Table of Contents

Chapter 1
Introduction

1.1. Objectives

The goal of this project is to develop, analyze and compare different algorithms to build convex hulls. “The convex hull is the smallest convex polygon containing the points.” (Cormen, Leiserson, Rivest, & Stein, 2009). Hence this project focuses on different ways of building that smallest polygon

The algorithms implemented are the following:

- Brute force
- Jarvis's March, also known as gift wrapping algorithm
- Graham scan
- Divide and conquer
- A special case: Convex hull of a simple polygon

We have programmed a simple GUI to graphically see the results, with the possibility of adding points with a mouse click. For studying the performance of these algorithms, we generate a random set of points, and we calculate the amount of time a given algorithm takes to find a convex hull. For every algorithm in this project, you will find a description of the process, and a complexity analysis.

The limits of this project are set to implement and analyse convex hull algorithms for 2d problems. The aim is to properly implement different proposed algorithms by using the python language, analyse the difference of approaches in each of these algorithms, and finally look at the performance of our implemented algorithms versus what the expected performance is.

The description for each algorithm is provided, as well as its pseudocode, and complexity analysis.

Chapter 2
Algorithms

We discuss the different algorithms which we implement in detail, with a simple intuitive analysis and pseudocode. The explanation of the algorithms and code is supplied below.

2.1. Brute Force Algorithm

Description:

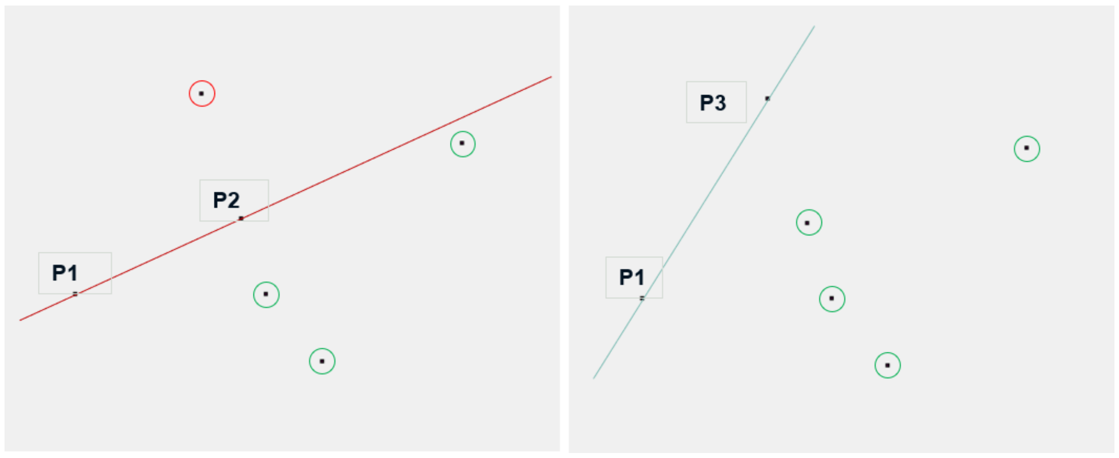


Figure 1 – Brute Force algorithm

The brute force method of analysis is just what the name itself implies. Forcing or trying all combinations of possible results for a given task until we get an appropriate result. This has been explained in Figure 1. For convex hull this translates into testing every possible pair of points to see if they lie on the convex hull. (Levitin, 2012) This can be done by checking if all points lay on one side of the line created by the pair of points and if they do to select them for convex hull. From Figure 1, we can observe that P1 and P2 are not a part of convex hull whereas P1 and P3 have all points on only one of their sides and are hence points on the convex hull.

Pseudocode:

This is described by Levitin (2012) using the following method-

```
Two points are selected with the following x and y axis positions
P1 – (x1,y1), P2 – (x2, y2)
Step 1 -The equation for the straight line passing through point 1 and point 2 is
ax + by = c where, a = y2 − y1, b = x1 − x2, c = x1y2 − y1x2.
Step 2 - For the points with coordinate (x,y), ax + by > c represents that the point is on one side of the line and ax + by < c represents that it is on the other.
Step 3 -If all the n-2 remaining points lay on one side of the convex hull only, then this line built by the points is a part of convex hull.
Step 4- Steps 1-3 are repeated for all n(n- 1)/2 pairs
```

2.2. Direction

All our algorithm to form a convex hull use an important metric to measure whether two lines connecting 3 points form a right (clockwise) or a left (anticlockwise) turn. To measure this, we get the cross product of (p2 - p0) X (p1 - p0) (Cormen, Leiserson, Rivest, & Stein, 2009)

This is represented as d = (p1.x-p0.x)*(p2.y-p0.y)- (p2.x-p0.x) *(p1.y-p0.y)

- d < 0 : points turn left,
- d > 0 : points turn right,
- d = 0 points collinear

(Note: In the code due a slightly different equation, the signs for direction are opposite)

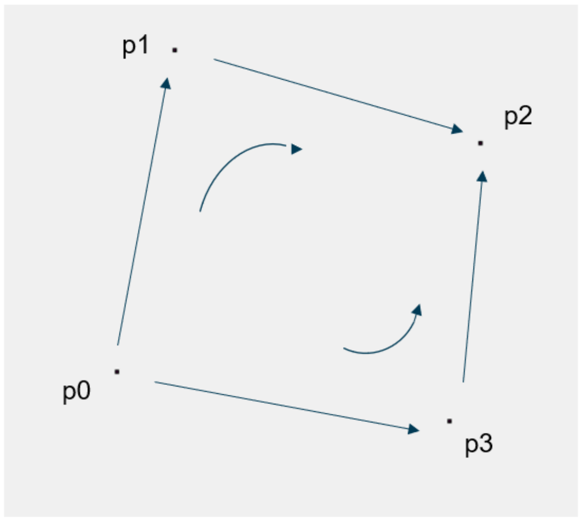


Figure 2 – Direction diagram

So, in Figure 2 p0, p1 and p2 make a right turn and will have d>0, while p0, p3 and p2 make a left turn with d<0.

2.2. Jarvis' march

Intuitively this algorithm imagines extending an arm extending from a point on the convex hull and then swinging the arm arbitrarily until we find the leftmost point which becomes the next point in the convex hull. (Jarvis, 1973)

Pseudocode:

Jarvis explained the method in the following manner.

Step 1-The first point picked is a part of the convex hull. It could be the leftmost and then bottommost point in the set of points. This is the origin of our arm

Step-2 – Next point picked is the next point in our set. We say that it is the left most point.

Step 3- This direction of the origin is compared with the leftmost point to check if any other n-1 points make a right turn to it. This is represented by direction (origin, i, leftmost).

Step 4- If the point makes a right turn it is the new leftmost point. When all the points are exhausted, we choose the new leftmost point and add it to convex hull. Now this becomes our new origin.

Step 5 – We repeat Steps 2-4 until we reach the first point in the hull

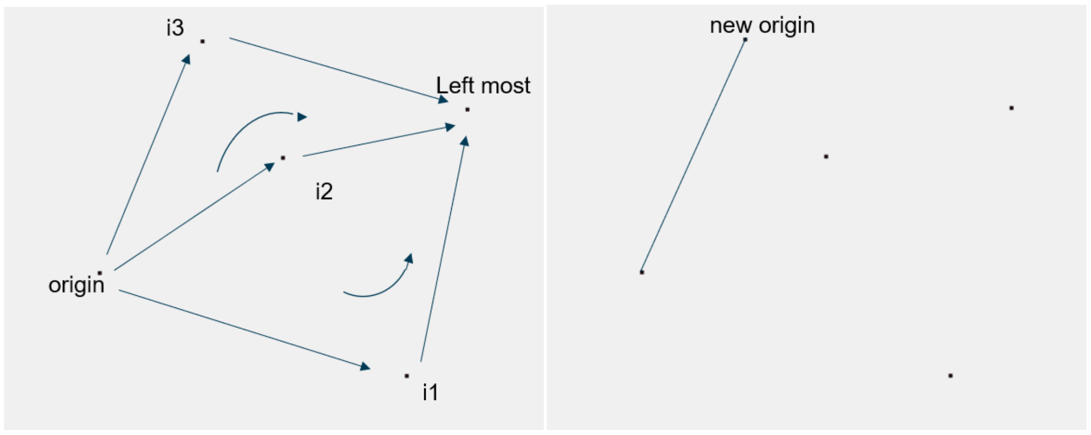


Figure 3 – Jarvis March Algorithm

We can see from Figure 3a that a leftmost point is selected arbitrarily, and then checked with all other points, until the new leftmost is found. In this case i3 is the new leftmost point. After the first run of the loop. The leftmost point i3 becomes the next vertex in the convex hull and is now used as the frame of reference or origin for the next round in the loop (Figure 3b).

2.3. Graham scan

The algorithm was initially proposed by Graham (Graham R. , 1972). Here is the pseudocode for the algorithm.

Pseudocode:

Step 1: Find P0 lowest points in Y-axis

Step 2: Sort points by polar angle with regards to P0. If points have the same angle, keep only the furthest.

Step 4: Add P0 and first point in sorted points list to convex hull list.

Step 3: For last point P(i) in convex hull list and next two points P(j), P(j+1) in sorted list:

```
        While direction is clockwise
            Pop P(j) from sorted list.
        Push P(j) to convex hull list
```

Description

The first step of the algorithm is to find the lowest point in the y axis. This point (P_0) will serve as a starting point for the algorithm. Polar angle is then calculated between the rest of the points and $P_{0.}$ Next, the points are ordered from smallest to greatest by angle.

If two points have the same angle, only the furthest point is kept in the set, as the other points are surely not in the convex hull. We encountered a difficulty in properly identifying these points and removing them. This bug caused the while loop to go back and forth between these colinear points in the next step of the algorithm, until fixed.

Next, we start looping over the points to find the ones part of the convex hull.

First, P_0 is initially part of the convex hull, as well as the first point in the ordered list of points (smallest polar angle with P_0). These points are added to a separate list “Hull”.

Next, we iterate over the ordered list of points by taking the last two points of the “Hull” list and the next point in the ordered points list. This gives a group of 3 points in every iteration.

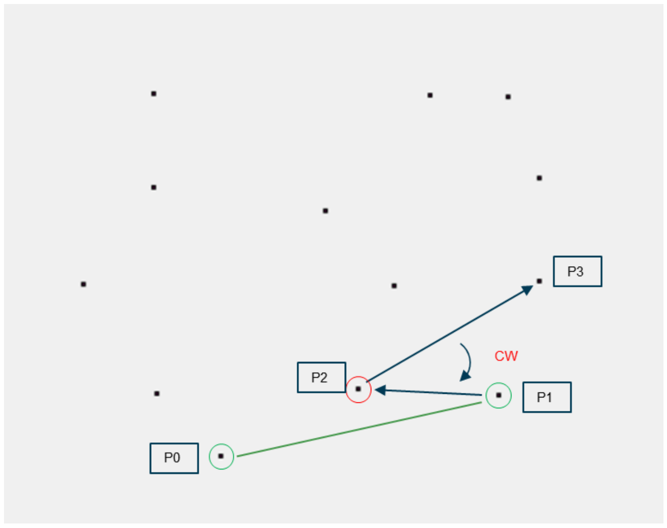


Figure 4 - Graham scan algorithm (Graham R. , 1972)

A direction is calculated with the vector of the first two points and the vector of the last two points. If the new point results in a clockwise direction, then the previous point is not part of the hull and can be replaced by the last one in the “Hull” list.

The algorithm iterates until the are no points left in the points’ list.

2.4. Divide and conquer

This algorithm is divided into three separate parts, Divide Conquer and Combine

1. **Divide**

   Division of the part of the algorithm is done in two steps.

   a. We sort the set of points by the x axis and y coordinates. We use quicksort algorithms for this purpose

   b. We divide the set of points into d portions by the x axis

2. **Conquer**

   We solve the n divisions for the convex hulls of each of them. We use Jarvis algorithm (Jarvis, 1973) to solve for d convex hulls.

3. **Combine**

   Combine the convex hulls with each other. We combine the last two convex hulls together until we are left with only one hull. We use the creation of upper and lower tangents for this. We describe the combination of steps below.

Combining two hulls:

The combination of convex hull involves creating upper and lower tangents of those two hulls. These tangents are built by taking direction of the tangent with the hulls into account. (Martinez, 2015)

First, we select the rightmost point in the left convex hull and the leftmost point in the right convex hull as our initial tangent points. Then start the combination process. The division and the tangent line initialization can be seen in Figure 5

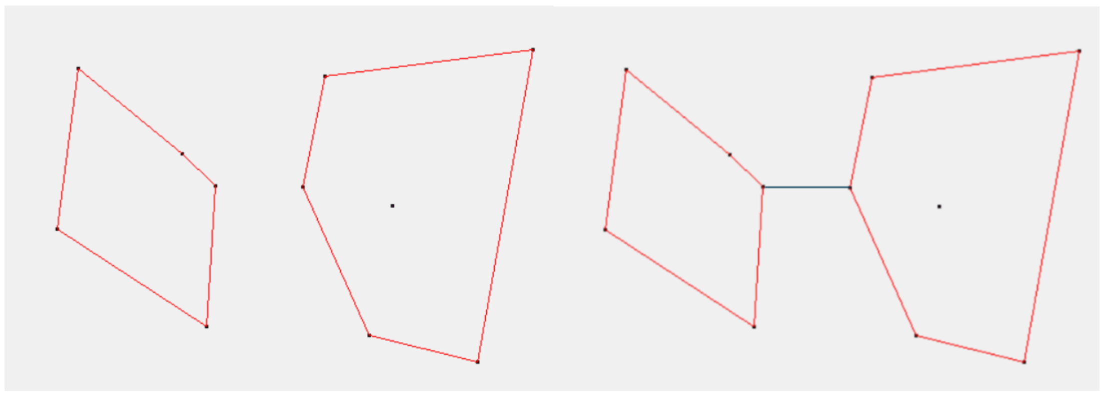


Figure 5-D&C Division and initial tangent line

Finding the upper and lower tangents:

1. **Upper Tangent**

   Start from initial tangent line

   Do until the line is tangential with both hulls

a. Check if right point of line makes a tangent with the right convex hull. If not move the right point up until it does.

b. Check if left point of line makes a tangent with the left convex hull. If not move the left point up until it does.

2. **Lower Tangent**

Start from initial tangent line

Do until the line is tangential with both hulls

c. Check if right point of line makes a tangent with the right convex hull. If not move the right point down until it does.

d. Check if left point of line makes a tangent with the left convex hull. If not move the left point down until it does.

The process of finding the upper tangent can be seen in Figure 6. As we can observe, the right side is incremented until it makes a tangent to the right hull. This can be done using direction (t1,t2,rh) for the right hull, where t1 and t2 are points on the tangent and rh is the next point in the hull. If the direction of these points is left, it is not tangential to right hull and if the direction is a right turn then the line is tangential. This is subsequently done for each the hull and for the upper and lower tangents.

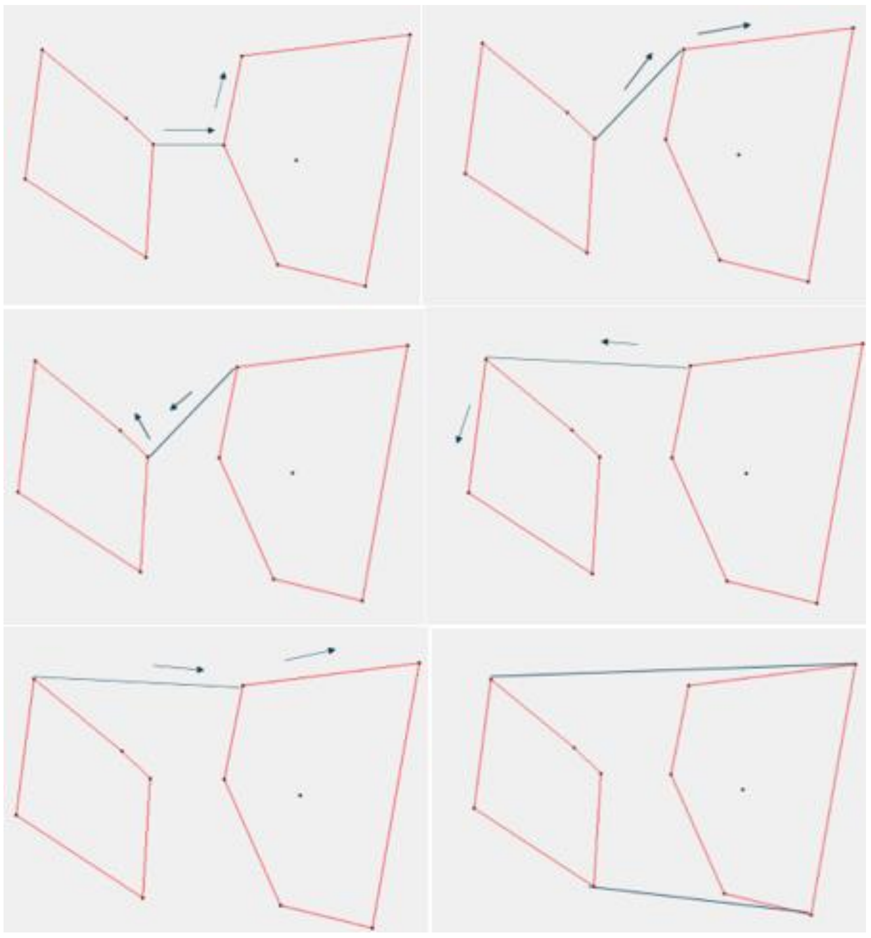


Figure 6- Creation of Upper Tangent

2.5. Convex hull of a simple polygon

This algorithm by Graham & Yao (1983) is called LeftHull as the algorithm is only for half of a simple polygon. Figure 8 represents the pseudocode of the algorithm and can be used to get the upper half of the hull. The algorithm uses a stack data structure and the notations for the algorithm are as follows.

3. $V(v_m, v_1, v_2, \ldots\ldots, v_{m-1})$ - Vertices of simple polygon :
4. $Q(q_0, q_2, \ldots.., q_t)$: Stack with convex hull vertices:
5. x : input

6. y: Vertex before q_t in V
7. Push x: push Q(x), t = t + 1
8. Reject x: Do nothing/Increase x by 1
9. L[p1,p2]: Create line

The algorithm runs until the input is exhausted

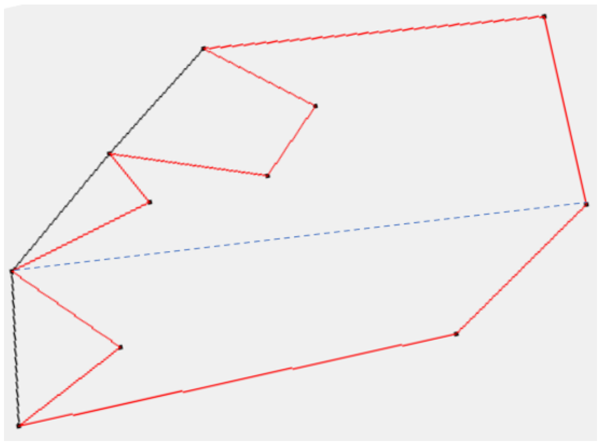


Figure 7-LeftHull Division

This algorithm has two checks. One check is for direction, and one check is for whether c lies on one side of the line L or the other. This check is done in a similar manner to Brute Force algorithm. This algorithm requires the first and last point of the vertex to be a part of the hull. This can be gotten using the leftmost and the rightmost point in the set. Then it divides the polygon by drawing a line through these points (Figure 7) (Note :This way is not completely reproduced in the code, The modifications are talked about later).

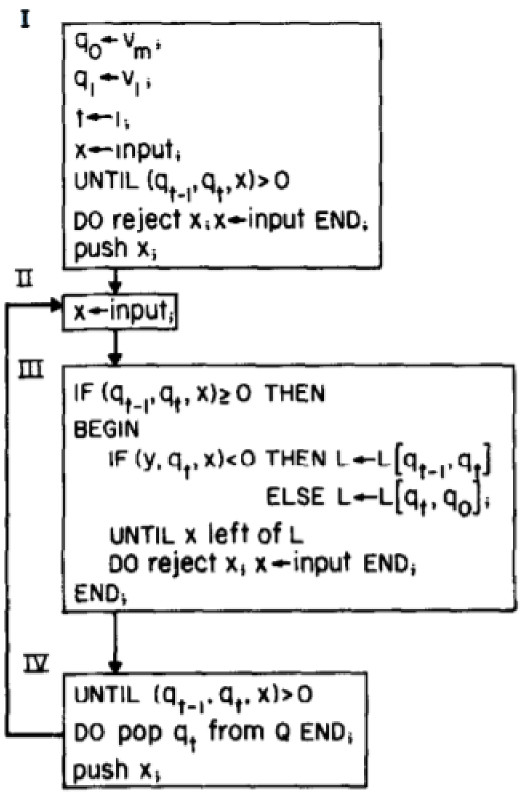
Let us look at the Figure 8 to understand the pseudocode,

Box I of Figure 8, Stack is filled with the two initial points part of the convex hull, the inputs are rejected until they are at the top of the dividing line and then input is pushed

Box II - Inputs are incremented by 1.

Box III - The two checks are done and input it incremented.

Box IV – The convex hull stack is updated



```
q_0 ← v_m;
q_1 ← v_1;
t ← 1;
x ← input;
UNTIL (q_{t-1}, q_t, x) > 0
DO reject x; x ← input END;
push x;
```

```
x ← input;
```

```
IF (q_{t-1}, q_t, x) ≥ 0 THEN
BEGIN
    IF (y, q_t, x) < 0 THEN L ← L[q_{t-1}, q_t]
                       ELSE L ← L[q_t, q_0];
    UNTIL x left of L
    DO reject x; x ← input END;
END;
```

```
UNTIL (q_{t-1}, q_t, x) > 0
DO pop q_t from Q END;
push x;
```

Figure 8- (Graham & Yao, 1983)

We can see from Figure 9 the two cases which can happen in box III. In Case a, the point selected q_{t-1} and q_t, will have to be popped from the stack. In Case b x will directly be selected as a point of the convex hull and pushed into stack for further checks.

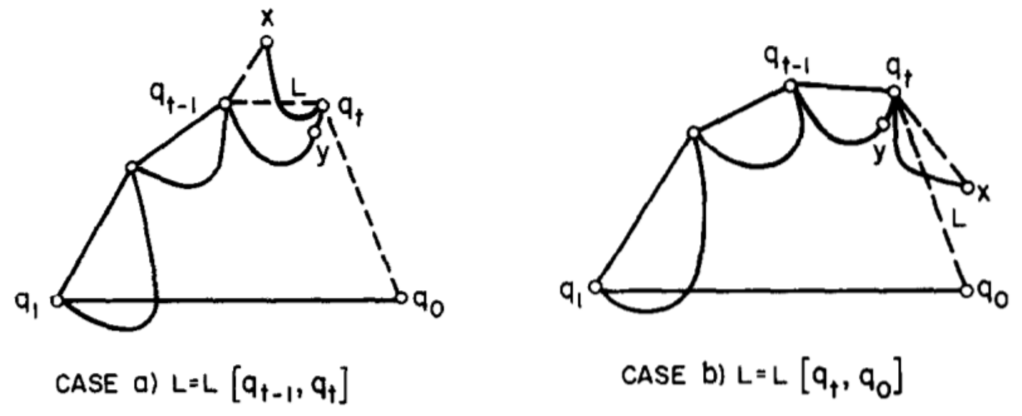


Figure 9- Cases of Box III (Graham & Yao, 1983)

We change the code and implement only half the convex hull. We wrap the algorithm from Box 3 until point q_2 in the hull. This means that the hull is created properly, and we don't need to implement the other half of the algorithm. The negatives are that the algorithm will run for a slightly more time and the second point p_1 still needs to be a part of the convex hull for the code to run properly.

Chapter 3
Results

3.1. Analysis of Implementation

1. Brute Force

Complexity:

There are n(n- 1)/2 pairs of points in the set. We also have to check each of the remaining n-2 points to see if all points lie on one side. Hence, the complexity of this algorithm is in **O(n^3)**.

This is very apparent in our implementation of our algorithm. The time required steeply increases as the number of points increase.

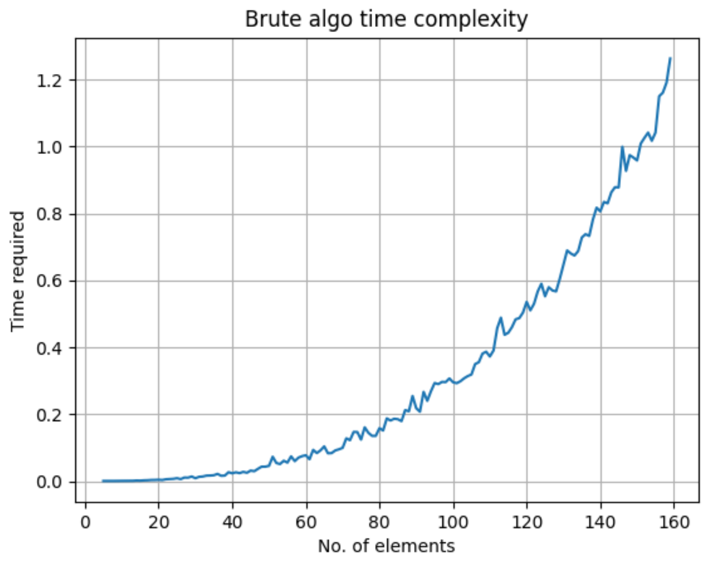


Figure 10- Brute force time complexity graph

We can see for a set of points going from 5 to 160 the time taken for the algorithm to find a convex hull. The results are what we expected as it shows an exponential curve.

2. Jarvis March

Complexity:

At every given iteration, the algorithm spends O(n) time. It repeats this until it finds all the vertices of the convex hull. If the convex hull has h number of vertices, then the algorithm needs to iterate h*n times. This makes the overall average complexity O(hn). If in the worst-case scenario all the points tested are a part of the convex hull, then this algorithm complexity becomes O(n^2)

Here is a plot of our actual complexity experimentation with the implementation of the algorithm.

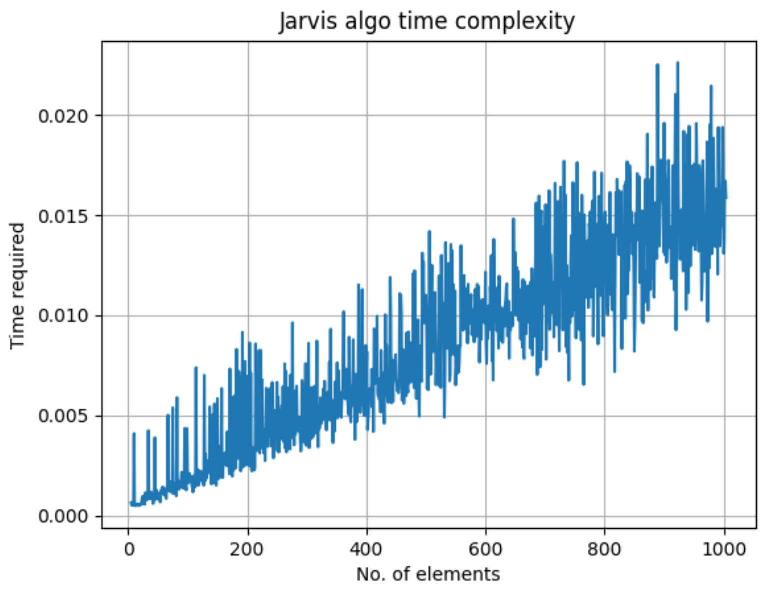


Figure 11- Jarvis march time complexity graph

As expected, the algorithm seems to find convex hulls in an almost linear time. Here the experience was done for a set of points going from 5 to 1000 points. Compared to the brute force algorithm, we are clearly much more efficient.

3. Graham Scan

The sorting phase of points by polar angle takes time O(nlogn). Then, the algorithms iteration over the points takes time O(n), as any given point is evaluated exactly once and is discarded at most once.

The total time complexity is then O(nlogn).

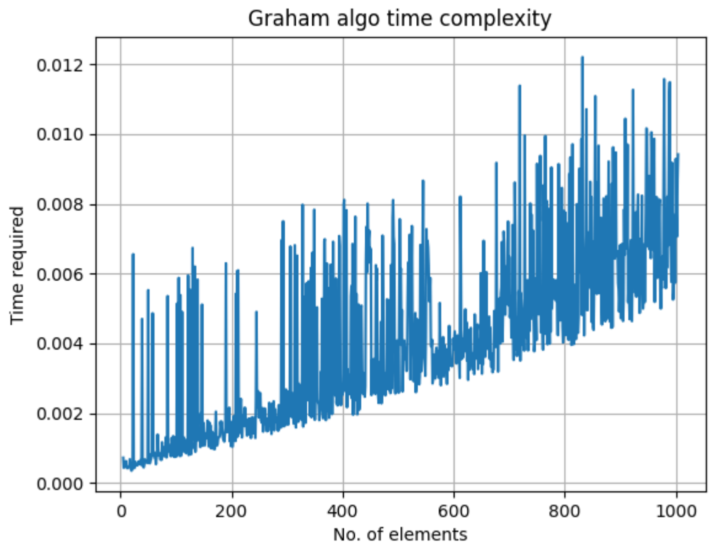


Figure 12- Graham scan time complexity graph

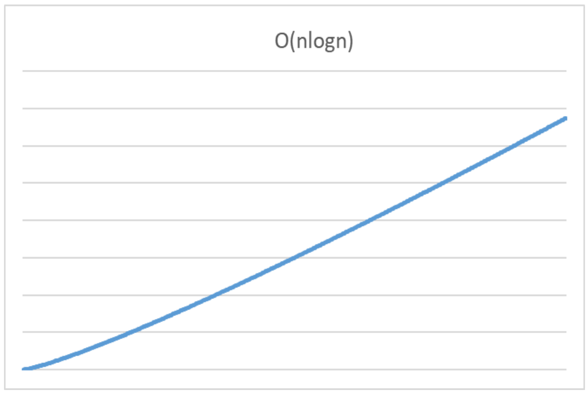


Figure 13- O(nlogn) time complexity shape

As we can see, for sets of points going from 5 to 1000 points, the algorithm behaves as expected. The curve is almost identical to a nlogn curve as shown in fig [13].

4. Divide and Conquer

Complexity:

Divide – worst case complexiy is O(n^2) (Quicksort) because we sort and then divide which takes linear time

(Note: Avg complexity of Quicksort is O(nlogn))

Conquer- Jarvis Algorithm times the division. O(d*n^2), So it is still O(n^2)

Combine – Each point is checked at most once, so this is O(n)

So, the overall **worst** complexity of the algorithm is O(n^2)

Here is again a plot with our own experimentation.

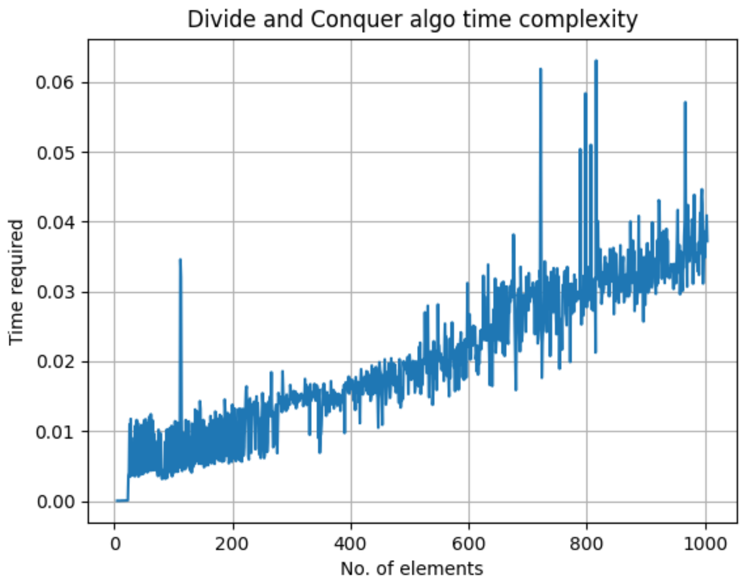


Figure 14- D&C algorithm time complexity

For a set of points going from 5 to 1000 points, we actually see a curve that is very similar to Jarvis march curve in fig[11]. That is to be expected as the overall time complexity of this algorithm is dependent on the conquer step, which is the step Jarvis march algorithm is applied. The time increases due to it being run d times.

5. Convex Hull of a simple polygon

The algorithm uses a stack data structure and as every element is pushed or pulled into the stack at most once the complexity of this algorithm is linear. So, the complexity is **O(n)**

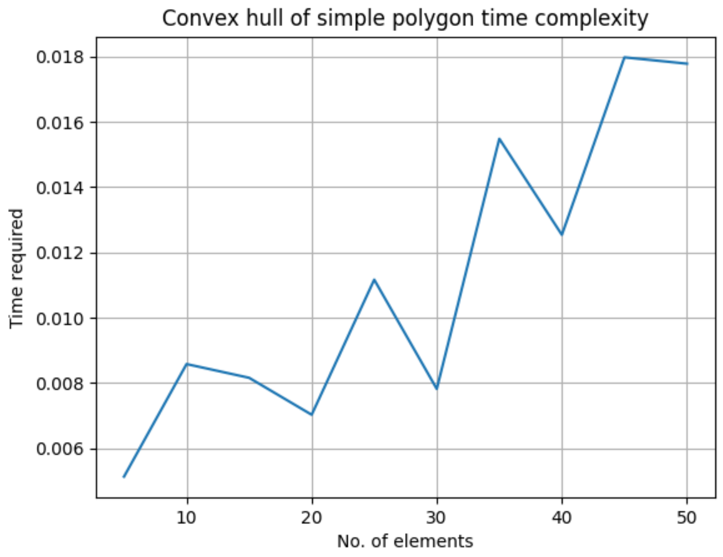


Figure 15- Convex hull of simple polygon algorithm time complexity

In this figure, we see the running time for the algorithm for polygons going from 5 points to 50, with a step of 5.

3.2. GUI:

We have built a GUI using python library tkinter to apply these algorithms. It can be used without knowing what is going on in the background. Figure 16 shows the display of the GUI

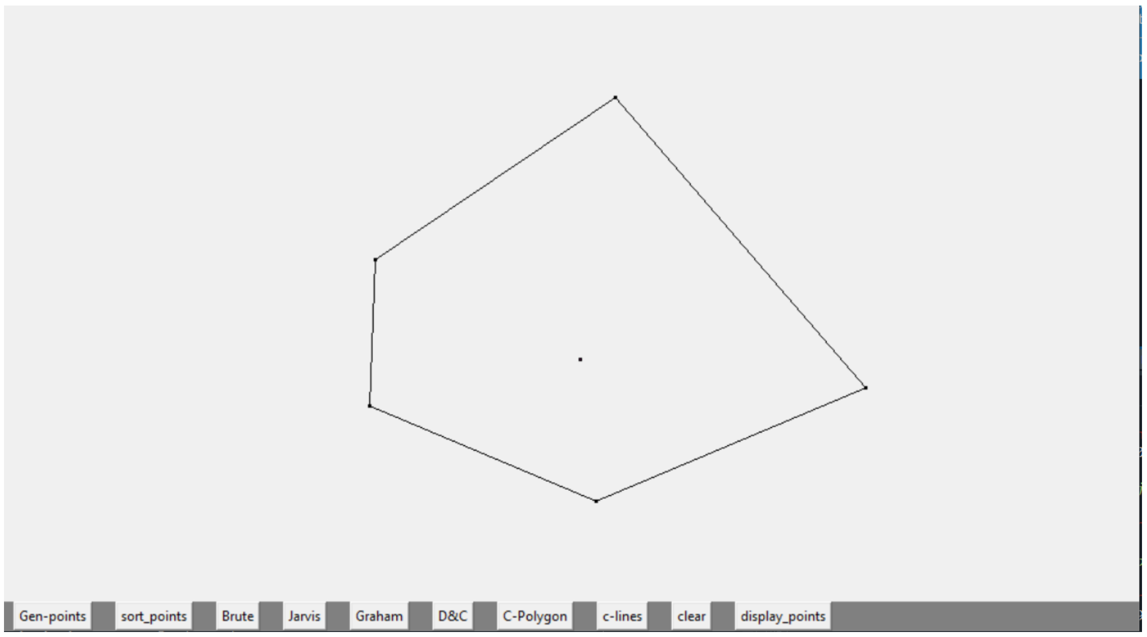


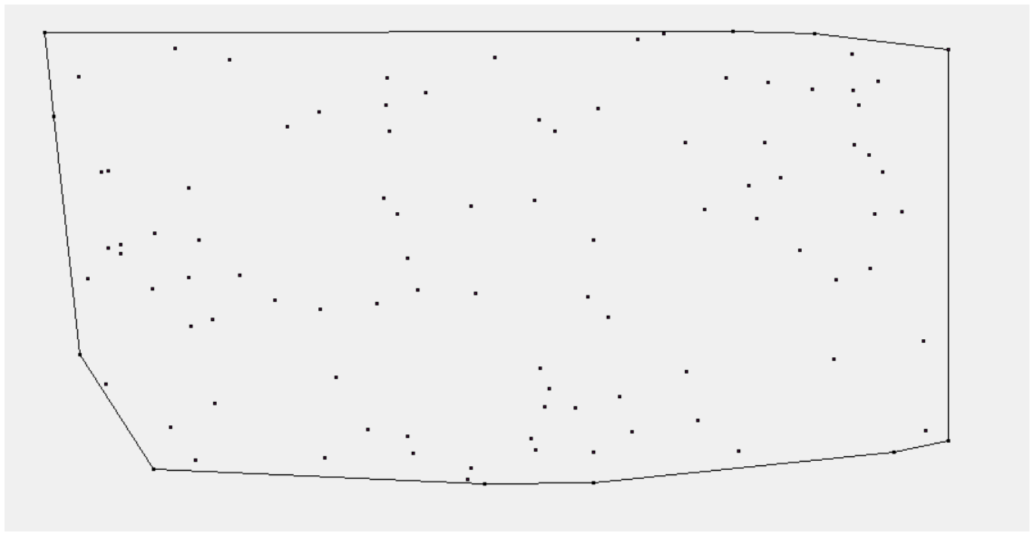
Figure 16 - GUI

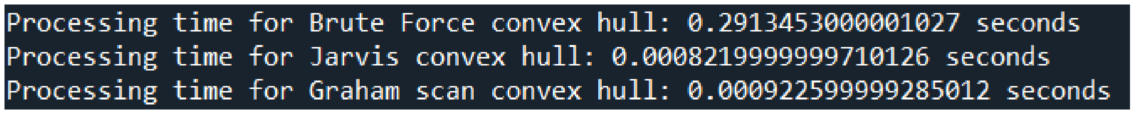
There are buttons in the GUI to do various things like Gen-points button, which could be pressed to create multiples of 100 points. We can also use a mouse to add points manually. The buttons can also be used to implement different algorithms. The c-lines button can then clear the hull so that we can see the implementation of all the algorithms.

The vertices of a simple polygon must be charted using mouse in a clockwise manner and cannot be charted randomly to get its convex hull. The clear button deletes all the points from the list.

3.3. GUI Results

For a randomly generated 100 points, we applied the discussed algorithms and obtained the following results:





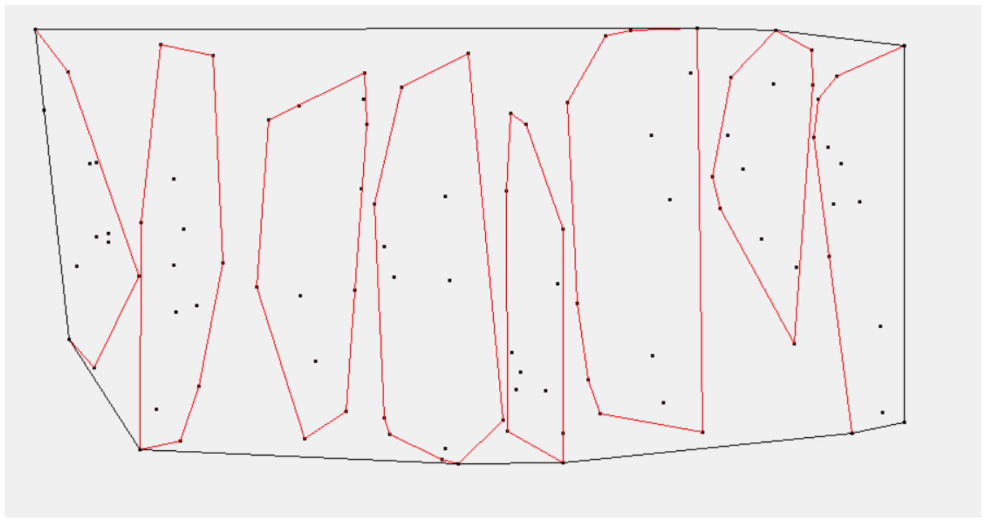
```
Processing time for Brute Force convex hull: 0.2913453000001027 seconds
Processing time for Jarvis convex hull: 0.0008219999999710126 seconds
Processing time for Graham scan convex hull: 0.000922599999285012 seconds
```

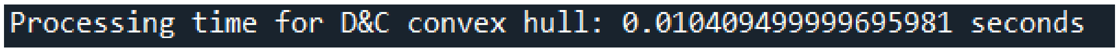
Figure 17- GUI implementation

Here we have the processing time for brute force algorithm, Jarvis march and Graham scan.

We clearly see that brute force is very time expensive taking 0.2 seconds, and Jarvis and graham have around the same processing time. Brute force takes a little less than 10^3 times the amount of time Jarvis or Graham takes

When applying divide and conquer algorithm with 8 division to the same set of points, we get the Figure 18. As we can see the algorithm divides the points into 8 portions before solving and then combining. The time it takes still less than 10 times brute force, but it lags the other two algorithms.

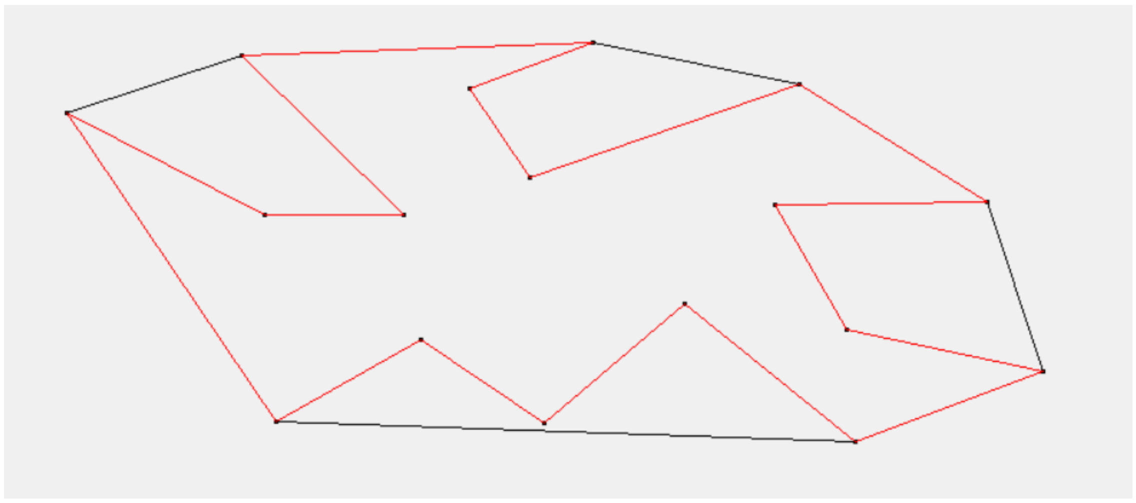


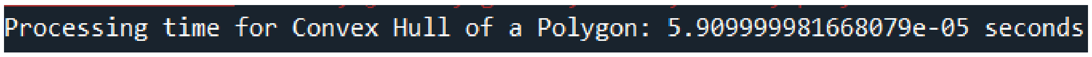


```
Processing time for D&C convex hull: 0.010409499999695981 seconds
```

Figure 18 – GUI implementation of D&C

The convex hull of simple polygon is created by putting vertices into the GUI in a clockwise manner. We can see the implementation in Figure 19. The red lines are the edges of the simple polygon and the black lines is the convex hull surrounding it. The simple polygon with 17 vertices takes a very low amount of time to process.





```
Processing time for Convex Hull of a Polygon: 5.909999981668079e-05 seconds
```

Figure 19 -GUI implementation of convex hull of a simple polygon

Chapter 4
Conclusion

4.1. Difficulties and Learnings

We have implemented multiple different algorithms in this project. The project taught us to structurally implement different algorithms. We faced multiple programming hurdles throughout the implementation, including a way to check the correctness of our implementation for a large number of points. The solution to this was creating a visual check viz. the GUI. All algorithms made us learn something new. The most difficult algorithms to implement were Divide and Conquer and the Hull of Simple Polygon. Divide and Conquer, required us to learn multiple data structures for storing the hulls. In the end we used lists of lists to divide the hulls into different section. The combination was extremely difficult to do, getting us different results. We learned to map our solutions on paper to chart what would happen next and printing results throughout our code.

The convex hull of a simple polygon did not have an intuitively understood paper. The difficulty in understanding the algorithm was somewhat same as Divide and Conquer. The code was then charted on paper to understand it. The implementation only was for half of the hull and needed two points which were a part of the convex hull as basis. We were able to reduce that necessity by running the loop of the algorithm until the second point in our set. This reduced our need for two anchor points at the sake of slightly higher processing time (Still linear) and now we only need the second point to be a part of the convex hull. We learned how to effectively challenging complex problems by using simple methods.

4.2. Future Work

There are many ways of speeding up the process of finding the convex hull for a set of points, one of them is to find the lowest and highest points, with regards to both the x axis and the y axis. With this set of 4 points, we can construct a square. All the point inside of this square are not part of the hull. By doing so, we reduce greatly the number of points to evaluate. There are also extensions for some of these algorithms to be applied to n dimensions, which might make a good subject to study as a continuation for this project.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms.* MIT press.

Graham, R. (1972). An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*.

Graham, R. L., & Yao, F. F. (1983). Finding the convex hull of a simple polygon. *Journal of Algorithms*.

Jarvis, R. A. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information processing letters*.

Levitin, A. (2012). Introduction to the design & analysis of algorithms. Pearson.

Martinez, T. (2015). *Convex hull project description*. Retrieved from CS 312 – Algorithm Analysis: http://axon.cs.byu.edu/~martinez/classes/312/Projects/Project2/project2.html