Convex hulls

Marc Glisse*

Feel free to use any reasonably common programming language, but whatever the language, make an effort so the code is readable even by someone who does not know this particular language. For fairness, since not all languages provide the same facilities, I placed a few restrictions on what functions you need to reimplement even if your language provides them, but other than that you are welcome to use the datastructures available, like an ArrayList in Java, a dictionary in Python, etc.

1 Introduction

The topic of this project is the computation of a polygon defined as the convex hull of a finite set of points in the plane. The input is the Cartesian coordinates of the points, and the output is the list of the vertices of the polygon in counterclockwise order, where vertices are represented either by their index in the input list, or by the coordinates of the corresponding point.

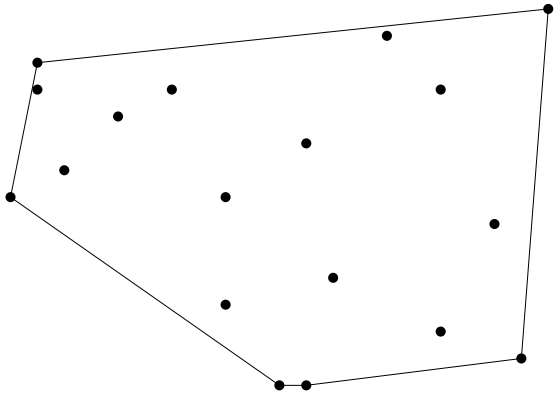


Figure 1: Convex hull of a point set.

We will implement 2 standard algorithms, and test them on different datasets.

Since this is 2D geometry, it is strongly recommended to plot the point set and the convex hull, to check your result visually. For this visualisation task, you may use any library or piece of code you can (legally) get your hands on, including software written by others in a different language.

*marc.glisse@inria.fr

2 Datasets

Before computing convex hulls, let us write functions that generate four (families of) datasets. You may use a function provided by your language to generate a random number between 0 and 1 (or several at the same time), but not an advanced one that already generates points in a disk or on a circle.

Dataset B is obtained by generating n random points uniformly and independently in the square $[0, 1]^2$. That is, for each point, its x and y coordinates are each given by the random generator.

Dataset A is obtained by generating $n-4$ points as in B, adding the 4 corners of the square, and rotating the whole point set by a random angle (the center of rotation does not matter). And randomly shuffle (permute) the n points so the extremes are not always at the same index.

Dataset C is obtained by generating n random points uniformly and independently in a disk. For this, we consider a square that contains the disk, generate random points in the square, keep only those that happen to fall in the disk, until we have the desired number of points.

Dataset D is obtained by generating n random points uniformly and independently on a circle. (Think of polar coordinates for an easy way to do that.)

Note that geometric computation is notoriously sensitive to numeric errors. These datasets should not be too bad, but there is always a tiny chance that a numerical imprecision may let your program think that a point is slightly above a line while it is slightly below, and that may cause strange errors. For this project, you may assume that 2 points cannot have the same x-coordinate, that 3 points cannot be aligned, etc.

For debugging, it may be convenient to try datasets with few points (less than 10), but afterwards you should certainly test your code with thousands of points, if not millions.

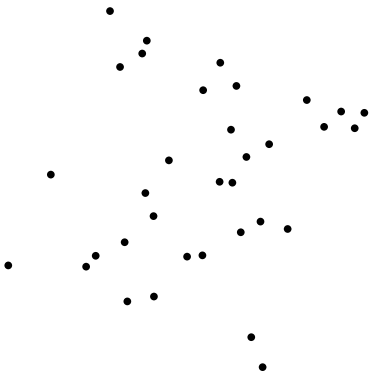
3 Sweeping

As a preprocessing step, this algorithm requires that we first sort the points according to their x-coordinate. Please use a sorting function provided by your language for that.

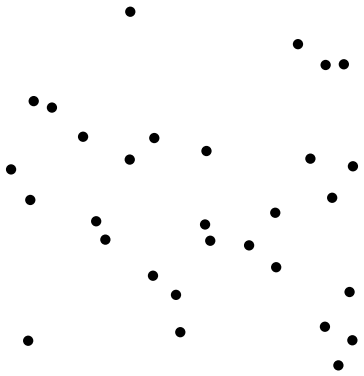
Implement a function that, given 3 points A, B, C, tells if the triangle ABC is oriented clockwise or counter-clockwise. If $A_x < B_x$, clockwise means that C is below the line passing through A and B, while if $A_x > B_x$, clockwise means that C is above the line passing through A and B. (possible hints: area, cross product, determinant)

We describe here how to compute the upper part of the hull, the computation of the lower part is symmetric. The file sweep.pdf, best viewed in presentation mode, shows a step-by-step execution of the algorithm.

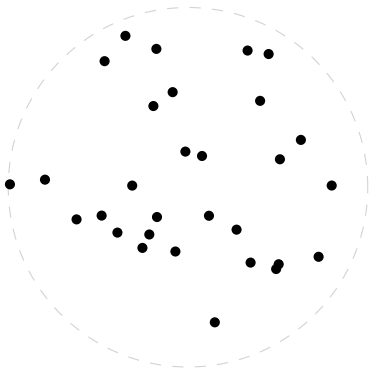
We add the points one by one, sorted by their x coordinate, and maintain the chain of vertices on the upper part of the convex hull of the points we have already seen. The sweeping algorithm starts from the leftmost point, which is obviously on the convex hull. We connect the second point to the first one (the convex hull of 2 points is the segment). Things really start with the third point. When we consider a new point p, we look at the edge it forms with the last point q of the chain. If this new edge "turns right" compared to the last edge of the



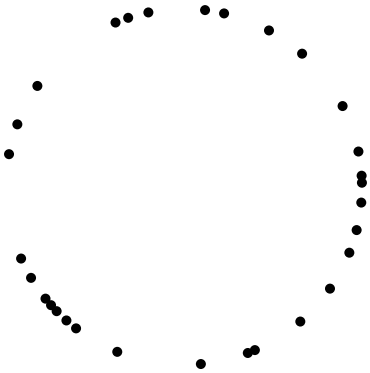
(a) Dataset A.



(b) Dataset B.



(c) Dataset C.



(d) Dataset D.

chain (the angle *below* is less than 180°), then the chain is still (upper-)convex, and we can move on to the next point. If on the other hand this new edge "turns left" (the angle *above* is less than 180°), appending p to the chain would create a concavity, we need to fix things. This left turn actually proves that the point q is not on the upper hull. We thus remove it from the chain, and consider p anew, which defines an edge with the new last vertex q' of the chain. We continue until we find a right turn, or until we reach the first vertex.

The procedure ends when we have handled the rightmost vertex. We can then similarly compute the lower hull and join them to output the convex hull.

Implement this algorithm, and test it on the datasets you generated earlier. Measure the time it takes for different values of n, and check if it roughly matches the theoretical complexity you expect (see below).

3.1 Complexity

- What is the (time) complexity of sorting at the beginning?
- What is the maximum time that adding one point can take?
- By considering how many times a point can be removed, give a bound on the complexity of this algorithm.

4 Output-sensitive algorithm

The complexity of the sweeping algorithm is optimal, it is impossible to do better in the general case. However, this complexity does not depend much on the configuration, even when very few of the points are on the convex hull. If the convex hull is a triangle, we should be able to notice that in linear time…

The gift-wrapping algorithm has a complexity $\Theta(nh)$ where n is the number of input points and h is the (unknown) number of vertices of the convex hull. It starts from the leftmost point, and finds the next point on the convex hull in linear time. This is nice if h is very small, but bad for large h. We will not implement it here. Instead, we will implement an algorithm that has an optimal complexity in terms of both h and n. We first need a few tools though. For both, the exact interface should be whatever works best with how you use them in the final algorithm, they are only presented separately to help you understand the algorithm and structure your code.

4.1 Median

Implement a function that takes a list of points ("list" is used here as a generic term, the datastructure is up to you) and returns a value x_m such that half of the points have an x coordinate strictly smaller than x_m, and the others strictly larger (the rounding of *half* when n is odd can be arbitrary).

This function needs to have linear complexity, and you are *not* allowed to use a function that directly gives the answer if your language provides one.

4.2 Linear programming

Given a set of points P with x coordinates different from x_m, we want to find a (non-vertical) line that intersects the vertical line of equation $x = x_m$ as low as

possible (smallest y), while passing above all the points of P. We further assume that there is at least one point with $x < x_m$ and one with $x > x_m$, so the minimal y is not $-\infty$.

The solution is always a line passing through 2 points of P, one with $x < x_m$ and one with $x > x_m$, and we call this pair of points the *basis* for the solution.

In this algorithm, we first pick an arbitrary point p_1 of P with $x < x_m$ and one p_2 with $x > x_m$. If we consider the problem with the set $P_2 = \{p_1, p_2\}$ instead of the whole P, (p_1, p_2) forms the basis of the solution. We add the other points of P one by one in *random* order (this is important) and maintain the basis of the solution for the subset P_k of k points already added. Call p_k the k-th point that we added. When we add p_{k+1} to P_k, if it is below the line defined by the basis of P_k, then the basis remains the same for P_{k+1}. If it is not, then we know that p_{k+1} is one of the two points of the new basis, and we look for the other one in P_k with complexity $O(k)$.

The file lp.pdf shows a step-by-step execution.

The expected complexity of this routine is linear. Checking if p_k is below the optimal line of P_{k-1} takes constant time, and we only change the basis (and do the expensive $O(k)$ search) with probability $\leq \frac{2}{k-2}$. Indeed, the basis of P_k does not depend on the order in which $\{p_1, \ldots, p_k\}$ were added, and the probability that the k-th insertion is expensive is the probability that, in the random permutation, the k-th point happens to be one of the 2 defining the basis.

4.3 Main algorithm

We describe here how to compute the upper hull. First, we select a value x_m such that half of the points are on the left and half on the right. Then, we find one left point l and one right point r such that the line passing through them intersects the line $x = x_m$ as low as possible while being above all the points of P. These 2 points define an edge of the convex hull. Now we split the points left of l into a first sub-problem, the points right of r into a second subproblem, and we drop the other points because they cannot appear on the upper hull. We apply the same algorithm recursively on each subproblem until we have constructed the entire convex hull.

The complexity of this variant of divide-and-conquer, sometimes called marriage-before-conquest, can be bounded not only by $O(n \log n)$, but even $O(n \log h)$ (where h again denotes the size of the output), because there are at most h divide steps in the execution (optionally, you can try to prove this $O(n \log h)$ bound in your report, but that's not the main goal here).

Implement this algorithm and test it on the datasets, like the first algorithm.

5 Comparison

Compare the result of your two algorithms on the various datasets. Do they give the same output? How does the performance compare? Does it depend on the type of dataset?

6 About the report

You are supposed to submit source files plus a report.

- Do not include code in the report. Especially do not include screenshots of code or photos of your screen... It is fine to add some pseudo-code to explain an algorithm though. Images of the graphical result of your program are also appreciated.
- How you tested your code is something worth explaining in the report.