Epsilon Geometry: Building Robust Algorithms from Imprecise Computations

Leonidas Guibas[1,2]   David Salesin[1]   Jorge Stolfi[2]

[1]*Stanford University*   [2]*DEC Systems Research Center*

1 Introduction

We describe a new general framework, called *Epsilon Geometry*, for coping with computational errors in geometric algorithms that arise from the use of finite precision arithmetic.

The Epsilon Geometry framework allows us to build robust geometric algorithms out of imprecise geometric primitives. Our method combines the techniques of interval arithmetic and backward error analysis, along with a good deal of geometric reasoning. Our algorithms compute an exact solution for a perturbed version of the input, and they return a bound on the size of this implicit perturbation.

The problem of building robust geometric algorithms has received a good deal of attention in the last few years. For example, Yap [9] and Edelsbrunner and Mücke [1] studied the problem of coping with geometric degeneracies, such as the possibility of three collinear points. The methods they propose are vaguely reminiscent of ours, in that they too make use of use perturbations on the input data. However, their perturbations are infinitesimal and bear only a superficial resemblance to ours. Furthermore, the problem of imprecise computations that we address is distinct and much harder, in the sense that rounding errors not only increase the likelihood of degenerate cases, but they also introduce the possibility of inconsistencies. For example, imprecise computations may tell us that points a, b, c and b, c, d are collinear, but that points $a, b,$ and d are not.

Ottmann, Thiemt, and Ullrich [8] showed how it is possible to create a robust segment-intersection algorithm assuming a scalar-product operator that is exact to within the machine's precision. By contrast, our framework is designed to cope with computations that are significantly less accurate than the machine's precision limits. Greene and Yao [2] showed how a discrete version of the segment-intersection problem can be formulated and satisfactorily solved, but they too assume precise computations.

Our approach is more similar to those of Milenkovic [6] and of Hoffmann, Hopcroft, and Karasick [3]. These methods compute an exact result for a perturbed version of the input data, but they assume a perturbation bounded by a constant chosen *a priori*. Among other differences, the algorithms in our framework can determine the size of the required perturbation based on the size of the rounding errors observed during the computation.

1.1 Epsilon-Predicates

The following definitions attempt to capture the notion of "'approximate tests" in a very general setting. Let $\mathcal{O}$ be a set of *objects* endowed with some distance metric $\| \cdot, \cdot \|$. Let P be a predicate defined on $\mathcal{O}$. Then for any $X \in \mathcal{O}$ and any $\varepsilon \geq 0$, we define $\varepsilon\text{-}P(X)$ as a shorthand for "$P(X')$ is true for *some* $X' \in \mathcal{O}$ such that $\|X, X'\| \leq \varepsilon$." That is, X is at most ε away from satisfying $P(X)$. Therefore, the truth-set of $\varepsilon\text{-}P$ is that of P, "fattened" by ε. Note that $0\text{-}P(X)$ is the same as $P(X)$.

In order to extend these definitions to an n-ary predicate P, we note that if $\mathcal{O}_1, \ldots, \mathcal{O}_n$ are metric spaces with distance functions $\| \; \|_1, \ldots, \| \; \|_n$, then $\mathcal{O}_1 \times \cdots \times \mathcal{O}_n$ is a metric space with the distance function given by

$$\|X, Y\| = \max \{\|X_1, Y_1\|_1, \ldots, \|X_n, Y_n\|_n\} \quad (1)$$

Thus, for example, if $\mathcal{O}_1 = \mathcal{O}_2 = \Re$ with metric $\|x, x'\| = |x - x'|$, then $\mathcal{O}_1 \times \mathcal{O}_2$ has the implicit metric $\|(x, y), (x', y')\| = \max \{|x - x'|, |y - y'|\}$. Therefore, if $P(x, y)$ is the predicate $(x > y)$, then $\varepsilon\text{-}P(x, y)$ means

that $x > y$ can be made true by perturbing x and y by at most ε.

Throughout this paper we will assume that $\| \ \|$ is a Minkowski metric. Minkowski metrics include all L_p distance functions, in particular the Euclidean (L_2) and Manhattan (L_1) metrics. Note that the composite metric (1) is Minkowski if the metrics $\| \ \|_i$ are Minkowski.

It follows immediately from the definition of an ε-predicate that

$$\varepsilon\text{-}P(X) \Rightarrow \varepsilon'\text{-}P(X) \quad \text{for all } \varepsilon' \geq \varepsilon \qquad (2)$$

and

$$\text{not}(\varepsilon\text{-}P(X)) \Rightarrow \text{not}(\varepsilon'\text{-}P(X)) \quad \text{for all } \varepsilon' \leq \varepsilon \qquad (3)$$

We can extend our definition of ε-predicate to negative values of ε, in such a way that properties (2) and (3) remain true for all ε. If $\varepsilon > 0$, we define $(-\varepsilon)\text{-}P(X)$ as a shorthand for "$P(X')$ is true for all $X' \in \mathcal{O}$ such that $\|X, X'\| \leq \varepsilon$." Intuitively, an object X that is $(-\varepsilon)\text{-}P$ is "extremely P," whereas an X that is only $\varepsilon\text{-}P$ is just "nearly P." This definition can also be expressed by the identity

$$(-\varepsilon)\text{-}P(X) \Leftrightarrow \text{not}(\varepsilon\text{-}(\text{not } P(X)))$$

The following properties of epsilon-predicates follow readily from the definitions and the triangle inequality for distances:

Lemma 1 *For any predicates, P and Q, and any $\varepsilon \geq 0$,*

$$\begin{aligned} \varepsilon\text{-}(P \vee Q)(X) \quad &\Leftrightarrow \quad \varepsilon\text{-}P(X) \vee \varepsilon\text{-}Q(X) \\ \varepsilon\text{-}(P \wedge Q)(X) \quad &\Rightarrow \quad \varepsilon\text{-}P(X) \wedge \varepsilon\text{-}Q(X) \end{aligned}$$

Note that in the case of $\wedge$ the implication only works in one direction, because even if it is possible to satisfy $P(X)$ and $Q(X)$ separately with ε-perturbations to X, it may not always be possible to satisfy both constraints simultaneously.

Lemma 2 *For any $\varepsilon, \delta \geq 0$, and any predicates P, Q, R, if $P(X)$ implies $\varepsilon\text{-}Q(X)$, and $Q(X)$ implies $\delta\text{-}R(X)$, then $P(X)$ implies $(\varepsilon + \delta)\text{-}R(X)$.*

1.2 Implementing Epsilon-Predicates

When geometric tests such as collinearity or convexity are implemented using floating point arithmetic in the straightforward way, their outcome becomes subject to errors. In order to quantify those errors and allow geometric algorithms to cope with them, we propose to implement each geometric test $P(X)$ as a procedure $\mathbf{P}(X)$ that, instead of simply returning `true` or `false`, returns an estimate of how far X is from satisfying P.

More precisely, the procedure $\mathbf{P}(X)$ should return a partition of the real line into three sets F, U, T, such that the predicate $\varepsilon\text{-}P(X)$ is false for $\varepsilon \in F$, true for $\varepsilon \in T$, and unknown for $\varepsilon \in U$. We call such a procedure an *epsilon-box* for P.

Given that $\varepsilon\text{-}P(X)$ is a monotonic boolean function of ε, for any fixed X, we can assume that F, U, and T are intervals of the real line (empty, bounded, or unbounded), with F before U before T. Since the floating-point numbers are a discrete set, such a partition can be represented by a pair of numbers $e = (e.lo, e.hi)$, such that $\varepsilon\text{-}P(X)$ is false for $\varepsilon < e.lo$, true for $\varepsilon \geq e.hi$, and unknown for $e.lo \leq \varepsilon < e.hi$. To cover all possible tri-partitions of $\Re$, we must let $e.lo$ and $e.hi$ assume also the special values $+\infty$ or $-\infty$.

We will call such pairs *intervals of uncertainty* or simply *intervals*. Note that this name is slightly misleading, because the pairs cannot be viewed as ordinary intervals of the real line. The difference is that two degenerate pairs (x, x) and (y, y) with $x \neq y$ are quite distinct outcomes, even though they define the same (empty) set U of "uncertain" ε-values.

Informally, the pair e returned by $\mathbf{P}(X)$ tells us that X is at least $e.lo$ and at most $e.hi$ away from satisfying the predicate P. In particular, if $e.lo > 0$, then $P(X)$ is definitely false; if $e.hi \leq 0$, then $P(X)$ is definitely true; and if $e.lo \leq 0 < e.hi$, the procedure was unable to decide whether $P(X)$ is true of false because of computation errors.

As we shall see, this approach allows us to build robust geometric algorithms out of arbitrarily inaccurate primitives. In general, such an algorithm will produce results that are correct only in an approximate sense. However, the algorithm will always be able to combine the uncertainty intervals returned by the primitives into a "warranty" for the result: that is, an interval of uncertainty that states how far the result that was returned may be from the exact one.

In principle, we do not make any assumptions on the size of the uncertainty intervals U returned by an epsilon-box. We consider an epsilon-box to be *correct* as long as $\varepsilon\text{-}P(X)$ is false for all $\varepsilon \in F$, and true for all $\varepsilon \in T$. An epsilon-box box is always allowed to say "I don't know" for an arbitrarily large range U or ε values. In particular, the trivial epsilon-box that always returns $T = F = \phi$, $U = \Re$ is a correct implementation of *any* predicate P.

In practice, of course, an epsilon-box should keep its uncertainty trange U reasonably small in order to be useful. Typically, a primitive box $\mathbf{P}(X)$ will compute the distance from X to the truth-set of P using floating point arithmetic, and estimate the uncertainty of the result according to standard numerical analysis techniques. For most primitives, this epsilon-box will be

only a few times slower than the naïve implementation of $P(X)$. We will come back to this topic in section 2.

1.3 Combining Epsilon-Boxes

The following operations on uncertainty intervals turn out to be useful for combining primitive epsilon-boxes into more complex algorithms. Given two uncertainty intervals a, b, we define

$$
\begin{aligned}
\min\{a,b\} &= (\min\{a.lo, b.lo\},\ \min\{a.hi, b.hi\}) \\
\max\{a,b\} &= (\max\{a.lo, b.lo\},\ \max\{a.hi, b.hi\}) \\
a \sqcup b &= (\min\{a.lo, b.lo\},\ \max\{a.hi, b.hi\}) \\
a \sqcap b &= (\max\{a.lo, b.lo\},\ \min\{a.hi, b.hi\})
\end{aligned}
$$

For example, suppose we managed to prove that the predicate $\varepsilon\text{-}R(X)$ is equivalent to $\varepsilon\text{-}P(X) \vee \varepsilon\text{-}Q(X)$, for all X and all ε. Then the predicate $R(X)$ can be implemented as the procedure $R(X)$ that evaluates the intervals $a = P(X)$ and $b = Q(X)$, and returns the interval $\min\{a, b\}$. Note that this interval is generally *not* the union of the intervals a and b.

This rule is easy to understand with the help of figure 2, which shows the "graphs" of the predicates $\varepsilon\text{-}P(X)$, $\varepsilon\text{-}Q(X)$, and $\varepsilon\text{-}R(X)$, for a particular X, as a function of ε. The "fuzzy" portion of each graph represents the uncertainty interval returned by the corresponding epsilon-box.
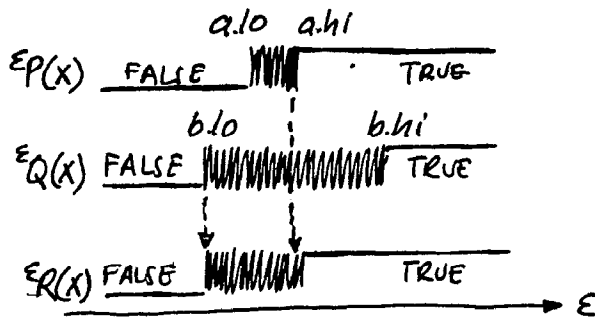

Figure 2.

Similary, if we know that the predicate $\varepsilon\text{-}R(X)$ is equivalent to $\varepsilon\text{-}P(X) \wedge \varepsilon\text{-}Q(X)$, for all X and all ε, then the procedure $R(X)$ should return $\max\{P(X), Q(X)\}$.

Note that in most machines these operations can be performed without any rounding errors. Note also that the uncertainty $c.hi - c.lo$ of the result c is no greater than the uncertainty in the inputs a and b.

Another situation that often arises in algorithms is the following. Suppose we computed two possible results a and b for the epsilon-box $R(X)$, by two different algorithms, and we know that the first algorithm is correct when some condition $P(X)$ is true, and the second one is correct when $P(X)$ is false. If we cannot decide whether $P(X)$ is true or false, we can still tell

that $\varepsilon\text{-}R(X)$ is false for $\varepsilon < \min\{a.lo, b.lo\}$, and true for $\varepsilon \geq \max\{a.hi, b.hi\}$. Therefore, the procedure $R(X)$ can safely return the pair $r = a \sqcup b$, which we call the *join* of a and b.

As a final example, suppose again that we have computed two possible results a and b for $R(X)$, but now we know that *both* intervals are correct. This tells us that $\varepsilon\text{-}R(X)$ is false for $\varepsilon < \max\{a.lo, b.lo\}$, and true for $\varepsilon \geq \min\{a.hi, b.hi\}$. Therefore, $R(X)$ can return the pair $r = a \sqcap b$, which we call the *meet* of a and b.

Note that if both a and b are correct outcomes, the result $r = a \sqcap b$ is bound to satisfy $r.lo \leq r.hi$. However, in more complicated expressions this need not be the case. For those situations, it is useful to define the "impossible" interval $\emptyset = (+\infty, -\infty)$, and define by convention $a \sqcap b = \emptyset$ whenever a and b are incompatible outcomes, that is, whenever $a.lo > b.hi$ or $b.lo > a.hi$. Note that $\emptyset \sqcap a = a \sqcap \emptyset = \emptyset$, and $a \sqcup \emptyset = \emptyset \sqcup a = a$, for any a.

2 Some Basic Predicates

Now let's consider in more detail the implementation of basic geometric predicates of two-dimensional geometry. Unless said otherwise, we will measure distances between points in the plane with the familiar Euclidean (L_2) metric, $\|p, q\| = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$. However, many of the results that follow can be adapted to any other L_p metric. In fact, some of the algorithms described below may be easier to code (but probably harder to prove) if we defined $\| \ \|$ to be the L_∞ or L_1 metric.

2.1 Coincidence

Let's consider first the *Coincident* predicate, which merely tests whether two points of the plane are coincident or distinct. According to the general definition, the derived predicate $\varepsilon\text{-}Coincident(p, q)$ is true if and only if $\varepsilon \geq \frac{1}{2}\|p, q\|$. Therefore, the procedure **Coincident** that implements this predicate should compute the distance $\|p, q\|$, and return a pair $(e.lo, e.hi)$ such that $e.lo \leq \frac{1}{2}\|p, q\| \leq e.hi$. This interval tells us that it is possible to make p and q coincident if we displace both points by $e.hi$ in suitable directions, and it is not possible to make p and q coincident if we displace them by less than $e.lo$.

Note that depending on the application, it may not be necessary to compute the Euclidean distance $\|p, q\|$ with high accuracy. For example, we can use the property that $\|p, q\|_\infty \leq \|p, q\| \leq \sqrt{2}\|p, q\|_\infty$, where $\|p, q\|_\infty = \max\{|p.x - q.x|, |p.y - q.y|\}$. The **Coincident** box may then compute $D = \|p, q\|_\infty$, and return the interval $(e.lo, e.hi) = (D/2, D/\sqrt{2})$. This is a

very coarse approximation, with a relative uncertainty of the result is almost 50%; however, it is somewhat faster to compute than the square root formula, and it may still be accurate enough for many applications.

2.2 Estimating Roundoff Errors

In practice, besides the approximation error that results from using the L_∞ norm to compute a distance, we also have rounding errors due to the subtraction and the division by $\sqrt{2}$. Fortunately, the magnitude of these errors is easy to estimate.

A fundamental "axiom" of numerical analysis [5] says that for each floating-point number system there is a constant u (the *machine precision*) such that the result c^* of computing $c = a * b$ in floating point (where $*$ is either $+$, $-$, $\cdot$, or $/$) satisfies $c^* = c(1 + \lambda)$, for some λ with $|\lambda| \le u$. Furthermore, the same guarantee applies to the basic numerical functions ($\sqrt{\ }$, sin, exp, etc.), if they are properly implemented.

So, for example, the computed value d^* of the distance

$$d = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2} \qquad (4)$$

is actually

$$d^* = \{[(p.x - q.x)(1 + \lambda_1)]^2(1 + \lambda_2)$$
$$+ [(p.y - q.y)(1 + \lambda_3)]^2(1 + \lambda_4)\}^{1/2}(1 + \lambda_5)(5)$$

where $|\lambda_i| \le u$ for all i.

We can simplify the last expression by resorting to a standard numerical analysis trick. Observe that the maximum relative rounding error $|\lambda|$ in a floating-point operation is normally very small, typically 10^{-6} or less. If we define u to be just a little bigger than this maximum error (say, twice as big), then we can prove that any expression of the form

$$\prod_{i=1}^{m}(1 + \lambda_i) \Big/ \prod_{j=1}^{n}(1 + \lambda_j')$$

lies in the interval $1 \pm (m + n)u$, provided m and n are not too big. In particular, this "safety factor" built into u allows us to ignore second and higher powers of the λ_i in error bounds, and freely move factors of $(1 \pm \lambda_i)$ between the numerator and denominator. Using this trick, formula (5) simplifies to

$$d^* = \sqrt{[(p.x - q.x)^2 + (p.y - q.y)]^2(1 + 3\lambda_6)}(1 + \lambda_5)$$

and finally to

$$d^* = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}(1 + \tfrac{5}{2}\lambda_7) \qquad (6)$$

for some λ_i with $|\lambda_i| \le u$. We conclude that the exact distance d is well inside the interval $d^* \pm 3u$. Therefore,

the Coincident procedure can return the uncertainty interval

$$e = (e.lo, e.hi) = (d^*(1 - 3u)/2, \ d^*(1 + 3u)/2) \qquad (7)$$

Note that the division by 2 is exact, and the rounding error in the multiplication by $(1 + 3u)$ is on the order of u^2 and is therefore covered by the safety factor implicit in u.

Of course there are many other correct implementations of *Coincident*, each with its own cost and accuracy. We are not concerned here with the problem of choosing between those alternatives. Our goal is not so much to design fast or accurate primitives, but to show how to make good use of arbitrarily inaccurate ones.

2.3 Collinearity and Orientation

The tests for collinearity and orientation of three given points deserve careful discussion, since they are basic building blocks of many two-dimensional geometric algorithms. We denote by $Collinear(p, q, r)$ the predicate that checks whether the points p, q and r of $\Re^2$ lie on a common straight line, in any order. Therefore, ε-$Collinear(p, q, r)$ is true if there exists a line l that passes within ε of all three points. The predicates *Collinear* and ε-*Collinear* are obviously symmetric in their three arguments.

We can visualize the ε-*Collinear* predicate as follows. Consider the disks P and Q of radius ε centered at p and q, respectively. The set of all lines passing through a point of P and a point of Q cover a bow-tie-shaped region of the plane bounded by the two inner and two outer tangents of P and Q. (If P and Q have a point in common, then this region degenerates to the entire plane.) We call this region the ε-*butterfly* determined by p and q. See figure 3.
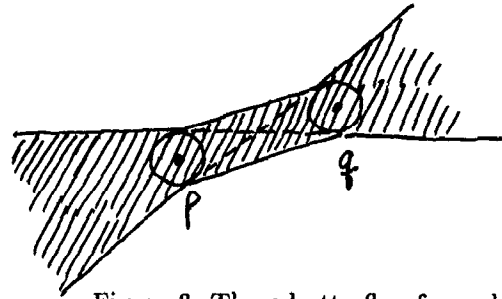

Figure 3. The ε-butterfly of p and q.

Obviously, the three points p, q, and r are ε-collinear if and only if the ε-disk centered at r intersects the ε-butterfly of p and q. Equivalently, the three points are ε-collinear if and only if one of the ε-disks intersects the ε-*stroke* of the other two points, which is how we call the convex hull of the two ε-disks centered at those
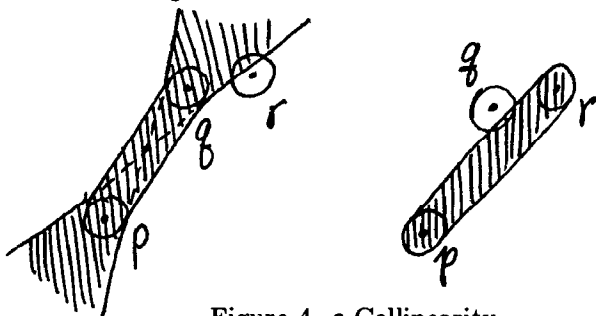
points. See figure 4.

Figure 4. ε-Collinearity.

In exact geometry, a triangle $T = (p, q, r)$ whose vertices are not collinear can be further classified by its orientation, either positive (counterclockwise) or negative (clockwise). The orientation is the sign of the determinant

$$D(p, q, r) = \begin{vmatrix} 1 & p.x & p.y \\ 1 & q.x & q.y \\ 1 & r.x & r.y \end{vmatrix} \tag{8}$$

We define the predicates $Pos(p, q, r)$ and $Neg(p, q, r)$ as meaning $D(p, q, r) \geq 0$ and $D(p, q, r) \leq 0$, respectively. Note that $Pos(T)$ is not the same thing as not $Neg(T)$; in fact $Pos(T) \wedge Neg(T) \equiv Collinear(T)$. (This convention is a bit confusing, but it seems to simplify many of the proofs and algorithms we will see later on.)

By definition, then, $\varepsilon\text{-}Pos(p, q, r)$ means that is possible to make $D(p, q, r) \geq 0$ by displacing the three points by at most ε in suitable directions. In graphical terms, $\varepsilon\text{-}Pos(p, q, r)$ means that the closed ε-disk centered at r either intersects the ε-butterfly of p and q, or lies fully to the left of it (as looking from p towards q). See figure 5.
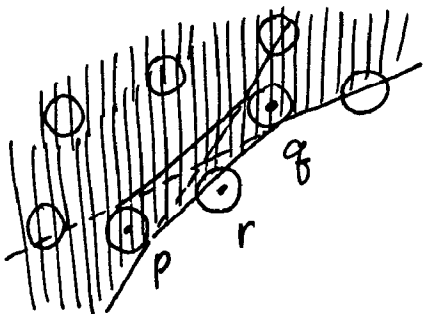

Figure 5. ε-Orientation.

From linear algebra we know that the the determinant D changes sign if we swap any two of the three points, and remains unchanged if the three permuted are permuted in a cyclic fashion. Thus,

$$\varepsilon\text{-}Pos(p, q, r) \equiv \varepsilon\text{-}Pos(q, r, p) \equiv \varepsilon\text{-}Pos(r, p, q)$$
$$\equiv \varepsilon\text{-}Neg(q, p, r) \equiv \varepsilon\text{-}Neg(r, q, p) \equiv \varepsilon\text{-}Neg(p, r, q)$$

Note again that $\varepsilon\text{-}Neg(T)$ is quite different from not $\varepsilon\text{-}Pos(T)$, and, in fact,

$$\varepsilon\text{-}Collinear(p, q, r) \equiv \varepsilon\text{-}Neg(p, q, r) \wedge \varepsilon\text{-}Pos(p, q, r)$$

Geometrically, the determinant D is twice the area of the triangle pqr, with a plus or minus sign depending on the orientation of the three points. In the Euclidean metric, the smallest perturbation that makes the three points collinear is one that moves them onto the perpendicular bisector of the shortest altitude of the triangle. See figure 6.
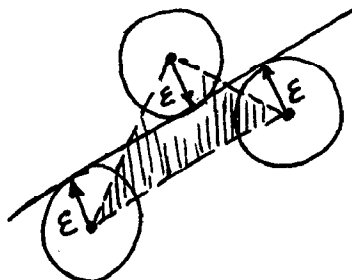

Figure 6.

Since the area of a triangle is given by one half its base times its height, the necessary perturbation ε is $\frac{1}{2}|D|/b$, where b is the length of the longest side.

We can use this result to implement a Pos box that uses only single-precision floating-point computations, as follows. First, we need to estimate the rounding errors incurred in the computation of $|D|/b$. From equation (8) we get

$$D = (q.x - p.x)(r.y - p.y) - (q.y - p.y)(r.x - p.x) \tag{9}$$

If we compute this formula using floating-point operations, we obtain an approximate result D^* satisfying

$$\begin{aligned} D^* &= ((q.x - p.x)(r.y - p.y)(1 + 3\lambda_1) \\ &\quad -(q.y - p.y)(r.x - p.x)(1 + 3\lambda_2))(1 + \lambda_3) \\ &= D(1 + \lambda_3) + 3\lambda_4 M \end{aligned}$$

where $M = |q.x - p.x||r.y - p.y| + |q.y - p.y||r.x - p.x|$, and $|\lambda_i| \leq u$. We can compute the longest side b by the obvious formula

$$b = \max\{\|p, q\|, \|q, r\|, \|r, p\|\}$$

where $\|\ \|$ is the familiar square root formula (4). As we discussed in section 2.2, if we assume that the square root operation is accurate to the machine precision, then the the computed value b^* satisfies $b^* = b(1 + 3\lambda_6)$. Therefore, the computed value h^* for the triangle's height $h = D/b$ satisfies

$$\begin{aligned} h^* &= \frac{D(1 + \lambda_3) + 3\lambda_5 M}{b(1 + 3\lambda_7)}(1 + \lambda_8) \\ &= \frac{D}{b}(1 + 4\lambda_9) + 3\lambda_{10}\frac{M}{b} \end{aligned}$$

In fact, since $|D| \leq K$, we can further simplify this to $h^* = h + 7\lambda_{11}M/b$. Therefore, we conclude that

$$h \in h^* \pm 7uM^*/b^* \tag{10}$$

Note that M can be as large as b^2, and that uM/b can easily be greater then h, which means that the *relative* error of h^* can be arbitrarily large. Formula (10) says that the *absolute* error of h^* is small compared to the distances between the three points. If this inaccuracy is a problem, one can make the relative error small by computing D with extended precision; more specifically, with at least $2m + 1$ fraction bits, where m is the number of fraction bits in single-precision floating-point numbers.

Even though *Pos* and *Neg* are not the exact opposites of each other, the asymmetric representation we use for the outcome of epsilon-boxes allows us to state the following result: if the pair $a = (a.lo, a.hi)$ is a valid outcome for Pos(T), then the pair $-a = (-a.hi, -a.lo)$ is a valid outcome of Neg(p, q, r). Therefore the procedures Pos and Neg can share most of their code, and if we evaluate Pos(T) we do not need to evaluate Neg(T).

2.4 Betweenness

We say that a point z is *between* two other points z and q if z lies on the closed segment pq. We denote this fact by $Between(z, pq)$. It is easy to see that $\varepsilon\text{-}Between(z, pq)$ if and only if the distance from z to the segment pq is at most 2ε; or, equivalently, if and only if the ε-disk centered at z intersects the ε-stroke of p and q. See figure 7.
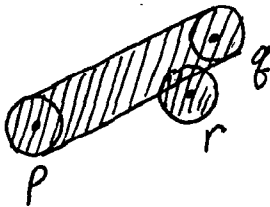

Figure 7. The $\varepsilon\text{-}Between$ test.

In order to compute the distance from z to the segment pq we must find the projection z' of z on the line l through p and q, and then return either $\frac{1}{2}\|z, p\|$, $\frac{1}{2}\|z, l\|$, or $\frac{1}{2}\|z, q\|$, depending on whether z' lies before p, between p and q, or after q, respectively.

We can check the position of z' by checking the signs of the dot products $\alpha = (z - p) \cdot (q - p)$ and $\beta = (z - q) \cdot (p - q)$. If $\alpha < 0$, then z' lies before p; if $\beta < 0$, then z' lies after q; otherwise z' lies between p and q. If we compute these dot products with single-precision floating point arithmetic, we must take into account the attendant rounding errors. An analysis similar to the one we did for the *Pos* predicate shows that the computed value α^* of α satisfies $\alpha \in \alpha^* \pm 4uK_p^*$, where K_p^* is the computed value of the quantity

$$K_p = |(z.x - p.x)(q.x - p.x)| + |(z.y - p.y)(q.y - p.y)|$$

A similar result holds for β. Therefore, these computations give us two intervals a and b, which may or may not include 0.

We obtain uncertainty intervals d_p and d_q for the distances $\frac{1}{2}\|z, p\|$ and $\frac{1}{2}\|z, q\|$ by calling the Coincident box. As for $\frac{1}{2}\|z, l\|$, its value is twice the area of the triangle zpq, divided by the length of pq; that is $|D(z, p, q)|/\|p, q\|$, where D is the determinant defined by equation (8). Therefore the analysis we did for the Pos subroutine applies here too (except that b is the length of pq, instead of the longest side), and we get the same uncertainty interval $d_l = h^* \pm 7uM^*/b^*$. (If $b^* = 0$ then we can conclude that $b = 0$, in which case we let d_l be the pair $(0, 0)$.)

We now have to put all this information together and deduce from it an uncertainty interval e for the distance from z to the segment pq. The algorithm is relatively straightforward:

1. $e \leftarrow \emptyset$
2. if $a.lo \leq 0 \wedge b.hi \geq 0$ then $e \leftarrow e \sqcup d_p$ fi
3. if $b.lo \leq 0 \wedge a.hi \geq 0$ then $e \leftarrow e \sqcup d_q$ fi
4. if $a.hi > 0 \wedge b.hi > 0$ then $e \leftarrow e \sqcup d_l$ fi
5. return e

Step 1 initializes e to the dummy interval $\emptyset$ that is a neutral element of $\sqcup$. Step 2 checks whether the projection z' could lie before p, in which case the uncertainty interval e is set to the range of possible values of $\frac{1}{2}\|z, p\|$. Step 3 does the same thing for q. Finally, step 5 checks whether z' could lie strictly between p and q, in which case it sets to e the range of $\frac{1}{2}\|z, l\|$. Note that if the algorithm cannot decide between any two cases, it will simply merge the corresponding uncertainty intervals.

3 Approximate Point Inclusion

3.1 Point Inclusion in a Triangle

We now show how these primitives can be combined into a more complex algorithm. We consider the problem of testing whether a point z is inside a triangle $T = (t_0, t_1, t_2)$, which we denote by the predicate $Inside(z, T)$. We define "inside" to include the triangle's boundary.

The ε-version of this predicate is true if the point z and the triangle T can be perturbed by at most ε so that $Inside(z, T)$ becomes true. Here, the distance between two triangles is defined as the maximum distance between a vertex of one triangle and the corresponding vertex of the other. Thus, $\varepsilon\text{-}Inside(z, T)$ is true if and only if Z it is within 2ε of some point of T.

Assume for the moment that the vertices t_0, t_1, t_2 of T are known to be non-collinear and positively (counterclockwise) oriented. Then ordinary geometry tells us

213

that z is inside T if and only if the triangles zt_0t_1, zt_1t_2, and zt_2t_0 are all positively oriented (or flat). Moreover, if z is inside T, any perturbation that puts z outside the triangle must reverse the orientation of one (or more) of these triangles. We conclude that

Lemma 3 *If a point z is inside a positively oriented triangle $T = (t_0, t_1, t_2)$, then for any ε (positive or negative),*

$$\varepsilon\text{-}Inside(z, T) \quad \Leftrightarrow \quad \bigwedge_{i=0}^{2} \varepsilon\text{-}Pos(z, t_i, t_{i+1}) \qquad (11)$$

If T is a negatively oriented triangle, formula (11) holds with *Pos* replaced by *Neg*.

Unfortunately, equation (11) does not hold if z is outside T. as figure 8 shows, the triangles zt_it_{i+1} can be all ε-positive, and yet z may be arbitrarily far from T. The explanation is that it is indeed possible to make either one of the triangles zt_0t_1 and zt_1t_2 positive with an ε-perturbation, but there is no such perturbation that makes them *both* positive at the same time.
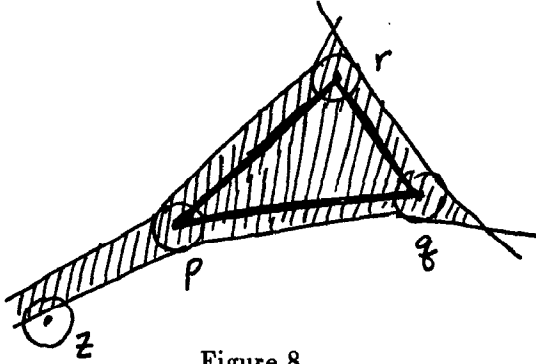

Figure 8.

Therefore, when z is outside T, the information returned by the **Pos** boxes is not sufficient. To handle this case, we introduce the predicate $Boundary(z, T)$, that tests whether the point z lies on the boundary of the triangle T. This predicate has a simple expression:

$$Boundary(z, T) \quad \Leftrightarrow \quad \bigvee_{i=0}^{2} Between(z, t_i, t_{i+1}). \qquad (12)$$

Recall that for $\varepsilon \geq 0$ the ε- qualifier distributes over $\vee$. Therfore, we can write

$$\varepsilon\text{-}Boundary(z, T) \quad \Leftrightarrow \quad \bigvee_{i=0}^{2} \varepsilon\text{-}Between(z, t_i, t_{i+1}). \qquad (13)$$

In other words, we are ε away from the boundary if and only if we are ε-away from some side. This identity holds also for $\varepsilon < 0$, since in that case both sides are **false**. The implementation of **Boundary** follows immediately from equation (13) and the $\vee$ construction described in

section 1.3: evaluate **Between**(z, p, q) for all edges p, q of T, and return the **min** of the resulting pairs. In fact, it is easy to check that this algorithm works even if T is replaced by an arbitrary polygon.

How does the *Boundary* predicate help us? The answer is given by the folowing trivial properties:

Lemma 4 *If a point z is outside a triangle T, then for all $\varepsilon \geq 0$*

$$\varepsilon\text{-}Inside(z, T) \quad \Leftrightarrow \quad \varepsilon\text{-}Boundary(z, T) \qquad (14)$$

Lemma 5 *If a point z is inside a triangle T, then for all $\varepsilon < 0$,*

$$\varepsilon\text{-}Inside(z, T) \quad \Leftrightarrow \quad \text{not}\,(-\varepsilon)\text{-}Boundary(z, T) \qquad (15)$$

With these results, we can implement the *Inside* tests for a point and a triangle by the procedure InTriangle below:

1. $a_0 \leftarrow \text{Pos}(z, t_0, t_1)$
2. $a_1 \leftarrow \text{Pos}(z, t_1, t_2)$
3. $a_2 \leftarrow \text{Pos}(z, t_2, t_0)$
4. $s_p \leftarrow \max\{a_0, a_1, a_2\}$
5. if $s_p.hi < 0$ then return s_p
6. $s_n \leftarrow \max\{-a_0, -a_1, -a_2\}$
7. if $s_n.hi < 0$ then return s_n
8. $s \leftarrow \text{Pos}(t_0, t_1, t_2)$
9. if $s.hi \leq 0$ then $r \leftarrow s_p$
10. elsif $s.lo \geq 0$ then $r \leftarrow s_n$
11. else $r \leftarrow s_n \sqcup s_p$
12. fi
13. $b \leftarrow \text{Boundary}(z, T)$
14. return $(-b \sqcap (r.lo \_ \infty)) \sqcup b$

In this algorithm, steps 1–4 compute the perturbation s_p needed to independently make each of the triangles zt_0t_1, zt_1t_2, and zt_2t_0 positive or collinear. If this perturbation is definitely negative, meaning all three triangles are positively oriented, then we can deduce that the triangle T itself is positively oriented, and that z is inside it. Lemma 3 then authorizes us to return s_p itself in step 5 as the uncertainty interval of **Pos**. Steps 6–7 perform the symmetric test for the case when T is negatively oriented.

If these two tests fail, the algorithm does some additional work in order to find the minimum perturbation r that makes each of the three triangles have (idependently) the same orientation as T. First the algorithm tries to determine the orientation of T by evaluating $\text{Pos}(t_0, t_1, t_2)$. If the result of this test has a definite sign, then r is taken to be either s_p or s_n. If the test is inconclusive, then r is set to the join of the two intervals. Note that in any case we will end up with $r.hi \geq 0$.

As we observed before, the interval r is still not the answer we want, since r was obtained by considering independent perturbations to each of the three edges, and those perturbations may not be simultaneously realizable. The only useful information contained in r is its lower endpoint $r.lo$, provided it is less than zero: if $-\varepsilon < r.lo < 0$, that is, if we cannot change the orientation of one of any of the three triangles with any ε-perturbation, then we cannot move z out of T with any ε-perturbation, which means z is $(-\varepsilon)$-inside T; and conversely. On the other hand, if $r.lo \geq 0$, we know only that z is outside, but we can't tell by how much. In other words, at step 13 the uncertainty interval of Pos is $(r.lo \_ \infty) \sqcup (0 \_ infty)$.

In order to reduce this uncertainty to a useful level, we evaluate $b = $ Boundary(z, T), the minimum perturbation needed to put z on the boundary of t. The resulting interval b is always non-negative; moreover, by lemmas 4 and 5, the result we want is either $-b$ or $+b$, depending on whether z is inside or outside T. So, the result we want is the intersection of those two intervals with $(r.lo \_ \infty) \sqcup (0 \_ infty)$, which simplifies to $(\text{-}b \sqcap (r.lo \_ \infty)) \sqcup b$, as returned in step 14.

Note that the width of the interval returned by this algorithm is at most twice the size of the widest interval returned by the Pos and Between boxes.

As usual, there is here a tradeoff between speed and accuracy. One could reduce the width of the uncertainty interval returned by InTriangle(z, T) by performing more elaborate tests, by using more accurate primitives, by combining their results in a more sophisticated fashion, or by implementing the *InTriangle* procedure as a primitive, as we did with *Pos*.

3.2 Point Inclusion in a Convex Polygon

Let's now consider the more general predicate that tests whether a point z lies inside a convex polygon $P = (p_0, p_1, \ldots, p_{n-1})$. One might think that the implementation InConvex of this predicate is a trivial generalization of InTriangle, but that is not the case. While lemmas 4 and 5 generalize nicely to arbitrary convex polygons, lemma 3 does not.
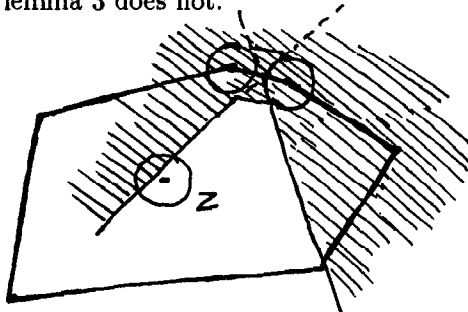

Figure 8.

As figure 8 shows, a point can be well inside the polygon P, even if it is possible to reverse the orientation of some triangle zp_ip_{i+1} by an arbitrarily small perturbation to those three points. Therefore, for InConvex must use a different approach, as follows. When z is inside P, we will estimate the degree of "insideness" from the output of the **Boundary** box alone, using lemma 5. When z is outside P, we will estimate its degree of "outsideness" with several calls to InTriangle, using the following obvious property:

Lemma 6 *Let* $T_1, T_2, \ldots, T_m$ *be a set of triangles whose union is the set P. If a point z is outside P, then for all $\varepsilon \geq 0$,*

$$\varepsilon\text{-}Inside(z, P) \quad \Leftrightarrow \quad \bigvee_{i=1}^{m} \varepsilon\text{-}Inside(z, T_i)$$

Note that lemma 6 cannot be extended to $\varepsilon < 0$, because z can be near an edge of some triangle T_i and still be arbitrarily far from the boundary of P. Nevertheless, lemma 6 allows us to write an epsilon-box InPolygon$(z, P, T_1, \ldots, T_m)$ that implements the *Inside* test for a planar figure P, given a collection of triangles $T_1, T_2, \ldots, T_m$ whose union is P:

1. $r \leftarrow \min \{$InTriangle$(z, T_i) : 1 \leq i \leq m\}$
2. if $r.lo \geq 0$ then return r fi
3. $r.lo \leftarrow -\infty$
4. $b \leftarrow$ Boundary(z, P)
5. return $(r \sqcap b) \sqcup (r \sqcap -b)$

Step 1 computes the perturbation r necessary to put z inside one of the triangles T_i. If $r.lo \geq 0$, it means z is outside P, and by lemma 6 we can return r itself as the outcome. If $r.lo < 0$, lemma 6 tells us nothing: the uncertainty interval for *Inside*(z, P) is not r but the whole interval $(-\infty, r.hi)$. In order to reduce this uncertainty, we look at the perturbation b needed to put z on the boundary of P (step 4). According to lemmas 4 and 5, the InPolygon box can safely return $r \sqcap b$ is z is outside P, and $r \sqcap -b$ if z is inside P; therefore, it can always return return the join of these two intervals (step 5).

Given this algorithm, the InConvex box that implements the *Inside* test for convex polygons is trivial: it suffices to note that a convex polygon $P = (p_0, p_1, \ldots, p_{n-1})$ is the union of the triangles $P_0 p_i p_{i+1}$ for $1 \leq i \leq n - 2$.

Note that, as in the the case of InTriangle, the width of the uncertainty interval returned by InPolygon or InConvex is at most twice that of the widest interval returned by any of the primitive epsilon-boxes called by them.

Some applications of the `InConvex` box may not require all the information that it returns. In such cases the algorithm can often be made simpler and faster, at the cost of returning a much wider uncertainty interval. For example, if we only want to know whether we are inside or outside P, but we don't care by how much, we can omit steps 4 and 5. In that case, when z is inside P or close enough to it the algorithm will return intervals of the form $(-\infty, \varepsilon)$.

4 Approximate Convexity

The `InConvex` test of the previous section assumes the polygon is convex in the ordinary sense. By itself, such a test isn't very useful, since in many applications we cannot always ensure that the polygons are strictly convex, because they can be the result of previous approximate computations, and the rounding errors incurred may have introduced slightly concave corners, self-intersections, and other similar defects. Worse still, if we are allowed to use only approximate primitives like the ones we described, we will not even be able to *check* whether a given polygon is convex. Therefore, we must learn how to handle polygons that are only "approximately convex." In order to do this, we must first define more carefully the meaning of "inside," and prove a few results about closed curves in general.

4.1 Closed Curves

We define a *closed curve* to be a continuous function from the unit circle S_1 into the plane, and we consider two curves to be the same if they differ only by a simple reparametrization, that is, by a homeomorphism of S_1 to itself. We say that a point z is *inside* such a curve C if the curve passes through z, or if its winding number around z is non-zero [4]. Note that the curve need not be simple. We denote this fact by $Inside(z, C)$.

We define the distance between two curves C, C' as the smallest value of $\max\{\|C(\varphi(t)), C'(t)\| : t \in S_1\}$, when φ ranges over all reparametrizations of C. In spite of this complicated metric, the meaning of $\varepsilon\text{-}Inside(z, C)$ is quite simple: if $\varepsilon \geq 0$, this predicate means that z is either inside C or at most $2|\varepsilon|$ away from C; and if $\varepsilon < 0$, it means that z is inside C but at least $|\varepsilon|$ away from C.

Lemma 7 *Let P and Q be two closed curves with the property that $\|P(t), Q(t)\| \leq \varepsilon$ for all t on the unit circle. Then any point that is inside P is $(\varepsilon/2)$-inside Q, and vice versa.*

Proof: Let x be a point inside P. If x is inside Q or x is on the curve P, then we are done. Let's assume that

x is inside P but not inside Q. Consider the rays from x to $P(t)$ and from x to $Q(t)$. As t goes once around the unit circle, the first ray makes a non-zero number of full turns, while the second makes zero full turns. Since the curves are continuous, there must exist a value of t for which the angle between the two rays is π. At that moment, x is on the segment connecting $P(t)$ and $Q(t)$, whose length is at most ε, by hypothesis. $\square$

This result is not as trivial as it may sound. For instance, it is not enough to merely require that every point of the curve P be within ε of Q and vice versa.

4.2 Polygons

The next lemma states an elementary property of Minkowski metrics is useful when applying lemma 7 to polygons:

Lemma 8 *Let $\|,\|$ be any Minkowski metric. Then for any four points p, q, p', q' such that $\|p, p'\| \leq \varepsilon$ and $\|q, q'\| \leq \varepsilon$, and for any $\alpha \in [0 _- 1]$, we have*

$$\|(1 - \alpha)p + \alpha q, (1 - \alpha)p' + \alpha q'\| \leq \varepsilon$$

The next result is a trivial corollary of lemmas 7 and 8:

Lemma 9 *If the polygons $P = (p_0, p_1, \ldots, p_{n-1})$ and $Q = (q_0, q_1, \ldots, q_{n-1})$ are such that $\|p_i, q_i\| \leq 2\varepsilon$ for all i, then any point inside P is ε-inside Q (and vice versa).*

4.3 Inclusion in an ε-Convex Polygon

Now let's try to extend the `InConvex` algorithm to polygons that are not necessarily convex, but merely ε-convex. According to the general definition, a polygon P is ε-*convex* if there exists a convex polygon P' that is at most ε away from P; or, equivalently, if we can perturb the vertices of P by at most ε in such a way that the result is a convex polygon. We will use the following result:

Lemma 10 *Let $P = (p_0, p_1, \ldots, p_{n-1})$ be an ε-convex polygon, with $\varepsilon > 0$, and z a point of the plane. Let P^* be the union of the triangles $p_0 p_i p_{i+1}$ for $1 \leq i \leq n - 2$. Then*

$$Inside(z, P) \Rightarrow Inside(z, P^*) \Rightarrow \varepsilon\text{-}Inside(z, P)$$

Proof: Let's prove first that $Inside(z, P) \Rightarrow Inside(z, P^*)$. If the point z is inside the polygon P, then any ray starting from z should meet at least one edge $p_i p_{i+1}$ of P for $1 \leq i \leq n - 2$. (Assuming each edge includes its endpoints.) In particular, this should happen for the ray out of z that is directed away from p_0.

216

This ray proves that z is inside the triangle $p_0 p_i p_{i+1}$, and hence inside P^*.

Now let's prove that $Inside(z, P^*) \Rightarrow \varepsilon\text{-}Inside(z, P)$. By definition there exists a convex polygon P' such that $\|p_i, p_i'\| \leq \varepsilon$ for all i. By corollary 9, every point inside P' is $(\varepsilon/2)$-inside P. By the same argument, any point of triangle $p_0 p_i p_{i+1}$ is $(\varepsilon/2)$-inside the corresponding triangle $p_0' p_i' p_{i+1}'$; therefore, by lemma 6, any point in P^* is $(\varepsilon/2)$-inside P'. By the triangle inequality (lemma 2), then, any point of P^* is ε-inside P. $\square$

This lemma gives us an algorithm $\texttt{InEpsConvex}(z, P, \varepsilon)$ that tests whether a point z is δ-Inside a polygon P, assuming that P is ε-convex. The algorithm is a slight modification of the $\texttt{InConvex}$ box that we gave in section 3.2:

1. $r \leftarrow \min\{\texttt{InTriangle}(z, p_0 p_i p_{i+1}) : 1 \leq i \leq n - 2\}$
2. $r.hi \leftarrow r.hi + \varepsilon$
3. if $r.lo \geq 0$ then return r fi
4. $r.lo \leftarrow -\infty$
5. $b \leftarrow \texttt{Boundary}(z, P)$
6. return $(r \sqcap b) \sqcup (r \sqcap -b)$

After step 1 we know that z is δ-inside some of the triangles $p_0 p_i p_{i+1}$, for $\delta \geq r.hi$, and not δ-inside any of them for $\delta < r.lo$. Step 2 adjusts $r.hi$ to account for the fact that the union of those triangles may include points that are up to 2ε away from P. This step is justified by the triangle inequality (lemma 2). The rest of the algorithm is shown correct by the same arguments used for $\texttt{InPolygon}$ in section 3.2.

5   Conclusions

The Epsilon Geometry framework we described in this paper allows us to build robust algorithms using imprecise computations. Because our framework allows us to use ordinary fixed- or floating-point arithmetic and substitute simpler approximations for hard-to-compute formulas, we believe it has great practical potential.

An important feature of the Epsilon Geometry approach is its flexibility, in that it gives the designer of a geometric algorithms great freedom to choose between accuracy, efficiency and simplicity. Our approach allows us to combine primitive epsilon-boxes into more complex algorithms, independently of the number representation and machine precision used inside each primitive box.

We have barely started to explore the application of this framework to classical computer geometry problems. If the examples we give in this paper are too elementary, it is only because we haven't had time yet to consider more complex ones.

Acknowledgements

References

[1] Herbert Edelsbrunner and Ernst Peter Mücke, "A technique to cope with degenerate cases in geometric algorithms." Proc. 4th Annual ACM Symp. on Computational Geometry (1988), 118–133.

[2] Daniel H. Greene and F. Frances Yao, "Finite-resolution computational geometry." Proc. 27th IEEE Symp. on the Foundations of Computer Science (1986), 143–152.

[3] Christoph M. Hoffman, John E. Hopcroft, and Michael S. Karasick, "Towards implementing robust geometric computations." Proc. 4th Annual ACM Symp. on Computational Geometry (1988), 106–117.

[4] Leo Guibas, Lyle Ramshaw, and Jorge Stolfi, "A kinetic framework for computational geometry." Proc. 24th IEEE Symp. on Foundations of Computer Science (October 1983), 100–111.

[5] Donald E. Knuth, "The art of computer programming, vol. 2: Seminumerical algorithms" (second edition), section 4.2.2. Addison-Wesley (1981).

[6] Victor J. Milenkovic, "Verifiable implementations of geometric algorithms using finite precision arithmetic." International Workshop on Geometric Reasoning (Oxford, England, July 1986).

[7] S. P. Mudur and P. A. Koparkar, "Interval methods for processing geometric objects." IEEE Computer Graphics and Applications (February 1984), 7–17.

[8] Thomas Ottmann, Gerald Thiemt, and Christian Ullrich, "Numerical stability of geometric algorithms." Proc. 3rd Annual ACM Symp. on Computational Geometry (1987), 119–125.

[9] Chee-Keng Yap, "A geometric consistency theorem for a symbolic perturbation scheme." Proc. 4th Annual ACM Symp. on Computational Geometry (1988), 134–142.