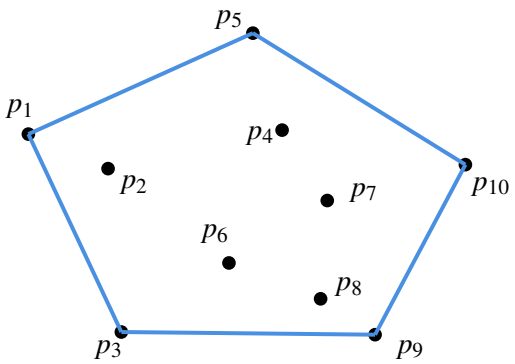
Convex Hull

Definitions

Definition 1 (Convex Hull)**.** Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of points on 2D plane, each of which is represented as $p_i = (x_i, y_i)$. The *convex hull* of P, denoted as $CH(P)$, is defined as the smallest, convex polygon that includes all points in P.

A convex hull is usually represented as the list of vertices (which is subset of P) of the polygon in counter-clockwise order. Such list is circular, and it's equivalent to shifting any number of vertices. If a point $p \in P$ is one of the vertices of the convex hull of P, we usually say that p is *on* the convex hull, and denote it as $p \in CH(P)$.



$$CH(P) = (p_1,\ p_3,\ p_9,\ p_{10},\ p_5)$$

Figure 1: A set of points $P = \{p_1, p_2, \cdots, p_{10}\}$ and its convex hull.

For most computational geometry problems, we always assume that the data (points, lines, etc) given to us are in *general* situation. For example, we usually assume no three (or more) lines go through the same point, no three (or more) points are colinear, and no four points are on the same circle, etc.

Property 1. Let $p \in P$. Then $p \in CH(P)$ if and only if there exists a line l such that p is on l, and that all other points, i.e. $P \setminus \{p\}$, are on the same side of l.

The above property gives us a way to determine if a point is on the convex hull, which will be used in the following algorithm (e.g., Graham-Scan).

Property 2. For any set of points P and another point q, we have $CH(P \cup \{q\}) = CH(CH(P) \cup \{q\})$.

The above property gives a way to gradually contruct convex hull. Suppose we already know the convex hull of P, i.e., $CH(P)$. Now we want to calculate the convex hull of $P \cup \{q\}$. We only need to consider these points in $CH(P)$ and q: those points that are inside of the convex hull of P can be safely discarded.

Graham-Scan Algorithm

The idea of Graham-Scan is to gradually construct the convex hull. The first step of this algorithm is the identification of the lowest point in P, i.e., the point with smallest y-coordinate, denoted as p_*. The second step is to sort all others points in counter-clockwise order w.r.t. p_*. Specifically, for any point $p \in P$, the

measure we use in sorting is the *angle* defined by points p, p_* and $(+\infty, 0)$. We define such angle for p_* is 0. All points are then sorted in ascending order by their angles. After sorting, for the sake of simplicity, we rename all points in P in this order, i.e., rename p_* as p_1, the point with second smallest angle as p_2, and so on. These two steps can be regarded as *preprocessing* steps of the Graham-Scan algorithm (i.e., first 3 lines of Algorithm Graham-Scan); the resulting of it is a sorted list of points P. See Figure 2.

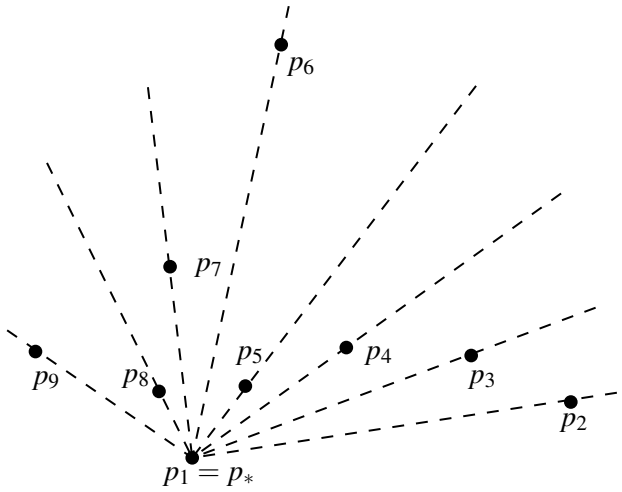


Figure 2: Preprocessing steps of Graham-Scan.

The main body of Graham-Scan is referred to as Graham-Scan-Core. It processess points in P (notice that now they are properly sorted) one by one and gradually construct the convex hull. It maintains a *stack* data structure, called S, and keeps the following variant:

Invariant: after processing the first k points of P, the convex hull of these k points, i.e., $CH(p_1, p_2, \cdots, p_k)$ will be stored in the stack S: the list of points in S from bottom to top gives the list of the vertices of the convex hull in counter-clockwise order.

How to guarantee above invariant? Consider the general case when we are processing p_k. Now we know that S stores $CH(p_1, p_2, \cdots, p_{k-1})$, and the goal is to determine $CH(p_1, p_2, \cdots, p_k)$. First, following above Property 2, we know that $CH(p_1, p_2, \cdots, p_k) = CH(CH(p_1, p_2, \cdots, p_{k-1}) \cup \{p_k\})$. Hence we only need to consider the points in S and the current point p_k.

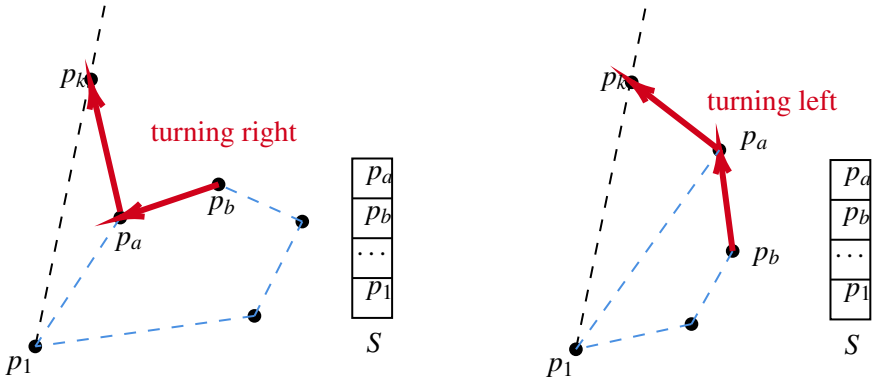


Figure 3: Procedure to decide if top element of S is on $CH(p_1, p_2, \cdots, p_k)$.

Second, notice that p_k must be on $CH(p_1, p_2, \cdots, p_k)$. Why? This can be proved using line p_1p_k following Property 1: line p_1p_k goes through point p_k and all others points in S are on the right side of this line (this is because points in P are sorted and we are processing them in counter-clockwise order; you can see now the reason why we sort points in this way).

Third, we need to determine and exclude points in S that are not in $CH(p_1, p_2, \cdots, p_k)$. See Figure 3. We use the following procedure to determine if the top element, called p_a, of S is on $CH(p_1, p_2, \cdots, p_k)$: we check the *orientation* of the three points $p_b \rightarrow p_a \rightarrow p_k$, where p_a and p_b are the top and second top elements of S respectively. If it's "turning right", then p_a will not be on $CH(p_1, p_2, \cdots, p_k)$. Why? Because in this case p_a will be within triangle $p_1p_bp_k$ (the angle of p_a is in the middle of that of p_b and p_k). If this happens, we can remove p_a safely, i.e., *pop (S)*, and continue to examine the top element of S iteratively. If the orientation is "turning left", then actually we get the convex hull of the first k points—that's the points in S plus p_k. Why? Becase in this case, points in S and p_k are all turning left, so the resulting polygon is convex, and it includes all first k points (this is because all removal we made are safe).

The complete Gramham-Scan algorithm is illustrated with pseudo-code below.

```
Algorithm Graham-Scan (P)
    calculate point p* in P with smallest y-coordinate;
    sort P in ascending order of the angles w.r.t. p* and (∞, 0);
    rename points in P so that points in P are following above order;
    Graham-Scan-Core (P);
end algorithm;

Algorithm Graham-Scan-Core (P = {p1, p2, ··· , pn})
    init empty stack S;
    push (S, p1);
    push (S, p2);
    push (S, p3);
    for k = 4 to n
        while (S is not empty)
            let pa and pb be the top two elements of S;
            check the orientation of pb → pa → pk;
            if "turning right": pop (S) and then continue the while loop;
            if "turning left": break the while loop;
        end while;
        push (S, pk);
    end for;
end algorithm;
```

The first 3 lines of Gramham-Scan algorithm corresponds to the preprocessing steps, which produce a sorted list of points. The main body of the algorithm, referred to as "Graham-Scan-Core" here, gradually construct the convex hull and guarantees above invariant. When the algorithm terminates, the convex hull of all points are stored in stack S.

Graham-Scan-Core can be regarded as an independent algorithm (you will find its use in the divide-and-

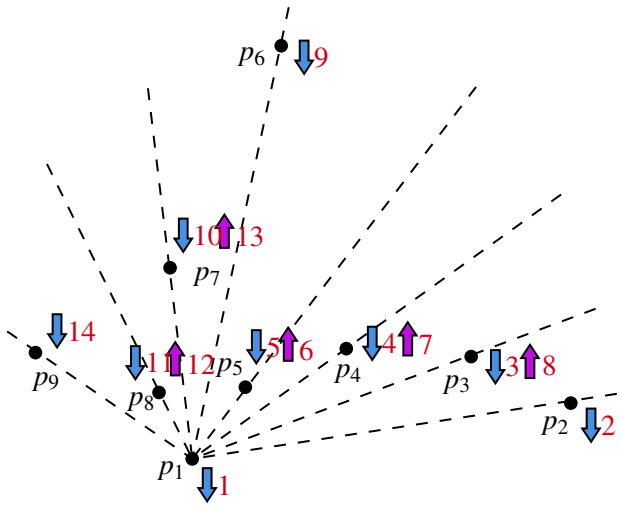


Figure 4: Running Graham-Scan-Core on the example given in Figure 2. Blue and pink arrows show the push and pop operations; the red numbers show the sequential of these operations.

conquer algorithm for convex hull). Its input is a list of points P, in which the first point, i.e. p_1, is the point with smallest y-coordinate, and the following points, i.e. $p_2, p_3, \cdots, p_n$, are placed in ascending order by their angle w.r.t. p_1 and $(+\infty, 0)$. The output of Graham-Scan-Core is $CH(P)$ which will be stored in the stack S.

There are two technical details in implementing above algorithms. First, in the preprocessing step, for each point p_i do we really need to calculate the actual angle of $\angle p_i p_* p_\infty$, where $p_\infty = (+\infty, 0)$? No. Notice that the only purpose of calculating these angles is sorting these points. Therefore, only the relative order matters but not the actual angles. Hence, we can calculate $\cos(\cdot)$ of the angle and use it to sort, as $\cos(\cdot)$ is monotone over $[0, \pi]$. Clearly, $\cos(\angle p_i p_* p_\infty) = \overrightarrow{p_* p_i} \cdot (1,0) / |\overrightarrow{p_* p_i}|$.

Second, in Graham-Scan-Core, how to determine the orientation of 3 points (i.e. $p_b \to p_a \to p_k$)? We will use "right-hand-rule". Define two vectors: $\overrightarrow{A} := \overrightarrow{p_b p_a}$ and $\overrightarrow{B} = \overrightarrow{p_a p_k}$. Let $\overrightarrow{A} = (x_A, y_A)$ and $\overrightarrow{B} = (x_B, y_B)$ be their coordinates, which can be calculated from the coordinates of p_b, p_a and p_k. The right-hand-rule gives that

$$\begin{cases} x_A y_B - x_B y_A > 0 & \text{if and only if } \ p_b \to p_a \to p_k \text{ "turning left"} \\ x_A y_B - x_B y_A < 0 & \text{if and only if } \ p_b \to p_a \to p_k \text{ "turning right"} \\ x_A y_B - x_B y_A = 0 & \text{if and only if } \ p_b, p_a, p_k \text{ "colinear"} \end{cases}$$

Although Graham-Scan-Core consists of two nested loop, in fact it runs in linear time! To see this, consider the types of operations and how many each type of operation the algorithm needs to do. There are three types of operations this algorithm does: *push*, *pop*, and *checking-orientation*, and each operation takes $\Theta(1)$ time. Now let's consider the times of each type will need to execute. First, notice that #(*push*) $= n$, as each point will be pushed to the stack exactly once. Second, #(*pop*) $\le n$, as #(*push*) $\le$ #(*pop*) starting from an empty stack (a point must be pushed into the stack prior to pop). Third, notice that right after *checking-orientation* it is either a *push* or a *pop* operation (i.e., it's not possible to have two *checking-orientation* operations next to each other). This implies that #(*checking-orientation*) $\le$ #(*push*) + #(*pop*) $\le 2n$. Combined, we have that Graham-Scan-Core takes $\Theta(n)$ time.

The preprocessing step of Graham-Scan takes $\Theta(n \log n)$, as it's dominated by the sorting step. The entire running time of Graham-Scan therefore takes $\Theta(n \log n)$ time.

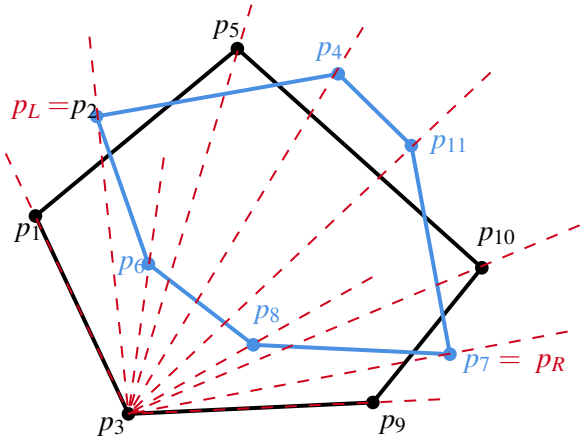
Divide-and-Conquer Algorithm for Convex Hull

We now design a divide-and-conquer algorithm for convex hull problem. As usual, we define recursive function, say, CHDC (P), which takes a set of points P as input and returns $CH(P)$, represented as the list of vertices of the convex polygon in counter-clockwise order.

function CHDC ($P = \{p_1, p_2, \cdots, p_n\}$)
  if $n \leq 3$: resolve this base case and return the convex hull;
  C_1 = CHDC ($P[1..n/2]$);
  C_2 = CHDC ($P[n/2+1..n]$);
  return combine (C_1, C_2);
end algorithm;

The "combine" function in above pseudo-code is missing. Before design an algorithm for "combine", we first make sure that it suffices to only consider points in C_1 and C_2. Formally, we can write $CH(P) = CH(C_1 \cup C_2)$. This can be proved using the Property 2 of convex hull. Intuitively, any point *inside* C_1 or C_2 with be inside $CH(P)$, and therefore won't be *on* $CH(P)$.

We can use, for example, Graham-Scan to calculate $CH(C_1 \cup C_2)$, which takes $\Theta(m \log m)$, where $m = |C_1 \cup C_2|$. Can we do better? Yes. We will design an $\Theta(m)$ time algorithm to calculate $CH(C_1, C_2)$. The idea is to take advantage of that, C_1 and C_2 are already sorted (in counter-clockwise order along the polygon), and then to use Graham-Scan-Core to calculate $CH(C_1 \cup C_2)$. Recall that, Graham-Scan-Core is linear-time algorithm for convex hull; it just requires that all points in its input is sorted in counter-clockwise order w.r.t. the first point in its input.



$C_1 = (p_5, p_1, p_3, p_9, p_{10})$ $\quad C_2 = (p_4, p_2, p_6, p_8, p_7, p_{11})$

$C_1' = (p_3, p_9, p_{10}, p_5, p_1)$ $\quad C_{2a} = (p_7, p_{11}, p_4)$

$C_{2b} = (p_8, p_6, p_2)$

$C_2' = (p_7, p_8, p_{11}, p_4, p_6, p_2)$

$C' = (p_3, p_9, p_7, p_{10}, p_8, p_{11}, p_4, p_5, p_6, p_2, p_1)$

Figure 5: Example of combining C_1 and C_2.

The pseudo-code for combine procedure is given below.

function combine (C_1, C_2)

find p_* in $C_1 \cup C_2$ with smallest y-coordinate;

assume that $p_* \in C_1$; otherwise exchange C_1 and C_2;

rewrite C_1 into C_1' with circular shifting so that p_* is the first point of C_1'; (*Note: now points in C_1' is sorted w.r.t. p_* in counter-clockwise order.*)

find p_R in C_2 with smallest angle (i.e., angle $\angle p_R p_* p_\infty$, where $p_\infty = (+\infty, 0)$;

find p_L in C_2 with largest angle (i.e., angle $\angle p_L p_* p_\infty$, where $p_\infty = (+\infty, 0)$;

let C_{2a} be the sublist of C_2 from p_R to p_L in counter-clockwise order; (*Note: now points in C_{2a} is sorted w.r.t. p_* in counter-clockwise order.*)

let C_{2b} be the sublist of C_2 from p_R to p_L in clockwise order; (*Note: now points in C_{2b} is sorted w.r.t. p_* in counter-clockwise order.*)

C_2' = merge-two-sorted-arrays (C_{2a}, C_{2b}); (*Note: merging is by the angle w.r.t. p_* and p_∞; after it points in C_2' is sorted w.r.t. p_* in counter-clockwise order.*)

C' = merge-two-sorted-arrays (C_1', C_2'); (*Note: merging is by the angle w.r.t. p_* and p_∞; after it points in C', i.e., points in $C_1 \cup C_2$, is sorted w.r.t. p_* in counter-clockwise order.*)

return Graham-Core-Scan (C');

end algorithm;

Clearly each step of above combine takes linear time. Therefore, the entire running time of combine is $\Theta(|C_1| + |C_2|)$.

To obtain the running time of CHDC, we write its recursion $T(n) = 2T(n/2) + \Theta(|C_1| + |C_2|)$. In worst case, $|C_1| + |C_2| = \Theta(n)$, and therefore $T(n) = 2T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$. In this (worst) case the running time of this divide-and-conquer algorithm is the same with Graham-Scan. However, if the size of $|C_1| + |C_2|$ is in the order of $o(n)$, we will have improved running time. For example, if $|C_1| + |C_2| = \Theta(n^{0.99})$, then $T(n) = \Theta(n)$. This might happen in practical cases, although in worst case divide-and-conquer runs in $\Theta(n \log n)$ time.

Half-Plane Intersection

Definitions

Definition 2 (half-planes)**.** A line l on 2D plane with function $y = ax - b$ defines two *half-planes*: the *upper half-plane*: $y \geq ax - b$ and the *lower half-plane*: $y \leq ax - b$.

Definition 3 (upper- and lower-envelop)**.** Let $L = \{y = a_i x - b_i \mid 1 \leq i \leq n\}$ be a set of lines on 2D plane. We define the *upper-envelop* of L, denoted as $UE(L)$, as the intersection of the corresponding n upper half-planes $\{y \geq a_i x - b \mid 1 \leq i \leq n\}$. We define the *lower-envelop* of L, denoted as $LE(L)$, as the intersection of the corresponding n lower half-planes $\{y \leq a_i x - b \mid 1 \leq i \leq n\}$.

Either upper-envolop or lower-envelop of a set of lines can be represented as the list of lines that define its boundary from left to right. In the example below, we can write $UE(L) = (l_1, l_2, l_4, l_7)$ and $LE(L) = (l_7, l_5, l_1)$.

We want to design efficient algorithms to calculate the upper- and lower-envelop of a set of lines. In fact, we don't need to design any new algorithm here. Below we will show that, the problem of finding upper- and lower-envolop of a set of lines is equivalent to the problem of finding the convex hull of a set of points.

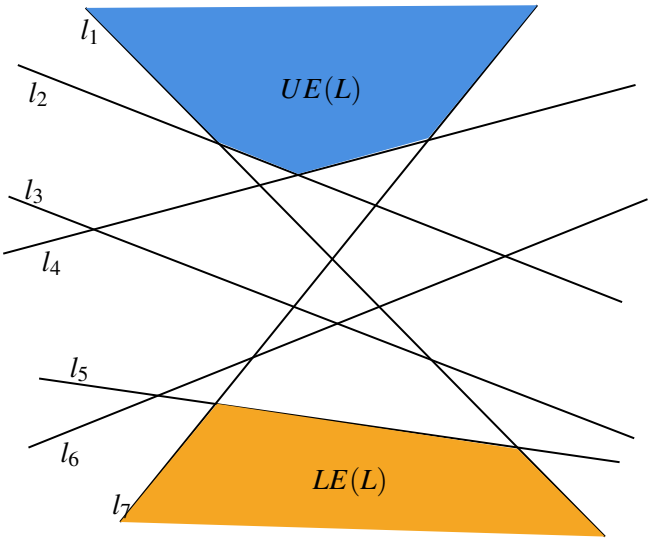


Figure 6: Illustration of upper-envelop and lower-envelop of lines $L = \{l_1, l_2, \cdots, l_7\}$.

Therefore, the algorithms we've designed for finding the convex hull can be directly used to find the upper- and lower-envelop of lines.

Duality

Definition 4 (dual of a point)**.** Let $p = (p_x, p_y)$ be a point on 2D plane. We define the *dual* of p, denoted as p^*, as a line with function $y = p_x x - p_y$ on 2D plane.

Definition 5 (dual of a line)**.** Let l be a line with function $y = ax - b$ on 2D plane. We define its *dual*, denoted as l^*, as a point with coordinates (a, b) on 2D plane.

The following three properties are direct consequences of above definitions. (Think how to prove them.)

Property 3. For any point p, we have $(p^*)^* = p$. For any line l, we have $(l^*)^* = l$.

Property 4. Point p is on line l if and only if point l^* is on line p^*.

Property 5. Point p is above (resp. below) line l if and only if point l^* is above (resp. below) line p^*.

Half-plane Intersection vs. Convex Hull

Definition 6 (upper- and lower-hull)**.** Let P be a set of points, and let $CH(P)$ be the convex hull of P. Let $p_L \in CH(P)$ be the vertex with smallest x-coordinate, and $p_R \in CH(P)$ be the vertex with largest x-coordinate. Therefore p_L and p_R partition $CH(P)$ into two parts: the list of vertices from p_L to p_R following the counter-clockwise order is called *lower hull* of P, denoted as $LH(P)$; the list of vertices from p_R to p_L following the counter-clockwise order is called *upper hull* of P, denoted as $UH(P)$.

We now show that upper- and lower-envelop of lines is essentially the same with lower- and upper-hull of points. We first prove the connection between upper-envelop and lower-hull; the other one, i.e., lower-envelop and upper-hull, can be proved symmetrically.

Let L be a set of lines, we define $L^* = \{l^* \mid l \in L\}$, i.e., the set of "dual points" of L.

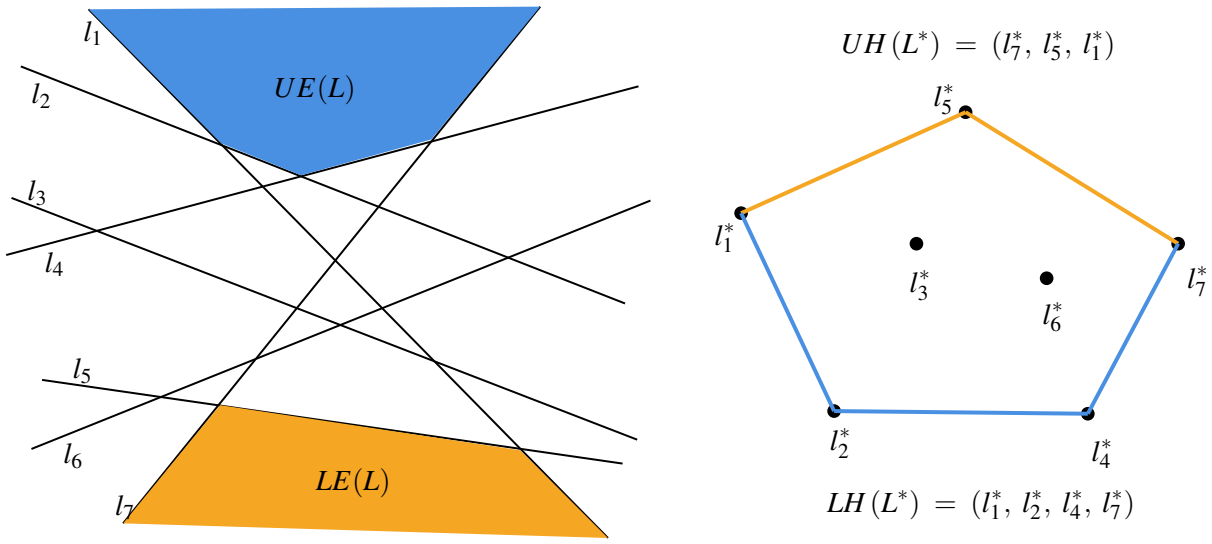


Figure 7: Illustration of duality between upper-/lower-envelop and lower-/upper-hull.

Claim 1. A line $l \in L$ is part of the boundary of $UE(L)$ if and only if l^* is one of the vertices of $LH(L^*)$.

Proof. Line l is part of $UE(L)$, implies that a piece of l is above all other lines. This is equivalent to: there exists a point p, such that p is on l, and that p is above all lines in $L \setminus \{l\}$. This statement is also equivalent to the following statement, by translating everything to their dual counterparts (and applying above Properties of duality): there exists a line p^*, such that l^* is on p^* and that all points in $L^* \setminus \{l^*\}$ are above line p^*. Clearly, this statement is also equivalent to that l^* is one vertex of the lower-hull of L^* (think the Properties of convex hull). □

The above claim shows that lines in $UE(L)$ and vertices in $LH(L^*)$ are in a "dual" relationship. We now show how their ordering are connected. Recall that we represent $UE(L)$ as a list of lines from left to right. Therefore, the *slope* of these lines are in increasing order. As the dual of line $y = ax - b$ is point (a,b), i.e., the slope of a line becomes the x-coordinate of its dual, we know that the corresponing "dual points" of $UE(L)$ are in the increasing order of their x-coordinates.

The above two facts can be combined as the following: $UE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $LH(L^*) = (l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. Formally, we can write

Fact 1. $UE(L) = (LH(L^*))^*$.

Symmetrically, with the same reasonging, we can prove that $LE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $UE(L^*) = (l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. (Recall that $LE(L)$ is represented as the list of lines from left to right, i.e., their slopes are decreasing, while $UH(L^*)$ is represented as the list of vertices from rightmost vertex to leftmost vertex in couter-clockwise order, i.e., their x-coordinates are also decreasing.) Formally, we can also write

Fact 2. $LE(L) = (UH(L^*))^*$.