Lecture 4
Exact Geometric Computation

We now expound the principles and assumptions of EGC. The name suggests that we want the geometry to be computed exactly. Of course, if our arithmetic is exact, then the geometry would be exact. But the converse is not true, and this is the reason for the power of EGC.

§1. Algebraic Numbers

What do we mean by "exact arithmetic"? In the case of rational numbers, this is clear enough. For numbers such as $\sqrt{2}$, it is often noted that there is "no finitely representable exact answer". But this is only true if we use some explicit (positional) number representation. It turns out that $\sqrt{2}$, and algebraic numbers more generally, do have precise albeit non-positional representations. These representations are quite usable since there are algorithms for performing arithmetic or other desired operations on them.

Let us describe one representation for algebraic numbers. By definition, an algebraic number α is the root of an integer polynomial $P(x) = \sum_{i=0}^{n} a_i x_i$. For instance $\sqrt{2}$ is the root of $P(x) = x^2 - 2$.

This point is important: we will perform "exact arithmetic" in the sense that some exact representation of the number will be retained, but we do not necessarily want to compute too precise a positional approximation for thees numbers. The approximation will be carried out as needed – this is the lazy arithmetic idea. But we go one step further: we drive this precision from the top in a way that is optimal in a suitable sense.

§2. Exact Geometry

Jiaxu Yu, in his 1990 Purdue thesis [9] on exact arithmetic for for solid modeling, concluded that the approach is not feasible in the near future. However, there is another approach which can avoid this problem by asking for the "exact geometry" instead of "exact arithmetic". What does this mean? Here is the line of reasoning that gives interprets the idea of "exact geometric computation".

- All conventional[1] algorithms, when applied to an input instance I, gives rise to a sequence of "primitive steps". These primitive steps are of two types: **assignment step** and **branching step**. In an assignment step has the form $x \leftarrow f(x_1, \ldots, x_n)$ where x and x_i's are programming variables and f is a primitive function in the programming language. In the branching steps, some predicate is evaluated, and there are 2 or more possible outcomes. Typically, the predicate is a polynomial $p(x_1, \ldots, x_n)$ where each x_i is a variable and the particular **branching outcome** depends on the sign of the polynomial evaluates at the values of the x_i's.

  If the algorithm is A and I is the input instance, we obtain a corresponding finite tree $T_A(I)$ that may be called a **tree program**. See figure 1.

  Note that in simple cases, we can often view the input instance I as a sequence of variables $I = (z_1, \ldots, z_m)$.

- Each input instance I for a geometric algorithm A corresponds to some underlying combinatorial structure, either to be computed or else used as the implicit search space for the problem. A simple but key observation is this: each distinct path in the tree program $T_A(I)$ corresponds to the computation for

[1] There are other models of programs such as found in constraint programming languages or in parallel programming models. In such models, each "step" must be further reduced to primitive steps of the conventional kind for our discussion to proceed.

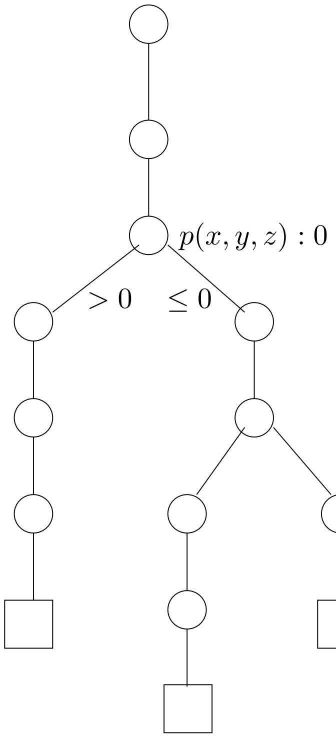

Figure 1: An tree program.

a distinct combinatorial structure $S = S(I)$. *Exact geometry geometry means that we want to operate on this correct structure* $S(I)$. In an inexact geometric algorithm, we often operate on some structure $S' = S(I')$ where I' is some perturbation of I. Catastrophic errors occur when S' is **inconsistent**, *i.e.*, $S' \neq S(I')$ for any input instance I'.

Advantages of EGC. First and foremost, we retain Euclidean geometry as our mathematical basis. This advantage cannot be under estimated: all our intuitions about geometry derives from Euclidean geometry! We design algorithms and reason about them based on this intuition.

Second, we can now exploit approximate arithmetic: as noted in Lecture 1 (Section 4), Big Float would be a real advantage over Big Rational. This fact was not fully appreciated in the beginning of this field, when it was automatically assumed that EGC entails exact arithmetic. Unfortunately, this "naive approach" to EGC is too inefficient for most problems [8].

Third, we can provide a systematic solution to nonrobustness problems based on EGC. In practical terms this means that we can provide a library to solve the nonrobustness problem. That is because we do not need to re-design our theoretical algorithms (which assumes exact real arithmetic) just to achieve robustness. We will demonstrate this with our `Core Library`. In contrast, inventing finite resolution geometries to achieve nonrobustness tend to lead to problem-specific solutions (in any case, not easily extendable).

Forth, the EGC approach reveals one of the difficulties of attempting to do rounding at the same time we perform geometric computation: these are two separate concerns which should be handled by separate algorithms. In other words, if we must round (and in practice, we often need to), then this should be addressed as a separate issue which we may call **geometric rounding problem**. That is, given a geometric structure D and a precision p, we want to round D into a new structure D' which has precision at most p. Typically, D is computed by some EGC method and may have high numerical precision.

Finally, it will turn out, there are many interesting techniques that open up once we focus on the geometry and not the arithmetic.

Nominal Exactness. A very important point about EGC is that we must assume the input instance I is exact. Some users resist this idea, saying that I is often not meant to be taken literally because there are uncertainty in the input values of I. But is this really a price of EGC? We argue no. If the input is really inexact, there are 2 possibilities: (1) it does not matter if we NORMINALLY treat it as exact. (2) the input is not consisten when taken literally. In the former case, it seems that the EGC approach will generally cause no harm. In the latter case, the user is asking our algorithm to do two things: to clean up the data to ensure consistency, and to solve whatever problem the algorithm is supposed to solve. Unfortunately, data clean-up is a generally difficult problem, not to say that it may also be an ill-defined problem. Therefore, we must insist that the user solve the data cleanup issue by some preprocessing algorithm before calling our EGC algorithm: we can only accept consistent data.

Efficiency Issues. Exact geometry removes some of the efficiency issues of exact arithmetic. But it must be said that efficiency remains a significant challenge. In particular, problems whose primitives have high algebraic degrees (as in geometric modeling problems) is a serious challenge. But we suggest that this is inherent in the problem itself, not specific to EGC. For instance, whenver FP approaches have success with a problem, EGC can easily duplicate this success in a more transparent way. But the converse is not true: EGC can address robustness in many problems that FP approaches are unable to reach.

The Consistency Approach. There is an approach closely related exact geometry: here, we want to compute outputs that is the exact geometry for some perturbed input. This approach was pursued by Hoffmann, Hopcroft and Karasick. Such an output is said to be **consistent**. Unfortunately, this approach may lead to some hard computational problems usually associated with theorem proving. We refer to the survey of Schirra [6] for more information on this approach.

Exercises

Exercise 2.1: □

End Exercises

§3. Fortune's Algorithm

One of the most intensively studied problems in computational geometry is Voronoi diagrams in its many forms. See Sugihara [] and Fortune [2].

Voronoi Diagrams for a Planar Point Set. We first collect some basic facts about Voronoi diagrams for a set S of n points in the plane. Points in S are called **sites**. This diagram, vor(S), is a planar subdivision comprising Voronoi cells, Voronoi edges and Voronoi vertices. There is a cell for each site, and the cell is a polygonal region comprising all points on the plane closer to the site than to other sites. For any point p and set $X \subseteq \mathbb{R}^2$, let $d(p, X) = \inf_{q \in X} d(p, q)$ where $d(p, q)$ denotes Euclidean distance. Let $\mathrm{circle}_S(p)$ denote the circle centered at p with radius $d(p, S)$.

PROPERTY: A point q is a Voronoi vertex of $\mathrm{vor}(S)$ iff $\mathrm{circle}_S(q)$ touches 3 or more sites in S. A point q lies on a Voronoi edge of $\mathrm{vor}(S)$ iff $\mathrm{circle}_S(q)$ touches 2 sites in S.

We can compute the Voronoi cell for a single site in $O(n \log n)$ – this amounts to computing the intersection of $n - 1$ half-spaces, a problem known to be equivalent to convex hulls. This gives an $O(n^2 \log n)$ algorithms for $\mathrm{vor}(S)$.

Using a divide-and-conquer approach, we can compute $\mathrm{vor}(S)$ in $O(n \log n)$ time: we divide the set into two equal halves S_L, S_R by a vertical line. Recursively compute the $\mathrm{vor}(S_L)$ and $\mathrm{vor}(S_R)$. It remains to merge these 2 Voronoi diagrams into $\mathrm{vor}(S)$. It turns out that there is a "separator curve" that is polygonal that separates the plane into a left and a right half. Moreover, $\mathrm{vor}(S_L)$ restricted to the left half, union with $\mathrm{vor}(S_R)$ restricted to the right half, yields $\mathrm{vor}(S)$. This separator curve can be computed in $O(n)$ time, leading to an $O(n \log n)$ complexity. This bound is also optimal in reasonable models of computation.

Sweepline Approach. Fortune [3] gave another $O(n \log n)$ approach based on the plane sweep paradigm. Let H be a horizontal line that sweeps from top to bottom. Let H^+ denote the upper half-space above H. Intuitively, we would like to maintain a representation of "$H + \cap \mathrm{vor}(S)$" as H sweeps. Moreover, we want this representation to depend only upon the set of sites in H^+. This idea by itself is wrong since the Voronoi diagram above H can be affected by the presence of sites in H^-. Instead, we introduce the concept of a **beachline** which is a front separating the sites in $H^+ \cap S$ from H.

Let $B = B(H, S)$ denote the beachline and B^+ denote the region above B. The set of points on B are easy to characterize: $q \in B$ iff $d(q, H) = d(q, S^+)$. It is not hard to see that B is a curve that is piecewise parabolic. Each parabolic part of B is called an **arc** and the endpoints of these arcs are called **breakpoints**. Then, $B^+ \cap \mathrm{vor}(S)$ is indeed unaffected by sites below H. Here is a picture of a beachline:

FIGURE: Beachline

Beachline: arcs and breakpoints.

Events. There are two kinds of events in our sweepline algorithm:
(1) Site Event: when H encounters a new site. This caused a new arc to appear in B.
(2) Circle Event: when an arc disappears. This corresponds to H reaching the bottom of a Voronoi circle.

Data structure. There are 3 data structures we need.

(1) DCEL: to represent the subdivision To handle infinite and semi-infinite edges of the subdivision, it is best to work in the projective plane $\mathbb{P}^2(\mathbb{R})$. A **projective coordinate** is a triple $(x, y, w) \in \mathbb{R}^2 \setminus \{(0, 0, 0)\}$. Two projective coordinates if they have the form (x, y, w) and (cx, cy, cw) for some $c \neq 0$. A **projective point** p is then the equivalence class $(x : y : w)$ of a projective coordinate (x, y, w). We say $(x : y : w)$ is **finite** if $w \neq 0$, else **infinite**. Intuitively, each infinite point $(x : y : 0)$ represents a direction x/y. All lines with slope x/y passes through this point. Note that the Euclidean plane $\mathbb{R}^2$ can be identified with the set of finite projective points. A projective line is the set of projective points whose coordinates satisfy an equation of the form $ax + by + cw = 0$. So $w = 0$ defines the "line at infinity". A model of the line at

infinity is a circle of infinite radius. Topologically, this cirlce is rather unusual: opposite points of this circle are identified. Actually, every line in the projective plane has the same topology, so the unusual nature of the line at infinity is simply a consequence of our model (its particular embedding in the Euclidean plane).

MODEL OF LINE AT INFINITY.

Suppose B comprises of just one parabolic arc (this is the situation just after the site event). We will represent the subdivision as a single cell bounded by the parabola and the line at infinity. Actually, there is an ambiguity here as there are 2 choices but we specify the region containing the first site, of course.

(2) Balanced Binary Tree T: to represent the beachline.

(3) Priority Queue Q: to order the events for processing.

THEOREM: the algorithm takes $O(n \log n)$ time.

Note: Fortune's algorithm can be adapted for computing the Voronoi diagram of a set disjoint of line segments, a problem first studied by Drysdale. An $O(n \log n)$ method for this problem was first proposed by Kirkpatrick in 1979 [5], and a provably correct solution is from Yap [7].

Exercises

Exercise 3.1: Provide the following details for Fortune's algorithm:
(i) How to initialize the subdivision structure that represents the partially constructed Voronoi diagram.
(ii) How to handle degeneracies – when two or more events are coincident. □

Exercise 3.2: Consider the following input set of point for Fortune's algorithm: $p_0 :: (0, 0)$, $p_1 :: (5, 0)$, $p_2 :: (5, 1)$, $p_3 :: (3, 1)$, $p_4 :: (3, 4)$, $p_5 :: (0, 4)$. How does your above method for handling degeneracies treat this input set? □

Exercise 3.3:
(i) Adapt Fortune's algorithm to compute the Delaunay triangulation of a set of points.
(ii) Give a "cheap implementation" of your algorithm in which you replace the balanced binary tree and the priority queue by sorted lists. So, we give up the $O(n \log n)$ complexity in exchange for a possibly smaller constant factor in an $O(n^2)$ time algorithm. □

End Exercises

§4. Primitives

We now address the problem of EGC in Fortune's algorithm, namely, what does it take to compute the primitives correctly? First we must understand the primitives needed for the algorithm.

Circumcircle of three points. Let p, q, r be 3 non-collinear points, and let c be the circumcenter of the triangle (p, q, r). We have the defining equations

$$\|p - c\|^2 = \|q - c\|^2 = \|r - c\|^2 = \rho^2$$

where ρ is the radius of the circumcircle defined by p, q, r. Writing $c = (c_x, c_y)$, $p = (p_x, p_y)$, $p^2 = p_x^2 + p_y^2$, etc, the equation $\|p - c\|^2 = \rho^2$ can be rewritten as

$$2p_x c_x + 2p_y c_y + (\rho^2 - c^2) = p^2.$$

Similar equations for q and r can be given. Putting them into a matrix form,

$$\begin{bmatrix} 2p_x & 2p_y & c^2-\rho^2 \\ 2q_x & 2q_y & c^2-\rho^2 \\ 2r_x & 2r_y & c^2-\rho^2 \end{bmatrix} \begin{pmatrix} c_x \\ c_y \\ 1 \end{pmatrix} = \begin{pmatrix} p^2 \\ q^2 \\ r^2 \end{pmatrix}$$

Let M be the matrix

$$M := \begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}$$

and let $\Delta = \det(M)$, then Kramer's rule (after some manipulation) gives

$$c_x = \frac{\det \begin{bmatrix} p^2 & p_y & 1 \\ q^2 & q_y & 1 \\ r^2 & r_y & 1 \end{bmatrix}}{2\Delta}$$

$$c_y = \frac{\det \begin{bmatrix} p_x & p^2 & 1 \\ q_x & q^2 & 1 \\ r_x & r^2 & 1 \end{bmatrix}}{2\Delta}$$

Now, the priority of the circle event defined by p, q, r is given by

$$\begin{aligned} y_c - \rho &= \frac{\det(M_x)}{2\Delta} - \sqrt{(p_x - c_x)^2 + (p_y - c_y)^2} \\ &= \frac{\det(M_x) + \sqrt{\cdots}}{2\Delta} \\ &= \frac{a + \sqrt{b}}{d} \end{aligned}$$

where a, b, d are suitable polynomials of the coordinates of p, q, r. Assuming each input coordinate is an L-bit integer, then a, b, d are $3L, 6L, 2L$ bits integers, respectively. [REMARK: here, we say "$3L$" bits even though it is really $3L + O(1)$ bits.] Thus, Fortune's algorithm requires comparisons of the form

$$\frac{a + \sqrt{b}}{d} : \frac{a' + \sqrt{b'}}{d} \tag{1}$$

which must be performed exactly, according to the dictates of EGC. Note: There are also comparisons between circle event priorities and a site event priorities, but these have lower complexity than the ones we focus on.

Reduction to integer arithmetic. It is well-known comparisons of radical expressions such as (1) can be reduced to integer arithmetic and integer comparisons. It is clear that we can reduce such questions to a comparison to zero:

$$E : 0 \tag{2}$$

where E is a radical expression. The idea is to use repeated squaring. There are two basic interpretations of the comparison (2): (1) a binary outcome of $E = 0$ or $E \neq 0$. (2) a ternary outcome $E = 0$ or $E > 0$ or $E < 0$. We call these the **2-way** and the **3-way comparison** respectively. In general, there is a considerable gap[2] between (1) and (2). Let us give a third interpretation of (2): we want a binary outcome of either $E \geq 0$ or $E < 0$. Call this a "2.5-way comparison" (for lack of a better term). Clearly, a 3-way comparison can be reduced to two 2.5-way comparisons.

In the literature, it is often the 2-way comparison of radical expressions that is assumed. But in our present application, we need the 3-way comparison. It was first pointed out in [1] that this is considerably more involved (at least from a programming viewpoint). To see why this is so, let us consider a 2.5-way comparison of the form:

$$A + B\sqrt{C} : 0 \tag{3}$$

Thus $A + B\sqrt{C} \geq 0$ if and only if one of the following three conjuncts is true:

$$(A \geq 0, B \geq 0), \qquad (A \geq 0, B \leq 0, A^2 - B^2C \geq 0), \qquad (A \leq 0, B \geq 0, A^2 - B^2C \leq 0).$$

If A, B, C are radical expressions, we must recursively perform suitable 2.5-way comparisons on the expressions $A, B, A^2 - B^2C$ (and their negations).

In the comparison (1) above, we transform the comparison to $d'(a + \sqrt{b}) \geq d(a' + \sqrt{b'})$ or

$$(I) : d'\sqrt{b} \geq d\sqrt{b'} + e$$

for $e = da' - d'a$. Assume that $d, d' \neq 0$, but not necessarily positive. Then (I) is equivalent to the disjunction of the following 3 conjuncts:

$$\begin{aligned}
(II): &\quad [d' \leq 0], [d\sqrt{b'} + e \leq 0], [d'^2 b \leq (d\sqrt{b'} + e)^2] && (4)\\
(III): &\quad [d' \leq 0], [d'^2 b \leq (d\sqrt{b'} + e)^2] && (5)\\
(IV): &\quad [d' \leq 0], [d\sqrt{b} + e \leq 0] && (6)
\end{aligned}$$

These can ultimately be expanded into a Boolean function of the signs of the following 6 expressions:

$$d, \quad d', \quad e, \quad d^2b' - e^2, \quad d'^2b - d^2b' - e^2, \quad 4d^2e^2b' - (d'^2b - d^2b' - e^2)^2.$$

Approximate Arithmetic Approach. Alternatively, we can compute the 2 expressions in (1) to a suitable precision, then compare them. A more direct argument shows $25L$-bits (Mahler's general root separation bound gives $60L$-bits of precision). Mehlhorn has pointed out that $20L$ bits suffices, using a general argument which we will reproduce below. Finally, with Sellen, we show that $15L$ bits suffices for our particular application.

Exercises

Exercise 4.1: Construct a simple example such that Fortune's algorithm when implemented in the straightforward way (using machine double precision arithmetic) will fail. □

Exercise 4.2: Carry out the same analysis of root separation bounds assuming that the inputs are rational numbers with L-bits in both the numerator and denominator. □

[2] In particular, Bloemer and Yap (1989) showed that if E is a sum of square-roots, then the 2-way comparison can be achieved in polynomial time. However, the best current method for performing the corresponding 3-way comparison appears to require exponential time.

Exercise 4.3: (Sellen-Yap) Let $\varepsilon = a + \sqrt{b} - \sqrt{c}$ where a, b, c are all L-bit integers.
(i) Show that $\log_2(1/|\varepsilon|) = 3L/2 + O(1)$. HINT: a is at most $1 + (L/2)$ bits long.
(ii) Show that this is the best possible by an infinite family of examples with $L \to \infty$. HINT: $a = 2^{L/2-1}$, $b = 2^{L-2} - 1$ and $c = 2^{L-1} - 2$. These numbers are the best possible for $L = 6, 8, 10$, as verified by exhaustive computation. □

End Exercises

§5. Misconceptions

In the area of floating point numerical computation, there are various misconceptions and common pitfalls that an un-informed computational scientist ought to be aware of. See for instance Higham's book [4, p. 31,35] which includes numerous references to papers and books (especially one by Acton) about these issues. There are similar misconceptions about exact arithmetic and EGC.

One of the items is "increasing the precision of a computation will increase the precision of the output" [4, Section 1.13, p. 19]. The answers given by numerical analysis is actually not as useful as it could be. While it does serve to caution the unwary, the real reason that such fluctuations occur is that once the computation has lost all significance, then the answer is completely unpredictable. This is the numerical equivalent of saying the logical statement that once a false premise is assumed, then we can prove anything we like. What is more useful is to bound the loss of significance in such examples. [CARRY OUT]

There is another common view that significance arithmetic is too pessimistic. In some sense, this is the only correct because significance arithmetic, like interval arithmetic, must balance between faster and more efficient arithmetic versus more accurate significance arithmetic. On the other hand, keeping wrong bits cannot really help.

References

[1] T. Dubé and C. K. Yap. A basis for implementing exact geometric algorithms (extended abstract), September, 1993. Paper from URL `http://cs.nyu.edu/cs/faculty/yap`.

[2] S. Fortune. Voronoi diagrams and Delaunay triangulations. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, pages 225–265. World Scientific Press, 1995. 2nd edition.

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

[4] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[5] D. G. Kirkpatrick. Efficient computation of continuous skeletons. *IEEE Foundations of Computer Science*, 20:18–27, 1979.

[6] S. Schirra. Robustness and precision issues in geometric computation. Report MPI-I-98-1-004, Max-Planck-Institut für Informatik, Saarbrücken, Germany, Jan 1998. A preliminary version of a chapter that will appear in the *Handbook on Computational Geometry*, edited by J.R. Sack and J. Urrutia. We give a survey on techniques that have been proposed and successfully used to attack robustness and precision problems in the implementation of geometric algorithms.

[7] C. K. Yap. An $O(n \log n)$ algorithm for the Voronoi diagram for a set of simple curve segments. *Discrete and Computational Geometry*, 2:365–394, 1987. Also: **NYU-Courant Institute, Robotics Lab., No. 43**, May 1985.

[8] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.

[9] J. Yu. *Exact arithmetic solid modeling.* Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992. Technical Report No. CSD-TR-92-037.