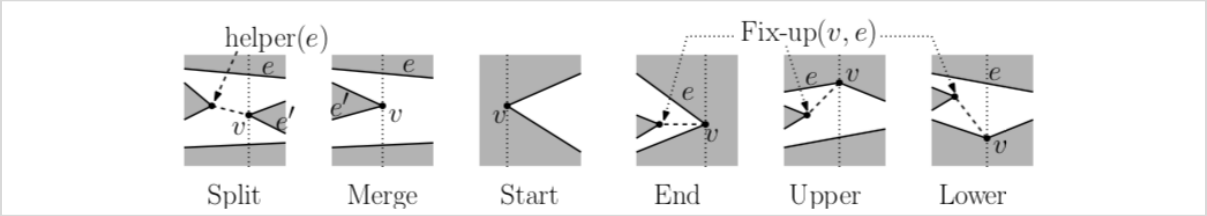
CS 6301.002.20S Lecture 6—January 29, 2020

Main topics are #polygon_triangulation.

Finish Monotone Subdivision

- Last time, we started discussing how to triangulate a simple polygon. We'll finish the discussion today by filing in the last details on how to compute a monotone subdivision of a polygon.
- The algorithm is largely based on the following observation: A polygon is not horizontally monotone if and only if there is a *scan reflex vertex*, a reflex vertex where both incident edges go left or both incident edges go right.
- The book calls the first kind *merge vertices* and the second kind *split vertices*. The goal is for the sweep line to discover these two kinds of vertices, and add diagonals to them when possible.
- We'll do a sweep line algorithm going left to right.
- Let e be an edge with the polygon interior below it. helper(e) is the rightmost vertex u left of the sweep line such that the vertical segment between e and u is entirely in the polygon.
- If e is immediately above or incident to vertex v, then we can add a diagonal back from v to helper(e) at the moment the sweep line hits v.
- Our goal will be to add diagonals from each split vertex back to a helper vertex. We'll also send diagonals back to helper merge vertices the moment before they stop being helper vertices. Each scan reflex vertex will therefore have a diagonal added in the useful direction.
- The sweep line status stores all edges on the sweep line with the polygon interior below them ordered top to bottom along with their helpers.
- Events occur when the sweep line hits a polygon vertex. We can just sort points left to right and then handle each vertex in sorted order, so no priority queue is needed.
- Fix-up(v, e): add a diagonal from v to helper(e) if helper(e) is a merge vertex.
- To keep things concise, I'll just list how to handle each event. The notes for the previous lecture give pseudocode for the whole algorithm.
- Split vertex v:
    - Find edge e immediately above v on sweep line.
    - Add diagonal to helper(e).
    - Let e' be the lower edge incident to v.
    - Add e' to sweep line status.
    - Make v the helper of e and e'.
- Merge vertex v:

    - Let e' be the lower edge incident to v.
    - Delete e' from status.
    - Fix-up(v, e')
    - Find edge e immediately above v on sweep line.
    - Fix-up(v, e).
    - helper(e) ← v
- *Start vertex* v (both incident edges go right and v is not reflex)
    - Let e be the higher incident edge on v.
    - Insert e into status and set helper(e) ← v.
- *End vertex* v (both incident edges go left and v is not reflex)
    - Let e be the higher incident edge on v.
    - Fix-up(v, e)
    - Delete e from status.
- *Upper chain vertex* v (incident edges go both directions and interior is below v)
    - Let e be incident edge left of v and e' be the incident edge to the right.
    - Fix-up(v, e)
    - Replace e with e' in sweep line status and set helper(e') ← v.
- *Lower chain vertex* v:
    - Find edge e immediately above v on sweep line.
    - Fix-up(v, e)
    - helper(e) ← v

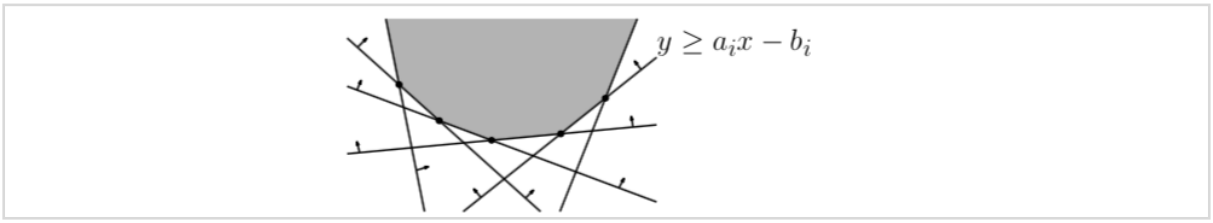


    - By only going for helpers of edges immediately above or incident to each v, we safely add diagonals.
    - We add diagonals going left for every split vertex, and we make sure to add a diagonal to helper merge vertices before their edges go away or they are replaced by new helpers. So the resulting polygons are horizontally monotone.
    - Finally, each event can be handled in O(log n) time and there are n events, so the whole thing + sorting takes O(n log n) time.

Halfplane Intersection

- For the rest of the day, we're going to focus on a another fundamental problem in computational geometry that has some nice connections to things you've seen before.
- A line in the plane partitions the plane into two regions called *halfplanes*.
- An *open* halfplane does not include the line. A *closed* halfplane includes the line.

- Today, we're going to work on the problem of computing the intersection of a set H = {h_1, …, h_n} of closed halfplanes.
- Since each halfplane is a convex set, this intersection will also be a convex set.
- Unlike in the convex hull problem, though, the convex set may be empty or unbounded.
- Like convex hull, halfplane intersection is useful for things like collision detection. It's also a vital component in optimization problems. I'll go into the latter point more on Tuesday.
- To make the problem concrete, we'll represent each halfplane in the following way. First, let's just assume by general position that there are no vertical lines. Any other line can be represented by an equation y = ax - b. Hopefully, the reason I'm using -b will be clear by the end of today's lecture.
- The *lower* halfplane for that line is y ≤ ax - b and the *upper* halfplane is y ≥ ax - b. So, each h_i is some y ? a_i x - b_i.

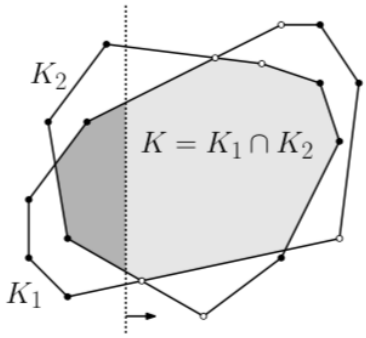


Divide-and-Conquer

- We'll start by briefly sketching a divide-and-conquer algorithm described in detail in the book. I won't spend too much time at, because the combine step is a simple example of plane sweep and what I'll talk about later is much more interesting.
- So, divide-and-conquer is paradigm for making recursive algorithms. You divide the input into simple pieces, conquer each piece by solving the problem recursively, and then combine them into the solution for your original problem.
- HalfplaneIntersectionDAC(H):
  - If n = 1, return h_1
  - divide H into subsets H_1 and H_2 of size floor(n/2) and ceil(n/2).
  - K_1 ← HalfplaneIntersectionDAC(H_1)
  - K_2 ← HalfplaneIntersectionDAC(H_2)
  - K ← intersection of K_1 and K_2
  - return K.
- Suppose it takes M(n) time to compute that intersection.
- If T(n) is the running time, then it follows the recurrence T(n) = 2T(n/2) + M(n)
- We'll use an O(n) time merge procedure, so the whole thing will take T(n) = O(n log n) time just like merge sort.
- The merge procedure is a sweep line algorithm.
- For the status, we need to store the segments of K_1 and K_2 intersected by the algorithm. However, there are at most four segments total, so we can just use a 4-element list to hold

the status and do anything we need in constant time.

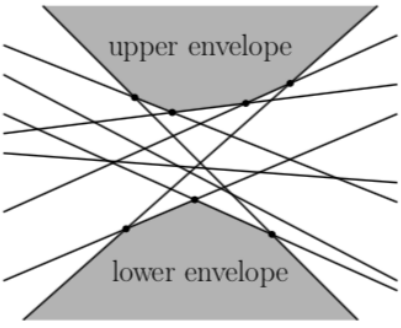
- Similarly, there are at most 8 event points we need to care about at any timeL the endpoints of the current segments and the up to four intersections between these segments. We'll just manually search the constant sized event queue for the next event each time we need one.
- There are only a constant number of possible cases for events and they're relatively easy to handle in constant time each.



- So the running time is proportional to the number of intersections between K_1 and K_2's segments, but that is only O(n) since each segment can enter or leave the other intersection at most once.
- Again, you can look at the book for details.

Point-line Duality

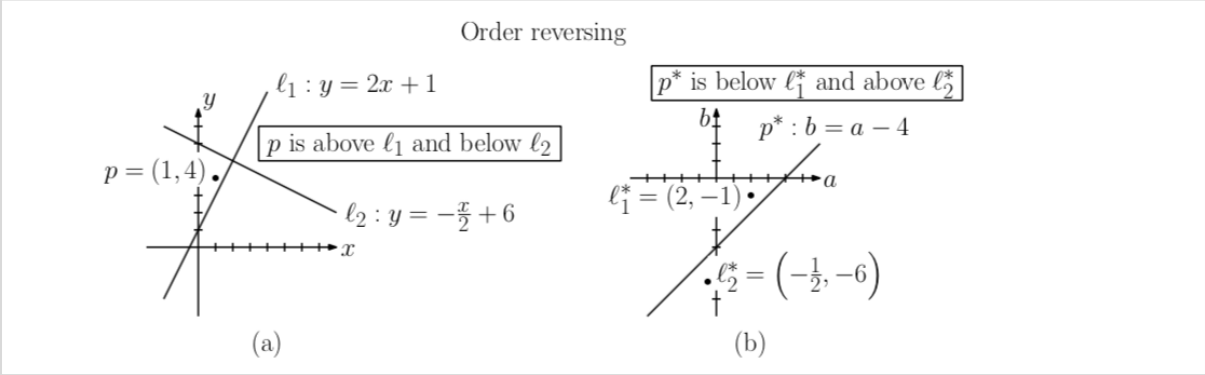
- For the rest of this lecture, I instead want to focus on a variant of halfplane intersection and its connections to an important topic in computational geometry.
- In the *lower envelope* problem, we're given L = {ell_1, …, ell_n} where each ell_i is of the form y = a_i x - b_i. We want to compute the intersection of their lower halfplanes y ≤ a_i x - b_i. The *upper envelope* problem has the symmetric definition.



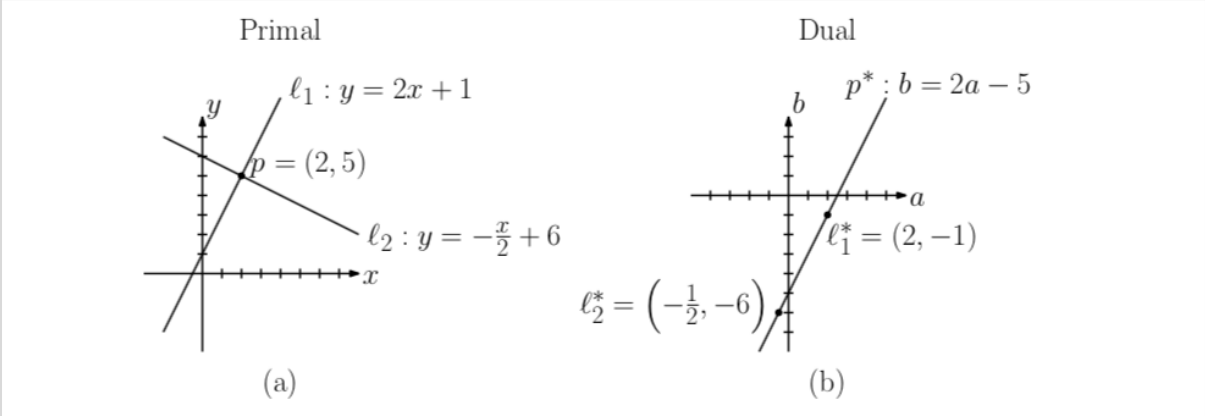
- So the lower envelope problem is just a special case of halfplane intersection. But the cool thing is that it is in some sense the exact same problem as computing an *upper* convex hull.
- Consider a line ell: y = ell_ax - ell_b. We can represent it as the pair (ell_a, ell_b), so you can think of it as a point living in some other 2-dimensional space. We denote this point as

ell*.

- We call the (x, y)-plane the *primal plane*. The new (a, b)-plane is the *dual plane*.
- Now, if you want to go backwards, you'd take the point (ell_a, ell_b) and turn it into the line y = ell_a x - ell_b. But we can do that exact same point to line transformation starting in the primal plane as well.
- So if p = (p_x, p_y), we can let p^* be the line b = p_x a - p_y.
- We have a way to turn primal lines and points into dual points and lines, respectively.
- This transformation is called *point-line duality*. It's a similar concept to other kinds of duality you may have seen such as linear programming duality and planar graph duality.
- And it comes with some interesting properties that are useful for solving certain problems in computational geometry:
  - Self inverse: p^ = p
    - Comes straight from the definition.
  - Order reversing: Point p is above*on*below line ell in the primal plane if and only if point ell^* is above*on*below line p^* in the dual plane
    - p is on or above ell <=> p_y ≥ ell_a p_x - ell_b <=> ell_b ≥ p_x ell_a - p_y <=> ell^* is on or above p^*



  - Intersection preserving: Lines ell_1 and ell_2 intersect at point p if and only if dual line p^* passes through points ell_1^* and ell_2^*.
    - (in other words, two lines determine a point and in the dual the two dual points determine its line)
    - Follows directly from order reversing.

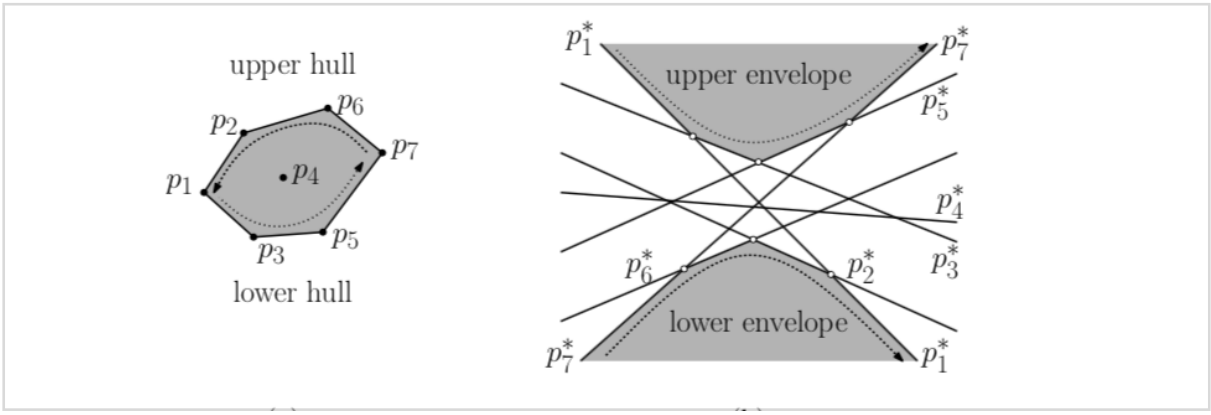


  - Collinearity/Coincidence: Three points are collinear in the primal plane if and only if

their dual lines intersect at a common point.
    - Follows directly from intersection preserving.

Convex Hulls and Envelopes

- So why did I want to introduce point-line duality today? Because, it turns out lower envelope and upper convex hull are dual problems! A solution for one is a solution for the other, and you don't even need to change code.
- Lemma: Let P be a set of points in the plane. The counterclockwise order of points along the upper (lower) convex hull of P is equal to the left-to-right order of the sequence of lines on the lower (upper) envelope of the dual P^*.



- Proof:
    - For simplicity, assume no three points are collinear.
    - Consider consecutive points p_i and p_j on the upper convex hull. All points lie below line ell_{i, j} that passes through them.
    - By intersection preserving, the dual point ell_{i, j}^* is the intersection of dual lines p_i^* and p_j^*.
    - By order reversing, ell_{i, j}^* lies below all the dual lines of P^*. Because ell_{i, j}^ * intersects two of these lines, it must lie on the boundary of the lower envelope.
    - Finally, as we move along the upper convex hull in counterclockwise order, each line's slope increases monotonically. So their dual points' a-coordinates are increasing, placing them in left-to-right order.
- Remember, the dual of the dual is the primal again, and this leads to a nice algorithm for computing a lower envelope:
- Given L, compute the dual points L^*. Compute the upper convex hull of L^ * using your favorite algorithm. The dual of the points on the upper hull are the lines on the lower envelope in order.
- Or, you could look very closely at what Graham's scan would do with those dual points and figure out how to directly manipulate the given lines in the same way.
- One final point. Notice how the upper and lower convex hulls are connected, but the lower and upper envelopes aren't? If you're familiar with *projective geometry*, this may

seem less surprising. Projective geometry kind of says as you go down to the bottom of the dual plane, you'll wrap around again at the top but left and right are flipped. So for the envelopes, we'd go from left to right along the bottom, follow the rightmost line down, and wrap around again on the left of the upper envelope.