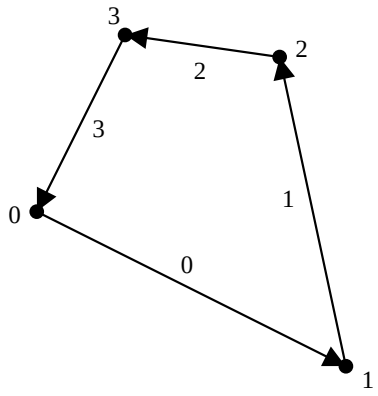
CS69001 Computing Laboratory – I

Assignment No: A1

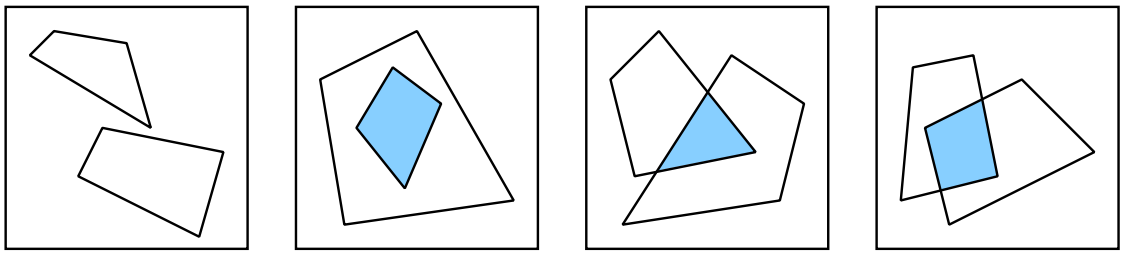
Date: 22–July–2019

A *quadrilateral* (also called a *quadrangle* and a *tetragon*) is a four-sided plane polygon. For simplicity, we concentrate only on simple and convex quadrilaterals. A quadrilateral is specified by the coordinates of its four corners. We assume that the corners are supplied in the counterclockwise direction starting from any corner. We number the corners as $0, 1, 2, 3$, and the sides (line segments, often directed) as $0, 1, 2, 3$ too.

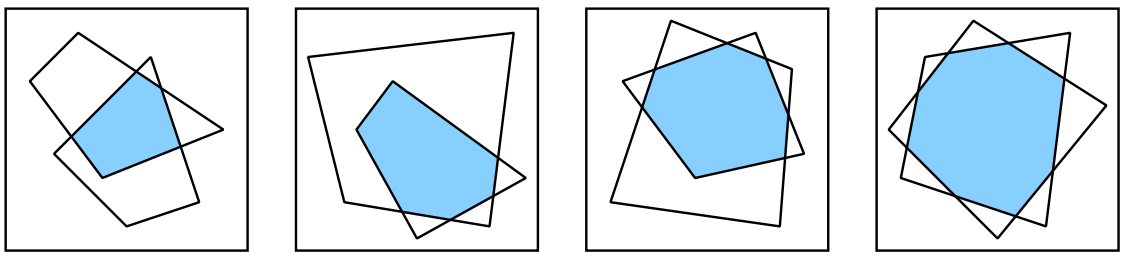


In this assignment, you write a program to compute the intersection polygon of two given quadrilaterals R_1 and R_2. We are mostly interested in the area of the intersection polygon. You are required to implement three different algorithms to compute the (approximate) intersection area. Assume that both the quadrilaterals lie completely inside the unit square U with corners at $(0,0)$, $(1,0)$, $(1,1)$ and $(0,1)$.

Before proceeding to a description of the algorithms, let us look at the following figure which demonstrates different possible cases of intersection. If the two quadrilaterals are disjoint (Part (a)), their intersection area is zero. If one of the quadrilaterals is completely inside the other (Part (b)), the intersection polygon is the inner quadrilateral. If the sides of the quadrilaterals intersect at one or more points, we have six possible cases (Parts (c)–(h)) characterized by the number of sides that the intersection polygon has.



(a) No intersection (b) One inside other (c) Three sides (d) Four sides



(e) Five sides (f) Six sides (g) Seven sides (h) Eight sides

We assume that the corners of the input polygons are in *general position*. In computational geometry, this means that no degenerate cases occur. For example, the sides of the given quadrilaterals are not parallel to one another, or to the x axis or the y axis. Moreover, we assume that there are no cases of touch, that is,

if two sides of the two quadrilaterals intersect, they do so somewhere in their interiors (not at one or more corners). Degenerate cases can often be handled with extra care, but you are not required to do so here.

Throughout this assignment, you are encouraged to use double-precision floating-point arithmetic. That is, use **`double`** variables instead of single-precision **`float`** variables.

Part 1: Some Useful Data Types and Functions

Define a user-defined data type (or class) to represent a point in the two-dimensional plane. This is a pair of **`double`** variables. Likewise, define a data type to store a polygon (like a quadrilateral or an intersection polygon). For the problem at hand, a polygon may have at most eight corners, so an array of eight points suffices to store a polygon.

Write a function **`segint(P1,P2,Q1,Q2)`** to compute the point of intersection of two line segments P_1P_2 and Q_1Q_2 (if it exists). Line segments are bounded in both the directions. Therefore, the segments may have no intersection even if the lines supporting the segments intersect. Generate the equation of the two lines P_1P_2 and Q_1Q_2, and if they are not parallel, compute their point of intersection. Finally, check whether this point of intersection belongs to both the segments.

Write a function **`side(P1,P2,P)`** to compute the position of the point P relative to the *directed* line P_1P_2. Compute the determinant

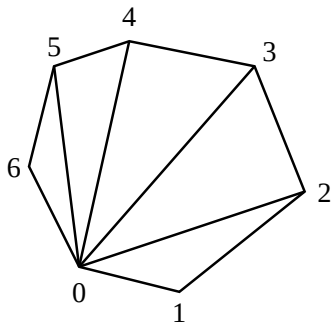
$$D = \begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x & y \end{vmatrix}.$$

P is on, to the left, or to the right of P_1P_2 according as whether $D = 0$, $D > 0$, or $D < 0$, respectively.

Write a function **`pointpos(R,P)`** to find the position of a point P relative to a quadrilateral R. We are interested in finding out whether P lies in the interior of R or not. Under the convention that the corners of R are arranged in the counterclockwise order, the four sides are given orientation from a corner to the next corner in the counterclockwise order. But then, P lies inside R if and only if P lies to the left of *all* of the directed sides of the quadrilateral R.

Write a function **`tarea(A,B,C)`** to compute the area of the triangle with corners at the points A, B, and C.

Write a function **`parea(P,n)`** to compute the area of a polygon P with n sides. Triangulate the polygon, and add the areas of the triangles. A scheme is suggested below.



Part 2: The Graphical Method

Recall that our region of interest is restricted to the unit square U with corners $(0,0)$, $(1,0)$, $(1,1)$, and $(0,1)$. Choose some positive integer N, and decompose U into an $N \times N$ grid of squares each of size $\frac{1}{N} \times \frac{1}{N}$. Treat each cell in the grid as a pixel. Count those of the N^2 pixels, the centers of which lie inside both the quadrilaterals. If the count is k, then the area of the intersection of R_1 and R_2 is approximated as $\frac{k}{N^2}$. As $N \to \infty$, this approximation approaches the actual area of intersection. However, this method requires running time proportional to N^2. So you can feasibly take N no larger than 10^4.

Write a function **`aoi1(R1,R2)`** to implement this method.

Part 3: The Numerical Method

This method is similar to numeric integration. Again, take a positive integer N, and break U into N vertical strips, each of height 1 and width $\frac{1}{N}$. Consider the vertical line passing through the center of each strip. Compute the points of intersection of this vertical line with the sides of the two quadrilaterals. This gives the amount of overlap (in one dimension, may be zero) at the center of the strip. Multiply this by the width $\frac{1}{N}$ of the strip to get the contribution of that strip to the intersection area. Add up the contributions of the N strips to obtain an approximate value for the area of intersection of R_1 and R_2. As $N \to \infty$, this approximation approaches the correct intersection area. For this method, the running time is proportional to N, so it is feasible to take N as large as 10^8.

Write a function **`aoi2(R1,R2)`** to implement this method.

Part 4: The Geometric Method

The winner enters the scene at the end. This method is guaranteed to give an almost exact computation of the intersection area. However, since floating-point arithmetic is involved, this will still lead to some small errors (this is unavoidable in *real* calculations). But most importantly, this method runs in constant time.

First, compute all the points of intersections of the sides of R_1 with the sides of R_2. Here, sides are to be treated as segments (not lines). Let there be l such points of side intersection. We have $0 \leqslant l \leqslant 8$. All these l points will be corners of the desired intersection polygon I. But the intersection polygon may contain one or more corners from one or both of the input quadrilaterals. You need to insert these quadrilateral corners in between the l side-intersection points computed above.

Arrange the l side-intersection points in the counterclockwise order (starting from any one of them). In order to do that, obtain the average (center of mass) of these l points, compute the angles of the l points with respect to the average (use **`atan2`**), and sort the l points with respect to these angles.

You now have three lists of points, all arranged in the counterclockwise order. The l side-intersection points, the four corners of R_1, and the four corners of R_2. By the next point in each list, we mean the next point in the counterclockwise direction in that list. Moreover, let P be a side-intersection point resulting from the intersection of the side s_1 of R_1 with the side s_2 of R_2. The corner of R_1 next to P is that endpoint of s_1, that follows P in the counterclockwise orientation. Likewise, the corner of R_2 next to P is defined.

Let us first handle the special case $l = 0$. This happens when either the quadrilaterals are disjoint or one of these is completely contained in the other. In the first case, all corners of each quadrilateral must lie outside the other quadrilateral, whereas in the second, all four of the corners of one quadrilateral lie inside the other (checking for one suffices). For disjoint quadrilaterals, the area of intersection is zero, whereas in the other case with $l = 0$, it is the area of the contained quadrilateral.

Now, suppose that $l > 0$. We build the list I of the corners of the intersection polygon of R_1 and R_2. Initialize I to empty. Add any side-intersection point to I (so now I contains only one corner at this time). Repeat the following until you have come back to the first (side-intersection) point added to I. You always need to remember the last side-intersection point added to I.

Let P be the last point (corner of the intersection polygon) added to I. This point may be one of the three: (i) a side-intersection point, (ii) a corner of R_1, and (iii) a corner of R_2. These cases are handled separately.

In Case (i), let $P_{1,next}$ and $P_{2,next}$ be the corners in R_1 and R_2 next to the side-intersection point P in the counterclockwise orientation (described above). There are three possibilities (the last two cannot happen together). First, if $P_{1,next}$ lies outside R_2, and $P_{2,next}$ lies outside R_1, append the next side-intersection point to I (you know the last side-intersection point added to I, don't you? Record it again for you have done the same thing once more). Second, if P_1 lies inside R_2, append P_1 to I. Third, if P_2 lies inside R_1, add P_2 to I.

Let us now look at Case (ii), that is, the last point P added to I is a corner of R_1. Let $P_{1,next}$ be the corner of R_1 next to P (in the counterclockwise direction). If $P_{1,next}$ lies inside R_2, then append $P_{1,next}$ to I. Otherwise, append the next side-intersection point to I (you remembered the last such point added).

Case (iii) can be handled *mutatis mutandis* like Case (ii).

Well done. You now have the list I of the corners of the intersection polygon. Call **`parea`**. You guessed correctly that the function implementing this method would be named **`aoi3(R1,R2)`**.

But wait! Don't submit yet. You still do not have the **`main()`** thing.

The *main*() function

- Read the four corners (x- and y-coordinates) of each input quadrilateral R_1 and R_2 from the user. It is the responsibility of the user (who, in your case, is your teachers) to supply valid input. That means that the four corners supplied for each quadrilateral must define a simple and convex quadrilateral residing inside the unit square U. Moreover, the corners must appear in the counterclockwise order. The listing of corners of each quadrilateral may, however, start from any corner.
- For $N = 10^{\lambda}$, $1 \leqslant \lambda \leqslant 3$, call and report the findings of **`aoi1()`**.
- For $N = 10^{\lambda}$, $1 \leqslant \lambda \leqslant 6$, call and report the findings of **`aoi2()`**.
- Call **`aoi3()`**, and report what you get. Our sample output shows the side-intersection points, and the points of I. Why don't you print those as well?
- It may hurt, but start learning how to comment your code. We are fussier about indentation though.

Samples

These are supplied in a standalone text file linked from the lab website.

Submit a single C/C++ source file. Do not use global/static variables.