CS 164 & CS 266: Computational Geometry

Lecture 1

Convex hulls and numerical issues

David Eppstein
University of California, Irvine

Fall Quarter, 2023

General Course Information

Instructor: David Eppstein

Teaching assistant: Alvin Chiu

Course info: Canvas and `https://www.ics.uci.edu/~eppstein/164`

Weekly problem sets (not graded!), two midterms, and final exam

Exams are not cumulative

Why are the problem sets not graded?

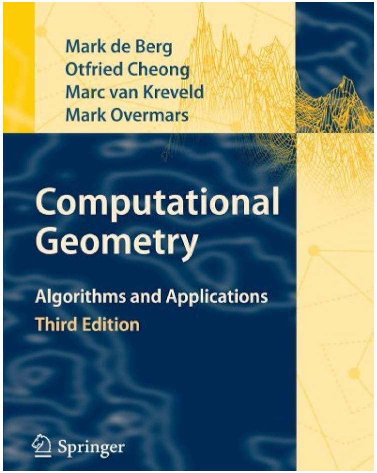
Main reason: I don't want to give an unfair advantage to students who look up the answers, compared to the students who do their own work

Secondary reasons:

- Similar questions will be on the exams (or in some cases the same), so having them in problem sets (rather than keeping homework and exam questions separate) should provide more opportunity for practice
- I can assign problems that match each week's lectures, rather than delaying them until those lectures are complete

Takeaway message: you should set aside time to work on the problems because it will show up in your exam scores

Textbook


Mark de Berg
Otfried Cheong
Marc van Kreveld
Mark Overmars

Computational Geometry

Algorithms and Applications
Third Edition

Springer

Computational Geometry: An Introduction (3rd ed.)

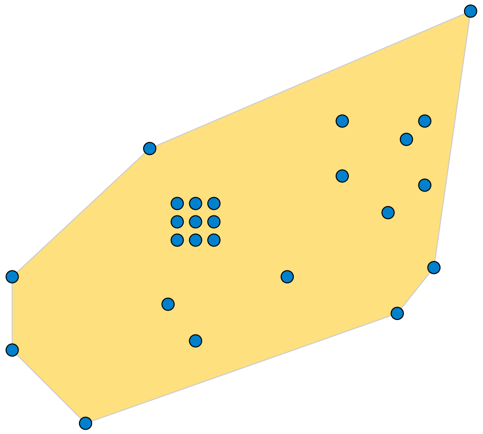
Mark de Berg, Otfried ("Mark") Cheong, Marc van Kreveld, Mark Overmars

Strongly recommended and free online from UCI internet addresses via a campus subscription to Springer books:
`https://link.springer.com/book/10.1007/978-3-540-77974-2`

Syllabus provides readings corresponding to lecture content

Convex hulls

Motivating question: What is the shape of a set of points?



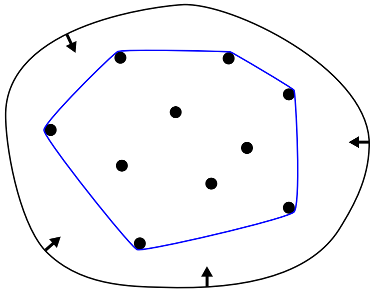
Approximate the points by a convex polygon, called the "convex hull"

Only boundary points affect its shape

Starting point for computing many other properties of the data

- Area
- Diameter (maximum distance between two points)
- Radius of smallest enclosing circle
- Classification in machine learning

What is the convex hull?



https://commons.wikimedia.org/wiki/File:ConvexHull.svg

Input: n points

Output (intuitive): polygon formed by stretching a rubber band around the points

Simplifying assumptions:

- No three in line
- No two with same x

More formal definitions

- Min perimeter polygon surrounding all points
- Min area convex polygon surrounding all points
- Intersection of all halfplanes that contain all points
- Union of all points, line segments through two points, and triangles formed by three points
- Set of all convex combinations (weighted averages)

$$c_0 p_0 + c_1 p_1 + c_2 p_2 + \cdots$$

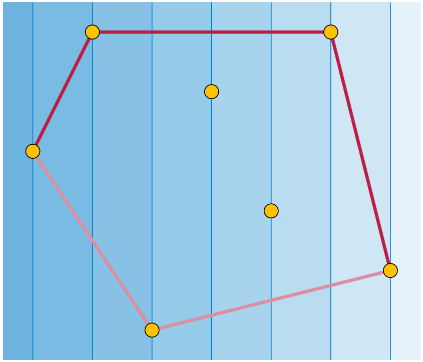
  for points p_i, positive coefficients c_i summing to one

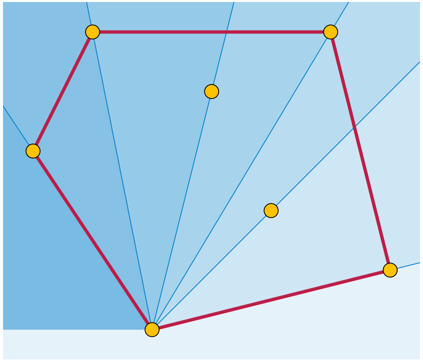
All equivalent to each other!

Graham scan

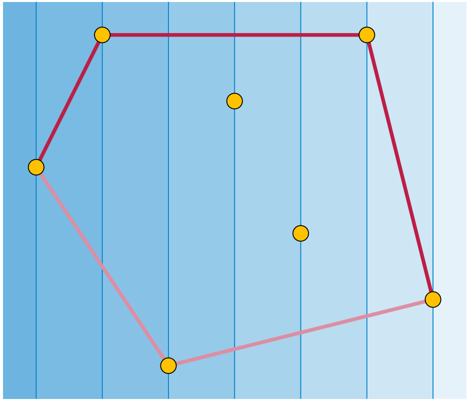
A simple, fast algorithm using only sorting and a stack. Two different versions:

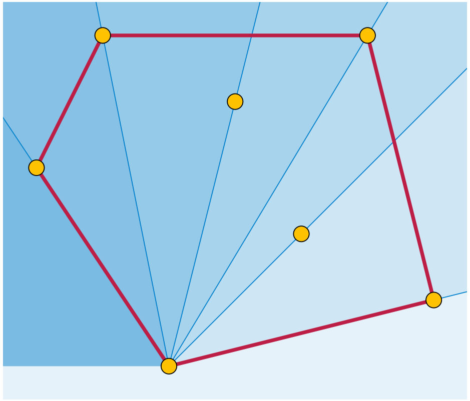
- Our book: sort by x-coordinates (easier)
  Find "upper hull" and "lower hull" separately
- Some sources: sort radially around bottom point; find whole hull in a single pass





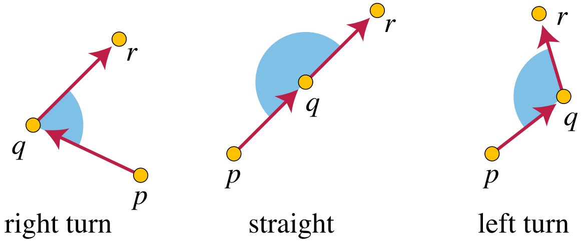
Upper and lower hulls





Split into two paths at leftmost and rightmost vertex

Left and right turns



If you travel in a consistent direction along the hull, you always turn the same way

- Left-to-right on upper hull: always turn right
- Left-to-right on lower hull: always turn left

Idea for algorithm: check if this is true and fix wrong-direction turns when we find them

To find upper hull of point set P

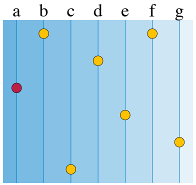
- Sort P by x-coordinates
- Create an empty stack S
  (Will contain upper hull of points seen so far)
- For each point p_i in sorted order:

```
While $S$ contains $\geq 2$ points
and second-last – last – $p_i$ is a left turn: $\Leftarrow$ Computational details next time!
        pop $S$
push $p_i$ onto $S$
```

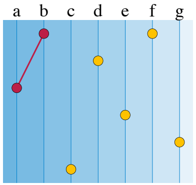
- Return S

Lower hull: Change “left” to “right”, or reverse the sorted order

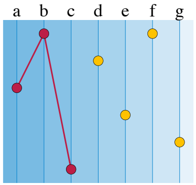
Example of Graham scan



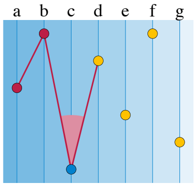
push a; stack = [a]



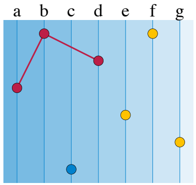
push b; stack = [a,b]



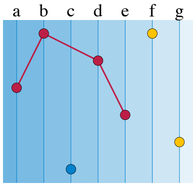
push c; stack = [a,b,c]



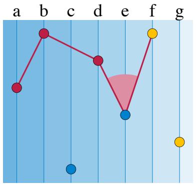
bcd=left; pop; stack = [a,b]



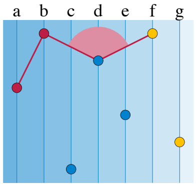
push d; stack = [a,b,d]



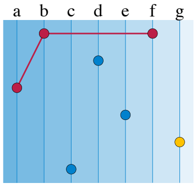
push e; stack = [a,b,d,e]



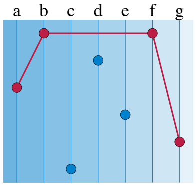
def=left; pop; stack = [a,b,d]



bdf=left; pop; stack = [a,b]



push f; stack = [a,b,f]



push g; stack = [a,b,f,g]

Analysis of Graham scan

Outer loop runs once per point, just does simple stack operations

Each time through the inner loop, we pop a point

Each point is only pushed once, so it can only be popped once

$\Rightarrow O(n)$ total times through inner loop

Total $=$ sorting $+ O(n)$

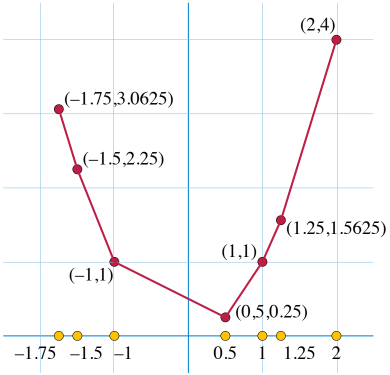
A lower bound on convex hulls

Given input list L of n numbers, $x_0, \ldots x_{n-1}$

Transform into n points (x_i, x_i^2)

Lower hull consists of all points in sorted order

So fast convex hull algorithm $\Rightarrow$ can sort equally fast



But this does not prove that convex hulls require $\Omega(n \log n)$ time

The $\Omega(n \log n)$ sorting lower bound is for a limited model of computing where we can only compare input numbers, but this model is unable to compute hulls!

Refined ("ultimate") hull algorithm

When hull has h vertices, time is $O(n \log h)$, fast for small h

First version: assume we already know the value of h

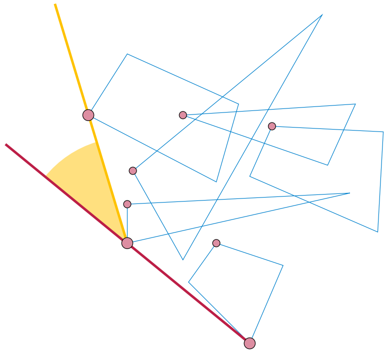
1. Divide points arbitrarily into $\lceil n/h \rceil$ subsets of size $\leq h$
2. Use Graham scan to find hull in each subset
3. Time so far: $O(n/h) \times O(h \log h) = O(n \log h)$

Next. . .

Find hull one edge at a time, starting @ bottom

"Gift wrapping": to find each edge, perform a separate binary search in each of many small overlapping convex polygons

h edges, $O(n/h) \times O(\log h)$ per edge = total $O(n \log h)$



Ultimate hulls when h is unknown

Try $h = 4, 16, 256, \ldots, 2^{2^i}$
until one of them works (we find a hull with $\leq h$ edges)

For each guess at h, stop early if we see too many edges

The final guess is $< h^2$ for the actual value of h

Total time for all tries: $O(n \log 4) + O(n \log 16) + \cdots + O(n \log h^2)$
It's a geometric series! Simplifies to $O(n \log h^2) = O(n \log h)$

Numerical precision

Primitives used for decisions must be exact!

If the proof of an algorithm's correctness depends on it making correct choices, then all the choices it makes must be correct

Primitives used for if-then-else tests, loop termination, etc., must always produce the intended results

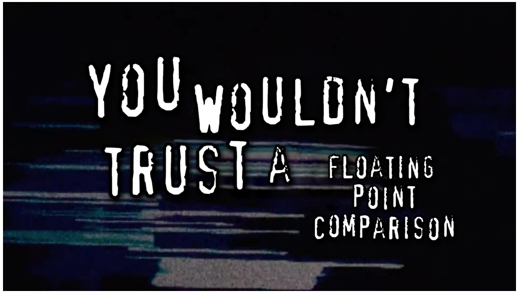
Otherwise, programs using them are likely to produce wrong results or, worse, crash!

https://commons.wikimedia.org/wiki/File:Kobaysahi_crash_Canadian_GP_2010_(cropped).jpg

What kind of numbers should we use?

Coordinates = numbers

The obvious choice is floating point, but that is inexact $\Rightarrow$ crashes



Instead, geometric algorithms generally need to represent coordinates as integers, and compute primitives to enough precision to guarantee exact results

Area / turn direction / crossing primitives use products of two coordinates and should be computed to twice as many bits of precision as the input coordinates