CS3110 Spring 2017 Lecture 14: Computational Geometry and Convex Hulls

Robert Constable

1 Lecture Plan

1. Cantor's Theorem, as stated by Bishop [1].
2. Background for geometry
3. Finding the convex hull of a point set

2 Cantor's theorem for the constructive reals

Here is a topic from Lecture 13 that I wanted to write a bit more about. I am considering extra credit for anyone who implements Cantor's Theorem as stated by Bishop [1], Theorem 2.19 (page 27): Let a_n be a computable sequence of real numbers, then we can construct a real number x that is not on the list, i.e. for all n, $x \neq a_n$.

The full version of Theorem 2.19 in the book is essentially this. The wording is changed a bit to stress that we can construct the real number x. The proof itself gives that construction exactly. This is the character of constructive mathematics, the proofs often provide implicit algorithms. In many cases we can directly "see the algorithm" from the proof. Even when we can't see it clearly, there are systematic methods of extracting the object claimed to exist from the proof itself. This is the basis of *program extraction from constructive proofs* that Cornell pioneered.

Full Theorem 2.19 Let (a_n) be a sequence of real numbers, and let x_0 and y_0 be real numbers with $x_0 < y_0$. Then we can construct a real number x such that $x_0 \leq x \leq y_0$ and $x \neq a_n$ for all positive integers n.

Recall defintion 2.12 from Bishop (page 24 at top) : $x \neq y$ *iff* $x < y$ or $x > y$. In this case, we say that x and y are separated.

3 Background for Geometry

In this lecture we will study computational geometry algorithms, specifically algorithms needed for finding the convex hull of a set of points in the plane. We will discuss how to implement these algorithms using the constructive real numbers. We will also consider informally how to demonstrate that the algorithms are correct. When using reals in the algorithm, it is considered analytic geometry. We can also build algorithms closer to the style of Euclid, without the use of the reals. This is sometimes called synthetic geometry; we use that term here at Cornell.

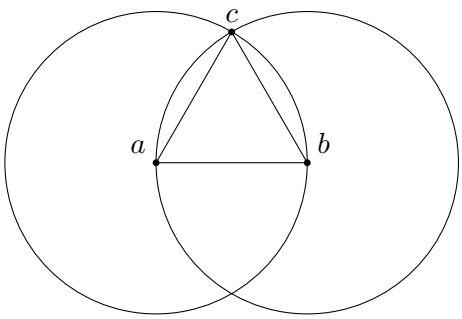
The classical approach to geometry represented by Euclid's *Elements* [4, 3] does not rely on understanding real numbers or even rational numbers. The geometric concepts are *abstract*. Historically they remained that way from 350 BCE until the 17th century. Many construction problems could not be resolved 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 for *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 (referred to historically as propositions), shows how to construct an equilateral triangle.

Proposition 1

To construct an equailateral triangle on a given finite straight line.



It is required to construct an equilateral triangle on the straight line ac.

Describe the circle with center a and radius ab.

Describe the circle with center b and radius ba.

Let c be the intersection of the circles.

Join the straight lines ba and bc from the point b at which the circles cut one another to the points a and b.

Now, since the point a is the center of the circle ab, therefore ac equals ab. Again, since the point b is the center of the circle ba, therefore bc equals ba.

But ac was proved equal to ab , therefore each of the straight lines ac and bc equals ab. And things which equal the same thing also equal one another, therefore ac also equals bc.

Therefore the three straight lines ac, ab and cb equal one another.

Therefore the triangle abc is equilateral, and it has been constructed on the given finite straight line ab.

3.1 Discussion

What are the primitive geometric concepts necessary for Proposition 1? One concept is the primitive notion of equality. To prove the theorem constructively, we can't rely on testing whether points are equal. *Why not?*

Proposition 1 emphasizes the necessity of the constructive notion of separate, $a \neq b$: if $\sim (a \neq b)$ then there is not a segment on which to begin

the construction. We can test $a \neq b$ as with the constructive reals, and we can define the notion of equlaity as $\sim (a \neq b)$.

$$(1)\ x \neq y \Rightarrow \forall z : Point.\ (z \neq x \vee z \neq y).$$

With this we can now prove $\sim (x \neq y)\ \&\ \sim (y \neq z)\ \Rightarrow \sim (x \neq z)$.

Assume $x \neq z$, then from (1) $(y \neq x\ \vee\ y \neq z)$ but each case contradicts $\sim (x \neq y)\ \&\ \sim (y \neq z)$.

Hence, $\sim (x \neq z)$.

4 Convex Hulls Abstractly

We can think about the problem of finding the convex hull of a set of points in the abstract way, in the Greek style. In this style functional programming is quite natural. We let the type of *Points* be the type of the 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 Points to come with an equality relation. But we do not assume that equality is decidable[1], that there there is no Boolean test, $eq(p_i, p_j)$ or said another way, we do not assume that for any two points p, q we have $(p = q\ v\ \neg(p = q))$.

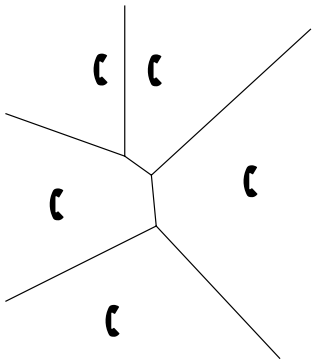
Starting on page 2 of the de Berg et al textbook, *Computational Geometry*, the authors give a coordinate free algorithm for computing the convex hull of a set of points in the Euclidean plane [2].

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

1 Computational Geometry

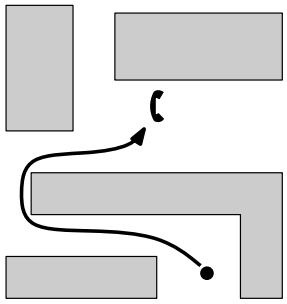
Introduction

Imagine you are walking on the campus of a university and suddenly you realize you have to make an urgent phone call. There are many public phones on campus and of course you want to go to the nearest one. But which one is the nearest? It would be helpful to have a map on which you could look up the nearest public phone, wherever on campus you are. The map should show a subdivision of the campus into regions, and for each region indicate the nearest public phone. What would these regions look like? And how could we compute them?



Even though this is not such a terribly important issue, it describes the basics of a fundamental geometric concept, which plays a role in many applications. The subdivision of the campus is a so-called *Voronoi diagram*, and it will be studied in Chapter 7 in this book. It can be used to model trading areas of different cities, to guide robots, and even to describe and simulate the growth of crystals. Computing a geometric structure like a Voronoi diagram requires geometric algorithms. Such algorithms form the topic of this book.

A second example. Assume you located the closest public phone. With a campus map in hand you will probably have little problem in getting to the phone along a reasonably short path, without hitting walls and other objects. But programming a robot to perform the same task is a lot more difficult. Again, the heart of the problem is geometric: given a collection of geometric obstacles, we have to find a short connection between two points, avoiding collisions with the obstacles. Solving this so-called *motion planning* problem is of crucial importance in robotics. Chapters 13 and 15 deal with geometric algorithms required for motion planning.



A third example. Assume you don't have one map but two: one with a description of the various buildings, including the public phones, and one indicating the roads on the campus. To plan a motion to the public phone we have to *overlay* these maps, that is, we have to combine the information in the two maps. Overlaying maps is one of the basic operations of geographic information systems. It involves locating the position of objects from one map in the other, computing the intersection of various features, and so on. Chapter 2 deals with this problem.

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.

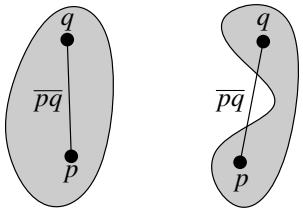
This book describes the most important notions, techniques, algorithms, and data structures from computational geometry in a way that we hope will be attractive to readers who are interested in applying results from computational geometry. Each chapter is motivated with a real computational problem that 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.

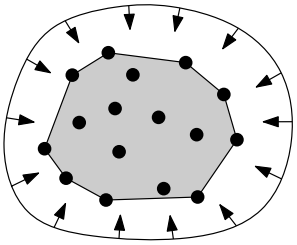
To illustrate the issues that arise in developing a geometric algorithm, this section deals with one of the first problems that was studied in computational geometry: the computation of planar convex hulls. We'll skip the motivation for this problem here; if you are interested you can read the introduction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

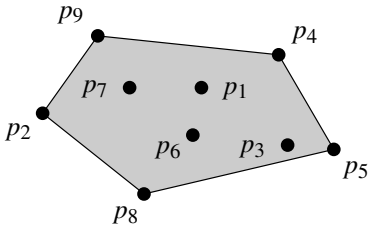
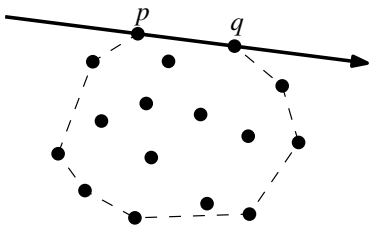


Figure 1.1
Computing a convex hull

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.

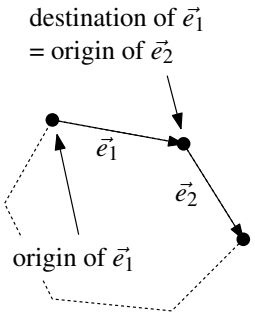
Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.
1. $E \leftarrow \emptyset$.
2. **for** all ordered pairs $(p,q) \in P \times P$ with p not equal to q
3. **do** *valid* $\leftarrow$ **true**

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.

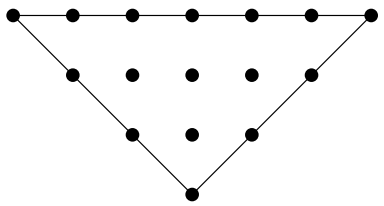
The other step of the algorithm that requires some explanation is the last one. In the loop of lines 2–7 we determine the set E of convex hull edges. From E we can construct the list $\mathcal{L}$ as follows. The edges in E are directed, so we can speak about the origin and the destination of an edge. 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

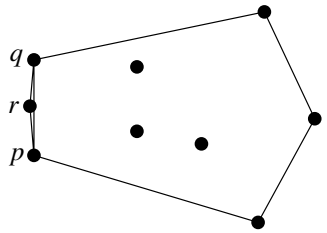
left of the line through p and q, it can also happen that it lies *on* this line. What should we do then? This is what we call a *degenerate case*, or a *degeneracy* for short. We prefer to ignore such situations when we first think about a problem, so that we don't get confused when we try to figure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

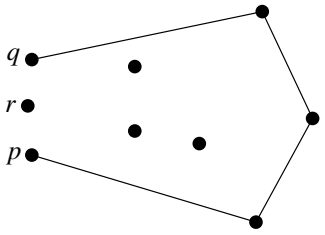


To make the algorithm correct in the presence of degeneracies we must reformulate the criterion above as follows: a directed edge $\overrightarrow{pq}$ is an edge of $\mathcal{CH}(P)$ if and only if all other points $r \in P$ lie either strictly to the right of the directed line through p and q, or they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

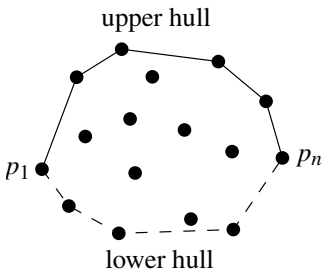
Imagine that there are three points p, q, and r, that are nearly collinear, and that all other points lie far to the right of them. Our algorithm tests the pairs (p,q), (r,q), and (p,r). Since these points are nearly collinear, it is possible that the rounding errors lead us to decide that r lies to 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.



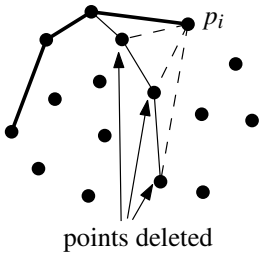


Although we have proven the algorithm to be correct and to handle all special cases, it is not *robust*: small errors in the computations can make it fail in completely unexpected ways. The problem is that we have proven the correctness assuming that we can compute exactly with real numbers.

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.



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.

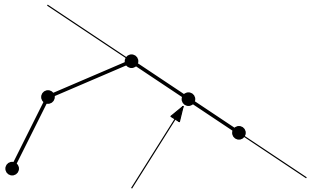
We now give the algorithm in pseudocode. The pseudocode computes both the upper hull and the lower hull. The latter is done by treating the points from right to left, analogous to the computation of the upper hull.

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.

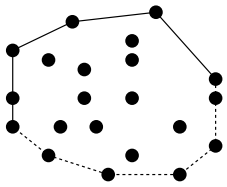
8. **for** $i \leftarrow n-2$ **downto** 1
9. **do** Append p_i to $\mathcal{L}_{\text{lower}}$.
10. **while** $\mathcal{L}_{\text{lower}}$ contains more than 2 points **and** the last three points in $\mathcal{L}_{\text{lower}}$ do not make a right turn
11. **do** Delete the middle of the last three 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.)


not a right turn

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.

References

[1] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

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

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

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