Solutions to Homework 1: Convex Hulls and Plane Sweep

Solution 1: As a convenience in describing our solution, let us imagine that we apply a rotation\(^1\) so that \(u\) is directed along the \(x\)-axis, that is \(u = (1, 0)\). Before accessing any vertex of the polygon we first apply this transformation.

Now the problem reduces to finding the vertex with the maximum \(x\)-coordinate, that is, the rightmost vertex of \(P\). We will make the usual general-position assumption that no two points have the same \(x\)-coordinates. Let \(\langle v_1, \ldots, v_n \rangle\) be the transformed vertices of \(P\) listed in counterclockwise order, and let \(x_i\) denote the \(x\)-coordinate of \(v_i\).

We apply a variant of binary search. Let \(\langle v_i, \ldots, v_j \rangle\) (for \(i \leq j\)) be a counterclockwise chain of \(P\). We assert that in \(O(1)\) time, we can determine whether the rightmost vertex of \(P\) lies within this subchain. Our test is based on a few easily verified observations. First, it follows by convexity that the \(x\)-coordinates of the vertices of a convex polygon form a sequence that has a unique local maximum and a unique local minimum. (By “local” we mean that the two vertices on either side of the vertex either are either both left or both right of the given vertex.) Second, since the vertices are in counterclockwise order, the edge leaving each vertex is directed from right to left if the vertex is on the upper hull and it is directed from left to right if it is on the lower hull. (Observe that the rightmost vertex of \(P\) is classified on the upper hull and the leftmost vertex is classified on the lower hull.)

First, observe that if \(i = j\) (the chain has a single vertex), we can test \(O(1)\) time whether this vertex is the rightmost vertex by checking the \(x\)-coordinates of its two neighbors. Otherwise, \(i < j\). Let us consider the possible cases regarding the hulls on which \(v_i\) and \(v_j\) lie:

- **(upper, upper):** If \(x_i > x_j\), the entire chain goes from right to left on the upper hull, and thus it cannot contain the rightmost vertex. Otherwise, it does (and it also contains the leftmost vertex). (See Fig. 1.)

- **(lower, lower):** If \(x_i < x_j\), the entire chain goes from left to right on the lower hull, and thus it cannot contain the rightmost vertex. Otherwise, it does (and it also contains the leftmost vertex).

- **(lower, upper):** Since the \(x\)-coordinates are increasing after \(v_i\) and decreasing after \(v_j\), we must encounter the local maximum, and therefore the chain does contain the rightmost vertex.

- **(upper, lower):** Since the \(x\)-coordinates are decreasing after \(v_i\) and increasing after \(v_j\), the complementary chain must contain the unique local maximum, and hence the chain does not contain the rightmost vertex.

The algorithm now applies the above test as part of a binary search. The initial chain is \(\langle v_1, \ldots, v_n \rangle\). Given any chain \(V = \langle v_i, \ldots, v_j \rangle\), we subdivide it at its median vertex \(v_k\) two subchains \(V_1 = \langle v_i, \ldots, v_k \rangle\) and \(V_2 = \langle v_k, \ldots, v_j \rangle\). We apply the above test to each of these subchains (actually, it suffices to apply it to only one), and recurse on the one that contains the rightmost vertex. When the subchain consists of a single vertex, we have found the rightmost vertex.

Solution 2: The algorithm is straightforward adaptation of the plane-sweep algorithm for line segment intersection. The algorithm begins by sorting the left endpoints of the chains. This can be done in \(O(m \log m)\)

---

\(^{1}\)This is a little exercise in linear algebra, which you are not responsible for. But, if you are interested, here are the details.

We need to rotate by the negation of the angle between \(u\) and the \(x\)-axis. This angle has tangent \(u_y/u_x\). Thus, the desired rotation matrix is

\[
\begin{pmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{pmatrix}
\]

where \(\theta = -\arctan(u_y/u_x)\).

Traditionally, arctangent produces a result over half the half the circle, but it can be computed over the entire circle using the Unix function atan2\((u_y, u_x)\).
time. These events are inserted into the event queue. The plane sweep then proceeds as in the standard line-segment intersection algorithm, but with the following modifications.

(i) If any intersection is detected between two chains, we terminate the algorithm immediately.

(ii) When the right endpoint of any segment is encountered, we replace this entry in the sweep-line status with the next edge of the chain (or delete it, if this is the last vertex of the chain) and check for intersections with the segments immediately above and below in the sweep-line status. Since the chains are strictly monotonic, it follows that this correctly handles this case. Next, we insert the right endpoint of this segment in the event queue.

Observe that, at any time, the sweep-line status has at most \( m \) entries, and hence each operation on the sweep-line status (insert, delete, previous, next) takes \( O(\log m) \) time. Also, because we only store one vertex of each chain in the event queue, there are at most \( m \) entries in the event queue, and hence, each operation can be performed in \( O(\log m) \) time. Thus, the total time to process all \( O(n) \) events is \( O(n \log m) \) time.

**Solution 3:** If a point on the boundary of \( P \) does not lie on the UVH, we say that this point is *shadowed*.

(a) First, we show that the UVH is a subset of \( C \). Let \( C' \) denote the (lower) chain of \( P \)'s boundary, which is formed by traversing the boundary of \( P \) in counter-clockwise order from \( v_1 \) to \( v_m \) (see Fig. 3(a)). Clearly, both \( C \) and \( C' \) are connected. Thus, any vertical line between \( v_1 \) and \( v_m \) must intersect both \( C \) and \( C' \). If, to the contrary, some point of \( C' \) were to lie on the UVH, then there would be a vertical line passing through this point, such that its intersection with \( C' \) lies above its intersection with \( C \). Since \( C' \) lies locally beneath \( C \) close to \( v_1 \) and \( v_m \), this inversion would imply that \( C \) and \( C' \) intersect at some point, which would contradict the assumption that \( P \) is a simple polygon.

Next, we argue that the points of the UVH lie only on right edges of \( C \). Since \( C \) traverses \( P \)'s boundary in clockwise order, if we encounter a left edge, the interior of \( P \) lies locally above the edge (see Fig. 3(b)). (The interior lies locally to the right of an observer standing on the edge and facing its direction.) This implies there must be some other edge of \( P \) lying above the current edge, which will shadow it and keep it out of the UVH.

(b) Suppose to the contrary there is a pair of points \( p_1, p_2 \) that appear in this order along a left-to-right order on the UVH (that is, \( x_1 < x_2 \)), but \( p_2 \) appears before \( p_1 \) along a traversal of the upper chain \( C \).
from $v_1$ to $v_m$ (see Fig. 4). We will derive a contradiction. Consider a region of the plane bounded by three parts: A vertical ray shot up from $v_1$, a vertical ray shot upwards from $p_2$, and the portion of upper chain $C$ from $v_1$ to $p_2$, which we’ll call $C'$. Since $p_2$ is seen before $p_1$ along $C$, it follows that $C'$ does not contain $p_1$. Since $p_1$ lies on the UVH by hypothesis, it follows that it lies within this three-part region. However, it is easy to see that $v_m$ lies outside this region. Thus, it is impossible for the upper chain to go from $p_1$ to $v_m$ without either crossing one of these rays or $C'$. Any such event, leads immediately to a contradiction.

(c) The algorithm is similar in spirit Graham’s algorithm. It processes the vertices from one end of the chain to the other, and maintains the UVH as it goes. We will maintain the UVH as a sequence of intervals along the $x$-axis, sorted from left to right. Within each interval, we will store a single line segment, which will correspond to the portion of an edge of $P$ that lies on the current UVH. We will intentionally not store certain edges that we know cannot be part of the UVH. As with Graham’s algorithm, these intervals will be stored on a stack, so that we can efficiently push new intervals (on the right) or remove intervals (from the right) which cannot be part of the UVH.

By part (a), it suffices to consider just the edges of $C$ (which we can compute in $O(n)$ time, by a traversal of $P$’s boundary). We start at $v_1$ and terminate on reaching $v_m$. The algorithm starts by creating a single interval containing the (left) edge $v_1; v_2$ and pushing it on the stack. We will maintain the invariant that the current vertex $v_i$ lies at the right side of the rightmost interval, which will be on top of the stack. Let $e = (v_i, v_{i+1})$ be the current edge being processed. We consider the following cases:

Right-edge: If $x_{i+1} > x_i$, we push the interval spanning $x_i, x_{i+1}$ onto the stack, and associate this interval with $e$. (See the comment below about entering the caves along right-moving edges.)

Left-edge: If $x_{i+1} < x_i$. We start tracing $e$ to the left. For each interval overlapped by $e$ we do the following. Let $s$ denote the segment associated with this interval. If the $e$ lies above $s$, we consider two subcases. First, if $e$ completely covers the interval, we pop this interval off the stack, and continue with the new stack top. Second, if $e$ covers only part of the interval, we keep the interval on the stack top, but we trim the interval so its right endpoint is now $x_{i+1}$. In either case, the operation is justified, since the segments being popped or trimmed are shadowed by $e$, and $e$ cannot be on the UVH by part (a).
On the other hand, if \( e \) lies beneath \( s \), we know that this part of \( e \) is shadowed by \( s \). Let us consider this case in a bit more generality, where the current edge \( e \) passes underneath the right endpoint of \( s \) (see Fig. 6(a)). Consider the vertical line passing through \( s \)'s right endpoint. Let \( p, r, \) and \( q \) be the intersection points of \( s, e \) and the segment that \( e \) shadowed just to the right of this vertical line, respectively (see Fig. 6(b)).

We think of the contour as "entering a cave" whose mouth is the segment \( \overline{pr} \). We enter a loop that traverses \( C \) starting from \( r \) until reaching a point \( r' \) where the chain reemerges through the mouth of the cave. We assert that (1) this must eventually happen, (2) when it does, the edge of \( C \) intersecting the mouth is a right edge, and (3) none of the points visited in this traversal lies on the UVH.

To prove these assertions we appeal to Fig. 6(c). Consider the portion of the boundary of \( C \) from \( p \) through \( q \) to \( r \). This forms the cave. It follows from parts and (b) above, that no part of this boundary is part of the UVH. Now, consider the region bounded by this portion of the boundary and the line segment \( \overline{rp} \). Because the contour enters the region at \( r \), and it cannot intersect the boundary of \( C \), it must eventually emerge through \( \overline{rp} \). Clearly it must be traveling from left to right when it does emerge. Finally, since it is entirely shadowed within this cave region, it cannot contribute the UVH.

Let \( r' \) denote the point where the contour emerges from the cave. We then continue the algorithm, with the right edge immediately following the emergence. A symmetric case is needed when the path is turns to the right and enters a cave. We trace the path until emerging leftwards through the mouth of the cave, and ignore vertices as we go. This completes the description of the algorithm.

To establish the running time of the algorithm we observe that each left-edge event can be handled in \( O(1) \) time. Each right-edge event takes time proportional to the number of entries popped off the stack, plus 1 for the trimmed interval. The time needed to traverse the cave is proportional to the number of edges of \( C \) that lie in the cave. Observe that every edge can be put on the stack at most once, it can be popped at most once. It can be trimmed many times, but each time it is trimmed, we can charge the processing to the shadowing edge that performed the trimming. Finally, each edge can be processed at most once for lying within a cave. It follows that the total running time is \( O(n) \), as
Solution to the Challenge Problem:  Let $P_B$ and $P_R$ denote the sets of blue and red points, respectively, and let $n_B$ and $n_R$ denote the respective sizes. We begin by characterizing the shape of the union of the RBB triangles. Let $H_B$ denote the convex hull of the blue points (see Fig. 7(a)). For any point $r \in P_r$, let $H_r$ be the convex hull of $\{r\} \cup P_B$ (see Fig. 7(b)).

We claim that the union of the RBB triangles having just $r$ as a vertex is equal to $H_r$. To see this, consider any point $q$ in this convex hull. Draw a line from $r$ through $q$, and continue the line until passes to the boundary of $H_R$. It is easy to see that the point where the line exits the shape must lie on an edge of $H_B$. Let $b_1$ and $b_2$ be the blue points defining this edge. It is easy to see that the triangle $\triangle rb_1b_2$ contains $q$, as desired. The converse is trivial, since any RBB triangle $\triangle rb_1b_2$ is the convex hull of the set $\{r, b_1, b_2\}$, and hence it is contained within the convex hull any superset, such as $\{r\} \cup P_B$, which is just what $H_r$ is.

Let $U = \bigcup_{r \in R} H_r$, be the desired union. Observe that if $r \in H_B$, this point does not contribute to $U$, and so we assume henceforth that $R$ consists only of points lying outside of $H_B$. (If there are no such points, $U = H_B$, and we are done.) For each $r \in R$, let $t_r'$ and $t_r''$ denote the two tangents from $r$ to $H_r$. As we saw in Chan’s algorithm, we may compute these two tangents in time $O(\log n_B)$, and therefore, we may compute all these tangents in time $O(n_R \log n_B) = O(n \log n)$.

We assert that the boundary complexity of $U$ is $O(n)$. The portion of the boundary formed by the edges of $H_B$ is clearly of size $O(n_B) = O(n)$. Observe that any tangent edge that contributes the boundary can provide at most one segment to the final boundary. The reason is that if part of some tangent segment of $t_r$ is hidden from the final boundary, it must lie within $H_{r'}$, for some other red point $r'$. If so, the remaining portion of the tangent, all the way to the boundary of $H_B$ is hidden from the final boundary. Therefore, each of the $2n_R$ tangents can contribute at most one line segment to the final boundary, which implies that the total boundary complexity is $O(n_B + 2n_R) = O(n)$.

Let $S$ be the set of line segments consisting of these tangents over all $r \in R$ and all the edges on the boundary of $H_B$. Let $o$ be any point in the interior of $H_B$. Let $T$ be the set of all triangles for which $o$ is one vertex and whose base is a segment of $S$ (see Fig. 7(c)). It is easy to see that $U$ can be equivalently expressed as $\bigcup_{t \in T} t$. Our approach will be to compute this latter union of $O(n_B + 2n_R) = O(n)$ triangles.

The algorithm is based on divide-and-conquer. It begins by partitioning $T$ into $T_1$ and $T_2$ of roughly equal sizes. We recursively compute the unions $U_1 = \bigcup_{t \in T_1} t$ and $U_2 = \bigcup_{t \in T_2} t$. Each set is represented as a counterclockwise collection of base edges (sorted radially about $o$). We merge these two sequences according to their angular orders about $o$. Within each interval of the resulting sorted sequence, there are at most two base edges present, one from $U_1$ and one from $U_2$ (see Fig. 7(d)). We compute the union of the two resulting triangles (which may create one additional vertex) in $O(1)$ time. Since each shape has $O(n/2)$ intervals, up to constant factors the merger can be performed in $O(n)$ time. Thus, the running time is given by the standard recurrence $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$. After repeating this for all the radial intervals, we obtain the desired shape $U$ (see Fig. 7(e)).