Chapter 4

Geometric Sweeping

Geometric sweeping technique is a generalization of a technique called *plane sweeping*, which is primarily used for 2-dimensional problems. In most cases, we will illustrate the technique for 2-dimensional cases. The generalization to higher dimensions is straightforward. This technique is also known as the *scan-line method* in computer graphics, and is used for a variety of applications, such as shading, polygon filling, among others.

The technique is intuitively simple. Suppose that we have a line in the plane. To collect the geometric information we are interested in, we slide the line in some way so that the whole plane will be "scanned" by the line. While the line is sweeping the plane, we stop at some points and update our recording. We continue this process until all interesting objects are collected.

There are two basic structures associated with this technique. One is for the *sweeping line status*, which is an appropriate description of the relevant information of the geometric objects at the sweeping line, and the other is for the *event points*, which are the places we should stop and update our recording. Note that the structures may be implemented in different data structures under various situations. In general, the data structures should support efficient operations that are necessary for updating the structures while the line is sweeping the plane.

4.1 Intersection of line segments

The geometric sweeping technique can be best illustrated by the following example. Recall the Segment-Intersection problem:

> **Segment-Intersection**:
> Given n line segments in the plane, find all intersections.

Suppose that we have a vertical line L that is used to sweep the plane from left to right. At every moment, the *sweeping line status* contains all segments intersecting the line L, sorted by the y-coordinates of their intersecting points with L. The sweeping line status is modified whenever one of the following three cases occurs:

1. The sweeping line L hits the left-end of a segment s. In this case, the segment s was not seen before and it may have intersections with other segments on the right side of the sweeping line L, so the segment s should be added to the sweeping line status;

2. The sweeping line L hits the right-end of a segment s. In this case, the segment s cannot have any intersections with other segments on the right side of the sweeping line L, so the segment s can be deleted from the sweeping line status;

3. The sweeping line L hits an intersection of two segments s_1 and s_2. In this case, the relative positions of the segments s_1 and s_2 in the sweeping line status should be swapped, since the segments in the sweeping line status are sorted by the y-coordinates of their intersection points with the line L.

It is easy to see that the sweeping line status of the sweeping line L will not be changed when it moves from left to right unless it hits either an endpoint of a segment or an intersection of two segments. Therefore, the set of *event points* consists of the endpoints of the given segments and the intersection points of the segments. We sort the event points by their x-coordinates.

We use two data structures `EV` and `ST` to store the event points and the sweeping line status, respectively, such that the set operations Minimum, Insert, and Delete can be performed efficiently (for example, they can be 2-3 trees). At very beginning, we suppose that the sweeping line L is far enough to the left so that no segments intersect L. At this moment, the sweeping line status `ST` is an empty set. We sort all endpoints of the segments by their x-coordinates and store them in the event point set `EV`. These are the event points at which the sweeping line L should stop and update the sweeping line status `ST`. However, the list is not complete since an intersection point of two segments should also be an event point. Unfortunately, these points are unknown to we at beginning. For this, we update the structure `EV` in the following way. Whenever we find an intersection point of two segments while the line L is sweeping the plane, we add the intersection point to `EV`. But how

do we find these intersection points? Note that if the next event point to be hit by the sweeping line L is an intersection point of two segments s_i and s_j, then the segments s_i and s_j should be adjacent in the sweeping line status ST. Therefore, whenever we change the adjacency relation in ST, we check for intersection points for new adjacent segments. When the sweeping line L reaches the right-most endpoint of the segments, all possible intersection points are collected.

These ideas are summarized in the following algorithm.

```
Algorithm  Segment-Intersection
Given:  n segments s1, s2, ... sn
Output: all intersections of these segments
/* We use a vertical line L to sweep the plane. At any moment, the segments
   intersecting L are stored in ST, sorted by the y-coordinates of their
   intersection points with the line L. The event points stored in EV are
   sorted by their x-coordinates. */
1.  EV = {};  ST = {};
2.  For (each endpoint p of the segments) Insert(EV, p);
3.  While (EV is not empty) Do
       p = Min(EV);  Delete(EV, p);
       If (p is a right-end of a segment s)
          let si and sj be the two segments adjacent to s in ST;
          If (p is an intersection point of s with si or sj) Report(p);
          Delete(ST, s);
          If ((si and sj intersect at p')&(x(p')>=x(p))) Insert(EV, p');
       Else If (p is a left-end of a segment s)
          Insert(ST, s);
          let si and sj be the adjacent segments of s in ST;
          If (p is an intersection point of s with si or sj) Report(p);
          If (s intersects si at pi) Insert(EV, pi);
          If (s intersects sj at pj) Insert(EV, pj);
       Else If (p is an intersection point of segments si and sj)
          /* suppose that sj follows si in ST */
          Report(p);
          swap the positions of si and sj in ST;
          suppose now sj follows sk and sh follows si in ST;
          If ((sk and sj intersect at q1)&(x(q1)>x(p))) Insert(EV, q1);
          If ((sh and si intersect at q2)&(x(q2)>x(p))) Insert(EV, q2);
```

Let us analyze the algorithm. As we suggested, we can use 2-3 trees for the structures ST and EV so that each of the following operations on the structures takes time $O(\log n)$: searching an element, finding the smallest element, deleting an element, and inserting an element. Thus, step 2 of the algorithm that initializes the structure EV takes time $O(n \log n)$. For step 3, we also need to find the neighboring segments of a given segment in the structure ST. This can be done by developing algorithms for 2-3 trees that find the neighbors of a given element in time $O(\log n)$ (this is not difficult but is an interesting exercise problem for 2-3 tree structures).

To count the time spent by the While loop in step 3, suppose there are

m intersection points for these n segments. In the `While` loop, each segment is inserted then deleted from the structure `ST` exactly once, and each event point is inserted then deleted from the structure `EV` exactly once. There are $n+m$ event points. Since each of the operations `Min`, `Insert`, `Delete`, and finding neighbors of a given segment in `ST` can be done in time $O(\log N)$ on a set of N elements, processing each segment takes time $O(\log n)$ time, and processing each event point takes time $O(\log(n+m))$ time. Therefore, step 3 of the algorithm runs in time

$$n \cdot O(\log n) + (n+m) \cdot O(\log(n+m)) = O((n+m)\log(n+m)).$$

Observe that m is at most n^2, so $\log(n+m) = O(\log n)$. Combining this with the analysis for steps 1-2 of the algorithm, we conclude that the algorithm `Segment-Intersection` runs in time $O((n+m)\log n)$.

We remark that the time complexity of the above algorithm depends on the number m of intersection points of the segments and the algorithm is not always efficient. For example, when the number m is of order $\Omega(n^2)$, then the algorithm runs in time $O(n^2 \log n)$, which is even worse than the straightforward method that picks every pair of segments and computes their intersection point. On the other hand, if the number m is of order $\Omega(n)$, then the algorithm runs efficiently in time $O(n \log n)$.

4.2 Constructing convex hulls

There are many algorithms for constructing convex hulls for point sets, in particular for the case where the point sets are in the 2-dimensional Euclidean space $\mathbf{E}^2$. The two most famous algorithms are based on the technique of plane sweeping, and will be discussed in this section.

4.2.1 Jarvis March

We start with an algorithm that is called *Jarvis March*, which is also known as the *gift wrapping method*.

The idea is based on the observation we gave in the proof of Theorem 3.1.3. Given a set S of n points in the plane, suppose that we move a straight line L sweeping the plane until L hits a point p_1 of S. The point p_1 must be on the boundary of the convex hull $\mathrm{CH}(S)$ of S since at this moment, all points of S are in one side of the line L and the point p_1 is on the line L. Now we rotate the line L around the point p_1, say counterclockwise, until L hits another point p_2 of S. The segment $\overline{p_1p_2}$ is then on the

boundary of the convex hull CH(S) since again all points of S are in one side of the line L and the segment $\overline{p_1p_2}$ is on the line L. Now we rotate the line L around p_2 counterclockwise until L hits a third point p_3 of S, then the line segment $\overline{p_2p_3}$ is the second boundary edge of CH(S). Then we rotate the line L around p_3, and so on. Continue this process until we come back to the first point p_1. The convex hull CH(S) then is constructed.

This process can also be regarded as a "wrapping" process. Suppose that we fix an end of a rope on a point p_1 that is known to be a hull vertex. Then we try to "wind" the points by the rope (or "wrap" the points by the rope). The rope obviously gives us the boundary of the convex hull when it comes back to the point p_1.

There are a few things we should mention in the above process. First of all, the sweeping manner is special: the line L is rotated around a point in the plane; secondly, the sweeping line status is very simple: it contains at any moment a single point that is the hull vertex most recently discovered; finally, the even points are the hull vertices.

Let us study the process in detail. Suppose that at some moment during the process, the consecutive hull vertices that have been found are p_1, p_2, $\cdots$, p_i. What point should be the next hull vertex? Obviously, the point p_{i+1} should be the one that is first touched by the rope when we rotate the rope around the point p_i. That is, the angle $\angle p_{i-1}p_ip_{i+1}$ should be the largest. We implement this idea in the following algorithm.

```
Algorithm Jarvis March
Input: a set S of n points in the plane
Output: the convex hull CH(S) of S
1. let p[1] be the point in the set S with the smallest y-coordinate;
2. let p[2] be the point in the set S such that the slope of the line
   segment {p[1], p[2]} is the smallest among all points in S - p[1];
3. Output(p[1]); Output(p[2]);
4. i = 2 ;
5. While (p[i] <> p[1]) Do
     let p[i+1] be the point in the set S such that the angle
       <(p[i-1]p[i]p[i+1]) is the largest;
     i= i + 1 ;
     Output(p[i]);
```

Let us study the complexity of the algorithm `Jarvis March`. Suppose there are k hull vertices in the convex hull CH(S) of the point set S. The points `p[1]` and `p[2]` are obviously hull vertices. Moreover, it is clear that to find the points `p[1]` and `p[2]` takes time $O(n)$, assuming S has n points. To find each next hull vertex `p[i+1]`, we check the angle $\angle$`p[i-1]p[i]p` for each point `p` in the set S. Thus, step 5 spends time $O(n)$ on each hull vertex. In conclusion, the algorithm `Jarvis March` runs in time $O(kn)$.

If k is small compared with n, for instance, if k is bounded by a constant, then the algorithm `Jarvis March` runs in linear time. On the other hand, if k is large, such as $k = \Omega(n)$, then `Jarvis March` runs in time $\Omega(n^2)$.

4.2.2 Graham Scan

Consider the algorithm `Jarvis March`. Suppose that the most recent hull vertex is p and the most recent hull edge is e. We find the next hull vertex by choosing the point q that makes the angle between e and $\overline{pq}$ the largest. To find the point q, we compute the angle between the segments e and $\overline{pp'}$ for every point p' in the set S. For each hull vertex, we apply this process to find the next hull vertex. In this process, even though we have found out that a point p' is not qualified for the next hull vertex, we still cannot exclude the possibility that the point p' is qualified for a later hull vertex. This is the reason that we have to consider the point again and again. A point can be considered up to n times in the worst case. A possible improvement is that we presort the set of points in some way so that once we find that a point is not qualified for the next hull vertex, we can exclude the point forever. For example, let p_0, p_1 and p_2 be three distinct hull vertices of the convex hull $\mathrm{CH}(S)$ for the set S. Suppose that the line segment $\overline{p_1p_2}$ is known to be on the boundary of the convex hull $\mathrm{CH}(S)$. Then the line segments $\overline{p_0p}$ for all points p of S that are between the angle $\angle p_1p_0p_2$ should be entirely in the triangle $\Delta p_0p_1p_2$. Therefore, if we start with the point p_1, and rotate the segment $\overline{p_0p_1}$ around the point p_0 counterclockwise to sweep the plan, then once we reach the point p_2, we can eliminate all points we have visited between the points p_1 and p_2. This elimination is permanent, i.e., once a point is eliminated, it will be ignored forever.

The above idea is implemented by the following well-known algorithm, known as *Graham Scan*.

```
Algorithm Graham-Scan
Input: a set S of n points in the plane
Output: the convex hull CH(S) of S
1. assume point p[0] in S with the smallest y-coordinate is the origin;
   /* if this is not the case, make a coordinate transformation. */
2. sort the points in the set S-p[0] by their polar angles:
    L = {p[1], p[2], ..., p[n-1]}  /* in increasing ordering */
3. Push(K,p[0]); Push(K,p[1]);  /* K is a stack */
4. For (i = 2; i < n; i++)
   /* K[1] and K[2] are the 1st and 2nd vertices on the top of K. */
     While (K[2]K[1]p[i] is a right-turn) Pop(K);
     Push(K,p[i]).
```

In `Graham-Scan`, the sweeping line rotates around a fixed point `p[0]`. All points in the set `S` are event points. Since the event points are presorted

in Step 2, it takes only constant time to find the next event point in the sorted list `L`. This makes `Graham-Scan` very efficient.

We make a few remarks on the algorithm `Graham-Scan`. First of all, there can be more than one point in the set `S` that are on the same ray from the point `p[0]`. This will cause no problem for the algorithm. However, if we want to be really specific, we can order the points first by their polar angles then by their distances to the point `p[0]`. Thus, with the same polar angle, points closer to `p[0]` will be considered first. Note that when a further point is considered, the points with the same polar angle but shorter distances will be right-turns and popped out from the stack `K`, which, obviously, is correct. Secondly, three points `K[2]K[1]p[i]` that make a straight line will be regarded as a right-turn so that step 4 of the algorithm will pop the point `K[1]` out of the stack `K`. This is correct since in this case, if `K[1]` is on the boundary of the convex hull $\mathrm{CH}(S)$ then the segment `K[2]p[i]` will be on the boundary of $\mathrm{CH}(S)$ so `K[1]` cannot be a hull vertex.

To show the correctness of the algorithm `Graham-Scan`, we first prove that a point p in S is a hull vertex *if and only if* for any two points p_1 and p_2 in S, where the polar angle of p is larger than that of p_1 but smaller than that of p_2, the segments p_1pp_2 is always a left-turn. First of all, if for some such two points p_1 and p_2, the segments p_1pp_2 is a right-turn, then the point p will be contained in the triangle given by the three points p_1, p_2, and `p[0]` in S, thus cannot be on the boundary of $\mathrm{CH}(S)$. On the other hand, if p_1pp_2 is always a left-turn for all such points p_1 and p_2 in S, then pick the points p_1' and p_2' in S such that the angle $\angle p_1'pp_2'$ is the largest, then all points in S are contained in the wedge between the ray from p that contains the point p_1' and the ray from p that contains the point p_2', and $p_1'pp_2'$ is a left-turn. That is, the point p must be a hull vertex.

Thus, in the algorithm `Graham-Scan`, a hull vertex can never be popped out from the stack `K` in step 4. We should also show that if a point is not a hull vertex, then it will be, sooner or later, popped out from the stack `K`. For this, let p_1 and p_2 be the two consecutive hull vertices. Then every point p in S whose polar angle is between that of p_1 and p_2 will make p_1pp_2 a right-turn. Thus, when we sweep from the ray from `p[0]` that contains p_1, the point p will sooner or later become a right-turn and get popped out from the stack `K`. Note that p may become a right-turn not because of the points p_1 and p_2 but because of two points p_1' and p_2' that are between p_1 and p_2. The point is that if these points p_1' and p_2' do not exist, then the points p_1 and p_2 will eventually make p a right-turn (note that, by the fact we proved above, the point p_1 will never be popped out of the stack).

Now we consider the time complexity of the algorithm `Graham-Scan`.

Step 1 that finds the point with the smallest y-coordinate in the set `S` can be done by comparing the y-coordinates of all points in the set `S`, thus it takes time $O(n)$. Step 2 can be done by any $O(n \log n)$-time sorting algorithm such as MergeSort. Step 3 obviously takes constant time. To discuss the time complexity of step 4, observe that each point of the set `S` is pushed into the stack `K` once and then may be popped out of the stack later. Whenever a point is popped out from the stack `K`, it will never be pushed into the stack again in later process. Thus, each point of `S` can cause at most one push and one pop on the stack `K`, and there are in total at most $2n$ stack pushes and pops in step 4 of the algorithm. Since each push and pop on the stack takes time $O(1)$, the total time taken by step 4 is bounded by $O(n)$. In summary, the time complexity of the algorithm `Graham-Scan` is $O(n \log n)$.

We remark that most of the time in the algorithm `Graham-Scan` is spent on step 2's sorting. Besides sorting, `Graham-Scan` runs in linear time.

Step 2 of the algorithm `Graham-Scan` sorts the points in the given set `S` by their polar angles. This involves in trigonometric operations. Although we have assumed that our RAM model can perform trigonometric operations in constant time per operation, trigonometric operations can be time-consuming in a real computer. We present a modified version of `Graham-Scan` that avoids using trigonometric operations.

The idea is as follows. Suppose that we are given a set S of n points in the plane. We add a new point p_0 to the set S such that p_0's y-coordinate is smaller than that of any point in the set S. Then we perform Graham Scan on this new set. Draw a line segment $\overline{p_0 p}$ for each point p in the set S. It can be easily seen that if the point p_0 moves toward the negative direction of the y-axis, these line segments are getting more and more parallel each other. Imagining that eventually the point p_0 reaches the infinite point along the negative direction of the y-axis, then all these line segments become vertical rays originating from the points of the set S. Now the ordering of the polar angles of the points of S around p_0 is identical with the ordering of the x-coordinates of these points. (In fact, p_0 does not have to be the infinite point, when p_0 is far enough from the set S, the above statement should be true.) Therefore, the convex hull of the new set can be constructed by first sorting the points in S by their x-coordinates instead of their polar angles. It is also easy to see that the convex hull of the new set consists of two vertical rays, originating from the two points $p_{\min}$ and $p_{\max}$ in the set S with the smallest and the largest x-coordinates, respectively, and a part H_U of the convex hull $\mathrm{CH}(S)$ of the original set S. This part H_U of the convex hull $\mathrm{CH}(S)$ is in fact the *upper hull* of the convex hull $\mathrm{CH}(S)$ in the sense that all points of the set S lie between the vertical lines $x = x_{\min}$ and

$x = x_{max}$ and below the part H_U. Similarly, the *lower hull* of the convex hull CH(S) can be constructed by the idea of adding an infinite point in the positive direction of the y-axis. The convex hull CH(S) is then simply the circular catenation of the upper hull and the lower hull.

The algorithm is presented as follows.

```
Algorithm Modified-Graham-Scan
Input: a set S of n points in the plane
Output: the convex hull CH(S) of S;
1. L = the points in S sorted in decreasing x-coordinate ordering;
2. pmax = the point (x,y) in S with the largest x-coordinate;
3. let p+ = (x,y-1); Push(K,p+); Push(K,pmax); /* K is a stack */
4. perform Graham Scan using the sorted list L;
   /* the resulting list minus the point p+ is the upper hull. */
5. construct the lower hull using the point in S with the smallest
   x-coordinate and the list L reversed;
6. catenate the upper and lower hulls to form the convex hull CH(S).
```

It is obvious that algorithm `Modified-Graham-Scan` runs in time $O(n \log n)$.

4.3 The farthest pair problem

The problem we shall discuss in this section is formally defined as follows:

> **Farthest-Pair**
> Find a pair of points in a given set S of points in $\mathbf{E}^2$ whose distance is the largest among all pairs of points in S.

A brute force algorithm is to examine every pair of points in S to find the pair with the largest distance. Thealgorithm obviously runs in time $O(n^2)$.

To get a more efficient algorithm, let us first investigate what kind of properties a farthest point pair in a set has. Let us suppose that S is a set of n points in the plane, and call a segment linking two farthest points in the set S a *diameter* of the set S.

Lemma 4.3.1 *Let $\overline{uv}$ be a diameter of a set S of points in the plane. Let l_u and l_v be two straight lines that are perpendicular to the segment $\overline{uv}$ such that l_u contains u and l_v contains v. Then all points of S are contained in the slab between the lines l_u and l_v.*

PROOF. Without loss of generality, suppose that the segment $\overline{uv}$ is horizontal and the point u is on the left of the point v. Draw a circle C centered at u of radius $|\overline{uv}|$, then the line l_v is tangent to C because l_v is perpendicular to $\overline{uv}$. Thus the circle C is entirely on the left of the line l_v. Since v is the

farthest point in the set S from the point u, all points of S are contained in the circle C. Consequently, all points of S are on the left of the line l_v. Similarly, we can prove that all points of S are on the right of the line l_u. Therefore, all points of the set S are between the lines l_u and l_v. □

Corollary 4.3.2 *Let $\overline{uv}$ be a diameter of a set S of points in the plane, then the points u and v are hull vertices of $CH(S)$.*

PROOF. As we discussed in Chapter 2, a point p in S is a hull vertex of $\mathrm{CH}(S)$ if and only if there is a line passing through p such that all points of S are on one side of the line. □

Two hull vertices u and v of the convex hull $\mathrm{CH}(S)$ are called an *antipodal pair* if we can draw two parallel supporting lines l_u and l_v of $\mathrm{CH}(S)$ such that l_u passes through u and l_v passes through v, and the convex hull $\mathrm{CH}(S)$ is entirely contained in the slab between the lines l_u and l_v.

Corollary 4.3.3 *Let $\overline{uv}$ be a diameter of the set S, then u and v make an antipodal pair.*

PROOF. By Corollary 4.3.2, the points u and v are hull vertices of $\mathrm{CH}(S)$. By Lemma 4.3.1, we can draw two parallel lines l_u and l_v such that l_u passes through u, that l_v passes through v, and that all points of S are contained in the slab between l_u and l_v. The slab between l_u and l_v is clearly a convex set. Since the convex hull $\mathrm{CH}(S)$ of S is the smallest convex set containing all points of S, i.e., the convex hull $\mathrm{CH}(S)$ is contained in all convex sets that contain all points of S, the convex hull $CH(S)$ is contained in the slab between the lines l_u and l_v. □

According to Lemma 4.3.1 and its corollaries, to find a farthest pair of a set S of n points in the plane, we only need to find a farthest pair among the hull vertices of the convex hull $\mathrm{CH}(S)$. Moreover, we only need to consider antipodal pairs on the convex hull $\mathrm{CH}(S)$. This greatly simplifies our problem. For this, we now consider the following problem: given a vertex u of a convex polygon P, which vertices of the polygon P can constitute an antipodal pair with the vertex u?

To answer this question, we first make some conventions and introduce some terminologies. We suppose that the vertices of the convex polygon P are given in counterclockwise ordering: $\{u_0, u_1, \ldots, u_{m-1}\}$. We may write a vertex of P as u_i where i is larger than $m-1$ or smaller than 0 – in this

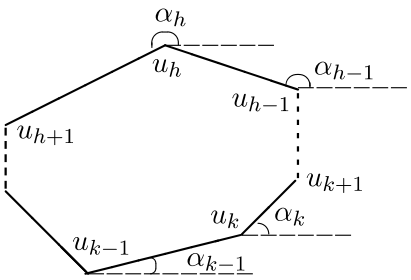

Figure 4.1: The convex polygon P

case, the index i is interpreted as $i \pmod m$. For each vertex u_i of P, let α_i be the angle from the x-axis to the edge $[u_i, u_{i+1}]$ (directed from u_i to u_{i+1}). Thus, we have $0 \leq \alpha_i < 2\pi$ for all i. See Figure 4.1 for an illustration. Again, we may write an angle α_i whose value is outside of the interval $[0, 2\pi)$ – in this case the angle α_i is interpreted as $\alpha_i \pmod{2\pi}$.

A vertex u_i of P is a *farthest* vertex from an edge $[u_{k-1}, u_k]$ of P if among all vertices of P, u_i is the farthest from the straight line that contains the edge $[u_{k-1}, u_k]$. A vertex u_i is the *first* (resp. *last*) farthest vertex from $[u_{k-1}, u_k]$ if u_i is the first (resp. last) farthest vertex from $[u_{k-1}, u_k]$ when we traverse the polygon P, counterclockwise, starting from the vertex u_k.

For each supporting line on the convex polygon P, we assign it an angle, as follows. First, draw a horizontal line L passing through the vertex of P with the smallest y-coordinate, and give the line L the same direction as that of the x-axis. Then rotate the line L around the convex polygon P, counterclockwise, keeping the line in touch with the polygon P. Thus, for every supporting line L' of P, the line L will become L' at some moment during this process. We then define the direction of the supporting line L' be to that of the corresponding line L at that moment. Now the *angle* of the supporting line L' is defined to be the angle from the x-axis to the supporting line L' with the given direction. Note that the angle of a supporting line is between 0 and 2π. In particular, if a supporting line L' contains an edge $[u_{i-1}, u_i]$ of the polygon P, then the angle of L' is equal to α_{i-1}.

Lemma 4.3.4 *Two vertices u_k and u_h of the convex polygon P make an antipodal pair if and only if the angle interval $[\alpha_{h-1}, \alpha_h]$ and the angle interval $[\pi + \alpha_{k-1}, \pi + \alpha_k]$ intersect.*

PROOF. For any i, the angle of the supporting line of P that contains the edge $[\alpha_{i-1}, \alpha_i]$ is α_{i-1}, and the angle of the supporting line of P that contains

the edge $[\alpha_i, \alpha_{i+1}]$ is α_i. Thus, a supporting line of P contains the vertex u_i if and only if its angle is in the angle interval $[\alpha_{i-1}, \alpha_i]$.

By the definition, the two vertices u_k and u_h make an antipodal pair if and only if there exist a supporting line L_k at u_k and a supporting line L_h at u_h such that L_k and L_h are in parallel, and that the polygon P is entirely contained in the slab between L_k and L_h. By the concept of the angle of a supporting line of P as we introduced earlier, this means that the angle of L_h is π plus the angle of L_k. Since the angle of L_k is in the angle interval $[\alpha_{k-1}, \alpha_k]$, the angle of L_h must be in the angle interval $[\pi + \alpha_{k-1}, \pi + \alpha_k]$. Since the angle of L_h is in the angle interval $[\alpha_{h-1}, \alpha_h]$, this means that the angle interval $[\alpha_{h-1}, \alpha_h]$ and the angle interval $[\pi+\alpha_{k-1}, \pi+\alpha_k]$ intersect. On the other hand, if α is a common angle in the intersection of the angle interval $[\alpha_{h-1}, \alpha_h]$ and the angle interval $[\pi+\alpha_{k-1}, \pi+\alpha_k]$, then we have a supporting line L'_h of angle α at u_h (because α is in $[\alpha_{h-1}, \alpha_h]$) and a support line L'_k of angle $\pi+\alpha$ at u_k (because $\pi+\alpha$ is in $[\pi+(\pi+\alpha_{k-1}), \pi+(\pi+\alpha_k)] = [\alpha_{k-1}, \alpha_k]$) where L'_h and L'_k are in parallel and the polygon P is contained in the slab between L'_h and L'_k, i.e., the vertices u_k and u_h make an antipodal pair. □

Lemma 4.3.4 leads to the following interesting observation.

Corollary 4.3.5 *Let u_h be the first farthest vertex from the edge $[u_{k-1}, u_k]$, and let u_r be the last farthest vertex from the edge $[u_k, u_{k+1}]$ of the polygon P. Then a vertex u_i makes an antipodal pair with the vertex u_k if and only if u_i is a vertex in the sequence $\{u_h, u_{h+1}, \ldots, u_{r-1}, u_r\}$ from u_h to u_r.*

PROOF. Let u_i be any vertex in the vertex sequence u_{k+1}, …, u_{h-1}. Since u_i is not a farthest vertex from the edge $[u_{k-1}, u_k]$, the angle difference $\alpha_i - \alpha_{k-1}$ must be smaller than π (because the edge $[u_i, u_{i+1}]$ is leading from vertex u_i to a farther vertex u_{i+1} from the edge $[u_{k-1}, u_k]$). Thus, $\pi + \alpha_{k-1} > \alpha_i$. Since $\alpha_k > \alpha_{k-1}$ and $\alpha_i > \alpha_{i-1}$, we get

$$\alpha_{i-1} < \alpha_i < \pi + \alpha_{k-1} < \pi + \alpha_k.$$

Thus, the angle intervals $[\alpha_{i-1}, \alpha_i]$ and $[\pi + \alpha_{k-1}, \pi + \alpha_k]$ do not intersect. By Lemma 4.3.4, vertex u_i cannot make an antipodal pair with vertex u_k.

Similarly, Let u_j be any vertex in the vertex sequence u_{r+1}, …, u_{k-1}. Since u_j is not a farthest vertex from the edge $[u_k, u_{k+1}]$, the angle difference $\alpha_{j-1} - \alpha_k$ must be larger than π (the edge $[u_{j-1}, u_j]$ is leading from vertex u_{j-1} to the vertex u_j closer to the edge $[u_{k-1}, u_k]$). Thus, $\pi + \alpha_k < \alpha_{j-1}$. Since $\alpha_{k-1} < \alpha_k$ and $\alpha_j > \alpha_{j-1}$, we get

$$\pi + \alpha_{k-1} < \pi + \alpha_k < \alpha_{j-1} < \alpha_j,$$

i.e., the angle intervals $[\alpha_{j-1}, \alpha_j]$ and $[\pi+\alpha_{k-1}, \pi+\alpha_k]$ do not intersect. By Lemma 4.3.4, vertex u_j cannot make an antipodal pair with vertex u_k.

Thus, if a vertex is not in the sequence $\{u_h, u_{h+1}, \ldots, u_{r-1}, u_r\}$, it cannot make an antipodal pair with u_k.

Now consider any vertex u_t from the sequence $\{u_h, u_{h+1}, \ldots, u_{r-1}, u_r\}$.

If u_t is a farthest vertex from the edge $[u_{k-1}, u_k]$, then the vertex u_{t-1} cannot be farther than vertex u_t from the edge $[u_{k-1}, u_k]$, so $\alpha_{t-1}-\alpha_{k-1} \leq \pi$. On the other hand, the vertex u_t cannot be closer than the vertex u_{t+1} to the edge $[u_{k-1}, u_k]$ so $\alpha_t-\alpha_{k-1} \geq \pi$. As a result, the angle interval $[\alpha_{t-1}, \alpha_t]$ contains the angle $\pi+\alpha_{k-1}$ that is also in the angle interval $[\pi+\alpha_{k-1}, \pi+\alpha_k]$. By Lemma 4.3.4, the vertex u_t makes an antipodal pair with the vertex u_k.

If u_t is a farthest vertex from the edge $[u_k, u_{k+1}]$, then the vertex u_{t-1} cannot be farther than vertex u_t from the edge $[u_k, u_{k+1}]$, so $\alpha_{t-1}-\alpha_k \leq \pi$, and the vertex u_t cannot be closer than the vertex u_{t+1} to the edge $[u_k, u_{k+1}]$ so $\alpha_t-\alpha_k \geq \pi$. As a result, the angle interval $[\alpha_{t-1}, \alpha_t]$ contains the angle $\pi+\alpha_k$ that is also in the angle interval $[\pi+\alpha_k, \pi+\alpha_{k+1}]$. By Lemma 4.3.4, the vertex u_t makes an antipodal pair with the vertex u_k.

Finally, suppose that u_i in the sequence $\{u_h, u_{h+1}, \ldots, u_{r-1}, u_r\}$ is a farthest vertex from neither the edge $[u_{k-1}, u_k]$ nor the edge $[u_k, u_{k+1}]$. Since u_h is a farthest vertex from $[u_{k-1}, u_k]$, $\alpha_h-\alpha_{k-1} \geq \pi$. Since $i > h$, $\alpha_i > \alpha_h$ so $\alpha_i-\alpha_{k-1} > \pi$, i.e., $\alpha_i > \pi+\alpha_{k-1}$. Moreover, because u_i is not a farthest vertex from $[u_k, u_{k+1}]$, $\alpha_i-\alpha_k < \pi$, i.e., $\alpha_i < \pi+\alpha_k$. Therefore, the angle intervals $[\alpha_{i-1}, \alpha_i]$ and $[\pi+\alpha_{k-1}, \pi+\alpha_k]$ intersect. By Lemma 4.3.4, the vertex u_i makes an antipodal pair with the vertex u_k.

Thus, in all cases, the vertex u_i in the sequence $\{u_h, u_{h+1}, \ldots, u_{r-1}, u_r\}$ makes an antipodal pair with the vertex u_k.

This completes the proof of the corollary. □

Now we are ready to present the algorithm, which is given below.

```
Algorithm Antipodal-Pair
Input: a convex polygon P = [u(0), ..., u(m-1)] in counterclockwise order
Output: all antipodal pairs of P
1. find the first farthest vertex u(h) from edge [u(m-1),u(0)];
2. For (k = 0; k <= m-1; k++)
     /* u(h) is the first farthest vertex from [u(k-1),u(k)] */
2.1  While (u(h) is not a farthest vertex from edge [u(k),u(k+1)])
        Output({u(k),u(h)}) as an antipodal pair;
        h = h + 1;
2.2  /* u(h) is the first farthest vertex from edge [u(k),u(k+1)] */
     Output({u(k),u(h)}) as an antipodal pair;
2.3  /* check the vertex u(h+1) */
     If (u(h+1) is also a farthest vertex from edge [u(k),u(k+1)])
       Output({u(k),u(h+1)}) as antipodal pairs.
```

The algorithm `Antipodal-Pair` works on each vertex `u(k)`, and finds all vertices that make antipodal pairs with `u(k)`. For each `k` $= 0, 1, \ldots, m-1$, the `k`-th execution of the `For`-loop at step 2 starts with the vertex `u(k)` and the first farthest vertex `u(h)` from the edge `[u(k-1),u(k)]`. Note that step 1 correctly prepares this configuration for the vertex `u(0)`. The correctness of steps 2.1-2.3 of the algorithm is ensured by Corollary 4.3.5, which output all vertices in the sequence that starts from the first farthest vertex to the edge `[u(k-1),u(k)]` and ends at the last farthest vertex to the edge `[u(k),u(k+1)]`. Note that the edge `[u(k),u(k+1)]` has at most two farthest vertices. Therefore, it suffices in step 2.3 to check only the vertex `u(h+1)` that is right after the first farthest vertex `u(h)` from `[u(k),u(k+1)]`. Also note that we do not change the index `h` so `u(h)` remains as the first farthest vertex from the edge `[u(k),u(k+1)]`, maintaining a correct configuration for the next execution of the `For`-loop.

Note that the operations on the index `h` in the algorithm (such as `h = h+1` in step 2.1) should be interpreted as `(mod m)` operations. To compute the distance from a vertex u_i to an edge $[u_{k-1}u_k]$, indeed the distance from the vertex u_i to the straight line containing $[u_{k-1}u_k]$, we observe that the distance is proportional to the area of the triangle $\triangle(u_i u_{k-1} u_k)$. Therefore, the vertex u_i is a farthest from the edge $[u_{k-1}u_k]$ if and only if the area of the triangle $\triangle(u_i u_{k-1} u_k)$ is less than neither the area of the triangle $\triangle(u_{i-1} u_{k-1} u_k)$ nor the area of the triangle $\triangle(u_{i+1} u_{k-1} u_k)$.

An intuitive description of the above algorithm is that we use two parallel lines to sandwich the convex polygon P, then rotate the lines along the boundary of P, keeping the lines in parallel. We report all pairs of vertices of P that are at some moment on the two parallel lines at the same time, respectively, when we rotate the lines.

To analyze the complexity of the algorithm, observe that we keep two indices `k` and `h` in the algorithm. In constant time, at least one of the indices is advanced. Since the index `k` is from 0 to $m-1$ and the index `h` marches the convex polygon `P` at most twice (the index `h` stops at the last farthest vertex from the edge `[u(m-1),u(0)]`), we conclude that the time complexity of the algorithm is bounded by $O(m)$.

A further improvement can be made in the algorithm `Antipodal-Pair` if we observe that when the index `h` reaches $m-1$, then, actually, all antipodal pairs of the polygon `P` have been found. In fact, when the index `h` is advanced from $m-1$ to 0, we are considering the vertex `u(0)` as a candidate that makes an antipodal pair with some other vertex of `P`. On the other hand, all vertices that make antipodal pairs with `u(0)` have been found when the index `k` is equal to 0. Although this improvement does not change the asymptotic order

of the time complexity of the algorithm, it may be useful from a practical point of view.

Now we give the algorithm for the original Farthest-Pair problem.

```
Algorithm Farthest-Pair
Input: a set S of n points in the plane
Output: the farthest pair in S
1. construct the convex hull CH(S) of S;
2. call Antipodal-Pair on CH(S);
3. output the pair with the largest distance from the result of step 2.
```

By the discussions given in this section, the above algorithm finds the farthest pair for a given set S correctly. Moreover, the algorithm runs in time $O(n \log n)$, which is dominated by the first step.

4.4 Triangulations

Triangulation is a fundamental problem in computational geometry. In many applications, the first step in working with complicated geometric objects is to break them into simple geometric objects. The simplest geometric objects in the plane are triangles. Classical applications of triangulation include finite element analysis and computer graphics.

Triangulating a set S of n points in the plane is to joint the points in the set S by non-intersecting straight line segments so that every region interior to the convex hull of S is a triangle (it is not difficult to see that if we insists on using straight line segments, then we cannot always make the exterior region a triangle). In this section we shall discuss a more general version of the triangulation problem: given a set S of n points in the plane and a set E of non-intersecting straight line segments whose endpoints are the points in S, construct a triangulation $T(S)$ of S such that all the segments in the set E appear in the triangulation $T(S)$.

Recall that a *planar straight line graph* (PSLG) $G = (S, E)$ is a finite set S of points in the plane plus a set E of non-intersecting straight line segments whose endpoints are the points in the set S. Note that a PSLG is a graph drawn in the plane but it is not necessarily connected. We always suppose that a PSLG G is represented by a doubly-connected edge list (DCEL).

The problem we shall discuss is called *Constrained Triangulation*, formally defined as follows.

> **Constrained Triangulation**
> Given a PSLG $G = (S, E)$ in the plane $\mathbf{E}^2$, construct a triangulation $T(S)$ of the point set S such that all segments of E are edges of $T(S)$.

4.4.1 Triangulating a monotone polygon

We start with the triangulation problem for a special class of PSLG's, called *monotone polygon*.

A *chain* $C = (v_1, \cdots, v_r)$ is a PSLG with a set of points $S = \{v_1, \cdots, v_r\}$ and a set of segments $E = \{\overline{v_i, v_{i+1}} \mid 1 \leq i \leq n-1\}$. A chain C is *monotone* with respect to a straight line l if any straight line orthogonal to l intersects the chain C at at most one point.

Definition A polygon P is *monotone* with respect to a straight line l if P is a simple polygon and the boundary of P can be decomposed into two chains monotone with respect to the straight line l. A polygon P is *monotone* if it is monotone with respect to the y-axis.

We first solve the following problem: given a monotone polygon P, triangulate the interior of P. That is, we add edges to the polygon P so that each region in the interior of P is a triangle.

A vertex u of a polygon P is *visible* from another vertex v of the polygon P if we can draw a straight line segment s connecting u and v such that the interior of the segment s is entirely in the interior of the polygon P. In particular, a vertex is *not* visible from any of its adjacent neighbors. Note that, by the definition, a vertex v is visible from a vertex u if and only if the vertex u is visible from the vertex v.

The method we are going to use is a "greedy" method. Standing at each vertex v of the polygon P, we look through the interior of the polygon P and see which vertex of the polygon P is visible. Whenever we find that a vertex u of the polygon P is visible from the vertex v, we add an edge between the vertices v and u. Keeping doing this until no vertex of P is visible from the vertex v, then we move to another vertex v' of P and add edges to those vertices that are visible from v', an so on. Note that once there is no vertex visible from a vertex v of P, then no vertex can become a visible vertex from v later, since the only operation we are performing is adding edges to the interior of the polygon P. Therefore, once we add edges to a vertex v of P so that there is no vertex of P visible from v, we do not have to come back and check the vertex v again. Moreover, if the interior of the polygon P is not triangulated, then there must be a pair of vertices v and u between which we can add a new edge e without edge-crossing. But this implies that the vertex u is still visible from the vertex v before we add the new edge e. Thus, if we process all vertices of P such that from any vertex v of P there is no visible vertex, then we must have triangulated the

interior of the polygon P.

The above method is valid in principle for triangulating *any* PSLG. However, to find all visible vertices from a vertex of a general PSLG may be difficult and time-consuming. On the other hand, if the PSLG is a monotone polygon, then the process above can be done very efficiently.

The following is the algorithm of triangulating a monotone polygon P. The vertices of the polygon P are processed in the way described above and in the ordering of decreasing y-coordinate. A stack K is used to store the vertices of P that have been processed such that no processed vertices are still visible from vertices in K but each vertex in K is still visible from some unprocessed vertices of P.

```
Algorithm Triangulate-MonoP
Input: a monotone polygon P
Output:  a triangulation of P
1. sort the vertices of P in decreasing y-coordinate: v(1),....,v(n);
2. Push(K, v(1)); Push(K, v(2)); /* K is a stack */
3. For (i = 3; i <= n; i++)
   /* assume K = [K(1), K(2); ..., K(s)], where K(1) is on top */
3.1  If (v(i) is adjacent to K(s) but not to K(1))
        w1 = K[1];
        While (K is not empty)
           add edge [v(i), K(1)]; Pop(K);
        Push(K, w1); Push(K, v(i));
3.2  Esle If (v(i) is adjacent to K(1) but not to K(s))
        While ((K has > 1 vertex) & (K(2) is visible from v(i)))
           add edge [v(i), K(2)]; Pop(K);
        Push(K, v(i));
3.3  Else /* now v(i) must be adjacent to both K(1) and K(s) */
        Pop(K);
        While (K has > 1 vertex)
            add edge [v(i), K(1)]; Pop(K);
        Pop(K); STOP.
```

We first discuss the correctness of the algorithm. Steps 1-2 initialize the configuration by setting a PSLG G_3 that is the given monotone polygon P. Also let P_3 be the region (i.e., the polygon) of G_3. For each i, $3 \leq i \leq n$, on the given PSLG G_i that has a region P_i, and a given stack configuration K_i = [K_i(1),...,K_i(s)] (where K_i(1) is on the top of the stack), the i-th execution of the For-loop in step 3 processes the vertex v(i) to construct the PSLG G_{i+1}. We first prove, by induction on i, that for all $3 \leq i \leq n+1$, the PSLG G_i, the polygon P_i, and the stack K_i satisfy the following conditions .

Properties of G_i

1. the region P_i is a monotone polygon, whose vertices consist of the vertices in the stack K_i, plus the vertices v(i), ..., v(n). The stack vertices [K_i(1),...,K_i(s)] make a monotone chain on the boundary of P_i;
2. the stack K_i contains at least two vertices, for $3 \leq i \leq n$;

3. a vertex v(j) not in the stack K_i and with j < i is not visible from any other vertex of G_i;
4. No two stack K_i vertices are visible from each other in G_i.

For $i = 3$, we have $G_3 = P_3$ being the given monotone polygon P, and the stack K_3 contains the first two vertices: K_3(1)=v(2) and K_3(2)=v(1). It is easy to verify that in this case, all four conditions hold true.

Now assume inductively that conditions 1-4 hold true for G_i, P_i, and K_i. We prove the conditions for G_{i+1}, P_{i+1}, and K_{i+1}.

Suppose that step 3.1 is executed in the i-th execution of the For-loop in step 3, then the vertex v(i) is adjacent to the bottom vertex K_i(s) in the stack K_i, but is not adjacent to the top vertex K_i(1) in K_i. We first show that in this case, all vertices in the stack K_i, except K_i(s), are visible from v(i). For this, consider the segment g = [v(i), K_i(s)]. Fix one end of g at v(i), and slide the other end of g along the edge [K_i(s),K_i(s-1)] (by induction, K_i contains at least two vertices, and [K_i(s), K_i(s-1)] is an edge of the monotone polygon P_i), keeping the interior of the segment g entirely in the interior of the polygon P_i until the segment g hits a vertex v. Since the vertex K_i(s-1) has a y-coordinate larger than that of v(i), the segment g cannot hit a vertex v(h) with $h > i$ before it hits K_i(s-1). Moreover, v cannot be a vertex v(h) with $h < i$ that is not in the stack K_i: by condition 3 inductively, such a v(h) is not visible from v(i). Thus, v must be a vertex in the stack K_i. If v is not K_i(s-1), then v would be a vertex in K_i that is visible from K_i(s), contradicting condition 4 by the inductive hypothesis. Thus, v must be K_i(s-1) so K_i(s-1) is visible from v(i). Also note that adding the edge [v(i),K_i(s-1)] makes the vertex K_i(s) not visible from any other vertex in the PSLG, which makes the vertex K_i(s) to satisfy condition 3. Now, similarly, using a segment that starts at the edge [v(i),K_i(s-1)] whose end at K_i(s-1) moves along the edge [K_i(s-1),K_i(s-2)], we can prove that the vertex K_i(s-2) is visible from v(i) so we can add the edge [v(i),K_i(s-2)] and make K_i(s-1) invisible from any other vertex, and so on. At the end of step 3.1, which has constructed the new PSLG G_{i+1}, the stack K_{i+1} contains two vertices K_{i+1}(1) = v(h), and K_{i+1}(2) = K_i(1), and a new polygon P_{i+1} is constructed whose vertices consist of the two vertices in K_{i+1} plus all vetices v(h) with $h > i$. Thus, all four conditions are satisfied by G_{i+1}, P_{i+1}, and K_{i+1}.

If step 3.2 is executed in the i-th execution of the For-loop in step 3, then the vertex v(i) is adjacent to the top vertex K_i(1) in the stack K_i, but is not adjacent to the bottom vertex K_i(s) in K_i. Step 3.2 repeatedly adds an edge [v(i),K_i(2)] and pops the top vertex K_i(1) out of K_i, as long

as the second top vertex K$_i$(2) in K$_i$ is visible from v(i). Since K$_i$(1) is adjacent to both v(i) and K$_i$(2), adding the edge [v(i),K$_i$(2)] will block the vertex K$_i$(1) from being visible from any other vertex, making vertex K$_i$(1), which is popped out of the stack, to satisfy condition 3. Since the While-loop in step 3.2 stops with at least one vertex in the stack K$_i$, and the vertex v(i) is pushed into the stack as a new stack vertex, the stack K$_{i+1}$ will contain at least two vertices, satisfying condition 2. For condition 1, inductively, the vertices in the stack K$_i$ make a monotone chain and the vertex v(i) has its y-coordinate smaller than that of all vertices in K$_i$, thus, pushing v(i) into the stack ensures that the vertices in the stack still make a monotone chain, which, plus the vertices v(h) with $h > i$, make a monotone polygon P_{i+1}, satisfying condition 1. Now the only remaining condition that still needs to verify is condition 4. Let [K$_{i+1}$(1),K$_{i+1}$(2),...,K$_{i+1}$(s)] be the stack configuration at the end of step 3.2, where K$_{i+1}$(1)=v(i) is at the top of the stack. If s=2, then condition 4 is automatically satisfied because [K$_{i+1}$(1),K$_{i+1}$(2)] is an edge of P_{i+1}. If s>2, then by the algorithm, K$_{i+1}$(1)=v(i) is not visible from K$_{i+1}$(3). If K$_{i+1}$(1) is visible from K$_{i+1}$(h) for some h>3, then pick such a K$_{i+1}$(h) with the smallest index h, and draw a segment g=[K$_{i+1}$(1),K$_{i+1}$(h)]. Fix the end K$_{i+1}$(h) of the segment g and slide the other end of g along the edge [K$_{i+1}$(1),K$_{i+1}$(2)], keeping the interior of the segment g entirely in the interior of the polygon P_i until the segment g hits a vertex v. Therefore, the vertex v is visible from K$_{i+1}$(h) (v could be K$_{i+1}$(2) but recall that h>3). However, the existence of the vertex v causes a contradiction: (1) v cannot be a vertex v(j) with j>i since the y-coordinate of K$_{i+1}$(h) is larger than that of K$_{i+1}$(1) and K$_{i+1}$(2); (2) v cannot be a vertex v(j) with j<i but not in the stack because by induction such a vertex is not visible from any other vertex; and (3) v cannot be a vertex in the stack because by the definition, v is not K$_{i+1}$(1) and [K$_{i+1}$(2),...,K$_{i+1}$(h),...,K$_{i+1}$(s)] is a subchain of the original chain in the stack K$_i$, in which by the induction no two vertices are visible from each other. This contradiction shows that the new stack vertex K$_{i+1}$(1)=v(h) cannot be visible from any other vertices in the stack K$_{i+1}$. Moreover, by induction on K$_i$, no two vertices from [K$_{i+1}$(2),...,K$_{i+1}$(s)], which are vertices in K$_i$, are visible from each other. This concludes that condition 4 is satisfied by the stack K$_{i+1}$ and the PSLG G_{i+1}.

Note that the vertex v(i) has to be adjacent to at least one of the vertices K$_i$(1) and K$_i$(s). Therefore, if steps 3.1 and 3.2 are not executed, then the vertex v(i) must be adjacent to both K$_i$(1) and K$_i$(s) in the monotone polygon P_i. This implies that v(i) must be the lowest vertex in the polygon P_i, i.e., i=n (this fact uses condition 1 inductively). In this

case, step 3.3 is executed. Since `v(i)` is adjacent to $\texttt{K}_i\texttt{(s)}$, as we proved for the case for step 3.1, in the case of step 3.3, all vertices in the stack $\texttt{K}_i$, except now $\texttt{K}_i\texttt{(1)}$ and $\texttt{K}_i\texttt{(s)}$, are visible from `v(i)`, and they will be popped out of the stack $\texttt{K}_i$ and become invisible from any other vertices after the corresponding edges are added. Note that in this case, condition 2 no longer holds true because the stack $\texttt{K}_{i+1}$ becomes empty.

This completes the proof that for all i, $3 \leq i \leq n+1$, the PSLG G_i, the polygon P_i, and the stack $\texttt{K}_i$ satisfy conditions 1-4. In particular, when $i = n+1$, by condition 3, no two vertices in the PSLG G_{n+1} are visible from each other. This means that the resulting PSLG G_{n+1} is a triangulation of the input monotone polygon `P`. This proves the correctness of the algorithm `Triangulate-MonoP`.

The analysis of the algorithm is easier. Since the polygon `P` is monotone, there are two vertices v_h and v_l of `P` with the largest and the smallest y-coordinates, respectively. Moreover, the boundary of the polygon `P` can be decomposed into two monotone chains

$$C = (u_0, u_1, \cdots, u_k) \quad \text{and} \quad C' = (u'_0, u'_1, \cdots, u'_h)$$

where $u_0 = u'_0 = v_h$ and $u_k = u'_h = v_l$ and the vertices in both chains C and C' are in decreasing y-coordinate ordering. We can merge the two chains C and C' in linear time to obtain the list `[v(1),v(2),...,v(n)]` of vertices of the polygon `P` sorted by decreasing y-coordinates. Therefore, step1 of the algorithm takes linear time.

Within the loop of step 3, we add each new edge in constant time. Since the final triangulation G_{n+1} is a planar graph that has at most $O(n)$ edges, the total time for adding new edges is bounded by $O(n)$. Moreover, since each vertex of `P` is pushed into then popped out the stack `K` exactly once, the total time of steps 2-3 is also bounded by $O(n)$.

We close this subsection with the conclusion that the problem of triangulating a monotone polygon can be solved in linear time.

4.4.2 Triangulating a general PSLG

Now we consider the problem of triangulating a general PSLG. Let G be a general PSLG. If each region of G is a monotone polygon, we can use the following algorithm to triangulate G: first construct all regions of G, which are monotone polygons P_1, P_2, ..., P_r; then triangulate each monotone polygon P_i by the algorithm `Triangulate-MonoP` given in the last subsection.

Constructing the regions P_1, ..., P_r of the PSLG G can be done using the algorithm `Trace-Region` given in section 2.4.2, which constructs a polygon

P_i in time $O(s_i)$, where s_i is the size of the polygon P_i, i.e., the number of sides of the polygon. To be more precise, the time for constructing the polygon P_i is bounded by $c \cdot s_i$ for a fixed constant c. Therefore, constructing all polygons $P_1, \ldots, P_r$ of the PSLG G takes time bounded by

$$c \cdot s_1 + \cdots + c \cdot s_r = c(s_1 + \cdots + s_r) = O(s_1 + \cdots + s_r).$$

Since each edge of G is used by exactly two regions of the PSLG G in their boundary, $s_1 + \cdots + s_r$ is twice the number of edges of G, which is bounded by $O(n)$ since G is a planar graph. That is, the regions of G can be constructed in time (n). Now we triangulate each region P_i of G using the algorithm `Triangulate-MonoP` in subsection 4.4.1, which triangulates the monotone polygon P_i in time $O(s_i)$, i.e., in time bounded by $d \cdot s_i$ for a fixed constant d. Therefore, triangulating all regions of G takes time

$$d \cdot s_1 + \cdots + d \cdot s_r = d(s_1 + \cdots + s_r) = O(s_1 + \cdots + s_r) = O(n).$$

It is easy to see that putting all these triangulated regions together to get a triangulation of the PSLG G can also be done in time $O(n)$. As a result, we conclude that if all regions of a PSLG G are monotone polygons then the triangulation of G can be done in linear time.

Therefore, the problem of triangulating a general PSLG G is reduced to the problem of converting the PSLG G into a PSLG G' such that all regions of G' are monotone polygons. Without loss of generality, we suppose that our PSLG G has no two points with the same y-coordinate (otherwise we can achieve this by rotating the coordinate system slightly). Let us first introduce some definitions.

Let G be a PSLG and let v be a vertex of G. An edge $[u, v]$ is an *upper edge* of v if the y-coordinate of u is larger than that of v, and an edge $[w, v]$ is a *lower edge* of v if the y-coordinate of w is smaller than that of v. A vertex v of G is *regular* if either v is the vertex of G with the maximum or the minimum y-coordinate or v has both upper edges and lower edges.

Definition A PSLG G is *regular* if every vertex of G is regular.

Note that if G is a regular PSLG, then G must be connected. To see this, suppose that G is not connected, then let v_h and v'_h be the vertices of the maximum y-coordinate for two different connected components of G, respectively. Now both v_0 and v'_0 have no upper edges, so one of them must be an irregular vertex.

Lemma 4.4.1 *All regions of a regular PSLG are monotone polygons.*

PROOF. Suppose that G is a regular PSLG but a region P of G is not a monotone polygon. Let v_h be the vertex of P that has the largest y-coordinate. Since P is a simple polygon and no other vertex of P has the same y-coordinate as v_h, when a horizontal straight line L is close enough to the vertex v_h, L intersects P at exactly two points. Because P is not monotone, there must be some horizontal lines intersecting P at more than two points. Let

$$r_0 = \sup\{r \mid \text{the line } y = r \text{ intersects } P \text{ at more than two points.}\}$$

Let L_0 be the horizontal straight line $y = r_0$. There are two possible cases.

The line L_0 intersects P at two points. Then since a slight moving down of the line L_0 would make the line intersect more than two points, there must be a vertex v of P on the line L_0 such that the vertex v has two lower edges. Since moving L_0 down by an arbitrarily small distance would make L_0 intersect P with more than two points, v is not v_h. However, this implies that v has no upper edges since each vertex of P is incident to exactly two edges of P. Thus, v is not a regular vertex and G is not a regular PSLG.

On the other hand, suppose that L_0 intersects P at more than two points, then a slight moving up the line L_0 would make the line intersect exactly two points of P. Thus one of those intersecting points of L_0 and P must be a vertex of P without upper edges. But this again contradicts the assumption that G is regular.

Therefore, the region P must be a monotone polygon. Since P is an arbitrary region of the PSLG G, this completes the proof of the lemma. □

Therefore, the problem Triangulation for regular PSLGs can be solved in linear time. In the next subsection, we will show that given a general PSLG G, in time $O(n \log n)$ we can convert G into a regular PSLG by adding edges to G. Consequently, the problem Constrained Triangulation can be solved in time $O(n \log n)$.

Remark: The problem of triangulating a simple polygon had drawn significant attention in the research in computational geometry. After much effort, Chazelle [8] was eventually able to develop a (highly non-trivial) linear-time algorithm for triangulating a general simple polygon. Since for a connected PSLG G, the regions of G can be constructed in linear time, Chazelle's linear time algorithm for simple polygons implies a linear-time algorithm for triangulating a general PSLG in which all regions are simple polygons (note that such a PSLG is necessarily connected).

4.4.3 Regularization of PSLGs

We thereby have the following problem.

> **RegularizingPSLG**
> Given a general PSLG G, add edges to G so that the resulting PSLG is regular.

Intuitively, the process of regularizing a PSLG is simple: we add an upper edge to a vertex if it does not have an upper edge, and add a lower edge to a vertex if it does not have a lower edge. The problem is, how do we add the edges so that edge-crossing is avoided. Therefore, when we are working on a vertex of a PSLG G, we should have enough information about the local environment of the vertex. But how do we maintain and update the information about the local environment efficiently when we move from one vertex to another vertex in the PSLG G?

Again, the plane sweeping technique helps. Let $V = \{v_1, v_2, \cdots, v_n\}$ be the vertex set of the PSLG G. Without loss of generality, suppose that no two vertices in V have the same y-coordinate.[1] We first sort the vertices in V by their y-coordinates. Then we sweep the plane by a horizontal line from the bottom up. The sweeping stops at each vertex of G and check if the vertex has an upper edge. If the vertex does not have an upper edge, then we record it and will add an upper edge to it later when we find a proper vertex with a larger y-coordinate. This process will ensure that all vertices, except the vertex of the maximum y-coordinate, have upper edges. We then sweep the plane one more time from the top down to add lower edges for those vertices that have no lower edges, except the vertex with the minimum y-coordinate. After these two sweeping processes, every vertex has at least one upper edge (except for the vertex with the maximum y-coordinate) and at least one lower edge (except for the vertex with the minimum y-coordinate). Thus the PSLG becomes regular. We discuss the bottom-up sweeping in detail. The top-down sweeping can be handled similarly.

Without loss of generality, suppose that the list $\{v_1, v_2, \cdots, v_n\}$ is the sorted list of the vertices of the PSLG G in ascending y-coordinates. Consider the horizontal sweeping line L that passes through a vertex v_i of G, where $i < n$. The sweeping line L partitions the PSLG G into three parts G_1, G_2, and G_3: (1) the structure G_1 is the "past history" containing those vertices of G that are below the line L and have upper edges, and those edges of G that are entirely below the line L; (2) the structure G_2 is the "current status" containing the vertices of G that are either on the line L or below

[1] In fact, with a minor modification, our algorithms will also work for the general case.

the line L but have no upper edges, and those edges of G that intersect the line L; and (3) the structure G_3 is the "unknown future" containing the vertices and edges of G that are entirely above the line L. The elements in G_1 are "done" elements that we have seen and we know that they do not need further processing, the elements in G_2 are the "current" elements that we are processing, and the elements in G_3 are "unknown" elements that have not been seen during the bottom-up sweeping. Therefore, the process of the plane sweeping is a process of updating the current status of the structure G_2 when we pass through each vertex of the PSLG G. Note that it is easy to see that during the sweeping between two consecutive vertices in the list $\{v_1, v_2, \cdots, v_n\}$, the status of the structure G_2 is invariant. The status of G_2 only changes when we pass through a vertex of the PSLG G. This is the reason why our sweeping is discrete (i.e., the sweeping only stops at the vertices of G and updates the current status of G_2).

We use a data structure D_2 to represent the current status of the structure G_2, which contains all edges in currently intersecting the sweeping line L, ordered from left to right (i.e., sorted by the x-coordinates of their intersecting points with the line L). We require that between two consecutive edges e_1 and e_2 in D_2, there is at most one "hung vertex", i.e., a vertex v that is below the sweeping line L and has no upper edges, here "between e_1 and e_2" means that if we draw a horizontal line through v, the line will intersect e_1 and e_2 on the two sides of v. This condition can be maintained, as follows: (1) if a second hung vertex v' is added between the edges e_1 and e_2, then we add an edge $[v, v']$ between v' and the first hung vertex v, which gives the first hung vertex v an upper edge thus unhang v; and (2) if the sweeping line L passes over the upper end v'' of one of the edges e_1 and e_2, then we add an edge $[v, v'']$ between v'' and the hung vertex v to give v an upper edge and unhang v. The information about the hung vertices between consecutive edges in the data structure D_2 is also recorded in D_2 with the corresponding edges so that for an edge in D_2, we can directly read if there are vertices hung on the left and/or the right of the edge.

The status of the structure G_2 recorded in D_2 can be dynamically updated in the following way when we are passing through a vertex v_i. Assume that D_2 records the status of G_2 just before L hits the vertex v_i. Now move the line L up until it hits v_i. Let e_1 be the edge whose intersection with L is just left to the vertex v_i on L, and let e_r be the edge whose intersection with L is just right to the vertex v_i on L (note that in certain cases the edges e_1 and e_r may not exist). Moreover, let e_2, ..., e_{r-1} be the lower edges incident on v_i in counterclockwise ordering. Note that before the line L hits the vertex v_i, the edges e_1, e_2, ..., e_{r-1}, e_r are consecutive in the

structure D_2, ordered from left to right. We then check if there is a hung vertex between each pair $[e_h, e_{h+1}]$ of edges, for $h = 1, \ldots, r-1$. If there is such a hung vertex v_h between $[e_h, e_{h+1}]$, then we add a new edge $[v_i, v_h]$ between v_i and v_h. Note that since the vertex v_h is the unique hang vertex between the consecutive edges e_h and e_{h+1} on L and v_i is an upper end of one of these two edges, adding the edge $[v_i, v_h]$ cannot cause edge crossings. Moreover, since the y-coordinate of v_i is larger than that of v_h, the new edge $[v_i, v_h]$ is an upper edge for v_h thus unhangs v_h. Now, after the sweeping line L passes over the vertex v_i, all lower edges of v_i will no longer belong to G_2. Thus we delete all of them from the data structure D_2 (so the edges e_1 and e_r become adjacent in D_2). On the other hand, all upper edges incident to v_i now belong to the structure G_2 so we insert all of them into the data structure D_2 (they will appear between e_1 and e_r in D_2). If v_i has no upper edges, then v_i will become hung between the edges e_1 and e_r (they are now adjacent in D_2). This completes the update of D_2 for the status of the structure G_2 when the sweeping line L passes through the vertex v_i.

Thus, the data structure D_2 is initiated as an empty set. The sweeping line L goes from vertex v_1, from the bottom up until meets the vertex v_n, with the data structure D_2 dynamically changed when the sweeping line L passes over each vertex, as described above. We will eventually finish adding upper edges to the vertices of the PSLG G. For any vertex v_i with no upper edges, the above process will hang v_i between two consecutive edges e_i and e_{i+1} in D_2 when the sweeping line passes through v_i. It is guaranteed that the vertex v_i will get unhung later: either the sweeping line L will hit a vertex v_j with $j > i$ such that v_j is also between e_i and e_{i+1} while e_i and e_{i+1} remain adjacent in D_2 so v_i is still hung between e_i and e_{i+1}, or the sweeping line L hits the upper end of one of edges e_i and e_{i+1}. In either case, the above process will add an upper edge to v_i and unhang v_i. Thus, after this process, each vertex of G, except v_n, has at least one upper edge.

To make the above process efficient, we need the data structure D_2 to support the following operations efficiently: (1) keeping a list S_2 for the edges in G_2, sorted by the x-coordinates of their intersecting points with the sweeping line L; (2) for a given vertex v_i, finding the sublist $T_i = [e_1, e_2, \ldots, e_r]$ of S_2 such that e_2, ..., e_{r-1} are the edges in G_2 that are incident to v_i; and (3) inserting edges into G_2 and deleting edges from G_2.

We can use a 2-3 tree for the data structure D_2. Thus, the edges in G_2 are sorted in D_2 increasingly by the x-coordinates of their intersecting points with the sweeping line L at its current position. On a given vertex $v_i = (x_i, y_i)$, we can make the sweeping line L as $y = y_i$ and search in the 2-3 tree D_2 for $x = x_i$. This will collect all lower edges of v_i: $[e_2, \ldots, e_{r-1}]$

of v_i, from which it is also easy to find the edge e_1 left to e_2 and the edge e_r right to e_{r-1} in D_2. In particular, the desired list $T_i = [e_1, e_2, \ldots, e_r]$ can be constructed in time $O(r \log n)$. Finally, the 2-3 tree D_2 supports insertion and deletion operations in time $O(\log n)$ per operation.

The following algorithm is based on the above discussion.

```
Algorithm Add-UpperEdges
Input: a PSLG G of n vertices
Output: a PSLG G', obtained by adding edges to G such that every
        vertex of G' (except the highest one) has upper edges.
1. sort vertices of G in increasing y-coordinates: v(1),....,v(n);
2. D2 = empty;  /* D2 is a 2-3 tree */
3. If (v(1) has upper edges)
     insert the upper edges of v(1) into D2;
   Else hang v(1);
4. For (i = 2; i <= n; i++)
4.1  find the sublist e(1),...,e(r) in D2, where e(2),....,e(r-1)
     are the lower edges of v(i);
4.2  For (h = 1; h < r; h++)
       If (there is a hung vertex v(h) between e(h) and e(h+1))
         add a new edge [v(h), v(i)]; unhang v(h);
4.3  delete the edges e(2), ...., e(r-1) from D2;
4.4  If (v(i) has upper edges)
       insert the upper edges of v(i) into D2;
     Else if (i < n)
       hang v(i) between the edges e(1) and e(r).
```

We analyze the complexity the algorithm. Step 1 can be done in time $(n \log n)$ by any optimal sorting algorithm. Since a PSLG of n vertices has only $O(n)$ edges, the 2-3 tree `D2` has at most $O(n)$ leaves. In the algorithm, each edge of the PSLG is inserted into `D2` only once (when it is an upper edge for a vertex, in steps 3 and 4.4), and deleted from `D2` once (when it becomes a lower edge of a vertex, in step 4.3). Moreover, the algorithm spends time $O(\log n)$ per edge when searching the edge list `e(1),e(2),...,e(r)` in step 4.1, and time $O(r)$ on processing the hung vertices in step 4.2. In summary, the algorithm spends time $O(\log n)$ on each edge of the PSLG. Since a PSLG of n vertices has $O(n)$ edges, we conclude that the algorithm `Add-UpperEdges` runs in time $O(n \log n)$.

Similarly, we can add lower edges to a PSLG in time $O(n \log n)$. This shows that we can regularize a PSLG of n vertices in time $O(n \log n)$.

Combining this result with the results of subsections 4.4.1-4.4.2, we conclude with the following theorem.

Theorem 4.4.2 *The problem Constrianed Triangulation can be solved in time $O(n \log n)$.*