Lecture Notes on
Geometric Robustness

Jonathan Richard Shewchuk
jrs@cs.berkeley.edu

April 15, 2013

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Supported in part by the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship, in part by the National Science Foundation under Awards CMS-9318163, ACI-9875170, CMS-9980063, CCR-0204377, and CCF-0430065, in part by an Alfred P. Sloan Research Fellowship, and in part by a gift from the Okawa Foundation. The claims in this document are those of the author. They are not endorsed by the sponsors or the U. S. Government.

Keywords: computational geometry, geometric robustness, geometric predicates, geometric constructors, geometric primitives, orientation test, incircle test

Contents

1 Geometric Robustness

Most geometric algorithms are not designed for numerical robustness; they are based on the *real RAM model*, in which quantities are allowed to be arbitrary real numbers, and all arithmetic is exact. There are several ways a geometric algorithm that is correct within the real RAM model can go wrong in an encounter with a modern microprocessor, in which floating-point operations are subject to roundoff error. The output might be incorrect, but be correct for some perturbation of the input. The result might be usable yet not be valid for any imaginable input. Or, the program may simply crash or fail to produce a result. I occasionally hear of implementations where more than half the developers' time is spent solving problems of roundoff error. For surveys of geometric robustness, see Fortune [9] and Hoffmann [15].

There are two types of geometric calculations, or *primitives*, found in geometric algorithms: *predicates*, which make a two-way or three-way decision (does this point lie to the left of, to the right of, or on this line?), and *constructors*, which create a new geometric object (what is the intersection point of these two lines?). Predicates usually make decisions based on the sign of an arithmetic expression—often a matrix determinant. Some geometric algorithms produce output that is purely combinatorial, such as a convex hull or an arrangement of hyperplanes, and rely exclusively on predicates. For example, a Delaunay triangulation can be computed using only orientation and incircle tests (which are discussed in the next section). Algorithms that use constructors are sometimes more difficult to make robust, because the objects they construct may take much greater numerical precision to represent than the input.

Geometric algorithms may be divided into several classes with varying amounts of robustness: *exact algorithms*, which are always correct; *robust algorithms*, which are always correct for some perturbation of the input; *stable algorithms*, for which the perturbation is small; *quasi-robust algorithms*, whose results might be geometrically inconsistent, but nevertheless satisfy some weakened consistency criterion; and *fragile algorithms*, which are not guaranteed to produce any usable output at all. The next several pages are devoted to a discussion of representative research in each class, and of the circumstances in which exact arithmetic and other techniques are or are not applicable. For more extensive surveys of geometric robustness, see Fortune [10] and Hoffmann [15].

Exact algorithms. A geometric algorithm is *exact* if it is guaranteed to produce a correct result when given an exact input. (Of course, the input to a geometric algorithm may only be an approximation of some real-world configuration, but this difficulty is ignored here.) Exact algorithms use exact arithmetic in some form, whether in the form of a multiprecision library or in a more disguised form.

There are several exact arithmetic schemes designed specifically for computational geometry; most are methods for exactly evaluating the sign of a determinant, and hence can be used to perform the orientation and incircle tests. These schemes are nearly always combined with a *floating-point filter*: each expression is first evaluated with ordinary floating-point arithmetic (roundoff error and all), and forward error analysis is used to compute an upper bound on the error of the expression. If the error bound is smaller than the magnitude of the expression, then the sign of the expression is known to be correct. Exact arithmetic is used only if the error bound is too large.

Clarkson [5] proposes an algorithm for using floating-point arithmetic to evaluate the sign of the determinant of a small matrix of integers. A variant of the modified Gram-Schmidt procedure is used to improve the conditioning of the matrix, so that the determinant can subsequently be evaluated safely by Gaussian elimination. The 53 bits of significand available in IEEE double precision numbers are sufficient to operate on 10×10 matrices of 32-bit integers. Clarkson's algorithm is naturally adaptive; its running time is small for matrices whose determinants are not near zero.

Avnaim, Boissonnat, Devillers, Preparata, and Yvinec [1] proposed an algorithm to evaluate signs of determinants of 2×2 and 3×3 matrices of p-bit integers using only p and $(p+1)$-bit arithmetic, respectively. Surprisingly, this is sufficient even to implement the insphere test (which is normally written as a 4×4 or 5×5 determinant), but with a handicap in bit complexity: 53-bit double precision arithmetic is sufficient to correctly perform the insphere test on points having 24-bit integer coordinates.

Fortune and Van Wyk [13, 12] propose a more general approach (not specific to determinants, or even to predicates) that represents integers using a standard extended precision technique, with digits of radix 2^{23} stored as double precision floating-point values. Rather than use a general-purpose arbitrary precision library, they have developed LN, an expression compiler that writes code to evaluate a specific expression exactly. The size of the operands is arbitrary, but is fixed when LN is run; an expression can be used to generate several functions, each for arguments of different bit lengths. Because the expression and the bit lengths of all operands are fixed in advance, LN can tune the exact arithmetic aggressively, eliminating loops, function calls, and memory management. The running time of a function produced by LN depends on the bit complexity of the inputs. Fortune and Van Wyk report an order-of-magnitude speed improvement over the use of multiprecision libraries (for equal bit complexity). Furthermore, LN gains another speed improvement by installing floating-point filters wherever appropriate, calculating error bounds automatically.

Karasick, Lieber, and Nackman [16] report their experiences optimizing a method for determinant evaluation using rational inputs. Their approach reduces the bit complexity of the inputs by performing arithmetic on intervals (with low precision bounds) rather than exact values. The determinant thus evaluated is also an interval; if it contains zero, the precision is increased and the determinant reevaluated. The procedure is repeated until the interval does not contain zero (or contains only zero), and its sign is certain. Their approach is thus somewhat adaptive, but it does not appear to use the results of one iteration to speed the next.

None of the methods mentioned so far is suitable if an algorithm must use floating-point coordinates (and cannot scale the inputs to be integers). Elsewhere [21], I propose a method for writing robust floating-point predicates that relies on new algorithms for arbitrary precision arithmetic and a technique for adaptively computing the value of a polynomial expression. As with Clarkson's method, the amount of time required for the computation depends on how close the value of the expression is to zero. The inputs may be ordinary single or double precision (24- or 53-bit significand) IEEE floating-point numbers.

The Clarkson and Avnaim et al. algorithms are effectively restricted to low precision integer coordinates. Floating-point inputs are more difficult to work with than integer inputs, partly because of the potential for the bit complexity of intermediate values to grow more quickly. The Karasick et al. algorithm also suffers this difficulty, and is too slow to be competitive with the other techniques discussed here, although it may be the best existing alternative for algorithms that require rational numbers, such as those computing exact line intersections.

Exact expression evaluation algorithms—especially those that accommodate inputs with only a limited bit complexity—do not satisfy the needs of all applications. A program that computes line intersections requires rational arithmetic; an exact numerator and exact denominator must be stored. If the intersections may themselves become endpoints of lines that generate more intersections, then intersections of greater and greater bit complexity may be generated. Even exact rational arithmetic is not sufficient for all applications. A solid modeler, for instance, might need to determine the vertices of the intersection of two independent solids that have been rotated through arbitrary angles. Yet exact floating-point arithmetic can't even cope with rotating a square 45° in the plane, because irrational vertex coordinates result. The problem of constructed irrational values has been partly attacked by the implementation of "real" numbers in the LEDA library of algorithms [3]. Values derived from square roots (and other arithmetic operations) are stored in symbolic form when necessary. Comparisons with such numbers are resolved with great numerical care, albeit sometimes at great cost; separation bounds are computed where necessary to ensure that the sign of an expression is determined accurately. Floating-point filters and another form of adaptivity (approximating a result repeatedly, doubling the precision each time) are used as well.

For the remainder of this discussion, I consider only algorithms that use only predicates, and not constructors.

Robust algorithms. There are algorithms that can be made correct with straightforward implementations of exact arithmetic, but suffer an unacceptable loss of speed. An alternative is to relax the requirement of a correct solution, and instead accept a solution that is "close enough" in some sense that depends upon the application. Without exact arithmetic, an algorithm must somehow find a way to produce sensible output despite the fact that geometric tests occasionally tell it lies. No general techniques have emerged yet, although bandages have appeared for specific algorithms, usually ensuring robustness or quasi-robustness through painstaking design and error analysis. The lack of generality of these techniques is not the only limitation of the relaxed approach to robustness; there is a more fundamental difficulty that deserves careful discussion.

When disaster strikes and a real RAM-correct algorithm implemented in floating-point arithmetic fails to produce a meaningful result, it is often because the algorithm has performed tests whose results are mutually contradictory. Figure 1 shows an error that arose in a two-dimensional Delaunay triangulation program I wrote. The program, which employs a divide-and-conquer algorithm presented by Guibas and Stolfi [14], failed in a subroutine that merges two triangulations into one. The geometrically nonsensical triangulation in the illustration was produced.

On close inspection with a debugger, I found that the failure was caused by a single incorrect result of the incircle test. At the bottom of Figure 1 appear four nearly collinear points whose deviation from collinearity has been greatly exaggerated for clarity. The points a, b, c, and d had been sorted by their x-coordinates, and b had been correctly established (by orientation tests) to lie below the line ac and above the line ad. In principle, a program could deduce from these facts that a cannot fall inside the circle dcb. Unfortunately, the incircle test incorrectly declared that a lay inside, thereby leading to the invalid triangulation.

It is significant that the incircle test was not just wrong about these particular points; it was inconsistent with the "known combinatorial facts." A correct algorithm (that computes a purely combinatorial result) will produce a meaningful result if its test results are wrong but are consistent with each other, because there exists an input for which those test results are correct. Following Fortune [8], an algorithm is *robust* if it always produces the correct output under the real RAM model, and under approximate arithmetic always produces an output that is consistent with some hypothetical input that is a perturbation of the true input; it is *stable* if this perturbation is small. Typically, bounds on the perturbation are proven by backward error analysis. Using only approximate arithmetic, Fortune gives an algorithm that computes a planar convex hull that is correct for points that have been perturbed by a relative error of at most $\mathcal{O}(\epsilon)$ (where ϵ is the *machine epsilon* of the floating-point unit), and an algorithm that maintains a triangulation that can be made planar by perturbing each vertex by a relative error of at most $\mathcal{O}(n^2\epsilon)$, where n is the number of vertices. If it seems surprising that a "stable" algorithm cannot keep a triangulation planar, consider the problem of inserting a new vertex so close to an existing edge that it is difficult to discern which side of the edge the vertex falls on. Only exact arithmetic can prevent the possibility of creating an "inverted" triangle.

One might wonder if my triangulation program can be made robust by avoiding any test whose result can be inferred from previous tests. Fortune [8] explains that

> [a]n algorithm is *parsimonious* if it never performs a test whose outcome has already been determined as the formal consequence of previous tests. A parsimonious algorithm is clearly robust,

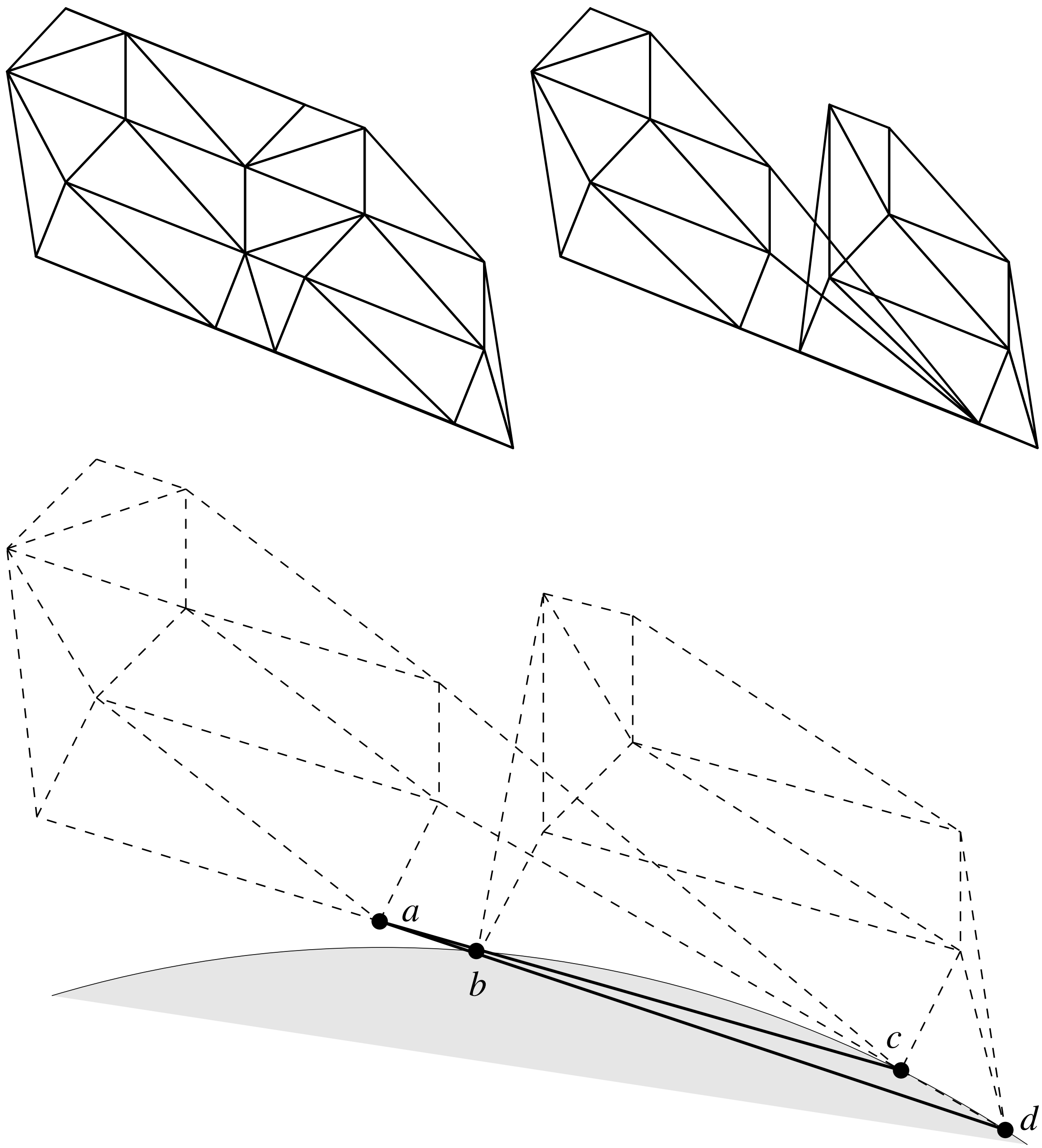

Figure 1: Top left: A Delaunay triangulation. Top right: An invalid triangulation created due to roundoff error. Bottom: Exaggerated view of the inconsistencies that led to the problem. The algorithm "knew" that the point b lay between the lines ac and ad, but an incorrect incircle test claimed that a lay inside the circle dcb.

> since any path through the algorithm must correspond to some geometric input; making an algorithm parsimonious is the most obvious way of making it robust. In principle it is possible to make an algorithm parsimonious: since all primitive tests are polynomial sign evaluations, the question of whether the current test is a logical consequence of previous tests can be phrased as a statement of the existential theory of the reals. This theory is at least NP-hard and is decidable in polynomial space [4]. Unfortunately, the full power of the theory seems to be necessary for some problems. An example is the *line arrangement problem*: given a set of lines (specified by real coordinates (a, b, c), so that $ax + by = c$), compute the combinatorial structure of the resulting arrangement in the plane. It follows from recent work of Mnev [19] that the problem of deciding whether a combinatorial arrangement is actually realizable with lines is as hard as the existential theory of the reals. Hence a parsimonious algorithm for the line arrangement problem ... seems to require the solution of NP-hard problems.

Because exact arithmetic does not require the solution of NP-hard problems, an intermediate course is possible: one could employ parsimony whenever it is efficient to do so, and resort to exact arithmetic otherwise. Consistency is guaranteed if exact tests are used to bootstrap the "parsimony engine." I am not aware of any algorithms in the literature that take this approach, although geometric algorithms are often designed by their authors to avoid the more obviously redundant tests.

Quasi-robust algorithms. The difficulty of determining whether a line arrangement is realizable suggests that, without exact arithmetic, robustness as defined above may be an unattainable goal. However, sometimes one can settle for an algorithm whose output might not be realizable. I place such algorithms in a bag labeled with the fuzzy term *quasi-robust*, which I apply to any algorithm whose output is somehow provably distinguishable from nonsense. Milenkovic [18] circumvents the aforementioned NP-hardness result while using approximate arithmetic by constructing pseudo-line arrangements; a *pseudo-line* is a curve constrained to lie very close to an actual line. Fortune [11] presents a 2D Delaunay triangulation algorithm that constructs, using approximate arithmetic, a triangulation that is nearly Delaunay in a well-defined sense using the pseudo-line-like notion of pseudocircles. Unfortunately, the algorithm's running time is $\mathcal{O}(n^2)$, which compares poorly with the $\mathcal{O}(n \log n)$ time of optimal algorithms. Milenkovic's and Fortune's algorithms are both *quasi-stable*, having small error bounds. Milenkovic's algorithm can be thought of as a quasi-robust algorithm for line arrangements, or as a robust algorithm for pseudo-line arrangements.

Barber [2] pioneered an approach in which uncertainty, including the imprecision of input data, is a part of each geometric entity. *Boxes* are structures that specify the location and the uncertainty in location of a vertex, edge, facet, or other geometric structure. Boxes may arise either as input or as algorithmic constructions; any uncertainty resulting from roundoff error is incorporated into their shapes and sizes. Barber presents algorithms for solving the point-in-polygon problem and for constructing convex hulls in any dimension. For the point-in-polygon problem, "can't tell" is a valid answer if the uncertainty inherent in the input or introduced by roundoff error prevents a sure determination. The salient feature of Barber's Quickhull convex hull algorithm is that it merges hull facets that cannot be guaranteed (through error analysis) to be clearly locally convex. The *box complex* produced by the algorithm is guaranteed to contain the true convex hull, bounding it, if possible, both from within and without.

The degree of robustness required of an algorithm is typically determined by how its output is used. For instance, many point location algorithms can fail when given a non-planar triangulation. For this very reason, my triangulator crashed after producing the flawed triangulation in Figure 1.

The reader should take three lessons from this discussion. First, problems due to roundoff can be severe and difficult to solve. Second, even if the inputs are imprecise and the user isn't picky about the accuracy of the output, internal consistency may still be necessary if any output is to be produced at all; exact arithmetic may be required even when exact results aren't. Attempts to solve problems by "tolerancing" (e.g., treating nearly-collinear points as if they were collinear) are not generally effective, because the use of tolerances does not restore internal consistency. Third, neither exact arithmetic nor clever handling of tests that tell falsehoods is a universal balm. However, exact arithmetic is attractive when it is applicable, because it can be plugged into a geometric algorithm with little effort.

2 Geometric Computations with Less Error

Here, I discuss how to write geometric primitives that incur as little error as possible if exact arithmetic is not available or is too slow. The main principle is this: many geometric computations involve geometric entities whose absolute coordinates (distance from the origin) are much greater than their relative coordinates (distance from each other). Furthermore, many common geometric calculations are translation-invariant: if we move the axes of the coordinate system, the answer is the same. Hence, if the operands of a geometric predicate or constructor are translated so that one of the relevant points lies at the origin, numerical precision is freed, and the result is more accurate.

Many geometric expressions found in the literature are already written as a function of differences of points, and these typically perform well. As a bad example, though, consider a well-known expression for the area of a polygon. If $p_1, p_2, \ldots, p_n$ are the vertices of a polygon listed in order around its boundary, let $p_{i,x}$ and $p_{i,y}$ be the x- and y-coordinates of p_i, and let $p_{n+1} = p_1$. The area of the polygon is

$$\frac{1}{2}\left|\sum_{i=1}^{n}(p_{i,x}p_{i+1,y} - p_{i,y}p_{i+1,x})\right|.$$

If the polygon's dimensions are small compared to its distance from the origin, this formula can give a terribly inaccurate result. To derive a better formula, translate p_n to the origin by replacing each p_i with p'_i, then substituting $p'_i = p_i - p_n$. The improved formula is

$$\frac{1}{2}\left|\sum_{i=1}^{n-2}\left[(p_{i,x} - p_{n,x})(p_{i+1,y} - p_{n,y}) - (p_{i,y} - p_{n,y})(p_{i+1,x} - p_{n,x})\right]\right|.$$

One can use forward error analysis [22] to derive estimates of the error (caused by floating-point roundoff) after computing each of these two expressions. The analysis reveals that the error of the first expression is a function of the sizes of the coordinates, whereas the error of the second expression is a function only of the distances between the points. If these distances are notably smaller than the absolute coordinates, the second expression is much more accurate. The products in the second expression benefit from the extra precision freed by the translation of points to near the origin. Unfortunately, the second formula uses more floating-point operations. (With a little clever coding, each difference between points can be computed once and used twice.) There is often a trade-off between numerical accuracy and speed.

It is straightforward to show that the two expressions above are equivalent, but if we believe the first expression is correct, it suffices that we understand that translating a polygon does not change its area. However, there are geometric calculations that are not translation-invariant.

The translation of points can often be done without roundoff error. Figure 2 demonstrates a toy problem: suppose we compute the area of each triangle in a triangulation (using the second formula above). An interesting property of most floating-point systems (including the IEEE standard) is that if two numbers have the same sign and differ by at most a factor of two, then their difference is computed with no roundoff

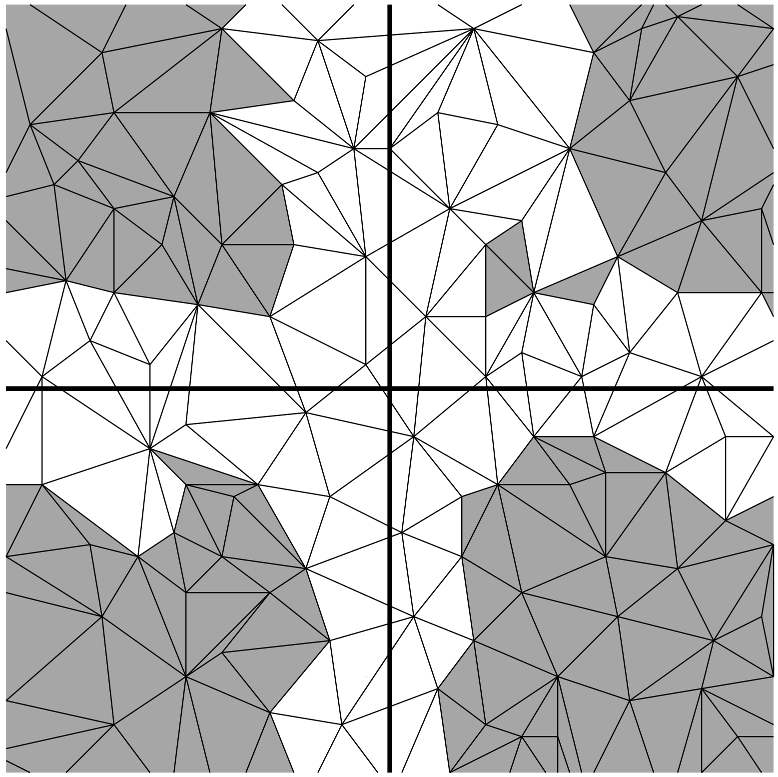

Figure 2: Shaded triangles can be translated to the origin without incurring roundoff error. In most triangulations, such triangles are the common case.

error. Hence, in Figure 2, any shaded triangle can be translated so that one of its vertices lies at the origin without roundoff error. The white triangles may or may not suffer from roundoff during such translation. In a large triangulation, only a small proportion of the triangles (those near a coordinate axis) will suffer roundoff during the translation.

Two other important examples of these ideas are the orientation predicate, used in most geometric codes, and the incircle predicate, used to construct Delaunay triangulations. Let a, b, c, and d be four points in the plane. Define a procedure ORIENT2D(a, b, c) that returns a positive value if the points a, b, and c are arranged in counterclockwise order, a negative value if the points are in clockwise order, and zero if the points are collinear. A more common (but less symmetric) interpretation is that ORIENT2D returns a positive value if a lies to the left of the directed line bc; for this purpose the orientation test is used by many geometric algorithms.

Define also a procedure INCIRCLE(a, b, c, d) that returns a positive value if d lies inside the oriented circle abc. By *oriented circle*, I mean the unique (and possibly degenerate) circle through a, b, and c, with these points occurring in counterclockwise order about the circle. (If these points occur in clockwise order, INCIRCLE will reverse the sign of its output, as if the circle's exterior were its interior.) INCIRCLE returns zero if and only if all four points lie on a common circle or line. Both ORIENT2D and INCIRCLE have the symmetry property that interchanging any two of their parameters reverses the sign of their result.

These definitions extend to arbitrary dimensions. To generalize the orientation test to dimensionality d, let $u_1, u_2, \ldots, u_d$ be the unit vectors. ORIENT is defined so that ORIENT$(u_1, u_2, \ldots, u_d, 0) = 1$.

In any dimension, the orientation and incircle tests may be implemented as matrix determinants. For two dimensions:

$$\textsc{Orient2D}(a,b,c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \quad (1)$$

$$= \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \quad (2)$$

$$\textsc{InCircle}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} \quad (3)$$

$$= \begin{vmatrix} a_x - d_x & a_y - d_y & (a_x - d_x)^2 + (a_y - d_y)^2 \\ b_x - d_x & b_y - d_y & (b_x - d_x)^2 + (b_y - d_y)^2 \\ c_x - d_x & c_y - d_y & (c_x - d_x)^2 + (c_y - d_y)^2 \end{vmatrix} \quad (4)$$

These formulae generalize to other dimensions straightforwardly. See Sections 3.1 and 3.2 for the three-dimensional versions.

Expressions (1) and (2) can be shown to be equivalent by simple algebraic transformations, as can Expressions (3) and (4) with a little more effort. Expression (2) also follows from Expression (1) by translating each point by $-c$, and Expression (4) follows from Expression (3) by translating each point by $-d$.

Although Expressions (1) and (3) are faster to compute, Expressions (2) and (4) tend to have much smaller errors, and are strongly preferred if floating-point computation with roundoff is used.

Once a determinant has been chosen for evaluation, there are several methods to evaluate it. A number of methods are surveyed by Fortune and Van Wyk [12], and only their conclusion is repeated here. The cheapest method of evaluating the determinant of a 5×5 or smaller matrix seems to be by dynamic programming applied to cofactor expansion. Let d be the dimensionality of the matrix. Evaluate the $\binom{d}{2}$ determinants of all 2×2 minors of the first two columns, then the $\binom{d}{3}$ determinants of all 3×3 minors of the first three columns, and so on.

The principles governing numerical accuracy apply to constructors as well as predicates. The main difference is that the final result must be translated back to its correct location. For instance, the following expressions compute the center O of a circle that passes through the three points a, b, and c.

$$O_x = c_x + \frac{\begin{vmatrix} (a_x - c_x)^2 + (a_y - c_y)^2 & a_y - c_y \\ (b_x - c_x)^2 + (b_y - c_y)^2 & b_y - c_y \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}, O_y = c_y + \frac{\begin{vmatrix} a_x - c_x & (a_x - c_x)^2 + (a_y - c_y)^2 \\ b_x - c_x & (b_x - c_x)^2 + (b_y - c_y)^2 \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}.$$

In both of these expressions, all arithmetic operations incur an error which is a function of the relative coordinates, until the final addition. Only this one final operation incurs an error proportional to the absolute coordinates.

Another advantage of these circumcenter expressions is that they are only unstable in cases where instability is unavoidable. Compare the common technique of computing the circumcenter by finding the intersections of the bisectors of two sides of the triangle $\triangle abc$. If $\triangle abc$ is an isosceles triangle with angles of 2°, 89°, and 89°, and the bisectors of its two nearly-parallel sides are used, the intersection computation

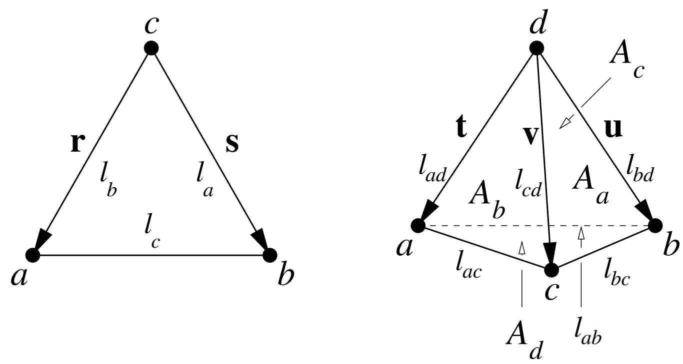

Figure 3: A triangle and a tetrahedron, both having positive orientation.

is unstable. On the other hand, if the bisector of the short side is used, the intersection computation is stable. A programmer might try to determine which two sides of the triangle will yield the most stable computation, but the expressions above achieve the same effect without any such decision. Unfortunately, finding expressions like these is a bit of an art. These expressions were found by expressing the circumcenter as the solution of a system of linear equations, then applying Cramer's Rule from linear algebra.

The denominator of the fractions above is precisely the expression computed by ORIENT2D. The computation of p is unstable if a, b, and c are nearly collinear (i.e. the triangle $\triangle abc$ has an angle very close to $180°$). In this case, the denominator is close to zero, and roundoff error in the denominator can dramatically change the result, or cause a division by zero. This is an instability that is unavoidable unless exact arithmetic is used to compute the denominator. (The accuracy of the numerator is less critical.)

3 Some (Relatively) Accurate Formulae

In these formulae, the norm $|\cdot|$ denotes the Euclidean length of a vector, and the operator $\times$ denotes the vector cross product. Formulae for triangles (in two or three dimensions) govern a triangle with vertices a, b, and c, and employ the vectors $\mathbf{r} = a - c$ and $\mathbf{s} = b - c$. Formulae for tetrahedra govern a tetrahedron with vertices a, b, c, and d, and employ the vectors $\mathbf{t} = a - d$, $\mathbf{u} = b - d$, and $\mathbf{v} = c - d$. These and other notations are illustrated in Figure 3.

3.1 Orientation of Points, Triangle Area, Tetrahedron Volume, Collinearity, and Coplanarity

The orientation test in d dimensions establishes the orientation of a set of $d + 1$ points. If the points are affinely independent, there are two possible orientations (regardless of the value of d), which are distinguished by the sign of an expression (given below for the two- and three-dimensional cases). If the points are not affinely independent, the expression evaluates to zero. The orientation of a set of points is invariant under rotation of the coordinate system, but is reversed in a mirror image.

Given three points a, b, and c in the plane, the expression ORIENT2D(a, b, c) is the signed area of the parallelogram determined by the vectors $\mathbf{r} = a - c$ and $\mathbf{s} = b - c$. It is positive if the points occur in counterclockwise order, negative if they occur in clockwise order, and zero if they are collinear.

$$\begin{aligned}
\textsc{Orient2D}(a,b,c) &= \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} \\
&= \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix} \\
&= \begin{vmatrix} \mathbf{r} & \mathbf{s} \end{vmatrix}.
\end{aligned}$$

As discussed in Section 2, the second (or equivalently, the third) of these determinants is better for fixed precision floating-point computation, because it loses less accuracy to roundoff error. The first expression is faster if an exact arithmetic library is used. If exact arithmetic is used, I recommend coupling it with a floating-point filter. The filter should use the second expression, and the exact computation should used the first expression. Public domain C code that does this, with additional levels of adaptivity (for greater speed), is freely available from http://www.cs.cmu.edu/∼quake/robust.html.

The signed area of the triangle with vertices a, b, and c is half the area of the parallelogram,

$$A = \frac{\textsc{Orient2D}(a,b,c)}{2}.$$

The area of a triangle in E^3 is proportional to the length of an orthogonal vector found by taking the vector cross product of two sides of the triangle. This cross product can be computed by three applications of ORIENT2D. Given a point a in E^3, let a_{yz} be a two-dimensional point whose coordinates are the y- and z-coordinates of a (in that order). Then

$$(a-c)\times(b-c) = \left(\textsc{Orient2D}(a_{yz}, b_{yz}, c_{yz}), \textsc{Orient2D}(a_{zx}, b_{zx}, c_{zx}), \textsc{Orient2D}(a_{xy}, b_{xy}, c_{xy})\right).$$

It is not accidental that the z-coordinate precedes the x-coordinate in the expression for the y-coordinate of the cross product. If the coordinates are not ordered thus, the sign of the y-coordinate will be wrong.

The unsigned area of a triangular face in E^3 with vertices a, b, and c is

$$\begin{aligned}
A_f &= \frac{|\mathbf{r}\times\mathbf{s}|}{2} \\
&= \frac{|(a-c)\times(b-c)|}{2} \\
&= \frac{\sqrt{\textsc{Orient2D}(a_{yz}, b_{yz}, c_{yz})^2 + \textsc{Orient2D}(a_{zx}, b_{zx}, c_{zx})^2 + \textsc{Orient2D}(a_{xy}, b_{xy}, c_{xy})^2}}{2}.
\end{aligned}$$

If an accurate implementation of ORIENT2D is available, this expression takes advantage of it.

An expression for the unsigned area of the triangle that works in any dimensionality $d \geq 2$ (albeit with less speed and accuracy than the foregoing expression) is

$$A_f = \frac{\sqrt{|\mathbf{r}|^2|\mathbf{s}|^2 - |\mathbf{r}\cdot\mathbf{s}|^2}}{2}.$$

This expression has two related pitfalls. First, floating-point error can cause the computation of $|\mathbf{r}|^2|\mathbf{s}|^2 - |\mathbf{r}\cdot\mathbf{s}|^2$ to produce a negative value; it is usually necessary to explicitly test for this possibility. Second, when this difference is much smaller than $|\mathbf{r}|^2|\mathbf{s}|^2$, the relative accuracy of the result can be very bad.

Given four points a, b, c, and d in E^3, the expression $\textsc{Orient3D}(a,b,c,d)$ is the signed volume of the parallelepiped determined by the vectors $\mathbf{t} = a - d$, $\mathbf{u} = b - d$, and $\mathbf{v} = c - d$. It is positive if the points occur in the orientation illustrated in Figure 3, negative if they occur in the mirror-image orientation, and zero if the four points are coplanar. You can apply a *right-hand rule*: orient your right hand with fingers curled to follow the circular sequence bcd. If your thumb points toward a, ORIENT3D returns a positive value.

$$\begin{aligned}
\textsc{Orient3D}(a,b,c,d) &= \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \\
&= \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \\
&= \begin{vmatrix} \mathbf{t} & \mathbf{u} & \mathbf{v} \end{vmatrix}.
\end{aligned}$$

As with ORIENT2D, the second (or equivalently, the third) of these determinants is more accurate for floating-point computation with roundoff, and the first is faster for computation with an exact arithmetic library.

The signed volume of the tetrahedron with vertices a, b, c, and d is one-sixth the volume of the parallelepiped,

$$V = \frac{\textsc{Orient3D}(a,b,c,d)}{6}.$$

Higher-dimensional ORIENT tests are easy to derive by generalizing ORIENT2D and ORIENT3D in the obvious way. The measure (higher-dimensional analogue to area and volume) of the d-dimensional parallelepiped determined by a set of d vectors (all emanating from a single point) is the determinant of the matrix whose column vectors are the vectors in question. The measure of a d-simplex equals the measure of the parallelepiped divided by $d!$.

ORIENT2D reveals whether three vertices in the plane are collinear. What about the collinearity of three vertices in three or more dimensions? Vertices a, b, and c are collinear if and only if $A_f = 0$. In three dimensions, there are two ways to test if $A_f = 0$: test if $\mathbf{r} \times \mathbf{s} = 0$, or test if $|\mathbf{r}|^2|\mathbf{s}|^2 = |\mathbf{r} \cdot \mathbf{s}|^2$. The first test can take advantage of a robust implementation of the ORIENT2D sign test, if one is available. The second test (only) works in any dimensionality.

3.2 The InCircle and InSphere Predicates

The insphere test in d dimensions establishes whether a vertex lies inside, outside, or on a sphere passing through $d + 1$ other points. These possibilities are distinguished by the sign of an expression (given below for the two- and three-dimensional cases).

Let a, b, c, and d be four points in the plane, and suppose the first three are oriented so that $\textsc{Orient2D}(a,b,c)$ is positive. The expression $\textsc{InCircle}(a,b,c,d)$ is positive if d lies inside the circle passing through a, b, and c; negative if d lies outside the circle; and zero if d lies on the circle. (If $\textsc{Orient2D}(a,b,c)$ is negative, the sign returned by INCIRCLE is reversed.)

$$\textsc{InCircle}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} a_x-d_x & a_y-d_y & (a_x-d_x)^2+(a_y-d_y)^2 \\ b_x-d_x & b_y-d_y & (b_x-d_x)^2+(b_y-d_y)^2 \\ c_x-d_x & c_y-d_y & (c_x-d_x)^2+(c_y-d_y)^2 \end{vmatrix}.$$

The second of these expressions is more accurate for floating-point computation with roundoff, and the first is faster for computation with an exact arithmetic library.

Let a, b, c, d, and e be five points in E^3, and suppose the first four are oriented so that $\textsc{Orient3D}(a,b,c,d)$ is positive. The expression $\textsc{InSphere}(a,b,c,d,e)$ is positive if e lies inside the sphere passing through a, b, c, and d; negative if e lies outside the sphere; and zero if e lies on the sphere. (If $\textsc{Orient3D}(a,b,c,d)$ is negative, the sign returned by $\textsc{InSphere}$ is reversed.)

$$\textsc{InSphere}(a,b,c,d,e) = \begin{vmatrix} a_x & a_y & a_z & a_x^2+a_y^2+a_z^2 & 1 \\ b_x & b_y & b_z & b_x^2+b_y^2+b_z^2 & 1 \\ c_x & c_y & c_z & c_x^2+c_y^2+c_z^2 & 1 \\ d_x & d_y & d_z & d_x^2+d_y^2+d_z^2 & 1 \\ e_x & e_y & e_z & e_x^2+e_y^2+e_z^2 & 1 \end{vmatrix}$$

$$= \begin{vmatrix} a_x-e_x & a_y-e_y & a_z-e_z & (a_x-e_x)^2+(a_y-e_y)^2+(a_z-e_z)^2 \\ b_x-e_x & b_y-e_y & b_z-e_z & (b_x-e_x)^2+(b_y-e_y)^2+(b_z-e_z)^2 \\ c_x-e_x & c_y-e_y & c_z-e_z & (c_x-e_x)^2+(c_y-e_y)^2+(c_z-e_z)^2 \\ d_x-e_x & d_y-e_y & d_z-e_z & (d_x-e_x)^2+(d_y-e_y)^2+(d_z-e_z)^2 \end{vmatrix}.$$

3.3 Angles and Minimum Angles

Let $\mathbf{r}$ and $\mathbf{s}$ be two vectors with a common origin. The angle separating the vectors is θ, where

$$\tan\theta = \frac{2A}{\mathbf{r}\cdot\mathbf{s}}, \quad \cos\theta = \frac{\mathbf{r}\cdot\mathbf{s}}{|\mathbf{r}||\mathbf{s}|}, \quad \text{and} \quad \sin\theta = \frac{2A}{|\mathbf{r}||\mathbf{s}|},$$

and A is the area of the triangle with sides $\mathbf{r}$ and $\mathbf{s}$. These formulae work equally well in two, three, or many dimensions. (In three or more dimensions, replace A with the unsigned A_f.) Only the tangent formula computes θ accurately over its entire range; the sine and cosine formulae are not generally recommended for calculating θ itself, although at least *one* of the two is accurate for any given value of θ.

The quadrant in which θ falls is determined by the signs of A and $\mathbf{r}\cdot\mathbf{s}$ (which is another argument for using the tangent formula, or perhaps the cosine formula when angles are not signed). A positive value of A indicates that θ is in the range $(0°, 180°)$, meaning that $\mathbf{s}$ is counterclockwise from $\mathbf{r}$; whereas a negative value indicates the range $(-180°, 0°)$, meaning that $\mathbf{s}$ is clockwise from $\mathbf{r}$. This distinction between positive and negative angles is only meaningful in two-dimensional space. A positive value of $\mathbf{r}\cdot\mathbf{s}$ indicates that θ is in the range $(-90°, 90°)$, whereas a negative value indicates the range $(90°, 180°)$ or $(-180°, -90°)$.

Again, the tangent formula is an accurate way to compute θ. However, it does require a check for division by zero. If $\mathbf{r}\cdot\mathbf{s} = 0$, θ is $90°$ or $-90°$, according to the sign of A. Otherwise, most computer arctangent functions return an angle in the range $(-90°, 90°)$, and it is necessary to correct the result if $\mathbf{r} \cdot \mathbf{s} < 0$ by adding or subtracting $180°$.

The cosine formula produces an angle θ in the range $[0°, 180°]$. This makes it the simplest formula to use when signed angles are not desired. (If signed angles are desired, an extra computation of A is necessary.) Unfortunately, the value of θ may be quite inaccurate; for θ very near 0° or 180°, the calculated value of θ loses about half its bits of accuracy. The formula is quite accurate for $\theta \in [45°, 135°]$, though. If the vectors $\mathbf{r}$ and $\mathbf{s}$ are known to be nonzero, there is no need to check for division by zero; however, for extreme values of θ, floating-point roundoff sometimes produces a value of $\mathbf{r} \cdot \mathbf{s}/(|\mathbf{r}||\mathbf{s}|)$ slightly greater than 1 or slightly less than -1, which may cause exceptions in arccosine libraries. An explicit test for this possibility is usually necessary.

The sine formula produces an angle θ in the range $[-90°, 90°]$. If θ might be obtuse, an extra test is needed to see if $\mathbf{r} \cdot \mathbf{s} < 0$, in which case the result must be corrected by subtracting it from $180°$ or from $-180°$. For θ near 90° or $-90°$, the calculated value of θ loses about half its bits of accuracy, and the value of $2A/(|\mathbf{r}||\mathbf{s}|)$ computed with roundoff may be slightly greater than 1 or slightly less than -1. The formula is quite accurate for $\theta \in [-45°, 45°]$ (including the range where the cosine formula is unreliable).

To find the minimum angle of a triangle (on the assumption that angles are never interpreted as being negative), there is no need to compute all three angles. Instead, use the fact that the sine of the minimum angle of the triangle is the minimum sine of the angles of the triangle, and by the Law of Sines, the minimum sine is found opposite the shortest edge. Hence, if the edge lengths of the triangle are $\ell_{\min}$, ℓ_{med}, and $\ell_{\max}$ in order from shortest to longest, then

$$\sin\theta_{\min} = \min_i \sin\theta_i = \frac{2|A|}{\ell_{\max}\ell_{\mathrm{med}}}.$$

Because the minimum angle of a triangle is never greater than $60°$, the arcsine operation that computes $\theta_{\min}$ is reasonably accurate.

3.4 Dihedral Angles and Minimum Dihedral Angles

Let c and d be two points in E^3, and suppose that two planar facets meet at the edge cd. Let a be a point lying on one of the facets, and let b be a point lying on the other. What is the dihedral angle separating the two facets? It is helpful to imagine the tetrahedron $abcd$, because the tetrahedron's volume and face areas appear in the formulae. The dihedral angle in question separates the faces $\triangle acd$ and $\triangle bcd$.

Let V be the signed volume of the tetrahedron $abcd$. Let A_a, A_b, A_c, and A_d be the unsigned areas of the faces of the tetrahedron opposite vertices a, b, c, and d, respectively. Let $\mathbf{n}_i$ be an outward-directed vector orthogonal to face i. The vectors $\mathbf{n}_i$ are produced as a by-product of computing A_i, because $A_i = |\mathbf{n}_i|/2$. If the vertices are oriented as depicted in Figure 3, then the outward-facing vectors are

$$\begin{aligned}
\mathbf{n}_a &= (c-d) \times (b-d) = \mathbf{v} \times \mathbf{u}, \\
\mathbf{n}_b &= (a-d) \times (c-d) = \mathbf{t} \times \mathbf{v}, \\
\mathbf{n}_c &= (b-d) \times (a-d) = \mathbf{u} \times \mathbf{t}, \text{ and} \\
\mathbf{n}_d &= (a-c) \times (b-c).
\end{aligned}$$

See Section 3.1 for advice on using an accurate implementation of ORIENT2D to compute these cross products. An alternative formula for the last vector is $\mathbf{n}_d = -\mathbf{n}_a - \mathbf{n}_b - \mathbf{n}_c$, but this loses a few more bits of accuracy.

Let $\langle i, j, k, l \rangle$ be a permutation of the vertices a, b, c, d. Let ℓ_{ij} be the length of the edge connecting vertices i and j, and let θ_{ij} be the dihedral angle at which face $\triangle ijk$ meets face $\triangle ijl$. For example, θ_{ab} is the dihedral angle at edge ab, and $\ell_{ab} = |a - b|$. As with planar angles, there are three formulae by which dihedral angles may be computed,

$$\tan \theta_{ij} = -\frac{6V\ell_{ij}}{\mathbf{n}_k \cdot \mathbf{n}_l}, \quad \cos \theta_{ij} = -\frac{\mathbf{n}_k \cdot \mathbf{n}_l}{4A_k A_l}, \quad \text{and} \quad \sin \theta_{ij} = \frac{3V\ell_{ij}}{2A_k A_l}.$$

As in two dimensions, only the tangent formula computes θ_{ij} accurately over its entire range. The sine and cosine formulae are not generally recommended for calculating θ_{ij} itself, although at least one of the two is accurate for any given value of θ_{ij}.

The quadrant in which θ_{ij} falls is determined by the signs of V and $\mathbf{n}_k \cdot \mathbf{n}_l$. By convention, a dihedral angle θ_{ij} is negative if the signed volume V is negative (and signed angles are desired). A negative value of $\mathbf{n}_k \cdot \mathbf{n}_l$ indicates that θ_{ij} is in the range $(-90^\circ, 90^\circ)$, whereas a positive value indicates the range $(90^\circ, 180^\circ)$ or $(-180^\circ, -90^\circ)$.

The tangent formula is an accurate way to compute θ_{ij}. It requires a check for division by zero. If $\mathbf{n}_k \cdot \mathbf{n}_l = 0$, θ_{ij} is 90° or -90°, according to the sign of V. Otherwise, most computer arctangent functions return an angle in the range $(-90^\circ, 90^\circ)$, and it is necessary to correct the result if $\mathbf{n}_k \cdot \mathbf{n}_l > 0$ by adding or subtracting 180°.

The cosine formula produces an angle θ_{ij} in the range $[0^\circ, 180^\circ]$. This makes it the simplest formula to use when signed angles are not desired. (If signed angles are desired, an extra computation of V is necessary.) Unfortunately, the value of θ_{ij} may be quite inaccurate; for θ_{ij} very near 0° or 180°, the calculated value of θ_{ij} loses about half its bits of accuracy. The formula is quite accurate for $\theta_{ij} \in [45^\circ, 135^\circ]$. If the vectors $\mathbf{n}_k$ and $\mathbf{n}_l$ are known to be nonzero, there is no need to check for division by zero; however, for extreme values of θ_{ij}, floating-point roundoff sometimes produces a value of $\cos \theta_{ij}$ slightly greater than 1 or slightly less than -1, which may cause exceptions in arccosine libraries. An explicit test for this possibility is usually necessary.

The sine formula produces an angle θ_{ij} in the range $[-90^\circ, 90^\circ]$. If θ_{ij} might be obtuse, an extra test is needed to see if $\mathbf{n}_k \cdot \mathbf{n}_l > 0$, in which case the result must be corrected by subtracting it from 180° or from -180°. For θ_{ij} near 90° or -90°, the calculated value of θ_{ij} loses about half its bits of accuracy, and the value of $\sin \theta_{ij}$ computed with roundoff may be slightly greater than 1 or slightly less than -1. The formula is quite accurate for $\theta_{ij} \in [-45^\circ, 45^\circ]$ (including the range where the cosine formula is unreliable).

To find the minimum sine of the dihedral angles of a tetrahedron (sometimes used as a quality measure for mesh smoothing), under the assumption that angles are never interpreted as being negative, compute the value

$$\min_{ij} \sin \theta_{ij} = \frac{3|V|}{2} \min_{i,j,k,l \text{ distinct}} \frac{\ell_{ij}}{A_k A_l}.$$

Unfortunately, unlike with triangles, it is not always true that $\sin \theta_{\min} = \min_{ij} \sin \theta_{ij}$, because the dihedral angle that minimizes the sine might be an obtuse angle. However, the smallest dihedral angle of a tetrahedron is always less than 70.53°, so $\theta_{\min}$ can be found by minimizing over the acute dihedral angles.

$$\tan \theta_{\min} = 6|V| \min_{\substack{\mathbf{n}_k \cdot \mathbf{n}_l < 0 \\ i,j,k,l \text{ distinct}}} \frac{-\ell_{ij}}{\mathbf{n}_k \cdot \mathbf{n}_l} \quad \text{and} \quad \sin \theta_{\min} = \frac{3|V|}{2} \min_{\substack{\mathbf{n}_k \cdot \mathbf{n}_l < 0 \\ i,j,k,l \text{ distinct}}} \frac{\ell_{ij}}{A_k A_l}.$$

Because $\theta_{\min}$ does not exceed 70.53°, the arcsine operation that computes $\theta_{\min}$ is reasonably accurate. The arctangent formula is slightly more accurate, though. (Beware, though, that the computation of $\mathbf{n}_k$, $\mathbf{n}_l$, A_k, and A_l might not be stable.)

3.5 Solid Angles

Let $abcd$ be a non-inverted tetrahedron. Each vertex of the tetrahedron is an endpoint of three edges. The solid angle at any vertex of the tetrahedron is the sum of the three dihedral angles at those three edges minus $180°$, and ranges from $0°$ to $360°$.

However, Eriksson [7] offers a faster and more accurate way to compute a solid angle. Let $\mathbf{t} = a - d$, $\mathbf{u} = b - d$, and $\mathbf{v} = c - d$. The solid angle at d is ϕ, where

$$\tan(\phi/2) = \frac{6|V|}{|\mathbf{t}||\mathbf{u}||\mathbf{v}| + |\mathbf{t}|\mathbf{u} \cdot \mathbf{v} + |\mathbf{u}|\mathbf{v} \cdot \mathbf{t} + |\mathbf{v}|\mathbf{t} \cdot \mathbf{u}}.$$

The arctangent should be computed so that ϕ is in the range $[0°, 360°]$ (and not $[-180°, 180°]$). If the denominator is zero, $\phi = 180°$; be careful to avoid a division by zero or overflow. If the denominator is negative, $\phi > 180°$. For degenerate tetrahedra (where $V = 0$), $\phi = 0°$ if the denominator is positive, and $\phi = 360°$ if the denominator is negative. When both the numerator and denominator are zero, the solid angle is undefined: an arbitrarily small perturbation of any vertex of the tetrahedron could set ϕ to any value.

Liu and Joe [17] provide an alternative formula. Let ϕ_i be the solid angle at vertex i, where i is one of a, b, c, or d. Let ℓ_{ij} be the length of the edge connecting vertices i and j. Then

$$\sin(\phi_i/2) = \frac{12|V|}{\sqrt{\prod_{i \neq j < k \neq i} (\ell_{ij} + \ell_{ik} + \ell_{jk})(\ell_{ij} + \ell_{ik} - \ell_{jk})}}.$$

Unfortunately, this formula does not distinguish between the cases where ϕ_i is less than or greater than $180°$. These cases may be distinguished by checking the sign of the denominator of the formula for $\tan(\phi_i/2)$ above.

Liu and Joe suggest using $\min_i \sin(\phi_i/2)$ as a quality measure for tetrahedra. For this purpose, there is no need for a sign check, and their formula is convenient.

Liu and Joe also show that the smallest solid angle of a tetrahedron never exceeds

$$4 \arcsin \sqrt{(9 - 5\sqrt{3})/18} \doteq 31.59°,$$

and the solid angles of an equilateral tetrahedron have exactly this value.

3.6 Intersection Points

The intersection of a line ab with a line cd in the plane is

$$p = a + \alpha(b - a),$$

where

$$\alpha = \frac{\begin{vmatrix} c_x - a_x & c_y - a_y \\ d_x - a_x & d_y - a_y \end{vmatrix}}{\begin{vmatrix} b_x - a_x & b_y - a_y \\ d_x - c_x & d_y - c_y \end{vmatrix}} = \frac{\textsc{Orient2D}(c, d, a)}{\begin{vmatrix} b_x - a_x & b_y - a_y \\ d_x - c_x & d_y - c_y \end{vmatrix}}.$$

The denominator is exactly zero if and only if ab is parallel to cd. If both the numerator and denominator are zero, $ab = cd$.

If p is very close to b, c, or d, the numerical accuracy can often be improved by repeating the calculation with the points appropriately swapped so that a is the point nearest p. If α is close to one, then p is close to b. Swapping a with b simply flips the sign of the denominator, and only the numerator needs to be recomputed from scratch.

The expression for α is numerically unstable when the denominator is close to zero. Exact arithmetic is sometimes necessary to compute the denominator with sufficient accuracy. Note that the denominator is not equivalent to an orientation test. However, the numerator is; it is twice the signed area of $\triangle cda$.

A related question is whether two line segments $\overline{ab}$ and $\overline{cd}$ intersect. The segments intersect if both of the following are true: $\textsc{Orient2D}(c,d,a) \cdot \textsc{Orient2D}(c,d,b) \leq 0$ and $\textsc{Orient2D}(a,b,c) \cdot \textsc{Orient2D}(a,b,d) \leq 0$. In words, the points a and b lie on opposite sides of cd (or one of them lies on cd), and c and d lie on opposite sides of ab (or one of them lies on ab). If an exact orientation test is available, an exact answer can be given.

For the intersection of two lines in E^3, see Section 3.9.

The intersection (in E^3) of a line ab with a plane passing through c, d, and e is

$$p = a + \alpha(b - a),$$

where

$$\alpha = \frac{\begin{vmatrix} a_x - c_x & a_y - c_y & a_z - c_z \\ d_x - c_x & d_y - c_y & d_z - c_z \\ e_x - c_x & e_y - c_y & e_z - c_z \end{vmatrix}}{\begin{vmatrix} a_x - b_x & a_y - b_y & a_z - b_z \\ d_x - c_x & d_y - c_y & d_z - c_z \\ e_x - c_x & e_y - c_y & e_z - c_z \end{vmatrix}} = \frac{\text{Orient3D}(a,d,e,c)}{\begin{vmatrix} a_x - b_x & a_y - b_y & a_z - b_z \\ d_x - c_x & d_y - c_y & d_z - c_z \\ e_x - c_x & e_y - c_y & e_z - c_z \end{vmatrix}}.$$

The denominator is exactly zero if and only if ab is parallel to the plane through c, d, and e. If both the numerator and denominator are zero, ab lies in that plane.

If p is very close to b (i.e. α is close to one), the numerical accuracy can often be improved by repeating the calculation with a and b swapped. Swapping a with b simply flips the sign of the denominator, and only the numerator needs to be recomputed from scratch.

The forgoing comments about the stability of the line intersection calculation apply here as well. The numerator is six times the signed volume of tetrahedron $adec$.

3.7 Intersection Points with Vertical Lines

Computing the intersection of a vertical line with a hyperplane is simpler than general intersection computation. In the plane, the intersection of a line ab with a vertical line with x-coordinate e_x has y-coordinate

$$e_y = b_y + \frac{(e_x - b_x)(a_y - b_y)}{a_x - b_x}.$$

In E^3, the intersection of a plane through the points a, b, and c with a vertical line with coordinates e_x

and e_y has z-coordinate

$$e_z = c_z - \frac{\begin{vmatrix} a_x - c_x & a_y - c_y & a_z - c_z \\ b_x - c_x & b_y - c_y & b_z - c_z \\ e_x - c_x & e_y - c_y & 0 \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}} = c_z - \frac{\begin{vmatrix} a_x - c_x & a_y - c_y & a_z - c_z \\ b_x - c_x & b_y - c_y & b_z - c_z \\ e_x - c_x & e_y - c_y & 0 \end{vmatrix}}{2A}.$$

In E^4, call the fourth coordinate axis the w-axis. The formula generalizes to

$$\begin{aligned} e_w &= d_w - \frac{\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z & a_w - d_w \\ b_x - d_x & b_y - d_y & b_z - d_z & b_w - d_w \\ c_x - d_x & c_y - d_y & c_z - d_z & c_w - d_w \\ e_x - d_x & e_y - d_y & e_z - d_z & 0 \end{vmatrix}}{\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix}} \\ &= d_w - \frac{\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z & a_w - d_w \\ b_x - d_x & b_y - d_y & b_z - d_z & b_w - d_w \\ c_x - d_x & c_y - d_y & c_z - d_z & c_w - d_w \\ e_x - d_x & e_y - d_y & e_z - d_z & 0 \end{vmatrix}}{6V}. \end{aligned}$$

To compute the intersection of a non-vertical axis-parallel line with a hyperplane, simply swap the two appropriate coordinates axes so that the axis-parallel line becomes vertical, apply the formulae above, and swap the axes back again.

3.8 Altitudes and Orthogonal Projections of Points

In d-dimensional space for any $d \geq 2$, let a be a point and let L be a line through two distinct points b and c. Let $\mathbf{r} = a - c$ and $\mathbf{s} = b - c$. The point on L nearest the point a—in other words, the orthogonal projection of a onto L—is

$$p = c + \frac{\mathbf{r} \cdot \mathbf{s}}{|\mathbf{s}|^2}\mathbf{s}.$$

If p is much closer to b than to c, the numerical accuracy can often be improved by repeating the calculation with b and c swapped. Only the numerator needs to be recomputed; swapping b and c does not change the denominator.

The distance between a and L (in other words, between a and its projection) is

$$h = \frac{2A}{|\mathbf{s}|} = \frac{2A}{\ell_a},$$

where A is the area of the triangle abc. The quantity h is also known as the altitude of a in the triangle $\triangle abc$. If $d = 2$, A may be signed, in which case the sign of h is the same as the sign of A, and thus indicates the orientation of the vertices a, b, and c. The signed altitude is useful in some triangle quality measures. In three dimensions or above, A is naturally unsigned. Formulae for A appear in Section 3.1.

The minimum-magnitude altitude of a triangle is

$$a_{\min} = \frac{2A}{\ell_{\max}}.$$

In three-dimensional space, let a be a point and let P be a plane through three points b, c, and d. Let $\mathbf{t} = a - d$, $\mathbf{u} = b - d$, and $\mathbf{v} = c - d$. The point on P nearest the point a—in other words, the orthogonal projection of a onto P—is

$$q = d + \frac{(\mathbf{u} \times \mathbf{v}) \cdot (\mathbf{t} \times \mathbf{v})}{|\mathbf{u} \times \mathbf{v}|^2}\mathbf{u} + \frac{(\mathbf{u} \times \mathbf{v}) \cdot (\mathbf{u} \times \mathbf{t})}{|\mathbf{u} \times \mathbf{v}|^2}\mathbf{v}.$$

If q is much closer to b or c than to d, the numerical accuracy can often be improved by repeating the calculation with d swapped with the point nearest q.

The distance between a and P (in other words, between a and its projection) is

$$h = \frac{6V}{|\mathbf{u} \times \mathbf{v}|} = \frac{3V}{A_a},$$

where V is the volume of the tetrahedron $abcd$, and A_a is the unsigned area of the triangle $\triangle bcd$. The quantity h is also known as the altitude of a in the tetrahedron $abcd$. V may be signed, in which case the sign of h is the same as the sign of V, and thus indicates the orientation of the vertices a, b, c, and d. The signed altitude is useful in some tetrahedron quality measures. Formulae for V and A_a appear in Section 3.1.

The expressions for q and h are numerically unstable when their denominators are close to zero. These denominators are exactly zero if and only if $\mathbf{u}$ is parallel to $\mathbf{v}$ (and thus b, c, and d are not affinely independent). Exact arithmetic is sometimes necessary to compute the denominator with sufficient accuracy.

The minimum-magnitude altitude of a tetrahedron is

$$a_{\min} = \frac{3V}{A_{\max}}.$$

3.9 Lines in Three Dimensions: Distance, Closest Points, and Intersection Points

Consider two lines ab and cd in three-dimensional space. Let $\mathbf{w} = a - b$, $\mathbf{v} = c - d$, and $\mathbf{u} = b - d$. The point on ab nearest cd (which is the intersection point if ab intersects cd) is

$$p = b + \frac{(\mathbf{w} \times \mathbf{v}) \cdot (\mathbf{v} \times \mathbf{u})}{|\mathbf{w} \times \mathbf{v}|^2}\mathbf{w}.$$

The denominator is exactly zero if and only if ab is parallel to cd. If $\mathbf{v} \times \mathbf{u} = 0$ as well, then $ab = cd$.

If p is much closer to a than to b (i.e. if the fraction is close to one), the numerical accuracy can often be improved by repeating the calculation with a and b swapped. Only the numerator needs to be recomputed; swapping a with b does not change the denominator.

The point on cd nearest ab is

$$q = d + \frac{(\mathbf{w} \times \mathbf{v}) \cdot (\mathbf{w} \times \mathbf{u})}{|\mathbf{w} \times \mathbf{v}|^2}\mathbf{v}.$$

If q is much closer to c than to d (i.e. if the fraction is close to one), the numerical accuracy can often be improved by repeating the calculation with c and d swapped. Only the numerator needs to be recomputed; swapping c with d does not change the denominator.

The lines ab and cd intersect if and only if

$$\textsc{Orient3D}(a, b, c, d) = 0.$$

The signed distance between ab and cd (at their closest points) is

$$h = \frac{6V}{|\mathbf{w} \times \mathbf{v}|} = \frac{\textsc{Orient3D}(a, b, c, d)}{|\mathbf{w} \times \mathbf{v}|},$$

where V is the signed volume of the tetrahedron $abcd$. The sign of h indicates the orientation of the vertices a, b, c, and d. The signed distance is useful in some tetrahedron quality measures.

The expressions for p, q, and h are numerically unstable when $|\mathbf{w} \times \mathbf{v}|$ is close to zero. Exact arithmetic is sometimes necessary to compute the denominator with sufficient accuracy.

3.10 The Minimum Aspect of a Tetrahedron

The minimum aspect of a tetrahedron is the smallest distance between two parallel planes such that the tetrahedron fits between the planes. It is either an altitude of the tetrahedron or the distance between two opposite edges of the tetrahedron. (Two edges of a tetrahedron are *opposite* if they do not share an endpoint. The six edges of a tetrahedron consist of three pairs of opposite edges.) A tetrahedron has four altitudes, for which Section 3.8 prescribes the formula $6V/|\mathbf{u} \times \mathbf{v}|$, where $\mathbf{u}$ and $\mathbf{v}$ represent two edges of a face of the tetrahedron. A tetrahedron has three pairs of opposite edges, for which Section 3.9 prescribes the formula $6V/|\mathbf{w} \times \mathbf{v}|$, where $\mathbf{w}$ and $\mathbf{v}$ represent opposite edges of the tetrahedron.

Hence, the minimum aspect of a tetrahedron is

$$h_{\min} = \frac{6V}{\max_{\mathbf{x},\mathbf{y}} |\mathbf{x} \times \mathbf{y}|}$$

where $\mathbf{x}$ and $\mathbf{y}$ vary over all the edges of the tetrahedron. However, it is not necessary to test all pairs of edges, because some pairs are redundant; any two edges of a face will yield the same altitude. Just seven pairs of edges must be tested, corresponding to the four faces and the three opposite pairs of edges.

Note that the minimum aspect of a triangle—the smallest distance between two parallel lines such that the triangle fits between the lines—is always its minimum altitude.

3.11 Circumcenters and Circumradii of Triangles and Tetrahedra

The following expressions compute the center O_{circ} of a circle that passes through the three points a, b, and c in the plane. This circle is known as the *circumscribing circle*, or *circumcircle*, of the triangle abc.

$$O_x = c_x + \frac{\begin{vmatrix} |\mathbf{r}|^2 & r_y \\ |\mathbf{s}|^2 & s_y \end{vmatrix}}{4A} = c_x + \frac{\begin{vmatrix} (a_x - c_x)^2 + (a_y - c_y)^2 & a_y - c_y \\ (b_x - c_x)^2 + (b_y - c_y)^2 & b_y - c_y \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}.$$

$$O_y = c_y + \frac{\begin{vmatrix} r_x & |\mathbf{r}|^2 \\ s_x & |\mathbf{s}|^2 \end{vmatrix}}{4A} = c_y + \frac{\begin{vmatrix} a_x - c_x & (a_x - c_x)^2 + (a_y - c_y)^2 \\ b_x - c_x & (b_x - c_x)^2 + (b_y - c_y)^2 \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}.$$

These expressions are numerically unstable when the denominator is close to zero (i.e. for a triangle that is nearly degenerate). An accurate ORIENT2D implementation should be used to calculate the denominator if one is available.

Although the circumradius of a triangle can be computed by computing $O_{\text{circ}} - c$ with the formulae above, then the distance $|O_{\text{circ}}c|$, a slightly more accurate formula is

$$r_{\text{circ}} = \frac{\ell_a \ell_b \ell_c}{4A} = \frac{|b-c||c-a||a-b|}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}},$$

where ℓ_a, ℓ_b, and ℓ_c are the edge lengths of the triangle. Because square root operations are expensive and introduce error, the numerator above is best computed as $\sqrt{|b-c|^2|c-a|^2|a-b|^2}$.

The following expressions compute the smallest sphere that passes through the vertices of a two-dimensional triangle in E^3. The center O_{circ} of the sphere is coplanar with the triangle.

$$\begin{aligned} O_{\text{circ}} &= c + \frac{\left(|\mathbf{r}|^2\mathbf{s} - |\mathbf{s}|^2\mathbf{r}\right) \times (\mathbf{r} \times \mathbf{s})}{8A_f^2} \\ &= c + \frac{\left[|a-c|^2(b-c) - |b-c|^2(a-c)\right] \times [(a-c) \times (b-c)]}{2|(a-c) \times (b-c)|^2}. \\ r_{\text{circ}} &= \frac{\left|\left(|\mathbf{r}|^2\mathbf{s} - |\mathbf{s}|^2\mathbf{r}\right) \times (\mathbf{r} \times \mathbf{s})\right|}{8A_f^2}. \end{aligned}$$

The following expressions compute the sphere that passes through four points a, b, c, and d in E^3. This sphere is known as the *circumscribing sphere*, or *circumsphere*, of the tetrahedron $abcd$.

$$\begin{aligned} O_{\text{circ}} &= d + \frac{|\mathbf{t}|^2\mathbf{u} \times \mathbf{v} + |\mathbf{u}|^2\mathbf{v} \times \mathbf{t} + |\mathbf{v}|^2\mathbf{t} \times \mathbf{u}}{12V} \\ &= d + \frac{|a-d|^2(b-d) \times (c-d) + |b-d|^2(c-d) \times (a-d) + |c-d|^2(a-d) \times (b-d)}{2\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix}}. \\ r_{\text{circ}} &= \frac{\left||\mathbf{t}|^2\mathbf{u} \times \mathbf{v} + |\mathbf{u}|^2\mathbf{v} \times \mathbf{t} + |\mathbf{v}|^2\mathbf{t} \times \mathbf{u}\right|}{12V}. \end{aligned}$$

These expressions, like the two-dimensional formula, are numerically unstable when the denominator is close to zero (i.e. for a tetrahedron that is nearly degenerate). An accurate ORIENT3D implementation should be used to calculate the denominator if one is available.

3.12 Orthocenters and Orthoradii of Triangles and Tetrahedra

Suppose the points a, b, and c are assigned *weights* w_a, w_b, and w_c; or *heights* h_a, h_b, and h_c. The height of a vertex is the x_{d+1}-coordinate to which it is lifted by the parabolic lifting map of Seidel [20, 6]. In a weighted Delaunay triangulation or power diagram, the weights and heights are related by $h_a = |a|^2 - w_a$. Imagine a circle around each vertex a with radius w_a, and similar circles for the other two vertices. The orthocircle of the triangle abc is the circle that is orthogonal to the circles around a, b, and c. In the special case where all three weights are zero, the orthocircle and the circumcircle are the same.

The numerically best-behaved expressions for computing the orthocircle depend on whether the input includes the heights or the weights. (Either one can be computed from the other, but the bit length increases.)

If the weights are given, use the following expressions for the orthcenter o_{orth} and orthoradius r_{orth}. Note that a triangle with vertex weights can have an imaginary orthoradius.

$$
\begin{aligned}
o_x &= c_x + \frac{\begin{vmatrix} |\mathbf{r}|^2 + (w_c - w_a) & r_y \\ |\mathbf{s}|^2 + (w_c - w_b) & s_y \end{vmatrix}}{4A} \\
&= c_x + \frac{\begin{vmatrix} (a_x - c_x)^2 + (a_y - c_y)^2 + (w_c - w_a) & a_y - c_y \\ (b_x - c_x)^2 + (b_y - c_y)^2 + (w_c - w_b) & b_y - c_y \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}. \\
o_y &= c_y + \frac{\begin{vmatrix} r_x & |\mathbf{r}|^2 + (w_c - w_a) \\ s_x & |\mathbf{s}|^2 + (w_c - w_b) \end{vmatrix}}{4A} \\
&= c_y + \frac{\begin{vmatrix} a_x - c_x & (a_x - c_x)^2 + (a_y - c_y)^2 + (w_c - w_a) \\ b_x - c_x & (b_x - c_x)^2 + (b_y - c_y)^2 + (w_c - w_b) \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}. \\
r_{\text{orth}}^2 &= |o_{\text{orth}}c|^2 - w_c = \frac{\begin{vmatrix} |\mathbf{r}|^2 + (w_c - w_a) & r_y \\ |\mathbf{s}|^2 + (w_c - w_b) & s_y \end{vmatrix}^2 + \begin{vmatrix} r_x & |\mathbf{r}|^2 + (w_c - w_a) \\ s_x & |\mathbf{s}|^2 + (w_c - w_b) \end{vmatrix}^2}{16A^2} - w_c.
\end{aligned}
$$

For the last formula, do not compute r_{orth} directly from o_{orth}. The value of $|o_{\text{orth}}c|^2$ can be computed much more accurately, as shown.

If the heights are given, use the following expressions.

$$
\begin{aligned}
o_x &= \frac{\begin{vmatrix} h_a - h_c & r_y \\ h_b - h_c & s_y \end{vmatrix}}{4A} = \frac{\begin{vmatrix} h_a - h_c & a_y - c_y \\ h_b - h_c & b_y - c_y \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}. \\
o_y &= \frac{\begin{vmatrix} r_x & h_a - h_c \\ s_x & h_b - h_c \end{vmatrix}}{4A} = \frac{\begin{vmatrix} a_x - c_x & h_a - h_c \\ b_x - c_x & h_b - h_c \end{vmatrix}}{2\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}}. \\
r_{\text{orth}}^2 &= |o_{\text{orth}}c|^2 + (h_c - |c|^2).
\end{aligned}
$$

In this case, r_{orth} is computed directly from o_{orth}. The parentheses indicate that it is wise to perform the subtraction before the addition.

The following formulae compute the orthosphere of a tetrahedron.

$$
o_{\text{orth}} = d + \frac{\left(|\mathbf{t}|^2 + (w_d - w_a)\right)\mathbf{u} \times \mathbf{v} + \left(|\mathbf{u}|^2 + (w_d - w_b)\right)\mathbf{v} \times \mathbf{t} + \left(|\mathbf{v}|^2 + (w_d - w_b)\right)\mathbf{t} \times \mathbf{u}}{12V}
$$

$$\begin{aligned}
&= \frac{(h_a - h_d)\mathbf{u}\times\mathbf{v} + (h_b - h_d)\mathbf{v}\times\mathbf{t} + (h_c - h_d)\mathbf{t}\times\mathbf{u}}{12V}. \\
r_{\mathrm{orth}}^2 &= |o_{\mathrm{orth}} d|^2 - w_d \\
&= \frac{\left|\left(|\mathbf{t}|^2 + (w_d - w_a)\right)\mathbf{u}\times\mathbf{v} + \left(|\mathbf{u}|^2 + (w_d - w_b)\right)\mathbf{v}\times\mathbf{t} + \left(|\mathbf{v}|^2 + (w_d - w_b)\right)\mathbf{t}\times\mathbf{u}\right|^2}{144V^2} - w_d \\
&= |o_{\mathrm{orth}} d|^2 + (h_d - |d|^2).
\end{aligned}$$

Again, the value of r_{orth} must be computed directly from o_{orth} if the heights are given, but if the weights are given, r_{orth} should be computed without using o_{orth} as an intermediate expression.

3.13 Min-Containment Circles and Spheres of Triangles and Tetrahedra

Let O_{mc} and r_{mc} denote the center and radius of the smallest circle or sphere that encloses a triangle or tetrahedron.

To find the min-containment circle of a triangle, first determine whether the triangle has an angle of 90° or greater. This is quickly accomplished by checking the signs of the dot products of each pair of edge vectors. If one of the angles is 90° or greater, then O_{mc} is the midpoint of the opposite edge and r_{mc} is half its length (and there is no need to test the remaining angles). If all three angles are acute, the min-containment circle is the circumcircle and the min-containment radius is the circumradius; see Section 3.11.

To find the min-containment sphere of a tetrahedron $abcd$, first compute its circumcenter O_{circ}. Next, test whether O_{circ} is in the tetrahedron by checking it against each triangular face. If ORIENT3D$(O_{\mathrm{circ}}, b, c, d)$, ORIENT3D$(O_{\mathrm{circ}}, a, d, c)$, ORIENT3D$(O_{\mathrm{circ}}, d, a, b)$, and ORIENT3D$(O_{\mathrm{circ}}, c, b, a)$ are all nonnegative, then $O_{\mathrm{mc}} = O_{\mathrm{circ}}$ and $r_{\mathrm{mc}} = r_{\mathrm{circ}}$. If exactly one orientation test returns a negative value, then O_{mc} and r_{mc} are the center and radius of the min-containment circle of the corresponding triangular face. If two of the volumes are negative, then O_{mc} and r_{mc} are the center and half the length of the edge where the two corresponding triangular faces meet.

In the case where O_{mc} is a circumcenter of one of the faces, it may be slightly faster, but slightly less accurate, to find the circumcenter of the triangular face by orthogonally projecting O_{circ} onto the face using the formula in Section 3.8, instead of using the face circumcenter formula in Section 3.11.

3.14 Incenters and Inradii of Triangles and Tetrahedra

The center of the smallest circle in the plane that intersects all three sides of a triangle (also known as the *inscribed circle*) is

$$O_{\mathrm{in}} = \frac{\sum_{i=1}^3 \ell_i v_i}{\sum_{i=1}^3 \ell_i},$$

and its radius is

$$r_{\mathrm{in}} = \frac{2A}{\sum_{i=1}^3 \ell_i}.$$

The center of the smallest sphere in E^3 that intersects all four faces of a tetrahedron (also known as the *inscribed sphere*) is

$$O_{\mathrm{in}} = \frac{\sum_{i=1}^4 A_i v_i}{\sum_{i=1}^4 A_i},$$

and its radius is

$$r_{\text{in}} = \frac{3V}{\sum_{i=1}^{4} A_i}.$$

3.15 Power Functions

For a circle or sphere S, the *power function* $\pi_S(p)$ maps a point p to a number such that $\sqrt{\pi_S(p)}$ is the radius of the circle or sphere centered at p that is orthogonal to S. Let $\pi_{\triangle abc}(p)$ denote $\pi_S(p)$ where S is the orthosphere or circumsphere of $\triangle abc$ (see Section 3.12).

Consider the two-dimensional case. Suppose the points a, b, and c are assigned *weights* w_a, w_b, and w_c and *heights* $h_a = |a|^2 - w_a$, $h_b = |b|^2 - w_b$, and $h_c = |c|^2 - w_c$. Define the *lifted point* $a^+ \in E^3$ to be $\langle a_x, a_y, h_a \rangle$, and likewise for b^+ and c^+. Let H be the plane in E^3 that contains a^+, b^+, and c^+. Let $H(p)$ be a function that maps any point $p \in E^2$ to the z-coordinate such that $\langle p_x, p_y, H(p) \rangle \in H$. Observe that computing $H(p)$ is equivalent to computing the intersection of a vertical line with H, which is addressed in Section 3.7.

The power function obeys the identity $\pi_S(p) = |p|^2 - H(p)$. As with computing orthospheres, the numerically best-behaved way to compute $\pi_S(p)$ depends on whether the input includes the heights or the weights of the vertices that define S. If the heights are provided, one may compute $H(p)$ as described in Section 3.7, treating each height as an extra coordinate, then calculate $\pi_S(p) = |p|^2 - H(p)$. If the weights are provided, the following formulae are preferable.

For a triangle abc in the plane,

$$\pi_{\triangle abc}(e) = \frac{\begin{vmatrix} a_x - c_x & a_y - c_y & (a_x - c_x)^2 + (a_y - c_y)^2 + (w_c - w_a) \\ b_x - c_x & b_y - c_y & (b_x - c_x)^2 + (b_y - c_y)^2 + (w_c - w_b) \\ e_x - c_x & e_y - c_y & (e_x - c_x)^2 + (e_y - c_y)^2 \end{vmatrix}}{\begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}} - w_c.$$

If all the weights are zero (i.e. $\pi_{\triangle abc}$ is determined by the circumcircle of the triangle), this formula reduces to

$$\pi_{\triangle abc}(e) = \frac{\text{INCIRCLE}(a, b, e, c)}{2A}.$$

For a tetrahedron $abcd$ in E^3,

$$\pi_{abcd}(e) = \frac{\begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z & (a_x - d_x)^2 + (a_y - d_y)^2 + (a_z - d_z)^2 + (w_d - w_a) \\ b_x - d_x & b_y - d_y & b_z - d_z & (b_x - d_x)^2 + (b_y - d_y)^2 + (b_z - d_z)^2 + (w_d - w_b) \\ c_x - d_x & c_y - d_y & c_z - d_z & (c_x - d_x)^2 + (c_y - d_y)^2 + (c_z - d_z)^2 + (w_d - w_c) \\ e_x - d_x & e_y - d_y & e_z - d_z & (e_x - d_x)^2 + (e_y - d_y)^2 + (e_z - d_z)^2 \end{vmatrix}}{\begin{vmatrix} a_x - d_x & a_y - d_y & a_y - d_z \\ b_x - d_x & b_y - d_y & b_y - d_z \\ c_x - d_x & c_y - d_y & c_y - d_z \end{vmatrix}} - w_d.$$

If all the weights are zero (i.e. $\pi_{\triangle abc}$ is determined by the circumsphere of the tetrahedron), this formula reduces to

$$\pi_{abcd}(e) = \frac{\text{INSPHERE}(a, b, c, e, d)}{6V}.$$

References

[1] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic*. Algorithmica **17**(2):111–132, February 1997.

[2] C. Bradford Barber. *Computational Geometry with Imprecise Data and Arithmetic*. Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, October 1992. Available as Technical Report CS-TR-377-92.

[3] Christoph Burnikel, Jochen Könemann, Kurt Mehlhorn, Stefan Näher, Stefan Schirra, and Christian Uhrig. *Exact Geometric Computation in LEDA*. Eleventh Annual Symposium on Computational Geometry (Vancouver, British Columbia, Canada), pages C18–C19. Association for Computing Machinery, June 1995.

[4] John Canny. *Some Algebraic and Geometric Computations in PSPACE*. 20th Annual Symposium on the Theory of Computing (Chicago, Illinois), pages 460–467. Association for Computing Machinery, May 1988.

[5] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation*. 33rd Annual Symposium on Foundations of Computer Science (Pittsburgh, Pennsylvania), pages 387–395. IEEE Computer Society Press, October 1992.

[6] Herbert Edelsbrunner and Raimund Seidel. *Voronoi Diagrams and Arrangements*. Discrete & Computational Geometry **1**:25–44, 1986.

[7] F. Eriksson. *On the Measure of Solid Angles*. Mathematics Magazine **63**(3):184–187, 1990.

[8] Steven Fortune. *Stable Maintenance of Point Set Triangulations in Two Dimensions*. 30th Annual Symposium on Foundations of Computer Science, pages 494–499. IEEE Computer Society Press, 1989.

[9] ———. *Progress in Computational Geometry*. Directions in Geometric Computing (R. Martin, editor), chapter 3, pages 81–128. Information Geometers Ltd., 1993.

[10] ———. *Progress in Computational Geometry*. Directions in Geometric Computing (R. Martin, editor), chapter 3, pages 81–128. Information Geometers Ltd., 1993.

[11] ———. *Numerical Stability of Algorithms for 2D Delaunay Triangulations*. International Journal of Computational Geometry & Applications **5**(1–2):193–213, March–June 1995.

[12] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry*. Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 163–172. Association for Computing Machinery, May 1993.

[13] ———. *Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry*. ACM Transactions on Graphics **15**(3):223–248, July 1996.

[14] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoï Diagrams*. ACM Transactions on Graphics **4**(2):74–123, April 1985.

[15] Christoph M. Hoffmann. *The Problems of Accuracy and Robustness in Geometric Computation*. Computer **22**(3):31–41, March 1989.

[16] Michael Karasick, Derek Lieber, and Lee R. Nackman. *Efficient Delaunay Triangulation Using Rational Arithmetic*. ACM Transactions on Graphics **10**(1):71–91, January 1991.

[17] Anwei Liu and Barry Joe. *Relationship between Tetrahedron Shape Measures*. BIT **34**:268–287, 1994.

[18] Victor Milenkovic. *Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections using Rounded Arithmetic*. 30th Annual Symposium on Foundations of Computer Science, pages 500–505. IEEE Computer Society Press, 1989.

[19] N. E. Mnev. *The Universality Theorems on the Classification Problem of Configuration Varieties and Convex Polytopes Varieties*. Topology and Geometry - Rohlin Seminar (O. Ya. Viro, editor), Lecture Notes in Mathematics, volume 1346, pages 527–543. Springer-Verlag, 1988.

[20] Raimund Seidel. *Voronoi Diagrams in Higher Dimensions*. Diplomarbeit, Institut für Informationsverarbeitung, Technische Universität Graz, 1982.

[21] Jonathan Richard Shewchuk. *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*. Discrete & Computational Geometry **18**(3):305–363, October 1997.

[22] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.