RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

DATA STRUCTURES & ALGORITHMS PROJECT REPORT

Collision Detection and Avoidance using Convex Hull Algorithms

Pratik Mistry, Shounak Rangwala, Pranit Ghag, Vikhyat Dhamija

April 28, 2020

Table of Contents

1. Introduction

1.1. What is a Convex Hull?

The convex hull of a set of points is defined as the smallest convex polygon that encloses all of the points in the set. Convex means that the polygon has no corner that is bent inwards. Convex hulls open sets are open, and convex hulls of compact sets are compact. Every convex set is the convex hull of its extreme sets [1]. The convex hull is a ubiquitous structure in computational geometry. Even though it is a useful tool in its own right, it is also helpful in constructing other structures like Voronoi diagrams, and in applications like unsupervised image analysis [2].

Convex hull is a concept used to detect objects using computational geometry. Even though it is a useful tool in its own right, it is also helpful in constructing other structures like Voronoi diagrams, and in applications like unsupervised image analysis. We can visualize what the convex hull looks like by a thought experiment. Imagine that the points are nails sticking out of the plane, take an elastic rubber band, stretch it around the nails and let it go. It will snap around the nails and assume a shape that minimizes its length. The area enclosed by the rubber band is called the convex hull of P. This leads to an alternative definition of the convex hull of a finite set P of points in the plane: it is the unique convex polygon whose vertices are points from Pand which contains all points of P [2].

1.2. Problem Domain 1: Self Driving Vehicles

A self-driving car, also known as an autonomous vehicle (AV), connected and autonomous vehicle (CAV), driverless car, or robotic car, is a vehicle that is capable of sensing its environment and moving safely with little or no human input. Collision avoidance and detection is essential for a system with almost no human input. A collision avoidance system (CAS), also known as a pre-crash system, forward collision warning system, or collision mitigation system, is an automobile safety system designed to prevent or reduce the severity of a collision [3]. Detection of objects is necessary for collision avoidance as the algorithm can find a route that is safest if an object is detected on its original path. Once a new path is found the system constantly has to check for obstacles to avoid damage to the vehicle. LiDAR points are used by self-driving car systems for collision detection and avoidance. LiDAR(Light Detection and Ranging) points are 3 dimensional points in space used to represent different objects. Object detection is a key task in autonomous driving. Autonomous cars are usually equipped with multiple sensors such as camera, LiDAR. These points in space are used to determine the shape of the object using a convex hull.

A convex polygon can be made using these LiDAR points to determine the shape of the object or obstacle. The shape of the obstacle will be used for detection of the object and the car. If the convex hull of a car avoids collision with obstacles then so does the car. Since the computation of paths that avoid collision is much easier with a convex car, then it is often used to plan paths. Finding the intersection between arbitrary contours is computationally much more expensive than finding the collision between two convex polygons. It is beneficial to use a convex hull for collision detection or avoidance.

1.3. Problem Domain 2: Fingerprint matching

High-resolution fingerprint identification system (HRFIS) has become a hot topic in the field of academic research. Compared to traditional automatic fingerprint identification systems, HRFIS reduces the risk of being faked by using level 3 features, such as pores, which cannot be detected in lower resolution images. However, there is a serious problem in HRFIS: there are hundreds of sweat pores in one fingerprint image, which will spend a considerable amount of time for direct fingerprint matching [4].

The purpose of fingerprint matching is to compare two fingerprint images and return a similarity score that represents the probability of match between the two fingerprints. The performance of an automatic fingerprint identification system is greatly determined by its fingerprint matching algorithm. Security can be compromised if the fingerprint authentication system is flawed.

Convex hulls of the fingerprint to be tested can be computed and compared with the convex hull of the fingerprint already in the system. If the convex hull of the fingerprint that is being tested matches with the convex hull of the fingerprint in the system then the fingerprints match. The convex hulls of unique fingerprints will be unique and using this principle, fingerprints matching can be implemented. In this method, fingerprints are aligned using singular points. Then minutiae are matched based on the alignment result. To reduce the impact of deformation, a convex hull is built for each of these fingerprints. Pores in these convex hulls are used for matching. Comparing this method with other fingerprint matching methods like random sample consensus method, minutia and ICP-based method, and direct pore matching method shows that this method using convex hull is more efficient [4].

1.4. Algorithms for Convex Hull constructions

Convex hull is an important concept in computational geometry with many applications as given above. Other applications include problems in mathematics, statistics, combinatorial optimization, economics, geometric modeling, ethology, quantum physics, etc. The algorithms used for building the convex hull are,

- Jarvis March Algorithm
- Quickhull Algorithm
- Graham Scan Algorithm
- Monotone Chain Algorithm

2. Jarvis March

2.1. Description

This algorithm is also called the gift-wrapping algorithm. The most intuitive algorithm which comes to our mind when we think of wrapping a string around some nails hammered into a piece of wood. This algorithm was discovered by R.A. Jarvis in 1973 and has been one of the most used algorithms for calculating convex hulls.

2.2. Methodology

The Jarvis March algorithm has a principle very similar to selection sort. In selection sort, we select the least number in the array and add it to the sorted array. Similarly, in the Jarvis-March algorithm, we find the point that is the rightmost point from where we stand and add this to the array containing the points for the convex hull.

To start off, we select the vertex that has the least x-coordinate (the leftmost point). This point is guaranteed to be in the convex hull array. If there are 2 such points having the same x-coordinate, select the one which has the least y-coordinate.

Basically, at every iteration, imagine yourself standing on the vertex and looking at all the other vertices in the group [5]. You will select the rightmost point, add it to your collection called "convex-hull" and then move onto that point. Repeat the same process till the right most point is the first point you stood on.

The orientation checker function is based on comparing the slopes of 2 lines [6]. Since the algorithm is all about finding the rightmost point. If the slope of the line joining point1 from the vertex you are "standing on" is more than the slope of the line joining point2 to that vertex, then point2 is more to the right than point1. If there is a case when both the points, point1 and point2, are collinear from the vertex, then the point selected is that whose distance from the vertex is most. Thus there is also an inbuilt distance method which calculates and compares the distance in the case as mentioned earlier.

2.3. Pseudocode

Below is the pseudocode of Jarvis-March Algorithm:

```
Jarvis_March(N):                        # N points in (x,y) form
    Lowest :=1
    For i := 2 to N:
        If i[x] < Lowest[x]:
            Lowest := i
    p := Lowest
    convex_hull.add(p)
    repeat
        cursor := p + 1                 # p and cursor points cannot be the same
        for j := 2 to N :
        # checks if the point j lies to the right of the cursor point
            if orientation_checker (p,j,cursor):
                cursor := j
        # cursor is new rightmost point, add to the convex_hull and move onto it
        convex_hull.add(cursor)
        p := cursor
    until p := Lowest                   # stop process when you return to starting
    point
```

2.4. Complexity Analysis

2.4.1. Time complexity

Average Case: For each point that is added to the convex hull boundary, the algorithm runs operations that have a time complexity of O(N), where N is the total number of points in the cluster. If there are going to be H points in the convex hull, the time complexity for the Jarvis-March algorithm is O(H*N) [7].

Worst Case: This happens when H=N i.e all the N points lie in a circle.

Best Case: Jarvis March is a very effective algorithm if we know that the number of points in the convex hull are going to be very small. Therefore if we know that H<<N, the algorithm almost has a linear time complexity of O(N).

Jarvis-March algorithm is therefore output-sensitive. The smaller the output, the faster the algorithm runs [5].

2.4.2. Space Complexity

Below is the space complexity of Jarvis Algorithm:

- O(n) - Store the points in the list
- O(n) - Store the convex hull points in the stack

Since, the space complexity is the same for both storing points in list and storing convex hull points in the array, the overall space complexity of the algorithm is O(n).

3. Quickhull

3.1. Description

Quickhull is an improvement on the naïve gift-wrapping algorithm. It was invented by C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa in 1996. Quickhull is a divide and conquer algorithm that works very similarly to quicksort by partitioning the dataset into 2 sets that individually go through the algorithm and the final results are clubbed into one forming the convex hull. The special property of quickhull is that the time complexity of the algorithm is actually determined by how the algorithm runs [8]. Unlike the quicksort algorithm, this is not a randomized algorithm and each dataset runs in a way that the big-Oh notation cannot be easily calculated. This will be better described later after explaining the functionalities of the algorithm.

3.2. Methodology

As mentioned earlier, quickhull is a divide and conquer algorithm. Assume you have a set of N points in the form of (x,y)

The following are the basic steps of the Quickhull algorithm [9]:

1. Find 2 starter points. These 2 points are those with the minimum (called *lowest*) and the maximum x-coordinates (called *highest*). These points are the rightmost and the leftmost points of the collection and so have to be a part of the convex hull. In case there are 2 or more points that have the same minimum x-coordinate, choose the one with the least y-coordinate. Similarly, if there are 2 or more points with the same maximum x-coordinate choose the one with the biggest y-coordinate.
2. Imagine a line joining these 2 extreme points, dividing the entire dataset into sets, one lying to the left of the line and the. others lying on the right of this line. Divide and conquer algorithm works simultaneously on these 2 sets and concatenates the result into 1.

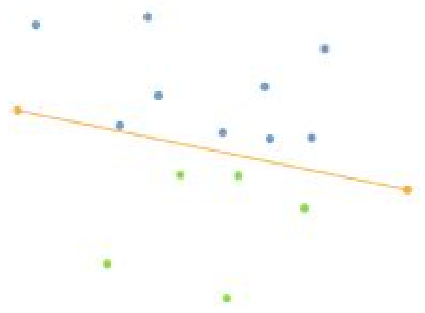


Figure 1: line joining the lowest and highest *(Best Viewed in Color)*

3. On the left side (considering one side for explanation purposes), we find the point that is farthest from the line lowest and highest. This point (called *furthest*) if the furthest point perpendicular to the line and by reason is also present in the convex hull.

4. We imagine a triangle between lowest, highest and furthest. All points in the left side, which lie inside this triangle are by default removed from consideration for the convex hull. Only the points that remain outside the triangle are possible candidates.

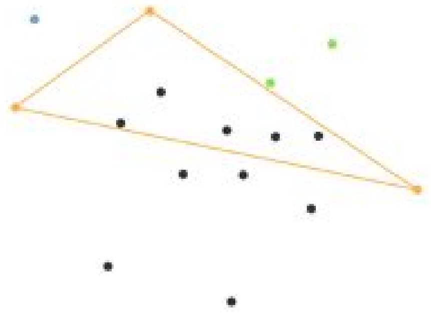


Figure 2: triangle made between lowest, highest and furthest *(Best Viewed in Color)*

5. We now repeat the steps 2,3,4 recursively for the lines made by joining lowest and furthest and the line made by joining farthest and highest. Consider the points that lie on the left side of these lines only. When carrying out this recursion on the right side of the main division, consider only the right side points of any recursive division formed.

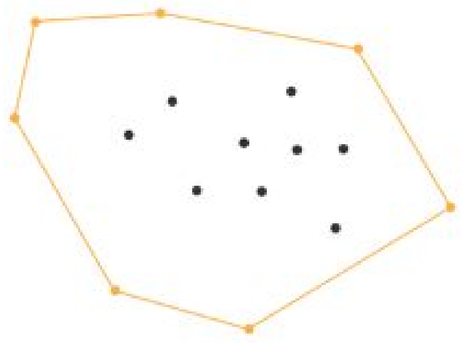


Figure 3: Final Result

3.3. Pseudocode

Below is the Pseudocode for QuickHull Algorithm [8]:

```
Input = a set S of n points
# Assume that there are at least 2 points in the input set S of points

function QuickHull(S):
    # Find convex hull from the set S of n points
    Convex Hull := {}
    Find left and right most points, say A & B, and add A & B to convex hull
    Segment AB divides the remaining (n − 2) points into 2 groups S1 and S2
        where S1 are points in S that are on the right side of the oriented line
        from A to B,
        and S2 are points in S that are on the right side of the oriented line from
        B to A

    FindHull(S1, A, B)
    FindHull(S2, B, A)
    Output := Convex Hull
end function

function FindHull(Sk, P, Q):
    # Find points on convex hull from the set Sk of points
    # that are on the right side of the oriented line from P to Q
    If Sk has no point then
        return
    From the given set of points in Sk, find farthest point, say C, from segment PQ

    Add point C to convex hull at the location between P and Q
    Three points P, Q, and C partition the remaining points of Sk into 3 subsets: S0,
    S1, and S2
        where S0 are points inside triangle PCQ,
        S1 are points on the right side of the oriented line from P to C,
        and
        S2 are points on the right side of the oriented line from C to Q.
    FindHull(S1, P, C)
    FindHull(S2, C, Q)
end function
```

3.4. Complexity Analysis

3.4.1. Time complexity

Average Case: The time complexity for quickhull is O(N*log(r)). Here, N is the total number of points in the dataset and r is the total number of "*furthest*" points that are found by recursion of the algorithm. Because we do not know the exact number of points that will be considered for recursion we cannot exactly calculate the time complexity [8].

Best case: The best case for the quickhull algorithm will be the case where the data is in such a format that the number of points that can be the extremes are very small in number. As compared to the total number of points. Imagine a dense cluster of points and a few outliers on the boundaries.

Worst Case: The worst case situation is of complexity $O(N^2)$. This situation happens when the points are arranged in a circle. This will call the recursion N times and will consider every last point as the furthest till the recursion runs.

3.4.2. Space Complexity

The space complexity of Quickhull is O(N) because we use arrays in the order of (number of points in the set).

4. Graham Scan

4.1. Description

Graham Scan is one of the first algorithms in computational geometry published by Ronald Graham in 1972 [13]. It finds a convex hull for a finite set of points placed in a 2D plane in linearithmic time. Thus, it has better performance than Jarvis-March [5] algorithm whose worst case time complexity is nearly N^2. Graham Scan is a very popular and widely used algorithm in many fields where convex hulls are needed.

4.2. Methodology

Now we will look into detailed working of Graham's Scan. The algorithm works in two main phases [14]:

- Sorting of the given points (x,y coordinates) based on their polar angles with respect to x-axis and anchor point
- Scanning the list of sorted points one by one to find the convex hull points

Consider a list of points as [(5,2),(9,6),(1,4),(0,0),(3,3),(5,5),(7,0),(3,1)]. Below are the detailed step by step execution of the algorithm [14]:

1. *Find the anchor point*

- This is the first and most important step of the algorithm. In this step, we find a point - *P0* having lowest y-coordinate among all the other sets of points.
- If there is a tie between two points i.e. two or more points has the same lowest y-coordinate, then we find consider the point having lowest x-coordinate.
- This point is referred to as an *Anchor Point.*
- In our case, since points (0,0) and (7,0) have the same lowest y-coordinate, we choose point (0,0) as anchor point. Figure 4 shows the points and the anchor point.

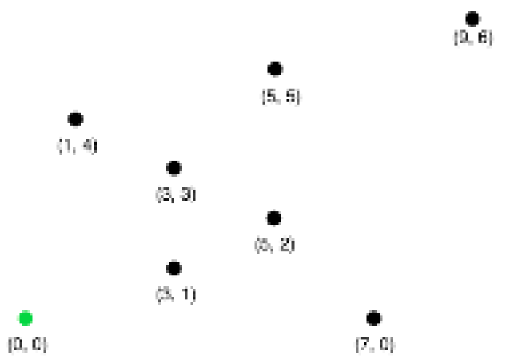


Figure 4: Points in 2D place having Anchor Point *(Best Viewed in Color)*

2. *Sorting the points*

- Again, this is the first phase and a very important one. In this phase, we sort out the points based on the polar angle θ of the line between point and anchor point with respect to the x-axis. Polar angle between two points P1 and P2 is calculated as [15]:

ϴ = tan (y2-y1/x2-x1)

- While implementing, we don't calculate the angle, instead, we calculate the relative orientation of two points to find out which point makes the larger angle.
- If there are two points whose polar angles are the same then we consider the point closest to the anchor point.
- To find the point closest to the anchor point among points having the same polar angle, we compare the distances calculated using the distance formula. The distance formula between two points can be given as:

$$d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

- The sorted points are: [(0,0),(7,0),(3,1),(5,2),(9,6),(3,3),(5,5),(1,4)]. Since point (3,3) and (5,5) has the same polar angle with anchor point, we sort them based on the distance w.r.t to anchor point. Thus, as point (3,3) is closer in terms of distance, it appears before (5,5) in the sorted list.
  Note: The first point in the list will always be the anchor point
- Sorting method plays a very important role here. We have used the Quick Sort method to sort the points as its time complexity is linearithmic.
- The sorted points are plotted in Figure 5 below

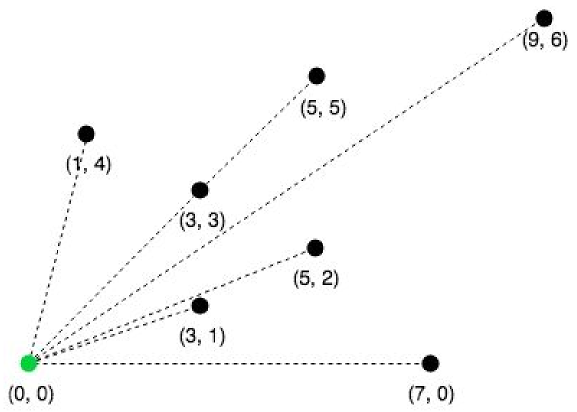


Figure 5: Points sorted by their polar angle and distances w.r.t to anchor point *(Best Viewed in Color)*

3. *Finding the convex hull points*

- This is the second phase of the algorithm where we scan each point from the sorted list of points and find convex hull points.
- We use *Stack* as a data structure to hold the convex hull points.
- Since, the first two points are always part of the convex hull, we add them into the stack.

- Then, we scan the rest N-2 points one by one to find the convex hull points.
- For each point *Pi*, we find the orientation of the point Pi and the top two points of the stack say P1, P2.
- If the direction of rotation of the convex hull line from point P2 to Pi is clockwise or straight, which means that the hull shape would result in concave i.e. point Pi is to the right to point P2. Thus, we discard or remove point P2 from the stack and again check the orientation of point Pi with respect to the top two points of stack.
- We keep on doing this until the orientation of three points is anti-clockwise. If orientation is anti-clockwise then we add point Pi to the stack
- Orientation of three points - P1, P2 and P3 is calculated by comparing slopes of lines P1P2 and P1P3 given by equation below [16]:

  Slope = (x2-x1)(y3-y1) - (y2-y1)(x3-x1)

  If 0 = Collinear, -ve = Right turn or clockwise, +ve = Left turn or counterclockwise
- Thus, if slope is 0, points are collinear, if slope is negative, orientation from point P2 to P3 is clockwise and if slope is positive then orientation is anti-clockwise.
- We can also say anti-clockwise means left turn from point P2 to P3 and clockwise means right turn from P2 to P3.
- In our case, consider point (3,1) and check orientation with respect to the top two points of stack - (7,0) and (0,0). Since, orientation is anti-clockwise, we add (3,1) to stack as seen in Figure 6 below:

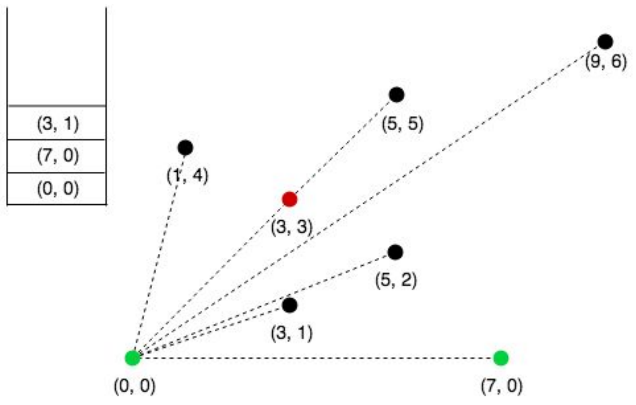


Figure 6: Adding Point (3,1) to stack *(Best Viewed in Color)*

- Similarly, the next point from the list is (5,2) and since orientation from point (3,1) is clockwise we discard (3,1) from the stack. Below figure 7 shows the step described:

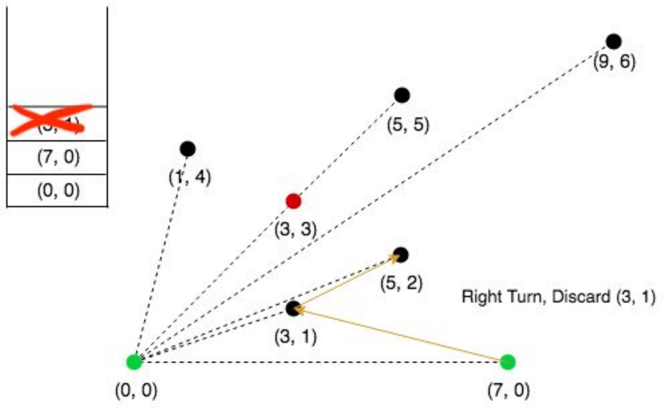


Figure 7: Removing Point (3,1) due to clockwise rotation to Point (5,2) *(Best Viewed in Color)*

- Then, since orientation from (7,0) is anti-clockwise then we add (5,2) to the stack. Below figure shows the step described:

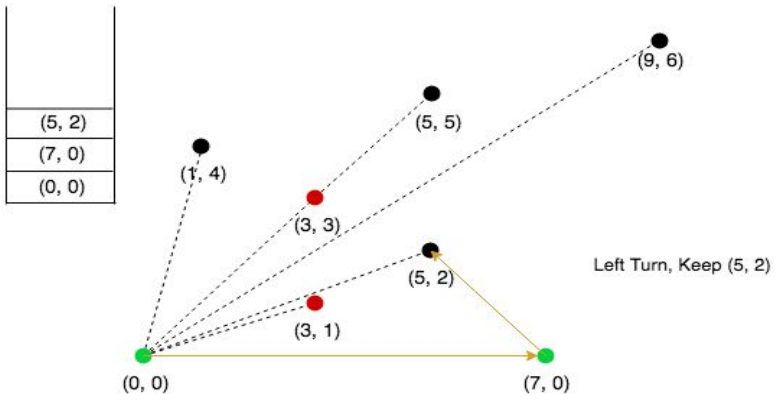


Figure 8: Adding Point (5,2) to stack *(Best Viewed in Color)*

- Now, consider the next point (9,6) and determine the rotation with (5,2) and (7,0) and find that it is clockwise as seen in image below. Hence we discard (5,2) from stack.

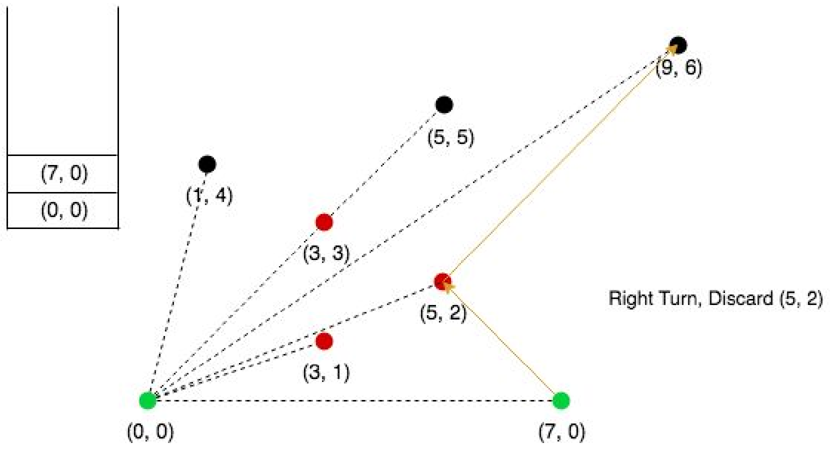


Figure 9: Removing Point (5,2) due to clockwise rotation to Point (9,6) *(Best Viewed in Color)*

- Further, since orientation of (9,6) is anticlockwise with respect to (7,0) as seen in the image below, we add it to stack.

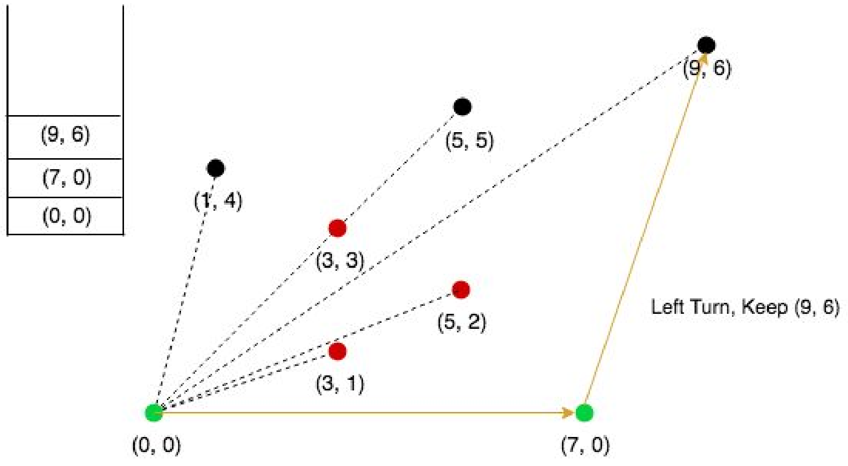


Figure 10: Adding Point (9,6) to stack *(Best Viewed in Color)*

- In the same way, (5,5) is pushed into the stack as seen in image below:

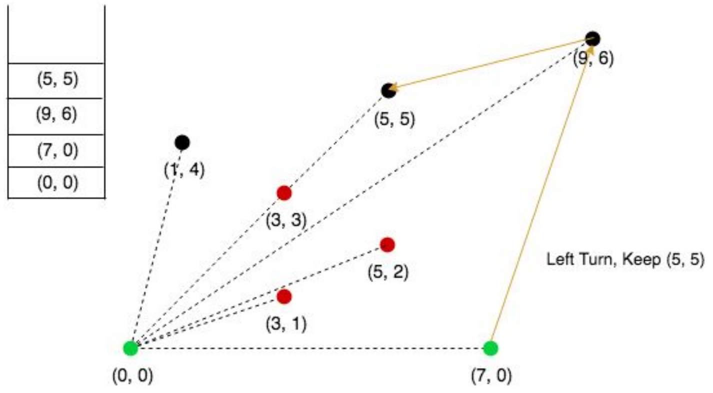


Figure 11: Adding Point (5,5) to stack *(Best Viewed in Color)*

- Next, point (3,3) and since orientation is anti-clockwise from point (5,5) we add it to stack. Next, point (1,4) is to the left of (3,3), we discard (3,3) from stack. Further, point (1,4) is collinear with points (9,6) and (5,5). In the case of collinearity, we discard the top of the stack and hence point (5,5) is popped from the stack.

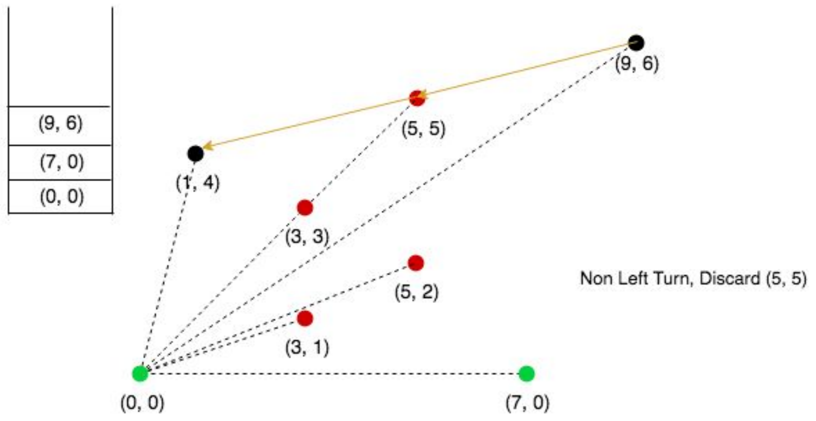


Figure 12: Removing Point (5,5) from stack because of collinearity with (1,4) *(Best Viewed in Color)*

- Since, (1,4) is last point in the list, we add it to the stack

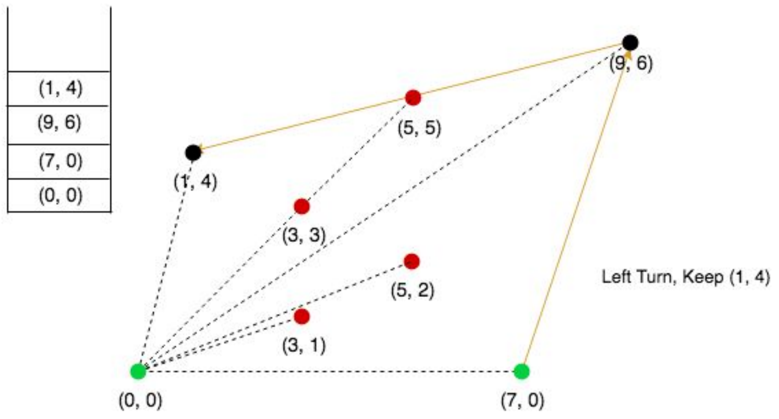


Figure 13: Adding Point (1,4) to the stack *(Best Viewed in Color)*

- Since, there are not points left, all the points on the stack i.e. (0,0), (7,0), (9,6) and (1,4) are the convex hull points as marked green in Figure below:

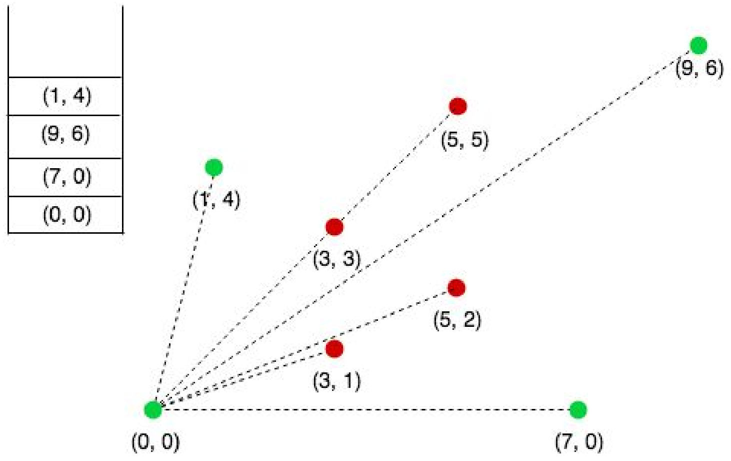


Figure 14: Final Convex Hull Points *(Best Viewed in Color)*

4.3. Pseudocode

Below is the Pseudocode for the Graham's Scan algorithm:

```
Stack s: GrahamScan(Points):
    Create empty stack for convex hull points
    Find the lowest y-coordinate and leftmost point, called anchor point P0
    Sort the points by polar with P0. If same polar angle sort by distance w.r.t P0

    For each point in sorted_points:
        While stack length > 1 and orientation(stack(length-1), stack(length) and point <= 0:
            Remove top element from stack i.e. pop stack
        Add point to the stack i.e. push point in stack

    Return stack
```

4.4. Complexity Analysis

Now since we have already discussed the detailed steps of the Graham's Scan algorithm, we will find out the time and space complexity of the algorithm.

4.4.1. Time Complexity

Below is the time complexity of Graham Scan Algorithm [14]:

- O(n) for finding anchor point
- O(nlogn) for sorting the points
- O(1) constant time for pushing items into the stack
- O(n) each point gets pushed once within the for loop
- O(n) for popping within the loop, each point gets popped once at most

Since the complexity is dominated by the step of sorting the points, the overall time complexity of the algorithm is O(nlogn).

4.4.2. Space Complexity

Below is the space complexity of Graham Scan Algorithm:

- O(n) - Store the points in the list
- O(n) - Store the convex hull points in the stack

Since, the space complexity is the same for both storing points in list and storing convex hull points in the stack, the overall space complexity of the algorithm is O(n).

5. Monotone Chain

5.1. Description

Monotone Chain [17] is an algorithm that finds a convex hull for a finite set of points placed in a 2D plane in linearithmic time. It was published by A. M. Andrew in 1979. This algorithm is very much similar to the Graham Scan algorithm [13] to find convex hulls with some minor differences in terms of execution which we will discuss in the following sections.

5.2. Methodology

Now we will look into detailed working of Graham's Scan. The algorithm works in four main phases:

- Sorting of the given points (x,y coordinates) lexicographically
- Scanning the list of sorted points i.e. from lowest point to highest point to find lower convex hull points
- Scanning the reverse list of sorted points i.e. from highest point to lowest point to find upper convex hull points
- Merge lower and upper convex hulls

Unlike previous algorithms, we won't discuss detailed steps with example points. We will just discuss the execution steps.

Figure 15 shows the final figure of Convex Hull calculated by Monotone's Chain algorithm:

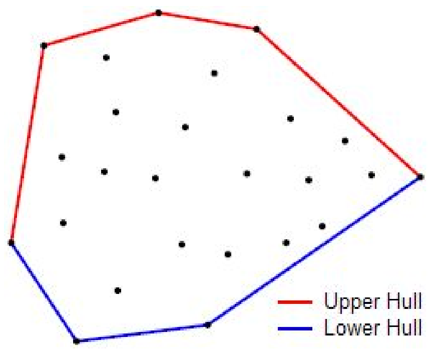


Figure 15: Resultant Convex Hull (Combination of Lower and Upper Convex Hull) *(Best Viewed in Color)*

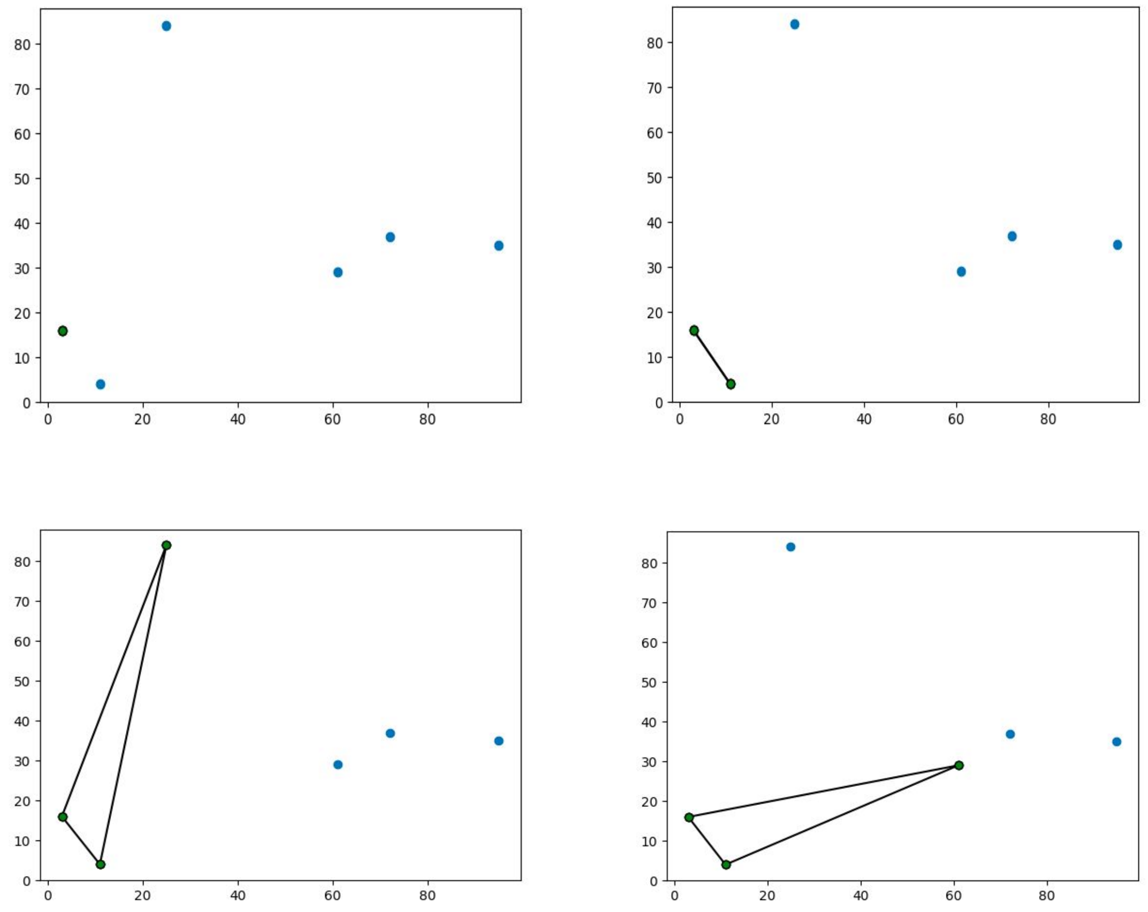
1. *Sorting the points lexicographically*

- This is the first phase of the algorithm and an important one. Like Graham Scan, we sort the points too.
- But the only difference in this step compared to Graham Scan is, we sort all the points lexicographically i.e. by x-coordinate.

- If two points have the same x-coordinate then we sort by y-coordinate.

2. *Finding the lower convex hull points*

- In this phase, we find all the lower convex hull points by the same procedure like Graham Scan where we scan each point of the sorted list to find the convex hull.
- Since the points are lexicographically sorted, we will start with the leftmost point and scan through each point to find convex hull points.
- Again, like the Graham Scan, we will use a lower convex hull stack which will store the results of this step. These results will contain all the lower hull convex points.
- The step to add points to the convex hull is the same as Graham Scan step 3 discussed.
- Below set of figures i.e. Figure 16 shows the execution of calculating lower convex hull points



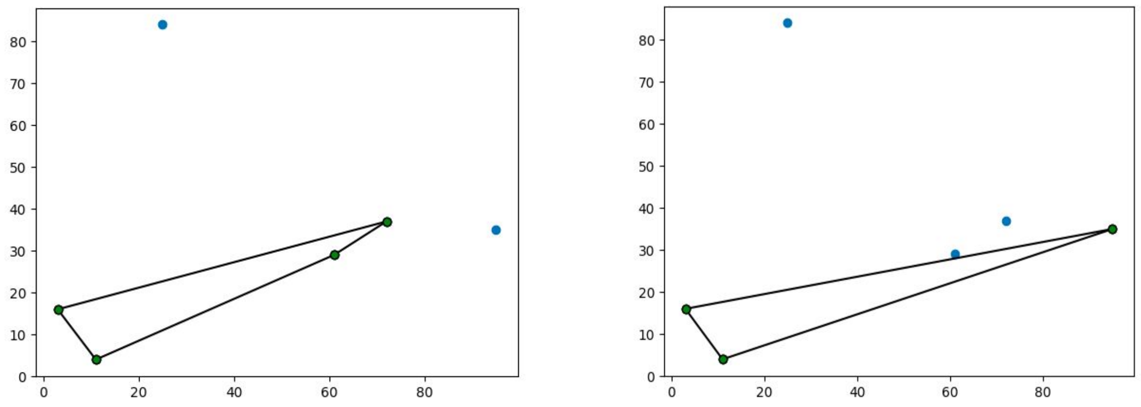
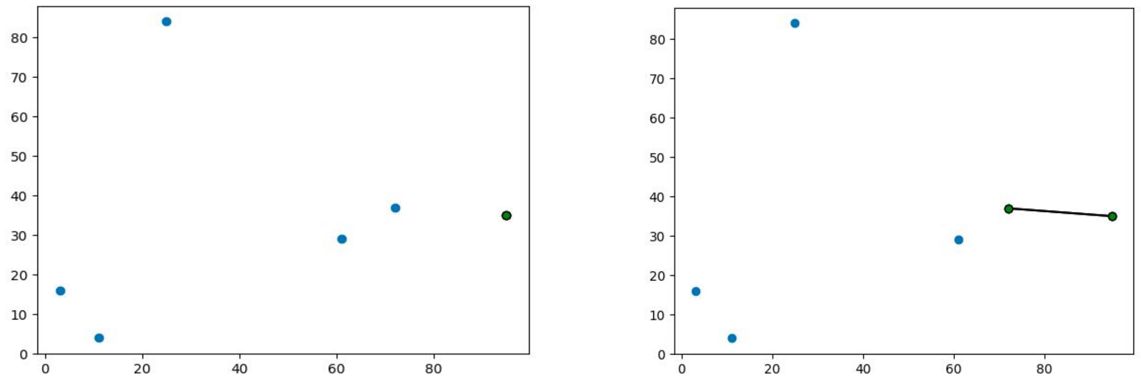


Figure 16: *(Left to Right and Top to Bottom)* Shows execution steps to calculate lower convex hull *(Best Viewed in Color)*

3. *Finding the upper convex hull points*

- This is the same phase as step 2 i.e. above phase discussed. Here we find all the upper convex hull points by the same procedure of scanning each point of the list and then add it to the upper convex hull stack based on orientation discussed in Graham Scan.
- The only difference in this step compared to step 2 is we use the reverse sorted list i.e. we start scanning from the rightmost point (highest point) of the list and then go till the leftmost point.
- Thus, it will result in finding the points present in upper convex hull points.
- Below set of figures i.e. Figure 17 shows the execution of calculating upper convex hull points



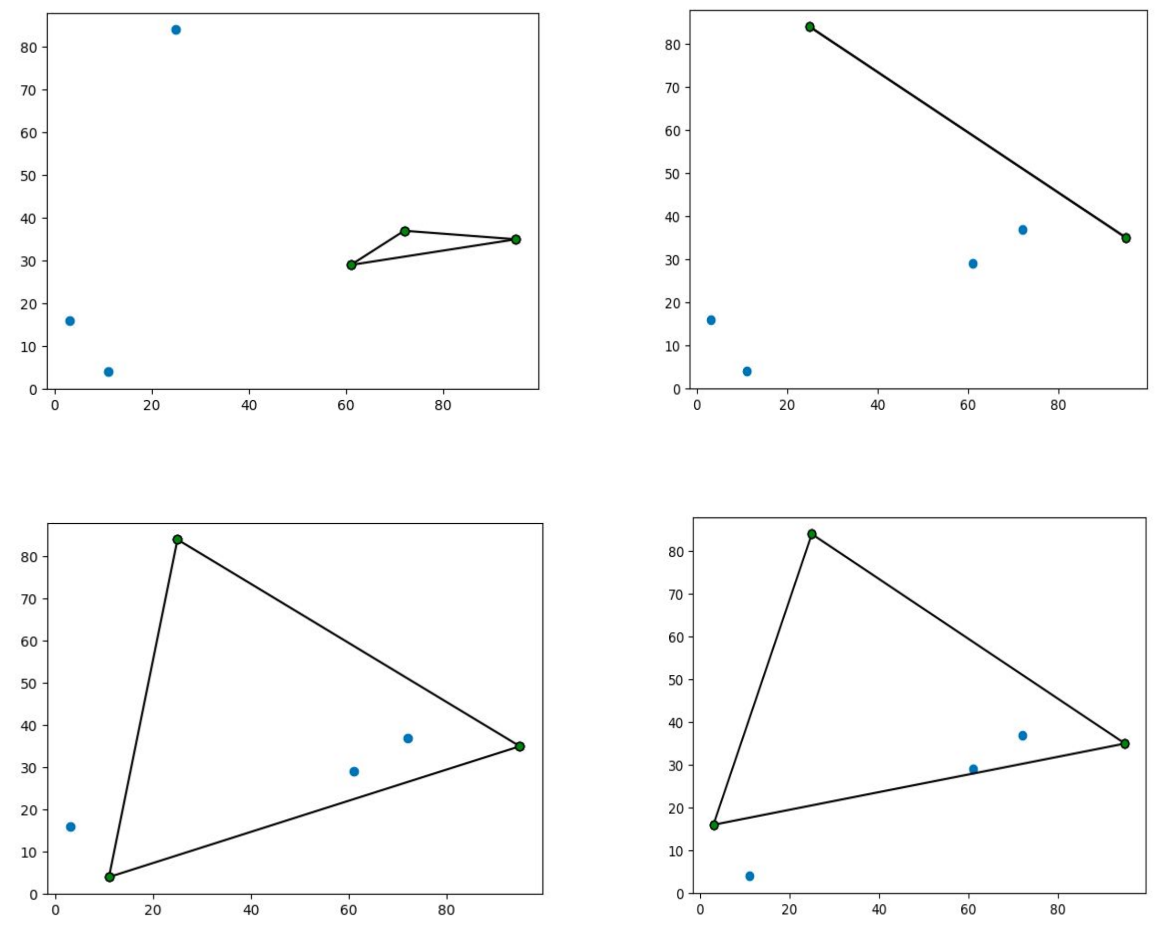


Figure 17: *(Left to Right and Top to Bottom)* Shows execution steps to calculate upper convex hull *(Best Viewed in Color)*

4. *Merging lower and upper convex hull*

- Since, we have calculated lower and upper convex hull points in step 2 and 3 discussed above, next is to merge both the lists to find the entire convex hull points.
- The merging in simple terms is appending upper (or lower) convex hull points to lower (or upper convex hull points)
- Below is the final figure 18 of the convex hull after merging lower and upper convex hulls shown in step 2 and 3

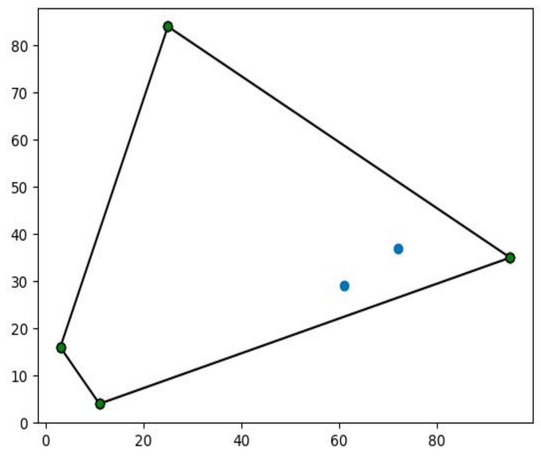


Figure 18: Resulting Convex Hull (Combination of Lower and Upper Convex Hull) *(Best Viewed in Color)*

5.3. Pseudocode

Below is the Pseudocode for the Monotone's Chain algorithm:

```
List L: MonotoneChain(Points):
        Create empty stacks for storing lower and upper convex hull points
        Create empty List for storing all the convex hull points
        Sort the points lexicographically i.e. by x-coordinate (sort by lowest y-coordinate if tie)

        # Finding Lower Convex Hull
        For each point in sorted_points:
            While stack length > 1 and orientation(stack(length-1), stack(length) and point <= 0:
                Remove top element from lower convex hull stack i.e. pop stack
            Add point to the lower convex hull stack i.e. push point in stack

        # Finding Upper Convex Hull
        For each point in reverse(sorted_points):
            While stack length > 1 and orientation(stack(length-1), stack(length) and point <= 0:
                Remove top element from upper convex hull stack i.e. pop stack
            Add point to the upper convex hull stack i.e. push point in stack

        # Merging Lower and Upper Convex Hull
        Append points of lower convex hull stack in List
        Append points of upper convex hull stack in List
        Return List
```

5.4. Complexity Analysis

5.4.1. Time Complexity

Below is the time complexity of Monotone's Chain Algorithm:

- O(nlogn) for sorting the points lexicographically
- Combined complexity for lower and upper hull:
    - 2*O(1) constant time for pushing items into stack
    - 2*O(n) each point gets pushed once within the for loop
    - 2*O(n) for popping within the loop, each point gets popped once at most
- O(n) to merge lower and upper convex hull

Since the complexity is dominated by the step of sorting the points, the overall time complexity of the algorithm is O(nlogn).

5.4.2. Space Complexity

Below is the space complexity of Monotone's Chain Algorithm:

- O(n) - Store the points in the list
- O(n/2) + O(n/2) - Store the lower and upper convex hull points in the stack
- O(n) - Store the convex hull points in list after merging lower and upper convex hull points

Since, the space complexity is the same for both storing points in list and storing convex hull points in the stack, the overall space complexity of the algorithm is O(n).

6. Experimental Results

6.1. Result Table

Below is the result table for comparative analysis between algorithms discussed:

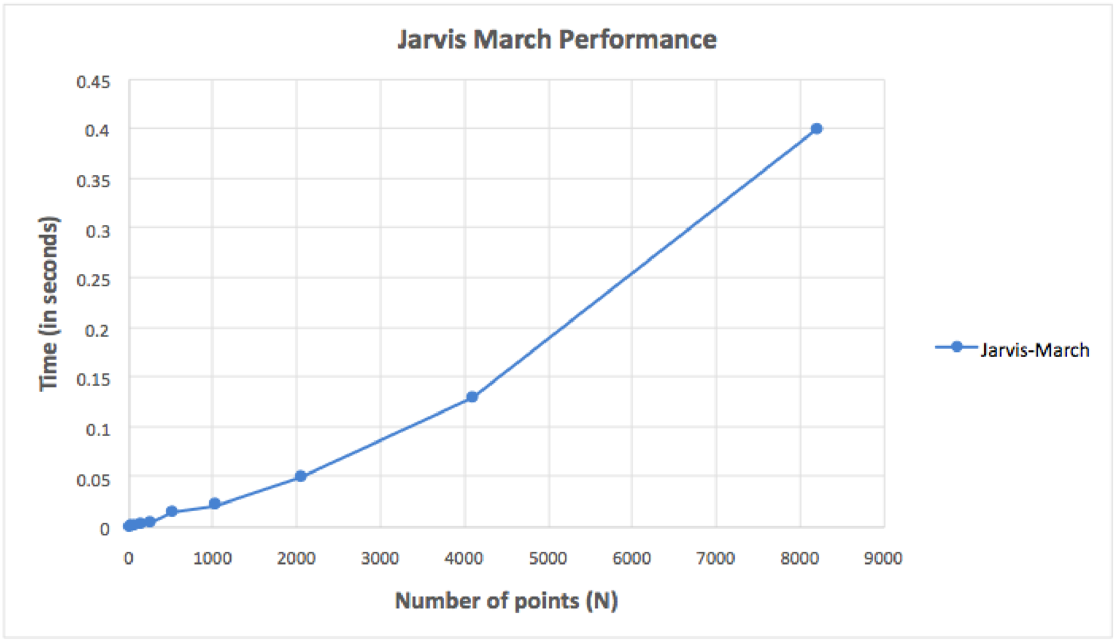
N	**Jarvis March (s)**	**QuickHull (s)**	**Graham Scan (s)**	**Monotone Chain (s)**
8	0.0001	0.0007	0.0001	0.00003
16	0.0016	0.001	0.0001	0.00005
32	0.0004	0.0038	0.0002	0.0001
64	0.001	0.005	0.0006	0.0002
128	0.002	0.012	0.0014	0.0004
256	0.0045	0.025	0.002	0.0009
512	0.015	0.04	0.005	0.002
1024	0.022	0.09	0.018	0.004
2048	0.05	0.15	0.03	0.008
4096	0.13	0.37	0.06	0.018
8192	0.4	0.78	0.16	0.035

Table 1: Size N vs Time (T in secs) Table for the performance comparison between algorithms

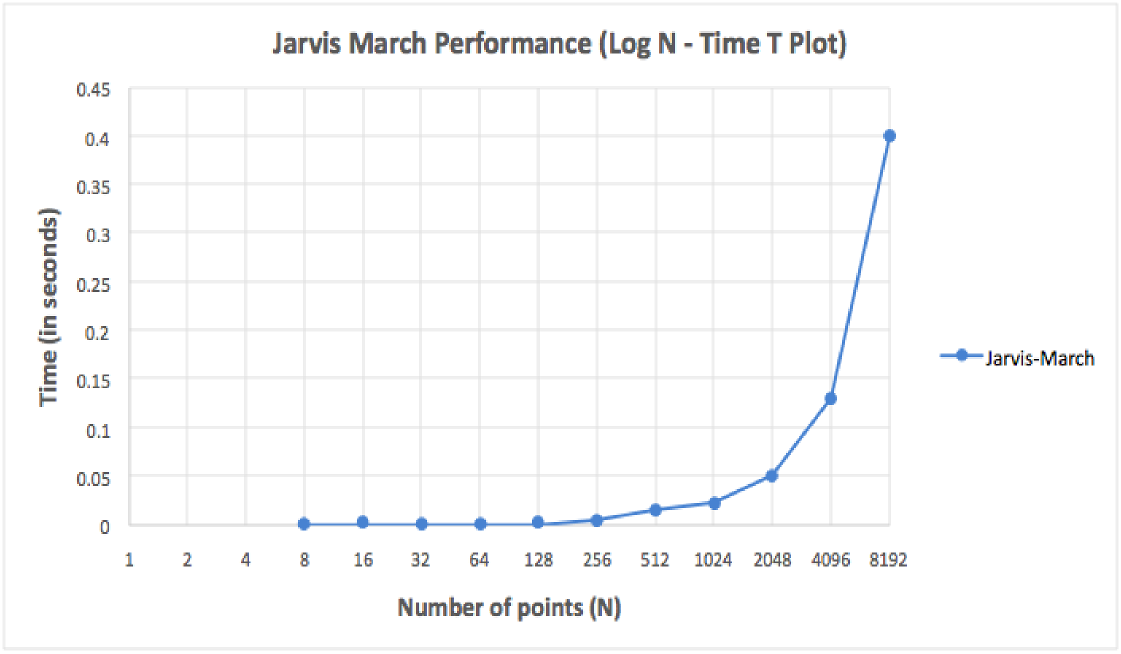
6.2. Performance Plots

6.2.1. Jarvis March

1. Size N Vs Time T Plot

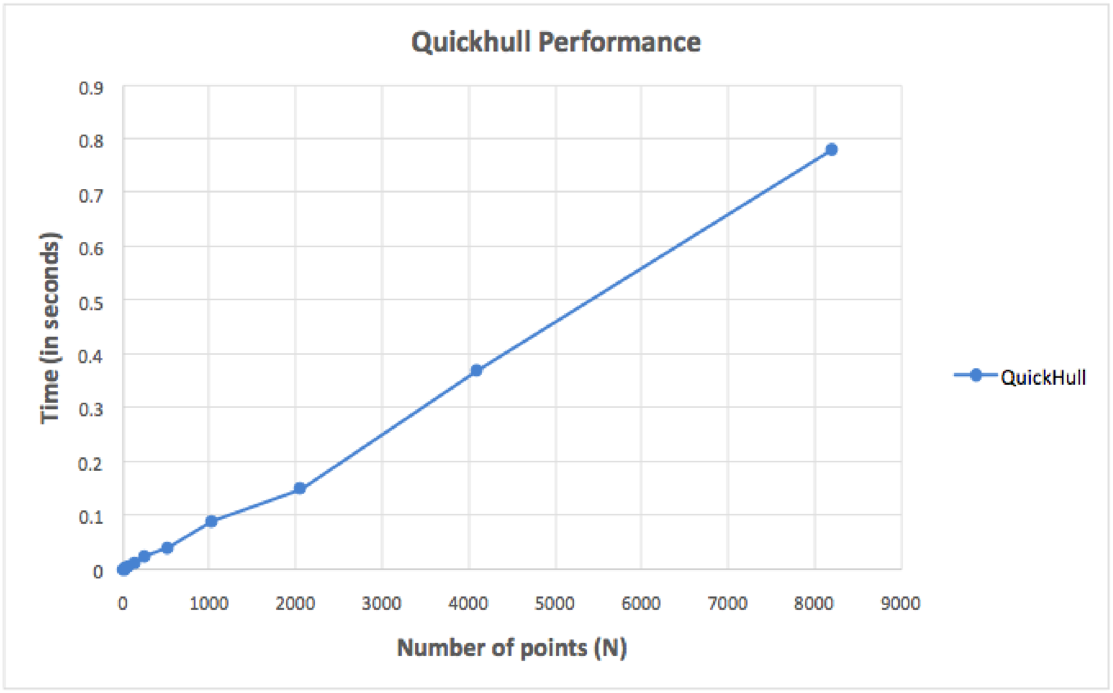


2. Log N vs Time T Plot

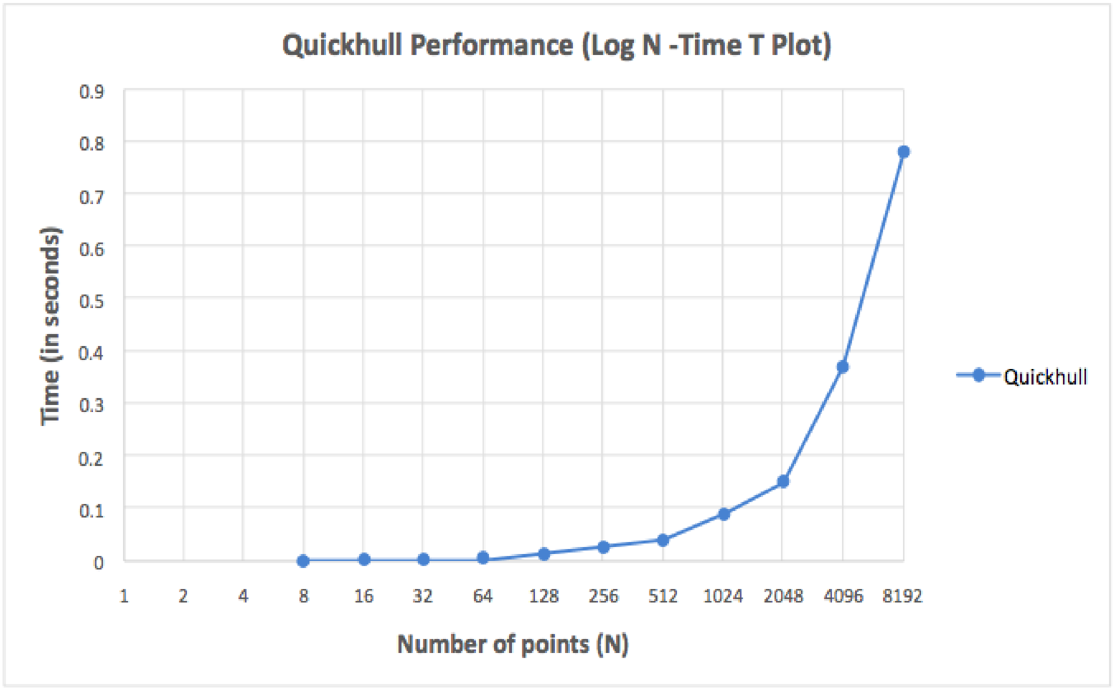


6.2.2. Quickhull

1. Size N Vs Time T Plot

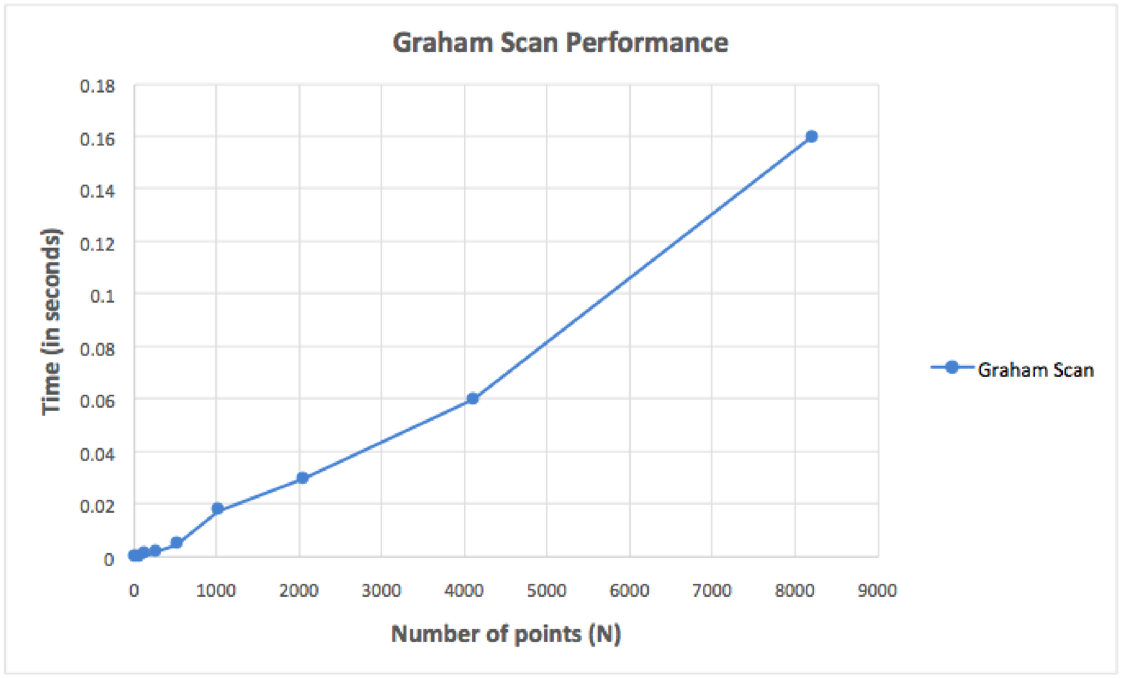


2. Log N vs Time T

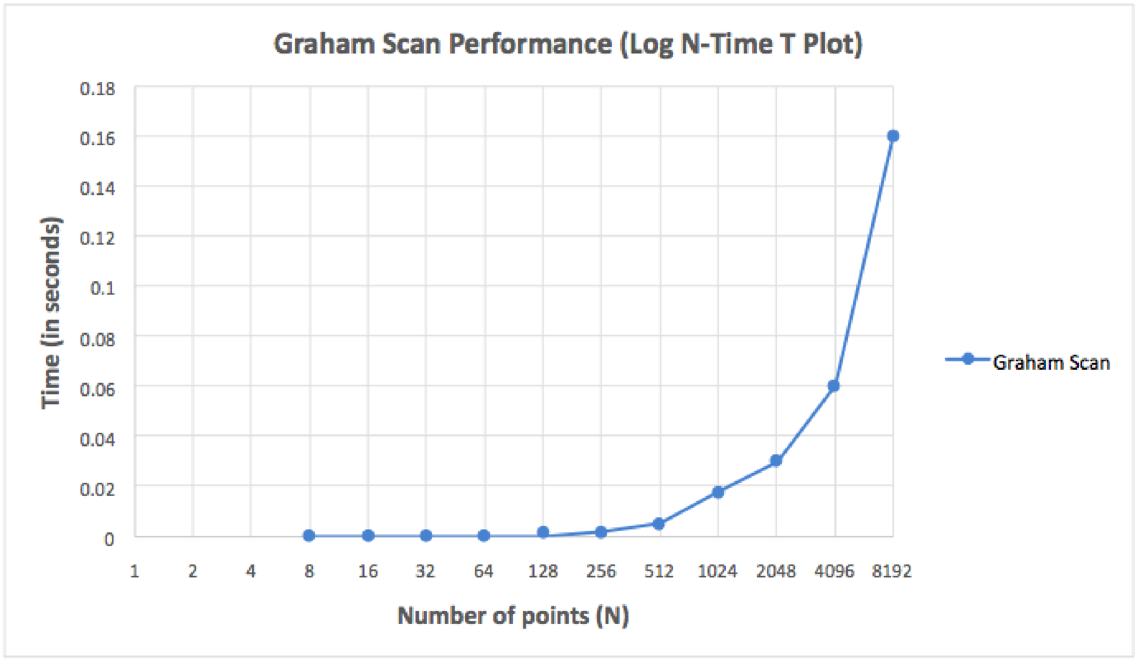


6.2.3. Graham Scan

1. Size N Vs Time T Plot

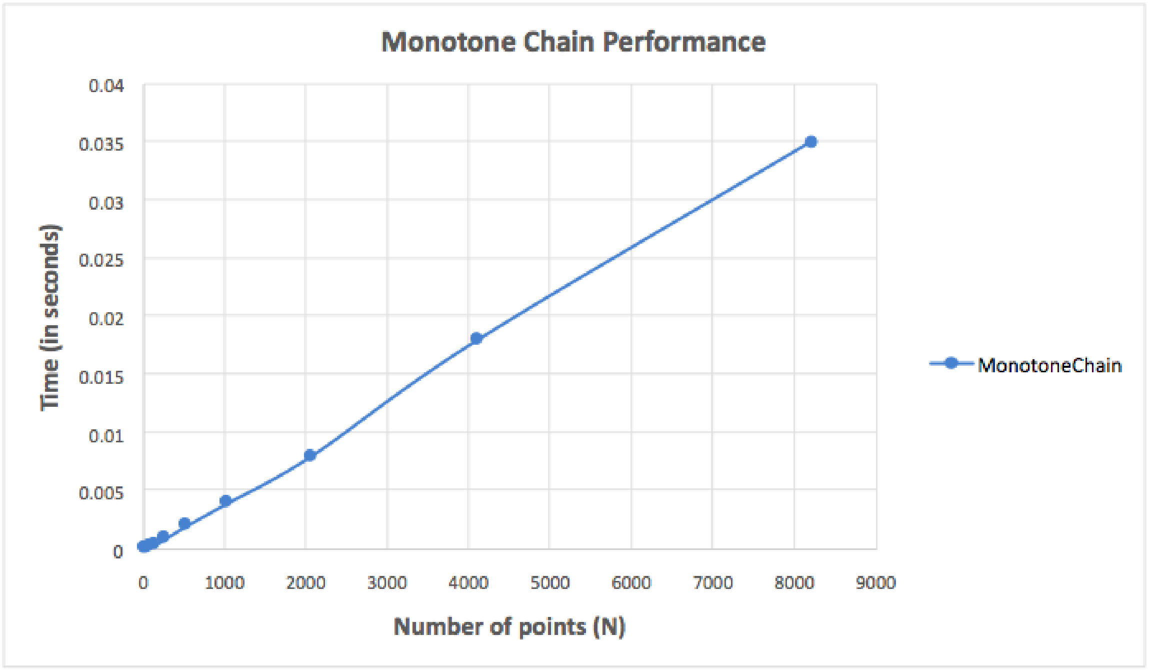


2. Log N vs Time T Plot

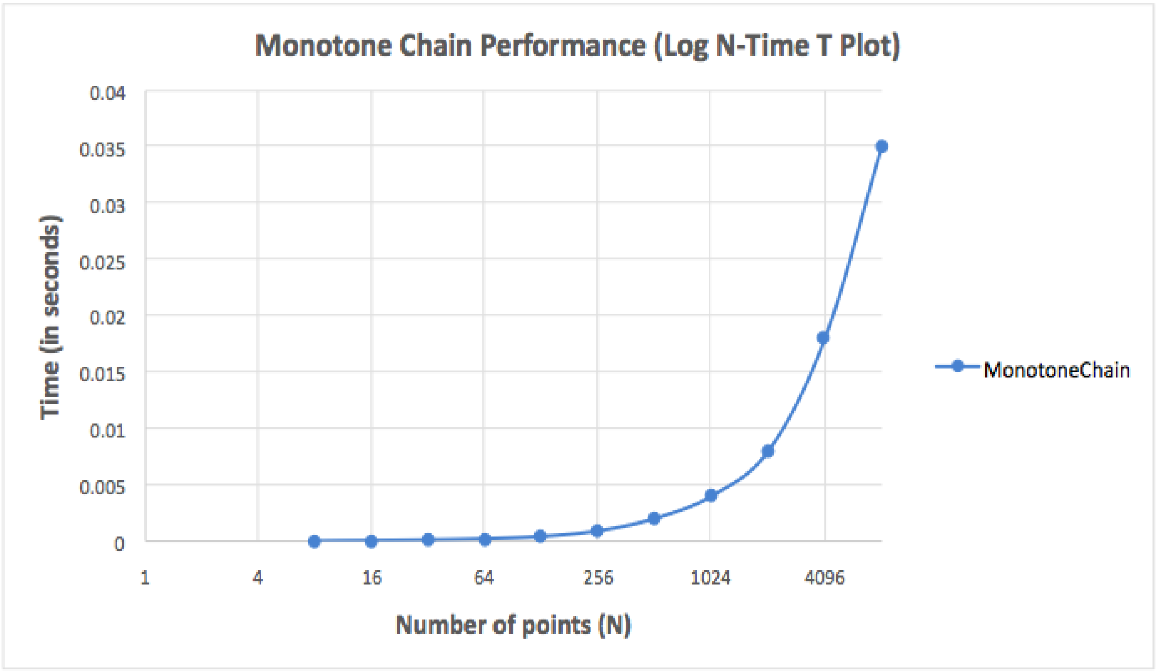


6.2.4. Monotone Chain

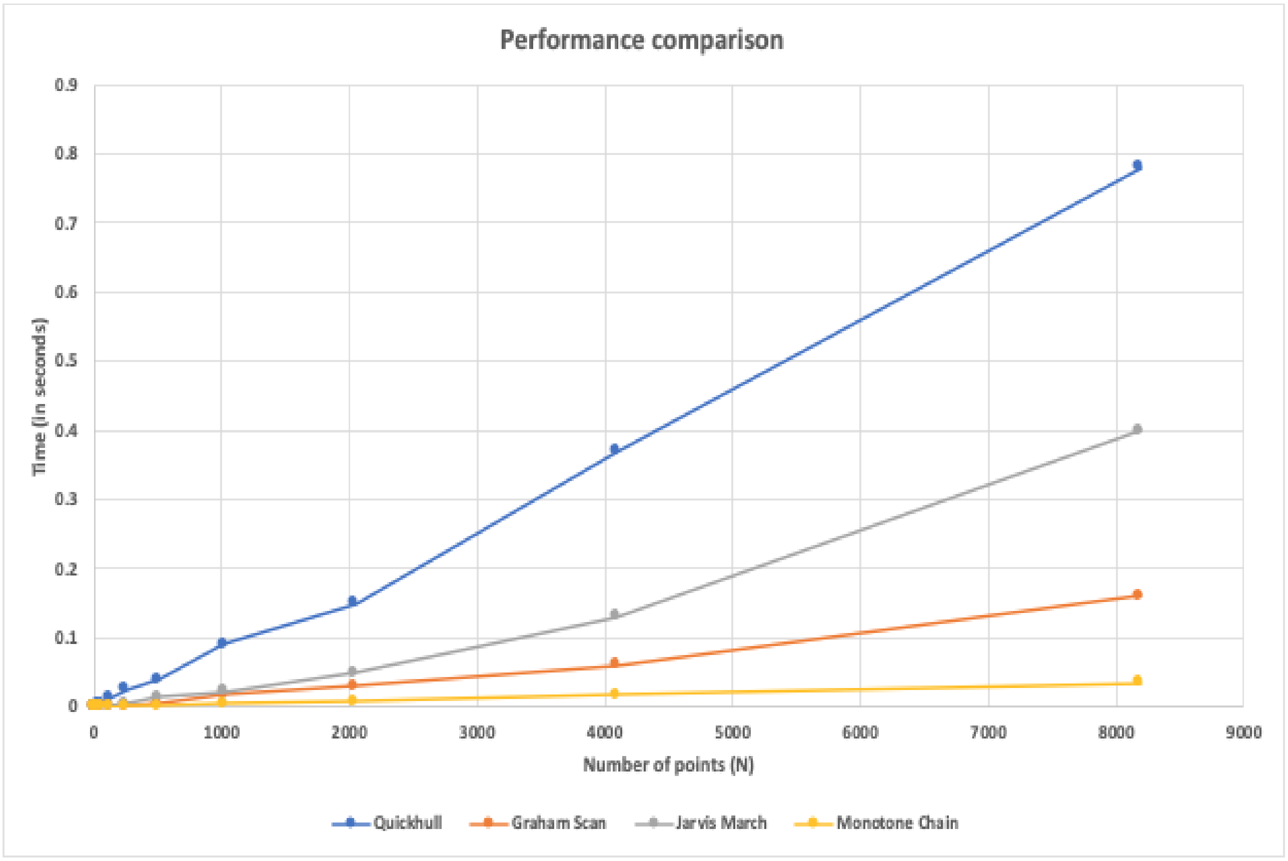
1. Size N Vs Time T Plot



2. Log N vs Time T Plot



6.2.5. Plot of Performance Comparison between Convex Hull Algorithms



6.3. Results Discussion

First when we look at the Graham scan and the Monotone chain algorithm, they are using the same methodology wherein the points are sorted first . Then, we choose the leftmost point and then we move around the points to check the rotations the coming points are causing and based on that we include them as a part of the convex hull or exclude them. Both have complexity in the order of N Log N.

As seen the monotone chain algorithm seems to be faster in comparison to Graham scan code. The plausible reason is that in Graham scan we are sorting the points in terms of the polar angles. So we need to do the extra calculation of calculating the polar angles for each point in case of Graham Scan but in case of Monotone Chain we simply sort the points based on their coordinates. So it saves us with some computations hence Monotone Chain is performing better than Graham Scan though algorithmically both are of the same order and quite similar.

Further as we see that the Jarvis March is an output sensitive algorithm . In the Jarvis March algorithm ,for each point on the convex hull starting from the left most , we move through all the points and find the most minimal or maximal angle relative to itself which is the next point in the Convex Hull. That is why the complexity of the algorithm is H (Points in the Convex Hull) * N(Total Points). In general when Data set is Random then it shows the performance of N Log N where h is in the order of Log N but can go to N i.e. the order of the an algorithm can be N^2 which is not the case with Monotone Chain and Graham Scan where the worst to best case is in the order of N Log N . From the time comparison data of various algorithms it can be clearly seen that both Graham Scan and Monotone Chain are performing better than the Gift wrapping algorithm which was as expected. We can also observe that N Vs T Plot for the Jarvis March is showing an almost linearithmic curve which is because of the randomness of the dataset wherein the H tends to be Log N.

Further it is being noticed that the QuickHull algorithm which was supposed to be the fastest has not done well in our data set. In view of this , we performed analysis of all the steps being performed in the implementation of the Quickhull algorithm to see whether they may be affecting the time performance of the whole algorithm . But after careful analysis and studying from various sources , plausible reason for such result was that when we say an algorithm to be order of some function g , then we are using the Big-O Asymptotic Notation which gives us the Upper Bound Idea (f = O(g) where f <= c times g) so actually the time complexity is represented by c times g and this c can be seen as an overhead in terms of amount of comparisons , memory allocations and deallocations etc. So by big O notation f gives an idea of order of growth. In our case , other algorithms except QuickHull are non recursive in nature But QuickHull is a recursive algorithm and recursion leads to various overheads associated with repetitive function calls and returns [18]. (Note: When a function is called , the memory is allocated to it on the stack and a recursive function calls itself where the memory for a called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call). In view of this, we can infer that this time performance of QuickHull is due to these overheads in its implementation.

6.4. Machine Specifications

Below are the machine specifications used for the performance analysis of the convex hull algorithms discussed in previous sections:

Machine used: MacBook Air (Retina, 13-inch, 2019)
Processor - 1.6 Ghz intel Core i5
Memory - 8 GB 2133 MHz LPDDR3
Graphics - Intel UHD Graphics 617 1563 MB
Operating System - MacOS 10

7. Future Scope and Applications

Convex hulls are predominantly used on 2-d datasets. Since most LIDAR datasets include clouds of 3-D points, we can figure out efficient ways to run convex hull on higher dimensional data, either by projecting it into 2-D planes or by innovating newer and better algorithms.

8. Conclusion

In this report, we discussed a couple of problem domains which involve the field of computational geometry. We then discussed what is convex hull and its importance in the mentioned problem domains. There are four algorithms which we introduced in this project report that can be used to solve the problems discussed by finding convex hulls. Also, we discussed their detailed description, methodology, pseudocode and complexity analysis for each of the algorithms introduced. Further, we discussed and analysed the experimental results of the experiments performed by implementing the algorithms and running it on different datasets. Lastly, we discussed the future scope and applications in the studies of convex hull algorithms.

9. References

1. https://en.wikipedia.org/wiki/Convex_hull
2. https://brilliant.org/wiki/convex-hull/
3. https://en.wikipedia.org/wiki/Collision_avoidance_system
4. https://digital-library.theiet.org/content/journals/10.1049/iet-ipr.2017.0138
5. http://jeffe.cs.illinois.edu/teaching/compgeom/notes/01-convexhull.pdf
6. https://algorithmtutor.com/Computational-Geometry/Convex-Hull-Algorithms-Jarvis-s-March/
7. http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/ConvexHull/jarvisMarch.htm
8. https://en.wikipedia.org/wiki/Quickhull
9. https://iq.opengenus.org/quick-hull-convex-hull/
10. https://www.cise.ufl.edu/~ungor/courses/fall06/papers/QuickHull.pdf
11. https://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/JarvisMarch.html
12. https://en.wikipedia.org/wiki/Gift_wrapping_algorithm
13. https://en.wikipedia.org/wiki/Graham_scan
14. https://algorithmtutor.com/Computational-Geometry/Convex-Hull-Algorithms-Graham-Scan/
15. https://www.geeksforgeeks.org/atan2-function-python/
16. https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
17. https://algorithmist.com/wiki/Monotone_chain_convex_hull
18. https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/

10. Work Distribution

(Overall contribution of all the team members was equal)

Below are the names of team members and their major responsibilities:

Team Member	Major Responsibilities
Pratik Mistry	● Implementation and research on Graham Scan Algorithm and Monotone Chain Algorithm ● Project presentation ● Reporting writing and complete formatting ● Dataset Generation
Shounak Rangwala	● Implementation and research on Jarvis March and Quickhull algorithms ● Project presentation ● Report writing and formatting
Vikhyat Dhamija	● Implementation and research on Monotone Chain Algorithm ● Performance analysis between algorithms ● Report Writing
Pranit Ghag	● Research on convex hull applications in industry and running performance tests for comparing the 4 algorithms ● Literature writing in project report