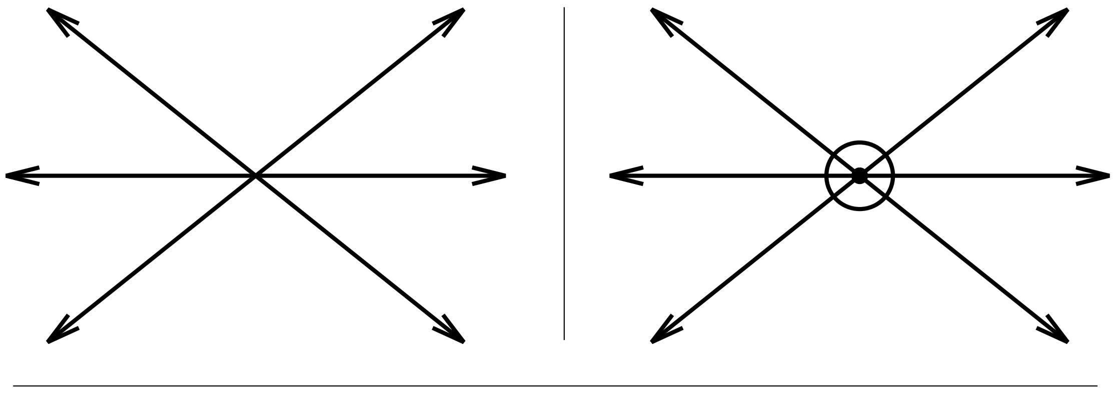

INPUT OUTPUT

17.1 Robust Geometric Primitives

Input description: A point p and line segment l, or two line segments l_1, l_2.

Problem description: Does p lie over, under, or on l? Does l_1 intersect l_2?

Discussion: Implementing basic geometric primitives is a task fraught with peril, even for such simple tasks as returning the intersection point of two lines. It is more complicated than you may think. What should you return if the two lines are parallel, meaning they don't intersect at all? What if the lines are identical, so the intersection is not a point but the entire line? What if one of the lines is horizontal, so that in the course of solving the equations for the intersection point you are likely to divide by zero? What if the two lines are almost parallel, so that the intersection point is so far from the origin as to cause arithmetic overflows? These issues become even more complicated for intersecting line segments, since there are many other special cases that must be watched for and treated specially.

If you are new to implementing geometric algorithms, I suggest that you study O'Rourke's *Computational Geometry in C* [O'R01] for practical advice and complete implementations of basic geometric algorithms and data structures. You are likely to avoid many headaches by following in his footsteps.

There are two different issues at work here: geometric degeneracy and numerical stability. *Degeneracy* refers to annoying special cases that must be treated in substantially different ways, such as when two lines intersect in more or less than a single point. There are three primary approaches to dealing with degeneracy:

- *Ignore it* – Make as an operating assumption that your program will work correctly only if no three points are collinear, no three lines meet at a point, no intersections happen at the endpoints of line segments, etc. This is probably the most common approach, and what I might recommend for short-term projects if you can live with frequent crashes. The drawback is that interesting data often comes from points sampled on a grid, and so is inherently very degenerate.

- *Fake it* – Randomly or symbolically perturb your data so that it becomes nondegenerate. By moving each of your points a small amount in a random direction, you can break many of the existing degeneracies in the data, hopefully without creating too many new problems. This probably should be the first thing to try once you decide that your program is crashing too often. A problem with random perturbations is that they can change the shape of your data in subtle ways, which may be intolerable for your application. There also exist techniques to "symbolically" perturb your data to remove degeneracies in a consistent manner, but these require serious study to apply correctly.
- *Deal with it* – Geometric applications can be made more robust by writing special code to handle each of the special cases that arise. This can work well if done with care at the beginning, but not so well if kludges are added whenever the system crashes. Expect to expend significant effort if you are determined to do it right.

Geometric computations often involve floating-point arithmetic, which leads to problems with overflows and numerical precision. There are three basic approaches to the issue of numerical stability:

- *Integer arithmetic* – By forcing all points of interest to lie on a fixed-size integer grid, you can perform exact comparisons to test whether any two points are equal or two line segments intersect. The cost is that the intersection point of two lines may not be exactly representable as a grid point. This is likely to be the simplest and best method, if you can get away with it.
- *Double precision reals* – By using double-precision floating point numbers, you may get lucky and avoid numerical errors. Your best bet might be to keep all the data as single-precision reals, and use double-precision for intermediate computations.
- *Arbitrary precision arithmetic* – This is certain to be correct, but also to be slow. This approach seems to be gaining favor in the research community. Careful analysis can minimize the need for high-precision arithmetic and thus the performance penalty. Still, you should expect high-precision arithmetic to be several orders of magnitude slower than standard floating-point arithmetic.

The difficulties associated with producing robust geometric software are still under attack by researchers. The best practical technique is to base your applications on a small set of geometric primitives that handle as much of the low-level geometry as possible. These primitives include:

- *Area of a triangle* – Although it is well known that the area $A(t)$ of a triangle $t = (a, b, c)$ is half the base times the height, computing the length of the base and altitude is messy work with trigonometric functions. It is better to use the determinant formula for *twice* the area:

  $$2 \cdot A(t) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

  This formula generalizes to compute $d!$ times the volume of a simplex in d dimensions. Thus, $3! = 6$ times the volume of a tetrahedron $t = (a, b, c, d)$ in three dimensions is

  $$6 \cdot A(t) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

  Note that these are signed volumes and can be negative, so take the absolute value first. Section 13.4 (page 404) explains how to compute determinants.

  The conceptually simplest way to compute the area of a polygon (or polyhedron) is to triangulate it and then sum up the area of each triangle. Implementations of a slicker algorithm that avoids triangulation are discussed in [O'R01, SR03].

- *Above-below-on test* – Does a given point c lie above, below, or on a given line l? A clean way to deal with this is to represent l as a directed line that passes through point a before point b, and ask whether c lies to the left or right of the directed line l. It is up to you to decide whether left means above or below.

  This primitive can be implemented using the sign of the triangle area as computed above. If the area of $t(a, b, c) > 0$, then c lies to the left of $\overline{ab}$. If the area of $t(a, b, c) = 0$, then c lies on $\overline{ab}$. Finally, if the area of $t(a, b, c) < 0$, then c lies to the right of $\overline{ab}$. This generalizes naturally to three dimensions, where the sign of the area denotes whether d lies above or below the oriented plane (a, b, c).

- *Line segment intersection* – The above-below primitive can be used to test whether a line intersects a line segment. It does iff one endpoint of the segment is to the left of the line and the other is to the right. Segment intersection is similar but more complicated. We refer you to implementations described below. The decision as to whether two segments intersect if they share an endpoint depends upon your application and is representative of the problems with degeneracy.

- *In-circle test* – Does point d lie inside or outside the circle defined by points a, b, and c in the plane? This primitive occurs in all Delaunay triangulation algorithms, and can be used as a robust way to do distance comparisons. Assuming that a, b, c are labeled in counterclockwise order around the circle, compute the determinant

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

  In-circle will return 0 if all four points are cocircular—a positive value if d is inside the circle, and negative if d is outside.

Check out the Implementations section before you build your own.

Implementations: CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) both provide very complete sets of geometric primitives for planar geometry written in C++. LEDA is easier to learn and to work with, but CGAL is more comprehensive and freely available. If you are starting a significant geometric application, you should at least check them out before you try to write your own.

O'Rourke [O'R01] provides implementations in C of most of the primitives discussed in this section. See Section 19.1.10 (page 662). These primitives were implemented primarily for exposition rather than production use, but they should be reliable and appropriate for small applications.

The *Core Library* (see *http://cs.nyu.edu/exact/*) provides an API, which supports the Exact Geometric Computation (EGC) approach to numerically robust algorithms. With small changes, any C/C++ program can use it to readily support three levels of accuracy: machine-precision, arbitrary-precision, and guaranteed.

Shewchuk's [She97] robust implementation of basic geometric primitives in C++ is available at *http://www.cs.cmu.edu/~quake/robust.html.*

Notes: O'Rourke [O'R01] provides an implementation-oriented introduction to computational geometry which stresses robust geometric primitives. It is recommended reading. LEDA [MN99] provides another excellent role model.

Yap [Yap04] gives a excellent survey on techniques for achieving robust geometric computation, with a book [MY07] in preparation. Kettner, et al. [KMP$^+$04] provides graphic evidence of the troubles that arise when employing real arithmetic in geometric algorithms such as convex hull. Controlled perturbation [MOS06] is a new approach for robust computation that is receiving considerable attention. Shewchuk [She97] and Fortune and van Wyk [FvW93] present careful studies on the costs of using arbitrary-precision arithmetic for geometric computation. By being careful about when to use it, reasonable efficiency can be maintained while achieving complete robustness.

Related Problems: Intersection detection (see page 591), maintaining arrangements (see page 614).