Computatinal Geometry: Theory and Experimentation (2023)
Project 3: Convex Hull Computation

Peyman Afshani

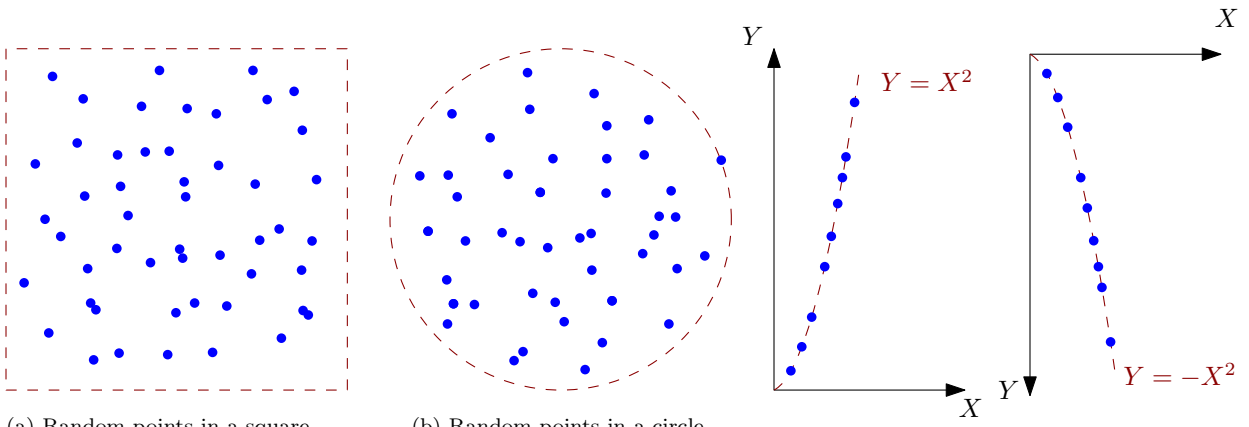
October 4, 2023

If you are **not** a PhD student, then the tasks at Section 3 are optional (without any extra points).

1 Introduction

In this project, you will implement a few convex hull algorithms and you will compare their performance. After implementing each algorithm and making sure that it runs correctly, you must evaluate its performance on some "test cases". You should try to follow sound algorithm engineering principles in doing so (have a plan, design your experiments with respect to your plans, and hypothesis, evaluate the results and potentially go back to repeat the process). For this project, and to reduce your workload, you can run all your experiments on the same machine. You can also choose whatever programming language you are comfortable with but you should be mindful of possible issues created by programming languages that have garbage collectors and so on.

Test cases. However, when running the tests, you should pay attention to create a diverse set of "input classes". The first class should be input points that are generated uniformly randomly **inside** a square, the second point sets generated uniformly randomly **inside** a circle, and the third point sets whose points lie **on** the curve $Y = X^2$ and the fourth test case is a set of points on the curve $Y = -X^2$. See Figure 1.



(a) Random points in a square

(b) Random points in a circle

(c) Points on the curves $Y = X^2$ and $Y = -X^2$

Figure 1: The test cases.

You are encouraged to pick additional test classes to improve the quality of your report.

Important remarks. Think carefully about the experiments that you are going to run. **For a good set of experiments, you should ideally spend more time doing experiments than coding!** There are a lot of things to discover here which will not be obvious at the beginning and without doing experiments. Also, think well about "performance metrics" that you wish to measure. The prominent one would be the running time, however, depending on your experiment or your hypothesis, you can measure the running time differently. For example, in the Graham's scan algorithm, you can separate the sorting time from the hull computational time. You are free to do similar things for the other algorithms. You are not in fact required to measure the actual running time and you are free to do all the experiments using software counters.

Your report. In addition to your code, you should also hand in a report. There is not a fixed formula on how to write a report and its details can vary from group to group and depending on your observations. However, here are some general tips.

For each algorithm, the report should contain a brief description of the algorithm, and its theoretical running time. If the algorithm has prominent parts, you can mention the theoretical running time of those. You can also mention any other interesting theoretical facts about the algorithms.

The report should also contain a brief description of the different input classes and your hypothesis on how they can impact the running time. An important metric here is the number of points on the convex hull which you can try to plot. Can you figure out how this number changes as a function of the input size? You can discuss your thoughts on how this can impact the running time. Try to tie your understanding of the output size to the theoretical analysis. To do this, it is important to understand the size of the convex hull for each different input class.

Then, you can move on to discuss your implementation and experiments. Most algorithms are described for finding the upper hull only. It is obviously possible to change them to compute the full convex hull but if you want to save time, you can focus only on the upper hull computation.

If you want, you can mention any interesting difficulties or problems you encountered in your implementation. The starting point of the experiments is trying to figure out if the algorithms behave as they theoretically should. There are a few different ways to look at this. One way is to look at each algorithm individually with respect to different values of n and different input classes and see if the results of the experiment comply with theory. Then, one can look at each input class and see how different algorithms behave on that input class. If things don't make sense or if you see a strange behavior, you can run follow up experiments to come up with an explanation. You don't have to explain everything and it is fine if a few things stay as mystery. Include any interesting finding that comes up during your experiments for a fuller and more satisfying report.

Algorithmic horse-race. Be mindful of the "algorithmic horse-race" trap. An algorithm like "Graham's scan" is very simple and its most prominent step (the sorting) is often very heavily optimized in most programming languages. At the opposite corner, it is almost certain that your implementation of "Marriage before Conquest Algorithm" will be nowhere near as optimized. These algorithms are much more involved and thus they even have greater opportunities for optimization and very likely you will not have the time to make them as optimized as "Graham's scan". Think about the conclusions you are about to make about algorithms with very different levels of optimizations.

2 The Convex Hull Algorithms

Part A(Graham's Scan). Implement and test the incremental convex hull algorithm (Graham's Scan) discussed in the class (the one discussed in pages 6 and 7 of BKOS). Let's call this algorithm INC_CH.

Part B(Gift Wrapping). Implement and test the gift-wrapping convex hull algorithm discussed in the class. Let's call this algorithm GIFT_CH. Remember that this computes the full convex hull. If you are only focusing on the upper hull, you need to change it slightly to only compute the upper hull.

- **GiftWrapping**(P): // Finds the convex hull of point set P
- To initialize, find the point q_1 with the smallest x-coordinate. Initialize an upwards ray $\overrightarrow{r}$ from q_1 (i.e., a vertical line segment with one endpoint being q_1 and the other endpoint at $Y = +\infty$). Set the pivot p to be q_1.

1. `Do` the following:
   (a) `Iterate` over all the points in P and find the point v that minimizes the angle between $\overrightarrow{r}$ and $\overrightarrow{pv}$.
   (b) Add v to the convex hull
   (c) Set $\overrightarrow{r} \leftarrow \overrightarrow{pv}$
   (d) Set $p \leftarrow v$
2. `Until` p equals q_1.

Part C(Marriage before Conquest). Implement the marriage-before-conquest convex hull algorithm. Note that this computes the upper hull. We call this MbC_CH.

1. Pick a random point $p_m = (x_m, y_m)$. Partition the input into two sets P_ℓ and P_r where P_ℓ contains all the points with x-coordinate smaller than x_m and P_r contains the rest of the points.
2. Find the "bridge" over the vertical line $X = x_m$ (i.e., the upper hull edge that intersects line $X = x_m$). You need to implement linear programming for this step. Let (x_i, y_i) and (x_j, y_j) be the left and right end points of the bridge.
3. Prune the points that lie under the line segment $(x_i, y_i), (x_j, y_j)$ (these will be the points whose x-coordinates lie between x_i and x_j.
4. Recursively compute the upper hull of P_ℓ and P_r.

Finding the Bridge. To find the bridge, you must implement the method based on linear programming. You can adapt your code that solves the linear program to the specific case of the mbC algorithm. In other words, you do not have to code a solution that can solve any generic 2D linear program, meaning, your code could be tailored to the specific linear programs that we get from the MbC algorithm. However, you need to pay attention for a few important details. First, pay attention to some of the points discussed in the exercise classes. These can help you overcome some of the floating point issues that otherwise can be very annoying. In addition, you can implement the linear program directly in the "primal space", meaning, by considering the points and using sidedness tests. Finally, pay attention that floating point inaccuracies do not cause *logical* errors in the computation that can result in completely wrong upper hulls, infinite loops or any other very undesirable situations.

Remarks about special cases. You algorithms are not required to handle inputs containing special cases such as three points on the line, two points having the exact same X or Y coordinates and so on. Your implementations could be based on the assumption that none of these cases exist in the solution.

Optimizations. In the MbC algorithm, there are a lot of possibilities for optimization (some obvious, some very non-obvious) that can significantly speed up the running time. It is not required that you spend a lot of time doing this. However, you must make sure that it is the correct implementation and that its asymptotic running time is correct. I have received some solutions in the past where the implementation had $\Omega(n^2)$ running time (usually caused by doing some vector operation or library operation n times where each operation could take $\Omega(n)$ time). If your experiments show a quadratic behavior, most often it is an indication that somewhere something has gone wrong and the implementation is not of the correct asymptotic behavior.

Part D(MbC Analysis). Prove that MbC_CH algorithm runs in $O(n \log h)$ time where h is the number of points on the convex hull (remark: you **cannot** assume that the recursion tree is balanced, in other words, you cannot assume that when we do the recursion, half of the upper hull edges are to the left and half to the right).

3 Mandatory Tasks for PhD Students

Part E$^+$ (MbC Entropy Analysis). Let $x_1 < \cdots < x_h$ be the x-coordinates of the points on the upper hull and let n_i be the number of input points $p = (x, y)$ such that $x_i \leq x < x_{i+1}$, $1 \leq i < h$. Show that the upper hull computation in MbC_CH runs in time

$$O\left(\sum_{i=1}^{h-1} n_i \log\left(\frac{n}{n_i}\right)\right).$$

We now introduce a variant of the MbC algorithm which we call MbC2_CH. We basically add one more pruning step to the MbC algorithm. This is shown below.

MbC2_CH:

1. Find the point with median x coordinate $p_m = (x_m, y)$ and partition the input into two sets P_ℓ and P_r where P_ℓ contains all the points with x-coordinate smaller than x_m and P_r contains the rest of the points.
2. Find the point p_ℓ with the smallest x-coordinate (if there are more than one, take the one with the largest y-coordinate) and the point p_r with the largest x-coordinate (if there are more than one, take the one with the smallest y-coordinate). Prune all the points that lie under the line segment $p_\ell p_r$.
3. Find the "bridge" over the vertical line $X = x_m$ (i.e., the upper hull edge that intersects line $X = x_m$). Let (x_i, y_i) and (x_j, y_j) be the left and right end points of the bridge.
4. Prune the points that lie under the line segment $(x_i, y_i), (x_j, y_j)$ (these will be the points whose x-coordinate lie between x_i and x_j.
5. Recursively compute the upper hull of P_ℓ and P_r.

Part F$^+$ (MbC Fine Analysis). Consider a partition, Δ, of the point set P into k disjoint subsets $P_1, \cdots, P_k$ and let $n_i = |P_i|$, $1 \leq i \leq k$. We say that Δ is "good" if for every i, $1 \leq i \leq k$, either $n_i = 1$ or P_i can be placed inside a triangle t_i such that t_i completely lies below the upper hull of P. Define the "entropy" of Δ as

$$H_\Delta = \sum_{i=1}^{k} \frac{n_i \log(\frac{n}{n_i})}{n}.$$

Prove that for any good partition Δ, the upper hull computation in MbC2_CH runs in $O(nH_\Delta)$ time. Prove that this is not the case for MbC_CH algorithm, that is, there exists a point set P and a good partition Δ such that nH_Δ is asymptotically smaller than the time it takes for MbC_CH algorithm to compute the upper hull of P.