INF421 PI - CONVEX HULLS

FARAJI Ossama - ZIAT Oussama

February 1, 2023

1 Introduction

The goal of this project is to determine the shape of the smallest convex polygon that encloses a given set of points in a two-dimensional plane. The input for this project is a list of the Cartesian coordinates of the points, and the output will be a list of the vertices of the polygon in a counterclockwise order. The vertices can be represented either by their position in the original list of points or by the coordinates of the point they correspond to.

There are many real-world applications for the computation of convex hulls, some examples include:

- Computer Graphics: Convex hulls can be used to quickly determine the silhouette of an object, which is useful for hidden surface removal and solid modeling.
- Image Processing: Convex hulls can be used to extract the shape of an object in an image, which can be used for object recognition and tracking.
- Robotics: Convex hulls can be used to determine the reachable workspace of a robotic arm, which can be used for motion planning and collision detection.
- Geographic Information Systems (GIS): Convex hulls can be used to determine the smallest convex polygon that encloses a set of geographical points, which can be used to represent regions or clusters of data.

And many other applications in Machine Learning and Game Development.

We provide the code along with several comments and implementation notes in the Jupiter Notebook along side this report. As we use Python 3, one could expect low performances, but Python's syntax is simple and easy to read, write, understand and debug the code, thanks to the large choice of libraries such as Numpy and Matplotlib.

2 Datasets

The main idea is to generate n random couples $(x_i, y_i)_{i\leq n}$ uniformly and independently in the square $[0,1]^2$, and undergo futher calculations if needed (shuffling, converting to polar coordinates...). Figure 1 below is an example for $n = 100$.

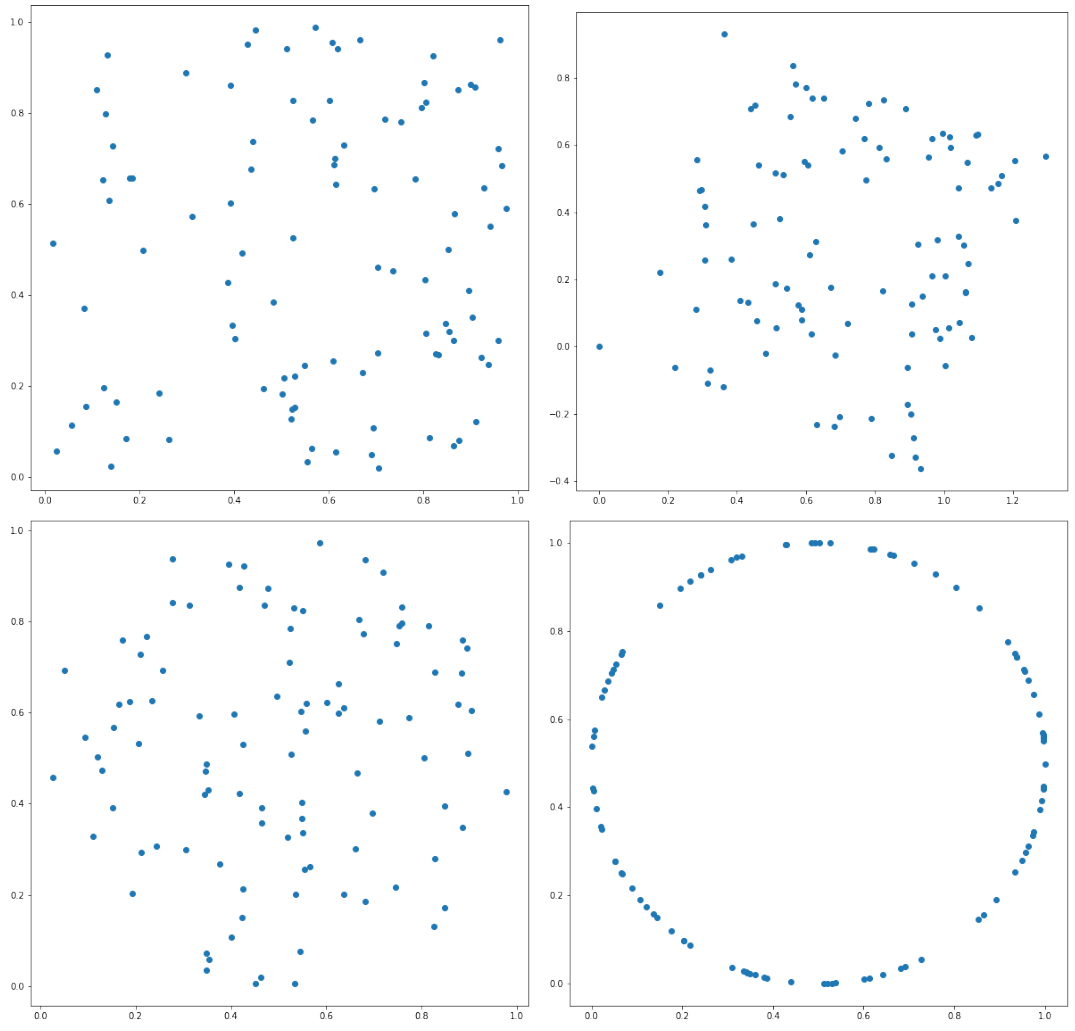


Figure 1: Datasets B, A, C and D in order

3 Sweeping

3.1 Preprocessing - Triangle Orientation

As a preprocessing step, this algorithm requires that we first sort the points according to their x-coordinate. For this task, we use the Python default sorting algorithm Timsort, whish is a hybrid sorting algorithm, derived from merge sort and insertion sort, It was designed to perform well on many kinds of real-world data and an average time complexity of $O(nlogn)$ which makes it suitable for sorting large datasets, it also has a good worst-case performance of $O(nlogn)$ and a best-case performance of $O(n)$ which makes it efficient for small datasets as well.

As for the helper function for determining the orientation of 3 points, we use the cross product of vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$. The function first checks if the x-coordinate of point A is less than the x-coordinate of point B or not. If it is, it calculates the cross product of the vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$ and compare the result with zero. If the cross product is less than zero, it means that the triangle is oriented clockwise, otherwise it is oriented counter-clockwise. If x-coordinate of point A is greater than x-coordinate of point B, the function checks if the cross product is greater than zero. If it is then the triangle is oriented clockwise, otherwise it's oriented counter-clockwise.

3.2 The algorithm

The implementation builds the upper and lower hulls separately by iterating through the sorted points and maintaining the chain of vertices on the upper and lower parts of the convex hull of the points that have already been seen.

Each time a new point is added, the function checks whether the new edge formed by this point and the last point on the chain "turns right" or "turns left" compared to the last edge of the chain. If it turns right, the chain is still (upper- or lower-) convex, and we can move on to the next point. If it turns left, this means that the last point on the chain is not on the convex hull, and it is removed from the chain.

The function returns the concatenation of the upper and lower hulls, with the last point of each hull removed. Figure 2 below shows the output of the algorithm for the 4 datasets, where we chose $n = 20$ for the sake of clarity.

3.3 Complexity

We measure the time it takes for values of n in range 100000 for the algorithm to fully excecute. Figure 3 below shows that the running complexity is linear for smaller values of n, and the curve shift toward the Y-axis for larger values, suggesting a overall time complexity of $O(nlogn)$.

3.3.1 What is the (time) complexity of sorting at the beginning?

The time complexity of this convex hull algorithm implementation is $O(nlogn)$ where n is the number of input points.

This is because the algorithm starts by sorting the input points by x-coordinate, which has a time complexity of $O(nlogn)$ using the python's built-in sorting function. Then the algorithm iterates over the sorted points twice, once to build the upper hull and once to build the lower hull. At each iteration, it checks the orientation of the last 3 points in the hull, which takes $O(1)$ time, and possibly remove the last point of the hull which takes $O(n)$ time in worst case. It's worth mentioning that the space complexity of the algorithm is $O(n)$ as we are storing all points in the hull.

3.3.2 What is the maximum time that adding one point can take?

When adding a point to the convex hull, the algorithm must determine if the new point makes a "left turn" or "right turn" with respect to the current hull. To do this, it must check the relative orientation of the new point and the previous two points in the hull, this operation can be done in $O(1)$ time.

After checking the orientation, the algorithm may need to remove some of the points in the hull, to preserve the convexity of the hull. The maximum number of points that could be removed from the hull in the worst case scenario is O(n).

So in the worst case scenario, adding one point to the hull can take $O(n)$ time.

This worst case scenario typically happens when the point set is close to a degenerate case like a collinear set where all the points are almost in a straight line, or when all the points are in a circular shape, the algorithm would have to backtrack for each new point to ensure convexity.

It's worth noting that this worst case scenario doesn't happen very often, as the expected running time of the algorithm is $O(nlogn)$.

3.3.3 By considering how many times a point can be removed, give a bound on the complexity of this algorithm.

By considering how many times a point can be removed, we can give a bound on the complexity of the algorithm.

In the worst-case scenario, each point in the hull can be removed once, which means that the number of the removed points would be at most n (i.e., total number of input points). So in the worst case scenario, adding one point to the hull can take O(n) time, and as the algorithm processes n points, the worst case time complexity of the algorithm is $O(n^2)$.

However, as I mentioned before, this worst-case scenario is not very likely to happen, as the expected running time of the algorithm is $O(nlogn)$. This is because the points are sorted by x-coordinate which reduces the number of iterations of the loop and the backtracking operations needed. Also, the number of points removed in practice is much smaller than n. In most cases, adding a new point to the hull requires removing very few, if any, of the existingpoints on the hull.

To summarize: The overall time complexity of the algorithm is $O(nlogn)$ on average, However, the worst-case time complexity is $O(n^2)$.

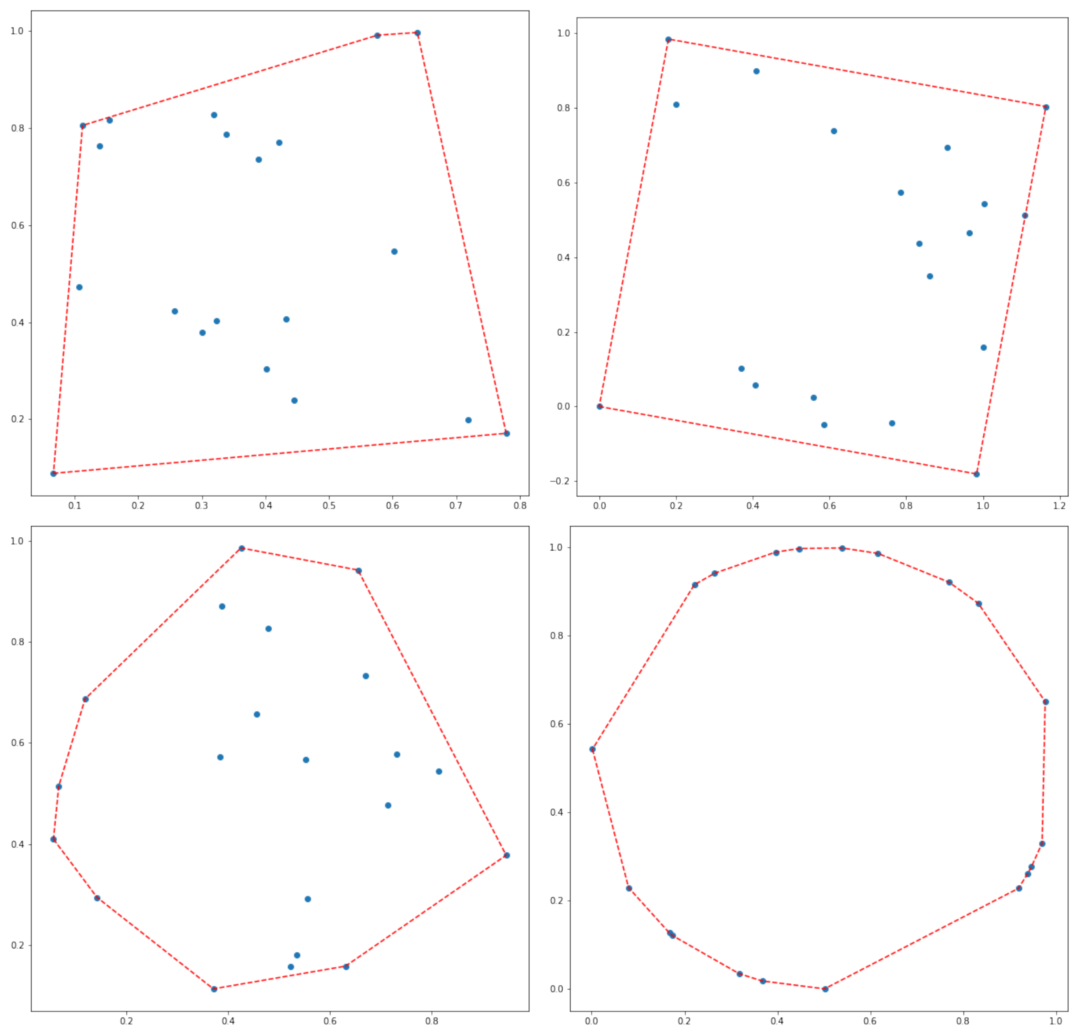


Figure 2: Output of the algorithm for the 4 datasets (n=20)

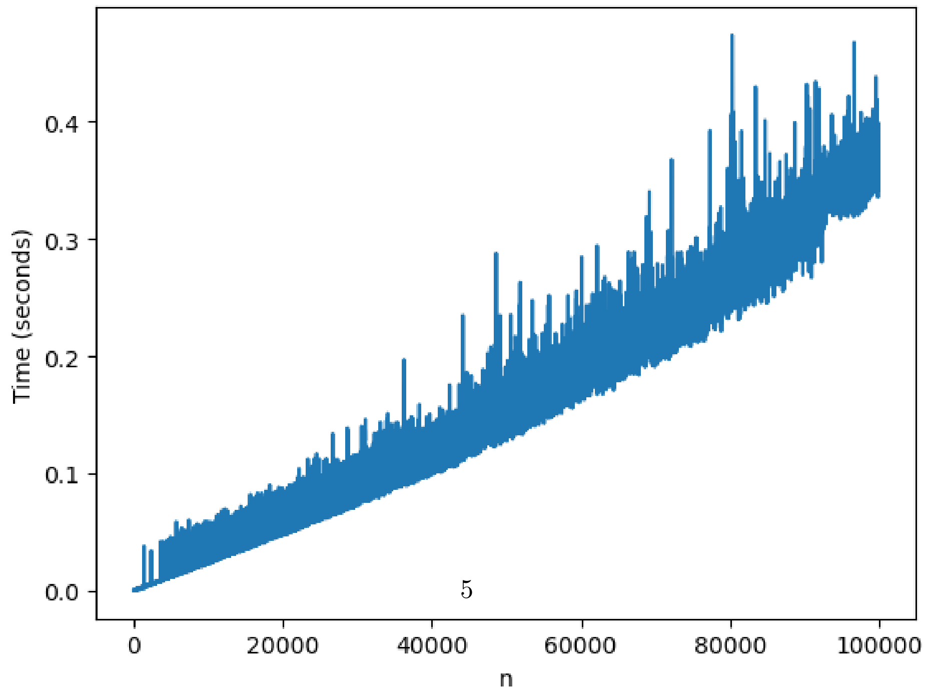


Figure 3: Running time for the sweeping algorithm

4 Output-sensitive algorithm

4.1 Median

We can implement a linear time complexity solution for finding the median of an unsorted list using the "quick select" algorithm, which is an efficient variant of the quicksort algorithm. Here is a rough pseudocode snippet :

- FindKMedian(A, K): Return the number in A which is the K-th in its size.
  - Pick randomly a number a from A = $a_1, ..., a_n$.
  - Partition the n numbers into two sets:
    - S - all the numbers smaller than a
    - S - all the numbers bigger than a
  - If $|S| = K - 1$ then a is the required K-median. Return a
  - If $|S| < K - 1$ then the K-median lies somewhere in B. Call recursively to FindKMedian(B, K - $|S|$ - 1)
  - Else, call recursively to FindKMedian(S, K).

The Python implementation uses the choice function from the random module to randomly select a pivot element from the input array, then partitions the array into three groups: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. It then recursively calls the quick_select function on the appropriate partition based on the value of k, which is the desired index of the median element. The find_median function uses the quick_select function to find the median of an array.

The time complexity of quick select is $O(n)$ on average and worst case $O(n^2)$

4.2 Linear programming

We can observe that this line must pass through at least two points, one with an x-coordinate less than x_m and the other with an x-coordinate greater than x_m. If we were to assume that the line passes above all points in the set P without passing through any point, it would be sufficient to translate the line in the y-direction until it touches one of the points. This would result in the line intersecting with the line $x = x_m$ at a lower ordinate, which contradicts our objective. Therefore, the desired line must pass through at least one point in the dataset P.
Furthermore, to prove that the line passes through another point in P, there are two cases to consider:

- **Case 1**: the line passes through point A with an x-coordinate less than x_m. In this case, we rotate the line in a clockwise direction until it touches another point, with the center of rotation being point A.
- **Case 2:** the line passes through point B with an x-coordinate greater than x_m. In this case, we rotate the line in a counter-clockwise direction until it touches another point, with the center of rotation being point B.

Figure 4 illustrates our approach:

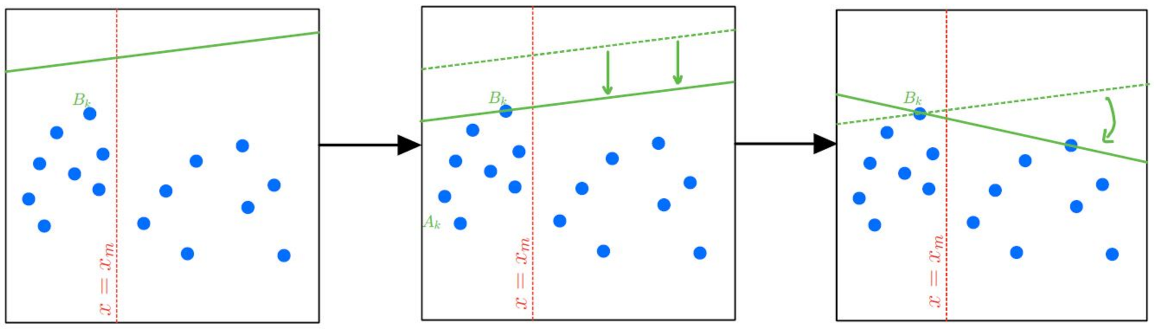


Figure 4: Linear programming approach

In order to find the line of best fit that intersects with $x = x_m$ at the lowest point, we begin by selecting an initial subset of two points, p_1 and p_2, such that the x-coordinate of p_1 is less than x_m and the x-coordinate of p_2 is greater than x_m. This initial subset, referred to as P_2, defines the line of best fit and serves as the basis for our computations. We then proceed by adding new points, p_{k+1}, to the subset P_k and determining the position of the point relative to the line defined by the previous basis. If the point is below the line, it is added to the subset and the old basis is retained. However, if the point is above the line, a new basis must be determined in linear time complexity.

To accomplish this, we take a geometric approach by first creating a line passing through the newly added point and with a slope that is equivalent to the old line defined by the previous basis. We then proceed to consider two cases:

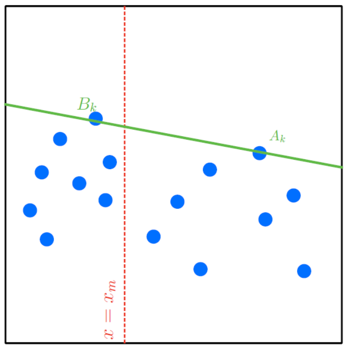
- **Case 1:** The x-coordinate of the newly added point is less than x_m. In this case, we rotate the line in a clockwise direction until it touches another point, B_{k+1}, with the center of rotation being point p_{k+11}. The new basis is then defined as (p_{k+1}, B_{k+1}).
- **Case 2:** The x-coordinate of the newly added point is greater than x_m. In this case, we rotate the line in a counter-clockwise direction until it touches another point, N_{k+1}, with the center of rotation being point p_{k+1}. The new basis is then defined as (p_{k+1}, B_{k+1}).

  Numerically, we can express this process as:

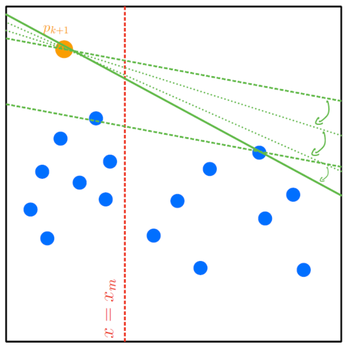
  $$B_{k+1} = argmin\{\|\text{slope}(\text{B}_k, A_k)\text{- } slope(p, p_{k+1})\| \text{ for } p \text{ in } P\}$$

  We can compute the slope of the line defined by two points in constant time complexity, thus we can find the point B_{k+1} with a complexity of $O(k)$

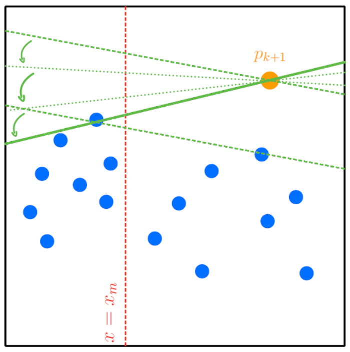
  Figure 5 illustrates our approach:



the subset P_k and his basis $(A_k,\, B_k)$



Case 1



Case 2

Figure 5:

In order to accurately assess the complexity of this algorithm, it is essential to take into consideration the likelihood of changing the basis after adding a new point. Based on the algorithm's design, it can be deduced that the only two points that may be above all the lines defined by the remaining points are those that constitute the basis. As such, the probability of changing the basis is equivalent to the probability that one of these two points is the last to be added, which is less than $\frac{2}{k-1}$.

It should be noted that the basis of the algorithm is independent of the order in which the points were added. This is a result of the uniqueness of the minimum, as the y-coordinate at which the line defined by the basis intersects with the line $x = x_m$ is unique. Furthermore, it can be proven that there cannot be two lines that intersect at the same point with the line $x = x_m$, and are both above all the points in the dataset. This is due to the condition where the basis is formed by a point where the x-coordinate is less than x_m, and the other point has an x-coordinate greater than x_m. It can be observed that if such a line exists and passes through a point of the dataset with $x < x_m$, the other point would be forced to be above the line defined by the basis. The same argument can be applied when this line passes through a point of the dataset where the x-coordinate is greater than x_m.

4.3 The main algorithm:

4.3.1 The idea:

The process of computing the upper hull can be broken down into several steps. Firstly, we select a value xm such that half of the points in the dataset are on the left side of xm and the other half are on the right side. This serves as a starting point for the algorithm.

Next, we find one point l on the left side of xm, and one point r on the right side of x_m, such that the line passing through them intersects the line $x = x_m$ at the lowest possible point while remaining above all the other points in the dataset P. These two points, l and r, define an edge of the convex hull.

We then split the points that are left of l into a first sub-problem, and the points that are right of r into a second sub-problem. The points that fall between l and r are dropped as they cannot be a part of the upper hull.

We then apply the same algorithm recursively on each sub-problem until we have constructed the entire convex hull. This recursive approach ensures that we are able to divide the problem into smaller sub-problems, making it easier to solve while still maintaining the precision of the final solution.

4.3.2 Complexicity:

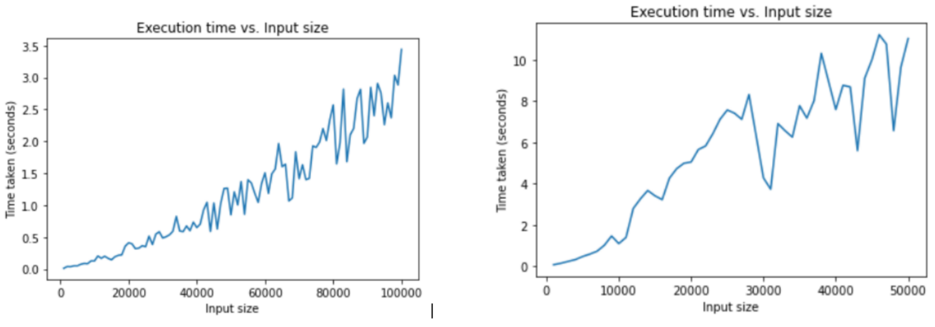
The algorithm for computing the upper hull has a time complexity of $O(n \log n)$. This is due to the fact that we are recursively applying the algorithm on subsets of the original dataset, with each subset being at most half the size of the previous subset. The complexity of finding one edge of the convex hull is $O(k)$, where k is the size of the subset on which the algorithm is applied. Since we are applying this algorithm on the original dataset and recursively on the two subsets defined earlier, the total complexity of the algorithm is $O(n \log h)$. This is because we are summing the complexity of each recursive call, which is $O(n + 2\frac{n}{2} + 2\frac{n}{4} + \dots) = O(n \log h)$, where n is the number of points in the dataset and h is the number of splits, which is equal to the number of edges in the upper hull.

4.3.3 Efficiency on some datasets:

The performance of the last algorithm on the defined datasets varies. Dataset A exhibits the best performance, due to the fact that the convex hull is a square and has a fixed number of edges, namely four. As a result, the number of edges does not increase with the number of points, resulting in a linear time complexity. This linearity is shown in the following figure

Dataset C is created by randomly generating n points within a disk, leading to an unknown number of edges. As a result, the performance of the algorithm is dependent on the points within the dataset. In contrast, dataset D exhibits the poorest performance. The number of edges in the convex hull of n points within a circle is h=n-1, leading to a clear illustration in the following figure.

The execution time for our algorithm to calculate the convex hull of a dataset of type A, with an input size of 50,000, is approximately 1 second. However, it takes 10 seconds to perform the same calculation on a dataset of type D. The logarithmic relationship between input size and execution time can be seen in the fact that $\log(50,000) = 10.82$, which supports the difference in performance between them.



DatasetA DatasetD

Figure 6: Execution time vs input size of our main algorithm

5 Comparison

The complexity of the sweeping algorithm is not impacted by the type of dataset. As previously noted, the difference between the sweeping algorithm and the main algorithm will be substantial for dataset A, contingent on the arrangement of the points in dataset C, and less pronounced for dataset A.

The main algorithm, however, takes longer than the sweeping algorithm when dealing with datasets smaller than 100,000 in size. This can be attributed to the fact that log(n)/log(h) is less than 8.3 for n smaller than 100,000 and h equal to 4 for a dataset of type A, the many intermediate steps involved in the main algorithm (such as calculating the median and the lower line) contribute to the difference in performance. This difference becomes more noticeable when the size of the dataset is huge, i.e. n greater than 10^{12}. However, it is important to note that the main algorithm may become more efficient for larger datasets, as the size of n grows beyond 10^{12}. The large number of intermediate steps in the main algorithm, may not have a significant impact on performance for these larger datasets.

Figure 7 illustrates the results of the comparison between the main and sweeping algorithm in datasets of type A, C, D respectively, where the Y-Axis is the ration of sweeping algorithm to main algorithm in execution time.

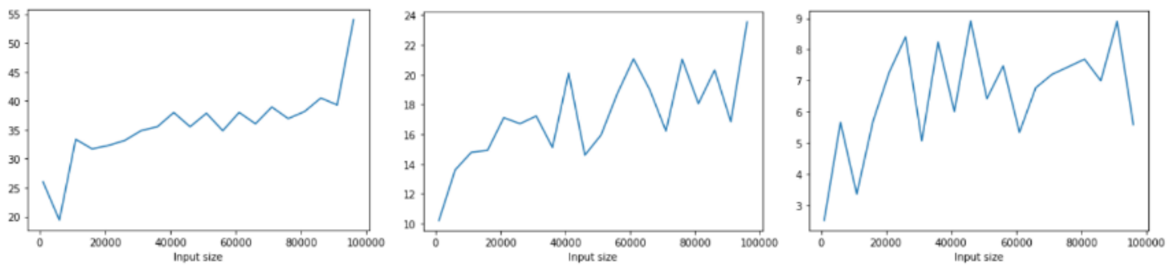


Figure 7: Comparison between the main and sweeping algorithm in datasets of type A, C, D respectively.