COMPUTATIONAL GEOMETRY IN C
SECOND EDITION

JOSEPH O’ROURKE
Assuming this determinant is the parallelepiped volume, argue that Equation (1.4) is twice the area of the indicated triangle.

2. Orientation of a polygon: from area [easy]. Given a list of vertices of a simple polygon in boundary traversal order, how can its orientation (clockwise versus counterclockwise) be determined using Theorem 1.3.3?

3. Orientation of a polygon. Use the proof of Lemma 1.2.1 to design a more efficient algorithm for determining the orientation of a polygon.

4. Volume of a cube. Compute the volume of a unit cube (side length 1) with the analog of Equation (1.12), using one vertex as $p$.

1.4. IMPLEMENTATION ISSUES

The remainder of the chapter takes a rather long “digression” into implementation issues. The goal is to present code to compute a triangulation. This hinges on detecting intersection between two segments, a seemingly trivial task that often is implemented incorrectly. We will approach segment intersection using the computation of areas from Section 1.3. We start with a few representation issues.

1.4.1. Representation of a Point

Arrays versus Records
All points will be represented by arrays of the appropriate number of coordinates. It is common practice to represent a point by a record with fields named $x$ and $y$, but this precludes the use of for-loops to iterate over the coordinates. There may seem little need to write a for-loop to iterate over only two indices, but I find it easier to understand, and it certainly generalizes to higher dimensions more easily.

Integers versus Reals
We will represent the coordinates with integers rather than with floating-point numbers wherever possible. This will permit us to avoid the issue of floating-point round-off error and allow us to write code that is verifiably correct within a range of coordinate values. Numerical error is an important topic and will be discussed at various points throughout the book (e.g., Sections 4.3.5 and 7.2). Obviously this habit of using integers will have to be relaxed when we compute, for example, the point of intersection between two line segments. The type definitions will be isolated so that modification of the code to handle different varieties of coordinate datatypes can be made in one location.

Point Type Definition
All type identifiers will begin with lowercase t. All defined constants will appear entirely in uppercase. The suffixes i and d indicate integer and double types respectively. See Code 1.1. In mathematical expressions, we will write $p_0$ and $p_1$ for $p[0]$ and $p[1]$.

\[19\text{That is, precludes it in most programming languages.}\]
1.4 Implementation Issues

```c
#define X 0
#define Y 1
typedef enum {FALSE, TRUE } bool;

#define DIM 2         /* Dimension of points */
typedef int tPointi[DIM]; /* Type integer point */
```

**Code 1.1** Point type.

1.4.2. Representation of a Polygon

The main options here are whether to use an array or a list, and if the latter, whether singly or doubly linked, and whether linear or circular.

Arrays are attractive for code clarity: The structure of loops and index increments are somewhat clearer with arrays than with lists. However, insertion and deletion of points is clumsy with arrays. As the triangulation code we develop will clip off ears, we will sacrifice simplicity to gain ease of deletion. In any case, we will need to use identical structures for the convex hull code in Chapters 3 and 4, so the investment here will reward us later. With an eye toward that generality, we opt to use a doubly linked circular list to represent a polygon. The basic cell of the data structure represents a single vertex, tVertexStructure, whose primary data field is tPoint. Pointers next and prev are provided to link each vertex to its adjacent vertices. See Code 1.2. An integer index vnum is included for printout, and other fields (such as bool ear) will be added as necessary.

```c
typedef struct tVertexStructure tsVertex; /* Used only in NEW(). */
typedef tsVertex *tVertex;
struct tVertexStructure {
    int     vnum;  /* Index */
    tPointi v;     /* Coordinates */
    bool    ear;   /* TRUE iff an ear */
    tVertex next, prev;
};
tVertex vertices = NULL;  /* “Head” of circular list. */
```

**Code 1.2** Vertex structure.

At all times, a global variable vertices is maintained that points to some vertex cell. This will serve as the “head” of the list during iterative processing. Loops over all vertices will take the form shown in Code 1.3. Care must be exercised if the processing in the loop deletes the cell to which vertex points.
tVertex v;
v = vertices;
do {
    /* Process vertex v */
    v = v->next;
} while ( v != vertices );

**Code 1.3** Loop to process all vertices.

We will need two basic list processing routines for vertex structures, one for allocating a new element (NEW) and another for adding a new element to the list (ADD). Looking ahead to later chapters, we write these as macros, with NEW taking the type as one parameter. This way the routines can be used for different types. (C does not permit manipulation of variables without regard to type, but macros are text based and oblivious to types). See Code 1.4. ADD first checks to see if head is non-NULL, and if so, it inserts the cell prior to head; if not, head points to the added cell, which is then the only cell in the list. The effect is that in a series of ADDS, the n-th point is added prior to the 0-th (the head) but after the (n-1)-st point.

```c
#define EXIT FAILURE 1
char *malloc();

#define NEW(p, type) (  
    if ((p=(type *)) malloc (sizeof(type))) == NULL) {  
        printf ("NEW: Out of Memory!\n");
        exit(EXIT FAILURE);
    }

#define ADD( head, p ) if ( head ) {  
    p->next = head;
    p->prev = head->prev;
    head->prev = p;
    p->prev->next = p;
}
else {  
    head = p;
    head->next = head->prev = p;
}

#define FREE(p) if (p) (free ((char *) p); p = NULL; )
```

**Code 1.4** NEW and ADD macros. (The backslashes continue the lines so that the preprocessor does not treat those as command lines.) FREE is used in Chapters 3 and 4.

### 1.4.3. Code for Area

Computing the area of a polygon is now a straightforward implementation of Equations (1.12) or (1.13). The former choice, with \( p = v_0 \), is shown in Code 1.5.
The data structures and conventions established in the previous section are employed.

```c
int Area2( tPointi a, tPointi b, tPointi c )
{
    return
    (b[X] - a[X]) * (c[Y] - a[Y]) -
    (c[X] - a[X]) * (b[Y] - a[Y]);
}

int AreaPoly2( void )
{
    int sum = 0;
    tVertex p, a;

    p = vertices; /* Fixed. */
    a = p->next; /* Moving. */
    do {
        sum += Area2( p->v, a->v, a->next->v );
        a = a->next;
    } while ( a->next != vertices );
    return sum;
}
```

**Code 1.5** Area2 and AreaPoly2.

There is an interesting potential problem with Area2: If the coordinates are large, the multiplications of coordinates could cause integer word overflow, which is unfortunately not reported by most C implementations. For Area2 we have followed the expression given by Equation (1.3) rather than that in (1.2), as the former both uses fewer multiplications and multiplies coordinate differences. Nevertheless, the issue remains, and we will revisit this point in Section 4.3.5. See Exercise 1.6.4[1].

### 1.5. SEGMENT INTERSECTION

#### 1.5.1. Diagonals

Our goal is to develop code to triangulate a polygon. The key step will be finding a diagonal of the polygon, a direct line of sight between two vertices $v_i$ and $v_j$. The segment $v_i v_j$ will not be a diagonal if it is blocked by a portion of the polygon's boundary. To be blocked, $v_i v_j$ must intersect an edge of the polygon. Note that if $v_i v_j$ only intersects an edge $e$ at its endpoint, perhaps only a grazing contact with the boundary, it is still effectively blocked, because diagonals must have clear visibility.

The following is an immediate consequence of the definition of a diagonal (Section 1.5.1):

**Lemma 1.5.1.** The segment $s = v_i v_j$ is a diagonal of $P$ iff
1. for all edges \( e \) of \( P \) that are not incident to either \( v_i \) or \( v_j \), \( s \) and \( e \) do not intersect:
\[ s \cap e = \emptyset \; ; \]
2. \( s \) is internal to \( P \) in a neighborhood of \( v_i \) and \( v_j \).

Condition (1) of this lemma has been phrased so that the "diagonalhood" of a segment can be determined without finding the actual point of intersection between \( s \) and each \( e \): Only a Boolean segment intersection predicate is required. Note that this would not be the case with the more direct implementation of the definition: A diagonal only intersects polygon edges at the diagonal endpoints. This phrasing would require computation of the intersection points and subsequent comparison to the endpoints. The purpose of condition (2) is to distinguish internal from external diagonals, as well as to rule out collinear overlap with an incident edge. We will revisit this condition in Section 1.6.2. We now turn our attention to developing code to check the nonintersection condition.

1.5.2. Problems with Slopes

Let \( v_i v_j = ab \) and \( e = cd \). A common first inclination when faced with the task of deciding whether \( ab \) and \( cd \) intersect is to find the point of intersection between the lines \( L_1 \) and \( L_2 \) containing the segments by solving the two linear equations in slope–intercept form, and then checking that the point falls on the segments. This method will clearly work, and it is not all that difficult to code. But the code is messy and error prone; it takes a surprising amount of diligence to get it exactly right. There are two special cases to handle: a vertical segment, whose containing line’s slope is infinite, and parallel segments, whose containing lines do not intersect. Both cases lead to division by zero in the computations, which must be avoided by special-case code. Even beyond this, checking that the point of intersection falls on the segments can lead to numerical precision problems.

To circumvent these problems, we avoid slopes altogether.

1.5.3. \texttt{Left}

Whether two segments intersect can be decided by using a \texttt{Left} predicate, which determines whether or not a point is to the left of a directed line. How \texttt{Left} is used to decide intersection will be shown in the next section. Here we concentrate on \texttt{Left} itself.

A directed line is determined by two points given in a particular order \((a, b)\). If a point \( c \) is to the left of the line determined by \((a, b)\), then the triple \((a, b, c)\) forms a counterclockwise circuit: This is what it means to be to the left of a line. See Figure 1.22.

Now the connection to signed area is finally clear: \( c \) is to the left of \((a, b)\) iff the area of the counterclockwise triangle, \(A(a, b, c)\), is positive. Therefore we may implement the \texttt{Left} predicate by a single call to \texttt{Area2} (Code 1.6).

Note that \texttt{Left} could be implemented by finding the equation of the line through \( a \) and \( b \), and substituting the coordinates of point \( c \) into the equation. This method would be straightforward but subject to the special case objections raised earlier. The area code in contrast has no special cases.
### 1.5 Segment Intersection

![Diagram](image)

**Figure 1.22** $c$ is left of $ab$ iff $\triangle abc$ has positive area; $\triangle abc'$ also has positive area.

```cpp
bool Left( tPointi a, tPointi b, tPointi c )
{
    return Area2( a, b, c ) > 0;
}

bool LeftOn( tPointi a, tPointi b, tPointi c )
{
    return Area2( a, b, c ) >= 0;
}

bool Collinear( tPointi a, tPointi b, tPointi c )
{
    return Area2( a, b, c ) == 0;
}
```

**Code 1.6** Left.

What happens when $c$ is collinear with $ab$? Then the determined triangle has zero area. Thus we have the happy circumstance that the exceptional geometric situation corresponds to the exceptional numerical result. As it will sometimes be useful to distinguish collinearity, we write a separate **Collinear** predicate\(^{20}\) for this, as well as **LeftOn**, giving us the equivalent of $=, <,$ and $\leq$; see again Code 1.6.

Note that we are comparing twice the area against zero in these routines: We are not comparing the area itself. The reason is that the area might not be an integer, and we would prefer not to leave the comfortable domain of the integers.

#### 1.5.4. Boolean Intersection

If the two segments $ab$ and $cd$ intersect in their interiors, then $c$ and $d$ are split by the line $L_1$ containing $ab$: $c$ is to one side and $d$ to the other. And likewise, $a$ and $b$ are

---

\(^{20}\)If floating-point coordinates are demanded by a particular application, this predicate would need modification, as it depends on exact equality with zero.
split by \( L_2 \), the line containing \( cd \). See Figure 1.23(a). Neither one of these conditions is alone sufficient to guarantee intersection, as Figure 1.23(b) shows, but it is clear that both together are sufficient. This leads to straightforward code to determine proper intersection, when two segments intersect at a point interior to both, if it is known that no three of the four endpoints are collinear. We can enforce this noncollinearity condition by explicit check; see Code 1.7.

```c
bool IntersectProp(tPointi a, tPointi b, tPointi c, tPointi d)
{
    /* Eliminate improper cases. */
    if (
        Collinear(a,b,c) ||
        Collinear(a,b,d) ||
        Collinear(c,d,a) ||
        Collinear(c,d,b)
    )
        return FALSE;

    return
        Xor( Left(a,b,c), Left(a,b,d) )
    &
        Xor( Left(c,d,a), Left(c,d,b) )
    ;
}

/* Exclusive or: True if exactly one argument is true. */
bool Xor( bool x, bool y )
{
    /* The arguments are negated to ensure that they are 0/1 values. */
    return !x ^ !y;
}
```

**Code 1.7 IntersectProp.**
There is unfortunate redundancy in this code, in that the four relevant triangle areas are being computed twice each. This redundancy could be removed by computing the areas and storing them in local variables, or by designing other primitives that fit the problem better. I would argue against storing the areas, as then the code would not be transparent. But it may be that the code can be designed around other primitives more naturally. It turns out that the first if-statement may be removed entirely for the purposes of triangulation, although then the routine no longer computes proper intersection (nor does it compute improper intersection). This is explored in Exercise 1.6.4[2]. I prefer to sacrifice efficiency for clarity and leave IntersectProp as is, for it is useful to look beyond the immediate programming task to possible other uses. In this instance, IntersectProp is precisely the function needed to compute clear visibility (Section 1.1.2).

One subtlety occurs here: It might be tempting to implement the exclusive-or by requiring that the products of the relevant areas be strictly negative, thus assuring that they are of opposite sign and that neither is zero:

\[
\text{Area2}(a, b, c) \times \text{Area2}(a, b, d) < 0 \\
\&\& \text{Area2}(c, d, a) \times \text{Area2}(c, d, b) < 0;
\]

The weakness in this formulation is that the product of the areas might cause integer word overflow! Thus a clever coding trick to save a few lines could hide a pernicious bug. This overflow problem can be avoided by having Area2 return +1, 0, or −1 rather than the true area (Sedgewick 1992, p. 350). I prefer to return the area for now, as this is useful in other contexts – for example, to compute the area of a polygon! In Chapter 4 we will revise this decision when we discuss overflow as a generic problem (Code 4.23).

Improper Intersection

Finally we must deal with the “special case” of improper intersection between the two segments, as Lemma 1.5.1 requires that the intersection be completely empty for a segment to be a diagonal. Improper intersection occurs precisely when an endpoint of one segment (say c) lies somewhere on the other (closed) segment ab. See Figure 1.24(a). This can only happen if a, b, c are collinear. But collinearity is not a sufficient condition

![Diagram](image)

**FIGURE 1.24** Improper intersection between two segments (a); collinearity is not sufficient (b).
for intersection, as Figure 1.24(b) makes clear. What we need to decide is if \( c \) is between \( a \) and \( b \).

**Betweenness**

We would like to compute this "betweenness" predicate without resorting to slopes, which would require special-case handling. Because we will only check betweenness of \( c \) when we know it lies on the line containing \( ab \), we may exploit this knowledge. If \( ab \) is not vertical, then \( c \) lies on \( ab \) iff the \( x \) coordinate of \( c \) falls in the interval determined by the \( x \) coordinates of \( a \) and \( b \). If \( ab \) is vertical, then a similar check on \( y \) coordinates determines betweenness. See Code 1.8.

```
bool Between( tPointi a, tPointi b, tPointi c )
{
    tPointi ba, ca;

    if ( ! Collinear( a, b, c ) )
        return FALSE;

    /* If ab not vertical, check betweenness on x; else on y. */
    if ( a[X] != b[X] )
        return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
                ((a[X] >= c[X]) && (c[X] >= b[X]));
    else
        return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
                ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
```

**Code 1.8** Between.

### 1.5.5. Segment Intersection Code

We finally can present code for computing segment intersection. Two segments intersect iff they intersect properly or one endpoint of one segment lies between the two endpoints of the other segment. The check for improper intersection is therefore implemented by four calls to \( \text{Between} \); see Code 1.9. Exercise 1.6.4[3] asks for an analysis of the inefficiencies of this routine.

### 1.6. TRIANGULATION: IMPLEMENTATION

#### 1.6.1. Diagonals, Internal or External

Having developed segment intersection code, we are nearly prepared to write code for triangulating a polygon. Our first goal is to find a diagonal of the polygon.
7

Search and Intersection

7.1. INTRODUCTION

In this (long) chapter we examine several problems that can be loosely classified as involving search or intersection (or both). This is a vast, well-developed topic, and I will make no attempt at systematic coverage. The chapter starts with two constant-time computations that are generally below the level considered in the computational geometry literature: intersecting two segments (Section 7.2) and intersecting a segment with a triangle (Section 7.3). Implementations are presented for both tasks. Next we employ these algorithms for two more difficult problems: determining whether a point is in a polygon – the “point-in-polygon problem” (Section 7.4), and the “point-in-polyhedron problem” (Section 7.5). The former is a heavily studied problem; the latter has seen less scrutiny. Again implementations are presented for both. We next turn to intersecting two convex polygons (Section 7.6), again with an implementation (the last in the chapter). Intersecting a collection of segments (Section 7.7) leads to intersection of nonconvex polygons (Section 7.8).

The theoretical jewel in this chapter is an algorithm to find extreme points of a polytope in any given query direction (Section 7.10). This leads naturally to planar point location (Section 7.11), which allows us to complete the explanation of the randomized triangulation algorithm from Chapter 2 (Section 2.4.1) with a presentation of a randomized algorithm to construct a search structure for a trapezoid decomposition (Section 7.11.4).

7.2. SEGMENT–SEGMENT INTERSECTION

In Chapter 1 (Section 1.5) we spent some time developing code that detects intersection between two segments for use in triangulation (Intersect, Code 1.9), but we never bothered to compute the point of intersection. It was not needed in the triangulation algorithm, and it would have forced us to leave the comfortable world of integer coordinates. For many applications, however, the floating-point coordinates of the point of intersection are needed. We will need this to compute the intersections between two polygons in Sections 7.6 and 7.8. Fortunately, it is not too difficult to compute the intersection point (although there are potential pitfalls), and the necessary floating-point calculations are not as problematical here as they sometimes are. In this section we develop code for this task.

See, e.g., de Berg et al. (1997).
Although the computation could be simplified a bit by employing the Boolean Intersect from Chapter 1, we opt here for an independent calculation. Let the two segments have endpoints \(a\) and \(b\) and \(c\) and \(d\), and let \(L_{ab}\) and \(L_{cd}\) be the lines containing the two segments. A common method of computing the point of intersection is to solve slope–intercept equations for \(L_{ab}\) and \(L_{cd}\) simultaneously:\(^2\) two equations in two unknowns (the \(x\) and \(y\) coordinates of the point of intersection). Instead we will use a parametric representation of the two segments, as the meaning of the variables seems more intuitive. We will see in Section 7.3 that the parametric approach generalizes nicely to more complex intersection computations.

Let \(A = b - a\) and \(C = d - c\); these vectors point along the segments. Any point on the line \(L_{ab}\) can be represented as the vector sum \(p(s) = a + sA\), which takes us to a point \(a\) on \(L_{ab}\), and then moves some distance along the line by scaling \(A\) by \(s\). See Figure 7.1. The variable \(s\) is called the parameter of this equation. Consider the values obtained for \(s = 0\), \(s = 1\), and \(s = \frac{1}{2}\): \(p(0) = a\), \(p(1) = a + A = a + b - a = b\), and \(p(\frac{1}{2}) = (a + b)/2\). These examples demonstrate that \(p(s)\) for \(s \in [0, 1]\) represents all the points on the segment \(ab\), with the value of \(s\) representing the fraction of the distance between the endpoints; in particular, the extremes of \(s\) yield the endpoints.

We can similarly represent the points on the second segment by \(q(t) = c + tC\), \(t \in [0, 1]\). A point of intersection between the segments is then specified by values of \(s\) and \(t\) that make \(p(s)\) equal to \(q(t)\): \(a + sA = c + tC\). This vector equation also comprises two equations in two unknowns: the \(x\) and \(y\) equations, both with \(s\) and \(t\) as unknowns. With our usual convention of subscripts 0 and 1 indicating \(x\) and \(y\) coordinates, its solution is

\[
\begin{align*}
    s &= \frac{a_0(d_1 - c_1) + c_0(a_1 - d_1) + d_0(c_1 - a_1)}{D}, \\
    t &= \frac{c_0(c_1 - b_1) + b_0(a_1 - c_1) + c_0(b_1 - a_1)}{D}, \\
    D &= a_0(d_1 - c_1) + b_0(c_1 - d_1) + d_0(b_1 - a_1) + c_0(a_1 - b_1)
\end{align*}
\]

Division by zero is a possibility in these equations. The denominator \(D\) happens to be zero iff the two lines are parallel, a claim left to Exercise 7.3.2[1]. Some parallel segments involve intersection, and some do not, as we detailed in Chapter 1 (Section 1.5.4). Temporarily, we will treat parallel segments as nonintersecting. The above equations lead to the rough code shown in Code 7.1. We will first describe this code, then criticize it, and finally revise it.

\(^2\)E.g., see Berger (1986, pp. 332–5).
# define X 0
# define Y 1
typedef enum {FALSE, TRUE} bool;
# define DIM 2
/* Dimension of points */
typedef int tPointi[DIM];
/* Type integer point */
typedef double tPointd[DIM];
/* Type double point */

bool SegSegInt(tPointi a, tPointi b, tPointi c, tPointi d,
tPointd p)
{
    double s, t;
    /* Parameters of the parametric eqns. */
    double num, denom;
    /* Numerator and denominator of eqns. */

    denom = a[X] * (d[Y] - c[Y]) +
           b[X] * (c[X] - d[Y]) +
           d[X] * (b[Y] - a[Y]) +
           c[X] * (a[Y] - b[Y]);

    /* If denom is zero, then segments are parallel. */
    if (denom == 0.0)
        return FALSE;

    num = a[X] * (d[Y] - c[Y]) +
          c[X] * (a[Y] - d[Y]) +
          d[X] * (c[Y] - a[Y]);
    s = num / denom;

    num = -(a[X] * (c[Y] - b[Y]) +
           b[X] * (a[Y] - c[Y]) +
           c[X] * (b[Y] - a[Y]));
    t = num / denom;

    p[X] = a[X] + s * (b[X] - a[X]);

    if ((0.0 <= s) && (s <= 1.0) &&
        (0.0 <= t) && (t <= 1.0))
        return TRUE;
    else return FALSE;
}

Code 7.1 Segment–segment intersection code: rough attempt.
The code takes the four integer-coordinate endpoints as input and produces two types of output: It returns a Boolean indicating whether or not the segments intersect, and it returns in the point \( p \) the double coordinates of the point of intersection; note that \( p \) is of type \texttt{double tPointd}. The computations of the numerators and denominators parallel Equations (7.1)–(7.3) exactly, and the test for intersection is \( 0 \leq s \leq 1 \) and \( 0 \leq t \leq 1 \).

There are at least three weaknesses to this code:

1. The code does not handle parallel segments. Most applications will need to know whether the segments overlap or not.
2. Many applications need to distinguish proper from improper intersections, just as we did for triangulation in Chapter 1. It would be useful to distinguish these in the output.
3. Although floating-point variables are used, the multiplications are still performed with integer arithmetic before the results are converted to \texttt{doubles}. Here is a simple example of how this code can fail due to overflow. Let the four endpoints be

\[
\begin{align*}
a &= (-r, -r), \\
b &= (+r, +r), \\
c &= (+r, -r), \\
d &= (-r, +r).
\end{align*}
\]

The segments form an 'X' shape intersecting at \( p = (0, 0) \). Calculation shows that the numerators from Equations (7.1) and (7.2) are both \(-4r^2\). For \( r = 10^5 \), this is \(-4 \times 10^{10}\), which exceeds what can be represented in 32 bits. In this case my machine returns \( p = (-267702.8, -267702.8) \) as the point of intersection!

We now address each of these three problems. First, we change the function from \texttt{bool to char} and have it return a "code" that indicates the type of intersection found. Applications that need to base decisions on whether or not the intersection is proper can use this code. Although the exact codes used should depend on the application, the following capture most needs:

- 'e': The segments collinearly overlap, sharing a point; 'e' stands for 'edge.'
- 'v': An endpoint of one segment is on the other segment, but 'e' doesn't hold; 'v' stands for 'vertex.'
- '1': The segments intersect properly (i.e., they share a point and neither 'v' nor 'e' holds); '1' stands for \texttt{TRUE}.
- '0': The segments do not intersect (i.e., they share no points); '0' stands for \texttt{FALSE}.

Note that the case where two collinear segments share just one point, an endpoint of each, is classified as 'e' in this scheme, although 'v' might be more appropriate in some contexts.

Second, we increase the range of applicability of the code by forcing the multiplications to floating-point by casting with \texttt{(double)}. This leads us to the code shown in Code 7.2. Before moving to the parallel segment case, let us point out a few features
char SegSegInt( tPointi a, tPointi b, tPointi c, tPointi d, tPointid p )
{

double s, t; /* The two parameters of the parametric eqns. */
double num, denom; /* Numerator and denominator of equations. */
char code = '"'; /* Return char characterizing intersection. */

c[X] * (double)( a[Y] - b[Y] );

/* If denom is zero, then segments are parallel: handle separately. */
if (denom == 0.0)
    return ParallelInt(a, b, c, d, p);

d[X] * (double)( c[Y] - a[Y] );
if ( (num == 0.0) || (num == denom) ) code = '"';

s = num / denom;

c[X] * (double)( b[Y] - a[Y] ) );
if ( (num == 0.0) || (num == denom) ) code = '"';
t = num / denom;

if ( (0.0 < s) && (s < 1.0) &&
    (0.0 < t) && (t < 1.0) )
    code = '"';
else if ( (0.0 > s) || (s > 1.0) ||
    (0.0 > t) || (t > 1.0) )
    code = '"';

p[X] = a[X] + s * ( b[X] - a[X] );
p[Y] = a[Y] + s * ( b[Y] - a[Y] );

return code;
}

Code 7.2 SegSegInt.
```c
char ParallelInt(tPointi a, tPointi b, tPointi c, tPointi d,
                 tPointd p)
{
    if (!Collinear(a, b, c))
        return '0';

    if (Between(a, b, c)) {
        Assigndi(p, c); return 'e';
    }
    if (Between(a, b, d)) {
        Assigndi(p, d); return 'e';
    }
    if (Between(c, d, a)) {
        Assigndi(p, a); return 'e';
    }
    if (Between(c, d, b)) {
        Assigndi(p, b); return 'e';
    }
    return '0';
}

void Assigndi(tPointd p, tPointi a)
{
    int i;
    for (i = 0; i < DIM; i++)
        p[i] = a[i];
}

bool Between(tPointi a, tPointi b, tPointi c)
{
    tPointi ba, ca;

    /* If ab not vertical, check betweenness on x; else on y. */
    if (a[X] != b[X])
        return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
                ((a[X] >= c[X]) && (c[X] >= b[X]));
    else
        return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
                ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
```

**Code 7.3** ParallelInt.
of this code. Checking the 'v' case is done with \( \text{num} \) rather than with \( s \) and \( t \) after division; this skirts possible floating-point inaccuracy in the division. The check for proper intersection is \( 0 < s < 1 \) and \( 0 < t < 1 \); the reverse inequalities yield no intersection.

With the computations forced to doubles, the range is greatly extended. I could only make it fail for coordinates each over a billion: \( r = 1234567809 \approx 10^9 \) in the previous overflow example. It is not surprising that it fails here, as \(-4r^2\) is now over \(10^{18}\), which requires 60 bits, exceeding the accuracy of double mantissas on most machines.

Finally we come to parallel segments, handled by a separate procedure ParallelInt. Collinear overlap was dealt with in Chapter 1 with the function Between (Code 1.8), which is exactly what we need here: The segments overlap iff an endpoint of one lies between the endpoints of the other. There is one small simplification. In the triangulation code, we had Between check collinearity, but here we can make one check: If \( c \) is not collinear with \( ab \), then the parallel segments \( ab \) and \( cd \) do not intersect. The straightforward code is shown in Code 7.3. Note that an endpoint is returned as the point of intersection \( p \). It is conceivable that some application might prefer to have the midpoint of overlap returned; in Section 7.6 we will need the entire segment of overlap.

It should be clear that minor modification of this intersection code can find ray-segment, ray-ray, ray-line, or line-ray intersection, by altering the acceptable \( s \) and \( t \) ranges. For example, accepting any nonnegative \( s \) corresponding to a positive stretch of the first segment yields ray-segment intersection.

### 7.3. SEGMENT–TRIANGLE INTERSECTION

We now turn to the more difficult, but still ultimately straightforward, computation of the point of intersection between a segment and a triangle in three dimensions. We will use this code in Section 7.5 to detect whether a point is in a polyhedron, but it has many other uses. In fact this is one of the most prevalent geometric computations performed today, because it is a key step in “ray tracing” used in computer graphics: finding the intersection between a light ray and a collection of polygons in space.

We will again use a parametric representation to derive the equations. Throughout we will let \( T = \triangle abc \) be the triangle and \( qr \) the segment, where \( q \) is viewed as the originating (“query”) endpoint in case \( qr \) represents a ray and \( r \) is the “ray” endpoint. We will assume throughout that \( r \neq q \), so the input segment has nonzero length.

#### 7.3.1. Segment–Plane Intersection

The first step is to determine if \( qr \) intersects the plane \( \pi \) containing \( T \). We will pursue this halfway goal throughout this subsection before turning to determining if the point of intersection lies in the triangle.
b. Design a Boolean function that determines if two such rationals are equal. Note that, e.g.,
\[ 2/6 = 127131/381393. \]
c. [programming] Implement the rational equal function.

4. **Clean code** [open]. Design a set of advance rules that handle the special cases more cleanly
than does the presented code. Ideally all the cases in Figure 7.16, as well as \( P \subset Q, Q \subset P, \)
and \( P \cap Q = \emptyset, \) would be handled naturally.

### 7.7. INTERSECTION OF SEGMENTS

Although we have seen that \( \Omega(n^2) \) is a lower bound on intersecting two polygons of \( n \)
edges each, in many applications the worst case is rare. This suggests a goal of developing an
output-size sensitive algorithm, one whose complexity depends on \( k, \) the size of (the number of vertices in) the output.\(^{18}\) It turns out that the hard part of this task is
a more general problem: finding the intersections among a collection of \( n \) segments in
the plane. This is more general in that no assumption is made that the segments connect
to form polygons. We will now pursue the segment intersection problem, presenting an
elegant algorithm due to Bentley & Ottmann (1979), one of the first output-size sensitive
algorithms in computational geometry, and return in Section 7.8 to the problem of
intersecting polygons.

A brute-force intersection algorithm takes \( \Omega(n^2) \) time: Check each segment against
every other (using, e.g., `SegSegInt` from Section 7.2). To achieve output sensitivity,
we want to compute intersections between only those pairs of segments that actually
intersect. This goal sounds circular, but even this formulation carries a hint of a solution, for
segments that intersect are close to one another – not throughout their length, but certainly
at their point of intersection! If we could somehow “travel down” the lengths of a pair of
segments until they become close to one another before deciding to intersect, we could
achieve output sensitivity. Plane sweep provides the needed “travel down” mechanism.

Recall from Section 2.4 that a trapezoidalization could be computed efficiently by
sweeping a horizontal line \( L \) over a collection of polygon edges, and only searching for
chord intersections in the local neighborhood of a vertex hit by \( L. \) It is just this sort of
local focus we need for the segment intersection problem.

Imagine sweeping the line \( L \) over a collection of segments \( S = \{s_0, s_1, \ldots, s_{n-1}\}. \)
Let \( x = s_i \cap s_j \) be an intersection point between two segments. Just before \( L \) reaches \( x, \)
\( L \) pierces both \( s_i \) and \( s_j, \) and they are adjacent along \( L: \) No other segment is between
them on \( L. \) Thus, at some time prior to every intersection “event” (when \( L \) crosses
an intersection point), the intersecting segments are adjacent on \( L. \) This gives us the
sought-for locality: Computing intersections between segments adjacent on \( L \) suffices
to capture all intersection points. Some of these adjacent segments do not in fact intersect,
but we will see that the “wasted effort” is small.

Let us make several simplifying assumptions to keep focused on the main idea:
Assume no segment is horizontal and no three segments pass through one point. The
plan is to sweep \( L \) over the segments, stopping at events of three types, when

\(^{18}\)See Sections 3.3, 4.6.4, and 6.7.2 for other output-size sensitive algorithms.
7.7 Intersection of Segments

All three of these events cause the list \( L \) of segments pierced by \( L \) to change: A segment is inserted, deleted, or two adjacent segments switch places, respectively. With each change, intersections between newly adjacent segments must be computed.

Although segments must become adjacent in \( L \) prior to their point of intersection \( x \), it is not guaranteed that \( x \) is the next intersection event when it is computed. Rather, the intersection events must be placed in a queue \( Q \) sorted by height, along with the segment endpoints. An illustration should make the algorithm clear. Consider the set of segments \( S = \{ s_0, s_1, \ldots \} \) shown in Figure 7.17. Let \( a_i \) be the upper endpoint of segment \( s_i \), and \( b_i \) its lower endpoint. Then the event queue is initialized to \( Q = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, b_2, \ldots) \), all the segment endpoints sorted top to bottom. When \( L \) reaches \( s_1 \) and \( a_2 \) (position 1), \( s_0 \) and \( s_1 \) become newly adjacent, and their intersection point \( x_{01} \) is added to the queue after \( b_2 \). \( s_1 \) and \( s_2 \) are also newly adjacent but do not intersect. Note that the higher intersection point \( x_{56} \) has not yet been constructed.

At position 2, \( L \) hits \( a_3 \); the newly adjacent segments \( s_3 \) and \( s_0 \) do not intersect. At this point \( L = (s_3, s_0, s_1, s_2) \). At position 3, \( L \) hits \( a_4 \), and intersection point \( x_{34} \) is added to \( Q \) at its appropriate location. Note that three intersection events “between” \( s_3 \) and \( s_4 \) will be encountered before \( x_{34} \) is reached. By the time \( L \) reaches the first intersection event at position 6, all the endpoints above have been processed, and \( L = (s_3, s_5, s_6, s_4, s_0, s_1) \). This event causes \( s_5 \) and \( s_6 \) to switch places in \( L \), introducing new adjacencies that result in \( x_{36} \) and \( x_{45} \) being added to \( Q \). \( Q \) now contains all the circled intersection points shown in the figure.

The algorithm needs to maintain two dynamic data structures: one for \( L \) and one for \( Q \). Both must support fast insertions and deletions in order to achieve an overall
low time complexity. We will not pursue the data structure details\(^\text{19}\) but only claim that balanced binary trees suffice to permit both \(L\) and \(Q\) to be stored in space proportional to their number of elements \(m\), with all needed operations performable in \(O(\log m)\) time. We now argue that such structures lead to a time complexity for intersecting \(n\) segments of \(O((n + k) \log n)\), where \(k\) is the number of intersection points between the segments. We will continue to assume that no three segments meet in one point.

The total number of events is \(2n + k = O(n + k)\): the \(2n\) segment endpoints and the \(k\) intersection points. Thus the length of \(Q\) is never more than this. Because each event is inserted once and deleted once from \(Q\), the total cost of maintaining \(Q\) is \(O((n + k) \log(n + k))\). Because \(k = O(n^2)\), \(O(\log(n + k)) = O(\log n + 2 \log n) = O(\log n)\). Thus maintaining \(Q\) costs \(O((n + k) \log n)\).

The total cost of maintaining \(L\) is \(O(n \log n)\): \(n\) segments inserted and deleted at \(O(\log n)\) each. It only remains to bound the number of intersection computations (each of which can be performed in constant time, by a call to \texttt{SegSegInt}, Code 7.2). Recall the earlier worry about \"wasted effort.\" However, the number of intersection calls is at most twice the number of events, because each event results in at most two new segment adjacencies: an inserted segment with its new neighbors, two new neighbors when the segment between is deleted, and new left and right neighbors created by a switch at an intersection event. Thus the total number of intersection calls is \(O(n + k)\).

The overall time complexity of the algorithm is therefore \(O((n + k) \log n)\), sensitive to the output size \(k\). We have seen that the space requirements are \(O(n + k)\) because this is how long \(Q\) can grow. It turns out that this can be reduced to \(O(n)\) (Exercise 7.8.1[2]). Moreover, both of these desirable complexities can be achieved without any of our simplifying assumptions (Exercise 7.8.1[1]).

These results were achieved by 1981 (Bentley & Ottmann 1979; Brown 1981), but more than a decade of further work was needed to reach an optimal algorithm in both time and space:

**Theorem 7.7.1.** The intersection of \(n\) segments in the plane may be constructed in \(O(n \log n + k)\) time (Chazelle & Edelsbrunner 1992) and \(O(n)\) space (Balaban 1995), where \(k\) is the number of intersection points between the segments.

Here \(k\) is not multiplied by \(\log n\) as in the original Bentley–Ottmann algorithm. The practical difference may be slight, but closing the theoretical gap required the development of new techniques.

## 7.8. INTERSECTION OF NONCONVEX POLYgons

It is not difficult to alter the Bentley–Ottmann sweepline algorithm to compute the intersection of two polygons. Let the two polygons be \(A\) and \(B\), with vertices labeled \(a_j\) and \(b_j\) respectively. The main idea is similar to that used by scan-line algorithms for filling (painting) a polygonal region on a graphics screen\(^\text{20}\) and is related to our ray-crossing

\(^{19}\)See de Berg, van Kreveld, Overmars & Schwarzkopf (1997, Sec. 2.1), Preparata & Shamos (1985, Sec. 7.2.3), or Mehlhorn (1984, Sec. VIII. 4.1).

\(^{20}\)See, e.g., Foley, van Dam, Feiner, Hughes & Phillips (1993, Sec. 3.5).
analysis in Section 7.4. One maintains along the length of the sweep line \( L \) a "status" indicator, which has the following value:

- \( \emptyset \): exterior to both polygons;
- \( A \): inside \( A \), but outside \( B \);
- \( B \): inside \( B \), but outside \( A \); or
- \( AB \): inside both \( A \) and \( B \).

The status is recorded for the span between each two adjacent segments pierced by \( L \); clearly it is constant throughout each span.

Consider the example shown in Figure 7.18. When \( L \) is at position 2 (event \( b_1 \)), the left-to-right status list is \( (\emptyset, A, AB, B, \emptyset) \). This information can be easily stored in the same data structure representing \( L \). We will not delve into the data structure details, but rather sketch how the status information can be updated in the same sweep that processes the segment intersection events, using the example in Figure 7.18.
At position 0, when \( L \) hits \( a_0 \), the fact that both \( A \)-edges are below \( a_0 \) indicates that we are inserting an \( A \)-span. At position 1, a \( B \)-span is inserted. Just slightly below \( b_0 \), an intersection event opens up an \( AB \)-span, easily recognized as such because the intersecting segments each bound \( A \) and \( B \) from opposite sides, with \( A \) and \( B \) below. At position 3, intersection event \( x \), the opposite occurs: The intersecting segments each bound \( A \) and \( B \) above them. Thus an \( AB \)-span disappears, replaced by an \( \emptyset \)-span between the switched segments. At \( a_2 \) (position 4), the inverse of the \( a_0 \) situation is encountered: The \( A \)-edges are above, and an \( A \)-span is engulfed by the surrounding \( B \)-spans. Although we have not provided precise rules (Exercise 7.8.1[5]), it should be clear that the span status information may be maintained by the sweepline algorithm without altering the asymptotic time or space complexity.

Although this enables us to “paint” the intersection \( A \cap B \) on a raster display, there is a further step or two to obtain lists of edges for each “polygonal” piece of \( A \cap B \). The reason for the scare quotes around “polygonal” is that the intersection may include pieces that are degenerate polygons: segments, points, etc. – what are sometimes collectively called “hair.” Whether this is desired as part of the output depends on the application. This issue aside, there is still further work. For example, at position 3 in Figure 7.18, an \( AB \)-span disappears at \( x \), but the polygonal piece that disappears locally at \( x \) continues on to lower sweepline positions elsewhere. Two \( AB \)-spans may merge, revealing that what appeared to be two separate pieces above \( L \) are actually joined below.

This aspect of the algorithm may be handled by growing polygonal chain boundaries for the pieces of the intersection as the sweepline progresses and then joining these pieces at certain events. Thus position 3 in the figure is an event that initiates joining a left-bounding \( AB \)-chain with a right-bounding \( AB \)-chain. Keeping track of the number of “dangling endpoints” of a chain permits detection of when a complete piece of the output has been passed: For example, at position 4 of the figure, \( a_2 \) closes up the chain and an entire piece can be printed, whereas at position 3, the chain joined at \( x \) remains open at its rightmost piercing with \( L \). Again we will not present details.

Finally, it is easy to see that we could just have easily computed \( A \cup B \), or \( A \setminus B \), or \( B \setminus A \) – the status indicator is all we need to distinguish these. Thus all “Boolean operations” between polygons may be constructed with variants of the Bentley–Ottmann sweepline algorithm, in the same time complexity. These Boolean operations are the heart of many CAD/CAM software systems, which, for example, construct complex parts for numerically controlled machining by subtracting one shape from another, joining shapes, slicing away part of a shape, etc., all of which are Boolean operations.

**Theorem 7.8.1.** The intersection, union, or difference of two polygons with a total of \( n \) vertices, whose edges intersect in \( k \) points, may be constructed in \( O(n \log n + k) \) time and \( O(n) \) space.

### 7.8.1. Exercises

1. **Handling degeneracies.**

   (a) [easy] Show that horizontal segments can be accommodated within the presented algorithm without increasing time or space complexity.