CMSC 754: Lecture 2
Convex Hulls in the Plane

Reading: Some of the material of this lecture is covered in Chapter 1 in the "4M's" book (by de Berg, Cheong, van Kreveld, and Overmars). The divide-and-conquer algorithm is given in Joseph O'Rourke's, "Computational Geometry in C."

Convex Hulls: In this lecture we will consider a fundamental structure in computational geometry, called the *convex hull*. We will give a more formal definition later, but, given a set P of points in the plane, the convex hull of P, denoted $\text{conv}(P)$, can be defined intuitively by surrounding a collection of points with a rubber band and then letting the rubber band "snap" tightly around the points (see Fig. 1).

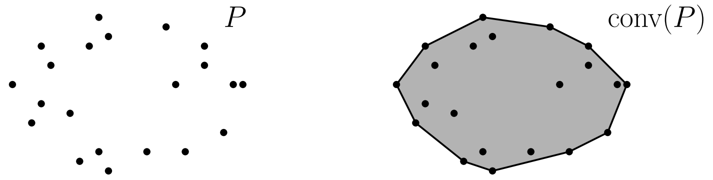

Fig. 1: A point set and its convex hull.

The (planar) *convex hull problem* is, given a discrete set of n points P in the plane, output a representation of P's convex hull. We may assume the points are given as a list of (x, y) coordinates. The convex hull is a closed convex polygon, the simplest representation is a cyclic (say, counterclockwise) enumeration of the vertices of the convex hull. In higher dimensions, the convex hull will be a convex polytope. We will discuss the representation of polytopes in future lectures, but in 3-dimensional space, the representation would consist of a vertices, edges, and faces that constitute the boundary of the polytope.

Applications: There are a number of reasons that the convex hull of a point set is an important geometric structure. One is that it is one of the simplest shape approximations for a set of points. (Other examples include minimum area enclosing rectangles, circles, and ellipses.) It can also be used for approximating more complex shapes. For example, the convex hull of a polygon in the plane or polyhedron in 3-space is the convex hull of its vertices.

Also many algorithms compute the convex hull as an initial stage in their execution or to filter out irrelevant points. For example, the *diameter* of a point set is the maximum distance between any two points of the set. It can be shown that the pair of points determining the diameter are both vertices of the convex hull. Also observe that minimum enclosing convex shapes (such as the minimum area rectangle, circle, and ellipse) depend only on the points of the convex hull.

Basic Definitions: Before getting to the discussion of the various convex-hull algorithms, let's begin with a few standard definitions, which will be useful throughout the semester. For any $d \geq 1$, let $\mathbb{R}^d$ denote real d-dimensional space, that is, the set of d-dimensional vectors over the real numbers. We refer to elements of $\mathbb{R}^d$ either as "points" or "vectors", depending on what we intend them to represent (a location in space or a displacement, respectively). We refer to real numbers as *scalars*.

A point/vector $p \in \mathbb{R}^d$ is expressed as a d-vector $(p_1, \ldots, p_d)$, where $p_i \in \mathbb{R}$. Following standard terminology from linear algebra, given two vectors $u, v \in \mathbb{R}^d$ and a scalar $\alpha \in \mathbb{R}$, let "$u + v$" be the vector-valued sum, αv be scalar-vector product, and let "$u \cdot v$" denote the standard scalar-valued dot-product.

Affine and convex combinations: Given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ in $\mathbb{R}^d$, we can express any point on the (infinite) line $\overleftrightarrow{pq}$ as a linear combination of their coordinates, where the coefficient sum to 1:

$$(1-\alpha)p + \alpha q \;=\; ((1-\alpha)p_x + \alpha q_x, (1-\alpha)p_y + \alpha q_y).$$

This is called an *affine combination* of p and q (see Fig. 2(a)).

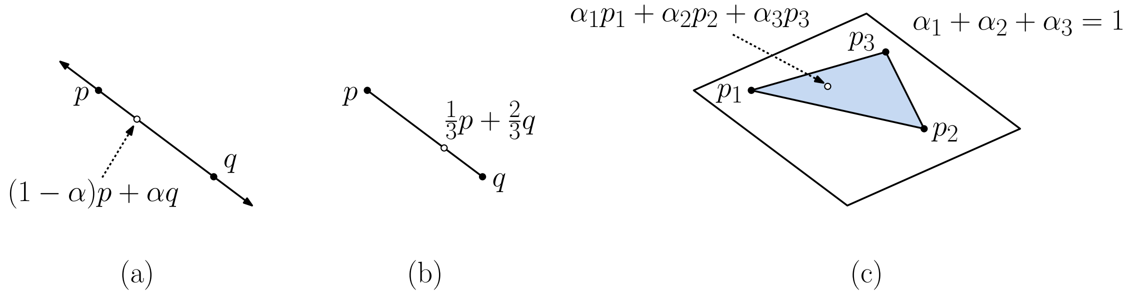
p
q
$(1-\alpha)p + \alpha q$
(a)
p
$\frac{1}{3}p + \frac{2}{3}q$
q
(b)
$\alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$
$\alpha_1 + \alpha_2 + \alpha_3 = 1$
p_3
p_1
p_2
(c)

Fig. 2: Affine and convex combinations.

By adding the additional constraint that $0 \le \alpha \le 1$, the set of points generated lie on the *line segment* $\overline{pq}$ (see Fig. 2(b)). This is called a *convex combination*. Notice that this can be viewed as taking a weighted average of p and q. As α approaches 1, the point lies closer to p and when α approaches zero, the point lies closer to q.

It is easy to extend both types of combinations to more than two points. For example, given k points $\{p_1, \ldots, p_k\}$ an affine combination of these points is the linear combination

$$\sum_{i=1}^{k} \alpha_i p_i, \qquad \text{such that } \alpha_1 + \cdots + \alpha_k = 1.$$

When $0 \le \alpha_i \le 1$ for all i, the result is called a *convex combination*.

The set of all affine combinations of three (non-collinear) points generates a plane, and generally, the resulting set is called the *affine span* or *affine closure* of the points. Given three noncollinear points in $\mathbb{R}^3$, their affine span generates the points on the plane passing through these points. The set of all convex combinations generates the triangle defined by the points. This generalizes in a natural way to all higher dimensions.

Hyperplanes/Halfspaces: Given a nonzero vector $v \in \mathbb{R}^d$ and a scalar $\alpha \in \mathbb{R}$, the set of points $\{p \mid p \cdot v = \alpha\}$ is a $(d-1)$-dimensional affine subspace, more often called a *hyperplane*. In the special cases $\mathbb{R}^2$ and $\mathbb{R}^3$, this defines a *line* and *plane*, respectively. If v is a unit vector and $\alpha \ge 0$, this hyperplane is orthogonal to v and lies at distance α from

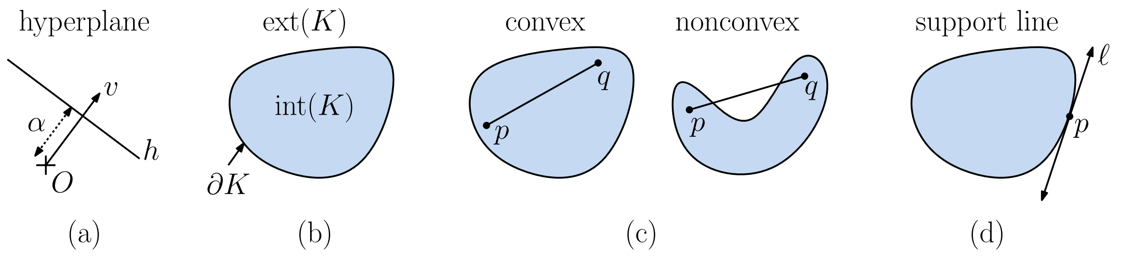

Fig. 3: Basic concepts.

the origin (see Fig. 3(a)). If we change the equality to an inequality, we obtain a *halfspace* consisting of points lying on one side of the hyperplane, for example $\{p \mid p \cdot v \leq \alpha\}$.

Concepts from Topology: There are a number of useful concepts that arise from topology: open and closed sets, interior, exterior, and boundary, connectivity, etc. We will use these without definitions, since for the simple objects that we will be working with, these concepts are easily understood at an intuitive level.[1] We will use $\text{int}(K)$, $\text{ext}(K)$, and ∂K to denote the interior, exterior, and boundary of a set K, respectively (see Fig. 3(b))).

Convexity: A set $K \subseteq \mathbb{R}^d$ is *convex* if given any points $p, q \in K$, the line segment $\overline{pq}$ is entirely contained within K (see Fig. 3(c)). Otherwise, it is called *nonconvex*. This is equivalent to saying that K is "closed" under convex combinations. Examples of convex sets in the plane include circular disks (the set of points contained within a circle), the set of points lying within any regular n-sided polygon, lines (infinite), line segments (finite), rays, and halfspaces.

Support line/hyperplane: An important property of any convex set K in $\mathbb{R}^2$ is that at every point p on the boundary of K, there exists a (not necessarily unique) line ℓ that passes through p such that K lies entirely to one side of ℓ (see Fig. 3(d)). This is called a *supporting line* (or *support line*) for K. In higher dimensions $\mathbb{R}^d$ this will generally by a $(d-1)$-dimensional hyperplane, called a *supporting hyperplane*. Observe that there may generally be multiple (indeed an infinite number of) supporting lines passing through a given boundary point of K. This happens when p is a vertex of K.

Convex Hulls: Formally, the convex hull of a point set P is defined to be smallest convex set containing P. It can be characterized in two provably equivalent ways (one additive and one subtractive). First, it is equal to the set of all convex combinations of points in P and second, it is equal to the intersection of all halfspaces that contain P.

When computing convex hulls, we will usually take P to be a finite set of points. In such a case, $\text{conv}(P)$ will be a convex polygon. Generally P could be an infinite set of points. For example, we could talk about the convex hull of a collection of circles. The boundary of such a shape would consist of a combination of circular arcs and straight line segments.

General Position: As in many of our algorithms, it will simplify the presentation to avoid lots of special cases by assuming that the points are in *general position*. This effectively means that "degenerate" geometric configurations do not arise in the input.

[1]See, for example https://en.wikipedia.org/wiki/Boundary_(topology) for definitions.

What do we mean by "degenerate"? This can be defined formally (by giving a definition on the *measure* of point sets and excluding certain configurations that have measure zero), but for our purposes, we will give a more intuitive, but less rigorous, definition. A *degeneracy* is any geometric property that can be destroyed by some infinitesimal perturbation of the geometric objects (or more accurately, the real-valued parameters that define these objects).

Here are some concrete examples of degenerate configurations (see Fig. 4):

- three points in the plane that are collinear
- two points that share the same x-coordinate
- three lines that pass through the same point (coincident)
- four points that lie on the same circle

Note, by the way that *three* points lying on the same circle is not a degeneracy, since generally three points uniquely determine a circle.

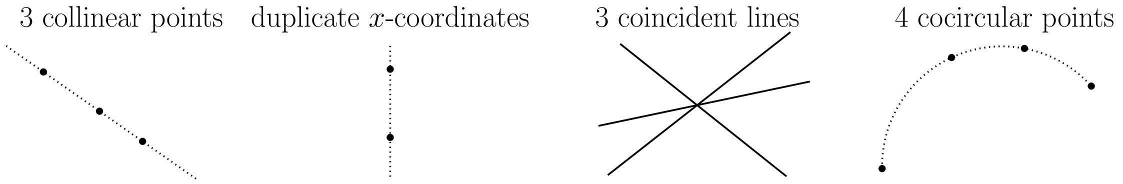

Fig. 4: Examples of degenerate configurations of geometric objects.

When we say that "we assume that our input is in general position", we mean that the input set is free of any degenerate configurations of points (e.g., no three points from the set are collinear). Whenever we do this, we should explain precisely which geometric configurations we are excluding. But, often we will be sloppy and just omit this, since it will usually be obvious by inspection of the algorithm.

Graham's Scan: We will begin with a presentation of a simple $O(n \log n)$ algorithm for the convex hull problem. It is a simple variation of a famous algorithm for convex hulls, called *Graham's Scan*, which dates back to the early 1970's (and named for its inventor Ronald Graham). The algorithm is loosely based on a common approach for building geometric structures called *incremental construction.* In such a algorithm object (points here) are added one at a time, and the structure (convex hull here) is updated with each new insertion.

Let us assume that the input consists of a set P of n points $\{p_1, \ldots, p_n\}$, where $p_i = (x_i, y_i)$. While we will not do so, it is easy to prove that the convex hull of a finite set of points is a convex polygon, whose vertices are a subset of P. A natural representation of such a polygon is a cyclic (for example, counterclockwise) listing of the vertices of this polygon.

An important issue with incremental algorithms is the order of insertion. If we were to add points in some arbitrary order, we would need some method of testing whether the newly added point is inside the existing hull. Instead, we will insert points in increasing order of x-coordinates. This guarantees that each newly added point is outside the current hull. (Note that Graham's original algorithm sorted points in a different way. It found the lowest point in the data set and then sorted points cyclically around this point. Sorting by x-coordinate seems to be a bit easier to implement, however.)

Since we are working from left to right, it would be convenient if the convex hull vertices were themselves ordered from left to right. To do this, we can break the boundary of the convex hull into two *chains*, an *upper chain* consisting of the vertices along the upper part of the hull and a *lower chain* consisting of the vertices along the lower part of the hull. Both chains will start with the leftmost point of P and will end with the rightmost point of P. We will make the general-position assumption that no two vertices have the same x-coordinates, so these two points are unique (see Fig. 5(a)).

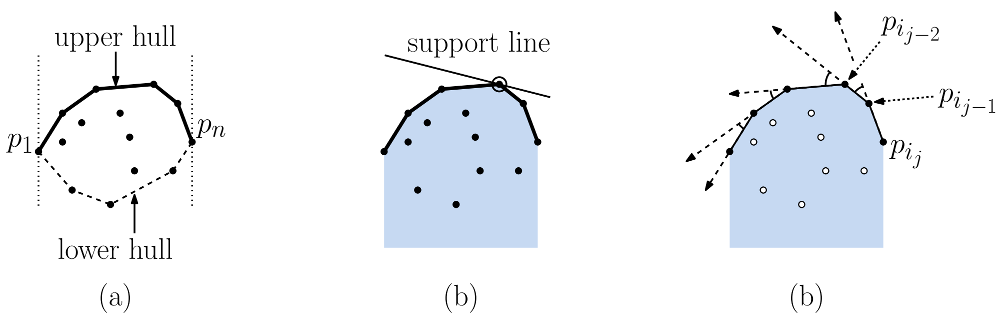

Fig. 5: (a) Upper and lower hulls, (b) supporting line, and (c) the left-hand turn property of points on the upper hull.

It suffices to show how to compute the upper hull, since the lower hull is symmetrical. (Just flip the picture upside down.) Once the two hulls have been computed, we can simply concatenate them with the reversal of the other to form the final hull.

Observe that a point $p \in P$ lies on the upper hull if and only if there is a supporting line passing through p such that all the points of P lie on or below this line (see Fig. 5(b)). Our algorithm will be based on the following lemma, which characterizes the upper hull of P. This is a simple consequence of the convexity. The first part says that the line passing through each edge of the hull is a supporting line, and the second part says that as we walk from right to left along the upper hull, we make successive left-hand turns (see Fig. 5(c)).

Lemma 1: Let $\langle p_{i_1}, \ldots, p_{i_m} \rangle$ denote the vertices of the upper hull of P, sorted from left to right. Then for $1 \le j \le m$, (1) all the points of P lie on or below the line $\overline{p_{i_j} p_{i_{j-1}}}$ joining consecutive vertices and (2) each consecutive triple $\langle p_{i_j} p_{i_{j-1}} p_{i_{j-2}} \rangle$ forms a left-hand turn.

Let $\langle p_1, \ldots, p_n \rangle$ denote the sequence of points sorted by increasing order of x-coordinates. For i ranging from 1 to n, let $P_i = \langle p_1, \ldots, p_i \rangle$. We will store the vertices of the upper hull of P_i on a stack S, where the top-to-bottom order of the stack corresponds to the right-to-left order of the vertices on the upper hull. Let $S[t]$ denote the stack's top. Observe that as we read the stack elements from top to bottom (that is, from right to left) consecutive triples of points of the upper hull form a (strict) left-hand turn (see Fig. 5(b)). As we push new points on the stack, we will enforce this property by popping points off of the stack that violate it.

Turning and orientations: Before proceeding with the presentation of the algorithm, we should first make a short digression to discuss the question of how to determine whether three points form a "left-hand turn." This can be done by a powerful primitive operation, called an *orientation test*, which is fundamental to many algorithms in computational geometry.

Given an ordered triple of points $\langle p, q, r \rangle$ in the plane, we say that they have *positive orientation* if they define a counterclockwise oriented triangle (see Fig. 6(a)), *negative orientation* if they define a clockwise oriented triangle (see Fig. 6(b)), and *zero orientation* if they are collinear, which includes as well the case where two or more of the points are identical (see Fig. 6(c)). Note that orientation depends on the order in which the points are given.

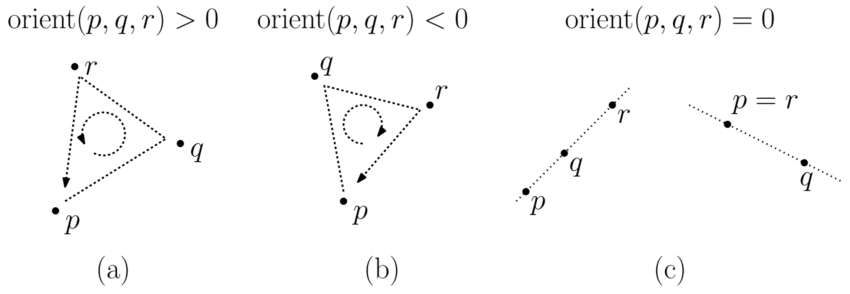

Fig. 6: Orientations of the ordered triple (p, q, r).

Orientation is formally defined as the sign of the determinant of the points given in homogeneous coordinates, that is, by prepending a 1 to each coordinate. For example, in the plane, we define

$$\text{orient}(p, q, r) = \text{sign}\left(\det\begin{pmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{pmatrix}\right).$$

Observe that in the 1-dimensional case, orient(p, q) is just $q - p$. Hence it is positive if $p < q$, zero if $p = q$, and negative if $p > q$. Thus orientation generalizes the familiar 1-dimensional binary relations $<, =, >$.

Also, observe that the sign of the orientation of an ordered triple is unchanged if the points are translated, rotated, or scaled (by a positive scale factor). A reflection transformation (e.g., $f(x, y) = (-x, y)$) reverses the sign of the orientation. In general, applying any affine transformation to the point alters the *sign* of the orientation according to the *sign* of the determinant of the matrix used in the transformation. (By the way, the notion of orientation can be generalized to $d + 1$ points in d-dimensional space, and is related to the notion of *chirality* in Chemistry and Physics. For example, in 3-space the orientation is positive if the point sequence defines a right-handed screw.)

Why use the orientation rather than computing the actual angle? Due to its discrete nature, the orientation test has many uses in computational geometry. This means that if we implement this test once in a manner that is very efficient and numerically very stable, we do not need to worry about designing our own ad hoc geometric primitives.

Given a sequence of three points p, q, r, we say that the sequence $\langle p, q, r \rangle$ makes a (strict) *left-hand turn* if orient$(p, q, r) > 0$.

Graham's Scan Details: We can now present the full algorithm. Recall that the input is a set

P of n points in $\mathbb{R}^2$. We may assume that $n \geq 3$, since a hulls with fewer points are rather trivial.

Let us consider just the case of the upper hull, and let's see what happens when we process the insertion of the ith point, p_i (see Fig. 7(a)). First observe that p_i is on the upper hull of P_i (since it is the rightmost point seen so far). Let p_j be its predecessor on the upper hull of P_i. We know from Lemma 1 that all the points of P_i lie on or below the line $\overline{p_i p_j}$. Let p_{j-1} be the point immediately preceding p_j on the upper hull. We also know from this lemma that $\langle p_i p_j p_{j-1} \rangle$ forms a left-hand turn. Clearly then, if any triple $\langle p_i, S[t], S[t-1] \rangle$ does *not* form a left-hand turn (that is, $\text{orient}(p_i, S[t], S[t-1]) \leq 0$), we may infer that $S[t]$ is *not* on the upper hull, and hence it is safe to delete it by popping it off the stack. We repeat this until we find a left-turning triple (see Fig. 7(b)) or hitting the bottom of the stack. Once this happens, we push p_i on top of the stack, making it the rightmost vertex on the upper hull (see Fig. 7(c)). The algorithm is presented in the code block below.

Graham's Scan

```
(1) Sort the points according to increasing order of their $x$-coordinates, denoted $\langle p_1, p_2, \ldots, p_n \rangle$.
(2) push $p_1$ and then $p_2$ onto $S$.
(3) for $i \leftarrow 3, \ldots, n$ do:
    (a) while ($|S| \geq 2$ and $\text{orient}(p_i, S[t], S[t-1]) \leq 0$) pop $S$.
    (b) push $p_i$ onto $S$.
```

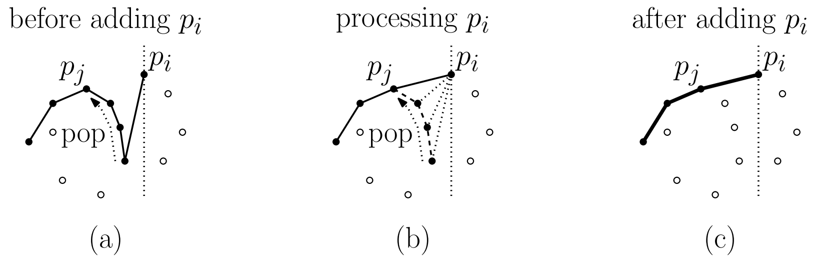

Fig. 7: Graham's scan.

Correctness: The correctness of Graham's scan follows from the following invariant. Let U_i denote the sequence of vertices forming the upper hull of $\{p_1, \ldots, p_i\}$, from left to right.

Lemma: Following any step $i \geq 2$, the stack contains (from bottom to top order) the vertices U_i of the upper convex hull.

Proof: The proof of this lemma naturally involves induction. The basis of induction is trivial, since clearly $U(2)$ just consists of p_1 and p_2.

Let us assume inductively that U_{i-1} has been correctly computed after stage $i-1$, and we will show that U_i is correctly computed after stage i. As we argued before, p_i must be the final vertex of U_i, and indeed the algorithm pushes it last. It remains to show that the algorithm will correctly pop all the points along U_{i-1} that are strictly between p_i and p_j from the stack, but leave p_j itself on the stack.

To see this, let H_{i-1} denote the convex body whose upper hull is U_{i-1} and no lower hull (that is, it extends down to $y = -\infty$) (see Fig. 8(a)). Clearly, all the vertices to be popped from the stack have x-coordinates that lie to the right of p_j and to the left of p_i. Since p_j is the predecessor of p_i on the final upper hull, all of the points of P lie below the supporting line $\ell = \overleftrightarrow{p_j p_i}$, and hence, all the points of U_{i-1} between p_j and p_i lie below the supporting line segment $\overline{p_j p_i}$.

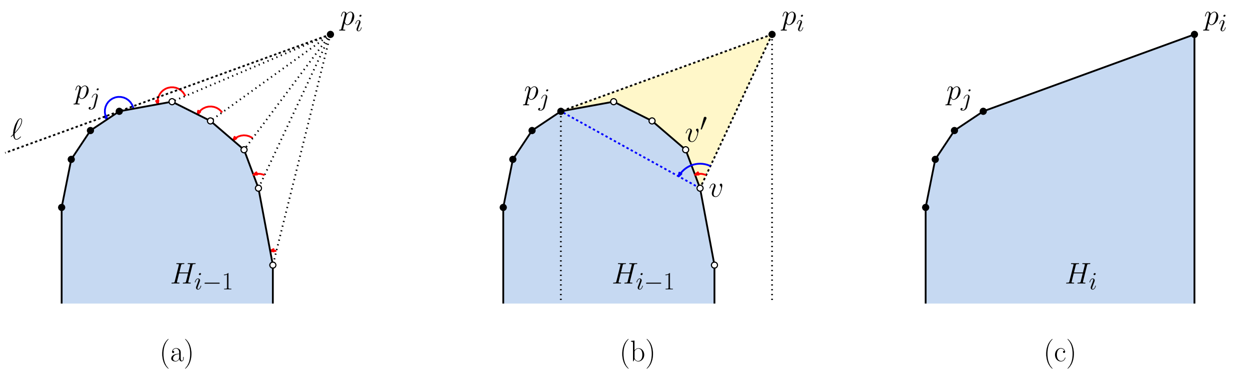

Fig. 8: Correctness of Graham's scan.

Let v denote the point on the stack currently under consideration and let v' denote its predecessor on the stack (see Fig. 8(b)). All the vertices along U_{i-1} between p_j and v lie below the segment $\overline{p_i p_j}$ and above the segment $\overline{p_j v}$. Therefore, they all lie within the triangle $\triangle p_i v p_j$. By basic geometry, the angle $\angle p_i v p_j$ is strictly smaller than 2π, which implies that orient$(p_i, v, p_j) < 0$. It follows that orient$(p_i, v, v') < 0$, and therefore v will be popped off the stack, as desired. By induction, all the vertices of U_{i-1} up to (but not including) p_j will be popped, and p_i will be pushed, as desired (see Fig. 8(c)).

How much detail? Wow, that was a lot of explaining! A question that often arises at this point of the semester is, "How much detail are you expecting in our geometrical proofs of correctness?" While geometric arguments are often aided by figures, you should not rely on your figures to do the "heavy lifting" in your proof. Your proof should reduce the task to a configuration involving a small number of points or lines. From there, you can appeal to easy geometric facts, without the need to spell them all out. There is a bit of art in doing this, and you will gain experience as the course progresses.

Don't be fooled by your drawings: There is a saying that "Geometry is reasoning well from badly drawn figures". Don't be fooled by coincidents that arise in your drawings. For example, if you were to draw Fig. 8 poorly, you might be tempted to assert that p_j is the point on U_{i-1} with the largest y-coordinate. While this might be true, it is clearly not always the case.

Running-time analysis: We will show that Graham's algorithm runs in $O(n \log n)$ time. Clearly, it takes this much time for the initial sorting of the points. After this, we will show that $O(n)$ time suffices for the rest of the computation.

Let d_i denote the number of points that are popped (deleted) on processing p_i. Because each orientation test takes $O(1)$ time, the amount of time spent processing p_i is $O(d_i + 1)$. (The extra +1 is for the last point tested, which is not deleted.) Thus, the total running time is proportional to

$$\sum_{i=1}^{n}(d_i + 1) \;=\; n + \sum_{i=1}^{n} d_i.$$

To bound $\sum_i d_i$, observe that each of the n points is pushed onto the stack once. Once a point is deleted it can never be deleted again. Since each of n points can be deleted at most once, $\sum_i d_i \leq n$. Thus after sorting, the total running time is $O(n)$. Since this is true for the lower hull as well, the total time is $O(2n) = O(n)$.

Convex Hull by Divide-and-Conquer: As with sorting, there are many different approaches to solving the convex hull problem for a planar point set P. Next, we will consider another $O(n \log n)$ algorithm, which is based on divide-and-conquer. It can be viewed as a generalization of the well-known MergeSort sorting algorithm (see any standard algorithms text). Here is an outline of the algorithm. As with Graham's scan, we will focus just on computing the upper hull, and the lower hull will be computed symmetrically.

The algorithm begins by sorting the points by their x-coordinate, in $O(n \log n)$ time. In splits the point set in half at its median x-coordinate, computes the upper hulls of the left and right sets recursively, and then merges the two upper hulls into a single upper hull. This latter process involves computing a common upper supporting line, called the *upper tangent*, for both hulls. The remainder of the algorithm is shown in the code section below.

Divide-and-Conquer (Upper) Convex Hull

```
(1) If $|P| \leq 3$, then compute the upper hull by brute force in $O(1)$ time and return.
(2) Otherwise, partition the point set $P$ into two sets $P'$ and $P''$ of roughly equal sizes by a vertical line.
(3) Recursively compute upper convex hulls of $P'$ and $P''$, denoted $H'$ and $H''$, respectively (see Fig. 9(a)).
(4) Compute the upper tangent $\ell = \overline{p'p''}$ (see Fig. 9(b)) by a process explained below.
(5) Merge the two hulls into a single upper hull by discarding all the vertices of $H'$ to the right of $p'$ and
    the vertices of $H''$ to the left of $p''$ (see Fig. 9(c)).
```

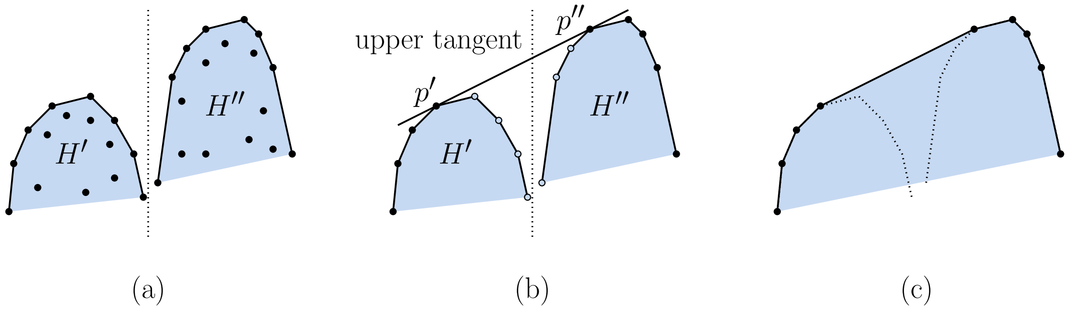

Fig. 9: Divide and conquer (upper) convex hull algorithm.

Computing the Upper Tangent: The only nontrival step is that of computing the common tangent line between the two upper hulls. Our algorithm will exploit the fact that the two hulls are separated by a vertical line. The algorithm operates by a simple "walking procedure."

We initialize p' to be the rightmost point of H' and p'' to be the leftmost point of H'' (see Fig. 10(a)). We will walk p' backwards along H' and walk p'' forwards along H'' until we hit the vertices that define the tangent line.

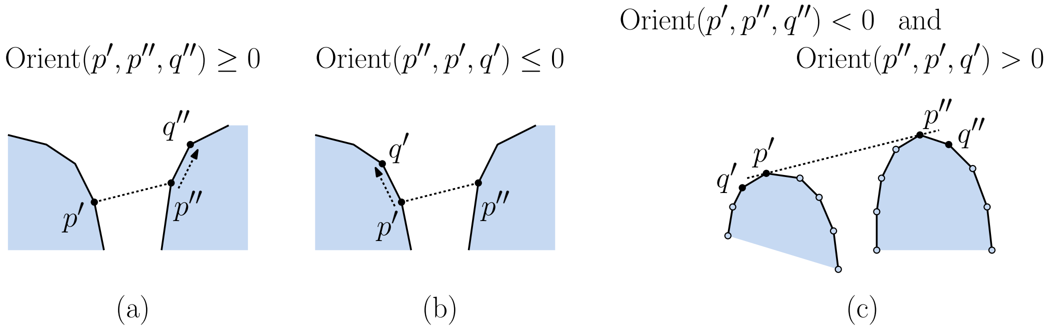
Orient(p', p'', q'') ≥ 0
Orient(p'', p', q') ≤ 0
Orient(p', p'', q'') < 0 and Orient(p'', p', q') > 0
q''
p'
p''
q'
p'
p''
p''
q''
p'
q'
(a)
(b)
(c)

Fig. 10: Computing the upper tangent.

As in Graham's scan, it is possible to determine just how far to walk simply by applying orientation tests. In particular, let q' be the point immediately preceding p' on H', and let q'' be the point immediately following p'' on H''. Observe that if orient$(p', p'', q'') \geq 0$, then we can advance p'' to the next point along H'' (see Fig. 10(a)). Symmetrically, if orient$(p'', p', q') \leq 0$, then we can advance p' to its predecessor along H' (see Fig. 10(b)). When neither of these conditions applies, that is, orient$(p', p'', q'') < 0$ and orient$(p'', p', q') > 0$, we have arrived at the desired points of mutual tangency (see Fig. 10(c)).

There is one rather messy detail in implementing this algorithm. This arises if either q' or q'' does not exist because we have arrived at the leftmost vertex of H' or the rightmost vertex of H''. We can avoid having to check for these conditions by creating two *sentinel points*. We create a new leftmost vertex for H' that lies infinitely below its original leftmost vertex, and we create a new rightmost vertex for H'' that lies infinitely below its original rightmost vertex. The tangency computation will never arrive at these points, and so we do not need to add a special test for the case when q' and q'' do not exist. The algorithm is presented in the following code block.

Computing the Upper Tangent

```
UpperTangent($H', H''$) :
    (1) Let $p'$ be the rightmost point of $H'$, and let $q'$ be its predecessor.
    (2) Let $p''$ be the leftmost point of $H''$, and let $q''$ be its successor.
    (3) Repeat the following until orient$(p', p'', q'') < 0$ and orient$(p'', p', q') > 0$:
        (a) while (orient$(p', p'', q'') \geq 0$) advance $p''$ and $q''$ to their successors on $H''$.
        (b) while (orient$(p'', p', q') \leq 0$) advance $p'$ and $q'$ to their predecessors on $H'$.
    (4) return $(p', p'')$.
```

Running-time analysis: The asymptotic running time of the algorithm can be expressed by a recurrence. Given an input of size n, consider the time needed to perform all the parts of the procedure, ignoring the recursive calls. This includes the time to partition the point set, compute the upper tangent line, and return the final result. Clearly, each of these can be performed in $O(n)$ time, assuming any standard list representation of the hull vertices. Thus, ignoring constant factors, we can describe the running time by the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n \le 3 \\ n + 2T(n/2) & \text{otherwise.} \end{cases}$$

This is the same recurrence that arises in Mergesort. It is easy to show that it solves to $T(n) \in O(n \log n)$ (see any standard algorithms text).

Jarvis's March: Our next convex hull algorithm, called *Jarvis's march*. This algorithm is *output sensitive*, meaning that its running time depends on the number of vertices on the final convex hull. Let n denote the number of input points, and let h denote the number of vertices on the final convex hull. We will see that Jarvis's march computes the convex hull in $O(nh)$ time by a process called "gift-wrapping." In the worst case, $h = n$, so this is inferior to Graham's algorithm for large h, it is superior if h is asymptotically smaller than $\log n$, that is, $h = o(\log n)$.

Jarvis's algorithm begins by identifying any one point of P that is guaranteed to be on the hull, say, the point with the smallest y-coordinate. (As usual, we assume general position, so this point is unique.) Call this v_1. It then repeatedly finds the next vertex on the hull in counterclockwise order.

Given a triple of distinct points $\langle p, q, r\rangle$, define the *turning angle* of r with respect to p and q to be the (CCW) angle between the directed line $\overline{pq}$ and the directed line $\overline{qr}$ (see Fig. 11(a)).

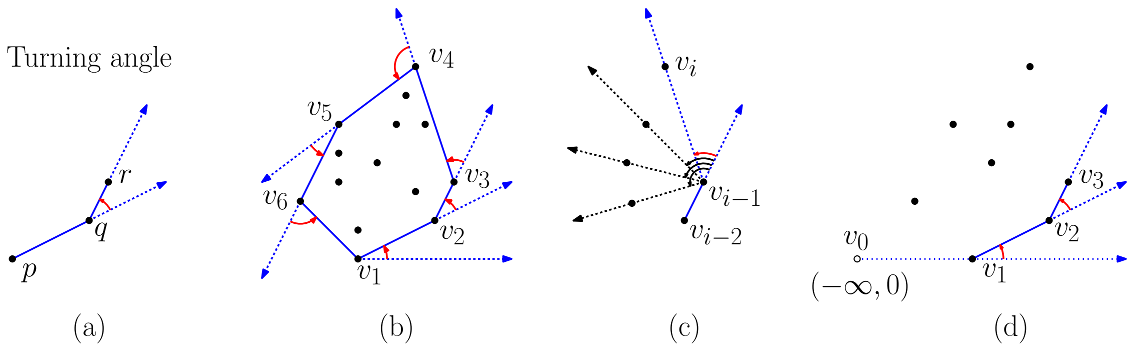

Fig. 11: Jarvis's march.

Jarvis's march works by repeatedly computing the next hull vertex v_i as the point of P that minimizes the turning angle with respect to the prior two, v_{i-2} and v_{i-1} (see Fig. 11(c)). Since we need two points, to get the ball rolling, it is convenient to define an imaginary "sentinel point" $v_0 = (-\infty, 0)$, which has the effect that the initial line $\overline{v_0 v_1}$ is directed horizontally to the right (see Fig. 11(d)).

The algorithm's correctness follows from the fact that (by induction) $\overline{v_{i-2}v_{i-1}}$ is a CCW-directed edge of the hull, and hence the next vertex of the hull is the one that minimizes the turning angle.

By basic trigonometry, turning angles can be computed in constant time. But it is interesting to note that it is possible to compare turning angles just using orientation tests. (Try this

Jarvis's March

```
(1) Given $P$, let $v_0 = (-\infty, 0)$ and let $v_1$ be the point of $P$ with the smallest $y$-coordinate
(2) For $i \leftarrow 2, 3, \ldots$
    (a) $v_i \leftarrow$ the point of $P \setminus \{v_{i-1}, v_{i-2}\}$ that minimizes the turning angle with respect to $v_{i-2}$ and $v_{i-1}$
    (b) If $v_i == v_1$, return $\langle v_1, \ldots, v_{i-1} \rangle$
```

yourself.) This implies that if the input coordinates are integers, the vertices of the hull can be computed exactly (assuming double-precision integer computations).

To obtain the running time, observe that v_1 can be computed in $O(n)$ time, and each iteration can be implemented in $O(n)$ time. After h iterations, the algorithm terminates, so the total running time is $O(n + nh) = O(nh)$.