# 20 Plane-sweep

Plane-sweep is an algorithmic paradigm that emerges in the study of two-dimensional geometric problems. The idea is to sweep the plane with a line and perform the computations in the sequence the data is encountered. In this section, we solve three problems with this paradigm: we construct the convex hull of a set of points, we triangulate the convex hull using the points as vertices, and we test a set of line segments for crossings.

**Convex hull.** Let $S$ be a finite set of points in the plane, each given by its two coordinates. The *convex hull* of $S$, denoted by $\mathrm{conv}\, S$, is the smallest convex set that contains $S$. Figure 91 illustrates the definition for a set of nine points. Imagine the points as solid nails in a planar board. An intuitive construction stretches a rubber band around the nails. After letting go, the nails prevent the complete relaxation of the rubber band which will then trace the boundary of the convex hull.

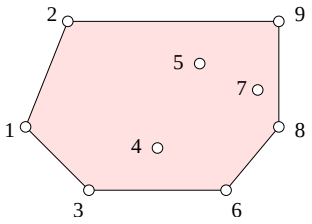


Figure 91: The convex hull of nine points, which we represent by the counterclockwise sequence of boundary vertices: 1, 3, 6, 8, 9, 2.

To construct the counterclockwise cyclic sequence of boundary vertices representing the convex hull, we sweep a vertical line from left to right over the data. At any moment in time, the points in front (to the right) of the line are untouched and the points behind (to the left) of the line have already been processed.

`Step 1.` Sort the points from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_n$.

`Step 2.` Construct a counterclockwise triangle from the first three points: $x_1x_2x_3$ or $x_1x_3x_2$.

`Step 3.` For $i$ from 4 to $n$, add the next point $x_i$ to the convex hull of the preceding points by finding the two lines that pass through $x_i$ and support the convex hull.

The algorithm is illustrated in Figure 92, which shows the addition of the sixth point in the data set.

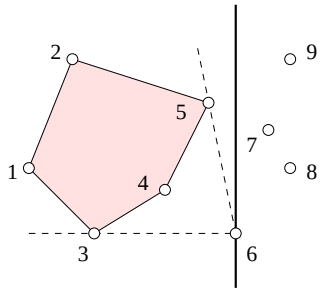


Figure 92: The vertical sweep-line passes through point 6. To add 6, we substitute 6 for the sequence of vertices on the boundary between 3 and 5.

**Orientation test.** A critical test needed to construct the convex hull is to determine the orientation of a sequence of three points. In other words, we need to be able to distinguish whether we make a left-turn or a right-turn as we go from the first to the middle and then the last point in the sequence. A convenient way to determine the orientation evaluates the determinant of a three-by-three matrix. More precisely, the points $a = (a_1, a_2)$, $b = (b_1, b_2)$, $c = (c_1, c_2)$ form a left-turn iff

$$\det \begin{bmatrix} 1 & a_1 & a_2 \\ 1 & b_1 & b_2 \\ 1 & c_1 & c_2 \end{bmatrix} > 0.$$

The three points form a right-turn iff the determinant is negative and they lie on a common line iff the determinant is zero.

```
boolean LEFT(Points a, b, c)
  return [a1(b2 − c2) + b1(c2 − a2)
                      + c1(a2 − b2) > 0].
```

To see that this formula is correct, we may convince ourselves that it is correct for three non-collinear points, e.g. $a = (0, 0)$, $b = (1, 0)$, and $c = (0, 1)$. Remember also that the determinant measures the area of the triangle and is therefore a continuous function that passes through zero only when the three points are collinear. Since we can continuously move every left-turn to every other left-turn without leaving the class of left-turns, it follows that the sign of the determinant is the same for all of them.

**Finding support lines.** We use a doubly-linked cyclic list of vertices to represent the convex hull boundary. Each

node in the list contains pointers to the next and the previous nodes. In addition, we have a pointer $last$ to the last vertex added to the list. This vertex is also the rightmost in the list. We add the $i$-th point by connecting it to the vertices $\mu \to pt$ and $\lambda \to pt$ identified in a counterclockwise and a clockwise traversal of the cycle starting at $last$, as illustrated in Figure 93. We simplify notation by using

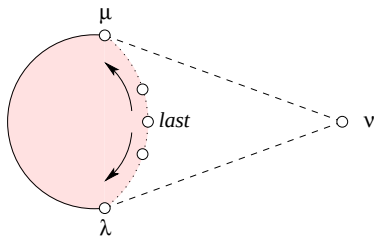


Figure 93: The upper support line passes through the first point $\mu \to pt$ that forms a left-turn from $\nu \to pt$ to $\mu \to next \to pt$.

nodes in the parameter list of the orientation test instead of the points they store.

$\mu = \lambda = last$; create new node with $\nu \to pt = i$;
`while` RIGHT$(\nu, \mu, \mu \to next)$ `do`
  $\mu = \mu \to next$
`endwhile`;
`while` LEFT$(\nu, \lambda, \lambda \to prev)$ `do`
  $\lambda = \lambda \to prev$
`endwhile`;
$\nu \to next = \mu$; $\nu \to prev = \lambda$;
$\mu \to prev = \lambda \to next = \nu$; $last = \nu$.

The effort to add the $i$-th point can be large, but if it is then we remove many previously added vertices from the list. Indeed, each iteration of the for-loop adds only one vertex to the cyclic list. We charge \$2 for the addition, one dollar for the cost of adding and the other to pay for the future deletion, if any. The extra dollars pay for all iterations of the while-loops, except for the first and the last. This implies that we spend only constant amortized time per point. After sorting the points from left to right, we can therefore construct the convex hull of $n$ points in time O($n$).

**Triangulation.** The same plane-sweep algorithm can be used to decompose the convex hull into triangles. All we need to change is that points and edges are never removed and a new point is connected to every point examined during the two while-loops. We define a *(geometric) triangulation* of a finite set of points $S$ in the plane as a maximally connected straight-line embedding of a planar graph whose vertices are mapped to points in $S$. Figure 94 shows the triangulation of the nine points in Figure 91 constructed by the plane-sweep algorithm. A triangulation is

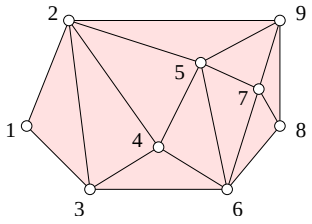


Figure 94: Triangulation constructed with the plane-sweep algorithm.

not necessarily a maximally connected planar graph since the prescribed placement of the points fixes the boundary of the outer face to be the boundary of the convex hull. Letting $k$ be the number of edges of that boundary, we would have to add $k - 3$ more edges to get a maximally connected planar graph. It follows that the triangulation has $m = 3n - (k + 3)$ edges and $\ell = 2n - (k + 2)$ triangles.

**Line segment intersection.** As a third application of the plane-sweep paradigm, we consider the problem of deciding whether or not $n$ given line segments have pairwise disjoint interiors. We allow line segments to share endpoints but we do not allow them to cross or to overlap. We may interpret this problem as deciding whether or not a straight-line drawing of a graph is an embedding. To simplify the description of the algorithm, we assume no three endpoints are collinear, so we only have to worry about crossings and not about other overlaps.

How can we decide whether or not a line segment with endpoint $u = (u_1, u_2)$ and $v = (v_1, v_2)$ crosses another line segment with endpoints $p = (p_1, p_2)$ and $q = (q_1, q_2)$? Figure 95 illustrates the question by showing the four different cases of how two line segments and the lines they span can intersect. The line segments cross iff $uv$ intersects the line of $pq$ and $pq$ intersects the line of $uv$. This condition can be checked using the orientation test.

`boolean` CROSS(`Points` $u, v, p, q$)
  `return` [(LEFT$(u, v, p)$ `xor` LEFT$(u, v, q)$) `and`
    (LEFT$(p, q, u)$ `xor` LEFT$(p, q, v)$)].

We can use the above function to test all $\binom{n}{2}$ pairs of line segments, which takes time O($n^2$).

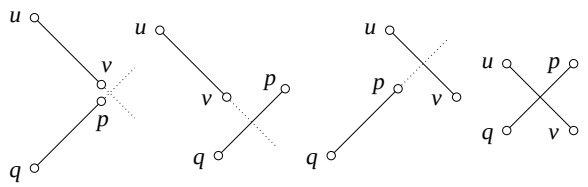


Figure 95: Three pairs of non-crossing and one pair of crossing line segments.

**Plane-sweep algorithm.** We obtain a faster algorithm by sweeping the plane with a vertical line from left to right, as before. To avoid special cases, we assume that no two endpoints are the same or lie on a common vertical line. During the sweep, we maintain the subset of line segments that intersect the sweep-line in the order they meet the line, as shown in Figure 96. We store this subset

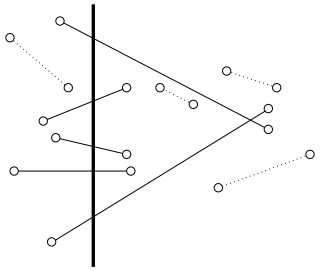

Figure 96: Five of the line segments intersect the sweep-line at its current position and two of them cross.

in a dictionary, which is updated at every endpoint. Only line segments that are adjacent in the ordering along the sweep-line are tested for crossings. Indeed, two line segments that cross are adjacent right before the sweep-line passes through the crossing, if not earlier.

`Step 1.` Sort the $2n$ endpoints from left to right and relabel them in this sequence as $x_1, x_2, \ldots, x_{2n}$. Each point still remembers the index of the other endpoint of its line segment.

`Step 2.` For $i$ from 1 to $2n$, process the $i$-th endpoint as follows:

`Case 2.1` $x_i$ is left endpoint of the line segment $x_ix_j$. Therefore, $i < j$. Insert $x_ix_j$ into the dictionary and let $uv$ and $pq$ be its predecessor and successor. If $\textsc{Cross}(u, v, x_i, x_j)$ or $\textsc{Cross}(p, q, x_i, x_j)$ then report the crossing and stop.

`Case 2.2` $x_i$ is right endpoint of the line segment $x_ix_j$. Therefore, $i > j$. Let $uv$ and $pq$ be the predecessor and the successor of $x_ix_j$. If $\textsc{Cross}(u, v, p, q)$ then report the crossing and stop. Delete $x_ix_j$ from the dictionary.

We do an insertion into the dictionary for each left endpoint and a deletion from the dictionary for each right endpoint, both in time $O(\log n)$. In addition, we do at most two crossing tests per endpoint, which takes constant time. In total, the algorithm takes time $O(n \log n)$ to test whether a set of $n$ line segments contains two that cross.