem holds. If p is the circumcenter of a skinny triangle (rejected for insertion because it encroaches upon s), $r_v \geq \frac{r_p}{\sqrt{2}}$ by Lemma 15, so by Lemma 17, $D_v \leq 1 + \sqrt{2}D_T$.

Alternatively, if p, like v, is a subsegment midpoint, and p and v lie on incident segments, then $r_v \geq \frac{r_p}{2\cos\alpha}$ by Lemma 15, and thus by Lemma 17, $D_v \leq 1 + 2D_S \cos\alpha$. It follows that one can prove that $D_v \leq D_S$ if D_S is chosen so that

$$D_S \geq 1 + \sqrt{2}D_T, \quad \text{and} \tag{3.2}$$

$$D_S \geq 1 + 2D_S \cos\alpha. \tag{3.3}$$

If the quality bound B is strictly larger than $\sqrt{2}$, conditions 3.1 and 3.2 are simultaneously satisfied by choosing

$$D_T = \frac{B+1}{B-\sqrt{2}}, \qquad D_S = \frac{(1+\sqrt{2})B}{B-\sqrt{2}}.$$

If the smallest input angle $\alpha_{\min}$ is strictly greater than $60°$, conditions 3.3 and 3.1 are satisfied by choosing

$$D_S = \frac{1}{1 - 2\cos\alpha_{\min}}, \qquad D_T = 1 + \frac{D_S}{B}.$$

One of these choices will dominate, depending on the values of B and $\alpha_{\min}$. However, if $B > \sqrt{2}$ and $\alpha_{\min} > 60°$, there are values of D_T and D_S that satisfy the lemma. ∎

Note that as B approaches $\sqrt{2}$ or α approaches $60°$, D_T and D_S approach infinity. In practice, the algorithm is better behaved than the theoretical bound suggests; the vertex spacing approaches zero only after B drops below one.

Theorem 19 (Ruppert [82]) *For any vertex v of the output mesh, the distance to its nearest neighbor w is at least $\frac{\text{lfs}(v)}{D_S+1}$.*

Proof: Inequality 3.2 indicates that $D_S > D_T$, so Lemma 18 shows that $\frac{\text{lfs}(v)}{r_v} \leq D_S$ for any vertex v. If v was added after w, then the distance between the two vertices is at least $r_v \geq \frac{\text{lfs}(v)}{D_S}$, and the theorem holds. If w was added after v, apply the lemma to w, yielding

$$|vw| \geq r_w \geq \frac{\text{lfs}(w)}{D_S}.$$

By Lemma 14, $\text{lfs}(w) + |vw| \geq \text{lfs}(v)$, so

$$|vw| \geq \frac{\text{lfs}(v) - |vw|}{D_S}.$$

It follows that $|vw| \geq \frac{\text{lfs}(v)}{D_S+1}$. ■

To give a specific example, consider triangulating a PSLG (having no acute input angles) so that no angle of the output mesh is smaller than 15°; hence $B \doteq 1.93$. For this choice of B, $D_T \doteq 5.66$ and $D_S \doteq 9.01$. Hence, the spacing of vertices is at worst about ten times smaller than the local feature size. Away from boundaries, the spacing of vertices is at worst 6.66 [58] times smaller than the local feature size.

Figure 3.19 shows the algorithm's performance for a variety of angle bounds. Ruppert's algorithm typically terminates for angle bounds much higher than the theoretically guaranteed 20.7, and typically exhibits much better vertex spacing than the provable worst-case bounds imply.

Ruppert [82] uses Theorem 19 to prove the size-optimality of the meshes his algorithm generates, and his result has been improved by Scott Mitchell. Mitchell's theorem is stated below, but the proof, which is rather involved, is omitted. The *cardinality* of a triangulation is the number of triangles in the triangulation.

Theorem 20 (Mitchell [68]) *Let* $\text{lfs}_T(p)$ *be the local feature size at* p *with respect to a triangulation* T *(treating* T *as a PSLG), whereas* $\text{lfs}(p)$ *remains the local feature size at* p *with respect to the input PSLG. Suppose a triangulation* T *with smallest angle* θ *has the property that there is some constant* $k_1 \geq 1$ *such that for every point* p, $k_1 \text{lfs}_T(p) \geq \text{lfs}(p)$. *Then the cardinality of* T *is less than* k_2 *times the cardinality of any other triangulation of the input PSLG with smallest angle* θ, *where* $k_2 = \mathcal{O}(k_1^2/\theta)$. ■

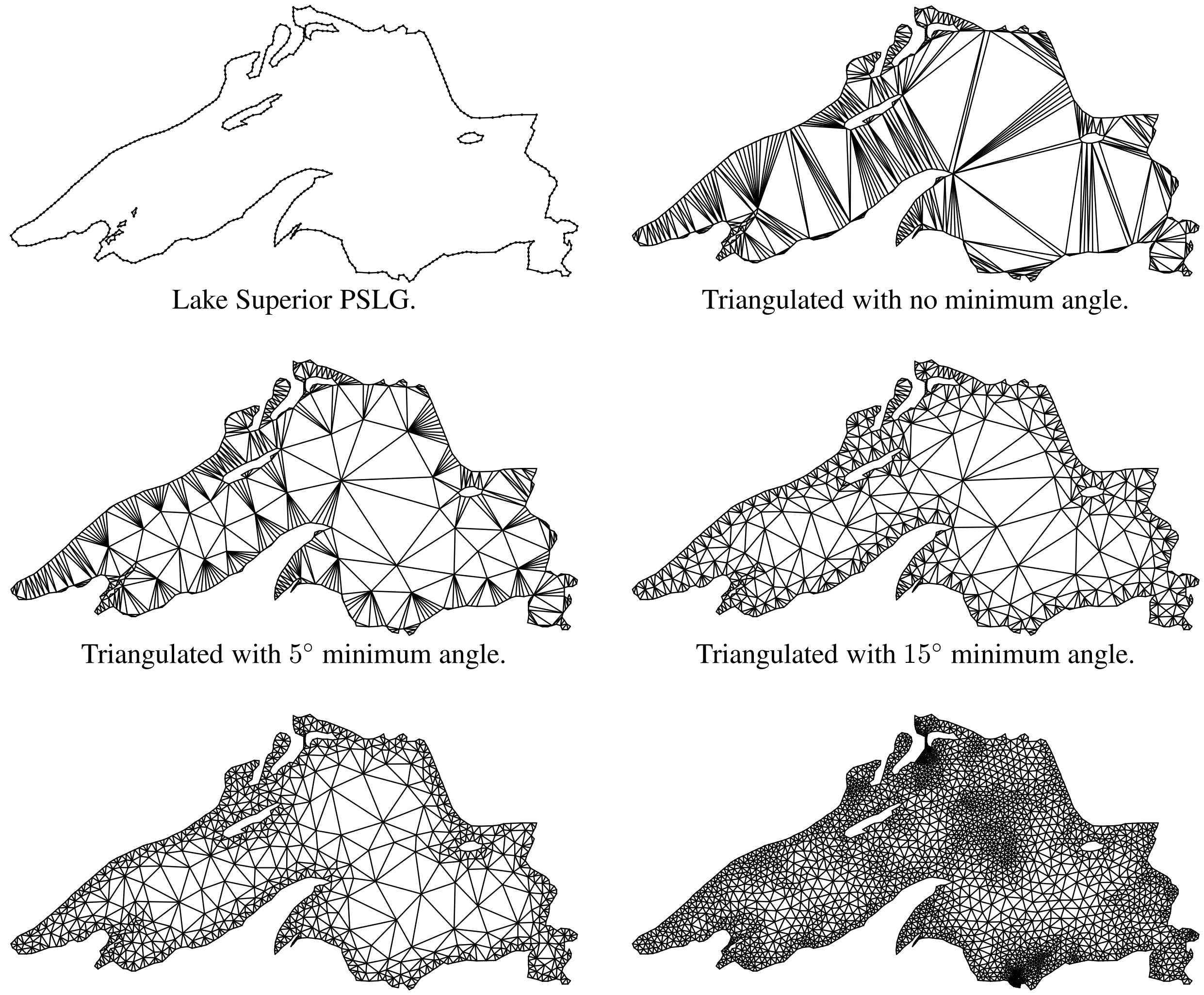
Theorem 19 can be used to show that the precondition of Theorem 20 is satisfied by meshes generated by Ruppert's algorithm. Hence, a mesh generated by Ruppert's algorithm has cardinality within a constant factor of the best possible mesh satisfying the angle bound.

3.4 Chew's Second Delaunay Refinement Algorithm and Diametral Lenses

This section presents two algorithms that offer an improved guarantee of good grading and that perform slightly better in practice: Chew's second Delaunay refinement algorithm [21], and a variant of Ruppert's algorithm that replaces diametral circles with narrower entities called diametral lenses. I will show that the two algorithms are equivalent, and exhibit good grading and size-optimality for angles bounds of up to 26.5°. Chew shows that his algorithm terminates for an angle bound of up to 30°, albeit with no guarantee of good grading. The means by which he obtains this bound is discussed in Section 3.5.2. Note that Chew's paper also discusses triangular meshing of curved surfaces in three dimensions, but I consider the algorithm only in its planar context.

3.4.1 Description of the Algorithm

Chew's algorithm begins with the constrained Delaunay triangulation of a segment-bounded PSLG, and uses Delaunay refinement with locked subsegments and a quality bound of $B = 1$, but there is no idea of encroached diametral circles. However, it may arise that a skinny triangle t cannot be split because t and its



Lake Superior PSLG. Triangulated with no minimum angle.

Triangulated with 5° minimum angle. Triangulated with 15° minimum angle.

Triangulated with 25° minimum angle. Triangulated with 34.2° minimum angle.

Figure 3.19: Meshes generated with Ruppert's algorithm for several different quality bounds. The algorithm does not terminate for angle bounds of 34.3° or higher on this PSLG.

circumcenter c lie on opposite sides of a subsegment s (possibly with c outside the triangulation). Because s is locked, inserting a vertex at c will not remove t from the mesh. Instead, c is rejected for insertion, and all free vertices (but not input vertices or vertices that lie on segments) that lie in the interior of the diametral circle of s and are visible from the midpoint of s are deleted. Then, a new vertex is inserted at the midpoint of s. The Delaunay property is maintained throughout all deletions and insertions, except that locked subsegments are not flipped. Figure 3.20 illustrates a subsegment split in Chew's algorithm.

If several subsegments lie between t and c, only the subsegment visible from the interior of t is split.

I claim that Chew's algorithm is roughly equivalent (enough for the purposes of analysis) to a variant of Ruppert's algorithm in which diametral circles are replaced with *diametral lenses*, illustrated in Figure 3.21(a). The diametral lens of a subsegment s is the intersection of two disks whose centers lie on each other's boundaries, and whose boundaries intersect at the endpoints of s. It follows that the defining disks have radius $2|s|/\sqrt{3}$, and their centers lie on the bisector of s at a distance of $|s|/\sqrt{3}$ from s. The subsegment s is split if there is a vertex, or an attempt to insert a vertex, in or on the boundary of its diametral lens, unless

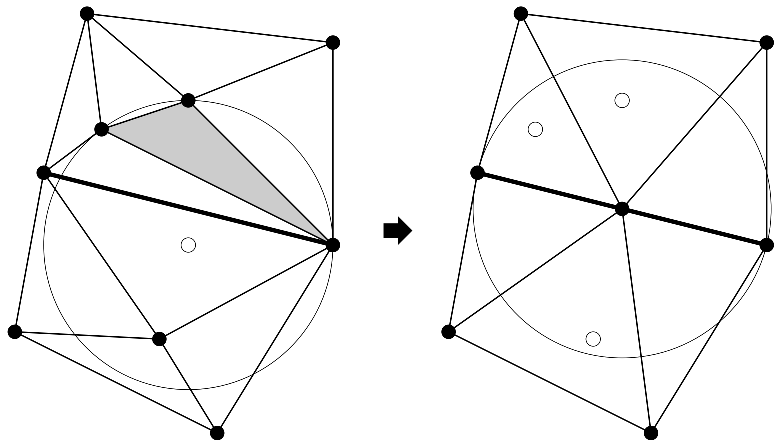
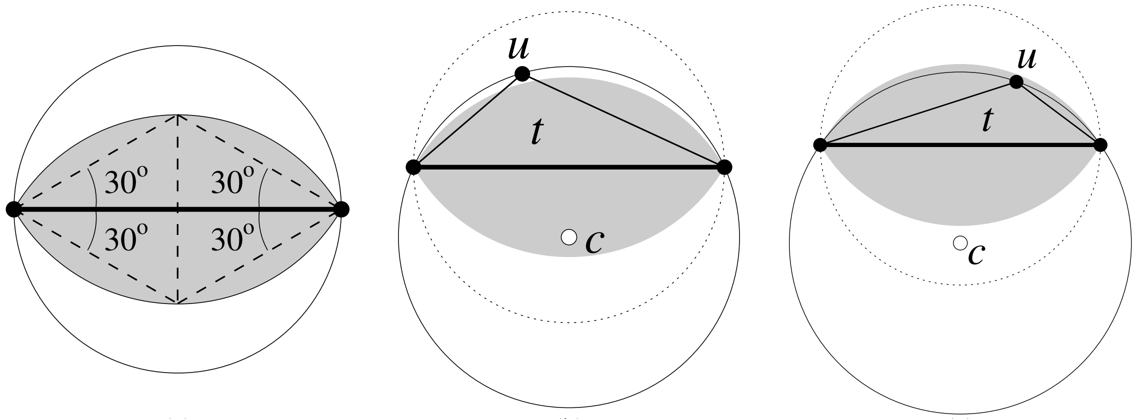


Figure 3.20: At left, a skinny triangle and its circumcenter lie on opposite sides of a subsegment. At right, all vertices in the subsegment's diametral circle have been deleted, and a new vertex has been inserted at the subsegment's midpoint.



(a) (b) (c)

Figure 3.21: (a) A hybrid Chew/Ruppert algorithm uses diametral lenses. Only vertices in the shaded region encroach upon this subsegment. Note the equilateral triangles; a circumcenter at the lowest point of the lens arises from any triangle whose vertices all lie on the upper lens's boundary. (b, c) If a skinny triangle and its circumcenter lie on opposite sides of a subsegment, then either the circumcenter or a vertex of the triangle lies within the subsegment's diametral lens.

another subsegment obstructs the line of sight between the encroaching vertex and the midpoint of s. As in Chew's algorithm, subsegments are locked, and all visible free vertices are deleted from a subsegment's diametral circle before the subsegment is bisected.

Why are these algorithms equivalent? Let t be a skinny triangle whose circumcenter c lies on the opposite side of a subsegment s. Lemma 13 shows that some vertex u of the skinny triangle lies inside the diametral circle of s, on the circumcircle of t. There are two possibilities: either c encroaches upon s (Figure 3.21(b)), or u encroaches upon s (Figure 3.21(c)). Hence, the modified Ruppert algorithm will split any subsegment Chew's algorithm would split.

Conversely, if a vertex lies in or on the boundary of the diametral lens of s, then the triangle containing s (on the same side of s as the encroaching vertex) is skinny, and its circumcenter is on the other side of s. Hence, Chew's algorithm will split any subsegment the modified Ruppert algorithm would split.

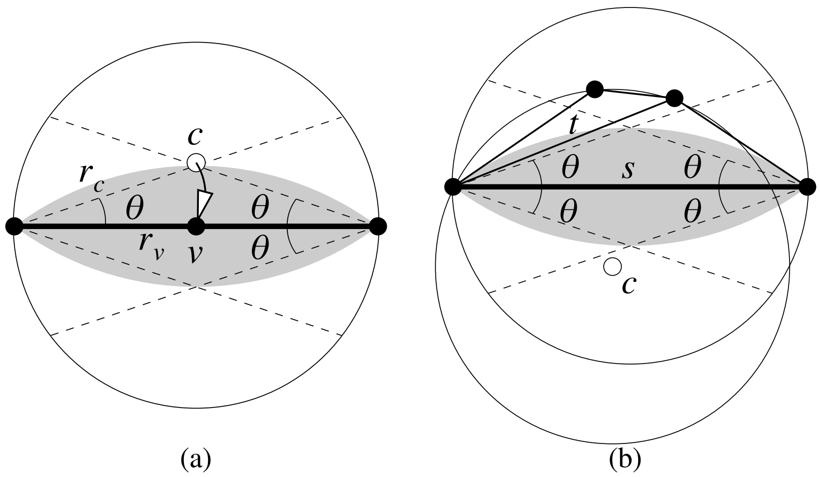


Figure 3.22: (a) Example where a subsegment is encroached upon by a vertex in its diametral lens. In the worst case, $r_v = r_c \cos\theta$. (b) Example where a subsegment is encroached because a skinny triangle and its circumcenter lie on opposite sides of the subsegment.

(Technically, this is not quite true; Chew's algorithm might decline to split a subsegment for which a vertex lies precisely at the intersection of the subsegment's bisector and the boundary of its diametral lens, thereby forming a triangle with two angles of precisely $30°$. This difference does not affect the analysis of the algorithms.)

The modified Ruppert algorithm has a small speed advantage because it avoids inserting many of the vertices that would be deleted later in Chew's algorithm.

Compared to diametral circles, diametral lenses have the disadvantage that the final mesh is not guaranteed to be Delaunay, but they have two advantages. First, many subsegment splits are avoided that would otherwise have occurred. Hence, the final mesh may have fewer triangles. Second, when a subsegment split does occur, the parent vertex $p = p(v)$ cannot be too near a pole of the diametral circle, and the ratio between r_v and r_p is better bounded. Whereas Lemma 15 could only guarantee that $r_v \geq \frac{r_p}{\sqrt{2}}$, diametral lenses make a better bound possible.

3.4.2 Proof of Good Grading and Size-Optimality

I generalize lenses to allow the angle θ that defines the shape of a lens, illustrated in Figure 3.22(a), to assume values other than $30°$. The lens angle θ is independent of the angle bound $\arcsin\frac{1}{2B}$; for instance, Ruppert's unmodified algorithm has $\theta = 45°$. There is little sense, though, in making θ smaller than the angle bound, because reducing θ below $\arcsin\frac{1}{2B}$ will only allow the insertion of more vertices that will be deleted. If $\theta < 30°$, there is the problem that a skinny triangle and its circumcircle might lie on opposite sides of a subsegment without any vertex falling in its lens, as illustrated in Figure 3.22(b). In this case, one must use Chew's formulation of the algorithm, so that the subsegment is properly considered encroached; but diametral lenses may be employed as well, because they save time in the cases where a vertex does fall inside.

It is this mixed formulation I envision for the proof that follows. I show that Chew's algorithm with $\theta = \arcsin\frac{1}{2B}$ exhibits guaranteed grading for $B > \frac{\sqrt{5}}{2} \doteq 1.12$, giving an angle bound of up to $\arcsin\frac{1}{\sqrt{5}} \doteq 26.56°$. (Although I shall not give details, if $\theta = 30°$, one may prove guaranteed grading only for $B > \frac{2}{\sqrt{3}} \doteq$

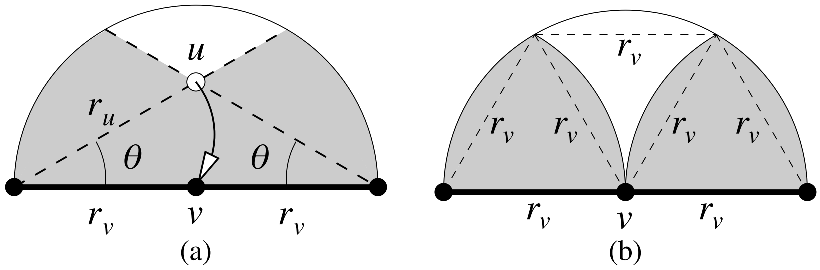


Figure 3.23: (a) Figure for the case where exactly one vertex is in the semicircle. (b) Figure for the case where more than one vertex is in the semicircle.

1.15, giving an angle bound of up to $\arcsin \frac{\sqrt{3}}{4} \doteq 25.65°$.)

The proof requires an adjustment of the definition of "insertion radius" to accommodate the necessary use of constrained Delaunay triangulations. The insertion radius r_c of the circumcenter c of a skinny triangle t is now defined to be the radius of t's circumcircle, whether or not the circumcircle contains any vertices. Recall that by the previous definition, r_c was the length of the longest edge that would adjoin c if c were inserted. The only circumstance in which these definitions differ is when t and c lie on opposite sides of a subsegment, and other vertices lie in t's circumcircle. These vertices do no harm because c is not actually inserted; c only acts as a placeholder relating the insertion radii of its parent p and its child. The change in definition is necessary, because otherwise the inequality $r_c \geq Br_p$, proven in Lemma 15, is invalidated.

Lemma 21 *Let θ be the angle that defines the shape of each diametral lens, as illustrated in Figure 3.22(a), where θ satisfies $\arcsin \frac{1}{2B} \leq \theta \leq 45°$. Let s be a subsegment encroached upon by the circumcenter c of a skinny triangle t. Suppose that all vertices in the diametral circle of s are deleted (except those not visible from the midpoint of s), and a vertex v is inserted at the midpoint of s. Then there is some vertex p, rejected for insertion in or deleted from the diametral circle of s, such that $r_v \geq r_p \cos\theta$.*

Proof: Because all vertices visible from v are deleted from inside the diametral circle of s, r_v is equal to the radius of that diametral circle. (Vertices not visible from v cannot affect v's insertion radius, because an edge cannot connect them to v.)

If the circumcenter c lies in (or on the boundary of) the diametral lens of s, then the maximum possible value of r_c occurs with c simultaneously on the boundary of the lens and on the bisector of s, as illustrated in Figure 3.22(a). The circumcircle centered at c can contain no vertices on or above s, so its radius is limited by the nearest endpoint of s. Hence, $r_v \geq r_c \cos\theta$; define the parent of v to be c.

The case just described is sufficient to prove the lemma if $\theta \geq 30°$. However, if $\theta < 30°$, then c might not fall in the lens; rather, s might be encroached because t and c lie on opposite sides of s, as illustrated in Figure 3.22(b). Assume without loss of generality that t lies above s, with c below. By Lemma 13, at least one vertex of t lies strictly inside the upper half of the diametral circle of s. There are two cases, depending on the number of vertices in the interior of this semicircle.

If the upper semicircle contains only one vertex u, then t is defined by u and s. Because t is skinny and $\theta \geq \arcsin \frac{1}{2B}$, u must lie in the shaded region of Figure 3.23(a), and therefore $r_v \geq r_u \cos\theta$. Define the parent of v to be u.

If the upper semicircle contains more than one vertex, consider Figure 3.23(b), in which the shaded region represents points within a distance of r_v from a subsegment endpoint. If some vertex u lies in the shaded region, then $r_u \leq r_v$; define the parent of v to be u. If no vertex lies in the shaded region, then there are at least two vertices in the white region of the upper semicircle. Let u be the most recently inserted of these vertices. The vertex u is no further than r_v from any other vertex in the white region, so $r_u \leq r_v$; define the parent of v to be u. ■

Lemma 21 extends the definition of parent to accommodate a new type of encroachment. If a subsegment is encroached because a triangle and its circumcenter lie on opposite sides of the subsegment, although no vertex encroaches upon the subsegment's diametral lens, then the parent of the newly inserted midpoint is defined to be a vertex in the upper half of the subsegment's diametral circle. It is important that the parent is inside the diametral circle of s, because Lemma 17, which bounds the density of vertices near a vertex, relies on the assumption that $|pv| \leq r_v$.

What if the diametral circle of s contains a vertex u that is visible from v but not deleted? Either u is an input vertex, or u lies on a segment. The analysis of Lemma 21 does not apply, because r_v is (at most) $|uv|$, which is smaller than the diametral radius of s. In this case, u and v lie on nonincident input features, so $r_v \geq \mathrm{lfs}(v)$; this case is already covered by the analysis of Lemma 15. Choose the input vertex or segment vertex closest to v to be the parent of v.

Do the differences between Chew's algorithm and Ruppert's original algorithm invalidate any of the assumptions used in Section 3.3 to prove termination? None of the bounds proven in Theorem 16 and Lemma 18 is invalidated. When a vertex is deleted from a Delaunay triangulation, no vertex finds itself adjoining a shorter edge than the shortest edge it adjoined before the deletion. (This fact follows because a constrained Delaunay triangulation connects every vertex to its nearest visible neighbor.) Hence, each vertex's insertion radius still serves as a lower bound on the lengths of all edges that connect the vertex to vertices older than itself, and therefore Theorem 16 and Lemma 18 are still true.

The only part of the termination proof that does not apply to Chew's second algorithm is the assumption that every operation inserts a new vertex. If vertices can be deleted, are we sure that the algorithm terminates? Observe that vertex deletions only occur when a subsegment is split, and vertices are never deleted from subsegments. Theorem 16 sets a lower bound on the length of each subsegment, so only a finite number of subsegment splits can occur. After the last subsegment split, no more vertex deletions occur, and termination may be proven in the usual manner.

The consequence of the bound proven by Lemma 21 is illustrated in the dataflow graph of Figure 3.24. Recall that termination is guaranteed if no cycle has a product less than one. Hence, a condition of termination is that $B\cos\theta \geq 1$. The best bound that satisfies this criterion, as well as the requirement that $\theta \geq \arcsin\frac{1}{2B}$, is $B = \frac{\sqrt{5}}{2} \doteq 1.12$, which corresponds to an angle bound of $\arcsin\frac{1}{\sqrt{5}} \doteq 26.5^\circ$.

Theorem 22 *Suppose the quality bound B is strictly larger than $\frac{\sqrt{5}}{2}$, and the smallest angle between two incident segments in the input PSLG is strictly greater than 60°. There exist fixed constants $D_T \geq 1$ and $D_S \geq 1$ such that, for any vertex v inserted (or considered for insertion and rejected) at the circumcenter of a skinny triangle, $D_v \leq D_T$, and for any vertex v inserted at the midpoint of an encroached subsegment, $D_v \leq D_S$.*

Proof: Essentially the same as the proof of Lemma 18, except that Lemma 21 makes it possible to replace Condition 3.2 with

$$\begin{aligned} D_S &\geq 1 + \frac{D_T}{\cos\theta} \\ &\geq 1 + \frac{2BD_T}{\sqrt{4B^2-1}} \end{aligned} \tag{3.4}$$

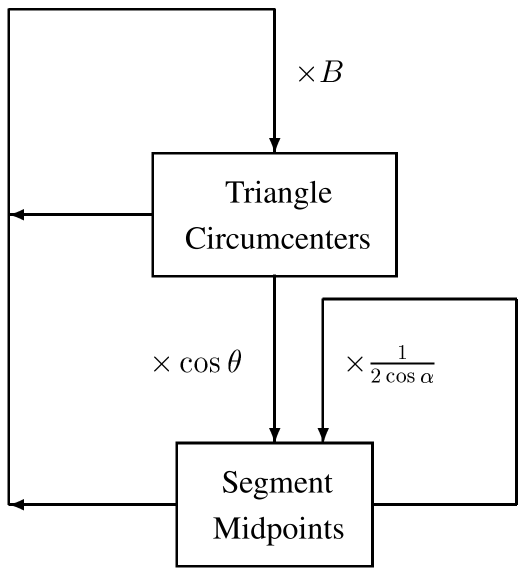


Figure 3.24: Dataflow diagram for Chew's algorithm (with a variable angle condition).

If the quality bound B is strictly larger than $\frac{\sqrt{5}}{2}$, Conditions 3.1 and 3.4 are simultaneously satisfied by choosing

$$D_T = \frac{\left(1 + \frac{1}{B}\right)\sqrt{4B^2 - 1}}{\sqrt{4B^2 - 1} - 2}, \qquad D_S = \frac{\sqrt{4B^2 - 1} + 2B}{\sqrt{4B^2 - 1} - 2}.$$

D_T and D_S must also satisfy the conditions specified in Lemma 18 related to the angles between segments. If $B > \frac{\sqrt{5}}{2}$ and $\alpha_{\min} > 60^\circ$, there are values of D_T and D_S that satisfy the theorem. ∎

Theorem 19, which bounds the edge lengths of the mesh, generalizes directly to cover Chew's algorithm, so we may compare this analysis with that of Ruppert's algorithm. As in Section 3.3, consider triangulating a PSLG (having no acute input angles) so that no angle of the output mesh is smaller than 15°; hence $B \doteq 1.93$. For this choice of B, $D_T \doteq 3.27$ and $D_S \doteq 4.39$, compared to the corresponding values of 5.66 and 9.01 for Ruppert's algorithm. Hence, the spacing of vertices is at worst a little more than five times the local feature size, and a little more than four times the local feature size away from boundaries. Because the worst-case number of triangles is proportional to the square of D_S, Chew's algorithm is size-optimal with a constant of optimality almost four times better than Ruppert's algorithm. Of course, worst-case behavior is never seen is practice, and the observed difference between the two algorithms is less dramatic.

3.5 Improvements

Here, I describe several modifications to the Delaunay refinement algorithms that improve the quality of the elements of a mesh. The first modification improves the quality of triangles away from the boundary of the mesh; the second, which generalizes an idea of Chew, improves the quality of triangles everywhere.

3.5.1 Improving the Quality Bound in the Interior of the Mesh

The first improvement arises easily from the discussion in Section 3.3.3. So long as no cycle having a product less than one appears in the insertion radius dataflow graph (Figure 3.18 or 3.24), termination is assured. The barrier to reducing the quality bound B below $\frac{\sqrt{5}}{2}$ is the fact that, when an encroached segment is split, the child's insertion radius may be a factor of $\frac{\sqrt{5}}{2}$ smaller than its parent's. However, not every segment bisection is a worst-case example, and it is easy to explicitly measure the insertion radii of a parent and its potential progeny before deciding to take action. One can take advantage of these facts with any one of the following strategies.

- Use a quality bound of $B = 1$ for triangles that are not in contact with segment interiors, and a quality bound of $B = \sqrt{2}$ (for diametral circles) or $B = \frac{\sqrt{5}}{2}$ (for diametral lenses) for any triangle having a vertex that lies in the interior of a segment.
- Attempt to insert the circumcenter of any triangle whose circumradius-to-shortest edge ratio is larger than one. If any subsegments would be encroached, the circumcenter is rejected as usual, but the encroached subsegments are split only if the triangle's circumradius-to-shortest edge ratio is greater than $\sqrt{2}$ (for diametral circles) or $\frac{\sqrt{5}}{2}$ (for diametral lenses).
- Attempt to insert the circumcenter of any triangle whose circumradius-to-shortest edge ratio is larger than one. If any subsegments would be encroached, the circumcenter is rejected as usual, and each encroached subsegment is checked to determine the insertion radius of the new vertex that might be inserted at its midpoint. The only midpoints inserted are those whose insertion radii are at least as large as the length of the shortest edge of the skinny triangle.

The first strategy is easily understood from Figure 3.25. Because segment vertices may have smaller insertion radii than free vertices, segment vertices are only allowed to father free vertices whose insertion radii are larger than their own by a factor of $\sqrt{2}$ or $\frac{\sqrt{5}}{2}$, as appropriate. Hence, no diminishing cycles are possible.

The other two strategies work for an even more straightforward reason: all vertices (except rejected vertices) are expressly forbidden from creating descendants having insertion radii smaller than their own. The third strategy is more aggressive than the second, as it always chooses to insert a vertex if the second strategy would do so.

The first strategy differs from the other two in its tendency to space segment vertices more closely than free vertices. The other two strategies tend to space segment vertices and free vertices equally, at the cost of spacing the latter more densely than necessary. The first strategy interrupts the propagation of reduced insertion radii from segment vertices to the free vertices, whereas the other two interrupt the process by which free vertices create segment vertices with smaller insertion radii. The effect of the first strategy is easily stated: upon termination, all angles are better than 20.7° or 26.5°, and all triangles whose vertices do not lie in segment interiors have angles of 30° or better. For the other two strategies, the delineation between 26.5° triangles and 30° triangles is not so clear, although the former only occur near boundaries.

None of these strategies compromises good grading or size-optimality, although the bounds may be weaker. Assume that a quality bound B is applied to all triangles, and a stronger quality bound $B_I > 1$ applies in the interior of the mesh. Then Equation 3.1 is accompanied by the equation

$$D_T \geq 1 + \frac{D_T}{B_I},$$

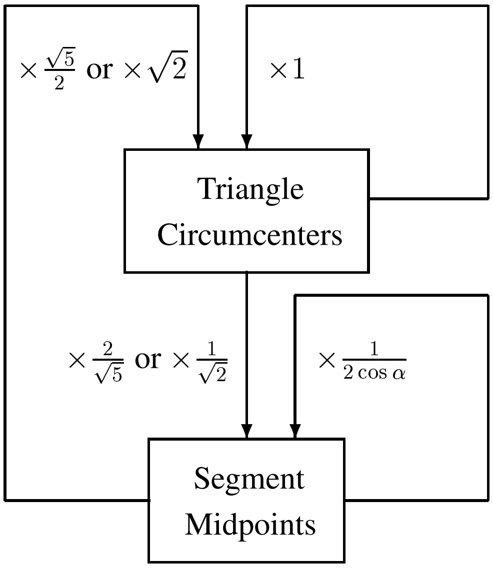


Figure 3.25: This dataflow diagram demonstrates how a simple modification to Ruppert's or Chew's algorithm can improve the quality of elements away from mesh boundaries.

which holds true when

$$D_T \geq \frac{B_I}{B_I - 1}.$$

If this bound is stricter than the bounds already given in the proof of Lemma 18 (for diametral circles) or 22 (for diametral lenses), then D_S must also be recalculated using Equation 3.2. Furthermore, if the second or third strategy is used, then D_T increases increased to match D_S (Condition 3.2 no longer holds.) However, if $B > \sqrt{2}$ (for Ruppert's algorithm) or $B > \frac{\sqrt{5}}{2}$ (for Chew's), $B_I > 1$, and $\alpha_{\min} > 60°$, there are values of D_T and D_S that satisfy the lemma.

3.5.2 Range-Restricted Segment Splitting

In this section, I suggest another algorithmic change that generalizes an idea of Chew [21]. Both Ruppert's and Chew's algorithms may be modified to make it possible to apply a quality bound of $B = 1$ to all triangles of the mesh, although there is no accompanying guarantee of good grading. I shall consider Ruppert's algorithm first, then Chew's.

Observe that the only mechanism by which a vertex can have a child with a smaller insertion radius than its own is by encroaching upon a subsegment. Furthermore, an encroaching circumcenter v cannot have a child whose insertion radius is smaller than $r_v/\sqrt{2}$, and hence it cannot cause the splitting of a segment whose length is less than $\sqrt{2}r_v$. On the other hand, if v causes the bisection of a segment whose length is $2r_v$ or greater, the child that results will have an insertion radius of at least r_v. I conclude that a vertex v can only produce a child whose insertion radius is less than r_v if a segment is present whose length is between $\sqrt{2}r_v$ and $2r_v$. If no such segment exists, the cycle of diminishing edge lengths is broken.

Thus the motivation for *range-restricted segment splitting*. Whenever possible, the length of each subsegment is restricted to the range $c2^x$, where $c \in (1, \sqrt{2}]$ and x is an integer. This restriction is illustrated in Figure 3.26, wherein darkened boxes on the number line represent legal subsegment lengths. The positive integers are partitioned into contiguous sets, each having a geometric width of $\sqrt{2}$, and alternate sets

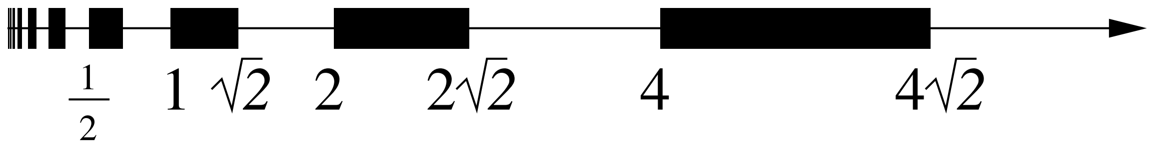


Figure 3.26: Legal subsegment lengths are of the form $c2^x$, where $c \in (1, \sqrt{2}]$ and x is an integer.

are made illegal. With this choice of partition, the bisection of any legal subsegment will produce legal subsegments.

Of course, input segments might not have legal lengths. However, when an encroached segment of illegal length is split, rather than place a new vertex at its midpoint, one may place the vertex so that the resulting subsegments fall within the legal range.

How does this restriction help? A vertex whose insertion radius is greater than 2^x for some integer x cannot have a descendant whose insertion radius is 2^x or smaller unless a subsegment having illegal length is split. However, each illegal subsegment can be split only once, yielding subsegments of legal length; hence, the fuel for diminishing edge lengths is in limited supply.

An illegal segment of the form $c2^x$, where $c \in (\sqrt{2}, 2]$ and x is an integer, is split into subsegments of lengths $c_1 2^x$ and $c_2 2^x$ as follows.

- If $c \in (\sqrt{2}, \frac{3}{2}]$, then $c_1 = \frac{\sqrt{2}}{4}$ and $c_2 = c - c_1$.
- If $c \in (\frac{3}{2}, \frac{2+\sqrt{2}}{2}]$, then $c_1 = 1 + \epsilon$ and $c_2 = c - c_1$. (Here, ϵ is an infinitesimal value used because 1 is technically not in the legal range. In practice, $\epsilon = 0$ is recommended.)
- If $c \in (\frac{2+\sqrt{2}}{2}, 2]$, then $c_1 = \frac{\sqrt{2}}{2}$ and $c_2 = c - c_1$.

The most unbalanced split occurs if $c = \frac{3}{2}$. Then, the ratio between c_1 and c is $\frac{\sqrt{2}}{6} \doteq 0.2357$.

I shall show formally that Delaunay refinement with range-restricted segment splitting terminates for any quality bound $B \geq 1$. Define the *insertion radius floor* r'_v of a vertex v to be the largest power of two that is strictly less than the vertex's insertion radius r_v.

Lemma 23 *Let* $\mathrm{lfs}_{\min}$ *be the shortest distance between two nonincident entities (vertices or segments) of the input PSLG. Suppose that a triangle is considered to be skinny if its circumradius-to-shortest edge ratio is larger than* $B \geq 1$. *Suppose also that the input PSLG has no angles smaller than* $60°$. *Let* v *be a vertex of the mesh, and let* $p = p(v)$ *be its parent, if one exists. Then either* $r'_v \geq \mathrm{lfs}_{\min}/6$*, or* $r'_v \geq r'_p$.

Proof: If v is an input vertex, then $\mathrm{lfs}_{\min} \leq \mathrm{lfs}(v) \leq r_v \leq 2r'_v$, and the theorem holds.

If v is inserted at the circumcenter of a skinny triangle, then by Lemma 15, $r_v \geq Br_p$. Because $B \geq 1$, it follows that $r'_v \geq r'_p$.

If v is inserted at the midpoint of an encroached subsegment s, there are three cases to consider.

- If the parent p is an input vertex, or was inserted on a segment not incident to the segment containing s, then $\mathrm{lfs}_{\min} \leq \mathrm{lfs}(v) \leq r_v \leq 2r'_v$.

- If v and p lie on incident segments separated by an angle α where $60^\circ \leq \alpha < 90^\circ$, then by Lemma 15, $r_v \geq \frac{r_p}{2\cos\alpha} \geq r_p$. Therefore, $r'_v \geq r'_p$.
- If p is a circumcenter that was considered for insertion but rejected because it encroaches upon s, there are two subcases to consider.
  - If s has legal length $c2^x$, where $c \in (1, \sqrt{2}]$ and x is an integer, then s is precisely bisected. By Lemma 15, $r_v \geq \frac{r_p}{\sqrt{2}}$. However, $r_v = c2^{x-1}$ is in a legal range, so $r'_v \geq r'_p$. If r'_v were smaller than r'_p, then r_v would lie in the illegal range $(\frac{r'_p}{\sqrt{2}}, r'_p]$.
  - If s has illegal length $c2^x$, where $c \in (\sqrt{2}, 2]$ and x is an integer, then the most unbalanced split possible occurs if $c = \frac{3}{2}$, in which case $c_1 = \frac{\sqrt{2}}{4}$. Because s has illegal length, it must be an input segment, and its endpoints are input vertices, so $\mathrm{lfs}_{\min}$ is no greater than $c2^x$. The insertion radius r_v is equal to $c_1 2^x$. Because $c_1 2^x$ is in a legal range, $r'_v \geq \frac{r_v}{\sqrt{2}}$. Hence,

$$\begin{aligned} \frac{\mathrm{lfs}_{\min}}{r'_v} &\leq \sqrt{2}\frac{\mathrm{lfs}_{\min}}{r_v} \\ &\leq \sqrt{2}\frac{c2^x}{c_1 2^x} \\ &\leq 6, \end{aligned}$$

and the theorem holds. ■

Theorem 24 *Suppose that any two incident segments are separated by an angle of at least* 60°*, and a triangle is considered to be skinny if its circumradius-to-shortest edge ratio is larger than* $B \geq 1$*. Ruppert's algorithm with range-restricted segment splitting will terminate, with no triangulation edge shorter than* $\mathrm{lfs}_{\min}/6$.

Proof: By Lemma 23, the insertion radius floor r'_v of every vertex v is either greater than or equal to $\mathrm{lfs}_{\min}/6$, or greater than or equal to the insertion radius floor of some preexisting vertex. Because a vertex's insertion radius floor is a lower bound on its insertion radius, no edge smaller than $\mathrm{lfs}_{\min}/6$ is ever introduced into the mesh, and the algorithm must terminate. ■

The bound can be improved to $\mathrm{lfs}_{\min}/4$. The bound of $\mathrm{lfs}_{\min}/6$ results because a segment of length $\mathrm{lfs}_{\min}$ may undergo a worst-case unbalanced segment split. To prevent this, define the legal ranges to be of the form $c2^x$ where $c \in [\mathrm{lfs}_{\min}, \sqrt{2}\mathrm{lfs}_{\min}]$, instead of $c \in (1, \sqrt{2}]$. With this choice of legal range, only segments longer than $\sqrt{2}\mathrm{lfs}_{\min}$ undergo an unbalanced split, and only segments of length at least $\frac{3}{2}\mathrm{lfs}_{\min}$ undergo a worst-case unbalanced split. To implement this modification, the initial triangulation must be scanned to determine the value of $\mathrm{lfs}_{\min}$.

I recommend two changes to range-restricted segment splitting for practical implementation. First, legal lengths may be defined by the closed range $[1, \sqrt{2}]$ rather than $(1, \sqrt{2}]$. Theoretically, this can be justified by the fact that a vertex inserted at the midpoint of a segment because of an encroaching circumcenter has an insertion radius *strictly* greater than $\frac{1}{\sqrt{2}}$ times its parent's. In practice, floating-point roundoff error renders such quibbles meaningless, but the choice of this legal range is justified because there is always some "slack" in the mesh; not every inserted vertex has the smallest possible insertion radius relative to its parent. If a

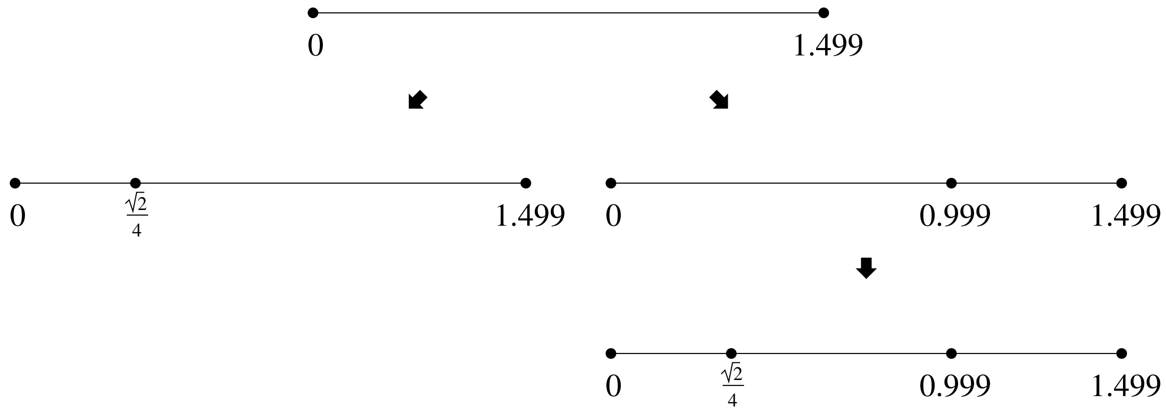


Figure 3.27: An example demonstrating a modification to the method of splitting certain illegal segments. The original method (left) splits an illegal segment of length 1.499 into legal subsegments of lengths $\frac{\sqrt{2}}{4}$ and $1.499 - \frac{\sqrt{2}}{4}$. The modified method (right) splits it into an illegal subsegment of length 0.999 and a legal subsegment of length 0.5. If the illegal subsegment is split in turn, both of its subsegments are legal, and neither is shorter than the smaller subsegment of the original method.

closed legal range is used, redefine the insertion radius floor r'_v to be the largest power of two that is less than or equal to r_v, rather than strictly less than r_v.

The second change modifies the rule for values of c in the range $(\sqrt{2}, \frac{3}{2}]$. In this case, choose $c_1 = \frac{1}{2}$ and $c_2 = c - c_1$. For this choice, c_2 is not a legal length, but if the illegal subsegment of length $c_2 2^x$ is itself split, two legal subsegments result, and the smaller one has length $\frac{\sqrt{2}}{4}2^x$. This subsegment is no worse than the smaller subsegment of the original scheme, as Figure 3.27 illustrates. If by good fortune the subsegment having length $c_2 2^x$ is not split, the creation of an unnecessarily small feature is avoided.

Theorem 24 holds even if the two practical changes discussed above are used, with small modifications to the proof.

I turn now to Chew's algorithm. Another advantage of diametral lenses over diametral circles is that they make it possible to use narrower illegal ranges. An encroaching circumcenter v cannot have a child whose insertion radius is smaller than $r_v \cos\theta$, so the width of each illegal range need only be $\frac{1}{\cos\theta}$. For instance, if $\theta = 30^\circ$, one may use illegal ranges having a geometric width of $\frac{2}{\sqrt{3}} \doteq 1.15$ instead of $\sqrt{2} \doteq 1.41$. In this case, illegal segment lengths are of the form $c2^x$, where $c \in (\sqrt{3}, 2)$, and x is an integer.

Because the legal range can be made wider, and the illegal range narrower, than when diametral circles are used, splitting an illegal segment to yield legal segments is easier. Chew handles illegal segments by trisecting them; one can do better by splitting them into two pieces, just unevenly enough to ensure that both subsegment lengths are legal. The following recipe is suggested.

- If $c \in (\sqrt{3}, 1 + \frac{\sqrt{3}}{2}]$, then $c_1 = 1$ and $c_2 = c - c_1$.
- If $c \in (1 + \frac{\sqrt{3}}{2}, 2)$, then $c_1 = \frac{\sqrt{3}}{2}$ and $c_2 = c - c_1$.

The most unbalanced split occurs if c is infinitesimally larger than $\sqrt{3}$. In this case, the ratio between c_2 and c is approximately $1 - \frac{1}{\sqrt{3}} \doteq 0.4226$, which isn't much worse than bisection.

For comparison, I shall describe how Chew [21] guarantees that his algorithm terminates for an angle bound of 30°. Chew employs range-restricted segment splitting, but uses only one range instead of an infinite

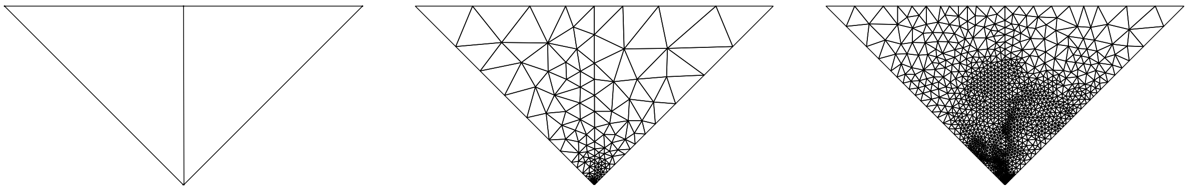


Figure 3.28: (a) A PSLG with a tiny segment at the bottom tip. (b) With a minimum angle of 32° ($B \doteq 0.944$), Ruppert's algorithm creates a well-graded mesh, despite the lack of theoretical guarantees. (c) With a minimum angle of 33.5° ($B \doteq 0.906$), grading is poor.

sequence of ranges. For an appropriate value h (see Chew for details, but one could use $h = \mathrm{lfs}_{\min}/2$), Chew declares the range $(\sqrt{3}h, 2h)$ invalid; subsegments with lengths between $2\sqrt{3}h$ and $4h$ are trisected rather than bisected. Hence, no edge smaller than h ever appears.

There are two disadvantages of using a single illegal range, rather than an infinite series of illegal ranges. The first is the inconvenience of computing h in advance. The second and more fundamental problem is that if small angles are present in the input PSLG, edges smaller than h may arise anyway; see Section 3.7 for a discussion of the problem and its cures.

It does not appear to be possible to prove that Delaunay refinement with range-restricted segment splitting produces graded or size-optimal meshes with circumradius-to-shortest edge ratios that are very close to one. The difficulty is that if a mesh contains a long segment with a small feature size at one end, the small feature size might be expected to propagate along the whole length of the segment. A small subsegment at one end of the segment might indirectly cause its neighboring subsegment to be split until the neighbor is the same size. The neighboring subsegment might then cause its neighbor to be split, and so on down the length of the segment.

As Figure 3.28 demonstrates, however, even if diametral circles are used, a chain reaction severe enough to compromise the grading of the mesh only seems to occur in practice if the quality bound is less than about 0.92 (corresponding to an angle of about 33°)!

The meshes in this figure were generated without range-restricted segment splitting, which is useful as a theoretical construct but unnecessary in practice. As I have mentioned before, there is a good deal of slack in the inequalities that underly the proof of termination, because newly inserted vertices rarely have worst-case insertion radii. As a result of this slack, any Delaunay refinement algorithm that handles boundaries in a reasonable way seems to achieve angle bounds higher than 30°. An examination of range-restricted segment splitting reveals why we should expect this to be true: an ever-diminishing sequence of edges is possible only through an endless chain reaction of alternating splits of segments of legal and illegal length, and only if the sequence of vertex insertions encounters little slack on its infinite journey. Such an occurrence is improbable.

The improvements described thus far are improvements to the circumradius-to-shortest edge ratio of the triangles of a mesh; however, they have not reduced the minimum permissible angle between input segments. The next two sections consider the problem of dealing with angles smaller than 60°. The first of these two sections sets limits on what is possible, and shows that we cannot be overly ambitious.

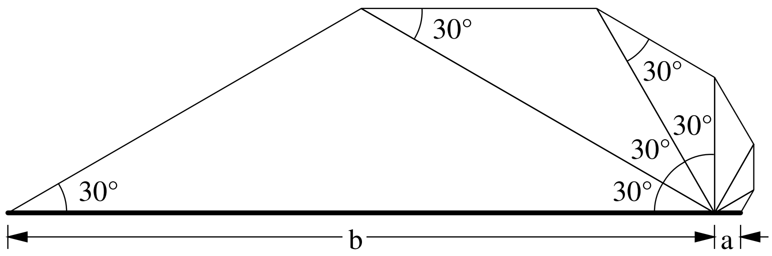


Figure 3.29: In any triangulation with no angles smaller than $30°$, the ratio b/a cannot exceed 27.

3.6 A Negative Result on Quality Triangulations of PSLGs That Have Small Angles

For any angle bound $\theta > 0$, there exists a PSLG $\mathcal{P}$ such that it is not possible to triangulate $\mathcal{P}$ without creating a new corner (not present in $\mathcal{P}$) whose angle is smaller than θ. This statement applies to any triangulation algorithm, and not just those discussed in this thesis. Here, I discuss why this is true.

The result holds for certain PSLGs that have an angle much smaller than θ. Of course, one must respect the PSLG; small input angles cannot be removed. However, one would like to believe that it is possible to triangulate a PSLG without creating any small angles that aren't already present in the input. Unfortunately, no algorithm can make this guarantee for all PSLGs.

The reasoning behind the result is as follows. Suppose a segment in a conforming triangulation has been split into two subsegments of lengths a and b, as illustrated in Figure 3.29. Mitchell [68] proves that if the triangulation has no angles smaller than θ, then the ratio b/a has an upper bound of $(2\cos\theta)^{180°/\theta}$. (This bound is tight if $180°/\theta$ is an integer; Figure 3.29 offers an example where the bound is obtained.) Hence any bound on the smallest angle of a triangulation imposes a limit on the gradation of triangle sizes along a segment (or anywhere in the mesh).

A problem can arise if a small angle ϕ occurs at the intersection vertex o of two segments of a PSLG, and one of these segments is separated by a much larger angle from a third segment incident at o. Figure 3.30 (top) illustrates this circumstance. Assume that the middle segment of the three is split by a vertex p, which may be present in the input or may be inserted to help achieve the angle constraint elsewhere in the triangulation. The insertion of p forces the narrow region between the first two segments to be triangulated (Figure 3.30, center), which may necessitate the insertion of a new vertex q on the segment containing p. Let $a = |pq|$ and $b = |op|$ as illustrated. If the angle bound is respected, the length a cannot be large; the ratio a/b is bounded below

$$\frac{\sin\phi}{\sin\theta}\left(\cos(\theta+\phi)+\frac{\sin(\theta+\phi)}{\tan\theta}\right).$$

If the region above the narrow region is part of the interior of the PSLG, the fan effect demonstrated in Figure 3.29 may necessitate the insertion of another vertex r between o and p (Figure 3.30, bottom); this circumstance is unavoidable if the product of the bounds on b/a and a/b given above is less than one. For an angle constraint of $\theta = 30°$, this condition occurs when ϕ is about six tenths of a degree. Unfortunately, the new vertex r creates the same conditions as the vertex p, but is closer to o; the process will cascade, eternally necessitating smaller and smaller triangles to satisfy the angle constraint. No algorithm can produce a finite triangulation of such a PSLG without violating the angle constraint.

This bound is probably not strict. It would not be surprising if a $30°$ angle bound is not obtainable by

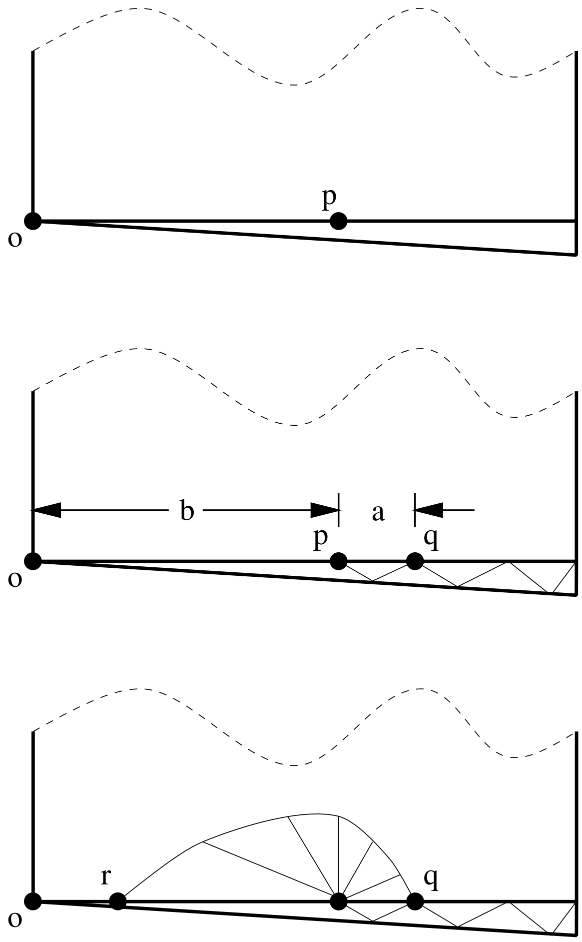


Figure 3.30: Top: A difficult PSLG with a small interior angle ϕ. Center: The vertex p and the angle constraint necessitate the insertion of the vertex q. Bottom: The vertex q and the angle constraint necessitate the insertion of the vertex r. The process repeats eternally.

any algorithm for $\phi = 1°$, and Delaunay refinement often fails in practice to achieve a 30° angle bound for $\phi = 5°$.

Oddly, it appears to be straightforward to triangulate this PSLG using an infinite number of well-shaped triangles. A vertex at the apex of a small angle can be shielded with a thin strip of well-shaped triangles, as Figure 3.31 illustrates. (This idea is related to Ruppert's technique of using *shield edges* [82]. However, Ruppert mistakenly claims that the region concealed behind shield edges always has a finite good-quality triangulation.) The strip is narrow enough to admit a quality triangulation at the smallest input angle. Its shape is chosen so that the angles it forms with the segments outside the shield are obtuse, and the region outside the shield can be triangulated by Delaunay refinement. The region inside the shield is triangulated by an infinite sequence of similar strips, with each successive strip smaller than the previous strip by a constant factor close to one.

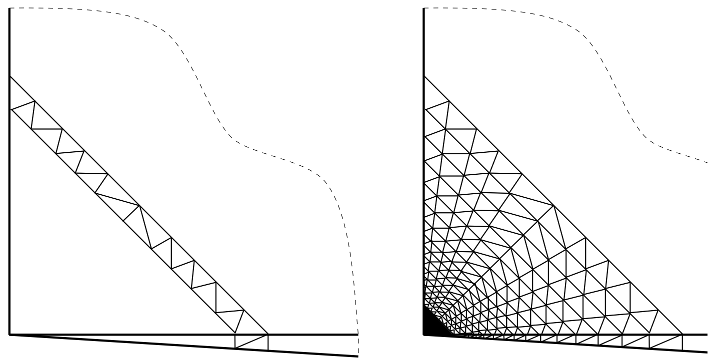


Figure 3.31: How to create a quality triangulation of infinite cardinality around the apex of a very small angle. The method employs a thin strip of well-shaped triangles about the vertex (left). Ever-smaller copies of the strip fill the gap between the vertex and the outer strip (right).

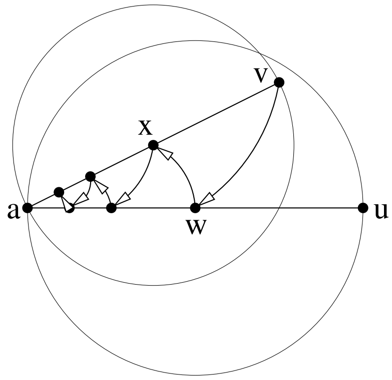


Figure 3.32: A problem caused by a small input angle. Vertex v encroaches upon au, which is split at w. Vertex w encroaches upon av, which is split at x. Vertex x encroaches upon aw, and so on.

3.7 Practical Handling of Small Input Angles

A practical mesh generator should not respond to small input angles by failing to terminate, even if the only alternative is to leave bad angles behind. The result of the previous section quashes all hope of finding a magic pill that will make it possible to triangulate any PSLG without introducing additional small angles. The Delaunay refinement algorithms discussed thus far will fail to terminate on PSLGs like that of Figure 3.30. Of course, Delaunay refinement algorithms should be modified so that they do not try to split any skinny triangle that bears a small input angle. However, even this change does not help with the bad PSLGs described in the previous section, because such PSLGs always have a small angle that is removable, but another small angle invariably takes its place. How can one detect this circumstance, and ensure termination of the algorithm without unnecessarily leaving many bad angles behind?

Figure 3.32 demonstrates one of the difficulties caused by small input angles. If two incident segments have unmatched lengths, a endless cycle of mutual encroachment may produce ever-smaller subsegments incident to the apex of the small angle. For diametral spheres, this phenomenon is only observed with angles smaller than $45°$; for diametral lenses, only with angles smaller than roughly $22.24°$.

To solve this problem, Ruppert [82] suggests “modified segment splitting using concentric circular shells”. Imagine that each input vertex is encircled by concentric circles whose radii are all the powers of two, as illustrated in Figure 3.33. When an encroached subsegment has an endpoint that is an input vertex

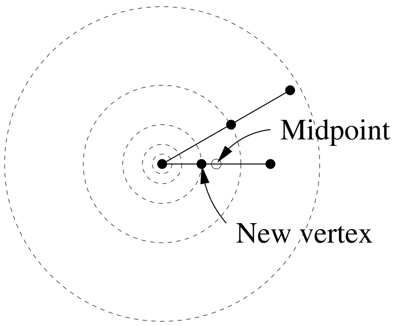


Figure 3.33: If an encroached subsegment has a shared input vertex for an endpoint, the subsegment is split at its intersection with a circular shell whose radius is a power of two.

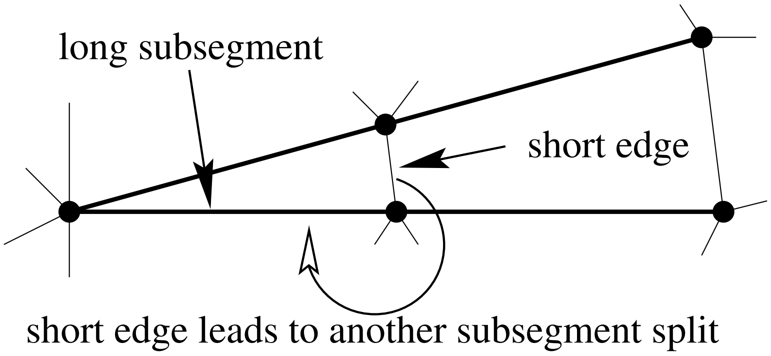


Figure 3.34: The short edge opposite a small angle can cause other short edges to be created as the algorithm attempts to remove skinny triangles. If the small insertion radii propagate around an endpoint and cause the supporting subsegments to be split, a shorter edge is created, and the cycle may repeat.

shared with another segment, the subsegment is split not at its midpoint, but at one of the circular shells, so that one of resulting subsegments has a power-of-two length. The shell that gives the best balanced split is chosen; in the worst case, the smaller resulting subsegment is one-third the length of the split subsegment. If both endpoints are shared input vertices, choose one endpoint's shells arbitrarily. Range-restricted segment splitting may optionally be used on all subsegments not subject to concentric shell splitting. Each input segment may undergo up to three unbalanced splits: two that create power-of-two subsegments at the ends of the segment, and one to split an illegal subsegment lying between these two. All other subsegment splits are bisections.

Concentric shell segment splitting prevents the runaway cycle of ever-smaller subsegments portrayed in Figure 3.32, because incident subsegments of equal length do not encroach upon each other. Again, it is important to modify the algorithm so that it does not attempt to split a skinny triangle that bears a small input angle, and cannot be improved.

Modified segment splitting using concentric circular shells is generally effective in practice for PSLGs that have small angles greater than 10°, and often for smaller angles. It is always effective for polygons with holes (for reasons to be discussed shortly). As the previous section hints, difficulties are only likely to occur when a small angle is adjacent to a much larger angle. The negative result of the previous section arises not because subsegment midpoints can cause incident subsegments to be split, but because the free edge opposite a small angle is shorter than the subsegments whose endpoints define it, as Figure 3.34 illustrates.

The two subsegments of Figure 3.34 are coupled, in the sense that if one is bisected then so is the

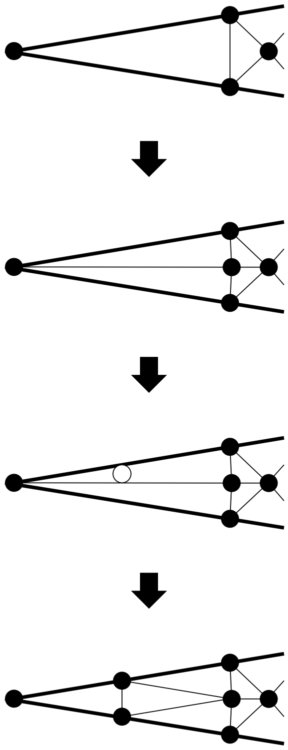


Figure 3.35: Concentric shell segment splitting ensures that polygons (with holes) can be triangulated, because it causes small angles to be clipped off. This sequence of illustrations demonstrate that if a clipped triangle's interior edge is flipped, a smaller clipped triangle will result.

other, because the midpoint of one encroaches upon the other. This holds true for any two segments of equal length separated by less than $60°$, if diametral circles are used, or $\arctan\sqrt{\frac{13}{3}} - 30° \doteq 34.34°$, if diametral lenses are used. Each time such a dual bisection occurs, a new edge is created that is smaller than the subsegments produced by the bisection; the free edge can be arbitrarily small if the angle is arbitrarily small. One of the endpoints of the free edge has a small insertion radius, though that endpoint's parent (typically the other endpoint) might have a large insertion radius. Hence, a small angle functions as an "insertion radius reducer". The new small edge will likely engender other small edges as the algorithm attempts to remove skinny triangles. If small insertion radii propagate around an endpoint of the small edge, the incident subsegments may be split again, commencing an infinite sequence of smaller and smaller edges.

If the PSLG is a polygon (possibly with polygonal holes), small insertion radii cannot propagate around the small edge, because the small edge partitions the polygon into a skinny triangle (which the algorithm does not attempt to split) and everything else. The small edge is itself flipped or penetrated only if there is an even smaller feature elsewhere in the mesh. If the small edge is thus removed, the algorithm will attempt to fix the two skinny triangles that result, thereby causing the subsegments to be split again, thus creating a new smaller edge (Figure 3.35).

For general PSLGs, how may one diagnose and cure diminishing cycles of edges? A sure-fire way to guarantee termination was hinted at in Section 3.5.1: never insert a vertex whose insertion radius is smaller than the insertion radius of its most recently inserted ancestor (its parent if the parent was inserted; its grandparent if the parent was rejected), unless the parent is an input vertex or lies on a nonincident segment.

This restriction is undesirably conservative for two reasons. First, if a Delaunay triangulation is desired,

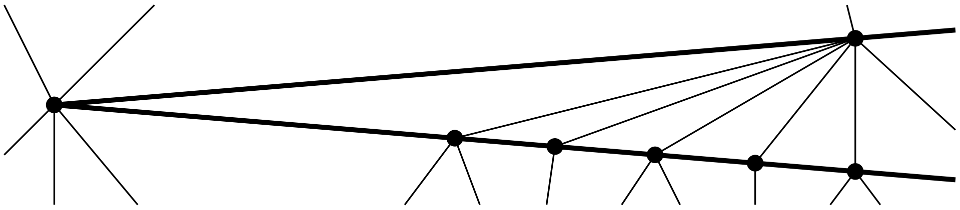
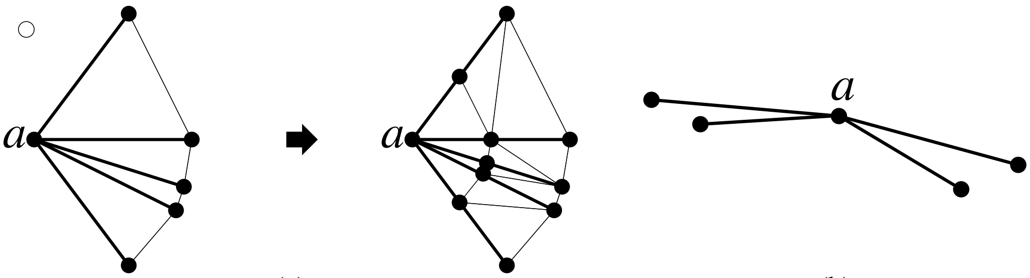


Figure 3.36: The simplest method of ensuring termination when small input angles are present has undesirable properties, including the production of large angles and many small angles.



(a) (b)

Figure 3.37: (a) Example of a subsegment cluster. If all the subsegments of a cluster have power-of-two lengths, then they all have the same length and are effectively split as a unit because of mutual encroachment. (b) Several independent subsegment clusters may share the same apex.

the restriction might prevent us from obtaining one, because segments may be left encroached. A second, more pervasive problem is demonstrated in Figure 3.36. Two subsegments are separated by a small input angle, and one of the two is bisected. The other subsegment is encroached, but is not bisected because a small edge would be created. One unfortunate result is that the triangle bearing the small input angle also bears a large angle of almost 180°. Recall that large angles can be worse than small angles, because they jeopardize convergence and interpolation accuracy in a way that small angles do not. Another unfortunate result is that many skinny triangles may form. The triangles in the figure cannot be improved without splitting the upper subsegment.

As an alternative, I suggest the following scheme.

The Quitter: A Delaunay refinement algorithm that knows when to give up. Guaranteed to terminate.

The Quitter is based on Delaunay refinement with concentric circular shells; range-restricted segment splitting is optional. When a subsegment s is encroached upon by the circumcenter of a skinny triangle, a decision is made whether to split it with a vertex v, or to leave it whole. (In either case, the circumcenter is rejected for insertion.) The decision process is somewhat elaborate.

If neither endpoint of s bears a small input angle (less than 60°), or if both endpoints do, then s is split. Otherwise, let a be the apex of the small angle. Define the *subsegment cluster* of s to be the set of subsegments incident to a that are separated from s, or from some other member of the subsegment cluster of s, by less than 60°. If diametral circles are used, once all the subsegments of a cluster have been split to power-of-two lengths, they must all be the same length to avoid encroaching upon each other. If one is bisected, the others follow suit, as illustrated in Figure 3.37(a). If v is inserted it is called a *trigger vertex*, because it may trigger the splitting of all the subsegments in a cluster.

If diametral lenses are used, it is no longer true that all the subsegments in a cluster split as a unit. However, clusters are still defined by a 60° angle, because diametral lenses do not diminish the problem of small edges appearing opposite a cluster apex.

The definition of subsegment cluster does not imply that all subsegments incident to an input vertex are part of the same cluster. For instance, Figure 3.37(b) shows two independent subsegment clusters sharing one apex, separated from each other by angles of at least 60°.

To decide whether s should be split, the Quitter determines the insertion radius r_g of v's grandparent g (which is the parent of the encroaching circumcenter), and the minimum insertion radius $r_{\min}$ of all the midpoint vertices (including v) that will be introduced into the subsegment cluster of s if all the subsegments in the cluster having length $|s|$ or greater are split. If all the subsegments in the cluster have the same length, then $r_{\min}$ depends upon the smallest angle in the subsegment cluster.

The vertex v is inserted, splitting s, only if one or more of the following three conditions hold.

- If $r_{\min} \geq r_g$, then v is inserted.
- If one of the segments in the subsegment cluster of s has a length that is not a power of two, then v is inserted.
- If no ancestor of v also lies in the interior of the segment containing s, then v is inserted. (Endpoints of the segment are exempt.)

End of description of the Quitter.

If there are no input angles smaller than 60°, the Quitter acts no differently from Ruppert's or Chew's algorithm by the following reasoning. Any encroached subsegment s is the only subsegment in its cluster, and $r_{\min} = r_v$. If s is precisely bisected, Theorem 16 states that the first condition ($r_{\min} \geq r_g$) always holds. If the length of s is not a power of two, s may be split unevenly, and hence the condition $r_{\min} \geq r_g$ may not be true, but the second condition above ensures that such splits are not prevented.

On the other hand, if small angles are present, and the first condition fails for some encroached segment, the third condition identifies situations in which the mesh can be improved without threatening the guarantee of termination. This rule attempts to distinguish between the case where a segment is encroached because of small input features, and the case where a segment is encroached because it bears a small angle.

Theorem 25 *The Quitter always terminates.*

Proof sketch: Suppose for the sake of contradiction that the Quitter fails to terminate. Then there must be an infinite sequence of vertices V with the property that each vertex of V (except the first) is the child of its predecessor, and for any positive real value d, some vertex in V has insertion radius smaller than d. (If there is no such sequence of descendants, then there is a lower bound on the length of an edge, and the algorithm must terminate.)

Say that a vertex v has the *diminishing property* if its insertion radius floor r'_v is less than that of all its ancestors. The sequence V contains an infinite number of vertices that have the diminishing property.

Thanks to Lemma 23, if a vertex v has an insertion radius floor smaller than that of all its ancestors, then v must have been inserted in a subsegment s under one of the following conditions:

- s bears a small input angle, and the length of s is not a power of two.
- s is of illegal length.

- s is encroached upon by an input vertex or a vertex lying on a segment not incident to s.
- s is encroached upon by a vertex p that lies on an segment incident to s at an angle less than 60°.

Only a finite number of vertices can be inserted under the first three conditions. The first condition can occur twice for each input segment (once for each end), and the second condition can occur once for each input segment. Any subsegment shorter than $\mathrm{lfs}_{\min}$ cannot be encroached upon by a nonincident feature, so only a finite number of vertex insertions of the third type are possible as well. Hence, V must contain an infinite number of vertices inserted under the fourth condition.

However, V cannot have arbitrarily long runs of such vertices, because power-of-two segment splitting prevents a cluster of incident segments from engaging in a chain reaction of ever-diminishing mutual encroachment. Specifically, let 2^x be the largest power of two less than or equal to the length of the shortest subsegment in the cluster. No subsegment of the cluster can be split to a length shorter than 2^{x-1} through the mechanism of encroachment alone. The edges opposite the apex of the cluster may be much shorter than 2^{x-1}, but some other mechanism is needed to explain how the sequence V can contain insertion radii even shorter than these edges. The only such mechanism that can be employed an infinite number of times is the attempted splitting of a skinny triangle. Hence, V must contain an infinite number of trigger vertices.

One of the rules is that a trigger vertex may only be inserted if it has no ancestor in the interior of the same segment. Hence, V may only contain one trigger vertex for each input segment. It follows that the number of trigger vertices in V is finite, a contradiction. ■

The Quitter eliminates all encroached subsegments, so if diametral circles are used, there is no danger that a segment will fail to appear in the final mesh (if subsegments are not locked), or that the final mesh will not be Delaunay (if subsegments are locked). Because subsegments are not encroached, an angle near 180° cannot appear immediately opposite a subsegment (as in Figure 3.36), although large angles can appear near subsegment clusters. The Quitter offers no guarantees on quality when small input angles are present, but skinny triangles in the final mesh occur only near input angles less than 60°.

The Quitter has the unfortunate characteristic that it demands more memory than would otherwise be necessary, because each vertex of the mesh must store its insertion radius and a pointer to its parent (or, if its parent was rejected, its grandparent). Hence, I suggest possible modifications to avoid these requirements.

The Quitter needs to know the insertion radius of a vertex only when a trigger vertex v is being considered for insertion. It is straightforward to compute the insertion radii of v and the other vertices that will be inserted into the cluster. However, the insertion radius of the grandparent of the trigger vertex is used for comparison, and may not be directly computable from the mesh, because other vertices may have been inserted near g since g was inserted. Nevertheless, it is reasonable to approximate r_g by using the length d of the shortest edge of the skinny triangle whose circumcenter is v's parent, illustrated in Figure 3.38. The length d is an upper bound on r_g, so its use will not jeopardize the Quitter's termination guarantee; the modified algorithm is strictly more conservative in its decision of whether to insert v. With this modification, there is no need to store the insertion radii of vertices for later use.

The only apparent way to avoid storing a pointer from each vertex to its nearest inserted ancestor is to eliminate the condition that a trigger vertex may be inserted if none of its ancestors lies in the same segment. The possible disadvantage is that a small nearby input feature might fail to cause the segment to be split even though it ought to have the privilege, and thus skinny triangles will unnecessarily remain in the mesh.

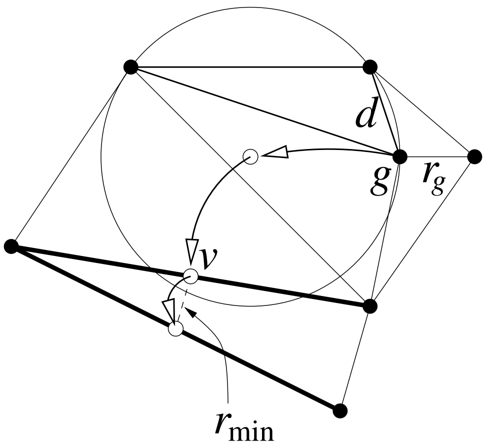


Figure 3.38: The length d of the shortest edge of a skinny triangle is an upper bound on the insertion radius r_g of the most recently inserted endpoint of that edge.

3.8 Conclusions

The intuition governing Delaunay refinement comes from an understanding of the relationship between the insertion radii of parents and their children. Hence, I use dataflow graphs such as Figure 3.18 to demonstrate these relationships. This manner of thinking brings clarity to ideas that otherwise might be hidden within proofs. For instance, Figure 3.18 provides an immediate explanation for why Ruppert's algorithm achieves an angle bound of up to 20.7° (which corresponds to a circumradius-to-shortest edge ratio of $\sqrt{2}$). The same ideas can be found in Ruppert's original paper, but are somewhat obscured by the mathematics. By bringing the intuition to the forefront, and by explicitly graphing the relationships between the insertion radii of related vertices, I have found a variety of improvements to Delaunay refinement and its analysis, which have been discussed in detail in this chapter and are listed again here.

- The minimum angle separating two input segments can be relaxed from the 90° bound specified by Ruppert to a 60° bound. This observation comes from the dataflow graph of Figure 3.18.
- My new analysis of Chew's algorithm arose from my attempts to understand the relationship between segment midpoints and their parents, which is reflected in the dataflow graph of Figure 3.24.
- The dataflow graphs for Ruppert's and Chew's algorithms sparked my recognition of the fact that a better quality bound can be applied in the interior of the mesh, as illustrated in Figure 3.25.
- The idea of range-restricted segment splitting arose from my attempts to find ways to weaken the spiral of diminishing insertion radii. (Only later did I realize that Chew had developed a very similar idea.)
- My method for handling small input angles works by preventing vertices from having children that might contribute to a sequence of vertices with endlessly diminishing insertion radii.

Hence, this manner of approaching Delaunay refinement has shown great fecundity. Simple as these dataflow graphs are, they have provided the clues that helped to unearth most of the new results in this thesis, and in Chapter 4 they will prove themselves invaluable in studying tetrahedral Delaunay refinement, in which the relationships between the insertion radii of different types of vertices become even more complicated. Most of the improvements in the list above will repeat themselves in the three-dimensional setting.

At this writing, I have not yet implemented diametral lenses. I expect them to outperform diametral circles in circumstances in which long segments are present, because diametral lenses are less prone to be split. On the other hand, diametral circles and diametral lenses will exhibit little or no difference for many inputs whose boundaries are composed of many tiny segments, such as Figure 3.7 (bottom).

My negative result on quality triangulations comes as a surprise, as researchers in mesh generation have been laboring for some time under false assumptions about what is possible in triangular mesh generation. A few have mistakenly claimed that they could provide triangulations of arbitrary PSLGs with no new small angles. Fortunately, a recognition of the fundamental difficulty of triangulating PSLGs with tiny angles makes it easier to formulate a strategy for handling them. Once one realizes that the best one can hope for is to minimize the unavoidable damage that small input angles can cause, it becomes relatively easy to develop a method that prevents vertices having smaller and smaller insertion radii from being inserted. The method I have suggested is somewhat more elaborate than what is necessary to guarantee termination, but is likely to reward the extra effort with better triangulations.

Chapter 4

Three-Dimensional Delaunay Refinement Algorithms

Herein, I build upon the framework of Ruppert and Chew to design a Delaunay refinement algorithm for tetrahedral meshes. The generalization to three dimensions is relatively straightforward, albeit not without complications. The basic operation is still the Delaunay insertion of vertices at the circumcenters of simplices, and the result is still a mesh whose elements have bounded circumradius-to-shortest edge ratios.

Unfortunately, unlike the two-dimensional case, such a mesh is not necessarily adequate for the needs of finite element methods. The reason is the existence of a type of tetrahedron called a *sliver* or *kite*. The canonical sliver is formed by arranging four vertices, equally spaced, around the equator of a sphere, then perturbing one of the vertices slightly off the equator, as illustrated in Figure 4.1. As is apparent in the figure, a sliver can have an admirable circumradius-to-shortest edge ratio (as low as $\frac{1}{\sqrt{2}}$!) yet be considered awful by most other measures, because its volume and its shortest altitude can be arbitrarily close to zero. Slivers have no two-dimensional analogue; any triangle with a small circumradius-to-shortest edge ratio is considered "well-shaped" by the usual standards of finite element methods.

Slivers often survive Delaunay-based tetrahedral mesh generation methods because their small circumradii minimize the likelihood of vertices being inserted in their circumspheres (Figure 4.2). A perfectly flat sliver whose edge lengths are $\mathrm{lfs}_{\min}$ about the equator and $\sqrt{2}\mathrm{lfs}_{\min}$ across the diagonals is guaranteed to survive any Delaunay refinement method that does not introduce edges smaller than $\mathrm{lfs}_{\min}$, because every

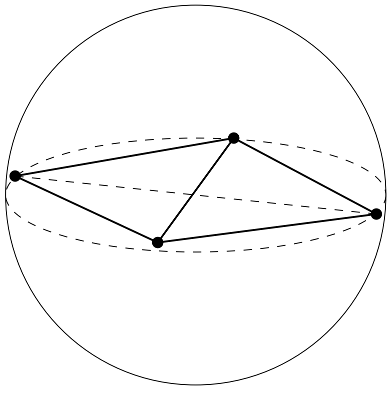


Figure 4.1: A sliver tetrahedron.

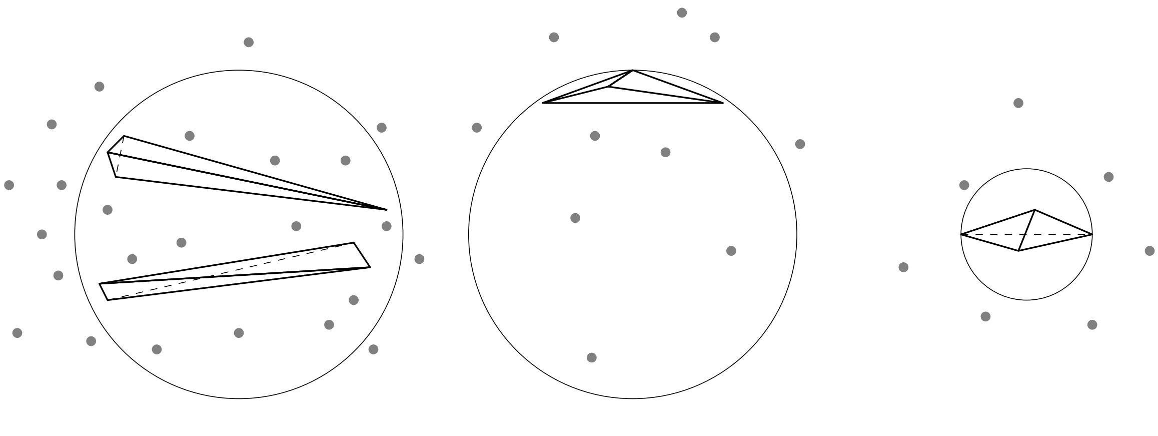


Figure 4.2: In three dimensions, skinny tetrahedra known as needles and caps have circumspheres significantly larger than their shortest edge, and are thus eliminated when additional vertices are inserted, spaced to match the shortest edge. A sliver can have a circumradius smaller than its shortest edge, and can easily survive in a Delaunay tetrahedralization of uniformly spaced vertices.

point in the interior of its circumsphere is a distance less than $\mathrm{lfs}_{\min}$ from one of its vertices; no vertex can be inserted inside the sphere.

Despite this problem, Delaunay refinement methods are valuable for generating three-dimensional meshes. The worst slivers can often be removed by Delaunay refinement, even if there is no theoretical guarantee. Meshes with bounds on the circumradius-to-shortest edge ratios of their tetrahedra are an excellent starting point for mesh smoothing and optimization methods designed to remove slivers and improve the quality of an existing mesh (see Section 2.2.4). Even if slivers are not removed, Delaunay refinement tetrahedralizations are sometimes adequate for such numerical techniques as the control volume method [66], which operates upon the Voronoi diagram rather than the Delaunay tetrahedralization. The Voronoi dual of a tetrahedralization with bounded circumradius-to-shortest edge ratios has nicely rounded cells, even if slivers are present in the tetrahedralization itself.

In this chapter, I present a three-dimensional generalization of Ruppert's algorithm that generates tetrahedralizations whose tetrahedra have circumradius-to-shortest edge ratios no greater than the bound $B = \sqrt{2} \doteq 1.41$. If B is relaxed to be greater than two, then good grading can also be proven. I enhance the algorithm with a structure similar to diametral lenses, and thereby achieve a tetrahedron quality bound of $B = \frac{2}{\sqrt{3}} \doteq 1.15$, or a well-graded mesh for any tetrahedron quality bound that satisfies $B > \frac{2\sqrt{2}}{\sqrt{3}} \doteq 1.63$. Size-optimality, however, cannot be proven.

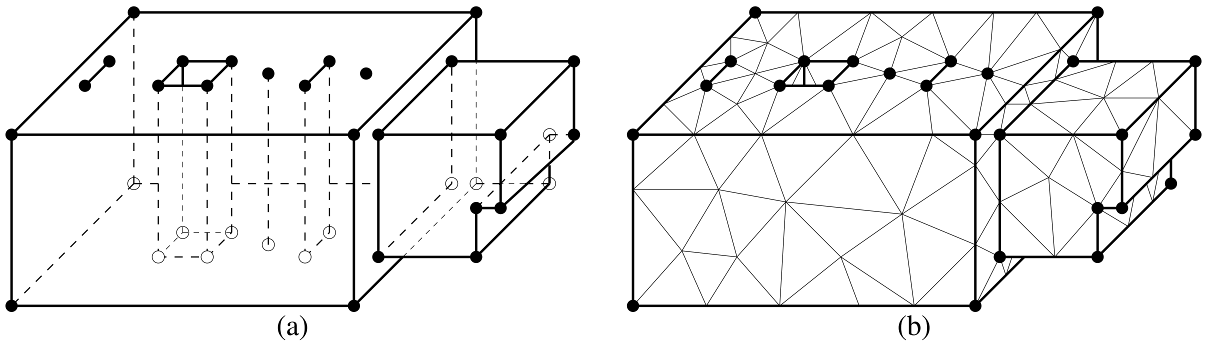


Figure 4.3: (a) Any facet of a PLC may contain holes, slits, and vertices; these may support intersections with other polytopes or allow a user of the finite element method to apply boundary conditions. (b) When a PLC is tetrahedralized, each facet of the PLC is partitioned into triangular subfacets, which respect the holes, slits, and vertices.

4.1 Preliminaries

4.1.1 Piecewise Linear Complexes and Local Feature Size

Before defining a three-dimensional Delaunay refinement algorithm, it is necessary to define the input upon which the algorithm will operate. I use a generalization of a planar straight line graph called a *piecewise linear complex* (PLC); see Miller, Talmor, Teng, Walkington, and Wang [67] for a similar definition that generalizes to any number of dimensions.

In three dimensions, a PLC is a set of vertices, segments, and facets. Vertices and segments are no different than in the two-dimensional case, except that they are embedded in three-dimensional space. Facets, however, can be quite complicated in shape. A facet is a planar boundary, such as the rectangular and nearly-rectangular facets that define the object depicted in Figure 4.3(a). As the figure illustrates, a facet may have any number of sides, may be nonconvex, and may have holes, slits, or vertices in its interior. However, an immutable requirement is that a facet must be planar.

A piecewise linear complex X is required to have the following properties.

- For any facet in X, every edge and vertex of the facet must appear as a segment or vertex of X. Hence, all facets are segment-bounded.
- X contains both endpoints of each segment of X.
- X is closed under intersection. Hence, if two facets of X intersect at a line segment, that line segment must be represented by a segment of X. If a segment or facet of X intersects another segment or facet of X at a single point, that point must be represented by a vertex in X.
- If a segment of X intersects a facet of X at more than a finite number of points, then the segment must be entirely contained in the facet. This rule ensures that facets "line up" with their boundaries. A facet cannot be bounded by a segment that extends beyond the boundary of the facet.

The process of tetrahedral mesh generation necessarily divides each facet into triangular faces, as illustrated in Figure 4.3(b). Just as the edges that compose a segment are called subsegments, the triangular faces that compose a facet are called *subfacets*. All of the triangular faces visible in Figure 4.3(b) are subfacets, but most of the faces in the interior of the tetrahedralization are not.

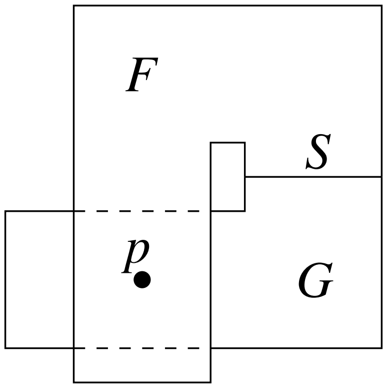


Figure 4.4: Two incident facets separated by a dihedral angle of nearly $180°$. What is the local feature size at p?

Recall that any vertex inserted into a segment remains there permanently. When examining the algorithms discussed in this chapter, keep in mind that vertices inserted into facets are also permanent, but the edges that partition a facet into subfacets are *not* permanent, are *not* treated like subsegments, and are always subject to flipping according to the Delaunay criterion.

Many approaches to tetrahedral mesh generation permanently triangulate the input facets as a separate step prior to tetrahedralizing the interior of a solid. The problem with this approach is that these independent facet triangulations may not be ideal for forming a good tetrahedralization when other surfaces are taken into account. For instance, a feature that lies near a facet (but not necessarily in the plane of the facet) may necessitate the use of smaller subfacets near that feature. The algorithms of this chapter use an alternative approach, wherein facet triangulations are refined in conjunction with the tetrahedralization. Each facet's triangulation can change in response to attempts to improve the tetrahedra of the mesh. The tetrahedralization process is not beholden to poor decisions made earlier.

Because the shape of a facet is versatile, the definition of local feature size does not generalize straightforwardly. Figure 4.4 demonstrates the difficulty. Two facets F and G are incident at a segment S, separated by a dihedral angle of almost $180°$. The facets are not convex, and they may pass arbitrarily close to each other in a region far from S. What is the local feature size at the point p? Because F and G are incident, a ball large enough to intersect two nonincident features must have diameter as large as the width of the prongs. However, the size of tetrahedra near p is determined by the distance separating F and G, which could be arbitrarily small. The straightforward generalization of local feature size does not account for this peccadillo of nonconvex facets.

To develop a more appropriate metric, I define a *facet region* to be any region of a facet visible from a single point on its boundary. (Visibility is defined solely by and within the facet in question.) Two facet regions on two different facets are said to be *incident* if they are defined by the same point. Figure 4.5 illustrates two incident facet regions, and the point that defines them. Two points, one lying in F and one lying in G, are said to lie in incident facet regions if there is any point on the shared boundary of F and G that is visible from both points. They are said to lie in nonincident facet regions if no such point exists. (For higher-dimensional mesh generation, this definition extends unchanged to polytopes of higher dimension.)

Similarly, if a segment S is incident to a facet F at a single vertex a, then S is said to be incident to the facet region of F visible from a. If a vertex v is incident to a facet F, then v is said to be incident to the facet region of F visible from v.

Given a piecewise linear complex X, I define the local feature size $\text{lfs}(p)$ at a point p to be the radius of

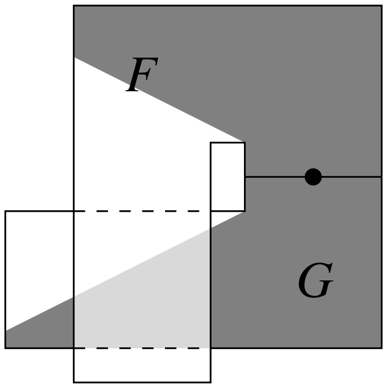


Figure 4.5: Shaded areas are two incident facet regions. Both regions are visible from the indicated point.

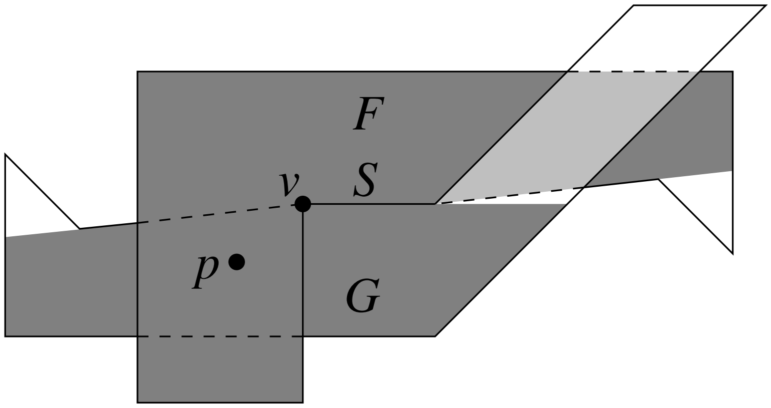


Figure 4.6: Two incident facets separated by a dihedral angle of nearly 180°. The definition of local feature size should not approach zero near v, but it is nonetheless difficult to mesh the region between F and G near v.

the smallest ball centered at p that intersects two points that lie on nonincident vertices, segments, or facet regions of X. (To be rigorous, $\text{lfs}(p)$ is sometimes the radius of the largest ball that doesn't intersect two such points.)

Unfortunately, careful specification of which portions of facets are incident doesn't solve all the problems attributable to nonconvex facets. Figure 4.6 demonstrates another difficulty. Again, two facets F and G are incident at a segment S, separated by a dihedral angle slightly less than $180°$. One endpoint v of S is a reflex vertex of F. The incident facet regions defined by the vertex v have the same problem we encountered in Figure 4.4: the local feature size at point p may be much larger than the distance between facets F and G at point p.

In this case, however, the problem is unavoidable. Suppose one chooses a definition of local feature size that reflects the distance between F and G at p. As p moves toward v, its local feature size approaches zero, suggesting that infinitesimally small tetrahedra are needed to mesh the region near v. Intuitively and practically, a useful definition of local feature size must have a positive lower bound.

The mismatch between the definition of local feature size proposed here and the small distance between F and G at p reflects a fundamental difficulty in meshing the facets of Figure 4.6—a difficulty that is not present in Figure 4.4. In Figure 4.6, it is not possible to mesh the region between F and G at v without resorting to poorly shaped tetrahedra. The facets of Figure 4.4 can be meshed entirely with well-shaped

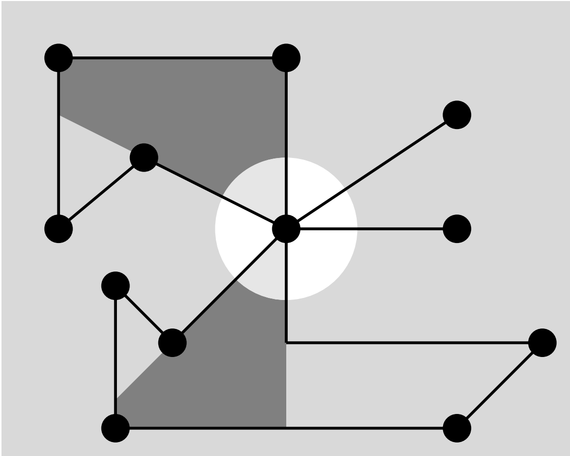


Figure 4.7: In a sufficiently small neighborhood around any vertex, a PLC looks like a set of rays emitted from that vertex.

tetrahedra. My three-dimensional Delaunay refinement algorithm outlaws inputs like Figure 4.6, at least for the purposes of analysis.

There are two reasons why it makes sense to use visibility to define incident features. First, if a point p is not visible from a boundary point v of the same facet, there must be an edge, not incident to v, that separates v from p. Second, if one studies a sufficiently small neighborhood around a vertex, any facet or segment incident to the vertex appears to be a union of rays emanating from that vertex, as illustrated in Figure 4.7. Hence, the local feature size does not approach zero anywhere. Incidentally, examination of an arbitrarily small neighborhood around each vertex is sufficient to diagnose problems like that in Figure 4.6, because the only input features that threaten the termination of Delaunay refinement are those that persist no matter how small the tetrahedra become.

Lemma 14, which states that $\text{lfs}(v) \leq \text{lfs}(u) + |uv|$ for any two points u and v, applies to this definition of local feature size just as it applies in two dimensions. The only prerequisite for the correctness of Lemma 14, besides the triangle inequality, is that there be a consistent definition of which pairs of points lie in incident regions, and which do not.

4.1.2 Orthogonal Projections

Frequently in this chapter, I will use the notion of the *orthogonal projection* of a geometric entity onto a line or plane. Given a facet or subfacet F and a point p, the orthogonal projection $\text{proj}_F(p)$ of p onto F is the point that is coplanar with F and satisfies the requirement that the line $p[\text{proj}_F(p)]$ is orthogonal to F, as illustrated in Figure 4.8. The projection exists whether or not it falls in F.

Similarly, the orthogonal projection $\text{proj}_S(p)$ of p onto a segment or subsegment S is the point that is collinear with S and satisfies the requirement that the direction of projection is orthogonal to S.

Sets of points, as well as points, may be projected. If F and G are facets, then $\text{proj}_F(G)$ is the set $\{\text{proj}_F(p) : p \in G\}$.

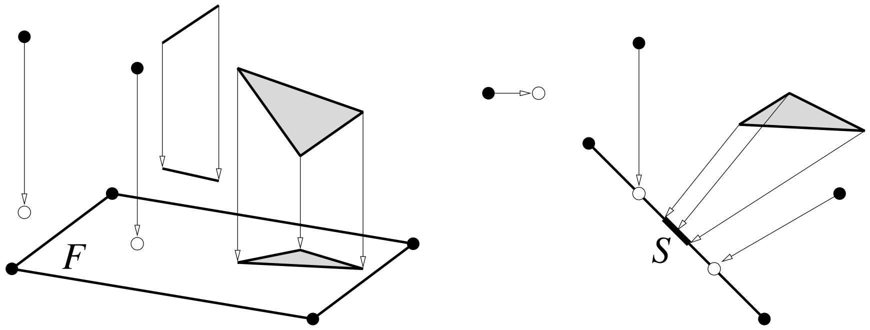


Figure 4.8: The orthogonal projections of points and sets of points onto facets and segments.

4.2 Generalization of Ruppert's Algorithm to Three Dimensions

In this section, I describe a three-dimensional Delaunay refinement algorithm that produces well-graded tetrahedral meshes for any circumradius-to-shortest edge ratio bound greater than two. Improvements to the algorithm are made in later sections. Miller, Talmor, Teng, Walkington, and Wang [67] have developed a related algorithm, which will be discussed in some detail in Section 4.5.

4.2.1 Description of the Algorithm

Three-dimensional Delaunay refinement takes a *facet-bounded* PLC as its input. Tetrahedralized and untetrahedralized regions of space must be separated by facets so that, in the final mesh, any triangular face not shared by two tetrahedra is a subfacet. The algorithm begins with an unconstrained Delaunay tetrahedralization of the input vertices, momentarily ignoring the input segments and facets. As in two dimensions, the tetrahedralization is then refined by inserting additional vertices into the mesh, using an incremental Delaunay tetrahedralization method such as the Bowyer/Watson method [12, 93] or an edge/face flipping method [52, 78], until all segments and facets are present and all constraints on tetrahedron quality are met. Vertex insertion is governed by three rules.

- The *diametral sphere* of a subsegment is the (unique) smallest sphere that contains the subsegment. As in the two-dimensional algorithm, a subsegment is encroached if a vertex lies strictly inside its diametral sphere, or if the subsegment does not appear in the tetrahedralization. Any encroached subsegment that arises is immediately split by inserting a vertex at its midpoint. See Figure 4.9(a).
- The *equatorial sphere* of a triangular subfacet is the (unique) smallest sphere that passes through the three vertices of the subfacet. (The *equator* of an equatorial sphere is the circle defined by the same three vertices.) A subfacet is encroached if a vertex lies strictly inside its equatorial sphere, or if the subfacet is expected to appear in the tetrahedralization but does not. (More on this shortly.) Each encroached subfacet is normally split by inserting a vertex at its circumcenter. However, if a new vertex would encroach upon any subsegment, it is not inserted; instead, all the subsegments it would encroach upon are split. See Figure 4.9(b).
- A tetrahedron is said to be *skinny* if its circumradius-to-shortest edge ratio is larger than some bound B. (By this definition, not all slivers are considered skinny.) Each skinny tetrahedron is normally split by inserting a vertex at its circumcenter, thus eliminating the tetrahedron; see Figure 4.9(c). However,

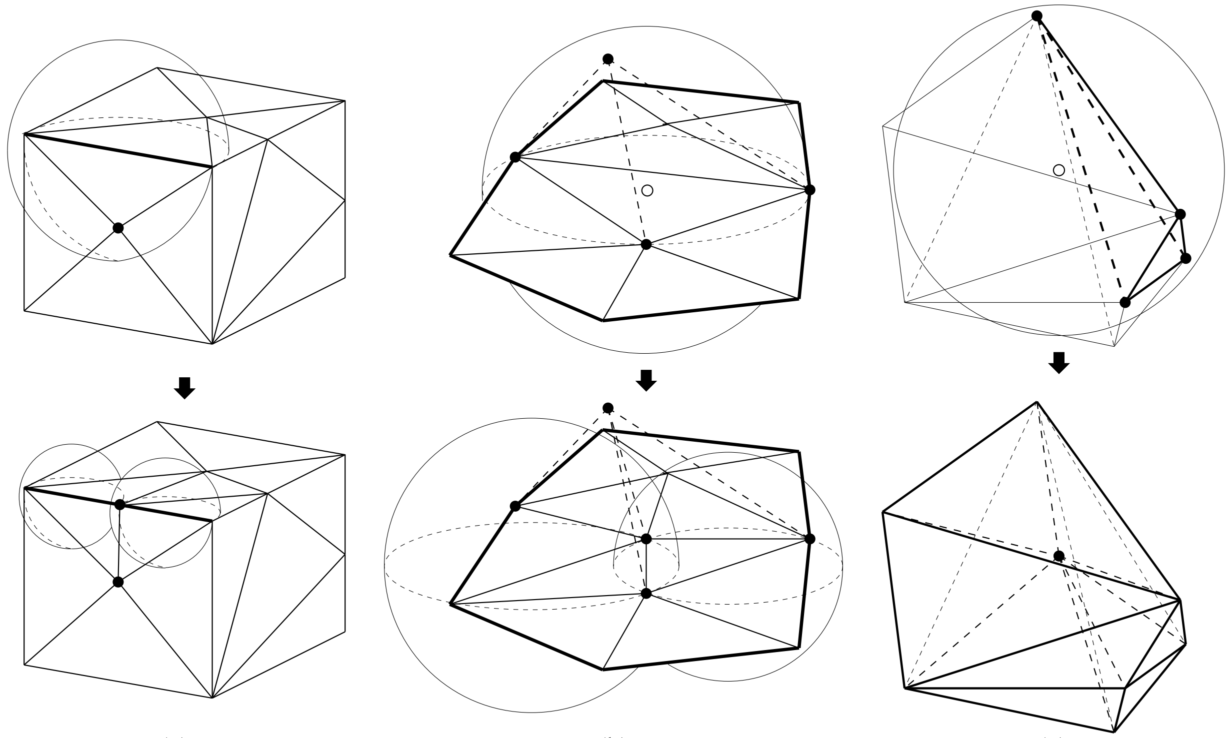


Figure 4.9: Three operations for three-dimensional Delaunay refinement. (a) Splitting an encroached subsegment. Dotted arcs indicate where diametral spheres intersect faces. The subsegment and the encroaching vertex could each be on the surface or in the interior of the mesh. (b) Splitting an encroached subfacet. The triangular faces shown are subfacets of a larger facet, with tetrahedra (not shown) atop them. A vertex in the equatorial sphere of a subfacet causes a vertex to be inserted at its circumcenter. Afterward, all equatorial spheres (included the two illustrated) are empty. (c) Splitting a bad tetrahedron. A vertex is inserted at its circumcenter.

> if a new vertex would encroach upon any subsegment or subfacet, then it is not inserted; instead, all the subsegments it would encroach upon are split. If the skinny tetrahedron is not eliminated as a result, then all the subfacets its circumcenter would encroach upon are split. (A subtle point is that, if the tetrahedron is eliminated by subsegment splitting, the algorithm should not split any subfacets that appear during subsegment splitting, or the bounds proven in the next section will not be valid. Lazy programmers beware.)

Encroached subsegments are given priority over encroached subfacets, which have priority over skinny tetrahedra.

The first obvious complication is that if a facet is missing from the mesh, it is difficult to say what its subfacets are. With segments there is no such problem; if a segment is missing from the mesh, and a vertex is inserted at its midpoint, one knows unambiguously where the two resulting subsegments should be. It is less clear how to identify subfacets that do not yet exist.

The solution is straightforward. For each facet, it is necessary to maintain a triangulation of its vertices, independent from the tetrahedralization in which we hope its subfacets will eventually appear. By comparing the triangles of a facet's triangulation against the faces of the tetrahedralization, one can identify subfacets that need help in forcing their way into the mesh. For each triangular subfacet in a facet triangulation, look

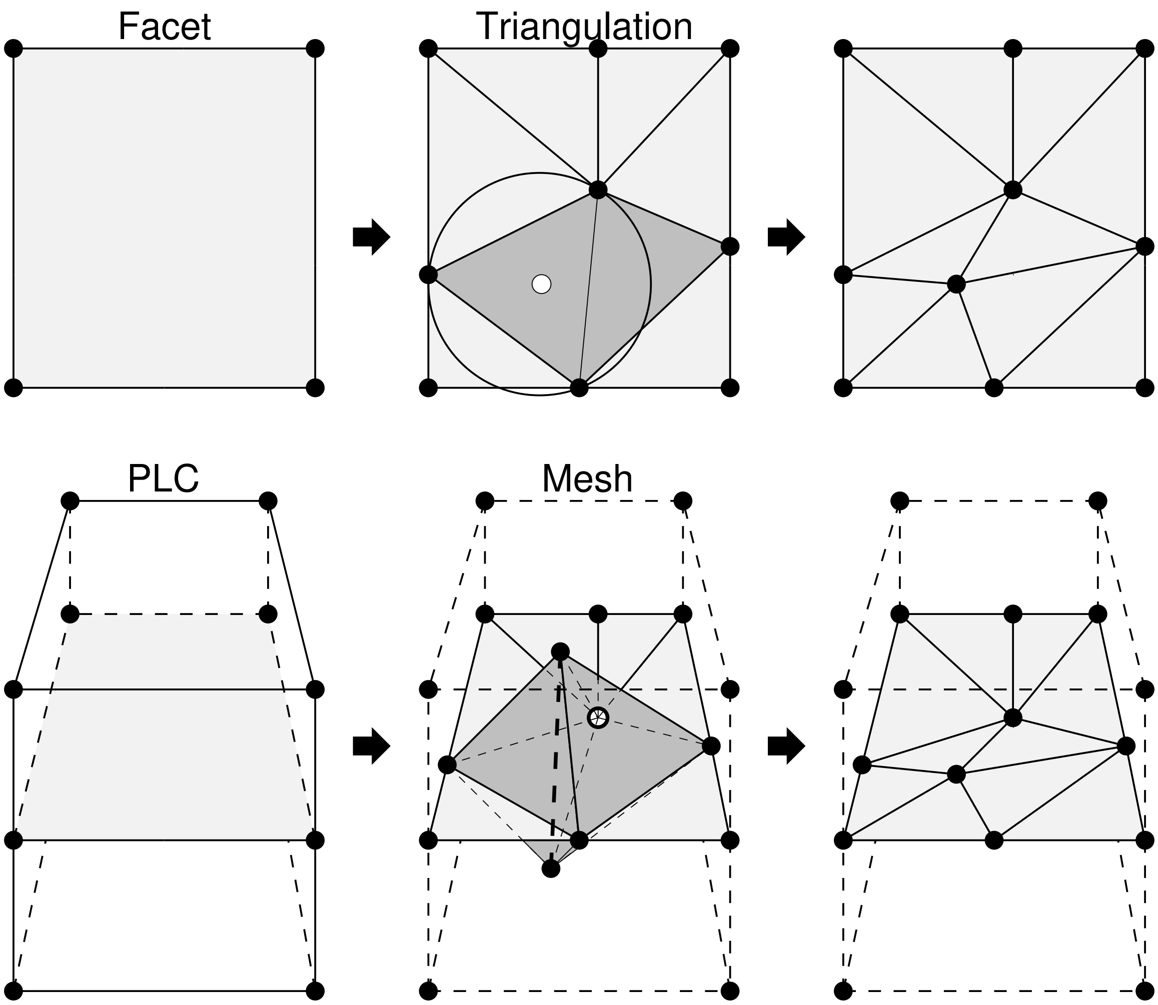


Figure 4.10: The top illustrations depict a rectangular facet and its triangulation. The bottom illustrations depict the facet's position as an interior boundary of a PLC, and its progress as it is inserted into the tetrahedralization. Most of the vertices and tetrahedra of the mesh are omitted for clarity. The facet triangulation and the tetrahedralization are maintained separately. Shaded triangular subfacets in the facet triangulation (top center) are missing from the tetrahedralization (bottom center). The bold dashed line (bottom center) represents a tetrahedralization edge that passes through the facet. Missing subfacets are forced into the mesh by inserting vertices at their circumcenters (right, top and bottom). Each of these vertices is independently inserted into both the triangulation and the tetrahedralization.

for a matching face in the tetrahedralization; if the latter is missing, insert a vertex at the circumcenter of the subfacet (subject to rejection if subsegments are encroached), as illustrated in Figure 4.10. The vertex is inserted into both the tetrahedralization and the facet triangulation. Similarly, midpoints of encroached subsegments are inserted into the tetrahedralization and into each containing facet triangulation.

In essence, Ruppert's algorithm uses the same procedure to recover segments. However, the process of forming a "one-dimensional triangulation" is so simple that it passes unnoticed.

Which vertices of the tetrahedralization need to be considered in a facet triangulation? It is a fact, albeit somewhat nonintuitive, that if a facet appears in a Delaunay tetrahedralization as a union of faces, then the triangulation of the facet is determined solely by the vertices of the tetrahedralization that lie in the plane of the facet. If a vertex lies close to a facet, but not in the same plane, it may cause a subfacet to be

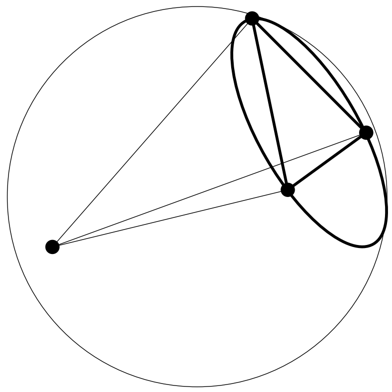


Figure 4.11: If a tetrahedron is Delaunay, the circumcircle of each of its faces is empty, because each face's circumcircle is a cross-section of the tetrahedron's circumsphere.

missing (as in Figure 4.10), but it cannot affect the shape of the triangulation if all subfacets are present. Why? Suppose a subfacet of a facet appears in the tetrahedralization. Then the subfacet must be a face of a Delaunay tetrahedron. The subfacet's circumcircle is empty, because its circumcircle is a cross-section of the tetrahedron's empty circumsphere, as illustrated in Figure 4.11. Therefore, if a facet appears in a Delaunay tetrahedralization, it appears as a Delaunay triangulation. Because the Delaunay triangulation is unique (except in nondegenerate cases), vertices that do not lie in the plane of the facet have no effect on how the facet is triangulated.

Hence, each separately maintained facet triangulation need only consider vertices lying in the plane of the facet. Furthermore, because each facet is segment-bounded, and segments are recovered (in the tetrahedralization) before facets, each facet triangulation can safely ignore vertices that lie outside the facet (even in the same plane). The requirements set forth in Section 4.1.1 ensure that all of the vertices and segments of a facet must be explicitly identified in the input PLC. The only additional vertices to be considered are those that were inserted on segments to force segments and other facets into the mesh. The algorithm maintains a list of the vertices on each segment, ready to be called upon when a facet triangulation is initially formed.

Unfortunately, if a facet triangulation is not unique because of cocircularity degeneracies, then the foregoing statement about extraplanar vertices having no effect on the triangulation does not apply. To be specific, suppose a facet triangulation has four or more cocircular vertices, which are triangulated one way, whereas the tetrahedralization contains a set of faces that triangulate the same vertices with a different (but also Delaunay) set of triangles, as illustrated in Figure 4.12. (If exact arithmetic is not used, nearly-degenerate cases may team up with floating-point roundoff error to make this circumstance more common.) An aggressive implementation might identify these cases and correct the facet triangulation so that it matches the tetrahedralization (it is not always possible to force the tetrahedralization to match the triangulation). However, inserting a new vertex at the center of the collective circumcircle is always available as a lazy alternative.

To appreciate why I should choose this rather unusual method of forcing facets into the mesh, it is worth comparing it with the most popular method [48, 96, 79]. In many tetrahedral mesh generators, facets are inserted by identifying points where the edges of the tetrahedralization intersect a missing facet, and inserting vertices at these points. The perils of so doing are illustrated in Figure 4.13. In the illustration, a vertex is inserted where a tetrahedralization edge (bold dashed line) intersects the facet. Unfortunately, the edge intersects the facet near one of the bounding segments of the facet, and the new vertex creates a feature that may be arbitrarily small. Afterward, the only alternatives are to refine the tetrahedra near the

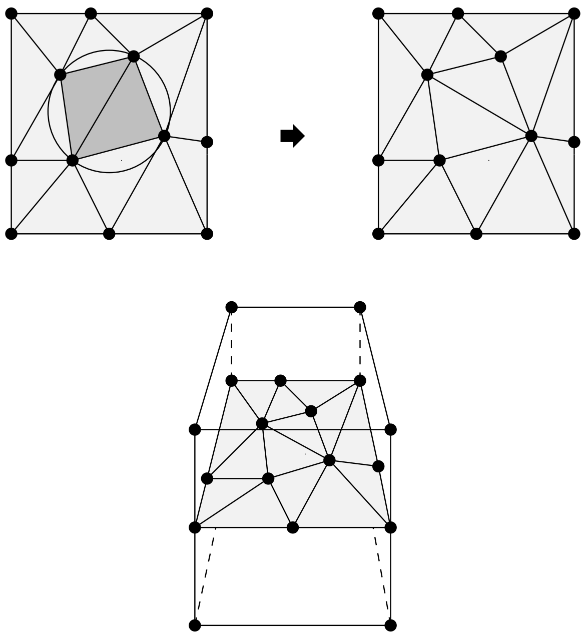


Figure 4.12: A facet triangulation and a tetrahedralization may disagree due to cocircular vertices. This occurrence may be diagnosed and fixed as shown here, or a new vertex may be inserted at the circumcenter, removing the degeneracy.

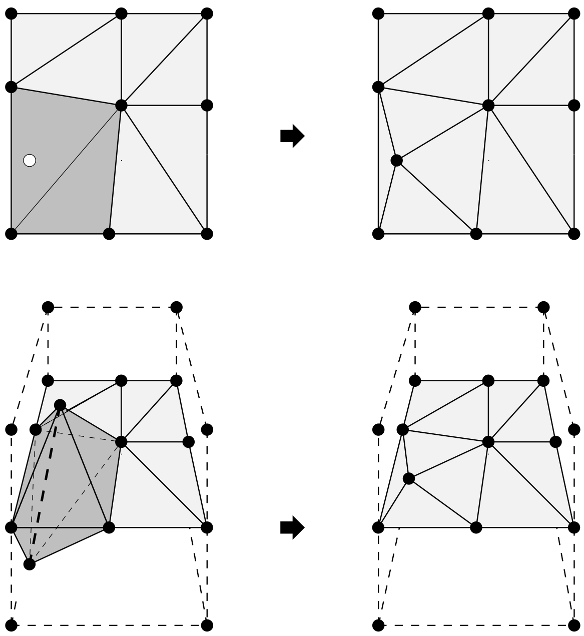


Figure 4.13: One may force a facet into a tetrahedralization by inserting vertices at the intersections of the facet with edges of the tetrahedralization, but this method might create arbitrarily small features by placing vertices close to segments.

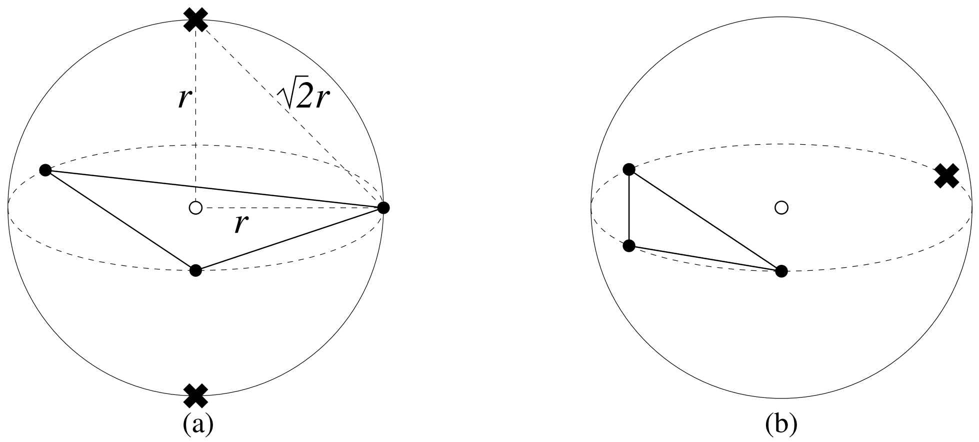


Figure 4.14: The relationship between the insertion radii of the circumcenter of an encroached subfacet and the encroaching vertex. Crosses identify the location of an encroaching vertex having maximum distance from the nearest subfacet vertex. (a) If the encroached subfacet contains its own circumcenter, the encroaching vertex is no further from the nearest vertex of the subfacet than $\sqrt{2}$ times the circumradius of the subfacet. (b) If the encroached subfacet does not contain its own circumcenter, the encroaching vertex may be further away.

new vertex to a small size, or to move or remove the vertex. Some mesh generators cope with this problem by smoothing the vertices on each facet after the facet is competely inserted.

My facet insertion method does not insert such vertices at all. A vertex considered for insertion so close to a segment is rejected, and a subsegment is split instead. This would not necessarily be true if edge-facet intersections were considered for insertion, because such an intersection may be near a vertex lying on the segment, and thus fail to encroach upon any subsegments. Subfacet circumcenters are better choices because they are far from the nearest vertices, and cannot create a new small feature without encroaching upon a subsegment.

Another advantage of my facet insertion method is that if a subfacet is missing from the mesh, there must be a vertex inside its equatorial sphere, or in a degenerate case, several vertices on its equatorial sphere. Hence, for the purposes of analysis, missing subfacets may be treated identically to facets that are present but encroached. As in the two-dimensional case, the same is true for missing subsegments.

I shall pass implementation difficulties aside to analyze the algorithm. In analysis, however, subfacets present another complication. It would be nice to prove, in the manner of Lemma 15, that whenever an encroached subfacet is split at its circumcenter, the insertion radius of the newly inserted vertex is no worse than $\sqrt{2}$ times smaller than the insertion radius of its parent. Unfortunately, this is not true for the algorithm described above.

Consider the two examples of Figure 4.14. If a subfacet that contains its own circumcenter is encroached, then the distance between the encroaching vertex and the nearest vertex of the subfacet is no more than $\sqrt{2}$ times the circumradius of the subfacet. This distance is maximized if the encroaching vertex lies at a pole of the equatorial sphere (where the *poles* are the two points of the sphere furthest from its equator), as illustrated in Figure 4.14(a). However, if a subfacet that does not contain its own circumcenter is encroached, the distance is maximized if the encroaching vertex lies on the equator, equidistant from the two vertices of the longest edge of the subfacet, as in Figure 4.14(b). Even if the encroaching vertex is well away from the equator, its distance from the nearest vertex of the subfacet can still be larger than $\sqrt{2}$ times the radius of the equatorial sphere. (I have confirmed through my implementation that such cases do arise in practice.)

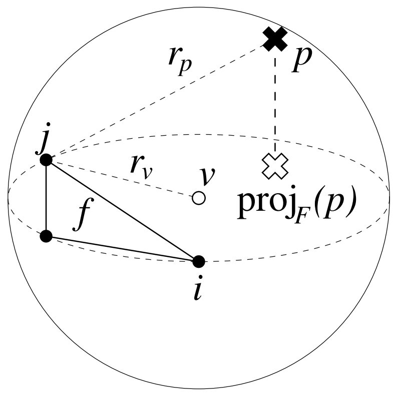


Figure 4.15: An encroached subfacet f that contains neither its own circumcenter v nor the projection of the encroaching vertex p onto the plane containing f.

Rather than settle for a looser guarantee on quality, one can make a small change to the algorithm that will yield a $\sqrt{2}$ bound. Let f be an encroached subfacet that does not contain its own circumcenter v, as illustrated in Figure 4.15. Let i and j be vertices of f, with ij the edge that separates f from v. Let p be the vertex that encroaches upon f; p will be the parent of v if the algorithm attempts to insert v.

Let F be the facet that contains f. Let $\text{proj}_F(p)$ be the orthogonal projection of p onto the plane containing F (and hence f). If $\text{proj}_F(p)$ lies on the same side of ij as f (or on ij), there is no problem; the ratio $\frac{r_p}{r_v}$ cannot be greater than $\sqrt{2}$. However, if $\text{proj}_F(p)$ lies on the same side of ij as v (as illustrated), there is no such guarantee.

In the latter case, however, one can show (with the following lemma) that there is some other subfacet g of F that is encroached by p and contains $\text{proj}_F(p)$. (There are two such subfacets if $\text{proj}_F(p)$ falls on an edge.) One can achieve the desired bound by modifying the algorithm to split g first and delay the splitting of f indefinitely.

Lemma 26 (Projection Lemma) *Let f be a subfacet of the Delaunay triangulated facet F. Suppose that f is encroached by some vertex p strictly inside the equatorial sphere of f, but p does not encroach upon any subsegment of F. Then* $\text{proj}_F(p)$ *falls within the facet F, and p encroaches upon a subfacet of F that contains* $\text{proj}_F(p)$.

Proof: First, I prove that $\text{proj}_F(p)$ falls inside F, using similar reasoning to that employed in Lemma 13. Suppose for the sake of contradiction that $\text{proj}_F(p)$ falls outside the facet F. Let c be the centroid of f; c clearly lies inside F. Because all facets are segment-bounded, the line segment connecting c to $\text{proj}_F(p)$ must intersect some subsegment s. Let $\mathcal{S}$ be the plane that contains s and is orthogonal to F, as illustrated in Figure 4.16(a).

Because f is a Delaunay subfacet of F, its circumcircle (in the plane of F) contains no vertices of F. However, its equatorial sphere may contain vertices—including p—and f might not appear in the tetrahedralization.

It is apparent that p and $\text{proj}_F(p)$ lie on the same side of $\mathcal{S}$, as the projection is defined orthogonally to F. Say that a point is *inside* $\mathcal{S}$ if it is on the same side of $\mathcal{S}$ as c, and *outside* $\mathcal{S}$ if it is on the same side as p and $\text{proj}_F(p)$. Because the circumcenter of f lies in F (Lemma 13), and the circumcircle of f cannot enclose the endpoints of s (f is Delaunay in F), the portion of f's equatorial sphere outside $\mathcal{S}$ lies entirely inside

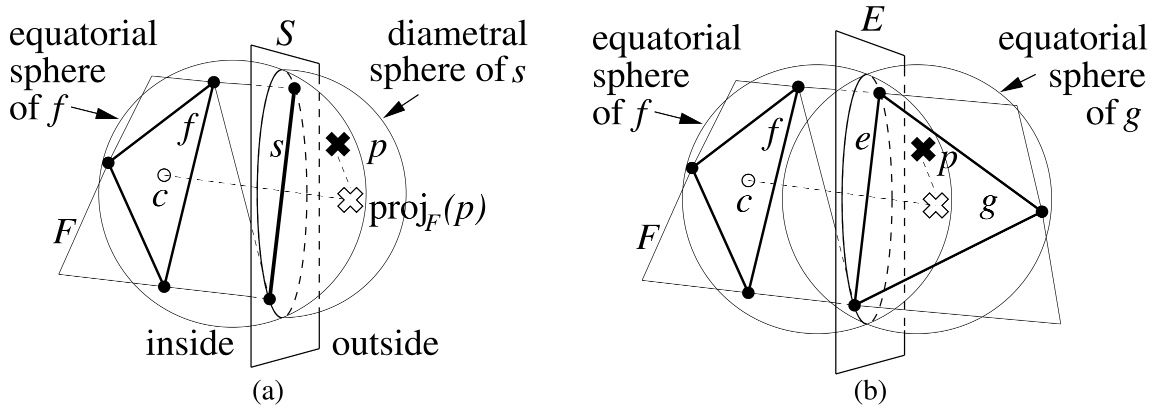


Figure 4.16: Two properties of encroached Delaunay subfacets. (a) If a vertex p encroaches upon a Delaunay subfacet f of a facet F, but its projection into the plane containing F falls outside F, then p encroaches upon some subsegment s of F as well. (b) If a vertex p encroaches upon a subfacet f of a Delaunay triangulated facet F, but does not encroach upon any subsegment of F, then p encroaches upon the subfacet(s) g of F that contains $\text{proj}_F(p)$.

the diametral sphere of s (as the figure demonstrates). Because p is strictly inside the equatorial sphere of f, p also lies strictly within the diametral sphere of s, contradicting the assumption that p encroaches upon no subsegment of F.

It follows that $\text{proj}_F(p)$ must be contained in some subfacet g of F. (The containment is not necessarily strict; $\text{proj}_F(p)$ may fall on an edge interior to F, and be contained in two subfacets.) To complete the proof of the lemma, I shall show that p encroaches upon g. If $f = g$ the result follows immediately, so assume that $f \neq g$.

Again, let c be the centroid of f. The line segment connecting c to $\text{proj}_F(p)$ must intersect some edge e of the subfacet g, as illustrated in Figure 4.16(b). Let $\mathcal{E}$ be the plane that contains e and is orthogonal to F. Say that a point is on the *g-side* if it is on the same side of $\mathcal{E}$ as g. Because the triangulation of F is Delaunay, the portion of f's equatorial sphere on the g-side is entirely enclosed by the equatorial sphere of g. The point p lies on the g-side or in $\mathcal{E}$ (because $\text{proj}_F(p)$ is contained in g), and p lies strictly within the equatorial sphere of f, so it must also lie strictly within the equatorial sphere of g, and hence encroaches upon g. ■

There is one case not covered by the Projection Lemma. If f is missing, the closest encroaching vertex v might lie precisely on the equatorial sphere of f, and also lie precisely on the diametral sphere of s, thereby failing to encroach s. In this case, however, the circumcenter of f precisely coincides with the midpoint of s. Hence, the algorithm's behavior will be no different than if s were encroached by v.

One way to interpret the Projection Lemma is to imagine that the facet F is orthogonally extended to infinity, so that each subfacet of F defines an infinitely long triangular prism (Figure 4.17). Each subfacet's equatorial sphere dominates its prism, in the sense that the sphere contains any point in the prism that lies within the equatorial sphere of any other subfacet of F. If a vertex p encroaches upon any subfacet of F, then p encroaches upon the subfacet in whose prism p is contained. If p encroaches upon some subfacet of F but is contained in none of the prisms, then p also encroaches upon some boundary subsegment of F.

In the latter case, any boundary subsegments encroached upon by p are split until none remains. The Projection Lemma guarantees that any subfacets of F encroached upon by p are eliminated in the process.

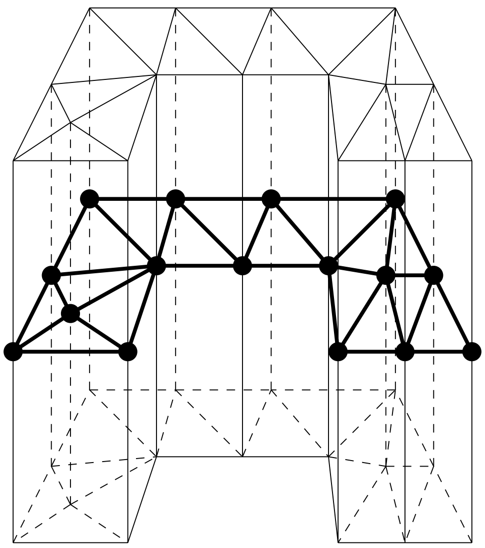


Figure 4.17: Each subfacet's equatorial sphere dominates the triangular prism defined by extending the subfacet orthogonally.

On the other hand, if the vertex p is contained in the prism of a subfacet g, and no subsegments are encroached, then splitting g is a good choice. Several new subfacets will appear, at least one of which contains $\text{proj}_F(p)$; if this subfacet is encroached, then it is split as well, and so forth until the subfacet containing $\text{proj}_F(p)$ is not encroached. The Projection Lemma guarantees that any other subfacets of F encroached upon by p (those that do not contain $\text{proj}_F(p)$) are eliminated in the process.

4.2.2 Proof of Termination

The proof of termination for the three-dimensional case is similar to that of the two-dimensional case. Assume that in the input PLC, any two incident segments are separated by an angle of 60° or greater. If a segment meets a facet at one vertex, and the projection of the segment onto the facet (using a projection direction orthogonal to the facet) intersects the interior of the facet, then the angle separating the segment from the facet must be greater than $\arccos \frac{1}{2\sqrt{2}} \doteq 69.3^\circ$. If the projection of the segment does not intersect the interior of the facet, the Projection Lemma implies that no vertex on the segment can encroach upon any subfacet of the facet without also encroaching upon a boundary segment of the facet, so the 60° separation between segments is sufficient to ensure termination.

The condition for two incident facets is more complicated. If both facets are convex and meet at a segment, then it is sufficient for the facets to be separated by a dihedral angle of 90° or greater. In general, the two facets must satisfy the following *projection condition*.

For any vertex v where two facets F and G meet, let $\text{vis}_v(F)$ be the facet region of F visible from v, and define $\text{vis}_v(G)$ likewise. By definition, $\text{vis}_v(F)$ and $\text{vis}_v(G)$ are incident facet regions. No point of the orthogonal projection of $\text{vis}_v(F)$ onto G may fall in the interior of $\text{vis}_v(G)$. Another way to word it is to say that $\text{vis}_v(F)$ is disjoint from the interior of the prism formed by projecting $\text{vis}_v(G)$ orthogonally (recall Figure 4.17). Formally, for any point v on $F \cap G$, the projection condition requires that $\text{proj}_G(\text{vis}_v(F)) \cap \text{interior}(\text{vis}_v(G)) = \emptyset$. This condition is equivalent to the converse condition, in which F and G trade places.

The payoff of this restriction is that, by Lemma 26, no vertex in $\mathrm{vis}_v(F)$ may encroach upon a subfacet contained entirely in $\mathrm{vis}_v(G)$ without also encroaching upon a subsegment of G or a subfacet of G not entirely in $\mathrm{vis}_v(G)$. The converse is also true. The purpose of this condition is to prevent any vertex from splitting a subfacet in an incident facet region. Otherwise, subfacets might be split to arbitrarily small sizes through mutual encroachment in regions arbitrarily close to v.

The projection condition just defined is always satisfied by two facets separated by a dihedral angle of exactly 90°. It is also satisfied by facets separated by a dihedral angle greater than 90° if the facets meet each other only at segments whose endpoints are not reflex vertices of either facet. (Recall Figure 4.6, which depicts two facets that are separated by a dihedral angle greater than 90° but fail the projection condition because v is a reflex vertex of F.)

The following lemma, which extends Lemma 15 to three dimensions, is true for the Delaunay refinement algorithm described heretofore, if one is careful never to split an encroached subfacet f that contains neither its own circumcenter nor the projection $\mathrm{proj}_f(p)$ of the encroaching vertex p. (Even more liberally, an implementation can easily measure the insertion radii of the parent and its potential progeny, and may split f if the latter is no less than $\frac{1}{\sqrt{2}}$ times the former.)

The insertion radius is defined as before: r_v is the length of the shortest edge incident to v immediately after v is inserted. The parent of a vertex is defined as before, with the following amendments. If v is the circumcenter of a skinny tetrahedron, its parent $p(v)$ is the most recently inserted endpoint of the shortest edge of that tetrahedron. If v is the circumcenter of an encroached subfacet, its parent is the encroaching vertex closest to v (whether that vertex is inserted or rejected).

Lemma 27 *Let v be a vertex of the mesh, and let $p = p(v)$ be its parent, if one exists. Then either $r_v \geq \mathrm{lfs}(v)$, or $r_v \geq Cr_p$, where*

- *$C = B$ if v is the circumcenter of a skinny tetrahedron,*
- *$C = \frac{1}{\sqrt{2}}$ if v is the midpoint of an encroached subsegment or the circumcenter of an encroached subfacet, and p is rejected for insertion,*
- *$C = \frac{1}{2\cos\alpha}$ if v and p lie on incident segments separated by an angle of α, or if v lies in the interior of a facet incident to a segment containing p at an angle α, where $45^\circ \leq \alpha < 90^\circ$,*
- *$C = \sin\alpha$ if v and p are positioned as in the previous case, but with $\alpha \leq 45^\circ$, and*
- *$C = \frac{\sin\alpha}{\sqrt{2}}$ if v and p lie within facet regions that are incident at a segment S, if $\mathrm{proj}_S(p)$ lies within S (this case is included only to demonstrate why it should be avoided),*

or v and p lie within incident facet regions that do not meet at a segment S for which $\mathrm{proj}_S(p)$ lies within S. For this case (which should also be avoided), I offer no analysis.

If one thinks of a subsegment's midpoint as its circumcenter, one can see this lemma as having a hierarchical form: if the circumcenter of a simplex encroaches upon a lower-dimensional simplex, then the circumcenter is rejected for insertion, and the circumcenter of the lower-dimensional simplex has an insertion radius up to $\sqrt{2}$ times smaller than that of the rejected circumcenter. If the circumcenter of a simplex encroaches upon another simplex having equal or higher dimension, then the circumcenter of the latter has an insertion radius that depends in part on the angle between the two simplices. I expect this framework to generalize to higher dimensions, and will elaborate in Section 4.7.

Proof of Lemma 27: If v is an input vertex, the circumcenter of a tetrahedron, or the midpoint of an encroached subsegment, then it may be treated exactly as in Lemma 15. One case from that lemma is worth briefly revisiting to show that nothing essential has changed.

If v is inserted at the midpoint of an encroached subsegment s, and its parent $p = p(v)$ is a circumcenter (of a tetrahedron or subfacet) that was considered for insertion but rejected because it encroaches upon s, then p lies strictly inside the diametral sphere of s. Because the circumsphere/circumcircle centered at p contains no vertices, and in particular does not contain the endpoints of s, $r_v > \frac{r_p}{\sqrt{2}}$; see Figure 4.18(a) for an example where the relation is nearly equality. Note that the change from circles (in the two-dimensional analysis) to spheres makes little difference. Perhaps the clearest way to see this is to observe that if one takes a two-dimensional cross-section that passes through s and p, the cross-section is indistinguishable from the two-dimensional case. (The same argument can be made for the case where p and v lie on incident segments.)

Only the circumstance where v is the circumcenter of an encroached subfacet f remains. Let F be the facet that contains f. There are four cases to consider.

- If the parent p is an input vertex, or lies in a segment or facet region not incident to any facet region containing v, then $\mathrm{lfs}(v) \leq r_v$.
- If p is a tetrahedron circumcenter that was considered for insertion but rejected because it encroaches upon f, then p lies strictly inside the equatorial sphere of f. Because the tetrahedralization is Delaunay, the circumsphere centered at p contains no vertices, so its radius is limited by the nearest vertex of f. By assumption, f contains either its own circumcenter or $\mathrm{proj}_f(p)$. In the former case, $\frac{r_v}{r_p}$ is minimized when p is at the pole of f's equatorial sphere, as illustrated in Figure 4.18(b). In the latter case, $\frac{r_v}{r_p}$ is minimized when $\mathrm{proj}_f(p)$ is the intersection of f's longest edge and the bisector of its second-longest edge, as illustrated in Figure 4.18(c). In either case, $r_v > \frac{r_p}{\sqrt{2}}$.
- If p was inserted on a segment that is incident to F at one vertex a, separated by an angle of $\alpha \geq 45^\circ$ (Figure 4.18(d)), the shared vertex a cannot lie inside the equatorial sphere of f because the facet F is Delaunay. (This is true even if f does not appear in the tetrahedralization.) Because the segment and facet are separated by an angle of α, the angle $\angle pav$ is at least α. Because f is encroached upon by p, p lies inside its equatorial sphere. (If f is not present in the tetrahedralization, p might lie on its equatorial sphere in a degenerate case.) Analogously to the case of two incident segments (see Lemma 15), if $\alpha \geq 45^\circ$, then $\frac{r_v}{r_p}$ is minimized when the radius of the equatorial sphere is $r_v = |vp|$, and p lies on the sphere. (If the equatorial sphere were any smaller, it could not contain p.) Therefore, $r_v \geq \frac{r_p}{2\cos\alpha}$. If $\alpha \leq 45^\circ$, then $\frac{r_v}{r_p}$ is minimized when $v = \mathrm{proj}_f(p)$; therefore, $r_v \geq r_p \sin\alpha$.
- If p and v lie within two facet regions that are incident at a segment S, the analysis is less optimistic than the previous case because there is no vertex that serves the function that a serves in Figure 4.18(d). As Figure 4.19 shows, the equatorial sphere centered at v is not constrained by the segment S (although it is constrained by the vertices on S).

  To find the minimum possible value of $\frac{r_v}{r_p}$, consider the point $\mathrm{proj}_S(p)$, which (by assumption) lies on some subsegment s of the segment S. Let d be the distance from p to $\mathrm{proj}_S(p)$. Because p does not encroach upon s (otherwise, the algorithm would split s in preference to f), the smallest possible value of $\frac{d}{r_p}$ is $\frac{1}{\sqrt{2}}$.

  In the absence of any constraints, $r_v = |vp|$ is minimized when $v = \mathrm{proj}_F(p)$, with the line segment vp orthogonal to F. With this choice of v, the angle $\angle p[proj_S(p)]v$ is precisely the dihedral angle α

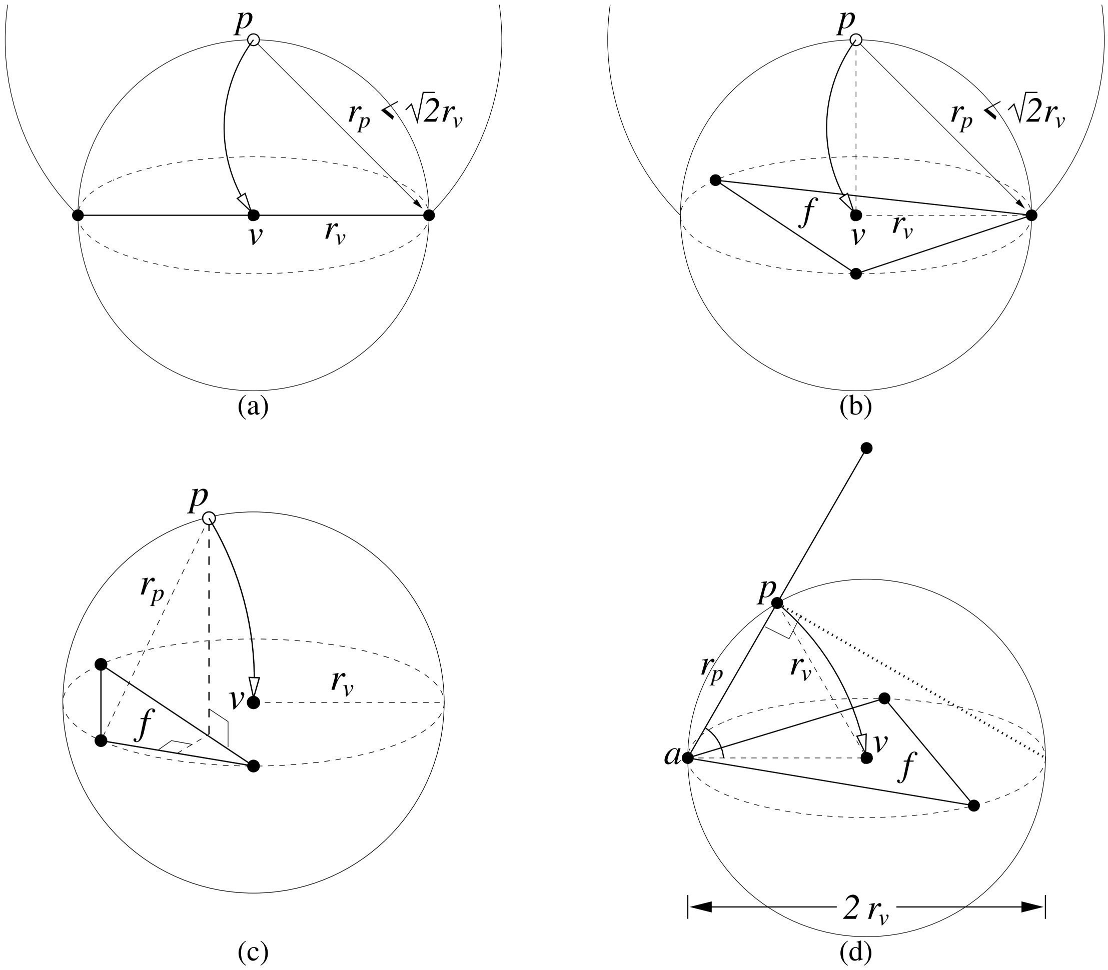


Figure 4.18: The relationship between the insertion radii of a child and its parent. (a) When a subsegment is encroached upon by a circumcenter, the child's insertion radius may be arbitrarily close to a factor of $\sqrt{2}$ smaller than its parent's. (b) When a subfacet that contains its own circumcenter is encroached upon by the circumcenter of a skinny tetrahedron, the child's insertion radius may be arbitrarily close to a factor of $\sqrt{2}$ smaller than its parent's. (c) A bound better than $\sqrt{2}$ applies to a subfacet that does not contain its own circumcenter, but does contain the projection of the encroaching vertex. (d) When a subfacet is encroached upon by the midpoint of a subsegment, and the corresponding facet and segment are incident at one vertex, the analysis differs little from the case of two incident segments.

separating the two facets. Hence, the minimum value of $\frac{r_v}{d}$ is $\sin\alpha$. Combining this with the result of the previous paragraph, $\frac{r_v}{r_p} \geq \frac{\sin\alpha}{\sqrt{2}}$. Note that, unlike the case where a segment meets a facet or another segment, the worst case is not achieved with p on the equatorial sphere of f for any angle less than $90°$. ■

Lemma 27 provides the information one needs to ensure that Delaunay refinement will terminate. As with the two dimensional algorithms, the key is to prevent any cycle wherein mesh vertices beget chains of descendants with ever-smaller insertion radii (Figure 4.20).

Mesh vertices are divided into four classes: input vertices (which cannot contribute to cycles), vertices inserted into segments, vertices inserted into facet interiors, and free vertices inserted at circumcenters of

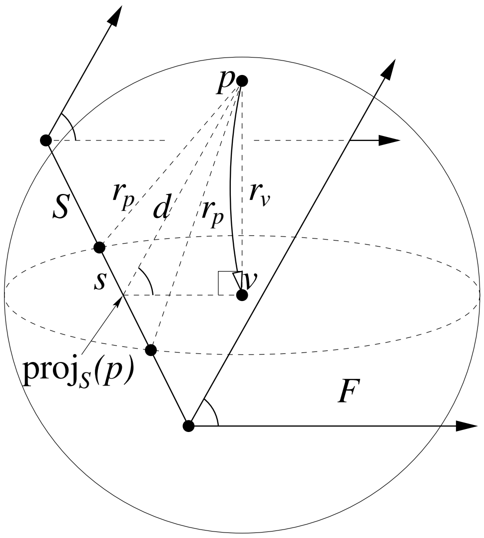


Figure 4.19: When two incident facet regions are separated by an angle less than $90°$, and a subfacet of one is encroached upon by a vertex in the interior of the other, the child's insertion radius r_v may be smaller than its parent's insertion radius r_p. Hence, a $90°$ minimum separation is imposed.

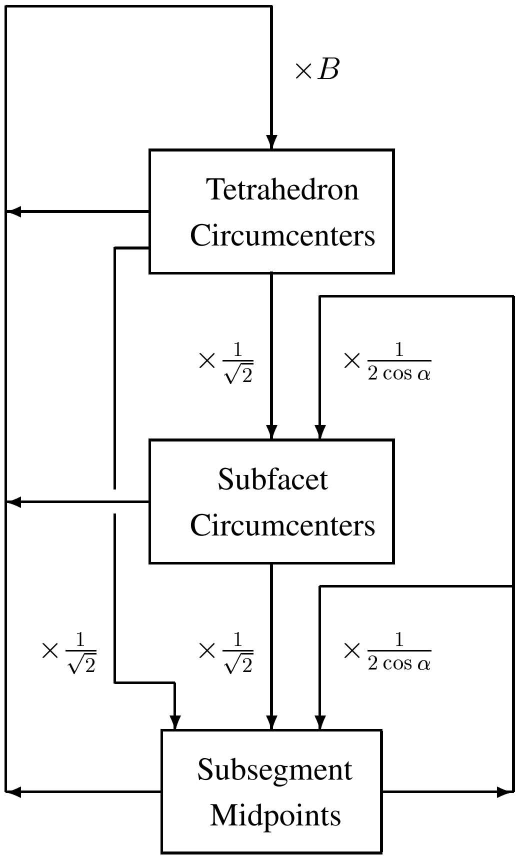


Figure 4.20: Dataflow diagram illustrating the worst-case relation between a vertex's insertion radius and the insertion radii of the children it begets. If no cycle has a product smaller than one, the three dimensional Delaunay refinement algorithm will terminate.

tetrahedra. As we have seen, free vertices can father facet vertices whose insertion radii are smaller by a factor of $\sqrt{2}$, and these facet vertices in turn can father segment vertices whose insertion radii are smaller by another factor of $\sqrt{2}$. Hence, to avoid spiralling into the abyss, it is important that segment vertices can only father free vertices whose insertion radii are at least twice as large. This constraint fixes the best guaranteed circumradius-to-shortest edge ratio at $B = 2$.

The need to prevent diminishing cycles also engenders the requirement that incident segments be separated by angles of $60°$ or more, just as it did in the two-dimensional case. A segment incident to a facet must be separated by an angle of at least $\arccos \frac{1}{2\sqrt{2}} \doteq 69.3°$ so that if a vertex on the segment encroaches upon a subfacet of the facet, the child that results will have an insertion radius at least $\sqrt{2}$ larger than that of its parent. (Recall from Lemma 27 that $r_v \geq \frac{r_p}{2\cos\alpha}$.)

Theorem 28 *Let* $\mathrm{lfs}_{\min}$ *be the shortest distance between two nonincident entities (vertices, segments, or facets) of the input PLC. Suppose that any two incident segments are separated by an angle of at least* $60°$*, any two incident facet regions satisfy the projection condition, and any segment incident to a facet at one vertex is separated from it by an angle of at least* $\arccos \frac{1}{2\sqrt{2}}$ *or satisfies the projection condition.*

Suppose a tetrahedron is considered to be skinny if its circumradius-to-shortest edge ratio is larger than $B \geq 2$*. The three-dimensional Delaunay refinement algorithm described above will terminate, with no tetrahedralization edge shorter than* $\mathrm{lfs}_{\min}$.

Proof: Suppose for the sake of contradiction that the algorithm introduces one or more edges shorter than $\mathrm{lfs}_{\min}$ into the mesh. Let e be the first such edge introduced. Clearly, the endpoints of e cannot both be input vertices, nor can they lie on nonincident segments or facet regions. Let v be the most recently inserted endpoint of e.

By assumption, no edge shorter than $\mathrm{lfs}_{\min}$ existed before v was inserted. Hence, for any ancestor a of v, $r_a \geq \mathrm{lfs}_{\min}$. Let $p = p(v)$ be the parent of v, let $g = p(p)$ be the grandparent of v (if one exists), and let $h = p(g)$ be the great-grandparent of v (if one exists). Because of the projection condition, v and p cannot lie on incident facet regions. Consider the following cases.

- If v is the circumcenter of a skinny tetrahedron, then by Lemma 27, $r_v \geq Br_p \geq 2r_p$.
- If v is the midpoint of an encroached subsegment or the circumcenter of an encroached subfacet, and p is the circumcenter of a skinny tetrahedron, then by Lemma 27, $r_v \geq \frac{1}{\sqrt{2}}r_p \geq \frac{B}{\sqrt{2}}r_g \geq \sqrt{2}r_g$.
- If v is the midpoint of an encroached subsegment, p is the circumcenter of an encroached subfacet, and g is the circumcenter of a skinny tetrahedron, then by Lemma 27, $r_v \geq \frac{1}{\sqrt{2}}r_p \geq \frac{1}{2}r_g \geq \frac{B}{2}r_h \geq r_h$.
- If v and p lie on incident segments, then by Lemma 27, $r_v \geq \frac{r_p}{2\cos\alpha}$. Because $\alpha \geq 60°$, $r_v \geq r_p$.
- If v is the circumcenter of an encroached subfacet and p lies on a segment incident (at a single vertex) to the facet containing v, then by Lemma 27, $r_v \geq \frac{r_p}{2\cos\alpha}$. Because $\alpha \geq \arccos\frac{1}{2\sqrt{2}}$, $r_v \geq \sqrt{2}r_p$.
- If v is the midpoint of an encroached subsegment, p is the (rejected) circumcenter of an encroached subfacet, and g lies on a segment incident (at a single vertex) to the facet containing p, then by Lemma 27, $r_v \geq \frac{1}{\sqrt{2}}r_p \geq \frac{1}{2\sqrt{2}\cos\alpha}r_g$. Because $\alpha \geq \arccos\frac{1}{2\sqrt{2}}$, $r_v \geq r_g$.

- If v is the midpoint of an encroached subsegment, and p has been inserted on a nonincident segment or facet region, then by the definition of parent, pv is the shortest edge introduced by the insertion of v. Because p and v lie on nonincident entities, p and v are separated by a distance of at least $\text{lfs}_{\min}$, contradicting the assumption that e has length less than $\text{lfs}_{\min}$.

In the first six cases, $r_p \geq r_a$ for some ancestor a of p in the mesh. It follows that $r_p \geq \text{lfs}_{\min}$, contradicting the assumption that e has length less than $\text{lfs}_{\min}$. Because no edge shorter than $\text{lfs}_{\min}$ is ever introduced, the algorithm must terminate. ■

4.2.3 Proof of Good Grading

As with the two-dimensional algorithm, a stronger termination proof is possible, showing that all edges in the final mesh are proportional in length to the local feature sizes of their endpoints, and thus ensuring nicely graded meshes. The proof makes use of Lemma 17, which generalizes unchanged to three or more dimensions. Recall that the lemma states that if $r_v \geq Cr_p$ for some vertex v with parent p, then their lfs-weighted vertex densities are related by the formula $D_v = \frac{\text{lfs}(v)}{r_v} \leq 1 + \frac{D_p}{C}$.

Lemma 29 *Suppose the quality bound B is strictly larger than 2, and all angles between segments and facets satisfy the conditions listed in Theorem 28, with all inequalities replaced by strict inequalities.*

There exist fixed constants $D_T \geq 1$, $D_F \geq 1$, and $D_S \geq 1$ such that, for any vertex v inserted (or rejected) at the circumcenter of a skinny tetrahedron, $D_v \leq D_T$; for any vertex v inserted (or rejected) at the circumcenter of an encroached subfacet, $D_v \leq D_F$; and for any vertex v inserted at the midpoint of an encroached subsegment, $D_v \leq D_S$. Hence, the insertion radius of every vertex is proportional to its local feature size.

Proof: Consider any non-input vertex v with parent $p = p(v)$. If p is an input vertex, then $D_p = \frac{\text{lfs}(p)}{r_p} \leq 1$. Otherwise, assume for the sake of induction that the lemma is true for p. Hence, $D_p \leq \max\{D_T, D_F, D_S\}$.

First, suppose v is inserted or considered for insertion at the circumcenter of a skinny tetrahedron. By Lemma 27, $r_v \geq Br_p$. Therefore, by Lemma 17, $D_v \leq 1 + \frac{\max\{D_T, D_F, D_S\}}{B}$. It follows that one can prove that $D_v \leq D_T$ if D_T is chosen so that

$$D_T \geq 1 + \frac{\max\{D_T, D_F, D_S\}}{B}. \tag{4.1}$$

Second, suppose v is inserted or considered for insertion at the circumcenter of a subfacet f. If its parent p is an input vertex or lies on a segment or facet region not incident to the facet region containing v, then $\text{lfs}(v) \leq r_v$, and the theorem holds. If p is the circumcenter of a skinny tetrahedron (rejected for insertion because it encroaches upon f), $r_v \geq \frac{r_p}{\sqrt{2}}$ by Lemma 27, so by Lemma 17, $D_v \leq 1 + \sqrt{2}D_T$.

Alternatively, if p lies on a segment incident to the facet containing f, then $r_v \geq \frac{r_p}{2\cos\alpha}$ by Lemma 27, and thus by Lemma 17, $D_v \leq 1 + 2D_S\cos\alpha$. It follows that one can prove that $D_v \leq D_F$ if D_F is chosen so that

$$\begin{aligned} D_F &\geq 1 + \sqrt{2}D_T, \qquad \text{and} & (4.2)\\ D_F &\geq 1 + 2D_S\cos\alpha. & (4.3) \end{aligned}$$

Third, suppose v is inserted at the midpoint of a subsegment s. If its parent p is an input vertex or lies on a segment or facet region not incident to the segment containing s, then $\mathrm{lfs}(v) \leq r_v$, and the theorem holds. If p is the circumcenter of a skinny tetrahedron or encroached subfacet (rejected for insertion because it encroaches upon s), $r_v \geq \frac{r_p}{\sqrt{2}}$ by Lemma 27, so by Lemma 17, $D_v \leq 1 + \sqrt{2} \max\{D_T, D_F\}$.

Alternatively, if p and v lie on incident segments, then $r_v \geq \frac{r_p}{2\cos\alpha}$ by Lemma 27, and thus by Lemma 17, $D_v \leq 1 + 2D_S \cos\alpha$. It follows that one can prove that $D_v \leq D_S$ if D_S is chosen so that

$$D_S \geq 1 + \sqrt{2}\max\{D_T, D_F\} \quad \text{and} \tag{4.4}$$

$$D_S \geq 1 + 2D_S \cos\alpha. \tag{4.5}$$

If the quality bound B is strictly larger than 2, conditions 4.1, 4.2, and 4.4 are simultaneously satisfied by choosing

$$D_T = \frac{B + 1 + \sqrt{2}}{B - 2}, \qquad D_F = \frac{(1+\sqrt{2})B + \sqrt{2}}{B - 2}, \qquad D_S = \frac{(3+\sqrt{2})B}{B-2}.$$

If the smallest angle α_{FS} between any facet and any segment is strictly greater than $\arccos \frac{1}{2\sqrt{2}} \doteq 69.3^\circ$, conditions 4.3 and 4.4 may be satisfied by choosing

$$D_F = \frac{1 + 2\cos\alpha_{FS}}{1 - 2\sqrt{2}\cos\alpha_{FS}}, \qquad D_S = \frac{1+\sqrt{2}}{1 - 2\sqrt{2}\cos\alpha_{FS}},$$

if these values exceed those specified above. In this case, adjust D_T upward if necessary according to condition 4.1.

If the smallest angle α_{SS} between two segments is strictly greater than 60°, condition 4.5 may be satisfied by choosing

$$D_S = \frac{1}{1 - 2\cos\alpha_{SS}},$$

if this value exceeds those specified above. In this case, adjust D_T and D_F upward if necessary according to conditions 4.1 and 4.2. ■

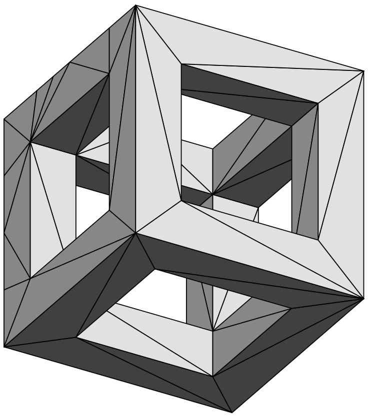
Note that as B approaches 2, α_{SS} approaches 60°, or α_{FS} approaches $\arccos \frac{1}{2\sqrt{2}}$, the values of D_T, D_F, and D_S approach infinity.

Theorem 30 *For any vertex v of the output mesh, the distance to its nearest neighbor is at least $\frac{\mathrm{lfs}(v)}{D_S+1}$.*

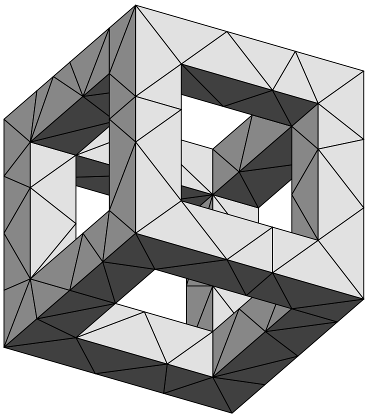
Proof: Inequality 4.4 indicates that D_S is larger than D_T and D_F. The remainder of the proof is identical to that of Theorem 19. ■

To provide an example, suppose $B = 2.5$ and the input PLC has no acute angles. Then $D_T \doteq 9.8$, $D_F \doteq 14.9$, and $D_S \doteq 22.1$. Hence, the spacing of vertices is at worst about 23 times smaller than the local feature size.

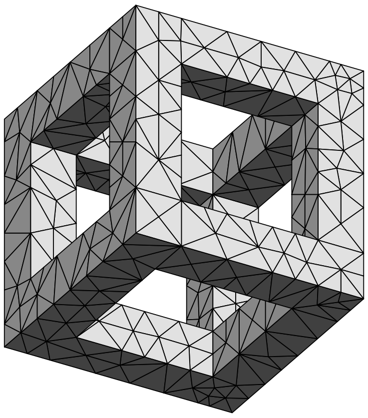
As Figure 4.21 shows, the algorithm performs much better in practice. The upper left mesh is the initial tetrahedralization after all segments and facets are inserted and unwanted tetrahedra have been removed from the holes. (Some subsegments remain encroached because during the segment and facet insertion stages,



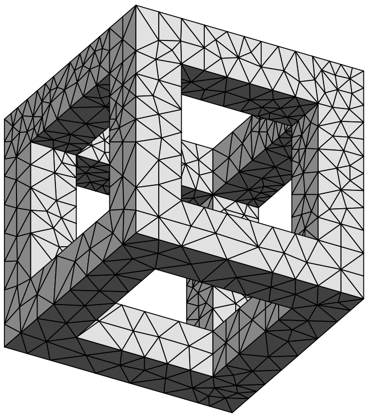
Initial tetrahedralization after segment and facet insertion. 54 vertices, 114 tetrahedra.



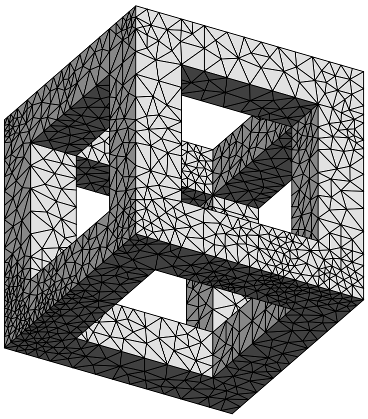
$B = 1.4$, $\theta_{\min} = 5.42°$, $\theta_{\max} = 171.87°$, $h_{\min} = 1.25$, 110 vertices, 211 tetrahedra.



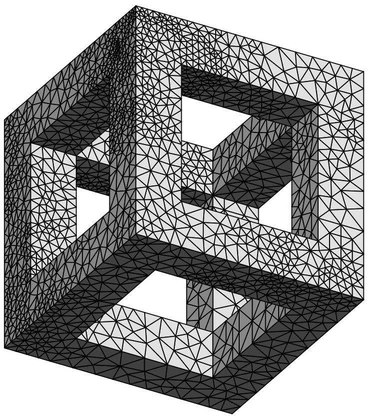
$B = 1.2$, $\theta_{\min} = 2.01°$, $\theta_{\max} = 175.89°$, $h_{\min} = 0.5$, 468 vertices, 1206 tetrahedra.



$B = 1.1$, $\theta_{\min} = 0.69°$, $\theta_{\max} = 178.83°$, $h_{\min} = 0.36$, 1135 vertices, 3752 tetrahedra.



$B = 1.05$, $\theta_{\min} = 1.01°$, $\theta_{\max} = 178.21°$, $h_{\min} = 0.24$, 2997 vertices, 11717 tetrahedra.



$B = 1.04$, $\theta_{\min} = 1.01°$, $\theta_{\max} = 178.21°$, $h_{\min} = 0.13$, 5884 vertices, 25575 tetrahedra.

Figure 4.21: Several meshes of a $10 \times 10 \times 10$ PLC generated with different bounds (B) on quality. Below each mesh is listed the smallest dihedral angle $\theta_{\min}$, the largest dihedral angle $\theta_{\max}$, and the shortest edge length $h_{\min}$. The algorithm does not terminate on this PLC for a bound of $B = 1.03$.

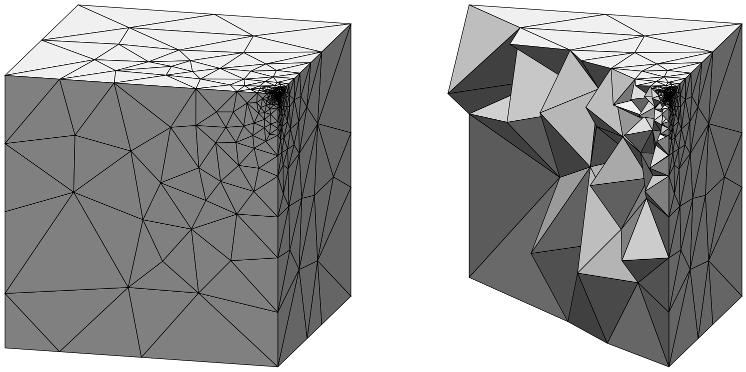


Figure 4.22: At left, a mesh of a truncated cube. At right, a cross-section through a diagonal of the top face.

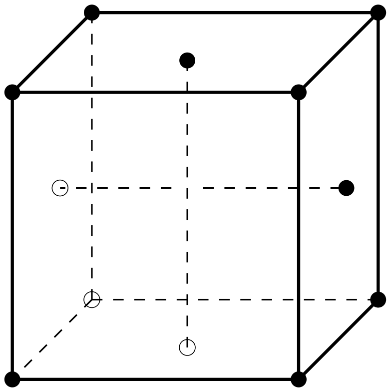


Figure 4.23: A counterexample demonstrating that the three-dimensional Delaunay refinement algorithm is not size-optimal.

my implementation only splits an encroached subsegment if it is missing or it is part of the facet currently being inserted.) After all encroached subsegments and subfacets have been split (upper right), the largest circumradius-to-shortest edge ratio is already less than 1.4, which is much better than the proven bound of 2. The shortest edge length is 1.25, and $\mathrm{lfs}_{\min} = 1$, so the spectre of edge lengths 23 times smaller than the local feature size has not materialized. As the quality bound B decreases, the number of elements in the final mesh increases gracefully until B drops below 1.05. At $B = 1.03$, the algorithm fails to terminate.

Not surprisingly, the object depicted is slightly harder to tetrahedralize if the unwanted tetrahedra are not removed from the holes before refining. At $B = 1.06$, the algorithm fails to terminate.

Figure 4.22 offers a demonstration of the grading of a tetrahedralization generated by Delaunay refinement. A cube has been truncated at one corner, cutting off a portion whose width is one-millionth that of the cube. Although this mesh satisfies a quality bound of $B = 1.2$, good grading is very much in evidence.

Unfortunately, the proof of good grading does not yield a size-optimality proof as it did in the two-dimensional case. Gary Miller and Dafna Talmor (private communication) have pointed out the counterexample depicted in Figure 4.23. Inside this PLC, two segments pass very close to each other without intersecting. The PLC might reasonably be tetrahedralized with a few dozen tetrahedra having bounded circumradius-to-shortest edge ratios, if these tetrahedra include a sliver tetrahedron whose four vertices are the endpoints of the two internal segments. However, the best my Delaunay refinement algorithm can promise is to fill the region with tetrahedra whose edge lengths are proportional to the distance between the two segments. Because this distance may be arbitrarily small, the algorithm is not size-optimal.

4.3 Delaunay Refinement with Equatorial Lenses

In this section, I improve the Delaunay refinement algorithm by replacing equatorial spheres with equatorial lenses, which are similar to the Chew-inspired diametral lenses introduced in Section 3.4. This modification ensures that the algorithm terminates and produces well-graded meshes for any bound on circumradius-to-shortest edge ratio greater than $\frac{2\sqrt{2}}{\sqrt{3}} \doteq 1.63$, which is a significant improvement over the bound of two given for Delaunay refinement with equatorial spheres.

4.3.1 Description of the Algorithm

My lens-based algorithm begins with the Delaunay tetrahedralization of a facet-bounded PLC, and performs Delaunay refinement with locked subsegments and subfacets. A constrained Delaunay tetrahedralization would be ideal if one could be generated, but this is not generally possible, so the algorithm attempts to recover all missing segments first, and then all missing facets, locking each subsegment and subfacet as soon as it appears.

Just as in the three-dimensional generalization of Ruppert's algorithm, subsegments are protected by diametral spheres. Missing subfacets are protected by equatorial spheres, but any subfacet that is present in the tetrahedralization is protected only by an equatorial lens, illustrated in Figure 4.24. The equatorial lens of a subfacet f is the intersection of two balls whose centers lie on each other's boundaries, and whose boundaries intersect at the circumcircle of f. If r_f is the circumradius of f, the defining balls have radius $2r_f/\sqrt{3}$, and their centers lie on the line orthogonal to f through its circumcenter, a distance of $r_f/\sqrt{3}$ from f. An equatorial lens is the revolution of a diametral lens about its shorter axis. Unlike in the two-dimensional case, it does not seem to be possible to achieve a result analogous to Lemma 21 for a lens angle smaller than 30°, so I shall use a lens angle of 30° throughout.

The subfacet f is considered for splitting if there is a vertex, or an attempt to insert a vertex, inside or on the boundary of its equatorial lens, unless another subfacet obstructs the line of sight between the encroaching vertex and the circumcenter of f. (Throughout this section, visibility is deemed to be obstructed only by interposing subfacets, and only if they are present in the mesh, and thus locked.) As usual, if the circumcenter of f encroaches upon any subsegments, the encroached subsegments are split instead. However, if f is split, all free vertices (but not input vertices or vertices that lie on segments or facets) that lie in the interior of the equatorial sphere of f and are visible from the circumcenter of f are deleted. Then, a new vertex is inserted at the circumcenter of f, as illustrated in Figure 4.25. The Delaunay property is maintained throughout, except that locked subfacets are not flipped. Hence, the final mesh is not guaranteed to be truly Delaunay, but is effectively constrained Delaunay.

As in two dimensions, the advantage of lenses is that when a vertex v with parent p is inserted at the center of a lens, its insertion radius is bounded by the inequality $r_v \geq r_p \cos 30^\circ = \frac{\sqrt{3}}{2} r_p$. As in the three-dimensional generalization of Ruppert's algorithm, the bound is only ensured if the algorithm refuses to split any subfacet that contains neither its own circumcenter nor the orthogonal projection of the encroaching vertex.

Here a new problem arises. Suppose a facet F contains a subfacet f whose equatorial lens is encroached upon by a vertex p, but f contains neither its own circumcenter nor $\mathrm{proj}_F(p)$. Let g be the subfacet of F that

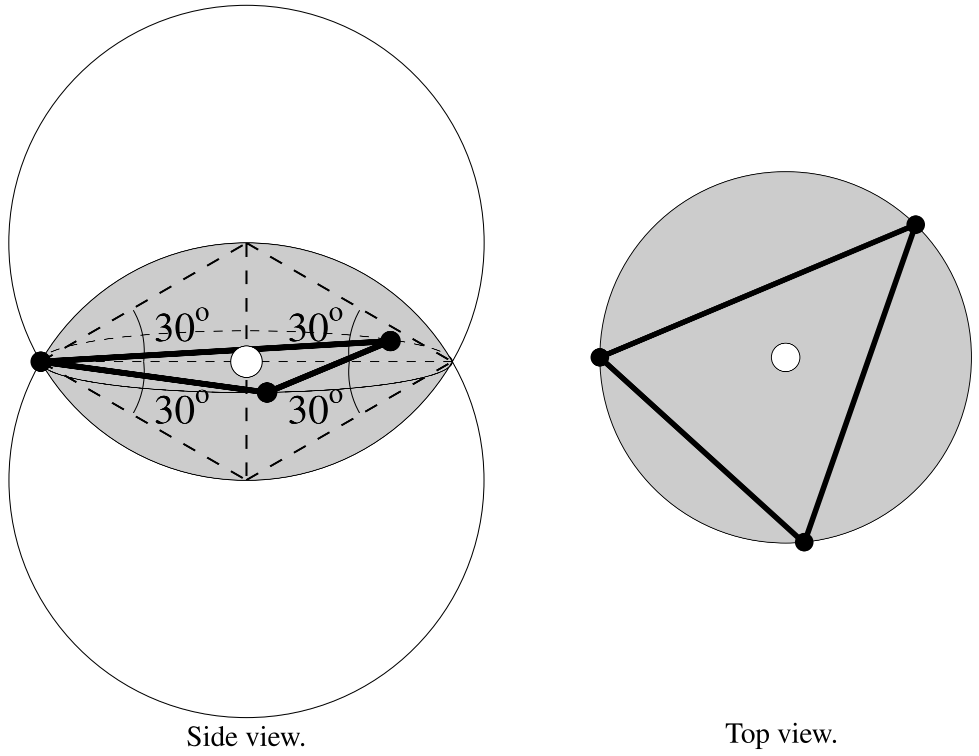


Figure 4.24: The equatorial lens (shaded) of a triangular subfacet is the intersection of two identical balls whose boundaries meet at the subfacet's circumcircle. Each ball's center lies on the surface of the other ball.

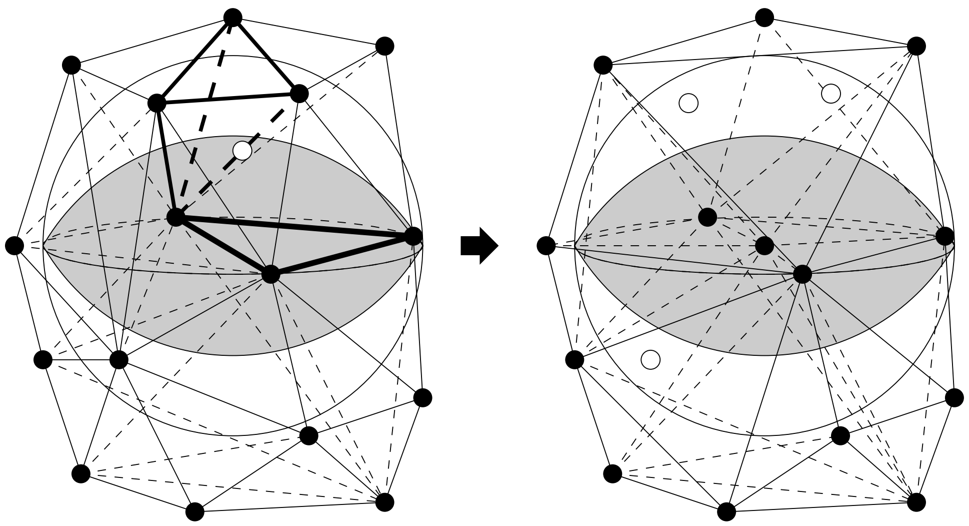


Figure 4.25: At left, the circumcenter of the bold tetrahedron encroaches upon the equatorial lens of the bold subfacet. At right, all vertices in the subfacet's equatorial sphere have been deleted, and a new vertex has been inserted at the subfacet's circumcenter.

contains $\text{proj}_F(p)$. One would like to have a guarantee, similar to the Projection Lemma, that the equatorial lens of g is encroached upon by p. Unfortunately, there is no such guarantee.

Fortunately, the next section will show that the inequality $r_v \geq \frac{\sqrt{3}}{2} r_p$ holds if v is the circumcenter of g, even if g is not encroached. Hence, the rule governing subfacet encroachment remains unchanged: if an encroached subfacet contains neither its own circumcenter nor the orthogonal projection of the encroaching vertex, then the subfacet containing the orthogonal projection is split instead.

The guarantees offered by the Projection Lemma are just as useful for understanding Delaunay refinement with equatorial lenses as they were with equatorial spheres. Suppose p encroaches upon the equatorial lens of f. Because the equatorial lens of f is contained in the equatorial sphere of f, p encroaches upon f's equatorial sphere as well. Because the modified algorithm still uses diametral spheres to protect subsegments, the Projection Lemma implies that either p encroaches upon a subsegment, or $\text{proj}_F(p)$ lies within F. In the former case, the encroached subsegment is split instead of f; in the latter case, the Projection Lemma guarantees that some subfacet of F contains $\text{proj}_F(p)$. Furthermore, the Projection Lemma guarantees that if one repeatedly splits the subfacet containing $\text{proj}_F(p)$, F will eventually contain no subfacets whose equatorial spheres are encroached, and thus also guarantees that F will eventually contain no subfacets whose equatorial lenses are encroached.

4.3.2 Proof of Termination and Good Grading

The three-dimensional Delaunay refinement algorithm with equatorial lenses, like Chew's algorithm, requires for its analysis that the insertion radius r_c of the circumcenter c of a skinny tetrahedron t be redefined to be the radius of t's circumsphere. A vertex w may lie in t's circumsphere, but only if there is some locked subfacet f separating w from t. Either c lies on the same side of f as t, and thus never interacts with w, or c lies on the same side of f as w, but is not inserted because it encroaches upon f. Either way, c does not participate in an edge shorter than r_c. Does the notion of separation become ambiguous near the boundaries of a facet? No, because facets are segment-bounded, and all subsegments are protected by diametral spheres.

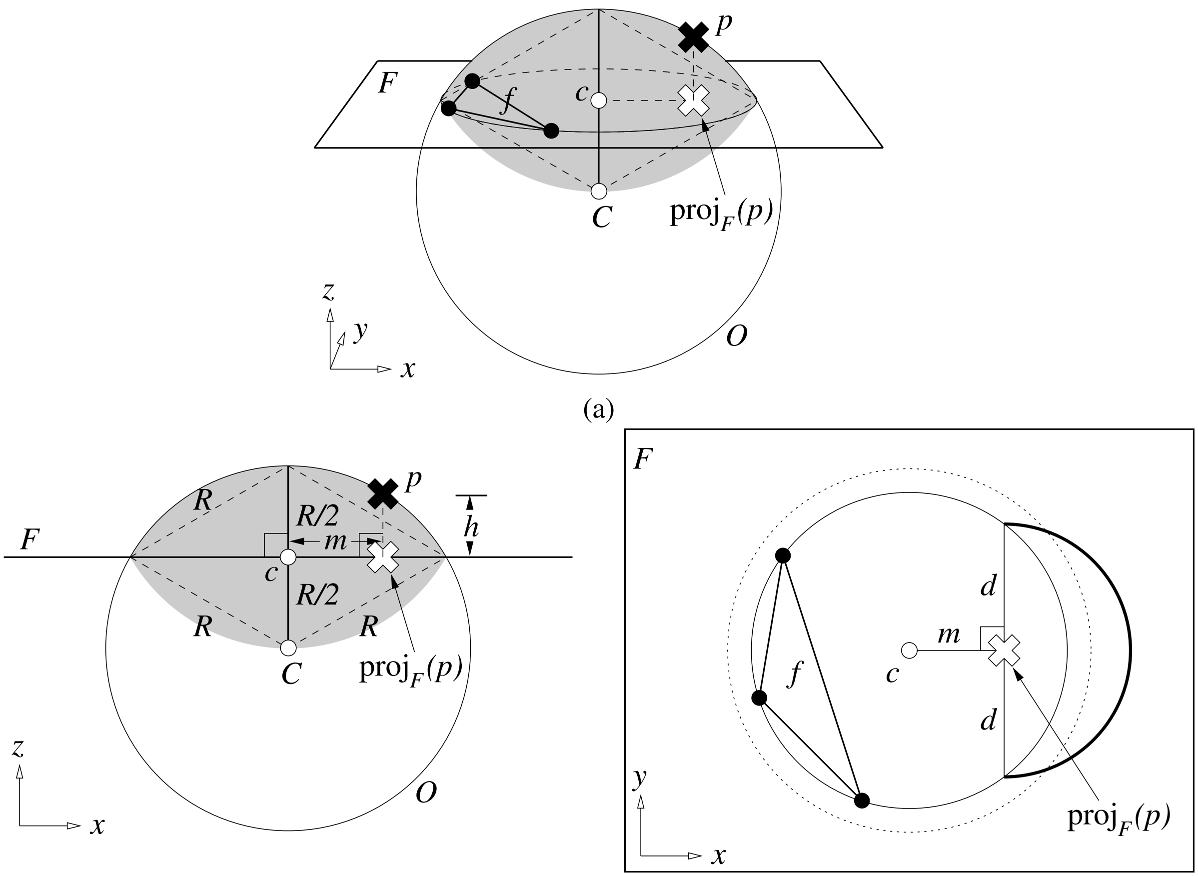
When a subfacet f is encroached, but no subsegment is encroached, the algorithm may choose to split the subfacet g that contains $\text{proj}_F(p)$. The following lemma shows that this choice produces a new vertex whose insertion radius is not much smaller than that of its parent.

Lemma 31 *Let f be a subfacet of a facet F. Let p be a tetrahedron circumcenter that encroaches upon the equatorial lens of f, and whose projection* $\text{proj}_F(p)$ *falls in some subfacet g of F (where g may or may not be f). Suppose that all vertices in the equatorial sphere of g are deleted (except those not visible from the circumcenter of g), and a vertex v is inserted at the circumcenter of g. Then $r_v \geq \frac{\sqrt{3}}{2} r_p$.*

Proof: Because all vertices visible from v are deleted from the equatorial sphere of g, r_v is equal to the radius of that equatorial sphere. (Vertices not visible from v cannot affect v's insertion radius, because an edge cannot connect them to v.)

Without loss of generality, define a coordinate system oriented so that F lies in the x-y plane, $\text{proj}_F(p)$ has the same y-coordinate as the circumcenter c of f (for instance, both might lie on the x-axis), and p is above F, as illustrated in Figure 4.26.

Let O be the lower of the two balls that define the equatorial lens of f. Let C be the center of O, and let R be the radius of O. The line segment Cc is aligned with the z-axis and has length $\frac{R}{2}$. The line segment



(b) Side view. (c) Top view.

Figure 4.26: Three views of an encroached equatorial lens.

$c[\text{proj}_F(p)]$ is aligned with the x-axis; let m be its length. The line segment $p[\text{proj}_F(p)]$ is orthogonal to F, and thus aligned with the z-axis; let h (for "height") be its length, as Figure 4.26(b) illustrates.

Draw a chord of the circumcircle of f whose midpoint is $\text{proj}_F(p)$. As Figure 4.26(c) shows, the chord is orthogonal to the line segment $c[\text{proj}_F(p)]$, and thus aligned with the y-axis. Let $2d$ be the length of the chord, so that $\text{proj}_F(p)$ bisects the chord into two line segments of length d.

The significance of d is that it is a lower bound on r_v, where v is the circumcenter of the subfacet g. Why? Recall that g contains $\text{proj}_F(p)$. However, because F is Delaunay, none of the vertices of g can lie inside the circumcircle of f. The circumcircle of g must be large enough that it can satisfy both these constraints; the smallest possible such circumcircle is outlined in bold in Figure 4.26(c), and has radius d.

As Figure 4.26(b) makes apparent, the x-coordinate of p differs from that of C by m, and their z-coordinates differ by $\frac{R}{2} + h$. Because p encroaches upon the equatorial lens of f, p lies inside or on the boundary of O. Hence, by Pythagoras' Law,

$$m^2 + \left(\frac{R}{2} + h\right)^2 \leq R^2.$$

Expanding gives

$$m^2 + \left(\frac{R}{2}\right)^2 + Rh + h^2 \leq R^2. \tag{4.6}$$

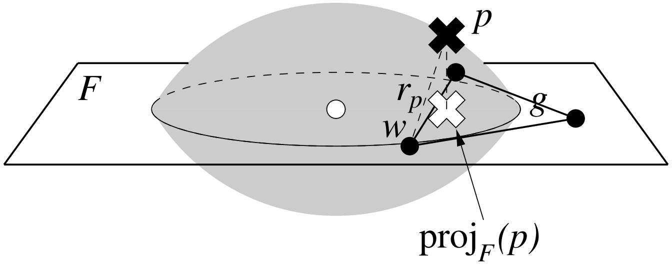


Figure 4.27: Because p lies in the diametral lens, its height above the plane cannot be more than $\frac{1}{\sqrt{3}}$ times the circumradius of the triangle that contains $\text{proj}_F(p)$.

Each endpoint of the chord in Figure 4.26(c) lies on the boundary of O, so by Pythagoras' Law,

$$m^2 + d^2 + \left(\frac{R}{2}\right)^2 = R^2. \tag{4.7}$$

Subtracting Equation 4.7 from Inequality 4.6 gives

$$Rh + h^2 \leq d^2.$$

As Figure 4.26(b) shows, it is always true that $h \leq \frac{R}{2}$, because the equatorial lens of f does not extend further than $\frac{R}{2}$ from the plane containing F. Recall that d is a lower bound on r_v. Combining these bounds,

$$r_v^2 \geq d^2 \geq Rh + h^2 \geq 3h^2.$$

Let w be the vertex of g nearest $\text{proj}_F(p)$, as illustrated in Figure 4.27. Because $\text{proj}_F(p)$ lies within g and the circumradius of g is r_v, the length $|w[\text{proj}_F(p)]|$ is at most r_v. The vertex p is the circumcenter of a constrained Delaunay tetrahedron; because the circumsphere of this tetrahedron contains no vertex of F, r_p can be no greater than the distance $|pw|$. This distance can be computed by Pythagoras' Law, because $w[\text{proj}_F(p)]$ is orthogonal to $p[\text{proj}_F(p)]$ (the former lies in F, whereas the latter is orthogonal to F). Hence,

$$\begin{aligned} r_p^2 &\leq |w[\text{proj}_F(p)]|^2 + |p[\text{proj}_F(p)]|^2 \\ &\leq r_v^2 + h^2 \\ &\leq \frac{4}{3} r_v^2. \end{aligned}$$

Therefore, $$r_p \leq \frac{2}{\sqrt{3}} r_v,$$

and the result follows. ■

Lemma 31 is only applicable if all the vertices in the equatorial sphere of g that are visible from v are deleted. If some such vertex u is not deleted, then u is an input vertex or lies on a subsegment or subfacet. The vertices u and v cannot lie on incident features, because of the 60° minimum angle between input entities and the projection condition between input facets. (The edge of a lens rises from the plane at an angle of 60°.) Hence, the local feature size at v is at most $|uv|$, and $r_v \geq \text{lfs}(v)$, as Lemma 27 indicates. Choose the input vertex, segment vertex, or facet vertex closest to v to be the parent of v.

Vertices are only deleted when a subfacet is split, and vertices are never deleted from subfacets. Theorem 28 sets a lower bound on the length of each facet edge, so only a finite number of subfacet splits can occur. After the last subfacet split, no more vertex deletions occur, so termination is ensured by Theorem 28.

Theorem 32 *Suppose the quality bound B is strictly larger than $\frac{2\sqrt{2}}{\sqrt{3}}$, and all angles between segments and facets satisfy the conditions listed in Theorem 28, with all inequalities replaced by strict inequalities.*

There exist fixed constants $D_T \geq 1$, $D_F \geq 1$, and $D_S \geq 1$ such that, for any vertex v inserted (or rejected) at the circumcenter of a skinny tetrahedron, $D_v \leq D_T$; for any vertex v inserted (or rejected) at the circumcenter of an encroached subfacet, $D_v \leq D_F$; and for any vertex v inserted at the midpoint of an encroached subsegment, $D_v \leq D_S$.

Proof: Essentially the same as the proof of Lemma 29, except that Lemma 31 makes it possible to replace Condition 4.2 with

$$D_F \geq 1 + \frac{2}{\sqrt{3}} D_T \tag{4.8}$$

If the quality bound B is strictly larger than $\frac{2\sqrt{2}}{3}$, Conditions 4.1, 4.4, and 4.8 are simultaneously satisfied by choosing

$$D_T = \frac{\sqrt{3}B + \sqrt{3} + \sqrt{6}}{\sqrt{3}B - 2\sqrt{2}}, \qquad D_F = \frac{(2+\sqrt{3})B + 2}{\sqrt{3}B - 2\sqrt{2}}, \qquad D_S = \frac{(\sqrt{3} + \sqrt{6} + 2\sqrt{2})B}{\sqrt{3}B - 2\sqrt{2}}.$$

D_T, D_F, and D_S must also satisfy the conditions specified in Lemma 29 regarding the angles between segments and facets and between segments. If $B > \frac{2\sqrt{2}}{3}$, $\alpha_{FS} > \arccos \frac{1}{2\sqrt{2}}$, and $\alpha_{SS} > 60^\circ$, there are values of D_T, D_F, and D_S that satisfy the theorem. ■

To compare equatorial lenses with equatorial spheres, consider again tetrahedralizing a PLC with no acute angles, applying a quality bound of $B = 2.5$. Using equatorial lenses, $D_T \doteq 5.7$, $D_F \doteq 7.5$, and $D_S \doteq 11.7$. Compare with the corresponding values 9.8, 14.9, and 22.1 derived for equatorial spheres at the end of Section 4.2.3. Hence, the worst-case vertex spacing for Delaunay refinement with equatorial lenses is a factor of 1.8 better than with equatorial spheres. Because the number of tetrahedra is inversely proportional to the cube of vertex spacing, equatorial lenses improve the worst-case cardinality of the mesh by a factor of about six.

Equatorial lenses have another advantage. With some effort, it is possible to show that if the dihedral angle separating two incident facet regions is 60° or more, a vertex in one facet region cannot encroach upon a subfacet of the other without encroaching upon a subsegment of the other. (Details are omitted.) However, because equatorial spheres must be used for missing subfacets, this fact does not lead to as nice a bound on edge lengths as one might hope. Given a PLC whose incident facets are separated by at least 60°, it is possible to show that Delaunay refinement will terminate if facets are recovered one at a time, but as Lemma 27 indicates, each successively recovered facet may have smaller edges than the previously inserted facet if it is encroached upon by vertices of the previous facet. The length of the shortest edge in the final mesh may be exponentially small, where the exponent is proportional to the number of facets. Section 5.3.1 suggests a facet recovery method that might partly ameliorate this problem.

4.3.3 Diametral Lemons?

The success of diametral lenses in two dimensions, and equatorial lenses in three, naturally leads one to ask whether it might be possible to further improve the quality bound by replacing diametral spheres with some smaller structure. The obvious choice, depicted in Figure 4.28, is the revolution of a diametral lens about its longer axis, yielding a pointed prolate spheroid I call a *diametral lemon*.

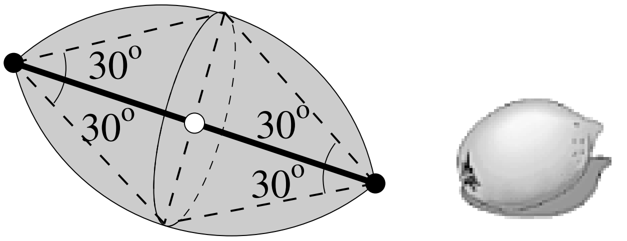


Figure 4.28: The diametral lemon of a subsegment is the revolution of a diametral lens about the subsegment.

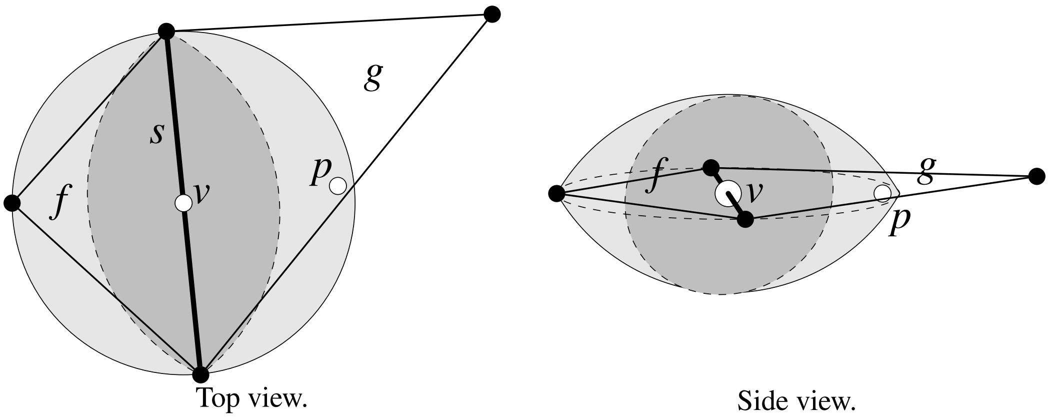


Figure 4.29: An example demonstrating that diametral lemons don't seem to improve the quality bound. In this illustration, the diametral lemon of s is contained in the equatorial lens of f.

Alas, diametral lemons are lemons indeed, because they do not seem to improve the worst-case ratio of $\sqrt{2}$ between the insertion radii of a facet vertex and a subsegment vertex it spawns. An example that demonstrates this failure is illustrated in Figure 4.29. A subfacet f meets another subfacet g at a subsegment s. The circumcenter of f coincides with the midpoint of s. The equatorial lens of f extends beyond the diametral lemon of s, and oddly, f can be encroached upon by a vertex that lies outside the facet containing f, but does not encroach upon s.

Suppose that the subfacet g is encroached upon by the circumcenter of some skinny tetrahedron, and g's circumcenter p is considered for insertion. If p encroaches upon f, f is considered for splitting. However, the circumcenter of f encroaches upon s, so s is split at its midpoint v. But neither p nor the apex of f lie in the diametral lemon of s, or particularly close to v; in the worst case, the insertion radius of v might be $\sqrt{2}$ smaller than that of either p or the apex of f.

Could we simply insert p, decline to split s, and leave the equatorial lens of f encroached? Unfortunately, f and p might together form a skinny tetrahedron, which must be removed, and splitting s may be the best way to accomplish the removal. There is no guarantee that the circumcenter of this tetrahedron is near f or s, and the usual analysis techniques do not seem to apply.

Diametral lemons have another fundamental problem. One purpose of any protective region, be it a sphere, a lens, or a lemon, is to handle the case where a skinny tetrahedron cannot be eliminated by inserting a vertex at its circumcenter, because a locked subsegment or subfacet prevents the tetrahedron from being eliminated. It is this requirement that dictates the 30° angle that defines the shape of an equatorial lens.

Consider Figure 4.30. At left, there appears a tetrahedron t whose vertices lie on the illustrated sphere, which is the circumsphere of t. Though none of the vertices of t lies in or on the diametral lemon of the

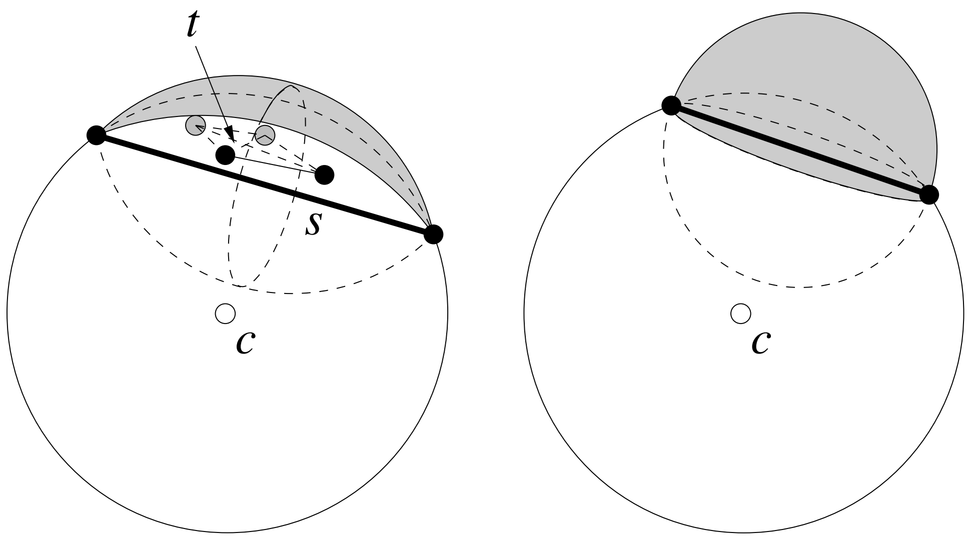


Figure 4.30: A diametral lemon fails to ensure that a locked subsegment will not stand between a tetrahedron and its circumcenter, whereas a diametral sphere succeeds.

subsegment s, several edges of t pass through the lemon, above the segment s. The circumcenter c of t lies outside the lemon as well, below the segment s. If a vertex is inserted at c, t will not be eliminated because s is locked and stands between t and its circumcenter.

At right, the lemon has been replaced with a diametral sphere. An equator has been drawn on the diametral sphere, oriented so that it will appear circular when viewed from the circumcenter c depicted. If the diametral sphere of s is empty, the vertices that lie on any empty circumsphere centered at c cannot lie above this equator. Hence, any Delaunay tetrahedron whose circumcenter is c lies below the subsegment, and the subsegment will not prevent the tetrahedron from being eliminated if a vertex is inserted at c. The diametral sphere appears to be the smallest protecting shape that can make this guarantee.

4.4 Improvements

The improvements to two-dimensional Delaunay refinement described in Section 3.5 apply in three dimensions as well. They are briefly revisited here.

4.4.1 Improving the Quality Bound in the Interior of the Mesh

Any of the following three strategies may be used to improve the quality of most of the tetrahedra of the mesh without jeopardizing the termination guarantee.

- Use a quality bound of $B = 1$ for tetrahedra that are not in contact with facet or segment interiors, a quality bound of $B = \sqrt{2}$ (for equatorial spheres) or $B = \frac{2}{\sqrt{3}}$ (for equatorial lenses) for any tetrahedron that is not in contact with a segment interior but has a vertex that lies in the interior of a facet, and a quality bound of $B = 2$ (for equatorial spheres) or $B = \frac{2\sqrt{2}}{\sqrt{3}}$ (for equatorial lenses) for any tetrahedron having a vertex that lies in the interior of a segment. The flow diagram for this strategy (with equatorial lenses) appears as Figure 4.31.

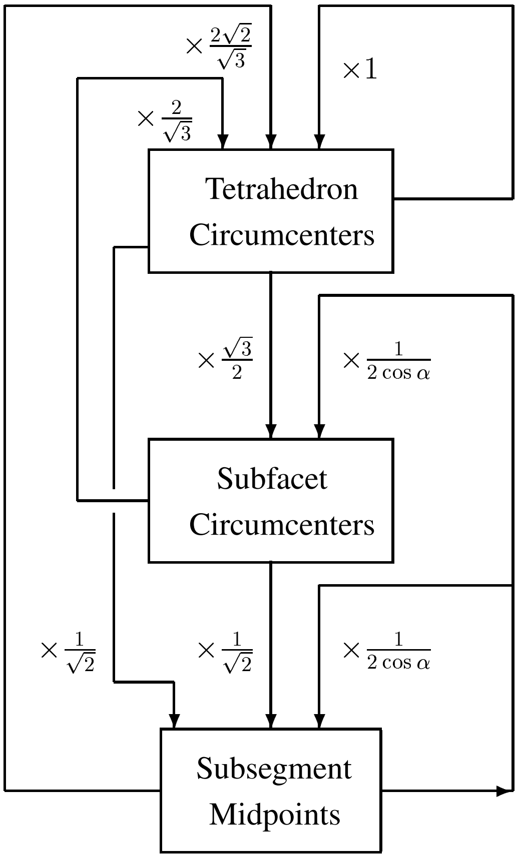


Figure 4.31: Dataflow diagram for three-dimensional Delaunay refinement with equatorial lenses and improved quality away from the boundaries.

- Attempt to insert the circumcenter of any tetrahedron whose circumradius-to-shortest edge ratio is larger than one. If any subsegments would be encroached, the circumcenter is rejected as usual, but the encroached subsegments are split only if the triangle's circumradius-to-shortest edge ratio is greater than $\sqrt{2}$. If any subfacets would be encroached, they are split only if the triangle's circumradius-to-shortest edge ratio is greater than 2 (for equatorial spheres) or $\frac{2\sqrt{2}}{\sqrt{3}}$ (for equatorial lenses).

- Attempt to insert the circumcenter of any tetrahedron whose circumradius-to-shortest edge ratio is larger than one. If any subsegments or subfacets would be encroached, the circumcenter is rejected as usual. Each encroached subsegment is checked to determine the insertion radius of the new vertex that might be inserted at its midpoint. Each encroached subfacet is checked to determine whether its circumcenter would encroach upon any subsegments, and if so, what the insertion radius of the new vertices at their midpoints would be. If a subfacet's circumcenter does not encroach upon any subsegments, the insertion radius of the subfacet's circumcenter is determined. The only midpoints and circumcenters inserted are those whose insertion radii are at least as large as the length of the shortest edge of the skinny tetrahedron.

As in the two-dimensional case, the second and third strategies tend to result in a denser spacing of vertices in the interior of the mesh than the first strategy. Also as in the two-dimensional case, good grading is maintained if the quality bound B_I in the interior of the mesh is greater than one. Then Equation 4.1 is accompanied by the equation

$$D_T \geq \frac{B_I}{B_I - 1},$$

which is familiar from Section 3.5.1.

Unlike in the two-dimensional case, this improvement is not rendered unnecessary by range-restricted segment splitting (discussed below). The two improvements combined offer even better bounds in the interior of the mesh. It is possible to apply a quality bound of $B = 1$ to tetrahedra that are not in contact with facet or segment interiors, and a quality bound of $B = \sqrt{2}$ (for equatorial spheres) or $B = \frac{2}{\sqrt{3}}$ (for equatorial lenses) to all tetrahedra.

4.4.2 Range-Restricted Segment Splitting

As in two dimensions, the quality bound of tetrahedra may be improved by range-restricted segment splitting, at the cost of sacrificing good grading in theory, if not in practice. Termination is proven below for a bound of $B = \sqrt{2}$ (if equatorial spheres are used) or $B = \frac{2}{\sqrt{3}}$ (if equatorial lenses are used). Furthermore, the constraint on the angle separating a segment from a facet may be relaxed from 69.3° to 60°.

In three dimensions, the illegal range must have a geometric width of $\sqrt{2}$ whether one uses equatorial spheres or equatorial lenses, because diametral spheres are always used. Hence, subsegments are restricted to the legal range $c2^x$, where $c \in (1, \sqrt{2}]$ and x is an integer. Segments of illegal length are split unevenly as described in Section 3.5.2.

To prove that the procedure terminates, I require a slightly different definition of *insertion radius floor* than I used for the two-dimensional proof. If v is an input vertex or lies on a subsegment or subfacet, then its insertion radius floor r'_v is still defined to be the largest power of two strictly less than its insertion radius r_v. However, if v is a free vertex inserted or rejected for insertion at the circumcenter of a skinny tetrahedron, then r'_v is defined to be the largest power of two strictly less than $\frac{r_v}{\sqrt{2}}$ (for equatorial spheres) or $\frac{\sqrt{3}}{2} r_v$ (for equatorial lenses). This change in the definition accounts for the case where a tetrahedron circumcenter encroaches upon a subfacet, and unavoidably engenders a child with smaller insertion radius.

Lemma 33 *Let* $\mathrm{lfs}_{\min}$ *be the shortest distance between two nonincident entities (vertices, segments, or facet regions) of the input PLC. Suppose that any two incident segments are separated by an angle of at least* 60°*, any two incident facet regions satisfy the projection condition, and any segment incident to a facet at one vertex is separated from it by an angle of at least* 60° *or satisfies the projection condition.*

Suppose that a triangle is considered to be skinny if its circumradius-to-shortest edge ratio is larger than $B \geq \sqrt{2}$ *if equatorial spheres are used, or* $B \geq \frac{2}{\sqrt{3}}$ *if equatorial lenses are used. Let* v *be a vertex of the mesh, and let* $p = p(v)$ *be its parent, if one exists. Then either* $r'_v \geq \mathrm{lfs}_{\min}/6$*, or* $r'_v \geq r'_p$*.*

Proof: If v is an input vertex, or if v lies on a segment or facet and its parent p is an input vertex or lies on a nonincident segment or facet region, then $\mathrm{lfs}_{\min} \leq \mathrm{lfs}(v) \leq r_v \leq 2r'_v$, and the theorem holds.

If v is inserted at the circumcenter of a skinny tetrahedron, then by Lemma 15, $r_v \geq Br_p > Br'_p$. Recall that r'_v is the largest power of two strictly less than $\frac{r_v}{\sqrt{2}}$ (for equatorial spheres) or $\frac{\sqrt{3}}{2} r_v$ (for equatorial lenses). Because B is chosen to cancel out the coefficient that biases r'_v, it follows that $r'_v \geq r'_p$.

If v is inserted at the circumcenter of an encroached subfacet f, the case where p is an input vertex or lies on a nonincident feature has been considered above, and p cannot lie on an incident facet, so there are two cases left.

- If p lies on an incident segment separated from the facet containing f by an angle α, where $60^\circ \leq \alpha < 90^\circ$, then by Lemma 27, $r_v \geq \frac{r_p}{2\cos\alpha} \geq r_p$. Therefore, $r'_v \geq r'_p$.
- If p is the circumcenter of a skinny tetrahedron, rejected for insertion because it encroaches upon f, then by Lemma 27, $r_v \geq \frac{r_p}{\sqrt{2}}$ if equatorial spheres are used, or by Lemma 31, $r_v \geq \frac{\sqrt{3}}{2} r_p$ if equatorial lenses are used. It follows that $r'_v \geq r'_p$, because the coefficient that biases the insertion radius floor of p is so chosen.

If v is inserted at the midpoint of an encroached subsegment, then the analysis presented in Lemma 23 for the two-dimensional case applies without change. ■

Theorem 34 *Suppose that the conditions on the quality bound and the angles between input entities specified in Lemma 33 hold. The Delaunay refinement algorithms described in this chapter, augmented with range-restricted segment splitting, will terminate with no tetrahedralization edge shorter than* $\mathrm{lfs}_{\min}/6$.

Proof: By Lemma 33, the insertion radius floor r'_v of every vertex v is either greater than or equal to $\mathrm{lfs}_{\min}/6$, or greater than or equal to the insertion radius floor of some preexisting vertex. Because a vertex's insertion radius floor is a lower bound on its insertion radius, no edge smaller than $\mathrm{lfs}_{\min}/6$ is ever introduced into the mesh, and the algorithm must terminate. ■

The bound can be improved to $\mathrm{lfs}_{\min}/4$ in the same manner described following Theorem 24. I recommend both of the practical modifications to range-restricted segment splitting described in Section 3.5.2: use the closed legal range $[1, \sqrt{2}]$, and use a splitting procedure that occasionally takes two splits to get rid of an illegal subsegment.

4.5 Comparison with the Delaunay Meshing Algorithm of Miller, Talmor, Teng, Walkington, and Wang

The general-dimensional mesh generation algorithm of Miller, Talmor, Teng, Walkington, and Wang [67] bears many similarities to the present research, and as I shall demonstrate, achieves theoretical bounds similar to those proven in Section 4.2.3 for Delaunay refinement with equatorial spheres. The Miller et al. algorithm differs from ordinary Delaunay refinement in that it begins by deciding what vertex spacing is needed to meet a desired bound on circumradius-to-shortest edge ratio, and then generates a set of vertices to match.

The algorithm relies upon a spacing function $f(v)$ defined over the domain to be meshed. Imagine that each vertex v of the mesh is the center of a ball of radius $f(v)$. No two balls are allowed to overlap. This rule implies that any edge vw has length at least $f(v) + f(w)$. Hence, the spacing function sets a lower bound on the distance between vertices throughout the mesh.

To achieve good bounds on the circumradius-to-shortest edge ratios of the tetrahedra of the mesh, Miller et al. form a *maximal sphere-packing*, which is a set of vertices having the property that no additional vertex may be added without creating overlapping balls. Maximality ensures that tetrahedra with large circumradii cannot exist; recall that Chew's first Delaunay refinement algorithm uses maximality (with a constant spacing function) to eliminate triangles having angles smaller than 30°. The sphere-packing is also subject to the restriction that vertices may not lie inside the *protective sphere* of a subsegment or subfacet, much like the diametral and equatorial spheres of Delaunay refinement. True maximality is troublesome to achieve, and Miller et al. suggest relaxed forms of maximality that do not compromise the tetrahedron quality bounds; for instance, *circumcenter-maximality*, in which no vertex may be inserted at a tetrahedron circumcenter without creating overlapping balls, is sufficient.

The advantage of generating a mesh from a spacing function is that the complete vertex set can be generated prior to and independently from the tetrahedralization, much like the earliest Delaunay meshing algorithms in the engineering literature. As a result, it is relatively easy to parallelize the Miller et al. algorithm, whereas Delaunay refinement algorithms are difficult to parallelize because of synchronization concerns when multiple processors are simultaneously changing the topology of the mesh. Miller et al. create a maximal sphere-packing on each segment, then on each facet, and finally in the interior of the mesh. Each segment may be packed independently, and once the segments are finished, so may each facet. Finally, three-dimensional regions are packed. Even within a single region or facet, maximal sphere-packing is easier to parallelize than Delaunay refinement. After sphere packing is complete, the vertices are tetrahedralized, perhaps with a standard parallel convex hull algorithm.

The key innovation of the algorithm over earlier algorithms that generate a complete vertex set before triangulation is the use of the local feature size to determine vertex spacing. Provable bounds on tetrahedron quality may be obtained by choosing the spacing function $f(v) = \beta \mathrm{lfs}(v)$ for a sufficiently small value of β. The function $\mathrm{lfs}(\cdot)$ may be computed with the help of octrees. Miller et al. show that, for this spacing function, the three-dimensional version of their algorithm achieves circumradius-to-shortest edge ratios bounded below

$$B = \frac{2}{1-(7+2\sqrt{2})\beta}.$$

To compare this bound with the results of Section 4.2.3, I must revise Theorem 19 so that the minimum length of an edge is expressed in terms of both of the edge's endpoints.

Theorem 35 *For the Delaunay refinement algorithms discussed in this chapter, any edge vw of the final mesh has length at least $\frac{\mathrm{lfs}(v)+\mathrm{lfs}(w)}{2D_S+1}$.*

Proof: Lemma 29 and Theorem 32 show (each for a different value of D_S) that $\frac{\mathrm{lfs}(v)}{r_v} \leq D_S$ for any vertex v. Assume without loss of generality that w was added after v, and thus the distance between the two vertices is at least $r_w \geq \frac{\mathrm{lfs}(w)}{D_S}$. It follows that

$$|vw| \geq r_w \geq \frac{\mathrm{lfs}(w)+\mathrm{lfs}(w)}{2D_S}.$$

By Lemma 14, $\mathrm{lfs}(w) + |vw| \geq \mathrm{lfs}(v)$, so

$$|vw| \geq \frac{\mathrm{lfs}(w)+\mathrm{lfs}(v)-|vw|}{2D_S}.$$

It follows that $|vw| \geq \frac{\mathrm{lfs}(v)+\mathrm{lfs}(w)}{2D_S+1}$. ■

Based on the value of D_S calculated in Lemma 29, Delaunay refinement with equatorial spheres ensures that edge lengths are bounded by the inequality

$$|vw| \geq \frac{B-2}{(7+2\sqrt{2})B-2}[\mathrm{lfs}(v)+\mathrm{lfs}(w)].$$

In a mesh produced by the algorithm of Miller et al., no edge vw is shorter than $f(v) + f(w)$. If $f(p) = \beta \mathrm{lfs}(p)$, edge lengths are bounded by the inequality

$$|vw| \geq \frac{B-2}{(7+2\sqrt{2})B}[\mathrm{lfs}(v) + \mathrm{lfs}(w)].$$

It is not surprising that these bounds are quite similar, as the algorithms are based upon similar ideas, and are ultimately subject to the same imperatives of mathematical law. I do not know to what difference between the algorithms one should attribute the slightly better bound for Delaunay refinement, nor whether it marks a real difference between the algorithms or is an artifact of the different methods of analysis. However, there is no doubt about the source of the additional improvement one obtains by using equatorial lenses instead of equatorial spheres. Taking the value for D_S from Theorem 32, Delaunay refinement with equatorial lenses produces a tetrahedralization whose edge lengths are bounded by the inequality

$$|vw| \geq \frac{\sqrt{3}B - 2\sqrt{2}}{(3\sqrt{3} + 2\sqrt{6} + 4\sqrt{2})B - 2\sqrt{2}}[\mathrm{lfs}(v) + \mathrm{lfs}(w)].$$

Edge lengths in the Miller et al. algorithm and in Delaunay refinement with equatorial spheres decrease to zero as the quality bound approaches two, whereas as I have already discussed, Delaunay refinement with equatorial lenses produces well-graded meshes for quality bounds as low as 1.63. If range-restricted segment splitting is used, the quality bound may be further reduced to 1.15, although I can no longer prove that the final mesh is not uniform.

The real difference between the algorithms, however, is one not exposed by mathematics. Delaunay refinement is *lazy*, in the sense that it inserts a vertex only if a skinny simplex is present. The Miller et al. algorithm is not lazy at all; it is blind to the mesh that would be formed by the vertices it creates. However, by creating a maximal sphere-packing it inserts enough vertices to ensure that skinny simplices simply cannot survive.

How much does maximality cost? Figure 4.32, reprinted from Ruppert's original paper, gives us some idea. The figure charts the progress of the smallest angle in a triangular mesh during a typical run of Ruppert's algorithm. (During this run, no specific angle bound was applied; rather, the algorithm repeatedly splits the worst triangle in the mesh, even if it is nicely shaped.) The analysis presented in Section 3.3.4 implies that eventually, the curve should never drop below 20.7°. It is not clear how long the algorithm would have to run before reaching this hallowed state, but it is clear that the algorithm arrives at a mesh satisfying a 20°, or even 30°, angle bound long before. A maximal sphere-packing algorithm guaranteed to obtain a 20° angle bound produces a mesh at the high end of the curve, where the curve cannot dip below 20° again. Hence, it generates many more elements than a lazy algorithm.

For a more direct example, return to Figure 4.21 in Section 4.2.3. In theory, to obtain a quality bound of $B = 2.5$, one might have to tolerate edge lengths more than twenty times smaller than the local feature size; in practice, the number appears to be closer to two. Laziness appear to buy you a thousand-fold smaller mesh.

Is it possible to simultaneously obtain the benefits of Delaunay refinement and the parallelizability of Miller et al.?

The benefits of equatorial lenses can perhaps be realized in the Miller et al. algorithm. It seems straightforward to form a sphere-packing in which subfacets are protected with lenses, rather than spheres. The sticking point is tetrahedralizing the vertices of the sphere/lens packing. Lenses do not guarantee a Delaunay mesh, and the improved bounds that accompany them are a direct benefit of relaxing the requirement

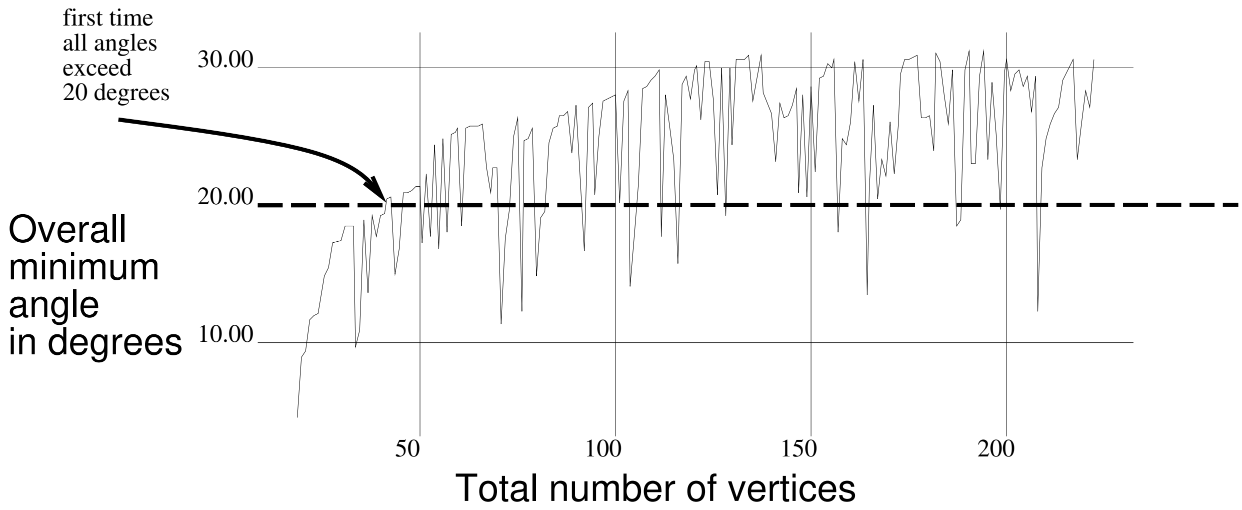


Figure 4.32: Progress of the minimum angle of a triangulation during a run of Ruppert's algorithm. (Courtesy Jim Ruppert.)

that meshes be Delaunay. As Section 2.1.3 demonstrates, a general algorithm for constructing constrained Delaunay tetrahedralizations is not going to appear. Does a maximal sphere/lens packing have special properties that guarantee that a constrained Delaunay tetrahedralization can be formed? If so, is there an algorithm for generating such tetrahedralizations? I will not pursue the question here.

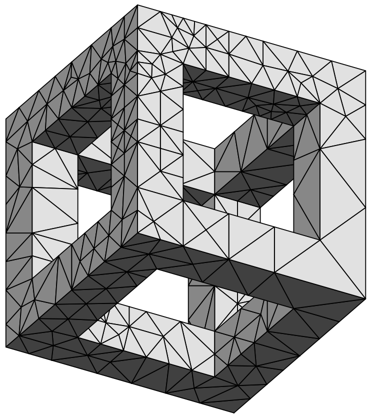
It also seems straightforward to use restricted subsegment lengths when generating a sphere-packing of a segment, but the only obvious way to reap the improved bounds proven in Section 4.4.2 is to put all the subsegments of the mesh into the same range, so that the shortest and longest subsegments of the mesh differ in length by a factor no greater than $\sqrt{2}$. While Delaunay refinement with range-restricted segment splitting might obtain the same unfortunate result in the worst case, it rarely happens in practice because of the algorithm's laziness.

And what of laziness? For a mesh generation algorithm to insert vertices lazily, it must be able to examine the quality of the simplices of the current mesh. Unfortunately, this implies maintaining a triangulation, which would seem to rule out the easy parallelization that Miller et al. offer. One hopes for, but does not expect to see, an elegant resolution to this dilemma. A suggestion is to use the Miller et al. algorithm to generate an initial mesh with coarser vertex spacing than the theory suggests, then refine it to remove the few poor quality elements that appear. Even if the latter step is sequential, it may be short enough that most of the speed benefits of parallelization are realized.

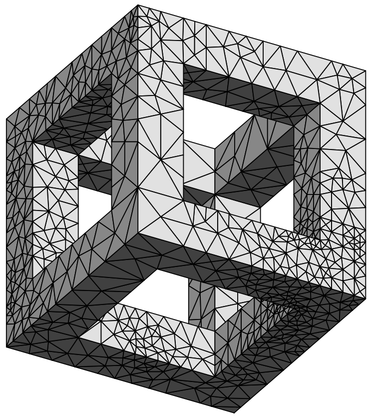
4.6 Sliver Removal by Delaunay Refinement

Although I have proven no theoretical guarantees about Delaunay refinement's ability to remove sliver tetrahedra, it is nonetheless natural to wonder whether Delaunay refinement might be effective in practice. If one inserts a vertex at the circumcenter of each sliver tetrahedron, will the algorithm fail to terminate?

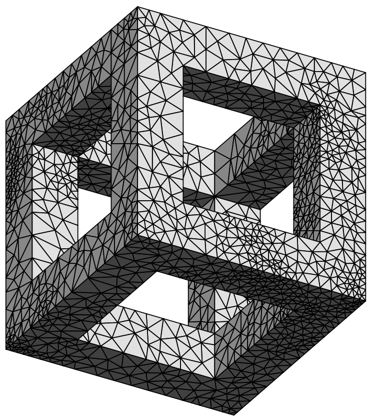
As Figure 4.33 demonstrates, Delaunay refinement can succeed for useful dihedral angle bounds. Each of the meshes illustrated was generated by applying a circumradius-to-shortest edge ratio bound B, and a dihedral angle bound $\theta_{\min}$. Not surprisingly, as the bound B was strengthened, the bound $\theta_{\min}$ had to be



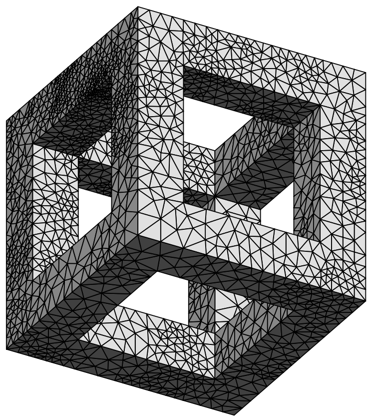
$B = 1.4$, $\theta_{\min} = 23.2°$, $\theta_{\max} = 146.2°$,
$h_{\min} = 0.47$, 505 vertices, 1331 tetrahedra.



$B = 1.2$, $\theta_{\min} = 19°$, $\theta_{\max} = 151.7°$,
$h_{\min} = 0.186$, 1937 vertices, 7303 tetrahedra.



$B = 1.1$, $\theta_{\min} = 14°$, $\theta_{\max} = 158.9°$,
$h_{\min} = 0.125$, 4539 vertices, 19048 tetrahedra.



$B = 1.06$, $\theta_{\min} = 11°$, $\theta_{\max} = 163.6°$,
$h_{\min} = 0.119$, 6514 vertices, 28543 tetrahedra.

Figure 4.33: Meshes created by Delaunay refinement using equatorial spheres and bounds on both circumradius-to-shortest edge ratio B and smallest dihedral angle $\theta_{\min}$. Also listed for each mesh is its largest dihedral angle $\theta_{\max}$ and its shortest edge length $h_{\min}$. The best lower bound on dihedral angles obtained for this PLC is $23.2°$. Compare with Figure 4.21 on Page 105.

weakened, or the algorithm did not terminate. For each mesh illustrated, raising the bound $\theta_{\min}$ by one degree causes the algorithm to fail to halt. It is not necessary to use a circumradius-to-shortest edge ratio bound at all. However, even if dihedral angles are the sole criterion for judging tetrahedron quaity, I have good reason to believe that smaller meshes are achieved if poor tetrahedra are ordered so that those with the largest circumradius-to-shortest edge ratios are split earliest. See Section 5.3.3 for further discussion.

Chew [22] offers hints as to why good results are obtained. A sliver can always be eliminated by splitting it, but how can one avoid creating new slivers in the process? Chew observes that a newly inserted vertex can only take part in a sliver if it is positioned badly relative to a triangular face already in the mesh. Figure 4.34 illustrates the set of bad positions. At left, a side view of the plane containing a face of the tetrahedralization is drawn. A tetrahedron formed by the face and a new vertex can have a small dihedral angle only if the new vertex lies within the slab depicted; this slab is the set of all points within a certain distance from the plane. Late in the Delaunay refinement process, such a tetrahedron can only arise if its circumradius-to-shortest edge ratio is small, which implies that it must lie in the region colored black in Figure 4.34 (left). This

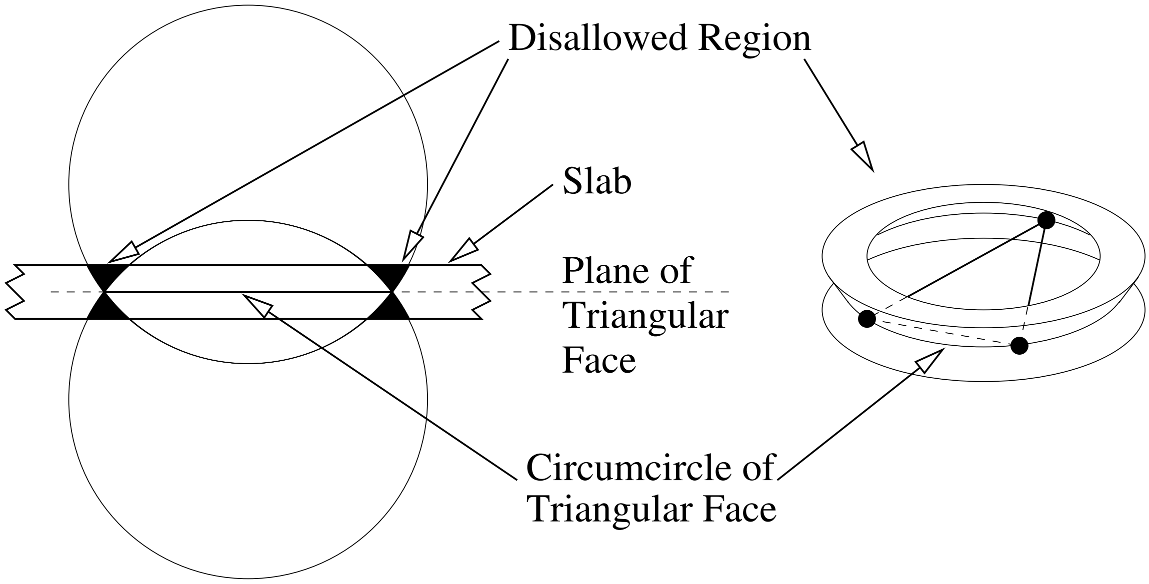


Figure 4.34: Left: A side view of the plane containing a triangular face. In conjunction with this face, a newly inserted vertex can form a sliver with both a small dihedral angle and a small circumradius-to-shortest edge ratio only if it is inserted in the disallowed region (black). Right: An oblique view of the disallowed region of a triangular face.

disallowed region, depicted at right, is shaped like a ring with an hourglass cross-section.

Chew shows that if the slab associated with each face is sufficiently thin, a randomized Delaunay refinement algorithm can avoid ever placing a vertex in the disallowed region of any face. The key idea is that each new vertex is not inserted precisely at a circumcenter; rather, a candidate vertex is generated at a randomly chosen location in the inner half of the circumsphere's radius. If the candidate vertex lies in some face's disallowed region, the candidate is rejected and a new one generated in its stead.

The algorithm will eventually generate a successful candidate, because the number of nearby triangular faces is bounded, and the volume of each disallowed region is small. If the sum of the volumes of the disallowed regions is less than the volume of the region in which candidate vertices are generated, a good candidate will eventually be found. To ensure that this condition is met, the slabs are made very thin.

Chew derives an explicit bound on the worst-case tetrahedron aspect ratio, which is too small to serve as a practical guarantee. However, there is undoubtedly a great deal of slack in the derivation. Even if the slabs are made thick enough to offer a useful bound on the minimum dihedral angle, the small volume of the disallowed region suggests that the practical prospects are good. My non-randomized Delaunay refinement algorithm seems to verify this intuition. I have not yet tested whether randomization is helpful in practice. Although randomization may reduce the frequency with which slivers are generated, the act of inserting vertices off-center in circumspheres weakens the bound on circumradius-to-shortest edge ratio.

Unfortunately, my practical success in removing slivers is probably due in part to the severe restrictions on input angle I have imposed upon Delaunay refinement. Practitioners report that they have the most difficulty removing slivers at the boundary of a mesh, especially near small angles. Figure 2.35 on Page 38 offers a demonstration of this observation. Mesh improvement techniques such as optimization-based smoothing and topological transformations, discussed in Section 2.2.4, can likely remove some of the imperfections that cannot be removed directly by Delaunay refinement.

4.7 Generalization to Higher Dimensions

I do not intend to carry out a full analysis of higher-dimensional Delaunay refinement algorithms here. However, I suspect that there are no further barriers to fully generalizing the method.

The most important result to generalize is the Projection Lemma (Lemma 26), which I have every reason to believe holds true for higher dimensional facets. The Projection Lemma is critical because it specifies a condition under which incident constrained polytopes can be guaranteed not to encroach upon each other. Specifically, if the lower-dimensional boundary polytopes of a constrained facet are not encroached, then the subfacets of that facet can only be encroached upon by vertices in the facet's orthogonal projection.

Additionally, the Projection Lemma makes it possible to choose an encroached simplex to split so that the insertion radius of the newly inserted vertex is no worse than $\sqrt{2}$ times smaller than that of its parent, regardless of the dimensionality of the simplices under consideration for splitting. Hence, in d dimensions one expects to achieve a quality bound $B > \sqrt{2}^{d-1}$ with good grading by using the straightforward generalization of Ruppert's algorithm, and $B = \sqrt{2}^{d-2}$ (without a guarantee of good grading) with the use of range-restricted segment splitting. I am also optimistic that lenses, rather than spheres, can be used to protect subfacets of dimension $d-1$, although spheres must be used to protect lower-dimensional subfacets. If so, one may achieve a quality bound $B > \frac{\sqrt{2}^d}{\sqrt{3}}$ with a guarantee of good grading, and (for $d \geq 3$) $B = \frac{\sqrt{2}^{d-1}}{\sqrt{3}}$ without.

If one believes that the Projection Lemma generalizes to higher dimensions, then Lemma 27, Theorem 28, Lemma 29, and Theorem 30 seem to generalize in straightforward ways. The most complicated pieces of Lemma 27 are those dealing with acute angles between simplices. Without some additional algorithmic insight, acute angles probably cannot be tolerated between simplices of dimension higher than one. An acute angle between a segment (1-simplex) and another simplex of dimension k may be permitted for any k, but the angle must be larger than $\arccos \frac{1}{\sqrt{2}^{k+1}}$. If range-restricted segment splitting is used, this angle may be relaxed to $\arccos \frac{1}{\sqrt{2}^k}$ for $k \geq 2$.

Of course, this discussion begs the question of whether anyone would want such an algorithm. A four-dimensional mesh generator might find use in space-time finite element methods, where time is represented as a fourth spatial dimension. This might be an ideal application for Delaunay refinement methods, because for some problems, no additional small angles will ensue from consideration of the time dimension. Commonly, the region to be meshed is nothing more complicated than an unchanging three-dimensional object extruded orthogonally in the time dimension. In this case, the reason to create a four-dimensional mesh is so that one may adjust the density of nodes through time in order to track time-dependent multiscale phenomena, such as turbulent fluid flow.

4.8 Conclusions

Delaunay refinement is an effective technique for three-dimensional mesh generation. Its theoretical guarantees on element quality and mesh grading make it attractive. Taken at face value, however, these guarantees are not wholly satisfying. There is no guarantee that slivers can be eliminated. Although the constant D_S derived in Section 4.3.2 gives us confidence that edge sizes cannot become smaller than one twelfth the local feature size when applying a quality bound of $B = 2.5$, this bound may seem insufficiently strong for practical purposes, especially when one recalls that the number of elements is inversely proportional to the cube of the edge length.

Fortunately, Delaunay refinement falls into the class of algorithms that usually outperform their worst-case bounds. The proof techniques used to analyze Delaunay refinement fail to take into account a great deal of "slack" in the mesh; the relationship between the insertion radii of a parent and its child is usually looser than in the worst case. This slack accumulates as one traces a sequence of descendants from an input vertex. One can apply a tighter bound on circumradius-to-shortest edge ratio than the theory suggests is possible, or even apply bounds on dihedral angles, and still produce a small, nicely graded mesh.

Despite this pleasant gap between theory and practice, the theory is helpful for suggesting innovations that are likely to bear fruit in practice, such as equatorial lenses.

The main outstanding problem in three-dimensional Delaunay refinement is the question of how to handle small input angles. Is there a method as effective as the Quitter, presented in Section 3.7? Modified segment splitting using concentric spherical shells is probably as good an idea in three dimensions as in two, but it only begins to address the possible problems. What about facets separated by small angles? How can their vertices be kept from encroaching upon each other's subfacets? One suggestion is to modify the method of subfacet splitting. If two subfacets meet at a subsegment, separated by a small angle, and one of the subfacets is encroached, perhaps it should be split in such a way that an equilateral triangle is formed at the subsegment. In this manner, subfacets separated by small angles are prevented from engaging in a diminishing spiral of mutual encroachment, just as subsegments are prevented from doing so by modified segment splitting. This idea holds promise, but falls short of a complete solution to the problem of small angles.

In two dimensions, there is a sure-fire solution: never insert a vertex whose insertion radius is smaller than the insertion radius of its most recently inserted ancestor. An impediment to using this strategy in three dimensions, besides the awful elements it produces, is that boundary recovery may fail if a missing subsegment or subfacet is not split because of this rule. This problem is surmounted in two dimensions by the constrained Delaunay triangulation, but this option is not available in three. Section 5.3.1 suggests a way to garner some of the advantages of constrained triangulation, but offers no guarantees.

Nevertheless, if segments and facets are inserted in sequence, and the subsegments and subfacets of each are locked as soon as they are recovered, then they will all be recovered eventually. As I mentioned at the end of Section 4.3.2, the length of the shortest edge in the final mesh may be exponentially small, where the exponent is proportional to the number of facets. After the boundaries have been completely recovered, the sure-fire solution can be applied. Hence, it is always possible to ensure that three-dimensional Delaunay refinement terminates, although the elements might be poor in quality and much smaller than desired.

Chapter 5

Implementation

Triangle is a C program for two-dimensional mesh generation and construction of Delaunay triangulations, constrained Delaunay triangulations, and Voronoi diagrams. *Pyramid* is a C program for three-dimensional mesh generation and Delaunay tetrahedralization. These programs are implementations of the Delaunay refinement algorithms discussed in the previous chapters. Triangle and Pyramid are fast, memory-efficient, and robust. Triangle computes Delaunay triangulations and constrained Delaunay triangulations exactly; Pyramid computes Delaunay tetrahedralizations exactly.

Features of both programs include user-specified constraints on element quality and size, user-specified holes and concavities, the ability to refine preexisting triangulations, and the economical use of exact arithmetic to improve robustness. This chapter discusses many of the key implementation decisions, including the choice of triangulation algorithms and data structures, the steps taken to create and refine a mesh, and other issues. The use of exact arithmetic to ensure the correctness of Delaunay triangulations and tetrahedralizations, and to improve the robustness of both mesh generators, is discussed at length in Chapter 6.

Many of the implementation decisions in a complex program like a mesh generator depend upon how one wishes to trade off speed and memory use. Triangle and Pyramid are designed to support large scientific computing projects, in which the sizes of the meshes that can be produced are limited by the available memory, and not by the amount of time the program can run. Therefore, many of the decisions described in this chapter are motivated by the desire to make space efficiency a priority, without unduly sacrificing speed.

Delaunay triangulation timings (seconds)									
Number of points	10,000			100,000			1,000,000		
Point distribution / Algorithm	Uniform Random	Boundary of Circle	Tilted Grid	Uniform Random	Boundary of Circle	Tilted Grid	Uniform Random	Boundary of Circle	Tilted Grid
Div&Conq, alternating cuts									
robust	0.33	0.57	0.72	4.5	5.3	5.5	58	61	58
non-robust	0.30	0.27	0.27	4.0	4.0	3.5	53	56	44
Div&Conq, vertical cuts									
robust	0.47	1.06	0.96	6.2	9.0	7.6	79	98	85
non-robust	0.36	0.17	failed	5.0	2.1	4.2	64	26	failed
Sweepline									
non-robust	0.78	0.62	0.71	10.8	8.6	10.5	147	119	139
Incremental									
robust	1.15	3.88	2.79	24.0	112.7	101.3	545	1523	2138
non-robust	0.99	2.74	failed	21.3	94.3	failed	486	1327	failed

Table 5.1: Timings for triangulation on a DEC 3000/700 with a 225 MHz Alpha processor, not including I/O. Robust and non-robust versions of the Delaunay algorithms were used to triangulate points chosen from one of three different distributions: uniformly distributed random points in a square, random approximately cocircular points, and a tilted square grid.

5.1 Triangulation Algorithms

5.1.1 Comparison of Three Delaunay Triangulation Algorithms

A mesh generator rests on the efficiency of its triangulation algorithms and data structures, so I discuss these first.

There are many Delaunay triangulation algorithms, some of which are surveyed and evaluated by Fortune [33] and Su and Drysdale [91, 90]. Their results indicate a rough parity in speed, to within a factor of two, among the incremental insertion algorithm of Lawson [59], the divide-and-conquer algorithm of Lee and Schachter [60], and the plane-sweep algorithm of Fortune [31]; however, the implementations they study were written by different people. I believe that Triangle is the first instance in which all three algorithms have been implemented with the same data structures and floating-point tests, by one person who gave roughly equal attention to optimizing each. (Some details of how these implementations were optimized appear in Section 5.1.2.)

Table 5.1 compares the algorithms, including versions that use exact arithmetic (see Chapter 6) to achieve robustness, and versions that use approximate arithmetic and are hence faster but may fail or produce incorrect output. (The robust and non-robust versions are otherwise identical.) As Su and Drysdale [91] also found, the divide-and-conquer algorithm is fastest, with the sweepline algorithm second. The incremental algorithm performs poorly, spending most of its time in point location. (Su and Drysdale produced a better incremental insertion implementation by using bucketing to perform point location, but it still ranks third. Triangle does not use bucketing because it is easily defeated, as discussed in Section 5.1.2.) The agreement between my results and those of Su and Drysdale lends support to their ranking of algorithms.

An important optimization to the divide-and-conquer algorithm, adapted from Dwyer [30], is to partition the vertices with alternating horizontal and vertical cuts (Lee and Schachter's algorithm uses only vertical cuts). Alternating cuts speed the algorithm and, when exact arithmetic is disabled, reduce its likelihood of failure. One million points can be triangulated correctly in a minute on a fast workstation.

All three triangulation algorithms are implemented so as to eliminate duplicate input points; if not eliminated, duplicates can cause catastrophic failures. The sweepline algorithm can easily detect duplicate points as they are removed from the event queue (by comparing each with the previous point removed from the queue), and the incremental insertion algorithm can detect a duplicate point after point location. My implementation of the divide-and-conquer algorithm begins by sorting the points by their x-coordinates, after which duplicates can be detected and removed. This sorting step is a necessary part of the divide-and-conquer algorithm with vertical cuts, but not of the variant with alternating cuts (which must perform a sequence of median-finding operations, alternately by x and y-coordinates). Hence, the timings in Table 5.1 for divide-and-conquer triangulation with alternating cuts could be improved slightly if one could guarantee that no duplicate input points would occur; the initial sorting step would be unnecessary.

5.1.2 Technical Implementation Notes

This section presents technical details of my Delaunay triangulation implementations that are important for anyone who wishes to evaluate the usefulness of my evaluations, or to modify the code.

The sweepline and incremental Delaunay triangulation implementations compared by Su and Drysdale [91] each use some variant of uniform bucketing to locate points. Bucketing yields fast implementations on uniform point sets, but is easily defeated; a small, dense cluster of points in a large, sparsely populated region may all fall into a single bucket. I have not used bucketing in Triangle, preferring algorithms that exhibit good performance with any distribution of input points. As a result, Triangle may be slower than necessary when triangulating uniformly distributed point sets, but will not exhibit asymptotically slower running times on difficult inputs.

Fortune's sweepline algorithm uses two nontrivial data structures in addition to the triangulation: a priority queue to store events, and a balanced tree data structure to store the sequence of edges on the boundary of the mesh. Fortune's own implementation, available from Netlib, uses bucketing to perform both these functions; hence, an $\mathcal{O}(n \log n)$ running time is not guaranteed, and Su and Drysdale [91] found that the original implementation exhibits $\mathcal{O}(n^{3/2})$ performance on uniform random point sets. By modifying Fortune's code to use a heap to store events, they obtained $\mathcal{O}(n \log n)$ running time on uniformly distributed point sets, and better performance for point sets having more than about 50,000 points. However, they found that bucketing outperforms a heap on smaller point sets.

Triangle's implementation uses a heap as well, and also uses a splay tree [88] to store mesh boundary edges, so that an $\mathcal{O}(n \log n)$ running time is attained, regardless of the distribution of points. Not all boundary edges are stored in the splay tree; when a new edge is created, it is inserted into the tree with probability 0.1. (The value 0.1 was chosen empirically to minimize the triangulation time for uniform random point sets.) At any time, the splay tree contains a random sample of roughly one tenth of the boundary edges. When the sweepline sweeps past an input point, the point must be located relative to the boundary edges; this point location involves a search in the splay tree, followed by a search on the boundary of the triangulation itself. By keeping the splay tree small, this scheme improves the speed and memory use of point location without changing the asymptotic performance. This is an example of how randomization can be used to reduce the constants, rather than the asymptotic behavior, associated with a geometric algorithm.

A splay tree adjusts itself so that frequently accessed items are near the top of the tree. Hence, a point set organized so that many new vertices appear at roughly the same location on the boundary of the mesh is likely to be triangulated quickly. This effect partly explains why Triangle's sweepline implementation triangulates points on the boundary of a circle more quickly than the other point sets, even though there are many more boundary edges in the cocircular point set and the splay tree grows to be much larger (containing $\mathcal{O}(n)$ boundary edges instead of $\mathcal{O}(\sqrt{n})$). For this reason, I believe that splay trees are better suited to sweepline Delaunay triangulation than other balanced tree algorithms, such as red-black trees.

Triangle's incremental insertion algorithm for Delaunay triangulation uses the point location method proposed by Mücke, Saias, and Zhu [72]. Their *jump-and-walk* method chooses a random sample of $\mathcal{O}(n^{1/3})$ vertices from the mesh (where n is the number of nodes *currently* in the mesh), determines which of these vertices is closest to the query point, and walks through the mesh from the chosen vertex toward the query point until the triangle containing that point is found. Mücke et al. show that the resulting incremental algorithm takes expected $\mathcal{O}(n^{4/3})$ time on uniform random point sets. Table 5.1 appears to confirm this analysis. Triangle uses a sample size of $0.45n^{1/3}$; the coefficient was chosen empirically to minimize the triangulation time for uniform random point sets. Triangle also checks the previously inserted point, because in many practical point sets, any two consecutive points have a high likelihood of being near each other.

I have not implemented the $\mathcal{O}(n \log n)$ point location scheme suggested by Guibas, Knuth, and Sharir [46], although it promises to outperform the method of Mücke et al. Even with asymptotically better point location, the incremental insertion algorithm seems unlikely to surpass the performance of the divide-and-conquer algorithm.

5.2 Data Structures

5.2.1 Data Structures for Triangulation in Two Dimensions

Should one choose a data structure that uses a record to represent each edge, or one that uses a record to represent each triangle? Triangle was originally written using Guibas and Stolfi's *quad-edge* data structure [47] (without the *Flip* operator), then rewritten using a triangle-based data structure. The quad-edge data structure is popular because it is elegant, because it simultaneously represents a graph and its geometric dual (such as a Delaunay triangulation and the corresponding Voronoi diagram), and because Guibas and Stolfi give detailed pseudocode for implementing the divide-and-conquer and incremental Delaunay algorithms using quad-edges.

Despite the fundamental differences between the data structures, the quad-edge-based and triangle-based implementations of Triangle are both faithful to the Delaunay triangulation algorithms presented by Guibas and Stolfi [47] (I did not implement a quad-edge sweepline algorithm), and hence offer a fair comparison of the data structures. Perhaps the most useful observation of this chapter for practitioners is that the divide-and-conquer algorithm, the incremental algorithm, and Ruppert's Delaunay refinement algorithm were all sped by a factor of two by the triangular data structure. (However, it is worth noting that the code devoted specifically to triangulation is roughly twice as long for the triangular data structure.) A difference so pronounced demands explanation.

First, consider the different storage demands of each data structure, illustrated in Figure 5.1. Each quad-edge record contains four pointers to neighboring quad-edges, and two pointers to vertices (the endpoints of the edge). Each triangle record contains three pointers to neighboring triangles, and three pointers to vertices. Hence, both structures contain six pointers. A triangulation contains roughly three edges for every two triangles. Hence, the triangular data structure is more space-efficient.

It is difficult to ascertain why the triangular data structure is superior in time as well as space, but one can make educated inferences. When a program makes structural changes to a triangulation, the amount of time used depends in part on the number of pointers that have to be read and written. This amount is smaller for the triangular data structure; more of the connectivity information is implicit in each triangle. Cacheing is improved by the fact that fewer structures are accessed. (For large triangulations, any two adjoining quad-edges or triangles are unlikely to lie in the same cache line.)

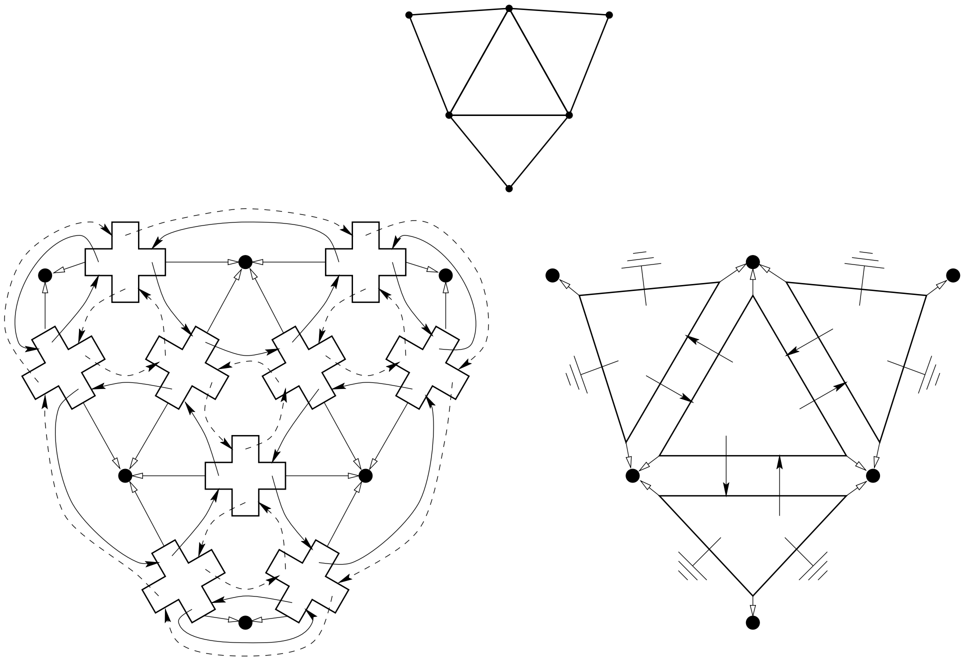


Figure 5.1: A triangulation (top) and its corresponding representations with quad-edge (left) and triangular (right) data structures. Each quad-edge and each triangle contains six pointers.

Both the quad-edge and triangle data structures must store not only pointers to their neighbors, but also the *orientations* of their neighbors, to make clear how they are connected. For instance, each pointer from a triangle to a neighboring triangle has an associated orientation (a number between zero and two) that indicates which edge of the neighboring triangle is contacted. An important space optimization is to store the orientation of each quad-edge or triangle in the bottom two bits of the corresponding pointer. Thus, each record must be aligned on a four-byte boundary. This space optimization is probably a speed optimization as well, as memory traffic in modern machines is becoming more and more expensive compared to bit operations.

Because the triangle-based divide-and-conquer algorithm proved to be fastest, it is worth exploring in some depth. At first glance, the algorithm and data structure seem incompatible. The divide-and-conquer algorithm recursively halves the input vertices until they are partitioned into subsets of two or three vertices each. Each subset is easily triangulated (yielding an edge, two collinear edges, or a triangle), and the triangulations are merged together to form larger ones. But how does one represent an edge or a sequence of collinear edges with a triangular data structure? If one uses a degenerate triangle to represent an isolated edge, the resulting code is clumsy because of the need to handle special cases. One might partition the input into subsets of three to five vertices, but this does not help if the points in a subset are collinear.

To preserve the elegance of Guibas and Stolfi's presentation of the divide-and-conquer algorithm, each triangulation is surrounded with a layer of "ghost" triangles, one triangle per convex hull edge. The ghost triangles are connected to each other in a ring about a "vertex at infinity" (really just a null pointer). A single edge is represented by two ghost triangles, as illustrated in Figure 5.2.

Ghost triangles are useful for efficiently traversing the convex hull edges during the merge step. Some are transformed into real triangles during this step; two triangulations are sewn together by fitting their ghost triangles together like the teeth of two gears. (Some edge flips are also needed. See Figure 5.3.) Each merge step creates only two new triangles; one at the bottom and one at the top of the seam. After all the

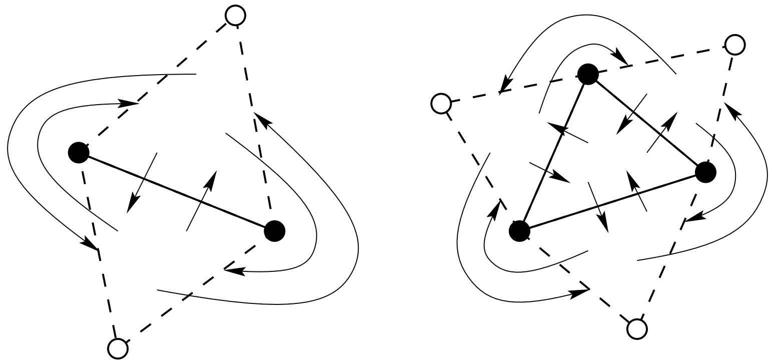


Figure 5.2: How the triangle-based divide-and-conquer algorithm represents an isolated edge (left) and an isolated triangle (right). Dashed lines represent ghost triangles. White vertices all represent the same "vertex at infinity"; only black vertices have coordinates.

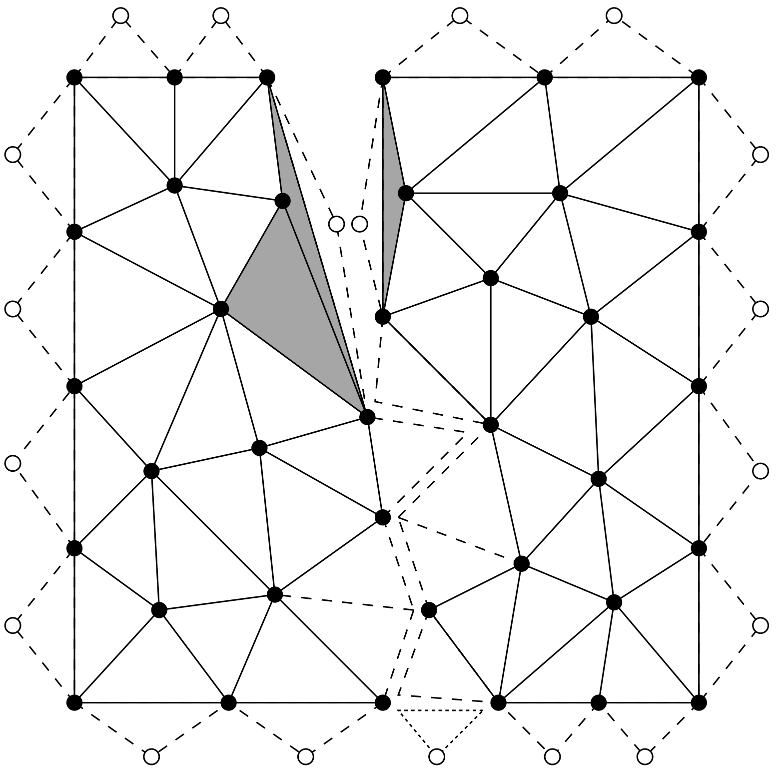


Figure 5.3: Halfway through a merge step of the divide-and-conquer algorithm. Dashed lines represent ghost triangles and triangles displaced by edge flips. The dotted triangle at bottom center is a newly created ghost triangle. Shaded triangles are not Delaunay and will be displaced by edge flips.

merge steps are done, the ghost triangles are removed and the triangulation is passed on to the next stage of meshing.

Ghost triangles are especially useful for reducing the amount of special-case code. For example, consider performing an edge flip between two triangles that lie at the boundary of the mesh. The two triangles must be detached from their neighbors, rotated a quarter turn, and reattached. One of the tasks performed during reattachment is adjusting the pointers of each of the four neighboring triangles. Without ghost triangles, some of the neighbors might be null pointers, and conditional code is required to check each. With ghost triangles, the conditionals are not required. Although this may seem like a trivial concern, the number of similar cases in which ghost triangles simplified the implementation of Triangle was large enough to make it worthy of note.

Precisely the same data structure, ghost triangles and all, is used in the sweepline implementation to represent the growing triangulation. Ghost triangles are handy for representing the dangling edges that appear on the advancing front of the triangulation, and for navigating along the front during point location. Details are omitted.

5.2.2 Data Structures for Mesh Generation in Two Dimensions

Augmentations to the quad-edge and triangle data structures are necessary to support the constrained triangulations needed for mesh generation. As Section 3.3 mentioned, Delaunay refinement can be implemented with or without locked edges. As a practical matter, though, subsegments need to be flagged so that encroached subsegments can be quickly detected. Hence, there is nothing to lose by locking subsegments, and a speed improvement will result, because unnecessary edge flips are not performed.

Other augmentations are needed too. Additional information may be associated with each subsegment, vertex, and element of the mesh. Commonly, subsegments and vertices must carry markers to identify which segments they lie upon, so that the correct boundary conditions may be applied to them in a finite element simulation or other numerical PDE solver. If a smoothing algorithm is implemented, it will need to know which subsegments are connected together into a single segment, so that vertices may be moved along the length of the segment. If curved segments are supported, information about the curvature of a subsegment is needed whenever that subsegment is split. (At the time of this writing, Triangle supports markers for boundary conditions and stores subsegment connectivity. Smoothing and curved surfaces are not implemented.)

Each triangle of the mesh may need to carry associated attributes such as its maximum permissible triangle area or physical constants needed for a finite element simulation. Vertices might also have physical constants associated with them. It is also useful for a mesh generator to be able to tag each element to identify the region of the mesh in which the element falls. Triangle, for instance, allows the user to specify segment-bounded regions of the mesh whose elements should be tagged with specified numerical markers.

While each element of the mesh may have associated attributes, the only edges that generally have any special information associated with them are subsegments. Hence, it is easier to augment the triangular data structure to include subsegment attributes, using a separate data structure that represents a subsegment, than to augment the quad-edge data structure to include element attributes.

I modify the triangular data structure to meet the requirements described above by augmenting each triangle with three extra pointers (one for each edge), which are usually null but may point to a subsegment data structure (Figure 5.4). In large meshes where the number of triangles is determined primarily by area constraints and not by the input geometry, only a minority of edges are subsegments, so the memory occupied by subsegments is small. However, the memory occupied by triangles is increased by one-half. In the special case where information is associated with subsegments but not with elements, the additional three pointers in each triangle eliminate the space advantage of the triangular data structure relative to quad-edges, but not its speed advantage. Triangle uses the longer nine-pointer record only if subsegments are present; six-pointer triangles are used for unconstrained Delaunay triangulation.

In large meshes, most of the pointers from triangles to subsegments are null, so each triangle record can be reduced to seven pointers by using just a single subsegment pointer. In a triangle that contacts no

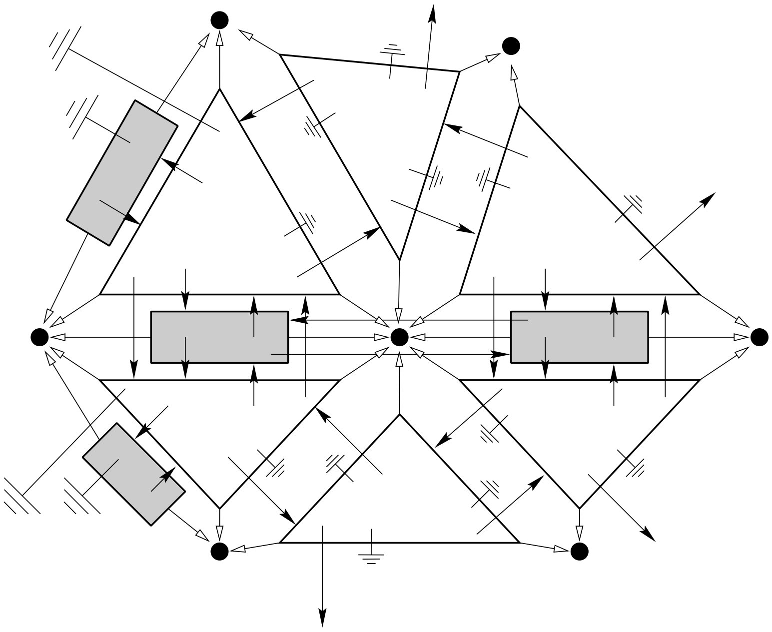


Figure 5.4: Shaded boxes represent subsegments, which may be linked together into segments. Each triangle has three additional pointers that reference adjoining subsegments.

subsegment, this pointer is null. In a triangle that contacts one or more subsegments, this pointer points to a separate record containing three pointers to subsegments. The number of these special records is small, so they increase the space requirements only modestly. I have not implemented this space optimization in Triangle.

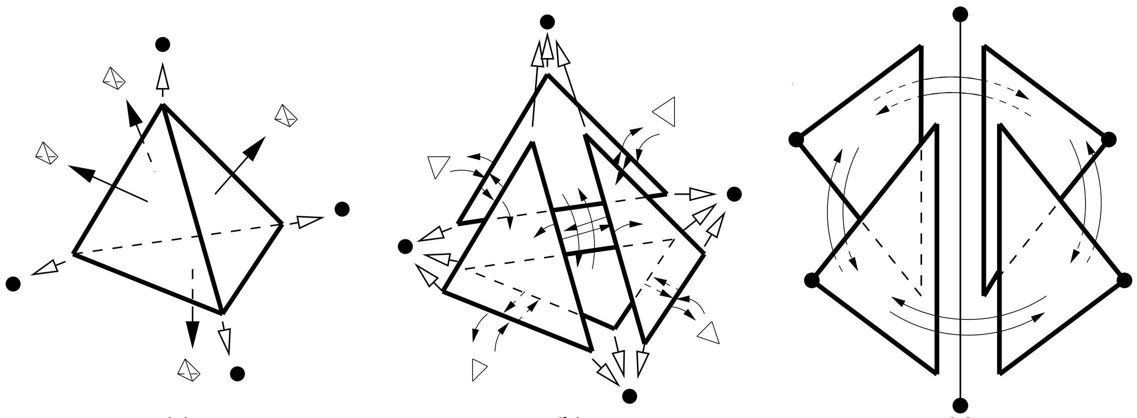
A more aggressive optimization would be to use the original six-pointer triangles, but each of a triangle's three pointers to neighbors can point to either a triangle or a subsegment. A one-bit tag (possibly hidden in the lower bits of each pointer) would distinguish between the two. This space optimization would increase the amount of conditional code executed; it is not clear how bad its effect on running time would be.

To save space and time, Triangle and Pyramid do not maintain pointers from mesh vertices to any other structure. Variables in each program often denote a vertex not by pointing directly to the vertex, but rather by pointing to an element that contains the vertex. Hence, mesh structures connected to the vertex may be identified.

5.2.3 Data Structures for Three Dimensions

There are at least three choices of data structure to represent a tetrahedralization. One could use a record to represent each tetrahedron, a record to represent each face, or a record to represent each pairing of a face and an edge (hence, three records per triangular face). The last structure, proposed by Dobkin and Laszlo [29], is the most general, and can be used to represent arbitrary spatial subdivisions. However, a tetrahedralizer does not need this generality, and memory considerations easily rule out all but the first option.

Consider, for instance, the minimum memory requirements for a tetrahedron-based Delaunay tetrahedralizer, and for a face-based tetrahedralizer. In the former case, illustrated in Figure 5.5(a), the record that represents a tetrahedron must have eight pointers: four for its vertices, and four for the adjoining tetrahedra. In the latter case, illustrated in Figure 5.5(b), the record that represents a triangular face must have six



(a) (b) (c)

Figure 5.5: (a) Tetrahedron-based data structure. (b) Face-based data structure. (c) A doubly linked list of faces about an edge.

pointers, and ideally has nine: three for its vertices, and three or six that point to adjacent triangular faces. The choice of three or six depends on whether one wishes to have a singly-linked or doubly-linked list of faces about each edge of the tetrahedralization; the latter is illustrated in Figure 5.5(c). A singly-linked list of faces is slower to traverse, and more cumbersome to program.

There are roughly two faces stored for each tetrahedron, because each tetrahedron has four faces, and each face (except on exterior boundaries) is shared by two tetrahedra. Hence, the cost of a face-based data structure is twelve or eighteen pointers per tetrahedron, which markedly exceeds the memory requirements of the tetrahedron-based data structure.

For mesh generation, as opposed to Delaunay tetrahedralization, the data structures must be able to represent constrained subfacets and subsegments, and be able to associate attributes with subfacets, subsegments, elements, and vertices. As in the two-dimensional case, such attributes might be associated with each element of the mesh, but the only edges and faces that generally carry such information are subsegments and subfacets. Hence, the tetrahedron-based data structure is more utilitarian that the face-based data structure. The tetrahedral data structure almost certainly results in a faster implementation, if the two-dimensional Delaunay implementations are any indication. I have not attempted implementing tetrahedralization algorithms with any other data structure.

The remainder of this section is devoted to a discussion of how the tetrahedron-based data structure is modified in Pyramid to accommodate subfacets and subsegments. Just as the triangular data structure uses three additional pointers to attach subsegments, the tetrahedral data structure uses four additional pointers to attach subfacets. As in the two-dimensional case, if the mesh is large, the data structures that represent subfacets and subsegments occupy only a small portion of memory, and the four pointers from a tetrahedron to adjoining subfacets can be reduced to one, or even zero. Hence, a tetrahedral record consists of eight pointers if only Delaunay tetrahedralization is performed, or eight, nine, or twelve pointers for mesh generation. (Pyramid currently uses twelve pointers in the latter case.)

The data structure that represents a subfacet contains three pointers to its vertices, three pointers to adjoining subfacets, and two pointers to adjoining tetrahedra. The three pointers to adjoining subfacets are used only to indicate coplanar neighbors in a common facet. These pointers are important to the Delaunay refinement algorithm, because they indicate that the shared edge can be flipped to satisfy the Delaunay criterion when a vertex is inserted in the facet. Figure 5.6 illustrates two subfacets, connected at a flippable

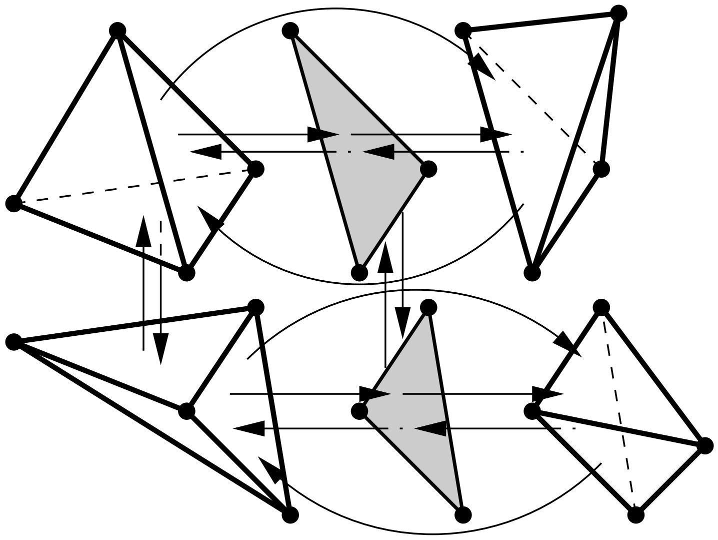
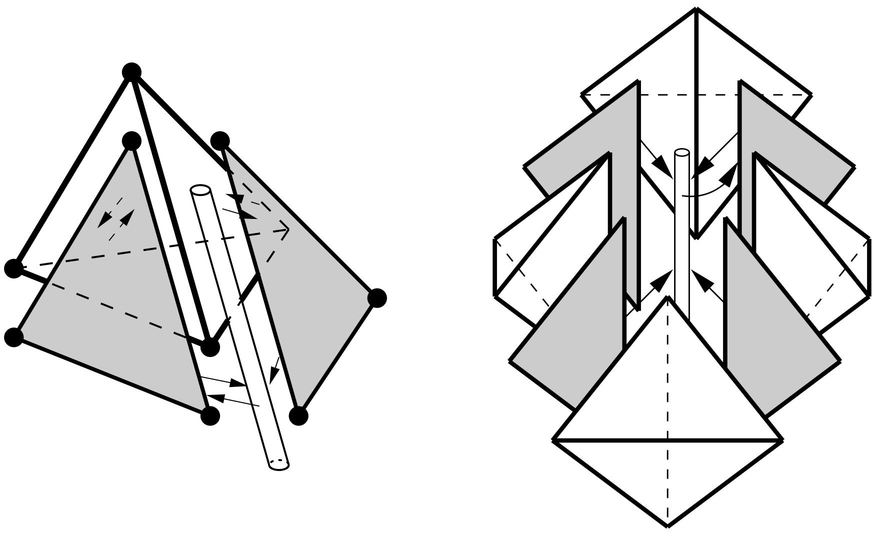


Figure 5.6: The records representing subfacets (shaded) have pointers to adjoining tetrahedra, subfacets, and vertices. Subfacets are directly linked to each other only if they are part of the same facet, and the edge they share is flippable.



(a) (b)

Figure 5.7: (a) Tetrahedra and subsegments are only connected via subfacets. (b) Each subsegment has a full complement of *wings*, which are subfacets that anchor it to adjoining tetrahedra.

edge and sandwiched between tetrahedra, that together form a quadrilateral facet. Figure 4.10 in Chapter 4 illustrates a circumstance in which such edge flips occur.

Each subfacet also has three pointers to adjoining subsegments. To save space, there are no pointers directly connecting tetrahedra and adjoining subsegments; connections between tetrahedra and subsegments are entirely mediated through subfacets, as illustrated in Figure 5.7(a). Because a subsegment may be shared by any number of subfacets and tetrahedra, each subsegment has a pointer to only one adjoining subfacet (chosen arbitrarily); the others must be found through the connectivity of the mesh. To ensure that every subsegment incident to a tetrahedron may be found, each subsegment has a full complement of *wings*, which

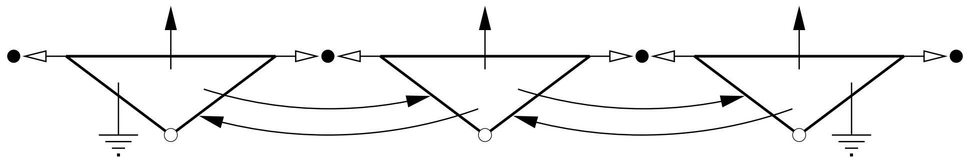


Figure 5.8: Subsegments are represented by degenerate subfacets. A chain of linked subsegments form a segment. Open circles represent null vertices. The pointers directed upward in the illustration point to adjoining subfacets, which may or may not be solid. Different subsegments of this segment may even point to subfacets of different facets.

are subfacets that link a subsegment to its adjoining tetrahedra, as illustrated in Figure 5.7(b). Some or all of these subfacets may be *nonsolid subfacets*, which are not "real" subfacets, but are present solely for the purpose of connecting tetrahedra to subsegments. Only *solid subfacets*, which lie within facets, are locked in place. The faces occupied by nonsolid subfacets are eligible for flipping according to the Delaunay criterion.

It has proven to be quite convenient to represent subsegments with the same data structure used for subfacets, in a manner illustrated in Figure 5.8. A subfacet record used to represent a subsegment has one null vertex opposite its "real" edge. A subsegment is similar to a ghost triangle: it is connected at its "real" edge to an adjoining triangular subfacet, and it is linked to neighboring subsegments (of the same segment) at its "fake" edges. The decision to represent subsegments with the same data structure used for subfacets has eliminated the need for much special-case code that Pyramid would otherwise incorporate.

5.3 Implementing Delaunay Refinement Algorithms

This section describes Delaunay refinement as it is implemented in Triangle and Pyramid. Figures 5.9 through 5.13 illustrate the process of meshing a PSLG that represents an electric guitar.

The first stage of both Triangle and Pyramid is to find the Delaunay triangulation or tetrahedralization of the input vertices, as in Figure 5.10. In general, some of the input segments and facets are missing from the triangulation; the second stage is to recover them. Figure 5.11 illustrates the constrained Delaunay triangulation of the input PSLG.

The third stage of the algorithm, which diverges from Ruppert [82], is to remove triangles or tetrahedra from concavities and holes (Figure 5.12). The fourth stage of the algorithm is to apply a Delaunay refinement algorithm to the mesh, as described in Chapters 3 and 4. Figure 5.13 illustrates a final mesh having no angles smaller than $20°$.

The last three stages are described in the following sections.

5.3.1 Segment and Facet Recovery

Although the theoretical treatment of encroached subsegments and subfacets is no different for those that are missing and those that are present in the mesh, they are treated very differently in practice. Whereas missing segments and facets require the maintenance of a separate triangulation of each segment and facet (which is almost trivial for segments), subsegments and subfacets that are present in the mesh do not, and their encroachment can be detected much more easily. Furthermore, missing subsegments and subfacets can sometimes be recovered without inserting a new vertex. For reasons to be stated shortly, this solution is often preferable.

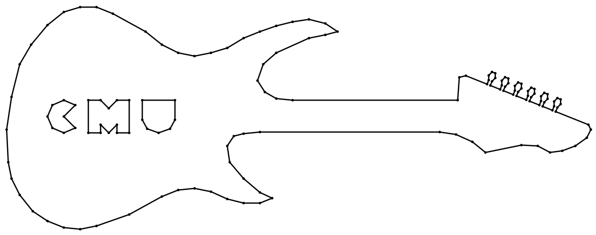


Figure 5.9: Electric guitar PSLG.

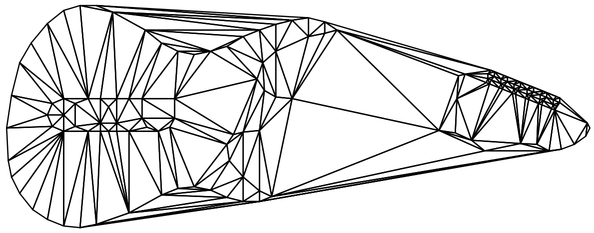


Figure 5.10: Delaunay triangulation of vertices of PSLG. The triangulation does not conform to all of the input segments.

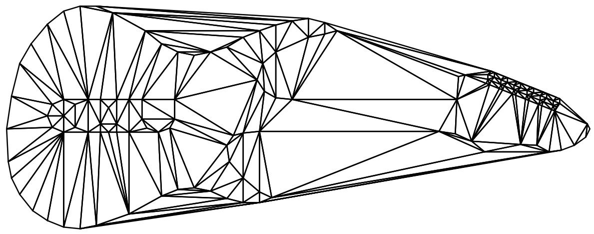


Figure 5.11: Constrained Delaunay triangulation of PSLG.

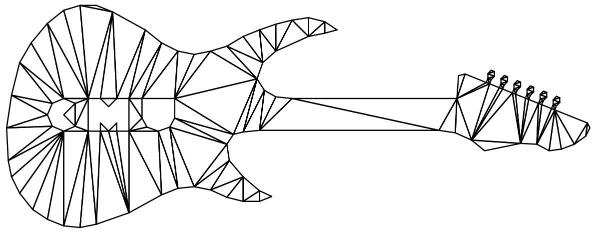


Figure 5.12: Triangles are removed from concavities and holes.

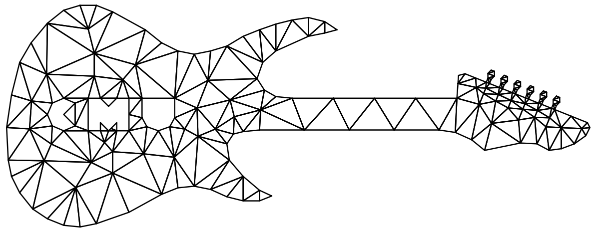


Figure 5.13: Conforming Delaunay triangulation with $20°$ minimum angle.

Triangle can force the mesh to conform to the input segments in one of two ways, selectable by the user. The first was described in Section 3.3; any segment that fails to appear in the mesh is recursively divided by inserting vertices along its length, using Lawson's incremental insertion algorithm to maintain the Delaunay property, until the entire segment is represented as a union of edges of the mesh. Subsegments are locked as they appear. Input segments that are not missing from the initial triangulation must also be identified and locked.

The second choice is to simply use a constrained Delaunay triangulation, as Figure 5.11 demonstrates. Each segment is inserted by deleting the triangles it overlaps, and the region on each side of the segment is Delaunay triangulated anew (recall Figure 2.16 on Page 21). No new vertices are inserted. Triangle uses the constrained Delaunay triangulation by default.

Incremental insertion of segments is not an optimal method of constructing a constrained Delaunay triangulation; I could have chosen the optimal $\mathcal{O}(n \log n)$ divide-and-conquer method of Chew [18] instead. However, practical inputs are usually composed mainly of short, easily inserted segments. Although Chew's algorithm is optimal, it carries a larger constant overhead than purely Delaunay divide-and-conquer triangulation, and would likely be slower on most practical inputs. I have not implemented Chew's constrained Delaunay triangulation algorithm, and hence cannot test this notion, but I doubt the effort would be worth the ends.

Although the definition of "PSLG" normally disallows segment intersections (except at segment endpoints), Triangle can detect segment intersections and insert vertices appropriately. When Triangle is finding and deleting the triangles that overlap a missing segment, it detects any subsegments that cross the missing segment, and splits each such subsegment by inserting a new vertex at the intersection point. Triangle also notices if a missing segment passes through an existing vertex, and recursively inserts the two subsegments yielded by splitting the segment at the intersecting vertex. However, if a segment passes very close to an existing vertex, but does not meet it precisely (as determined by the exact predicates described in Chapter 6), a very small feature is formed; hence, users should be wary of placing vertices in segment interiors in the hopes that Triangle will deem them collinear. Instead, input segments should be split into smaller input segments at the vertices they are intended to intersect.

Pyramid, unfortunately, does not have the choice of forming a constrained Delaunay tetrahedralization, because constrained Delaunay tetrahedralizations do not always exist. However, subsegments and subfacets can sometimes be introduced into the mesh not by vertex insertion, but by the use of appropriate edge flips. For instance, a 2-3 flip might be used to restore a missing subsegment, and a 3-2 flip might be used to restore a missing subfacet. More generally, a missing subsegment or subfacet might be restored by a clever sequence of flips. However, recall from Chapter 2 that a tetrahedralization that conforms to the missing subsegments and subfacets does not necessarily exist, and the NP-hardness result of Ruppert and Seidel [83] suggests that it might not be feasible to determine whether such a tetrahedralization exists. Hence, one must rely on heuristics when attempting to recover subsegments and subfacets without inserting new vertices. One must resort to inserting a new vertex, in the manner described in Section 4.2, if the heuristics fail.

One might ask, why go to such trouble to avoid inserting new vertices when recovering missing subsegments and subfacets? After all, if a subsegment or subfacet is missing, there must be a vertex in its protecting sphere (except in rare degenerate cases), and the subsegment or subfacet will be split anyway. There are two answers. First, when a subsegment (in two dimensions) or subfacet (in three) has been recovered, its protecting diametral circle or equatorial sphere can be replaced with a diametral lens or equatorial lens, possibly averting a vertex insertion. Second, overrefinement due to small external features, as described in the next section, may be reduced or averted.

5.3.2 Concavities and Holes

In both Triangle and Pyramid, users may create holes in their meshes by specifying *hole points* where an "element-eating virus" is planted and spread by depth-first search until its advance is halted by segments (in two dimensions) or facets (in three). This simple mechanism saves both the user and the implementation from a common outlook of solid modeling wherein one must define oriented curves whose insides are clearly distinguishable from their outsides. Exterior boundaries (which separate a triangulated region from an untriangulated region, and include boundaries of holes) and interior boundaries (which separate two triangulated regions) are treated in a unified manner.

If the region being meshed is not convex, concavities are recognized from unlocked edges (in two dimensions) or faces (in three dimensions) on the boundary of the mesh, and the same element-eating virus is used to hollow them out (recall Figure 5.12). The user may select an option that causes the convex hull of the input to be automatically protected with subsegments or subfacets. If this option is chosen, the user is relieved of the responsibility of providing a segment-bounded or facet-bounded input. Concavities can still be created by specifying appropriate hole points just inside the convex hull, but segments or facets must be used to demarcate the internal boundary of the concavity.

Triangle and Pyramid remove extraneous elements from holes and concavities before the refinement stage. This presents no problem for the refinement algorithms. The main concern is that general point location is difficult in a nonconvex triangulation. Fortunately, the most general form of point location is not needed for Delaunay refinement. Point location is used only to find the circumcenter of an element, which may be accomplished by walking from the centroid of the element toward the circumcenter. If the path is blocked by a subsegment or subfacet, the culprit is marked as encroached, and the search may be abandoned. (Recall that this is precisely how Chew's second Delaunay refinement algorithm [21] decides to split a segment.) Because the mesh is segment-bounded (in two dimensions) or facet-bounded (in three), the search must either succeed or be foiled by an encroached entity. Moreover, in two dimensions, if diametral circles (rather than lenses) are used, Lemma 13 guarantees that any circumcenter considered for insertion falls inside the mesh, although roundoff error might perturb it to just outside the mesh. The analogous result can be proven in three dimensions.

An advantage of removing elements before refinement is that computation is not wasted refining elements that will eventually be deleted. A more important advantage is illustrated in Figure 5.14. If extraneous elements remain during the refinement stage, overrefinement can occur if very small features outside the object being meshed cause the creation of small elements inside the mesh. Ruppert [82] suggests solving this problem by using the constrained Delaunay triangulation and ignoring interactions that take place outside the region being triangulated. Early removal of triangles provides a nearly effortless way to accomplish this effect. Subsegments and subfacets that would normally be considered encroached are ignored (Figure 5.14, right), because encroached subsegments are diagnosed by noticing that they occur opposite an obtuse angle in a triangle. (See the next section for details.)

This advantage can be cast into a formal framework by redefining the notion of local feature size. Let the *geodesic distance* between two points be the length of the shortest path between them that does not pass through an untriangulated region of the plane. In other words, any geodesic path must go around holes and concavities. Given a PSLG X and a point p in the triangulated region of X, define the local feature size $\operatorname{lfs}(p)$ to be the smallest value such that there exist two points u and v that lie on nonincident vertices or segments of X, and each of u and v is within a geodesic distance of $\operatorname{lfs}(p)$ from p. This is essentially the same as the definition of local feature size given in Section 3.3.2, except that Euclidean distances are replaced with geodesic distances. All of the proofs in Chapter 3 can be made to work with geodesic distances, because Lemma 14 depends only upon the triangle inequality, which holds for geodesic distances as well as

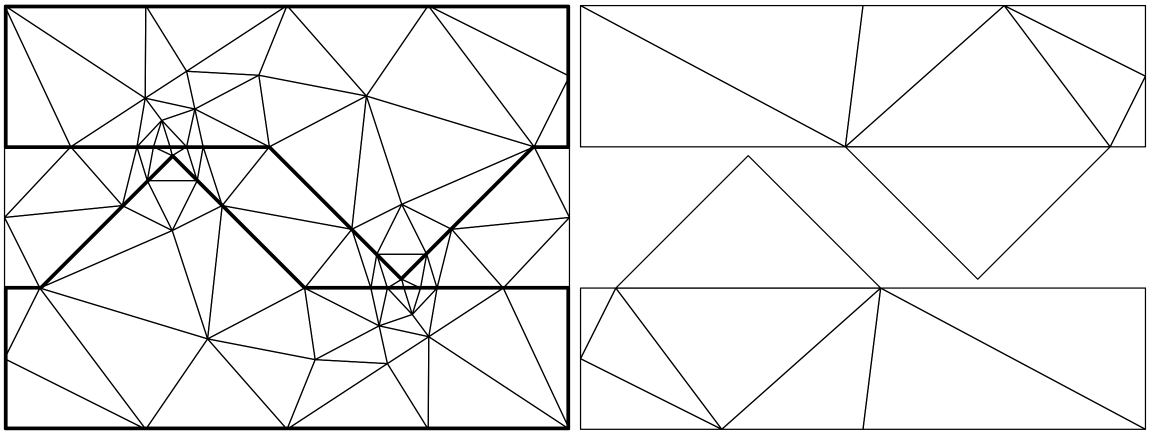


Figure 5.14: Two variations of Ruppert's Delaunay refinement algorithm with a 20° minimum angle. Left: Mesh created using segment recovery by recursive splitting and late removal of triangles. This illustration shows exterior triangles, just prior to their removal, to show why overrefinement occurs. Right: Mesh created using the constrained Delaunay triangulation and early removal of triangles.

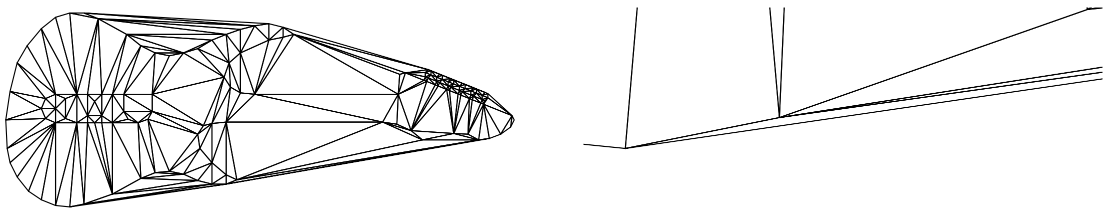
Euclidean distances, and because a child and its parent are never separated by an untriangulated region of the plane. This change can yield improved bounds on edge lengths in some regions of the mesh, because small exterior features are no longer taken into account in the definition of local feature size.

These observations about geodesic distance can also be applied to surface meshing, wherein one meshes two-dimensional planar surfaces embedded in three dimensions. These surfaces may meet at shared segments, so that small feature sizes in one surface may propagate into another. Again, the geodesic distance between two points is the length of the shortest path between them that does not pass through an untriangulated region. Hence, the path is restricted to lie in the input surfaces. Two-dimensional Delaunay refinement algorithms may be applied in this context with virtually no change.

The problem of overrefinement due to small external features is not solved for tetrahedral meshing, however. Constrained Delaunay tetrahedralizations are not an option, and the vertex insertion method for recovering segments and facets can overrefine. However, if missing subsegments and subfacets are given priority over other encroached subsegments and subfacets; if they are recovered with as few vertex insertions as possible (using heuristic methods based on flips, as described in the previous section); and if holes are emptied immediately after all missing subsegments and subfacets are recovered, much or all of the potential overrefinement can be avoided. However, I can offer no guarantee.

Another source of spurious small features is the convex hull of the input, which appears as the boundary edges or faces of the initial triangulation. To give a two-dimensional example, if an input vertex lies just inside the convex hull, and the nearest convex hull edge is treated as a subsegment, then the local feature size near the vertex may be artificially reduced to an arbitrarily small length. In three dimensions, this problem may be caused not only by vertices just inside the convex hull, but also by segments that pass near convex hull edges. These may arise, for instance, when the input is a pre-triangulated surface mesh with exterior dihedral angles that are slightly less than $180°$, just short of convexity. When the input is tetrahedralized, sliver tetrahedra may fill these crevices.

Hence, it is important that convex hull edges and faces are not treated as subsegments and subfacets, except for those edges and faces specifically identified by the user as such. However, if the mesh is not segment-bounded or facet-bounded, it is not clear how to treat exterior skinny triangles or tetrahedra whose circumcenters fall outside the mesh. The removal of elements from concavities yields a segment-bounded



(a) (b)

Figure 5.15: (a) Constrained Delaunay triangulation of the guitar PSLG and its convex hull. (b) Close-up of a small angle formed at the bottom of PSLG because of the convex hull.

or facet-bounded mesh, so that Delaunay refinement may proceed.

Small angles present another motivation for removing elements from holes and concavities prior to applying the Delaunay refinement algorithm. If a small angle is present within a hole or concavity (rather than in the triangulated portion of the PSLG), the small angle has no effect on the meshing process. However, if the mesh were refined prior to the carving of concavities and holes, unnecessarily small elements might be produced, or the refinement stage might fail to terminate. This problem can appear with dastardly stealth when meshing certain nonconvex objects that do not appear to have small angles. A very small angle may be unexpectedly formed between a defining segment of the object and an edge of the convex hull, as illustrated in Figure 5.15. The user, unaware of the effect of the convex hull edge, would be mystified why the Delaunay refinement algorithm fails to terminate on what appears to be an easy PSLG. (In fact, this is how the negative result of Section 3.6 first became evident to me.) Early removal of elements from concavities evades this problem.

In Triangle and Pyramid, the same segment-bounded or facet-bounded depth-first search used to demarcate holes and concavities is also used to tag the elements of selected regions of the mesh with markers that indicate which region they lie in.

5.3.3 Delaunay Refinement

The refinement stage is illustrated on a two-dimensional PSLG in Figure 5.16. As was noted in the previous section, holes and interior boundaries are easily accommodated by the Delaunay refinement algorithm.

Triangle maintains a queue of encroached subsegments and a queue of skinny triangles, each of which are initialized at the beginning of the refinement stage and maintained throughout; every vertex insertion may add new members to either queue. Pyramid maintains queues of encroached subsegments, encroached subfacets, and skinny tetrahedra. The queues of encroached subsegments and subfacets rarely contain more than a few items, except at the beginning of the refinement stage, when they may contain many.

Each queue is initialized by traversing a list of all subsegments, subfacets, triangles, or tetrahedra present in the mesh. Detection of encroached subsegments and subfacets is a local operation. For instance, a subsegment may be tested for encroachment by inspecting only those vertices that appear directly opposite the subsegment in a triangle (a triangular face in three dimensions).

To see why this fact is true, consider Figure 5.17(a). Both of the vertices (v and w) opposite the segment s lie outside the diametral circle of s. Because the mesh is constrained Delaunay, each triangle's circumcircle is empty (on its side of s), and therefore the diametral circle of s is empty. As Figure 5.17(b) shows, the same argument is true of diametral lenses, because a diametral lens is defined by circular arcs passing through a segment's endpoints.

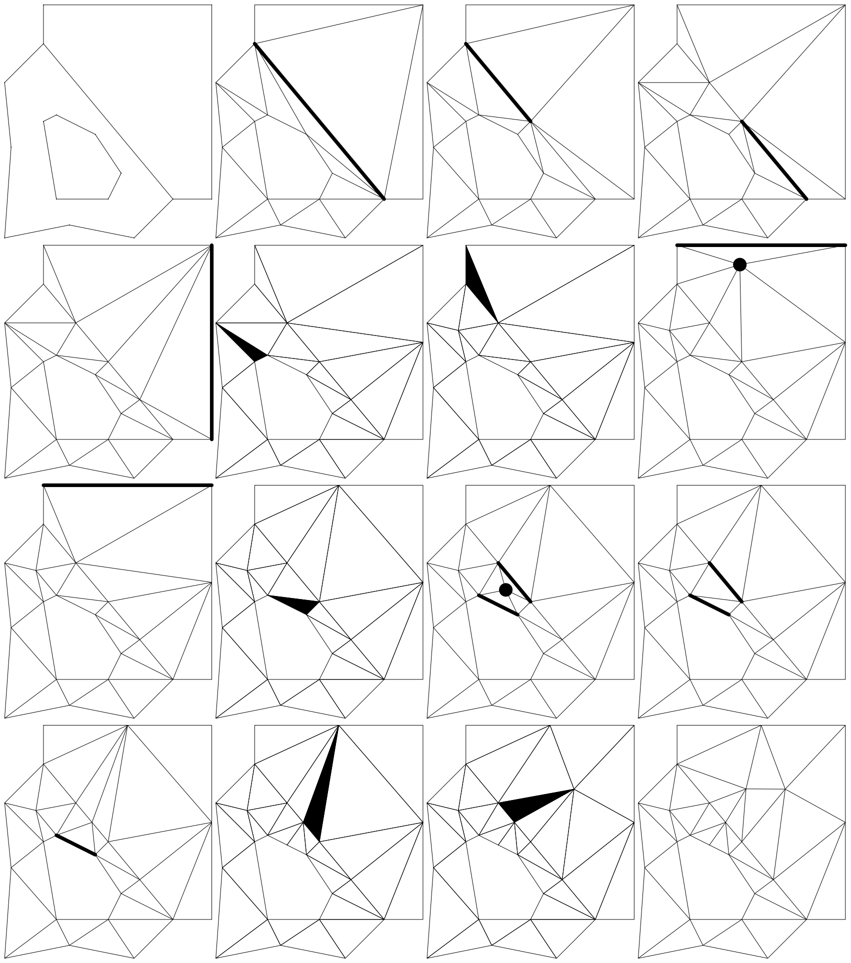


Figure 5.16: Demonstration of the refinement stage. The first two images are the input PSLG and its constrained Delaunay triangulation. In each image, highlighted segments or triangles are about to be split, and highlighted vertices are rejected for insertion. Note that the algorithm easily accommodates interior boundaries and holes.

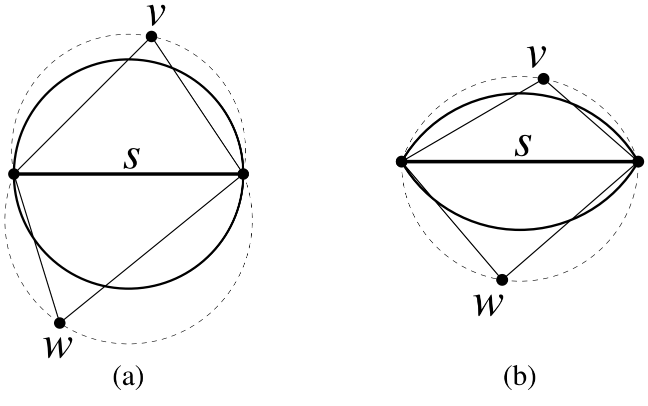


Figure 5.17: (a) If the apices (here, v and w) of the triangles that contain a subsegment s are outside the diametral circle of s, then no vertex lies in the diametral circle of s, because the triangles are Delaunay. (b) The same statement is true for the diametral lens of s.

The same arguments apply in three dimensions to diametral spheres, equatorial spheres, and equatorial lenses. In each case, a subsegment or subfacet may be quickly tested for encroachment by testing only the tetrahedra that contain the subsegment or subfacet in question.

After the queues are initialized, the Delaunay refinement process may cause other subsegments and subfacets to become encroached. The most obvious way to test whether a new vertex encroaches upon some subsegment or subfacet is to insert it into the triangulation, then test each of the edges and faces that appear opposite the vertex in some triangle, tetrahedron, or triangular face. If a subsegment or subfacet is encroached, it is inserted into the appropriate queue, and the new vertex may have to be deleted from the mesh. The decision to accept or reject a vertex depends on the type of vertex being inserted.

- *Subsegment midpoints:* These are never rejected.
- *Subfacet circumcenters:* According to the description of three-dimensional Delaunay refinement given in Chapter 4, these are rejected if they encroach upon a subsegment. However, to obtain the bounds proven in Chapter 4, it is only necessary to reject a subfacet circumcenter if it encroaches upon a subsegment of the same facet. This fact reduces the amount of testing that must be done.
- *Circumcenters of skinny triangles and tetrahedra:* These are rejected if they encroach upon any subsegment or subfacet.

The test for subsegments, if they are protected by diametral circles or spheres, is quite simple. Let t be a triangle formed by a subsegment s and a vertex v opposite it. If the angle at v is greater than 90°, then v encroaches upon s; this test reduces to a dot product. The tests for encroachment of diametral lenses, equatorial spheres, and equatorial lenses are more complicated.

I turn from the topic of detecting encroachment to the topic of managing the queue of skinny elements (which also holds elements that are too large, as dictated by bounds specified by the user). Each time a vertex is inserted or deleted, each new triangle or tetrahedron that appears is tested, and is inserted into the queue if its quality is too poor, or its area or volume too large. The number of triangles or tetrahedra in the final mesh is determined in part by the order in which skinny elements are split, especially when a strong quality bound is used. Figure 5.18 demonstrates how sensitive Ruppert's algorithm is to the order. For this example with a 33° minimum angle, a heap of skinny triangles indexed by their smallest angle confers a 35% reduction in mesh size over a first-in first-out queue. (This difference is typical for strong angle bounds, but thankfully seems to disappear for small angle bounds.) The discrepancy probably occurs because circumcenters of very skinny triangles are likely to eliminate more skinny triangles than circumcenters of mildly skinny triangles. Unfortunately, a heap is slow for large meshes, especially when small area constraints force all of the elements into the heap. Delaunay refinement usually takes $\mathcal{O}(n)$ time in practice, but the use of a heap increases the complexity to $\mathcal{O}(n \log n)$.

The solution used in Triangle and Pyramid, chosen experimentally, is to use 64 FIFO queues, each representing a different interval of circumradius-to-shortest edge ratios. Oddly, it is counterproductive in practice to order well-shaped elements, so one queue is used for well-shaped but too-large elements whose quality ratios are all roughly smaller than 0.8 (in Triangle, corresponding to an angle of about 39°) or one (in Pyramid). Elements with larger quality ratios are partitioned among the remaining queues. When a skinny element is chosen for splitting, it is taken from the "worst" nonempty queue. A queue of nonempty queues is maintained so that a skinny element may be chosen quickly. This method yields meshes comparable with those generated using a heap, but is only slightly slower than using a single queue.

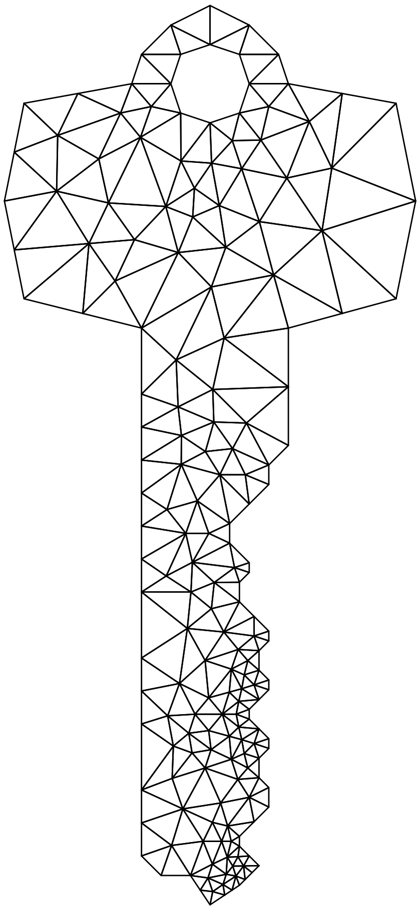
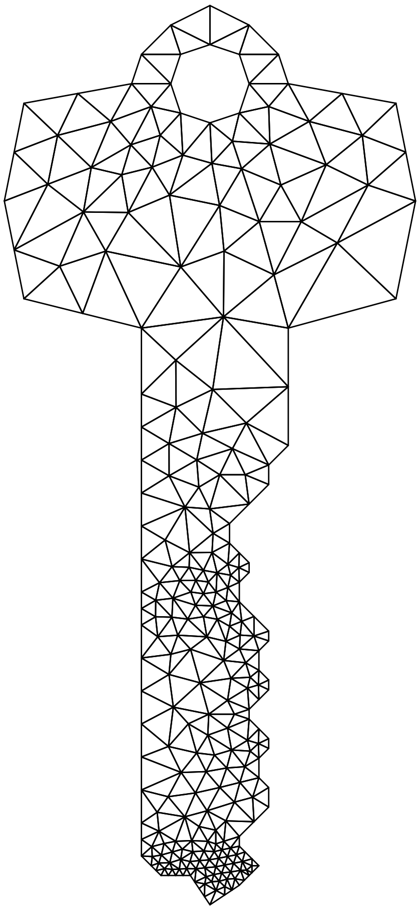



Figure 5.18: Two meshes with a 33° minimum angle. The left mesh, with 290 triangles, was formed by always splitting the worst existing triangle. The right mesh, with 450 triangles, was formed by using a first-come first-split queue of skinny triangles.

During the refinement phase, Triangle generates about 22,800 new vertices per second on a DEC 3000/700. Pyramid generates a more modest 800 vertices per second. Vertices are inserted using a flip-based incremental Delaunay algorithm, but in Triangle are inserted much more quickly than Table 5.1 would suggest because a triangle's circumcenter can be located quickly by starting the search at the triangle.

5.4 Conclusions

Triangle and Pyramid were originally designed and implemented to meet the needs of the Quake project [7] at Carnegie Mellon University, a multidisciplinary effort to study earthquake-induced ground motion in the Los Angeles basin. Such a study is necessarily of large magnitude, and exerts a great deal of stress on the software infrastructure that supports it. Triangle and Pyramid have risen to the challenge, generating meshes of up to 77 million tetrahedral elements. As well as using Triangle within the Quake project, I have released it for public use, and expect to release Pyramid in a similar manner within the next year.

In the two years since Triangle was released to the public, I have heard from researchers and developers who are using Triangle for a surprising variety of applications. Triangle seems to be particularly popular for triangulating survey and map data; for maintaining terrain databases, especially for use in real-time simulations; and for discontinuity meshing for global illumination methods such as radiosity. I have also heard from individuals using Triangle for more surprising applications, such as stereo vision, interpolation of speech signals, computing the orientation of text images, modeling reflections of high frequency radio from structures in cities, modeling the density of stars in the sky, the triangulation of virtual worlds for a video game, and ecological research culminating in a paper by N. V. Joshi, entitled "The Spatial Organization of Plant Communities in a Deciduous Forest."

Of course, Triangle has also been used by many researchers for numerical simulation. Applications include electrical current propagation in the myocardium, simulation of surgery on a model of the human cornea, transport processes in estuaries and coastal waters, tomographic models of the seismic structure beneath Eurasia, Schrödinger's equation in quantum confined structures, two-and-a-half-dimensional waveguide problems, electrostatic and magnetostatic modeling of complex multielectrode systems, control volume FEM for fluid flow and heat transfer, and surface meshing for BEM on integrated circuits.

These examples represent only a selected few of the applications that people have written to tell me about, which in turn surely represent only a fraction of the people who are using Triangle. Several companies have also purchased licenses to use Triangle in their commercial products, for purposes including beam element visualization, thermal analysis, interpolation between grids for an ocean floor database, visualization of mining data, and cartoon animation.

The upshot is that there has long been an unanswered need for robust mesh generation in a great variety of application domains. Although there was triangular meshing software available freely on the net prior to Triangle, many of my users report that none had the combination of flexibility, robustness, and ease of use of Triangle.

Chapter 6

Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates

From the beginning of the implementation of Triangle, and well into the development of Pyramid, floating-point roundoff problems plagued me. Each program would sometimes crash, sometimes find itself stuck in an endless loop, and sometimes produce garbled output. At first I believed that I would be able to fix the problems by understanding how the algorithms went wrong when roundoff error produced incorrect answers, and writing special-case code to handle each potential problem. Some of the robustness problems yielded to this approach, but others did not. Fortunately, Steven Fortune of AT&T Bell Laboratories convinced me, in a few brief but well-worded email messages (and in several longer and equally well-worded technical papers), to choose the alternative path to robustness, which led to the research described in this chapter. For reasons that will become apparent, exact arithmetic is the better approach to solving many, if not all, of the robustness worries associated with triangulation.

Herein, I make three contributions to geometric robustness, the first two of which I hope will find application elsewhere in numerical analysis. The first is to offer fast software-level algorithms for exact addition and multiplication of arbitrary precision floating-point values. The second is to propose a technique for adaptive precision arithmetic that can often speed these algorithms when one wishes to perform multiprecision calculations that do not always require exact arithmetic, but must satisfy some error bound. The third is to provide a practical demonstration of these techniques, in the form of implementations of several common geometric calculations whose required degree of accuracy depends on their inputs. These robust geometric predicates are adaptive; their running time depends on the degree of uncertainty of the result, and is usually small.

These algorithms work on computers whose floating-point arithmetic uses radix two and exact rounding, including machines complying with the IEEE 754 standard. The inputs to the predicates may be arbitrary single or double precision floating-point numbers. C code is publicly available for the 2D and 3D orientation and incircle tests, and is used very successfully in both Triangle and Pyramid. Timings of the implementations demonstrate their effectiveness.

6.1 Introduction

Software libraries for arbitrary precision floating-point arithmetic can be used to accurately perform many error-prone or ill-conditioned computations that would be infeasible using only hardware-supported approximate arithmetic. Some of these computations have accuracy requirements that vary with their input. For instance, consider the problem of finding the center of a circle, given three points that lie on the circle. Normally, hardware precision arithmetic will suffice, but if the input points are nearly collinear, the problem is ill-conditioned and the approximate calculation may yield a wildly inaccurate result or a division by zero. Alternatively, an exact arithmetic library can be used and will yield a correct result, but exact arithmetic is slow; one would rather use it only when one really needs to.

This chapter presents two techniques for writing fast implementations of extended precision calculations like these, and demonstrates them with implementations of four commonly used geometric predicates. The first technique is a suite of algorithms, several of them new, for performing arbitrary precision arithmetic. The method has its greatest advantage in computations that process values of extended but small precision (several hundred or thousand bits), and seems ideal for computational geometry and some numerical methods, where much benefit can be realized from a modest increase in precision. The second technique is a way to modify these algorithms so that they compute their result adaptively; they are quick in most circumstances, but are still slow when their results are prone to have high relative error. A third subject of this chapter is a demonstration of these techniques with implementations and performance measurements of four commonly used geometric predicates. An elaboration of each of these three topics follows.

Methods of simulating exact arithmetic in software can be classified by several characteristics. Some exact arithmetic libraries operate on integers or fixed-point numbers, while others operate on floating-point numbers. To represent a number, the former libraries store a significand of arbitrary length; the latter store an exponent as well. Some libraries use the hardware's integer arithmetic units, whereas others use the floating-point units. Oddly, the decision to use integers or floating-point numbers internally is orthogonal to the type of number being represented. It was once the norm to use integer arithmetic to build extended precision floating-point libraries, especially when floating-point hardware was uncommon and differed between computer models. Times have changed, and modern architectures are highly optimized for floating-point performance; on many processors, floating-point arithmetic is faster than integer arithmetic. The trend is reversing for software libraries as well, and there are several proposals to use floating-point arithmetic to perform extended-precision integer calculations. Fortune and Van Wyk [37, 36], Clarkson [23], and Avnaim, Boissonnat, Devillers, Preparata, and Yvinec [2] have described algorithms of this kind, designed to attack the same computational geometry robustness problems considered later in this chapter. These algorithms are surveyed in Section 6.2.

Another differentiating feature of multiprecision libraries is whether they use multiple exponents. Most arbitrary precision libraries store numbers in a *multiple-digit* format, consisting of a sequence of digits (usually of large radix, like 2^{32}) coupled with a single exponent. A freely available example of the multiple-digit approach is Bailey's MPFUN package [4], a sophisticated portable multiprecision library that uses digits of machine-dependent radix (usually 2^{24}) stored as single precision floating-point values. An alternative is the *multiple-component* format, wherein a number is expressed as a sum of ordinary floating-point words, each with its own significand and exponent [76, 26, 61]. This approach has the advantage that the result of an addition like $2^{300} + 2^{-300}$ (which may well arise in calculations like the geometric predicates discussed in Section 6.5.1) can be stored in two words of memory, whereas the multiple-digit approach will use at least 601 bits to store the sum, and incur a corresponding speed penalty when performing arithmetic with it. On the other hand, the multiple-digit approach can more compactly represent most numbers, because only one exponent is stored. (MPFUN sacrifices this compactness to take advantage of floating-point hardware; the exponent of each digit is unused.) More pertinent is the difference in speed, discussed briefly in Section 6.3.1.

The algorithms described herein use floating-point hardware to perform extended precision floating-point arithmetic, using the multiple-component approach. These algorithms, described in Section 6.3, work under the assumption that hardware arithmetic is performed in radix two with exact rounding. This assumption holds on processors compliant with the IEEE 754 floating-point standard. Proofs of the correctness of all algorithms are given.

The methods herein are closely related to, and occasionally taken directly from, methods developed by Priest [76, 77], but are faster. The improvement in speed arises partly because Priest's algorithms run on a wide variety of floating-point architectures, with different radices and rounding behavior, whereas mine are limited to and optimized for radix two with exact rounding. This specialization is justified by the wide acceptance of the IEEE 754 standard. My algorithms also benefit from a relaxation of Priest's normalization requirement, which is less strict than the normalization required by multiple-digit algorithms, but is nonetheless time-consuming to enforce.

I demonstrate these methods with publicly available code that performs the two-dimensional and three-dimensional orientation and incircle tests, calculations that commonly arise in computational geometry. The orientation test determines whether a point lies to the left of, to the right of, or on a line or plane; it is an important predicate used in many (perhaps most) geometric algorithms. The incircle test determines whether a point lies inside, outside, or on a circle or sphere, and is used for Delaunay triangulation. Inexact versions of these tests are vulnerable to roundoff error, and the wrong answers they produce can cause geometric algorithms to hang, crash, or produce incorrect output. Although exact arithmetic banishes these difficulties, it is common to hear reports of implementations being slowed by factors of ten or more as a consequence [56, 36]. For these reasons, computational geometry is an important arena for evaluating extended precision arithmetic schemes.

The orientation and incircle tests evaluate the sign of a matrix determinant. It is significant that only the sign, and not the magnitude, of the determinant is needed. Fortune and Van Wyk [36] take advantage of this fact by using a floating-point filter: the determinant is first evaluated approximately, and only if forward error analysis indicates that the sign of the approximate result cannot be trusted does one use an exact test. I carry their suggestion to its logical extreme by computing a sequence of successively more accurate approximations to the determinant, stopping only when the accuracy of the sign is assured. To reduce computation time, approximations reuse a previous, less accurate computation when it is economical to do so. Procedures thus designed are adaptive; they refine their results until they are certain of the correctness of their answer. The technique is not limited to computational geometry, nor is it limited to finding signs of expressions; it can be employed in any calculation where the required degree of accuracy varies. This adaptive approach is described in Section 6.4, and its application to the orientation and incircle tests is described in Section 6.5.

Readers who wish to use these predicates in their own applications are encouraged to download them and try them out. However, be certain to read Section 6.6, which covers two important issues that must be considered to ensure the correctness of the implementation: your processor's floating-point behavior and your compiler's optimization behavior. Furthermore, be aware that exact arithmetic is not a panacea for all robustness woes; its uses and limitations are discussed in Section 6.2. Exact arithmetic can make robust many algorithms that take geometric input and return purely combinatorial output; for instance, a fully robust convex hull implementation can be produced with recourse only to an exact orientation test. However, in algorithms that construct new geometric objects, exact arithmetic is sometimes constrained by its cost and its inability to represent arbitrary irrational numbers.

A few words are appropriate to describe some of the motivation for pursuing robust predicates for floating-point, rather than integer, operands. One might argue that real-valued input to a geometric program can be scaled and approximated in integer form. Indeed, there are few geometric problems that truly require the range of magnitude that floating-point storage provides, and integer formats had a clear speed advantage over floating-point formats for small-scale exact computation prior to the present research. The best argument for exact floating-point libraries in computational geometry, besides convenience, is the fact that many existing geometric programs already use floating-point numbers internally, and it is easier to replace their geometric predicates with robust floating-point versions than to retrofit the programs to use integers throughout. Online algorithms present another argument, because they are not always compatible with the scaled-input approach. One cannot always know in advance what resolution will be required, and repeated rescalings may be necessary to support an internal integer format when the inputs are real and unpredictable. In any case, I hope that this research will make it easier for programmers to choose between integer and floating-point arithmetic as they prefer.

6.2 Related Work in Robust Computational Geometry

Most geometric algorithms are not originally designed for robustness at all; they are based on the *real RAM model*, in which quantities are allowed to be arbitrary real numbers, and all arithmetic is exact. There are several ways a geometric algorithm that is correct within the real RAM model can go wrong in an encounter with roundoff error. The output might be incorrect, but be correct for some perturbation of its input. The result might be usable yet not be valid for any imaginable input. Or, the program may simply crash or fail to produce a result. To reflect these possibilities, geometric algorithms are divided into several classes with varying amounts of robustness: *exact algorithms*, which are always correct; *robust algorithms*, which are always correct for some perturbation of the input; *stable algorithms*, for which the perturbation is small; *quasi-robust algorithms*, whose results might be geometrically inconsistent, but nevertheless satisfy some weakened consistency criterion; and *fragile algorithms*, which are not guaranteed to produce any usable output at all. The next several pages are devoted to a discussion of representative research in each class, and of the circumstances in which exact arithmetic and other techniques are or are not applicable. For more extensive surveys of geometric robustness, see Fortune [34] and Hoffmann [50].

Exact algorithms. A geometric algorithm is *exact* if it is guaranteed to produce a correct result when given an exact input. (Of course, the input to a geometric algorithm may only be an approximation of some real-world configuration, but this difficulty is ignored here.) Exact algorithms use exact arithmetic in some form, whether in the form of a multiprecision library or in a more disguised form.

There are several exact arithmetic schemes designed specifically for computational geometry; most are methods for exactly evaluating the sign of a determinant, and hence can be used to perform the orientation and incircle tests. Clarkson [23] proposes an algorithm for using floating-point arithmetic to evaluate the sign of the determinant of a small matrix of integers. A variant of the modified Gram-Schmidt procedure is used to improve the conditioning of the matrix, so that the determinant can subsequently be evaluated safely by Gaussian elimination. The 53 bits of significand available in IEEE double precision numbers are sufficient to operate on 10×10 matrices of 32-bit integers. Clarkson's algorithm is naturally adaptive; its running time is small for matrices whose determinants are not near zero[1].

[1]The method presented in Clarkson's paper does not work correctly if the determinant is exactly zero, but Clarkson (personal communication) notes that it is easily fixed. "By keeping track of the scaling done by the algorithm, an upper bound can be maintained for the magnitude of the determinant of the matrix. When that upper bound drops below one, the determinant must be zero, since the matrix entries are integers, and the algorithm can stop."

Recently, Avnaim, Boissonnat, Devillers, Preparata, and Yvinec [2] proposed an algorithm to evaluate signs of determinants of 2×2 and 3×3 matrices of p-bit integers using only p and $(p+1)$-bit arithmetic, respectively. Surprisingly, this is sufficient even to implement the insphere test (which is normally written as a 4×4 or 5×5 determinant), but with a handicap in bit complexity; 53-bit double precision arithmetic is sufficient to correctly perform the insphere test on points having 24-bit integer coordinates.

Fortune and Van Wyk [37, 36] propose a more general approach (not specific to determinants, or even to predicates) that represents integers using a standard multiple-digit technique with digits of radix 2^{23} stored as double precision floating-point values. (53-bit double precision significands make it possible to add several products of 23-bit integers before it becomes necessary to normalize.) Rather than use a general-purpose arbitrary precision library, they have developed LN, an expression compiler that writes code to evaluate a specific expression exactly. The size of the operands is arbitrary, but is fixed when LN is run; an expression can be used to generate several functions, each for arguments of different bit lengths. Because the expression and the bit lengths of all operands are fixed in advance, LN can tune the exact arithmetic aggressively, eliminating loops, function calls, and memory management. The running time of a function produced by LN depends on the bit complexity of the inputs. Fortune and Van Wyk report an order-of-magnitude speed improvement over the use of multiprecision libraries (for equal bit complexity). Furthermore, LN gains another speed improvement by installing floating-point filters wherever appropriate, calculating error bounds automatically.

Karasick, Lieber, and Nackman [56] report their experiences optimizing a method for determinant evaluation using rational inputs. Their approach reduces the bit complexity of the inputs by performing arithmetic on intervals (with low precision bounds) rather than exact values. The determinant thus evaluated is also an interval; if it contains zero, the precision is increased and the determinant reevaluated. The procedure is repeated until the interval does not contain zero (or contains only zero), and the result is certain. Their approach is thus adaptive, although it does not appear to use the results of one iteration to speed the next.

Because the Clarkson and Avnaim et al. algorithms are effectively restricted to low precision integer coordinates, I do not compare their performance with that of my algorithms, though theirs may be faster. Floating-point inputs are more difficult to work with than integer inputs, partly because of the potential for the bit complexity of intermediate values to grow more quickly. (The Karasick et al. algorithm also suffers this difficulty, and is probably not competitive with the other techniques discussed here, although it may be the best existing alternative for algorithms that require rational numbers, such as those computing exact line intersections.) When it is necessary for an algorithm to use floating-point coordinates, the aforementioned methods are not currently an option (although it might be possible to adapt them using the techniques of Section 6.3). I am not aware of any prior literature on exact determinant evaluation that considers floating-point operands, except for one limited example: Ottmann, Thiemt, and Ullrich [74] advocate the use of an *accurate scalar product* operation, ideally implemented in hardware (though the software-level distillation algorithm described in Section 6.3.8 may also be used), as a way to evaluate some predicates such as the 2D orientation test.

Exact determinant algorithms do not satisfy the needs of all applications. A program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored. If the intersections may themselves become endpoints of lines that generate more intersections, then intersections of greater and greater bit complexity may be generated. Even exact rational arithmetic is not sufficient for all applications; a solid modeler, for instance, might need to determine the vertices of the intersection of two independent solids that have been rotated through arbitrary angles. Yet exact floating-point arithmetic can't even cope with rotating a square 45° in the plane, because irrational vertex coordinates result. The problem of constructed irrational values has been partly attacked by the implementation of "real" numbers in the LEDA library of algorithms [13]. Values derived from square roots (and other arithmetic operations) are stored in symbolic form when necessary. Comparisons with such numbers are resolved with great numerical care, albeit sometimes at great cost; separation bounds are computed where necessary to ensure that the sign of an expression is determined accurately. Floating-point filters and another form of adaptivity (approximating a result repeatedly, doubling the precision each time) are used as well.

For the remainder of this discussion, consideration is restricted to algorithms whose input is geometric (e.g. coordinates are specified) but whose output is purely combinatorial, such as the construction of a convex hull or an arrangement of hyperplanes.

Robust algorithms. There are algorithms that can be made correct with straightforward implementations of exact arithmetic, but suffer an unacceptable loss of speed. An alternative is to relax the requirement of a correct solution, and instead accept a solution that is "close enough" in some sense that depends upon the application. Without exact arithmetic, an algorithm must somehow find a way to produce sensible output despite the fact that geometric tests occasionally tell it lies. No general techniques have emerged yet, although bandages have appeared for specific algorithms, usually ensuring robustness or quasi-robustness through painstaking design and error analysis. The lack of generality of these techniques is not the only limitation of the relaxed approach to robustness; there is a more fundamental difficulty that deserves careful discussion.

When disaster strikes and a real RAM-correct algorithm implemented in floating-point arithmetic fails to produce a meaningful result, it is often because the algorithm has performed tests whose results are mutually contradictory. Figure 6.1 shows an error that arose in the triangulation merging subroutine of Triangle's divide-and-conquer Delaunay triangulation implementation. The geometrically nonsensical triangulation in the illustration was produced.

On close inspection with a debugger, I found that the failure was caused by a single incorrect result of the incircle test. At the bottom of Figure 6.1 appear four nearly collinear points whose deviation from collinearity has been greatly exaggerated for clarity. The points a, b, c, and d had been sorted by their x-coordinates, and b had been correctly established (by orientation tests) to lie below the line ac and above the line ad. In principle, a program could deduce from these facts that a cannot fall inside the circle dcb. Unfortunately, the incircle test incorrectly declared that a lay inside, thereby leading to the invalid result.

It is significant that the incircle test was not just wrong about these particular points; it was inconsistent with the "known combinatorial facts." A correct algorithm (that computes a purely combinatorial result) will produce a meaningful result if its test results are wrong but are consistent with each other, because there exists an input for which those test results are correct. Following Fortune [32], an algorithm is *robust* if it always produces the correct output under the real RAM model, and under approximate arithmetic always produces an output that is consistent with some hypothetical input that is a perturbation of the true input; it is *stable* if this perturbation is small. Typically, bounds on the perturbation are proven by backward error analysis. Using only approximate arithmetic, Fortune gives an algorithm that computes a planar convex hull that is correct for points that have been perturbed by a relative error of at most $\mathcal{O}(\epsilon)$ (where ϵ is the machine epsilon, defined in Section 6.4.2), and an algorithm that maintains a triangulation that can be made planar by perturbing each vertex by a relative error of at most $\mathcal{O}(n^2\epsilon)$, where n is the number of vertices. If it seems surprising that a "stable" algorithm cannot keep a triangulation planar, consider the problem of inserting a new vertex so close to an existing edge that it is difficult to discern which side of the edge the vertex falls on. Only exact arithmetic can prevent the possibility of creating an "inverted" triangle.

One might wonder if my triangulation program can be made robust by avoiding any test whose result can be inferred from previous tests. Fortune [32] explains that

> [a]n algorithm is *parsimonious* if it never performs a test whose outcome has already been determined as the formal consequence of previous tests. A parsimonious algorithm is clearly robust,

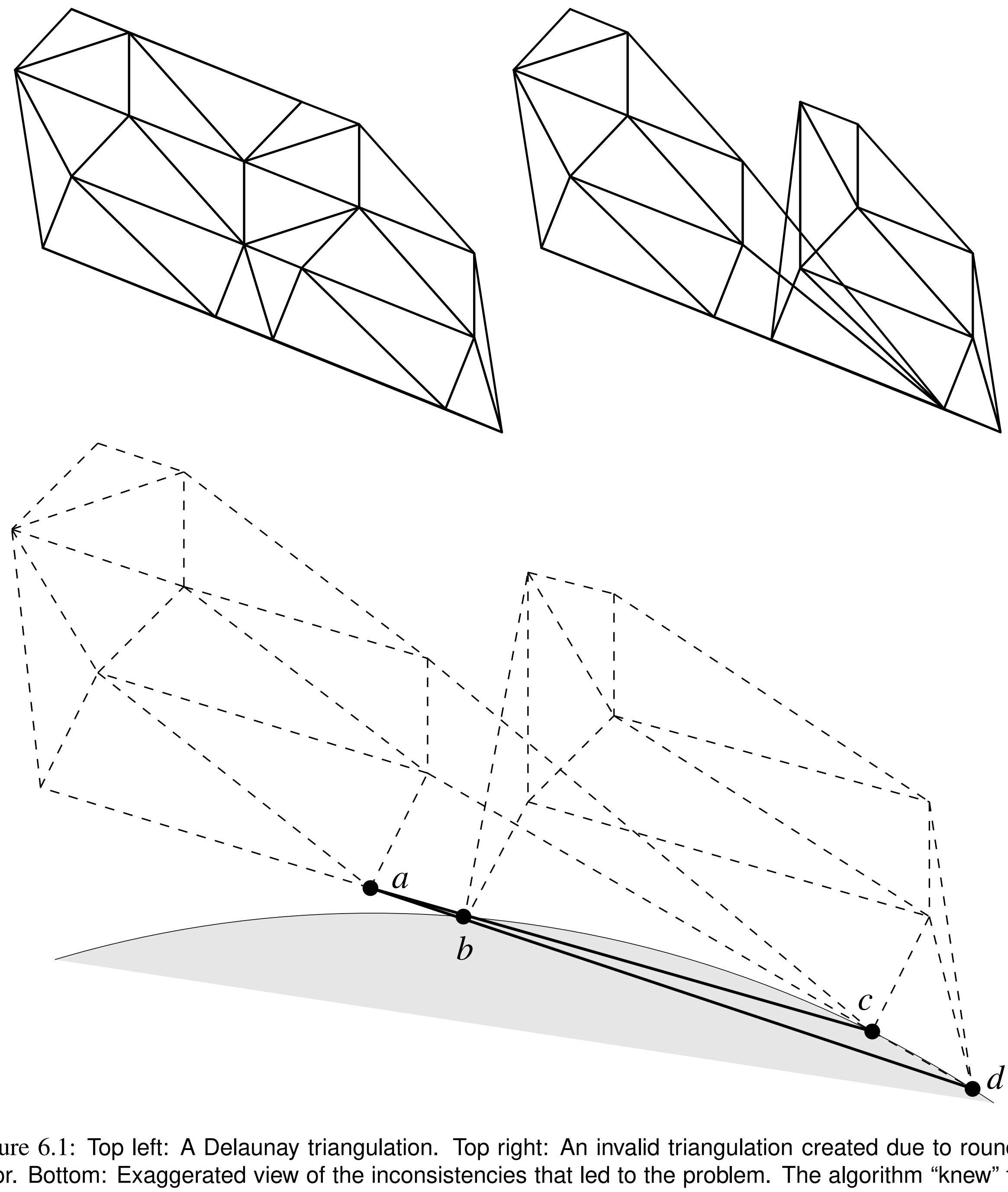


Figure 6.1: Top left: A Delaunay triangulation. Top right: An invalid triangulation created due to roundoff error. Bottom: Exaggerated view of the inconsistencies that led to the problem. The algorithm "knew" that the point b lay between the lines ac and ad, but an incorrect incircle test claimed that a lay inside the circle dcb.

since any path through the algorithm must correspond to some geometric input; making an algorithm parsimonious is the most obvious way of making it robust. In principle it is possible to make an algorithm parsimonious: since all primitive tests are polynomial sign evaluations, the question of whether the current test is a logical consequence of previous tests can be phrased as a statement of the existential theory of the reals. This theory is at least NP-hard and is decidable in polynomial space [15]. Unfortunately, the full power of the theory seems to be necessary for

> some problems. An example is the *line arrangement problem*: given a set of lines (specified by real coordinates (a, b, c), so that $ax + by = c$), compute the combinatorial structure of the resulting arrangement in the plane. It follows from recent work of Mnev [71] that the problem of deciding whether a combinatorial arrangement is actually realizable with lines is as hard as the existential theory of the reals. Hence a parsimonious algorithm for the line arrangement problem . . . seems to require the solution of NP-hard problems.

Because exact arithmetic does not require the solution of NP-hard problems, an intermediate course is possible; one could employ parsimony whenever it is efficient to do so, and resort to exact arithmetic otherwise. Consistency is guaranteed if exact tests are used to bootstrap the "parsimony engine." I am not aware of any algorithms in the literature that take this approach, although geometric algorithms are often designed by their authors to avoid the more obviously redundant tests.

Quasi-robust algorithms. The difficulty of determining whether a line arrangement is realizable suggests that, without exact arithmetic, robustness as defined above may be an unattainable goal. However, sometimes one can settle for an algorithm whose output might not be realizable. I place such algorithms in a bag labeled with the fuzzy term *quasi-robust*, which I apply to any algorithm whose output is somehow provably distinguishable from nonsense. Milenkovic [65] circumvents the aforementioned NP-hardness result while using approximate arithmetic by constructing pseudo-line arrangements; a *pseudo-line* is a curve constrained to lie very close to an actual line. Fortune [35] presents a 2D Delaunay triangulation algorithm that constructs, using approximate arithmetic, a triangulation that is nearly Delaunay in a well-defined sense using the pseudo-line-like notion of pseudocircles. Unfortunately, the algorithm's running time is $\mathcal{O}(n^2)$, which compares poorly with the $\mathcal{O}(n \log n)$ time of optimal algorithms. Milenkovic's and Fortune's algorithms are both *quasi-stable*, having small error bounds. Milenkovic's algorithm can be thought of as a quasi-robust algorithm for line arrangements, or as a robust algorithm for pseudo-line arrangements.

Barber [8] pioneered an approach in which uncertainty, including the imprecision of input data, is a part of each geometric entity. *Boxes* are structures that specify the location and the uncertainty in location of a vertex, edge, facet, or other geometric structure. Boxes may arise either as input or as algorithmic constructions; any uncertainty resulting from roundoff error is incorporated into their shapes and sizes. Barber presents algorithms for solving the point-in-polygon problem and for constructing convex hulls in any dimension. For the point-in-polygon problem, "can't tell" is a valid answer if the uncertainty inherent in the input or introduced by roundoff error prevents a sure determination. The salient feature of Barber's Quickhull convex hull algorithm is that it merges hull facets that cannot be guaranteed (through error analysis) to be clearly locally convex. The *box complex* produced by the algorithm is guaranteed to contain the true convex hull, bounding it, if possible, both from within and without.

The degree of robustness required of an algorithm is typically determined by how its output is used. For instance, many point location algorithms can fail when given a non-planar triangulation. For this very reason, my triangulator crashed after producing the flawed triangulation in Figure 6.1.

The reader should take three lessons from this section. First, problems due to roundoff can be severe and difficult to solve. Second, even if the inputs are imprecise and the user isn't picky about the accuracy of the output, internal consistency may still be necessary if any output is to be produced at all; exact arithmetic may be required even when exact results aren't. Third, neither exact arithmetic nor clever handling of tests that tell falsehoods is a universal balm. However, exact arithmetic is attractive when it is applicable, because it can be employed by naïve program developers without the time-consuming need for careful analysis of a particular algorithm's behavior when faced with imprecision. (I occasionally hear of implementations where more than half the developers' time is spent solving problems of roundoff error and degeneracy.) Hence, efforts to improve the speed of exact arithmetic in computational geometry are well justified.

6.3 Arbitrary Precision Floating-Point Arithmetic

6.3.1 Background

Most modern processors support floating-point numbers of the form $\pm$*significand* $\times\ 2^{exponent}$. The significand is a p-bit binary number of the form $b.bbb\ldots$, where each b denotes a single bit; one additional bit represents the sign. This research does not address issues of overflow and underflow, so I allow the exponent to be an integer in the range $[-\infty, \infty]$. (Fortunately, many applications have inputs whose exponents fall within a circumscribed range. The four predicates implemented for this chapter will not overflow nor underflow if their inputs have exponents in the range $[-142, 201]$ and IEEE 754 double precision arithmetic is used.) Floating-point values are generally *normalized*, which means that if a value is not zero, then its most significant bit is set to one, and the exponent adjusted accordingly. For example, in four-bit arithmetic, binary 1101 (decimal 13) is represented as 1.101×2^3. See the survey by Goldberg [44] for a detailed explanation of floating-point storage formats, particularly the IEEE 754 standard.

Exact arithmetic often produces values that require more than p bits to store. For the algorithms herein, each arbitrary precision value is expressed as an *expansion*[2] $x = x_n + \cdots + x_2 + x_1$, where each x_i is called a *component* of x and is represented by a floating-point value with a p-bit significand. To impose some structure on expansions, they are required to be *nonoverlapping* and ordered by magnitude (x_n largest, x_1 smallest). Two floating-point values x and y are nonoverlapping if the least significant nonzero bit of x is more significant than the most significant nonzero bit of y, or vice versa; for instance, the binary values 1100 and -10.1 are nonoverlapping, whereas 101 and 10 overlap.[3] The number zero does not overlap any number. An expansion is nonoverlapping if all its components are mutually nonoverlapping. Note that a number may be represented by many possible nonoverlapping expansions; consider $1100 + -10.1 = 1001 + 0.1 = 1000 + 1 + 0.1$. A nonoverlapping expansion is desirable because it is easy to determine its sign (take the sign of the largest component) or to produce a crude approximation of its value (take the component with largest magnitude).

Two floating-point values x and y are *adjacent* if they overlap, if x overlaps $2y$, or if $2x$ overlaps y. For instance, 1100 is adjacent to 11, but 1000 is not. An expansion is *nonadjacent* if no two of its components are adjacent. Surprisingly, any floating-point value has a corresponding nonadjacent expansion; for instance, 11111 may appear at first not to be representable as a nonoverlapping expansion of one-bit components, but consider the expansion $100000 + -1$. The trick is to use the sign bit of each component to separate it from its larger neighbor. We will later see algorithms in which nonadjacent expansions arise naturally.

Multiple-component algorithms (based on the expansions defined above) can be faster than multiple-digit algorithms because the latter require expensive normalization of results to fixed digit positions, whereas multiple-component algorithms can allow the boundaries between components to wander freely. Boundaries are still enforced, but can fall at any bit position. In addition, it usually takes time to convert an ordinary floating-point number to the internal format of a multiple-digit library, whereas any ordinary floating-point number *is* an expansion of length one. Conversion overhead can account for a significant part of the cost of small extended precision computations.

The central conceptual difference between standard multiple-digit algorithms and the multiple-component algorithms described herein is that the former perform exact arithmetic by keeping the bit complexity of operands small enough to avoid roundoff error, whereas the latter allow roundoff to occur, then account for

[2] Note that this definition of *expansion* is slightly different from that used by Priest [76]; whereas Priest requires that the exponents of any two components of the expansion differ by at least p, no such requirement is made here.

[3] Formally, x and y are nonoverlapping if there exist integers r and s such that $x = r2^s$ and $|y| < 2^s$, or $y = r2^s$ and $|x| < 2^s$.

it after the fact. To measure roundoff quickly and correctly, a certain standard of accuracy is required from the processor's floating-point units. The algorithms presented herein rely on the assumption that addition, subtraction, and multiplication are performed with *exact rounding*. This means that if the exact result can be stored in a p-bit significand, then the exact result is produced; if it cannot, then it is rounded to the nearest p-bit floating-point value. For instance, in four-bit arithmetic the product $111 \times 101 = 100011$ is rounded to 1.001×2^5. If a value falls precisely halfway between two consecutive p-bit values, a tiebreaking rule determines the result. Two possibilities are the round-to-even rule, which specifies that the value should be rounded to the nearest p-bit value with an even significand, and the round-toward-zero rule. In four-bit arithmetic, 10011 is rounded to 1.010×2^4 under the round-to-even rule, and to 1.001×2^4 under the round-toward-zero rule. The IEEE 754 standard specifies round-to-even tiebreaking as a default. Throughout this chapter, the symbols $\oplus$, $\ominus$, and $\otimes$ represent p-bit floating-point addition, subtraction, and multiplication with exact rounding. Due to roundoff, these operators lack several desirable arithmetic properties. Associativity is an example; in four-bit arithmetic, $(1000 \oplus 0.011) \oplus 0.011 = 1000$, but $1000 \oplus (0.011 \oplus 0.011) = 1001$. A list of reliable identities for floating-point arithmetic is given by Knuth [57].

Roundoff is often analyzed in terms of *ulps*, or "units in the last place." An ulp is the effective magnitude of the low-order (pth) bit of a p-bit significand. An ulp is defined relative to a specific floating point value; I shall use $\text{ulp}(a)$ to denote this quantity. For instance, in four-bit arithmetic, $\text{ulp}(-1100) = 1$, and $\text{ulp}(1) = 0.001$.

Another useful notation is $\text{err}(a \circledast b)$, which denotes the roundoff error incurred by using a p-bit floating-point operation $\circledast$ to approximate a real operation $*$ (addition, subtraction, multiplication, or division) on the operands a and b. Note that whereas ulp is an unsigned quantity, err is signed. For any basic operation, $a \circledast b = a * b + \text{err}(a \circledast b)$, and exact rounding guarantees that $|\text{err}(a \circledast b)| \leq \frac{1}{2}\text{ulp}(a \circledast b)$.

In the pages that follow, various properties of floating-point arithmetic are proven, and algorithms for manipulating expansions are developed based on these properties. Throughout, binary and decimal numbers are intermixed; the base should be apparent from context. A number is said to be *expressible in p bits* if it can be expressed with a p-bit significand, *not* counting the sign bit or the exponent. I will occasionally refer to the *magnitude of a bit*, defined relative to a specific number; for instance, the magnitude of the second nonzero bit of binary -1110 is four. The remainder of this section is quite technical; the reader may wish to skip the proofs on a first reading. The key new results are Theorems 48, 54, and 59, which provide algorithms for summing and scaling expansions.

6.3.2 Properties of Binary Arithmetic

Exact rounding guarantees that $|\text{err}(a \circledast b)| \leq \frac{1}{2}\text{ulp}(a \circledast b)$, but one can sometimes find a smaller bound for the roundoff error, as evidenced by the two lemmata below. The first lemma is useful when one operand is much smaller than the other, and the second is useful when the sum is close to a power of two. For Lemmata 36 through 40, let a and b be p-bit floating-point numbers.

Lemma 36 *Let $a \oplus b = a + b + \text{err}(a \oplus b)$. The roundoff error $|\text{err}(a \oplus b)|$ is no larger than $|a|$ or $|b|$. (An analogous result holds for subtraction.)*

Proof: Assume without loss of generality that $|a| \geq |b|$. The sum $a \oplus b$ is the p-bit floating-point number closest to $a + b$. But a is a p-bit floating-point number, so $|\text{err}(a \oplus b)| \leq |b| \leq |a|$. (See Figure 6.2.) ■

Corollary 37 *The roundoff error $\text{err}(a \oplus b)$ can be expressed with a p-bit significand.*

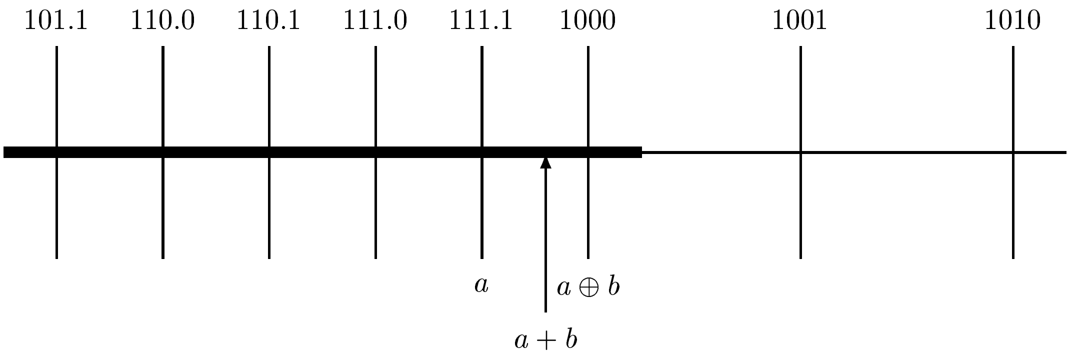


Figure 6.2: Demonstration of the first two lemmata. Vertical lines represent four-bit floating-point values. The roundoff error is the distance between $a + b$ and $a \oplus b$. Lemma 36 states that the error cannot be larger than $|b|$. Lemma 38(b) states that if $|a + b| \leq 2^i(2^{p+1} + 1)$ (for $i = -2$ and $p = 4$, this means that $a + b$ falls into the darkened region), then the error is no greater than 2^i. This lemma is useful when a computed value falls close to a power of two.

Proof: Assume without loss of generality that $|a| \geq |b|$. Clearly, the least significant nonzero bit of $\text{err}(a \oplus b)$ is no smaller in magnitude than $\text{ulp}(b)$. By Lemma 36, $|\text{err}(a \oplus b)| \leq |b|$; hence, the significand of $\text{err}(a \oplus b)$ is no longer than that of b. It follows that $\text{err}(a \oplus b)$ is expressible in p bits.

Lemma 38 *For any basic floating-point operation* $*$, *let* $a \circledast b = a * b + \text{err}(a \circledast b)$. *Then:*

(a) If $|\text{err}(a \circledast b)| \geq 2^i$ *for some integer* i, *then* $|a * b| \geq 2^i(2^p + 1)$.

(b) If $|\text{err}(a \circledast b)| > 2^i$ *for some integer* i, *then* $|a * b| > 2^i(2^{p+1} + 1)$.

Proof:

(a) The numbers $2^i(2^p), 2^i(2^p - 1), 2^i(2^p - 2), \ldots, 0$ are all expressible in p bits. Any value $|a * b| < 2^i(2^p + 1)$ is within a distance less than 2^i from one of these numbers.

(b) The numbers $2^i(2^{p+1}), 2^i(2^{p+1} - 2), 2^i(2^{p+1} - 4), \ldots, 0$ are all expressible in p bits. Any value $|a * b| \leq 2^i(2^{p+1} + 1)$ is within a distance of 2^i from one of these numbers. (See Figure 6.2.) ■

The next two lemmata identify special cases for which computer arithmetic is exact. The first shows that addition and subtraction are exact if the result has smaller magnitude than the operands.

Lemma 39 *Suppose that* $|a + b| \leq |a|$ *and* $|a + b| \leq |b|$. *Then* $a \oplus b = a + b$. *(An analogous result holds for subtraction.)*

Proof: Without loss of generality, assume $|a| \geq |b|$. Clearly, the least significant nonzero bit of $a + b$ is no smaller in magnitude than $\text{ulp}(b)$. However, $|a + b| \leq |b|$. It follows that $a + b$ can be expressed in p bits. ■

Many of the algorithms will rely on the following lemma, which shows that subtraction is exact for two operands within a factor of two of each other:

$$\begin{array}{rcrrrr}
a & = & 1 & 1 & 0 & 1 \\
b & = & 1 & 0 & 1 & 0 \\
\hline
a-b & = & & & 1 & 1
\end{array}
\qquad\qquad
\begin{array}{rcrrrrrl}
a & = & 1 & 0 & 0 & 1 & & \times 2^1 \\
b & = & & 1 & 0 & 0 & 1 & \\
\hline
a-b & = & & 1 & 0 & 0 & 1 &
\end{array}$$

Figure 6.3: Two demonstrations of Lemma 40.

Lemma 40 (Sterbenz [89]) *Suppose that* $b \in [\frac{a}{2}, 2a]$. *Then* $a \ominus b = a - b$.

Proof: Without loss of generality, assume $|a| \geq |b|$. (The other case is symmetric, because $a \ominus b = -b \ominus -a$.) Then $b \in [\frac{a}{2}, a]$. The difference satisfies $|a - b| \leq |b| \leq |a|$; the result follows by Lemma 39. ■

Two examples demonstrating Lemma 40 appear in Figure 6.3. If a and b have the same exponent, then floating-point subtraction is analogous to finding the difference between two p-bit integers of the same sign, and the result is expressible in p bits. Otherwise, the exponents of a and b differ by one, because $b \in [\frac{a}{2}, 2a]$. In this case, the difference has the smaller of the two exponents, and so can be expressed in p bits.

6.3.3 Simple Addition

An important basic operation in all the algorithms for performing arithmetic with expansions is the addition of two p-bit values to form a nonoverlapping expansion (of length two). Two such algorithms, due to Dekker and Knuth respectively, are presented.

Theorem 41 (Dekker [26]) *Let a and b be p-bit floating-point numbers such that $|a| \geq |b|$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$, where x is an approximation to $a + b$ and y represents the roundoff error in the calculation of x.*

FAST-TWO-SUM(a, b)
1 $\quad x \Leftarrow a \oplus b$
2 $\quad b_{\text{virtual}} \Leftarrow x \ominus a$
3 $\quad y \Leftarrow b \ominus b_{\text{virtual}}$
4 $\quad$ **return** (x, y)

Proof: Line 1 computes $a + b$, but may be subject to rounding, so we have $x = a + b + \text{err}(a \oplus b)$. By assumption $|a| \geq |b|$, so a and x must have the same sign (or $x = 0$).

Line 2 computes the quantity b_{virtual}, which is the value that was *really* added to a in Line 1. This subtraction is computed exactly; this fact can be proven by considering two cases. If a and b have the same sign, or if $|b| \leq \frac{|a|}{2}$, then $x \in [\frac{a}{2}, 2a]$ and one can apply Lemma 40 (see Figure 6.4). On the other hand, if a and b are opposite in sign and $|b| > \frac{|a|}{2}$, then $b \in [-\frac{a}{2}, -a]$ and one can apply Lemma 40 to Line 1, showing that x was computed exactly and therefore $b_{\text{virtual}} = b$ (see Figure 6.5). In either case the subtraction is exact, so $b_{\text{virtual}} = x - a = b + \text{err}(a \oplus b)$.

Line 3 is also computed exactly. By Corollary 37, $b - b_{\text{virtual}} = -\text{err}(a \oplus b)$ is expressible in p bits.

It follows that $y = -\text{err}(a \oplus b)$ and $x = a + b + \text{err}(a \oplus b)$, hence $a + b = x + y$. Exact rounding guarantees that $|y| \leq \frac{1}{2}\text{ulp}(x)$, so x and y are nonoverlapping. ■

$$
\begin{array}{rcccccccccccc}
a & = & & & & 1 & 1 & 1 & 1 & & & & \times 2^2 \\
b & = & & & & & & 1 & 0 & 0 & 1 & & \\ \cline{4-11}
x & = & a \oplus b & = & 1 & 0 & 0 & 1 & & & & & \times 2^3 \\
a & = & & & & 1 & 1 & 1 & 1 & & & & \times 2^2 \\ \cline{4-11}
b_{\text{virtual}} & = & x \ominus a & = & & & & 1 & 1 & 0 & 0 & & \\
& & & & & & & & & & & & \\
y & = & b \ominus b_{\text{virtual}} & = & & & & & - & 1 & 1 & &
\end{array}
$$

Figure 6.4: Demonstration of FAST-TWO-SUM where a and b have the same sign. The sum of 111100 and 1001 is the expansion $1001000 + -11$.

$$
\begin{array}{rcccccccccc}
a & = & & & 1 & 0 & 0 & 1 & & \times 2^1 \\
b & = & & & - & 1 & 0 & 1 & 1 & \\ \cline{5-9}
x & = & a \oplus b & = & & & 1 & 1 & 1 & \\
a & = & & & 1 & 0 & 0 & 1 & & \times 2^1 \\ \cline{5-9}
b_{\text{virtual}} & = & x \ominus a & = & - & 1 & 0 & 1 & 1 & \\
& & & & & & & & & \\
y & = & b \ominus b_{\text{virtual}} & = & & & & & 0 &
\end{array}
$$

Figure 6.5: Demonstration of FAST-TWO-SUM where a and b have opposite sign and $|b| > \frac{|a|}{2}$.

Note that the outputs x and y do *not* necessarily have the same sign, as Figure 6.4 demonstrates. Two-term subtraction ("FAST-TWO-DIFF") is implemented by the sequence $x \Leftarrow a \ominus b; b_{\text{virtual}} \Leftarrow a \ominus x$; $y \Leftarrow b_{\text{virtual}} \ominus b$. The proof of the correctness of this sequence is analogous to Theorem 41.

The difficulty with using FAST-TWO-SUM is the requirement that $|a| \geq |b|$. If the relative sizes of a and b are unknown, a comparison is required to order the addends before invoking FAST-TWO-SUM. With most C compilers[4], perhaps the fastest portable way to implement this test is with the statement "`if ((a > b) == (a > -b))`". This test takes time to execute, and the slowdown may be surprisingly large because on modern pipelined and superscalar architectures, an **if** statement coupled with imperfect microprocessor branch prediction may cause a processor's instruction pipeline to drain. This explanation is speculative and machine-dependent, but the TWO-SUM algorithm below, which avoids a comparison at the cost of three additional floating-point operations, is usually empirically faster[5]. Of course, FAST-TWO-SUM remains faster if the relative sizes of the operands are known *a priori*, and the comparison can be avoided.

Theorem 42 (Knuth [57]) *Let a and b be p-bit floating-point numbers, where $p \geq 3$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$, where x is an*

[4] The exceptions are those few that can identify and optimize the `fabs()` math library call.

[5] On a DEC Alpha-based workstation, using the bundled C compiler with optimization level 3, TWO-SUM uses roughly 65% as much time as FAST-TWO-SUM conditioned with the test "`if ((a > b) == (a > -b))`". On a SPARCstation IPX, using the GNU compiler with optimization level 2, TWO-SUM uses roughly 85% as much time. On the other hand, using the SPARCstation's bundled compiler with optimization (which produces slower code than gcc), conditional FAST-TWO-SUM uses only 82% as much time as TWO-SUM. The lesson is that for optimal speed, one must time each method with one's own machine and compiler.

$$
\begin{array}{rcccccccccccl}
a & = & & & & & & 1 & 1. & 1 & 1 & \\
b & = & & & & 1 & 1 & 0 & 1 & & & \\
\cline{5-11}
x & = & a \oplus b & = & 1 & 0 & 0 & 0 & & & & \times 2^1 \\
a & = & & & & & & 1 & 1. & 1 & 1 & \\
\cline{5-11}
b_{\text{virtual}} & = & x \ominus a & = & & 1 & 1 & 0 & 0 & & & \\
 & & & & & & & & & & & \\
a_{\text{virtual}} & = & x \ominus b_{\text{virtual}} & = & & & 1 & 0 & 0 & & & \\
 & & & & & & & & & & & \\
b_{\text{roundoff}} & = & b \ominus b_{\text{virtual}} & = & & & & & 1 & & & \\
a_{\text{roundoff}} & = & a \ominus a_{\text{virtual}} & = & & & & - & 0. & 0 & 1 & \\
\cline{5-11}
y & = & a_{\text{roundoff}} \oplus b_{\text{roundoff}} & = & & & & & 0. & 1 & 1 &
\end{array}
$$

Figure 6.6: Demonstration of TWO-SUM where $|a| < |b|$ and $|a| \leq |x|$. The sum of 11.11 and 1101 is the expansion $10000 + 0.11$.

approximation to $a + b$ and y is the roundoff error in the calculation of x.

```
TWO-SUM(a, b)
1    x ⇐ a ⊕ b
2    b_virtual ⇐ x ⊖ a
3    a_virtual ⇐ x ⊖ b_virtual
4    b_roundoff ⇐ b ⊖ b_virtual
5    a_roundoff ⇐ a ⊖ a_virtual
6    y ⇐ a_roundoff ⊕ b_roundoff
7    return (x, y)
```

Proof: If $|a| \geq |b|$, then Lines 1, 2, and 4 correspond precisely to the FAST-TWO-SUM algorithm. Recall from the proof of Theorem 41 that Line 2 is calculated exactly; it follows that Line 3 of TWO-SUM is calculated exactly as well, because $a_{\text{virtual}} = a$ can be expressed exactly. Hence, a_{roundoff} is zero, $y = b_{\text{roundoff}}$ is computed exactly, and the procedure is correct.

Now, suppose that $|a| < |b|$, and consider two cases. If $|x| < |a| < |b|$, then x is computed exactly by Lemma 39. It immediately follows that $b_{\text{virtual}} = b$, $a_{\text{virtual}} = a$, and b_{roundoff}, a_{roundoff}, and y are zero.

Conversely, if $|x| \geq |a|$, Lines 1 and 2 may be subject to rounding, so $x = a + b + \text{err}(a \oplus b)$, and $b_{\text{virtual}} = b + \text{err}(a \oplus b) + \text{err}(x \ominus a)$. (See Figure 6.6.) Lines 2, 3, and 5 are analogous to the three lines of FAST-TWO-DIFF (with Line 5 negated), so Lines 3 and 5 are computed exactly. Hence, $a_{\text{virtual}} = x - b_{\text{virtual}} = a - \text{err}(x \ominus a)$, and $a_{\text{roundoff}} = \text{err}(x \ominus a)$.

Because $|b| > |a|$, we have $|x| = |a \oplus b| \leq 2|b|$, so the roundoff errors $\text{err}(a \oplus b)$ and $\text{err}(x \ominus a)$ each cannot be more than $\text{ulp}(b)$, so $b_{\text{virtual}} \in [\frac{b}{2}, 2b]$ (for $p \geq 3$) and Lemma 40 can be applied to show that Line 4 is exact. Hence, $b_{\text{roundoff}} = -\text{err}(a \oplus b) - \text{err}(x \ominus a)$. Finally, Line 6 is exact because by Corollary 37, $a_{\text{roundoff}} + b_{\text{roundoff}} = -\text{err}(a \oplus b)$ is expressible in p bits.

It follows that $y = -\text{err}(a \oplus b)$ and $x = a + b + \text{err}(a \oplus b)$, hence $a + b = x + y$. ■

Two-term subtraction ("TWO-DIFF") is implemented by the sequence $x \Leftarrow a \ominus b$; $b_{\text{virtual}} \Leftarrow a \ominus x$; $a_{\text{virtual}} \Leftarrow x \oplus b_{\text{virtual}}$; $b_{\text{roundoff}} \Leftarrow b_{\text{virtual}} \ominus b$; $a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$; $y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$.

Corollary 43 *Let x and y be the values returned by* FAST-TWO-SUM *or* TWO-SUM.

(a) If $|y| \geq 2^i$ for some integer i, then $|x + y| \geq 2^i(2^p + 1)$.

(b) If $|y| > 2^i$ for some integer i, then $|x + y| > 2^i(2^{p+1} + 1)$.

Proof: y is the roundoff error $-\text{err}(a \oplus b)$ for some a and b. By Theorems 41 and 42, $a + b = x + y$. The results follow directly from Lemma 38. ■

Corollary 44 *Let x and y be the values returned by* FAST-TWO-SUM *or* TWO-SUM. *On a machine whose arithmetic uses round-to-even tiebreaking, x and y are nonadjacent.*

Proof: Exact rounding guarantees that $y \leq \frac{1}{2}\text{ulp}(x)$. If the inequality is strict, x and y are nonadjacent. If $y = \frac{1}{2}\text{ulp}(x)$, the round-to-even rule ensures that the least significant bit of the significand of x is zero, so x and y are nonadjacent. ■

6.3.4 Expansion Addition

Having established how to add two p-bit values, I turn to the topic of how to add two arbitrary precision values expressed as expansions. Three methods are available. EXPANSION-SUM adds an m-component expansion to an n-component expansion in $\mathcal{O}(mn)$ time. LINEAR-EXPANSION-SUM and FAST-EXPANSION-SUM do the same in $\mathcal{O}(m + n)$ time.

Despite its asymptotic disadvantage, EXPANSION-SUM can be faster than the linear-time algorithms in cases where the size of each expansion is small and fixed, because program loops can be completely unrolled and indirection overhead can be eliminated (by avoiding the use of arrays). The linear-time algorithms have conditionals that make such optimizations untenable. Hence, EXPANSION-SUM and FAST-EXPANSION-SUM are both used in the implementations of geometric predicates described in Section 6.5.

EXPANSION-SUM and LINEAR-EXPANSION-SUM both have the property that their outputs are nonoverlapping if their inputs are nonoverlapping, and nonadjacent if their inputs are nonadjacent. FAST-EXPANSION-SUM is faster than LINEAR-EXPANSION-SUM, performing six floating-point operations per component rather than nine, but has three disadvantages. First, FAST-EXPANSION-SUM does not always preserve either the nonoverlapping nor the nonadjacent property; instead, it preserves an intermediate property, described later. Second, whereas LINEAR-EXPANSION-SUM makes no assumption about the tiebreaking rule, FAST-EXPANSION-SUM is designed for machines that use round-to-even tiebreaking, and can fail on machines with other tiebreaking rules. Third, the correctness proof for FAST-EXPANSION-SUM is much more tedious. Nevertheless, I use FAST-EXPANSION-SUM in my geometric predicates, and relegate the slower LINEAR-EXPANSION-SUM to Appendix A. Users of machines that have exact rounding but not round-to-even tiebreaking should replace calls to FAST-EXPANSION-SUM with calls to LINEAR-EXPANSION-SUM.

A complicating characteristic of all the algorithms for manipulating expansions is that there may be spurious zero components scattered throughout the output expansions, even if no zeros were present in the input expansions. For instance, if the expansions $1111+0.0101$ and $1100+0.11$ are passed as inputs to any of the three expansion addition algorithms, the output expansion in four-bit arithmetic is $11100+0+0+0.0001$. One may want to add expansions thus produced to other expansions; fortunately, all the algorithms in this chapter cope well with spurious zero components in their input expansions. Unfortunately, accounting for

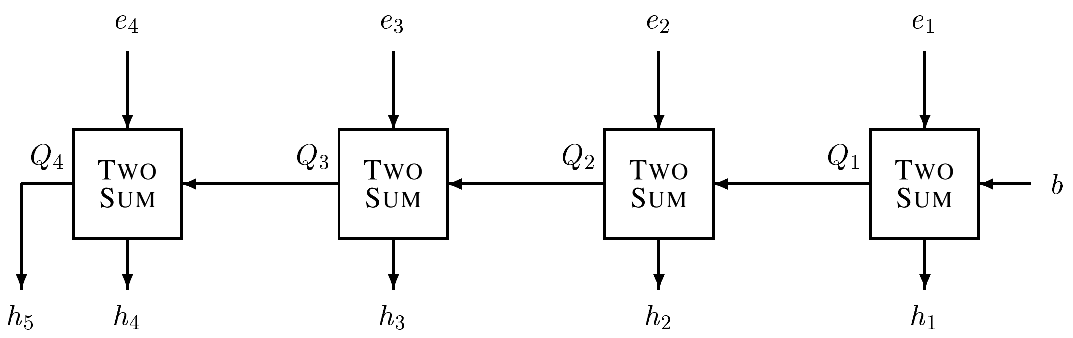


Figure 6.7: Operation of GROW-EXPANSION. The expansions e and h are illustrated with their most significant components on the left. All TWO-SUM boxes in this chapter observe the convention that the larger output (x) emerges from the left side of each box, and the smaller output (y) from the bottom or right. Each Q_i term is an approximate running total.

these zero components could complicate the correctness proofs significantly. To avoid confusion, most of the proofs for the addition and scaling algorithms are written as if all input components are nonzero. Spurious zeros can be integrated into the proofs (after the fact) by noting that the effect of a zero input component is always to produce a zero output component without changing the value of the accumulator (denoted by the variable Q). The effect can be likened to a pipeline delay; it will become clear in the first few proofs.

Each algorithm has an accompanying dataflow diagram, like Figure 6.7. Readers will find the proofs easier to understand if they follow the diagrams while reading the proofs, and keep several facts in mind. First, Lemma 36 indicates that the down arrow from any TWO-SUM box represents a number no larger than either input to the box. (This is why a zero input component yields a zero output component.) Second, Theorems 41 and 42 indicate that the down arrow from any TWO-SUM box represents a number too small to overlap the number represented by the left arrow from the box.

I begin with an algorithm for adding a single p-bit value to an expansion.

Theorem 45 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, and let b be a p-bit value where $p \geq 3$. Suppose that the components $e_1, e_2, \ldots, e_m$ are sorted in order of* **increasing** *magnitude, except that any of the e_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{m+1} h_i = e + b$, where the components $h_1, h_2, \ldots, h_{m+1}$ are also in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e is nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

```
GROW-EXPANSION(e, b)
    Q_0 ⇐ b
    for i ⇐ 1 to m
        (Q_i, h_i) ⇐ TWO-SUM(Q_{i-1}, e_i)
    h_{m+1} ⇐ Q_m
    return h
```

Q_i is an approximate sum of b and the first i components of e; see Figure 6.7. In an implementation, the array Q can be collapsed into a single scalar.

Proof: At the end of each iteration of the **for** loop, the invariant $Q_i + \sum_{j=1}^{i} h_j = b + \sum_{j=1}^{i} e_j$ holds. Certainly this invariant holds for $i = 0$ after Line 1 is executed. From Line 3 and Theorem 42, we have that $Q_i + h_i = Q_{i-1} + e_i$; from this one can deduce inductively that the invariant holds for all (relevant values of) i. Thus, after Line 4 is executed, $\sum_{j=1}^{m+1} h_j = \sum_{j=1}^{m} e_j + b$.

For all i, the output of TWO-SUM (in Line 3) has the property that h_i and Q_i do not overlap. By Lemma 36, $|h_i| \leq |e_i|$, and because e is a nonoverlapping expansion whose nonzero components are arranged in increasing order, h_i cannot overlap any of $e_{i+1}, e_{i+2}, \ldots$. It follows that h_i cannot overlap any of the later components of h, because these are constructed by summing Q_i with later e components. Hence, h is nonoverlapping and increasing (excepting zero components of h). If round-to-even tiebreaking is used, then h_i and Q_i are nonadjacent for all i (by Corollary 44), so if e is nonadjacent, then h is nonadjacent.

If any of the e_i is zero, the corresponding output component h_i is also zero, and the accumulator value Q is unchanged ($Q_i = Q_{i-1}$). (For instance, consider Figure 6.7, and suppose that e_3 is zero. The accumulator value Q_2 shifts through the pipeline to become Q_3, and a zero is harmlessly output as h_3. The same effect occurs in several algorithms in this chapter.) ■

Corollary 46 *The first m components of h are each no larger than the corresponding component of e. (That is, $|h_1| \leq |e_1|, |h_2| \leq |e_2|, \ldots, |h_m| \leq |e_m|$.) Furthermore, $|h_1| \leq |b|$.*

Proof: Follows immediately by application of Lemma 36 to Line 3. (Both of these facts are apparent in Figure 6.7. Recall that the down arrow from any TWO-SUM box represents a number no larger than either input to the box.) ■

If e is a long expansion, two optimizations might be advantageous. The first is to use a binary search to find the smallest component of e greater than or equal to ulp(b), and start there. A variant of this idea, without the search, is used in the next theorem. The second optimization is to stop early if the output of a TWO-SUM operation is the same as its inputs; the expansion is already nonoverlapping.

A naïve way to add one expansion to another is to repeatedly use GROW-EXPANSION to add each component of one expansion to the other. One can improve this idea with a small modification.

Theorem 47 *Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be nonoverlapping expansions of m and n p-bit components, respectively, where $p \geq 3$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{m+n} h_i = e + f$, where the components of h are in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e and f are nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

```
EXPANSION-SUM(e, f)
1    h ⇐ e
2    for i ⇐ 1 to n
3        ⟨h_i, h_{i+1}, ..., h_{i+m}⟩ ⇐ GROW-EXPANSION(⟨h_i, h_{i+1}, ..., h_{i+m-1}⟩, f_i)
4    return h
```

Proof: That $\sum_{i=1}^{m+n} h_i = \sum_{i=1}^{m} e_i + \sum_{i=1}^{n} f_i$ upon completion can be proven by induction on Line 3.

After setting $h \Leftarrow e$, EXPANSION-SUM traverses the expansion f from smallest to largest component, individually adding these components to h using GROW-EXPANSION (see Figure 6.8). The theorem would follow directly from Theorem 45 if each component f_i were added to the whole expansion h, but to save

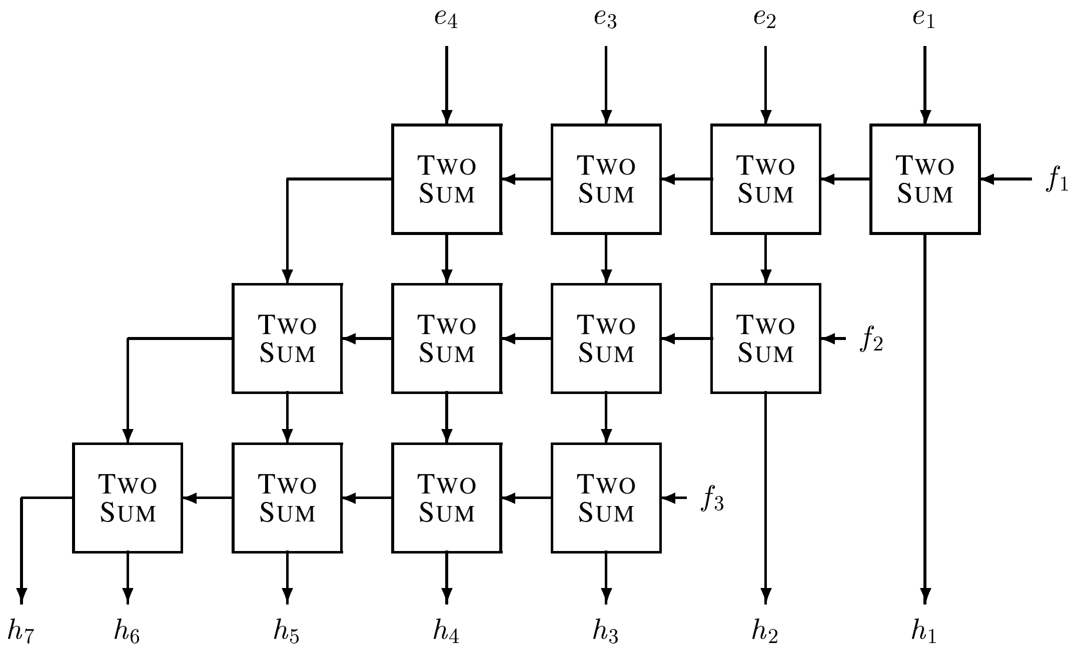


Figure 6.8: Operation of EXPANSION-SUM.

time, only the subexpansion $\langle h_i, h_{i+1}, \ldots, h_{i+m-1}\rangle$ is considered. (In Figure 6.8, this optimization saves three TWO-SUM operations that would otherwise appear in the lower right corner of the figure.)

When f_i is considered, the components $f_1, f_2, \ldots, f_{i-1}$ have already been summed into h. According to Corollary 46, $|h_j| \leq |f_j|$ after iteration j of Line 3. Because f is an increasing nonoverlapping expansion, for any $j < i$, h_j cannot overlap f_i, and furthermore $|h_j| < |f_i|$ (unless $f_i = 0$). Therefore, when one sums f_i into h, one can skip the first $i-1$ components of h without sacrificing the nonoverlapping and increasing properties of h. Similarly, if e and f are each nonadjacent, one can skip the first $i-1$ components of h without sacrificing the nonadjacent property of h.

No difficulty ensues if f_i is a spurious zero component, because zero does not overlap any number. GROW-EXPANSION will deposit a zero at h_i and continue normally. ■

Unlike EXPANSION-SUM, FAST-EXPANSION-SUM does not preserve the nonoverlapping or nonadjacent properties, but it is guaranteed to produce a strongly nonoverlapping output if its inputs are strongly nonoverlapping. An expansion is *strongly nonoverlapping* if no two of its components are overlapping, no component is adjacent to two other components, and any pair of adjacent components have the property that both components can be expressed with a one-bit significand (that is, both are powers of two). For instance, $11000 + 11$ and $10000 + 1000 + 10 + 1$ are both strongly nonoverlapping, but $11100 + 11$ is not, nor is $100 + 10 + 1$. A characteristic of this property is that a zero bit must occur in the expansion at least once every $p + 1$ bits. For instance, in four-bit arithmetic, a strongly nonoverlapping expansion whose largest component is 1111 can be no greater than $1111.01111011110\ldots$. Any nonadjacent expansion is strongly nonoverlapping, and any strongly nonoverlapping expansion is nonoverlapping, but the converse implications do not apply. Recall that any floating-point value has a nonadjacent expansion; hence, any floating-point value has a strongly nonoverlapping expansion. For example, 1111.1 may be expressed as $10000 + -0.1$.

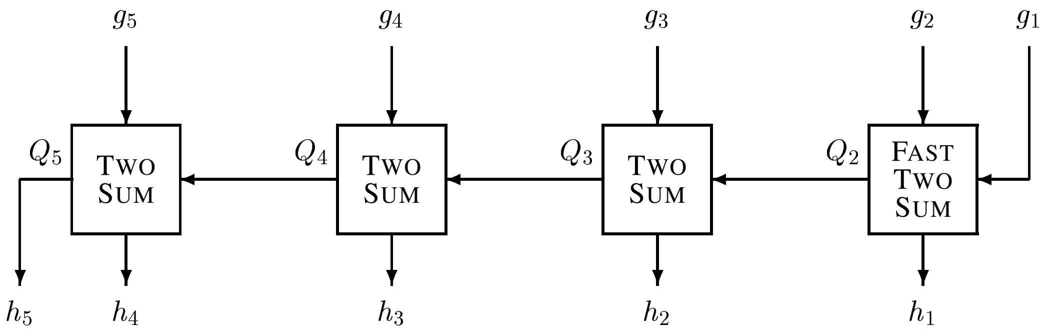


Figure 6.9: Operation of FAST-EXPANSION-SUM. The Q_i terms maintain an approximate running total.

Under the assumption that all expansions are strongly nonoverlapping, it is possible to prove the first key result of this chapter: the FAST-EXPANSION-SUM algorithm defined below behaves correctly under round-to-even tiebreaking. The algorithm can also be used with round-toward-zero arithmetic, but the proof is different. I have emphasized round-to-even arithmetic here due to the IEEE 754 standard.

A variant of this algorithm was presented by Priest [76], but it is used differently here. Priest uses the algorithm to sum two nonoverlapping expansions, and proves under general conditions that the components of the resulting expansion overlap by at most one digit (i.e. one bit in binary arithmetic). An expensive renormalization step is required afterward to remove the overlap. Here, by contrast, the algorithm is used to sum two strongly nonoverlapping expansions, and the result is also a strongly nonoverlapping expansion. Not surprisingly, the proof demands more stringent conditions than Priest requires: binary arithmetic with exact rounding and round-to-even tiebreaking, consonant with the IEEE 754 standard. No renormalization is needed.

Theorem 48 *Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be strongly nonoverlapping expansions of m and n p-bit components, respectively, where $p \geq 4$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. On a machine whose arithmetic uses the round-to-even rule, the following algorithm will produce a strongly nonoverlapping expansion h such that $h = \sum_{i=1}^{m+n} h_i = e + f$, where the components of h are also in order of increasing magnitude, except that any of the h_i may be zero.*

FAST-EXPANSION-SUM(e, f)
Merge e and f into a single sequence g, in order of
  nondecreasing magnitude (possibly with interspersed zeros)
$(Q_2, h_1) \Leftarrow$ FAST-TWO-SUM(g_2, g_1)
for $i \Leftarrow 3$ **to** $m + n$
  $(Q_i, h_{i-1}) \Leftarrow$ TWO-SUM(Q_{i-1}, g_i)
$h_{m+n} \Leftarrow Q_{m+n}$
return h

Q_i is an approximate sum of the first i components of g; see Figure 6.9.

Several lemmata will aid the proof of Theorem 48. I begin with a proof that the sum itself is correct.

Lemma 49 (Q Invariant) *At the end of each iteration of the* **for** *loop, the invariant* $Q_i + \sum_{j=1}^{i-1} h_j = \sum_{j=1}^{i} g_j$ *holds. This assures us that after Line 5 is executed,* $\sum_{j=1}^{m+n} h_j = \sum_{j=1}^{m+n} g_j$*, so the algorithm produces a correct sum.*

Proof: The invariant clearly holds for $i = 2$ after Line 2 is executed. For larger values of i, Line 4 ensures that $Q_i + h_{i-1} = Q_{i-1} + g_i$; the invariant follows by induction. ■

Lemma 50 *Let* $\widehat{g} = \sum_{j=1}^{k} \widehat{g}_j$ *be a series formed by merging two strongly nonoverlapping expansions, or a subseries thereof. Suppose that* $\widehat{g}_k$ *is the largest component and has a nonzero bit of magnitude* 2^i *or smaller for some integer* i*. Then* $|\sum_{j=1}^{k} \widehat{g}_j| < 2^i(2^{p+1} - 1)$*, and* $|\sum_{j=1}^{k-1} \widehat{g}_j| < 2^i(2^p)$*.*

Proof: Let $\widehat{e}$ and $\widehat{f}$ be the expansions (or subsequences thereof) from which $\widehat{g}$ was formed, and assume that the component $\widehat{g}_k$ comes from the expansion $\widehat{e}$. Because $\widehat{g}_k$ is the largest component of $\widehat{e}$ and has a nonzero bit of magnitude 2^i or smaller, and because $\widehat{e}$ is strongly nonoverlapping, $|\widehat{e}|$ is less than $2^i(2^p - \frac{1}{2})$. (For instance, if $p = 4$ and $i = 0$, then $|\widehat{e}| \leq 1111.0111101111\ldots$.) The same bound applies to the expansion $\widehat{f}$, so $|\widehat{g}| = |\widehat{e} + \widehat{f}| < 2^i(2^{p+1} - 1)$.

If we omit $\widehat{g}_k$ from the sum, there are two cases to consider. If $\widehat{g}_k = 2^i$, then $|\widehat{e} - \widehat{g}_k|$ is less than 2^i, and $|\widehat{f}|$ is less than $2^i(2)$. (For instance, if $p = 4$, $i = 0$, and $\widehat{g}_k = 1$, then $|\widehat{e} - \widehat{g}_k| \leq 0.10111101111\ldots$, and $|\widehat{f}| \leq 1.10111101111\ldots$.) Conversely, if $\widehat{g}_k \neq 2^i$, then $|\widehat{e} - \widehat{g}_k|$ is less than $2^i(\frac{1}{2})$, and $|\widehat{f}|$ is less than $2^i(2^p - \frac{1}{2})$. (For instance, if $p = 4$, $i = 0$, and $\widehat{g}_k = 1111$, then $|\widehat{e} - \widehat{g}_k| \leq 0.0111101111\ldots$, and $|\widehat{f}| \leq 1111.0111101111\ldots$.) In either case, $|\widehat{g} - \widehat{g}_k| = |\widehat{e} - \widehat{g}_k + \widehat{f}| < 2^i(2^p)$. ■

Lemma 51 *The expansion* h *produced by* FAST-EXPANSION-SUM *is a nonoverlapping expansion whose components are in order of increasing magnitude (excepting zeros).*

Proof: Suppose for the sake of contradiction that two successive nonzero components of h overlap or occur in order of decreasing magnitude. Denote the first such pair produced[6] h_{i-1} and h_i; then the components $h_1, \ldots, h_{i-1}$ are nonoverlapping and increasing (excepting zeros).

Assume without loss of generality that the exponent of h_{i-1} is zero, so that h_{i-1} is of the form $\pm 1.*$, where an asterisk represents a sequence of arbitrary bits.

Q_i and h_{i-1} are produced by a TWO-SUM or FAST-TWO-SUM operation, and are therefore nonadjacent by Corollary 44 (because the round-to-even rule is used). Q_i is therefore of the form $\pm * 00$ (having no bits of magnitude smaller than four). Because $|h_{i-1}| \geq 1$, Corollary 43(a) guarantees that

$$|Q_i + h_{i-1}| \geq 2^p + 1. \tag{6.1}$$

Because the offending components h_{i-1} and h_i are nonzero and either overlapping or of decreasing magnitude, there must be at least one nonzero bit in the significand of h_i whose magnitude is no greater than one. One may ask, where does this offending bit come from? h_i is computed by Line 4 from Q_i and g_{i+1}, and the offending bit cannot come from Q_i (which is of the form $\pm * 00$), so it must have come from g_{i+1}. Hence, $|g_{i+1}|$ has a nonzero bit of magnitude one or smaller. Applying Lemma 50, one finds that $|\sum_{j=1}^{i} g_j| < 2^p$.

[6] It is implicitly assumed here that the first offending pair is not separated by intervening zeros. The proof could be written to consider the case where intervening zeros appear, but this would make it even more convoluted. Trust me.

$$
\begin{array}{rr|ccccc|ccccccccccccc}
 & |\sum_{j'} e_{j'}| \leq & & \multicolumn{3}{c}{\boxed{g_{i+1}}} & . & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
|\sum_{j=1}^{i} g_i| \Big\{ & |\sum_{j''} f_{j''}| \leq & & 1 & 1 & 1 & 1. & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 & |\sum_{j=1}^{i-2} h_j| \leq & & & & & 0. & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
 & |Q_i + h_{i-1}| \leq & 1 & 0 & 0 & 0 & 0. & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1
\end{array}
$$

Figure 6.10: Demonstration (for $p = 4$) of how the Q Invariant is used in the proof that h is nonoverlapping. The top two values, e and f, are being summed to form h. Because g_{i+1} has a nonzero bit of magnitude no greater than 1, and because g is formed by merging two strongly nonoverlapping expansions, the sum $|\sum_{j=1}^{i} g_i| + |\sum_{j=1}^{i-2} h_j|$ can be no larger than illustrated in this worst-case example. As a result, $|Q_i + h_{i-1}|$ cannot be large enough to have a roundoff error of 1, so $|h_{i-1}|$ is smaller than 1 and cannot overlap g_{i+1}. (Note that g_{i+1} is not part of the sum; it appears above in a box drawn as a placeholder that bounds the value of each expansion.)

A bound for $\sum_{j=1}^{i-2} h_j$ can be derived by recalling that h_{i-1} is of the form $\pm 1.*$, and $h_1, \ldots, h_{i-1}$ are nonoverlapping and increasing. Hence, $|\sum_{j=1}^{i-2} h_j| < 1$.

Rewrite the Q Invariant in the form $Q_i + h_{i-1} = \sum_{j=1}^{i} g_j - \sum_{j=1}^{i-2} h_j$. Using the bounds derived above, we obtain

$$|Q_i + h_{i-1}| < 2^p + 1. \tag{6.2}$$

See Figure 6.10 for a concrete example.

Inequalities 6.1 and 6.2 cannot hold simultaneously. The result follows by contradiction. ∎

Proof of Theorem 48: Lemma 49 ensures that $h = e + f$. Lemma 51 eliminates the possibility that the components of h overlap or fail to occur in order of increasing magnitude; it remains only to prove that h is strongly nonoverlapping. Suppose that two successive nonzero components h_{i-1} and h_i are adjacent.

Assume without loss of generality that the exponent of h_{i-1} is zero, so that h_{i-1} is of the form $\pm 1.*$. As in the proof of Lemma 51, Q_i must have the form $\pm * 00$.

Because h_{i-1} and h_i are adjacent, the least significant nonzero bit of h_i has magnitude two; that is, h_i is of the form $\pm * 10$. Again we ask, where does this bit come from? As before, this bit cannot come from Q_i, so it must have come from g_{i+1}. Hence, $|g_{i+1}|$ has a nonzero bit of magnitude two. Applying Lemma 50, we find that $|\sum_{j=1}^{i+1} g_j| < 2^{p+2} - 2$ and $|\sum_{j=1}^{i} g_j| < 2^{p+1}$.

Bounds for $\sum_{j=1}^{i-1} h_j$ and $\sum_{j=1}^{i-2} h_j$ can also be derived by recalling that h_{i-1} is of the form $\pm 1.*$ and is the largest component of a nonoverlapping expansion. Hence, $|\sum_{j=1}^{i-1} h_j| < 2$, and $|\sum_{j=1}^{i-2} h_j| < 1$.

Rewriting the Q Invariant in the form $Q_{i+1} + h_i = \sum_{j=1}^{i+1} g_j - \sum_{j=1}^{i-1} h_j$, we obtain

$$|Q_{i+1} + h_i| < 2^{p+2}. \tag{6.3}$$

The Q Invariant also gives us the identity $Q_i + h_{i-1} = \sum_{j=1}^{i} g_j - \sum_{j=1}^{i-2} h_j$. Hence,

$$|Q_i + h_{i-1}| < 2^{p+1} + 1. \tag{6.4}$$

Recall that the value $|h_i|$ is at least 2. Consider the possibility that $|h_i|$ might be greater than 2; by Corollary 43(b), this can occur only if $|Q_{i+1} + h_i| > 2^{p+2} + 2$, contradicting Inequality 6.3. Hence, $|h_i|$ must be exactly 2, and is expressible in one bit. (Figure 6.11 gives an example where this occurs.)

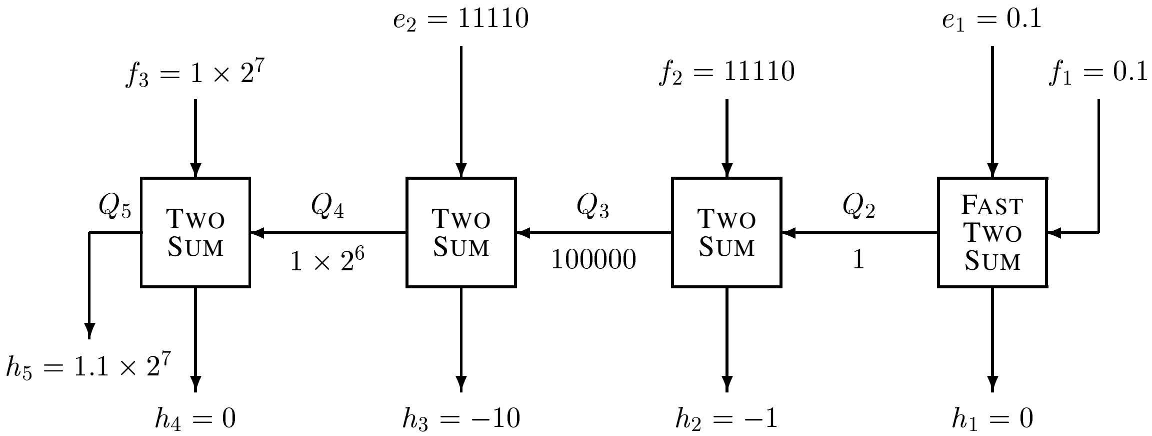


Figure 6.11: A four-bit example where FAST-EXPANSION-SUM generates two adjacent components h_2 and h_3. The figure permits me a stab at explaining the (admittedly thin) intuition behind Theorem 48: suppose h_2 is of the form $\pm 1.*$. Because h_2 is the roundoff term associated with Q_3, Q_3 must be of the form $*00$ if round-to-even arithmetic is used. Hence, the bit of magnitude 2 in h_3 must have come from e_2. This implies that $|e_2|$ is no larger than 11110, which imposes bounds on how large $|Q_3|$ and $|Q_4|$ can be (Lemma 50); these bounds in turn imply that $|h_2|$ can be no larger than 1, and $|h_3|$ can be no larger than 10. Furthermore, h_4 cannot be adjacent to h_3 because neither Q_4 nor f_3 can have a bit of magnitude 4.

Similarly, the value $|h_{i-1}|$ is at least 1. Consider the possibility that $|h_{i-1}|$ might be greater than 1; by Corollary 43(b), this can occur only if $|Q_i + h_{i-1}| > 2^{p+1} + 1$, contradicting Inequality 6.4. Hence, $|h_{i-1}|$ must be exactly 1, and is expressible in one bit.

By Corollary 43(a), $|Q_i + h_{i-1}| \geq 2^p + 1$ (because $|h_{i-1}| = 1$). Using this inequality, the inequality $|\sum_{j=1}^{i-2} h_j| < 1$, and the Q Invariant, one can deduce that $|\sum_{j=1}^{i} g_j| > 2^p$. Because g is formed from two nonoverlapping increasing expansions, this inequality implies that $|g_i| \geq 2^{p-2} \geq 100$ binary (recalling that $p \geq 4$), and hence $g_{i+2}, g_{i+3}, \ldots$ must all be of the form $\pm * 000$ (having no bits of magnitude smaller than 8). Q_{i+1} is also of the form $\pm * 000$, because Q_{i+1} and h_i are produced by a TWO-SUM or FAST-TWO-SUM operation, and are therefore nonadjacent by Corollary 44 (assuming the round-to-even rule is used).

Because Q_{i+1} and $g_{i+2}, g_{i+3}, \ldots$ are of the form $\pm * 000$, $h_{i+1}, h_{i+2}, \ldots$ must be as well, and are therefore not adjacent to h_i. It follows that h cannot contain three consecutive adjacent components.

These arguments prove that if two components of h are adjacent, both are expressible in one bit, and no other components are adjacent to them. Hence, h is strongly nonoverlapping. ■

The proof of Theorem 48 is more complex than one would like. It is unfortunate that the proof requires strongly nonoverlapping expansions; it would be more parsimonious if FAST-EXPANSION-SUM produced nonoverlapping output from nonoverlapping input, or nonadjacent output from nonadjacent input. Unfortunately, it does neither. For a counterexample to the former possibility, consider adding the nonoverlapping expansion $11110000 + 1111 + 0.1111$ to itself in four-bit arithmetic. (This example produces an overlapping expansion if one uses the round-to-even rule, but not if one uses the round-toward-zero rule.) For a counterexample to the latter possibility, see Figure 6.11. On a personal note, it took me quite a bit of effort to find a property between nonoverlapping and nonadjacent that is preserved by FAST-EXPANSION-SUM. Several conjectures were laboriously examined and discarded before I converged on the strongly nonoverlapping property. I persisted only because the algorithm consistently works in practice.

It is also unfortunate that the proof requires explicit consideration of the tiebreaking rule. FAST-EXPANSION-SUM works just as well on a machine that uses the round-toward-zero rule. The conditions under which it works are also simpler—the output expansion is guaranteed to be nonoverlapping if the input expansions are. One might hope to prove that FAST-EXPANSION-SUM works regardless of rounding mode, but this is not possible. Appendix B demonstrates the difficulty with an example of how mixing round-toward-zero and round-to-even arithmetic can lead to the creation of overlapping expansions.

The algorithms EXPANSION-SUM and FAST-EXPANSION-SUM can be mixed only to a limited degree. EXPANSION-SUM preserves the nonoverlapping and nonadjacent properties, but not the strongly nonoverlapping property; FAST-EXPANSION-SUM preserves only the strongly nonoverlapping property. Because nonadjacent expansions are strongly nonoverlapping, and strongly nonoverlapping expansions are nonoverlapping, expansions produced exclusively by one of the two algorithms can be fed as input to the other, but it may be dangerous to repeatedly switch back and forth between the two algorithms. In practice, EXPANSION-SUM is only preferred for producing small expansions, which are nonadjacent and hence suitable as input to FAST-EXPANSION-SUM.

It is useful to consider the operation counts of the algorithms. EXPANSION-SUM uses mn TWO-SUM operations, for a total of $6mn$ flops (floating-point operations). FAST-EXPANSION-SUM uses $m + n - 2$ TWO-SUM operations and one FAST-TWO-SUM operation, for a total of $6m + 6n - 9$ flops. However, the merge step of FAST-EXPANSION-SUM requires $m+n-1$ comparison operations of the form "**if** $|e_i| > |f_j|$". Empirically, each such comparison seems to take roughly as long as three flops; hence, a rough measure is to estimate that FAST-EXPANSION-SUM takes as long to execute as $9m + 9n - 12$ flops.

These estimates correlate well with the measured performance of the algorithms. I implemented each procedure as a function call whose parameters are variable-length expansions stored as arrays, and measured them on a DEC Alpha-based workstation using the bundled compiler with optimization level 3. By plotting their performance over a variety of expansion sizes and fitting curves, I found that EXPANSION-SUM runs in $0.83(m + n) - 0.7$ microseconds, and FAST-EXPANSION-SUM runs in $0.54mn + 0.6$ microseconds. FAST-EXPANSION-SUM is always faster except when one of the expansions has only one component, in which case GROW-EXPANSION should be used.

As I have mentioned, however, the balance shifts when expansion lengths are small and fixed. By storing small, fixed-length expansions as scalar variables rather than arrays, one can unroll the loops in EXPANSION-SUM, remove array indexing overhead, and allow components to be allocated to registers by the compiler. Thus, EXPANSION-SUM is attractive in this special case, and is used to advantage in my implementation of the geometric predicates of Section 6.5. Note that FAST-EXPANSION-SUM is difficult to unroll because of the conditionals in its initial merging step.

On the other hand, the use of arrays to store expansions (and non-unrolled loops to manage them) confers the advantage that spurious zero components can easily be eliminated from output expansions. In the procedures GROW-EXPANSION, EXPANSION-SUM, and FAST-EXPANSION-SUM, as well as the procedures SCALE-EXPANSION and COMPRESS in the sections to come, *zero elimination* can be achieved by maintaining a separate index for the output array h and advancing this index only when the procedure produces a nonzero component of h. In practice, versions of these algorithms that eliminate zeros are almost always preferable to versions that don't (except when loop unrolling confers a greater advantage). Zero elimination adds a small amount of overhead for testing and indexing, but the lost time is virtually always regained when further operations are performed on the resulting shortened expansions.

Experience suggests that it is economical to use unrolled versions of EXPANSION-SUM to form expansions of up to about four components, tolerating interspersed zeros, and to use FAST-EXPANSION-SUM with zero elimination when forming (potentially) larger expansions.

6.3.5 Simple Multiplication

The basic multiplication algorithm computes a nonoverlapping expansion equal to the product of two p-bit values. The multiplication is performed by splitting each value into two halves with half the precision, then performing four exact multiplications on these fragments. The trick is to find a way to split a floating-point value in two. The following theorem was first proven by Dekker [26]:

Theorem 52 *Let a be a p-bit floating-point number, where $p \geq 3$. Choose a* splitting point *s such that $\frac{p}{2} \leq s \leq p-1$. Then the following algorithm will produce a $(p-s)$-bit value a_{hi} and a nonoverlapping $(s-1)$-bit value a_{lo} such that $|a_{\text{hi}}| \geq |a_{\text{lo}}|$ and $a = a_{\text{hi}} + a_{\text{lo}}$.*

```
SPLIT(a, s)
1    c ⇐ (2^s + 1) ⊗ a
2    a_big ⇐ c ⊖ a
3    a_hi ⇐ c ⊖ a_big
4    a_lo ⇐ a ⊖ a_hi
5    return (a_hi, a_lo)
```

The claim may seem absurd. After all, a_{hi} and a_{lo} have only $p-1$ bits of significand between them; how can they carry all the information of a p-bit significand? The secret is hidden in the sign bit of a_{lo}. For instance, the seven-bit number 1001001 can be split into the three-bit terms 1010000 and -111. This property is fortunate, because even if p is odd, as it is in IEEE 754 double precision arithmetic, a can be split into two $\lfloor \frac{p}{2} \rfloor$-bit values.

Proof: Line 1 is equivalent to computing $2^s a \oplus a$. (Clearly, $2^s a$ can be expressed exactly, because multiplying a value by a power of two only changes its exponent, and does not change its significand.) Line 1 is subject to rounding, so we have $c = 2^s a + a + \text{err}(2^s a \oplus a)$.

Line 2 is also subject to rounding, so $a_{\text{big}} = 2^s a + \text{err}(2^s a \oplus a) + \text{err}(c \ominus a)$. It will become apparent shortly that the proof relies on showing that the exponent of a_{big} is no greater than the exponent of $2^s a$. Both $|\text{err}(2^s a \oplus a)|$ and $|\text{err}(c \ominus a)|$ are bounded by $\frac{1}{2}\text{ulp}(c)$, so the exponent of a_{big} can only be larger than that of $2^s a$ if every bit of the significand of a is nonzero except possibly the last (in four-bit arithmetic, a must have significand 1110 or 1111). By manually checking the behavior of SPLIT in these two cases, one can verify that the exponent of a_{big} is never larger than that of $2^s a$.

The reason this fact is useful is because, with Line 2, it implies that $|\text{err}(c \ominus a)| \leq \frac{1}{2}\text{ulp}(2^s a)$, and so the error term $\text{err}(c \ominus a)$ is expressible in $s-1$ bits (for $s \geq 2$).

By Lemma 40, Lines 3 and 4 are calculated exactly. It follows that $a_{\text{hi}} = a - \text{err}(c \ominus a)$, and $a_{\text{lo}} = \text{err}(c \ominus a)$; the latter is expressible in $s-1$ bits. To show that a_{hi} is expressible in $p-s$ bits, consider that its least significant bit cannot be smaller than $\text{ulp}(a_{\text{big}}) = 2^s \text{ulp}(a)$. If a_{hi} has the same exponent as a, then a_{hi} must be expressible in $p-s$ bits; alternatively, if a_{hi} has an exponent one greater than that of a (because $a - \text{err}(c \ominus a)$ has a larger exponent than a), then a_{hi} is expressible in one bit (as demonstrated in Figure 6.12).

Finally, the exactness of Line 4 implies that $a = a_{\text{hi}} + a_{\text{lo}}$ as required. ∎

Multiplication is performed by setting $s = \lceil \frac{p}{2} \rceil$, so that the p-bit operands a and b are each split into two $\lfloor \frac{p}{2} \rfloor$-bit pieces, a_{hi}, a_{lo}, b_{hi}, and b_{lo}. The products $a_{\text{hi}}b_{\text{hi}}$, $a_{\text{lo}}b_{\text{hi}}$, $a_{\text{hi}}b_{\text{lo}}$, and $a_{\text{lo}}b_{\text{lo}}$ can each be computed exactly by the floating-point unit, producing four values. These could then be summed using the FAST-EXPANSION-SUM procedure in Section 6.3.4. However, Dekker [26] provides several faster ways to accomplish the computation. Dekker attributes the following method to G. W. Veltkamp.

$$
\begin{array}{rcccccccccccccl}
a & = & & & & & & & 1 & 1 & 1 & 0 & 1 & \\
2^3a & = & & & & 1 & 1 & 1 & 0 & 1 & & & & \times 2^3 \\
\cline{4-12}
c & = & (2^3+1)\otimes a & = & 1 & 0 & 0 & 0 & 0 & & & & & \times 2^4 \\
a & = & & & & & & & 1 & 1 & 1 & 0 & 1 & \\
\cline{4-12}
a_{\text{big}} & = & c \ominus a & = & & 1 & 1 & 1 & 0 & 0 & & & & \times 2^3 \\
a_{\text{hi}} & = & c \ominus a_{\text{big}} & = & & & & 1 & 0 & 0 & 0 & 0 & & \times 2^1 \\
a_{\text{lo}} & = & a \ominus a_{\text{hi}} & = & & & & & & & - & 1 & 1 &
\end{array}
$$

Figure 6.12: Demonstration of SPLIT splitting a five-bit number into two two-bit numbers.

$$
\begin{array}{rcrcccccccccccccl}
a & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
b & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
\cline{5-16}
x & = & a \otimes b & = & 1 & 1 & 0 & 1 & 1 & 0 & & & & & & & \times 2^6 \\
 & & a_{\text{hi}} \otimes b_{\text{hi}} & = & 1 & 1 & 0 & 0 & 0 & 1 & & & & & & & \times 2^6 \\
\cline{5-10}
err_1 & = & x \ominus (a_{\text{hi}} \otimes b_{\text{hi}}) & = & & & & 1 & 0 & 1 & 0 & 0 & 0 & & & & \times 2^3 \\
 & & a_{\text{lo}} \otimes b_{\text{hi}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
\cline{8-14}
err_2 & = & err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}}) & = & & & & & 1 & 0 & 0 & 1 & 1 & 0 & & & \times 2^2 \\
 & & a_{\text{hi}} \otimes b_{\text{lo}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
\cline{9-14}
err_3 & = & err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 0 & 0 & 0 & 0 & \\
 & & a_{\text{lo}} \otimes b_{\text{lo}} & = & & & & & & & & & 1 & 0 & 0 & 1 & \\
\cline{11-16}
-y & = & err_3 \ominus (a_{\text{lo}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 1 & 0 & 0 & 1 &
\end{array}
$$

Figure 6.13: Demonstration of TWO-PRODUCT in six-bit arithmetic where $a = b = 111011$, $a_{\text{hi}} = b_{\text{hi}} = 111000$, and $a_{\text{lo}} = b_{\text{lo}} = 11$. Note that each intermediate result is expressible in six bits. The resulting expansion is $110110 \times 2^6 + 11001$.

Theorem 53 *Let a and b be p-bit floating-point numbers, where $p \geq 6$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$, where x is an approximation to ab and y represents the roundoff error in the calculation of x. Furthermore, if round-to-even tiebreaking is used, x and y are nonadjacent. (See Figure 6.13.)*

TWO-PRODUCT(a, b)
- $x \Leftarrow a \otimes b$
- $(a_{\text{hi}}, a_{\text{lo}}) = \text{SPLIT}(a, \lceil \frac{p}{2} \rceil)$
- $(b_{\text{hi}}, b_{\text{lo}}) = \text{SPLIT}(b, \lceil \frac{p}{2} \rceil)$
- $err_1 \Leftarrow x \ominus (a_{\text{hi}} \otimes b_{\text{hi}})$
- $err_2 \Leftarrow err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}})$
- $err_3 \Leftarrow err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}})$
- $y \Leftarrow (a_{\text{lo}} \otimes b_{\text{lo}}) \ominus err_3$
- **return** (x, y)

Proof: Line 1 is subject to rounding, so we have $x = ab + \text{err}(a \otimes b)$. The multiplications in Lines 4 through 7 are all exact, because each factor has no more than $\lfloor \frac{p}{2} \rfloor$ bits; it will be proven that each of the subtractions is also exact, and thus $y = -\text{err}(a \otimes b)$.

Without loss of generality, assume that the exponents of a and b are $p-1$, so that $|a|$ and $|b|$ are integers in the range $[2^{p-1}, 2^p - 1]$. In the proof of Theorem 52 it emerged that $|a_{\text{hi}}|$ and $|b_{\text{hi}}|$ are integers in the range $[2^{p-1}, 2^p]$, and $|a_{\text{lo}}|$ and $|b_{\text{lo}}|$ are integers in the range $[0, 2^{\lceil p/2 \rceil - 1}]$. From these ranges and the assumption that $p \geq 6$, one can derive the inequalities $|a_{\text{lo}}| \leq \frac{1}{8}|a_{\text{hi}}|$, $|b_{\text{lo}}| \leq \frac{1}{8}|b_{\text{hi}}|$, and $\text{err}(a \otimes b) \leq 2^{p-1} \leq \frac{1}{32}|a_{\text{hi}} b_{\text{hi}}|$.

Intuitively, $a_{\text{hi}} b_{\text{hi}}$ ought to be within a factor of two of $a \otimes b$, so that Line 4 is computed exactly (by Lemma 40). To confirm this hunch, note that $x = ab + \text{err}(a \otimes b) = a_{\text{hi}} b_{\text{hi}} + a_{\text{lo}} b_{\text{hi}} + a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b) = a_{\text{hi}} b_{\text{hi}} \pm \frac{19}{64}|a_{\text{hi}} b_{\text{hi}}|$ (using the inequalities stated above), which justifies the use of Lemma 40. Because Line 4 is computed without roundoff, $err_1 = a_{\text{lo}} b_{\text{hi}} + a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$.

We are assured that Line 5 is executed without roundoff error if the value $err_1 - a_{\text{lo}} b_{\text{hi}} = a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$ is expressible in p bits. I prove that this property holds by showing that the left-hand expression is a multiple of $2^{\lceil p/2 \rceil}$, and the right-hand expression is strictly smaller than $2^{\lceil 3p/2 \rceil}$.

The upper bound on the absolute value of the right-hand expression follows immediately from the upper bounds for a_{hi}, a_{lo}, b_{lo}, and $\text{err}(a \otimes b)$. To show that the left-hand expression is a multiple of $2^{\lceil p/2 \rceil}$, consider that err_1 must be a multiple of 2^{p-1} because $a \otimes b$ and $a_{\text{hi}} b_{\text{hi}}$ have exponents of at least $2p - 2$. Hence, $err_1 - a_{\text{lo}} b_{\text{hi}}$ must be a multiple of $2^{\lceil p/2 \rceil}$ because a_{lo} is an integer, and b_{hi} is a multiple of $2^{\lceil p/2 \rceil}$. Hence, Line 5 is computed exactly, and $err_2 = a_{\text{hi}} b_{\text{lo}} + a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$.

To show that Line 6 is computed without roundoff error, note that $a_{\text{lo}} b_{\text{lo}}$ is an integer no greater than 2^{p-1} (because a_{lo} and b_{lo} are integers no greater than $2^{\lceil p/2 \rceil - 1}$), and $\text{err}(a \otimes b)$ is an integer no greater than 2^{p-1}. Thus, $err_3 = a_{\text{lo}} b_{\text{lo}} + \text{err}(a \otimes b)$ is an integer no greater than 2^p, and is expressible in p bits.

Finally, Line 7 is exact simply because $y = -\text{err}(a \otimes b)$ can be expressed in p bits. Hence, $ab = x + y$.

If round-to-even tiebreaking is used, x and y are nonadjacent by analogy to Corollary 44. ∎

6.3.6 Expansion Scaling

The following algorithm, which multiplies an expansion by a floating-point value, is the second key new result of this chapter.

Theorem 54 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, and let b be a p-bit value where $p \geq 4$. Suppose that the components of e are sorted in order of increasing magnitude, except that any of the e_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{2m} h_i = be$, where the components of h are also in order of increasing magnitude, except that any of the h_i may be zero. Furthermore, if e is nonadjacent and round-to-even tiebreaking is used, then h is nonadjacent.*

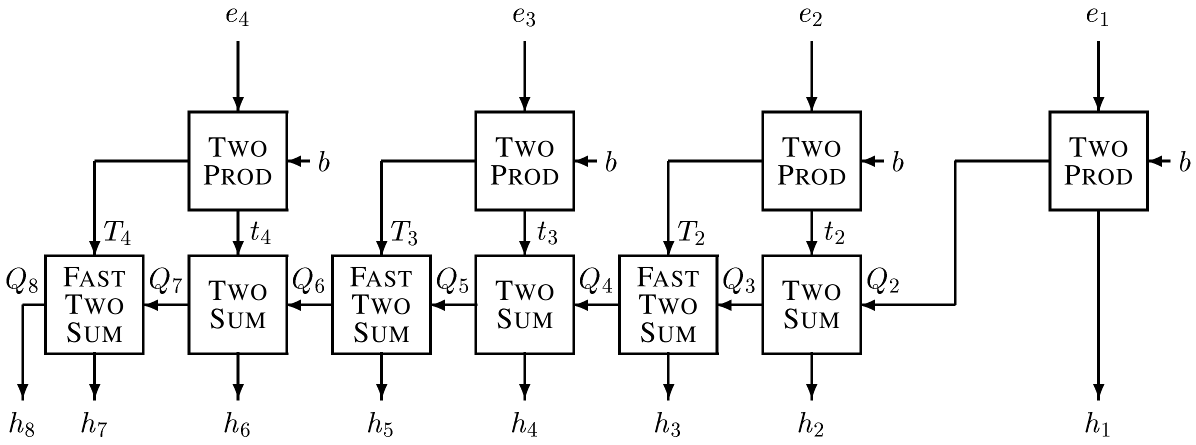


Figure 6.14: Operation of SCALE-EXPANSION.

SCALE-EXPANSION(e, b)
1 $(Q_2, h_1) \Leftarrow$ TWO-PRODUCT(e_1, b)
2 **for** $i \Leftarrow 2$ **to** m
3 $\quad (T_i, t_i) \Leftarrow$ TWO-PRODUCT(e_i, b)
4 $\quad (Q_{2i-1}, h_{2i-2}) \Leftarrow$ TWO-SUM(Q_{2i-2}, t_i)
5 $\quad (Q_{2i}, h_{2i-1}) \Leftarrow$ FAST-TWO-SUM(T_i, Q_{2i-1})
6 $h_{2m} \Leftarrow Q_{2m}$
7 **return** h

As illustrated in Figure 6.14, SCALE-EXPANSION multiplies each component of e by b and sums the results. It should be apparent why the final expansion h is the desired product, but it is not so obvious why the components of h are guaranteed to be nonoverlapping and in increasing order. Two lemmata will aid the proof.

Lemma 55 *Let e_i and e_j be two nonoverlapping nonzero components of e, with $i < j$ and $|e_i| < |e_j|$. Let T_i be a correctly rounded approximation to $e_i b$, and let $T_i + t_i$ be a two-component expansion exactly equal to $e_i b$. (Such an expansion is produced by Line 3, but here is defined also for $i = 1$.) Then t_i is too small in magnitude to overlap the double-width product $e_j b$. Furthermore, if e_i and e_j are nonadjacent, then t_i is not adjacent to $e_j b$.*

Proof: By scaling e and b by appropriate powers of 2 (thereby shifting their exponents without changing their significands), one may assume without loss of generality that e_j and b are integers with magnitude less than 2^p, and that $|e_i| < 1$ (and hence a radix point falls between e_j and e_i).

It follows that $e_j b$ is an integer, and $|e_i b| < 2^p$. The latter fact and exact rounding imply that $|t_i| \leq \frac{1}{2}$. Hence, $e_j b$ and t_i do not overlap.

If e_i and e_j are nonadjacent, scale e so that e_j is an integer and $|e_i| < \frac{1}{2}$. Then $|t_i| \leq \frac{1}{4}$, so $e_j b$ and t_i are not adjacent. ∎

Lemma 56 *For some i, let r be the smallest integer such that $|e_i| < 2^r$ (hence e_i does not overlap 2^r). Then $|Q_{2i}| \leq 2^r|b|$, and thus $|h_{2i-1}| \leq 2^{r-1}\mathrm{ulp}(b)$.*

Proof: The inequality $|Q_{2i}| \leq 2^r|b|$ holds for $i = 1$ after Line 1 is executed even if Q_2 is rounded to a larger magnitude, because $|e_1b| < 2^r|b|$, and $2^r|b|$ is expressible in p bits. For larger values of i, the bound is proven by induction. Assume that R is the smallest integer such that $|e_{i-1}| < 2^R$; by the inductive hypothesis, $|Q_{2i-2}| \leq 2^R|b|$.

Because e_i and e_{i-1} are nonoverlapping, e_i must be a multiple of 2^R. Suppose that r is the smallest integer such that $|e_i| < 2^r$; then $|e_i| \leq 2^r - 2^R$.

Lines 3, 4, and 5 compute Q_{2i}, an approximation of $Q_{2i-2} + e_ib$, and are subject to roundoff error in Lines 4 and 5. Suppose that Q_{2i-2} and e_ib have the same sign, that $|Q_{2i-2}|$ has its largest possible value $2^R|b|$, and that $|e_i|$ has its largest possible value $2^r - 2^R$. For these assignments, roundoff does not occur in Lines 4 and 5, and $|Q_{2i}| = |Q_{2i-2} + e_ib| = 2^r|b|$. Otherwise, roundoff may occur, but the monotonicity of floating-point multiplication and addition ensures that $|Q_{2i}|$ cannot be larger than $2^r|b|$.

The inequality $|h_{2i-1}| \leq 2^{r-1}\text{ulp}(b)$ is guaranteed by exact rounding because h_{2i-1} is the roundoff term associated with the computation of Q_{2i} in Line 5. ■

Proof of Theorem 54: One can prove inductively that at the end of each iteration of the **for** loop, the invariant $Q_{2i} + \sum_{j=1}^{2i-1} h_j = \sum_{j=1}^{i} e_jb$ holds. Certainly this invariant holds for $i = 1$ after Line 1 is executed. By induction on Lines 3, 4, and 5, one can deduce that the invariant holds for all (relevant values of) i. (The use of FAST-TWO-SUM in Line 5 will be justified shortly.) Thus, after Line 6 is executed, $\sum_{j=1}^{2m} h_j = b\sum_{j=1}^{m} e_j$.

I shall prove that the components of h are nonoverlapping by showing that each time a component of h is written, that component is smaller than and does not overlap either the accumulator Q nor any of the remaining products (e_jb); hence, the component cannot overlap any portion of their sum. The first claim, that each component h_j does not overlap the accumulator Q_{j+1}, is true because h_j is the roundoff error incurred while computing Q_{j+1}.

To show that each component of h is smaller than and does not overlap the remaining products, I shall consider h_1, the remaining odd components of h, and the even components of h separately. The component h_1, computed by Line 1, does not overlap the remaining products ($e_2b, e_3b, \ldots$) by virtue of Lemma 55. The even components, which are computed by Line 4, do not overlap the remaining products because, by application of Lemma 36 to Line 4, a component $|h_{2i-2}|$ is no larger than $|t_i|$, which is bounded in turn by Lemma 55.

Odd components of h, computed by Line 5, do not overlap the remaining products by virtue of Lemma 56, which guarantees that $|h_{2i-1}| \leq 2^{r-1}\text{ulp}(b)$. The remaining products are all multiples of $2^r\text{ulp}(b)$ (because the remaining components of e are multiples of 2^r).

If round-to-even tiebreaking is used, the output of each TWO-SUM, FAST-TWO-SUM, and TWO-PRODUCT statement is nonadjacent. If e is nonadjacent as well, the arguments above are easily modified to show that h is nonadjacent.

The use of FAST-TWO-SUM in Line 5 is justified because $|T_i| \geq |Q_{2i-1}|$ (except if $T_i = 0$, in which case FAST-TWO-SUM still works correctly). To see this, recall that e_i is a multiple of 2^R (with R defined as in Lemma 56), and consider two cases: if $|e_i| = 2^R$, then T_i is computed exactly and $t_i = 0$, so $|T_i| = 2^R|b| \geq |Q_{2i-2}| = |Q_{2i-1}|$. If $|e_i|$ is larger than 2^R, it is at least twice as large, and hence T_i is at least $2|Q_{2i-2}|$, so even if roundoff occurs and t_i is not zero, $|T_i| > |Q_{2i-2}| + |t_i| \geq |Q_{2i-1}|$.

Note that if an input component e_i is zero, then two zero output components are produced, and the accumulator value is unchanged ($Q_{2i} = Q_{2i-2}$). ■

The following corollary demonstrates that SCALE-EXPANSION is compatible with FAST-EXPANSION-SUM.

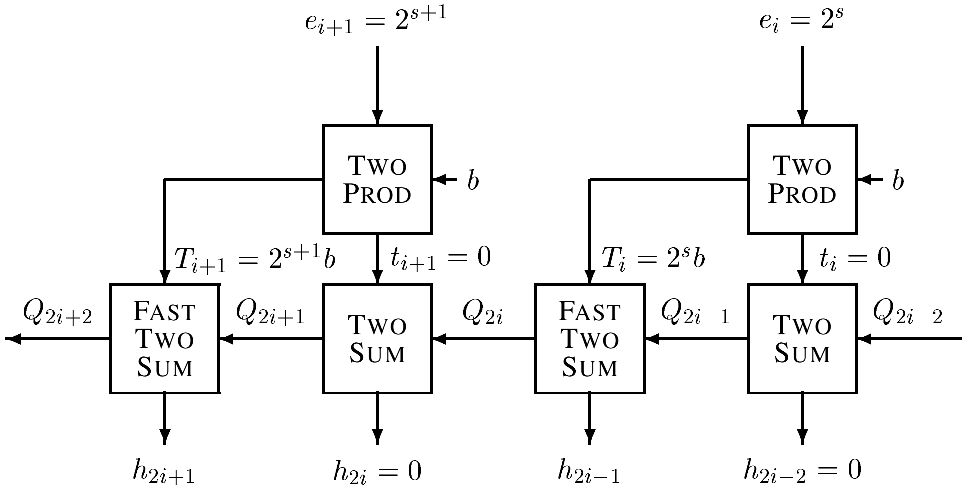


Figure 6.15: An adjacent pair of one-bit components in a strongly nonoverlapping input expansion may cause SCALE-EXPANSION to produce an adjacent pair of one-bit components in the output expansion.

Corollary 57 *If e is strongly nonoverlapping and round-to-even tiebreaking is used, then h is strongly nonoverlapping.*

Proof: Because e is nonoverlapping, h is nonoverlapping by Theorem 54. We have also seen that if e is nonadjacent, then h is nonadjacent and hence strongly nonoverlapping; but e is only guaranteed to be strongly nonoverlapping, and may deviate from nonadjacency.

Suppose two successive components e_i and e_{i+1} are adjacent. By the definition of strongly nonoverlapping, e_i and e_{i+1} are both powers of two and are not adjacent to e_{i-1} or e_{i+2}. Let s be the integer satisfying $e_i = 2^s$ and $e_{i+1} = 2^{s+1}$. For these components the multiplication of Line 3 is exact, so $T_i = 2^s b$, $T_{i+1} = 2^{s+1}b$, and $t_i = t_{i+1} = 0$. Applying Lemma 36 to Line 4, $h_{2i-2} = h_{2i} = 0$. However, the components h_{2i-1} and h_{2i+1} may cause difficulty (see Figure 6.15). We know h is nonoverlapping, but can these two components be adjacent to their neighbors or each other?

The arguments used in Theorem 54 to prove that h is nonadjacent, if e is nonadjacent and round-to-even tiebreaking is used, can be applied here as well to show that h_{2i-1} and h_{2i+1} are not adjacent to any components of h produced before or after them, but they may be adjacent to each other. Assume that h_{2i-1} and h_{2i+1} are adjacent (they cannot be overlapping).

h_{2i+1} is computed in Line 5 from T_{i+1} and Q_{2i+1}. The latter addend is equal to Q_{2i}, because $t_{i+1} = 0$. Q_{2i} is not adjacent to h_{2i-1}, because they are produced in Line 5 from a FAST-TWO-SUM operation. Hence, the least significant nonzero bit of h_{2i+1} (that is, the bit that causes it to be adjacent to h_{2i-1}) must have come from T_{i+1}, which is equal to $2^{s+1}b$. It follows that h_{2i+1} is a multiple of $2^{s+1}\mathrm{ulp}(b)$. Because $|e_{i+1}| < 2^{s+2}$, Lemma 56 implies that $|h_{2i+1}| \leq 2^{s+1}\mathrm{ulp}(b)$. Hence, $|h_{2i+1}| = 2^{s+1}\mathrm{ulp}(b)$.

Similarly, because $|e_i| < 2^{s+1}$, Lemma 56 implies that $|h_{2i-1}| \leq 2^s\mathrm{ulp}(b)$. The components h_{2i+1} and h_{2i-1} can only be adjacent in the case $|h_{2i-1}| = 2^s\mathrm{ulp}(b)$. In this case, both components are expressible in one bit.

Hence, each adjacent pair of one-bit components in the input can give rise to an isolated adjacent pair of one-bit components in the output, but no other adjacent components may appear. If e is strongly nonoverlapping, so is h. ■

6.3.7 Compression and Approximation

The algorithms for manipulating expansions do not usually express their results in the most compact form. In addition to the interspersed zero components that have already been mentioned (and are easily eliminated), it is also common to find components that represent only a few bits of an expansion's value. Such fragmentation rarely becomes severe, but it can cause the largest component of an expansion to be a poor approximation of the value of the whole expansion; the largest component may carry as little as one bit of significance. Such a component may result, for instance, from cancellation during the subtraction of two nearly equal expansions.

The COMPRESS algorithm below finds a compact form for an expansion. More importantly, COMPRESS guarantees that the largest component is a good approximation to the whole expansion. If round-to-even tiebreaking is used, COMPRESS also converts nonoverlapping expansions into nonadjacent expansions.

Priest [76] presents a more complicated "Renormalization" procedure that compresses optimally. Its greater running time is rarely justified by the marginal reduction in expansion length, unless there is a need to put expansions in a canonical form.

Theorem 58 *Let $e = \sum_{i=1}^{m} e_i$ be a nonoverlapping expansion of m p-bit components, where $m \geq 3$. Suppose that the components of e are sorted in order of increasing magnitude, except that any of the e_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h (nonadjacent if round-to-even tiebreaking is used) such that $h = \sum_{i=1}^{n} h_i = e$, where the components h_i are in order of increasing magnitude. If $h \neq 0$, none of the h_i will be zero. Furthermore, the largest component h_n approximates h with an error smaller than* $\mathrm{ulp}(h_n)$.

```
COMPRESS(e)
    Q ⇐ e_m
    bottom ⇐ m
    for i ⇐ m − 1 downto 1
        (Q, q) ⇐ FAST-TWO-SUM(Q, e_i)
        if q ≠ 0 then
            g_bottom ⇐ Q
            bottom ⇐ bottom − 1
            Q ⇐ q
    g_bottom ⇐ Q
    top ⇐ 1
    for i ⇐ bottom + 1 to m
        (Q, q) ⇐ FAST-TWO-SUM(g_i, Q)
        if q ≠ 0 then
            h_top ⇐ Q
            top ⇐ top + 1
    h_top ⇐ Q
    Set n (the length of h) to top
    return h
```

Figure 6.16 illustrates the operation of COMPRESS. For clarity, g and h are presented as two separate arrays in the COMPRESS pseudocode, but they can be combined into a single working array without conflict by replacing every occurrence of "g" with "h".

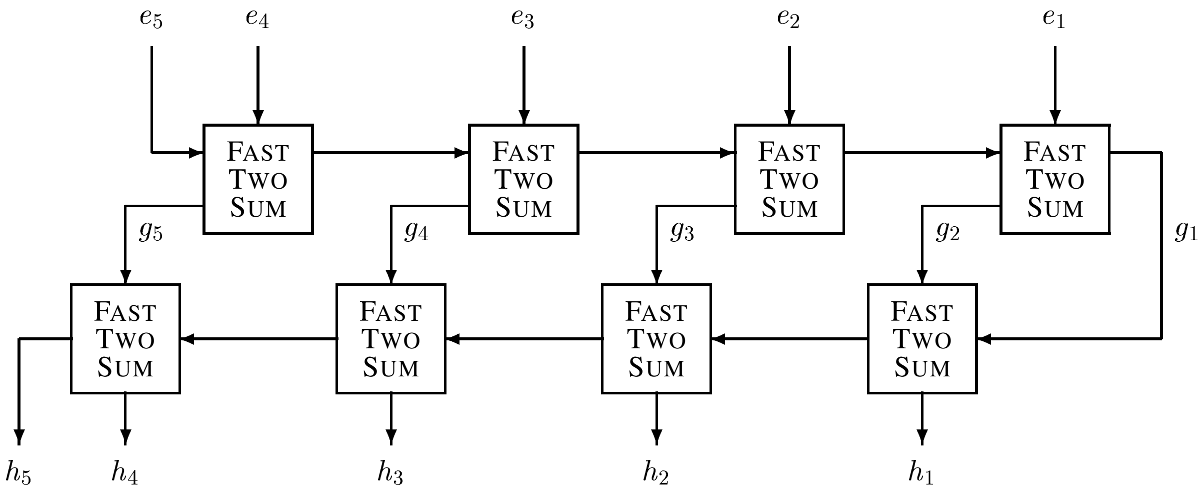


Figure 6.16: Operation of COMPRESS when no zero-elimination occurs.

Proof Sketch: COMPRESS works by traversing the expansion from largest to smallest component, then back from smallest to largest, replacing each adjacent pair with its two-component sum. The first traversal, from largest to smallest, does most of the compression. The expansion $g_m + g_{m-1} + \cdots + g_{bottom}$ produced by Lines 1 through 9 has the property that $g_{j-1} \leq \text{ulp}(g_j)$ for all j (and thus successive components overlap by at most one bit). This fact follows because the output of FAST-TWO-SUM in Line 4 has the property that $q \leq \frac{1}{2}\text{ulp}(Q)$, and the value of q thus produced can only be increased slightly by the subsequent addition of smaller nonoverlapping components.

The second traversal, from smallest to largest, clips any overlapping bits. The use of FAST-TWO-SUM in Line 12 is justified because the property that $g_{i-1} \leq \text{ulp}(g_i)$ guarantees that Q (the sum of the components that are smaller than g_i) is smaller than g_i. The expansion $h_{top} + h_{top-1} + \cdots + h_2 + h_1$ is nonoverlapping (nonadjacent if round-to-even tiebreaking is used) because FAST-TWO-SUM produces nonoverlapping (nonadjacent) output.

During the second traversal, an approximate total is maintained in the accumulator Q. The component h_{n-1} is produced by the last FAST-TWO-SUM operation that produces a roundoff term; this roundoff term is no greater than $\frac{1}{2}\text{ulp}(h_n)$. Hence, the sum $|h_{n-1} + h_{n-2} + \cdots + h_2 + h_1|$ (where the components of h are nonoverlapping) is less than $\text{ulp}(h_n)$, therefore $|h - h_n| < \text{ulp}(h_n)$. ■

To ensure that h_n is a good approximation to h, only the second traversal is necessary; however, the first traversal is more effective in reducing the number of components. The fastest way to approximate e is to simply sum its components from smallest to largest; by the reasoning used above, the result errs by less than one ulp. This observation is the basis for an APPROXIMATE procedure that is used in the predicates of Section 6.5.

Theorem 58 is not the strongest statement that can be made about COMPRESS. COMPRESS is effective even if the components of the input expansion have a certain limited amount of overlap. Furthermore, the bound for $|h - h_n|$ is not tight. (I conjecture that the largest possible relative error is exhibited by a number that contains a nonzero bit every pth bit; observe that $1 + \frac{1}{2}\text{ulp}(1) + \frac{1}{4}[\text{ulp}(1)]^2 + \cdots$ cannot be further compressed.) These improvements complicate the proof and are not explored here.

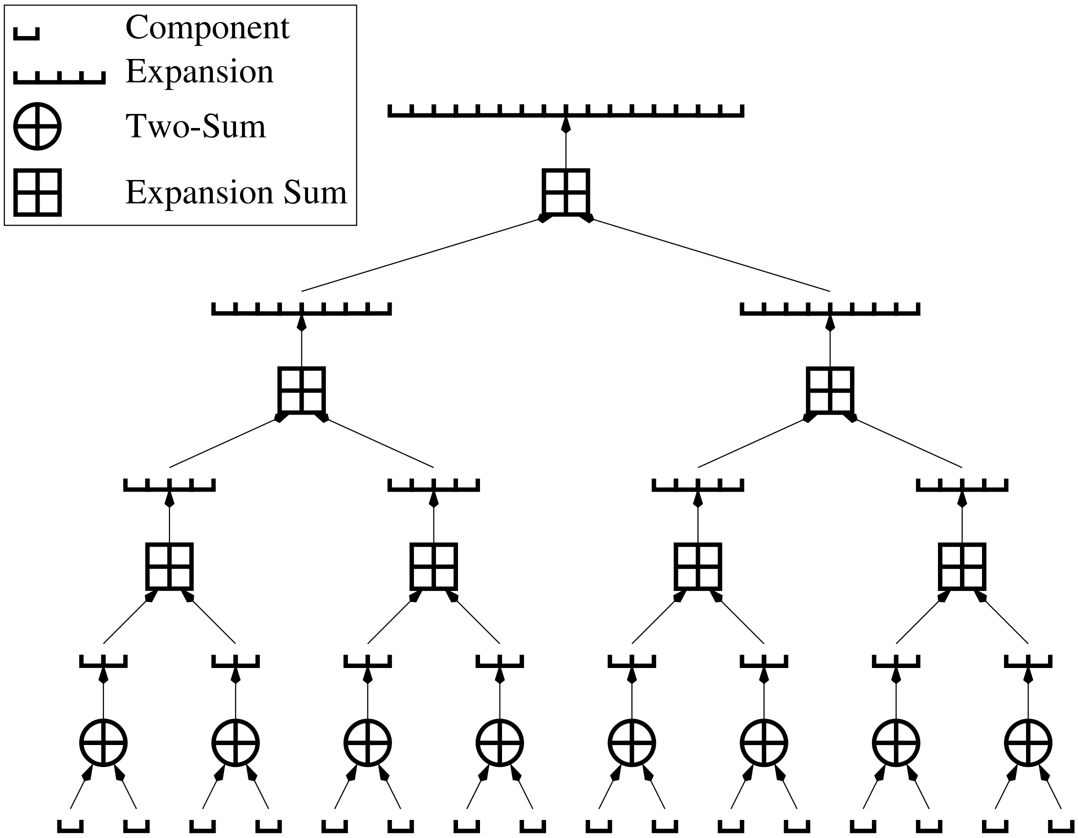


Figure 6.17: Distillation of sixteen p-bit floating-point values.

6.3.8 Other Operations

Distillation is the process of summing k unordered p-bit values. Distillation can be performed by the divide-and-conquer algorithm of Priest [76], which uses any expansion addition algorithm to sum the values in a tree-like fashion as illustrated in Figure 6.17. Each p-bit addend is a leaf of the tree, and each interior node represents a call to an expansion addition algorithm. If EXPANSION-SUM is used (and zero elimination is not), then it does not matter whether the tree is balanced; distillation will take precisely $\frac{1}{2}k(k-1)$ TWO-SUM operations, regardless of the order in which expansions are combined. If FAST-EXPANSION-SUM is used, the speed of distillation depends strongly on the balance of the tree. A well-balanced tree will yield an $\mathcal{O}(k \log k)$ distillation algorithm, an asymptotic improvement over distilling with EXPANSION-SUM. As I have mentioned, it is usually fastest to use an unrolled EXPANSION-SUM to create expansions of length four, and FAST-EXPANSION-SUM with zero elimination to sum these expansions.

To find the product of two expansions e and f, use SCALE-EXPANSION (with zero elimination) to form the expansions $ef_1, ef_2, \ldots$, then sum these using a distillation tree.

Division cannot always, of course, be performed exactly, but it can be performed to arbitrary precision by an iterative algorithm that employs multiprecision addition and multiplication. Consult Priest [76] for one such algorithm.

The easiest way to compare two expansions is to subtract one from the other, and test the sign of the result. An expansion's sign can be easily tested because of the nonoverlapping property; simply check the sign of the expansion's most significant nonzero component. (If zero elimination is employed, check the component with the largest index.) A nonoverlapping expansion is equal to zero if and only if all its components are equal to zero.

6.4 Adaptive Precision Arithmetic

6.4.1 Why Adaptivity?

Exact arithmetic is expensive, and when it can be avoided, it should be. Some applications do not need exact results, but require the absolute error of a result to fall below some threshold. If this threshold is known before the computation is performed, it is economical to employ *adaptivity by prediction*. One writes several procedures, each of which approximates the result with a different degree of precision, and with a correspondingly different speed. Error bounds are derived for each of these procedures; these bounds are typically much cheaper to compute than the approximations themselves, except for the least precise approximation. For any particular input, the application computes the error bounds and uses them to choose the procedure that will attain the necessary accuracy most cheaply.

Sometimes, however, one cannot determine whether a computation will be accurate enough before it is done. An example is when one wishes to bound the relative error, rather than the absolute error, of the result. (A special case is determining the sign of an expression; the result must have relative error less than one.) The result may prove to be much larger than its error bound, and low precision arithmetic will suffice, or it may be so close to zero that it is necessary to evaluate it exactly to satisfy the bound on relative error. One cannot generally know in advance how much precision is needed.

In the context of determinant evaluation for computational geometry, Fortune and Van Wyk [36] suggest using a floating-point filter. An expression is evaluated approximately in hardware precision arithmetic first. Forward error analysis determines whether the approximate result can be trusted; if not, an exact result is computed. If the exact computation is only needed occasionally, the application is slowed only a little.

One might hope to improve this idea further by computing a sequence of increasingly accurate results, testing each one in turn for accuracy. Alas, whenever an exact result is required, one suffers both the cost of the exact computation and the additional burden of computing several approximate results in advance. Fortunately, it is often possible to use intermediate results as stepping stones to more accurate results; work already done is not discarded but is refined.

6.4.2 Making Arithmetic Adaptive

FAST-TWO-SUM, TWO-SUM, and TWO-PRODUCT each have the feature that they can be broken into two parts: Line 1, which computes an approximate result, and the remaining lines, which calculate the roundoff error. The latter, more expensive calculation can be delayed until it is needed, if it is ever needed at all. In this sense, these routines can be made *adaptive*, so that they only produce as much of the result as is needed. I describe here how to achieve the same effect with more general expressions.

Any expression composed of addition, subtraction, and multiplication operations can be calculated adaptively in a manner that defines a natural sequence of intermediate results whose accuracy it is appropriate to test. Such a sequence is most easily described by considering the tree associated with the expression, as in Figure 6.18(a). The leaves of this tree represent floating-point operands, and its internal nodes represent operations. Replace each node whose children are both leaves with the sum $x_i + y_i$, where x_i represents the

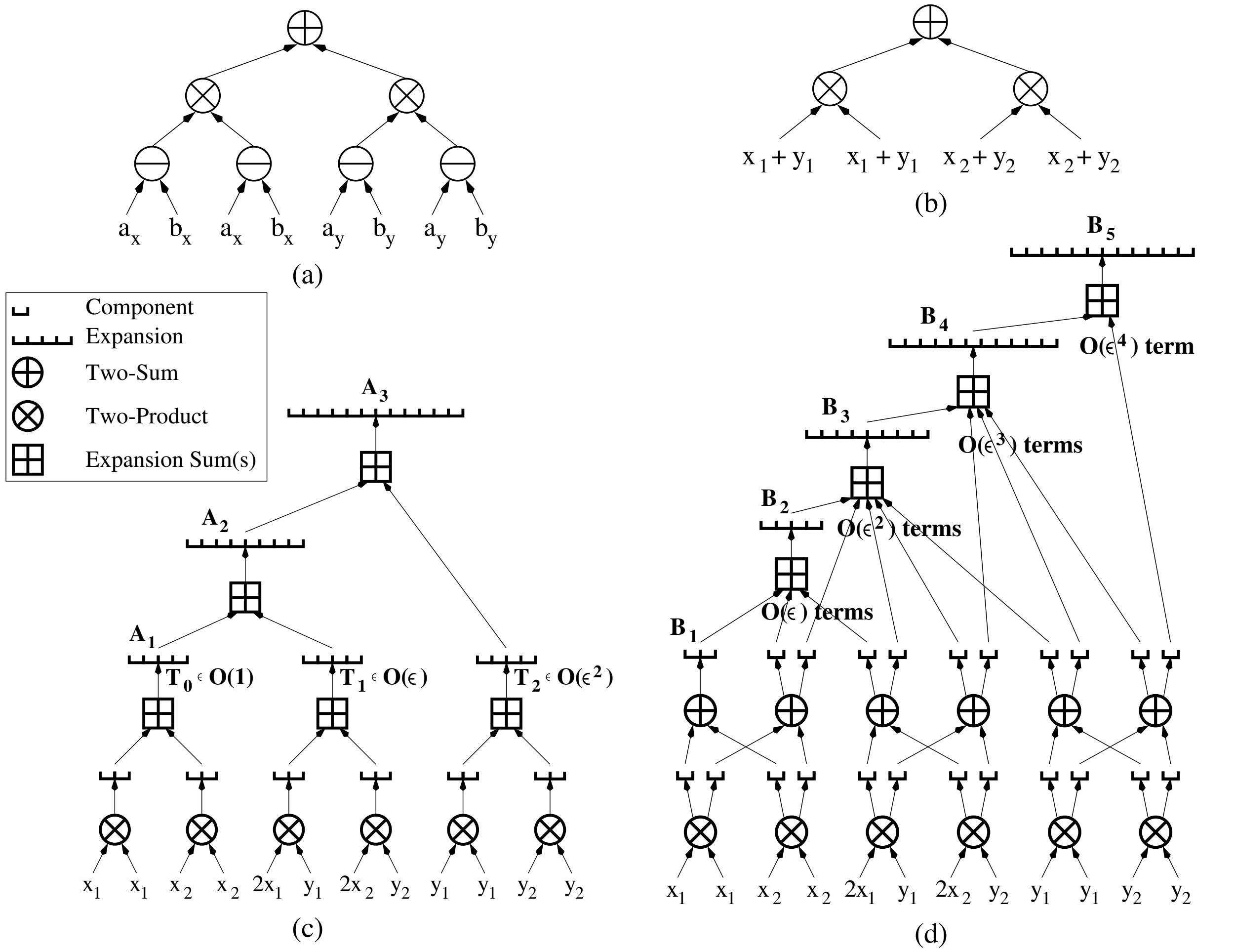


Figure 6.18: (a) Formula for the square of the distance between two points a and b. (b) The lowest subexpressions in the tree are expressed as the sum of an approximate value and a roundoff error. (c) A simple incremental adaptive method for evaluating the expression. The approximations A_1 and A_2 are generated and tested in turn. The final expansion A_3 is exact. Each A_i includes all terms of size $\mathcal{O}(\epsilon^{i-1})$ or larger, and hence has error no greater than $\mathcal{O}(\epsilon^i)$. (d) Incremental adaptivity taken to an extreme. The three subexpression trees T_0, T_1, and T_2 are themselves calculated adaptively. Each B_i incorporates only the terms needed to reduce its error to $\mathcal{O}(\epsilon^i)$.

approximate value of the subexpression, and y_i represents the roundoff error incurred while calculating x_i, as illustrated in Figure 6.18(b). Expand the expression to form a polynomial.

In the expanded expression, the terms containing many occurrences of y variables (roundoff errors) are dominated by terms containing fewer occurrences. As an example, consider the expression $(a_x - b_x)^2 + (a_y - b_y)^2$ (Figure 6.18), which calculates the square of the distance between two points in the plane. Set $a_x - b_x = x_1 + y_1$ and $a_y - b_y = x_2 + y_2$. The resulting expression, expanded in full, is

$$(x_1^2 + x_2^2) + (2x_1y_1 + 2x_2y_2) + (y_1^2 + y_2^2). \tag{6.5}$$

It is significant that each y_i is small relative to its corresponding x_i. Using standard terminology from forward error analysis [97], the quantity $\frac{1}{2}\mathrm{ulp}(1)$ is called the *machine epsilon*, denoted ϵ. Recall that exact rounding guarantees that $|y_i| \leq \epsilon|x_i|$; the quantity ϵ bounds the *relative error* $\mathrm{err}(a \circledast b)/(a \circledast b)$ of any basic floating-point operation. Note that $\epsilon = 2^{-p}$. In IEEE 754 double precision arithmetic, $\epsilon = 2^{-53}$; in single precision, $\epsilon = 2^{-24}$.

Expression 6.5 can be divided into three parts, having magnitudes of $\mathcal{O}(1)$, $\mathcal{O}(\epsilon)$, and $\mathcal{O}(\epsilon^2)$, respectively. Denote these parts T_0, T_1, and T_2. More generally, for any expression expanded in this manner, let T_i be the sum of all products containing i of the y variables, so that T_i has magnitude $\mathcal{O}(\epsilon^i)$.

One can obtain an approximation A_j with error no larger than $\mathcal{O}(\epsilon^j)$ by computing exactly the sum of the first j terms, T_0 through T_{j-1}. The sequence $A_1, A_2, \ldots$ of increasingly accurate approximations can be formed incrementally; A_j is the exact sum of A_{j-1} and T_{j-1}. Members of this sequence are generated and tested, as illustrated in Figure 6.18(c), until one is sufficiently accurate.

The approximation A_j is not the way to achieve an error bound of $\mathcal{O}(\epsilon^j)$ with the least amount of work. For instance, a floating-point calculation of $(x_1^2 + x_2^2)$ using no exact arithmetic techniques will achieve an $\mathcal{O}(\epsilon)$ error bound, albeit with a larger constant than the error bound for A_1. Experimentation has shown that the fastest adaptive predicates are written by calculating an approximation having bound $\mathcal{O}(\epsilon^j)$ as quickly as possible, then moving on to the next smaller order of magnitude. Improvements in the constant prefacing each error bound will make a difference in only a small number of cases. Hence, I will consider two modifications to the technique just described. The first modification computes each error bound from the minimum possible number of roundoff terms. This lazy approach is presented here for instructional purposes, but is not generally the fastest. The second modification I will consider, and the one I recommend for use, is faster because it spends less time collating small data.

The first modification is to compute the subexpressions T_0, T_1, and T_2 adaptively as well. The method is the same: replace each bottom-level subexpression of T_0 (and T_1 and T_2) with the sum of an approximate result and an error term, and expand T_0 into a sum of terms of differing order. An approximation B_j having an error bound of magnitude $\mathcal{O}(\epsilon^j)$ may be found by approximating each T term with error $\mathcal{O}(\epsilon^j)$. Because the term T_k has magnitude at most $\mathcal{O}(\epsilon^k)$, it need not be approximated with any better relative error than $\mathcal{O}(\epsilon^{j-k})$.

Figure 6.18(d) shows that the method is as lazy as possible, in the sense that each approximation B_j uses only the roundoff terms needed to obtain an $\mathcal{O}(\epsilon^j)$ error bound. (Note that this is true at every level of the tree. It is apparent in the figure that every roundoff term produced is fed into a different calculation than the larger term produced with it.) However, the laziest approach is not necessarily the fastest approach. The cost of this method is unnecessarily large for two reasons. First, recall from Section 6.3.8 that FAST-EXPANSION-SUM is most effective when terms are summed in a balanced manner. The additions in Figure 6.18(d) are less well balanced than those in Figure 6.18(c). Second, and more importantly, there is a good deal of overhead for keeping track of many small pieces of the sum; the method sacrifices most of the advantages of the compressed form in which expansions are represented. Figure 6.18(d) does not fully reveal how convoluted this extreme form of adaptivity can become for larger expressions. In addition to having an unexpectedly large overhead, this method can be exasperating for the programmer.

The final method for incremental adaptivity I shall present, which is used to derive the geometric predicates in Section 6.5, falls somewhere between the two described above. As in the first method, compute the sequence $A_1, A_2, \ldots$, and define also $A_0 = 0$. We have seen that the error bound of each term A_j may be improved from $\mathcal{O}(\epsilon^j)$ to $\mathcal{O}(\epsilon^{j+1})$ by (exactly) adding T_j to it. However, because the magnitude of T_j itself is $\mathcal{O}(\epsilon^j)$, the same effect can be achieved (with a slightly worse constant in the error bound) by computing T_j with floating-point arithmetic and tolerating the roundoff error, rather than computing T_j exactly. Hence, an approximation C_{j+1} having an $\mathcal{O}(\epsilon^{j+1})$ error bound is computed by summing A_j and an inexpensive *correctional term*, which is merely the floating-point approximation to T_j, as illustrated in Figure 6.19. C_{j+1} is nearly as accurate as A_{j+1} but takes much less work to compute. If C_{j+1} is not sufficiently accurate, then it

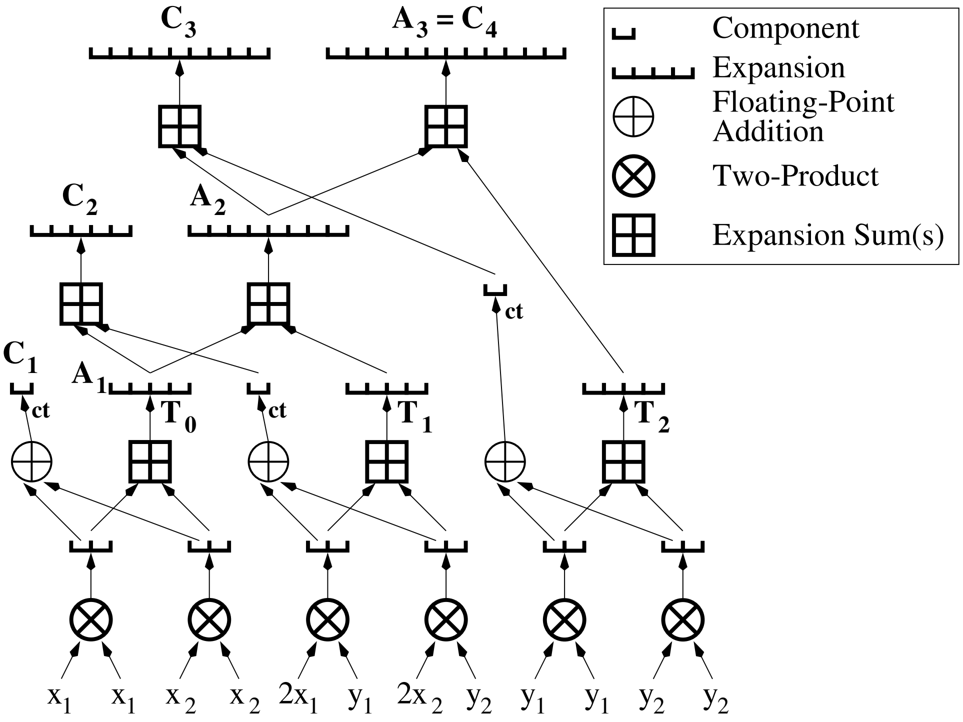


Figure 6.19: An adaptive method of intermediate complexity that is frequently more efficient than the other two. Each C_i achieves an $\mathcal{O}(\epsilon^i)$ error bound by adding an inexpensive correctional term (labeled "ct") to A_{i-1}.

is thrown away, and the exact value of T_j is computed and added to A_j to form A_{j+1}. This scheme reuses the work done in performing exact calculations, but does not reuse the correctional terms. (In practice, no speed can be gained by reusing the correctional terms.)

The first value (C_1) computed by this method is an approximation to T_0; if C_1 is sufficiently accurate, it is unnecessary to compute the y terms, or use any exact arithmetic techniques, at all. (Recall that the y terms are more expensive to compute than the x terms.) This first test is identical to Fortune and Van Wyk's floating-point filter.

This method does more work during each stage of the computation than the first method, but typically terminates one stage earlier. It is slower when the exact result must be computed, but is faster in applications that rarely need an exact result. In some cases, it may be desirable to test certain members of both sequences A and C for accuracy; the predicates defined in Section 6.5 do so.

All three methods of making expressions adaptive are mechanical and can be automated. An expression compiler similar to Fortune and Van Wyk's [37], discussed in Section 6.2, would be valuable; it could convert expressions into code that evaluates these expressions adaptively, with automatically computed error bounds.

The reader may wonder if writing an expression in sum-of-products form isn't inefficient. In ordinary floating-point arithmetic it often is, but it seems to make little difference when using the exact arithmetic algorithms of Section 6.3. Indeed, the multiplication operation described in Section 6.3.8 multiplies two expansions by expanding the product into sum-of-products form.

These ideas are not exclusively applicable to the multiple-component approach to arbitrary precision arithmetic. They will work with multiple-digit formats as well, though the details differ.

6.5 Implementation of Geometric Predicates

6.5.1 The Orientation and Incircle Tests

Let a, b, c, and d be four points in the plane. Define a procedure ORIENT2D(a, b, c) that returns a positive value if the points a, b, and c are arranged in counterclockwise order, a negative value if the points are in clockwise order, and zero if the points are collinear. A more common (but less symmetric) interpretation is that ORIENT2D returns a positive value if c lies to the left of the directed line ab; for this purpose the orientation test is used by many geometric algorithms.

Define also a procedure INCIRCLE(a, b, c, d) that returns a positive value if d lies inside the oriented circle abc. By *oriented circle*, I mean the unique (and possibly degenerate) circle through a, b, and c, with these points occurring in counterclockwise order about the circle. (If these points occur in clockwise order, INCIRCLE will reverse the sign of its output, as if the circle's exterior were its interior.) INCIRCLE returns zero if and only if all four points lie on a common circle. Both ORIENT2D and INCIRCLE have the symmetry property that interchanging any two of their parameters reverses the sign of their result.

These definitions extend trivially to arbitrary dimensions. For instance, ORIENT3D(a, b, c, d) returns a positive value if d lies below the oriented plane passing through a, b, and c. By *oriented plane*, I mean that a, b, and c appear in counterclockwise order when viewed from above the plane. (One can apply a *left-hand rule*: orient your left hand with fingers curled to follow the circular sequence abc. If your thumb points toward d, ORIENT3D returns a positive value.) To generalize the orientation test to dimensionality d, let $u_1, u_2, \ldots, u_d$ be the unit vectors; ORIENT is defined so that ORIENT$(u_1, u_2, \ldots, u_d, 0) = 1$.

In any dimension, the orientation and incircle tests may be implemented as matrix determinants. For three dimensions:

$$\text{ORIENT3D}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \tag{6.6}$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \tag{6.7}$$

$$\text{INSPHERE}(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & a_x^2 + a_y^2 + a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2 + b_y^2 + b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2 + c_y^2 + c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2 + d_y^2 + d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2 + e_y^2 + e_z^2 & 1 \end{vmatrix} \tag{6.8}$$

$$= \begin{vmatrix} a_x - e_x & a_y - e_y & a_z - e_z & (a_x - e_x)^2 + (a_y - e_y)^2 + (a_z - e_z)^2 \\ b_x - e_x & b_y - e_y & b_z - e_z & (b_x - e_x)^2 + (b_y - e_y)^2 + (b_z - e_z)^2 \\ c_x - e_x & c_y - e_y & c_z - e_z & (c_x - e_x)^2 + (c_y - e_y)^2 + (c_z - e_z)^2 \\ d_x - e_x & d_y - e_y & d_z - e_z & (d_x - e_x)^2 + (d_y - e_y)^2 + (d_z - e_z)^2 \end{vmatrix} \tag{6.9}$$

These formulae generalize to other dimensions in the obvious way. Expressions 6.6 and 6.7 can be shown to be equivalent by simple algebraic transformations, as can Expressions 6.8 and 6.9 with a little more effort. These equivalences are unsurprising because one expects the result of any orientation or incircle test not to change if all the points undergo an identical translation in the plane. Expression 6.7, for instance, follows from Expression 6.6 by translating each point by $-d$.

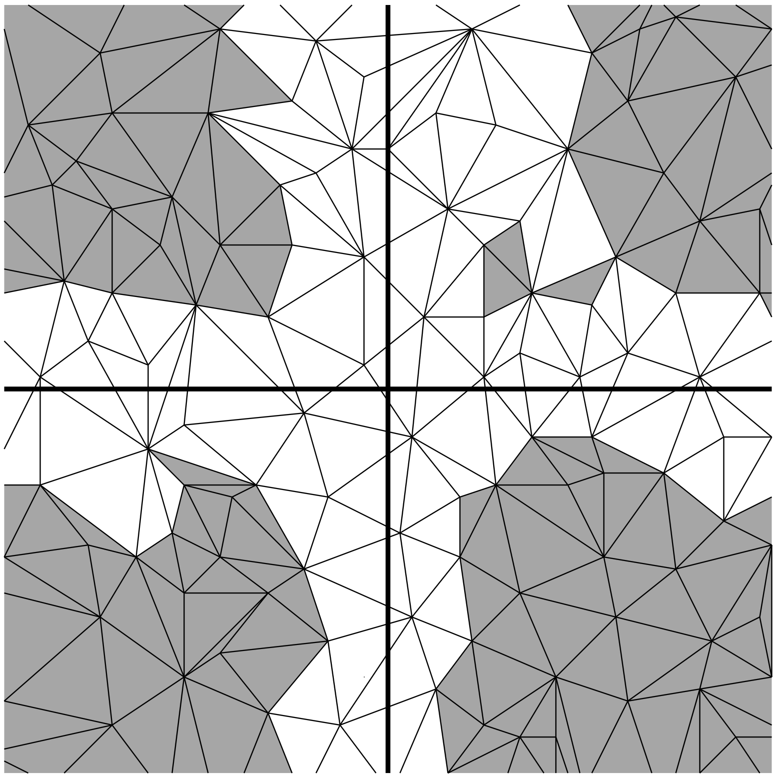


Figure 6.20: Shaded triangles can be translated to the origin without incurring roundoff error (Lemma 40). In most triangulations, such triangles are the common case.

When computing these determinants using the techniques of Section 6.3, the choice between Expressions 6.6 and 6.7, or between 6.8 and 6.9, is not straightforward. In principle, Expression 6.6 seems preferable because it can only produce a 96-component expansion, whereas Expression 6.7 could produce an expansion having 192 components. These numbers are somewhat misleading, however, because with zero-elimination, expansions rarely grow longer than six components in real applications. Nevertheless, Expression 6.7 takes roughly 25% more time to compute in exact arithmetic, and Expression 6.9 takes about 50% more time than Expression 6.8. The disparity likely increases in higher dimensions.

Nevertheless, the mechanics of error estimation turn the tide in the other direction. Important as a fast exact test is, it is equally important to avoid exact tests whenever possible. Expressions 6.7 and 6.9 tend to have smaller errors (and correspondingly smaller error estimates) because their errors are a function of the relative coordinates of the points, whereas the errors of Expressions 6.6 and 6.8 are a function of the absolute coordinates of the points.

In most geometric applications, the points that serve as parameters to geometric tests tend to be close to each other. Commonly, their absolute coordinates are much larger than the distances between them. By translating the points so they lie near the origin, working precision is freed for the subsequent calculations. Hence, the errors and error bounds for Expressions 6.7 and 6.9 are generally much smaller than for Expressions 6.6 and 6.8. Furthermore, the translation can often be done without roundoff error. Figure 6.20 demonstrates a toy problem: suppose ORIENT2D is used to find the orientation of each triangle in a triangulation. Thanks to Lemma 40, any shaded triangle can be translated so that one of its vertices lies at the origin without roundoff error; the white triangles may or may not suffer from roundoff during such translation. If the complete triangulation is much larger than the portion illustrated, only a small proportion of the triangles (those near a coordinate axis) will suffer roundoff. Because exact translation is the common case, my adaptive geometric predicates test for and exploit this case.

Once a determinant has been chosen for evaluation, there are several methods to evaluate it. A number of methods are surveyed by Fortune and Van Wyk [36], and only their conclusion is repeated here. The cheapest method of evaluating the determinant of a 5×5 or smaller matrix seems to be by dynamic programming applied to cofactor expansion. Evaluate the $\binom{d}{2}$ determinants of all 2×2 minors of the first two columns, then the $\binom{d}{3}$ determinants of all 3×3 minors of the first three columns, and so on. All four of my predicates use this method.

6.5.2 ORIENT2D

My implementation of ORIENT2D computes a sequence of up to four results (labeled A through D) as illustrated in Figure 6.21. The exact result D may be as long as sixteen components, but zero elimination is used, so a length of two to six components is more common in practice.

A, B, and C are logical places to test the accuracy of the result before continuing. In most applications, the majority of calls to ORIENT2D will end with the floating-point approximation A, which is computed without resort to any exact arithmetic techniques. Although the four-component expansion B, like A, has an error of $\mathcal{O}(\epsilon)$, it is an appropriate value to test because B is the exact result if the four subtractions at the bottom of the expression tree are performed without roundoff error (corresponding to the shaded triangles in Figure 6.20). Because this is the common case, ORIENT2D explicitly tests for it; execution continues only if roundoff occurred during the translation of coordinates and B is smaller than its error bound. The corrected estimate C has an error bound of $\mathcal{O}(\epsilon^2)$. If C is not sufficiently accurate, the exact determinant D is computed.

There are two unusual features of this test, both of which arise because only the sign of the determinant is needed. First, the correctional term added to B to form C is not added exactly; instead, the APPROXIMATE procedure of Section 6.3.7 is used to find an approximation B′ of B, and the correctional term is added to B′ with the possibility of roundoff error. The consequent errors may be of magnitude $\mathcal{O}(\epsilon \mathrm{B})$, which would normally preclude obtaining an error bound of $\mathcal{O}(\epsilon^2)$. However, the sign of the determinant is only questionable if B is of magnitude $\mathcal{O}(\epsilon)$, so an $\mathcal{O}(\epsilon^2)$ error bound for C can be established.

The second interesting feature is that, if C is not sufficiently accurate, no more approximations are computed before computing the exact determinant. To understand why, consider three collinear points a, b, and c; the determinant defined by these points is zero. If a coordinate of one of these points is perturbed by a single ulp, the determinant typically increases to $\mathcal{O}(\epsilon)$. Hence, one might guess that when a determinant is no larger than $\mathcal{O}(\epsilon^2)$, it is probably zero. This intuition seems to hold in practice for all the predicates considered herein, on both random and "practical" point sets. Determinants that don't stop with approximation C are nearly always zero.

The derivation of error bounds for these values is tricky, so an example is given here. The easiest way to apply forward error analysis to an expression whose value is calculated in floating-point arithmetic is to express the exact value of each subexpression in terms of the computed value plus an unknown error term whose magnitude is bounded. For instance, the error incurred by the computation $x \Leftarrow a \oplus b$ is no larger than $\epsilon|x|$. Furthermore, the error is smaller than $\epsilon|a + b|$. Each of these bounds is useful under different circumstances. If t represents the true value $a + b$, an abbreviated way of expressing these notions is to write $t = x \pm \epsilon|x|$ and $t = x \pm \epsilon|t|$. Henceforth, this notation will be used as shorthand for the relation $t = x + \lambda$ for some λ that satisfies $|\lambda| \leq \epsilon|x|$ and $|\lambda| \leq \epsilon|t|$.

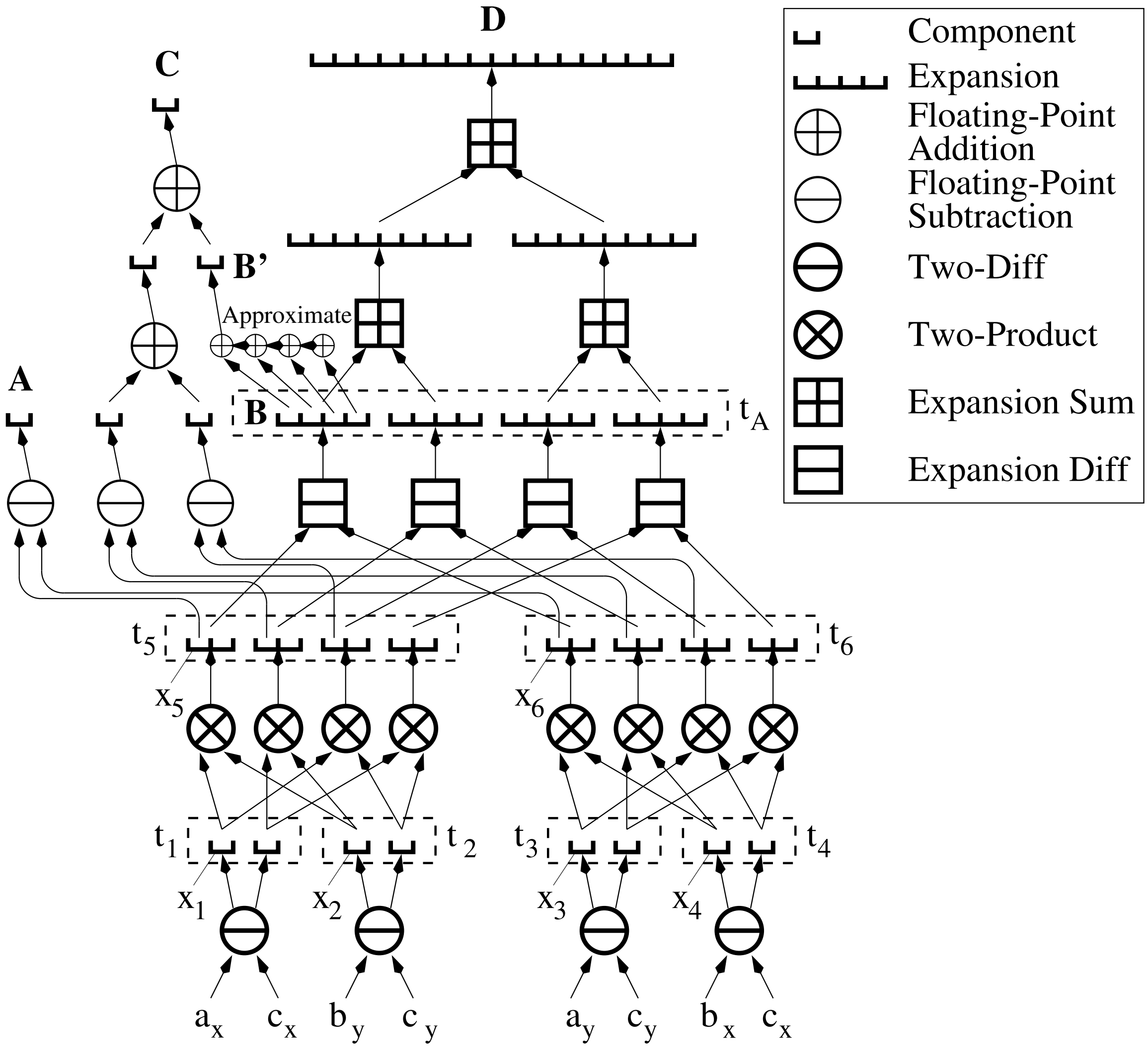


Figure 6.21: Adaptive calculations used by the 2D orientation test. Dashed boxes represent nodes in the original expression tree.

Let us consider the error bound for A. For each subexpression in the expression tree of the orientation test, denote its true (exact) value t_i and its approximate value x_i as follows.

$$
\begin{array}{ll}
t_1 = a_x - c_x & x_1 = a_x \ominus c_x \\
t_2 = b_y - c_y & x_2 = b_y \ominus c_y \\
t_3 = a_y - c_y & x_3 = a_y \ominus c_y \\
t_4 = b_x - c_x & x_4 = b_x \ominus c_x \\
t_5 = t_1 t_2 & x_5 = x_1 \otimes x_2 \\
t_6 = t_3 t_4 & x_6 = x_3 \otimes x_4 \\
t_A = t_5 - t_6 & \mathrm{A} = x_5 \ominus x_6
\end{array}
$$

From these definitions, it is clear that $t_1 = x_1 \pm \epsilon|x_1|$; similar bounds hold for t_2, t_3, and t_4. Observe

Approximation	Error bound
A	$(3\epsilon + 16\epsilon^2) \otimes (\lvert x_5\rvert \oplus \lvert x_6\rvert)$
B′	$(2\epsilon + 12\epsilon^2) \otimes (\lvert x_5\rvert \oplus \lvert x_6\rvert)$
C	$(3\epsilon + 8\epsilon^2) \otimes \lvert \mathrm{B}'\rvert \oplus (9\epsilon^2 + 64\epsilon^3) \otimes (\lvert x_5\rvert \oplus \lvert x_6\rvert)$

Table 6.1: Error bounds for the expansions calculated by ORIENT2D. B′ is a p-bit approximation of the expansion B, computed by the APPROXIMATE procedure. Note that each coefficient is expressible in p bits.

also that $x_5 = x_1 \otimes x_2 = x_1x_2 \pm \epsilon|x_5|$. It follows that

$$\begin{aligned} t_5 = t_1t_2 &= x_1x_2 \pm (2\epsilon + \epsilon^2)|x_1x_2| \\ &= x_5 \pm \epsilon|x_5| \pm (2\epsilon + \epsilon^2)(|x_5| \pm \epsilon|x_5|) \\ &= x_5 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|x_5|. \end{aligned}$$

Similarly, $t_6 = x_6 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)|x_6|$.

It may seem odd to be keeping track of terms smaller than $\mathcal{O}(\epsilon)$, but the effort to find the smallest machine-representable coefficient for each error bound is justified if it ever prevents a determinant computation from becoming more expensive than necessary. An error bound for A can now be derived.

$$\begin{aligned} t_A = t_5 - t_6 &= x_5 - x_6 \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|) \\ &= \mathrm{A} \pm \epsilon|\mathrm{A}| \pm (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|) \end{aligned}$$

One can minimize the effect of the term $\epsilon|\mathrm{A}|$ by taking advantage of the fact that we are only interested in the sign of t_A. One can conclude with certainty that A has the correct sign if

$$(1 - \epsilon)|\mathrm{A}| > (3\epsilon + 3\epsilon^2 + \epsilon^3)(|x_5| + |x_6|),$$

which is true if

$$|\mathrm{A}| \geq (3\epsilon + 6\epsilon^2 + 8\epsilon^3)(|x_5| + |x_6|).$$

This bound is not directly applicable, because its computation will incur roundoff error. To account for this, multiply the coefficient by $(1 + \epsilon)^2$ (a factor of $(1 + \epsilon)$ for the addition of $|x_5|$ and $|x_6|$, and another such factor for the multiplication). Hence, we are secure that the sign of A is correct if

$$|\mathrm{A}| \geq (3\epsilon + 12\epsilon^2 + 24\epsilon^3) \otimes (|x_5| \oplus |x_6|).$$

This bound is not directly applicable either, because the coefficient is not expressible in p bits. Rounding up to the next p-bit number, we have the coefficient $(3\epsilon + 16\epsilon^2)$, which should be exactly computed once at program initialization and reused during each call to ORIENT2D.

Error bounds for A, B′, and C are given in Table 6.1. The bound for B′ takes advantage of Theorem 58, which shows that B′ approximates B with relative error less than 2ϵ. (Recall from Section 6.3.7 that the largest component of B might have only one bit of precision.)

These bounds have the pleasing property that they are zero in the common case that all three input points lie on a horizontal or vertical line. Hence, although ORIENT2D usually resorts to exact arithmetic when given collinear input points, it only performs the approximate test (A) in the two cases that occur most commonly in practice.

Double precision ORIENT2D timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Collinear
Approximate (6.7)	0.15	0.15	0.16
Exact (6.6)	6.56	6.89	6.31
Exact (6.7)	8.35	8.48	8.13
Exact (6.6), MPFUN	92.85	94.03	84.97
Adaptive A (6.7), approximate	0.28	0.27	0.22
Adaptive B (6.7)			1.89
Adaptive C (6.7)			2.14
Adaptive D (6.7), exact			8.35
LN adaptive (6.7), approximate	0.32	n/a	
LN adaptive (6.7), exact		n/a	4.43

Table 6.2: Timings for ORIENT2D on a DEC 3000/700 with a 225 MHz Alpha processor. All determinants use the 2D version of either Expression 6.6 or the more stable Expression 6.7 as indicated. The first two columns indicate input points generated from a uniform random distribution and a geometric random distribution. The third column considers two points chosen from one of the random distributions, and a third point chosen to be approximately collinear to the first two. Timings for the adaptive tests are categorized according to which result was the last generated. Each timing is an average of 60 or more randomly generated inputs. For each such input, time was measured by a Unix system call before and after 10,000 iterations of the predicate. Individual timings vary by approximately 10%. Timings of Bailey's MPFUN package and Fortune and Van Wyk's LN package are included for comparison.

Compiler effects affect the implementation of ORIENT2D. By separating the calculation of A and the remaining calculations into two procedures, with the former calling the latter if necessary, I reduced the time to compute A by 25%, presumably because of improvements in the compiler's ability to perform register allocation.

Table 6.2 lists timings for ORIENT2D, given random inputs. Observe that the adaptive test, when it stops at the approximate result A, takes nearly twice as long as the approximate test because of the need to compute an error bound. The table includes a comparison with Bailey's MPFUN [4], chosen because it is the fastest portable and freely available arbitrary precision package I know of. ORIENT2D coded with my (nonadaptive) algorithms is roughly thirteen times faster than ORIENT2D coded with MPFUN.

Also included is a comparison with an orientation predicate for 53-bit integer inputs, created by Fortune and Van Wyk's LN. The LN-generated orientation predicate is quite fast because it takes advantage of the fact that it is restricted to bounded integer inputs. My exact tests cost less than twice as much as LN's; this seems like a reasonable price to pay for the ability to handle arbitrary exponents in the input.

These timings are not the whole story; LN's static error estimate is typically much larger than the runtime error estimate used for adaptive stage A, and LN uses only two stages of adaptivity, so the LN-generated predicates are slower in some applications, as Section 6.5.4 will demonstrate. It is significant that for 53-bit integer inputs, the multiple-stage predicates will rarely pass stage B because the initial translation is usually done without roundoff error; hence, the LN-generated ORIENT2D usually takes more than twice as long to produce an exact result. It should be emphasized, however, that these are not inherent differences between LN's multiple-digit integer approach and my multiple-component floating-point approach; LN could, in principle, employ the same runtime error estimate and a similar multiple-stage adaptivity scheme.

Approximation	Error bound
A	$(7\epsilon + 56\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
B′	$(3\epsilon + 28\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
C	$(3\epsilon + 8\epsilon^2) \otimes \vert\mathrm{B}'\vert \oplus (26\epsilon^2 + 288\epsilon^3) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$

$$\begin{aligned}
\alpha_a &= |x_1| \otimes (|x_6| \oplus |x_7|) \\
&= |a_z \ominus d_z| \otimes (|(b_x \ominus d_x) \otimes (c_y \ominus d_y)| \oplus |(b_y \ominus d_y) \otimes (c_x \ominus d_x)|) \\
\alpha_b &= |b_z \ominus d_z| \otimes (|(c_x \ominus d_x) \otimes (a_y \ominus d_y)| \oplus |(c_y \ominus d_y) \otimes (a_x \ominus d_x)|) \\
\alpha_c &= |c_z \ominus d_z| \otimes (|(a_x \ominus d_x) \otimes (b_y \ominus d_y)| \oplus |(a_y \ominus d_y) \otimes (b_x \ominus d_x)|)
\end{aligned}$$

Table 6.3: Error bounds for the expansions calculated by ORIENT3D.

Double precision ORIENT3D timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Coplanar
Approximate (6.7)	0.25	0.25	0.25
Exact (6.6)	33.30	38.54	32.90
Exact (6.7)	42.69	48.21	42.41
Exact (6.6), MPFUN	260.51	262.08	246.64
Adaptive A (6.7), approximate	0.61	0.60	0.62
Adaptive B (6.7)			12.98
Adaptive C (6.7)			15.59
Adaptive D (6.7), exact			27.29
LN adaptive (6.7), approximate	0.85	n/a	
LN adaptive (6.7), exact		n/a	18.11

Table 6.4: Timings for ORIENT3D on a DEC 3000/700. All determinants are Expression 6.6 or the more stable Expression 6.7 as indicated. Each timing is an average of 120 or more randomly generated inputs. For each such input, time was measured by a Unix system call before and after 10,000 iterations of the predicate. Individual timings vary by approximately 10%.

6.5.3 ORIENT3D, INCIRCLE, and INSPHERE

Figure 6.22 illustrates the implementation of ORIENT3D, which is similar to the ORIENT2D implementation. A is the standard floating-point result. B is exact if the subtractions at the bottom of the tree incur no roundoff. C represents a drop in the error bound from $\mathcal{O}(\epsilon)$ to $\mathcal{O}(\epsilon^2)$. D is the exact determinant.

Error bounds for the largest component of each of these expansions are given in Table 6.3, partly in terms of the variables x_1, x_6, and x_7 in Figure 6.22. The bounds are zero if all four input points share the same x-, y-, or z-coordinate, so only the approximate test is needed in the most common instances of coplanarity.

Table 6.4 lists timings for ORIENT3D, given random inputs. The error bound for A is expensive to compute, and increases the amount of time required to perform the approximate test in the adaptive case by a factor of two and a half. The gap between my exact algorithm and MPFUN is smaller than in the 2D case, but is still a factor of nearly eight.

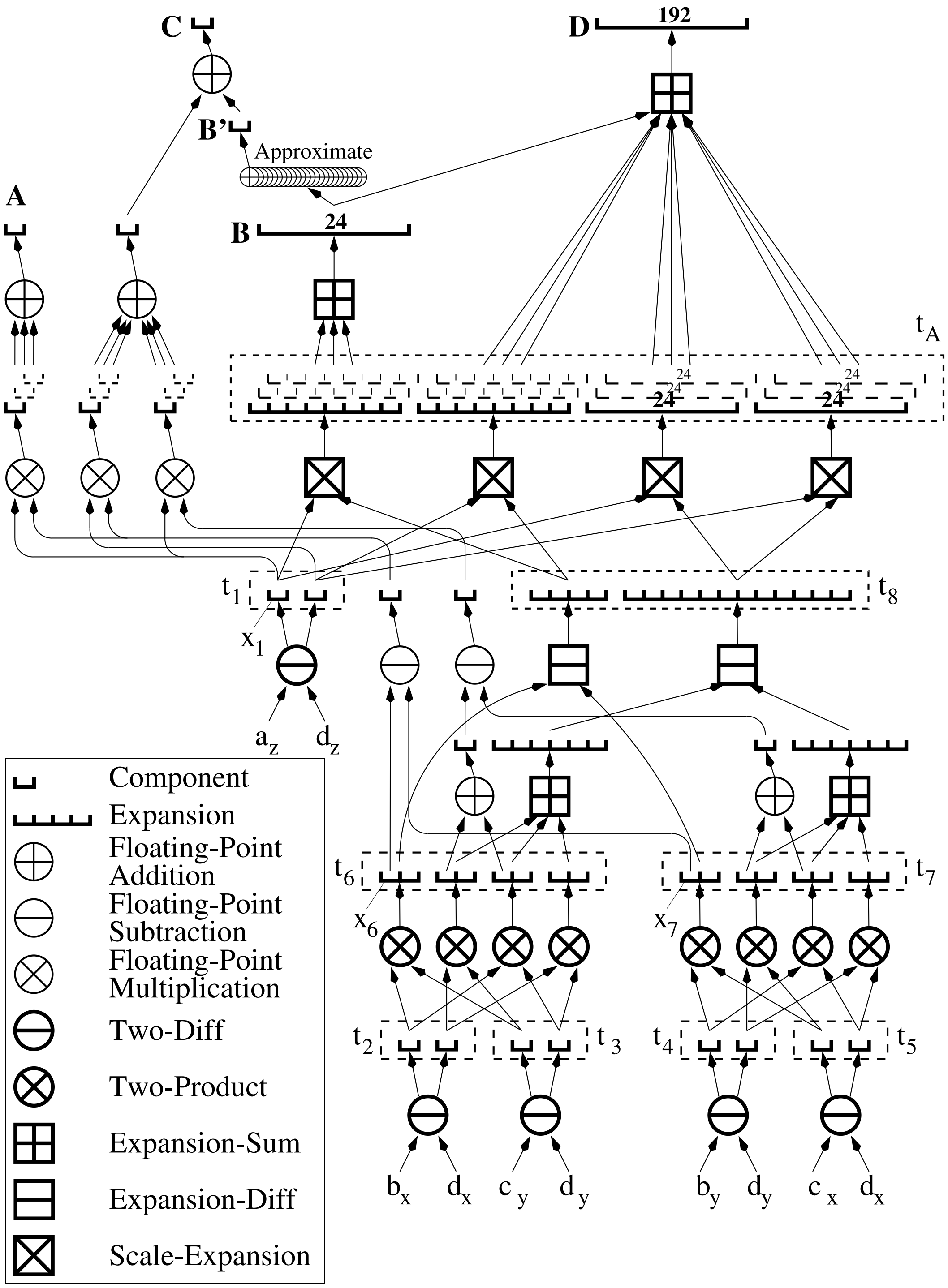


Figure 6.22: Adaptive calculations used by the 3D orientation test. Bold numbers indicate the length of an expansion. Only part of the expression tree is shown; two of the three cofactors are omitted, but their results appear as dashed components and expansions.

Approximation	Error bound
A	$(10\epsilon + 96\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
B′	$(4\epsilon + 48\epsilon^2) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$
C	$(3\epsilon + 8\epsilon^2) \otimes \lvert \mathrm{B}' \rvert \oplus (44\epsilon^2 + 576\epsilon^3) \otimes (\alpha_a \oplus \alpha_b \oplus \alpha_c)$

$$\begin{aligned}
\alpha_a &= ((a_x \ominus d_x)^2 \oplus (a_y \ominus d_y)^2) \otimes (|(b_x \ominus d_x) \otimes (c_y \ominus d_y)| \oplus |(b_y \ominus d_y) \otimes (c_x \ominus d_x)|) \\
\alpha_b &= ((b_x \ominus d_x)^2 \oplus (b_y \ominus d_y)^2) \otimes (|(c_x \ominus d_x) \otimes (a_y \ominus d_y)| \oplus |(c_y \ominus d_y) \otimes (a_x \ominus d_x)|) \\
\alpha_c &= ((c_x \ominus d_x)^2 \oplus (c_y \ominus d_y)^2) \otimes (|(a_x \ominus d_x) \otimes (b_y \ominus d_y)| \oplus |(a_y \ominus d_y) \otimes (b_x \ominus d_x)|)
\end{aligned}$$

Table 6.5: Error bounds for the expansions calculated by INCIRCLE. Squares are approximate.

Double precision INCIRCLE timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Cocircular
Approximate (6.9)	0.31	0.28	0.30
Exact (6.8)	71.66	83.01	75.34
Exact (6.9)	91.71	118.30	104.44
Exact (6.8), MPFUN	350.77	343.61	348.55
Adaptive A (6.9), approximate	0.64	0.59	0.64
Adaptive B (6.9)			44.56
Adaptive C (6.9)			48.80
Adaptive D (6.9), exact			78.06
LN adaptive (6.9), approximate	1.33	n/a	
LN adaptive (6.9), exact		n/a	32.44

Table 6.6: Timings for INCIRCLE on a DEC 3000/700. All determinants are the 2D version of either Expression 6.8 or the more stable Expression 6.9 as indicated. Each timing is an average of 100 or more randomly generated inputs, except adaptive stage D. (It is difficult to generate cases that reach stage D.) For each such input, time was measured by a Unix system call before and after 1,000 iterations of the predicate. Individual timings vary by approximately 10%.

Oddly, the table reveals that D is calculated more quickly than the exact result is calculated by the non-adaptive version of ORIENT3D. The explanation is probably that D is only computed when the determinant is zero or very close to zero, hence the lengths of the intermediate expansions are smaller than usual, and the computation time is less. Furthermore, when some of the point coordinates are translated without roundoff error, the adaptive predicate ignores branches of the expression tree that evaluate to zero.

INCIRCLE is implemented similarly to ORIENT3D, as the determinants are similar. The corresponding error bounds appear in Table 6.5, and timings appear in Table 6.6.

Timings for INSPHERE appear in Table 6.7. This implementation differs from the other tests in that, due to programmer laziness, D is not computed incrementally from B; rather, if C is not accurate enough, D is computed from scratch. Fortunately, C is usually accurate enough.

The LN exact tests have an advantage of a factor of roughly 2.5 for INCIRCLE and 4 for INSPHERE, so the cost of handling floating-point operands is greater with the larger expressions. As with the orientation

Double precision INSPHERE timings in microseconds			
Points Method	Uniform Random	Geometric Random	Nearly Cospherical
Approximate (6.9)	0.93	0.95	0.93
Exact (6.8)	324.22	378.94	347.16
Exact (6.9)	374.59	480.28	414.13
Exact (6.8), MPFUN	1,017.56	1,019.89	1,059.87
Adaptive A (6.9), approximate	2.13	2.14	2.14
Adaptive B (6.9)			166.21
Adaptive C (6.9)			171.74
Adaptive D (6.8), exact			463.96
LN adaptive (6.9), approximate	2.35	n/a	
LN adaptive (6.9), exact		n/a	116.74

Table 6.7: Timings for INSPHERE on a DEC 3000/700. All determinants are Expression 6.8 or the more stable Expression 6.9 as indicated. Each timing is an average of 25 or more randomly generated inputs, except adaptive stage D. For each such input, time was measured by a Unix system call before and after 1,000 iterations of the predicate. Individual timings vary by approximately 10%.

tests, this cost is mediated by better error bounds and four-stage adaptivity.

The timings for the exact versions of all four predicates show some sensitivity to the distribution of the operands; they take 5% to 30% longer to execute with geometrically distributed operands (whose exponents vary widely) than with uniformly distributed operands. This difference occurs because the intermediate and final expansions are larger when the operands have broadly distributed exponents. The exact orientation predicates are cheapest when their inputs are collinear/coplanar, because of the smaller expansions that result, but this effect does not occur for the exact incircle predicates.

6.5.4 Performance in Two Triangulation Programs

To evaluate the effectiveness of the adaptive tests in applications, I integrated them into Triangle and Pyramid, and recorded the speeds of 2D divide-and-conquer Delaunay triangulation and 3D incremental Delaunay tetrahedralization under various conditions. For both 2D and 3D, three types of inputs were tested: uniform random points, points lying (approximately) on the boundary of a circle or sphere, and a square or cubic grid of lattice points, tilted so as not to be aligned with the coordinate axes. The latter two were chosen for their nastiness. The lattices have been tilted using approximate arithmetic, so they are not perfectly cubical, and the exponents of their coordinates vary enough that LN cannot be used. (I have also tried perfect lattices with 53-bit integer coordinates, but ORIENT3D and INSPHERE never pass stage B; the perturbed lattices are preferred here because they occasionally force the predicates into stage C or D.)

The results for 2D, which appear in Table 6.8, indicate that the four-stage predicates add about 8% to the total running time for randomly distributed input points, mainly because of the error bound tests. For the more difficult point sets, the penalty may be as great as 30%. Of course, this penalty applies precisely for the point sets that are most likely to cause difficulties when exact arithmetic is not available.

The results for 3D, outlined in Table 6.9, are less pleasing. The four-stage predicates add about 35% to the total running time for randomly distributed input points; for points distributed approximately on the

2D divide-and-conquer Delaunay triangulation			
	Uniform Random	Perimeter of Circle	Tilted Grid
Input sites	1,000,000	1,000,000	1,000,000
ORIENT2D calls			
Adaptive A, approximate	9,497,314	6,291,742	9,318,610
Adaptive B			121,081
Adaptive C			118
Adaptive D, exact			3
Average time, μs	0.32	0.38	0.33
LN approximate	9,497,314	2,112,284	n/a
LN exact		4,179,458	n/a
LN average time, μs	0.35	3.16	n/a
INCIRCLE calls			
Adaptive A, approximate	7,596,885	3,970,796	7,201,317
Adaptive B		50,551	176,470
Adaptive C		120	47
Adaptive D, exact			4
Average time, μs	0.65	1.11	1.67
LN approximate	6,077,062	0	n/a
LN exact	1,519,823	4,021,467	n/a
LN average time, μs	7.36	32.78	n/a
Program running time, seconds			
Approximate version	57.3	59.9	48.3
Robust version	61.7	64.7	62.2
LN robust version	116.0	214.6	n/a

Table 6.8: Statistics for 2D divide-and-conquer Delaunay triangulation of several point sets. Timings are accurate to within 10%.

surface of a sphere, the penalty is a factor of eleven. Ominously, however, the penalty for the tilted grid is uncertain, because the tetrahedralization program using approximate arithmetic failed to terminate. A debugger revealed that the point location routine was stuck in an infinite loop because a geometric inconsistency had been introduced into the mesh due to roundoff error. Robust arithmetic is not always slower after all.

In these programs (and likely in any program), three of the four-stage predicates (INSPHERE being the exception) are faster than their LN equivalents. This is a surprise, considering that the four-stage predicates accept 53-bit floating-point inputs whereas the LN-generated predicates are restricted to 53-bit integer inputs. However, the integer predicates would probably outperform the floating-point predicates if they were to adopt the same runtime error estimate and a similar four-stage adaptivity scheme.

3D incremental Delaunay tetrahedralization			
	Uniform Random	Surface of Sphere	Tilted Grid
Input sites	10,000	10,000	10,000
ORIENT3D calls			
Adaptive A, approximate	2,735,668	1,935,978	5,542,567
Adaptive B			602,344
Adaptive C			1,267,423
Adaptive D, exact			28,185
Average time, μs	0.72	0.72	4.12
LN approximate	2,735,668	1,935,920	n/a
LN exact		58	n/a
LN average time, μs	0.99	1.00	n/a
INSPHERE calls			
Adaptive A, approximate	439,090	122,273	3,080,312
Adaptive B		180,383	267,162
Adaptive C		1,667	548,063
Adaptive D, exact			
Average time, μs	2.23	96.45	48.12
LN approximate	438,194	104,616	n/a
LN exact	896	199,707	n/a
LN average time, μs	2.50	70.82	n/a
Program running time, seconds			
Approximate version	4.3	3.0	∞
Robust version	5.8	34.1	108.5
LN robust version	6.5	30.5	n/a

Table 6.9: Statistics for 3D incremental Delaunay tetrahedralization of several point sets. Timings are accurate to within 10%. The approximate code failed to terminate on the tilted grid input.

6.6 Caveats

Unfortunately, the arbitrary precision arithmetic routines described herein are not universally portable; both hardware and compilers can prevent them from functioning correctly.

Compilers can interfere by making invalid optimizations based on misconceptions about floating-point arithmetic. For instance, a clever but incorrect compiler might cause expansion arithmetic algorithms to fail by deriving the “fact” that b_{virtual}, computed by Line 2 of FAST-TWO-SUM, is equal to b, and optimizing the subtraction away. This optimization would be valid if computers stored arbitrary real numbers, but is incorrect for floating-point numbers. Unfortunately, not all compiler developers are aware of the importance of maintaining correct floating-point language semantics, but as a whole, they seem to be improving. Goldberg [44, §3.2.3] presents several related examples of how carefully designed numerical algorithms can be utterly ruined by incorrect optimizations.

Even floating-point units that use binary arithmetic with exact rounding, including those that conform to the IEEE 754 standard, can have subtle properties that undermine the assumptions of the algorithms. The most common such difficulty is the presence of extended precision internal floating-point registers, such as those on the Intel 80486 and Pentium processors. While such registers usually improve the stability of floating-point calculations, they cause the methods described herein for determining the roundoff of an operation to fail. There are several possible workarounds for this problem. In C, it is possible to designate a variable as volatile, implying that it must be stored to memory. This ensures that the variable is rounded to a p-bit significand before it is used in another operation. Forcing intermediate values to be stored to memory and reloaded can slow down the algorithms significantly, and there is a worse consequence. Even a volatile variable could be *doubly rounded*, being rounded once to the internal extended precision format, then rounded again to single or double precision when it is stored to memory. The result after double rounding is not always the same as it would be if it had been correctly rounded to the final precision, and Priest [77, page 103] describes a case wherein the roundoff error produced by double rounding may not be expressible in p bits. This might be alleviated by a more complex (and slower) version of FAST-TWO-SUM. A better solution is to configure one's processor to round internally to double precision. While most processors with internal extended precision registers can be thus configured, and most compilers provide support for manipulating processor control state, such support varies between compilers and is not portable. Nevertheless, the speed advantage of multiple-component methods makes it well worth the trouble to learn the right incantation to correctly configure your processor.

The algorithms do work correctly without special treatment on most current Unix workstations. Nevertheless, users should be careful when trying the routines, or moving to a new platform, to ensure that the underlying assumptions of the method are not violated.

6.7 Conclusions

The algorithms presented herein are simple and fast; looking at Figure 6.9, it is difficult to imagine how expansions could be summed with fewer operations without special hardware assistance. Two features of these techniques account for the improvement in speed relative to other techniques, especially for numbers whose precision is only a few components in length. The first is the relaxation of the usual condition that numbers be normalized to fixed digit positions. Instead, one enforces the much weaker condition that expansions be nonoverlapping (or strongly nonoverlapping). Expansions can be summed and the resulting components made nonoverlapping at a cost of six floating-point operations and one comparison per component. It seems unlikely that normalization to fixed digit positions can be done so quickly in a portable way on current processors. The second feature to which I attribute the improved speed is the fact that most packages require expensive conversions between ordinary floating-point numbers and the packages' internal formats. With the techniques Priest and I describe, no conversions are necessary.

The reader may be misled and attribute the whole difference between my algorithms and MPFUN to the fact that I store double precision components, while MPFUN stores single precision digits, and imagine the difference would go away if MPFUN were reimplemented in double precision. Such a belief betrays a misunderstanding of how MPFUN works. MPFUN uses double precision arithmetic internally, and obtains exact results by using digits narrow enough that they can be multiplied exactly. Hence, MPFUN's half-precision digits are an integral part of its approach: to calculate exactly by avoiding roundoff error. The surprise of multiple-component methods is that reasonable speed can be attained by allowing roundoff to happen, then accounting for it after the fact.

As well as being fast, multiple-component algorithms are also reasonably portable, making no assumptions other than that a machine has binary arithmetic with exact rounding (and round-to-even tiebreaking if FAST-EXPANSION-SUM is to be used instead of LINEAR-EXPANSION-SUM). No representation-dependent tricks like bit-masking to extract exponent fields are used. There are still machines that cannot execute these algorithms correctly, but their numbers seem to be dwindling as the IEEE standard becomes entrenched.

Perhaps the greatest limitation of the multiple-component approach is that while it easily extends the precision of floating-point numbers, there is no simple way to extend the exponent range without losing much of the speed. The obvious approach, associating a separate exponent field with each component, is sure to be too slow. A more promising approach is to express each multiprecision number as a *multiexpansion* consisting of digits of very large radix, where each digit is an expansion coupled with an exponent. In this scheme, the true exponent of a component is the sum of the component's own exponent and the exponent of the expansion that contains it. The fast algorithms described in this chapter can be used to add or multiply individual digits; digits are normalized by standard methods (such as those used by MPFUN). IEEE double precision values have an exponent range of -1022 to 1023, so one could multiply digits of radix 2^{1000} with a simple expansion multiplication algorithm, or digits of radix 2^{2000} with a slightly more complicated one that splits each digit in half before multiplying.

The C code I have made publicly available might form the beginning of an extensive library of arithmetic routines similar to MPFUN, but a great deal of work remains to be done. In addition to the problem of expanding the exponent range, there is one problem that is particular to the multiple-component approach: it is not possible to use FFT-based multiplication algorithms without first renormalizing each expansion to a multiple-digit form. This normalization is not difficult to do, but it costs time and puts the multiple-component method at a disadvantage relative to methods that keep numbers in digit form as a matter of course.

As Priest points out, multiple-component algorithms can be used to implement extended (but finite) precision arithmetic as well as exact arithmetic; simply compress and then truncate each result to a fixed number of components. Perhaps the greatest potential of these algorithms lies not with arbitrary precision libraries, but in providing a fast and simple way to extend slightly the precision of critical variables in numerical algorithms. Hence, it would not be difficult to provide a routine that quickly computes the intersection point of two segments with double precision endpoints, correctly rounded to a double precision result. If an algorithm can be made significantly more stable by using double or quadruple precision for a few key values, it may save a researcher from spending a great deal of time devising and analyzing a stabler algorithm; Priest [77, §5.1] offers several examples. Speed considerations may make it untenable to accomplish this by calling a standard extended precision library. The techniques Priest and I have developed are simple enough to be coded directly in numerical algorithms, avoiding function call overhead and conversion costs.

A useful tool in coding such algorithms would be an expression compiler similar to Fortune and Van Wyk's LN [37, 36], which converts an expression into exact arithmetic code, complete with error bound derivation and floating-point filters. Such a tool could also automate the process of breaking an expression into adaptive stages as described in Section 6.4.

To see how adaptivity can be used for more than just determining the sign of an expression, suppose one wishes to find, with relative error no greater than 1%, the center d of a circle that passes through the three points a, b, and c. One may use the following expressions.

$$d_x = c_x - \frac{\begin{vmatrix} a_y - c_y & (a_x - c_x)^2 + (a_y - c_y)^2 \\ b_y - c_y & (b_x - c_x)^2 + (b_y - c_y)^2 \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}, \quad d_y = c_y + \frac{\begin{vmatrix} a_x - c_x & (a_x - c_x)^2 + (a_y - c_y)^2 \\ b_x - c_x & (b_x - c_x)^2 + (b_y - c_y)^2 \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}.$$

The denominator of these fractions is precisely the expression computed by ORIENT2D. The computation of d is unstable if a, b, and c are nearly collinear; roundoff error in the denominator can dramatically change the result, or cause a division by zero. Disaster can be avoided, and the desired error bound enforced, by computing the denominator with a variant of ORIENT2D that accepts an approximation only if its relative error is roughly half of one percent. A similar adaptive routine could accurately compute the numerators.

It might be fruitful to explore whether the methods described by Clarkson [23] and Avnaim et al. [2] can be extended by fast multiprecision methods to handle arbitrary double precision floating-point inputs. One could certainly relax their constraints on the bit complexity of the inputs; for instance, the method of Avnaim et al. could be made to perform the INSPHERE test on 64-bit inputs using expansions of length three. Unfortunately, it is not obvious how to adapt these integer-based techniques to inputs with wildly differing exponents. It is also not clear whether such hybrid algorithms would be faster than straightforward adaptivity. Nevertheless, Clarkson's approach looks promising for larger determinants. Although my methods work well for small determinants, they are unlikely to work well for sizes much larger than 5×5. Even if one uses Gaussian elimination rather than cofactor expansion (an important adjustment for matrices larger than 5×5), the adaptivity technique does not scale well with determinants, because of the large number of terms in the expanded polynomial. Clarkson's technique may be the only economical approach for matrices larger than 10×10.

Whether or not these issues are resolved in the near future, researchers can make use today of tests for orientation and incircle in two and three dimensions that are correct, fast in most cases, and applicable to single or double precision floating-point inputs. I invite working computational geometers to try my code in their implementations, and hope that it will save them from worrying about robustness so they may concentrate on geometry.

Appendix A

Linear-Time Expansion Addition without Round-to-Even Tiebreaking

Theorem 59 *Let $e = \sum_{i=1}^{m} e_i$ and $f = \sum_{i=1}^{n} f_i$ be nonoverlapping expansions of m and n p-bit components, respectively, where $p \geq 3$. Suppose that the components of both e and f are sorted in order of increasing magnitude, except that any of the e_i or f_i may be zero. Then the following algorithm will produce a nonoverlapping expansion h such that $h = \sum_{i=1}^{m+n} h_i = e + f$, where the components of h are also in order of increasing magnitude, except that any of the h_i may be zero.*

```
LINEAR-EXPANSION-SUM(e, f)
1    Merge e and f into a single sequence g, in order of
         nondecreasing magnitude (possibly with interspersed zeroes)
2    (Q_2, q_2) ⇐ FAST-TWO-SUM(g_2, g_1)
3    for i ⇐ 3 to m + n
4        (R_i, h_{i-2}) ⇐ FAST-TWO-SUM(g_i, q_{i-1})
5        (Q_i, q_i) ⇐ TWO-SUM(Q_{i-1}, R_i)
6    h_{m+n-1} ⇐ q_{m+n}
7    h_{m+n} ⇐ Q_{m+n}
8    return h
```

$Q_i + q_i$ is an approximate sum of the first i components of g; see Figure A.1.

Proof: At the end of each iteration of the **for** loop, the invariant $Q_i + q_i + \sum_{j=1}^{i-2} h_j = \sum_{j=1}^{i} g_j$ holds. Certainly this invariant holds for $i = 2$ after Line 2 is executed. From Lines 4 and 5, we have that $Q_i + q_i + h_{i-2} = Q_{i-1} + q_{i-1} + g_i$; the invariant follows by induction. (The use of FAST-TWO-SUM in Line 4 will be justified shortly.) This assures us that after Lines 6 and 7 are executed, $\sum_{j=1}^{m+n} h_j = \sum_{j=1}^{m+n} g_j$, so the algorithm produces a correct sum.

The proof that h is nonoverlapping and increasing relies on the fact that the components of g are summed in order from smallest to largest, so the running total $Q_i + q_i$ never grows much larger than the next component to be summed. Specifically, I prove by induction that the exponent of Q_i is at most one greater than the exponent of g_{i+1}, and the components $h_1, \ldots, h_{i-1}$ are nonoverlapping and in order of increasing magnitude (excepting zeros). This statement holds for $i = 2$ because $|Q_2| = |g_1 \oplus g_2| \leq 2|g_2| \leq 2|g_3|$. To prove the statement in the general case, assume (for the inductive hypothesis) that the exponent of Q_{i-1} is at most one greater than the exponent of g_i, and the components $h_1, \ldots, h_{i-2}$ are nonoverlapping and increasing.

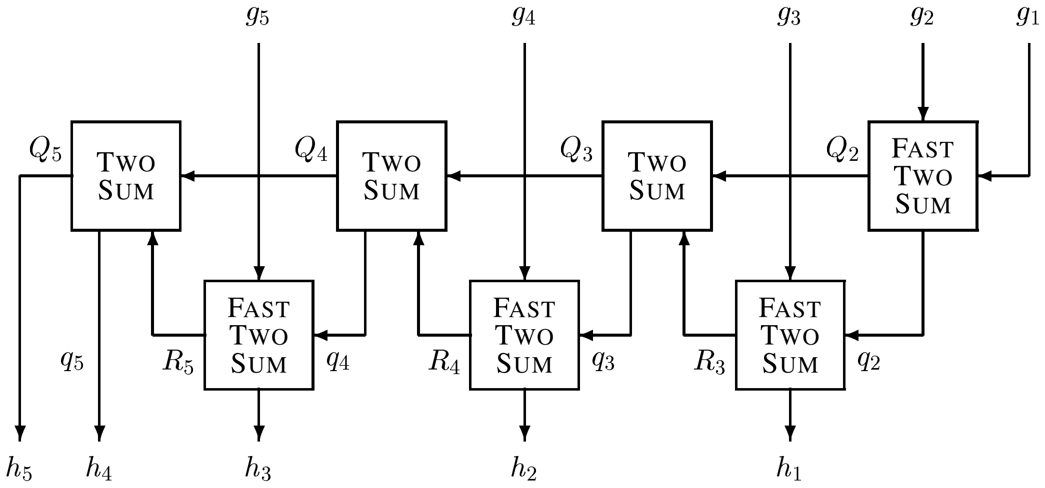


Figure A.1: Operation of LINEAR-EXPANSION-SUM. $Q_i + q_i$ maintains an approximate running total. The FAST-TWO-SUM operations in the bottom row exist to clip a high-order bit off each q_i term, if necessary, before outputting it.

q_{i-1} is the roundoff error of the TWO-SUM operation that produces Q_{i-1}, so $|q_{i-1}| \leq \frac{1}{2}\text{ulp}(Q_{i-1})$. This inequality and the inductive hypothesis imply that $|q_{i-1}| \leq \text{ulp}(g_i)$, which justifies the use of a FAST-TWO-SUM operation in Line 4. This operation produces the sum $|R_i + h_{i-2}| = |g_i + q_{i-1}| < (2^p + 1)\text{ulp}(g_i)$. Corollary 43(a) implies that $|h_{i-2}| < \text{ulp}(g_i)$. Because $h_1, \ldots, h_{i-2}$ are nonoverlapping, we have the bound $|\sum_{j=1}^{i-2} h_j| < \text{ulp}(g_i) \leq \text{ulp}(g_{i+1})$.

Assume without loss of generality that the exponent of g_{i+1} is $p - 1$, so that $\text{ulp}(g_{i+1}) = 1$, and $|g_1|, |g_2|, \ldots, |g_{i+1}|$ are bounded below 2^p. Because g is formed by merging two nonoverlapping increasing expansions, $|\sum_{j=1}^{i} g_j| < 2^p + 2^{p-1}$. Consider, for instance, if $g_{i+1} = 1000$ (in four-bit arithmetic); then $|\sum_{j=1}^{i} g_j|$ can be no greater than the sum of $1111.1111\ldots$ and $111.1111\ldots$.

Substituting these bounds into the invariant given at the beginning of this proof, we have $|Q_i + q_i| \leq |\sum_{j=1}^{i-2} h_j| + |\sum_{j=1}^{i} g_j| < 2^p + 2^{p-1} + 1$, which confirms that the exponent of Q_i is at most one greater than the exponent of g_{i+1}.

To show that h_{i-1} is larger than previous components of h (or is zero) and does not overlap them, observe from Figure A.1 that h_{i-1} is formed (for $i \geq 3$) by summing g_{i+1}, R_i, and Q_{i-1}. It can be shown that all three of these are either equal to zero or too large to overlap h_{i-2}, and hence so is h_{i-1}. We have already seen that $|h_{i-2}| < \text{ulp}(g_i)$, which is bounded in turn by $\text{ulp}(g_{i+1})$. It is clear that $|h_{i-2}|$ is too small to overlap R_i because both are produced by a FAST-TWO-SUM operation. Finally, $|h_{i-2}|$ is too small to overlap Q_{i-1} because $|h_{i-2}| \leq |q_{i-1}|$ (applying Lemma 36 to Line 4), and $|q_{i-1}| \leq \frac{1}{2}\text{ulp}(Q_{i-1})$.

The foregoing discussion assumes that none of the input components is zero. If any of the g_i is zero, the corresponding output component h_{i-2} is also zero, and the accumulator values Q and q are unchanged ($Q_i = Q_{i-1}$, $q_i = q_{i-1}$). ■

Appendix B

Why the Tiebreaking Rule is Important

Theorem 48 is complicated by the need to consider the tiebreaking rule. This appendix gives an example that proves that this complication is necessary to ensure that FAST-EXPANSION-SUM will produce nonoverlapping output. If one's processor does not use round-to-even tiebreaking, one might use instead an algorithm that is independent of the tiebreaking rule, such as the slower LINEAR-EXPANSION-SUM in Appendix A.

Section 6.3.4 gave examples that demonstrate that FAST-EXPANSION-SUM does not preserve the nonoverlapping or nonadjacent properties. The following example demonstrates that, in the absence of any assumption about the tiebreaking rule, FAST-EXPANSION-SUM does not preserve any property that implies the nonoverlapping property. (As we have seen, the round-to-even rule ensures that FAST-EXPANSION-SUM preserves the strongly nonoverlapping property.)

For simplicity, assume that four-bit arithmetic is used. Suppose the round-toward-zero rule is initially in effect. The incompressible expansions $2^{14} + 2^8 + 2^4 + 1$ and $2^{11} + 2^6 + 2^2$ can each be formed by summing their components with any expansion addition algorithm. Summing these two expansions, FAST-EXPANSION-SUM (with zero elimination) yields the expansion $1001 \times 2^{11} + 2^8 + 2^6 + 2^4 + 2^2 + 1$. Similarly, one can form the expansion $1001 \times 2^{10} + 2^7 + 2^5 + 2^3 + 2^1$. Summing these two in turn yields $1101 \times 2^{11} + 2^{10} + 1111 \times 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 1$, which is nonoverlapping but not strongly nonoverlapping.

Switching to the round-to-even rule, suppose FAST-EXPANSION-SUM is used to sum two copies of this expansion. The resulting "expansion" is $111 \times 2^{13} + -2^{11} + 2^{10} + -2^5 + 2^5 + -2^1$, which contains a pair of overlapping components. Hence, it is not safe to mix the round-toward-zero and round-to-even rules, and it is not possible to prove that FAST-EXPANSION-SUM produces nonoverlapping expansions for any tiebreaking rule.

Although the expansion above is not nonoverlapping, it is not particularly bad, in the sense that APPROXIMATE will nonetheless produce an accurate approximation of the expansion's value. It can be proven that, regardless of tiebreaking rule, FAST-EXPANSION-SUM preserves what I call the *weakly nonoverlapping* property, which allows only a small amount of overlap between components, easily fixed by compression. (Details are omitted here, but I am quite certain of the result. I produced a proof similar to that of Theorem 48, and rivalling it in complexity, before I discovered the strongly nonoverlapping property.) I conjecture that the geometric predicates of Section 6.5 work correctly regardless of tiebreaking rule.

Bibliography

[1] Franz Aurenhammer. *Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure*. ACM Computing Surveys **23**(3):345–405, September 1991.

[2] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic*. Algorithmica **17**(2):111–132, February 1997.

[3] Ivo Babuška and A. K. Aziz. *On the Angle Condition in the Finite Element Method*. SIAM Journal on Numerical Analysis **13**(2):214–226, April 1976.

[4] David H. Bailey. *A Portable High Performance Multiprecision Package*. Technical Report RNR-90-022, NASA Ames Research Center, Moffett Field, California, May 1993.

[5] B. S. Baker, E. Grosse, and C. S. Rafferty. *Nonobtuse Triangulation of Polygons*. Discrete and Computational Geometry **3**(2):147–168, 1988.

[6] T. J. Baker. *Automatic Mesh Generation for Complex Three-Dimensional Regions using a Constrained Delaunay Triangulation*. Engineering with Computers **5**:161–175, 1989.

[7] Hesheng Bao, Jacobo Bielak, Omar Ghattas, David R. O'Hallaron, Loukas F. Kallivokas, Jonathan Richard Shewchuk, and Jifeng Xu. *Earthquake Ground Motion Modeling on Parallel Computers*. Supercomputing '96 (Pittsburgh, Pennsylvania), November 1996.

[8] C. Bradford Barber. *Computational Geometry with Imprecise Data and Arithmetic*. Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, October 1992. Available as Technical Report CS-TR-377-92.

[9] T. J. Barth and D. C. Jesperson. *The Design and Application of Upwind Schemes on Unstructured Meshes*. Proceedings of the AIAA 27th Aerospace Sciences Meeting (Reno, Nevada), 1989.

[10] Marshall Bern and David Eppstein. *Mesh Generation and Optimal Triangulation*. Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 23–90. World Scientific, Singapore, 1992.

[11] Marshall Bern, David Eppstein, and John R. Gilbert. *Provably Good Mesh Generation*. 31st Annual Symposium on Foundations of Computer Science, pages 231–241. IEEE Computer Society Press, 1990.

[12] Adrian Bowyer. *Computing Dirichlet Tessellations*. Computer Journal **24**(2):162–166, 1981.

[13] Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. *Exact Geometric Computation in LEDA*. Eleventh Annual Symposium on Computational Geometry (Vancouver, British Columbia, Canada), pages C18–C19. Association for Computing Machinery, June 1995.

[14] Scott A. Canann, S. N. Muthukrishnan, and R. K. Phillips. *Topological Refinement Procedures for Triangular Finite Element Meshes*. Engineering with Computers **12**(3 & 4):243–255, 1996.

[15] John Canny. *Some Algebraic and Geometric Computations in PSPACE*. 20th Annual Symposium on the Theory of Computing (Chicago, Illinois), pages 460–467. Association for Computing Machinery, May 1988.

[16] Graham F. Carey and John Tinsley Oden. *Finite Elements: Computational Aspects*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[17] James C. Cavendish, David A. Field, and William H. Frey. *An Approach to Automatic Three-Dimensional Finite Element Mesh Generation*. International Journal for Numerical Methods in Engineering **21**(2):329–347, February 1985.

[18] L. Paul Chew. *Constrained Delaunay Triangulations*. Algorithmica **4**(1):97–108, 1989.

[19] ———. *Guaranteed-Quality Triangular Meshes*. Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[20] ———. *Building Voronoi Diagrams for Convex Polygons in Linear Expected Time*. Technical Report PCS-TR90-147, Department of Mathematics and Computer Science, Dartmouth College, 1990.

[21] ———. *Guaranteed-Quality Mesh Generation for Curved Surfaces*. Proceedings of the Ninth Annual Symposium on Computational Geometry (San Diego, California), pages 274–280. Association for Computing Machinery, May 1993.

[22] ———. *Guaranteed-Quality Delaunay Meshing in 3D*. Proceedings of the Thirteenth Annual Symposium on Computational Geometry. Association for Computing Machinery, 1997.

[23] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation*. 33rd Annual Symposium on Foundations of Computer Science (Pittsburgh, Pennsylvania), pages 387–395. IEEE Computer Society Press, October 1992.

[24] Kenneth L. Clarkson and Peter W. Shor. *Applications of Random Sampling in Computational Geometry, II*. Discrete & Computational Geometry **4**(1):387–421, 1989.

[25] E. F. D'Azevedo and R. B. Simpson. *On Optimal Interpolation Triangle Incidences*. SIAM Journal on Scientific and Statistical Computing **10**:1063–1075, 1989.

[26] T. J. Dekker. *A Floating-Point Technique for Extending the Available Precision*. Numerische Mathematik **18**:224–242, 1971.

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

[28] Tamal K. Dey, Chanderjit L. Bajaj, and Kokichi Sugihara. *On Good Triangulations in Three Dimensions*. Proceedings of the Symposium on Solid Modeling Foundations and CAD/CAM Applications. Association for Computing Machinery, 1991.

[29] David P. Dobkin and Michael J. Laszlo. *Primitives for the Manipulation of Three-Dimensional Subdivisions*. Algorithmica **4**:3–32, 1989.

[30] Rex A. Dwyer. *A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations*. Algorithmica **2**(2):137–151, 1987.

[31] Steven Fortune. *A Sweepline Algorithm for Voronoi Diagrams*. Algorithmica **2**(2):153–174, 1987.

[32] ——. *Stable Maintenance of Point Set Triangulations in Two Dimensions*. 30th Annual Symposium on Foundations of Computer Science, pages 494–499. IEEE Computer Society Press, 1989.

[33] ——. *Voronoi Diagrams and Delaunay Triangulations*. Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 193–233. World Scientific, Singapore, 1992.

[34] ——. *Progress in Computational Geometry*. Directions in Geometric Computing (R. Martin, editor), chapter 3, pages 81–128. Information Geometers Ltd., 1993.

[35] ——. *Numerical Stability of Algorithms for 2D Delaunay Triangulations*. International Journal of Computational Geometry & Applications **5**(1–2):193–213, March–June 1995.

[36] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry*. Proceedings of the Ninth Annual Symposium on Computational Geometry (San Diego, California), pages 163–172. Association for Computing Machinery, May 1993.

[37] ——. *Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry*. ACM Transactions on Graphics **15**(3):223–248, July 1996.

[38] Lori A. Freitag, Mark Jones, and Paul Plassman. *An Efficient Parallel Algorithm for Mesh Smoothing*. Fourth International Meshing Roundtable (Albuquerque, New Mexico), pages 47–58. Sandia National Laboratories, October 1995.

[39] Lori A. Freitag and Carl Ollivier-Gooch. *A Comparison of Tetrahedral Mesh Improvement Techniques*. Fifth International Meshing Roundtable (Pittsburgh, Pennsylvania), pages 87–100. Sandia National Laboratories, October 1996.

[40] ——. *Tetrahedral Mesh Improvement Using Swapping and Smoothing*. To appear in International Journal for Numerical Methods in Engineering, 1997.

[41] William H. Frey. *Selective Refinement: A New Strategy for Automatic Node Placement in Graded Triangular Meshes*. International Journal for Numerical Methods in Engineering **24**(11):2183–2200, November 1987.

[42] William H. Frey and David A. Field. *Mesh Relaxation: A New Technique for Improving Triangulations*. International Journal for Numerical Methods in Engineering **31**:1121–1133, 1991.

[43] P. L. George, F. Hecht, and E. Saltel. *Automatic Mesh Generator with Specified Boundary*. Computer Methods in Applied Mechanics and Engineering **92**(3):269–288, November 1991.

[44] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys **23**(1):5–48, March 1991.

[45] N. A. Golias and T. D. Tsiboukis. *An Approach to Refining Three-Dimensional Tetrahedral Meshes Based on Delaunay Transformations*. International Journal for Numerical Methods in Engineering **37**:793–812, 1994.

[46] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. *Randomized Incremental Construction of Delaunay and Voronoi Diagrams*. Algorithmica **7**(4):381–413, 1992. Also available as Stanford University Computer Science Department Technical Report STAN-CS-90-1300 and in Springer-Verlag Lecture Notes in Computer Science, volume 443.

[47] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams*. ACM Transactions on Graphics **4**(2):74–123, April 1985.

[48] Carol Hazelwood. *Approximating Constrained Tetrahedrizations*. Computer Aided Geometric Design **10**:67–87, 1993.

[49] L. R. Hermann. *Laplacian-Isoparametric Grid Generation Scheme*. Journal of the Engineering Mechanics Division of the American Society of Civil Engineers **102**:749–756, October 1976.

[50] Christoph M. Hoffmann. *The Problems of Accuracy and Robustness in Geometric Computation*. Computer **22**(3):31–41, March 1989.

[51] A. Jameson, T. J. Baker, and N. P. Weatherill. *Calculation of Inviscid Transonic Flow over a Complete Aircraft*. Proceedings of the 24th AIAA Aerospace Sciences Meeting (Reno, Nevada), 1986.

[52] Barry Joe. *Three-Dimensional Triangulations from Local Transformations*. SIAM Journal on Scientific and Statistical Computing **10**:718–741, 1989.

[53] ______. *Construction of Three-Dimensional Triangulations using Local Transformations*. Computer Aided Geometric Design **8**:123–142, 1991.

[54] ______. *Construction of k-Dimensional Delaunay Triangulations using Local Transformations*. SIAM Journal on Scientific Computing **14**(6):1415–1436, November 1993.

[55] ______. *Construction of Three-Dimensional Improved-Quality Triangulations Using Local Transformations*. SIAM Journal on Scientific Computing **16**(6):1292–1307, November 1995.

[56] Michael Karasick, Derek Lieber, and Lee R. Nackman. *Efficient Delaunay Triangulation Using Rational Arithmetic*. ACM Transactions on Graphics **10**(1):71–91, January 1991.

[57] Donald Ervin Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, second edition, volume 2. Addison Wesley, Reading, Massachusetts, 1981.

[58] Anton Szandor LaVey. *The Satanic Bible*. Avon Books, New York, 1969.

[59] Charles L. Lawson. *Software for C^1 Surface Interpolation*. Mathematical Software III (John R. Rice, editor), pages 161–194. Academic Press, New York, 1977.

[60] Der-Tsai Lee and Bruce J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation*. International Journal of Computer and Information Sciences **9**(3):219–242, 1980.

[61] Seppo Linnainmaa. *Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums*. BIT **14**:167–202, 1974.

[62] Rainald Löhner. *Generation of Three-Dimensional Unstructured Grids by the Advancing Front Method*. Proceedings of the 26th AIAA Aerospace Sciences Meeting (Reno, Nevada), 1988.

[63] David L. Marcum and Nigel P. Weatherill. *Unstructured Grid Generation Using Iterative Point Insertion and Local Reconnection*. Twelfth AIAA Applied Aerodynamics Conference (Colorado Springs, Colorado), number AIAA 94-1926, June 1994.

[64] Dimitri J. Mavriplis. *An Advancing Front Delaunay Triangulation Algorithm Designed for Robustness*. Technical Report 92-49, ICASE, October 1992.

[65] Victor Milenkovic. *Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections using Rounded Arithmetic*. 30th Annual Symposium on Foundations of Computer Science, pages 500–505. IEEE Computer Society Press, 1989.

[66] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. *A Delaunay Based Numerical Method for Three Dimensions: Generation, Formulation, and Partition*. Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing (Las Vegas, Nevada), May 1995.

[67] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, Noel Walkington, and Han Wang. *Control Volume Meshes using Sphere Packing: Generation, Refinement and Coarsening*. Fifth International Meshing Roundtable (Pittsburgh, Pennsylvania), pages 47–61, October 1996.

[68] Scott A. Mitchell. *Cardinality Bounds for Triangulations with Bounded Minimum Angle*. Proceedings of the Sixth Canadian Conference on Computational Geometry (Saskatoon, Saskatchewan, Canada), pages 326–331, August 1994.

[69] Scott A. Mitchell and Stephen A. Vavasis. *Quality Mesh Generation in Three Dimensions*. Proceedings of the Eighth Annual Symposium on Computational Geometry, pages 212–221, 1992.

[70] ———. *Quality Mesh Generation in Higher Dimensions*. Submitted to SIAM Journal on Computing. Manuscript available from http://www.cs.cornell.edu/Info/People/vavasis/vavasis.html, February 1997.

[71] N. E. Mnev. *The Universality Theorems on the Classification Problem of Configuration Varieties and Convex Polytopes Varieties*. Topology and Geometry - Rohlin Seminar (O. Ya. Viro, editor), Lecture Notes in Mathematics, volume 1346, pages 527–543. Springer-Verlag, 1988.

[72] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. *Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations*. Proceedings of the Twelfth Annual Symposium on Computational Geometry, pages 274–283. Association for Computing Machinery, May 1996.

[73] Friedhelm Neugebauer and Ralf Diekmann. *Improved Mesh Generation: Not Simple but Good*. Fifth International Meshing Roundtable (Pittsburgh, Pennsylvania), pages 257–270. Sandia National Laboratories, October 1996.

[74] Thomas Ottmann, Gerald Thiemt, and Christian Ullrich. *Numerical Stability of Geometric Algorithms*. Proceedings of the Third Annual Symposium on Computational Geometry, pages 119–125. Association for Computing Machinery, June 1987.

[75] V. N. Parthasarathy and Srinivas Kodiyalam. *A Constrained Optimization Approach to Finite Element Mesh Smoothing*. Finite Elements in Analysis and Design **9**:309–320, 1991.

[76] Douglas M. Priest. *Algorithms for Arbitrary Precision Floating Point Arithmetic*. Tenth Symposium on Computer Arithmetic (Los Alamitos, California), pages 132–143. IEEE Computer Society Press, 1991.

[77] ______. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. thesis, Department of Mathematics, University of California at Berkeley, Berkeley, California, November 1992. Available by anonymous FTP to `ftp.icsi.berkeley.edu` as `pub/theory/priest-thesis.ps.Z`.

[78] V. T. Rajan. *Optimality of the Delaunay Triangulation in* $\mathbb{R}^d$. Proceedings of the Seventh Annual Symposium on Computational Geometry, pages 357–363, 1991.

[79] James Martin Ruppert. *Results on Triangulation and High Quality Mesh Generation*. Ph.D. thesis, University of California at Berkeley, Berkeley, California, 1992.

[80] Jim Ruppert. *A New and Simple Algorithm for Quality 2-Dimensional Mesh Generation*. Technical Report UCB/CSD 92/694, University of California at Berkeley, Berkeley, California, 1992.

[81] ______. *A New and Simple Algorithm for Quality 2-Dimensional Mesh Generation*. Proceedings of the Fourth Annual Symposium on Discrete Algorithms, pages 83–92. Association for Computing Machinery, January 1993.

[82] ______. *A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation*. Journal of Algorithms **18**(3):548–585, May 1995.

[83] Jim Ruppert and Raimund Seidel. *On the Difficulty of Tetrahedralizing 3-Dimensional Non-Convex Polyhedra*. Proceedings of the Fifth Annual Symposium on Computational Geometry, pages 380–393. Association for Computing Machinery, 1989.

[84] H. Samet. *The Quadtree and Related Hierarchical Data Structures*. Computing Surveys **16**:188–260, 1984.

[85] Raimund Seidel. *Backwards Analysis of Randomized Geometric Algorithms*. Technical Report TR-92-014, International Computer Science Institute, University of California at Berkeley, Berkeley, California, February 1992.

[86] Michael I. Shamos and Dan Hoey. *Closest-Point Problems*. 16th Annual Symposium on Foundations of Computer Science (Berkeley, California), pages 151–162. IEEE Computer Society Press, October 1975.

[87] Mark S. Shephard and Marcel K. Georges. *Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique*. International Journal for Numerical Methods in Engineering **32**:709–749, 1991.

[88] Daniel Dominic Sleator and Robert Endre Tarjan. *Self-Adjusting Binary Search Trees*. Journal of the Association for Computing Machinery **32**(3):652–686, July 1985.

[89] Pat H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[90] Peter Su. *Efficient Parallel Algorithms for Closest Point Problems*. Ph.D. thesis, Dartmouth College, Hanover, New Hampshire, May 1994.

[91] Peter Su and Robert L. Scot Drysdale. *A Comparison of Sequential Delaunay Triangulation Algorithms*. Proceedings of the Eleventh Annual Symposium on Computational Geometry (Vancouver, British Columbia, Canada), pages 61–70. Association for Computing Machinery, June 1995.

[92] Joe F. Thompson and Nigel P. Weatherill. *Aspects of Numerical Grid Generation: Current Science and Art*. 1993.

[93] David F. Watson. *Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes*. Computer Journal **24**(2):167–172, 1981.

[94] Nigel P. Weatherill. *Delaunay Triangulation in Computational Fluid Dynamics*. Computers and Mathematics with Applications **24**(5/6):129–150, September 1992.

[95] Nigel P. Weatherill and O. Hassan. *Efficient Three-Dimensional Grid Generation using the Delaunay Triangulation*. Proceedings of the First European Computational Fluid Dynamics Conference (Brussels, Belgium) (Ch. Hirsch, J. Périaux, and W. Kordulla, editors), pages 961–968, September 1992.

[96] Nigel P. Weatherill, O. Hassan, D. L. Marcum, and M. J. Marchant. *Grid Generation by the Delaunay Triangulation*. Von Karman Institute for Fluid Dynamics 1993–1994 Lecture Series, 1994.

[97] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[98] M. A. Yerry and Mark S. Shephard. *A Modified Quadtree Approach to Finite Element Mesh Generation*. IEEE Computer Graphics and Applications **3**:39–46, January/February 1983.

[99] ———. *Automatic Three-Dimensional Mesh Generation by the Modified-Octree Technique*. International Journal for Numerical Methods in Engineering **20**:1965–1990, 1984.