Typeset with NDThesiS version 2.18 (2004/04/05)
on February 3, 2006

for

Joseph E. Lammersfeld

entitled

IMPLEMENTING FOUR-DIMENSIONAL TRIANGULATIONS IN CGAL

This class conforms to the University of Notre Dame style guidelines established Spring 2004. However it is still possible to generate a non-conformant document if the published instructions are not followed! Be sure to refer to the published Graduate School guidelines as well.

THIS IS A TEMPORARY VERSION OF THIS CLASSFILE. IT IS ONLY INTENDED TO BE USED FOR DISSERTATIONS IN THE SPRING OF 2004. A new version of this classfile will be available after that, and should be used for all future dissertations.

This summary page can be disabled by specifying the nosummary *option to the class invocation.*(i.e., \documentclass[nosummary]{ndthesis})

THIS PAGE IS *NOT* PART OF THE THESIS, BUT SHOULD BE TURNED IN TO THE PROOFREADER!

NDThesiS documentation can be found at these locations:

`http://www.nd.edu/~afsunix/faq/tetexdoc/latex/ndthesis/`
`http://www.cse.nd.edu/~jsquyres/ndthesis/`

General LaTeX documentation and info:

On-line docs:	
ND installation	`http://www.nd.edu/~afsunix/faq/tetexdoc/`
TeX User's Group	`http://www.tug.org/`
Books:	
A Guide...for Beg. & Adv. Users	by Kopka/Daly
LaTeX User's Guide ...	by Lamport
The LaTeX Companion	by Goossens/Mittelbach/Samarin
Packages: (check on-line docs)	
rotating	sideways tables and figures
longtable	multi-page tables
graphicx	using Postscript and other figures

IMPLEMENTING FOUR-DIMENSIONAL TRIANGULATIONS IN CGAL

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

by

Joseph E. Lammersfeld, B.S.

Danny Z. Chen, Ph.D., Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
February 2006

IMPLEMENTING FOUR-DIMENSIONAL TRIANGULATIONS IN CGAL

Abstract

by

Joseph E. Lammersfeld

This thesis presents the implementation of a four-dimensional triangulation package. Computation of both basic triangulations of point sets and regular triangulations of weighted point sets using an incremental insertion algorithm is presented. The regular triangulation is the dual of the power diagram, which is the Voronoi diagram of weighted points under the power distance. The implementation follows the Computational Geometry Algorithms Library (CGAL), and will be contributed.

A key part of the package lies with the data structure, which allows the user to easily traverse the constructed triangulations using circulators, enumerators, and iterators. Additionally, the geometric predicates are generalized from their lower-dimensional counterparts. Arithmetic filtering is used to keep the predicates exact, yet efficient. The result is a triangulation package that is easy to use, efficient, and robust.

To everyone who patiently waited for me while I completed this thesis, although the work will never feel complete.

CONTENTS

FIGURES

TABLES

SYMBOLS

$\mathbb{R}^d$	Euclidean space of dimension d
p_i	Point
$P_i = \{p_i, w_i\}$	Weighted point
$\mathcal{P}$	Set of points or weighted points
$dist$	Euclidean distance function
pow	Power distance function
$Reg(p_i)$	Voronoi region of point site p_i
$Reg(P_i)$	Power region of weighted point site P_i
$VD_{\mathcal{P}}$	Voronoi diagram of point set $\mathcal{P}$
$PD_{\mathcal{P}}$	Power diagram of weighted point set $\mathcal{P}$
$\mathcal{T}_{\mathcal{P}}$	Triangulation of point set $\mathcal{P}$
$\mathcal{DT}_{\mathcal{P}}$	Delaunay triangulation of point set $\mathcal{P}$
$\mathcal{RT}_{\mathcal{P}}$	Regular triangulation of weighted point set $\mathcal{P}$

ACKNOWLEDGMENTS

Everyone in the Computer Science and Engineering Department has been supportive of my unusual advising situation. Danny Chen has been especially supportive as my advisor after Menelaos left, and I owe him much gratitude for his time and good advice. I thank Menelaos Karavelas for introducing me to CGAL, answering my questions from halfway around the world, and returning at the right time to push me to the finish line. Finally, I must thank Kevin Bowyer for giving me the opportunity to teach two courses at Notre Dame in 2005. I am confident that these experiences will be useful in my future.

The Computer Science and Engineering Department and the Arthur J. Schmitt Fellowship need to be acknowledged for providing financial support of my graduate education.

The following software was used in some way. Most importantly, the Computational Geometry Algorithms Library (CGAL) (including the Windows demo programs). MWSnap was used for taking screen captures of the CGALdemos, and Ipe was used to produce several figures.

CHAPTER 1

INTRODUCTION

1.1 Voronoi diagrams and Delaunay triangulations

The Voronoi diagram is one of the most fundamental and well studied geometric data structures in the fields of computational geometry and computer science. Suppose you are given a set of geometric objects (called input sites) in a space. Assigning every point of the space to the closest site under a distance function results in a partition of the space into regions where each region is associated with one of the input sites. This is called the Voronoi assignment model, and the resulting space partitioning is called the Voronoi diagram. Because of the Voronoi assignment model, the points in the region of input site S_i are closer to S_i than to any other input site.

The most well known and studied Voronoi diagram is for the case of point sites in the plane under the usual Euclidean distance function (L_2-metric). This was introduced to computer science and computational geometry by Shamos and Hoey [37], along with a divide-and-conquer algorithm to compute it. Fortune [22, 23] describes a sweepline algorithm to compute it. In this case, the Voronoi diagram consists of convex regions bounded by segments, half-lines, or lines. Also, it is related via graph duality to the Delaunay triangulation of the point sites. The Delaunay triangulation has the property that the circumscribing circle of each triangle is empty of any other site. Properties, data structures, algorithms, and geometric predicates

for the Voronoi diagram and Delaunay triangulation of point sites in $\mathbb{R}^2$ have been collected in computational geometry textbooks [12, 15, 33, 35].

It is quite natural to generalize the Voronoi diagrams and their duals by introducing new types of sites, changing the distance function, or increasing the dimension. Properties for many such generalizations have appeared in the literature along with data structures and algorithms for representing and constructing them. To enumerate this work is beyond the scope of this thesis. Fortune [24] summarizes the work for the case of point sites in general dimensions. Aurenhammer [6] discusses Voronoi diagrams and generalizations from both mathematical and computer science perspectives, enumerates several applications, describes algorithms, and provides an extensive bibliography (see also [7, 34]). Voronoi diagrams are intimately related to triangulations of the sites via graph duality, and we will examine one such generalization of a Voronoi diagram and its dual in this thesis: the power diagram and the regular triangulation in $\mathbb{R}^4$.

1.2 Motivation and applications

Power diagrams are known to be generalizations of several other types of Voronoi diagrams. Therefore, constructing these other Voronoi diagrams can be reduced to constructing a power diagram (perhaps in a higher dimension). For example, power diagrams in $\mathbb{R}^{d+1}$ are related to Voronoi diagrams of spheres[1] in $\mathbb{R}^d$ (cf. [4]). Computing the Voronoi diagram of spheres has applications in biology and chemistry, where atoms are represented as spheres[2]. Power diagrams in $\mathbb{R}^{d+1}$ are

[1] The Voronoi diagram of spheres is also known as the additively weighted Voronoi diagram or the Apollonius diagram.

[2] According to Will [39], biologists have used several different space partitioning techniques in order to answer questions about proteins using a geometric approach. Using the point set Voronoi diagram is not desirable because it does not take into consideration the relative differences in size between atoms. Goede et al. [26] proposes the use of the Voronoi diagram of spheres as an accurate model for space partitioning among atoms in order to compute the volume occupied by atoms and to estimate the densities of proteins.

also related to Möbius diagrams in $\mathbb{R}^d$ in which doubly weighted points are used as sites (cf. [11]). As a final example, the skew Voronoi diagram[3] is a special case of the Voronoi diagram of spheres [1, 2]. Therefore, skew Voronoi diagrams are related indirectly to power diagrams via Voronoi diagrams of spheres.

The computation of the power diagram is a problem of interest for direct application as well. Aurenhammer [4] describes applications for sphere packing and illuminating balls. Aurenhammer [5] also uses power diagrams for computing the union and intersection of spheres. Imai et al. [30] discuss applications to finding connected components of circles, finding the contour of the union of circles, and testing whether or not a query point is in the union of circles.

1.3 Problem description

In this thesis, we consider only Euclidean spaces, sometimes in a general sense as in $\mathbb{R}^d$ and other times in a specific dimension such as $\mathbb{R}^2$, $\mathbb{R}^3$, or $\mathbb{R}^4$. The type of site of the power diagram is the weighted point, which consists of a point in $\mathbb{R}^d$ and an additional real value that is the weight of the point. Let $P_i = \{p_i, w_i\}$ be a weighted point and let p be a point. When $w_i \geq 0$, P_i can be thought of in a geometric sense as a sphere with center p_i and radius $\sqrt{w_i}$. Let l be a line through p and intersecting the sphere at points B and C, which may be the same point if p lies outside the sphere. Then the power distance is defined as $pow(p, P_i) = pB \cdot pC$. The power distance can be written algebraically as $pow(p, P_i) = dist^2(p, p_i) - w_i$, where $dist$ is the Euclidean distance between two points. Under this interpretation, the power distance from a point p outside sphere P_i is positive and equal to the square of the distance from p to P_i along a tangent line to P_i. See Figure 1.1 for a geometric illustration. If p lies on sphere P_i then the power distance is zero, and if p lies inside

[3] Also known as the Voronoi diagram for direction sensitive distances.

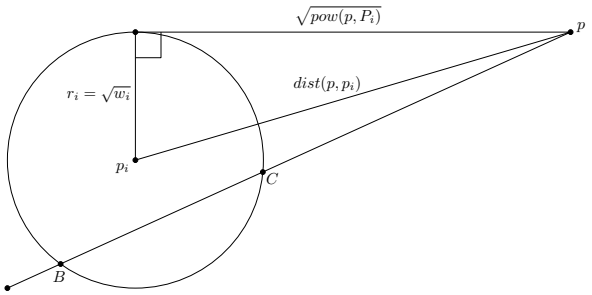


Figure 1.1. An illustration of the geometric interpretation of the power distance function. The sphere on the left represents the weighted point $P_i = \{p_i, w_i\}$ with positive weight w_i and radius $r_i = \sqrt{w_i}$. By drawing a line from p intersecting P_i at B and C, $pow(p, P_i)$ can be computed as $pB \cdot pC$. When the line becomes tangent to the sphere, B is the same point as C and $pow(p, P_i)$ becomes $dist^2(p, p_i) - r_i^2$.

sphere P_j, then the power distance is negative. The power product between two weighted points P_i and P_j is defined as $pow(P_i, P_j) = dist^2(p_i, p_j) - w_i - w_j$. Two weighted points are said to be orthogonal if their power product equals zero.

Now we define the power diagram, whose definition can be generalized to other types of Voronoi diagrams quite easily. Let $\mathcal{P} = \{P_1, ..., P_n\}$ be a set of n weighted point input sites. Given two sites $P_i, P_j \in \mathcal{P}$, let $H_{ij} = \{p \in \mathbb{R}^d | pow(p, P_i) \leq pow(p, P_j)\}$ be the set of points closer to P_i than to P_j. Then the Voronoi region of site P_i is $Reg(P_i) = \bigcap_{j \neq i} H_{ij}$. In other words, the Voronoi region of P_i is formed as the intersection of H_{ij} for every other input site P_j. The collection of regions forms the Voronoi diagram of the sites: $V_{\mathcal{P}} = \bigcup_{i=1}^{n} Reg(P_i)$. Weighted points belonging to two regions lie on bisectors, which bound the regions. The bisector between two weighted point sites P_i and P_j is the set of points equidistant to P_i and P_j, and in the case of the power diagram it is a hyperplane. In this sense, the power diagram of a set of weighted point sites generates a space partitioning into convex regions similar to that generated under the point set Voronoi assignment model. In addition, it is possible for an input site to have an empty power cell. In this

case, the site is termed hidden. If a site P_i is hidden, then for every point p in the space there exists another site P_j such that $pow(p, P_j) < pow(p, P_i)$. In other words, there are no points that are closer to P_i than to any other site. In this case, the power diagram differs from the ordinary Voronoi diagram because sites of an ordinary Voronoi diagram are never hidden. Sites that are not hidden are called visible.

The dual of the power diagram is called the regular triangulation. The regions of the power diagram are in one-to-one correspondence with the (visible) vertices of the regular triangulation. When two regions are adjacent through a facet in the power diagram, the corresponding weighted points are connected by an edge in the regular triangulation. Also, the vertices of the power diagram (weighted points that are equidistant to $d+1$ sites under the power distance function) are in one-to-one correspondence with the cells of the regular triangulation. This is conditioned on a general position assumption on the set of input sites. However, the implementation provided by this thesis will impose no such assumptions on the input sites, and correctly handle degenerate situations when they arise. When degeneracies are detected (for example when there is a weighted point that is equidistant to more than $d+1$ sites under the power distance), an implicit perturbation scheme is used and the resulting triangulation depends on the order of insertion of the sites.

Dual structures of Voronoi diagrams have properties that often make them easier to represent and manipulate from an implementation point of view. For example, consider the planar Delaunay triangulation mentioned previously. It consists of (constant size) triangles, whereas the Voronoi diagram consists of convex polygons. The complexity of the entire Voronoi diagram can be concentrated in one Voronoi region (in other words, one region can have $O(n)$ edges, where n is the number of input sites). Also, the Delaunay triangulation is always connected, whereas the

Voronoi diagram may consist of disconnected components for some sets of point sites. In this thesis, implementation details for the power diagram will be based on the dual regular triangulation, but theoretical results and properties may come out of either the primary or the dual structure.

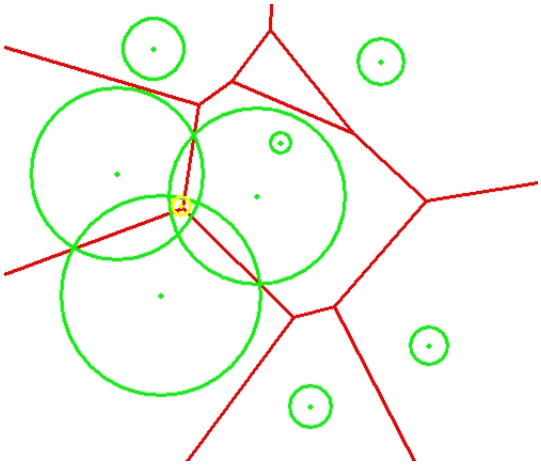
The Computational Geometry Algorithms Library (CGAL) is a C++ library that provides generic algorithms and data structures (via C++ templates) for many computational geometry problems. The goal of the CGAL project is to "make the large body of geometric algorithms developed in the field of computational geometry available for industrial application" (cf. [21]). CGAL has the sub-goals of providing correct, flexible, easy-to-use, efficient, and robust packages, which will be discussed further in Chapter 2. This library includes packages for Delaunay and regular triangulations in $\mathbb{R}^2$ and $\mathbb{R}^3$, convex hulls, line segment arrangements, and many more packages that are too numerous to mention here.

1.4 Contributions

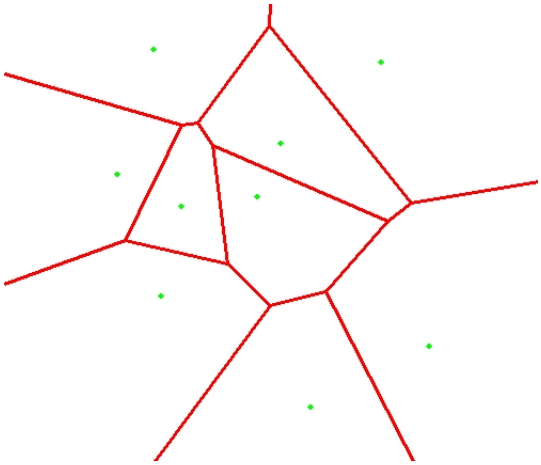
The main contributions of this thesis are as follows:

- Develop a data structure and an algorithm for the construction of basic triangulations of point sites and regular triangulations of weighted point sites in $\mathbb{R}^4$.
- Contribute a correct, flexible, easy-to-use, efficient, and robust implementation to CGAL.

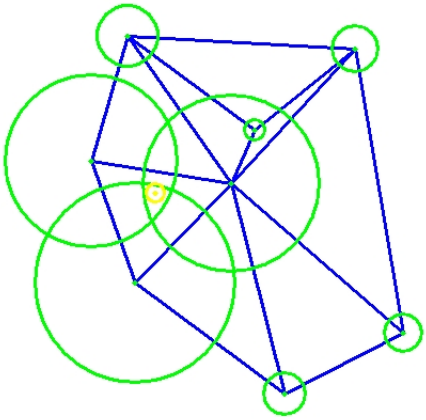
Constructing the CGAL package for regular triangulations in $\mathbb{R}^4$ requires abstracting the properties, data structure, geometric predicates, and algorithm from the lower-dimensional cases that have already been implemented in CGAL. A goal is to provide a similar interface as the triangulation packages in lower dimensions in order to maintain a similar look and feel for users familiar with these other packages. Because the structure of a triangulation in $\mathbb{R}^4$ is naturally more complicated than



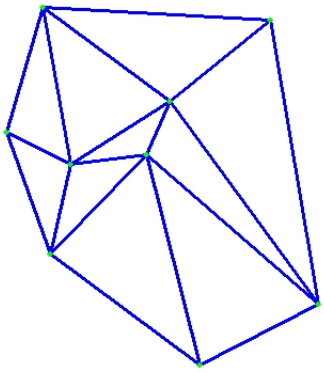
(a) Power diagram of weighted points



(b) Ordinary Voronoi diagram of points



(c) Regular triangulation of weighted points



(d) Delaunay triangulation of points

Figure 1.2. A power diagram, an ordinary Voronoi diagram, a regular triangulation, and a Delaunay triangulation are illustrated. In all four figures, the same point set is used. For the power diagram and regular triangulation, weights were added to each point. Notice that the power diagram has a weighted point that does not have an associated region, and the same weighted point is not included in the regular triangulation. We say that the weighted point is hidden. In contrast, points of ordinary Voronoi diagrams and Delaunay triangulations are never hidden.

that in $\mathbb{R}^3$, new ideas (not used in the existing CGAL triangulation packages) for traversing the resulting triangulations will be introduced, which are necessary for the construction process.

When all points have equal weight, the power diagram is exactly the ordinary point site Voronoi diagram, and therefore the regular triangulation of the weighted points is equal to the Delaunay triangulation of the corresponding non-weighted points. Therefore, the regular triangulation implementation yields a Delaunay triangulation implementation as a special case.

1.5 Organization

Chapter 2 summarizes previous results on the power diagram and regular triangulation problems. Also, more detailed descriptions of CGAL and the existing triangulation packages will be discussed. Chapter 3 discusses the generalizations that are necessary to engineer the data structure, geometric predicates, and algorithm to construct triangulations in $\mathbb{R}^4$. Chapter 4 details CGAL specific implementation issues, and discusses how filtering the geometric predicates is a practical technique to improve efficiency. Finally, Chapter 5 concludes the thesis and suggests possibilities for future work.

CHAPTER 2

PREVIOUS WORK

2.1 Triangulations

Triangulations (especially Delaunay triangulations) have been studied extensively throughout the literature, and many results have been collected in the previously sited computational geometry textbooks. For example, Boissonnat and Yvinec [12] devote several chapters to triangulations (both point set triangulations and polygon triangulations). Properties of (Delaunay) triangulations have been studied by Lawson [32] and Rajan [36], and regular triangulations have been studied directly by Edelsbrunner and Shah [19, 20]. Triangulations (and more generally simplicial complexes) also appear in the field of topology, where they are studied from a topological (instead of computational) point of view. A topology text such as Armstrong [3] is a useful reference for the topological viewpoint of triangulations and simplicial complexes. Providing an extensive survey of computational and topological triangulation literature is beyond the scope of this thesis, but some of the important properties and algorithms will be summarized in the following sections.

2.1.1 Properties

Boissonnat and Yvinec [12] state that, "To triangulate a region is to describe it as the union of a collection of simplices whose interiors are pairwise disjoint."

To be mathematically precise, definitions of simplices, simplicial complexes[1], and triangulations must be given. The following descriptions are consistent with both Boissonnat and Yvinec [12] and Armstrong [3]. A k-simplex in $\mathbb{R}^d$ for $0 \leq k \leq d$ is the convex hull of $k+1$ *affinely independent* points[2]. Intuitively, a k-simplex is the simplest object that spans k dimensions. For example, a 0-simplex is a point, a 1-simplex is a segment, a 2-simplex is a triangle, a 3-simplex is a tetrahedron, and so on. In this thesis, a 4-simplex will be referred to as a pentahedron. An l-face of a k-simplex is itself a simplex: it is the convex hull of $l+1$ points of the k-simplex, where $0 \leq l \leq k$. A (simplicial) complex is defined as a set of simplices such that:

1. Given a k-simplex in the complex, every l-face of the simplex is also in the complex, and
2. Given two k-simplices, either they do not intersect or they intersect at a shared l-face, for some $l < k$.

A d-complex is a complex that contains d-simplices, but no k-simplex for any $k > d$. A triangulation is a complex but not every complex is a triangulation, so we must strengthen these ideas before arriving at the definition of a triangulation.

First, triangulations must be *connected* complexes. The 1-skeleton of a complex is a subset of the complex consisting of only 0- and 1-simplices. A 1-skeleton of a complex is essentially an undirected graph where the vertices are the 0-simplices (points) of the complex, and the edges are the 1-simplices (segments) of the complex. A complex is connected if its 1-skeleton is a connected undirected graph.

Secondly, triangulations must be *pure* complexes. This means that any k-simplex of a d-triangulation must be either a d-simplex itself ($k = d$) or a k-face of a d-simplex of the triangulation ($k < d$).

[1]Referred to simply as *complexes* from now on.

[2]Let $\{p_1, ..., p_k, p_{k+1}\}$ be a set of $k+1$ points in $\mathbb{R}^d$, where $k \leq d$. Take all linear combinations $\sum_{i=1}^{k+1} \lambda_i p_i$, where $\sum_{i=1}^{k+1} \lambda_i = 1$ for each linear combination. The result is a set of points that is a hyperplane of some dimension called the affine hull of the points. The $k+1$ points are affinely independent if their affine hull has dimension k.

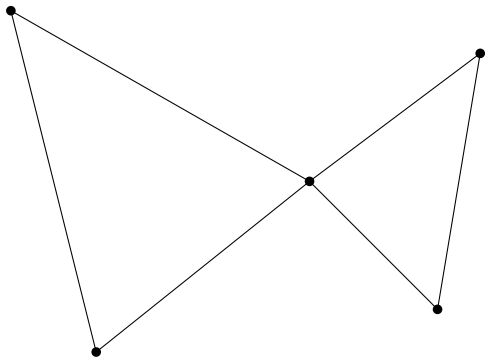


Figure 2.1. An example of a topological singularity in two dimensions. Notice that the two triangles intersect at a vertex but not at an edge. While this is a complex, it is not a triangulation.

Finally, triangulations are complexes that are free of *singularities*. Defining a singularity precisely requires too much topology machinery to be appropriate for this thesis. Intuitively, consider a d-simplex in a d-complex. For the complex to be a triangulation, we expect each $(d-1)$-face of this simplex to be adjacent to one d-simplex when the face is on the convex hull of the 0-faces of the triangulation, or to two d-simplices otherwise. For example, consider a 2-complex consisting of only two triangles that intersect at a point but not at a segment (see Figure 2.1), or a 3-complex with two tetrahedra that share a common point or edge but not a common triangle. Both examples have singularities.

Define a d-triangulation as pure connected d-complex that is free of singularities. Triangulating a set of input points $\{p_1, ..., p_n\}$ in $\mathbb{R}^d$ means constructing a k-triangulation for some $k \leq d$ where each 0-face is one of the input points p_i. Notice that this allows input site p_j to not appear as a 0-face in the simplicial complex. Also, although the input points have d coordinates, all may lie on a common lower-dimensional hyperplane, which results in a k-triangulation for some $k < d$.

Lawson [32] describes point site triangulations in arbitrary dimension and discusses the *sphere test* that is the basic geometric predicate of Delaunay triangulations. The main result proved by Lawson is that given $d+2$ (affinely independent) points in $\mathbb{R}^d$, there exist at most two different ways to triangulate the points (in some cases, there may only be one way). Given $d+2$ points in $\mathbb{R}^d$, the sphere test determines whether the last point lies inside or outside of the $(d-1)$-sphere defined by the first $d+1$ points. If the points are not cospherical, then the sphere test can be used to choose a unique preferred triangulation of the points that will result in the Delaunay triangulation.

Regular triangulations are generalizations of Delaunay triangulations by generalizing the sphere test of points to the power test of weighted points. Because of this, many results about Delaunay triangulations also apply to regular triangulations.

Boissonnat and Yvinec [12] discusses the optimality of the Delaunay triangulation: "Delaunay triangulations maximize two criteria, compactness and equiangularity." The compactness result is due to Rajan [36]. Define the min-containment sphere of a triangle[3] as the smallest sphere containing the triangle. When the center of the circumscribing sphere lies within the triangle, then the min-containment sphere is exactly the circumscribing sphere. However, when the center of the circumscribing sphere lies outside the triangle, then the radius of the min-containment sphere is less than the radius of the circumscribing sphere. One of the main results of Rajan [36] is that "the maximum min-containment radius of the Delaunay triangulation is less than the maximum min-containment radius of any other triangulation of the point set." Therefore, the Delaunay triangulation results in a compact triangulation. The angle-optimal result only applies in $\mathbb{R}^2$, and it is discussed by de Berg et al. [15]. Given a triangulation $\mathcal{T}_\mathcal{P}$ of a set of n points $\mathcal{P}$ containing t triangles,

[3]Rajan [36] uses the term triangle to mean a d-simplex in $\mathbb{R}^d$.

define its angle vector of $3t$ angles to contain the value of each angle of each triangle in the triangulation sorted by increasing value. Among all possible triangulations of the set of points, define the angle-optimal triangulation as the triangulation whose minimum angle in the vector is maximized. $\mathcal{T}_\mathcal{P}$ is an angle-optimal triangulation if and only if $\mathcal{T}_\mathcal{P}$ is a Delaunay triangulation.

2.1.2 Algorithms

Incremental algorithms have been commonly used for constructing triangulations in practice. Incremental algorithms are typically easier to program than sweepline or divide-and-conquer algorithms, and the input data need not be known in advance. The price is that incremental algorithms typically have higher worst case running times. The general idea is to start with some base case triangulation. The input objects are then added one by one, and a *valid* triangulation is maintained after each insertion. Validity in this sense depends on the type of triangulation. For example, a Delaunay triangulation is valid if every simplex satisfies the sphere test, and a regular triangulation is valid if every simplex satisfies the power test. After all input sites have been added, the resulting triangulation is output by the algorithm. In the $\mathbb{R}^2$ case, the base case triangulation may consist of a single fictitious triangle of points not in the input set such that all input points are contained inside this triangle. Such a base case eliminates the issue of inserting a point outside the convex hull of the triangulation. After all of the points are inserted, this fictitious triangle is removed since its points are not input points. In other cases, a fictitious triangle is not used as the base case, and inserting outside the convex hull is allowed.

Boissonnat and Yvinec [12] describes a sweepline algorithm for incrementally constructing triangulations in $\mathbb{R}^2$. The points are sorted in lexicographic order[4].

[4]In $\mathbb{R}^2$, a point (x_1, y_1) is lexicographically smaller than (x_2, y_2) if $x_1 < x_2$, or if $x_1 = x_2$ and $y_1 < y_2$. This generalizes easily to arbitrary dimension.

From left-to-right, each point is inserted (outside the convex hull) and the new triangulation is built from the existing triangulation by adding edges from the point to be inserted to points along the convex hull that are *visible* to the new point. In this case, visibility means that the added edges do not intersect existing edges of the triangulation. This algorithm will produce a *basic* triangulation in that it does not necessarily satisfy any kind of geometric constraint (such as the sphere test or power test). Sorting the n input points requires $O(n \log n)$ time, and building the triangulation takes an additional $O(n)$ time because the combinatorial complexity of a triangulation in $\mathbb{R}^2$ is well known to be $O(n)$.

An important result in Lawson [32] that has become ubiquitous in the Delaunay triangulation literature is that "if a triangulation $\mathcal{T}_\mathcal{P}$ of a point set $\mathcal{P}$ has the property that every pair of simplices sharing a common facet satisfies the local sphere test, then $\mathcal{T}_\mathcal{P}$ satisfies the global sphere test." This theorem is the basis of incremental insertion algorithms for Delaunay and regular triangulations, where sphere and power tests are used to detect conflicting simplices during an insert operation. One method of repairing the triangulation by removing conflicts is flipping, which is based on another result in [32] described above that $d + 2$ affinely independent points have at most two triangulations. A flip essentially changes the structure of a triangulation locally at the $d + 2$ points from one configuration to the other, while leaving the remaining triangulation unchanged.

Guibas and Stolfi [27] provides divide-and-conquer and incremental algorithms for computing the Delaunay triangulation in $\mathbb{R}^2$. We will focus on the incremental algorithm because it is the approach taken in this thesis for solving the generalized problem. When a new point p is inserted in a Delaunay triangulation $\mathcal{T}$, the triangle t in which it is located is found in the point location phase. Then edges are added from p to the points of t. In [27] the authors prove that these new edges are Delaunay:

they will remain as edges in the triangulation after repairing is completed. In the flipping phase, the triangulation is transformed into the Delaunay triangulation by performing the circle test and identifying *suspect* edges. The initial suspect edges are the edges of t. If suspect edges pass the circle test, then they are Delaunay. However, if a suspect edge fails the circle test, then the edge is flipped and two new suspect edges from one of the neighboring triangles are identified. Then it is proved that new edges created by the flip are Delaunay and are adjacent to p. Search proceeds outward from t until no suspect edges remain. At the end of this process, the Delaunay property of the triangulation is restored.

Randomization is sometimes used to shuffle the order of the input objects before insertion in order to improve the expected running time of the algorithm. Using randomized analysis, Guibas et al. [28] prove that "for any collection of n sites (regardless of their distribution), if we randomize over the sequence of their insertions by the incremental algorithm, then the expected total number of structural changes that happen to the diagram is only $O(n)$, and the overall cost of the algorithm is $O(n \log n)$." Another contribution of this paper is the data structure used for point location. Instead of maintaining only the current triangulation, all versions are maintained "on top of one another" with links between versions. More precisely, whenever a triangle is replaced by new triangles, the old triangle is kept in the data structure with pointers from it to the new triangles. Although this seems costly from a space complexity point of view, the authors prove that the expected number of structural changes is $O(n)$, and so the expected space complexity remains $O(n)$. Point location is executed by starting at the original triangulation and tracing down through all the old versions of the triangulation until a triangle of the current Delaunay triangulation that contains the query point is found. The original triangulation is a single fictitious triangle that surrounds the input points as discussed previously.

By using this data structure for point location, the expected cost of locating all sites as they are inserted incrementally is at most $O(n \log n)$ (cf. [28]). Therefore, the expected total running time of the algorithm is $O(n \log n)$.

Rajan [36] provides an incremental insertion algorithm based on flipping to compute the Delaunay triangulation in $\mathbb{R}^d$. The same general idea is used: a list of triangles[5] to be flipped is maintained. A flip operation obtains the next triangle to be flipped out of the list, and then inserts additional suspect triangles into the list as a result. The repairing step is complete when the list is empty. Edelsbrunner and Shah [20] extend this algorithm to the case of regular triangulations in $\mathbb{R}^d$. The data structure used is a *minimalist data structure* that stores an array of vertices, and each simplex as an array of indices into the vertex array and an array of pointers to neighboring simplices. Faces of intermediate dimension are not stored directly. Given the $d+2$ points of two adjacent simplices, the common $(d-1)$-face is flipped to an edge. The algorithm repeats this type of flip by maintaining a list of $(d-1)$-faces called *link facets*. When the list is empty of link facets, then the regular property of the triangulation is restored, and the triangulation is ready for the next point to be inserted.

An important result of that is summarized in Rajan [36] states, "when a point is inserted into a Delaunay triangulation, then every new triangle or facet created during the modification to satisfy the Delaunay criteria, has the new point as one of its vertices." Although this paper uses a flipping based approach, this theorem implies another way to restore the Delaunay property to a triangulation after a new point is inserted. Since each new object in the triangulation will be adjacent to the point to be inserted, it is possible to simply identify all triangles that are *in conflict* with the new point. A triangle is in conflict with a point if its circumscribing circle

[5] Again, Rajan [36] refers to d-simplices in $\mathbb{R}^d$ as triangles.

contains the point in its interior. The set of triangles that are in conflict forms the conflict region, which is a connected set of triangles whose boundary is a polyhedron. The conflict region can be discovered by searching outward (via depth first search for example) from the starting triangle (found during point location), and performing a sphere test on each triangle encountered. The conflict region is then destroyed, and new triangles are formed to fill it with one vertex as the point to be inserted and the other vertices belonging to facets on the boundary of the conflict region.

2.2 Power diagrams

The power diagram was introduced in the previous chapter as the Voronoi diagram of weighted point sites under the power distance function. Recall that the power distance from a point p to a weighted point $P_i = \{p_i, w_i\}$ is defined as $pow(p, P_i) = dist^2(p, p_i) - w_i$. Power diagrams have been studied in many works (e.g., [4, 5, 6, 30]).

2.2.1 Properties

Imai et al. [30] considers the problem of computing the planar Voronoi diagram in the Laguerre geometry, where the distance function from a point to a circle is defined as the length of the tangent line. This is similar, but slightly different from the power distance as it is defined in this thesis. The square of the distance function of [30] is equal to the power distance defined here. Therefore, both distance functions result in the same bisector, and thus the Voronoi diagram in the Laguerre geometry and the power diagram are equivalent. Many of the $\mathbb{R}^2$ results described in [30] easily generalize to $\mathbb{R}^d$.

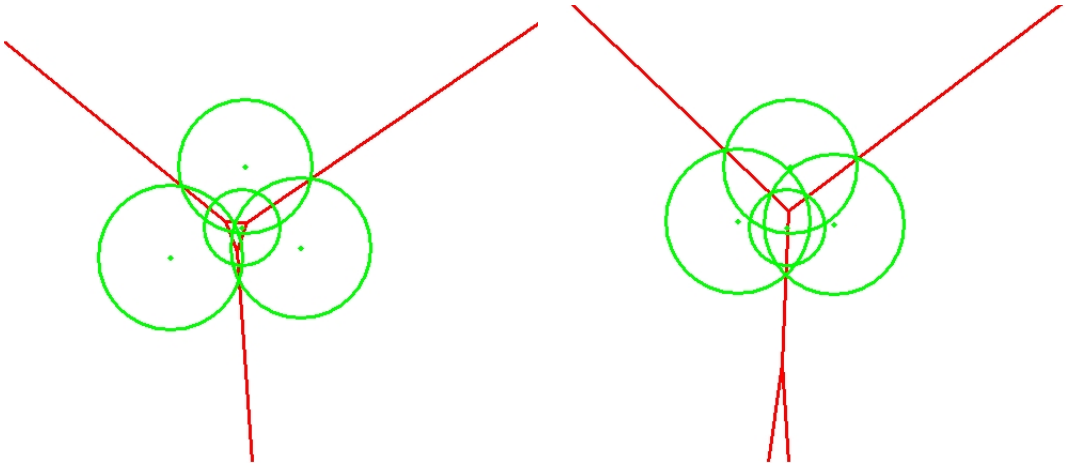
The bisector between two sites is termed the radical axis, and Imai et al. [30] points out that the radical axis is a straight line and is perpendicular to the line joining the centers of the two sites. Since bisectors partition the plane into two half-

planes and Voronoi regions are formed by intersecting these halfplanes, the regions are convex polygons. An interesting case arises when one circle is entirely contained inside another: the bisector if it exists lies outside of the outer circle. A radical center is a single point that is equidistant to three sites.

It is possible for a site to have empty intersection with its Voronoi region. Such a site is termed *improper*. A necessary but not sufficient condition for a site to be improper is that it is contained in the union of other sites. Because a hidden site has an empty Voronoi region, the intersection of that region with the corresponding circle site must be empty. Therefore, a hidden site is necessarily improper. We have seen that if a circle is contained inside a larger circle, then the inner circle does not intersect its halfspace and so it is improper. More interestingly, consider the case where a circle is contained in the union of a collection of other circles (but is not necessarily contained in any single circle in the collection). This does not necessarily imply that the inner circle is improper. However, the converse is true: if a circle is improper then it is contained in the union of proper circles. See Figure 2.2 for an example in $\mathbb{R}^2$. These results generalize easily into higher dimensions.

A Voronoi region may be bounded, unbounded, or empty. If the center of a site is on a corner of the convex hull, then its Voronoi region is non-empty and unbounded. However, if the center a sphere is on the convex hull (but not on a corner), then the corresponding region is either unbounded or empty. Finally, if the center of a sphere is interior to the convex hull of the center points, then its Voronoi region is either bounded or empty. The behavior of points on the convex hull is consistent with the ordinary point site Voronoi diagram except when regions are empty (every site of the ordinary point site diagram will have a non-empty Voronoi region).

Aurenhammer [4] discusses the power diagram problem in $\mathbb{R}^d$, and defines the



(a) Proper (b) Improper

Figure 2.2. Demonstration that a sphere contained in the union of other spheres (but not necessarily contained in any one other sphere) may be proper or improper. In (a), the center sphere is proper because it intersects its power cell. In (b), the three surrounding spheres are the same size as in (a), but the left and right spheres have been repositioned to squeeze the region of the center sphere toward the bottom of the diagram. The result is that the center sphere is improper because it does not intersect its power cell.

power diagram as a cell complex[6] in $\mathbb{R}^d$. An important contribution of this paper is the description of the relationship of power diagrams in $\mathbb{R}^d$ to convex hulls in $\mathbb{R}^{d+1}$, which will be discussed in the following section.

2.2.2 Algorithms

Imai et al. [30] gives an $O(n \log n)$ time divide-and-conquer algorithm for computing the power diagram in $\mathbb{R}^2$. The algorithm is similar in spirit to the algorithm in Shamos and Hoey [37] for computing the point site Voronoi diagram in $\mathbb{R}^2$. The input sites are sorted lexicographically by the x- and y-coordinates of their centers and divided equally into left and right sets. The power diagrams of the left and right sets are computed recursively, and then merged to form the power diagram of the entire set. The merge step can be performed in $O(n)$ time, which results in an $O(n \log n)$ time algorithm. The merge step essentially finds the *dividing polygonal line*, which consists of parts of bisectors between pairs of sites with one site coming from the left set and the other site coming from the right set. Therefore, it consists of a ray extending to infinity above the points, followed by some number of segments, and then a second ray that extends to infinity below the points.

Aurenhammer [4] addresses the power diagram problem in $\mathbb{R}^d$ and exploits a relationship to a convex hull in $\mathbb{R}^{d+1}$ in order to compute the diagram in arbitrary dimension. Let the space $\mathbb{R}^{d+1}$ be spanned by coordinate axes $x_1, ..., x_d, x_{d+1}$, and let the centers of the sites of the power diagram in $\mathbb{R}^d$ be located on the $x_{d+1} = 0$ hyperplane. Consider the paraboloid $\mathcal{Q} : x_{d+1} = x_1^2 + x_2^2 + ... + x_d^2$. Map a weighted point site P_i on $x_{d+1} = 0$ to a hyperplane h in $\mathbb{R}^{d+1}$ so that the projection to

[6] A cell complex is a partition of $\mathbb{R}^d$ into polyhedra. More precisely, define a j-flat as the intersection of $(d-j)$ hyperplanes. For example, in $\mathbb{R}^3$ a 0-flat is a point, a 1-flat is a line, a 2-flat is a plane, and a 3-flat is $\mathbb{R}^3$. A polyhedron is the intersection of a finite number of halfspaces, or the convex hull of its vertices. A j-polyhedron is contained in a j-flat, but not contained in a $(j-1)$-flat. Intuitively, a j-polyhedron uses exactly j dimensions. A cell complex is a collection of j-polyhedra for $(0 \leq j \leq d)$, so it is a partition of $\mathbb{R}^d$ into polyhedra.

$x_{d+1} = 0$ of the intersection of h and $\mathcal{Q}$ results in P_i. Aurenhammer [4] proves that bisectors in the power diagram in $\mathbb{R}^d$ are projections of intersections of hyperplanes in $\mathbb{R}^{d+1}$ to the $x_{d+1} = 0$ hyperplane under this mapping. Therefore, a power diagram in $\mathbb{R}^d$ corresponds to a polyhedron in $\mathbb{R}^{d+1}$. Working in the dual space, a regular triangulation in $\mathbb{R}^d$ corresponds to a convex hull of the lifted sites in $\mathbb{R}^{d+1}$. This algorithm relies on the fact that well known algorithms exist for constructing convex hulls in arbitrary dimension. Chazelle [14] presents an optimal $O(n \log n + n^{\lfloor (d+1)/2 \rfloor})$ deterministic algorithm for computing the convex hull of n points in $\mathbb{R}^{d+1}$. Therefore, the $\mathbb{R}^d$ power diagram algorithm in [4] has time complexity $O(n \log n + n^{\lfloor (d+1)/2 \rfloor})$ as well. To summarize, the algorithm lifts the points into $\mathbb{R}^{d+1}$, computes the convex hull of the lifted points, and projects the result to $x_{d+1} = 0$, which is the power diagram.

2.3 CGAL

There may exist issues that prevent one from taking an algorithm described in the computational geometry literature and translating it directly into an implementation that is free of errors. Two such issues that frequently arise are precision and degenerate input. First, algorithms in the literature sometimes assume infinite precision when dealing with real numbers, while implementations typically use imprecise floating-point types built into programming languages (such as `double` in C++). Secondly, problem specific assumptions are sometimes made about the set of geometric objects input to the algorithm, and these are referred to as general position assumptions. For example, when computing the Delaunay triangulation of point sites in $\mathbb{R}^d$, sometimes the assumption is made that no $d + 2$ input points may be cospherical. Given $d + 2$ points, the sphere test normally distinguishes the preferred of the (at most) two possible triangulations of the $d + 2$ points. However,

when $d+2$ points are cospherical, it is not clear which of the two possible triangulations to choose. One way to avoid the issue is to assume that the input points will be in general position. This kind of assumption can be generalized to the case of constructing the regular triangulation of weighted points in $\mathbb{R}^d$ by not allowing $d+2$ weighted points to be cospherical under the power distance.

The Computational Geometry Algorithms Library (CGAL) is a C++ software library that makes common useful computational geometry algorithms available for application (cf. [21]). All CGAL packages share the common goals of correctness, flexibility, ease-of-use, efficiency, and robustness. Included within these goals are properly dealing with degenerate inputs and handling potential precision problems. One design strategy employed by CGAL packages is to use geometric traits classes to separate numerics from combinatorics. This way, exact numeric computation can be used on the geometry of the objects while the data structure simply represents the combinatoric information. This also allows different implementations of the traits classes to be plugged in without modifying the algorithm or data structure.

Correctness in software design involves code operation coinciding with code documentation. Therefore, verifying correctness involves ensuring that the code behaves as documented. Under this definition, code that does not handle degenerate situations is still considered correct as long as the documentation of the code states such issues. However, CGAL packages do appropriately handle degeneracies. Robustness in CGAL means that a software package does not suffer from precision problems, and CGAL solves this by using number types that support exact computation. However, the trade-off is that working with exact computation slows running time in practice. Flexibility starts by dividing CGAL into smaller packages so that components can be replaced or extended easily. For example, it may be convenient for the user to define his or her own Vertex or Cell type that stores information additional

information such as a color. As long as the new class provides the required public interface, it can be plugged in and used instead of the default Vertex and Cell types. The ease-of-use goal involves how quickly a new user can learn to use the library. Finally, CGAL has the goal of efficiently implementing geometric algorithms. While time and space complexity results are known in the literature, implementing them so that the algorithms run efficiently in practice on large input sets is a challenging goal.

2.3.1 Triangulation packages

There are many packages available in CGAL, but we will only be concerned with further describing the triangulation packages in this thesis. The implementations of the $\mathbb{R}^2$ and $\mathbb{R}^3$ packages use three levels of abstraction. At the bottom, simplices and vertices are represented, and neighbor relationships are defined between simplices by maintaining handles. Simplices are called faces in $\mathbb{R}^2$, and cells in $\mathbb{R}^3$. At the next level, the triangulation data structure maintains the collection of simplices and vertices and allows topological operations to be executed on the entire triangulation. For example, new vertices and simplices can be created and linked together at the data structure level. Finally, the geometric information appears at the triangulation level. The triangulation uses geometric predicates and a data structure to maintain the triangulation. The important functionalities of the existing triangulation packages are location of a query point within the triangulation, insertion/removal of existing vertices, and the ability to traverse the structure.

The triangulation algorithm of each package maintains an *infinite vertex* to allow for convenience when dealing with edges and faces on the convex hull of the input sites. The triangulation algorithm creates an infinite vertex when a new triangulation is created and inserts it into the data structure, which treats it like any

other vertex because the data structure ignores the geometric aspect of a vertex. Simplices are formed from the infinite vertex to the faces on the convex hull. This means triangulations in Euclidean space are mapped to triangulations on a sphere, which has several benefits. It provides uniformity of representation: all edges are adjacent to two faces in $\mathbb{R}^2$, and all faces are adjacent to two cells in $\mathbb{R}^3$, even when these edges and faces are on the convex hull of the sites. The convex hull of the sites can be extracted simply by visiting every infinite cell. Iterators over lower-dimensional features such as edges and faces do not need to handle special cases for edges and faces on the convex hull.

C++ templates are used to parametrize the triangulation class with a geometric traits class and the data structure class. This type of organization provides a clear distinction between the geometry of the triangulation and its structural representation. The traits class provides operations on geometric objects. For example, for the basic triangulation in $\mathbb{R}^2$, the traits class provides an orientation test that takes three points and determines if they have a clockwise or counterclockwise orientation[7]. The geometric traits of the Delaunay triangulation in $\mathbb{R}^2$ provides in addition to the predicates required for constructing a triangulation the circle test that determines whether or not a fourth point lies inside the circle defined by the first three points. The corresponding sphere test is also implemented for Delaunay triangulations in $\mathbb{R}^3$. Finally, the geometric traits for regular triangulations contain the power test in addition to the predicates required for constructing a triangulation, which is essentially a sphere test under the power distance instead of the normal Euclidean distance[8]. For more information on CGAL triangulations, see [8, 10, 13].

[7] In degenerate situations, the three points may be collinear, which results in a return of zero (`COLLINEAR`) from the orientation test.

[8] In degenerate situations, points may be cocircular, cospherical, or equidistant under the power distance, which results in a return of zero (`ON_ORIENTED_BOUNDARY`) from the circle, sphere, and power tests.

CHAPTER 3

BUILDING TRIANGULATIONS INCREMENTALLY

3.1 Introduction

The triangulation algorithms of this thesis are implemented in $\mathbb{R}^4$, which means that points consist of four real coordinate values. A weighted point in $\mathbb{R}^4$ is a point plus an additional real value called the weight. When the weight is non-negative, weighted points can be viewed geometrically as spheres with radius equal to the square root of the weight.

Given six points in $\mathbb{R}^4$, there are at most two different ways to triangulate these points (cf. Lawson [32]). A geometric test (such as the sphere or power test) distinguishes a preferred triangulation between the two possibilities. When triangulating input points that are in general position (non-degenerate), satisfying a geometric constraint locally throughout the triangulation leads to a unique global triangulation. With this in mind, the definition of a basic triangulation is a triangulation of points that does not necessarily satisfy any geometric constraint. A basic triangulation algorithm constructs an arbitrary triangulation (among all possible triangulations). Because of the use of incremental construction in the algorithms of this thesis, the resulting basic triangulation depends on the order of insertion of the points.

We extend basic triangulations to incorporate the power test in order to construct regular triangulations of weighted points. Given a pentahedron, define the

power sphere as the weighted point orthogonal to each of the weighted points of the pentahedron. The regular property establishes that a pentahedron is a cell of a regular triangulation if the power product of any other weighted point with the cell's power sphere is positive. In non-degenerate situations, the power test provides a way to distinguish a preferred triangulation of six weighted points.

In this thesis, we implement data structures, geometric predicates, and algorithms specifically in $\mathbb{R}^4$ to be able to construct basic triangulations of points and regular triangulations of weighted points.

As a prerequisite, we will make the distinction between vertices, points, and weighted points. Points and weighted points are geometric objects that store geometric data only, and serve as the sites of the triangulations that we discuss. Vertices are objects that store sites as one of their components, and are involved in the topological structure of the triangulation. For example, an additional component of a vertex is a reference to one of the cells adjacent to it. This allows the data structure and other classes to use vertices without worrying about the geometric information stored within them. This is important for situations in which only topological connectivity information is important, but actual values of geometric coordinates is not. The same distinction between geometry and combinatorics is made between cells and pentahedra. A cell is a 4-simplex, and a pentahedron is the geometric embedding of a cell.

The chapter is organized as follows. First, we present the data structure including the traversal operations of circulation, enumeration, and iteration. Secondly, we discuss how the geometric predicates are generalized from lower dimensions to $\mathbb{R}^4$. Finally we discuss the incremental insertion algorithm including point location, conflict region determination, conflict region starring, and inserting outside the affine hull. We save the CGAL implementation details for Chapter 4.

3.2 Data structure

The data structure used to represent triangulations in $\mathbb{R}^4$ is similar to the data structure of Edelsbrunner and Shah [20] for representing triangulations in $\mathbb{R}^d$, as described in Section 2.1.2. Namely, the vertices of the triangulation are stored along with the simplices (or cells). The j-faces for $1 \leq j \leq 3$ are represented implicitly in this data structure. Since a 4-simplex consists of five vertices, each cell maintains five handles to vertices, and five handles to neighboring cells. Although it is possible to traverse the list of cells and vertices of the entire triangulation at a global level, the local neighbor relationships at each cell determine the adjacency structure of the triangulation. Therefore, discussion of the entire data structure amounts to discussing individual cells and their adjacency relationships.

Consider a single cell or 4-simplex as described above. Opposite each vertex is a 3-face (facet or tetrahedron), so there are $\binom{5}{1} = 5$ 3-faces adjacent to each cell. Each 3-face is shared by exactly two neighboring cells. Similarly, opposite each pair of vertices is a 2-face (face or triangle), so there are $\binom{5}{2} = 10$ 2-faces adjacent to each cell. In this case, 2-faces may be shared by potentially many cells. Consider the analogous case in $\mathbb{R}^3$ where an edge is shared by potentially many tetrahedra. Continuing, opposite each triple of vertices is a 1-face (edge or segment), so there are $\binom{5}{3} = 10$ 1-faces adjacent to each cell. Again, 1-faces may be shared by potentially many cells, just as in the analogous case in $\mathbb{R}^3$ where a vertex is shared by many tetrahedra.

The input points may all lie on a common lower-dimensional hyperplane (not necessarily axis aligned). This degenerate situation results in a lower-dimensional triangulation. Since the data structure is maintained at the cell level by maintaining five handles to vertices and five handles to neighboring cells, it is capable of representing lower-dimensional triangulations by using a proper subset of these vertex

and cell handles. For example, if the input points lie on a two-dimensional plane, then the highest-dimensional simplex in the triangulation is a 2-simplex, or a triangle. In this case, each cell maintains three vertex handles and three neighboring cell handles, and the remaining two vertex handles and two cell handles are not used. This idea can be applied to other degenerate situations as well in which all input points lie on a common line or a three-dimensional hyperplane.

3.2.1 Traversing

A facet is adjacent to exactly two neighboring cells, but faces, edges, and vertices may be adjacent to possibly many cells. Using our data structure, it will be possible to visit all cells adjacent to a face, edge, or vertex using circulators and enumerators.

Part of the conflict region detection algorithm described above relies on finding all cells adjacent to a lower-dimensional face. Circulating will be used when there is a geometric order among the cells that are adjacent to a lower-dimensional face. A geometric order exists around a $(d-2)$-face of a d-cell. For example, given a triangulation in $\mathbb{R}^2$, there is a circular order of triangles around a point. In $\mathbb{R}^3$, there is a circular order of the tetrahedra around an edge, but there is no circular order of cells adjacent to a vertex. In $\mathbb{R}^4$, there is a circular order of the pentahedra adjacent to a triangle, but there is no circular geometric order of cells adjacent to an edge or a vertex. Since an inserted point may lie on an edge in a four-dimensional triangulation, there must be a way to enumerate all the cells that are adjacent to this edge. Since there is no circular order, the term *enumerator* is used instead of *circulator*. Enumerators are used for retrieving all d-simplices adjacent to a lower-dimensional k-simplex when $k < d-2$. The actual implementation details of circulators and enumerators will be discussed in Section 4.2.1.

3.3 Geometric predicates

After the data structure has been implemented to represent triangulations in $\mathbb{R}^4$, the next step is to implement the geometric predicates. As the most important predicates, we generalize the orientation test to operate on five points, and the power test to operate on five weighted points.

Given four points in $\mathbb{R}^4$ that determine an oriented hyperplane and a fifth query point, the orientation test determines on which side of the oriented hyperplane the fifth point lies. The evaluation of the orientation test in $\mathbb{R}^4$ amounts to determining the sign of the determinant of a certain five-by-five matrix. One twenty-fourth of the determinant of this matrix is equal to the signed volume of the pentahedron formed by the five points. The sign of the volume is positive (negative) if the fifth point lies on the positive (negative) side of the oriented hyperplane determined by the first four points. This is analogous to the $\mathbb{R}^2$ case in which three (ordered) points make either a left turn or a right turn. O'Rourke [35] develops matrices for determining the signed areas of triangles and volumes of tetrahedra, which generalize easily to $\mathbb{R}^4$. We will use a similar approach here.

Let $h = (p, q, r, s)$ be an oriented hyperplane determined by four points in $\mathbb{R}^4$, and let t be the fifth point. Each point has four coordinates, so $p = (p_1, p_2, p_3, p_4)$ and similarly for q, r, s, and t. The predicate returns positive (negative) if t lies in the positive (negative) halfspace of h. In degenerate situations, t may lie on h and so the orientation test returns zero. The orientation test in $\mathbb{R}^4$ is as follows.

$$orientation\,(p,q,r,s,t) = sign\left(\begin{vmatrix} 1 & p_1 & p_2 & p_3 & p_4 \\ 1 & q_1 & q_2 & q_3 & q_4 \\ 1 & r_1 & r_2 & r_3 & r_4 \\ 1 & s_1 & s_2 & s_3 & s_4 \\ 1 & t_1 & t_2 & t_3 & t_4 \end{vmatrix}\right) \tag{3.1}$$

One method of calculating the determinant of matrix A is to evaluate a series of *cofactors*, which are essentially determinants of matrices with one less row and column than A. We can take advantage of the fact that all entries in the first column equal 1 by subtracting the first row from every other row besides the first. The result is the following:

$$orientation\,(p,q,r,s,t) = sign\left(\begin{vmatrix} 1 & p_1 & p_2 & p_3 & p_4 \\ 0 & q_1 - p_1 & q_2 - p_2 & q_3 - p_3 & q_4 - p_4 \\ 0 & r_1 - p_1 & r_2 - p_2 & r_3 - p_3 & r_4 - p_4 \\ 0 & s_1 - p_1 & s_2 - p_2 & s_3 - p_3 & s_4 - p_4 \\ 0 & t_1 - p_1 & t_2 - p_2 & t_3 - p_3 & t_4 - p_4 \end{vmatrix}\right) \tag{3.2}$$

The first column of the resulting matrix contains a 1 followed by all 0 entries. Because of this, all of the cofactors except for one become 0 and can be ignored. Therefore, the orientation test can be evaluated by determining the sign of the determinant of a four-by-four matrix instead of a five-by-five matrix. At this point, we may evaluate this determinant in the fastest way possible, not necessarily by evaluating cofactors. We use the CGAL provided methods for evaluating determinants, which happen to be based on evaluating cofactors.

$$orientation\,(p,q,r,s,t) = sign\left(\begin{vmatrix} q_1-p_1 & q_2-p_2 & q_3-p_3 & q_4-p_4 \\ r_1-p_1 & r_2-p_2 & r_3-p_3 & r_4-p_4 \\ s_1-p_1 & s_2-p_2 & s_3-p_3 & s_4-p_4 \\ t_1-p_1 & t_2-p_2 & t_3-p_3 & t_4-p_4 \end{vmatrix}\right) \tag{3.3}$$

The power test has a similar matrix based implementation. We will essentially map the weighted points in $\mathbb{R}^4$ to points in $\mathbb{R}^5$ and then perform an orientation test. Let $P = \{p, w\}$ be a weighted point in $\mathbb{R}^4$. Map P to $(p, \|p\|^2 - w)$, which is a point in $\mathbb{R}^5$. If this transformation is performed for all the input sites, then we have a collection of points in $\mathbb{R}^5$. By taking the convex hull and projecting those facets that face downward to the $\mathbb{R}^4$ hyperplane, we get the regular triangulation of the original weighted point sites. This transformation helps us in evaluating the predicates because the power test of five weighted points in $\mathbb{R}^4$ essentially turns into an orientation test of five points in $\mathbb{R}^5$, which we already know how to evaluate. For convenience, we introduce the following new variables, which correspond to the last coordinate in our mapping, and then present the six-by-six matrix for evaluating the power test.

$$M_P = p_1^2 + p_2^2 + p_3^2 + p_4^2 - w_P \tag{3.4}$$

$$M_Q = q_1^2 + q_2^2 + q_3^2 + q_4^2 - w_Q \tag{3.5}$$

$$M_R = r_1^2 + r_2^2 + r_3^2 + r_4^2 - w_R \tag{3.6}$$

$$M_S = s_1^2 + s_2^2 + s_3^2 + s_4^2 - w_S \tag{3.7}$$

$$M_T = t_1^2 + t_2^2 + t_3^2 + t_4^2 - w_T \tag{3.8}$$

$$M_U = u_1^2 + u_2^2 + u_3^2 + u_4^2 - w_U \tag{3.9}$$

$$power\_test\,(P,Q,R,S,T,U) = sign\left(\begin{vmatrix} 1 & p_1 & p_2 & p_3 & p_4 & M_P \\ 1 & q_1 & q_2 & q_3 & q_4 & M_Q \\ 1 & r_1 & r_2 & r_3 & r_4 & M_R \\ 1 & s_1 & s_2 & s_3 & s_4 & M_S \\ 1 & t_1 & t_2 & t_3 & t_4 & M_T \\ 1 & u_1 & u_2 & u_3 & u_4 & M_U \end{vmatrix}\right) \tag{3.10}$$

Using a similar manipulation as in the orientation test, we can reduce the power test in $\mathbb{R}^4$ from evaluating a six-by-six determinant to evaluating the following five-by-five determinant.

$$power\_test\,(P,Q,R,S,T,U) =$$
$$sign\left(\begin{vmatrix} q_1-p_1 & q_2-p_2 & q_3-p_3 & q_4-p_4 & M_Q-M_P \\ r_1-p_1 & r_2-p_2 & r_3-p_3 & r_4-p_4 & M_R-M_P \\ s_1-p_1 & s_2-p_2 & s_3-p_3 & s_4-p_4 & M_S-M_P \\ t_1-p_1 & t_2-p_2 & t_3-p_3 & t_4-p_4 & M_T-M_P \\ u_1-p_1 & u_2-p_2 & u_3-p_3 & u_4-p_4 & M_U-M_P \end{vmatrix}\right) \tag{3.11}$$

Notice that when the weights of all points are zero, the points are mapped onto the paraboloid $x_5 = x_1^2 + x_2^2 + x_3^2 + x_4^2$, where each x_i represents a coordinate axis in $\mathbb{R}^5$. For this reason, the power test turns into the circle test when all the weights are zero, and Delaunay triangulations are a special case of regular triangulations. Therefore, the regular triangulation package presented here can be used for computing Delaunay triangulations as well.

In the last paragraph of Section 3.2, we discussed degenerate situations in which all input points lie on a common lower-dimensional hyperplane. We wish to keep the triangulation in its true dimension. From the point of view of the data structure,

dealing with such cases is easy: we just ignore some of the vertex and cell handles in each cell. However, the points still have full dimensionality in these situations, which means that the four coordinate values are all meaningful. For example, this allows for a two-dimensional triangulation of four-dimensional points that all lie on a common two-dimensional plane. But this plane is not necessarily axis aligned. This situation leads to difficulties in the implementation of the predicates because it is not as easy as just ignoring the last coordinate value of each point. This means that it does not blend well with the matrices presented for evaluating the predicates. For example, the orientation test takes five points as input. But if our triangulation lies on a two-dimensional plane, then we wish to test the orientation of only three points and cannot use the orientation test as defined. In addition to the fully-dimensional orientation and power test predicates, we must provide lower-dimensional predicates as well.

The solution is to systematically project the points (or weighted points) to lower-dimensional axis aligned hyperplanes, and then perform orientation tests (or power tests) on these hyperplanes. The four-dimensional Euclidean space $\mathbb{R}^4$ has four axes which we name x_1, x_2, x_3, and x_4. For example, if we wish to determine the planar orientation of three points, we project the points to the x_1x_2-plane and perform the two-dimensional orientation test using a two-by-two matrix similar to the four-by-four matrix given above. If this lower-dimensional orientation test returns positive or negative, then that result is returned. If zero is returned, this means that the projected points are collinear on the x_1x_2 plane. However, this does not necessarily mean that the original points are collinear in $\mathbb{R}^4$. The three points are then projected to the x_1x_3-plane and the process continues. After testing all possible projections (x_1x_4-plane, x_2x_3-plane, x_2x_4-plane, and x_3x_4-plane), some result for the orientation is known. If all projections have collinear orientations on the lower-dimensional

planes, then the three points are collinear in $\mathbb{R}^4$. Otherwise, one of the lower-dimensional orientation tests returns positive or negative, and that is returned as the result of the orientation test of the three fully-dimensional points. A similar process is followed for determining the orientation of four points when all points of the triangulation lie on the same three-dimensional hyperplane in $\mathbb{R}^4$, for determining the orientation of two points when all points of the triangulation lie on the same line, and for performing the lower-dimensional power tests.

3.4 Algorithm

Incremental insertion will be used to construct the triangulations. First, an empty data structure is created that represents a (-1)-dimensional triangulation. As points are inserted, the dimension of the triangulation increases as necessary so that the current dimension equals the dimension of the smallest affine space[1] that contains the point set. This is allowed because the geometric predicates support the lower-dimensional operations, and we can query the data structure for its dimension. For example, when the first point is inserted, the dimension increases from -1 to 0. When the second distinct point is inserted, the dimension increases from 0 to 1. When the next point that is not collinear with the existing points is inserted, the dimension increases to 2. This process continues until all points have been inserted. At any intermediate step in the algorithm, a valid triangulation exists, and so the structure may be traversed and output. Note that this triangulation may not be fully-dimensional after all the points have been inserted.

An additional fictitious vertex called the infinite vertex is added to allow for convenience when dealing with faces on the convex hull of the points, and any cell containing the infinite vertex is called an infinite cell. Every cell is allowed to have

[1] Smallest refers to the dimension of the affine space.

at most one handle to the infinite vertex. Because of the use of the infinite vertex, every cell has the full set of neighboring cells and lower-dimensional faces even if the cell is on the convex hull of the triangulation. Additionally, a d-dimensional triangulation as it is represented here with a vertex at infinity is homeomorphic[2] to a d-sphere in $\mathbb{R}^{d+1}$. For example, by starting with a triangulation of points on the plane and connecting all points on the convex hull to the infinite vertex, the result is a triangulation that is homeomorphic to a 2-sphere. We generalize this idea so that a triangulation in $\mathbb{R}^4$ plus the infinite vertex is homeomorphic to a 4-sphere.

As discussed in Section 2.2.2, a regular triangulation in $\mathbb{R}^d$ corresponds to a polyhedron in $\mathbb{R}^{d+1}$. The upper bound theorem for polyhedra states that any d-polyhedron with n vertices has $O(n^{\lfloor d/2 \rfloor})$ faces of all dimensions. Therefore, the number of cells of a triangulation in $\mathbb{R}^d$ is $O(n^{\lfloor (d+1)/2 \rfloor})$, where n is the number of input sites. Specifically, the number of cells of a four-dimensional triangulation is $\Theta(n^2)$, and any algorithm must take $\Omega(n^2)$ in the worst case. Since we use incremental insertion the worst case running time of the algorithm presented here is $O(n^3)$.

The pseudocode of the incremental insertion algorithm is presented in Algorithm 1. It applies to constructing both basic and regular triangulations. The main difference lies in the determination of the conflict region. We will discuss how to determine the conflict region in both cases in Section 3.4.2 and in implementation detail in Section 4.4.

[2]Formally, a homeomorphism (or topological equivalence) is a function from one topological space to another that is one-to-one, onto, continuous, and has continuous inverse. Intuitively, you may imagine the triangulation (including the vertex at infinity) to be a rubber sheet that can be stretched into a sphere. Although this intuitive approach is useful in many cases, it does not represent all homeomorphisms between topological spaces. For example, a Möbius strip with one twist is indeed homeomorphic to a Möbius strip with three twists, but cannot be stretched to add additional twists. See a topology text such as Armstrong [3] for more information.

Algorithm 1 Constructing a triangulation using incremental insertion.

Input: $\mathcal{P}$, a set of input sites.
Output: $\mathcal{T}$, the computed triangulation of $\mathcal{P}$.
Create the infinite vertex v and initialize a (-1)-dimensional triangulation $\mathcal{T}$ containing v.
for all $p \in \mathcal{P}$ **do**
  Locate cell c within $\mathcal{T}$ that contains p.
  Determine the set of cells $\mathcal{C}$ of $\mathcal{T}$ that are in conflict with p.
  Let $\partial\mathcal{C}$ be the set of facets on the boundary of $\mathcal{C}$.
  for all $f \in \partial\mathcal{C}$ **do**
    Create a new cell composed of f and p, and add it to $\mathcal{T}$.
  end for
  for all $c \in \mathcal{C}$ **do**
    Remove c from $\mathcal{T}$.
  end for
end for

3.4.1 Point location

Point location is the first step of the incremental insertion algorithm when a new point is inserted into an existing triangulation. Given a triangulation and a query point, the point location problem is to find the cell of the triangulation that contains the query point. When the query point lies on the boundary of cells, any cell will suffice as the result of the point location. In the case where the point lies outside the convex hull of the triangulation, any infinite cell whose non-infinite facet is visible to the point suffices. Finally, it may be the case that the point lies outside the affine hull of the current triangulation. This occurs when the current triangulation is not fully-dimensional, and the point to be inserted will cause the dimension of the triangulation to increase by one. Further details regarding how to handle all the insertion cases just mentioned will be discussed in the next sections.

To accomplish point location, orientation tests are used along with the following observation. Let c be a cell consisting of points p, q, r, s, t and let u be a query point. Assume that c has positive orientation: $\mathit{orientation}(p, q, r, s, t) > 0$. If u is

inside c, then all of the following orientation tests will have non-negative values:

- $orientation(u, q, r, s, t)$,
- $orientation(p, u, r, s, t)$,
- $orientation(p, q, u, s, t)$,
- $orientation(p, q, r, u, t)$,
- $orientation(p, q, r, s, u)$.

If all of the orientation tests have positive values, then the query point lies in the interior c. If all of the orientation tests are non-negative and one or more equal zero, then the query point lies on a lower-dimensional face of c.

Point location operates by walking on the current triangulation. A starting cell may or may not be specified (when not specified it is chosen arbitrarily). The above orientation tests are used to determine whether or not the query point is inside (or on the boundary of) the current cell. If one or more of those orientation tests returns a negative value, then the query point must lie outside the current cell. One of the neighboring cells is chosen and the current cell is updated to be this neighboring cell. The neighboring cell chosen always corresponds to an orientation test that returned a negative result. This ensures that the location is moving in the correct direction. For example, if $orientation(u, q, r, s, t)$ is negative, then the neighboring cell that shares q, r, s, t may be be chosen as the next cell. It may be the case that several of the orientation tests return negative values, so any neighboring cell corresponding to a negative orientation test may be chosen. In fact, the next cell is chosen randomly among all the possible neighboring cells corresponding to negative orientation tests. This prevents the algorithm from entering a deterministic infinite loop where the same sequence of cells is visited repeatedly with no forward progress being made.

The same point location algorithm is used by both the basic and regular triangulation packages. For basic triangulations we locate the point to be inserted, and for regular triangulations we locate the center of the weighted point to be inserted.

3.4.2 Conflict region determination

The next step in the algorithm is to identify cells that are in conflict with the point u to be inserted. The cells that are in conflict form the conflict region, which is *star-shaped* with respect to u[3] but not necessarily convex. The conflict region is then filled with one new cell for each facet on the boundary of the conflict region, and each new cell has u as one of its points. Identifying the conflict region is one place where the basic and regular triangulation algorithms differ.

In the case of basic triangulations, if the point to be inserted lies in the interior of some cell c, then only c is considered to be in conflict, and the boundary of the conflict region consists of only five facets. If u lies on a lower-dimensional face, then all cells adjacent to this lower-dimensional face must be discovered and marked as in conflict with u. This can be accomplished by traversing the data structure using the appropriate circulator or enumerator, and marking the cells as in conflict. The data structure provides all necessary circulators and enumerators to accomplish this, which has been described in Section 3.2.1.

Finding the conflict region in a regular triangulation is more interesting. First, the initial cell c that contains the weighted point to be inserted P is found by the point location. There are two cases to consider, either P is hidden (has empty Voronoi region in the power diagram) or c is the first conflict cell.

1. If P is not hidden, then c must be in conflict because P lies within (or on the boundary of) c. If P were both visible and not in conflict with c, this would contract our definition of a point set triangulation.

[3] "A simple polygon $\mathcal{P}$ is star-shaped if it contains a point q such that for any point p in $\mathcal{P}$ the line segment $\overline{pq}$ is contained in $\mathcal{P}$" (cf. [15]). This definition generalizes to polyhedra in higher dimensions.

2. If c is not in conflict with P, then P is hidden. It is impossible for cells outside of c to be in conflict with P because this would result in the same contradiction stated above.

If P is hidden, then it is stored in a list maintained by c because the sites of c hide P. As an alternative, P could have been stored in a global list of all hidden weighted points maintained by the data structure. The algorithm currently does not support deletion of points, so this hidden point will never become visible. However, when the package is modified to allow for deletions, hidden points stored in cells may become visible when vertices of the cell in which they are hidden are deleted.

Because of the power test, a cell may be in conflict with P even when it does not contain the weighted point inside or on the boundary of the cell. If a cell is found to be in conflict with P, the cell is marked and search proceeds to each of its neighbors via a recursive call. This results in depth first search of cells in order to discover the conflict region. The conflict region is star-shaped with respect to the inserted point, so the hole is filled by starring the region (as was done for basic triangulations).

Both basic and regular triangulations contain the convex hull of their points. Because of this, when point to be inserted lies outside the convex hull of the current triangulation, new cells outside the convex hull must be created. A collection of infinite cells will be identified as being in conflict with the point to be inserted. An infinite cell is in conflict if its non-infinite facet is visible to the point to be inserted. This visibility test can be carried by using an orientation test. If the point is on the positive side of the finite facet, then the infinite cell containing that finite facet is in conflict. Again, depth first search is used to discover the conflict region by recursing on neighboring infinite cells.

3.4.3 Conflict region starring

A recursive algorithm is used to fill, or star, the conflict region. Given a star-shaped hole of conflicting cells, a new cell must be created for each facet on the boundary of the hole. Importantly, neighboring relationships between the newly created cells must be built so that traversal of the triangulation after the star operation works properly. In other words, part of reforming a valid triangulation is to correctly set the neighbors of each new cell created. The fact that the conflict region is star-shaped means that the boundary of the conflict region is homeomorphic to a sphere. Therefore, it is not necessary to examine geometry when starring the conflict region. The starring algorithm is implemented at the data structure level.

A conflict cell on the boundary of the conflict region starts the process. A new cell c is created with the same vertices as the facet that is on the boundary of the conflict region but with one vertex corresponding to the vertex to be inserted. Next, search proceeds to the neighboring facets that are also on the boundary of the conflict region. Since the conflict region is simply a collection of cells, the neighboring facets on the boundary are not known immediately. A circulator is used to visit cells until the next boundary facet is discovered. For example, in two dimensions a segment or edge lies on the boundary of the conflict region. Triangles adjacent to one of the vertices of this edge are circulated around until a triangle is found that is on the boundary of the conflict region. In three dimensions a triangle lies on the boundary of the conflict region, so tetrahedra adjacent to one of the edges of this triangle are circulated around until a tetrahedron is found that is on the boundary of the region. Finally, in four dimensions a tetrahedron lies on the boundary of the conflict region, so pentahedra adjacent to one of the triangles of this tetrahedron are circulated around until a pentahedron is found that is on the boundary of the conflict region. At this point, the next boundary facet is known,

and a recursive call is made to create a neighboring cell c_n. The recursive call returns c_n, and the adjacency relationship between c and c_n is set. When a cell is visited that has already been created by this process, this branch of recursion ends.

At the end of the starring process, all the new cells have been created and neighboring relationships have been set. The old cells (that are still marked as in conflict) are simply destroyed.

Conflict region determination and starring occurs when the point to be inserted does not increase the dimension of the triangulation.

3.4.4 Weighted points that become hidden

We discussed in Section 3.4.2 what happens when a weighted point to be inserted is hidden. The weighted point is simply stored in a list maintained by the cell in which its center is located, and there is no conflict region to determine and star. However, it is possible for one or more sites to become hidden after the insertion of a weighted point site. As an output of the conflict region determination and starring, a vector of vertices that were involved in the conflict region is produced. At the beginning of the starring procedure, each vertex is marked. As the conflict region is starred, vertices on the boundary of the region are unmarked. Such vertices obviously do not become hidden. The marked vertices that remain at the end of the starring procedure are interior to the conflict region, and will be stored in the cell in which they located.

The cell in which these hidden vertices are stored is determined by locating the center of the weighted point in the updated triangulation. The location returns a cell, and the weighted point is then added to the cell's list of hidden weighted points. These locate operations will be fast because it is possible to specify that the starting cell is one of the cells adjacent to the newly inserted weighted point.

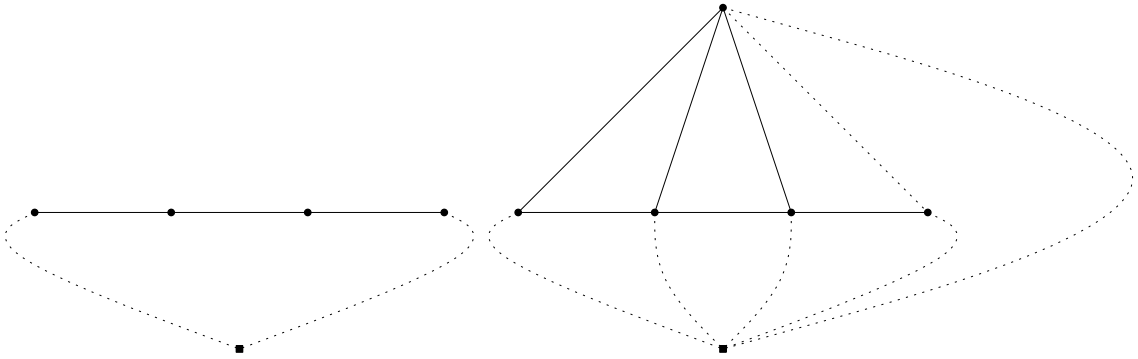
Therefore, the entire triangulation does not have to be searched repeatedly to find the cells in which the hidden weighted points are located. Instead, only the area local to the conflict region (the cells adjacent to the newly inserted weighted point) will be searched.

Since deletions are not yet supported, these hidden weighted points will not become visible. But storing them within the cells that hide them is the first step toward implementing a dynamic algorithm.

3.4.5 Inserting outside the affine hull

When the point to be inserted causes the dimension of the triangulation to increase by one, the conflict region approach cannot be used. As discussed in Section 3.2, the data structure represents lower-dimensional triangulations by using a subset of the vertices to represent lower-dimensional cells. When a new point is inserted that causes the dimension of the triangulation to increase, each existing cell uses the next available vertex to store the vertex of the point to be inserted. The first step when inserting the new point is to loop though all the existing cells and add the new vertex to each cell. This increases the dimension of each existing cell by one.

Imagine the simplest case of a one-dimensional triangulation. Think of this as a circular sequence of alternating edges and vertices, with one of the vertices being the infinite vertex (therefore, there are two infinite edges). When a new vertex is added outside the affine hull, each edge (including both infinite edges) is augmented with the new vertex. Therefore, each finite edge becomes a finite face, and the two infinite edges become infinite faces. However, each of these faces now has a adjacent face across from the new vertex that is not yet defined. Each of the finite vertices now lies on the convex hull, and so infinite faces need to be created for each of the



(a) A one-dimensional triangulation

(b) The triangulation after its dimension is increased

Figure 3.1. Demonstration of inserting a point outside the affine hull of a one-dimensional triangulation.

original finite edges to be the corresponding neighbors of the finite faces across from the new vertex. See Figure 3.1 for an illustration of increasing a one-dimensional triangulation into two dimensions.

Now imagine the lower-dimensional case of a two-dimensional triangulation on the plane. Each face uses three of five of its vertex handles, and each edge on the convex hull is an edge of an infinite face. Then a new point is inserted that causes the dimension to increase to three. Then each face is visited and its fourth vertex is set to the vertex to be inserted (even infinite faces, which essentially connects the vertex to be inserted to the infinite vertex). Each such cell must have a neighboring cell across from the vertex to be inserted, and this will be discussed shortly. All points are now on the convex hull of the three-dimensional triangulation, which means that all of the the original faces now lie on the convex hull. The neighbor of each cell opposite the new vertex is undefined, and it should be an infinite cell. Therefore, new infinite cells are created for each of the original finite faces, and each

face in the original triangulation is now shared between a new infinite cell and a finite cell that existed at the beginning of the insert.

This process is abstracted to allow for inserting a point outside the affine hull of a three-dimensional triangulation. Originally, each cell uses four out of five of its vertex handles, and each triangle on the convex hull of the triangulation is a triangle of an infinite cell. Each existing cell is visited and has its fifth vertex set to the vertex to be inserted. Then, new infinite cells are created to be neighbors of the original finite cells because all of the finite points are now part of the convex hull in four dimensions. Although it is much more difficult to imagine inserting a point outside the affine hull of a three-dimensional triangulation, abstracting the method described in the previous paragraphs to operate in four dimensions becomes straightforward.

CHAPTER 4

CGAL TRIANGULATION IMPLEMENTATION

4.1 Introduction

CGAL packages typically use multiple levels of abstraction in order to separate geometry from topological structure, and to allow advanced users to easily extend certain components of the library without rewriting unrelated components. The code design of the $\mathbb{R}^4$ triangulation package follows the organization of the $\mathbb{R}^2$ and $\mathbb{R}^3$ packages. Although a similar high-level breakdown is used, generalizations are made to reflect the more complicated structure of $\mathbb{R}^4$ triangulations. Descriptions involving the data structure, geometric predicates, and algorithm were discussed in Chapter 3, and implementation details will be discussed here. The triangulation package employs a three-level design. At the lowest level, classes for individual vertices and cells are written. One level higher, the data structure maintains a collection of vertices and cells to represent the structure of the entire triangulation, but it does not operate on the geometric data of the objects. At the highest level, the triangulation class presents the interface of the package to the user. Advanced users may explore into the lower layers and add code if their application demands additional functionality. A high level view of the code organization can be found in Figure 4.1.

We will first present the code design of the data structure (including circulators/enumerators/iterators), followed by the geometric traits classes, and the trian-

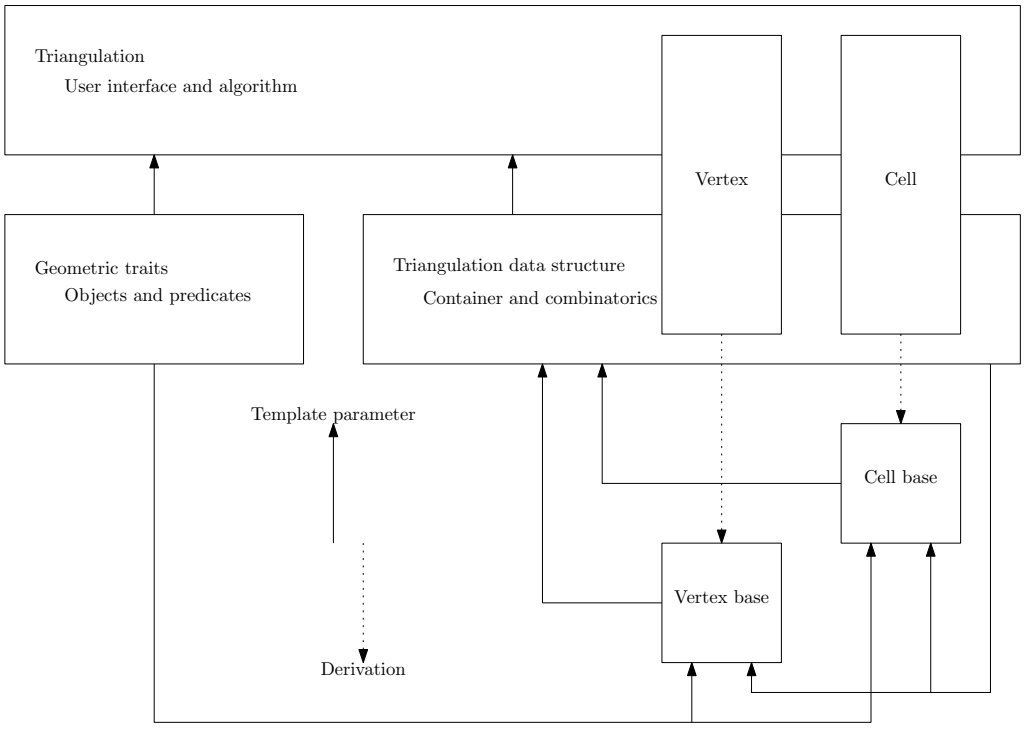


Figure 4.1. An illustration of the triangulation package design. The geometric traits and triangulation data structure are the two template parameters for the interface classes. Additionally, vertex and cell classes are template parameters of the data structure class. Because of this, users may inject their own vertex and/or cell classes into the data structure, or their own geometric predicates or data structure into the algorithm.

gulation and regular triangulation algorithm classes. Although these are discussed in sequence, they are really individual components. At a high level, a geometric traits class and a triangulation data structure class are plugged into the triangulation class as template parameters. In this sense, the traits classes and the data structure class are really implementations of a concept. Other classes may be plugged in as long as they export the public interface required by the triangulation. Finally, we discuss verification methods used to determine whether or not the constructed triangulations are correct.

4.2 The `Triangulation_data_structure_4<Vb,Cb>` class

Perhaps the most important contribution of this thesis is the implementation of a data structure that represents triangulations (both basic and regular) in $\mathbb{R}^4$, including the operations on the data structure (such as starring a conflict region and the implementation of all the circulators and enumerators that allow the structure to be traversed and output). This implementation is provided by `Triangulation_data_structure_4<Vb,Cb>` and its helper classes. The data structure takes two template parameters. The first template parameter is named `Vb`, which stands for vertex base. It is used to define the `Vertex` type, which is used in turn to define the container of vertices stored by the data structure that maintains all the vertices of the triangulation. A `Vertex_handle` is essentially a pointer to a `Vertex` within the container. The second template parameter is named `Cb`, which stands for cell base. The `Cell_handle` type is essentially a pointer to a `Cell` stored within the container of cells maintained by the data structure.

Although this code organization is seemingly complicated, it allows an advanced user to inject his or her own vertex or cell class. For example, the user may wish to augment the default vertex or cell with additional information such as the color of

the vertex or cell. However, beginners need not worry about specifying these classes because they will be chosen automatically at compile time according to default template parameters.

As mentioned in Section 3.2, only cells and vertices are stored directly by the data structure, and faces of intermediate dimension are stored implicitly. A 3-face (which we will call a facet or tetrahedron) is represented with a cell and a single integer index (which indexes a vertex of the cell). This pair represents the facet opposite this vertex within the cell. Notice that there are two possible ways to represent the same facet since exactly two cells share a facet. Similarly, a 2-face (face or triangle) is represented with a cell and two integer indices. This triple represents the face opposite these two vertices within the cell, which is the common face of two facets. Finally, a 1-face (edge or segment) is represented with a cell and three indices, which is the common edge of three facets. Faces and edges may be shared by possibly many cells, so there may be many ways to represent the same face or edge. Indeed, the types `Edge`, `Face`, and `Facet` are public within `Triangulation_data_structure_4<Vb,Cb>`, so the types are exported by the class.

The data structure presented here is similar to the data structure found in the three-dimensional CGAL triangulation package. However, there is a difference in how the intermediate-dimensional simplices are represented. In the four-dimensional data structure, we always use indices into a cell to indicate the simplices that are opposite the vertices indexed. This is not always the case in the three-dimensional triangulation data structure. Specifically, edges in that data structure are represented by cells and two indices that index the actual vertices that comprise the edge. The four-dimensional triangulation data structure implementation provides a uniform way of representing the intermediate-dimensional simplices.

One of the most important functions of the data structure is `insert_in_hole`,

which is used for filling the conflict region that has already been identified. The `insert_in_hole` function must create a new vertex v (which is added to the container of vertices), then fill the hole by creating new cells with one vertex being the vertex to be inserted, and the other vertices belonging to facets on the boundary of the conflict region.

This starring operation is performed by the `create_star_{1,2,3,4}` functions, which are private in the data structure class. The starring function is implemented in the data structure because it does not need the geometric information of the cells. The starring procedure simply assumes a star-shaped conflict region, which may consist of either finite or infinite cells. In the fully-dimensional case, `create_star_4` starts with an existing cell c that is in conflict and on the boundary of the conflict region. The function creates a new cell *cnew* using the vertices of c, and with one of the vertices of *cnew* being replaced by v. The adjacency relationship is set so that *cnew* and the original neighbor of c across from v are now neighbors. But four of the neighbors of *cnew* remain undefined. Using a `Cell_around_face_circulator`, a neighboring facet on the boundary of the conflict region is discovered and a recursive call to `create_star_4` is made, which creates and returns a new cell *nnn* with v as one of the vertices. After the return, the adjacency between *nnn* and *cnew* is set. However, as this recursion develops, it will be the case that `create_star_4` finds a cell that was already created by this process. When this happens, the adjacency relationship is set, but no new recursive calls are made. Therefore, the process stops after all facets on the boundary of the conflict region have had new cells created with v as one of the vertices. At this point, control returns to the `insert_in_hole` function, which deletes all cells that are no longer linked in the current triangulation.

The public interface of the data structure class can be found in Appendix A.5.

4.2.1 Circulators and enumerators

Circulators and enumerators are used for visiting the local neighborhood of a lower-dimensional simplex within the triangulation. For example, given a 2-simplex (face or triangle), visit all cells that contain this face as a lower-dimensional simplex. This is analogous to the situation in $\mathbb{R}^3$ where the user wishes to visit all 3-simplices around an edge. As another example, consider visiting all cells adjacent to a given vertex. In the first case, there is a natural order of cells around a face, and in the second case there is no such natural circular order of cells around a vertex.

A cell and two integer indices are stored to represent a face. Let $f = (c, i_1, i_2)$ be the face where c is a cell of the triangulation and i_1 and i_2 are indices. Let $v_1 = c(i_1)$ and $v_2 = c(i_2)$ be two vertices of the cell. Within c, f is the face opposite v_1 and v_2, and so f contains neither v_1 nor v_2. Now consider two adjacent cells of c. The cell c_1 opposite v_1 shares four vertices with c: the three vertices of f and v_2. Similarly, cell c_2 opposite v_2 shares four vertices with c: the three vertices of f and v_1. Notice that in both cases, the neighboring cell contains f as one of its lower-dimensional faces. The key point is that given a cell c and a face f of that cell, exactly two neighboring cells of c will also contain f as a face. This also implies that there is both a forward and backward way to traverse to the next cell. It is possible to repeatedly visit adjacent cells that all share f in order until the original cell c is reached. In the process, all cells adjacent to f are visited exactly once. This kind of traversal of cells is similar to iterating over a circular doubly linked list of cells in either the forward or backward directions. Hence, the name circulator is used. Users increment and decrement by using the `++` and `--` operators on a circulator object.

Consider the approach of storing the previous cell every time the circulator is incremented. Using the example from above, let the new current cell be c_1 and

let c be stored as the previous node. When the next increment operation occurs, both eligible neighbors of c_1 are compared to c, and the neighbor that is not c is visited next and c_1 becomes the new previous node. This straightforward approach involves storing a temporary variable and executing a comparison for each increment operation, which in inefficient. Instead, an indexing scheme and lookup tables are used so that no temporary variable is stored and no comparisons are made. Memory is used to store the lookup tables. Given integer indices into a cell, a table lookup is performed and an index is returned so that the next (or previous) neighbor can be visited immediately.

Now that the general ideas for circulators are presented, we can discuss the details of the indices and lookup tables. First, the order in which the indices are applied to a cell determine the orientation of the lower-dimensional face it represents. For example, the face $f_1 = (c, i_1, i_2)$ has opposite orientation as $f_2 = (c, i_2, i_1)$. Therefore, a forward circulator around f_1 traverses the same sequence of cells as a reverse circulator around f_2.

Enumerators are like circulators in that they collect simplices around some lower-dimensional simplex. But they are different from circulators in that the simplices in the collection have no circular geometric order. Because of this, enumerators are like iterators in that they have a beginning and an end. In fact, the enumerator classes export iterator types so that users of the these classes can call the `begin` and `end` member functions, and use the `++` and `--` operators to traverse the simplices collected.

Enumeration is an idea introduced in the four-dimensional triangulation package, although it could be added to the three-dimensional triangulation package for enumerating tetrahedra around a vertex. Since enumerating tetrahedra around a vertex is not necessary for the insertion process, the functionality was not imple-

mented in the three-dimensional package. However, enumeration of cells adjacent to an edge is necessary for the insertion of a point that lies on that edge, so there must be some way to mark all the cells adjacent to an edge. Enumerators provide an abstraction to encapsulate this functionality.

Visiting cells around a face

The `Cell_around_face_circulator` class requires a fully-dimensional triangulation. There are several constructors available, all of which take a face. Internally, three `Vertex_handles` $v1$, $v2$, $v3$ that comprise the face are stored, and do not change after the circulator object has been constructed. This is necessary because as the circulator moves from cell to cell, the indices of $v1$, $v2$, and $v3$ in the current cell may change, but the actual vertices remain unchanged. In addition, there is a private `Cell_handle` *pos* that references the current cell location in the circulation process. The cell *pos* always contains $v1$, $v2$, and $v3$.

At construction time, the `map_to_actual_face` table is used to map the two indices that represent the opposite face to three indices that represent the actual face. Given a pair of indices (i, j), a triple of indices (k, m, n) is returned, where i, j, k, m, and n are all different and take values between 0 and 4 inclusive. Notice that shifting the indices of the return triple as in (n, k, m) and (m, n, k) result in additional valid returns. However, looking up (j, i) in the table will return (m, k, n) so that the face returned has the opposite orientation, and the circulator increments in the opposite direction. The contents of the `map_to_actual_face` table can be found in Table 4.1, and the table is implemented in `Triangulation_utils_4` from which both `Triangulation_4<Gt,Tds>` and `Triangulation_data_structure_4<Vb,Cb>` inherit.

The `next_around_face` lookup table is used to map the actual indices of a face f within the current cell to a single index whose neighbor is the next cell when

TABLE 4.1

CONTENTS OF THE `MAP_TO_ACTUAL_FACE` LOOKUP TABLE

Opposite indices	Actual indices	Opposite indices	Actual indices
(0, 1)	(2, 3, 4)	(2, 3)	(0, 1, 4)
(0, 2)	(3, 1, 4)	(2, 4)	(1, 0, 3)
(0, 3)	(1, 2, 4)	(3, 0)	(2, 1, 4)
(0, 4)	(2, 1, 3)	(3, 1)	(0, 2, 4)
(1, 0)	(3, 2, 4)	(3, 2)	(1, 0, 4)
(1, 2)	(0, 3, 4)	(3, 4)	(0, 1, 2)
(1, 3)	(2, 0, 4)	(4, 0)	(1, 2, 3)
(1, 4)	(0, 2, 3)	(4, 1)	(2, 0, 3)
(2, 0)	(1, 3, 4)	(4, 2)	(0, 1, 3)
(2, 1)	(3, 0, 4)	(4, 3)	(1, 0, 2)

circulating in the forward direction. By finding the indices of $v1$, $v2$, and $v3$ in the current cell, performing a lookup in the `next_around_face` table, moving to the adjacent cell, and repeating, all cells adjacent to f are visited exactly once before returning to the starting cell. Notice that if the actual indices of f within the current cell are (i, j, k) then the next index m cannot equal i, j, or k. Since each cell can be indexed 0 to 4 inclusive, there are two choices for m. Since shifting the indices of a face does not change its orientation, (i, j, k), (k, i, j), and (j, k, i) all have the same next index m. However, swapping two indices causes the circulator to move in the opposite direction, so (j, i, k), (k, j, i), and (i, k, j) all have the same next index n, but $m \neq n$. The contents of `next_around_face` can be found in Table 4.2.

Other circulators

Lower-dimensional circulators have existed in the $\mathbb{R}^2$ and $\mathbb{R}^3$ CGAL triangulation packages, so details and the lookup tables will not be discussed here. The data structure exports a type called `Facet_around_edge_circulator` for when the tri-

TABLE 4.2

CONTENTS OF THE `NEXT_AROUND_FACE` LOOKUP TABLE

Actual indices	Index to next	Actual indices	Index to next
(0, 1, 2)	3	(2, 3, 0)	1
(0, 1, 3)	4	(2, 3, 1)	4
(0, 1, 4)	2	(2, 3, 4)	0
(0, 2, 1)	4	(2, 4, 0)	3
(0, 2, 3)	1	(2, 4, 1)	0
(0, 2, 4)	3	(2, 4, 3)	1
(0, 3, 1)	2	(3, 0, 1)	4
(0, 3, 2)	4	(3, 0, 2)	1
(0, 3, 4)	1	(3, 0, 4)	2
(0, 4, 1)	3	(3, 1, 0)	2
(0, 4, 2)	1	(3, 1, 2)	4
(0, 4, 3)	2	(3, 1, 4)	0
(1, 0, 2)	4	(3, 2, 0)	4
(1, 0, 3)	2	(3, 2, 1)	0
(1, 0, 4)	3	(3, 2, 4)	1
(1, 2, 0)	3	(3, 4, 0)	1
(1, 2, 3)	4	(3, 4, 1)	2
(1, 2, 4)	0	(3, 4, 2)	3
(1, 3, 0)	4	(4, 0, 1)	2
(1, 3, 2)	0	(4, 0, 2)	3
(1, 3, 4)	2	(4, 0, 3)	1
(1, 4, 0)	2	(4, 1, 0)	3
(1, 4, 2)	3	(4, 1, 2)	0
(1, 4, 3)	0	(4, 1, 3)	2
(2, 0, 1)	3	(4, 2, 0)	1
(2, 0, 3)	4	(4, 2, 1)	3
(2, 0, 4)	1	(4, 2, 3)	0
(2, 1, 0)	4	(4, 3, 0)	2
(2, 1, 3)	0	(4, 3, 1)	0
(2, 1, 4)	3	(4, 3, 2)	1

angulation is three-dimensional, and `Face_around_vertex_circulator` for the two-dimensional case. Each use a similar scheme as in the fully-dimensional case where the actual vertices of the lower-dimensional face are stored, and then a lookup table is accessed to transition to the correct neighboring cell when an increment or decrement operation is performed on the circulator. The `Vertex_around_vertex_circulator` is valid in two-dimensions only, and is useful for visiting neighboring sites in the dual power diagram.

Visiting cells adjacent to an edge

The `Cell_around_edge_enumerator` class keeps a private member that is a container of `Cell_handles`. Currently, an STL vector (`std::vector<Cell_handle>`) is used as the container, but any container that implements a bidirectional iterator type may be used (such as an STL list, for example). At construction time, an edge is passed to the constructor, and the container of cells is filled immediately via a call to the private `build_cells` member function of the class. First, `build_cells` uses the `map_to_actual_edge` lookup table in order to store `Vertex_handles` v and u of the edge.

The table essentially maps the internal representation of an edge (cell and three indices) into indices that can be used to retrieve the actual vertices that make up the edge. It may be possible to compute this where necessary instead of performing the table lookup, but there are several advantages to using a lookup table. First, it allows encapsulating the operation so that functions do not become cluttered with this translation code. Second, it is implemented with a multi-dimensional array, so lookups are fast. Finally, it allows the encoding of an orientation of the lower-dimensional faces. For example, looking up (0, 1, 2) in the table results in a return of (3, 4), but looking up (1, 0, 2) results in a return of (4, 3). In fact,

map_to_actual_edge and next_around_face are essentially the same table, yet they are used for different purposes. Both take a triple of indices, and the first index returned by map_to_actual_edge is the same as the return value of next_around_face. The contents of map_to_actual_edge are listed in Table 4.3, and the table is implemented in Triangulation_utils_4 from which both Triangulation_4<Gt,Tds> and Triangulation_data_structure_4<Vb,Cb> inherit.

Within build_cells, the recursive incident_cells function is called, which performs a depth first search and marks cells that contain both v and u. Three vertices of the starting cell c are not equal to v or u, so there are three adjacent cells that contain both v and u. Recursion stops when search reaches a cell that has already been marked. At each step along the way, a cell is added to the container. The in_conflict_flag of each cell is used as the marker, and after search completes all in_conflict_flags are reset to 0. The result is that the STL vector is now full of cells adjacent to the edge defined by v and u. The incident_cells function is demonstrated in the following code snippet. Note that in this and all other code snippets, we may eliminate parts of the code and present only those parts that are essential for presenting the underlying ideas of the implementation.

```
void build_cells ()
{
  // The edge is represented by cell c_ and three integer indices
  // i1_, i2_, and i3_. Translate this representation into the
  // actual vertices v and u that comprise the edge. Use the
  // map_to_actual_edge lookup table to accomplish this.
  Vertex_handle v =
             c_->vertex(Tds::map_to_actual_edge(i1_, i2_, i3_, 0));
  Vertex_handle u =
             c_->vertex(Tds::map_to_actual_edge(i1_, i2_, i3_, 1));

  // Fill the container with cells adjacent to the edge. Call
  // recursive function incident_cells (see below).
  Cell_container tmp_cells;
  incident_cells(v, u, c_, std::back_inserter(tmp_cells));
```

```
  // Reset the conflict flags of the cells in question.
  for (Cell_container_iterator cit = tmp_cells.begin();
       cit  != tmp_cells.end(); ++cit) {
    (*cit)->set_in_conflict_flag(0);
    cells.push_back(*cit);
  }

  // Sort the cells by their memory address. This is necessary so
  // that the Edge_iterator (which uses a
  // Cell_around_edge_enumerator) only reports each edge exactly
  // once.
  std::sort(cells.begin(), cells.end());
}

template <typename OutputIterator>
void incident_cells (const Vertex_handle & v,
                     const Vertex_handle & u,
                     const Cell_handle & c,
                     OutputIterator tmp_cells) const
  // v and u are the vertices of the edge.
  // c is the current cell containing both v and u.
  // As cells are discovered, they are added to a container for
  // which tmp_cells is an iterator into.
{
  // Keep track that we have visited this cell so that we do not
  // return.
  c->set_in_conflict_flag(1);

  // Add it to the container of cells.
  *tmp_cells++ = c;

  // Try to explore each adjacent cell.
  for (int i = 0; i < 5; ++i) {

    // Do not explore cells that do not contain both v and u.
    if (c->vertex(i) == v || c->vertex(i) == u)
      continue;

    // Do not explore cells that have already been explored.
    Cell_handle next = c->neighbor(i);
    if (next->get_in_conflict_flag() != 0)
      continue;
```

```
    // Otherwise, the unexplored neighbor contains both v and
    // u. Perform a recursive call to add it to the container.
    incident_cells(v, u, next, tmp_cells);
  }
}
```

After construction, the enumerator operates as an iterator. The `begin` and `end` member functions of the enumerator are called, which return iterators into the STL vector. The increment `++` and decrement `--` operators can be used because the iterator is simply `std::vector<Cell_handle>::iterator`.

Other enumerators

The data structure also exports types for `Cell_around_vertex_enumerator`, `Facet_around_vertex_enumerator`, and `Vertex_around_vertex_enumerator`. A container of cells is built at construction time via a recursive depth first search algorithm. Then the `begin` and `end` member functions of the enumerator are called that return iterators into the container of cells, which can be incremented and decremented with the `++` and `--` operators.

The `Vertex_around_vertex_enumerator` is valid for both three-dimensional and four-dimensional triangulations, and operates much like the cell and facet enumerators, where the internal container stores `Vertex_handle`s instead of `Cell_handle`s. This enumerator is convenient for accessing the neighboring sites in the dual power diagram.

4.2.2 Iterators

The data structure exports types for iteration through the cells, facets, faces, edges, and vertices stored. Specifically, it exports `Cell_iterator`, `Facet_iterator`, `Face_iterator`, `Edge_iterator`, and `Vertex_iterator`. The implementations of `Cell_iterator` and `Vertex_iterator` are provided by the cell and vertex contain-

TABLE 4.3

CONTENTS OF THE `MAP_TO_ACTUAL_EDGE` LOOKUP TABLE

Opposite indices	Actual indices	Opposite indices	Actual indices
(0, 1, 2)	(3, 4)	(2, 3, 0)	(1, 4)
(0, 1, 3)	(4, 2)	(2, 3, 1)	(4, 0)
(0, 1, 4)	(2, 3)	(2, 3, 4)	(0, 1)
(0, 2, 1)	(4, 3)	(2, 4, 0)	(3, 1)
(0, 2, 3)	(1, 4)	(2, 4, 1)	(0, 3)
(0, 2, 4)	(3, 1)	(2, 4, 3)	(1, 0)
(0, 3, 1)	(2, 4)	(3, 0, 1)	(4, 2)
(0, 3, 2)	(4, 1)	(3, 0, 2)	(1, 4)
(0, 3, 4)	(1, 2)	(3, 0, 4)	(2, 1)
(0, 4, 1)	(3, 2)	(3, 1, 0)	(2, 4)
(0, 4, 2)	(1, 3)	(3, 1, 2)	(4, 0)
(0, 4, 3)	(2, 1)	(3, 1, 4)	(0, 2)
(1, 0, 2)	(4, 3)	(3, 2, 0)	(4, 1)
(1, 0, 3)	(2, 4)	(3, 2, 1)	(0, 4)
(1, 0, 4)	(3, 2)	(3, 2, 4)	(1, 0)
(1, 2, 0)	(3, 4)	(3, 4, 0)	(1, 2)
(1, 2, 3)	(4, 0)	(3, 4, 1)	(2, 0)
(1, 2, 4)	(0, 3)	(3, 4, 2)	(0, 1)
(1, 3, 0)	(4, 2)	(4, 0, 1)	(2, 3)
(1, 3, 2)	(0, 4)	(4, 0, 2)	(3, 1)
(1, 3, 4)	(2, 0)	(4, 0, 3)	(1, 2)
(1, 4, 0)	(2, 3)	(4, 1, 0)	(3, 2)
(1, 4, 2)	(3, 0)	(4, 1, 2)	(0, 3)
(1, 4, 3)	(0, 2)	(4, 1, 3)	(2, 0)
(2, 0, 1)	(3, 4)	(4, 2, 0)	(1, 3)
(2, 0, 3)	(4, 1)	(4, 2, 1)	(3, 0)
(2, 0, 4)	(1, 3)	(4, 2, 3)	(0, 1)
(2, 1, 0)	(4, 3)	(4, 3, 0)	(2, 1)
(2, 1, 3)	(0, 4)	(4, 3, 1)	(0, 2)
(2, 1, 4)	(3, 0)	(4, 3, 2)	(1, 0)

ers of the data structure. However, class implementations for the intermediate-dimensional iterators were written since facets, faces, and edges are stored implicitly in the data structure. CGAL iterators operate similarly as STL iterators. In order to use them, the appropriate `begin` and `end` member functions must be called, which exist in the data structure. Also, the dereference operator `*` must be used to access the elements within the iterators.

In lower-dimensional cases, it is possible to create iterators over higher-dimensional features, but such iterators will be empty. For example, when the triangulation is two-dimensional, the `Facet_iterator` and `Cell_iterator` are empty, but the user can iterate over faces using `Face_iterator` and over edges using `Edge_iterator`. More importantly, it is possible to create meaningful iterators over lower-dimensional features. For example, it is possible to iterate over all edges in a four-dimensional triangulation using `Edge_iterator`. Internally, the data structure has the ability to loop over all cells, even in a lower-dimensional case where one or more vertices are ignored.

The method used by the iterator classes is to loop over the cells and report the lower-dimensional simplices within each cell. However, this poses some difficulty because lower-dimensional simplices are shared by more than one cell. Let c be the current cell in the iteration process, and f the lower-dimensional simplex of c that may or may not be reported. We only wish to report f exactly once, so we choose a convention of examining memory addresses of cells containing f to ensure this. If c has the lowest memory address among all cells that contain f then f is reported, otherwise f is skipped. This kind of scheme ensures that each lower-dimensional simplex is reported exactly once, even though it may be shared by many cells. To implement these operations, circulators or enumerators must be used in order to examine the memory addresses of all cells adjacent to a lower-dimensional simplex.

Each of the following iterator classes stores three private data members. The first is a pointer to the data structure object so that the cells stored in the data structure may be accessed. Secondly, a `Cell_handle` *pos* is stored that keeps track of the current position within the data structure's container of cells. The final member is of type `Facet`, `Face`, or `Edge`, depending on the iterator. These types provide the necessary integer indices that are used within the iterator classes, and also serve as the return values when the corresponding iterator is dereferenced.

Iterating over all facets

The `Facet_iterator` class is used to loop over objects that are geometrically tetrahedra. While this iterator can be used in any dimension, it is non-empty only when the dimension of the triangulation is greater than or equal to three. The `Facet` member of the class is a pair consisting of a `Cell_handle` $first$ and an integer *second*.

In three dimensions the facet is the highest-dimensional simplex possible, and so the facet iterator is simply a wrapper around the iterator of the cell container of the data structure. The `++` and `--` operators simply increment and decrement *pos*, while *second* remains 4 throughout.

In four dimensions facets are lower-dimensional simplices, so memory addresses must be examined. As discussed in Section 3.2, each facet of a four-dimensional triangulation is shared by exactly two cells. When the triangulation is four-dimensional and this iterator is created, *pos* starts at the beginning of the cell container and *second* equals 0. When the `++` operator is called, if *second* equals 4 then *second* is set to 0 and *pos* is incremented to the next cell, otherwise *second* increments. This essentially increments to the next facet, which may be skipped depending on the memory address of the cell of *pos* compared to the memory address of the adjacent

cell that shares the facet. Because of this, several facets may be skipped before the `++` operator returns. The opposite behavior occurs on a call to the `--` operator.

Iterating over all faces

The `Face_iterator` class is used to loop over objects that are geometrically triangles, and is non-empty when the dimension of the triangulation is greater than or equal to two. The `Face` member of the class is a triple consisting of a `Cell_handle` *first* and two integers *second* and *third*.

In two dimensions the face is the highest-dimensional simplex, and so the face iterator is a wrapper around the iterator of the cell container of the data structure. The `++` and `--` operators simply increment and decrement *pos* while (*second*, *third*) remains (3, 4) throughout. When the triangulation is three-dimensional, the face iterator operates much like the facet iterator in four dimensions. In this case, *third* always equals 4, but *second* varies from 0 to 3 inclusive.

When the triangulation is four-dimensional, (*second*, *third*) is incremented as follows: (0, 1), ..., (0, 4), (1, 2), ..., (1, 4), ..., (3, 4), (0, 1). At the last transition from (3, 4) to (0, 1), the *pos* iterator is incremented to move to the next cell and this process repeats. This visits all 10 faces of *pos* before incrementing *pos* to the next cell. Again, each triple consisting of (*pos*, *second*, *third*) represents a face, and faces are reported only when *pos* has the lowest memory address among all cells that share the face. Otherwise, the face is skipped. In order to examine the memory addresses of all adjacent cells, a `Cell_around_face_circulator` is used.

Iterating over all edges

The `Edge_iterator` class is used to loop over objects that are geometrically segments, and is non-empty when the dimension of the triangulation is greater than or equal to one. The `Edge` member of the class is a quadruple consisting of a

`Cell_handle` $first$ and three integers *second*, *third*, and $fourth$.

In one dimension the edge is the highest-dimensional simplex, and so the edge iterator is a wrapper around the iterator of the data structure cell container. The `++` and `--` operators simply increment and decrement *pos* while (*second*, *third*, and $fourth$) remains (2, 3, 4) throughout. When the triangulation is two-dimensional, the edge iterator operates much like the `Face_iterator` in three dimensions or `Facet_iterator` in four dimensions. In this case, $fourth$ always equals 4, *third* always equals 3, and *second* varies from 0 to 2 inclusive. In three dimensions the edge iterator operates much like the face iterator in four dimensions. In this case, $fourth$ always equals 4 while (*second*, *third*) varies from (0, 1) to (2, 3) to visit all 6 edges of a facet before incrementing *pos* to the next cell. A `Facet_around_edge_circulator` is used to examine memory addresses of all facets around the current edge.

In four dimensions, (*second*, *third*, $fourth$) is incremented as follows: (0, 1, 2), ..., (0, 1, 4), (0, 2, 3), ..., (2, 3, 4), (0, 1, 2). The *pos* iterator is incremented when the counter resets to (0, 1, 2). This visits all 10 edges of *pos* before incrementing *pos* to the next cell. In this case, a `Cell_around_edge_enumerator` is used to retrieve all the cells adjacent to the edge represented. Conveniently, the enumerator returns cells sorted by their memory addresses, so only the memory address of the first cell of the list returned by the enumerator needs to be compared against the memory address of *pos*.

4.3 Geometric traits and kernel classes

A geometric kernel in CGAL is a class that contains a set of functors that are either geometric objects or geometric predicates that may be common to many packages. "The term *kernel* refers to a collection of representations for constant-size geometric objects and operations on these representations" (cf. [29]). For example,

the orientation test on points is a typical predicate provided by a CGAL kernel. Since CGAL does not currently provide packages specifically in $\mathbb{R}^4$, a kernel was implemented to meet the needs for basic triangulations. This kernel implementation is provided by the `Simple_cartesian_4<Nt>` class, which has a single template parameter that is the number type (which may or may not be exact) used to store objects and perform computations. This kernel does not use reference counting (hence simple), and represents points by their Cartesian coordinates (hence cartesian). Because it is currently used only by the triangulation package, `Simple_cartesian_4<Nt>` does not contain as many geometric objects and predicates when compared to the two-dimensional and three-dimensional kernels. Specifically, it provides the `Point_4` geometric object and the `Orientation_4` geometric predicate. Additionally, lower-dimensional orientation tests are included: `Cohyperplanar_orientation_4`, `Coplanar_orientation_4`, and `Compare_x1x2x3x4_4`. Each of the predicates is a functor[1]. `Cohyperplanar_orientation_4` takes four points and determines the orientation of the tetrahedron they form. Similarly, `Coplanar_orientation` takes three points and determines the orientation of the triangle they form. The orientation tests are evaluated by projecting the points to a lower-dimensional hyperplane and computing signs of determinants, as discussed in Section 3.3. CGAL provides global functions for evaluating determinants. The orientation tests return values of the enumeration type `Orientation` defined in CGAL, which may take the values `NEGATIVE`, `ZERO`, or `POSITIVE`. `Compare_x1x2x3x4_4` performs lexicographical comparison of two points, and it is essentially a one-dimensional orientation test. This is evaluated by comparing the coordinate values of the two points involved.

A traits class in CGAL is either a kernel, or a kernel augmented with additional problem-specific objects and operations. Therefore, a traits class also provides ge-

[1]A functor is a class that defines `operator()`.

ometric objects and predicates to the algorithms in the form of functors. An algorithm class essentially ties together a traits class (geometry) with a data structure class (combinatorial representation) in order to provide the interface and functionality of the package. For the case of basic triangulations, the traits class used is simply the `Simple_cartesian_4<Nt>` kernel. However, a traits class customized for regular triangulations was implemented to include the `Weighted_point_4` geometric object and the `Power_test_4` geometric predicate. The power tests return values of the enumeration type `Oriented_side` defined in CGAL, which may take the values `ON_NEGATIVE_SIDE`, `ON_ORIENTED_BOUNDARY`, or `ON_BOUNDED_SIDE`. The traits class implemented is named `Regular_triangulation_traits_4<Kernel>`, which inherits from whatever kernel is used (e.g., `Simple_cartesian_4<Nt>`). The `Power_test_4` predicate also calls CGAL's global functions for evaluating determinants. Note that although `Simple_cartesian_4<Nt>` and `Regular_triangulation_traits_4<Kernel>` were implemented, users are able to plug in their own class as long as they implement the necessary interface, which is composed of the functors mentioned.

4.3.1 Arithmetic filtering

The use of a traits class allows a clean separation between geometry and combinatorics. Although traits classes export geometric objects and operations on those objects, there is a representational issue that has not yet been addressed. Namely, the geometric objects must use variables of some number type to store the underlying representation. The operations are evaluated by performing arithmetic on the data of the underlying representation. For example, a `Point_4` object needs to store four coordinates, and a `Weighted_point_4` object needs to store a weight in addition to the four coordinates. The operations are implemented by computing determinants using these values.

If `double` is used as the number type, predicates may be evaluated incorrectly during the course of the algorithm. Not only may this cause an incorrect triangulation to be constructed, but it may cause more disastrous results such as an infinite loop. For example, during the locate phase of the insertion process it is possible for the query point to lie on or very close to a lower-dimensional facet f that is the boundary between two cells c_1 and c_2. If the orientation test returns an incorrect result then the algorithm may insert the new point inside one of the cells instead of on f. This kind of error may be deemed acceptable. However, if the orientation test returns that the query point is outside of both c_1 and c_2, then an infinite loop will result. This kind of situation is possible when using an inexact number type.

There are several approaches that have been developed. For example, the *simulation of simplicity* approach in Edelsbrunner and Mücke [18] essentially perturbs the input objects slightly so that degenerate cases (like a point lying on a lower-dimensional face) disappear. Using this approach, algorithms may be written under general position assumptions on the input objects, yet operate correctly in degenerate situations because of the perturbation. See also Sugihara et al. [38] for a topology based approach, Yap [40] for a theoretical framework for exact computation, and Kettner et al. [31] for examples under which algorithms fail due to arithmetic errors.

The approach taken by CGAL is to develop algorithms using an exact number type and exact computation of predicates. This adds additional computational overhead, especially considering that operations on `double` can be performed in hardware, while operations on exact number types must be implemented in software. To address this, CGAL uses *dynamic arithmetic filtering*, where inexact computations are performed, and the reliability of the results is checked. If floating point errors have occurred, then the same computation is performed using an exact number type. The decision as to whether or not floating point errors occurred is made at runtime,

hence the filtering is dynamic.

CGAL makes it easy to incorporate arithmetic filtering into the development of packages by providing a `Filtered_predicate<...>` class. A package developer typically implements a package including a traits class by exclusively using exact computation. Then the `Filtered_predicate<...>` class is used almost mechanically to provide a filtered version of the traits class. Indeed,
`Triangulation_filtered_traits_4<Kernel>` and
`Regular_triangulation_filtered_traits_4<Kernel>` classes were written to provide filtered versions of the traits class. Note that both of these may be plugged instead of the previously mentioned kernel and traits classes because they provide the same interface. See Appendix A.4.

4.4 The `Triangulation_4<Gt,Tds>` class

This class provides the high level interface to the user of the basic triangulation package. It uses a geometric traits class containing the objects and predicates, which is plugged in as the first template parameter. Either `Simple_cartesian_4<Nt>` or `Triangulation_filtered_traits_4<Kernel>` may be used, both of which are provided. It also uses a data structure class, which is plugged in as the second template parameter. The user builds a triangulation by first creating a
`Triangulation_4<Gt,Tds>` object with a constructor, and then using an `insert` function to input the points into the triangulation.

There are a number of `insert` functions that are used internally within the class to handle various cases, but there is one important `insert` function that is part of the public interface. It takes a `Point_4` object as its first parameter, and has an optional second parameter that is a `Cell_handle`. The `Cell_handle` is used as the starting cell when the `locate` function is called, which is the first step within

`insert`. The `locate` function finds where the new point will fit into the current basic triangulation. Depending on the result of the `locate` call and the dimension of the triangulation, the new point will be inserted in an edge (geometrically a segment), in a face (triangle), in a facet (tetrahedron), in a cell (pentahedron), outside the convex hull, or outside the affine hull (the dimension increases).

There are two public `locate` member functions available to the user. The simplest one takes a query `Point_4` object and returns a `Cell_handle` that contains the query point. While this `locate` function may be adequate for some purposes, it does not indicate whether the query point lies interior to the cell or on a lower-dimensional face. To extract this additional information, the second version of the `locate` function must be used. It also returns a `Cell_handle`, but contains four additional reference parameters that are set within the function so that the caller can access the additional information. The first additional parameter is of the enumerated type `Locate_type`, which may hold the value `VERTEX`, `EDGE`, `FACE`, `FACET`, `CELL`, `OUTSIDE_CONVEX_HULL`, or `OUTSIDE_AFFINE_HULL`. The next three reference parameters are integer indices into the cell that is returned. If the query point is located on a lower-dimensional face of the cell, then one to three of these indices are set in order to indicate the lower-dimensional face on which the query point lies. If the query point is located outside the convex hull, then an infinite cell is returned whose finite facet is visible to the query point (there may be many cells that satisfy this criteria, and any one of them may be returned). It is this version of the `locate` method that is used as the first step when `insert` is called.

Table 4.4 summarizes what `locate` may return. In this table, c is used for the `Cell_handle` that is returned, the indices li, lj, and lk may take values between 0 and 4 inclusive, and an X character indicates that a field is not used. When a vertex is returned, the li index is used to indicate the vertex within c on which the query

TABLE 4.4

POSSIBLE RETURN VALUES OF THE `LOCATE` METHOD

D	Locate_type is	Returns	Represented By
4	CELL	finite cell	(c, X, X, X)
4	FACET	finite facet	(c, X, X, lk)
4	FACE	finite face	(c, X, lj, lk)
4	EDGE	finite edge	(c, li, lj, lk)
4	VERTEX	finite vertex	(c, li, X, X)
4	OUTSIDE_CONVEX_HULL	infinite cell	(c, X, X, lk)
3	FACET	finite facet	(c, X, X, 4)
3	FACE	finite face	(c, X, lj, 4)
3	EDGE	finite edge	(c, li, lj, 4)
3	VERTEX	finite vertex	(c, li, X, X)
3	OUTSIDE_CONVEX_HULL	infinite facet	(c, X, lj, 4)
3	OUTSIDE_AFFINE_HULL	nothing else	(X, X, X, X)
2	FACE	finite face	(c, X, 3, 4)
2	EDGE	finite edge	(c, li, 3, 4)
2	VERTEX	finite vertex	(c, li, X, X)
2	OUTSIDE_CONVEX_HULL	infinite face	(c, li, 3, 4)
2	OUTSIDE_AFFINE_HULL	nothing else	(X, X, X, X)
1	EDGE	finite edge	(c, 2, 3, 4)
1	VERTEX	finite vertex	(c, li, X, X)
1	OUTSIDE_CONVEX_HULL	infinite edge	(c, 2, 3, 4)
1	OUTSIDE_AFFINE_HULL	nothing else	(X, X, X, X)
0	VERTEX	finite vertex	(c, 0, X, X)
0	OUTSIDE_AFFINE_HULL	nothing else	(X, X, X, X)
-1	OUTSIDE_AFFINE_HULL	nothing else	(X, X, X, X)

point lies. When an infinite cell, facet, or face is returned, one of the indices is used to indicate which vertex is the infinite vertex. In the case of an infinite edge in one dimension, no such index is provided because a fourth reference parameter would have to be added to the `locate` function specifically for this case. In the case when the `Locate_type` is `OUTSIDE_AFFINE_HULL`, the `Cell_handle` and three indices are not assigned and have no meaning.

We will discuss only the $\mathbb{R}^4$ case of the operation of `locate`, but keep in mind that essentially the same algorithm is implemented in the lower-dimensional cases as well. The `locate` implementation walks from cell to cell until a cell that contains the query point is found. The original starting cell may be provided by the user as the last parameter to the function. If no starting cell is specified, then the neighbor of an infinite cell (in other words, a finite cell) is chosen. At each step, a random integer i between 0 and 4 inclusive is chosen, and five `Point_4` objects $p0$, ..., $p4$ are retrieved by indexing the cell with i, $(i+1)\%5$, ..., and $(i+4)\%5$. Regardless of the value of i, the orientation of the simplex $(p0, p1, p2, p3, p4)$ will be positive. In other words, the orientation test returns `POSITIVE`[2]. Then a sequence of at most five orientation tests are performed as described in Section 3.4.1. If all the orientation tests return non-negative values, then the query point lies inside or on the boundary of the current cell. Otherwise, at least one of the orientation tests returned a negative value, so the new current cell is set to the neighboring cell corresponding to the first negative

[2]This occurs because the dimension is even. Consider an analogous case of determining the orientation of a triangle using the right-hand rule. In this case, the orientation of $(p0, p1, p2)$ is the same as the orientations of both $(p1, p2, p0)$ and $(p2, p0, p1)$. However, in odd dimensions the value of i that is chosen determines the orientation of the resulting simplex. For example, in three dimensions the orientation of $(p0, p1, p2, p3)$ is the same as the orientation of $(p2, p3, p0, p1)$, but not the same as the orientations of $(p1, p2, p3, p0)$ and $(p3, p0, p1, p2)$. This is also related to the size of the matrix whose determinant is evaluated. In even dimensions a square matrix of odd size is evaluated, and in odd dimensions a square matrix of even size is evaluated. The sign of the determinant of an odd sized matrix does not change when the rows are shifted, while the sign of the determinant of an even sized matrix does change when the rows are shifted. The `locate` method correctly deals with both situations.

orientation test encountered. Choosing i randomly is important because it leads to a random choice among all possible neighboring cells that correspond to negative orientation tests. Otherwise, it is possible for the `locate` method to enter an infinite loop by visiting a sequence of cells in a cycle. In this way, it uses randomness to get out of local minima in its search to find a cell in which the query point lies. A code snippet from the `locate` method is provided next.

```
template <class Gt, class Tds>
typename Triangulation_4<Gt,Tds>::Cell_handle
Triangulation_4<Gt,Tds>::locate
(const Point_4 & p,
 Locate_type & lt, int & li, int & lj, int & lk,
 Cell_handle start) const
{
  // Some code omitted.
  // ...

  // Store orientations of the new point with respect to facets.
  // This allows testing if a point is inside or on boundary of a
  // cell.
  Orientation o[5];

  // When the cell is located, then we will break out of this loop
  // and return the proper Cell_handle. Until it is found, walk
  // from cell to cell by using the neighbor(index) function.
  while (1) {

    // If the current cell has the infinite vertex, then p is
    // outside the convex hull of the triangulation.
    if (c->has_vertex(infinite, lk)) {
      lt = OUTSIDE_CONVEX_HULL;
      // lk is set with the has_vertex function.
      return c;
    }

    // There are 5 possible vertices to choose from in the
    // cell. Choose a random index as the starting point.
    i = rand_5();

    // Get the five points of the vertices.
    Point_4 p0 = c->vertex( i     )->point();
```

```
Point_4 p1 = c->vertex((i+1)%5)->point();
Point_4 p2 = c->vertex((i+2)%5)->point();
Point_4 p3 = c->vertex((i+3)%5)->point();
Point_4 p4 = c->vertex((i+4)%5)->point();

// This is what the orientation should NOT be. Replace each of
// p0,...,p4 by p and compute the orientation. If all are not
// NEGATIVE then p is inside or on the boundary of the current
// cell.
Orientation test_or = NEGATIVE;

// Look at the neighboring cell to make sure we do not go
// backward to a cell that we already know does not contain p.
Cell_handle next = c->neighbor(i);
if (previous != next) {

  // Compute the orientation of p with the first facet.
  o[0] = orientation(p, p1, p2, p3, p4);

  // Since the dimension is 4, test_or is always NEGATIVE. If
  // the orientation just computed is negative, we know p lies
  // on the other side of the facet and p cannot be in c. Go to
  // the adjacent cell across from i and continue.
  if (o[0] == test_or) {
    previous = c;
    c = next;

    // Jump immediately to the top of the while(1) loop.
    continue;
  }
}

// Since we just came from the neighbor, we know that this
// orientation must be correct. Because the dimension is 4,
// this orientation is positive.
else
  o[0] = POSITIVE;

// Repeat this four more times for the remaining facets.
// ...

// If we make it here without continuing to the top of the
// loop, then we have found a cell c that contains p. Break the
// loop and determine the exact location.
```

```
    break;

  } // end while (1) loop

  // Now p is in c or on its boundary. We have the o[0...4] matrix
  // filled with values that are either POSITIVE or ZERO. Use this
  // to determine the exact location (p may be interior to c or on
  // a facet, face, edge, or vertex of c.

  // Some code omitted.
  // ...
}
```

After the `locate` method returns, the next step is to identify the conflict region. Since the `Triangulation_4<Gt,Tds>` class constructs basic triangulations, determining the conflict region is straightforward. If the new point lies in the interior of some cell, only that cell comprises the conflict region. Otherwise, the new point lies on a lower-dimensional face. In this case, all cells adjacent to that lower-dimensional face comprise the conflict region. The conflict region is marked (possibly by using circulators or enumerators to visit all cells adjacent to a lower-dimensional face) by setting each cell's `in_conflict_flag` to 1, and then the conflict region is filled using the `insert_in_hole` function of the data structure described in Section 4.2.

The `Triangulation_4<Gt,Tds>` class also exports types for all iterators, circulators, and enumerators from the data structure so that users of the class can traverse and output the resulting structure. Whereas the data structure implements iterators for all (finite and infinite) cells, facets, faces, edges, and vertices, the `Triangulation_4<Gt,Tds>` class uses checks in order to filter out infinite components. Hence, `Triangulation_4<Gt,Tds>` also provides iterators for finite cells, facets, faces, edges, and vertices. An example of using some of these features will be provided in Section 4.6.

The public interface of the basic triangulation package can be found in Appendix

A.1.

4.4.1 The `Regular_triangulation_4<Gt,Tds>` class

The `Regular_triangulation_4<Gt,Tds>` class inherits from the `Triangulation_4<Gt,Tds>` class, so much of the functionality is provided by the base class. For example, the `locate` methods are inherited from the base class. However, some functions need to be overridden, and the most important of these are the `insert` functions. First, `insert` now operates on `Weighted_point_4` objects instead of `Point_4` objects. Second, `insert` must now deal with hidden sites properly, both when a weighted point to be inserted is hidden and also when other weighted points become hidden in the insertion process.

A conflict region is again determined, but the `Power_test_4` is now used to find cells that conflict with the new point. Recall from Section 3.1 that the power test determines if a query weighted point is inside, outside, or on the boundary of a power sphere, which is defined by the weighted points of a cell of a regular triangulation. If a cell is determined to be in conflict, then it is saved in a list, and all of its vertices are marked because some of them may become hidden. After the entire conflict region is determined, the starring function (`insert_in_hole`) of the data structure is called, which unmarks vertices that lie on the boundary of the conflict region. Some vertices may still be marked, and their geometric data will be hidden in the proper cell. An additional `locate` call is performed to determine the cell in which to hide the site, and then it is simply added to a container within that cell. A snippet of the code that discovers the conflict region is presented in the following.

```
// in_conflict_flag value :
// 0 -> unknown
// 1 -> in conflict
// 2 -> not in conflict (== on boundary)
template <int D,
```

```
          class Conflict_test,
          class OutputIteratorBoundaryFacets,
          class OutputIteratorCells,
          class OutputIteratorInternalFacets>
Triple <OutputIteratorBoundaryFacets,
        OutputIteratorCells,
        OutputIteratorInternalFacets>
// The return is a triple of items.
// The first item is an iterator into a container that stores
//   facets lying on the boundary of the conflict region.
// The second item is an iterator into a container that stores
//   cells in the conflict region.
// The third item is an iterator into an container that stores
//   facets inside the conflict region (but not on the boundary).
find_conflicts (const Cell_handle & c,
                const Conflict_test & tester,
                Triple<OutputIteratorBoundaryFacets,
                       OutputIteratorCells,
                       OutputIteratorInternalFacets> it) const
  // c     : The current cell, which must be in conflict.
  // tester: A functor that tests if a cell is in conflict with the
  //         point to be inserted, which is stored inside tester.
{
  // Make sure c is in conflict. Useful in the first call to this
  // recursive function.
  CGAL_triangulation_precondition(tester(c));

  // Set the in_conflict_flag and add c to the conflict region cell
  // container.
  c->set_in_conflict_flag(1);
  *it.second++ = c;

  // Examine every neighbor of c.
  for (int i = 0; i < D+1; ++i) {
    Cell_handle test = c->neighbor(i);

    // If this neighbor already has in_conflict_flag set, then do
    // not explore it. Keep track of internal facets by adding it
    // to the internal facet container. Compare memory addresses of
    // c and test to ensure adding the facet once into this
    // container.
    if (test->get_in_conflict_flag() == 1) {
      if (c < test)
        *it.third++ = Facet(c, i);
```

```
      // Go to the next adjacent cell.
      continue;
    }

    // If the neighbor does not have its in_conflict_flag set, then
    // we use tester to see if it is in conflict. If so, we explore
    // it by making a recursive call to find_conflicts. Also, we
    // compare memory addresses of c and test and add a facet to
    // the internal facet container if appropriate.
    if (test->get_in_conflict_flag() == 0) {
      if (tester(test)) {
        if (c < test)
          *it.third++ = Facet(c, i); // Internal facet.
        it = find_conflicts<D>(test, tester, it);

        // Go to the next adjacent cell.
        continue;
      }

      // We know c is in conflict and test is not in
      // conflict. Therefore, we add a facet to the container of
      // facets on the boundary of the conflict region. We reset
      // the in_conflict_flag within the insert_in_hole_ function
      // of the data structure.
      test->set_in_conflict_flag(2);
    }

    *it.first++ = Facet(c, i);
  }

  return it;
}

template <int D, typename Conflict_test>
Vertex_handle
insert_conflict (const Cell_handle & c,
                 const Conflict_test & tester)
{
  CGAL_triangulation_precondition(tester(c));

  // A container of cells that will be filled with conflict cells.
  std::vector<Cell_handle> cells;
```

```
  // Will store a facet on the boundary of the conflict region.
  Facet facet;

  // Oneset_iterator means that only one item will be stored (only
  //   one facet on the boundary of the conflict region is
  //   necessary).
  // std::back_inserter allows operations like *cells++ = c (all
  //   the conflict cells will be added to the cells vector).
  // Emptyset_iterator means that any attempts to add objects via
  //   this iterator will be ignored (we do not need internal
  //   facets here).
  find_conflicts<D>(c, tester,
                    make_triple(Oneset_iterator<Facet>(facet),
                                std::back_inserter(cells),
                                Emptyset_iterator()));

  // Call the data structure function. It will star the region and
  // then delete the old cells from the triangulation.
  return tds_.insert_in_hole_(cells.begin(), cells.end(),
                              facet.first, facet.second);
}
```

In addition to the vertex iterator provided by the `Triangulation_4<Gt,Tds>` class, the `Regular_triangulation_4<Gt,Tds>` class also provides classes named `Visible_points_iterator`, `Hidden_points_iterator`, and `Points_iterator`. The last iterator simply concatenates visible and hidden weighted points. These are iterators over weighted points (not vertices). Whereas visible sites are stored within vertices, hidden sites are stored within the cells that hide them, so there is no vertex associated with a hidden site. Also, providing an iterator over all hidden weighted points becomes tricky because they are stored throughout the cells of the triangulation. We essentially need to loop over all cells, and then within each cell loop over all hidden weighted points. CGAL's `Nested_iterator<...>` class is used to encapsulate this behavior, which results in the appearance that the hidden weighted points are all stored in a single flat container.

The public interface of the regular triangulation class can be found in Appendix

A.2.

4.5 Correctness/verification (the `is_valid` methods)

Inevitably, there were errors encountered during the development process. Testing the code was performed throughout by inserting points or weighted points randomly distributed on the grid, traversing the resulting triangulation using all available circulators, enumerators, and iterators, and then using the `is_valid` methods. In some cases, the program would crash during insertion or traversal. In other cases, all points would be inserted successfully, but errors were still present in the code so that an incorrect triangulation was formed. In both cases, errors were identified and corrected. The `is_valid` methods were written to provide checks locally throughout the structure in order to ensure that the global triangulation constructed is correct. In addition to debugging, users may use `is_valid` to verify correctness of the constructed triangulation.

At the lowest level, each cell and vertex has `is_valid` methods. First, each vertex stores a `Cell_handle` to an adjacent cell. The `is_valid` method checks that a valid cell is stored. Additionally, the vertices of this adjacent cell are examined to ensure that it contains the vertex in question. The `is_valid` method for cells checks that exactly the correct number of `Vertex_handle`s are used, depending on the dimension of the triangulation. Also, adjacency relationships are examined to check that a neighbor cell n of cell c has c as a neighbor and that the correct `Vertex_handle`s are shared. These are simple sanity checks to make sure that the data structure is operating correctly on the lowest level.

At the level of the data structure, the `is_valid` method counts vertices, edges, faces, facets, and cells to make sure that the triangulation satisfies the Euler relation. Depending on the dimension, the following formulas need to be satisfied.

- $d = 4 : cells - facets + faces - edges + vertices = 2$
- $d = 3 : facets - faces + edges - vertices = 0$
- $d = 2 : faces - edges + vertices = 2$
- $d = 1 : edges - vertices = 0$
- $d = 0 : vertices = 2$

Note that infinite simplices are included in these counts. For example, when the dimension is zero, an infinite vertex and one finite vertex compose the triangulation. A one-dimensional triangulation can be thought of as a ring of alternating edges and vertices, so the edge count must equal the vertex count. Also note the alternating value of the Euler relation between 0 and 2 as the dimension increases.

The `Triangulation_4::is_valid` method checks for badly oriented cells. For each (finite) cell in the triangulation, an orientation test is performed, and each result is expected to be `POSITIVE`. The `Regular_triangulation_4::is_valid` method checks that the power test is satisfied locally throughout the triangulation. For each cell c, each neighbor n is examined to ensure that the opposite point in n is not contained in the power sphere determined by c (as long as the point is not infinite).

A call to the `is_valid` method of `Regular_triangulation_4<Gt,Tds>` generates a call to `is_valid` of `Triangulation_4<Gt,Tds>`, which generates a call to `is_valid` of the data structure, and so on until all `is_valid` methods mentioned have been verified. If at any point an error in the triangulation is found, the program exits and provides the condition that was violated.

4.6 Using the package

Before we can declare triangulation objects and start inserting points or weighted points into them, we must define a triangulation type, which we will simply call `Triangulation`. Recall from Sections 4.4 and 4.4.1 that triangulations take two

template parameters. The first is the geometric traits class, which we will need to define. The second is the data structure class, which has a default type that will be sufficient for demonstration purposes here. Therefore, we must plug a traits class into `Gt` of `Triangulation_4<Gt,Tds>` to get the `Triangulation` type that we desire.

We must parameterize our traits and kernels with a number type, and we will use an exact number type so as to avoid floating point errors. CGAL provides `Gmpq` for performing addition, subtraction, multiplication and division exactly and a multi-precision floating-point type called `MP_Float`. We choose one of them declare the `Nt` type:

```
// Choose one:
#include <CGAL/Gmpq.h>
typedef CGAL::Gmpq Nt;

#include <CGAL/MP_Float.h>
typedef CGAL::MP_Float Nt;
```

The first thing we will do is take this number type `Nt` and plug it into `Simple_cartesian_4<Nt>` to define a kernel type. We will use this kernel somehow in the type definitions that follow, so it is convenient to do this here.

```
#include <CGAL/Simple_cartesian_4.h>
typedef CGAL::Simple_cartesian_4<Nt> Kernel;
```

For basic triangulation, we will use either `Simple_cartesian_4<Nt>` directly or `Triangulation_filtered_traits_4<Kernel>` as our geometric traits class. The second option will perform arithmetic filtering, so it is recommended from an efficiency point of view.

```
// Choose one (basic triangulations):
typedef Kernel Traits;

#include <CGAL/Triangulation_filtered_traits_4.h>
typedef CGAL::Triangulation_filtered_traits_4<Kernel> Traits;
```

For regular triangulations, we cannot use the `Kernel` type directly because it does not contain the necessary power tests. There are two traits classes available, one that performs exact computation exclusively, and another that performs arithmetic filtering. Again, the filtered traits class is recommended for efficiency reasons.

```
// Choose one (regular triangulations):
#include <CGAL/Regular_triangulation_traits_4.h>
typedef CGAL::Regular_triangulation_traits_4<Kernel> Traits;

#include <CGAL/Regular_triangulation_filtered_traits_4.h>
typedef CGAL::Regular_triangulation_filtered_traits_4<Kernel>
Traits;
```

For working with points and weighted points later on in the program, we would like to define a type that allows us to do this easily, so we define the `Point` type in one of two ways (depending on whether the triangulation is basic or regular).

```
// Choose this for basic triangulations:
typedef Traits::Point_4 Point;

// Choose this for regular triangulations:
typedef Traits::Weighted_point_4 Point;
```

Finally, we define our `Triangulation` type by plugging `Traits` into the first template parameter of the interface class.

```
// Choose this for basic triangulations:
#include <CGAL/Triangulation_4.h>
typedef CGAL::Triangulation_4<Traits> Triangulation;

// Choose this for regular triangulations:
#include <CGAL/Regular_triangulation_4.h>
typedef CGAL::Regular_triangulation_4<Traits> Triangulation;
```

This organization seems complicated, but it allows the user to easily change number types, the type of traits (filtered versus non-filtered), or even the type of the triangulation between regular and basic. From now on, we will assume that

we have the `Point` and `Triangulation` types defined. First, we need points or weighted points to insert into the basic or regular triangulations. Assume that we have a text file of points (where each has either four coordinates or four coordinates plus a weight). We can build the triangulation simply by reading the points from this file and inserting them into the `tri` object.

```
int main (int argc, char* argv[])
{
  // Error checking arguments to main omitted.
  ifstream ifs(argv[1]);

  Triangulation tri;
  Point p;

  while (ifs >> p) {
    tri.insert(p);
  }

  // ...
}
```

At this point, all the points have been inserted into the triangulation. We can verify correctness of the triangulation by calling the `is_valid` method. As discussed in Section 4.5, this will cause a chain of `is_valid` calls into the data structure, cells, and vertices.

```
// We pass true into the verbose parameter so that the method
// prints that the triangulation is valid (or prints an error
// message). The method returns a bool, so the program will stop
// immediately if the triangulation is invalid because of the
//assert.
assert(tri.is_valid(true));
```

At this point, we have a valid `Triangulation` object `tri`, and we would like to extract information about the resulting structure. Describing all such functionality (including what happens when `tri` is less than four-dimensional) would be excessive,

so we work with the four-dimensional cases and try to keep the descriptions and code snippets brief.

We can output the number of cells, number of finite cells, and the number of vertices, using the corresponding member functions. We can also print counts of the intermediate-dimensional facets, faces, and edges.

```
cout << "number_of_cells() = " << tri.number_of_cells() << endl;
cout << "number_of_finite_cells() = "
     << tri.number_of_finite_cells() << endl;

cout << "number_of_facets() = " << tri.number_of_facets() << endl;
cout << "number_of_finite_facets() = "
     << tri.number_of_finite_facets() << endl;

cout << "number_of_vertices() = " << tri.number_of_vertices()
     << endl;
cout << "number_of_finite_vertices() = "
     << tri.number_of_finite_vertices() << std::endl;
```

If we have a regular triangulation object, we can also print counts of the number of visible and hidden points.

```
cout << "number_of_visible_points() = "
     << tri.number_of_visible_points() << endl;
cout << "number_of_hidden_points() = "
     << tri.number_of_hidden_points() << endl;
```

But we are probably more interested in the cells of the triangulation, so we use an iterator to traverse and print each cell. In this case, we use the `All_cells_iterator` that includes infinite cells. Since the infinite vertex does not have a geometric point or weighted point associated with it, we must be careful to distinguish it from the finite vertices by calling the `is_infinite` member function.

```
int all_cells_count = 0;
Triangulation::All_cells_iterator acells_it;
for (acells_it = tri.all_cells_begin();
     acells_it != tri.all_cells_end(); ++acells_it) {
```

```
  cout << "Cell " << ++all_cells_count << ": ";
  for (int i = 0; i <= tri.dimension(); ++i) {
    cout << "P" << i << ":";
    if (tri.is_infinite(acells_it->vertex(i)))
      cout << "INF ";
    else
      cout << acells_it->vertex(i)->point() << " ";
  }
  cout << endl;
}
```

The following iterators are also provided and can be used in a similar manner. The corresponding examples are omitted.

- `Finite_cells_iterator`
- `All_facets_iterator`
- `Finite_facets_iterator`
- `All_faces_iterator`
- `Finite_faces_iterator`
- `All_edges_iterator`
- `Finite_edges_iterator`
- `All_vertices_iterator`
- `Finite_vertices_iterator`

The final examples we provide involve circulating cells around a face, and enumerating cells around an edge. We would retrieve a face or edge of interest using the iterators described above. Then, we construct circulator and enumerator objects, and use them to visit all the adjacent cells.

```
// Assume we grab a face from the fface_it Finite_faces_iterator.
Triangulation::Cell_around_face_circulator cc;
cc = tri.incident_cells(fface_it->first, fface_it->second,
                        fface_it->third);
Triangulation::Cell_handle start(cc);

int k = 0;
```

```
do {

  // Print the current cell.
  cout << "Cell " << ++k << ": ";
  Triangulation::Cell_handle cur(cc);
  for (int i = 0; i <= tri.dimension(); ++i) {
    cout << "P" << i << ":";
    if (tri.is_infinite(cur->vertex(i)))
      cout << "INF ";
    else
      cout << cur->vertex(i)->point() << " ";
  }
  cout << endl;

  // Move to the next cell
  ++cc;

// Stop when we return to the starting cell.
} while (Triangulation::Cell_handle(cc) != start);

// Assume we grab an edge from the fedge_it Finite_edges_iterator.
Triangulation::Cell_around_edge_enumerator ce(*fedge_it);
Triangulation::Cell_around_edge_enumerator::Cell_container_iterator
cit;

int k = 0;
for (cit = ce.begin(); cit != ce.end(); ++cit) {

  // Print the current cell.
  cout << "Cell " << ++k << ": ";
  Triangulation::Cell_handle cur(*cit);
  for (int i = 0; i <= tri.dimension(); ++i) {
    cout << "P" << i << ":";
    if (tri.is_infinite(cur->vertex(i)))
      cout << "INF ";
    else
      cout << cur->vertex(i)->point() << " ";
  }
  cout << endl;

  // Move to the next cell.
}
```

In addition, the triangulation interface also provides the following circulators

and enumerators, which operate similarly to those described above. The examples are again omitted.

- `Cell_around_vertex_enumerator` (4D only)
- `Facet_around_edge_circulator` (3D only)
- `Facet_around_vertex_enumerator` (3D only)
- `Face_around_vertex_circulator` (2D only)
- `Vertex_around_vertex_circulator` (2D only)
- `Vertex_around_vertex_enumerator` (3D and 4D)

CHAPTER 5

CONCLUSION

5.1 Summary

A four-dimensional triangulation package was designed and implemented. To represent this, a data structure was designed and implemented. The data structure stores cells and vertices directly, and represents intermediate-dimensional simplices implicitly. This data structure may be viewed as a container of cells where each cell has five neighbors. In essence, the data structure is an abstraction of a circular doubly linked list. A circular doubly linked list can be used to represent a one-dimensional triangulation that includes an infinite vertex. Each node in the linked list represents a cell, and each cell has exactly two neighbors. The four-dimensional triangulation data structure is a generalization of this where each node has five neighbors instead of the usual two. Additionally, circulators, enumerators, and iterators were implemented to allow for convenient traversal of the structure after the triangulation has been built. The second component of the design involved abstracting the geometric predicates in order for them to operate on four-dimensional points and weighted points. Finally, the two components were tied together via the basic and regular triangulation algorithms, which also provided the interfaces to the user.

The implementation will be contributed to CGAL, the Computational Geometry Algorithms Library. After that, it may be used in the design of other packages, such

as the three-dimensional Apollonius diagram or Möbius diagram packages. It may also be used for direct application as well.

5.2 Future work

Because the Delaunay triangulation is a special case of the regular triangulation, the regular triangulation package can be used to construct Delaunay triangulations simply by setting the weights of all the input points equal to each other. However, creating a specific Delaunay triangulation interface would be desirable from the point of view of the users and efficiency. This could be accomplished in one of two ways:

1. Create a wrapper around the regular triangulation package, or
2. Create a package that makes use of the same data structure, but has separate implementations (using the sphere test instead of the power test).

Both approaches can be used to create the same interface for constructing and traversing Delaunay triangulations. Creating a wrapper around the regular triangulation package would be the easiest approach from the implementation point of view. In this case, no new data structures, algorithms, or geometric predicates would need to be written. The power test would simply operate on weighted points whose weight is always zero. However, creating a customized Delaunay triangulation package would lead to a more efficient implementation. First, memory would not be wasted storing the weight coordinate of each point when this is not necessary for the Delaunay triangulation. Secondly, a direct implementation of the sphere test would execute faster than the more general power test. This is because fewer operations would be evaluated due to the fact that the weight would no longer be stored. The data structure and many details of the algorithm could be reused, but the sphere test geometric predicate would need to be written.

Currently, the packages are not dynamic in that they do not support deletion of sites. Storage of hidden weighted points within cells of the regular triangulation is already implemented, which is the first step toward unhiding points after a visible weighted point has been deleted. Devillers [16] and Devillers and Teillaud [17] discuss vertex removal in three-dimensional Delaunay triangulations. Perhaps a similar approach will be applicable to four-dimensional basic and regular triangulations.

A practical optimization for point location that has been implemented in the two-dimensional CGAL triangulation package is called the triangulation hierarchy. The full triangulation is stored at the bottom of several levels of triangulations. A subset of sites are kept from one level to the next higher level. Point location is executed in a top down fashion, and uses the result of location at one level as the starting cell in the next lower level. Point location finishes when the query point has been located in the full triangulation. Because of the overhead involved, the triangulation hierarchy typically works best for large input sets. This hierarchy idea may be generalized to operate in the four-dimensional triangulations presented here in order to speed up point location queries on triangulations with many input sites.

Generalizing the code to operate in $\mathbb{R}^d$ is an obvious direction for future work. One way to do this would be to make the dimension a template parameter, which means that the value would need to be known at compile time. Some components would extend straightforwardly (but not necessarily easily) into general dimensions. For example, using an array to represent points or evaluating determinants of arbitrary size by introducing a `Matrix` type. However, data structure issues may be difficult to resolve. Currently, the dimension is fixed at four, which allows the types `Facet`, `Face`, and `Edge` to be hardcoded, along with code for circulation and enumeration of cells around these lower-dimensional simplices. While fully-dimensional cells and vertices may still be stored, types for intermediate-dimensional simplices

can no longer be hardcoded. Furthermore, circulation currently depends on integer values stored in lookup tables. These values were determined at implementation time and programmed into the lookup tables. A scheme not involving hardcoded types and programmed lookup tables would need to be devised if circulation and enumeration of simplices around lower-dimensional faces is desired.

As mentioned in the introduction, power diagrams (and their dual regular triangulations) are general structures that are related to other diagrams such as the Voronoi diagram of spheres and the Möbius diagram. The four-dimensional regular triangulation package may be used in the development of these other packages. Boissonnat and Delage [9] compute each region of the three-dimensional Voronoi diagram of spheres individually using a reduction to a three-dimensional power diagram, and then link the regions together appropriately. Will [39] computes a single region of the Voronoi diagram of spheres, and Gavrilova and Rokne [25] discuss the Voronoi diagram of spheres in general dimension for moving spheres. However, these works do not compute the entire diagram via a reduction to a power diagram in one dimension higher.

5.3 Final thoughts

Recall the problem of inserting a point outside the affine hull of a three-dimensional triangulation to increase the dimension to four, which was discussed in Section 3.4.5. Attempting to solve this problem directly would be difficult because of the difficulty in visualizing and thinking in four dimensions. The key is to first solve the related lower-dimensional problems, and then abstract the solutions to work for the four-dimensional case. Although this approach allows one to more easily think in four dimensions in many cases, it does not take away the complexity and difficulty of solving problems in higher dimensions. Solving these problems are not

always straightforward extensions of lower-dimensional cases, and new techniques needed to be developed accordingly. Most notably, the data structure and the type of traversal operations supported needed to be developed to represent four-dimensional triangulations correctly. Enumeration was introduced not only as a convenience, but because it is used in the algorithm for inserting a point on an edge in a four-dimensional triangulation. For the same reason, circulation of cells around a face was implemented, which was not a straightforward extension of the three-dimensional situation of circulation of facets around an edge. In three dimensions, the edge has a direction depending on the order of its vertices, and the right-hand rule can be used to determine which adjacent cell to visit on an increment operation. In four dimensions, the face being circulated around has an orientation, and the spirit of moving to the correct adjacent cell on an increment operation remains, but the right hand rule can no longer be used to determine the correct adjacent cell. Instead, something more complicated involving encoding orientations into lookup tables was devised to accomplish this.

The point is that complexity usually increases when increasing the dimension of the problem, and partly this has to do with the difficulty in visualizing higher-dimensional space. Sometimes solutions are easily generalized from lower-dimensional cases, and in other situations a generalization is not intuitively obvious.

APPENDIX A

PUBLIC INTERFACES AND CODE EXAMPLES

The Appendix provides more extensive code examples compared to those presented in the text. Specifically, we present the public interfaces of the basic and regular triangulation packages. This is followed by an example of the regular triangulation traits class, and how we use CGAL's `Filtered_predicate<...>` to build a traits class that performs arithmetic filtering. Finally, we present an extended code example corresponding to the snippets presented in Section 4.6. This example demonstrates usage of most of the types and functions in the public interfaces of the triangulation and regular triangulation classes.

A.1 Public interface of `Triangulation_4<Gt,Tds>`

```
template <class Gt,
          class Tds = Triangulation_data_structure_4<
                       Triangulation_vertex_base_4<Gt>,
                       Triangulation_cell_base_4<Gt> > >
class Triangulation_4
  : public Triangulation_utils_4
{
 public:
  typedef Gt Geom_traits;
  typedef Tds Triangulation_data_structure;

  typedef typename Gt::Point_4       Point_4;
  typedef typename Gt::Segment_4     Segment_4;
  typedef typename Gt::Triangle_4    Triangle_4;
  typedef typename Gt::Tetrahedron_4 Tetrahedron_4;
  typedef typename Gt::Pentahedron_4 Pentahedron_4;

  typedef typename Tds::Vertex       Vertex;
  typedef typename Tds::Edge         Edge;
  typedef typename Tds::Face         Face;
  typedef typename Tds::Facet        Facet;
  typedef typename Tds::Cell         Cell;

  typedef Vertex                     Simplex_0;
  typedef Edge                       Simplex_1;
  typedef Face                       Simplex_2;
  typedef Facet                      Simplex_3;
  typedef Cell                       Simplex_4;

  typedef typename Tds::size_type size_type;
  typedef typename Tds::difference_type difference_type;

  typedef typename Tds::Vertex_handle Vertex_handle;
  typedef typename Tds::Cell_handle Cell_handle;

  // CIRCULATOR AND ENUMERATOR TYPES.
  typedef typename Tds::Cell_around_edge_enumerator
  Cell_around_edge_enumerator;

  typedef typename Tds::Cell_around_vertex_enumerator
  Cell_around_vertex_enumerator;
```

```
typedef typename Tds::Cell_around_face_circulator
Cell_around_face_circulator;

typedef typename Tds::Facet_around_vertex_enumerator
Facet_around_vertex_enumerator;

typedef typename Tds::Facet_around_edge_circulator
Facet_around_edge_circulator;

typedef typename Tds::Vertex_around_vertex_enumerator
Vertex_around_vertex_enumerator;

typedef typename Tds::Face_around_vertex_circulator
Face_around_vertex_circulator;

typedef typename Tds::Vertex_around_vertex_circulator
Vertex_around_vertex_circulator;

// ITERATOR TYPES.
typedef typename Tds::Cell_iterator   All_cells_iterator;
typedef typename Tds::Facet_iterator  All_facets_iterator;
typedef typename Tds::Face_iterator   All_faces_iterator;
typedef typename Tds::Edge_iterator   All_edges_iterator;
typedef typename Tds::Vertex_iterator All_vertices_iterator;

class Finite_cells_iterator    {/* ... */};
class Finite_vertices_iterator {/* ... */};

// Filter_iterator is a standard CGAL class template used to
// transform one iterator into another. In this case, it causes
// infinite features to be skipped.
typedef Filter_iterator<All_edges_iterator, Infinite_tester>
Finite_edges_iterator;

typedef Filter_iterator<All_faces_iterator, Infinite_tester>
Finite_faces_iterator;

typedef Filter_iterator<All_facets_iterator, Infinite_tester>
Finite_facets_iterator;

// Iterator_project is a standard CGAL class template used to
// transform one iterator into another. In this case, we are
// transforming a vertex iterator into a point iterator.
typedef Iterator_project<Finite_vertices_iterator,
```

```
                          Proj_point,
                          const Point_4 &,
                          const Point_4 *,
                          std::ptrdiff_t,
                          std::bidirectional_iterator_tag>
Point_4_iterator;

typedef Point_4            value_type;
typedef const value_type & const_reference;

enum Locate_type {
  VERTEX = 0,
  EDGE,
  FACE,
  FACET,
  CELL,
  OUTSIDE_CONVEX_HULL,
  OUTSIDE_AFFINE_HULL
};

// CONSTRUCTORS.
Triangulation_4 (const Gt & gt = Gt());
Triangulation_4 (const Triangulation_4 & tr);

template <typename InputIterator>
Triangulation_4 (InputIterator first, InputIterator last,
                 const Gt & gt = Gt());

// ASSIGNMENT.
Triangulation_4 & operator= (const Triangulation_4 & tr);
void clear ();
void swap (Triangulation_4 & tr);

// ACCESS.
const Gt & geom_traits () const;
const Tds & tds () const;
Tds & tds (); // Advanced.

// QUERIES that return numbers.
size_type number_of_finite_cells    () const;
size_type number_of_cells           () const;
size_type number_of_finite_facets   () const;
size_type number_of_facets          () const;
size_type number_of_finite_faces    () const;
```

```
size_type number_of_faces           () const;
size_type number_of_finite_edges    () const;
size_type number_of_edges           () const;
size_type number_of_finite_vertices () const;
size_type number_of_vertices        () const;

int       dimension                 () const;

unsigned int degree (const Vertex_handle & v) const;

// ACCESS.
Vertex_handle infinite_vertex () const;
Vertex_handle finite_vertex   () const;
Cell_handle   infinite_cell   () const;

Pentahedron_4 pentahedron_4 (const Cell_handle & c) const;
Tetrahedron_4 tetrahedron_4 (const Cell_handle & c,
                             int i)                 const;
Tetrahedron_4 tetrahedron_4 (const Facet & f)       const;
Triangle_4    triangle_4    (const Cell_handle & c,
                             int i, int j)          const;
Triangle_4    triangle_4    (const Face & f)        const;
Segment_4     segment_4     (const Cell_handle & c,
                             int i, int j, int k)   const;
Segment_4     segment_4     (const Edge & e)        const;

// QUERIES that return bool.
bool is_infinite (const Vertex_handle & v)   const;
bool is_infinite (const Cell_handle & c)     const;
bool is_infinite (const Cell_handle & c,
                  int i)                     const;
bool is_infinite (const Facet & f)           const;
bool is_infinite (const Cell_handle & c,
                  int i, int j)              const;
bool is_infinite (const Face & f)            const;
bool is_infinite (const Cell_handle & c,
                  int i, int j, int k)       const;
bool is_infinite (const Edge & e)            const;

bool is_vertex   (const Point_4 & p,
                  Vertex_handle & v)         const;
bool is_vertex   (const Vertex_handle & v)   const;
bool is_edge     (const Vertex_handle & u,
                  const Vertex_handle & v,
```

```
                 Cell_handle & c,
                 int & i, int & j) const;
bool is_face     (const Vertex_handle & u,
                  const Vertex_handle & v,
                  const Vertex_handle & w,
                  Cell_handle & c,
                  int & i, int & j, int & k) const;
bool is_facet    (const Vertex_handle & u,
                  const Vertex_handle & v,
                  const Vertex_handle & w,
                  const Vertex_handle & x,
                  Cell_handle & c,
                  int & i, int & j, int & k,
                           int & l)          const;
bool is_cell     (const Cell_handle & c)     const;
bool is_cell     (const Vertex_handle & u,
                  const Vertex_handle & v,
                  const Vertex_handle & w,
                  const Vertex_handle & x,
                  const Vertex_handle & y,
                  Cell_handle & c,
                  int & i, int & j, int & k,
                  int & l, int & m) const;
bool is_cell     (const Vertex_handle & u,
                  const Vertex_handle & v,
                  const Vertex_handle & w,
                  const Vertex_handle & x,
                  const Vertex_handle & y,
                  Cell_handle & c)           const;
bool has_vertex  (const Facet & f,
                  Vertex_handle v, int & j)  const;
bool has_vertex  (const Cell_handle & c,
                  int i,
                  const Vertex_handle & v,
                  int & j)                   const;
bool has_vertex  (const Facet & f,
                  const Vertex_handle & v)   const;
bool has_vertex  (const Cell_handle & c,
                  int i,
                  const Vertex_handle & v)   const;
bool are_equal   (const Cell_handle & c,
                  int i,
                  const Cell_handle & n,
                  int j)                     const;
```

```
bool are_equal   (const Facet & f,
                  const Facet & g)           const;
bool are_equal   (const Facet & f,
                  const Cell_handle & n,
                  int j)                     const;

// POINT LOCATION.
Cell_handle locate (const Point_4 & p,
                    Locate_type & lt,
                    int & li, int & lj, int & lk,
                    Cell_handle start = Cell_handle()) const;
Cell_handle
locate (const Point_4 & p,
        const Cell_handle & start = Cell_handle()) const;

// INSERTION.
Vertex_handle
insert (const Point_4 & p,
        const Cell_handle & start = Cell_handle());

Vertex_handle
insert (const Point_4 & p, const Locate_type & lt,
        const Cell_handle & c,
        int li, int lj, int lk);

template <class InputIterator>
int insert (InputIterator first, InputIterator last);

template <class CellIt>
Vertex_handle insert_in_hole (const Point_4 & p,
                              CellIt cell_begin, CellIt cell_end,
                              Cell_handle begin, int i);

// ITERATORS.
Finite_cells_iterator    finite_cells_begin    () const;
Finite_cells_iterator    finite_cells_end      () const;
All_cells_iterator       all_cells_begin       () const;
All_cells_iterator       all_cells_end         () const;
Finite_facets_iterator   finite_facets_begin   () const;
Finite_facets_iterator   finite_facets_end     () const;
All_facets_iterator      all_facets_begin      () const;
All_facets_iterator      all_facets_end        () const;
Finite_faces_iterator    finite_faces_begin    () const;
Finite_faces_iterator    finite_faces_end      () const;
```

```
  All_faces_iterator       all_faces_begin       () const;
  All_faces_iterator       all_faces_end         () const;
  Finite_edges_iterator    finite_edges_begin    () const;
  Finite_edges_iterator    finite_edges_end      () const;
  All_edges_iterator       all_edges_begin       () const;
  All_edges_iterator       all_edges_end         () const;
  Finite_vertices_iterator finite_vertices_begin () const;
  Finite_vertices_iterator finite_vertices_end   () const;
  All_vertices_iterator    all_vertices_begin    () const;
  All_vertices_iterator    all_vertices_end      () const;
  Point_4_iterator         points_begin          () const;
  Point_4_iterator         points_end            () const;

  // CIRCULATORS.
  Cell_around_face_circulator
  incident_cells  (const Cell_handle & c,
                   int i, int j)              const;
  Cell_around_face_circulator
  incident_cells  (const Face & f)            const;

  Facet_around_edge_circulator
  incident_facets (const Cell_handle & c,
                   int i, int j)              const;
  Facet_around_edge_circulator
  incident_facets (const Edge & e)            const;

  Face_around_vertex_circulator
  incident_faces  (const Vertex_handle & v)   const;

  Vertex_around_vertex_circulator
  incident_vertices (const Vertex_handle & v) const;

  // VALIDITY CHECKING.
  bool is_valid (bool verbose = false, int level = 0) const;
};
```

A.2 Public interface of `Regular_triangulation_4<Gt,Tds>`

```
template <typename Gt,
          typename Tds = Triangulation_data_structure_4 <
                          Triangulation_vertex_base_4<Gt>,
                          Regular_triangulation_cell_base_4<Gt> > >
class Regular_triangulation_4
  : public Triangulation_4<Gt, Tds>
{
public:
  typedef Tds Triangulation_data_structure;
  typedef Gt  Geom_traits;

  typedef typename Base::Vertex_handle  Vertex_handle;
  typedef typename Base::Cell_handle    Cell_handle;

  typedef typename Base::Vertex         Vertex;
  typedef typename Base::Edge           Edge;
  typedef typename Base::Face           Face;
  typedef typename Base::Facet          Facet;
  typedef typename Base::Cell           Cell;

  typedef typename Base::Locate_type    Locate_type;
  typedef typename Base::size_type      size_type;

  typedef typename Gt::Bare_point_4     Bare_point_4;
  typedef typename Gt::Weighted_point_4 Weighted_point_4;
  typedef typename Gt::Weight           Weight;

  // ITERATOR TYPES.
  typedef typename Base::All_cells_iterator
  All_cells_iterator;
  typedef typename Base::All_facets_iterator
  All_facets_iterator;
  typedef typename Base::All_faces_iterator
  All_faces_iterator;
  typedef typename Base::All_edges_iterator
  All_edges_iterator;
  typedef typename Base::Finite_cells_iterator
  Finite_cells_iterator;
  typedef typename Base::Finite_facets_iterator
  Finite_facets_iterator;
  typedef typename Base::Finite_faces_iterator
  Finite_faces_iterator;
```

```
typedef typename Base::Finite_edges_iterator
Finite_edges_iterator;
typedef typename Base::Finite_vertices_iterator
Finite_vertices_iterator;

// Iterator_project is a standard CGAL class template used to
// transform one iterator into another. In this case, we are
// transforming a vertex iterator into a weighted point iterator.
typedef CGALi::Project_point_4<Vertex> Proj_point;
typedef Iterator_project<Finite_vertices_iterator, Proj_point>
Visible_points_iterator;

// Nested_iterator is a standard CGAL class template used to
// provide a flat view when containers are nested inside of
// containers. In this case, instead of looping over all cells
// and then looping over all hidden weighted points within each
// cell, we simply loop over all hidden weighted points.
typedef
Regular_triangulation_cell_base_nested_iterator_traits_4
<Finite_cells_iterator>
Hidden_points_nested_iterator_traits_4;
typedef Nested_iterator<Finite_cells_iterator,
                        Hidden_points_nested_iterator_traits_4>
Hidden_points_iterator;

// Concatenate_iterator is a standard CGAL class template used to
// provide an iterator that ties together two separate
// containers. In this case we loop over all weighted points by
// concatenating the visible and hidden points.
typedef Concatenate_iterator<Visible_points_iterator,
                             Hidden_points_iterator>
Points_iterator;

// CONSTRUCTION.
Regular_triangulation_4 (const Gt & gt = Gt());
Regular_triangulation_4 (const Regular_triangulation_4 & rt);

template <typename InputIterator>
Regular_triangulation_4 (InputIterator first, InputIterator last,
                         const Gt & gt = Gt());

// ITERATORS.
using Base::all_cells_begin;
using Base::all_cells_end;
```

```
using Base::all_facets_begin;
using Base::all_facets_end;
using Base::all_faces_begin;
using Base::all_faces_end;
using Base::all_edges_begin;
using Base::all_edges_end;
using Base::finite_cells_begin;
using Base::finite_cells_end;
using Base::finite_facets_begin;
using Base::finite_facets_end;
using Base::finite_faces_begin;
using Base::finite_faces_end;
using Base::finite_edges_begin;
using Base::finite_edges_end;
using Base::finite_vertices_begin;
using Base::finite_vertices_end;

Visible_points_iterator visible_points_begin () const;
Visible_points_iterator visible_points_end () const;
Hidden_points_iterator hidden_points_begin () const;
Hidden_points_iterator hidden_points_end () const;
Points_iterator points_begin () const;
Points_iterator points_end () const;

// ACCESS.
using Base::tds;

// QUERIES that return numbers.
size_type number_of_vertices () const;
size_type number_of_visible_points () const;
size_type number_of_hidden_points () const;

// INSERTION.
Vertex_handle insert (const Weighted_point_4 & p,
                      Cell_handle start = Cell_handle());

template <class InputIterator>
size_type insert (InputIterator first, InputIterator last);
Vertex_handle insert (const Weighted_point_4 & p, Locate_type lt,
                      const Cell_handle & c,
                      int li, int lj, int lk);

// VALIDITY CHECKING.
bool is_valid (bool verbose = false, int level = 0) const;
```

```
};
```

A.3 Regular_triangulation_traits_4<Kernel>

```
template <typename K>
class Regular_triangulation_traits_4
  : public K
{
public:
  // OBJECTS.
  typedef typename K::Point_4                  Bare_point_4;
  typedef typename K::RT                       Weight;

  // External Weighted_point class provided in CGAL.
  typedef CGAL::Weighted_point<Bare_point_4, Weight>
  Weighted_point_4;
  typedef Weighted_point_4                     Point_4;

  // PREDICATES.
  // External Power_test_4 class provided.
  typedef CGAL::Power_test_4<Weighted_point_4> Power_test_4;

  Power_test_4 power_test_4_object() const
  {
    return Power_test_4();
  }
};
```

A.4 Regular_triangulation_filtered_traits_4<Kernel>

```
template <typename CK_t,
        typename CK_MTag = Ring_tag,
        typename EK_t    = Simple_cartesian_4<MP_Float>,
        typename EK_MTag = CK_MTag,
        typename FK_t    = Simple_cartesian_4<Interval_nt<false> >,
        typename FK_MTag = CK_MTag,
        typename C2E_t   = Cartesian_converter<CK_t, EK_t>,
        typename C2F_t   =
           Cartesian_converter<CK_t, FK_t,
                    To_interval<typename CK_t::RT> > >
class Regular_triangulation_filtered_traits_4
  : public Triangulation_filtered_traits_4<CK_t, CK_MTag,
                                           EK_t, EK_MTag,
                                           FK_t, FK_MTag,
                                           C2E_t, C2F_t>
{
private:
  typedef Regular_triangulation_traits_4<CK_t> CK_traits;
  typedef Regular_triangulation_traits_4<EK_t> EK_traits;
  typedef Regular_triangulation_traits_4<FK_t> FK_traits;

  typedef Regular_triangulation_cartesian_converter_4
  <CK_traits, EK_traits, C2E_t> C2E;
  typedef Regular_triangulation_cartesian_converter_4
  <CK_traits, FK_traits, C2F_t> C2F;

  // Types for the construction kernel.
  typedef typename CK_traits::Bare_point_4
  CK_traits_Bare_point_4;
  typedef typename CK_traits::Weight
  CK_traits_Weight;
  typedef typename CK_traits::Weighted_point_4
  CK_traits_Weighted_point_4;
  typedef typename CK_traits::Point_4
  CK_traits_Point_4;

  // Types for the exact kernel.
  typedef typename EK_traits::Bare_point_4
  EK_traits_Bare_point_4;
  typedef typename EK_traits::Weight
  EK_traits_Weight;
  typedef typename EK_traits::Weighted_point_4
```

```
  EK_traits_Weighted_point_4;
  typedef typename EK_traits::Point_4
  EK_traits_Point_4;

  // Types for the filtering kernel.
  typedef typename FK_traits::Bare_point_4
  FK_traits_Bare_point_4;
  typedef typename FK_traits::Weight
  FK_traits_Weight;
  typedef typename FK_traits::Weighted_point_4
  FK_traits_Weighted_point_4;
  typedef typename FK_traits::Point_4
  FK_traits_Point_4;

public:
  typedef CK_t      R;
  typedef CK_MTag   Method_tag;

  typedef CK_traits Construction_traits;
  typedef EK_traits Exact_traits;
  typedef FK_traits Filtering_traits;

  typedef CK_MTag   Construction_traits_method_tag;
  typedef EK_MTag   Exact_traits_method_tag;
  typedef FK_MTag   Filtering_traits_method_tag;

  typedef typename CK_traits::Bare_point_4
  Bare_point_4;
  typedef typename CK_traits::Weight
  Weight;
  typedef typename CK_traits::Weighted_point_4
  Weighted_point_4;
  typedef typename CK_traits::Point_4
  Point_4;

private:
  // No predicates for the construction kernel.

  // Predicates for the exact kernel.
  typedef typename EK_traits::Power_test_4
  EK_traits_Power_test_4;

  // Predicates for the filtering kernel.
  typedef typename FK_traits::Power_test_4
```

```
  FK_traits_Power_test_4;

public:
  // External Filtered_predicate class provided by CGAL.
  typedef Filtered_predicate
  <EK_traits_Power_test_4, FK_traits_Power_test_4, C2E, C2F>
  Power_test_4;

public:
  Power_test_4
  power_test_4_object () const
  {
    return Power_test_4();
  }
};
```

A.5 Public interface of Triangulation_data_structure_4<Vb,Cb>

```
template <class Vb = Triangulation_ds_vertex_base_4<>,
          class Cb = Triangulation_ds_cell_base_4<> >
class Triangulation_data_structure_4
  : public Triangulation_utils_4
{
  typedef Triangulation_data_structure_4<Vb,Cb>         Self;
  typedef typename Vb::template Rebind_Tds<Self>::Other
  Vertex_base;
  typedef typename Cb::template Rebind_Tds<Self>::Other Cell_base;

  // These classes are provided in this implementation.
  typedef Triangulation_ds_vertex_4<Vertex_base>   Vertex;
  typedef Triangulation_ds_cell_4<Cell_base>       Cell;

  typedef typename Cell_container::size_type       size_type;
  typedef typename Cell_container::difference_type difference_type;

  typedef typename Cell_container::iterator        Cell_iterator;
  typedef typename Vertex_container::iterator      Vertex_iterator;

  // These classes are provided in this implementation.
  typedef Triangulation_ds_facet_iterator_4<Self>  Facet_iterator;
  typedef Triangulation_ds_face_iterator_4<Self>   Face_iterator;
  typedef Triangulation_ds_edge_iterator_4<Self>   Edge_iterator;

  // Represent handles with iterators
  typedef Vertex_iterator                          Vertex_handle;
  typedef Cell_iterator                            Cell_handle;

  // CGAL provides Quadruple and Triple.
  typedef Quadruple<Cell_handle, int, int, int>    Edge;
  typedef Triple<Cell_handle, int, int>            Face;
  typedef std::pair<Cell_handle, int>              Facet;

  // ENUMERATOR TYPES.
  // A class for each circulator and enumerator is provided by this
  // implementation.

  // 4D only.
  typedef Triangulation_ds_cell_around_edge_enumerator_4<Self>
  Cell_around_edge_enumerator;
```

```
// 4D only.
typedef Triangulation_ds_cell_around_vertex_enumerator_4<Self>
Cell_around_vertex_enumerator;

// 3D only.
typedef Triangulation_ds_facet_around_vertex_enumerator_4<Self>
Facet_around_vertex_enumerator;

// 3D or 4D. Need to pass the dimension when constructing the
// class.
typedef Triangulation_ds_vertex_around_vertex_enumerator_4<Self>
Vertex_around_vertex_enumerator;

// CIRCULATOR TYPES.

// 4D only.
typedef Triangulation_ds_cell_around_face_circulator_4<Self>
Cell_around_face_circulator;

// 3D only.
typedef Triangulation_ds_facet_around_edge_circulator_4<Self>
Facet_around_edge_circulator;

// 2D only.
typedef Triangulation_ds_face_around_vertex_circulator_4<Self>
Face_around_vertex_circulator;

// 2D only.
typedef Triangulation_ds_vertex_around_vertex_circulator_4<Self>
Vertex_around_vertex_circulator;

// CONSTRUCTION/ASSIGNMENT.
// Initially, the data structure has dimension -2. When the
// infinite vertex is inserted, the dimension increases to -1.
// When the first finite vertex is inserted, the dimension
// increases to 0, and so on.
Triangulation_data_structure_4 ();
Triangulation_data_structure_4 (const Self & tds);
Self & operator= (const Self & tds);

// QUERIES that return numbers.
// These include infinite cells, facets, faces, edges, vertices.
size_type number_of_cells    () const;
size_type number_of_facets   () const;
```

```
size_type number_of_faces    () const;
size_type number_of_edges    () const;
size_type number_of_vertices () const;

int       dimension          () const; // Current dimension.

int degree (const Vertex_handle & v) const;

// ADVANCED. May invalidate the data structure.
void set_dimension (int n);

// QUERIES that return bool.
bool is_vertex (const Vertex_handle & v)          const;
bool is_edge   (const Cell_handle & c,
                int i, int j, int k)              const;
bool is_edge   (const Vertex_handle & u,
                const Vertex_handle & v,
                Cell_handle & c,
                int & i, int & j)                 const;
bool is_edge   (const Vertex_handle & u,
                const Vertex_handle & v)          const;
bool is_face   (const Cell_handle & c,
                int i, int j)                     const;
bool is_face   (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w,
                const Cell_handle & c,
                int & i, int & j, int & k)        const;
bool is_face   (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w)          const;
bool is_facet  (const Cell_handle & c,
                int i)                            const;
bool is_facet  (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w,
                const Vertex_handle & x,
                const Cell_handle & c,
                int & i, int & j, int & k,
                         int & l)                 const;
bool is_facet  (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w,
                const Vertex_handle & x)          const;
```

```
bool is_cell   (const Cell_handle & c)           const;
bool is_cell   (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w,
                const Vertex_handle & t,
                const Vertex_handle & x,
                const Cell_handle & c,
                int & i, int & j, int & k,
                         int & l, int & m)       const;
bool is_cell   (const Vertex_handle & u,
                const Vertex_handle & v,
                const Vertex_handle & w,
                const Vertex_handle & t,
                const Vertex_handle & x)         const;
bool are_equal (const Cell_handle & c, int i,
                const Cell_handle & n, int j)    const;
bool are_equal (const Facet & f, const Facet & g) const;

// INSERTION.
Vertex_handle insert_in_cell  (const Cell_handle & c);
Vertex_handle insert_in_facet (const Facet & f);
Vertex_handle insert_in_facet (const Cell_handle & c,
                               int i);
Vertex_handle insert_in_face  (const Face & f);
Vertex_handle insert_in_face  (const Cell_handle & c,
                               int i, int j);
Vertex_handle insert_in_edge  (const Edge & e);
Vertex_handle insert_in_edge  (const Cell_handle & c,
                               int i, int j, int k);

Vertex_handle insert_increase_dimension
(const Vertex_handle & star = Vertex_handle());

// This one assumes in_conflict_flags are set, it stars region
// immediately.
template <class CellIt>
Vertex_handle insert_in_hole_ (CellIt cell_begin,
                               CellIt cell_end,
                               const Cell_handle & begin, int i);

// This one first marks in_conflict_flags, then stars region.
template <class CellIt>
Vertex_handle insert_in_hole (CellIt cell_begin, CellIt cell_end,
                              const Cell_handle & begin, int i);
```

```
// ADVANCED. May invalidate the data structure.
Vertex_handle create_vertex (const Vertex & v);
Vertex_handle create_vertex ();
Vertex_handle create_vertex (const Vertex_handle & v);
Cell_handle   create_cell   (const Cell & c);
Cell_handle   create_cell   ();
Cell_handle   create_cell   (const Cell_handle & c);
Cell_handle   create_cell   (const Vertex_handle & v0,
                             const Vertex_handle & v1,
                             const Vertex_handle & v2,
                             const Vertex_handle & v3,
                             const Vertex_handle & v4);
Cell_handle   create_cell   (const Vertex_handle & v0,
                             const Vertex_handle & v1,
                             const Vertex_handle & v2,
                             const Vertex_handle & v3,
                             const Vertex_handle & v4,
                             const Cell_handle & n0,
                             const Cell_handle & n1,
                             const Cell_handle & n2,
                             const Cell_handle & n3,
                             const Cell_handle & n4);
Cell_handle create_facet    ();
Cell_handle create_facet    (const Vertex_handle & v0,
                             const Vertex_handle & v1,
                             const Vertex_handle & v2,
                             const Vertex_handle & v3);
Cell_handle create_face     ();
Cell_handle create_face     (const Vertex_handle & v0,
                             const Vertex_handle & v1,
                             const Vertex_handle & v2);
Cell_handle create_edge     ();
Cell_handle create_edge     (const Vertex_handle & v0,
                             const Vertex_handle & v1);

void delete_vertex (const Vertex_handle & v);
void delete_cell   (const Cell_handle & c);

template <class InputIterator>
void delete_vertices (InputIterator begin, InputIterator end);

template <class InputIterator>
void delete_cells (InputIterator begin, InputIterator end);
```

```
// ITERATORS. Infinite features included.
Cell_iterator cells_begin      () const;
Cell_iterator cells_end        () const;
Facet_iterator facets_begin    () const;
Facet_iterator facets_end      () const;
Face_iterator faces_begin      () const;
Face_iterator faces_end        () const;
Edge_iterator edges_begin      () const;
Edge_iterator edges_end        () const;
Vertex_iterator vertices_begin () const;
Vertex_iterator vertices_end   () const;

// CIRCULATORS. Infinite features included.
Face_around_vertex_circulator
incident_faces    (const Vertex_handle & v) const;

Vertex_around_vertex_circulator
incident_vertices (const Vertex_handle & v) const;

Facet_around_edge_circulator
incident_facets   (const Cell_handle & c,
                   int i, int j)            const;

Facet_around_edge_circulator
incident_facets   (const Edge & e)          const;

Cell_around_face_circulator
incident_cells    (const Cell_handle & ce,
                   int i, int j)            const;

Cell_around_face_circulator
incident_cells    (const Face & f)          const;

template <typename OutputIterator>
OutputIterator
incident_cells    (const Vertex_handle & v,
                   OutputIterator cells)    const;

template <class OutputIterator>
OutputIterator
incident_vertices (const Vertex_handle & v,
                   OutputIterator vertices) const;
```

```
  // VALIDITY CHECKING.
  bool is_valid (bool verbose = false, int level = 0) const;
};
```

```
#include <CGAL/basic.h>

#include <iostream>
#include <fstream>
#include <cassert>

#if 1
  #include <CGAL/Gmpq.h>
  typedef CGAL::Gmpq Nt;
#else
  #include <CGAL/MP_Float.h>
  typedef CGAL::MP_Float Nt;
#endif

#include <CGAL/Simple_cartesian_4.h>
typedef CGAL::Simple_cartesian_4<Nt> Kernel;

#if 0
  #include <CGAL/Triangulation_4.h>
  typedef Kernel Traits;
  typedef Kernel::Point_4 Point;
  typedef CGAL::Triangulation_4<Traits> Triangulation;
  #define BASIC_TRIANGULATION
#elif 0
  #include <CGAL/Triangulation_traits_4.h>
  #include <CGAL/Triangulation_4.h>
  typedef CGAL::Triangulation_traits_4<Kernel> Traits;
  typedef Traits::Point_4 Point;
  typedef CGAL::Triangulation_4<Traits> Triangulation;
  #define BASIC_TRIANGULATION
#elif 0
  #include <CGAL/Triangulation_filtered_traits_4.h>
  #include <CGAL/Triangulation_4.h>
  typedef CGAL::Triangulation_filtered_traits_4<Kernel> Traits;
  typedef Traits::Point_4 Point;
  typedef CGAL::Triangulation_4<Traits> Triangulation;
  #define BASIC_TRIANGULATION
#elif 1
  #include <CGAL/Regular_triangulation_traits_4.h>
  #include <CGAL/Regular_triangulation_4.h>
  typedef CGAL::Regular_triangulation_traits_4<Kernel> Traits;
  typedef Traits::Weighted_point_4 Point;
```

```
  typedef CGAL::Regular_triangulation_4<Traits> Triangulation;
  #define REGULAR_TRIANGULATION
#elif 0
  #include <CGAL/Regular_triangulation_filtered_traits_4.h>
  #include <CGAL/Regular_triangulation_4.h>
  typedef CGAL::Regular_triangulation_filtered_traits_4<Kernel>
          Traits;
  typedef Traits::Weighted_point_4 Point;
  typedef CGAL::Regular_triangulation_4<Traits> Triangulation;
  #define REGULAR_TRIANGULATION
#endif

// Need only Point and Triangulation defined past this comment.

int main(int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "pass the name of an input file" << std::endl;
    return 1;
  }

  std::ifstream ifs(argv[1]);
  assert(ifs);

  Triangulation tri;

  Point p;

  while (ifs >> p) {
    tri.insert(p);
  }



  if (tri.dimension() >= 4) {
    std::cout << "number_of_cells() = "
              << tri.number_of_cells() << std::endl;
    std::cout << "number_of_finite_cells() = "
              << tri.number_of_finite_cells() << std::endl;
  }

  if (tri.dimension() >= 3) {
    std::cout << "number_of_facets() = "
              << tri.number_of_facets() << std::endl;
```

```
    std::cout << "number_of_finite_facets() = "
              << tri.number_of_finite_facets() << std::endl;
  }

  if (tri.dimension() >= 2) {
    std::cout << "number_of_faces() = "
              << tri.number_of_faces() << std::endl;
    std::cout << "number_of_finite_faces() = "
              << tri.number_of_finite_faces() << std::endl;
  }

  if (tri.dimension() >= 1) {
    std::cout << "number_of_edges() = "
              << tri.number_of_edges() << std::endl;
    std::cout << "number_of_finite_edges() = "
              << tri.number_of_finite_edges() << std::endl;
  }

  if (tri.dimension() >= 0) {
    std::cout << "number_of_vertices() = "
              << tri.number_of_vertices() << std::endl;
    std::cout << "number_of_finite_vertices() = "
              << tri.number_of_finite_vertices() << std::endl;
#ifdef REGULAR_TRIANGULATION
    std::cout << "number_of_visible_points() = "
              << tri.number_of_visible_points() << std::endl;
    std::cout << "number_of_hidden_points() = "
              << tri.number_of_hidden_points() << std::endl;
#endif
  }
  std::cout << std::endl;



  if (tri.dimension() >= 0) {
    std::cout
      << "Printing the points using finite_vertices_iterator"
      << std::endl;
    Triangulation::Finite_vertices_iterator fvertex_it;
    for (fvertex_it = tri.finite_vertices_begin();
         fvertex_it != tri.finite_vertices_end();
         fvertex_it++) {
      Triangulation::Vertex_handle v(fvertex_it);
      std::cout << v->point() << std::endl;
```

```
    }
    std::cout << std::endl;
  }


#ifdef BASIC_TRIANGULATION
  if (tri.dimension() >= 0) {
    std::cout
      << "Printing the points using Point_4_iterator"
      << std::endl;
    Triangulation::Point_4_iterator point_it;
    for (point_it = tri.points_begin();
         point_it != tri.points_end();
         point_it++) {
      std::cout << (*point_it) << std::endl;
    }
    std::cout << std::endl;
  }
#endif

#ifdef REGULAR_TRIANGULATION
  if (tri.dimension() >= 0) {
    std::cout
     << "Printing the visible points using Visible_points_iterator"
     << std::endl;
    Triangulation::Visible_points_iterator visible_points_it;
    for (visible_points_it = tri.visible_points_begin();
         visible_points_it != tri.visible_points_end();
         ++visible_points_it) {
      std::cout << *visible_points_it << std::endl;
    }
    std::cout << std::endl;
  }

  if (tri.dimension() >= 0) {
    std::cout
      << "Printing the hidden points using Hidden_points_iterator"
      << std::endl;
    Triangulation::Hidden_points_iterator hidden_points_it;
    for (hidden_points_it = tri.hidden_points_begin();
         hidden_points_it != tri.hidden_points_end();
         ++hidden_points_it) {
      std::cout << *hidden_points_it << std::endl;
    }
```

```
    std::cout << std::endl;
  }

  if (tri.dimension() >= 0) {
    std::cout
      << "Printing all points using Points_iterator"
      << std::endl;
    Triangulation::Points_iterator points_it;
    for (points_it = tri.points_begin();
         points_it != tri.points_end(); ++points_it) {
      std::cout << *points_it << std::endl;
    }
    std::cout << std::endl;
  }
#endif

  if (tri.dimension() >= 4) {
    std::cout
    << "Printing the points for each cell using All_cells_iterator"
    << std::endl;
    int all_cells_count = 0;
    Triangulation::All_cells_iterator acells_it;
    for (acells_it = tri.all_cells_begin();
         acells_it != tri.all_cells_end();
         ++acells_it) {
      std::cout << "Cell " << ++all_cells_count << ": ";
      for (int i = 0; i <= tri.dimension(); ++i) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(acells_it->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << acells_it->vertex(i)->point() << " ";
      }
      std::cout << std::endl;
    }
    std::cout << std::endl;

    std::cout
    << "Printing the points for each cell using Finite_cells_iterator"
    << std::endl;
    int finite_cells_count = 0;
    Triangulation::Finite_cells_iterator fcells_it;
    for (fcells_it = tri.finite_cells_begin();
         fcells_it != tri.finite_cells_end();
```

```
       ++fcells_it) {
    std::cout << "Cell " << ++finite_cells_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(fcells_it->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << fcells_it->vertex(i)->point() << " ";
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}



if (tri.dimension() >= 3) {
  std::cout
  << "Printing the points for each facet using All_facets_iterator"
  << std::endl;
  int all_facets_count = 0;
  Triangulation::All_facets_iterator afacet_it;
  for (afacet_it = tri.all_facets_begin();
       afacet_it != tri.all_facets_end();
       ++afacet_it) {
    std::cout << "Facet " << ++all_facets_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != afacet_it->second) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(afacet_it->first->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << afacet_it->first->vertex(i)->point()
                    << " ";
      }
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;

  std::cout
  << "Printing the points for each facet using Finite_facets_iterator"
  << std::endl;
  int finite_facets_count = 0;
```

```
Triangulation::Finite_facets_iterator ffacet_it;
for (ffacet_it = tri.finite_facets_begin();
     ffacet_it != tri.finite_facets_end();
     ++ffacet_it) {
  std::cout << "Facet " << ++finite_facets_count << ": ";
  for (int i = 0; i <= tri.dimension(); ++i) {
    if (i != ffacet_it->second) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(ffacet_it->first->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << ffacet_it->first->vertex(i)->point()
                  << " ";
    }
  }
  std::cout << std::endl;
}
std::cout << std::endl;

std::cout
  << "Printing the points around each vertex using "
  << "Vertex_around_vertex_enumerator" << std::endl;
Triangulation::All_vertices_iterator avertices_it;
for (avertices_it = tri.all_vertices_begin();
     avertices_it != tri.all_vertices_end();
     ++avertices_it) {

  Triangulation::Vertex_around_vertex_enumerator
  vave(avertices_it, tri.dimension());

  Triangulation::Vertex_around_vertex_enumerator
               ::Vertex_container_iterator vit;

  std::cout << "Vertices adjacent to " << avertices_it->point()
            << " are:" << std::endl;

  for (vit = vave.begin(); vit != vave.end(); ++vit) {
    Triangulation::Vertex_handle v(*vit);
    if (tri.is_infinite(v)) {
      std::cout << " INF" << std::endl;
    }
    else {
      std::cout << " " << v->point() << std::endl;
    }
```

```
    }

  }
  std::cout << std::endl;
}



if (tri.dimension() >= 2) {
  std::cout
  << "Printing the points for each face using All_faces_iterator"
  << std::endl;
  int all_faces_count = 0;
  Triangulation::All_faces_iterator aface_it;
  for (aface_it = tri.all_faces_begin();
       aface_it != tri.all_faces_end();
       ++aface_it) {
    std::cout << "Face " << ++all_faces_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != aface_it->second && i != aface_it->third) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(aface_it->first->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << aface_it->first->vertex(i)->point()
                    << " ";
      }
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;

  std::cout
  << "Printing the points for each face using Finite_faces_iterator"
  << std::endl;
  int finite_faces_count = 0;
  Triangulation::Finite_faces_iterator fface_it;
  for (fface_it = tri.finite_faces_begin();
       fface_it != tri.finite_faces_end();
       ++fface_it) {
    std::cout << "Face " << ++finite_faces_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i)
      if (i != fface_it->second && i != fface_it->third) {
        std::cout << "P" << i << ":";
```

```
      if (tri.is_infinite(fface_it->first->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << fface_it->first->vertex(i)->point()
                  << " ";
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}



if (tri.dimension() >= 1) {
  std::cout
  << "Printing the points for each edge using All_edges_iterator"
  << std::endl;
  Triangulation::All_edges_iterator aedge_it;
  int all_edges_count = 0;
  for (aedge_it = tri.all_edges_begin();
       aedge_it != tri.all_edges_end();
       ++aedge_it) {
    std::cout << "Edge "  << ++all_edges_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != aedge_it->second && i != aedge_it->third &&
          i != aedge_it->fourth) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(aedge_it->first->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << aedge_it->first->vertex(i)->point()
                    << " ";
      }
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;

  std::cout
  << "Printing the points for each edge using Finite_edges_iterator"
  << std::endl;
  Triangulation::Finite_edges_iterator fedge_it;
  int finite_edges_count = 0;
  for (fedge_it = tri.finite_edges_begin();
```

```
       fedge_it != tri.finite_edges_end();
       ++fedge_it) {
    std::cout << "Edge "  << ++finite_edges_count << ": ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != fedge_it->second && i != fedge_it->third &&
          i != fedge_it->fourth) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(fedge_it->first->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << fedge_it->first->vertex(i)->point()
                    << " ";
      }
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}



if (tri.dimension() == 4) {
  std::cout
    << "Printing the cells adjacent to each face"
    << std::endl;
  Triangulation::Finite_faces_iterator fface_it;
  for (fface_it = tri.finite_faces_begin();
       fface_it != tri.finite_faces_end();
       ++fface_it) {
    std::cout << "Face: ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != fface_it->second && i != fface_it->third) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(fface_it->first->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << fface_it->first->vertex(i)->point()
                    << " ";
      }
    }
    std::cout << std::endl;

    Triangulation::Cell_around_face_circulator cc;
    cc = tri.incident_cells(fface_it->first,
```

```
                          fface_it->second, fface_it->third);
  Triangulation::Cell_handle start(cc);

  int k = 0;
  do {
    std::cout << "Cell " << ++k << ": ";
    Triangulation::Cell_handle cur(cc);
    for (int i = 0; i <= tri.dimension(); ++i) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(cur->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << cur->vertex(i)->point() << " ";
    }
    std::cout << std::endl;
    ++cc;
  } while (Triangulation::Cell_handle(cc) != start);

  std::cout << "Going around the other way..." << std::endl;
  start = cc = tri.incident_cells(fface_it->first,
                                  fface_it->third,
                                  fface_it->second);
  k = 0;
  do {
    std::cout << "Cell " << ++k << ": ";
    Triangulation::Cell_handle cur(cc);
    for (int i = 0; i <= tri.dimension(); ++i) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(cur->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << cur->vertex(i)->point() << " ";
    }
    std::cout << std::endl;
    ++cc;
  } while (Triangulation::Cell_handle(cc) != start);
}
std::cout << std::endl;

std::cout
  << "Printing the cells adjacent to each edge"
  << std::endl;
Triangulation::Finite_edges_iterator fedge_it;
for (fedge_it = tri.finite_edges_begin();
```

```
     fedge_it != tri.finite_edges_end();
     ++fedge_it) {
  std::cout << "Edge: ";
  for (int i = 0; i <= tri.dimension(); ++i) {
    if (i != fedge_it->second && i != fedge_it->third &&
        i != fedge_it->fourth) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(fedge_it->first->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << fedge_it->first->vertex(i)->point()
                  << " ";
    }
  }
  std::cout << std::endl;

  Triangulation::Cell_around_edge_enumerator ce(*fedge_it);
  Triangulation::Cell_around_edge_enumerator
              ::Cell_container_iterator cit;

  int k = 0;
  for (cit = ce.begin();
       cit != ce.end();
       ++cit) {
    std::cout << "Cell " << ++k << ": ";
    Triangulation::Cell_handle cur(*cit);
    for (int i = 0; i <= tri.dimension(); ++i) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(cur->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << cur->vertex(i)->point() << " ";
    }
    std::cout << std::endl;
  }
}
std::cout << std::endl;

std::cout
  << "Printing the cells adjacent to each vertex"
  << std::endl;
Triangulation::All_vertices_iterator avertices_it;
for (avertices_it = tri.all_vertices_begin();
     avertices_it != tri.all_vertices_end();
```

```
       ++avertices_it) {

    std::cout << "Cells adjacent to ertex: "
              << avertices_it->point() << " are" << std::endl;

    Triangulation::Cell_around_vertex_enumerator
    cave(avertices_it);

    Triangulation::Cell_around_vertex_enumerator
                 ::Cell_container_iterator cave_it;

    int k = 0;
    for (cave_it = cave.begin(); cave_it != cave.end();
         ++cave_it) {
      std::cout << "Cell " << ++k << ": ";
      Triangulation::Cell_handle cur(*cave_it);
      for (int i = 0; i <= tri.dimension(); ++i) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(cur->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << cur->vertex(i)->point() << " ";
      }
      std::cout << std::endl;
    }
  }
  std::cout << std::endl;
}



else if (tri.dimension() == 3) {
  std::cout
    << "Printing the facets adjacent to each edge"
    << std::endl;
  Triangulation::Finite_edges_iterator fedge_it;
  for (fedge_it = tri.finite_edges_begin();
       fedge_it != tri.finite_edges_end();
       ++fedge_it) {
    std::cout << "Edge: ";
    for (int i = 0; i <= tri.dimension(); ++i) {
      if (i != fedge_it->second && i != fedge_it->third &&
          i != fedge_it->fourth) {
        std::cout << "P" << i << ":";
```

```
    if (tri.is_infinite(fedge_it->first->vertex(i)))
      std::cout << "INF ";
    else
      std::cout << fedge_it->first->vertex(i)->point()
                << " ";
  }
}
std::cout << std::endl;

Triangulation::Facet_around_edge_circulator fc;
fc = tri.incident_facets(fedge_it->first,
                         fedge_it->second, fedge_it->third);
Triangulation::Cell_handle start(fc);

int k = 0;
do {
  std::cout << "Facet " << ++k << ": ";
  Triangulation::Cell_handle cur(fc);
  for (int i = 0; i <= tri.dimension(); ++i) {
    std::cout << "P" << i << ":";
    if (tri.is_infinite(cur->vertex(i)))
      std::cout << "INF ";
    else
      std::cout << cur->vertex(i)->point() << " ";
  }
  std::cout << std::endl;
  ++fc;
} while (Triangulation::Cell_handle(fc) != start);

std::cout << "Going around the other way..." << std::endl;
start = fc = tri.incident_facets(fedge_it->first,
                                 fedge_it->third,
                                 fedge_it->second);
k = 0;
do {
  std::cout << "Facet " << ++k << ": ";
  Triangulation::Cell_handle cur(fc);
  for (int i = 0; i <= tri.dimension(); ++i) {
    std::cout << "P" << i << ":";
    if (tri.is_infinite(cur->vertex(i)))
      std::cout << "INF ";
    else
      std::cout << cur->vertex(i)->point() << " ";
  }
```

```
      std::cout << std::endl;
      ++fc;
    } while (Triangulation::Cell_handle(fc) != start);
  }
  std::cout << std::endl;

  std::cout
    << "Printing the facets adjacent to each vertex"
    << std::endl;
  Triangulation::All_vertices_iterator avertices_it;
  for (avertices_it = tri.all_vertices_begin();
       avertices_it != tri.all_vertices_end();
       ++avertices_it) {

    std::cout << "Facets adjacent to vertex: "
              << avertices_it->point() << " are" << std::endl;

    Triangulation::Facet_around_vertex_enumerator
    fave(avertices_it);

    Triangulation::Facet_around_vertex_enumerator
                ::Cell_container_iterator fave_it;

    int k = 0;
    for (fave_it = fave.begin(); fave_it != fave.end();
         ++fave_it) {
      std::cout << "Cell " << ++k << ": ";
      Triangulation::Cell_handle cur(*fave_it);
      for (int i = 0; i <= tri.dimension(); ++i) {
        std::cout << "P" << i << ":";
        if (tri.is_infinite(cur->vertex(i)))
          std::cout << "INF ";
        else
          std::cout << cur->vertex(i)->point() << " ";
      }
      std::cout << std::endl;
    }
  }
  std::cout << std::endl;
}



else if (tri.dimension() == 2) {
```

```
std::cout
  << "Printing the faces adjacent to each vertex"
  << std::endl;
Triangulation::Finite_vertices_iterator fvertex_it;
for (fvertex_it = tri.finite_vertices_begin();
     fvertex_it != tri.finite_vertices_end();
     ++fvertex_it) {
  Triangulation::Vertex_handle v(fvertex_it);

  std::cout << "Point: " << v->point() << std::endl;

  Triangulation::Face_around_vertex_circulator vc;
  vc = tri.incident_faces(v);
  Triangulation::Cell_handle start(vc);

  int k = 0;
  do {
    std::cout << "Face " << ++k << ": ";
    Triangulation::Cell_handle cur(vc);
    for (int i = 0; i <= tri.dimension(); ++i) {
      std::cout << "P" << i << ":";
      if (tri.is_infinite(cur->vertex(i)))
        std::cout << "INF ";
      else
        std::cout << cur->vertex(i)->point() << " ";
    }
    std::cout << std::endl;
    ++vc;
  } while (Triangulation::Cell_handle(vc) != start);
}
std::cout << std::endl;

std::cout
  << "Printing the vertices adjacent to each vertex"
  << std::endl;
for (fvertex_it = tri.finite_vertices_begin();
     fvertex_it != tri.finite_vertices_end();
     ++fvertex_it) {
  Triangulation::Vertex_handle v(fvertex_it);

  std::cout << "Point: " << v->point() << std::endl;

  Triangulation::Vertex_around_vertex_circulator vavc(v);
  Triangulation::Vertex_handle start(vavc);
```

```
      int k = 0;
      do {

        std::cout << "Vertex " << ++k << ": ";

        Triangulation::Vertex_handle v(vavc);
        if (tri.is_infinite(v))
          std::cout << "INF";
        else
          std::cout << v->point();

        std::cout << std::endl;

        ++vavc;
      } while (Triangulation::Vertex_handle(vavc) != start);
    }
    std::cout << std::endl;
  }



  assert(tri.is_valid(true));

  return 0;
}
```

BIBLIOGRAPHY

[1] O. Aichholzer, F. Aurenhammer, D. Z. Chen, D. T. Lee, and E. Papadopoulou. Skew Voronoi diagrams. *International Journal of Computational Geometry and Applications*, 9(3):235–247, June 1999.

[2] O. Aichholzer, D. Z. Chen, D. T. Lee, A. Mukhopadhyay, E. Papadopoulou, and F. Aurenhammer. Voronoi diagrams for direction-sensitive distances. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 418–420. ACM Press, 1997.

[3] M. A. Armstrong. *Basic Topology.* Springer-Verlag, first edition, 1983.

[4] F. Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM Journal on Computing*, 16(1):78–96, 1987.

[5] F. Aurenhammer. Improved algorithms for discs and balls using power diagrams. *Journal of Algorithms*, 9(2):151–161, 1988.

[6] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[7] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and G. Urrutia, editors, *Handbook of Computational Geometry, Chapter V*, pages 201–290. Elsevier Science Publishing, 2000. [SFB Report F003-092, TU Graz, Austria, 1996].

[8] J.-D. Boissonnat, F. Cazals, F. Da, O. Devillers, S. Pion, F. Rebufat, M. Teillaud, and M. Yvinec. Programming with CGAL: the example of triangulations. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 421–422. ACM Press, 1999.

[9] J.-D. Boissonnat and C. Delage. Convex hull and Voronoi diagram of additively weighted points. In *Proceedings of the Thirteenth Annual European Symposium on Algorithms*, pages 367–378, 2005.

[10] J.-D. Boissonnat, O. Devillers, M. Teillaud, and M. Yvinec. Triangulations in CGAL (extended abstract). In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry*, pages 11–18. ACM Press, 2000.

[11] J.-D. Boissonnat and M. I. Karavelas. On the combinatorial complexity of Euclidean Voronoi cells and convex hulls of d-dimensional spheres. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 305–312. Society for Industrial and Applied Mathematics, 2003.

[12] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, first edition, 1998.

[13] The CGAL homepage. `http://www.cgal.org/`.

[14] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete and Computational Geometry*, 10:377–409, 1993.

[15] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[16] O. Devillers. On deletion in Delaunay triangulations. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 181–188. ACM Press, 1999.

[17] O. Devillers and M. Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 313–319. Society for Industrial and Applied Mathematics, 2003.

[18] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[19] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 43–52. ACM Press, 1992.

[20] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, March 1996.

[21] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Software-Practice and Experience*, 30(11):1167–1202, 2000.

[22] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 313–322. ACM Press, 1986.

[23] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[24] S. J. Fortune. Voronoi diagrams and Delaunay triangulations. *Euclidean Geometry and Computers*, pages 193–233, 1992. World Scientific Publishing Co., D.A. Du, F.K. Hwang, eds.

[25] M. L. Gavrilova and J. G. Rokne. Updating the topology of the dynamic Voronoi diagram for spheres in Euclidean d-dimensional space. *Computer Aided Geometric Design*, 20(4):231–242, 2003.

[26] A. Goede, R. Preissner, and C. Frömmel. Voronoi cell: New method for allocation of space among atoms: elimination of avoidable erros in calculation of atomic volume and density. *Journal of Computational Chemistry*, 18(9):1113–1123, 1997.

[27] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[28] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[29] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proceedings of the Fifth International Workshop on Algorithm Engineering*, pages 79–90, London, UK, 2001. Springer-Verlag.

[30] H. Imai, M. Iri, and K. Murota. Voronoi diagram in the Laguerre geometry and its applications. *SIAM Journal on Computing*, 14(1):93–105, February 1985.

[31] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proceedings of the Twelfth Annual European Symposium on Algorithms*, pages 702–713, 2004.

[32] C. L. Lawson. Properties of n-dimensional triangulations. *Computer Aided Geometric Design*, 3(4):231–246, December 1986.

[33] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, first edition, 1998.

[34] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations : Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, second edition, 2000.

[35] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 1998.

[36] V. T. Rajan. Optimality of the Delaunay triangulation in $\mathbb{R}^d$. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 357–363. ACM Press, 1991.

[37] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the Sixteenth IEEE Symposium on Foundations of Computer Science*, pages 151–162, 1975.

[38] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation - an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, May 2000.

[39] H.-M. Will. *Computation of Additively Weighted Voronoi Cells for Applications in Molecular Biology*. PhD thesis, Swiss Federal Institute of Technology, 1999.

[40] C.-K. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1-2):3–23, 1997.