CS3110 Spring 2016 Lecture 16: Computational Geometry Algorithms

Robert Constable

1 Lecture Plan

1. Logic in OCaml Continued
2. Background for geometry
3. Comments on PS4
4. Convex polygons
5. Finding the convex hull of a point set

2 Logic in OCaml Continued

We noted in Lecture 14 that there is a program having the polymorphic type:

$$((\alpha \ \vee \ \beta) \rightarrow \gamma) \rightarrow ((\alpha \rightarrow \gamma) \vee (\beta \rightarrow \gamma)).$$

We showed informally that there is no program in

$$((\alpha \rightarrow \gamma) \vee (\beta \rightarrow \gamma)) \rightarrow ((\alpha \vee \beta) \rightarrow \gamma).$$

(Take $\alpha = \mathrm{T}$, $\beta = \mathrm{F}$, $\gamma = \mathrm{F}$.)

We used this important principle.

If a polymorphic OCaml type is programmable, i.e. there is an OCaml program with that type, then it is a tautology in Boolean logic.

Note, this principle is not “if and only if.” There are tautologies that are not programmable, e.g. $(\sim \beta \rightarrow \sim \alpha) \rightarrow (\alpha \rightarrow \beta)$.

In due course, we will show why this is true. Meanwhile, we will collect more examples. Here is an especially simple one:

(a) $((\alpha \to \beta)\ \&\ (\beta \to \gamma)) \to (\alpha \to \gamma)$

What is this program?

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

Is there a program?

Yes: $(\alpha \to \beta) \to ((\beta \to \text{Void}) \to (\alpha \to \text{Void}))$

$\text{fun } ab \to \text{fun } nb \to \text{ fun } a \to\ nb(ab(a))$

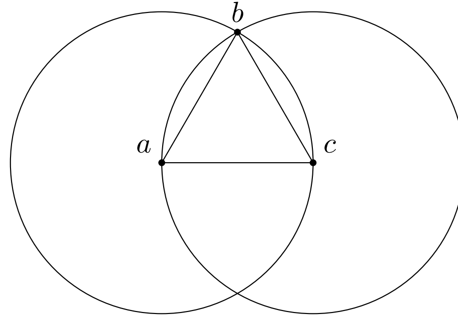
3 Background for Geometry

In the two lectures this week, we will study computational geometry algorithms, specifically algorithms needed for finding the convex hull of a set of points in the plane. We will examine how to implement these algorithms using the constructive real numbers. We will also consider informally how to demonstrate that the algorithms are correct.

The classical approach to geometry represented by Euclid's *Elements* [6, 5] does not rely on understanding real numbers or even rational numbers. The geometric concepts are *abstract*. Historically they remained that way from 350 BCE until the 17th century, and abstract geometry is still an active area of research [7, 8, 1, 2]. Many construction problems could not be resolved in the abstract setting, e.g. the problem of trisecting an angle or "squaring the circle." Nevertheless, geometry was developed to a very high level and was constructive in a fundamental way.

The Greeks tried to find methods of *constructing polygons* and showing that they are congruent or similar. Their method of construction was to use a straight edge and (collapsing) compass. With those tools they could construct polygons, which they called *rectilinear figures*, and prove that two such figures were congruent and that they were similar.

To make these ideas concrete, recall how Euclid, in his first theorem, shows how to construct an equilateral triangle.



3.1 Computational Geometry

Computational geometry (CG) is an important area of theoretical computer science concerned with developing algorithms for efficiently processing geometric data. CG has applications in computer vision, graphics, computational biology, robotics, pattern recognition, and so forth.

Typically geometric algorithms are developed using floating point numbers. In this case there are several subtle issues that must be addressed in evaluating the results. If the algorithms are not sufficiently robust against the inherent limitation on the precision of floating point input and output, they can fail to provide geometrically sensible results. In this course we overcome this limitation by using constructive real numbers. This is a very precise approach, and it can perhaps account for all geometric results and capture precisely all geometric truths.

On the other hand, using constructive real numbers requires considerable mathematical sophistication of the kind not typically taught in undergraduate classes. So we would like to supplement the insights from constructive reals by referencing inherently geometric concepts. There seems to be a notion of geometric truth that can be precisely expressed independently of floating point numbers or even infinite precision constructive real numbers – at least in the view of this author.

It is possible that these truths can be axiomatized and developed deductively using proof assistants such as Agda, Coq, and Nuprl. But this is not yet accomplished in sufficient detail to be taught in college courses. Some Cornell researchers believe this is a viable path to a precise complementary account of geometric concepts that uses real numbers to implement geometric concepts, insights, and intuitive algorithms.

We will cite results from two excellent textbooks [4, 3]. We cite these textbooks and present algorithms from them and from accounts on the web – another excellent source of explanation for the algorithms taught in this part of the course.

3.2 Convex hulls abstractly

We can think about the problem of finding the convex hull of a set of points in the Greek "coordinate free" style. This approach provides the *basic intuition* behind most (perhaps all) convex hull algorithms. This is one precise sense in which our approach is abstract. In this style functional programming is also quite natural as we will see. We let the type of *Points* be the type of the entire Euclidean plane. The type *Points list* is an OCaml like representation of a finite (possibly empty) collection of points in the plane.

We typically want the points to be distinct, so we need to say that $[p_1, ..., p_n]$ is a list of *distinct* points. By this we mean that we do not have $p_i = p_j$ unless $i = j$. So we want the type of Points to come with an equality relation.[1] But we do not assume that equality is decidable on every type. That is, we do not require a Boolean test, $eq(p_i, p_j)$ deciding whether points are equal or not.[2] Expressed another way, we do not assume that for any two points p, q we have $(p = q \vee \neg(p = q))$ where the $\vee$ disjoint union is given by the OCaml variant type as defined in the OCaml propositional type system.

Starting on page 2 of the de Berg et al textbook, *Computational Geometry*, the authors give a coordinate free algorithm for computing the convex hull of a set of points in the Euclidean plane [4]. The input is simply a list of distinct points. They write $p_1, p_2, p_3, ..., p_n$ as the given non-empty list of points. If there are only two distinct points, then there is no convex hull, so we require at least three distinct non-colinear points to state the problem. The first step of the algorithm is to pick two of these points and construct a directed line segment connecting them, say from p_1 to p_2. This provides a base line and a direction. We can speak of the region of the plane to the left and the one to the right of this directed line segment. We

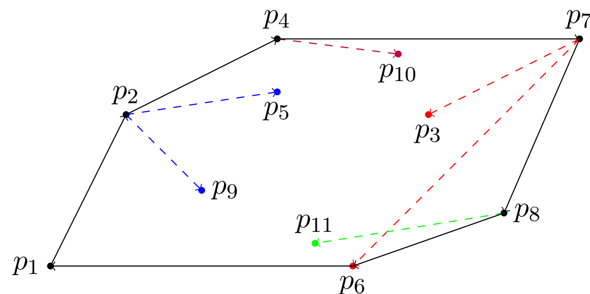
[1] This is a general property of all types in the formal constructive type theories, they come with an equality relation. Without such a relation, it is less clear how we reason precisely about the type, indeed, whether it is a mathematically sensible collection.

[2] Recall that we know that the type of constructive real numbers does not have a decidable equality function.

check the region to the *left of the segment* to see if there is a point r in it. If there is, then we try another pair of points, say p, q and make this check. We continue until we find a pair h_1, h_2 with no points in the right region. We know that this segment h_1, h_2 will be an edge of the convex hull. Now we pick another pair of points that we have not tried before and perform the same tests.

If we know that there is a convex hull of any finite list of points, then this algorithm will find it, but it will be quite slow, n^3 in the length of the list; that is why the method is called *SlowConvexHull* by de Berg et al. Moreover, it assumes that there is such a geometric object without showing us how to construct it. How do we really know that there is always such a geometric object for any finite list of planar points? The textbook hints at the problems with this paragraph on page 5: "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." We can add to this list of issues the fact that it assumes there is a convex hull without giving any reason for us to believe that it exists, except perhaps for the "rubber band" argument, which is hardly simple mathematics. So logically, this purported construction assumes the existence of the object we are tying to construct.

We can be more efficient and still use an abstract coordinate free approach if we try adding points one at a time, say starting from the left most point.



References

[1] Michael J. Beeson. Proof and computation in geometry. In Tetsuo Ida and Jacques Fleuriot, editors, *Automated Deduction in Geometry*,

volume 7993 of *Springer Lecture Notes in Artificial Intelligence*, pages 1–30. Springer, 2013.

[2] Michael J. Beeson. A constructive version of Tarski's geometry. *Annals of Pure and Applied Logic*, 2015.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmans. *Computational Geometry*. Springer-Verlag Berlin Heidelberg, 2008.

[5] Euclid. *The Elements*. Green Lion Press, Santa Fe, New Mexico, 2007.

[6] Euclid. *Elements*. Dover, approx 300 BCE. Translated by Sir Thomas L. Heath.

[7] David Hilbert. *Foundations of Geometry*. Open Court Publishing.

[8] Alfred Tarski. What is elementary geometry? In *The axiomatic method with special reference to geometry and physics: Proceedings of an international symposium held at the University of California, Berkeley, December 26, 1957-January 4, 1958*, Studies in Logic and the Foundation of Mathematics, North Holland, pages 16–29. Brouwer Press, 1959.

These are just three examples of geometric problems requiring carefully designed geometric algorithms for their solution. In the 1970s the field of computational geometry emerged, dealing with such geometric problems. It can be defined as the systematic study of algorithms and data structures for geometric objects, with a focus on exact algorithms that are asymptotically fast. Many researchers were attracted by the challenges posed by the geometric problems. The road from problem formulation to efficient and elegant solutions has often been long, with many difficult and sub-optimal intermediate results. Today there is a rich collection of geometric algorithms that are efficient, and relatively easy to understand and implement.

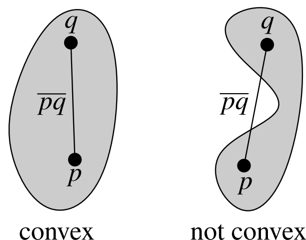
This book describes the most important notions, techniques, algorithms, and data structures from computational geometry in a way that we hope will be attractive to readers who are interested in applying results from computational geometry. Each chapter is motivated with a real computational problem that requires geometric algorithms for its solution. To show the wide applicability of computational geometry, the problems were taken from various application areas: robotics, computer graphics, CAD/CAM, and geographic information systems.

You should not expect ready-to-implement software solutions for major problems in the application areas. Every chapter deals with a single concept in computational geometry; the applications only serve to introduce and motivate the concepts. They also illustrate the process of modeling an engineering problem and finding an exact solution.

1.1 An Example: Convex Hulls

Good solutions to algorithmic problems of a geometric nature are mostly based on two ingredients. One is a thorough understanding of the geometric properties of the problem, the other is a proper application of algorithmic techniques and data structures. If you don't understand the geometry of the problem, all the algorithms of the world won't help you to solve it efficiently. On the other hand, even if you perfectly understand the geometry of the problem, it is hard to solve it effectively if you don't know the right algorithmic techniques. This book will give you a thorough understanding of the most important geometric concepts and algorithmic techniques.

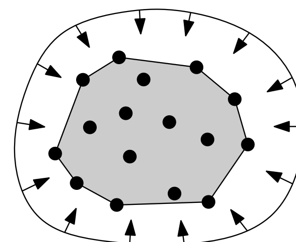
To illustrate the issues that arise in developing a geometric algorithm, this section deals with one of the first problems that was studied in computational geometry: the computation of planar convex hulls. We'll skip the motivation for this problem here; if you are interested you can read the introduction to Chapter 11, where we study convex hulls in 3-dimensional space.



convex not convex

A subset S of the plane is called *convex* if and only if for any pair of points $p,q \in S$ the line segment $\overline{pq}$ is completely contained in S. The *convex hull* $\mathcal{CH}(S)$ of a set S is the smallest convex set that contains S. To be more precise, it is the intersection of all convex sets that contain S.

We will study the problem of computing the convex hull of a finite set P of n points in the plane. We can visualize what the convex hull looks like by a thought experiment. Imagine that the points are nails sticking out of the plane, take an elastic rubber band, hold it around the nails, and let it go. It will snap around the nails, minimizing its length. The area enclosed by the rubber band is the convex hull of P. This leads to an alternative definition of the convex hull of a finite set P of points in the plane: it is the unique convex polygon whose vertices are points from P and that contains all points of P. Of course we should prove rigorously that this is well defined—that is, that the polygon is unique—and that the definition is equivalent to the one given earlier, but let's skip that in this introductory chapter.



How do we compute the convex hull? Before we can answer this question we must ask another question: what does it mean to compute the convex hull? As we have seen, the convex hull of P is a convex polygon. A natural way to represent a polygon is by listing its vertices in clockwise order, starting with an arbitrary one. So the problem we want to solve is this: given a set $P = \{p_1, p_2, \ldots, p_n\}$ of points in the plane, compute a list that contains those points from P that are the vertices of $\mathcal{CH}(P)$, listed in clockwise order.

input = set of points:
$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

output = representation of the convex hull:
p_4, p_5, p_8, p_2, p_9

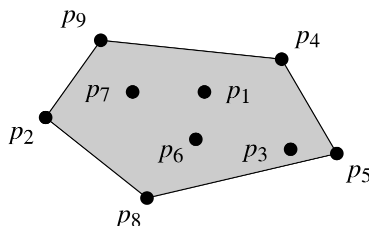
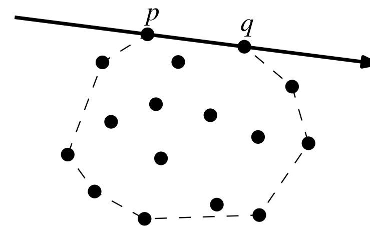


Figure 1.1
Computing a convex hull

The first definition of convex hulls is of little help when we want to design an algorithm to compute the convex hull. It talks about the intersection of all convex sets containing P, of which there are infinitely many. The observation that $\mathcal{CH}(P)$ is a convex polygon is more useful. Let's see what the edges of $\mathcal{CH}(P)$ are. Both endpoints p and q of such an edge are points of P, and if we direct the line through p and q such that $\mathcal{CH}(P)$ lies to the right, then all the points of P must lie to the right of this line. The reverse is also true: if all points of $P \setminus \{p, q\}$ lie to the right of the directed line through p and q, then $\overline{pq}$ is an edge of $\mathcal{CH}(P)$.



Now that we understand the geometry of the problem a little bit better we can develop an algorithm. We will describe it in a style of pseudocode we will use throughout this book.

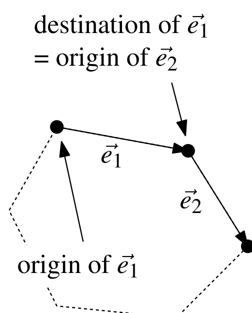
Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.
1. $E \leftarrow \emptyset$.
2. **for** all ordered pairs $(p, q) \in P \times P$ with p not equal to q
3. **do** *valid* $\leftarrow$ **true**

4.   **for** all points $r \in P$ not equal to p or q
5.   **do if** r lies to the left of the directed line from p to q
6.   **then** $valid \leftarrow$ **false**.
7.   **if** $valid$ **then** Add the directed edge $\overrightarrow{pq}$ to E.
8. From the set E of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, sorted in clockwise order.

Two steps in the algorithm are perhaps not entirely clear.

The first one is line 5: how do we test whether a point lies to the left or to the right of a directed line? This is one of the primitive operations required in most geometric algorithms. Throughout this book we assume that such operations are available. It is clear that they can be performed in constant time so the actual implementation will not affect the asymptotic running time in order of magnitude. This is not to say that such primitive operations are unimportant or trivial. They are not easy to implement correctly and their implementation will affect the actual running time of the algorithm. Fortunately, software libraries containing such primitive operations are nowadays available. We conclude that we don't have to worry about the test in line 5; we may assume that we have a function available performing the test for us in constant time.

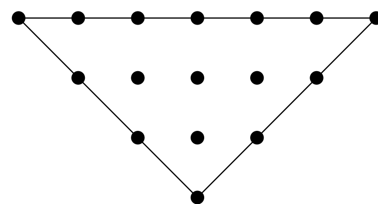
The other step of the algorithm that requires some explanation is the last one. In the loop of lines 2–7 we determine the set E of convex hull edges. From E we can construct the list $\mathcal{L}$ as follows. The edges in E are directed, so we can speak about the origin and the destination of an edge. Because the edges are directed such that the other points lie to their right, the destination of an edge comes after its origin when the vertices are listed in clockwise order. Now remove an arbitrary edge $\vec{e}_1$ from E. Put the origin of $\vec{e}_1$ as the first point into $\mathcal{L}$, and the destination as the second point. Find the edge $\vec{e}_2$ in E whose origin is the destination of $\vec{e}_1$, remove it from E, and append its destination to $\mathcal{L}$. Next, find the edge $\vec{e}_3$ whose origin is the destination of $\vec{e}_2$, remove it from E, and append its destination to $\mathcal{L}$. We continue in this manner until there is only one edge left in E. Then we are done; the destination of the remaining edge is necessarily the origin of $\vec{e}_1$, which is already the first point in $\mathcal{L}$. A simple implementation of this procedure takes $O(n^2)$ time. This can easily be improved to $O(n \log n)$, but the time required for the rest of the algorithm dominates the total running time anyway.



Analyzing the time complexity of SLOWCONVEXHULL is easy. We check $n^2 - n$ pairs of points. For each pair we look at the $n-2$ other points to see whether they all lie on the right side. This will take $O(n^3)$ time in total. The final step takes $O(n^2)$ time, so the total running time is $O(n^3)$. An algorithm with a cubic running time is too slow to be of practical use for anything but the smallest input sets. The problem is that we did not use any clever algorithmic design techniques; we just translated the geometric insight into an algorithm in a brute-force manner. But before we try to do better, it is useful to make several observations about this algorithm.

We have been a bit careless when deriving the criterion of when a pair p,q defines an edge of $\mathcal{CH}(P)$. A point r does not always lie to the right or to the

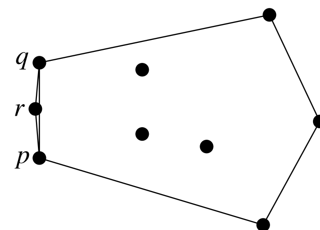
left of the line through p and q, it can also happen that it lies *on* this line. What should we do then? This is what we call a *degenerate case*, or a *degeneracy* for short. We prefer to ignore such situations when we first think about a problem, so that we don't get confused when we try to figure 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.

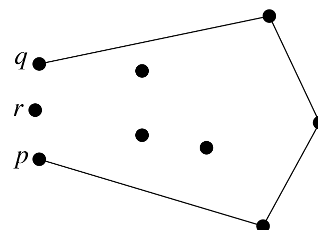


To make the algorithm correct in the presence of degeneracies we must reformulate the criterion above as follows: a directed edge $\overrightarrow{pq}$ is an edge of $\mathcal{CH}(P)$ if and only if all other points $r \in P$ lie either strictly to the right of the directed line through p and q, or they 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.