CS3110 Spring 2016 Lecture 17: Computational Geometry Algorithms Continued

Robert Constable

1 Lecture Plan

1. Another OCaml logic question: are these specifications programmable? Explain.

   - $(\alpha \to \beta) \to \sim (\alpha \to \sim \beta)$
   - $\alpha \to \beta \to \sim (\alpha \to \sim \beta)$

2. How is CG used?

   - Computer graphics/vision
   - Robotics/CPS/pattern recognition/tracking
   - Computational biology (URMS - unit vector RMS)/chemistry

3. When we use numerical methods, e.g. floats, robustness becomes a problem as the textbooks indicate.

   Numerical stability.

4. Numerical methods for convex hull.

   Look at a numerical version of the incremental algorithm from de Berg Chapter 1.

Convex Hull Algorithm

From de Berg et al. pages 6-8. (This algorithm is similar to the "Jarvis March" algorithm from Cormen pages 1037-1038.)

We now use real numbers and "coordinate geometry" to find the convex hull of a set of points. This algorithm is similar to the problem on PS4.

Theorem: Finding the Convex Hull.

Let P be a list of n points in the Euclidean plane represented as distinct pairs of constructive real numbers.

We can find a sublist of P listing all and only the points of the convex hull of the points on P.

Proof

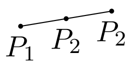
We give the constructive content of the claim and an informal account that this is the correct list of points.

1. Sort the n points of p by their x-coordinate.* There can be multiple points with the same x-coordinate.

   We first build the upper hull from left to right. We start with the first two points, P_1, P_2. Note, $P_1 \neq P_2$ but they might have the same x-coordinate. Pick P_1 as the lowest.

   *Sort in ascending order.[1]

2. Now consider points P_3, $P_4, ...$ starting with $i = 3$ in building the *upper-hull.* For points P_3 to P_n, while upper hull contains more than 2 points and the last three points do *not* make a right turn,

   (eg. not 

) delete the middle point. This computes the *upper hull.* Now put points P_n, P_{n-1} into a list for the lower_ hull and use the same method to compute the lower hull, from P_{n-2} to P_1.

3. Finally remove the first and last points of the lower hull list, to avoid duplicates, and return the combined lists, upper_hull appended to lower_hull

[1]**Note as well: The Graham Scan algorithm in PS4 Exercise 2 page 3 is *NOT* the same as the convex hull algorithm from Lecture 17. Graham Scan starts with a point with the *lowest y coordinate* and sorts the other points by increasing angle. In lecture we started with the lowest x coordinate and sorted by the value of the x-coordinate.**

This finishes the construction. It is easy to see that all points are in the convex hull and no points of the hull are missing. See proof in de Berg.

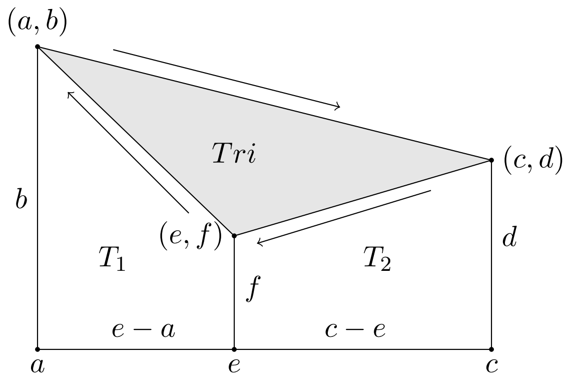
Signed Area

There are several ways to determine whether points make a left or a right turn. The most numerically simple is to us the *signed area test*.

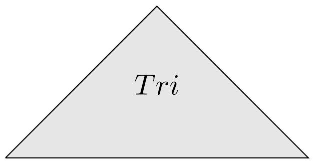
area positive (+) then turns *right*
area negative (-) then turns *left*

the equation in PS4 implements the sign test (Exercise 3)

Signed area of a triangle



The area (signed) of the triangle with base (a,b) to (c,d).



The trapezoid has area $\dfrac{(b+d)(c-a)}{2}$.

The two smaller trapezoids, T_1, T_2 have areas:

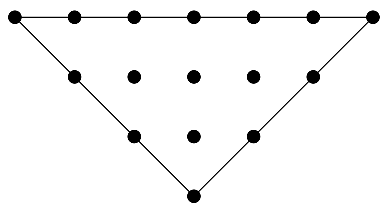
$$T_1 = \frac{(b+f)(e-a)}{2} \qquad T_2 = \frac{(f+d)(c-e)}{2}$$

Hence the area of triangle Tri is

$$\frac{1}{2}((b+d)(c-a) - (b+f)(e-a) - (f+d)(c-e)) \\ = -\frac{1}{2}((ad + be + cf - ed - fa - bc))$$

Note, the determinant is $\begin{vmatrix} a & b & 1 \\ c & d & 1 \\ e & f & 1 \end{vmatrix} = (ad + be + cf - ed - fa - bc)$.

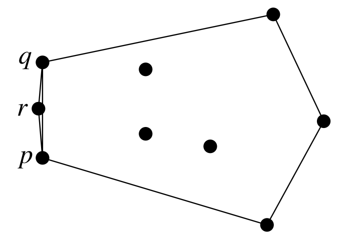
left of the line through p and q, it can also happen that it lies *on* this line. What should we do then? This is what we call a *degenerate case*, or a *degeneracy* for short. We prefer to ignore such situations when we first think about a problem, so that we don't get confused when we try to figure out the geometric properties of a problem. However, these situations do arise in practice. For instance, if we create the points by clicking on a screen with a mouse, all points will have small integer coordinates, and it is quite likely that we will create three points on a line.

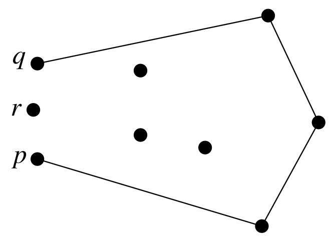


To make the algorithm correct in the presence of degeneracies we must reformulate the criterion above as follows: a directed edge $\overrightarrow{pq}$ is an edge of $\mathcal{CH}(P)$ if and only if all other points $r \in P$ lie either strictly to the right of the directed line through p and q, or they lie on the open line segment $\overline{pq}$. (We assume that there are no coinciding points in P.) So we have to replace line 5 of the algorithm by this more complicated test.

We have been ignoring another important issue that can influence the correctness of the result of our algorithm. We implicitly assumed that we can somehow test exactly whether a point lies to the right or to the left of a given line. This is not necessarily true: if the points are given in floating point coordinates and the computations are done using floating point arithmetic, then there will be rounding errors that may distort the outcome of tests.

Imagine that there are three points p, q, and r, that are nearly collinear, and that all other points lie far to the right of them. Our algorithm tests the pairs (p,q), (r,q), and (p,r). Since these points are nearly collinear, it is possible that the rounding errors lead us to decide that r lies to the right of the line from p to q, that p lies to the right of the line from r to q, and that q lies to the right of the line from p to r. Of course this is geometrically impossible—but the floating point arithmetic doesn't know that! In this case the algorithm will accept all three edges. Even worse, all three tests could give the opposite answer, in which case the algorithm rejects all three edges, leading to a gap in the boundary of the convex hull. And this leads to a serious problem when we try to construct the sorted list of convex hull vertices in the last step of our algorithm. This step assumes that there is exactly one edge starting in every convex hull vertex, and exactly one edge ending there. Due to the rounding errors there can suddenly be two, or no, edges starting in vertex p. This can cause the program implementing our simple algorithm to crash, since the last step has not been designed to deal with such inconsistent data.

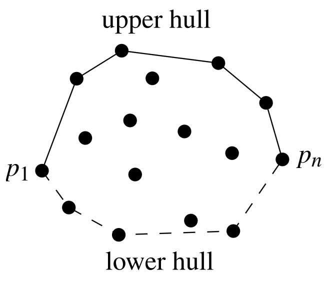




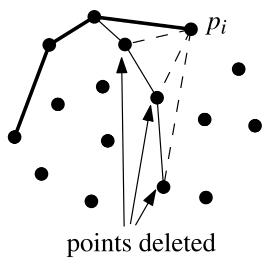
Although we have proven the algorithm to be correct and to handle all special cases, it is not *robust*: small errors in the computations can make it fail in completely unexpected ways. The problem is that we have proven the correctness assuming that we can compute exactly with real numbers.

We have designed our first geometric algorithm. It computes the convex hull of a set of points in the plane. However, it is quite slow—its running time is $O(n^3)$—, it deals with degenerate cases in an awkward way, and it is not robust. We should try to do better.

To this end we apply a standard algorithmic design technique: we will develop an *incremental algorithm*. This means that we will add the points in P one by one, updating our solution after each addition. We give this incremental approach a geometric flavor by adding the points from left to right. So we first sort the points by x-coordinate, obtaining a sorted sequence $p_1,\ldots,p_n$, and then we add them in that order. Because we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right as they occur along the boundary. But this is not the case. Therefore we first compute only those convex hull vertices that lie on the *upper hull*, which is the part of the convex hull running from the leftmost point p_1 to the rightmost point p_n when the vertices are listed in clockwise order. In other words, the upper hull contains the convex hull edges bounding the convex hull from above. In a second scan, which is performed from right to left, we compute the remaining part of the convex hull, the *lower hull*.



The basic step in the incremental algorithm is the update of the upper hull after adding a point p_i. In other words, given the upper hull of the points $p_1,\ldots,p_{i-1}$, we have to compute the upper hull of $p_1,\ldots,p_i$. This can be done as follows. When we walk around the boundary of a polygon in clockwise order, we make a turn at every vertex. For an arbitrary polygon this can be both a right turn and a left turn, but for a convex polygon every turn must be a right turn. This suggests handling the addition of p_i in the following way. Let $\mathcal{L}_{\text{upper}}$ be a list that stores the upper vertices in left-to-right order. We first append p_i to $\mathcal{L}_{\text{upper}}$. This is correct because p_i is the rightmost point of the ones added so far, so it must be on the upper hull. Next, we check whether the last three points in $\mathcal{L}_{\text{upper}}$ make a right turn. If this is the case there is nothing more to do; $\mathcal{L}_{\text{upper}}$ contains the vertices of the upper hull of $p_1,\ldots,p_i$, and we can proceed to the next point, p_{i+1}. But if the last three points make a left turn, we have to delete the middle one from the upper hull. In this case we are not finished yet: it could be that the new last three points still do not make a right turn, in which case we again have to delete the middle one. We continue in this manner until the last three points make a right turn, or until there are only two points left.



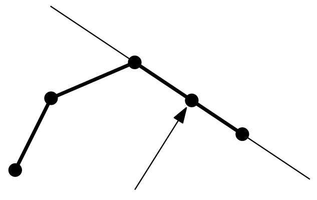
We now give the algorithm in pseudocode. The pseudocode computes both the upper hull and the lower hull. The latter is done by treating the points from right to left, analogous to the computation of the upper hull.

Algorithm CONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

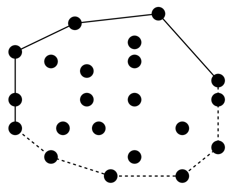
```
1. Sort the points by $x$-coordinate, resulting in a sequence $p_1,\ldots,p_n$.
2. Put the points $p_1$ and $p_2$ in a list $\mathcal{L}_{\text{upper}}$, with $p_1$ as the first point.
3. for $i \leftarrow 3$ to $n$
4.     do Append $p_i$ to $\mathcal{L}_{\text{upper}}$.
5.        while $\mathcal{L}_{\text{upper}}$ contains more than two points and the last three points
                in $\mathcal{L}_{\text{upper}}$ do not make a right turn
6.           do Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$.
7. Put the points $p_n$ and $p_{n-1}$ in a list $\mathcal{L}_{\text{lower}}$, with $p_n$ as the first point.
```

```
8.  for $i \leftarrow n-2$ downto 1
9.      do Append $p_i$ to $\mathcal{L}_{\text{lower}}$.
10.         while $\mathcal{L}_{\text{lower}}$ contains more than 2 points and the last three points
                in $\mathcal{L}_{\text{lower}}$ do not make a right turn
11.             do Delete the middle of the last three points from $\mathcal{L}_{\text{lower}}$.
12. Remove the first and the last point from $\mathcal{L}_{\text{lower}}$ to avoid duplication of the
    points where the upper and lower hull meet.
13. Append $\mathcal{L}_{\text{lower}}$ to $\mathcal{L}_{\text{upper}}$, and call the resulting list $\mathcal{L}$.
14. return $\mathcal{L}$
```

Once again, when we look closer we realize that the above algorithm is not correct. Without mentioning it, we made the assumption that no two points have the same x-coordinate. If this assumption is not valid the order on x-coordinate is not well defined. Fortunately, this turns out not to be a serious problem. We only have to generalize the ordering in a suitable way: rather than using only the x-coordinate of the points to define the order, we use the lexicographic order. This means that we first sort by x-coordinate, and if points have the same x-coordinate we sort them by y-coordinate.



Another special case we have ignored is that the three points for which we have to determine whether they make a left or a right turn lie on a straight line. In this case the middle point should not occur on the convex hull, so collinear points must be treated as if they make a left turn. In other words, we should use a test that returns true if the three points make a right turn, and false otherwise. (Note that this is simpler than the test required in the previous algorithm when there were collinear points.)



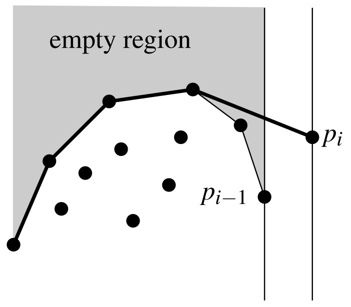
With these modifications the algorithm correctly computes the convex hull: the first scan computes the upper hull, which is now defined as the part of the convex hull running from the lexicographically smallest vertex to the lexicographically largest vertex, and the second scan computes the remaining part of the convex hull.

What does our algorithm do in the presence of rounding errors in the floating point arithmetic? When such errors occur, it can happen that a point is removed from the convex hull although it should be there, or that a point inside the real convex hull is not removed. But the structural integrity of the algorithm is unharmed: it will compute a closed polygonal chain. After all, the output is a list of points that we can interpret as the clockwise listing of the vertices of a polygon, and any three consecutive points form a right turn or, because of the rounding errors, they almost form a right turn. Moreover, no point in P can be far outside the computed hull. The only problem that can still occur is that, when three points lie very close together, a turn that is actually a sharp left turn can be interpretated as a right turn. This might result in a dent in the resulting polygon. A way out of this is to make sure that points in the input that are very close together are considered as being the same point, for example by rounding. Hence, although the result need not be exactly correct—but then, we cannot hope for an exact result if we use inexact arithmetic—it does make sense. For many applications this is good enough. Still, it is wise to be careful in the implementation of the basic test to avoid errors as much as possible.

We conclude with the following theorem:

Theorem 1.1 *The convex hull of a set of n points in the plane can be computed in $O(n \log n)$ time.*

Proof. We will prove the correctness of the computation of the upper hull; the lower hull computation can be proved correct using similar arguments. The proof is by induction on the number of point treated. Before the **for**-loop starts, the list $\mathcal{L}_{\text{upper}}$ contains the points p_1 and p_2, which trivially form the upper hull of $\{p_1, p_2\}$. Now suppose that $\mathcal{L}_{\text{upper}}$ contains the upper hull vertices of $\{p_1, \ldots, p_{i-1}\}$ and consider the addition of p_i. After the execution of the **while**-loop and because of the induction hypothesis, we know that the points in $\mathcal{L}_{\text{upper}}$ form a chain that only makes right turns. Moreover, the chain starts at the lexicographically smallest point of $\{p_1, \ldots, p_i\}$ and ends at the lexicographically largest point, namely p_i. If we can show that all points of $\{p_1, \ldots, p_i\}$ that are not in $\mathcal{L}_{\text{upper}}$ are below the chain, then $\mathcal{L}_{\text{upper}}$ contains the correct points. By induction we know there is no point above the chain that we had before p_i was added. Since the old chain lies below the new chain, the only possibility for a point to lie above the new chain is if it lies in the vertical slab between p_{i-1} and p_i. But this is not possible, since such a point would be in between p_{i-1} and p_i in the lexicographical order. (You should verify that a similar argument holds if p_{i-1} and p_i, or any other points, have the same x-coordinate.)



To prove the time bound, we note that sorting the points lexicographically can be done in $O(n \log n)$ time. Now consider the computation of the upper hull. The **for**-loop is executed a linear number of times. The question that remains is how often the **while**-loop inside it is executed. For each execution of the **for**-loop the **while**-loop is executed at least once. For any extra execution a point is deleted from the current hull. As each point can be deleted only once during the construction of the upper hull, the total number of extra executions over all **for**-loops is bounded by n. Similarly, the computation of the lower hull takes $O(n)$ time. Due to the sorting step, the total time required for computing the convex hull is $O(n \log n)$. ◻

The final convex hull algorithm is simple to describe and easy to implement. It only requires lexicographic sorting and a test whether three consecutive points make a right turn. From the original definition of the problem it was far from obvious that such an easy and efficient solution would exist.

1.2 Degeneracies and Robustness

As we have seen in the previous section, the development of a geometric algorithm often goes through three phases.

In the first phase, we try to ignore everything that will clutter our understanding of the geometric concepts we are dealing with. Sometimes collinear points are a nuisance, sometimes vertical line segments are. When first trying to design or understand an algorithm, it is often helpful to ignore these degenerate cases.

In the second phase, we have to adjust the algorithm designed in the first phase to be correct in the presence of degenerate cases. Beginners tend to do this by adding a huge number of case distinctions to their algorithms. In many situations there is a better way. By considering the geometry of the problem again, one can often integrate special cases with the general case. For example, in the convex hull algorithm we only had to use the lexicographical order instead of the order on x-coordinate to deal with points with equal x-coordinate. For most algorithms in this book we have tried to take this integrated approach to deal with special cases. Still, it is easier not to think about such cases upon first reading. Only after understanding how the algorithm works in the general case should you think about degeneracies.

If you study the computational geometry literature, you will find that many authors ignore special cases, often by formulating specific assumptions on the input. For example, in the convex hull problem we could have ignored special cases by simply stating that we assume that the input is such that no three points are collinear and no two points have the same x-coordinate. From a theoretical point of view, such assumptions are usually justified: the goal is then to establish the computational complexity of a problem and, although it is tedious to work out the details, degenerate cases can almost always be handled without increasing the asymptotic complexity of the algorithm. But special cases definitely increase the complexity of the implementations. Most researchers in computational geometry today are aware that their *general position* assumptions are not satisfied in practical applications and that an integrated treatment of the special cases is normally the best way to handle them. Furthermore, there are general techniques—so-called *symbolic perturbation schemes*—that allow one to ignore special cases during the design and implementation, and still have an algorithm that is correct in the presence of degeneracies.

The third phase is the actual implementation. Now one needs to think about the primitive operations, like testing whether a point lies to the left, to the right, or on a directed line. If you are lucky you have a geometric software library available that contains the operations you need, otherwise you must implement them yourself.

Another issue that arises in the implementation phase is that the assumption of doing exact arithmetic with real numbers breaks down, and it is necessary to understand the consequences. Robustness problems are often a cause of frustration when implementing geometric algorithms. Solving robustness problems is not easy. One solution is to use a package providing exact arithmetic (using integers, rationals, or even algebraic numbers, depending on the type of problem) but this will be slow. Alternatively, one can adapt the algorithm to detect inconsistencies and take appropriate actions to avoid crashing the program. In this case it is not guaranteed that the algorithm produces the correct output, and it is important to establish the exact properties that the output has. This is what we did in the previous section, when we developed the convex hull algorithm: the result might not be a convex polygon but we know that the structure of the output is correct and that the output polygon is very close to the convex hull. Finally, it is possible to predict, based on the input, the precision in