Computatinal Geometry: Theory and Experimentation (2025)
Project 2: Convex Hull Computation

Peyman Afshani

October 2, 2025

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 $Y = -X^2$. See the figure below. You are encouraged to pick additional test classes to improve the quality of your report.

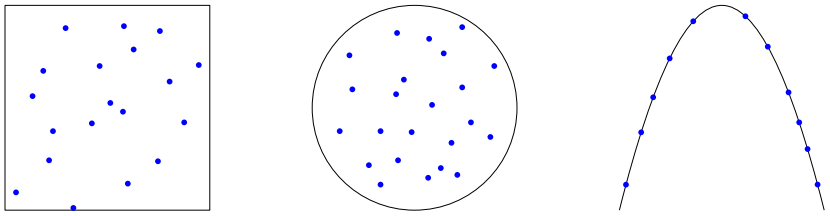

Figure 1: The three required test cases. You are encouraged to add more test cases if you want to.

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!** 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, you can measure the following times separately:

1. The time it takes to read the input
2. The time needed for computation
3. The time for smaller computational tasks (e.g., separate the sorting time in Graham's scan).

You are also encouraged to use software counters to obtain more robust results.

Your report. You need to focus on the following in your report:

- 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.

  If you have changed some aspects of the algorithms that you have implemented, also mention those.

- 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.

  Plot the number of points on the convex hull (or upper hull) for the square and the circle case (or any other class of inputs where it is not immediately clear what is the number of points on the hull). Decide whehter to use logarithmic scale on the X and Y axis for this to make sense. These plot should enable you to come up with a hypothesis on what is the expected number of points on the convex hull, for the first two input types (inside a square and inside a circle).

  Also mention how this impact the running time of the algorithms studied here.

- 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.

- Make sure that your algorithms are tested for large enough input sizes. **Test cases of just a few thousand points are too small to get any meaningful conclusion from them.** You don't need to run all the algorithms on all the test sizes (some algorithms might end up too slow) but make sure you are testing large inputs.

In addition to the above points, you can also discuss your implementation and experiments. The algorithms are described for finding the upper hull only and it is fine if you only find the upper hull in your project. It is obviously possible to change them to compute the full convex hull.

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. For example, if you choose to use Python, it is likely that the sorting step of Graham's scan will be faster than the scan part.

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(QuickHull). Implement the `QUICKHULL` algorithm, described by the following pseudocode. Let's call this algorithm QH_CH. This algorithm builds the upper hull.

· To initialize, find the point q_1 with the smallest x-coordinate and the point q_2 with the largest x-coordinate, and form the line segment s by connecting them. Then prune all the points below s.

· **QuickHull**$(\overline{q_1q_2}, P)$: // Finds the upper hull of point set P above segment $s = \overline{q_1q_2}$

1. Find the point q above s that has the largest distance to s.

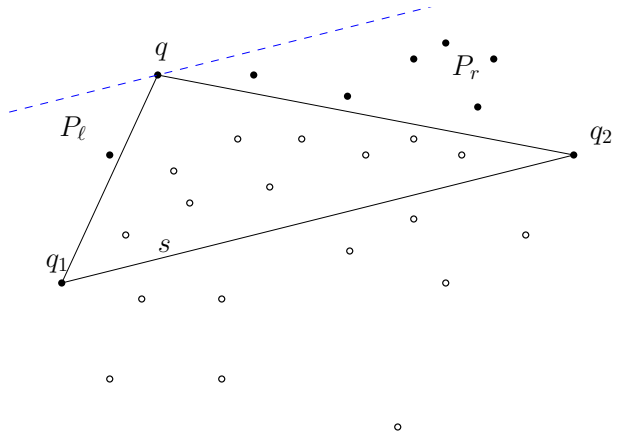

Figure 2: QuickHull finds the upper hull between q_1 and q_2. It is assumed that all the points below the segment $s = \overline{q_1 q_2}$ have been pruned. We find the point furthest away from the line segment s, then we prune all the points inside the triangle $q_1 q_2 s$ and then we recurse on the point sets P_ℓ and P_r that are to the left and right of q respectively.

2. Add q to the upper hull
3. Prune all the points inside the triangle formed by q, q_1 and q_2, and partition the remaining points into two subsets P_ℓ and P_r consisting of points that lie to the left of q and points that lie to the right of q.
4. Connect q_1 to q to form segment s_1 and then connect q_2 to q to form segment s_2.
5. Recurse on **QuickHull**(s_1, P_ℓ) and **QuickHull**(s_2, P_r)

In your report, explain what is the worst-case running time of this algorithm.

Part C(Marriage before Conquest). Implement the marriage-before-conquest convex hull algorithm. Follow the pseudocode given below for the upper hull construction (use a similar code for the lower hull). Let's call this algorithm MbC_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 "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.

Next, add one more pruning step to the above algorithm and call it MbC2_CH. This extra pruning step is the step 2 in the algorithm below.

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). Note that these can be done at the same time as step 1. 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.

Remarks on the MbC algorithm. Instead of finding the median of a set S exactly, you can simply pick a random point and use its X-coordinate. Also, you need to pay attention to a few details: first, observe that if the random point is the leftmost or the rightmost point, then you might have a strange situation in your LP. In in general, it is best if x_m, the parameter that is used in your LP is not the X-coordinate of an actual point in your input and it lies between two input points. Finally, it is encouraged that you look at the exercises for the LP lecture as it has a lot of hints on how to implement the MbC algorithm.

Important. You must include code snippets from your LP solution and discuss how you implemented the LP step. Using libraries such as simplex is **not** acceptable since in our case, simplex corresponds to essentially gift wrapping and thus it would be very inefficient.