Computations of Delaunay and higher order triangulations, with applications to splines

by
Yuanxin Liu

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2007

Approved by:

Jack Snoeyink, Advisor

Leonard McMillan, Reader

Wenjie He, Reader

Dinesh Manocha, Committee Member

Helena Mitasova, Committee Member

© 2007
Yuanxin Liu
ALL RIGHTS RESERVED

ABSTRACT

YUANXIN LIU: Computations of Delaunay and higher order triangulations, with applications to splines.

(Under the direction of Jack Snoeyink.)

Abstract goes here!

Contents

List of Figures **xi**

List of Tables **xvii**

1 Introduction **1**

2 Geometric Preliminaries **7**
2.1 Geometric primitives 7
2.2 Convex cell complexes and triangulations 8
2.3 Delaunay diagrams and triangulations 9

3 Faraway Point: A Sentinel Point for Delaunay tessellation **12**
3.1 Geometric Preliminaries 15
3.2 The lifting map 16
3.3 The Polytope with a Faraway Point 18
3.4 Implementation policy for sidedness test 21
3.5 Discussion 22

4 Engineering Delaunay Tessellation Programs **25**
4.1 Incremental construction of Delaunay triangulation 25
4.2 Exact predicate computation 27
4.3 Handling of degeneracies 28
4.4 TESS3: A Delaunay triangulation program for protein molecular data . 29
  4.4.1 Bit-leveling and Ordering with Space filling curves 30
  4.4.2 Selected implementation details 32
  4.4.3 Accuracy limitations 33
4.5 Comparison of five Delaunay triangulation programs 35
  4.5.1 Implementation Comparison 36

4.5.2 Performance comparison 39
4.5.3 Conclusion 43
4.6 Computation of Delaunay Diagrams on Points from 4d Grids 44
4.6.1 Preliminaries 45
4.6.2 Arithmetic complexity and algebraic degree 47
4.6.3 Sphere construction 47
4.6.4 Incremental construction of 4d Delaunay diagrams 48
4.6.5 Implementation and experiments 50
4.6.6 Conclusion 52
4.6.7 Factoring the expression for spheres 52

5 Streaming Delaunay Triangulations 56
5.1 Previous works 58
5.2 Spatial finalizing data 59
5.3 Streaming 2D Delaunay triangulations 61
5.3.1 Delaunay triangulation with finalization 61
5.3.2 Delaunay triangulation with finalization 62
5.3.3 Identifying final triangles 63
5.4 Streaming 3D Delaunay triangulation 65
5.5 Comparisons 66
5.6 Conclusions 68
5.7 Error bounds for computing sphere centers and radii 70
5.7.1 center error bound 71
5.7.2 squared radius error bound 72

6 Multivariate B-splines 74
6.1 Univariate B-splines 76
6.2 Multivariate B-splines and Delaunay configurations 77
6.2.1 simplex splines 78
6.2.2 Delaunay configurations 79

7 Generalization of Delaunay Configurations 84
7.1 k-sets and boundary-interior configurations 86
7.2 Boundary-interior configurations for convex sets in three dimensions . . 89
7.3 Projective boundary-interior configurations: A generalization of Delaunay configurations 94

7.4 Generalized configs in two dimensions by computational procedures . . 95
    7.4.1 Preliminaries 100
    7.4.2 $\leq$ 3-set links bound null or simple polygons 103
    7.4.3 Generalization of order two and three centroid triangulations . . 106

8 Simplex Splines from Generalized Delaunay Configurations 108
8.1 Collecting simplex splines to B-splines 110
8.2 Reproduction of ZP elements 112
    8.2.1 Polyhedron splines 112
    8.2.2 ZP elements and B-splines 113
    8.2.3 Tessellation of 4-cube 114
    8.2.4 Main result 116
    8.2.5 Application: patch-blending 118
8.3 Reproduction of Bezier patches 120
    8.3.1 Application: representing sharp features 123
8.4 Data fitting with bivariate B-splines 125

Bibliography 129

List of Figures

1.1 a) Delaunay triangulation; the dotted circle circumscribes the vertices of a triangle and is empty of other vertices. b) A three dimensional triangulation in 3-space. c) A two dimensional triangulation on a sphere. 2

1.2 Left: The PL interpolation function over a triangulation. Right: The PL interpolation of terrain data from Crater Lake (Garland and Heckbert, 1995). 3

3.1 For input P, points of 4×4 grid, we draw the 2d projections of the convex hulls of lifted points: at left, $\mathrm{conv}(\hat{P} \cup \{e\}$, where e is the point at infinity, and at right, $\mathrm{conv}(\hat{P} \cup \{q\})$, where q is the faraway point. Dashed segments connect to the point at infinity or the faraway point. 14

3.2 The two dimensional oriented projective space $\mathbb{P}^2$ can be identified with a sphere. Through central projection, the upper open hemisphere maps to the Euclidean plane H, and the lower one maps to H'—the antipodes of the Euclidean plane. The shaded region on the sphere is a polytope in which one vertex is antipodal to an Euclidean point—drawn as a hollow dot. The projection of the polytope onto the two Euclidean planes are two open polygons. 16

3.3 Computing Delaunay diagram using the lifting map and the faraway point q. The shaded planar subdivision is the Delaunay diagram. The open polyhedron is the Euclidean portion of an oriented projective polytope with vertex q. Since we cannot draw q in Euclidean space, we draw its antipode $-q$ as an unfilled circle. 18

4.1 Perturbing b to b' produces the shaded flat triangle. 29

4.2 Hilbert curve for an $8 \times 8 \times 8$ grid. 31

4.3 Running times of incremental Delaunay tessellation for ten input sets of 100K randomly distributed points under four spatial orders: Hilbert, row-major, Gray, Z-order and random order 32

4.4 The atoms from the PDB file 1H1K. The tetrahedra drawn are those whose spheres are not empty. 33

4.5 Semilog plot showing percentage of InSphere tests with round-off errors by number of points n and number of coordinate bits, for three orderings. We plot a dot for each of 10 runs for given n and bit number, and draw lines through the averages of 10 runs. 35

4.6 Running time of the programs with 10 bit random points. 40

4.7 Running time of the programs with PDB files. 41

4.8 Running time of the CGAL Delaunay hierarchy using random, BRIO and Hilbert point orders. 43

4.9 Growth in the number of data structure elements against the number of points inserted in randomized incremental construction for the Delaunay diagram (gray) and Delaunay triangulation (black). The number of data structure elements is the sum of cells, faces and vertex references 45

5.1 Three terrain data sets: the 6 million-point “grbm” (LIDAR data of Baisman Run at Broadmoor, Maryland, left), the 67 million-point “puget” (middle) and the 0.5 billion-point “neuse” data set (right). Colors illustrate spatial coherence in selected grid cells: each cell’s center is colored by the time of its first point, and each cell’s boundary is colored by the time of its last point, with time increasing from black to white along the color ramp (bottom). . . 59

5.2 A closeup of streaming Delaunay in 2D. The points on the left have been processed, and their triangles written out. All triangles in this figure are *active*. We have drawn a few representative circumcircles, all of which intersect unfinalized space. At this moment, points are being inserted into the leftmost cell, which will be finalized next. 62

5.3 Skinny temporary triangles (left) are avoided by lazily sprinkling one point into each unfinalized quadrant at each level of the evolving quadtree (right). 63

5.4 Points sampled on a closed curve. Most of space has been finalized, yet few triangles are final—most circumcircles intersect the unfinalized region. . . . 65

5.5 Streaming Delaunay tetrahedralization of the f16 point set. Sprinkle points are turned off for clarity. Most of this model’s points are clustered near its center. 66

6.1 In counter-clockwise order: B-splines of degree 0, 1 and 2 and quadratic B-spline basis. 76

6.2 Simplex splines as shadows. The heavy dots are the defining knots. The length of the vertical segment, or the area of the vertical polygon, is the value of the function at the point x. 78

6.3 Illustration of Proposition 1. The dotted circle goes through two points and has a pair of points I inside. The solid circles are circumcircles of three points—indicated by triangles. The un-shaded triangles abut on points from I, while the colored triangles do not. 81

7.1 The edges and triangles in F_0 form a belt. 91

7.2 Illustration for Lemma 7.2.2. The balls above and below the hyperplane h represent the X and Y sets. The red and green dots represent the projections of X and Y sets. For each bounding edge of the convex hull (shaded), the triplet of points whose plane goes through it gives a triangle in $F(X, P)$. Left: The red and green dots can not be separated by a line; $v \notin V_1(X \cup Y \cup \{v\})$. Right: the red and green dots can be separated by line; $v \in V_1(X \cup Y \cup \{v\})$. 92

7.3 Illustration of the link triangulation algorithm. The first iteration takes a triangulation of the knots (top left), triangulates the link of each knot (for example, the second of the top row shows the triangulation for the link of knot 9), and outputs a set of degree 1 configs. The output degree 1 configs, along with the input degree 0 configs, are transformed by the union map to the centroid triangulation on the first of the bottom row, in which the triangles from degree 0 configs are colored green and the shaded part is the transformed link triangulation for knot 9. Each centroid is labeled by its corresponding set of knots; these labels are more legible if magnified. The second iteration is illustrated the same way: The third of the top row shows the link triangulation for a knot pair $\{9, 12\}$ and the picture below it shows the centroid triangulation formed by the output degree 2 configs and the input degree 1 configs. The third iteration is not completed. To complete it, take each knot set correponding to a vertex in the second centroid triangulation and triangulate its link, for example, the link of the knot set $\{4, 8, 9\}$ on the top right. Some of these links bound null polygons, for example, the link of $\{12, 15, 16\}$, colored orange. 99

7.4 Illustration for the proof of Lemma 7.4.4. 101

7.5 Illustration for the proof of Lemma 7.4.6. The gray edges are edges of the triangulation Δ_0. The red and green edges are diagonal edges of the triangulations Δ_u and Δ_v, respectively. 103

7.6 Illustration for the proof of Lemma 7.4.7. The gray edges are edges of the triangulation Δ_0. The dark edges are edges of the diagonals in Δ_1. Left: Case I. Center and right: Case II. 104

8.1 Left: The control polygon for a univariate B-spline. Right: The control mesh for a bivariate B-spline. 111

8.2 The four directional mesh over $\mathbb{Z} \times \mathbb{Z}$ and the ZP-element. 113

8.3 The triangulation of $\mathbb{Z} \times \mathbb{Z}$, Δ_0, the degree one configurations Δ_1. Left: Δ_0 and a link triangulation. Middle: centroid triangulation from Δ_1 and Δ_0, in which the shaded portion corresponds to the link triangulation on the left. Right: the P and Q type B-splines associated with Δ_1; the knot pairs that index them are colored black. 114

8.4 Illustration for the proof of Lemma 8.2.3. Left: the pyramid $[C_0]$ and tetrahedron $[D_0]$ in the tessellation of a cube; the symmetrical polytopes $[C_1]$ and $[D_1]$, which tessellate the half of the cube drawn in light gray, are not shown. Center: the bottom and top of the 4-prism $[C_0] \times [0,1]$ are drawn in two colors; the coordinates are those of P_0'. Right: the bottom and top of the 4-prism $[D_0] \times [0,1]$ are drawn in two colors; the coordinates are those of Q_0'. . . . 115

8.5 Illustration for the proof of Lemma 8.2.5. 116

8.6 A triangulation and examples of its associated vertex and edge tent functions. The functions on the right are indexed by the highlighted vertex and edge on the left. 121

8.7 Left: a pulled apart triangulation $\widetilde{\Delta_0}$ with a vertex and edge magnified; note that there are three possible number of duplicate edges between two pulled apart vertices. Right: the link triangulations of a pair of duplicate vertices; the triangulation edges of $\widetilde{\Delta_0}$ are drawn as thick gray edges while the edges of the link triangulations are drawn as think black edges. 122

8.8 Interpolating the bivariate functions $\sin(xy)e^{x+y}$, above, and $\sin(\pi x)$, below, with B-splines defined over the knot set $\{1..6\}\times\{1..6\}$. For each interpolation, shown from left to right on a row are: the knot triangulation, the centroid triangulation from the knot triangulation and the degree one configurations, the error measured over a grid, the maximum error and the condition number of the interpolation matrix. 128

List of Tables

3.1 The InSphere test requires little change with the faraway point q, whether perturbation is used or not. The code assumes that the cells are represented by point sets. When it discovers a cell using q, it obtains a neighbor cell c' from the data structure and tests q against the sphere for c'. 23

4.1 The percentage of Delaunay tetrahedra with neighboring points inside or on their spheres using 100K random points as input. 34

4.2 Program comparison summary. 39

4.3 Summary of timings and memory usage, running the programs against the same 100k randomly generated points with 10 bit coordinates. Notes: For pyramid tetrahedra creation, numbers marked f include all initialized by flipping and marked n include only those for which new memory is allocated—equivalently, only those not immediately destroyed by a flip involving the same new point. For CGAL timings, p indicates profiler and t direct timing. For tess3 point location, h includes the preprocessing to order the points along a Hilbert curve; w is walk only. 42

4.4 Comparing running time and memory of QHULL and DD4 with TESS4. The input for each row is a fraction of the 16^4 grid. Both the size of the process memory and the output are reported. The output includes each cell's vertex and neighbor references, stored as integers. For the output size of DD4, the number in the parenthesis is the fraction of cells that are not simplicial. . . 51

5.1 Performance of `spdelaunay2d` on large terrains. The `spfinalize` option "l$_i$" selects a quadtree of depth i, and "e$_j$" finalizes all empty quadrants in the bottom j levels of the tree at the beginning of the stream. Rows list `spdelaunay2d`'s memory footprint (MB) and two timings: one for reading pre-finalized points from *disk*, and one for reading finalized points via a *pipe* from `spfinalize`. Timings and memory footprints do *not* include `spfinalize`, except that the "pipe" timings include `spfinalize`'s third pass, which runs concurrently. For total "pipe" running times, add pass 1 & 2 timings from Table **??**. For total "pipe" memory footprints, add the footprint from Table **??**. For "disk" mode, add the running times of all three finalizer passes, and take the larger memory footprint. Disk timings for the "neuse" tiling are omitted—we do not have enough scratch disk space. 64

5.2 Performance of `spdelaunay3d` tetrahedralizing pre-finalized 3D points sampled from the ppm isosurface. The output is a streaming tetrahedral mesh. Option "l$_i$" indicates that the points are spatially finalized with an *octree* of depth i. The middle third of the table shows the maximum number of active tetrahedra, the running time (hours:minutes:seconds), and the memory footprint (MB). 65

5.3 Running times (minutes:seconds) and memory footprints (MB) of triangulators on an old laptop (top of each time box) with 512 MB memory and a new laptop (bottom of each time box) with 1 GB memory, for several 2D and 3D point sets. `spfinalize` pipes its output to `spdelaunay2d` or `spdelaunay3d`; timings for `spfinalize` reflect only the first two passes over the input stream, and timings for `spdelaunay2d` or `spdelaunay3d` reflect the combined times for the triangulator and the finalizer's third pass. The "total" column lists the start-to-end running time and memory footprint of the triangulation pipe. For the in-core triangulators Triangle and Pyramid, we report total running time and the running time excluding I/O ("–I/O"). Option "m$_5$" means subtrees with less than 5K points are collapsed into their root cell. 67

7.1 The link triangulation procedure 98

Chapter 1

Introduction

Digital data that consist of discrete points are frequently captured and processed by scientific and engineering applications—for example, digital images captured by cameras, atom coordinates of a protein molecule produced by x-ray crystallography, sample points of a surface in 3D gathered by laser scanning devices, and sample points in fluid dynamic simulation. Due to the rapid advance of new data gathering technologies, data set sizes are increasing, and the data distributions are becoming more irregular. These trends call for new computational tools that are both efficient enough to handle large data sets and flexible enough to accommodate irregularity.

A mathematical foundation that is well-suited for developing such tools is *triangulation*. For a set of points P in the plane, a two dimensional triangulation is a set of triplets in P whose corresponding triangles form a tiling in the plane; analogously, a three dimensional triangulation is a set of quadruples such that the corresponding tetrahedra form a tiling in the 3-space (Figure 1.1b). For a chosen point set, many possible triangulations exist. A particularly well known one is the *Delaunay triangulation*, which, in two dimensions, satisfies the property that the circumcircles through each triplet has no other point of P inside (See Figure 1.1a).

Triangulations are used for a wide variety of applications. For example, they are used in molecular biology for analyzing neighboring relations (Ban et al., 2004; Liang et al., 1998; Bandyopadhyay and Snoeyink, 2004), in geo-sciences and CAD/CAM to represent and analyze surfaces (Edelsbrunner et al., 2003) and in fluid dynamics to tessellate the domain of simulation (Bern and Eppstein, 1992; Shewchuk, 2002). These applications benefit from two properties of triangulations: First, a triangulation decomposes the space into simple regions of fixed complexity, which localizes the computation on data; second, it can be constructed around any set of points, which accommodates irregularity.

To appreciate more concretely the benefit of using triangulation, consider the application of terrain modeling in geo-sciences, where data are following the trend of becoming larger and more irregular: Gigabytes-size data sets are now routinely gathered by sensors such as LIDAR; irregular data are generated both by sensors and by integration of data from different sources. A terrain surface is modeled as a bivariate (two-variable) function, so that the domain of the function represents the latitude-longitude coordinate plane, the range represents the elevations and the plot of the function represents the terrain surface. For a particular piece of terrain, its model function is constructed from a set of 3D points sampled from its surface. One possible function that can be constructed is the *piecewise linear (PL) interpolation* function. To construct such a function, f, the locations of the data in the planar domain are triangulated. Then, for a point x in the domain, the function value $f(x)$ is the linear combination of three data points whose triangle contains x (See Figure 1.2). Several nice features of the PL interpolation function follow directly from the properties of triangulation:

- The function is defined *locally*: Evaluating the function references precisely three nearby points.

- The function can represent the data more compactly: Suppose that the terrain has a large piece of flat region—from a lake, for example—then, the PL-interpolation can represent the region using only the few data points around the the boundary of the region.

- The function can be generalized to handle data in non-planar domains, thanks to the well-known topological fact that a triangulation can be constructed over any surfaces in 3D (See figure 1.1c).

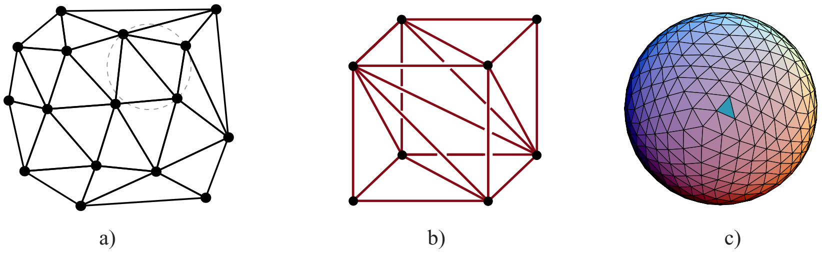

Figure 1.1: a) Delaunay triangulation; the dotted circle circumscribes the vertices of a triangle and is empty of other vertices. b) A three dimensional triangulation in 3-space. c) A two dimensional triangulation on a sphere.

The potential benefits from using triangulation are far from being fully exploited. The challenges fundamentally stem from the complexity of the triangulation structure, which generally takes more space to represent than the input points. This complexity makes developing a triangulation program a delicate task, particularly when it is important that the program runs fast and robustly over large data. Specifically,

- Many computer programs are designed with the assumption that the memory required by the program will not exceed the main memory size. When this assumption is violated, programs *thrash*, slowing down dramatically as the operating system starts to page memory pages out to disks. Thrashing is particularly pronounced for triangulation programs because the size of a triangulation is about 10 times as large as the input point set.
- In designing triangulation algorithms, it is common to assume that the input points are drawn from the ideal Euclidean plane, so that special point configurations, such as a triplet of points that are collinear, rarely occur. It is also common to assume that geometric tests, such as testing which side of the line a point is on, can always be performed exactly. Because neither of these assumptions are true in practice, before an algorithm can be translated to a working program, careful analysis must be done to show how to handle the situations when these assumptions are violated. If this is not done, the resulting program can produce inconsistent output or even crash.

Mathematically, the complexity of the triangulation structure makes it challenging to build additional structure "on top of" triangulation. Consequently, there only a few types of function spaces defined over triangulation—one example is the *Bezier patch*, which includes the PL interpolation as a special case—and none of them uses more than

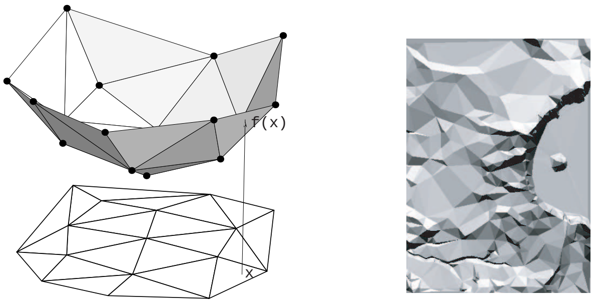

Figure 1.2: Left: The PL interpolation function over a triangulation. Right: The PL interpolation of terrain data from Crater Lake (Garland and Heckbert, 1995).

the basic tiling property of triangulation. The lack of variety of mathematical functions makes it difficult for triangulation to meet the modeling needs of many applications. For example, although terrain modeling could benefit from the ability of triangulation to handle large and irregular data, it often resorts to other tools that provide a better variety of functions, at the expense of handling irregularity or speed. In particular, if speed is more critical than handling irregularity, then it borrows from the large number of tools available for image processing, which require that the data to lie on a grid; if handling irregularity is more important than speed, then it uses various radial basis functions (Mitasova and Mitas, 1993; Wendland, 2004), which are slow for large data sets because they are globally defined.

This thesis addresses these challenges in two parts. The first part concentrates on techniques designed for efficiently and robustly computing Delaunay triangulations of three kinds of practical data: the terrain data from LIDAR sensors commonly found in GIS, the atom coordinate data used for biological applications, and the time varying volume data generated from from scientific simulations.

The second part addresses the problem of defining spline spaces over triangulations in two dimensions. It does so by generalizing Delaunay configurations, defined as follows. For a given point set P in two dimensions, a *Delaunay configuration* is a pair of subsets (T, I) from P, where T, called the *boundary set*, is a triplet and I, called the *interior set*, is the set of points that fall in the circumcircle through T. The size of the interior set is the *degree* of the configuration. As recently discovered by Neamtu (2004), for a chosen point set, the set of all degree k Delaunay configurations can be associated with a set of degree $k+1$ splines that form the basis of a spline space. In particular, for the trivial case of $k = 0$, the spline space coincides with the PL interpolation functions over the Delaunay triangulation. Neamtu's definition of the spline space relies only on a few structural properties of the Delaunay configurations. This raises the question whether there exist other sets of configurations with identical structural properties. If there are, then these sets of configurations—let us call them *generalized configurations* from hereon—can be substituted for Delaunay configurations in Neamtu's definition of spline space thereby yielding a family of splines over the same point set. This has the following applications:

- *Framework of splines.* When choosing a basis for spline representation, an important criteria is its generality. In the univariate setting, the popularity of B-splines can be partly attributed to its ability to represent a variety of splines. In the bivariate setting, the splines from the generalized configurations hold the promise to provide an analog of B-splines in providing a general spline representation, because the definition has no restrictions on the positions of the input points and, for a fixed point set, admit a large number of spline bases.

- *Data dependent configurations.* Delaunay triangulation is the canonical triangulation to use for PL-interpolation, because it has a number of optimal interpolation properties (Shewchuk, 2002). However, if certain characteristics of the data is available, such as the preferred gradient direction of the true function, non-Delaunay triangulations can achieve better PL-interpolation. This is referred to in the literature as *data dependent triangulation* (Dyn and Rippa, 1993). The generalization of Delaunay configurations could support "data-dependent configurations": Guided by knowledge about the data, non-Delaunay configurations can be constructed to improve the data fitting.

It can be observed that, for the trivial case of $k = 0$, the generalized configurations are simply sets of triangles that form planar triangulations. Therefore, this generalization problem can be more broadly considered as one of generalizing triangulations. The relation of Delaunay triangulations, configurations, planar triangulations, and the solution of the generalization problem is depicted by the diagram below:

$$\begin{array}{rcl} \text{Delaunay triangulations} & \subset & \text{Delaunay configurations} \\ \cap & & \cap \\ \text{triangulations} & \subset & \text{generalized configurations} \end{array}$$

The following summarizes the main results of the thesis, many of which have been published.

- [Chapter 3] I define a sentinel point for computing Delaunay triangulations that allows perturbation methods to enforce general position. The work will be published in International Journal of Computational Geometry and Applications (Liu and Snoeyink, 2006a).

- [Chapter 5] I show how to reduce the memory usage of a Delaunay triangulation program to a fraction by exploiting the fact that huge data sets tend to be in *spatially coherent* order—spatially near points are also near in their ordering. This work is done in collaboration with Martin Isenburg and published in SIGGRAPH'06 (Isenburg et al., 2006)

- [Section 4.4] I give techniques to efficiently compute 3D Delaunay triangulations of protein data and compare the performances of several 3D Delaunay triangulations programs which implement similar algorithms but make different decisions on how to meet the algorithm assumptions. The work is published in Combinatorial and Computational Geometry (Liu and Snoeyink, 2005a) and Voronoi diagram conference (Liu and Snoeyink, 2005b).

- [Section 4.6] I give techniques to efficiently compute 4D Delaunay triangulations of time varying volume data. The work is published in Voronoi diagram conference ’06 (Liu and Snoeyink, 2006c).

- [Chapter 7.4] I generalize two dimensional Delaunay configurations through a computational procedure. This procedure takes a planar triangulation as input and iteratively compute configurations of one degree higher. The procedure can be varied by varying a polygon-triangulation subroutine. This subroutine may be specialized to produce Delaunay configurations, but other subroutines can be designed to suit application needs. Preliminary results of the work are published in Symposium of Computational Geometry ’06 (Liu and Snoeyink, 2006b).

- [Chapter 8] I give examples of applications of quadratic bivariate splines from the generalized configurations. Parts of the work are published in Symposium of Computational Geometry ’06 (Liu and Snoeyink, 2006b).

Chapter 2

Geometric Preliminaries

I review foundamental geometrc objects in the s dimensional Euclidean space, which include oriented hyperplanes, affine and convex hulls, convex cell complexes, triangulations, and, finally, Delaunay diagrams and triangulations.

2.1 Geometric primitives

The *affine hull* of a set of points generalizes the notion of the line through a set of points: For a set of points $P \subset \mathbb{R}^s$, the affine hull of P, denoted $\mathrm{aff}(P)$, is defined:

$$\mathrm{aff}(P) : \{\sum_{p \in P} \lambda_p p \mid \sum_{p} \lambda_p = 1\}. \tag{2.1}$$

The *dimension* of an affine hull $\mathrm{aff}(P)$ one less than the size of the smallest subset of P that still gives the same hull. A set of n points P are *affinely independent* if the dimension of $\mathrm{aff}(P)$ is $n-1$.

In $\mathbb{R}^s$, an $s-1$-dimensional affine hull is called a *hyperplane.* A hyperplane h divides the space into two open half spaces. By labeling one side of h as positve and the other negative, we *orient* the hyperplane h. The positive and negatives side of h are denoted h^+ and h^-, respectively. The *closed* positive and negative side are denoted $\overline{h^+} \equiv h \cup h^+$ and $\overline{h^-} \equiv h \cup h^+$, respectively. An oriented hyperplane h can be represented by an $s+1$-tuple of reals: There exists a tuple of reals $(h_0, h_1, \ldots, h_s) \in \mathbb{R}^{s+1}$ such that for a point $p \in \mathbb{R}^s$, p belongs to h, or h^- or h^+ if and only if $(h_0, h_1, \ldots, h_s) \cdot (1, p_1, \ldots, p_s) = 0$, < 0 or > 0. Clearly, positive multiples of the tuple represents the same oriented hyperplane.

The tuple representing a hyperplane can be computed by first choosing s affinely independent points $(A_1, \ldots, A_s) \subset \mathbb{R}^s$ on the hyperplane, and compute a $(s+1)$-tuple of $s \times s$ determinants:

$$H(A_1, \ldots, A_s) := \left((-1)^i \begin{vmatrix} (A_0)_0 & \ldots & (A_0)_{i-1} & (A_0)_i & \ldots & (A_0)_s \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ (A_s)_0 & \ldots & (A_s)_{i-1} & (A_s)_i & \ldots & (A_s)_s \end{vmatrix} \right)_{0 \leq i \leq s} \tag{2.2}$$

The convex hull of a set of points generalize the notion of the line segment between two points: For a set of points $P \subset \mathbb{R}^s$, its convex hull $\mathrm{conv}(P)$ is defined:

$$\mathrm{conv}(P) := \{\sum_{p \in P} \lambda_p p \mid \sum_p \lambda_p = 1, \lambda_p \geq 0\}. \tag{2.3}$$

The *dimension* of a convex hull $\mathrm{conv}(P)$ is the dimension of the affine hull $\mathrm{aff}(P)$.

The convex hull of a finite set of points is called a *polytope*. The most interesting thing about a polytope is its boundary, which consists of *faces*: Given a polytope $\mathcal{P}$, a subset $\mathcal{F} \subset \mathcal{P}$ is a *face* of $\mathcal{P}$ if there is a hyperplane h such that $\mathcal{F} = h \cap \mathcal{P}$ and $h^- \cap \mathcal{P} = \emptyset$. The faces are themselves polytopes (of lower dimension). The lowest dimensional (0) faces are called *vertices*; the highest dimensional ($s-1$) faces are called *facets*. Given a polytope $\mathcal{P}$, we denote the set of all faces by $\mathcal{F}(\mathcal{P})$,

A polytope can be either represented by its vertex set or the set of hyperplanes that support its facets(one proof can be found in Ziegler (1994)):

Fact 2.1.1. *Let $\mathcal{P}$ be a polytope with vertices V and facet support hyperplanes H. Then,*

$$\mathcal{P} = \mathrm{conv}(V) = \bigcap_{h \in H} \overline{h^+}.$$

For problems that take points as input, it is natural to use the vertex sets to represent polytopes. When using vertex sets, it is convenient to speak of "polytope V", when V is a set of vertices, but this can be confusing when we need to choose a point from $\mathrm{conv}(V)$. To avoid such confusion—and still be brief, we use brackets over V, so that $[V] \equiv \mathrm{conv}(V)$.

2.2 Convex cell complexes and triangulations

A set of polytopes $\mathcal{C}$ form a *convex cell complex* if it satisfies that, for any polytope $\mathcal{P} \in \mathcal{C}$, all faces of $\mathcal{P}$ also belong to $\mathcal{C}$, and that, for a pair of polytopes $\mathcal{P}$, $\mathcal{Q} \in \mathcal{C}$, their intersection, $\mathcal{P} \cap \mathcal{Q}$, is a face for both.

In $\mathbb{R}^s$, a *simplex* is the convex hull of any $1 \leq r \leq s+1$ affinely independent points. It is easy to see that the set of vertices of a simplex is the same as the point set defining it, and that the convex hull of any subset of the vertices is its face.

A set of simplices that form a convex cell complex is called a *simplicial complex.* In a simplicial complex $\mathcal{C}$, the subset of simplices that are not faces of any other simplices is its *inclusion-maximal* set. The inclusion-maximal set gives a compact representation of $\mathcal{C}$: Suppose that $\mathcal{C}$ is in vertex representation and $\mathcal{C}'$ is its inclusion-maximal set, then C can be generated from $\mathcal{C}'$ by taking all-subsets: $\mathcal{C} = \{S \mid S \subseteq T, T \in \mathcal{C}'\}$.

A simplicial complex whose inclusion-maximal simplices have the same dimension is called a *triangulation.* The dimension of a triangulation is the dimension of its inclusion-maximal simplices. The underlying space of a triangulation is the union of its simplices. The following common kinds of triangulations differ mainly in their underlying spaces:

- *The face complex of a polytope.* For a finite set of points $P \subset \mathbb{R}^{s+1}$ in general position—every $s+1$-subset of P is affinely independent, the faces of the polytope $\text{conv}(P)$ are simplices therefore they form an s-dimensional triangulation whose underlying space is the polytope boundary.

- *Point-set triangulation.* For a finite set of points $P \subset \mathbb{R}^s$ that are fully dimensional, a triangulation whose vertices are precisely P and whose underlying space is the polytope $\text{conv}(P)$ is a point-set triangulation and, in particular, a triangulation of the points P. Note that a point set triangulation of P induces a triangulation of the boundary of $\text{conv}(P)$, and if P are in general position, this induced triangulation is exacly the face complex of the convex hull of P.

- *Triangulation of simple polygons.* A simple polygon is a 1 dimensional triangulation whose underlying space is a topological circle. The triangulation of a simple polygon $\mathcal{P}$ is a two dimensional triangulation that satisfies that the boundary of its underlying space is $\mathcal{P}$.

XXX Define $d(t)$.

2.3 Delaunay diagrams and triangulations

For a finite set of point *sites* $P \subset \mathbb{R}^s$, a Delaunay diagram of P, denoted $\mathcal{D}(P)$, is a convex cell complex defined as follows: For a set of points $f \subset P$, the polytope $\text{conv}(f)$ is a face of the Delaunay diagram if there is a sphere S such that f are on S and $P \backslash f$

are outside of S. If we make the general position assumption that no $d+2$ sites are cospherical, then the diagram is a triangulation.

By definition, every site is a vertex in the Delaunay diagram, therefore, we usually represent a face $\operatorname{conv}(f) \in \mathcal{D}(P)$ by the vertex set f, so that $\mathcal{D}(P)$ is represented as a set of subsets of P.

Delaunay diagrams are related to convex hulls in one dimension higher via a *lifting map*, introduced by Brown(1980). To show this relation, let us study the representations of spheres and derive the lifting map.

Geometrically, a sphere in $\mathbb{R}^s$ is the set of points that is some fixed distance away from a chosen point. Simple algebra shows that for any sphere S, there is a tuple $(S_0, \ldots, S_{s+1}) \in \mathbb{R}^{s+2}$, where $S_{s+1} > 0$, such that a point $x \in \mathbb{R}^s$ is on, inside or outside the sphere if and only if the dot product $(S_0, \ldots, S_{s+1}) \cdot (1, x_1, \ldots, x_s, x \cdot x) = 0$, < 0 or > 0. The tuple $(S_0, \ldots, S_{s+1})$ therefore can be taken as a representation of the sphere; any positive multiple clearly represents the same sphere.

If we consider a tuple S representing a sphere in $\mathbb{R}^s$ as representing a hyperplane in $\mathbb{R}^{s+1}$, then we can view the sidedness relation between a point and a sphere as the sidedness relation of a "lifted" point against a hyperplane, as illustrated in Figure 5.2. Formally, let $f : \mathbb{R}^s \to \mathbb{R} : x \mapsto x \cdot x$ denote the unit parabolid function. Let the caret ($\hat{}$) denote the lifting map that takes $\mathbb{R}^s$ to the plot of f in $\mathbb{R}^{s+1}$, i.e. $\hat{} : \mathbb{R}^s \to \mathbb{R}^{s+1} : x \mapsto (x; f(x))$. Then, for a point $p \in \mathbb{R}^s$ and a tuple S representing a sphere, p is inside, on or outside the sphere S if and only if $\hat{p}$ is on the negative side, on or on the positive side of the hyperplane represented by S.

By viewing spheres as hyperplanes in one dimension higher, it is easy to see that circumspheres can be derived by computing a hyperplane after lifting: For a tuple of positively oriented points $s+1$ points $P = (P_0, \ldots, P_s) \subset \mathbb{R}^s$, i.e. $d(P) > 0$, the tuple $H(\hat{P}_0, \ldots, \hat{P}_s)$ represents the circumsphere through the points $P_0, \ldots, P_s$. Note that positive orientation condition implies that the last entry in the tuple is positive so that the computed tuple is a valid representation of a sphere.

Viewing sphere as hyperplanes in one dimension higher, we can generalize the definition of the Delaunay diagram by varing the lifting map. For a hyperplane H in tuple representation, call H *down-facing* if $H_{s+1} > 0$. Let $f : \mathbb{R}^s \to \mathbb{R}$ be any convex function and $\hat{}$ be the associated lifting map. Then, the set of polytopes

$$\{\operatorname{conv}(f) \mid \hat{f} = H \cap \hat{P}, H \text{ is a down-facing hyperplane}, H^- \cap \hat{P} = \emptyset\} \tag{2.4}$$

forms a convex cell complex that generalize Delaunay diagram, i.e. if we set the lifting function to be parabolid, then the complex is the Delaunay diagram. This family of diagrams are known by different names in slightly different contexts. In the dual setting, they are known as the *power diagrams*(Aurenhammer, 1987). If general position is assumed, the diagrams are themselves *regular triangulations*(Edelsbrunner and Shah, 1996).

The definition in Eq. 2.4 makes it clear that the Delaunay diagram can be viewed as the "lower half" of the convex hull $\hat{P}$: If we allow H in Eq. 2.4 to be vertical and up-facing, then we get all the faces of the convex hull $\mathrm{conv}(\hat{P})$. The view of Delaunay diagrams as convex hulls imply that the bound on the size of Delaunay diagrams can be established by the known bound on the size of convex hulls: By the upper bound theorem (McMullen, 1970), the size of a Delaunay diagram is $O(n^{\lceil s/2 \rceil})$ (there are examples of Delaunay diagrams that achieves the bound). This bound implies that in three dimension, Delaunay diagrams have size $O(n^2)$. However, practitioners have always observed size $O(n)$ Delaunay diagrams. This discrepency is explained by many theoretical works that make various realistic assumptions about the input, such as that they are uniformly randomly sampled from space or from surfaces (Attali et al., 2003; Dwyer, 1991; Erickson, 2002).

Chapter 3

Faraway Point: A Sentinel Point for Delaunay tessellation

For a set of points $P \subset \mathbb{R}^s$, a Delaunay diagram of P is bounded by the convex hull of P. The Delaunay faces that belong to the boundary of the convex hull of P are called *boundary faces*. If we assume general position—-there are no coplanar $(s+1)$-subsets nor cospherical $(s+2)$-subsets—then the boundary faces are the same as the convex hull faces. If there are degeneracies, however, the boundary faces subdivide the convex hull faces (see Figure 3.1.)

Because boundary faces are not "surrounded" by higher dimensional faces in the diagram as the non-boundary faces are, they must be treated specially in computation. One simple way to avoid special treatments for the boundary is to lift the points P to $\hat{P}$ and compute the Delaunay diagram as lower faces of $\operatorname{conv}(\hat{P})$, but this leads to computing all the upper faces, which can be undesirable. A good alternative is to compute $\operatorname{conv}(\hat{P} \cup \{e\})$, where e is a *point at infinity* that can be thought of as the intersection of all vertical upward rays. If we assume general positions, the faces of $\operatorname{conv}(\hat{P} \cup \{e\})$ satisfies that each face F which does not correspond to a Delaunay face is incident on e and the face $F \backslash \{e\}$ projects to a boundary Delaunay face. This simple correpondence between the boundary faces of a Delaunay triangulation and the sentinel faces for computing it is attractive for implementation.

The point at infinity e has issues with degeneracies in the input. First, if we are to compute the Delaunay diagram exactly without perturbation, then the simple correspondence between the sentinel faces and the boundary Delaunay faces no longer hold; second, recall from Section 4.1 that, to avoid "flat simplices" in a perturbation scheme, the points in $\hat{P}$ should be perturbed only in the vertical perturbation, but this can not remove degeneracies involving e, since e lies on all vertical rays. In order to preserve the simplicity of implementation with e, but resolve the issues with degeneracies, we define a *faraway point* q and propose computing $\text{conv}(\hat{P} \cup \{q\})$ instead of $\text{conv}(\hat{P} \cup \{e\})$. In an oriented projective geometry, this faraway point differs from the point at infinity only infinitesimally. Therefore, if general position is assumed, the replacement makes no difference in the output. However, for degenerate input, the replacement brings two improvements. First, to compute a triangulation, a vertical perturbation scheme is sufficient to simulate general position. This is because, by Lemma 3.3.5, the support hyperplanes of polytope $\text{conv}(\hat{P} \cup \{q\})$ are down-facing so an incremental algorithm never encounters vertical hyperplanes. Second, for computing the exact diagram, faces incident on e correspond to the boundary Delaunay faces, by Theorem 3.3.6. In comparison, the extra faces with the point at infinity match the convex hull faces, which can introduce complicated relations between the boundary faces and the extra faces. A two dimensional example of this is shown in Figure 3.1.

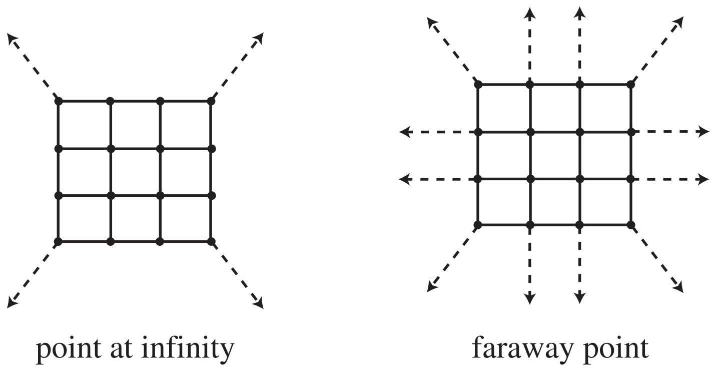

Figure 3.1: For input P, points of 4×4 grid, we draw the 2d projections of the convex hulls of lifted points: at left, $\text{conv}(\hat{P} \cup \{e\}$, where e is the point at infinity, and at right, $\text{conv}(\hat{P} \cup \{q\})$, where q is the faraway point. Dashed segments connect to the point at infinity or the faraway point.

Replacing the point at infinity with our faraway point has a simple implementation policy in code. It adds just one conditional to the sphere inside/outside test. We provide the pseudocode of the sphere sidedness test in Section 3.4, and discuss how it simplifies algorithms based on flipping in Section 3.5.

3.1 Geometric Preliminaries

We review representations of points, hyperplanes and polytopes in an oriented projective space, which contains the familiar Euclidean space. We need this because we will define the faraway point in the non-Euclidean portion of oriented projective space.

Geometric problems in the Euclidean space regarding sidedness relations between points and hyperplanes can be stated in the more general oriented projective setting. Doing so often remove special cases in the solution. We sketch the basics; see also Chapter 7 of Boissonnat and YvinecBoissonnat and Yvinec (1998), or Stolfi's workStolfi (1991) for more mathematical details and artistic drawings.

The points in an n dimensional oriented projective space $\mathbb{P}^n$ are represented by $n+1$-tuples of reals called *homogeneous coordinates.* Let $\mathbf{0} := [0, \ldots, 0]$ be the origin of $\mathbb{R}^{n+1}$. Any tuple $p = [p_0, \ldots, p_n] \in \mathbb{R}^{n+1} \backslash \{\mathbf{0}\}$ is the homogeneous coordinates representation of a point in $\mathbb{P}^n$; any positive multiple of p represents the same point.

The points in $\mathbb{P}^n$ can be identified with the points on an n-dimensional unit sphere centered at the origin in $\mathbb{R}^{n+1}$. The points on the sphere, except those lying on a great circle, can be central-projected to two parallel hyperplanes H and H', as shown in Figure 3.2. We can identify one of these hyperplanes, say H, with the n dimensional Euclidean space $\mathbb{R}^n$, and partition the oriented projective space into three sets: $\mathbb{R}^n$, the antipodes of $\mathbb{R}^n$, and the set of *points at infinity*, which corresponds to the great circle that project to neither H nor H'. Canonically, we choose H to be the hyperplane of points whose first coordinate—the *homogenizing coordinate*—is equal to unity. A point $p \in \mathbb{P}^n$, therefore, represents the Euclidean point $(\frac{p_1}{p_0}, \ldots, \frac{p_n}{p_0})$ if the homogenizing coordinate $p_0 > 0$, a *point at infinity* in the direction of vector $(p_1, \ldots, p_n)$ if $p_0 = 0$, and an antipode to the Euclidean point $(\frac{p_1}{p_0}, \ldots, \frac{p_n}{p_0})$ if $p_0 < 0$.

The affine hull of a set of points $P \subset \mathbb{P}^n$, denoted $\operatorname{aff}(P)$, is the subspace of $\mathbb{P}^n$ generated by taking all linear combinations of the coordinate tuples of P and identifying the tuples that differ by scalar multiples. The dimension of $\operatorname{aff}(P)$ is the rank of the matrix constructed by listing the coordinate tuples from P as columns. An $n-1$ dimensional affine hull is called a *hyperplane.*

Identifying $\mathbb{P}^n$ with an n-sphere, a hyperplane H in $\mathbb{P}^n$ is a great circle that divides the sphere to two open hemispheres. If we label the open hemispheres as the positive and negative sides of H, denoted H^+ and H^- respectively, then a hyperplane can be oriented in two ways, which reverse its positive/negative sides. Algebraically, a hyperplane H is represented by an $n+1$-tuple of reals $\langle H_0, \ldots, H_n \rangle$, so that each a point

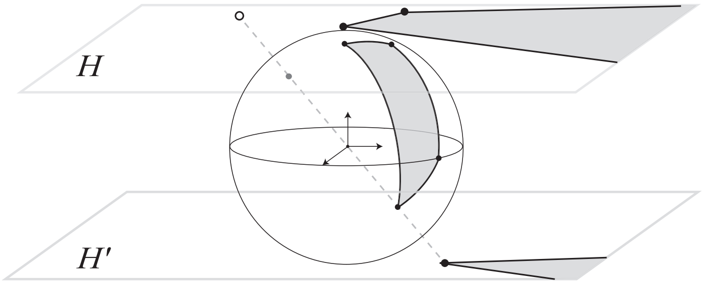

Figure 3.2: The two dimensional oriented projective space $\mathbb{P}^2$ can be identified with a sphere. Through central projection, the upper open hemisphere maps to the Euclidean plane H, and the lower one maps to H'—the antipodes of the Euclidean plane. The shaded region on the sphere is a polytope in which one vertex is antipodal to an Euclidean point—drawn as a hollow dot. The projection of the polytope onto the two Euclidean planes are two open polygons.

$p \in \mathbb{P}^n$, satisfies $p \in H$, or H^+, or H^- , if the dot product $[p_0, \ldots, p_n] \cdot \langle H_0, \ldots, H_n \rangle = 0$, or > 0, or < 0.

The tuple representing a hyperplane can be computed by first choosing n affinely independent points $(A_1, \ldots, A_n)$ on the hyperplane, adding a symbolic point p, and computing the $n{+}1$ minors of the determinant $\left| A_1, \ldots, A_n, p \right|$ with respect to $p_0, \ldots, p_n$. We denote the resulting tuple by $H(A_1, \ldots, A_n)$.

If we continue to identify $\mathbb{P}^n$ with an n-sphere, any two points $a, b \in \mathbb{P}^n$ that are not anti-podal define a line segment $\overline{ab}$ as the shortest geodesic curve between a and b. Therefore, in the oriented projective space, the notion of convexity is well-defined for point sets that lie to one side of some halfplane. An *oriented projective polytope* is the convex hull of such a set of points. The faces of a polytope are defined as follows.

Definition 1. *Let $P \subset \mathbb{P}^n$ be a set of points. A subset $f \subset P$ forms a face* $\mathrm{conv}(f)$ *of* $\mathrm{conv}(P)$ *if there is a hyperplane H such that $H \cap P = f$ and one side of H is empty of points in P.*

The defining hyperplanes of the faces of a polytope are called *support hyperplanes.* If the non-empty side of a support hyperplane is designated its positive side, then we say that the support is positive; otherwise a support is negative.

3.2 The lifting map

The goal of this section is to study points and hyperplanes in d and $d+1$ dimensions that are related by a lifting map. For a real function f over $\mathbb{R}^s$, the lifting map

$\hat{}: \mathbb{R}^s \to \mathbb{R}^{s+1} : x \mapsto (x; f(x))$ takes points in d dimensions vertically to the plot of f in $d+1$ dimensions. The vertical direction in $d+1$ dimensions can be formally established by the point at infinity $e := [0, \ldots, 0, 1]$, so that a hyperplane H is vertical whenever $e \in H$.

Lifting a point set does not change its affine dimension. Inversely, for a lifted point set $\hat{P} \subset \mathbb{P}^s$, if there is a non-vertical hyperplane through $\hat{P}$, then the affine dimension of P is the same as $\hat{P}$. Otherwise, the affine dimension of P can be one smaller than that of $\hat{P}$.

If the lifting function is chosen to be the paraboloid, then the hyperplanes whose positive sides contain e—*the down-facing hyperplanes*—correspond to spheres: For a sphere S with equation $\langle S_0, \ldots, S_{s+1}\rangle[1; x; x \cdot x] = 0$, a point $p \in \mathbb{R}^s$ is on, inside or outside S if and only if $\hat{p}$ is on, on the positive side, or on the negative side of the hyperplane represented by the tuple S. The hyperplane is down-facing because $\langle S_0, \ldots, S_{s+1}\rangle \cdot e = S_{s+1} > 0$.

A vertical hyperplane in $\mathbb{P}^{s+1}$ vertically projects to a hyperplane in $\mathbb{P}^s$. Testing sidedness of lifted points with vertical hyperplanes is equivalent to testing sidedness of the original points with the projection of the vertical hyperplanes. To be precise, for a point $p \in \mathbb{P}^s$ and a set of affinely independent points $\{f_1, \ldots, f_d\} \subset \mathbb{P}^s$, $\left| f_1, \ldots, f_d, p \right| = -\left| \hat{f}_1, \ldots, \hat{f}_d, e, \hat{p} \right|$.

The set of down-facing hyperplanes through a non-vertical $(d-1)$-flat, i.e., the affine hull of d affinely independent points, is a one-parameter family. A point on the vertical hyperplane through the flat lies on the same side of all members of the family.

Lemma 3.2.1. *Let $f \subset \mathbb{P}^s$ be a set of points whose affine dimension is $d-1$. Let H^e denote the vertical hyperplane though f. A point $p \in H^e$ is on the same side of all down-facing hyperplanes through f.*

Proof. Let $v \in \mathbb{P}^s$ be a point such that $v \notin \mathrm{aff}(f)$. Choose a set of affinely independent points $\{f_1, \ldots, f_d\} \subset f$. Any down-facing hyperplane can be represented the tuple $H(\hat{f}_1, \ldots, \hat{f}_d, [v; \lambda])$, for some $\lambda \in \mathbb{R}$. The sidedness of p with respect to this hyperplane is decided by the determinant:

$$\left| \hat{f}_1, \ldots, \hat{f}_d, [v; \lambda], p \right| = \left| \hat{f}_1, \ldots, \hat{f}_d, [v; 0], p \right| + \lambda \left| \hat{f}_1, \ldots, \hat{f}_d, e, p \right| = \left| \hat{f}_1, \ldots, \hat{f}_d, [v; 0], p \right|.$$

Since the value of the determinant is independent of λ, we have proved the claim. □

3.3 The Polytope with a Faraway Point

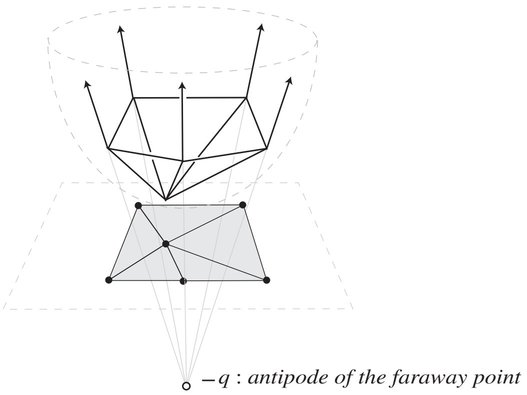

Figure 3.3: Computing Delaunay diagram using the lifting map and the faraway point q. The shaded planar subdivision is the Delaunay diagram. The open polyhedron is the Euclidean portion of an oriented projective polytope with vertex q. Since we cannot draw q in Euclidean space, we draw its antipode $-q$ as an unfilled circle.

We define the *faraway point* as follows: For a set of points $P \subset \mathbb{R}^s$ such that $\mathrm{aff}(P)$ is d-dimensional, the *faraway point* with respect to P is a projective point $q \in \mathbb{P}^{s+1}$ with coordinates $[q_0, \ldots, q_{s+1}]$ such that $q_0 = -1$, $-(q_1, \ldots, q_d)$ is interior to the convex hull of P, and q_{s+1} is a symbol ∞ that is a place holder for a large enough number.

For a set of points P with faraway point q, we propose computing its Delaunay diagram by computing the oriented projective polytope $\mathcal{P} := \mathrm{conv}(\hat{P} \cup \{q\})$. The polytope $\mathcal{P}$ is well-defined because:

- $\hat{P}$ belong to the Euclidean half space of $\mathbb{P}^{s+1}$, defined by the hyperplane at infinity $\langle 1, 0, \ldots, 0 \rangle$. If we perturb the hyperplane slightly to $\langle 1, 0, \ldots, 0, \frac{1}{\infty} \rangle$, the corresponding half space contains both $\hat{P}$ and q.

- For large enough ∞, the sidedness relations of q with respect to the hyperplanes determined by points in $\hat{P}$ do not change.

The structural properties of the polytope $\mathcal{P}$ are stated formally in Lemma 3.3.5 and Theorem 3.3.6. Lemma 3.3.5 says that the support hyperplanes of $\mathcal{P}$ are all down-facing. Therefore a vertical perturbation scheme, which removes any non-vertical degeneracies, is enough for its incremental construction. Theorem 3.3.6 says that, regardless of whether general position is assumed or not, the non-Delaunay faces of this polytope correspond one-to-one to the boundary Delaunay faces.

The rest of this paper contains proofs of the main results. We will make the standing assumption that P is a set of points such that $\operatorname{aff}(P)$ is d-dimensional, that q is its faraway point, and that $\mathcal{P}$ is the polytope $\operatorname{conv}(\hat{P} \cup \{q\}$.

Note that our definition of Delaunay diagrams (Definition **??**) places no restrictions on the choices of the lifting function. We formally define boundary Delaunay faces as follows:

Definition 2. *For a set of points* $P \subset \mathbb{R}^s$*, a Delaunay face* $\operatorname{conv}(f \subset P)$ *is a boundary face if the hyperplane* H *through* f *is a support of* $\operatorname{conv}(P)$*, i.e.* $H \cap P \supset f$ *and* $H^- \cap P = \emptyset$ *or* $H^+ \cap P = \emptyset$*.*

The faraway point q satisfies the following properties:

- Because of the arbitrarily large coordinate ∞, the faraway point q must be a vertex of $\mathcal{P}$ and can not belong to any affine flat from $\hat{P}$. Therefore, for any n-dimensional face $\operatorname{conv}(\hat{f} \cup \{q\})$ of $\mathcal{P}$, $\operatorname{conv}(\hat{f})$ is an $n-1$ dimensional face.
- q is an infinitesimal perturbation of the point at infinity e: $[q_0, \dots, q_d, \infty]$ represents the same projective point as $[\frac{q_0}{\infty}, \dots, \frac{q_d}{\infty}, 1]$, which, as ∞ increases, converges to $[0, \dots, 0, 1]$. The statement that q differs from e only infinitesimally can also be expressed in terms of sidedness relations: If the points are in general position, we can replace the faraway point q by the point at infinity e without changing any sidedness relations, as formally stated in Lemma 3.3.1.

Lemma 3.3.1. *Let* $\{f_1, \dots, f_{s+1}\} \subset \mathbb{P}^s$ *be a set of affinely independent points. Then* $\operatorname{sign}\left|\hat{f}_1, \dots, \hat{f}_{s+1}, e\right| = \operatorname{sign}\left|\hat{f}_1, \dots, \hat{f}_{s+1}, q\right|$.

Proof. Because of the affine independence of the set $\{f_1, \dots, f_{s+1}\}$,

$$\left|f_1, \dots, f_{s+1}\right| = \left|\hat{f}_1, \dots, \hat{f}_{s+1}, e\right| \neq 0.$$

Thus, the determinant

$$\left|\hat{f}_1, \dots, \hat{f}_{s+1}, [q_0, \dots, q_d, \infty]\right| = \left|\hat{f}_1, \dots, \hat{f}_{s+1}, [q_0, \dots, q_d, 0]\right| + \infty \left|\hat{f}_1, \dots, \hat{f}_{s+1}, e\right|$$

is dominated by the determinant $\left|\hat{f}_1, \dots, \hat{f}_{s+1}, e\right|$. Therefore, the signs of the determinants are the same. □

Lemma 3.3.2. *The support hyperplanes of* $\mathcal{P}$ *through* q *are not vertical.*

Proof. Let $o := -[q_0, \ldots, q_d]$, so that o represent an Euclidean point interior to $\operatorname{conv}(P)$. Any vertical hyperplane H^e through q projects to a hyperplane H through o. Since o is interior to $\operatorname{conv}(P)$, H has points of P on both sides. Therefore H^e does not support $\operatorname{conv}(\hat{P} \cup \{q\})$. □

Lemma 3.3.3. *Let* $\operatorname{conv}(\hat{f} \cup \{q\})$ *be a facet incident on* q *in* $\mathcal{P}$. *Then, there is a hyperplane through* f *that is a support of* $\operatorname{conv}(P)$.

Proof. Since the affine dimension of $\hat{f}$ is d and, by Lemma 3.3.2, $\hat{f}$ are on a non-vertical hyperplane, the affine dimension of f is d.

Choose a set of points $\{f_1, \ldots, f_d\} \subset f$ such that $H(\hat{f}_1, \ldots, \hat{f}_d, q)$ is the support of $\operatorname{conv}(\hat{f} \cup \{q\})$. Then, for any point $p \in P$, $\left|\hat{f}_1, \ldots, \hat{f}_d, q, \hat{p}\right| \geq 0$. Then, by Lemma 3.3.1, $\left|\hat{f}_1, \ldots, \hat{f}_d, e, \hat{p} \geq 0\right|$. Equivalently, $\left|f_1, \ldots, f_d, p\right| \leq 0$. Therefore, $H(f_1, \ldots, f_d)$ is a support of $\operatorname{conv}(P)$. □

Lemma 3.3.4. *Let* $\operatorname{conv}(f)$ *denote a boundary Delaunay facet. Then,* $\operatorname{conv}(\hat{f} \cup \{q\})$ *is facet of* $\mathcal{P}$.

Proof. Let $o := -[q_0, \ldots, q_d]$ so that o is interior to $\operatorname{conv}(P)$. Choose $\{f_1, \ldots, f_d\} \subset f$ so that $H(f_1, \ldots, f_d)$ represents the negative support of $\operatorname{conv}(f)$ in $\operatorname{conv}(P)$. Then, $\left|f_1, \ldots, f_d, o\right| < 0$.

Let $H^q := H(\hat{f}_1, \ldots, \hat{f}_d, q)$. H^q is down-facing because $H^q \cdot e = \left|\hat{f}_1, \ldots, \hat{f}_d, q, e\right| = \left|f_1, \ldots, f_d, -o\right| > 0$. Let H denote the Delaunay support for $\operatorname{conv}(f)$.

Choose any $p \in P$. Consider two cases. *Case I*: p is on the hull support $H(f_1, \ldots, f_d)$. Then, by Lemma 3.2.1, the sidedness of p with respect to H^q is the same as the Delaunay support H. Therefore, if $\hat{p} \in \hat{f}$, $\hat{p} \in H^q$ and otherwise, $\hat{p} \in (H^q)^+$.

Case II: p is on the negative side of the hull support $H(f_1, \ldots, f_d)$. Then, $\left|f_1, \ldots, f_d, p\right| < 0$. Then, $\left|\hat{f}_1, \ldots, \hat{f}_d, e, \hat{p}\right| > 0$. Then, by Lemma 3.3.1, $\left|\hat{f}_1, \ldots, \hat{f}_d, q, \hat{p}\right| > 0$.

By these two cases, H^q supports the facet $\operatorname{conv}(\hat{f} \cup \{q\})$ in $\mathcal{P}$.

□

Lemma 3.3.5. *Let* $P \subset \mathbb{R}^s$ *be a finite set of points such that* $\operatorname{aff}(P)$ *is d-dimensional. Let* q *be the faraway point with respect to* P. *Then, the positively oriented support hyperplanes of the polytope* $\operatorname{conv}(\hat{P} \cup \{q\})$ *are down-facing.*

Proof. Let $o := -[q_0, \ldots, q_d]$ so that o is interior to $\operatorname{conv}(P)$. Let H denote a positive facet support of $\operatorname{conv}(\hat{P} \cup \{q\})$.

If H is not vertical and does not contain q, then, because $q \in H^+$, by Lemma 3.3.1, $e \in H^+$. Otherwise, either H is vertical or H contains q. In either case, there exists a set of affinely independent points $\{f_1, \ldots, f_d\}$ such that $H(f_1, \ldots, f_d)$ supports $\operatorname{conv}(P)$. The set exists in the first case because H is vertical and in the second by Lemma 3.3.3. Then, for any $p \in P$, $\left|f_1, \ldots, f_d, p\right| \leq 0$ Then, since o is interior to $\operatorname{conv}(P)$, $\left|f_1, \ldots, f_d, o\right| < 0$. Then, $\left|\hat{f}_1, \ldots, \hat{f}_d, [-o; \infty], e\right| = \left|\hat{f}_1, \ldots, \hat{f}_d, q, e\right| > 0$. The last inequality implies that, first, if q belongs to H, then H is down-facing, and, second, H can not be vertical otherwise q will be on its negative side, which violates the condition that H is a positive support. Therefore H is down-facing. □

Theorem 3.3.6. *Let $P \subset \mathbb{R}^s$ be a finite set of points such that $\operatorname{aff}(P)$ is d-dimensional. Let q be the faraway point with respect to P. Then, for a set of points $f \subset P$,*

1. $\operatorname{conv}(f)$ *is in the Delaunay diagram of P if and only* $\operatorname{conv}(\hat{f})$ *is a face of the polytope* $\operatorname{conv}(\hat{P} \cup \{q\})$.
2. $\operatorname{conv}(f)$ *is a boundary face of the Delaunay diagram of P if and only if* $\operatorname{conv}(\hat{f} \cup \{q\})$ *is a face of the polytope* $\operatorname{conv}(\hat{P} \cup \{q\})$.

Proof. It is enough to show that the theorem is true for the highest dimensional faces.

Let $\operatorname{conv}(f)$ be a Delaunay facet. Its support hyperplane H is down-facing, i.e. $e \in H^+$. By Lemma 3.3.1, $q \in H+$. Therefore, H supports the facet $\operatorname{conv}(\hat{f})$ in $\operatorname{conv}(\hat{P} \cup \{q\})$.

Let $\operatorname{conv}(f)$ be a boundary Delaunay facet, by Lemma 3.3.3, $\operatorname{conv}(\hat{f} \cup \{q\})$ is a facet of $\operatorname{conv}(\hat{P} \cup \{q\})$.

Let $\operatorname{conv}(\hat{f} \cup \{q\})$ be a facet of $\operatorname{conv}(\hat{P} \cup \{q\})$. By Lemma 3.3.5, its support hyperplane H is down-facing. Therefore H is a Delaunay support for $\operatorname{conv}(f)$. □

As a corollary of Lemma 3.3.5 and Theorem 3.3.6, the vertical projection of the faces of $\operatorname{conv}(\hat{P} \cup q)$ subdivide the space $\mathbb{R}^s$. This property does not hold for either $\operatorname{conv}(\hat{P})$ or $\operatorname{conv}(\hat{P} \cup \{e\})$: the vertical projection of $\operatorname{conv}(\hat{P})$ has two overlapping subdivisions, and the vertical projection of $\operatorname{conv}(\hat{P} \cup \{e\})$ is not well defined since the the point at infinity e projects to $[0, \ldots, 0]$, which does not represent any point.

3.4 Implementation policy for sidedness test

Computing Delaunay diagrams via convex hulls needs only one geometric predicate—the point/hyperplane sidedness test. Since the hyperplanes of the polytope $\operatorname{conv}(\hat{P} \cup q)$

are down-facing, and hence represent spheres, these are the *InSphere* testsBoissonnat and Yvinec (1998). We now describe an implementation policy for the InSphere test consistent with the symbolic coordinate ∞ in the faraway point q. The "spheres" touching q are arbitrarily large because of the symbolic coordinate in q. To test a point p against such a sphere through q and a Delaunay facet $\text{conv}(f)$, first test whether p is on the hyperplane through f. If p is not on the hyperplane, by Lemma 3.3.1, the sidedness of p with respect to this hyperplane can be returned. Otherwise, by Lemma 3.2.1, we can choose any other sphere through f to test sidedness. For example, the following implementation choices are possible.

- When $d = 2$ only, the $d - 1$ sphere through f is a line segment. Instead of choosing a circle through f, it is easy to test whether p is interior to the line segment determined by f.

- Choose any point v not on the hyperplane through f; test p against the sphere through f and v.

- Let $\text{conv}(f, v)$ be the neighboring Delaunay cell across the facet $\text{conv}(f)$; test p against the sphere through f and v.

The last implementation is particularly efficient. This is because an incremental algorithm often has a search phase that looks for Delaunay spheres invaded by p, and the search phase might have already performed the sidedness test with respect to the neighboring sphere. We provide pseudocode for the last approach in Table 3.1. In the pseudocode, by choosing one of the two return statements, a programmer can choose whether to implement the test exactly or simulate general position in a way consistent with an incremental vertical perturbation scheme. Unlike Devillier's vertical perturbation scheme, which assigns perturbation to the input points prior to insertion, the perturbation here is simpler, returning "inside" whenever an "on" is encountered, but is dependent on the insertion order.

3.5 Discussion

We have computed Delaunay diagrams as convex hulls in one dimension higher. Another way to compute Delaunay triangulations, without explicitly using a higher dimension, is called *flipping*Edelsbrunner and Shah (1996); Shewchuk (1996b), so named

The **InSphere Test**

global variables and subroutines

```
//the faraway point,
//   represented by its limit
$\mathbf{q}$ := (0, ..., 0, 1)

//Toggle perturbation
PERT := True; // False for exact

Neighbor($c$: cell, $f$: facet)
  Let $c'$ be the neighbor cell of $c$
     across $f$;
  return $c'$;

Sphere($c$: cell)
  return $H$, the tuple representing
     the support hyperplane of $c$;
```

pseudocode

```
//Test whether a point is inside (−1), on(0)
//  or outside(1) the sphere for a Delaunay cell.
InSphere($c$: cell, $p$: lifted site in
                  homogeneous coordinates
   $d$ := Sphere($c$)·$p$; // Orientation determinant
   if ($d \neq 0$) return sign($d$); // non-zero (not on)
   if ($\mathbf{q} \notin c$) // $c$ does not have the faraway point
      if (PERT)
         return -1; // vertically perturb
      else
         return sign($d$); // no perturbation
   else      // $c$ has faraway point
      $c'$ := Neighbor($c$, $c \backslash \mathbf{q}$);
      return InSphere($c'$, $p$); // test neighbor
```

Table 3.1: The InSphere test requires little change with the faraway point q, whether perturbation is used or not. The code assumes that the cells are represented by point sets. When it discovers a cell using q, it obtains a neighbor cell c' from the data structure and tests q against the sphere for c'.

because it applies flip operations to small groups of simplices. Flipping uses two geometric predicates, point-sphere sidedness test and point-hyperplane sidedness, used to determine whether a flip is possible and in a routine to locate a point inside a simplex. To avoid special cases for the boundary, the flipping approach places $d+1$ “points at infinity” in d dimensions away from the convex hull of the sites (these have symbolic coordinates and are not to be confused with the projective points at infinity) and use special codes in the predicates to handle their symbolic coordinates.

We notice that a single faraway point may be substituted for the $d+1$ “points at infinity” to implement a flipping algorithm: The point-sphere sidedness test can be performed as described in Section 3.4; the point-hyperplane sidedness can be performed with respect to the projection of faraway point, which becomes an antipode to a point in the d dimensional Euclidean space. The substitution of the faraway point simplifies the implementation considerably: Not only do we reduce the number of special points from $d+1$ to one, but the one special point, when projected to d-dimensions, does not even have a symbolic coordinate so needs no special codes for point-plane sidedness test. However, the correctness analysis of the flipping algorithm inEdelsbrunner and Shah (1996) is carried out for a bounded triangulation. It will be interesting to check whether the correctness holds when the flip operations are applied to maintain a triangulation in which some simplices are infinite.

Chapter 4

Engineering Delaunay Tessellation Programs

In implementing an incremental Delaunay triangulation algorithm, there are a number of engineering decisions that must be made by implementors, including the type of arithmetic, degeneracy handling, data structure representation, and low-level algorithms. In Section 4.1, I compare how these decisions are made by a number of publicly available 3D Delaunay triangulation programs. These programs include my own program TESS3, which is designed to work well with atom coordinates data from biological applications. The details of TESS3 is described in Section 4.4. The engineering ideas from TESS3 also goes into my 4D Delaunay diagram program, DD4, which is designed to work well with time-varying volume data from scientific simulations. The details of DD4 and performance comparison is presented in Section 4.6.

4.1 Incremental construction of Delaunay triangulation

In $\mathbb{R}^s$, assuming that a set of sites $P = \{P_1, \ldots, P_i\}$ are in general position—P has no $s+2$-subsets that are cospherical, the Delaunay triangulation $\mathcal{D}\{P_1 \ldots, P_i\}$ can be computed from the Delaunay triangulation $\mathcal{D}\{P_1, \ldots, P_{i-1}\}$ as follows:

- Delete all d-simplices in $\mathcal{D}\{P_1, \ldots, P_{i-1}\}$ whose circumsphere have P_i inside. These simplices are said to be *in conflict with* P_i.

- For each $(s-1)$-simplex F in $\mathcal{D}\{P_1, \ldots, P_{i-1}\}$ that is a common facet between a

d-simplex in conflict with P_i and one that is not—*a hole facet*, construct a new simplex $F \cup \{P_i\}$.

Therefore, the Delaunay triangulation of n sites can be constructed by initializing with a single s-simplex and run the above incremental construction $n - (s + 1)$ times. The simplicity of this incremental construction scheme makes it a popular basis for designing Delaunay triangulation algorithms.

An incremental Delaunay triangulation algorithm usually represents a Delaunay triangulation not only by its set of d-simplices but also by their *neighbor relations*: Two d-simplices are *neighbors* if they share a common $d - 1$-simplex. Given the neighbor relations of a Delaunay triangulation, to find the simplices in conflict with P_i, it is necessary to search only for one simplex that is in conflict with P_i—the rest can be identified by performing a graph search from that one simplex.

An incremental Delaunay triangulation algorithm usually randomizes the insertion order, i.e., $\{P_1, \ldots, P_n\}$ is a random permutation of the points in P. Randomizing guarantees that the expected cost of the ith incremental step is the size of $\mathcal{D}\{P_1, \ldots, P_i\}$ divided by i—the best that can be hoped for. In particular, in two dimensions, randomizing the insertion order guarantees that the expected cost of an incremental step is six.

Incremental algorithms differ from each each other mainly in how they locate the first simplex in conflict with P_i—the *point location proecdure*—and and how they create the new simplices around P_i—the *update* procedure. Let use survey the existing solutions for these procedures.

For point location, if its performance is measured by worst case time, then, in two dimensions, the fastest procedure uses a *history DAG*, which takes $O(\log(i))$ expected worst case time (Guibas et al., 1992); in higher dimensions, the fastest procedure uses linear programming and takes $O(i)$ expected time (Seidel, 1991). In practice, however, these procedures are often not used because they are complicated to implement and impose a large computational overhead. Instead, a more practical procedure locates a point by performing a *walk*: Start from some initial simplex and keep stepping into a neighboring simplex until a simplex is found to be in conflict with P_i. There are two main variants on the walk:

- *Straight-line-walk.* The walk starts from some point x in the initial simplex and visits the simplices stabbed by the line between x and P_i. For uniformly distributed points, the expected number of steps is $O(n^{1/s})$.

- *Remembering-stochastic-walk* (Devillers et al., 2002). Suppose that the walk pauses at a simplex T, to decide the next simplex to step into, choose a facet F of T so that P_i and T are on the opposite side of the hyperplane $\operatorname{aff}(F)$—there are at most d of them—and step to the neighboring simplex across F. The walk always terminates by the acyclic theorem by Edelsbrunner (1989).

For the update step, there are mainly the following two ways:

- *Bowyer-Watson* (Bowyer, 1981; Watson, 1981). The update is performed in three phases: Deleting the simplices in conflict; creating the new simplices; and establish the neighboring relations among the new simplices.

- *Flipping* (Edelsbrunner and Shah, 1992). In two dimensions, for two neighboring triangles $\{a, b, u\}$ and $\{a, b, v\}$ in a triangulation such that the quadrilateral $[a, b, u] \cup [a, b, v]$ is convex, a *flip* operation replaces the triangles $\{a, b, u\}$ and $\{a, b, v\}$ by $\{u, v, a\}$ and $\{u, v, a\}$. To perform the update procedure with flips, locate an old Delaunay triangle that contains P_i, split this triangle into three new triangles abutting on P_i, and keep applying flip operation to a neighboring pair of new and old triangles whenever the old triangle is in conflict with P_i. This process is guaranteed to terminate in a number of steps proportional to the number of triangles incident on P_i in $\mathcal{D}\{P_1, \ldots, P_i\}$. In higher dimensions, flipping can be defined analogously and the result on the number of flips holds.

4.2 Exact predicate computation

In geometric computations, a *predicate* is an algebraic expression whose sign is used by an algorithm to make decisions. The most common predicate used for Delaunay triangulation is the InSphere predicate. For a set of $s + 1$ points $\{A_0, \ldots, A_s\} \subset \mathbb{R}^s$, the InSphere predicate is some expansion of the determinant $d(\hat{A}, \ldots, \hat{A}_s)$, which is a polynomial of degree $s + 2$ in terms of the input coordinate.

It is important that the signs of the predicates are computed correctly, since an error can cause a program to make a wrong branching decision and, from that point on, behave unpredictably. Unfortunately, on one hand, the fast floating point arithmetics commonly built into the computer hardwares have round-off errors therefore can not guarantee that the signs are always evaluated correctly; on the other hand, exact arithmetics—such as those provided by the GMP library—are slow. To use the fast floating point arithmetic in a way that still guarantees the correct evaluation of predicates, two main approaches are often used:

- *Floating point filter.* Let f denote the predicate expression. Replacing the arithmetic operation in f by the floating point operation gives another expression $\tilde{f}$. Then, $|\tilde{f} - f|$ represents the round-off error from floating point arithmetics. A floating point filter is an upper bound on $|\tilde{f} - f|$. If a filter B is derived for $|\tilde{f} - f|$ before a program run, then during the program run, every evaluation $\tilde{f}$ is compared with B: If $\tilde{f} < B$, then the sign of $\tilde{f}$ must be correct; otherwise, an exact arithmetic operation is used to evaluate the sign. There are several variants on the floating point filters, depending on how much run time information is used. In particular, the filters that do not use any run time information are called static filters (For example, the filters used by the computational geometry library CGAL (Melquiond and Pion, 2005)).

- *Exact algorithms.* Although the exact evaluation of a degree k polynomial requires k times the number of input bits, the exact evaluation of the sign of the polynomial can require less. Examples include the algorithms by Clarkson (1992) and Avnaim et al. (1997).

4.3 Handling of degeneracies

Delaunay triangulation is well defined only after assuming general position—the input point set does not contain cospherical $(s + 2)$-subsets, or *degeneracies.* However, this assumption is frequently violated in the real world. For example, in a set of points positioned on an integer grid, every subset around a grid cell is cospherical. In order to compute a triangulation, the degeneracies must be removed. This can be done either by actual perturbation of the coordinates or by symbolical perturbation, which perturbs the input coordinates by functions of ϵ that goes to zero as ϵ goes to zero. Of these two methods, the symbolic perturbation is superior: It guarantees that no degeneracy exists after the perturbation and its alteration on the input data is only infinitesmal. The simplest symbolic perturbation can be performed implicitly during an incremental construction: At an incremental construction step for a new point P_i, if P_i is found to be a sphere, simply treats P_i as if it is inside (or outside) the sphere. Since the perturbation depends on the order of the input points, the output of a triangulation program can be different for different ordering of the input, which might not be desirable for some applications. For these applications, two alternatives are available:

- *Perturbing the world* (Alliez et al., 2000). An infinitesimal affine transform is applied to the coordinate system so that cospherical points disappear.
- *Simulation of simplicity* (Edelsbrunner and Mücke, 1990). The coordinates of each input point is translated by symbolic expressions defined with respect to the index of the point. The indexing of the points therefore can be used to control the perturbation.

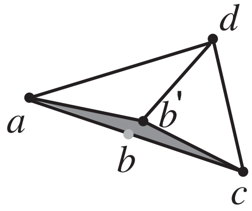

Figure 4.1: Perturbing b to b' produces the shaded flat triangle.

A common issue of the perturbation scheme is that it might produce artificial flat simplices—simplices whose actual volume (without perturbation) is zero, as illustrated in Figure 4.1. For example, Mücke (1998), using the simulation-of-simplicity perturbation, observed that for 3D grid points, more than one third of the tetrahedra were flat. One ad-hoc way to avoid flat simplices is to be vigilant in the incremental update and never create them, but this requires special cases in the code. Devillers (2003) suggests avoiding flat simplices by a simple "vertical" perturbation scheme. Recall that Delaunay triangulation can be considered more generally as a *regular triangulation* defined by first lifting the points to one dimension higher. If the perturbation is applied only to the lift coordinates, the resulting regular triangulation can not have flat simplices because the input coordinates are not altered. This vertical perturbation, however, does not completely resolve the issue, due to the way the boundary of Delaunay triangulation is usually handled. We will fully resolve this issue in Chapter 3.

4.4 TESS3: A Delaunay triangulation program for protein molecular data

Biological applications often model the atoms of a protein molecules as points in $\mathbb{R}^3$ and analyze the geometric structure of a protein molecule by first computing the Delaunay triangulation of its atoms (Richards, 1974; Liang et al., 1998) XXXmore citations. This motivates me to engineer a Delaunay triangulation program, TESS3, that is designed to be fast for protein molecular data.

All available protein molecule data sets are stored in the PDB (Protein Data Bank) (Berman et al., 2000) depository. They have the following characteristics:

- *Even distribution.* The atoms in a protein are well-packed. Therefore, the points representing the atoms tend to be evenly distributed, with physically-enforced minimum separation distances.
- *Limited precision.* By PDB file format, atom coordinates have an 8.3f field specification in units of Ångstroms; they may have three decimal digits before the decimal place (four if the number is positive), and three digits after. Thus, an atom coordinate has at most 24 bits, with differences between neighboring atoms usually needing 12 bits. Since the experimental techniques do not give accuracies of thousandths or even hundredths of Ångstroms, we may even reduce these limits.

Since the coordinates have limited precision, I decide to see if it is possible to stretch the use of standard IEEE 754 double precision floating point arithmetic (, 1985) to evaluate predicates, which can produce errors because of round-offs. Specifically, I try to partition the points by the coordinate bits to reduce the precision needed to evaluate predicates. Because the data are evenly distributed, I try to speed up the point location procedure—the bottle neck of an incremental construction—by ordering the points in a spatially coherent manner. The rest of this section describe these techniques in details and show results of testing the program against all available PDB files.

4.4.1 Bit-leveling and Ordering with Space filling curves

Point location, if not implemented carefully, becomes the bottleneck in 3d Delaunay construction. In the literature, algorithms designed for optimal worst-case performance may use randomization to avoid “bad” point orders and may maintain separate point locations structures on top of the tessellation.

With the evenly distributed points encountered in practice, it is simpler and more efficient to spatially sort the points and use some form of walk in the tessellation from a recently created tetrahedron to a tetrahedron or sphere containing the new point. By spatially sorting the input points, we also improve the “cache coherence” of an incremental construction program. This is particularly important considering the memory hierarchies of modern computer architectures and the large size of the data input frequently encountered in practice, such as point clouds from laser scans.

There is a tension between wanting to insert points near previously inserted points, so that the walk is short, and wanting to insert points evenly across space so that local clusters do not increase the size of the tessellation by creating long, skinny tetrahedron. This tension was seen in the construction “con BRIO” of Amenta *et al.* Amenta et al. (2003a), which first partition input randomly to exponentially larger sets, and order points spatially within each random partition.

We partition our input points by an approach that we call “bit leveling,” which is designed to reduce the number of floating point bits needed to perform the computations correctly. While computing the bounding box of the input sites P, we take histograms of the 8-bit suffixes for the sites’ x, y, and z coordinates. Then we determine the most common 1-bit suffix for all three coordinates, and the most common k-bit suffix, given the common $(k-1)$-bit suffix. The level of a point is the number of common suffixes it matches, from 0 to 8. To avoid the histogram computation, simply assign the level k of a point p as the minimum number of trailing zero bits in p_x, p_y, and p_z. Note that if the least-significant coordinate bits are random, we would expect 7/8 of the points to appear at the bottom level and 1/8 to be passed up. If more are passed up, that means more correlation among their coordinates. During the incremental Delaunay computation, the levels are inserted in the reverse order, i.e., the points on the 0th level are inserted first and those on the 8th level last.

Bit leveling ensures that all points inserted in levels k and above share k least significant bits; in the InSphere() computation, these will be subtracted off from the mantissa, so that fewer bits of precision are needed for correct evaluation. At the lower levels, if our assumption of even distribution of points is valid, then we can hope that the InSphere predicate is performed on nearby points, so that higher-order bits will be subtracted off.

Within each level, TESS3 orders the input points by bucketing them into a grid so that only a small number of points remain in each grid cell and then ordering the grid cells. We experimented with a number of grid cell orderings, which includes row-major ordering, , Z-ordering, Gray code Gray (1953) and Hilbert curve (Figure 4.2), which is known to have good locality-preserving properties when used for indexing a grid (Moon et al., 2001; Niedermeier et al., 2002). We found that indeed, Hilbert curve gives the best result, but the winning margin is small. Figure 4.3 shows the running time comparison using 10 sets of randomly generated 100K points and five ordering, as one

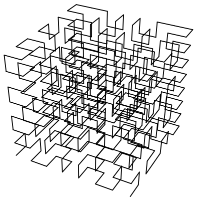

Figure 4.2: Hilbert curve for an $8 \times 8 \times 8$ grid.

typical example. Bit leveling was used with each example.

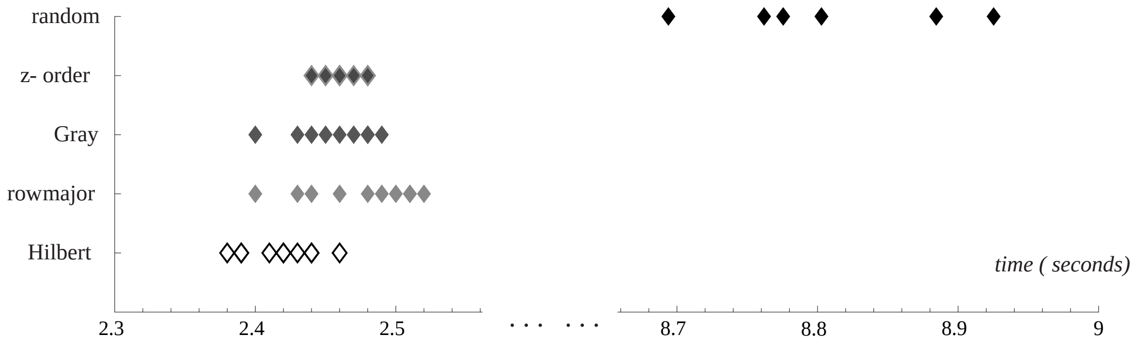

Figure 4.3: Running times of incremental Delaunay tessellation for ten input sets of 100K randomly distributed points under four spatial orders: Hilbert, row-major, Gray, Z-order and random order

4.4.2 Selected implementation details

For point location, TESS3 uses the remembering stochastic walk (defined in Section 4.1). TESS3 uses Bowyer-Watson update, so the point location walk stops when we find a tetrahedron whose sphere has p inside. We have observed that the walk performs remarkably well in practice: With spatial sorting, it usually terminates after 2-3 steps.

We speed up point location in TESS3 by storing the sphere tuple for each tetrahedron. Even though spheres are not tested too many times in our point location, it is still faster to compute and store the sphere equations so the point-in-sphere test becomes a simple dot product. We have compared implementations of TESS3 with and without storing sphere equations and found that the version storing spheres is faster by about 20 percent.

We note that we can reuse the InSphere computations in point location in performing the walk. Consider a single step of the walk that goes from the tetrahedron $t_1 = \{a, b, c, v_1\}$ to tetrahedron $t_2 = \{a, b, c, v_2\}$, whose spheres, in tuple representation, are S_1 and S_2, respectively. Suppose that we want to test p against the plane through $\operatorname{aff}\{a, b, c\}$. Let q_1 and q_2 be the lift coordinates of v_1 and v_2. An equation for H is $q_1(S_2) - q_2(S_1)$. Therefore, the sign of the orientation determinant of p with respect to H can be computed by computing the weighted difference of the two already-computed InSphere values: $q_1(S_2 \cdot p) - q_2(S_1 \cdot p)$. This is cheaper than performing the determinant computation with $\{a, b, c\}$ and p. When the spheres of t_1 and t_2 are identical, i.e., S_1 and S_2 differ only by a multiple, we have a degenerate configuration and have to perform the determinant computation, but this happens rarely enough that TESS3 simply chooses the side randomly.

To order a set of points with a Hilbert curve, TESS3 subdivides a bounding cube into $(2^i)^3$ boxes and reorders the points using counting sort on the index of the box on the Hilbert curve that contains each point. Points in a box can be reordered recursively until the number of points in each subbox is small. Parameter i is chosen large enough so that few recursive steps are needed, and small enough that the permutation can be done in a cache-coherent manner. We find that having $(2^3)^3 = 512$ boxes works well; ordering 1 million points takes between 1–2 seconds on common desktop machines.

4.4.3 Accuracy limitations

Since TESS3 uses floating point arithmetics to evaluate predicates, there is no guarantee that the output is a true Delaunay triangulation. In particular, in PDB data, the coordinates have 16 bits—more than the 10 bits that can be computed exactly with double precision floating point arithmetic. We therefore decide to measure the amount of errors in the output, by running TESS3 against all the 20393 files present in the PDB at the time of the experiment. We measure two types of errors: the percentage of tetrahedra that are not positively oriented and the percentage of tetrahedra whose spheres are not empty. We find that, of all these files, there is only one for which TESS3 produced errors, namely the PDB file 1H1K, shown in Figure 4.4. In the tessellation for 1H1K, 266 tetrahedra, out of the total 263K, have non-empty sphere. We can see in the figure that our assumption of even distribution is egregiously violated. This violation is explained by the comments in the 1H1K file: "This entry corresponds to only the RNA model which was built into the blue tongue virus structure data. In order to view the whole virus in conjunction with the nucleic acid template,

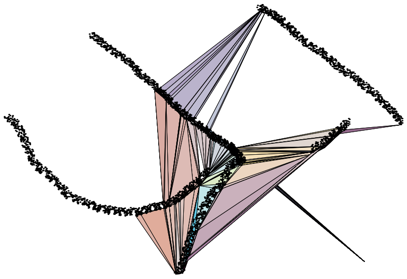

Figure 4.4: The atoms from the PDB file 1H1K. The tetrahedra drawn are those whose spheres are not empty.

To further study the errors in the output of TESS3, we generate random input points with fixed number of bits and measure the amount of errors in the output in relation to the number of input coordinates bits. The findings are reported in table 4.1. From this table, we observe that the bit number threshold beyond which TESS3 produces large number of Delaunay errors is 16, but the percentage of errors observed by the floating point auditing is quite small, even when the bit number is large. We also observe that a large portion of the errors in the incorrect output would not be detectable if the auditing is implemented with floating point arithmetic. Perhaps a triangulation that can be viewed as Delaunay by floating point arithmetic can be useful for applications that do not require the exact output.

# of bits		≤ 16	17	18	19	20	21	22	23
exact	Delaunay	0	0.1	15.5	81.9	96.7	98.9	99.2	99.3
	orientation	0	0	0	0	0	0	0	0
float	Delaunay	0	0	0	0	0	0	0	0
	orientation	0	0	0	0	0	0	0	0

Table 4.1: The percentage of Delaunay tetrahedra with neighboring points inside or on their spheres using 100K random points as input.

We investigate how much ordering points along a Hilbert curve and bit-leveling helps speed up tess3 and make it more resistant to numerical problems. Figure 4.5 shows a log-log plot of the percentage of InSphere tests that contain round-off errors with three different orderings: random, Hilbert ordering only, and Hilbert ordering combined with bit-leveling. The percentages of errors are affected by both the number of coordinate bits and the number of points in the input; the plot illustrates variations in both of these controls. Given an input with a certain number of coordinate bits, we can see that the combined ordering has the lowest amount of numerical errors—and the difference becomes more dramatic as the number of input points increases.

In figure 4.5, we show a log-log plot of the percentage of InSphere tests that contain round-off errors with three different orderings: random, Hilbert ordering only, and Hilbert ordering combined with bit leveling. The percentages of errors are affected by both the number of coordinate bits and the number of points in the input; the plot illustrates variations in both of these controls. Given an input with a certain number of coordinate bits, we can see that the combined ordering has the lowest amount of numerical errors—and the difference becomes more dramatic as the number of input points increases. We should emphasize that the InSphere errors here are observed

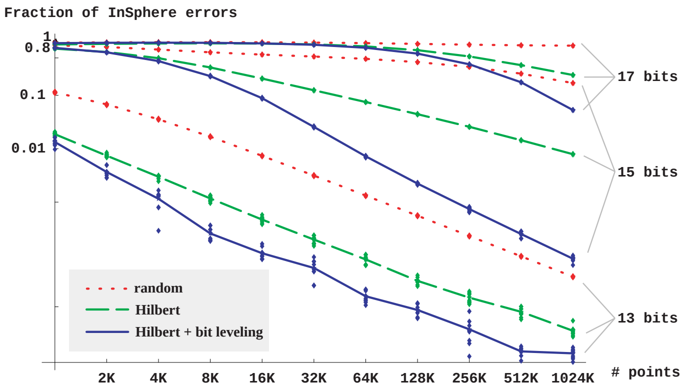

Figure 4.5: Semilog plot showing percentage of InSphere tests with round-off errors by number of points n and number of coordinate bits, for three orderings. We plot a dot for each of 10 runs for given n and bit number, and draw lines through the averages of 10 runs.

during the incremental construction, not the errors reported by the auditing of the output, which do not appear until the number of coordinate bits reaches 17.

4.5 Comparison of five Delaunay triangulation programs

We compare the implementations of five 3D triangulation programs:

- CGAL (Boissonnat et al., 2002; Devillers, 1998) is a C++ geometric algorithm library that includes a DELAUNAY_TRIANGULATION_3 class that encapsulates functions for Delaunay triangulation. It also supports vertex removal (Devillers and Teillaud, 2003).

- HULL by Clarkson (1992) is a convex hull program for dimension up to four, which inclues routines for computing Delaunay triangulations.

- QHULL (Barber et al., 1996), initially developed at the geometry center of University of Minnesota, is a popular program for computing convex hulls in arbitrary dimensions.

- PYRAMID by Shewchuk (1998a) is designed to construct triangulation of a solid shape. In addition to taking points as input, it also can take line segments and triangles and compute a *constrained Delaunay triangulation* which include the line segments and triangles as faces.

- TESS3 is engineered to work with protein molecular data. Its implementation is described in Section 4.4.

There are many other programs that can compute the Delaunay triangulation: DELTRI by Edelsbrunner (1994), NNSORT by Watson (1981; 1992) and Proshape (Koehl et al., 2002), to cite just a few. The above programs are selected because of their comparable speeds and their interesting implementation choices. All of these programs constructs a Delaunay triangulation incrementally, as described in Section 4.1, but make different implementation choices regarding point location, update, and predicates. In Section **??**, we compare the implemetation choices of the programs, ending with a summary table that allow side-by-side comparisons. In Section **??**, we compare the performance of these program.

The speed of computer programs in general are strongly affected by whether they make *coherent memory references*: A sequence of memory references are coherent if adjacent ones in the sequence reference nearby memory addresses. Therefore, randomizing the input points, as often done by a theoratically optimal algorithm to defeat worst time cases, often lead to poor performances. This observations means that, for Delaunay triangulations, inserting points in a spatially coherent manner—by ordering them along a space filling curve, for example—often improves the speed of the program. Amenta, Choi and Rote (2003a) study how their Biased Randomized Insertion Order (BRIO) preserves enough randomness in the input points so that the performance of a randomized incremental algorithm is unchanged but orders the points by spatial locality to improve memory coherence. In Section 4.5.2, we compare how their ordering improve the speed of Delaunay triangulation against other ordering such as using space filling curves.

4.5.1 Implementation Comparison

For representation of Delaunay triangulations, all five program use variations of the basic representation discussed in Section **??**, i.e., they store the set of tetrahedra and their neighboring relations. The difference of the program are how they refine the neighbor relations to establish correspondence between vertex references—or *corners*—of neighboring tetrahedra. For two neighboring tetrahedron $T = F \cup \{v\}$ and $T' = F \cup \{v\}'$, vertices v and v' are the *opposite corners* of T and T'. Knowing the opposite corners of neighboring tetrahedra allow their shared triangle to be immediately recovered. It is also useful to know, for operations such as cycling around an edge, how the rest of the corners of T and T' correspond, i.e. given vertex v in the shared triangle F, we want to know which corner in T and T' references v. PYRAMID and TESS3 have special ways to do this: PYRAMID stores four bits with each opposite corner pointer to indicate the orientation of the neighboring tetrahedron and location of the vertices of the shared triangle. Tess3 uses a representation that is a refinement of the structure of Paoluzziet al. (Paoluzzi et al., 1993) or Kettner et al. (2003): Each tetrahedra stores its vertex references as an array in lexicographic order, except that the first two may be swapped to keep the orientation positive. The correspondence between vertices in neighboring tetrahedra, where vertex $0 \leq i < 4$ is replaced by vertex at position $0 \leq j < 4$, can be recorded in a table indexed by i, j.

The boundary triangles in a Delaunay triangulation cause special cases to appear. All the programs, except, PYRAMID, handle the boundary by using a *point at infinity* e, so that for every triangle $\{a, b, c\}$ on the boundary, there is a tetrahedron $\{a, b, c, e\}$.

In theory, point location is not the bottleneck for devising optimal 3D Delaunay algorithms. In practice, however, the size of the neighborhood updated by inserting a new point is close to constant, and point location to find the tetrahedron containing a new point p can be more costly than updating the triangulation if not done carefully.

HULL and QHULL implement the two standard ways to perform point location in randomized incremental constructions of the convex hull: Hull maintains the history of all simplices, and searches the history DAG to insert a new point. QHull maintains a conflict list for each facet of the convex hull in the form of an *outside set*, which is the set of points yet to be processed that can "see" the facet. These are equivalent in the amount of work done, although the history dag is larger, and the conflict list requires that all points be known in advance.

CGAL implements the Delaunay hierarchy scheme invented by Devillers (2002). It combines a hierarchical point location data structure with the remembering stochastic walk. The Delaunay hierarchy first creates a sequence of levels so that the 0th level is P, and each subsequent level is produced by random sampling a constant fraction of the points from the previous level. Next, Delaunay triangulation is created for each level, and the tetrahedra that share vertices between levels are linked. To locate p, at each step, a walk is performed within a level to find the vertex closest to p. This vertex is then used as the starting point for the next step. The hierachical triangulation makes the asymptopic point location time to be $\emptyset(\log(n))$, which is optimal, while the walk, along with appropriately chosen parameter for the sizes of the levels, allow the space used the data structure to be small.

PYRAMID uses the *jump-and-walk* introduced by Mücke et al. (1996). To locate p in a mesh of m tetrahedra, it measures the distance from p to a random sample of $m^{1/4}$ tetrahedra, then walk from the closest of these to the tetrahedron containing p. Each step of the walk visits a tetrahedron t, shoots a ray from the centroid of t towards p, and go to the neighboring tetrahedron intersected by the ray. In the worst case, this walk may visit almost all tetrahedra, but under some uniformity assumptions the walk takes $O(n^{1/4})$ steps, which is an improvement over $O(n^{1/3})$ steps that a walk would have required without the initial sampling.

Contrasting the asymptotic behavior of the Delaunay hierarchy and the jump-and-walk, we should point out that the difference between $(n^{1/4})$ and $\log(n)$ is small for practical value of n; the Delaunay hierarchy, however, makes no assumption about the point distribution.

CGAL has many options for evaluating predicates exactly. It can use interval arithmetic (Pion, 1999), without or with static filtering (Devillers and Pion, 2003) or an adapted filtering that guarantees correctness for integers of no more than 24 bits. We list these options in increasing speed, though static filtering is usually recommended because it makes no assumption about the input and is still quite competitive in speed.

HULL uses a low bit-complexity algorithm for evaluating the sign of an orientation determinant that is based on Graham-Schmidt orthogonalization. The idea is that since we care only about the sign of the determinant, we can manipulate the determinant so far as its sign does not change. The implementation uses only double precision floating operations and is able to compute the signs of InSphere determinants exactly for input whose coordinates have less than 26 bits.

PYRAMID uses multilevel filtering (Shewchuk, 1996a) and an exact arithmetic to implement its geometric tests.

QHULL and TESS3 use floating point operations exclusively, and are written so that they do not crash if the arithmetic is faulty, but they may compute incorrect structures. Qhull checks for structural errors, and can apply heuristics to repair them in postprocessing. TESS3 assumes that input points have limited precision and are well distributed, and uses bit-leveling and Hilbert curve orders to try to reduce the precision needed to evaluate predicates (see Section 4.4 for details.) Since a tetrahedron's sphere can be used repeatedly in the InSphere predicate, TESS3 stores spheres so that the InSphere predicate can be expressed as a dot product instead of a determinant.

The update step in the Delaunay triangulation can be performed either in the Bowyer-Watson style or with flipping. All the programs, except PYRAMID, chooses the

Bowyer-Watson style. TESS3 uses Bowyer-Watson update because have observed in our experiments that flipping assigns neighbor pointers to twice as many tetrahedra, since many tetrahedra created by flips with a new vertex p are almost immediately destroyed by other flips with p.

To handle degeneracies, QHULL allows the user to select a policy when the input contains degeneracies or the output contains errors: either it perturbs the input numerically and tries again, or it attempts to repair the outputs with some heuristics. CGAL uses the vertical perturbation scheme (Devillers and Teillaud, 2003). The perturbation of a point is determined by its index and guarantees that there are no flat tetrahedra. TESS3 also perturbs the points vertically, but the perturbation is implicit.

Program	flip?	point location	exact?	caching spheres?	degeneracy	prog. lang.
CGAL (version 2.4)	no	Delaunay hierarchy	yes	no	perturbing E^3	`C++`
HULL (obtained in 3/2004)	no	history DAG	yes	yes	perturb points into hull in E^4	`C`
PYRAMID (obtained in 3/2004)	yes	jump-and-walk	yes	no	Perturb points into hull in E^4. Remove flat tetrahedra by post-processing	`C`
QHULL (version 2003.1)	no	outside set	no	no	Perturb points into hull in E^4. Remove flat tetrahedra by post-processing.	`C`
TESS3 (last revised in 9/2003)	no	Hilbert ordering, zig-zag walk	no	yes	Perturbation in E^4 with no flat tetrahedra.	`C`

Table 4.2: Program comparison summary.

4.5.2 Performance comparison

In this section we report on experiments running the five programs on randomly generated points and on PDB files. We first report on running time. Then, because TESS3 uses only standard floating point arithmetic, we report on the (small number of) errors that it makes.

We have tried to use the latest available codes of these programs. HULL and PYRA-

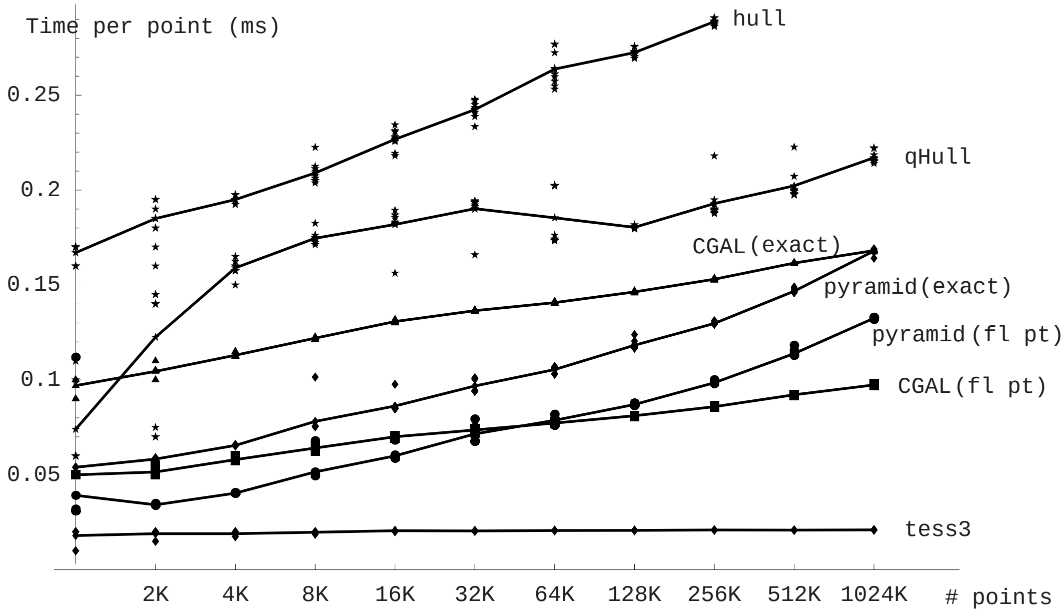

Figure 4.6: Running time of the programs with 10 bit random points.

MID codes were given to us by the authors. CGAL and Qhull codes were downloaded from their web sites. The latest version of CGAL in April, 2004 is 3.0.1; however, we found that it is more than two times slower than CGAL 2.4 due to compiler issues. (Sylvain Pion, an author of the CGAL code, has found a regression in the numerical computation code generated by `gcc` that probably explains the slow-down.) We therefore proceed to use CGAL 2.4. Qhull 2003.1 we used is the latest version.

The plots in Figures 4.6 and 4.7 show the running time comparisons using random data and PDB data as input, respectively, using a logarithmic scale on the x axis and the running time per point in micro-seconds on the y axis. Hull's running time is much slower than the rest of the programs, with time per point between 0.4–0.6 ms. In Figure 4.7, we omitted it so other plots can be compared more easily. The timings are performed on a single processor of an AMD Athlon 1.4GHZ machine with 2GB of memory, running Red Hat Linux 7.3. Using time per point removes the expected linear trend and allows easier comparison across the entire x-coordinate range. Lines indicate the averages of ten runs; individual runs are plotted with markers. We should also mention that CGAL's running time seems to be affected most by compiler changes, with the slowest as much as 2.5 times slower than the fastest (the timing plots are produced with a version that is roughly 1.5 times slower than the fastest we have seen).

We generated random data by choosing coordinates uniformly from 10-bit non-negative integers. This ensures that the floating point computations of both Qhull and

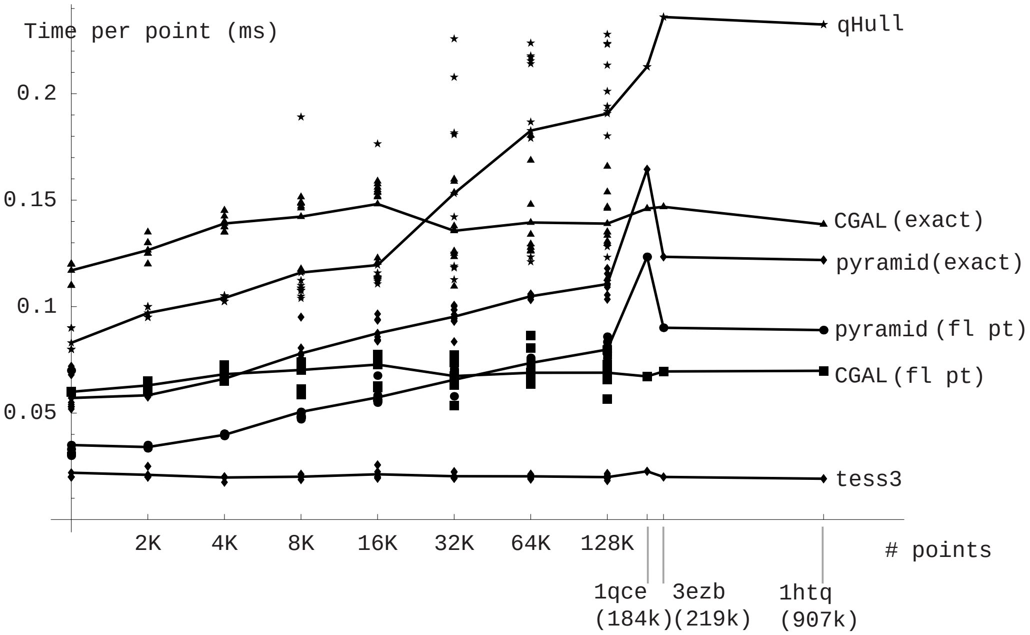

Figure 4.7: Running time of the programs with PDB files.

tess3 are correct. For the PDB data, for each input size n that is indicated on the x-axis, we try to find 10 files whose number of atoms are closest to n, though there is only one (with the indicated name) for each of the three largest sizes. We have posted online (Liu and Snoeyink, 2005b) the names of these PDB files, as well as the program used to generate the random data.

There are a few immediate conclusions: The ordering of programs, tess3 < CGAL (fp) < pyramid (fp) < pyramid(ex) & CGAL < Qhull < hull, is consistent, although hull is particularly slow with the PDB files in comparison and is therefore not shown. In Figure 4.6 and 4.7, we can see a clear penalty for exact arithmetic, because even when an exact arithmetic package is able to correctly evaluate a predicate with a floating point filter, it must still evaluate and test an error bound to know that it was correct. Time per point shows some increase for everything but CGAL and tess3, which we believe is due to point location.

To further explain the difference in these programs' running time, we used the `gcc` profiler to determine the time-consuming routines. There are caveats to doing so; function level profiling turns off optimizations such as inlining, and adds overhead to each function call, which is supposed to be factored out, but may not be. (This affects CGAL the most, with its templated `C++` functions we could not get reasonable profiler numbers, so we also tried to time its optimized code, but this has problems with clock

	total created spheres/tetra	MakeSphere (μs)	InSphere (μs)		Update (μs)	Point Location		Memory (MB)
			fl. pt.	exact		fl. pt.	exact	
CGAL (2.4)	2,760,890	–	0.06^p 0.24^t	18.5^p 1.72^t	0.1^p 16.1^t	$21.8\%^p$ $22.1\%^t$	$25.3\%^p$ $27.9\%^t$	39
HULL	2,316,338	10.02	0.14	–	2.40	–	73.1%	401
PYRAMID	$5{,}327{,}541^f$ $2{,}662{,}496^n$	-	0.21	0.72	2.44	50.2%	38.1%	57
QHULL (2003.1)	2,583,320	0.65	0.12		> 4.39	9.0%	–	172
TESS3	2,784,736	0.13	0.04	–	2.42	$3.88\%^h$ $0.43\%^w$	–	77

Table 4.3: Summary of timings and memory usage, running the programs against the same 100k randomly generated points with 10 bit coordinates. Notes: For pyramid tetrahedra creation, numbers marked f include all initialized by flipping and marked n include only those for which new memory is allocated—equivalently, only those not immediately destroyed by a flip involving the same new point. For CGAL timings, p indicates profiler and t direct timing. For tess3 point location, h includes the preprocessing to order the points along a Hilbert curve; w is walk only.

resolution.) The table shows some of our findings for running the programs against the same 100k randomly generated points with 10 bit coordinates.

The "total created spheres/tetra" column shows that flipping must initialize many more tetrahedra. "MakeSphere" and "InSphere" columns, which record time to make sphere equations and test points against them, indicate that there are speed advantages to using native floating point arithmetic for numerical computations. Even simple floating point filters must check error bounds for computations. Note that for the programs that do not cache spheres, the InSphere test is a determinant computation. The "Update" column indicates the time to update the tetrahedral complex and does not include any numerical computation time. The "Point Location" column indicates the percentage of time a program spends in point location (for tess3, this number includes the time for sorting the points along the Hilbert curve). The "Memory" column indicates the total amount of memory the programs occupy in the end.

As we can see from the table, tess3 benefited particularly from its fast point location. Caching sphere equations also helped speed up the numerical computation. A version of tess3 that does not cache sphere equations is about 20 percent slower. We observed some bottlenecks of the other programs: QHULL's data structure is expensive to update and the code contains debugging and option tests; Hull's exact arithmetic incurs a significant overhead even when running on points with few bits; PYRAMID was bogged down mainly by its point location, which samples many tetrahedra.

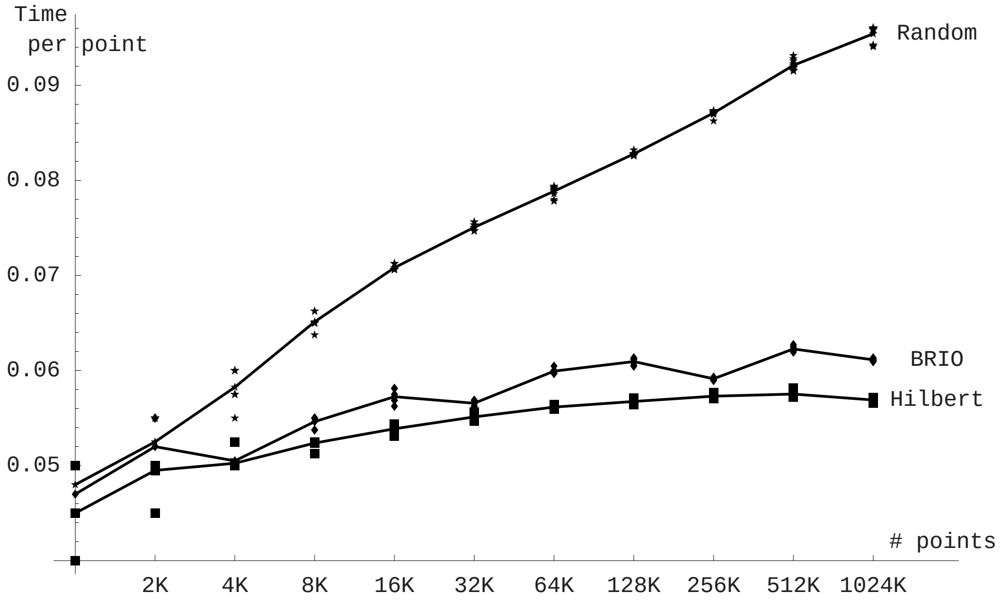

Figure 4.8: Running time of the CGAL Delaunay hierarchy using random, BRIO and Hilbert point orders.

We close by comparing BRIO insertion order with a Hilbert curve order. A BRIO order first partitions the input points into $O(\log n)$ sets as follows: Randomly sample half of the input points and put them into the first set; repeatedly make the next set by randomly sampling half of the previous set. Order the sets in the reverse order they are created. Finally, the points within each set are ordered by first bucketing them with an octree and traverse the buckets in a depth-first order. Figure 4.8 compares the running times of CGAL, which uses a randomized point location data structure, under the BRIO and Hilbert insertion orders. The Hilbert curve is faster on average and has a smaller deviation. This suggests that for input points that are uniformly distributed, adding randomness into the insertion ordering perhaps will only slow down the program.

4.5.3 Conclusion

We have surveyed five implementations of 3D Delaunay triangulation and compared their speed on PDB files and randomly generated data. The experiments show that

Hull and QHull, the two programs that solve the more general problem of convex hull construction in 4D, are slower, penalized by not doing point location in 3D. Amongst the other three programs, tess3 is the fastest because its point location is carefully engineered for input points that are uniformly distributed in space. Exact arithmetic with filtering is quite efficient, as demonstrated by CGAL and PYRAMID, but still incurs an overhead. We show that it is possible to have an implementation that works well even when straightforward bit-complexity analysis suggests otherwise.

4.6 Computation of Delaunay Diagrams on Points from 4d Grids

Time-varying volume data, from scientific computations or engineering simulations, are produced on space-time grids that may range in size from $30^3 \times 150$ to $1024^3 \times 1000$ or larger. To visualize the data, one may wish to use the following computational pipeline that invokes the Delaunay triangulation: sample the grid points irregularly, perhaps using more samples in regions with special interests; Delaunay triangulate the sample locations to interpolate the samples; finally, for display, extract isosurfaces from the triangulation.

In order to use the Delaunay triangulation, the *degeneracies* in the input must be handled. A common approach to handle the degeneracies is to perturb the input sites infinitesimally so that all degeneracies disappear. An alternative approach, one that we prefer, is to compute the *Delaunay diagram* exactly without perturbation. Because the cells in a Delaunay diagram are generally not simplicial, they must be further triangulated to support piecewise linear interpolation. For example, they can be triangulated with barycentric subdivision, which is particularly simple and symmetrical. At first, the requirement for “post-processing” non-simplicial cells seem to be a shortcoming of computing Delaunay diagrams. However, it also can be seen as an advantage: instead of letting the triangulations of the cells be induced by an perturbation policy, the user directly control the triangulations. When the input contains many degeneracies, we have another advantage: The size of the Delaunay diagram is much smaller than that of the Delaunay triangulation of the perturbed sites. The sites from time-varying volume data are highly degenerate, because they contain many sets of points sampled from the corners of a grid cell, which are cospherical. Figure 4.9 demonstrates this by plotting the size of the the Delaunay diagram and the size of the Delaunay triangulation over a sequence of input; the input sequence is produced by randomly sampling an increasing fraction of a 10 × 10 × 10 × 10 grid in order to mimic the time varying volume data. Given the degenerate characteristics of time-varying volume data, we are motivated to develop techniques to compute the Delaunay diagrams in four dimensions. In Section 4.6.4, we give a simple incremental algorithm for constructing the Delaunay diagram that maintains only the cells and their neighboring relations.

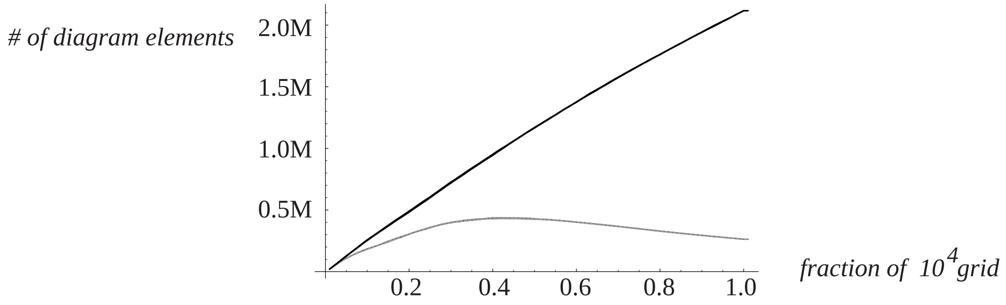

Figure 4.9: Growth in the number of data structure elements against the number of points inserted in randomized incremental construction for the Delaunay diagram (gray) and Delaunay triangulation (black). The number of data structure elements is the sum of cells, faces and vertex references

Our geometric predicates are based on representation of spheres as tuples as opposed to determinants. In d dimensions, our predicates and sphere construction use $O(d)$ arithmetic operations, which compares favorably with the $O(d!)$ arithmetic operations used for computing determinants. In four dimensions, the actual running time advantage is not significant. Nonetheless, we choose to use the sphere-based approach because it simplifies the implementation. We describe our sphere construction in Section 4.6.3.

We have implemented our algorithm in C++. In Section 4.6.5, we show experimental results with our program.

4.6.1 Preliminaries

We review Delaunay diagrams and representations of points and spheres.

The Delaunay diagram is a collection of convex hulls, defined with empty spheres. These convex hulls form what is known as a cell complex (Boissonnat and Yvinec, 1998, page 245) and the convex hulls in a cell complex are called faces. The set of faces of a cell complex has the property that the intersection of any subset belongs to the set. Faces of the highest dimension are called *cells*; faces of the second highest dimension are called *facets*; and faces of the third highest dimension are called *ridges*. The faces of a Delaunay diagram of a finite set of sites partitions the convex hull of the sites. A facet in the Delaunay diagram is always the intersection of two *neighboring* cells, except for those on the boundary. To avoid the boundary, it is convenient to place a sentinel point infinitely far away from the sites and define "infinite spheres" through it so that the Delaunay diagram partitions the whole space. For the sentinel point, we choose to use the *faraway point* that we have introduced earlier Liu and Snoeyink (2006a). The faraway point requires no special treatment in the code other than one conditional in the point-sphere sidedness test.

To represent a point, we use lifting Brown (1980) and homogenizing Riesenfeld (1981). We represent a point p by a $(d+2)$-vector $[p_0, \ldots, p_{d+1}]$, where $[p_1, \ldots, p_d]$ are the Cartesian coordinates, $p_0 = 1$ is the homogenizing coordinate, and $p_{d+1} = \sum_{i=1}^{d} p_i^2$ is the lifting coordinate.

Delaunay triangulation often implements the point-sphere sidedness test with the Orientation determinant and the InSphere determinant. Given a simplex represented with by a tuple of vertices, $v^0, \ldots, v^d$, the Orientation determinant computes the signed volume of the simplex; if the sign is positive, the simplex is positively oriented. Given a positively oriented simplex, the sign of the InSphere determinant gives the sidedness of a point p with respect to the sphere of the simplex.

$$\begin{array}{ll} \text{Orientation}(v^0, \ldots, v^d) & \text{InSphere}(v^0, \ldots, v^d; q) \\ := \begin{vmatrix} v^0_0 & \cdots & v^0_d \\ \vdots & \ddots & \vdots \\ v^d_0 & \cdots & v^d_d \end{vmatrix} & := \begin{vmatrix} v^0_0 & \cdots & v^0_d & v^0_{d+1} \\ \vdots & \ddots & \vdots & \vdots \\ v^d_0 & \cdots & v^d_d & v^d_{d+1} \\ p_0 & \cdots & p_d & p_{d+1} \end{vmatrix} \end{array}$$

We implement the point-sphere sidedness test more directly using spheres. A sphere S is represented by a $(d+2)$-vector such that $S \cdot p = 0$ for all points on the sphere, with the sign chosen so that $S \cdot p < 0$ for all points inside the sphere. Any positive scalar multiple represents the same sphere. Planes can be considered and represented as special spheres whose last entry is zero.

A sphere can be constructed from a tuple of positively oriented $d+1$ affinely independent points $(v^0, \ldots, v^d)$ by using the minors of the InSphere determinant $\text{InSphere}(v^0, \ldots, v^d; x)$ with respect to a symbolic point x.

Alternatively, we can construct spheres from spheres by linear combinations. The affine combination of two spheres, $\lambda S_1 + (1-\lambda)S_2$ for real λ, is a one parameter family of spheres called a *pencil of spheres.* The linear combination of two spheres, $\beta S_1 + \gamma S_2$ for reals β and γ that do not sum to zero, is actually the same pencil—since scalar multiples of a tuple represent the same sphere, we can set $\lambda = \beta/(\beta+\gamma)$ and $1-\lambda = \gamma/(\beta+\gamma)$.

When two distinct spheres, S_1 and S_2, intersect in a lower-dimensional sphere $C = \{p \mid S_1 \cdot p = 0, S_2 \cdot p = 0\}$, then all spheres in their pencil contain C. Each point $v \notin C$ is on a unique (up to a scalar multiple) sphere S_v in the pencil:

$$S_v = (S_1 \cdot q)S_2 - (S_2 \cdot q)S_1. \tag{4.1}$$

It is easily checked that $S_v \cdot v = 0$, and that $S_v \cdot p = 0$ for all $p \in C$.

Let e be the $n+1$-vector $[0, \ldots, 0, 1]$. The plane that contains the intersection between two spheres is

$$P_{12} = (S_1 \cdot e)S_2 - (S_2 \cdot e)S_1. \tag{4.2}$$

4.6.2 Arithmetic complexity and algebraic degree

Computers perform computations on integers with limited precision. Therefore, it is important to analyze the largest integer required—measured in bits—to perform arithmetic operations. In particular, with Delaunay computation, we assume that the point coordinates are represented by b-bit integers and want to know how large the numbers in our arithmetics can grow as a function of b, and d, the dimension.

Liotta et al. Liotta et al. (1997) gave a simple algebra for estimating the precision of a computation by tracking the degree of the polynomials involved. Recall the representation of a point p by a $d+2$ tuple, $p_{d+1} = \sum_{i=1}^{d} p_i^2$. The degree of these coordinates are $[0, 1, \ldots, 2]$. When computed from determinants of point coordinates, spheres have degrees $[d+2, d+1, \ldots, d+1, d]$, and planes have degrees $[d, d-1, \ldots, d-1, 0]$. The expression $S \cdot p$ is therefore a degree $d+2$ polynomial when S is a sphere and degree d when S is a plane. Each term of the expression could be evaluated with db-bit integers. Potential carries from summing $d^{O(1)}$ terms may require additional $O(\lceil \log_2 d \rceil)$ bits.

4.6.3 Sphere construction

For Delaunay diagram or triangulation algorithms, the most important geometric predicate is the point-sphere sidedness test. The common way to implement the test is to use the InSphere determinant. We avoid using any determinants: given a point p to test sidedness against a sphere S, we compute the dot product $S \cdot p$; to compute spheres, we use equation 4.1, which only have dot products. In d dimensions, our sphere computations use $O(d)$ arithmetic operations, compared with the $O(d!)$ arithmetic operations used by a InSphere determinant. There is also an implementation advantage of our sphere-based approach: For direct computation of Delaunay diagrams, the InSphere determinant requires finding $d+1$ affinely independent vertices of a cell, but this is expensive when a cell has many vertices; also, ordering the $d+1$ vertices to give positive orientation determinant complicates the code.

The caveat of the sphere-based approach is that a new sphere, constructed using equation 4.1 takes twice the number of bits to represent as an old sphere. Repeated applications of equation 4.1 therefore would exponentially increase the storage size for the sphere vectors. However, we know that, when computed from determinants of point coordinates, the components of the spheres are polynomials of fixed degree: $[d+2, d+1, \ldots, d+1, d]$. Therefore, the components of a sphere computed using equation 4.1 must have some common scalar multiple that can be factored out. For points in general position, we derive a formula for computing this factor, as stated in Theorem 4.6.1. The formula uses only a dot product of an old sphere and a point, which requires $O(d)$ arithmetic operations. Unfortunately, if there are degeneracies, Theorem 4.6.1 cannot be used for finding the common factor. Therefore, more expensive procedures, such as Euler's GCD algorithm, have to be used.

Theorem 4.6.1. *In E^d, suppose that S_a and S_b are circumspheres of neighboring simplices with common vertices f and opposite vertices a and b such that $S_a \cdot b \neq S_b \cdot a$. Let p be any point. The expression $S_p = ((S_a \cdot p)S_b - (S_a \cdot p)S_b)/V$, where $V = (S_b \cdot a) = (S_b \cdot a)$, is the sphere computed from the points f and p using the InSphere determinant.*

Proof. This is immediate from the determinant identity in Lemma 4.6.4. □

4.6.4 Incremental construction of 4d Delaunay diagrams

We describe an algorithm to construct a Delaunay diagram incrementally. The algorithm is based on Edelsbrunner's beneath-beyond method Edelsbrunner (1987). However, we simplify it for the special case of four dimensional Delaunay diagram construction, maintaining no more data structures than for typical Delaunay triangulation algorithms. We do not analyze our algorithm but will show later actual running times with our implementation.

The algorithm maintains a cell complex, representing each cell as a set of vertices and each facet as an arc in the *neighbor graph*: the nodes are the cells; each arc is between two cells that have the same facet.

The cell complex is initialized by constructing the simplex of six site–the sentinel faraway point and five points that are affinely independent.

To insert a point p , we perform the procedures listed below. Using a terminology from Edelsbrunner Edelsbrunner (1987), we let the cells have *colors* with respect to p: A cell is red, green, or blue if its sphere has p inside, on or outside.

- Locate finds a red cell by performing a walk in the neighbor graph. The walk visits one cell c_1 at each step and proceeds to a neighboring cell c_2 such that p and c_2 are on the same side of the plane through the facet between c_1 and c_2. The walk always terminates by the acyclic theorem from Edelsbrunner Edelsbrunner (1989).

- Search finds all the red cells. Starting from the cell located by the locate procedure, a depth first search of the neighbor graph is performed. A branch of the search terminates when a blue cell is found. Each pair of red-blue or red-green cells identifies a horizon facet.

- Update deletes all red cells and create new cells. Two kinds of new cells are created: For each yellow cell, a new cell is created by inserting p into the vertices of the yellow cell; for each horizon facet between red-blue cells, a new cell is created by instantiating a new vertex set that includes p and the vertices of the horizon facet.

- Connect updates the neighbor graph. The new neighboring relations are either between the old and the new cells or amongst the new cells. The former case is easy to handle: For each horizon facet between red-blue cells, the blue cell and the new cell created from the facet become new neighbors. For the latter case, three steps are performed.

    1. Associate each horizon facet to a new cell. For each horizon facet between red-blue cells, associate it to the Delaunay cell created from it; for each of the other horizon facets, which are between red-green cells, map it to the new cell created from modifying the green cell.
    2. Compute horizon ridges—the two dimensional faces of the horizon facets. Each horizon ridge is shared by exactly two horizon facets.

3. For each horizon ridge, connect the cells associated with the two facets sharing the ridge.

Each step in the procedures above is simple except the computation of horizon ridges in the connect procedure. To compute the ridges of a horizon facet f, we choose a cell c containing f, e.g. the red cell; for each neighbors c' of c, collect the vertex set $c' \cap f$ as a ridge if their affine span is two dimensional. The last dimension check is easy because testing dimension—$\dim \operatorname{aff} c' \cap f = 2$—is the same as testing cardinality—$\#(c' \cap f) \geq 3$, as justified by Observation 4.6.2.

As an implementation detail, the point-plane sidedness test needed by the locate procedure can be implemented using spheres. Consider the neighboring cells c_1 and c_2 in the locate procedure. Let their respective spheres be S_1 and S_2. To decide whether to terminate the walk, we first compute $S_2 \cdot p$. To test p against the plane between c_1 and c_2, we use equation 4.2 and compute the sign of $(S_1 \cdot e)(S_2 \cdot p) - (S_2 \cdot e)(S_1 \cdot p)$. In this expression, $S_2 \cdot p$ is already computed; $S_1 \cdot p$ was saved from a previous walk step. Since $S_1 \cdot e$ and $S_2 \cdot e$ just extract vector components, the only additional arithmetics left are scalar multiples and additions.

Lemma 4.6.2. *For a face with vertices f in a four dimensional Delaunay diagram, if $\#f \geq 3$, the dimension of the face is at least two.*

Proof. Consider the lifted sites $\hat{f}$ from f. If their affine span is a line, then the line intersects at the unit paraboloid in five dimensions at more than two points, which is impossible. $\square$

4.6.5 Implementation and experiments

We have implemented our algorithm in `C++`. We call our program DD4. For each cell, we store its vertex and neighbor pointers in STL (Standard Template Library) vectors, to allow them to have variable length. To test and construct spheres, we use double precision floating point arithmetics. To find the common factor among the vector components of a sphere, we apply a GCD function 6 times.

As candidates for comparison, we have looked at a popular convex hull program, QHULL Barber et al. (1996), and our own program, TESS4 Kettner et al. (2003). QHULL performs Delaunay triangulation after the sites are lifted and can output Delaunay diagrams with a post-processing step that merges simplices with identical spheres. TESS4 was implemented with many similar techniques as DD4. For example, TESS4 uses a walk for point location and computes spheres for geometric predicates predicates—although, unlike dd4, the spheres are computed with determinants. The biggest difference between tess4 and dd4 is the manipulation of ridges: tess4, which maintains a triangulation, can identify a horizon ridge in constant time, while dd4 has to perform set intersections to identify a horizon ridge, which uses time proportional to the total number of cell vertices around the ridge. Therefore, we expect dd4 to perform slower than tess4 when there are few degeneracies in the input.

For input data, we randomly sample an increasing fraction of a 16^4 grid to mimic the time varying volume data. The experiments are performed on a Linux machine with Intel Pentium III 700MHz processor, with 1MB cache and 2.5GB RAM. We show the results in Table 4.4. When input contains few degeneracies, as expected, dd4 is slower than tess4. Furthermore, dd4 uses more memory because, to manage the vertex and neighbor reference for each cell, it uses STL vectors, which has overhead and often allocates twice the amount of memory than used; on the other hand, tess4 allocates fixed memory blocks for its simplices. When there are many degeneracies, the speed of dd4 catches up with tess4 because it maintains fewer cells. For comparison of memory use, we show both the process memory and the output size. For dd4, its cell representation in the output differs slightly from that for the running process: instead of using STD vectors, a cell's vertex or cell list is stored as a length followed by an array. For tess4, the cell representation in the output is identical to that used for the running process.

input (% of grid)	time (sec.)			size (MB)				
				process			output	
	Qhull	tess4	dd4	Qhull	tess4	dd4	tess4	dd4
10	21.5	2.8	10.1	39.5	14.3	16.2	6.4	5.4 (0.2)
20	69.7	6.4	19.1	72.3	27.1	26.1	12.2	9.0 (0.3)
40	245	14.0	32.2	124.8	52.5	35.0	23.7	12.4(0.6)
80	665.3	35.4	48.9	150.0	97.1	39.9	43.9	8.7 (0.9)
100	824.1	51.2	54.4	161.1	116.3	39.6	52.5	6.9 (1.0)

Table 4.4: Comparing running time and memory of Qhull and dd4 with tess4. The input for each row is a fraction of the 16^4 grid. Both the size of the process memory and the output are reported. The output includes each cell's vertex and neighbor references, stored as integers. For the output size of dd4, the number in the parenthesis is the fraction of cells that are not simplicial.

4.6.6 Conclusion

We have shown a simple algorithm to compute 4d Delaunay diagrams directly. Our implementation is competitive against a Delaunay triangulation program when there are many degeneracies. On the down side, the price paid for handling non-simplicial cells is high: we can not traverse the simplices around a ridge efficiently so the update procedure becomes the bottleneck of our program; we have to use dynamic arrays to maintain vertex and neighbor references, which adds a substantial amount of storage overhead. It will be interesting to find some representation of cell complex "in between" a minimalist data structure as we use and a full face lattice as used by many convex hull algorithms that do not assume general position.

The common way of constructing spheres, implicit in the evaluation of InSphere determinants, are based on point coordinates; we choose to construct spheres based on spheres, which is asymptotically faster with respect to dimension than the point-based approach. The main difficult for the sphere-based approach is to control the size of the sphere representation. We have tackled this difficulty by deriving a simple factor-finding formula in Theorem 4.6.1. Unfortunately, the formula is most useful when the input is in general position. It will be interesting to find alternative ways for sphere construction that do not compute full determinants but provide guarantees on the size of the spheres.

4.6.7 Factoring the expression for spheres

To show that the expression for spheres factors according to Theorem 4.6.1, we can establish two determinant identities.

Lemma 4.6.3. *Let M be a real, symmetric $d \times d$ matrix, and let e, f, g, and h be column vectors in $\mathbb{R}^d$. The following determinant identity holds for $(d+2) \times (d+2)$ matrices:*

$$\det \begin{vmatrix} M & e & h \\ f^T & 0 & 0 \\ g^T & 0 & 0 \end{vmatrix} - \det \begin{vmatrix} M & f & h \\ e^T & 0 & 0 \\ g^T & 0 & 0 \end{vmatrix} = \det \begin{vmatrix} M & g & h \\ e^T & 0 & 0 \\ f^T & 0 & 0 \end{vmatrix}.$$

Proof. Recall that a matrix determinant is a sum in which each term is product taking one entry per column according to some permutation of the rows, with positive or negative sign depending on whether the permutation has an even or odd number of transpositions.

Since our matrices have zeros in the lower right corner, any non-zero term from our determinants has one entry from each of e, f, g, and h. Let's express both sides as polynomials in $e_i f_j g_k h_l$ for indices $i, j, k, l \in \{1, 2, \ldots, d\}$, then show that the coefficients are equal. We can divide into cases based on the number of distinct indices in $\{i, j, k, l\}$.

First, consider four distinct indices, as in the term $e_1 f_2 g_3 h_4$. Note that we need $d \geq 4$ to have this case. In fact, we may assume that these are the indices in this case, since by interchanging both rows and columns we can bring the four indices $\{i, j, k, l\}$ to $\{1, 2, 3, 4\}$ without changing the value of the determinant or the symmetry of the upper left block. If $d > 4$ then the coefficients of $e_1 f_2 g_3 h_4$ in each of the three matrices of 4.6.3 has a common factor of $\det |M_{\overline{1234}}|$, which is the matrix minor formed by striking rows and columns 1 through 4. The remaining factors are 2×2 minors of the first four rows and columns of M. They can be shown to satisfy the desired equation using the symmetry of M:

$$\det \begin{vmatrix} m_{21} & m_{41} \\ m_{23} & m_{43} \end{vmatrix} - \det \begin{vmatrix} m_{12} & m_{42} \\ m_{13} & m_{43} \end{vmatrix} \tag{4.3}$$

$$= m_{21}m_{43} - m_{23}m_{41} - m_{12}m_{43} + m_{13}m_{42} \tag{4.4}$$

$$= m_{12}m_{34} - m_{32}m_{41} - m_{12}m_{34} + m_{31}m_{42} \tag{4.5}$$

$$= \det \begin{vmatrix} m_{31} & m_{41} \\ m_{32} & m_{42} \end{vmatrix}. \tag{4.6}$$

In verifying the signs, the reader will note that, in each matrix, the calculation of the coefficient for $e_1 f_2 g_3 h_4$ involves an even number of minus signs.

Second, consider terms $e_i f_j g_k h_l$ with $i = j$ or $k = l$. Note that these do not arise on the right-hand side of the equation. On the left-hand side, when $i = j$ we take minors by striking rows and columns containing e_i and f_j, and what remains, in both cases, is $\det \begin{vmatrix} M_{\bar{i}} & h \\ g^T & 0 \end{vmatrix}$. These determinants cancel, as they appear with opposite signs. When $k = l$ we obtain $\det \begin{vmatrix} M_{\bar{k}} & f \\ e^T & 0 \end{vmatrix}$ and its transpose, which also cancel with opposite signs.

Third, consider terms $e_i f_j g_k h_l$ with $i \neq j$ and $k \leq l$, but with $i = k$ or $j = l$. In either case, the leftmost determinant has no contribution. When $i = k$, the second determinant equals the minor $\det \begin{vmatrix} M_{\bar{i}} & h \\ f^T & 0 \end{vmatrix}$, which is negated and so equals the right-hand minor. When $j = l$ the second determinant is $\det \begin{vmatrix} M_{\bar{j}} & e \\ g^T & 0 \end{vmatrix}$, again negated and so equal to the transpose of the right-hand minor.

Fourth, consider terms $e_i f_j g_k h_l$ with $i \neq j$ and $k \leq l$, but with $i = l$ or $j = k$. Now the second determinant has no contribution. The first equals the right-hand determinant when $j = k$ and its transpose when $i = l$.

Every term $e_i f_j g_k h_l$ falls into one of these four cases, so the lemma is established. □

Let M be a $(d+2) \times d$ matrix whose columns are the d points defining the intersection of spheres S_a and S_b. Let us assume that $S_a = \det|Ma\mathcal{P}|$ and $S_b = -\det|Mb\mathcal{P}|$ are the canonical spheres through these points. Theorem 4.6.1 claims that $(S_a \cdot q)S_b - (S_b \cdot q)S_a$ factors into $(S_b \cdot a)$ and the canonical sphere S_q. To prove this, it is sufficient to establish the following determinant identity.

Lemma 4.6.4. *Let M be a $(d+2) \times d$ matrix, and let a, b, q and $\mathcal{P}$ be $(d+2)$ vectors.*

Proof.

$$\begin{aligned} & \det\begin{vmatrix} M & a & q \end{vmatrix} * -\det\begin{vmatrix} M & b & \mathcal{P} \end{vmatrix} \\ - & \det\begin{vmatrix} M & b & q \end{vmatrix} * \det\begin{vmatrix} M & a & \mathcal{P} \end{vmatrix} \\ = & \det\begin{vmatrix} M & a & b \end{vmatrix} * \det\begin{vmatrix} M & q & \mathcal{P} \end{vmatrix} \end{aligned}$$

.

By using the properties of determinants of matrix transpose and product, namely $\det|A^T| = \det|A|$ and $\det|A| \cdot \det|B| = \det|A \cdot B|$, it is equivalent to show that

$$\det\begin{vmatrix} M^TM & M^Tb & M^T\mathcal{P} \\ a^TM & a^Tb & a^T\mathcal{P} \\ q^TM & q^Tb & q^T\mathcal{P} \end{vmatrix} - \det\begin{vmatrix} M^TM & M^Ta & M^T\mathcal{P} \\ b^TM & b^Ta & b^T\mathcal{P} \\ q^TM & q^Ta & q^T\mathcal{P} \end{vmatrix}$$

$$= \det\begin{vmatrix} M^TM & M^Tq & M^T\mathcal{P} \\ a^TM & a^Tq & a^T\mathcal{P} \\ b^TM & b^Tq & b^T\mathcal{P} \end{vmatrix}.$$

The terms of the first two determinants that involve $a^Tb = b^Ta$ or $q^T\mathcal{P}$ cancel because they have opposite signs and multiply identical minors (formed by striking rows and columns with a or b) or transposed minors (formed by striking rows and columns with q or $\mathcal{P}$. Thus, we obtain an equivalent expression if we replace these entries with zeros.

Likewise, the terms on the left-hand side that involve a^Tq, $a^T\mathcal{P}$, b^Tq, and $b^T\mathcal{P}$ cancel with the corresponding terms on the right-hand side, since they have appropriate signs and multiply identical or transposed minors. Again, we obtain an equivalent expression if we replace these entries with zeros:

$$
\begin{aligned}
&\det\begin{vmatrix} M^TM & M^Tb & M^T\mathcal{P} \\ a^TM & 0 & 0 \\ q^TM & 0 & 0 \end{vmatrix} - \det\begin{vmatrix} M^TM & M^Ta & M^T\mathcal{P} \\ b^TM & 0 & 0 \\ q^TM & 0 & 0 \end{vmatrix} \\
= \;&\det\begin{vmatrix} M^TM & M^Tq & M^T\mathcal{P} \\ a^TM & 0 & 0 \\ b^TM & 0 & 0 \end{vmatrix}
\end{aligned}
$$

Lemma 4.6.3 establishes this equality, and completes the proof. □

Chapter 5

Streaming Delaunay Triangulations

In the previous chapter, we have seen that, when the input points follow some spatially coherent order, the speed of an incremental Delaunay triangulation improves because of faster point location times and improved memory coherence. In this chapter, we further exploit the spatial coherence in the input by enginnering a *streaming Delaunay triangulation program*, which, instead of letting the operating system decide when to swap out infrequently used memories onto disks, monitors its own memory and recycles memories that will no longer be needed by the triangulation algorithm. Doing so, on one hand, further improves the cache coherence of the program, and on the other hand, dramatically reduces the total memory claimed by the triangulation program, which enables processing of giggantic data sets too large to fit into the main memory. Furthermore, the streaming triangulation program can generate output as the input is still being read. Therefore, it can be pipelined with another program that consumes the triangulation.

In order for our streaming Delaunay triangulation program to recycle its memories, it must be able to determine whether a piece of data structure will be needed in the future. This requires the program to have knowledge of the future points that are yet to be processed. This necessiates some preprocessing of the raw input data points. We have designed a preprocessing method called *spatial finalization*, which inserts *finalization flags* into the input point sequence and performs limited amount of reordering, by taking a few passes over the data using only a small amount of memory.

The effiency of our programs—one for spatial finalization, the other for triangulation—is demonstrated with the largest 2D scattered data set we can find, namely the Neuse River Basin data, which has over 500 million points (double-precision x, y, and height coordinates). This data comes from the NC Floodplain Mapping project (`http:`

`//www.ncfloodmaps.com`), begun after Hurricane Floyd in 1999. Our programs triangulate the 11.2 GB Neuse River Basin data in 50 minutes using 132 MB of memory and output a 16.9 GB mesh. This is about a factor of twelve faster than the previous best out-of-core Delaunay triangulator, by Agarwal, Arge, and Yi (2005).

5.1 Previous works

The problem of Delaunay triangulating large data sets has been addressed in a number of ways. Some focus on improving the memory use of traditional Delaunay triangulation programs: Blandford et al. (2005) describe data structures for dynamically maintaining compressed triangulations in two or three dimensions, thereby increasing the size of triangulation that fits in memory by a factor of three to five; Amenta, Choi, and Rote (2003b) design a point ordering called *biased randomized insertion order* (BRIO) for incremental Delaunay triangulation that is random enough that the algorithm is still worst case optimal but is biased enough—towards spatial coherence—-that the resulting program has good memory coherence. Other work design algorithms that explicitly use disks. These algorithms, called *external memory algorithms*, assume that the memory comprise a bounded but fast RAM and a slow but unbounded disk storage and try to minimize the number of disk accesses. The first such algorithm for Delaunay triangulation is the divide-and-conquer algorithm by Crauser et.al (2001). Their algorithm is subsequently modified by Agarwal et.al (2005) to compute constrained Delaunay triangulations, who also give an implementation that gives the previous fastest performance for Delaunay triangulating large data sets.

Let us review Agarwal at al.'s result. Given a set of n points, a machine whose RAM is of size M and disk blocks are of size B, Agarwal et al.'s algorithm runs in $O((n/B)\log_{M/B} n/B)$ disk operations. This running time can be translated as the time of taking a small number of scans of the input file on disks, because $\log_{M/B} n/B$ is small for reasonably large M and $O(n/B)$ is the time to scan the input file once. In fact, Agarwal et al. point out that, because for realistic M, $\log_{M/B} n/B$ is no more than two, the algorithm can be simplified so that it takes precisely three scans: The first scan sample a fraction of the input points, small enough so that its Delaunay triangulation $\mathcal{D}$ can be computed in the main memory; the second scan distribute the rest of the input points over the edges of $\mathcal{D}$; finally, on the third scan, for each edge of $\mathcal{D}$, its distribution of points are triangulated and a certain subset of them are certified to be in the final output.

The main differences between our approach and Argawal's are that, first, for computing Delaunay triangulations, our program only takes a single pass over the input points and theirs take three; second, our program uses the disks only for input and output while theirs use the disks to store intermediate data structures.

5.2 Spatial finalizing data

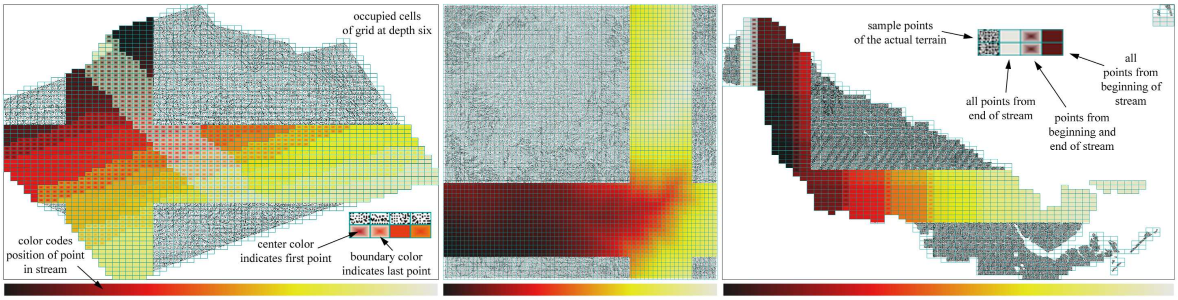
occupied cells
of grid at depth six
color codes
position of point
in stream
center color
indicates first point
boundary color
indicates last point
sample points
of the actual terrain
all
points from
beginning of
stream
all points from
end of stream
points from
beginning and
end of stream

Figure 5.1: Three terrain data sets: the 6 million-point "grbm" (LIDAR data of Baisman Run at Broadmoor, Maryland, left), the 67 million-point "puget" (middle) and the 0.5 billion-point "neuse" data set (right). Colors illustrate spatial coherence in selected grid cells: each cell's center is colored by the time of its first point, and each cell's boundary is colored by the time of its last point, with time increasing from black to white along the color ramp (bottom).

Real-world huge data sets are usually spatially coherent—if not, the programs that created them would have been slowed down. In Figure 5.1, we examine the spatial coherence of a few example huge data sets. There, each data set is bucketed into the cells of a spatial grid and their coherence is shown both locally and globally: For a single cell, its *width* measures the difference between the first point in the cell and the last, so that a smaller width means better coherence; The cells are themselves indexed according to the first points that fall in them, so that small index differences between adjacent cells means better coherence. The widths of the cells and the cell indices are represented by colors in the Figure 5.1 so we can clearly see the coherence. We should note that the measures we use here for spatial coherence are quite natural. The maximum cell width measures roughly the memory footprint of a program whose work can be divided by cells; similar notion of width has been used to measure the coherence of meshes (Isenburg and Lindstrom, 2005). The relation between index differences of grid cells and their Euclidean distances are commonly used to measure the spatial coherence of space filling curves (Alber and Niedermeier, 1998).

The fact that real-world huge data sets are usually spatially coherent leads to two observations: First, the programs that created the data should document the coherence in some way; second, ignoring the coherence in a huge data set then fully sorting it seems to be wasteful. These two observations lead to two ideas for preprocessing huge data points: The first idea is to document the coherence by what we call *spatial finalization*; the second idea is to perform limited amount of reordering we call *chunking* to repair the "incoherence" in the data caused by a small number of points. The chunking is simple enough that it can be coded as a small add-on to the spatial finalization program.

For a chosen subdivision of the space into cells, a finalization tag is represented by a cell ID. Given an input point sequence, inserting a finalization tag c into the sequence signals that no point after the tag falls into the cell c—the cell is *finalized*. Formally, we call a point sequence augmented with finalization tags, together with a header specifiying the spatial subdivision, a *spatially finalized point stream*. The spatial finalization tags in the stream allows an application that has read a prefix of the stream to construct a bounding region of the future points. This may be useful for many applications. In particular, we will demonstrate how to use it for incremental Delaunay triangulations.

Our implementation of spatial finalization is called *finalizer*. The finalizer uses a rectangular $2^k \times 2^k$ grid of cells for spatial subdivision, where k is chosen beforehand by a user. The finalizer's first pass over the input stream simply computes the smallest axis-parallel bounding box of the input, which is then subdivided into grid cells. The second pass counts how many points fall into each cell. The third pass is like the second pass, except that the finalizer decrements these counters instead of incrementing them. When a cell's counter reaches zero, the finalizer inserts a finalization tag for that cell into the stream.

Although huge data sets are mostly coherent, they can have a small number of points that seriously degrades its coherence measures. In particular, the maximum width of the cells used by a spatially finalized point stream can be increased by a few cells of large width. To repair this "incoherence" in the data caused by a small number of points, our finalizer, during the third pass, buffers all the points in each cell until the cell's counter reaches zero, then it releases all the cell's points into the output stream, followed by the cell's finalization tag. We call this act *chunking*. Clearly, chunking requires far less work than fully sorting the input.

An observant reader might object that a point-creating application could destroy the coherence that our finalizer is expecting simply by delaying one point in each cell to the end of the stream. Indeed, the "grbm" data set makes the finalizer buffer many points, because there is a diagonal stripe across the terrain at the end of the file. (It appears that the airplane was still collecting data on the way home.) This vulnerability is not an inherent limitation of streaming, only of our current implementation of the finalizer. Although we did not find it necessary with our current data sets, we could reorder such points by identifying them during the second pass, and storing them in a memory buffer or a temporary file.

5.3 Streaming 2D Delaunay triangulations

Conventional Delaunay triangulation programs output triangles after all the input points have been processed. By taking as input a spatially finalized point stream, our triangulator `spdelaunay2d` constructs a Delaunay triangulation incrementally and outputs a triangle whenever it determines that the triangle is *final*—that its circumcircle does not touch or enclose an unfinalized cell. Such a triangle must be in the Delaunay triangulation, since no point arriving in the future can be inside the triangle's circumcircle. We call a triangle *active* if it is not final.

We created `spdelaunay2d` by modifying an existing Delaunay triangulator so that it keeps in memory only the active triangles and their vertices. This change dramatically reduces the program's memory footprint. The main addition to the triangulator is a component that discovers when active triangles become final, writes them to the output stream, and frees their memory. This component uses a small fraction of the total running time.

5.3.1 Delaunay triangulation with finalization

Our triangulator maintains two data structures: a triangulation, and a dynamic quadtree that remembers which regions have been finalized. Both are illustrated in Figure 5.2. The purpose of the quadtree is to identify final triangles, as described in Section 5.3.3. If the quadtree were fully expanded, its leaves would be the cells of the finalization grid; but there is no need to store the descendants of a quadrant unless it contains both finalized and unfinalized cells. Thus, our quadtree's branches extend and contract dynamically to maintain the finalization state without consuming more memory than necessary.

When `spdelaunay2d` reads a point, it inserts it into the Delaunay triangulation.

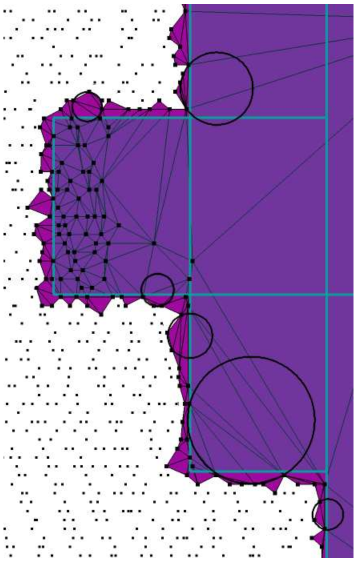

Figure 5.2: A closeup of streaming Delaunay in 2D. The points on the left have been processed, and their triangles written out. All triangles in this figure are *active*. We have drawn a few representative circumcircles, all of which intersect unfinalized space. At this moment, points are being inserted into the leftmost cell, which will be finalized next.

When it reads a finalization tag, it notes the finalized cell in the quadtree, determines which active triangles become final, writes them to the output stream, and frees their memory. Before a final triangle is written out, any vertex of that triangle that has not yet been output is written out. (Each vertex is delayed in the output stream until the first triangle that depends on it.) After a final triangle is written out, each of its vertices has its memory freed if it is no longer referenced by active triangles.

5.3.2 Delaunay triangulation with finalization

We use a triangle-based (not edge-based) data structure. Each triangle stores pointers to its three corners and its three neighbors. If a neighboring triangle is final, the corresponding pointer is null. The point insertion is done with Bowyer-Waterson style and the point location is done by walking along a straight-line (See Section 4.1 for details of these procedures.)

The point location walk might fail because of finalization. It might attempt to walk through final triangles, which are no longer in memory. The active triangles do not, in general, define a convex region. We modified the walking point locator so when it walks into a final triangle (i.e., a null pointer), the walk is restarted from a different starting point. For reasons described in the next section, each leaf of the quadtree maintains a

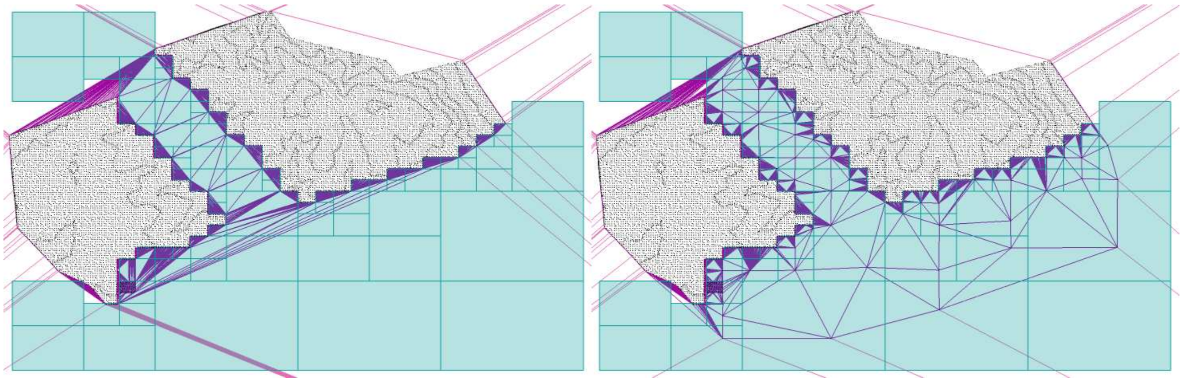

Figure 5.3: Skinny temporary triangles (left) are avoided by lazily sprinkling one point into each unfinalized quadrant at each level of the evolving quadtree (right).

list containing some of the triangles whose circumcircles intersect the leaf's quadrant. We find the quadrant enclosing p and start a new walk from one of the triangles on the quadrant's list. If this walk fails as well, we first try starting from another triangle, and then from triangles on neighboring quadrants' lists, before resorting to an exhaustive search through all the active triangles. In theory we could do better than exhaustive search, but in practice these searches account for an insignificant fraction of our running times. Fewer than 0.001% of point insertions require exhaustive search, and because we retain comparatively few triangles in memory and maintain a linked list of them with the most-recently created triangles at the front of the list, the exhaustive searches are faster than you would expect.

Final triangles pose no problem for the Bowyer-Watson update. We simply modified the depth-first search so it does not try to follow null pointers. For numerical robustness, we use the robust geometric predicates of Shewchuk (1997) to perform circle tests (deciding whether a circle encloses a point) and orientation tests (deciding which side of a line a point lies on). These tests suffice to produce a robust Delaunay triangulator.

5.3.3 Identifying final triangles

When `spdelaunay2d` reads a finalization tag, it needs to check which active triangles become final—that is, which triangles have circumcircles that no longer touch or enclose an unfinalized cell. We first check whether the circumcircle of a triangle is completely inside the cell that was just finalized—this cheap test certifies many newly created triangles as final. If that test fails, we use the fast circle-rectangle intersection checking code by Shaffer (1990) to test circumcircles against cells. We exploit the quadtree

finalized input points			`spdelaunay2d`				output mesh
name	# of points file size	op-tions	max active triangles	h:mm:ss		MB	# of triangles file size
				disk	pipe		
puget (single)	67,125,109 768 MB	l_6	56,163	4:41	5:10	6	(single) 134,207,228 2.3 GB
		l_7	48,946	4:23	4:43	7	
		l_8	49,316	4:24	4:45	7	
neuse (double)	500,141,313 11.2 GB	l_8e_4	76,337	37:42	39:41	10	(single) 1,000,282,528 16.9 GB
		l_9e_4	60,338	34:27	36:12	10	
		$l_{10}e_4$	54,802	31:57	33:46	7	
neuse 3 × 3 (double)	4,501,271,817 101 GB	$l_{10}e_5$	75,081	– : –	5:30:56	11	(single) 9,002,543,628 152 GB
		$l_{11}e_5$	67,497	– : –	4:54:40	11	
		$l_{12}e_5$	68,854	– : –	4:48:47	11	

Table 5.1: Performance of `spdelaunay2d` on large terrains. The `spfinalize` option "l_i" selects a quadtree of depth i, and "e_j" finalizes all empty quadrants in the bottom j levels of the tree at the beginning of the stream. Rows list `spdelaunay2d`'s memory footprint (MB) and two timings: one for reading pre-finalized points from *disk*, and one for reading finalized points via a *pipe* from `spfinalize`. Timings and memory footprints do *not* include `spfinalize`, except that the "pipe" timings include `spfinalize`'s third pass, which runs concurrently. For total "pipe" running times, add pass 1 & 2 timings from Table **??**. For total "pipe" memory footprints, add the footprint from Table **??**. For "disk" mode, add the running times of all three finalizer passes, and take the larger memory footprint. Disk timings for the "neuse" tiling are omitted—we do not have enough scratch disk space.

hierarchy to minimize the number of circle-box intersection tests—if a circumcircle does not intersect a quadrant, then it cannot intersect the quadrant's descendants. When it does intersect, we recurse on each child quadrant that intersects the circle.

When a triangle's circumcircle is found to intersect or enclose an unfinalized cell, it would be wasteful to check the triangle again before that cell is finalized. Thus, we link the triangle into a list maintained with the unfinalized cell, and ignore it until the cell's finalization tag arrives (or until a point insertion deletes the triangle). When we check the triangle again, we do not test it against the entire quadtree; we continue searching the quadtree (in preorder traversal) from the cell it is linked with, where the check last failed.

For our algorithm to be correct, circle-box intersection tests cannot report false negatives. False positives are acceptable because they only cause final triangles to stay longer in memory, though we prefer not to have too many of them. Rather than resorting to exact arithmetic (which is slow), we make the intersection tests conservative by computing error bounds E_x, E_y, and E_r on the center coordinates and radius of a triangle's circumcircle. Before we invoke Shaffer's code (or the simpler circle-inside-last-finalized-cell test), we enlarge the box by E_x and E_y and the circle by E_r.

5.4 Streaming 3D Delaunay triangulation

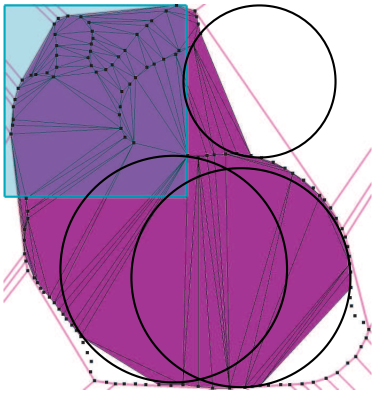

Figure 5.4: Points sampled on a closed curve. Most of space has been finalized, yet few triangles are final—most circumcircles intersect the unfinalized region.

From the stunning performance of streaming Delaunay triangulation in 2D, one would hope for a similar success story for tetrahedralizing points in 3D. Unfortunately, many gigantic data sets in 3D come from scans of surface models, and these are not amenable to a straightforward extension of the finalization procedures we developed for 2D. Delaunay tetrahedra of 3D surface points often have large cirumspheres that touch many cells; only when all touched cells are finalized do such tetrahedra become final. Figure 5.4 illustrates the 2D analog of this circumstance.

Nonetheless, we extended `spfinalize` to produce finalized points based on a octree, and implemented `spdelaunay3d` to construct a Delaunay tetrahedralization, using the same techniques described in Sections 5.2 and 5.3. Table 5.2 shows the performance of `spdelaunay3d` on pre-finalized points on the laptop described in Section 5.5. The "ppm$_k$" input points consist of every kth vertex of an isosurface mesh extracted from one timestep of a simulation of Richtmyer–Meshkov instability. In this turbulent surface, the points distribute somewhat evenly over a 3D volume and are more suitable for streaming tetrahedralizaton than surface scans. The table shows that the memory for `spdelaunay3d` is 5–10% of the output size.

finalized input points				`spdelaunay3d`			output mesh	
name	# of points	MB	opt	max active	h:mm:ss	MB	# tetrahedra	GB
ppm$_{16}$	11,737,698	136	l_4	951,683	7:42	137	80,751,131	1.4
ppm$_8$	29,362,621	341	l_5	1,903,241	22:19	306	201,721,882	3.5
ppm$_4$	58,725,279	686	l_6	4,010,296	56:23	592	405,940,587	7.0
ppm$_2$	117,450,465	1,422	l_7	6,907,250	2:41:06	795	815,321,347	14

Table 5.2: Performance of `spdelaunay3d` tetrahedralizing pre-finalized 3D points sampled from the ppm isosurface. The output is a streaming tetrahedral mesh. Option "l_i" indicates that the points are spatially finalized with an *octree* of depth i. The middle third of the table shows the maximum number of active tetrahedra, the running time (hours:minutes:seconds), and the memory footprint (MB).

Results on two smaller data sets, "sf1" and "f16," appear in Table 5.3. Each comes from volumetric data used for finite element analysis: points in "sf1" are from a pos-

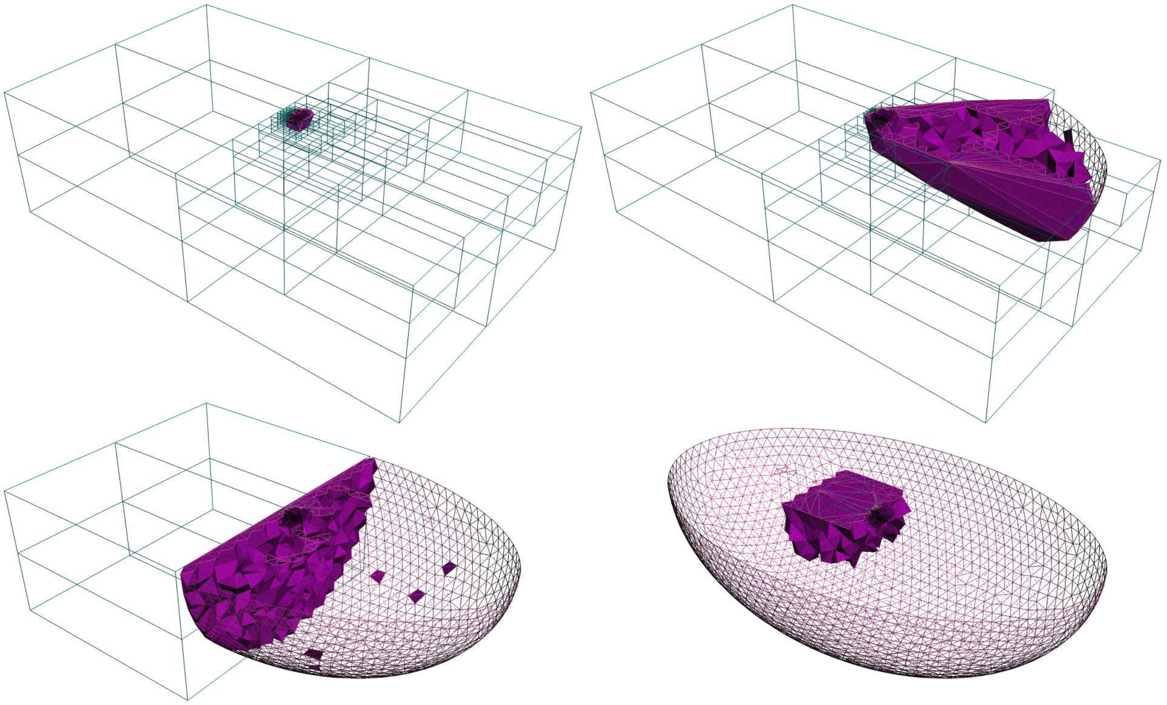

Figure 5.5: Streaming Delaunay tetrahedralization of the f16 point set. Sprinkle points are turned off for clarity. Most of this model's points are clustered near its center.

torder traversal of an adaptive octree mesh used in CMU's Quake earthquake simulation project. "sf1" is tetrahedralized slowly because its points lie on a grid, often forcing the robust geometric predicates Shewchuk (1997) to resort to exact arithmetic. The points in "f16" are the vertices of a tetrahedral mesh ordered along a space-filling z-order curve. Figure 5.5 depicts `spdelaunay3d` as it triangulates "f16."

5.5 Comparisons

Here we compare the performance of our streaming triangulators with in-core triangulators and with the previous fastest external memory Delaunay triangulator, by Agarwal, Arge, and Yi (2005), which also constructs *constrained* Delaunay triangulations.

Agarwal et al. "process 10 GB of real-life LIDAR data"—the 500 million point Neuse Basin point set (see Table 5.1), plus 755,000 segments that constrain the triangulation—"using only 128 MB of main memory in roughly 7.5 hours." This timing omits a preprocessing step that sorts the points along a space-filling Hilbert curve, taking about three additional hours. Their total time is thus 10–11 hours, compared to our 48 minutes to triangulate the unsorted points. This comparison is skewed (in opposite

input		spfinalize			spdelaunay2d		total		Triangle			output
name # of points	MB	opt	old new	MB	old new	MB	old new	MB	old new	−I/O	MB	MB triangles
grbm 6,016,883	69	l_6	0:04 0:04	15	1:07 0:23	3	1:11 0:27	18	1:47 0:34	1:02 0:17	495	208 12,018,597
puget$_5$ 13,423,821	154	l_6	0:10 0:10	7	2:17 0:55	4	2:27 1:05	11	thrash 3:45	 1:22	863	460 26,840,720

input		spfinalize			spdelaunay3d		total		Pyramid			output
name # of points	MB	opt	old new	MB	old new	MB	old new	MB	old new	−I/O	MB	MB tetrahedra
f16 1,124,648	13	l_9m_5	0:01 0:01	5	1:16 0:34	28	1:17 0:35	33	2:53 1:37	2:46 1:26	262	125 7,027,642
sf1 2,461,694	29	l_6m_5	0:02 0:02	16	9:57 4:15	29	9:59 4:17	45	thrash 5:16	 4:57	537	251 13,980,309

Table 5.3: Running times (minutes:seconds) and memory footprints (MB) of triangulators on an old laptop (top of each time box) with 512 MB memory and a new laptop (bottom of each time box) with 1 GB memory, for several 2D and 3D point sets. `spfinalize` pipes its output to `spdelaunay2d` or `spdelaunay3d`; timings for `spfinalize` reflect only the first two passes over the input stream, and timings for `spdelaunay2d` or `spdelaunay3d` reflect the combined times for the triangulator and the finalizer's third pass. The "total" column lists the start-to-end running time and memory footprint of the triangulation pipe. For the in-core triangulators Triangle and Pyramid, we report total running time and the running time excluding I/O ("−I/O"). Option "m_5" means subtrees with less than 5K points are collapsed into their root cell.

directions) by two differences. First, our triangulator does not read or respect the segments (we plan to add that capability and expect it will cost less than 20% more time for the Neuse data). Second, Agarwal et al. used a slightly faster processor, and much faster disks, than we did.

Our streaming Delaunay triangulators do more work than standard in-core algorithms, because they must identify final Delaunay triangles and tetrahedra. Nevertheless, Table 5.3 shows that they can outperform state-of-the-art in-core triangulators even for data sets that fit in memory. We compare them with the 2D triangulator Triangle Shewchuk (1996c) and the 3D triangulator Pyramid Shewchuk (1998b), modified to read input points from and write output meshes to the same binary format as our triangulators. Triangle, based on a divide-and-conquer algorithm, is the fastest sequential 2D implementation. Pyramid uses an incremental algorithm.

We used two laptops for our timings to get a sense of when the in-core triangulators start to thrash: a newer laptop described in Section 5.5, with 1 GB of memory, and an older laptop with a 1.1 GHz mobile Pentium III processor and 512 MB of memory. Four data sets appear in Table 5.3: The two 2D data sets, "grbm" and "puget," are described in Section 5.2 and depicted in Figure 5.1. The smaller "puget$_5$" is obtained by sampling every fifth point from "puget." The two 3D data sets, "f16" and "sf1," are described in Section 5.4.

The most striking differences are the memory footprints. `spdelaunay2d` uses less than 1% of the space of Triangle; `spdelaunay2d` and `spfinalize` together use less than 5%. `spdelaunay3d` uses less than 11% of the space of Pyramid; `spdelaunay3d` and `spfinalize` together use less than 13%. Moreover, Triangle and Pyramid's memory footprints increase linearly with the size of the triangulation, whereas the streaming triangulators' memory footprints increase more slowly with the stream size. Of course, the in-core triangulators begin thrashing long before the streaming triangulators would. Triangle begins to thrash on the new laptop at about 14 million input points. Compare this with the 4.5 billion points we have triangulated by streaming.

The running times are more surprising. How can the streaming triangulators, with the extra work of finalization, run faster than dedicated in-core triangulators? First, they offset the extra work by overlapping computation with file I/O, whereas Triangle and Pyramid do not. The speed of the streaming triangulators on pre-finalized points is almost entirely CPU-bound. If `spdelaunay2d`, while triangulating the Neuse Basin point stream (recall Table 5.1), discards the 16.9 GB output mesh stream instead of writing it to disk, it saves only three minutes of the 35-minute processing time.

Second, the streaming triangulators benefit from improved cache performance because of their much smaller memory footprints.

5.6 Conclusions

Researchers with whom we have discussed external memory triangulation suggest, almost by reflex, sorting the points first. For data sets with no spatial coherence at all, we too advocate sorting. But in our experience, large, real-world data sets have plenty of spatial coherence. The power of exploiting that spatial coherence is perhaps best illustrated by two facts. First, it takes Agarwal et al. (2005) three hours to Hilbert sort the same point set we triangulate in 50 minutes. Second, our triangulator runs as quickly on the original Neuse point data as on the Hilbert-sorted Neuse points (both kindly provided by Agarwal et al.)

We realize the benefits of sorting, at much less cost, by documenting the existing spatial coherence with *spatial finalization* and enhancing it by reordering points. In analogy to aikido, we use the data's spatial coherence to control and direct the data with small efforts, rather than fight it head on (by sorting it). One advantage is speed. Another advantage is that we can visualize the early output of a large pipeline of streaming modules soon after starting it.

We have described just one method of spatial finalization for point sets. We choose a depth-k quadtree/octree because we can describe it succintly with a bounding box and integer k, and it is relatively simple to determine which cells a sphere intersects. Binary space partitions, k-d trees, and many other spatial subdivisions would work too. If a point stream is sorted along a space-filling curve like a Hilbert or z-order curve, the stream is chunked, and finalization can be implicit—a cell is finalized when the next point leaves it. Sweepline algorithms, such as Fortune's (1992) for Delaunay triangulation, generate a point stream with implicit spatial finalization: they sort the points by one coordinate, thereby partitioning the plane into slabs. At each event, they finalize a slab, and could potentially produce partial output and free data structures. Likewise, Pajarola's (2005) streaming k-neighbor computation finalizes slabs of space with a sweep plane. But these methods bring with them the disadvantages of sorting discussed above.

The Achilles' heel of our 3D streaming triangulator is that it performs poorly on point clouds sampled from surfaces. The Delaunay triangulations of these point clouds have many tetrahedra with large circumspheres, which intersect many cells and are thus long-lived. We believe this problem can be solved by using more sophisticated, non-disjoint finalization regions computed by a randomized divide-and-conquer technique of Clarkson (1988). Clarkson's method covers space with overlapping spherical regions tailored to the point set, and guarantees that each Delaunay circumsphere is covered by a constant number of these regions; yet no region contains too many points. (Agarwal et al. use the same random sampling technique to divide a constrained Delaunay triangulation into subproblems. We propose to use it for spatial finalization instead.)

Implementations of traditional 2D divide-and-conquer Delaunay algorithms Shamos and Hoey (1975) are faster than incremental implementations, and even run in expected linear time on random points from some distributions Katajainen and Koppinen (1988). 2D divide-and-conquer algorithms seem amenable to a streaming implementation using our spatial finalization method. The key to fast streaming is to merge adjacent triangulations in an order dictated by the data, instead of an *a priori* order. Unfortunately, this rules out the best-known generalization of the divide-and-conquer approach to dimensions above two, the Delaunay Wall algorithm Cignoni et al. (1998), which constructs tetrahedra in an inflexible, predetermined order. We do not know how to create a 3D streaming divide-and-conquer Delaunay algorithm.

As huge data sets become ubiquitous in geometry processing, we hope that streaming geometry with finalization information and low width will become common. If point-creating programs would include finalization tags in their output streams, we could pipe them directly to our Delaunay triangulators, and begin producing triangles or tetrahedra even before all the points are created. The advantages of stream processing are so strong that we believe the producers of huge geometric data sets will have a profound incentive to make the modest efforts required to improve their spatial coherence and include finalization information.

5.7 Error bounds for computing sphere centers and radii

We sketch a forward error analysis to bound error in floating point computation of the center o and squared radius r^2 of the circumcircle of three points, a, b, and c.

We assume that our points are single precision IEEE 754 floating point numbers, and that double precision computations satisfy the IEEE 754 requirement that the result of a floating point operation equals the result of the true mathematical operation on double precision input, correctly rounded to double precision. Thus, for a point $p = (p_x, p_y)$, we assume that we correctly compute $p_q := p_x^2 + p_y^2$ in double precision.

The equation for the circle through $\{a, b, c\}$,

$$m_1 + m_x x + m_y y + m_q(x^2 + y^2) = 0, \tag{5.1}$$

has coefficients that are are determinants of the point coordinates:

$$\begin{aligned}
m_1 &= (a_x b_y - b_x a_y)c_q - (a_x c_y - c_x a_y)b_q + (b_x c_y - c_x b_y)a_q && (5.2)\\
m_x &= (a_y - c_y)(b_q - c_q) - (b_y - c_y)(a_q - c_q) && (5.3)\\
m_y &= (a_x - c_x)(b_q - c_q) - (b_x - c_x)(a_q - c_q) && (5.4)\\
m_q &= (a_x - c_x)(b_y - c_y) - (b_x - c_x)(a_y - c_y). && (5.5)
\end{aligned}$$

All expressions within parentheses can be evaluated exactly in double precision. We can easily derive that the center coordinates $o_\bullet = -\frac{m_\bullet}{2m_q}$ and squared radius $r^2 = -\frac{m_1}{m_q} + {o_x}^2 + {o_y}^2$.

Suppose that real number a is defined by an algebraic expression on floating point numbers using operations $+, -, \times, /$. We use $\overline{a}$ to denote the floating point number that results from computing a with the corresponding floating point operations, $\oplus, \ominus, \otimes, \oslash$. The error in representing a by $\overline{a}$ is denoted $\underline{a} := a - \overline{a}$. We use the following results from Shewchuk Shewchuk (1997): $\underline{m_q} \le \epsilon|\overline{m_q}| + (3\epsilon + 16\epsilon^2)\overline{n_q}$, where n_q is the permanent of the absolute values of the matrix for m_q, which equals $(|a_x| + |c_x|)(|b_y| + |c_y|) + (|a_y| + |c_y|)(|b_x| + |c_x|)$.

5.7.1 center error bound

For the center coordinate o_x, we derive an error bound of the form $o_x \le \overline{o_x} + \epsilon(|\overline{o_x}| + C)$, where $\epsilon \approx 2^{-52}$ is the machine epsilon, and C is an expression in intermediate terms of the circle equation that can be thought of as a "condition number."

First, let $T = \underline{m_x}\overline{m_q} - \overline{m_x}\underline{m_q}$ and $B = \overline{m_q}(\overline{m_q} + \underline{m_q})$. Then $o_x = (1+\epsilon)\overline{o_x} - \frac{T}{2B}$. We can bound B from below if we assume that $|\underline{m_q}| \ll |\overline{m_q}|$ and $\overline{m_q} \neq 0$. This assumption is almost always valid, because m_q is twice the area of the triangle abc; when the assumption fails we simply declare that the error in $\overline{o_x}$ is infinite.

$$
\begin{aligned}
|B| &= |\overline{m_q}(\overline{m_q} + \underline{m_q})| \\
&\ge |\overline{m_q}|(|\overline{m_q}| - (\epsilon|\overline{m_q}| + (3\epsilon + 16\epsilon^2)\overline{n_q})) \\
&\ge |\overline{m_q}|(|\overline{m_q}| - (\epsilon|\overline{m_q}| + 4\epsilon\overline{n_q})) \\
&\ge |\overline{m_q}|(|\overline{m_q}| - \epsilon(|\overline{m_q}| + 4\overline{n_q})) \\
&\ge |\overline{m_q}|(|\overline{m_q}| - \epsilon(1+\epsilon)(|\overline{m_q}| \oplus 4\overline{n_q})) \\
&\ge |\overline{m_q}|(|\overline{m_q}| - 2\epsilon(|\overline{m_q}| \oplus 4\overline{n_q})) \\
&\ge (1-\epsilon)^3|\overline{m_q}| \otimes (|\overline{m_q}| \ominus 2\epsilon \otimes (|\overline{m_q}| \oplus 4\overline{n_q}))
\end{aligned}
\tag{5.6}
$$

Now we bound T from above.

$$\begin{align*}
|T| &= |\underline{m_x}\overline{m_q} - \overline{m_x}\underline{m_q}| \\
&\le |\underline{m_x}\overline{m_q}| + |\overline{m_x}\underline{m_q}| \\
&\le |\overline{m_x}|(\epsilon|\overline{m_q}| + (3\epsilon + 16\epsilon^2)\overline{n_q}) + |\overline{m_q}|(\epsilon|\overline{m_x}| + (3\epsilon + 16\epsilon^2)\overline{n_x}) \\
&= \epsilon\left(2|\overline{m_x m_q}| + (3 + 16\epsilon)|\overline{m_x n_q}| + (3 + 16\epsilon)|\overline{m_q n_x}|\right) \tag{5.7} \\
&\le 2\epsilon\left(|\overline{m_x m_q}| + 2|\overline{m_x n_q}| + 2|\overline{m_q n_x}|\right) \\
&\le 2\epsilon(1+\epsilon)^3 \otimes \left(|\overline{m_x} \otimes \overline{m_q}| \oplus 2|\overline{m_x n_q}| \oplus 2|\overline{m_q n_x}|\right) \\
&\le 4\epsilon\left(|\overline{m_x} \otimes \overline{m_q}| \oplus 2|\overline{m_x n_q}| \oplus 2|\overline{m_q n_x}|\right)
\end{align*}$$

Denote the above bound for $|T|$ as $4\epsilon E_T$ and the bound for $|B|$ as $(1-\epsilon)^3 E_B$. We are now ready to bound the error $|\underline{o_x}|$

$$\begin{aligned}
|\underline{o_x}| &= \epsilon|\overline{o_x}| + \frac{1}{2}\left|\frac{T}{B}\right| \\
&\le \epsilon|\overline{o_x}| + \frac{1}{2}\frac{4\epsilon}{(1-\epsilon)^3}\frac{E_T}{E_B} \\
&\le \epsilon\left(|\overline{o_x}| + \frac{2(1+\epsilon)}{(1-\epsilon)^3} E_T \oslash E_B\right) \\
&\le \epsilon\left(|\overline{o_x}| + 3E_T \oslash E_B\right) \\
&\le 2\epsilon\left(|\overline{o_x}| \oplus 3 \otimes E_T \oslash E_B\right)
\end{aligned}$$

5.7.2 squared radius error bound

Denote $m1/m_q$ by A. The squared radius

$$\begin{aligned}
r^2 = -\frac{m_1}{m_q} + {o_x}^2 + {o_y}^2 &= -(\overline{A} + \underline{A}) + (\overline{o_x} + \underline{o_x})^2 + (\overline{o_y} + \underline{o_y})^2 \\
&= -\overline{A} + \overline{o_x}^2 + \overline{o_y}^2 - \underline{A} + 2\overline{o_x}\underline{o_x} + 2\overline{o_y}\underline{o_y} + \underline{o_x}^2 + \underline{o_y}^2 \\
&= (1+\epsilon)^3\left(-\overline{A} \oplus \overline{o_x} \otimes \overline{o_x} \oplus \overline{o_y} \otimes \overline{o_y}\right) - \underline{A} + 2\overline{o_x}\underline{o_x} + 2\overline{o_y}\underline{o_y} + \underline{o_x}^2 + \underline{o_y}^2 \\
&= (1+\epsilon)^3\overline{r^2} - \underline{A} + 2\overline{o_x}\underline{o_x} + 2\overline{o_y}\underline{o_y} + \underline{o_x}^2 + \underline{o_y}^2
\end{aligned}$$

We already upper bounded the errors $|\underline{o_x}|$ and $|\underline{o_y}|$. We need to bound the error $|\underline{A}|$ before bounding $|\underline{r^2}|$. Bounding $|\underline{A}|$ is similar to that for $|\underline{o_x}|$, except that the bound for $|\underline{m_1}|$ is slightly different from that for $|\underline{m_x}|$.

$$\begin{aligned}
m_1 &= c_x(a_y b_q - b_y a_q) - c_y(a_x b_q - b_x a_q) + c_q(a_x b_y - b_x a_y) \\
&\leq (1+\epsilon)^5 \,(\, c_x \otimes (a_y \otimes b_q \ominus b_y \otimes a_q) \ominus c_y(a_x \otimes b_q \ominus b_x \otimes a_q) \\
&\oplus c_q \otimes (a_x \otimes b_y \ominus b_x \otimes a_y) \,) \\
&= (1+\epsilon)^5 \overline{m_1} \leq (1+6\epsilon)\overline{m_1}
\end{aligned}$$

$$\begin{aligned}
|\underline{A}| &= |A - \overline{A}| \\
&\leq \epsilon\overline{A} + \frac{|\underline{m_1}\overline{m_q}| + |\overline{m_1}\underline{m_q}|}{|\overline{m_q}|(|\overline{m_q}| - |\underline{m_q}|)} \\
&\leq \epsilon\overline{A} + \frac{6\epsilon|\overline{m_1 m_q}| + \epsilon|\overline{m_1}(|\overline{m_q}| + 4\overline{n_q})|}{(1-\epsilon)^3 E_B} \\
&\leq \epsilon\overline{A} + \frac{\epsilon}{(1-\epsilon)^3}\frac{7|\overline{m_1 m_q}| + 4|\overline{m_1 n_q}|}{E_B} \\
&\leq \epsilon\overline{A} + \frac{\epsilon(1+\epsilon)^2}{(1-\epsilon)^3}\left(7|\overline{m_1 m_q}| \oplus 4|\overline{m_1 n_q}|\right) \oslash E_B \\
&= \epsilon\left(\overline{A} + \frac{(1+\epsilon)^2}{(1-\epsilon)^3}\left(7|\overline{m_1 m_q}| \oplus 4|\overline{m_1 n_q}|\right) \oslash E_B\right) \\
&= \epsilon\left(\overline{A} + 2\left(7|\overline{m_1 m_q}| \oplus 4|\overline{m_1 n_q}|\right) \oslash E_B\right) \\
&= 2\epsilon\left(\overline{A} \oplus 2\left(7|\overline{m_1 m_q}| \oplus 4|\overline{m_1 n_q}|\right) \oslash E_B\right)
\end{aligned}$$

Denote the upper bounds we established for $|\underline{o_x}|$, $|\underline{o_y}|$ and $|\underline{A}|$ as ϵE_x, ϵE_y and ϵE_A, respectively. Now we are ready to bound $|\underline{r^2}|$.

$$\begin{aligned}
|\underline{r^2}| &= |r^2 - \overline{r^2}| \\
&\leq 2\epsilon\overline{r^2} + |\underline{A}| + 2|\overline{o_x}\underline{o_x}| + 2|\overline{o_y}\underline{o_y}| + \underline{o_x}^2 + \underline{o_y}^2 \\
&\leq 2\epsilon\overline{r^2} + \epsilon\left(E_A + 2|\overline{o_x}|E_x + 2|\overline{o_y}|E_y + \epsilon {E_x}^2 + \epsilon {E_y}^2\right) \\
&\leq 2\epsilon\overline{r^2} + \epsilon(1+\epsilon)^6\left(E_A \oplus 2|\overline{o_x}| \otimes E_x \oplus 2|\overline{o_y}| \otimes E_y \oplus \epsilon \otimes E_x \otimes E_x \oplus \epsilon \otimes E_y \otimes E_y\right) \\
&\leq 2\epsilon\overline{r^2} + 2\epsilon\left(E_A \oplus 2|\overline{o_x}| \otimes E_x \oplus 2|\overline{o_y}| \otimes E_y \oplus \epsilon \otimes E_x \otimes E_x \oplus \epsilon \otimes E_y \otimes E_y\right) \\
&\leq 3\epsilon\left(\overline{r^2} \oplus 2\left(E_A \oplus 2|\overline{o_x}| \otimes E_x \oplus 2|\overline{o_y}| \otimes E_y \oplus \epsilon \otimes E_x \otimes E_x \oplus \epsilon \otimes E_y \otimes E_y\right)\right)
\end{aligned}$$

Chapter 6

Multivariate B-splines

Univariate *B-splines* are representations of smooth piecewise polynomials—or *splines*. (The reader should be note that, in literature, the word "spline" has been used for two kinds of mathematical objects. Besides the splines just defined, there are also *thin splate splines*, which are solutions to a type of data fitting problem that asks for fitting functions that maximize certain smoothness measures. In the multivariate setting, thin plate splines are not polynomials.)

We first review basic concepts related to B-splines, and review their multivariate generalizations. In particular, we describe an elegant construction by Marian Neamtu (2004). His construction reveals an interesting connection between splines and certain geometric properties in triangulations. In the coming chapters of this thesis, we will study these properties in detail and show how they may be relaxed to obtain a more general construction.

The following standard notation for real-valued functions will be used:

- For a set S, $\chi_S(x)$ is the membership function for S, i.e. 1 if x belongs to S and 0 otherwise.

- For an s-variate degree k polynomial p, the *polar form* P for p is the *unique* k-variate function of k vector variables $x_1, \ldots, x_k \in \mathbb{R}^s$ such that P is symmetric—P does not change if we reorder its parameters, multi-affine—P is linear with respect to each variable x_i when the other variables are fixed, and, for $x \in \mathbb{R}^s$, if $x = x_1 = \ldots = x_k$, then $P(x_1, \ldots, x_k) = p(x)$. For example, the polar form of the univariate polynomial $ax^2 + bx + c$ is $ax_1x_2 + \frac{bx_1}{2} + \frac{bx_2}{2} + c$.

6.1 Univariate B-splines

For degree $k \geq 0$ and $k+2$ reals $X_0 \leq X_1 \ldots \leq X_{k+1}$ called *knots*, the B-spline with respect to the knots is defined recursively as follows:

$$M(x \mid X_0, \ldots, X_k) := \frac{x - X_0}{X_{k+1} - X_0} M(x \mid X_1, \ldots, X_{k+1}) + \frac{X_{k+1} - x}{X_{k+1} - X_0} M(x \mid X_0, \ldots, X_k). \tag{6.1}$$

In the special case $k = 0$, the B-spline is the constant function over the interval from X_0 to X_1:

$$M(x \mid X_0, X_1) := \begin{cases} \frac{1}{X_1 - X_0} & \text{if } X_0 \leq x < X_1, \\ 0 & \text{otherwise.} \end{cases}$$

A degree k B-spline satisfy the following important properties:

- *Locally and compactly supported.* It is non-zero over the interval $[X_0, X_{k+1}]$.
- *Optimally smooth.* Assuming that the knots are distinct, it is C^{k-1}—it is continuous and can be differentiated $k-1$ times.

The optimal smoothness property assumes distinct knots. If the knots are not distinct, B-splines are still well-defined but have lower smoothness around the duplicate knots: Each time a knot is duplicated, the smoothness at the knot is lowered by one. Lowering the smoothness by using duplicating knots gives a way to introduce sharp corners along a smooth curve, which can be useful for designing curves in CAD.

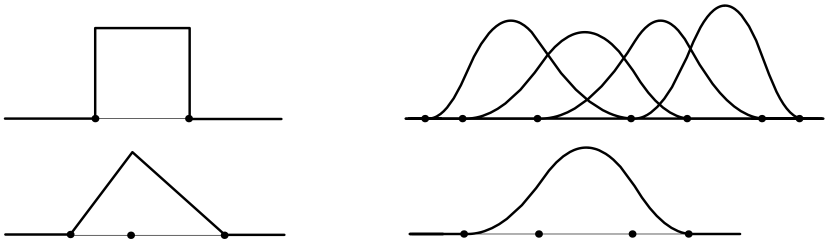

Figure 6.1: In counter-clockwise order: B-splines of degree 0, 1 and 2 and quadratic B-spline basis.

A set of B-splines can be constructed and combined linearly to form the basis of a function space. Given a universe of knots $\{\ldots \leq K_{i-1} \leq K_i \leq K_i \leq \ldots\}$, for each $i \in Z$, construct the normalized B-spline function $(K_{i+k+1} - K_i)M(\cdot \mid K_i, \ldots, K_{i+k+1})$ (See Figure 6.1). The function space spanned by the B-spline basis *reproduces polynomials*:

For a degree k polynomial p with polar form P,

$$p = \sum_{i \in \mathbb{Z}} P(K_{i+1}, \ldots, K_{i+k})(K_{i+k+1} - K_i) M(\cdot \mid X_i, \ldots, X_{i+k+1}). \tag{6.2}$$

This *polynomial reproduction* property is the "… *sine qua non* in the consideration of the approximation order of spline spaces" (Neamtu, 2004).

Note on terminology: As is the convention for most literature, the word "B-spline" can mean either a single B-spline basis function or a function chosen from a B-spline function space. The context usually resolves the ambiguity.

6.2 Multivariate B-splines and Delaunay configurations

Given the nice properties of univariate B-splines, one naturally desires their multivariate analog. However, finding an analog is not straightforward. In particular, the fundamental problem of choosing the "right" basis functions is still not settled: For $k \geq 0$, $s \geq 1$ and a universe of knots $K \subset \mathbb{R}^s$ (in arbitrary positions), we would like to construct a set of degree k basis splines that resemble univariate B-splines "as much as possible". It seems reasonable that they should at least possess the following properties:

i. Each basis spline is a locally and compactly supported, and optimally smooth.

ii. The spline space contain all polynomials of degree k.

iii. For $s = 1$, the spline space should coincide with the B-spline space over K.

There have been a number of works (Dahmen and Micchelli, 1983; Dahmen et al., 1992; Neamtu, 2004) that address this problem, but only the last solution, by Neamtu (2004), achieves all of the above properties. His solution, like all of the earlier ones, relies on the so called *simplex splines* as generalizations of single univariate basis splines and focus on solving the combinatoric problem of finding the right simplex spline sets to form bases. His rule of choosing simplex spline sets is based on the so called *Delaunay configurations*. I will review the basics of simplex splines and then review Delaunay configurations. Neamtu proves that his simplex splines reproduce polynomials. The special case of reproducing polynomials is reproducing constants—or *partition of unity*. I present a proof of partition-of-unity proof by trimming from Neamtu's proof the technical portions that involve polar forms. The goal of my presentation is to highlight how a combinatoric property of Delaunay configurations plays the essential role in establishing the polynomial-reproduction property of the spline basis.

6.2.1 simplex splines

For $s \geq 0$ and $k \geq 1$, an s-variate degree k simplex splines generalize the univariate B-splines by considering them as shadows of simplices. Figure 6.2 illustrates the simplex splines as shadows—in particular, a univariate B-spline as the shadow of a tetrahedron. Formally, for a set of $k+s+1$ knots $X \subset \mathbb{R}^s$, the simplex spline with respect to X, $M(\cdot \mid X)$, is defined as follows. Let π denote the vertical projection map $\pi : \mathbb{R}^{k+s} \to \mathbb{R}^s : (x_1, \ldots, x_{k+s}) \mapsto (x_1, \ldots, x_s)$. Let Y be any set of $k+s+1$ points in $\mathbb{R}^{k+s}$ such that $\pi(Y) = X$. Then,

$$M(x \mid X) := \frac{\mathrm{vol}_k\{y \mid y \in [Y], \pi(y) = x\}}{\mathrm{vol}_{k+s}[Y]}.$$

In the special case $k = 0$, the simplex spline is the membership function over the simplex $[X]$ normalized by its volume:

$$M(x \mid X) := \frac{1}{|d(X)|}\chi_{[X]}(x).$$

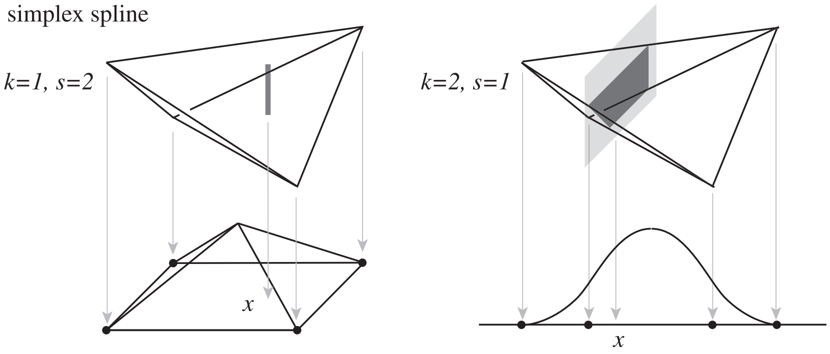

Figure 6.2: Simplex splines as shadows. The heavy dots are the defining knots. The length of the vertical segment, or the area of the vertical polygon, is the value of the function at the point x.

The simplex spline can be evaluated by the Miccelli recurrence: For $x \in \mathbb{R}^s$, let

$\{\lambda_v\}_{v\in X}$ be a set of reals such that $\sum_{v\in X}\lambda_v v = x$ and $\sum_{v\in X}\lambda_v = 1$. Then,

$$M(x \mid X) = \begin{cases} \chi_{[X]}(x) & \text{if} \# X = s+1 \\ \sum_{v\in X}\lambda_v M(x \mid X\backslash\{v\}) & \text{otherwise} \end{cases} \tag{6.3}$$

The Miccelli recurrence leaves some freedom in choosing the coefficients $\{\lambda_v\}_{v\in X}$. A canonical choice can be made: First select a set of $s+1$ affinely independent points $Y \subset X$; then, for each $v \in V$, if v belongs to the set Y, set λ_v to be the barycentric coordinate $d(\substack{x\\v}Y)/d(Y)$, otherwise set λ_v to be zero. We should note that, if we specialize this canonical Miccelli recurrence to the univariate setting and take the first and last knots in X to form the Y set, we get precisely the recurrence that has been given as the computational definition of the B-splines (Eq. 6.1).

Simplex splines possess many properties that are multivariate analogs of those possessed by B-splines. In particular, for a set of $k+s+1$ knots X in generic position—X has no affinely dependent $s+1$-subsets, The simplex spline $M(\cdot \mid X)$ satisfies that

- it is C^{k-1};
- it is supported over the polytope $\text{conv}(X)$.

These properties are multivariate analogs of those of the (individual) univariate B-splines. Therefore, if simplex splines are used for basis, the problem of constructing multivariate B-splines is reduced to one of choosing knot subsets: Given a knot universe K, choose a set of $k+s+1$-subsets—or *degree k configurations* —Δ, so that the function space spanned by the set of simplex splines $\{M(\cdot \mid X)\}_{X\in\Delta}$ satisfy properties ii and iii.

Existing solutions that address the problem of multivariate B-splines all use simplex splines as basis and study how to choose configurations. The first construction uses all possible $k+s+1$-subsets—the *complete configurations* (Dahmen and Micchelli, 1983). This construction satisfies ii but not iii; also, the size of the basis, $\binom{\#K}{k+s+1}$, makes it impractical. Most other solutions avoid the large size but do not fulfill property iii. The only solution that attains all of the properties is Neamtu's construction that uses *Delaunay configurations*. (For a more thorough comparison between the existing solutions, we refer the readers to the survey by Neamtu (2001).)

6.2.2 Delaunay configurations

For a knot set $K \subset \mathbb{R}^s$ in generic position—K has no $s+2$ subsets that are cospherical, a pair of disjoint subsets (B, I), of size $s+1$ and k respectively, is a degree k Delaunay configuration if the circumsphere of B has, of all knots in K, strictly I inside. The set of all degree k Delaunay configurations for K is denoted $\Delta_k(K)$.

For a Delaunay configuration (B, I), a normalized simplex spline with respect to the configuration, $N(\cdot \mid B, I)$ is defined:

$$N(\cdot \mid t, I) := d(t)M(\cdot \mid B \cup I).$$

The simplex spline basis associated with $\Delta_k(K)$ is

$$\{N(\cdot \mid B, I)\}_{(B,I)\in\Delta_k(K)}.$$

We should note that, since we union the pair (B, I) for the construction of simplex splines, a degree k Delaunay configuration —defined as a pair of sets—is consistent with the notion of a degree k configuration described earlier as a single set of $k + s + 1$ points.

We should note that, in the univariate setting, $\Delta_k(K)$ are precisely the set of all consecutive $k+2$-sequences in K. Then, the simplex spline basis associated with $\Delta_k(K)$ is exactly the univariate B-spline basis, as desired.

The simplex spline basis from $\Delta_k(K)$ reproduce polynomials:

Theorem 6.2.1. *(Neamtu, 2004) Let $K \subset \mathbb{R}^s$ be an infinite, locally finite set of knots in generic position—K has no $s + 2$ cospherical subsets. Then, for an s-variate, degree k polynomial p with polar form P,*

$$p = \sum_{(B,I)\in\Delta_k(K)} P(I)N(\cdot \mid B, I). \tag{6.4}$$

For the rest of the section, we will prove *partition of unity*, the special case of the above theorem when the polynomial is a constant, i.e. $p(\cdot) = 1$ and $P(\cdot) = 1$.

We will make the standing assumption about K as in Theorem 6.2.1 (The "infinite and locally finite" set assumption is made so that function properties can be stated for any point $x \in \mathbb{R}^s$. In practice, when K is finite, x has to be sufficiently far from the boundary (We will discuss what this boundary is in Chapter 7).) For brevity, we let $\Delta_n := \Delta_n(K)$ for any n.

Let us first see two equalities that express a normalized simplex spline as a sum of simplex splines of either the same degree or one degree lower. From simple algebra, we have the equality

$$N(x \mid B, I) = \sum_{0 \le i \le s} d(\tbinom{x}{i}B) M(x \mid B \cup I). \tag{6.5}$$

From Micchelli's recurrence (Eq.6.3), we have the equality

$$N(x \mid B, I) = \sum_{0 \le i \le s} d(\tbinom{x}{i}B) M(x \mid B \cup I \backslash \{B_i\}). \tag{6.6}$$

Define a degree k Delaunay *facet configuration* to be a pair (F, I) where F, the boundary set, is an s-subset of K, and I, the interior set, is a k-subset of K such that there is a sphere F that has I inside and $K \backslash I$ outside. The following proposition relates the degree k Delaunay facet configurations to Delaunay configurations of degree $k - 1$ and k:

Proposition 1. *Let $K \subset \mathbb{R}^s$ be an infinite, locally finite set of knots in generic position—K has no $s+2$ cospherical subsets. Let (F, I) be a disjoint pair of subsets of K such $\#F = s$ and there is a sphere through F that has exactly I inside, of all points in K. Then, there are exactly two points, say p_1 and p_2, with the property that the closed ball $B_i := \mathrm{ball}(F \cup \{p_i\})$ contains just the points $F \cup I \cup \{p_i\} = B_i \cap K$. Furthermore, we can observe three cases by whether these points are in I:*

i. Neither belongs to I; they are on the opposite sides of $\mathrm{aff}(F)$.

ii. Exactly one belongs to I; they are on the same side of $\mathrm{aff}(F)$.

iii. Both belong to I; they are on the opposite sides of $\mathrm{aff}(F)$.

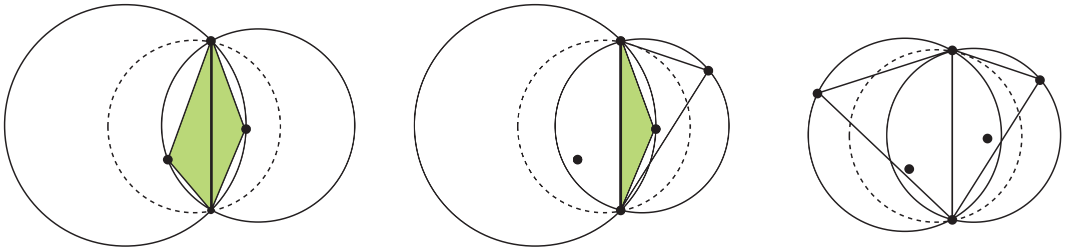

Figure 6.3: Illustration of Proposition 1. The dotted circle goes through two points and has a pair of points I inside. The solid circles are circumcircles of three points—indicated by triangles. The un-shaded triangles abut on points from I, while the colored triangles do not.

The proposition declares a relation between Δ_{k-1} and Δ_k via the facet configurations. This relation allows the following recurrence to be derived.

$$\sum_{(B,I)\in\Delta_k} N(\cdot \mid B, I) = \sum_{(B,I)\in\Delta_{k-1}} N(\cdot \mid B, I) \tag{6.7}$$

Proof. Let us consider the oriented version of facet configurations: For each facet configuration (F, I), we get its two oriented versions by ordering the boundary set F into tuples of two different parities. Let us extend the $\sim$ notation on page XXX for opposite and equivalent tuples to the oriented facet configurations: For two oriented facet configurations (F, I), (F', I')

$$(F, I) \sim (F', I') \Leftrightarrow F \sim F', I = I'$$
$$(F, I) \sim -(F', I') \Leftrightarrow F \sim -F', I = I'$$

Equivalent or opposite facet configurations give rise to equal or opposite expressions of real numbers: For any $x \in \mathbb{R}^s$,

$$(F, I) \sim (F', I') \Rightarrow d({}^xF)M(x \mid F \cup I) = d({}^xF')M(x \mid F \cup I') \tag{6.8}$$
$$(F, I) \sim -(F', I') \Rightarrow d({}^xF)M(x \mid F \cup I) = -d({}^xF')M(x \mid F \cup I') \tag{6.9}$$

Given a degree n configuration (B, I) and an integer $0 \leq i \leq s$, two oriented facet configurations, of degree n and $n+1$, can be derived: $({}_iB, I)$ and $({}_iB, I\backslash\{B_i\})$. Proposition 1 can be read as describing the converse: Given a degree k facet configuration, three cases describe how its oriented versions can be derived from configuration in Δ_{k-1} or Δ_k:

i. Both of its oriented versions can be derived from Δ_{k-1};

ii. Only one oriented version of it can be derived, but from both Δ_k and from Δ_{k-1}.

iii. Both of its oriented versions can be derived from Δ_k

Let $\mathcal{F}$ denote the set of oriented facet configurations in the last case. Let $\sum$ denote the formal addition operation over a set of oriented configurations that "cancels" opposites.

Then, $\mathcal{F}$ can be computed either from Δ_{k-1} or from Δ_k:

$$\begin{aligned}\mathcal{F} &= \sum\{({}_iB, I) \mid (B, I) \in \Delta_k, 0 \leq i \leq s\} \\ &= \sum\{({}_iB, I \cup \{B_i\}) \mid (B, I) \in \Delta_{k-1}, 0 \leq i \leq s\}\end{aligned}$$

The above equalities, together with equalities 6.8 and 6.9 imply that

$$\sum_{(F,I)\in\mathcal{F}} d({}^xF)M(x \mid F \cup I) = \sum_{(B,I)\in\Delta_k} \sum_{0\leq i\leq s} d({}^x_iB)M(x \mid B \cup I) \tag{6.10}$$

$$= \sum_{(B,I)\in\Delta_{k-1}} \sum_{0\leq i\leq s} d({}^x_iB)M(x \mid B \cup I \backslash \{B_i\}) \tag{6.11}$$

By equality 6.5,

$$\text{Expression } \ 6.10 = \sum_{(B,I)\in\Delta_k} N(\cdot \mid B, I)$$

By equality 6.6,

$$\text{Expression } \ 6.11 = \sum_{(B,I)\in\Delta_{k-1}} N(\cdot \mid B, I)$$

By the above two equalities and equality 6.11, we have

$$\sum_{(B,I)\in\Delta_k} N(\cdot \mid B, I) = \sum_{(B,I)\in\Delta_{k-1}} N(\cdot \mid B, I)$$

□

Then,

$$\sum_{(B,I)\in\Delta_k} N(\cdot \mid B, I) = \sum_{(B,I)\in\Delta_{k-1}} N(\cdot \mid B, I) = \ldots = \sum_{(B,I)\in\Delta_0} N(\cdot \mid B, I) = \sum_{(B,\emptyset)\in\Delta_0} \chi_{[B]} = 1$$

Chapter 7

Generalization of Delaunay Configurations

In the previous chapter, Delaunay configurations are introduced as the geometric elements in building a simplex spline space. Two essential properties of the Delaunay configurations establish the polynomial reproduction property of the simplex spline space: First, the degree zero configurations form a triangulation; second, the facets of degree k and degree $k+1$ configurations correspond in the way stated in proposition 1. In this chapter, I generalize Delaunay configurations by constructing *generalized configurations* in two dimensions that retain the two essential properties. The direct application of the result is that the generalized configurations can be substituted for Delaunay configurations in building simplex spline spaces. Furthermore, it initiates the study of the problem of generalizing Delaunay configurations, which is interesting in its own right.

I describe two approaches. The first approach views the Delaunay configurations in one dimension higher by using a lifting map and then varies this lifting map to produce generalized configurations. This approach is fairly obvious but deserve to be described because it connects Delaunay configurations to well studied objects in computational geometry, namely k-sets and boundary-interior configurations, and gives a general setting to study properties of Delaunay configurations, such as the number of Delaunay configurations in two dimensions. The second approach views the Delaunay configurations as the output of an iterative computational procedure, which takes the Delaunay configurations of some fixed degree as input and output the Delaunay configurations of one degree higher (This procedure is dual version of Lee's algorithm (Lee, 1982).) Varying a polygon-triangulation subroutine of this procedure output the generalized configurations. The main challenge for using this approach is to show that the procedure is indeed well-defined. More specifically, it must be shown what rules—if there is any—must be obeyed by the polygon triangulation subroutine in order for one application of the procedure to generate output that are valid input of the next application of the procedure. I am able to show that no rule has to be followed for at least three iterations. For higher degree cases, although there is no proof, experiments show that the conjecture is likely to be true.

The rest of the chapter is organized as follows. Section 7.1 reviews the fundamental geometric concepts related to Delaunay configurations, namely k-sets and k-set polytopes. Much of the material in this section is taken from (Andrzejak and Welzl, 2003). Section **??** presents a property of the k-sets in three dimensions and a counting of the number of k-sets, which generalizes Lee's theorem. Section 7.3 describes the generalization of Delaunay configurations through the lifting map. Section 7.4 presents the iterative procedure for generalizing Delaunay configurations.

I should mention that, around the time of this thesis, I learned that Neamtu had similar ideas regarding the generalized configurations. He has communicated to me that the result is in manuscript form, and is expected to be published soon.

The following set notation will be used. For a discrete set X,

- let $\#X$ denote the size of X;
- let $\binom{X}{k}$ denote the set of all subsets of size $k \geq 1$ from X;
- let $\overline{X}$ denote the *centroid of* X, which is the weighted sum of the points $\sum_{p \in X} \frac{p}{\#X}$.

7.1 k-sets and boundary-interior configurations

In s dimensional Euclidean space, for a point set P, a subset of k points $X \subset P$ is a *k-set* if X can be separated from $P \backslash X$ be a hyperplane. A closely related object is *boundary-interior configuration*: A pair of subsets (B, I) of P is said to be a *boundary-interior configuration* if there is a hyperplane h such that $h \cap P = B$ and $h^+ \cap P = I$.

k-sets and boundary-interior configurations are fundamentally important objects in computational geometry that have been studied in various settings. For example, in the study of hyperplane arrangements: If one applies a point-hyperplane duality transform, a k-set in P become the k hyperplanes facing a chosen point in an arrangement of hyperplanes (that are dual to P); and the boundary-interior configurations correspond to faces in this arrangement. The structural properties of the k-sets and boundary-interior configurations are studied most extensively by Andrzejak (2003), who assumes that P are in general position, and by Schmitt (2006), who does not assume that P are in general position.

It should be noted that the term *boundary-interior configuration* is not standard. In fact, the same object has been named differently in literature—for example, as (i, j)-*partitions* in (Andrzejak and Welzl, 2003) or k-*couples* in (Schmitt and Spehner, 2006).

The goal of this section is to describe a number of structural properties of the configurations, and in particular, Theorem 7.1.2 (Andrzejak and Welzl, 2003) that connects boundary-interior configurations to the faces of the so called *k-set polytope* (in the dual setting, k-set polytopes are called k-levels (Edelsbrunner, 1987).)

For the following discussion, I will make the standing assumption that $P \subset \mathbb{R}^s$ is a set of points in general position—every $s+1$-subset of P are affinely independent. I will also use the following notation throughout:

- $\Delta(P)$ denotes the set of all configurations from P.
- For an integer $k \geq 1$, $V_k(P)$ denotes the set of all k-sets from P.
- For integer i and set X, $\Delta_{i,X}(P)$ means the configurations in $\Delta(P)$ whose boundary set has size i and interior set is X.
- For integers i and j, $\Delta_{i,j}(P)$ means the configurations in $\Delta(P)$ whose boundary set has size i and interior set has size j.

The first important structural property relates the configurations around a chosen k-set. Choose a k-set $X \in V_k(P)$. The hyperplanes that separate X from the rest of P—either strictly or touching—support a set of simplices:

$$F(X, P) := \{f \mid f = h \cap P, \overline{h^+} \supset X, \overline{h^+} \supset P \backslash X\}. \tag{7.1}$$

These simplices—I will call them the *separating simplices for* X—-can be viewed the faces of an oriented projective polytope: Let $Y := P \backslash X$; consider X and Y as matrices of homogeneous coordinates tuples; then, the simplices in $F(X, P)$ correspond the faces of the oriented projective polytope $\operatorname{conv}(-X \cup Y)$. More formally, for a set of points $f \subset P$,

$$f \in F(X, P) \Leftrightarrow -(f \cap X) \cup (f \cap Y) \in F(\operatorname{conv}(-X \cup Y)). \tag{7.2}$$

For proof, observe that, for an Euclidean point $x \in \mathbb{R}^s$ and a hyperplane h , the oriented projective point $(1, x)$ and its antipode $-(1, x)$ are on the opposite sides of h.

The second important structural property relates the configurations around a chosen number k. It turns out that subsets of the configurations whose degrees are "around k" form a certain polytope called the k-set polytope, defined as follows. For an integer $k \geq 1$, the *k-set polytope* with respect to P, denoted $\mathcal{Q}_k(P)$, is the convex hull of the centroids of all k-subsets in P:

$$\mathcal{Q}_k(P) := \text{conv}\{\overline{X} \mid X \in \binom{P}{k}\}.$$

Note that the 1-set polytope $\mathcal{Q}_1(P)$ is precisely the convex hull of P. Therefore, k-set polytopes generalize convex hulls.

The vertices of a k-set polytope correspond 1-1 to the k-sets:

Lemma 7.1.1. *For a set of k points $X \subset P$, X is a k-set if and only if $\overline{X}$ is a vertex of $\mathcal{Q}_k(P)$.*

Proof. The proof can be reduced to one dimension: For a set of real numbers $P \subset \mathbb{R}$ and integer k, prove that the k-subset of P whose sum is the largest is precisely the set of k largest numbers in P. The proof for this is easy. $\square$

The correspondence implies that, when describing the faces of a k-set polytope, it does not matter whether a vertex is represented as a centroid point or a set of k points. If the latter is used, the face description is called the *combinatorial description* of the k-set polytope.

The relations between all configurations from P and the faces of all k-set polytopes from P (for all possible k) can be established by a combinatorial map, defined as follows. Let P denote a set of $n \geq 2$ elements. Given a pair of disjoint subsets (B, I) from P such that $\#B \geq 2$, and an integer $1 \leq i < \#B - 1$, the *union map* $\mathcal{U}$ returns a set of $(i + \#I)$-subsets of P:

$$\mathcal{U}((B, I), i) := \{X \cup I \mid X \in \binom{B}{i}\}. \tag{7.3}$$

It can be easily checked that the union map is injective and that its inverse satisfies the following equality: Let V be an element from the range of $\mathcal{U}$, so that V is a set of

k-subsets of P, then

$$\mathcal{U}^{-1}(V) = ((\bigcup V \setminus \bigcap V, \bigcap V), k - \# \bigcap V). \tag{7.4}$$

Theorem 7.1.2. *Let P be a set of n points in $\mathbb{R}^s$. The ≥ 1 dimensional faces of the $[1..n)$-set polytopes of P, described combinatorially, is precisely the image of the following set under the union map:*

$$\{((B, I), i) \mid (B, I) \in \Delta(P), \#B \geq 2, 1 \leq i < \#B\}.$$

This theorem implies that, for a chosen k, the k-sets and the following configurations

face dimension	configurations
1	$\Delta_{2,k-1}(P)$
2	$\Delta_{3,k-1}(P), \Delta_{3,k-2}(P)$
$\vdots$	$\vdots$
s	$\Delta_{s,k-1}(P), \Delta_{s,k-2}(P), \ldots, \Delta_{s,k-s}(P)$

correspond to the faces of a k-set polytope through the union map and taking centroids.

This theorem also implies that in dimensions greater than three, a k-set polytope include non-simplicial faces. For example, a configuration $(\{a, b, c, d\}, \emptyset)$ in four dimensions corresponds to the three dimensional facet $\mathcal{U}((\{a, b, c, d\}, \emptyset), 2) = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d$ which is not simplicial.

7.2 Boundary-interior configurations for convex sets in three dimensions

I study boundary interior configurations for convex point sets in three dimensions. These point sets are important because, as will be seen in Section 7.3, their respective configurations correspond to Delaunay configurations in two dimensions. I prove two properties: First, the configurations with a common interior set give a triangulation of a simple polygon (Lemma 7.2.1); second, the number of k-sets is linear with respect to the number of points and the degree k (Theorem 7.2.3).

I will make the standing assumption that P is a set of points in $\mathbb{R}^3$ in general, convex position.

For a chosen a k-set $X \in V_k(P)$, consider its separating simplices $F(X,P)$ (defined in Eq.7.1.) The set can be partitioned into three types: Those using only vertices in $P \backslash X$:

$$F_+(X,P) := \{F \mid F \in F(X,P), F \subset P \backslash X\};$$

those using only vertices in X:

$$F_-(X,P) := \{F \mid F \in F(X,P), F \subset X\};$$

and the rest:

$$F_0(X,P) := F(X,P) \backslash F_+(X,P) \backslash F_-(X,P).$$

For brevity, let $F_+ := F_+(X,P)$, $F_- := F_-(X,P)$ and $F_0 := F_0(X,P)$. By definition, the set F_+ and F_- are simplicial complexes, but the set F_0 is not. F_0 consists of edges and triangles: Each edge has one vertex in F_+ and one vertex in F_-; each triangle has one edge in F_+ or F_- and two edges in F_0. The edges and triangles in F_0 can be ordered into a *belt*, by the following proposition ((See Figure 7.1). Note that this proposition is actually stated for faces of a polytope in $\mathbb{R}^3$ but becomes applicable if we view the faces $F(X,P)$ as a polytope in the oriented projective 3-space (in the sense of Eq. 7.2).

Proposition 2. *Let $X \subset \mathbb{R}^3$ and $Y \subset \mathbb{R}^3$ be two set of points that can be separated by a plane h. Let F_0 be the polytope edges and triangles "between" X and Y,*

$$F_0 := \{f \mid f \in F(\text{conv}(X \cup Y)), f \cap X \neq \emptyset, f \cap Y \neq \emptyset\}$$

or, equivalently,

$$F_0 := \{f \mid f \in F(\text{conv}(X \cup Y)), [f] \cap h \neq \emptyset\}.$$

Then, F_0 can be ordered cyclically (by their intersection along h) into a belt*: Triangles and edges alternate in this ordering and satisfy that for an adjacent triangle-edge-triangle triplet (T, E, T'), $E = T \cap T'$, and for an adjacent edge-triangle-edge triplet (E, T, E'), $T = E \cup E'$.*

The simplices in $F_+(X,P)$ are either a single edge (and its vertices) or the triangulation of a simple polygon with no internal points. The formal statement and the proof can be found in Lemma 7.2.1. I should point out that the proof uses a simple but useful Lemma 7.2.2 that analyzes the faces of $F(X,P)$ incident upon a single vertex.

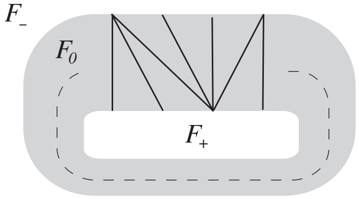

Figure 7.1: The edges and triangles in F_0 form a belt.

Lemma 7.2.1. *For a set of points $P \subset \mathbb{R}^3$ in convex position and a chosen k-set $X \subset P$, the simplicial complex $F_+(X, P)$ is either a single edge or the triangulation of a simple polygon with no internal vertex.*

Proof. Let $F_+ := F_+(X, P)$. Let $F_0 := F_0(X, P)$. Let $Y := P \backslash X$.

i. *A vertex link is connected.* Choose a vertex $v \in F_+$. Denote the cycle of vertices around v in $F(X, P)$ by L, i.e., $L = (\ldots, L_i, \ldots)$ such that the cycle of triangles $(\ldots, [L_i, L_{i+1}, v], \ldots)$ are precisely those incident on v in F_+. The subsequence of L restricted to Y, $L|Y$, is precisely the link of v in F_+. We prove that $L|Y$ is a non-empty proper subsequence of L. Let $L|X$ denote subsequence of L restricted to X, so that the vertices in $L|X$ and $L|Y$ partition L. By Lemma 7.2.2, there is a map π such that $\pi(L)$ becomes the cycle of vertices of the polytope $\text{conv}(\pi(P))$ in a plane h. Furthermore, the lemma implies that, because $v \in V(\text{conv}(P))$, $\pi(X)$ and $\pi(Y)$ can be separated by a line in h. This implies that $\pi(L|X)$ and $\pi(L|Y)$ are disjoint non-empty subsequences in $\pi(L)$. Therefore, $L|X$ is non-empty proper subsequence of L.

ii. *The boundary complex is connected.* By Proposition 2, we can assume that F_0 can be ordered into a belt. Let T denote the sub-cycle of triangles in F_0 that are incident on two points in Y:

$$T := (\{u, v, a\} \mid \{u, v, a\} \in F_0, \{u, v\} \subset Y, a \in X),$$

and B denote the associated edge cycle:

$$B := (\{u, v\} \mid \{u, v, a\} \in F_0, \{u, v\} \subset Y, a \in X)$$

Choose two adjacent triangles $\{u, v, a\}$ and $\{v', w, b\}$ in T, so that $a \in X$, $b \in X$ and the triangles after $\{v, a\}$ and before $\{v', b\}$ in F_0 do not belong to T. These triangles then all have only one vertex incident on Y. The belt property implies that this vertex must be $v = v'$. Therefore, B is a cycle of edges in which adjacent pair shares a vertex. Then, since B is precisely the edges in the boundary complex, the boundary simplex is connected.

If F_+ only a single edge, then the hypothesis is trivially true. Suppose F_+ has more than a single edge. Let β denote the boundary complex of F_+. By part 1, each vertex in β is incident on exactly two edges so β must consist of topological circles, but there is only one, by part 2. Also by part 2, F_+ has no internal vertex. Therefore, F_+ is the triangulation of a simple polygon with no internal vertex. $\square$

Lemma 7.2.2. *Let X, Y and $\{v\}$ be point sets in $\mathbb{R}^s$ such that there is a hyperplane h through v that separates X and Y. Let $n(h)$ denote the unit length normal of h; Let h' denote the hyperplane with normal $n(h)$ through the point $0+n(h)$. Let π denote the central projection map: $\pi : \mathbb{R}^s \backslash h \to h' : x \mapsto \frac{x-v}{(x-v)\cdot n(h)}$. Then,*

i. A hyperplane through v supports a face of $F(X,P)$ if and only if its restriction to h' supports a face of $\mathrm{conv}(\pi(X \cup Y))$.

ii. $v \in V_1(X \cup Y \cup \{v\})$ if and only if there is a hyperplane separating $\pi(X)$ and $\pi(Y)$ in h'.

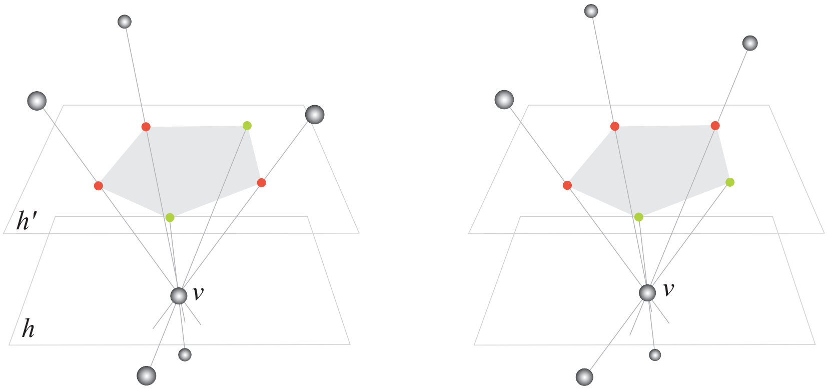

Figure 7.2: Illustration for Lemma 7.2.2. The balls above and below the hyperplane h represent the X and Y sets. The red and green dots represent the projections of X and Y sets. For each bounding edge of the convex hull (shaded), the triplet of points whose plane goes through it gives a triangle in $F(X,P)$. Left: The red and green dots can not be separated by a line; $v \notin V_1(X \cup Y \cup \{v\})$. Right: the red and green dots can be separated by line; $v \in V_1(X \cup Y \cup \{v\})$.

Theorem 7.2.3. *Let P be a set of n points in $\mathbb{R}^3$ in general and convex position. Then, for $1 \leq k < n$, the degree k-configurations are linear to both n and k. Specifically, let $v_k := \#V_k(P)$, $e(k) := \#\Delta_{2,k}(P)$ and $f_k := \#\Delta_{3,k}(P)$ denote the number k-sets, edge and triangle configurations. Then, $k \geq 1$, $f_k = 2nk - 8k + 4$. The expressions for v_k and e_k can be can be derived from f_k by equality 7.5 and 7.2.*

Proof. Since the configurations $\Delta_{0,\{2,3\}}(P)$ and P triangulate the boundary of a polytope, we have $3f_0 = 2e_0$, and, by Euler's equation, $v_1 + f_0 - e_0 = 2$. Combining these two equalities gives:

$$f_0 = 2n - 4; e_0 = 3n - 6.$$

Lemma 7.2.1 implies that the degree 0 edges correspond one-to-one to 2-sets. Therefore,

$$v_2 = e_0 = 3n - 6.$$

Theorem 7.1.2 implies that, for $k > 2$, the k-sets, the degree $k-1$ edge configurations and the degree $k-1$ and $k-2$ triangle configurations correspond to the faces of a polytope. Therefore,

$$v_k + f_{k-1} + f_{k-2} - e_{k-1} = 2;$$

$$3(f_{k-1} + f_{k-2}) = 2e_{k-1}.$$

Combining the above two equalities gives

$$v_k = \frac{f_{k-1} + f_{k-2}}{2} + 2. \tag{7.5}$$

To derive the next equality 7.6, let us consider the following set of *oriented* degree k edge configurations derived from the degree k triangle configurations.

$$L := \{{}_{t_i}t, I \cup \{t_i\}) \mid i \in \{0, 1, 2\}, (t, I) \in \Delta_{3,k-1}(P)\}.$$

Let L_X denote the subset of L with interior set X, so that $L = \bigcup_{X \in V_k(P)} L_X$. Then, by Lemma 7.2.1, L_X is either a pair of oppositely oriented edges of the single edge in $F_+(X, P)$ or the edges that bound the simple polygon represented by $F_+(X, P)$. In either case, we have $\#L_X - 2 = \#\text{triangles}(F_+(X,P))$. Then, we have the equality

$$\begin{aligned} f_k = \sum_{X \in V_k(P)} \#\text{triangles}(F_+(X,P)) = \sum_{X \in V_k(P)} \#L_X - 2 = \#L - 2v_k \\ = 3f_{k-1} - 2v_k. \quad (7.6) \end{aligned}$$

Combining equalities 7.6 and 7.5 gives

$$f_k - f_{k-1} = f_{k-1} - f_{k-2} - 4. \quad (7.7)$$

The above recurrence relations for f_k can be solved, since f_1 and f_0 are known:

$$f_k = 2nk - 8k + 4.$$

□

7.3 Projective boundary-interior configurations: A generalization of Delaunay configurations

Recall that a Delaunay triangulation can be constructed by first lifting the input points to a unit paraboloid and then computing the faces of the convex hull of the lifted points. This construction reveals a simple generalization of Delaunay triangulations to the so called regular triangulations via varying the lifting function. This section explains the analogous construction and generalization for Delaunay configurations.

Let $f : \mathbb{R}^s \to \mathbb{R}$ denote a convex function. Let the caret ($\hat{}$) denote the lifting map that takes $\mathbb{R}^s$ to the plot of f in $\mathbb{R}^{s+1}$. Then, for a set of points points $P \subset \mathbb{R}^s$ in general position—there is no $s+2$-subset of $\hat{P}$ on the same non-vertical hyperplane,

- A set of k points $X \subset P$ is a *projective k-set* with respect to P if $\hat{X}$ is a k-set with respect to $\hat{P}$ and the separating hyperplane for $\hat{X}$ is a down-facing
- A pair of disjoint subsets (B, I) of P is a *projective boundary-interior configuration* if $(\hat{B}, \hat{I})$ is a boundary-interior configuration with respect to $\hat{P}$ with a down-facing support hyperplane.

Therefore, according to the above definition, the projective k-sets and configurations correspond to the "lower half" of the k-sets and configurations in one dimension higher. The *boundary configurations* shared between the lower and upper half can be characterized in two ways:

- A projective boundary-interior configuration has a support hyperplane that is arbitrarily close to being vertical.
- Suppose the general position assumption is strengthened to forbid any linearly dependent $s+1$-subsets of P. Then, a boundary projective configuration is the same as the set of all configurations defined without the lifting map.

Let $V_k(P)$ denote the projective k-sets and $\Delta_{i,k}(P)$ denote the projective boundary-interior configurations of degree k and dimension $i-1$. The configurations $\Delta_{s+1,k}(P)$ generalize Delaunay configurations: If the unit paraboloid function is used as the lifting functions, they are precisely the degree k Delaunay configurations defined earlier. Also, for any lifting function, the set $\Delta_{s+1,k}(P)$ satisfy the facet-matching property stated in proposition 1. Therefore, these projective configurations can be substituted for Delaunay configurations in the proof of Theorem 6.2.1.

In two dimensions, by Theorem 7.1.2, for $k \geq 2$, the set of points $\{\overline{X} \mid X \in V_k(P)\}$, the set of edges $\{\mathcal{U}((B,I),1) \mid (B,I) \in \Delta_{2,k-1}(P)\}$, and the two sets of triangles $\{\mathcal{U}((B,I),1) \mid (B,I) \in \Delta_{3,k-1}(P)\}$ and $\{\mathcal{U}((B,I),2) \mid (B,I) \in \Delta_{3,k-2}(P)\}$ form a planar triangulation. This will be called *order k Delaunay triangulation.*

7.4 Generalized configs in two dimensions by computational procedures

In two dimensions, point-set triangulations can be considered as generalization of Delaunay triangles. The generalization is predicated upon only incidence and orientation relations between points and lines and retains the tessellation property of the Delaunay triangles. It is natural to wonder whether similar kinds of generalizations exist for higher degree Delaunay configurations. This Delaunay-configuration-generalization problem is seldom asked, perhaps due to two reasons: First, it is not clear what properties of Delaunay configurations are useful for applications therefore should be retained; second, such a generalization must answer a fundamental question that seems to be rather difficult: How do we define k-sets—if not by separating hyperplanes?

I present one possible approach to address this question. I generalize Delaunay configurations as the output of iterative applications of a computational procedure(Definition 3).

This procedure (Table 7.1) has a step that performs polygon triangulations. The exact choice of the polygon triangulation is left open. When Delaunay polygon triangulations are chosen, the procedure generates Delaunay configurations. When non-Delaunay polygon triangulations are chosen, the procedure generates configurations that satisfy a facet-matching property of Delaunay configurations described in Proposition 1. Therefore, they can be used to build bivariate simplex spline basis. The simplicity of the construction procedure means that, unlike with Delaunay configurations, a builder of simplex spline bases needs not to worry about how to satisfy the in-circle conditions and can concentrate better on building the desired basis functions. I will demonstrate examples of this in Chapter **??**.

The main technical challenge for using this generalization is to find conditions—if there is any—that must be satisfied for the non-Delaunay polygon triangulations so that the procedure can be iterated to generate configurations of any degree. I show that, for *arbitrary polygon triangulations*, the procedure can be iterated for up to three times (Theorem 7.4.2) to generate configurations of degree up to three. Beyond three times, experiments suggests that arbitrary polygon triangulations "still work", but presently I have no proof.

In addition to the facet-matching property, the generalized configurations also retain various other properties of the Delaunay configurations. For example, k-sets correspond to edge-configurations(Lemma 7.4.3); the degree 0-1, 1-2 and 2-3 configurations form centroid triangulations of the k-set polytope (defined with separating lines) **??**. These results suggest that the generalization may be an appropriate way to generalize Delaunay configurations for "general purposes"—rather than for the purpose of constructing splines.

The rest of the section defines the generalized configurations and presents Theorem 7.4.2.

Let us first establish some simple notation

- Our objects will be constructed from a discrete set of points $P \subset \mathbb{R}^2$ called *knots*. To simplify the generalization, we will temporarily assume that P is an infinite set. This assumption will be removed in Section 7.4.3.

- A pair of knot sets, $X := (B, I)$, is called a *configuration* (config for short) if B and I are disjoint. If $\#B = 2$, (B, I) is an *edge-config*. If $\#B = 3$, (B, I) is a *triangle-config*.

- We use $\uplus$ to denote the multiset union operation. Therefore, for two sets A and

B, $A \uplus B$ has size $\#A + \#B$ and each element in $A \cap B$ is repeated twice in $A \uplus B$.

The following operations construct objects from a set of triangle-configs of fixed degree .

- Let Δ be a set of triangle-configs. For a set of points I, we let Δ_I denote the set of triangles $\{T \mid (T, I) \in \Delta\}$. It is also convenient to adopt the following convention: For a set of degree k configurations Δ_k, and a set of k points I, we write Δ_I for $(\Delta_k)_I$. Dropping k does not cause ambiguity because we will always consider one set of triangle-configs of a chosen degree.

- The *corners* of a set of triangle-configs Δ is defined:

$$C(\Delta) := \{(v, I) \mid (T, I) \in \Delta, v \in T.\}$$

- The set of *k-sets* of a set of degree $k-1$ triangle-configs Δ is defined:

$$V(\Delta) := \bigcup_{(v,I) \in C(\Delta)} \{\{v\} \cup I\}$$

- The corners of a set of triangle-configs Δ are partitioned by the k-sets of Δ:

$$C(\Delta) = \biguplus_{J \in V(\Delta)} \{(v, I) \mid (v, I) \in C(\Delta), \{v\} \cup I = J\}.$$

  For a generalized k-set $J \in V(\Delta)$, we let $C_J(\Delta)$ denote its corner partition.

- A *k-set link* is a multiset of edges defined for chosen k-set: For a k-set $J \in V(\Delta)$, the link of J is the multiset:

$$\mathrm{Lk}(J, \Delta) := \biguplus_{(v,I) \in C_J(\Delta)} \mathrm{Lk}(v, \Delta_I).$$

  If a link consists of two oppositely oriented edges, we say that a link bounds a *null polygon*; the link bounds a simple polygon if its edges form a simple closed polygonal curve in counter clockwise order.

We can now define the generalized triangle-configs:

The following *link triangulation procedure* takes a set of triangle-configs of degree k whose links bound either null or simple polygons, triangulates the links and outputs a set of triangle-configs of degree $k+1$. This procedure should be more aptly called a procedure template, since the choices of the polygon triangulations are left unspecified.

```
LinkTriangulate
Input: $\Delta$: a set of triangle-configs of degree $k$ whose links bound null or simple polygons.
Output: $\Delta'$: a set of triangle-configs of degree $k+1$.
  For each $I \in V(\Delta)$, $\Delta_I := \begin{cases} \emptyset & \text{, if } \#\,\mathrm{Lk}(I,\Delta) = 2 \\ \text{polygon-triangulate } \mathrm{Lk}(I,\Delta) & \text{, otherwise} \end{cases}$
  $\Delta' := \bigcup_{I \in V(\Delta), T \in \Delta_I} (T, I)$
```

Table 7.1: The link triangulation procedure

Definition 3. *Let Δ be a set of triangle-configs of degree $k \geq 1$ with respect to a set of knots $P \subset \mathbb{R}^2$. We say that Δ* generalizes Delaunay triangle-configs *if Δ can be generated by k iterations of the link triangulation procedure, starting with a planar triangulation, i.e., there is a triangulation Δ_0 of P such that $\Delta = \underbrace{\texttt{LinkTriangulate}\ldots\texttt{LinkTriangulate}}_{k} \Delta_0$.*

The definition is indeed a generalization: If we initialize with the Delaunay triangulation of P, use Delaunay polygon triangulations for the link triangulation procedure and apply the procedure k times, we get the degree k Delaunay configurations of P. In fact, in the Voronoi-dual setting, this algorithm for generating Delaunay configurations becomes essentially Lee's algorithm for constructing higher order Voronoi diagrams (Lee, 1982).

The generalized triangle-configs retain the facet property of the Delaunay configurations stated in Proposition 1:

Lemma 7.4.1. *Let Δ and Δ' be the input and output set of the link triangulation procedure.*

i. For $(S, I) \in \Delta$, $u \in S$, either there are $(T, J) \in \Delta$, $v \in T$, such that ${}_uS \sim -{}_vT$, $\{u\} \cup I = \{v\} \cup J$ or there are $(S', I') \in \Delta'$, $u' \in S'$, such that ${}_{u'}S' \sim {}_uS$, $\{u\} \cup I = I'$.

ii. For $(S', I') \in \Delta'$, $u' \in S'$, either there are $(T', J') \in \Delta'$, $v' \in T'$, such that ${}_{u'}S' \sim -{}_{v'}T'$, $I' = J'$, or there are $(S, I) \in \Delta$, $u \in S$, such that ${}_{u'}S' \sim {}_uS$, $\{u\} \cup I = I'$.

Proof. For a configuration $(S, I) \in \Delta$ and a chosen vertex $u \in S$, the edge ${}_uS$ belongs to the link $\mathrm{Lk}(I \cup \{u\}, \Delta)$, which either bounds a null polygon or a simple one. In the

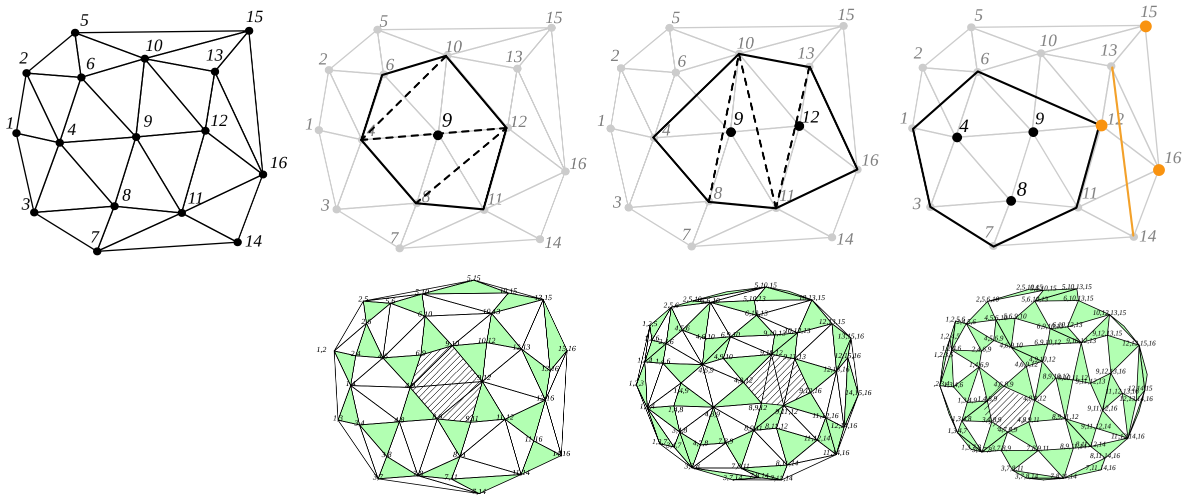

Figure 7.3: Illustration of the link triangulation algorithm. The first iteration takes a triangulation of the knots (top left), triangulates the link of each knot (for example, the second of the top row shows the triangulation for the link of knot 9), and outputs a set of degree 1 configs. The output degree 1 configs, along with the input degree 0 configs, are transformed by the union map to the centroid triangulation on the first of the bottom row, in which the triangles from degree 0 configs are colored green and the shaded part is the transformed link triangulation for knot 9. Each centroid is labeled by its corresponding set of knots; these labels are more legible if magnified. The second iteration is illustrated the same way: The third of the top row shows the link triangulation for a knot pair $\{9, 12\}$ and the picture below it shows the centroid triangulation formed by the output degree 2 configs and the input degree 1 configs. The third iteration is not completed. To complete it, take each knot set corresponding to a vertex in the second centroid triangulation and triangulate its link, for example, the link of the knot set $\{4, 8, 9\}$ on the top right. Some of these links bound null polygons, for example, the link of $\{12, 15, 16\}$, colored orange.

former case, an opposite edge $-_uS$ also belongs to the link, which implies that there is a knot $v \in I$ such that $(-^v_uS, I \cup \{u\} \backslash \{v\}) \in \Delta$. In the latter case, the edge $_uS$ is incident to exactly one triangle in the polygon triangulation. This triangle implies that there is a knot v so that $(^v_uS, I \cup \{u\}) \in \Delta$.

A configuration $(S', I') \in \Delta'$ is constructed from the triangulation of a simple polygon bounded by edges $\mathrm{Lk}(I', \Delta)$. Therefore, for any $u' \in S'$, the edge $_{u'}S'$ either belongs to the set $\mathrm{Lk}(I, \Delta)$ or is a diagonal in the polygon triangulation. In the former case, by definition of link edges, there is a knot $v' \in I'$ such that $(^{v'}_{u'}S', I' \backslash \{u'\}) \in \Delta$; in the latter case, the neighboring triangle across $_{u'}S'$ in the polygon triangulation implies that there is a knot v' such that $(-^{v'}_{u'}S', I') \in \Delta'$. $\square$

Then, by observing the role that the facet-matching property plays in the proof of Eq. 6.7, it is easy to check that the simplex splines associated with the degree k

generalized configurations reproduce polynomials of degree k.

To construct generalized configurations, the obvious way is to initialize with a planar triangulation and iterate the link triangulation procedure. By varying the choices of polygon triangulation in the procedure, one may construct configurations that suit application needs. However, it is not immediate clear that the link triangulation procedure can indeed be iterated: It must be shown that the output configs of one iteration satisfy the input condition of the next iteration—the links in the configs must bound either null of simple polygons. In Section 7.4.2, with Lemma 7.4.6 and 7.4.7, we show that this is indeed the case up for at least three iterations, using arbitrary polygonal triangulations. Therefore, we will establish the following theorem.

Theorem 7.4.2. *Starting from a planar triangulation, the link triangulation procedure can be iterated at least three times for any choices of polygon triangulation in the link triangulation procedure.*

7.4.1 Preliminaries

We adopt the convention that whenever we consider a set of degree k generalized configs Δ_k, the symbols $\Delta_0, \ldots, \Delta_{k-1}$ denote the config sets of degree 0 to $k-1$ that are generated in the construction of Δ_k.

Let us first describe a few simple properties of the generalized triangle-configs that come directly from the definition: Let Δ_k be a set of generalized triangle-configs of degree k, then

- For a corner $(v, I) \in C(\Delta_k)$, the vertex v belongs to the triangulation boundary $\partial\Delta_I$. Let a and b be the vertices before and after v in $\partial\Delta_I$. Then, the vertex link $\mathrm{Lk}(v, \Delta_I)$ forms a simple polygon curve that starts at b and ends at a.

In the projective setting, a set of k points X is a k-set if and only if there is an edge config (E, I) such that $E \cup I = X$. The same is true for the generalized k-sets:

Lemma 7.4.3. *Let J be a set of k knots and Δ_{k-1} denote a set of degree $k-1$ generalized triangle-configs. Then, $J \in V(\Delta_{k-1})$ if and only if there is a pair $\{u, v\} \subset J$ such that $\{u, v\}$ is an edge in the polygon triangulation $\Delta_{J\setminus\{u,v\}}$.*

Proof. Let $\{u, v\}$ be a pair of knots and I be a set of $k-2$ knots such that $\{u, v\}$ is an edge of Δ_I. Either the vertex link $\mathrm{Lk}(u, \Delta_I)$ or $\mathrm{Lk}(v, \Delta_I)$ has at least two edges. Withuot loss of generality, assume it is $\mathrm{Lk}(u, \Delta_I)$. Since $\mathrm{Lk}(u, \Delta_I)$ has two edges that are not opposite, $\mathrm{Lk}(\{u\} \cup I, \Delta_{k-2})$ is not a pair of opposite edges. Therefore, $\Delta_{\{u\}\cup I} \neq \emptyset$. Since $\mathrm{Lk}(u, \Delta_I) \ni v$, $\Delta_{\{u\}\cup I} \ni v$. Therefore, $(v, \{u\} \cup I)$ is a corner in Δ_{k-1}. Therefore, $\{u, v\} \cup I \in V(\Delta_{k-1})$.

Conversely, choose $J \in V(\Delta_{k-1})$. Let $v \in J$ be the knot such that $(v, J\backslash\{v\})$ is a corner of Δ_{k-1}. Therefore, v is a vertex of the polygon triangulation $\Delta_{J\backslash\{v\}}$. Then, there must be a knot $u \in J\backslash\{v\}$ such that $v \in \mathrm{Lk}(u, J\backslash\{u, v\})$. Therefore, $\{u, v\}$ is an edge of the polygon triangulation $\Delta_{J\backslash\{u,v\}}$. □

Let us present a useful technical lemma that will be used repeatedly. The lemma refers to objects called *wedges*: For points v,a and b in the plane, define $\mathrm{Wedge}(v; a, b)$ as the region swept by rotating the the ray $a - v$ to $b - v$ counter clockwise around v.

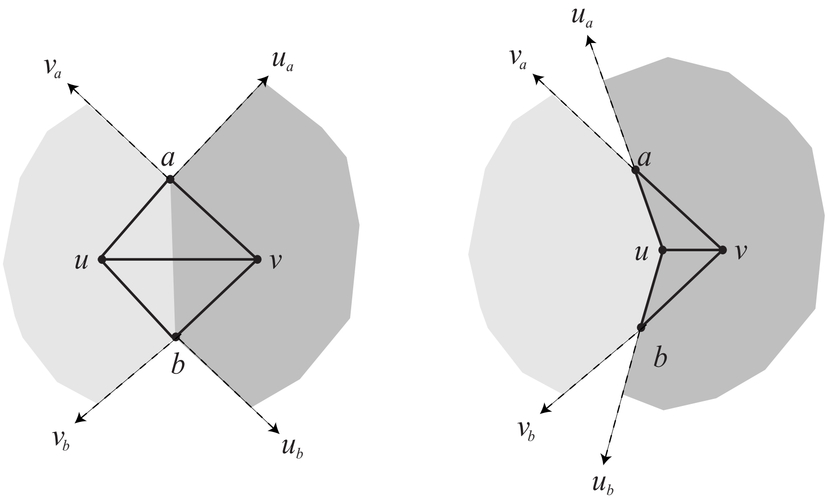

Figure 7.4: Illustration for the proof of Lemma 7.4.4.

Lemma 7.4.4. *Let $\{u, v, a, b\}$ be a set of four points in the plane. Let $[S]$ and $[T]$ be two line segments. If*

i. Points a and b are on the left and right side of the directed line (u, v).

ii. For vertex $x \in S \cup T$, $x \notin ([u, v, a] \cup [u, v, b]\backslash\{a, b\})$.

iii. $[S] \subset \mathrm{Wedge}(v; a, b)$ and $[T] \subset \mathrm{Wedge}(u; b, a)$.

Then, the intersection of the line segments $[S]$ and $[T]$ satisfies that either $[S] = [T] = [a, b]$, or, $[S] \cap [T] \in \{a, b\}$.

Proof. We consider two cases, depending on if one of the angles in $\angle a, u, b$ and $\angle b, v, a$ is concave. The illustration of the proof is found in Figure 7.4.

We construct the following points at infinity: $v_a := a - v, v_b := b - v, u_a = a - u$ and $u_b = b - u$.

Case I: $\angle a, u, b < \pi$ and $\angle b, v, a < \pi$.

If $S = T$, which can indeed be true, then the claim is trivially true. Otherwise, there must be one vertex pair, among S and T, that uses a point not in $\{a, b\}$. Assume it is S. Then, $S \cap \{a, b\} \in \{a, b\}$. Then, by condition 2, $[S] \cap [a, b]\} \in \{a, b\}$.

Let R_1 denote the open polygon with vertices (v_a, v_b, b, a). Let R_2 denote the open polygon with vertices (u_b, u_a, a, b). By the case assumption, R_1 and R_2 are convex and $R_1 \cap R_2 = [a, b]$.

By condition 2 and 3, the vertices $S \subset R_1$, and the vertices $T \subset R_2$. Therefore, since R_1 and R_2 are convex, $[S] \subset R_1$ and $[T] \subset R_2$. Then, $[S] \cap [T] \subseteq R_1 \cap R_2 = [a, b]$. Therefore, since $[S] \cap [a, b] \in \{a, b\}$, $[S] \cap [T] \in \{a, b\}$.

Case II: $\angle a, u, b \geq \pi$

Let R_1 denote the open polygon with vertices (v_a, v_b, b, u, a). Let $R_2 := \text{Wedge}(u; b, a)$. By the case assumption, R_1 is convex, and $R_1 \cap R_2 = [a, u] \cup [b, u]$. Also, because R_2 is a concave wedge and condition 2, $[T] \cap ([a, u] \cup [b, u]) \in \{a, b\}$.

By condition 2 and 3, the vertices $S \subset R_1$. Therefore, since R_1 is convex, $[S] \subset R_1$. Then, since $[T] \subset R_2$, $[S] \cap [T] \subseteq R_1 \cap R_2 = [a, u] \cup [b, u]$. Therefore, since $[T] \cap ([a, u] \cup [b, u]) \in \{a, b\}$, $[S] \cap [T] \in \{a, b\}$. □

Lemma 7.4.5. *Let T be the planar triangulation of a simple polygon—with or without internal vertices. Let $\{u, v\}$ denote a pair of vertices of T. If $\{u, v\}$ is a planar 2-set with respect to the vertices of T and the segment $[u, v]$ belongs to the region T, then $[u, v]$ is an edge of T.*

Proof. If $[u, v]$ is a boundary edge of T, then the claim is trivially true. Otherwise, the interior of $[u, v]$ belongs to the interior of the region of T. Assume that $[u, v]$ is a not a diagonal, then, by property of triangulations, there must be an edge $[a, b]$ that crosses $[u, v]$. Then, any half space that contains $[u, v]$ must also contains either a or b. This contradicts the fact that $\{u, v\}$ is a planar 2-set. Therefore, $[u, v]$ must be a diagonal of T. □

7.4.2 ≤ 3-set links bound null or simple polygons

The set of 1-sets V_1 is clearly the same as the knot set, i.e., $V_1 = \{\{p\} \mid p \in P\}$. Therefore, for any 1-set $\{v\} \in V_1$, the link $\mathrm{Lk}(\{v\}, \Delta_0)$ is the same as the vertex link in a planar triangulation. Therefore, it bounds a star-shaped simple polygon. Also, it is clear that each triangle in Δ_v contains at most one knot, namely v.

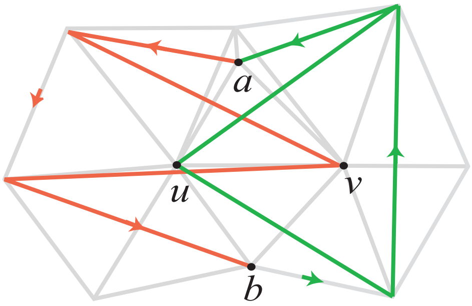

Figure 7.5: Illustration for the proof of Lemma 7.4.6. The gray edges are edges of the triangulation Δ_0. The red and green edges are diagonal edges of the triangulations Δ_u and Δ_v, respectively.

Lemma 7.4.6. *For a generalized 2-set* $I \in V(\Delta_1)$*, the collection of edges* $\mathrm{Lk}(I, \Delta_1)$ *bound a null or simple polygon.*

Proof. By Lemma 7.4.3, $\{u, v\}$ is a an edge of the triangulation Δ_0. Let $\{a, b\}$ be the pair of knots such that $(u, v, a), (v, u, b) \in \Delta_0$. Then, $\mathrm{Lk}(u, \Delta_v)$ represents a simple polygonal curve that starts at b and ends at a, and $\mathrm{Lk}(v, \Delta_u)$ represents a simple polygonal curve that starts at a and ends at b, as shown in Figure 7.5.

Choose $S \in \mathrm{Lk}(u, \Delta_v)$ and $T \in \mathrm{Lk}(v, \Delta_u)$. Because

- a and b are on the left and right side of the line (u, v), because (u, v, a) and (v, u, b) are opposite triangles in Δ_0..
- The vertices of S and T, except for $\{a, b\}$, do not belong to the triangle $[u, v, a]$ or $[u, v, b]$, since they are triangles of the knot triangulation Δ_0.
- $[S] \subset \mathrm{Wedge}(u; b, a)$ and $[T] \subset \mathrm{Wedge}(v; a, b)$, by the property of vertex links in polygon-triangulations.

, by Lemma 7.4.4, either $S = (b, a)$ and $T = (a, b)$, or $[S]$ and $[T]$ intersect only at either a or b.

Therefore, we have that either $\mathrm{Lk}(u, \Delta_v) = \{(b, a)\}$ and $\mathrm{Lk}(v, \Delta_u) = \{(a, b)\}$, or, the only common points of the curves $\mathrm{Lk}(v, \Delta_u)$ and $\mathrm{Lk}(u, \Delta_v)$ are a and b. Therefore, $\mathrm{Lk}(\{u, v\}, \Delta_1) = \mathrm{Lk}(u, \Delta_v) \uplus \mathrm{Lk}(v, \Delta_u)$ is either a pair of opposite edges or bounds a simple polygon. □

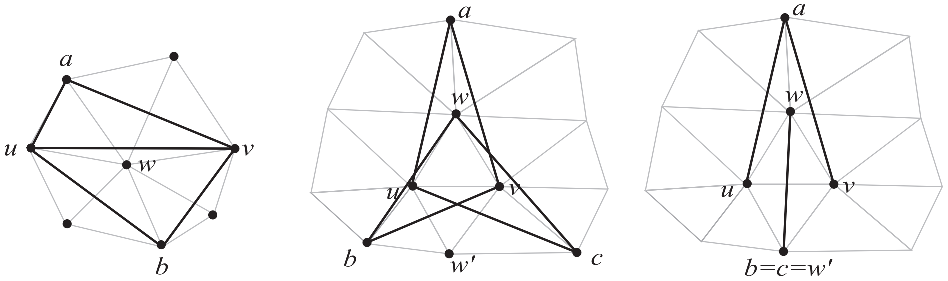

Figure 7.6: Illustration for the proof of Lemma 7.4.7. The gray edges are edges of the triangulation Δ_0. The dark edges are edges of the diagonals in Δ_1. Left: Case I. Center and right: Case II.

Lemma 7.4.7. *For a generalized 3-set* $I \in V(\Delta_2)$*, the collection of edges* $\mathrm{Lk}(I, \Delta_2)$ *bound a null or simple polygon.*

Proof. By Lemma 7.4.3, we can assume without loss of generality that $\{u, v\}$ is an edge of the polygon triangulation Δ_w. The edge $\{u, v\}$ is either a diagonal or boundary edge of the polygon triangulation. The second case implies that $\{u, v, w\}$ is a triangle in Δ_0. Let us consider these two cases separately. The basic idea is to set up the conditions so that Lemma 7.4.4 can be invoked.

The illustration for the proof is shown in Figure 7.6.

Case I: $\{u, v\}$ is a diagonal edge of Δ_w.

Let $\{a, b\}$ be the pair of knots such that $(u, v, a) \in \Delta_w$ and $(v, u, b) \in \Delta_w$. Therefore, a and b are on the left and right side of the line (u, v).

Choose $S \in \mathrm{Lk}(u, \Delta_{v,w})$ and $T \in \mathrm{Lk}(v, \Delta_{u,w})$. Because

- a and b are on the left and right side of the line (u, v).
- The vertices of S and T, except for $\{a, b\}$, do not belong to the triangle $[u, v, a]$ or $[u, v, b]$, by the property of Δ_1.
- $[S] \subset \mathrm{Wedge}(u; b, a)$ and $[T] \subset \mathrm{Wedge}(v; a, b)$, by the property of vertex links in polygon-triangulations.

, by Lemma 7.4.4, either $S = (b, a)$ and $T = (a, b)$, or $[S]$ and $[T]$ intersect only at either a or b. Therefore, either $\mathrm{Lk}(u, \Delta_{v,w}) = \{(b, a)\}$ and $\mathrm{Lk}(v, \Delta_{u,w}) = \{(a, b)\}$, or, the only common points of the curves $\mathrm{Lk}(v, \Delta_{u,w})$ and $\mathrm{Lk}(u, \Delta_{v,w})$ are their end points, a and b. Therefore, $\mathrm{Lk}(\{u, v, w\}, \Delta_2) = \mathrm{Lk}(u, \Delta_{v,w}) \uplus \mathrm{Lk}(v, \Delta_{u,w})$ bounds a null or simple polygon.

Case II: $\{u, v, w\}$ is triangle of Δ_0.

Assume that (u, v, w) is positively oriented. Let a, b and c be the three points such that $(u, v, a) \in \Delta_w$, $(v, w, b) \in \Delta_u$ and $(w, u, c) \in \Delta_v$. Therefore, a, b and c are on the left of the lines (u, v), (v, w) and (w, u), respectively.

Let w' be the knot such that $(v, u, w') \in \Delta_0$. Therefore,

- w' is on the right side of the line (u, v).
- $\mathrm{Wedge}(u; c, a) \subseteq \mathrm{Wedge}(u; w', a)$, because c is left of or on the line (u, w').
- $\mathrm{Wedge}(v; a, b) \subseteq \mathrm{Wedge}(v; a, w')$, because b is right of or on the line (v, w').

The points a, b and c are not necessarily distinct. Suppose that $b = c$. Then, we have $\mathrm{Lk}(v, \Delta_u) \supseteq \{(w, b)\}$ and $\mathrm{Lk}(u, \Delta_v) \supseteq \{(b, w)\}$. By Lemma 7.4.6, $\mathrm{Lk}(\{u, v\}, \Delta_1) = \{(w, b), (b, w)\}$ and b is the same point as w'. To be precise,

$$b = c = w' \Leftrightarrow \mathrm{Lk}(\{u, v\}, \Delta_1) = \{(w, w'), (w', w)\}$$

Without loss of generality, we assume from now on that $a \neq b$ and $a \neq c$, but it maybe that $b = c = w'$.

Choose $S \in \mathrm{Lk}(u, \Delta_{v,w})$ and $T \in \mathrm{Lk}(v, \Delta_{u,w})$. Because

- a and w' are on the left and right side of the line (u, v), respectively.
- The vertices of S and T, except for $\{a, w'\}$, do not belong to the triangle $[u, v, a]$ or $[u, v, w']$, because (u, v, w') contains no knot, and (u, v, a) contain at most one knot, namely w, but w does not belong to S and T because they are the boundary sets of the configs with interior sets $\{v, w\}$ and $\{u, w\}$.
- $[S] \subset \mathrm{Wedge}(u; c, a) \subseteq \mathrm{Wedge}(u; w', a)$ and $[T] \subset \mathrm{Wedge}(v; a, b) \subseteq \mathrm{Wedge}(v; a, w')$.

, by Lemma 7.4.4, either $S = (w', a)$ and $T = (a, w')$, or $[S]$ and $[T]$ intersect only at either a or w'.

We now prove the claim of the lemma by considering two cases $b \neq c$ or $b = c$. First suppose $b \neq c$. There must be one point among $\{b, c\}$ that is not identical to w'. Then,

$[S]$ and $[T]$ can only intersect at the knot a. Therefore, the curves $\mathrm{Lk}(v, \Delta_{u,w})$ and $\mathrm{Lk}(w, \Delta_{u,v})$ intersect at exactly at the knot a. Similarly, the curves $\mathrm{Lk}(v, \Delta_{u,w})$ and $\mathrm{Lk}(w, \Delta_{u,v})$ intersect at exactly the knot b and the curves $\mathrm{Lk}(w, \Delta_{u,v})$ and $\mathrm{Lk}(u, \Delta_{v,w})$ intersect at exactly at the knot c. Therefore, the edges $\mathrm{Lk}(\{u, v, w\}, \Delta_2) = \mathrm{Lk}(u, \Delta_{v,w}) \uplus \mathrm{Lk}(v, \Delta_{u,w}) \uplus \mathrm{Lk}(w, \Delta_{u,v})$ bound a simple polygon.

Suppose $b = c = w'$. Then, either $S = (w', a)$, $T = (a, w')$ or the edges $[S]$ and $[T]$ intersect at a or w'. Therefore, the curves $\mathrm{Lk}(v, \Delta_{u,w})$ and $\mathrm{Lk}(w, \Delta_{u,v})$ is either a pair of opposite edges or intersect at exactly at the knots a and w'. Because of this, and because $\Delta_{u,v} = \emptyset$, the edges $\mathrm{Lk}(\{u, v, w\}, \Delta_2) = \mathrm{Lk}(u, \Delta_{v,w}) \uplus \mathrm{Lk}(v, \Delta_{u,w})$ bound a null or simple polygon.

□

7.4.3 Generalization of order two and three centroid triangulations

For $k \geq 2$, Delaunay configurations of degree $k - 1$ and degree $k - 2$ form an order k centroid triangulation. The triangulation partition the planar k-set polytope of P.

Lemma 7.4.8. *The generalized 2-sets include the planar 2-sets.*

Proof. Let Δ_1 denote the degree 1 generalized configs that define the 2-sets. Let $\{u, v\}$ be a planar 2-set. Then, by Lemma 7.4.5, $\{u, v\}$ is an edge of Δ_0. Then, by Lemma 7.4.3, $\{u, v\} \in V(\Delta_1)$. □

Lemma 7.4.9. *The generalized 3-sets include the planar 3-sets.*

Proof. Let $\{u, v, w\}$ be a planar 3-set. At least two pairs in $\{u, v, w\}$ form edges of the triangulation Δ_0: If not, there must be an edge $[a, b]$ in Δ_0 that two edges of the triangle $[u, v, w]$, which contradicts the fact that $\{u, v, w\}$ is a 3-set. Suppose that these two edges are $[u, v]$ and $[u, w]$. Therefore, v and w belongs to the link polygon $\mathrm{Lk}(v, \Delta_0)$. We will prove that $[v, w]$ is an edge of the link polygon triangulation, which implies that, by Lemma 7.4.3, $\{u, v, w\}$ is a generalized 3-set.

We consider two cases, depending on if $\mathrm{Lk}(v, \Delta_0)$ is proper. First suppose it is. $[v, w]$ must belong to the region of $\mathrm{Lk}(v, \Delta_0)$: If not, the vertices from v to w around the link all belong to the triangle $[u, v, w]$, which contradicts the fact that $\{u, v, w\}$ is a 3-set. Then, because $\{v, w\}$ is a 2-set with respect to the vertices of $\mathrm{Lk}(v, \Delta_0)$, by Lemma 7.4.5, $[v, w]$ must be an edge in the link triangulation. Next suppose the polygon $\mathrm{Lk}(v, \Delta_0)$ is not proper, so that

$\square$

Chapter 8

Simplex Splines from Generalized Delaunay Configurations

In this chapter, I apply the generalization of Delaunay configurations presented in Chapter 7 to certain construction problems of bivariate splines. The basic ingredient is the simplex splines associated with a set of configurations. However, these simplex splines are not used directly. Instead, they are first collected into sets and each set is summed to give a new basis function called *bivariate B-spline*, following Neamtu **??**. The details and the reason for using this coarser basis will be explained in Section 8.1.

In Section 8.2 and 8.3, I specialize the generalized configurations so that the associated bivariate B-splines reproduce *Zwart-Powell (ZP) elements*, a textbook example of smooth box splines (de Boor et al., 1993), and *Bezier patches*, a widely used piecewise smooth spline defined over triangulations (Prautzsch et al., 2002). The results are interesting for two reasons: First, they provide evidence that bivariate B-splines provide a general frame work for bivariate splines; second, the reproduction rules for these splines can be used to mix splines of different types, which can be useful for blending patches of smooth box splines (Section 8.2.5) or to model sharp features on an otherwise smooth surface (Section 8.3.1).

In Section 8.4, the quality of bivariate B-splines for scattered data interpolation is studied experimentally. I find that, just as in the PL setting, the Delaunay configurations generally give good interpolation but for data from anisotropic functions, properly aligned non-Delaunay configurations can improve the interpolation.

8.1 Collecting simplex splines to B-splines

Recall that the simplex spline basis from the generalized configurations generalize the univariate B-splines. In this section, the simplex spline basis is modified in a simple way so that the resulting basis satisfy even more generalization property. This construction is discovered by Neamtu **??**, who names the resulting basis *bivariate B-spline* basis.

For a set of degree k generalized configurations Δ_k, let V denote the set of all interior sets of Δ_k, and, for each interior set $I \in V$, denote the set of all configurations whose interior set is I by Δ_I. Then, the *B-spline* basis with respect to Δ_k is defined by summing the normalized simplex splines associated with each Δ_I. More precisely, for each $I \in V$, there is a B-spline

$$B_I(\cdot) := \sum_{t \in \Delta_I} N(\cdot \mid t, I).$$

Collecting the simplex splines into B-splines preserves the polynomial reproduction property: For a polynomial $p \in \Pi_k$ with polar form P, the polynomial reproduction formula in Theorem 6.2.1 for simplex splines can be rewritten to become one for B-splines:

$$p = \sum_{(t,I) \in \Delta_k} P(I) N(\cdot \mid t, I) = \sum_{I \in V} P(I) \sum_{t \in \Delta_I} N(\cdot \mid t, I) = \sum_{I \in V} P(I) B_I(\cdot).$$

Therefore, the bivariate B-splines are indeed generalization of univariate B-splines.

The bivariate B-spline basis holds one more generalization property than the simplex spline basis: They generalize *control polygons* for univariate B-splines. For some real coefficients $\{\lambda_I\}_{I \in V}$, the plot of a B-spline function $\sum_{I \in V} \lambda_I B_I$, can be expressed as a parameterized surface in $\mathbb{R}^3$:

$$\{\sum_{I \in V} (\overline{I}; \lambda_I) B_I(x)\}_{x \in \mathbb{R}^2}$$

, in which the set of 3D points $\{(\overline{I}; \lambda_I)\}_{I \in V}$ can be called the *control points* (This can be easily established by reproducing the linear polynomial x with Eq. 8.1). Since the projection of the control points, $\{\overline{I}\}_{I \in V}$, are precisely the vertices of the centroid triangulation obtained by transforming the configurations Δ_{k-1} and Δ_{k-2} (See Section 7.3), the control points can be connected by a PL surface—a *control mesh*—constructed by lifting the centroid triangulation from the plane. This generalizes the control polygon for a B-spline curve and gives a visualization of the spatial relations between the B-spline basis functions (See Figure 8.1). Note that, in the linear case, the control mesh is identical to the plot of the (linear) B-spline function.

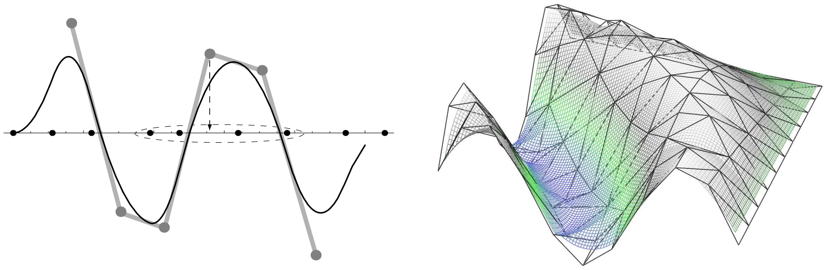

Figure 8.1: Left: The control polygon for a univariate B-spline. Right: The control mesh for a bivariate B-spline.

A computational advantage of using B-splines, rather than the finer simplex splines, is that they can be expressed in terms of configurations of one degree lower, therefore they requires less computation. To be precise, for $I \in V$, let Δ_{k-1} denote the set of degree $k-1$ configurations such that $\Delta_k = \texttt{LinkTriangulate}(\Delta_{k-1})$. Then,

$$B_I(x) = \sum_{f \in \mathrm{Lk}(I, \Delta_{k-1})} d({}^x f) M(x \mid I \cup f) \tag{8.1}$$

Proof.

$$\begin{aligned} B_I(x) = \sum_{t \in \Delta_I} N(\cdot \mid t, I) = \sum_{t \in \Delta_I} \sum_{0 \le i \le 2} d({}^x_i t) M(x \mid I \cup t \backslash \{t_i\}) &= \sum_{f \in \partial \Delta_I} d({}^x f) M(x \mid I \cup f) \\ &= \sum_{f \in \mathrm{Lk}(I, \Delta_{k-1})} d({}^x f) M(x \mid I \cup f) \end{aligned}$$

□

Expressed in this form, it is clear that the linear B-splines, defined over a planar triangulation Δ_0, are precisely the basis of the PL interpolation function over Δ_0.

Thus, B-splines, rather than simplex splines, will be used for the rest of the chapter, which study applications of quadratic B-splines. A quadratic B-spline basis will be specified by a diagram that shows the planar triangulation of the knots and how the vertex links of the triangulation are triangulated. A single quadratic B-spline will be expressed in several forms: In the form of a sum of simplex splines, in the form of a basis function indexed by some edge e in the triangulation, denoted $B_e(\cdot)$, or in the form $B(\cdot \mid P, I)$, where P is the link polygon and I is a set of two knots.

8.2 Reproduction of ZP elements

Box splines were introduced by de Boor (1976) as a generalization of univariate B-splines. Bivariate box splines—in particular their generalizations via subdivision schemes are widely used in CAD/CAM to represent surfaces.

A bivariate box spline of degree k is defined as the "shadow" of a unit cube in $\mathbb{R}^{k+2}$. A ZP-element is a quadratic box spline that is the "shadow" of a unit cube in $\mathbb{R}^4$. A ZP-element basis is generated by translating a single ZP-element over the integer grid.

I show that B-splines *reproduce* ZP-element basis in the following sense: Construct a set of quadratic B-splines associated with the configurations over the integer grid points as shown in Figure 8.3; then, each ZP-element can be expressed as a weighted sum of the B-splines. The formal statement is presented in Theorem 8.2.4. The basic idea of the proof is to first express both the ZP-element and the B-splines as sums of simplex splines and show that the sums match. The rest of the section presents the proof and is organized as follows

- As preparation, *polyhedron splines* will be introduced. Polyhedron splines generalize both the simplex and box splines and their notation can be specialized to notate both simplex and box splines.
- The B-splines and ZP elements are defined over the integer grid $\mathbb{Z} \times \mathbb{Z}$.
- Tessellation of cubes by *prisms* is studied.
- Proof of the main theorem is presented.

8.2.1 Polyhedron splines

Consider the spaces $\mathbb{R}^m$ and $\mathbb{R}^n$, where $m \leq n$. Let Ξ denote an $m \times n$ matrix, whose columns span $\mathbb{R}^m$. Ξ represents a projection map from $\mathbb{R}^n$ to $\mathbb{R}^m$, i.e., a point $y \in \mathbb{R}^n$ is taken to $\Xi y \in \mathbb{R}^m$.

Let $\mathcal{P}$ denote a polyhedron in $\mathbb{R}^n$. Then, the polyhedron shadow function, $M_\Xi(\cdot \mid \mathcal{P}) : \mathbb{R}^m \to \mathbb{R}$, is the measure of P that projects onto a point $x \in \mathbb{R}^m$:

$$M_\Xi(x \mid \mathcal{P}) := \mathrm{vol}_{n-m}\{y \mid y \in \mathcal{P}, \Xi y = x\}.$$

Two important examples of polyhedron splines are simplex and box splines:

- *Box splines.* The box spline $M_\Xi : \mathbb{R}^m \to \mathbb{R}$ is the shadow of an n-cube $[0,1]^n$: $M_\Xi(x) := M_\Xi(x \mid [0,1]^n)$.
- *Simplex splines.* Let $X \subset \mathbb{R}^m$ denote a set of $n+1$ points. Let Ξ be the matrix that projects a point $(x_1, \ldots, x_m, \ldots, x_n) \in \mathbb{R}^n$ "vertically" to the point $(x_1, \ldots, x_m) \in \mathbb{R}^m$. Let Y denote a set of points such that $\Xi Y = X$ and the simplex $[Y]$ has unit volume. Then, the simplex spline with respect to X, $M(\cdot \mid X)$, is the shadow of an n-simplex: $M(x \mid X) := M_\Xi(x \mid [Y])$.

By this definition, a box spline can be expressed as a weighted sum of simplex splines, by first triangulating the box into simplices. The weight of a simplex spline is the reciprocal of the volume of the simplex.

8.2.2 ZP elements and B-splines

This section defines two quadratic spline basis over the integer grid $\mathbb{Z} \times \mathbb{Z}$: the ZP elements and the B-splines.

The integer grid $\mathbb{Z} \times \mathbb{Z}$ can be refined by axis aligned and diagonal segments to form a mesh, as shown in Figure 8.2. The edges in the mesh follow four directions: $(1,0)$, $(0,1)$, $(1,1)$ and $(1,-1)$. These four directions define the quadratic box spline $M_{\left[\begin{smallmatrix}1 & 0 & 1 & 1\\ 0 & 1 & 1 & -1\end{smallmatrix}\right]}$, named *ZP-element*. The ZP-element is C^1 and has support over a polygon centered around the grid cell with lower left corner $(1,0)$. Translating this ZP-element by integer coordinates generate a basis.

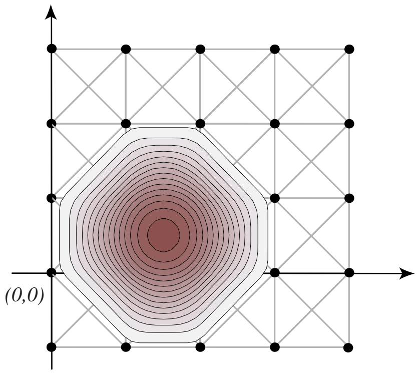

Figure 8.2: The four directional mesh over $\mathbb{Z} \times \mathbb{Z}$ and the ZP-element.

To form the B-spline basis, the first step is to refine the integer grid by diagonal edges to form a triangulation, Δ_0, as shown in Figure 8.3. The second step is to compute a set of degree one configurations Δ_1, as specified by Figure 8.3. The B-spline basis are then defined over Δ_1. It is not diffcult to check that the B-splines over Δ_1 are indexed only by the axis aligned edges of Δ_0. Let E denote the axis aligned edges of Δ_0. Then, the quadratic B-spline basis over Δ^1 is

$$\{B_{\{u,v\}}\}_{\{u,v\}\in E}. \tag{8.2}$$

Note that there are two types of these B-splines, as shown in Figure 8.3: One type is indexed by vertical edges in E and will be called P *type*; the other is indexed by horizontal edges in E and will be called the Q type.

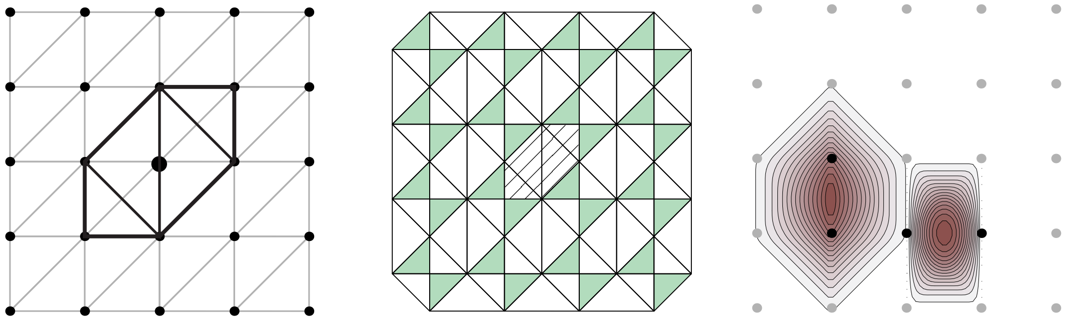

Figure 8.3: The triangulation of $\mathbb{Z}\times\mathbb{Z}$, Δ_0, the degree one configurations Δ_1. Left: Δ_0 and a link triangulation. Middle: centroid triangulation from Δ_1 and Δ_0, in which the shaded portion corresponds to the link triangulation on the left. Right: the P and Q type B-splines associated with Δ_1; the knot pairs that index them are colored black.

8.2.3 Tessellation of 4-cube

An n-dimensional prism is an n-polytope in $\mathbb{R}^n$ constructed by taking the Cartesian product of an $(n-1)$-polytope $[P]\subset\mathbb{R}^n$ with an interval $I\subset\mathbb{R}$. More specifically, we write a prism in the form $[P]\times[0,1]$, with $[0,1]$ "multiplied into" $[P]$ as the second coordinate:

$$[P]\times[0,1] := \{(x_1\ldots,x_n)\mid(x_1,x_3,\ldots,x_n)\in[P],x_2\in[0,1]\}.$$

$[P]$ will be referred to as the cross section of the prism $[P]\times[0,1]$. The prism facets $[P]\times\{0\}$ and $[P]\times\{1\}$ will be referred to as the *bottom* and the *top*, respectively.

Observation 8.2.1. *If a set of polytopes Π tessellate a polytope $\mathcal{P}$ in $\mathbb{R}^n$, then the set of prisms $\{T\times[0,1]\}_{T\in\Pi}$ tessellate the prism $\mathcal{P}\times[0,1]$ in $\mathbb{R}^{n+1}$.*

Observation 8.2.2. *Let $[A]$ be an n-simplex in $\mathbb{R}^n$ with vertices $\{A_0 \ldots, A_n\}$. Consider the prism $[A] \times [0,1]$, with bottom $A' := A \times [0]$ and top $A^* := A \times [1]$. The prism can be triangulated with the following $n+1$-simplices, generated by taking consecutive $n+1$ vertices in the list $A'_0 \ldots, A'_n, A^*_0 \ldots, A^*_n$:*

$$
\begin{array}{llr}
[& A'_0, A'_1, \ldots, A'_n, A^*_0 &] \\
[& \quad A'_1, \ldots, A'_n, A^*_0, A^*_1 &] \\
[& \qquad\qquad \ldots &] \\
[& \qquad\quad A'_n, A^*_0, A^*_1, \ldots, A^*_n &]
\end{array}.
$$

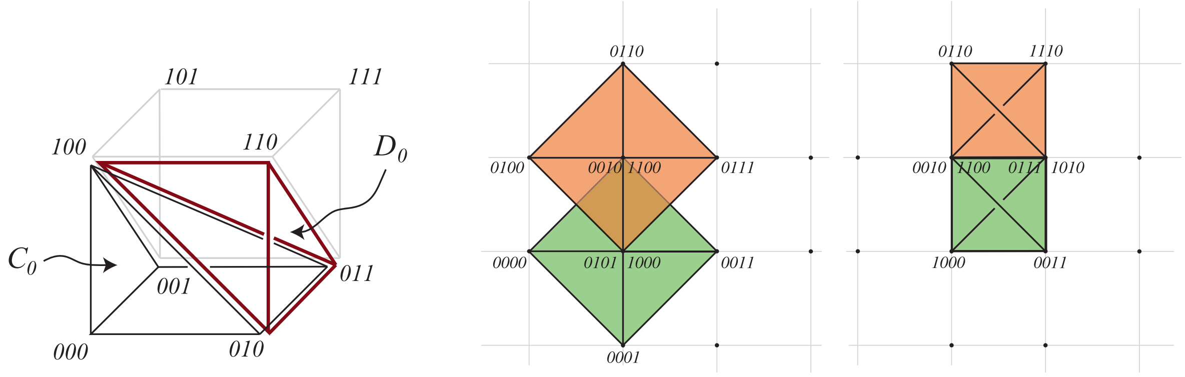

Figure 8.4: Illustration for the proof of Lemma 8.2.3. Left: the pyramid $[C_0]$ and tetrahedron $[D_0]$ in the tessellation of a cube; the symmetrical polytopes $[C_1]$ and $[D_1]$, which tessellate the half of the cube drawn in light gray, are not shown. Center: the bottom and top of the 4-prism $[C_0] \times [0,1]$ are drawn in two colors; the coordinates are those of P'_0. Right: the bottom and top of the 4-prism $[D_0] \times [0,1]$ are drawn in two colors; the coordinates are those of Q'_0.

Lemma 8.2.3. *Choose vertex subsets of a 3-cube:*

$$C_0 := \{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0)\} \tag{8.3}$$

$$C_1 := \{(1,0,0),(1,0,1),(1,1,0),(1,1,1),(0,1,1)\} \tag{8.4}$$

$$D_0 := \{(0,1,0),(0,1,1),(1,1,0),(1,0,0)\} \tag{8.5}$$

$$D_1 := \{(1,0,1),(1,0,0),(0,0,1),(0,1,1)\} \tag{8.6}$$

. *Then, the prisms in four dimensions $[C_0] \times [0,1]$, $[C_1] \times [0,1]$, $[D_0] \times [0,1]$, and $[D_1] \times [0,1]$ tessellate the 4-cube.*

Proof. As illustrated on Figure 8.4, a 3-cube can be tessellated by $[C_0]$, $[C_1]$, $[D_0]$ and $[D_1]$. By Observation 8.2.1, the four prisms stated in the claim tessellate the 4-cube. □

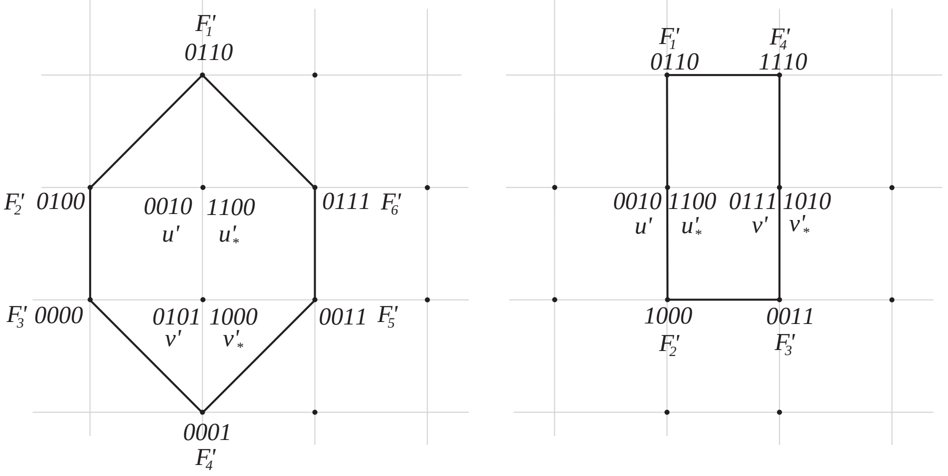

Figure 8.5: Illustration for the proof of Lemma 8.2.5.

8.2.4 Main result

This section presents the following theorem:

Theorem 8.2.4. *Let E denote the horizontal and vertical edges of the integer grid $\mathbb{Z}\times\mathbb{Z}$. Let $\{B_e\}_{e\in E}$ denote the set of B-splines associated with the degree one configurations over $\mathbb{Z}\times\mathbb{Z}$, as specified in Figure 8.3. Let $M_{\left[\begin{smallmatrix}1&0&1&1\\0&1&1&-1\end{smallmatrix}\right]}$ denote the ZP-element. Let e_1, e_2, e_3 and e_4 denote the bounding edges of the grid cell that the ZP-element is centered around, which has lower left corner $(1,0)$. Then,*

$$M_{\left[\begin{smallmatrix}1&0&1&1\\0&1&1&-1\end{smallmatrix}\right]} = \frac{1}{4!}\big(B_{e_1} + \ldots + B_{e_4}\big).$$

Lemma 8.2.5. *Let Ξ be the projection map for the ZP element. Let $P' = C_0 \times \{0,1\}$ and $Q' = D_0 \times \{0,1\}$, where C_0 and D_0 are specified in (8.3) and (8.5). Then, the projection of P' and Q' can be partitioned into sets—$F \uplus I = \Xi P'$ and $G \uplus J = \Xi Q'$ so that $\frac{1}{4!}B(\cdot \mid F, I) = M_\Xi(\cdot \mid [P'])$, and $\frac{1}{4!}B(\cdot \mid G, J) = M_\Xi(\cdot \mid [Q'])$.*

Proof. A notation convention: To relate points in $\mathbb{R}^4$ to their projection in $\mathbb{R}^2$, the same letter will be used for a point $a' \in \mathbb{R}^4$ and its projection $a = \Xi a'$, with the superscript $'$ as the distinguishing symbol.

Name the vertices in $[P']$ by $F'_{1..6}$, u', u'_*, v', and v'_* with the help of Figure 8.5. Let the projection of these points in $\mathbb{R}^2$ be $F := \Xi F'$, $u := \Xi u' = \Xi u'_*$, $v := \Xi v' = \Xi v'_*$.

The triangles $\Delta(F) = \{[F_{1,3,4}], [F_{1,4,5}], [F_{1,2,3}], [F_{1,5,6}]\}$ form a triangulation of the polygon F. Then, the B-spline $B(\cdot \mid F, I)$ can be expressed as a sum of simplex splines and derive the following equalities.

$$
\begin{aligned}
B(\cdot \mid F, I) = & \sum_{[t]\in\Delta(F)} d(t) M(\cdot \mid t \cup \{u, v\}) \\
& = 3M(\cdot \mid F_1, F_3, F_4, u, v) + 3M(\cdot \mid F_1, F_4, F_5, u, v) \\
& \quad + M(\cdot \mid F_1, F_2, F_3, u, v) + M(\cdot \mid F_1, F_5, F_6, u, v) \\
& = \quad M(\cdot \mid u, v, F_4, F_3, F_1) + M(\cdot \mid u, v, F_4, F_5, F_1) \\
& \quad + M(\cdot \mid v, F_4, F_3, F_1, u) + M(\cdot \mid v, F_4, F_5, F_1, u) \\
& \quad + M(\cdot \mid F_4, F_3, F_1, u, v) + M(\cdot \mid F_4, F_5, F_1, u, v) \\
& \quad + M(\cdot \mid F_3, F_1, u, v, F_2) + M(\cdot \mid F_5, F_1, u, v, F_6) \\
= \frac{1}{4!}\Big(& \; M_\Xi(\cdot \mid [u', v'_*, F'_4, F'_3, F'_1]) + M_\Xi(\cdot \mid [u', v'_*, F'_4, F'_5, F'_1]) \\
& + M_\Xi(\cdot \mid [v'_*, F'_4, F'_3, F'_1, u'_*]) + M_\Xi(\cdot \mid [v'_*, F'_4, F'_5, F'_1, u'_*]) \\
& + M_\Xi(\cdot \mid [F'_4, F'_3, F'_1, u'_*, v']) + M_\Xi(\cdot \mid [F'_4, F'_5, F'_1, u'_*, v']) \\
& + M_\Xi(\cdot \mid [F'_3, F'_1, u'_*, v', F'_2]) + M_\Xi(\cdot \mid [F'_5, F'_1, u'_*, v', F'_6])\Big)
\end{aligned}
$$

The simplices whose associated polyhedron splines tessellate the prism $[P']$, as the following argument proves. By Observation 8.2.2, the first column of simplices triangulates a prism with bottom tetrahedron $[u', v'_*, F'_4, F'_3]$ and top tetrahedron $[F'_1, u'_*, v', F'_2]$; the second column of simplices triangulates the prism with bottom tetrahedron $[u', v'_*, F'_4, F'_5]$ and top tetrahedron $[F'_1, u'_*, v', F'_6]$. Denoting the prisms corresponding to the first column and second column by $[S] \times [0, 1]$ and $[T] \times [0, 1]$, it can be checked that S and T triangulate the pyramid $[C_0]$ (Eq. 8.3), which is the cross section of the prism $[P']$. Then, by Observation 8.2.1, $[S] \times [0, 1]$ and $[T] \times [0, 1]$ tessellate $[P']$. Finally, it can be concluded that the simplices in the sum tessellate $[P']$.

Name the vertices in $[Q']$ by $F'_{1..4}$, u', u'_*, v', and v'_* with the help of Figure 8.5. The projections of these points in $\mathbb{R}^2$ are: $F := \Xi F$, $u := \Xi u' = \Xi u'_*$, $v := \Xi v' = \Xi v'_*$. It can be checked that $F \cup \{u, v\} = Q$.

The triangles $\Delta(F) = \{[F_{1,2,3}], [F_{1,3}, v], [F_{1,4,v}]\}$ form a triangulation of polygon F.

Then, the B-spline $B(\cdot \mid F, I)$ can be expressed as a sum of simplex splines.

$$
\begin{aligned}
B(\cdot \mid F, I) &= \sum_{[t]\in\Delta(F)} d(t) M(\cdot \mid t \cup \{u, v\}) \\
&= 2M(\cdot \mid F_1, F_2, F_3, u, v) + M(\cdot \mid F_1, F_3, v, u, v) + M(\cdot \mid F_1, F_4, v, u, v) \\
&= \frac{1}{4!}\Big(M_\Xi(\cdot \mid [u', F_2', F_3', v', F_1']) \\
&\quad + M_\Xi(\cdot \mid [F_2', F_3', v', F_1', u_*']) \\
&\quad + M_\Xi(\cdot \mid [F_3', v', F_1', u_*', v_*']) \\
&\quad + M_\Xi(\cdot \mid [v', F_1', u_*', v_*', F_4']).\Big)
\end{aligned}
$$

By Observation 8.2.2, the simplices in the sum triangulate a prism with bottom tetrahedron $[u', F_2', F_3', v']$ and top tetrahedron $[F_1', u_*', v_*', F_4']$. Furthermore, recalling that $[D_0]$ is the cross section of $[Q']$ (Eq. 8.5), one can check that this prism is $[D_0] \times [0, 1] = [Q]$. $\square$

By Lemma 8.2.3, the unit 4-cube can be tessellated into four polytopes, either of type P or Q. By Lemma 8.2.5, the polyhedron splines from the P and Q polytopes are exactly the P and Q type B-splines, which proves the main theorem.

8.2.5 Application: patch-blending

A common problem of using box splines is how to join patches of box splines smoothly. Common solutions either use subdivision schemes, which treats certain *extraordinary vertices* along the boundary specially, or require that some degrees of freedom from the patches are used to satisfy the smoothness conditions along the boundaries. The draw back of the first approach is that the surface is generally not a polynomial around the extraordinary surfaces and have lower smoothness than elsewhere; the draw back of the second approach is that it is often complicated and requires special cases for different patch connectivities. The approach used here is to form a single triangulation of the patch meshes and construct B-splines over the triangulation in such a way that the box splines are reproduced within each patch. The reproduction leaves a set of B-splines around the boundary "free" so that they can be used to fillet between box splines. This approach has the advantage that it is mathematically simple and works the same way for any connectivity of the patches. The purpose of this section is present examples of this approach.

Patch blending with ZP-elements is based on the following basic construction of a quadratic spline space. The input is a set of rectlinear mesh-patches, assumed to be non-overlapping and infinite (the latter is assumed only to simplify the description and can be removed in practice with some care.)

i. *Stich patches.* Refine the collection of patches to form a subdivision, by adding just enough edges so that the induced faces are triangular and the subdivision covers the plane. The details depend on how the patches fit. For example,

   - If the patches join seamlessly (Figure **??**a), then there is nothing to do;
   - If the patches are separated by gaps or holes(Figure **??**b), then add edges to triangulate these gaps or holes—for example, by using constrained Delaunay triangulation ();
   - If the patches meet at *T-junctions* (Figure **??**c), then add edges to triangulate the faces around the T-junctions.

ii. *Construct configurations.*

   (a) For each patch, label a face *internal* if all its surrounding faces are rectangular. For each rectangular face in the patch, if it is not labeled internal, split it into two triangles by inserting a diagonal in the subdivision along some canonical direction; otherwise, draw a *hidden diagonal* on the face along the same direction but do not actually alter the subdivision.

   (b) Assuming the presence of hidden diagonals, triangulate the links of each vertex. If a vertex belongs to the boundary of the patch, an arbitrary triangulation can be be used; otherwise, triangulate it in the way as directed in Figure 8.3.

iii. *Construct basis.* For each edge in the subdivison incident to at least one patch-boundary vertex, construct a B-spline according to the triangulations the two vertex links. For each internal face of some patch of the subdivision, construct a ZP-element according to the directions of the grid lines of the patch. The union of the B-splines and ZP-elements is the basis.

The spline space constructed above reproduce quadratic polynomials. XXX

8.3 Reproduction of Bezier patches

Triangular Bezier patches are a popular spline representation and have been used by both CAD/CAM applications to model surfaces of 3D solids, as parametric surfaces (Farin, 1997), and by GIS applications to model terrain surfaces, as plots of bivariate functions (Haber et al., 2001).

The theories and applications of triangular Bezier patches can be found in many textbooks and surveys (Prautzsch et al., 2002). The following presents the basic definition of a single quadratic Bezier patch and the C^0 condition for a set of quadratic Bezier patches from a triangulation. Let $T = [a, b, c]$ be a triangle in $\mathbb{R}^2$. For a point $x \in T$, denote the barycentric coordinates of x in T by $\alpha_a^T(x), \alpha_b^T(x)$, and $\alpha_c^T(x)$. The basis of the quadratic Bezier patch is the set of polynomials from expanding the expression $(\alpha_a^T(x) + \alpha_b^T(x) + \alpha_c^T(x))^2$:

$$\big\{(\alpha_a^T(x))^2, (\alpha_b^T(x))^2, (\alpha_c^T(x))^2, 2\alpha_a^T(x)\alpha_b^T(x), 2\alpha_a^T(x)\alpha_c^T(x), 2\alpha_b^T(x)\alpha_c^T(x)\big\}. \tag{8.7}$$

Given a planar triangulation Δ_0, the function from joining the Bezier patches associated with the triangles in Δ_0 is C^0 if and only if:

- Whenever two triangles $S, T \in \Delta_0$ share a vertex a, then the coefficients of the basis functions $(\alpha_a^S(x))^2$ and $(\alpha_a^T(x))^2$ are equal.
- Whenever two triangles $S, T \in \Delta_0$ share an edge $[a, b]$, the coefficients of the basis functions $2\alpha_a^S(x)\alpha_b^S(x)$ and $2\alpha_a^T(x)\alpha_b^T(x)$ are equal.

Equivalently, let $V(\Delta_0)$ and $E(\Delta_0)$ denote the vertex and edge set of Δ_0; For each vertex $v \in V(\Delta_0)$, define the *vertex tent* function $\text{tent}_v(x) := \sum_{T\in \text{St}(v,\Delta_0)} \big(\alpha_v^T(x)\big)^2$; for each edge $[u, v] \in E(\Delta_0)$, let S and T be the triangles incident on $[u, v]$ and define the *edge tent* function $\text{tent}_{u,v}(x) := 2\alpha_u^S(x)\alpha_v^S(x) + 2\alpha_u^T(x)\alpha_v^T(x)$, then, the set of C^0 functions constructed by joining the Bezier patches is the linear span of all vertex and edge tents (illustrated in Figure 8.6):

$$\Big\{\text{tent}_v(\cdot)\Big\}_{v\in V} \bigcup \Big\{\text{tent}_{u,v}(\cdot)\Big\}_{[u,v]\in E}. \tag{8.8}$$

In the rest of the section, I construct a B-spline basis over Δ_0 to reproduce the Bezier basis in (8.8). The formal statement is presented in Theorem 8.3.1. The construction first pulls apart the vertices infinitesmally, which introduces degenerate edges

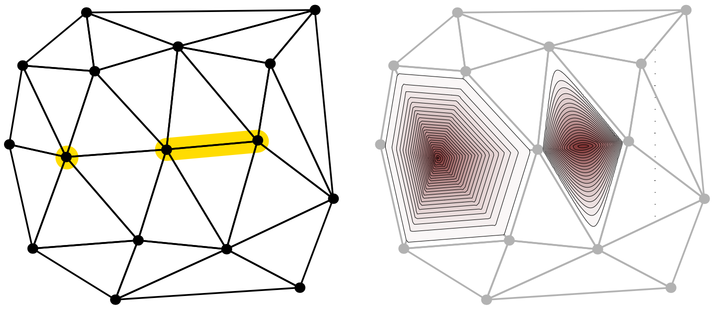

Figure 8.6: A triangulation and examples of its associated vertex and edge tent functions. The functions on the right are indexed by the highlighted vertex and edge on the left.

and triangles, then triangulates the vertex links specially as specified in Figure 8.7. The introduction of degeneracy is necessary because any quadratic B-spline function over Δ_0 is C^1 but the Bezier patch function is only C^0. It should also be noted that pulling apart vertices is analogous to duplicating knots in the univariate setting, which reproduces Bezier basis locally in a B-spline basis.

Theorem 8.3.1. *Let Δ_0 be a planar triangulation. Denote the basis of C^0 quadratic Bezier patches associated with Δ_0 by $\{\mathrm{tent}_v(\cdot)\}_{v\in V(\Delta_0)} \cup \{\mathrm{tent}_{u,v}(\cdot)\}_{\{u,v\}\in E(\Delta_0)}$. Denote the triangulation from pulling apart vertices of Δ_0 by $\widetilde{\Delta_0}$. Triangulating the vertex links of $\widetilde{\Delta_0}$ as specified by Figure 8.7 gives a set of degree one configurations. Denote its associated B-splines by $\{B_e\}_{e\in E(\widetilde{\Delta_0})}$. Then, for a vertex $v \in V(\Delta_0)$, denoting the edge from pulling apart v by $\{v_0, v_1\}$,*

$$B_{v_0,v_1}(\cdot) = \mathrm{tent}_v(\cdot);$$

for an edge $\{u, v\} \in E(\Delta_0)$, denoting the edges from pulling apart u and v by $\{u_0, u_1\}$ and $\{v_0, v_1\}$,

$$\sum_{\{u_i,v_j\}\in E(\widetilde{\Delta_0})} B_{u_i,v_j}(\cdot) = \mathrm{tent}_{u,v}(\cdot).$$

The first step of the construction is to pull apart the vertices of Δ_0 infinitesimally: For each vertex $v \in V(\Delta_0)$, an edge (v_0, v_1) is constructed, where v_0 and v_1 are infinitesimally close to v. Pulling-apart the vertices cause duplicate edges to appear: For each original edge $\{u, v\} \in \Delta_0$, pulling apart u and v induces one to three duplicate edges from the pair $\{u_0, u_1\}$ to the pair $\{v_0, v_1\}$, as illustrated in Figure 8.7.

Denote the pulled-apart triangulation by $\widetilde{\Delta_0}$. The second step is to triangulate the vertex links of $\widetilde{\Delta_0}$ in the following way. For each vertex pair $\{v_0, v_1\}$ of $\widetilde{\Delta_0}$, triangulate

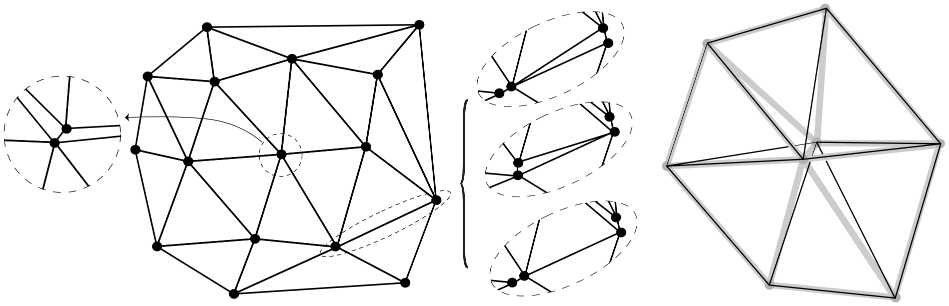

Figure 8.7: Left: a pulled apart triangulation $\widetilde{\Delta_0}$ with a vertex and edge magnified; note that there are three possible number of duplicate edges between two pulled apart vertices. Right: the link triangulations of a pair of duplicate vertices; the triangulation edges of $\widetilde{\Delta_0}$ are drawn as thick gray edges while the edges of the link triangulations are drawn as think black edges.

the polygon $\mathrm{Lk}(v_0, \widetilde{\Delta_0})$ by drawing diagonals from v_1 to all the other vertices (See Figure 8.7); and, symmetrically, triangulate the polygon $\mathrm{Lk}(v_1, \widetilde{\Delta_0})$ by drawing diagonals from v_0 to all the other vertices. Denote the resulting degree one configurations by Δ_1. The link polygons in Δ_1 can be characterized as follows:

- For an infinitesimal edge $\{v_0, v_1\}$, the link polygon $\mathrm{Lk}(\{v_0, v_1\}, \widetilde{\Delta_0})$ is infinitesimally close to $\mathrm{Lk}(v, \Delta_0)$.
- For an edge (u, v) in the original triangulation Δ_0, Let $[a, u, v]$ and $[b, u, v]$ denote the two triangles incident on edge $\{u, v\}$. If the set of edges from $\{u_0, u_1\}$ to $\{v_0, v_1\}$ contains
    - a single edge, namely $\{u_0, v_0\}$, then the polygon $\mathrm{Lk}(\{u_0, v_0\}, \Delta_1)$ is infinitesmally close to the quadrilateral (a, u, v, b).
    - two edges, namely $\{u_0, v_0\}$ and $\{u_0, v_1\}$, then the polygon $\mathrm{Lk}(\{u_0, v_0\}, \Delta_1)$ and $\mathrm{Lk}(\{u_0, v_1\}, \Delta_1)$ are infintesmally close to the triangles $[a, u, v]$ and $[b, u, v]$, respectively.
    - three edges, namely $\{u_0, v_1\}$,$\{u_1, v_0\}$ and $\{u, v\}$, where $\{u, v\}$ is the diagonal of the quadrilateral (u_0, u_1, v_0, v_1), then the polygons $\mathrm{Lk}(\{u_0, v_1\}, \Delta_1)$ and $\mathrm{Lk}(\{u_1, v_0\}, \Delta_1)$ are infinitesmally close to the triangle $[a, u, v]$ and $[b, u, v]$, respectively, and the polygon $\mathrm{Lk}(\{u, v\}, \Delta_1)$ is null.

Then, the following equalities between the B-splines defined with respect to Δ_1 and the vertex and edge tents from Δ_0 are satisfied, as can be easily checked by expanding the expression for B-spline using Michelli recurrence (6.3):

- For a vertex v of Δ_0, let $\{v_0, v_1\}$ be the edge from pulling apart v. Then, in the limit,
$$B_{v_0,v_1}(\cdot) = \text{tent}_v(\cdot);$$

- For an edge $[u, v]$ of Δ_0, let $\{u_0, u_1\}$ and $\{v_0, v_1\}$ be the edge from pulling apart u and v, respectively. Then, if there is one edge between $\{u_0, u_1\}$ and $\{v_0, v_1\}$, namely, $\{u, v\}$, then, in the limit,
$$B_{u,v}(\cdot) = \text{tent}_{u,v}(\cdot);$$
otherwise, there are two exactly two edges between $\{u_0, u_1\}$ and $\{v_0, v_1\}$ whose links in Δ_1 are not null, namely $\{u_0, v_1\}$ and $\{u_1, v_0\}$ and
$$B_{u_0,v_1}(\cdot) + B_{u_1,v_0}(\cdot) = \text{tent}_{u,v}(\cdot)$$

The above equalities proves that the B-splines reproduce the C^0 Bezier patches, as stated more formally:

8.3.1 Application: representing sharp features

In the univariate setting, B-splines are optimally smooth over knots in generic positions—i.e., no duplicates are allowed—but have reduced smoothness where knots are in degenerate positions. Specifically, if a knot has multiplicity i, then the smoothness of the B-spline at the knot is lowered by i. For a curve designer, this means that B-splines are “generically smooth” but can be made sharp locally by duplicating knots. It seems that analogous properties in the bivariate setting will be useful for a surface designer: Imagine a bivariate spline representation that is “generically smooth” but allow sharp corners or creases to appear by introducing degenerate configurations. I will show how this can be achieved, in the quadratic case, by using bivariate B-splines.

The main idea is to locally apply variations of the reproduction rule for C^0 Bezier patches described earlier. For the rest of the section, I describe scenarios of modeling sharp features and show how to realize the scenarios with B-splines. Figure **??** illustrate the various scenarios. A standing assumption: Δ_0 denotes a triangulation whose vertices are in generic position; Δ_1 denotes a set of configurations from Δ_0; Whenever Δ_0 is modified, the result is denoted $\widetilde{\Delta_0}$.

Scenario 1: Modeling a sharp corner on the B-spline surface at vertex v of Δ_0. First, pull apart the vertex v to an edge $\{v_0, v_1\}$. This creates a pulled-apart triangulation $\widetilde{\Delta_0}$. The degree one configurations Δ_1 over Δ_0 is modified by deleting all configurations with the interior set v and triangulating the links of v_0 and v_1 as directed by Figure 8.7. The B-splines are then collected around v to reproduce the vertex tent for v and the edge tents for all edges incident on v, so that the resulting basis is:

$$\{B_e\}_{e\in E(\Delta_0), e\not\ni v} \bigcup \{\text{tent}_e\}_{e\in E(\Delta_0), e\ni v} \bigcup \{\text{tent}_v\}$$

It should be noted that the function space spanned by the above splines is *interpolatory* at v: If the coefficient of tent_v is set to λ, then evaluating any function from the space gives precisely λ.

Scenario 2: Modeling a crease along an edge $\{u, v\}$ of Δ_0. This can be done most simply by performing the operations in Scenario 1 for both vertex u and v, which gives a crease that is sharp at both ends. To make a crease that is not sharp at both ends, first choose a point w in the interior of the edge $\{u, v\}$, subdivides the edge $\{u, v\}$ to $\{u, w\}$ and $\{w, v\}$ and the two triangles $\{u, v, a\}$ and $\{u, v, b\}$ incident on $\{u, v\}$ to four: $\{u, w, a\}$, $\{u, w, b\}$, $\{v, w, a\}$ and $\{v, w, b\}$. Second, retriangulate the vertex links of u, v, w, a and b in the new triangulation $\widetilde{\Delta_0}$, treating w specially: the vertex link of w, which is the quadrilateral (u, a, v, b), is triangulated by drawing the diagonal from u to v. The, the following set of splines from collecting the B-splines $\{B_e\}_{e\in E(\widetilde{\Delta_0})}$, is used as basis:

$$\{B_e\}_{e\in E(\widetilde{\Delta_0}), e\neq\{u,w\}, e\neq\{v,w\}} \bigcup \{B_{u,w} + B_{v,w}\}$$

The set has exactly one member that is not smooth, namely, $B_{u,w} + B_{v,w}$, which is sharp along the edge $\{u, v\}$.

Scenario 3: Interpolating linear elements. One can imagine scenarios where it is necessary to create linear elements on an otherwise smooth B-spline—for example, to represent man-made structures on a natural terrain. Such scenario can be supported by locally reproducing C^0 quadratic Bezier patches. The input linear elements, L, are represented as a set of edges and triangles in a triangulation, where each vertex v of L is associated with a height value λ_v. To interpolate the 3D line segments and triangles represented by L, the first step is to pull apart the vertices of L to reproduce the vertex and edge tent functions as specified in the previous section. The second step is to modify the tent functions as follows: For each edge $\{u, v\}$ of L—whether $\{u, v\}$ is an isolated edge in L or belongs to a triangle in L, replace the functions tent_u, tent_v and $\text{tent}_{u,v}$ by the functions $\text{tent}_u + \text{tent}_{u,v}/2$ and $\text{tent}_v + \text{tent}_{u,v}/2$ (Note that there is no longer a basis function associated with the edge). Then, setting the coefficients of $\text{tent}_u + \text{tent}_{u,v}/2$ and $\text{tent}_v + \text{tent}_{u,v}/2$ to be λ_u and λ_v interpolate the 3D line segment $\{(u; \lambda_u), (v; \lambda_v)\}$. If three edges, say $\{u, v\}$, $\{v, w\}$ and $\{u, w\}$, surround a triangle, then the 3D triangle $\{(u; \lambda_u), (v; \lambda_v), (w; \lambda_w)\}$ will also be interpolated.

8.4 Data fitting with bivariate B-splines

A scattered data fitting problem is typically described as follows: In s dimensions, for a set of data locations $P \subset \mathbb{R}^s$ and a set of data values $\{f_p\}_{p \in P} \subset \mathbb{R}$, first choose an appropriate space of s-variate functions $\text{span}\{g_v\}_{v \in V}$, where V is the index set of the basis, and then choose a function from this space by solving the system of equations $\{\sum_{v \in V} \lambda_v g_v(p) = f_p\}_{p \in P}$ for the coefficients $\{\lambda_v\}_{v \in V}$.

Common choices of function spaces used for this problem include *interpolatory* functions, such as PL interpolations and Lagrange polynomials, and *radial basis functions*, where each basis function is centered about a point and is parametrized by the Euclidean distance to the center point. For both of these spaces, the basis are naturally associated with the data points, i.e. $V = P$. In other words, it is easy to construct the basis according to the data locations, which is important for fitting scattered data.

B-splines can also be used for scattered data fitting. Compared with interpolatory or radial basis functions, B-splines have the advantage that they have compact and local support, which means that both the evaluation of B-splines and solving the system of interpolation equations can be performed efficiently. However, it is more difficult to construct the B-spline basis according to the data points. Specifically, for a given data set, one must choose an appropriate set of knots $K \subset \mathbb{R}^s$ so that the B-splines from K satisfy certain interpolation properties with respect to the data set. A basic interpolation property is that the system of interpolation equations with the B-splines has a unique solution. In the univariate setting, for the unique solution to exist, the following necessary and sufficient condition in the well known Schoenberg-Whitney Theorem () must be satisfied: denoting the ordered knot set and data locations by $K = \{\ldots < K_i < K_{i+1} \leq \ldots\}_{i \in \mathbb{Z}}$ and $P = \{\ldots < P_i < P_{i+1} \leq \ldots\}_{i \in \mathbb{Z}}$, for any $i \in \mathbb{Z}$, the inequality $K_i \leq P_i \leq K_{i+k}$ must hold. Additional interpolation properties can be stated in terms of some measure of the fitted function—for example, the total curvature. Such measures naturally lead to the study of optimization problems which try to achieve the best measure over all possible placement of knots (Jupp, 1978). In the bivariate setting, because the B-splines are so new, not much is known about their interpolation properties—not even the analog of Schoenberg-Whitney Theorem. So far, the only study is carried out by Dembart et al. (2004), who study experimentally the goodness of the new B-splines for data fitting. For that purpose, they forgo the more difficult problem of knot positioning and study a more restricted data fitting problem, where the data locations are the *Greville sites.* The work in this section extends their experimental study to B-splines associated with non-Delaunay configurations.

Greville sites are the centroids of the indices of B-splines: For a set of B-splines $\{B_I\}_{I\in V}$, its Greville sites are the set of points $\{\overline{I}\}_{I\in V}$. The restricted data fitting problem requires that the data locations are the same as the Greville sites, i.e. $P = \{\overline{I}\}_{I\in V}$. Another way to describe the restricted problem is that, instead of assuming that the input for interpolation is a discrete set of points, assume that the input is a continuous function $f : \mathbb{R}^s \to \mathbb{R}$ and the goal is to interpolate f at a discrete set of points P with a B-spline to obtain an approximation. In this setting, the points P are called *collocation sites.* It is well-known that using Greville sites as the collocation sites gives good interpolation, although there is no proven optimal property. To understand why Greville sites give good interpolation, it is helpful to look at a picture of B-splines to usee why: A B-spline B_I, whether in the univariate or bivariate setting, typically achieves the maximal around the Greville site $\overline{I}$.

It should also be mentioned that while it is tempting to use the data locations for knots, this generally does not lead to good interpolation. In particular, in the bivariate setting, for degree k, there are roughly k times as many B-splines as knots therefore, using data locations for knots lead to severely under-determined system of interpolation equations.

In Dembart et al.'s study (2004), the knots are either positioned on a grid or randomly; the input function f is chosen to be either $\sin(xy)e^{x+y}$ or the Franke function; the interpolation (with B-splines at Greville sites) is measured in two ways: the approximation error, defined as the maximal over a set of test locations on a grid, and the condition number of the interpolation matrix. They found that the converge rate, calculated from the measured approximate errors, is in the order of h^3.

As a continuation of the study, I perform similar experiments with B-splines associated with the generalized configurations. The main objective is to study the effect of element shapes on the quality of the B-splines for interpolation. For this purpose, two input data are used: The bivariate functions $\sin(xy)e^{x+y}$ and $\sin(\pi x)$ over the domain $[0, 1] \times [0, 1]$. The second function is *anisotropic*—Its gradients have a dominant direction, namely the direction of the x-axis. The results are shown in Figure 8.8. There, for $\sin(xy)e^{x+y}$, three basis are used for interpolation: The first comes Delaunay configurations; the second and third are constructed by first randomly flipping edges of the Delaunay triangulation. For $\sin(\pi x)$, the first basis comes Delaunay configurations; the second basis is constructed by first flipping edges of the Delaunay triangulation to align with the y-axis. To reduce the boundary effect, the configurations around the boundaries are constructed in a special way in order to locally reproduce Bezier patches, as described in Section 8.3, so that each boundary vertex or edge is associated with an interpolatory basis function.

The results show that, just as in the PL case, although Delaunay configurations generally give good interpolation, for anisotropic functions, aligning the configurations orthogonal to the dominant gradient direction gives better interpolation.

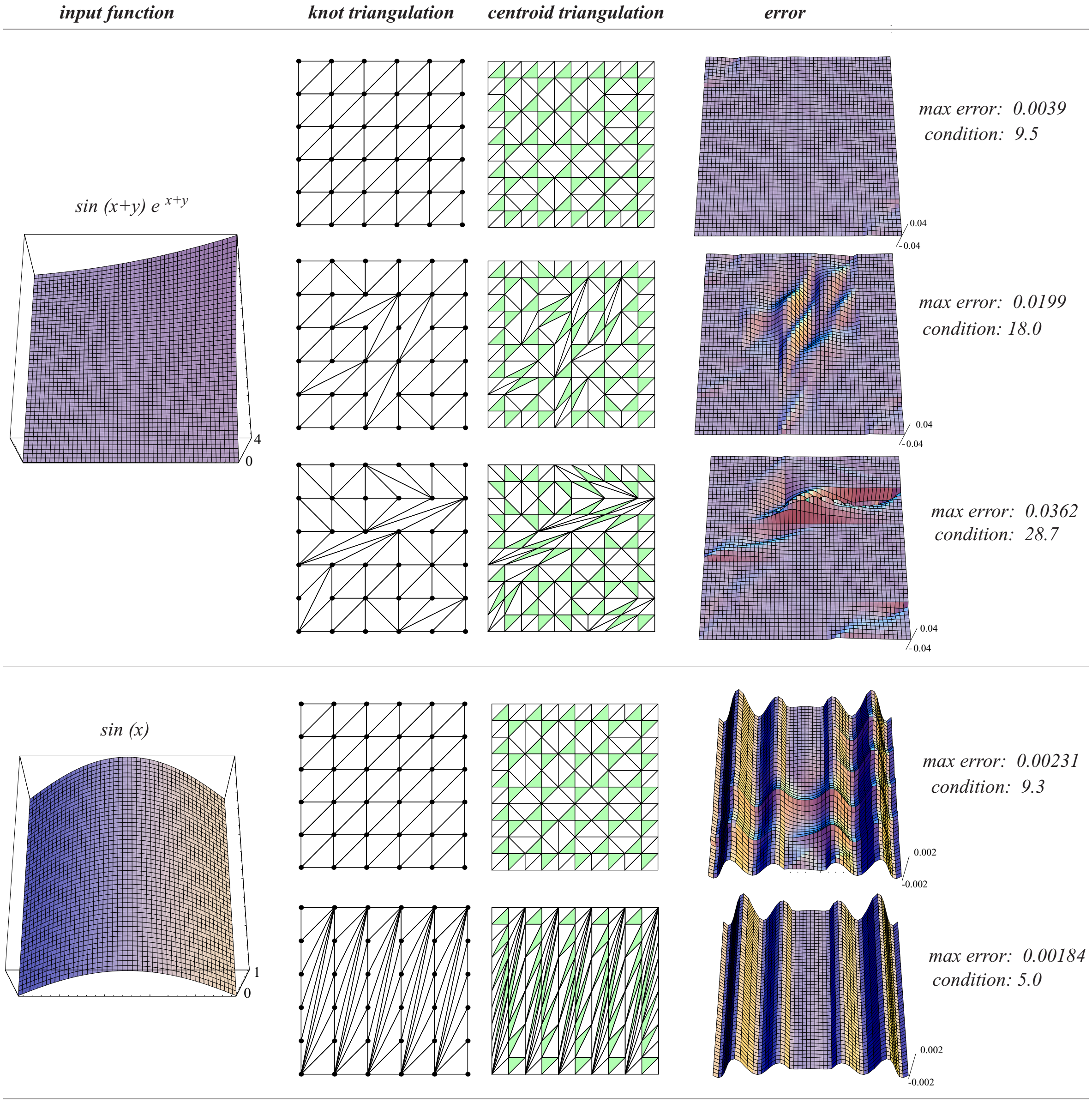

Figure 8.8: Interpolating the bivariate functions $\sin(xy)e^{x+y}$, above, and $\sin(\pi x)$, below, with B-splines defined over the knot set $\{1..6\} \times \{1..6\}$. For each interpolation, shown from left to right on a row are: the knot triangulation, the centroid triangulation from the knot triangulation and the degree one configurations, the error measured over a grid, the maximum error and the condition number of the interpolation matrix.

Bibliography

(1985). Ieee standard for binary floating point arithmetic, ANSI/IEEE std $754 - 1985$. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

Agarwal, P. K., Arge, L., and Yi, K. (2005). I/O-efficient construction of constrained Delaunay triangulations. In *Proceedings of the Thirteenth European Symposium on Algorithms*, pages 355–366, Mallorca, Spain.

Alber, J. and Niedermeier, R. (1998). On multi-dimensional hilbert indexings. In *Computing and Combinatorics*, pages 329–338.

Alliez, P., Devillers, O., and Snoeyink, J. (2000). Removing degeneracies by perturbing the problem or the world. *Reliable Computing*, 6:61–79. Special Issue on Computational Geometry.

Amenta, N., Choi, S., and Rote, G. (2003a). Incremental constructions con BRIO. In *Proceedings of the Nineteenth ACM Symposium on Computational Geometry*, pages 211–219.

Amenta, N., Choi, S., and Rote, G. (2003b). Incremental constructions con BRIO. In *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, pages 211–219, San Diego, California. Association for Computing Machinery.

Andrzejak, A. and Welzl, E. (2003). In between k-sets, j-facets, and i-faces: (i,j)-partitions. *Discrete and Computational Geometry*, 29(1):105–131.

Attali, D., Boissonnat, J.-D., and Lieutier, A. (2003). Complexity of the Delaunay triangulation of points on surfaces: the smooth case. In *Proceedings of the Nineteenth ACM Symposium on Computational Geometry*, pages 201–210.

Aurenhammer, F. (1987). Power diagrams: properties, algorithms and applications. *SIAM J. Comput.*, 16:78–96.

Avnaim, F., Boissonnat, J.-D., Devillers, O., Preparata, F., and Yvinec, M. (1997). Evaluating signs of determinants using single precision arithmetic. *Algorithmica*, 17(2):111–132.

Ban, Y.-E. A., Edelsbrunner, H., and Rudolph, J. (2004). Interface surfaces for protein-protein complexes. In *Proceedings of the eighth annual international conference on Computational molecular biology*, pages 205–212.

Bandyopadhyay, D. and Snoeyink, J. (2004). Almost-delaunay simplices: nearest neighbor relations for imprecise points. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 410–419, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483. `http://www.qhull.org/`.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N., and Bourne, P. E. (2000). The protein data bank. *Nucleic Acids Research*, 28(1):235–242. `http://www.rcsb.org`.

Bern, M. and Eppstein, D. (1992). Mesh generation and optimal triangulation. In Du, D.-Z. and Hwang, F. K., editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 23–90. World Scientific, Singapore.

Blandford, D. K., Blelloch, G. E., Cardoze, D. E., and Kadow, C. (2005). Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications*, 15(1):3–24.

Boissonnat, J.-D., Devillers, O., Pion, S., Teillaud, M., and Yvinec, M. (2002). Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19.

Boissonnat, J.-D. and Yvinec, M. (1998). *Algorithmic geometry*. Cambridge University Press, UK. Translated by Hervé Brönnimann.

Bowyer, A. (1981). Computing Dirichlet tesselations. *Comput. J.*, 24:162–166.

Brown, K. Q. (1980). *Geometric transforms for fast geometric algorithms*. Ph.D. thesis, Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA. Report CMU-CS-80-101.

Cignoni, P., Montani, C., and Scopigno, R. (1998). DeWall: A fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design*, 30(5):333–341.

Clarkson, K. L. (1988). A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847.

Clarkson, K. L. (1992). Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395. `http://cm.bell-labs.com/netlib/voronoi/hull.html`.

Crauser, A., Ferragina, P., Mehlhorn, K., Meyer, U., and Ramos, E. A. (2001). Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry and Applications*, 11(3):305–337. Special issue edited by J Hershberger: Selected Papers from the Fourteenth ACM Symposium on Computational Geometry, Minneapolis, MN, Jtune, 1998.

Dahmen, W., Micchelli, C. A., and Seidel, H.-P. (1992). Blossoming begets B-spline bases built better by B-patches. *Math. Comp.*, 59(199):97–115.

Dahmen, W. A. and Micchelli, C. A. (1983). On the linear independence of multivariate b-splines ii: complete configurations. *Math. Comp.*, 41(163):143–163.

de Boor, C. (1976). Splines as linear combinations of B-splines. a survey. *Approximation Theory, II*, pages 1–47.

de Boor, C., Höllig, K., and Riemenschneider, S. D. (1993). *Box Splines.* Springer-Verlag, New York.

Dembart, B., Gonsor, D., and Neamtu, M. (2004). Bivariate quadratic B-splines used as basis functions for data fitting. to appear, `http://math.vanderbilt.edu/~neamtu/papers/papers.html`.

Devillers, O. (1998). Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115.

Devillers, O. (2002). The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13:163–180.

Devillers, O. and Pion, S. (2003). Efficient exact geometric predicates for Delaunay triangulations. In *Proc. 5th Workshop Algorithm Eng. Exper.*, pages 37–44.

Devillers, O., Pion, S., and Teillaud, M. (2002). Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199.

Devillers, O. and Teillaud, M. (2003). Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 313–319.

Dwyer, R. A. (1991). Higher-dimensional Voronoi diagrams in linear expected time. *Discrete Comput. Geom.*, 6:343–367.

Dyn, N. and Rippa, S. (1993). Data-dependent triangulations for scattered data interpolation and finite element approximation. *Applied Numerical Mathematics*, 12(1-3):89–105.

Edelsbrunner, H. (1987). *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany.

Edelsbrunner, H. (1989). An acyclicity theorem for cell complexes in d dimensions. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 145–151.

Edelsbrunner, H., Harer, J., and Zomorodian, A. (2003). Hierarchical Morse-Smale complexes for piecewise linear 2-manifolds. *Discrete and Computational Geometry*, 30:87–107.

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

Edelsbrunner, H. and Mücke, E. P. (1994). Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72.

Edelsbrunner, H. and Shah, N. R. (1992). Incremental topological flipping works for regular triangulations. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 43–52.

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

Erickson, J. (2002). Dense point sets have sparse Delaunay triangulations. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 125–134. Society for Industrial and Applied Mathematics.

Farin, G. (1997). *Curves and Surfaces for CAGD: A Practical Guide*, pages 141–170. Morgan-Kaufmann.

Fortune, S. (1992). Voronoi diagrams and Delaunay triangulations. In Du, D.-Z. and Hwang, F., editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 193–233. World Scientific, Singapore.

Garland, M. and Heckbert, P. S. (1995). Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Dept. Computer Science, Carnegie-Mellon University.

Gray, F. (1953). Pulse code communication. United States Patent Number 2632058.

Guibas, L. J., Knuth, D. E., and Sharir, M. (1992). Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413.

Haber, J., Zeilfelder, F., Davydov, O., and Seidel, H. P. (2001). Smooth approximation and rendering of large scattered data sets. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 341–348, Washington, DC, USA. IEEE Computer Society.

Isenburg, M. and Lindstrom, P. (2005). Streaming meshes. In *Visualization '05 Proceedings*, pages 231–238, Minneapolis, Minnesota.

Isenburg, M., Liu, Y., Shewchuk, J., and Snoeyink, J. (2006). Streaming computation of delaunay triangulations. submitted to SIGGRAPH '06.

Jupp, D. L. B. (1978). Approximation to data by splines with free knots. *SIAM Journal on Numerical Analysis*, 15(2):328–343.

Katajainen, J. and Koppinen, M. (1988). Constructing Delaunay triangulations by merging buckets in quadtree order. *Fundamenta Informaticae*, XI(11):275–288.

Kettner, L., Rossignac, J., and Snoeyink, J. (2003). The Safari interface for visualizing time-dependent volume data using iso-surfaces and a control plane. *Comp. Geom. Theory Appl.*, 25(1-2):97–116.

Koehl, P., Levitt, M., and Edelsbrunner, H. (2002). Proshape. `http://biogeometry.duke.edu/software/proshape/`.

Lee, D. T. (1982). On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Comput.*, C-31:478–487.

Liang, J., Edelsbrunner, H., , and Woodward, C. (1998). Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design. *Protein Science*, 7:1884–1897.

Liotta, G., Preparata, F. P., and Tamassia, R. (1997). Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165.

Liu, Y. and Snoeyink, J. (2005a). A comparison of five implementations of 3d Delaunay tessellation. *Combinatorial and Computational Geometry, MSRI series*, pages 439–458.

Liu, Y. and Snoeyink, J. (2005b). TESS3: a program to compute 3d Delaunay tessellations for well-distributed points. In *International Symposium on Voronoi Diagrams in Science and Engineering*.

Liu, Y. and Snoeyink, J. (2006a). Faraway point: A sentinel point for Delaunay computation. *International Journal of Computational Geometry and Applications*. accepted.

Liu, Y. and Snoeyink, J. (2006b). Quadratic and cubic b-splines by generalizing higher-order voronoi diagrams. In *Symposium on Computational Geometry*.

Liu, Y. and Snoeyink, J. (2006c). Sphere based computation of Delaunay diagrams on points from 4d grids. In *International Symposium on Voronoi Diagrams in Science and Engineering*.

McMullen, P. (1970). The maximal number of faces of a convex polytope. *Mathematika*, 17:179–184.

Melquiond, G. and Pion, S. (2005). Formally certified floating-point filters for homogeneous geometric predicates. Research Report 5644, INRIA.

Mitasova, H. and Mitas, L. (1993). Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical Geology*, 25(6):641–655.

Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. (2001). Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141.

Mücke, E. P. (1998). A robust implementation for three-dimensional Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 8(2):255–276.

Mücke, E. P., Saias, I., and Zhu, B. (1996). Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283.

Neamtu, M. (2001). What is the natural generalization of univariate splines to higher dimensions? *Mathematical Methods for Curves and Surfaces: Oslo 2000*, pages 355–392.

Neamtu, M. (2004). Delaunay configurations and multivariate splines: A generalization of a result of B. N. Delaunay. *Trans. Amer. Math. Soc.* to appear, `http://math.vanderbilt.edu/~neamtu/papers/papers.html`.

Niedermeier, R., Reinhardt, K., and Sanders, P. (2002). Towards optimal locality in mesh-indexings. *Discrete Applied Mathematics*, 117(1-3):211–237.

Pajarola, R. (2005). Stream-processing points. In *Visualization '05 Proc.*, pages 239–246, Minneapolis, Minnesota.

Paoluzzi, A., Bernardini, F., Cattani, C., and Ferrucci, V. (1993). Dimension-independent modeling with simplicial complexes. *ACM Trans. Graph.*, 12(1):56–102.

Pion, S. (1999). Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to systems and Control*, pages 99–110.

Prautzsch, H., Boehm, W., and Paluszny, M. (2002). *Bezier and B-Spline Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Richards, F. M. (1974). The interpretation of protein structures: total volume, group volume distributions, and packing density. *J. Molecular Biology*, 82(1).

Riesenfeld, R. F. (1981). Homogeneous coordinates and projective planes in computer graphics. *IEEE Comput. Graph. Appl.*, 1:50–55.

Schmitt, D. and Spehner, J.-C. (2006). k-set polytopes and order-k Delaunay diagrams. In *International Symposium on Voronoi Diagrams in Science and Engineering*, pages 173–185.

Seidel, R. (1991). Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434.

Shaffer, C. A. (1990). Fast circle-rectangle intersection checking. In *Graphics Gems*, pages 51–53. Academic Press Professional, Inc., San Diego, CA, USA.

Shamos, M. I. and Hoey, D. (1975). Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162, Berkeley, California. IEEE Press.

Shewchuk, J. (2002). What is a good linear finite element? Interpolation, conditioning, anisotropy, and quality measures. The manuscript can be downloaded at `http://www.cs.cmu.edu/~jrs/jrspapers.html#quality`. A shorter version is published at the 11th International Meshing Roundtable.

Shewchuk, J. R. (1996a). Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150.

Shewchuk, J. R. (1996b). Triangle: engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*. Association for Computing Machinery.

Shewchuk, J. R. (1996c). Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag.

Shewchuk, J. R. (1997). Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363.

Shewchuk, J. R. (1998a). Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 86–95.

Shewchuk, J. R. (1998b). Tetrahedral mesh generation by Delaunay refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 86–95, Minneapolis, Minnesota. Association for Computing Machinery.

Stolfi, J. (1991). *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, New York, NY.

Watson, D. F. (1981). Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Comput. J.*, 24(2):167–172.

Watson, D. F. (1992). *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon.

Wendland, H. (2004). *Scattered Data Approximation*. Cambridge University Press.

Ziegler, G. M. (1994). *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg.