Bilkent University

CS-478

Computational Geometry

Spring 2023/2024

Project: Convex Hull Algorithms

Final Report

Erkin Aydın

22002956

1) Layout of the Report

I implemented 4 convex hull algorithms:

- Graham's Scan
- Jarvis March
- Quickhull
- Mergehull

I will go over the pseudo codes and explanations of convex hull algorithms. Later, certain implementation details for these algorithms will be mentioned. Then I will discuss the runtime benchmarks of these algorithms. Lastly, I will provide details for the application implementation and a guide for visualization.

2) Algorithms to be Used & Their Explanations

2.1) Graham's Scan

Graham's Scan can be used to find the convex hull of a finite set of points in a 2D space. It's time complexity is *O(nlogn)* and its space complexity is *O(n)*, where *n* is the number of points. It is a vertex-based convex hull algorithm, in which the relation between vertices are the primary concern and use in constructing the convex hull.

The main idea behind the Graham's Scan is selecting a good pivot point which is for sure internal to the convex hull of the set of points, sorting other points with respect to the polar angles they make with the selected pivot, and using this information to construct the convex hull.

Graham's Scan Algorithm consists of two stages: preparation stage and scan stage. I will focus on each stage one by one.

Preparation Stage of Graham's Scan

In the preparation stage, we need to select a good pivot point and sort other points with respect to the polar angles and distance from the point and then give a data structure for the scan stage.

A common approach for selecting a pivot point is selecting an extreme point that is for sure a point in the convex hull. Such an extreme point can be selected for looking the smallest or largest x and y coordinates. I prefer to select the point with the smallest y ordinate, and in case there are multiple points with that smallest y ordinate, choose the one with the smallest x ordinate. This can be done in *O(n)* time, where n is the number of points in the finite set of points.

The next step of the preparation stage is to transform the coordinates of other points so that the selected pivot point will be the origin. This can be done in *O(n)* time.

Then, the rest of the points will be sorted with respect to the angle they make with the pivot point and the x-axis of the new coordinate system. Where there are multiple point making the same polar angle, they should be sorted with respect to their distance from the origin. This can be done in *O(nlogn)* time.

Lastly, we need to convert these sorted points into a doubly-linked circular list, which can be done in *O(n)* time with a simple traversal of the sorted points.

Therefore, due to sorting points with respect to the polar angles they make and the distance from the origin, the preparation stage of the Graham's Scan Algorithm takes *O(nlogn)* time.

Here is the pseudo code for the preparation stage:

```
PREPARATION_GRAHAMS_SCAN(S)
1:  p_min_y := NULL
2:  min_y := INTEGER_MAX
3:  for each point p in S do
4:  |   if p.y < min_y then
5:  |   |       p_min_y := p
6:  |   |       min_y := p.y
7:  |   else if p.y = min_y and p.x < p_min_y.x then
8:  |   |       p_min_y := p
9:  negative_min_x := p_min_y.x
10: negative_min_y := p_min_y.y
11: for each point p in S do
12: |   p.x := p.x + negative_min_x
13: |   p.y := p.y + negative_min_y
14: sorted_points_list := Sort points lexicographically first with respect to the polar angles they
                          make, then by the distance to the origin, that is, p_min_y
15: sorted_dlcl := By a traversal of sorted_points_list, add the sorted points to the sorted_dlcl
16: return p_min_y, sorted_dlcl
```

Scan Stage of Graham's Scan

In the scan stage, we will use the p_min_y calculated in the preparation stage as the origin point, and then traverse the doubly-linked circular list to construct the convex hull. One thing to note here is that all the remaining points should be on the LEFT side of two consecutive points of the convex hull. If not, that is, at some point if we find a point on the RIGHT side of two consecutive points of the convex hull, then we need to back trace the constructed convex hull to delete the points causing this mistake.

In the scan stage, since no point can be visited more than twice, the time complexity is *O(n)*

Here is the pseudo code for the scan stage, a slightly changed version of the one given by Preparata:

```
SCAN_GRAHAMS_SCAN(p_min_y, sorted_dlcl)
1:  v := p_min_y := START
2:  w :=sorted_dlcl.PRED[v]
3:  f = FALSE
4:  while sorted_dlcl .NEXT[V] ≠ START or f = FALSE do
5:  |   if sorted_dlcl .NEXT[V] = w then
6:  |   |       f = TRUE
7:  |   if Left(v, sorted_dlcl .NEXT[V], sorted_dlcl .NEXT[sorted_dlcl .NEXT[V]]) then
8:  |   |       v = sorted_dlcl .NEXT[V]
9:  |   else then
10: |   |       delete sorted_dlcl .NEXT[V]
11: |   |       v := sorted_dlcl .PRED[V]
```

At the end of this scan, sorted_dlcl contains the convex hull.

2.2) Jarvis March

Jarvis March is a 2D specialization of the Gift Wrapping algorithm, in which the convex hull is found for higher dimensions. Just like Graham's Scan, Jarvis March can be used to find the convex hull of a finite set of points in a 2D space. It's time complexity is $O(n^2)$ while the space complexity is the same with Graham's Scan, $O(n)$. While it may seem like the Jarvis March algorithm runs slower than Graham's Scan, in the average case Jarvis March is actually faster.

Let h be the number of points in the convex hull of a finite set of points. Then in the Jarvis March algorithm only $O(hn)$ comparisons are necessary. Therefore the expected runtime for Jarvis March is $O(hn)$, and therefore Jarvis March Algorithm performs better than Graham's Scan Algorithm when $h < logn$.

Jarvis March Algorithm can be divided into two stages: Marching Up and Marching Down.

Marching Up
First, we use a similar logic used in Graham's Scan for finding the starting point of our algorithm: We find the points with smallest y coordinates, but this time instead of choosing the one with the smallest x coordinate of them, we choose the one with the greatest x coordinate of them. This starting point is in the hull for sure.

Then, we need to find the next point of the hull one by one. Each next point of the hull will make the smallest polar angle with the current point and the x-axis. In each step, finding the next point can be made in $O(n)$ time and there can be $O(n)$ steps, therefore marching up takes $O(n^2)$.

We can stop marching up when we reach to a point having the largest y coordinate.

Marching Down
Everything is the same as in marching up, but this time, the negative x-axis will be used in calculating polar angles. In each step, finding the next point can be made in $O(n)$ time and there can be $O(n)$ steps, therefore marching down also takes $O(n^2)$.

We can stop marching down when we reach to the starting point of the Jarvis March.

Here, I give the pseudo-code that I will use for Jarvis March Algorithm:

```
JARVIS_MARCH(S)
1:  p_min_y := NULL
2:  min_y := INTEGER_MAX
3:  max_y := INTEGER_MIN
4:  for each point p in S do
5:  |   if p.y < min_y then
6:  |   |       p_min_y := p
7:  |   |       min_y := p.y
8:  |   else if p.y = min_y and p.x > p_min_y.x then
9:  |   |       p_min_y := p
10: |   if p.y > max_y then
11: |   |       max_y := p.y
12: current_point := p_min_y
13: axis_to_check := x-axis
14: prev_angle := 0
15: prev_point := NULL
```

```
16: convex_hull := {}
17: repeat
18: |    smallest_angle := 360
19: |    next_point := NULL
20: |    for each point p of S do
21: |    |        if angle made by p, current_point and axis_to_check < smallest_angle then
22: |    |        |        next_point := p
23: |    |        |        smallest_angle := angle made by p, current_point and axis_to_check
24: |    if smallest_angle = prev_angle then
25: |    |        convex_hull := convex_hull / prev_point
26: |    convex_hull := convex_hull ∪ next_point
27: |    prev_point := next_point
28: |    prev_angle := next_angle
29: |    if next_point.y = max_y then
30: |    |        axis_to_check := negative x-axis
31: until current_point = p_min_y
32: return convex_hull
```

2.3) Quickhull

Quickhull is inspired by the Quicksort algorithm. It recursively partitions the points to find a convex hull for each subset. Then, concatenates the convex hull of the subsets to construct the convex hull of finite set of points.

Quickhull divides the finite set of points into two subsets, $S^{(1)}$ and $S^{(2)}$, by a line L. This line L is the line passing from the points with smallest and largest abscissa, l and r respectively. One thing to note here is that l and r points are the intersection of $S^{(1)}$ and $S^{(2)}$, therefore L does not partition the finite set of points. Therefore, to avoid this duplicate points in sets which will be duplicate in the final convex hull, in the initial call we use the smallest abscissa, where r is slightly lower in the y coordinate than l by a constant. Therefore, initial call is as follows:

```
1:  l0 := (x0, y0) // The point with smallest abscissa
2:  r0 := (x0, y0 – ε) // ε is an arbitrarily small constant
3:  convex_hull := QUICKHULL(S, l0, r0) / r0
```

Then, for both $S^{(1)}$ and $S^{(2)}$, we will find an apex point, that is a point furthest from the line L. Apex point will make the triangle with the greatest area with l and r points. Later on, this apex point will be used to eliminate the points that are for sure not in the convex hull, that are the points internat to the triangle the apex point made with l and r points. This process will continue recursively to the points not in the triangle.

While the worst case time complexity of Quickhull is $O(n^2)$, its expected time complexity is $O(nlogn)$, just like in Quicksort. The worst case appears when after elimination, all the points remain on one side. This is possible, but unlikely. Space complexity of Quickhull is $O(n^2)$.

Here, I provide a pseudo code for my future implementation of Quickhull (slightly modified from Course Slides):

```
QUICKHULL(S, l, r)
1:  if S = {l, r} then
2:  |    return {l, r}
3:  else
4:  |    h := FURTHEST(S, l, r)
```

5: | $S^{(1)}$:= Set of points in S that are on or left of line lh
6: | $S^{(2)}$:= Set of points in S that are on or left of line hr
7: | return (QUICKHULL($S^{(1)}$, l, h) || QUICKHULL($S^{(2)}$, h, r)) / h

2.4) Merge Hull

Merge Hull is inspired by Merge Sort. It consists of three stages: 1) Partitioning the finite set of points into two subsets S_1 and S_2 of approximately equal size, 2) Recursively find the convex hulls of S_1 and S_2, 3) Merge two hulls to construct the convex hull. When there are a small number of points in a set, we can simply return the set. Of course, this assumes the points in such a small set construct a valid convex hull for that small set. One other thing to note here is that even if it does not construct a valid convex hull for input set, we will eventually apply Graham's Scan in the merge step to these points, which does handle collinear points.

Merge step is the tricky step in Merge Hull, as areas covered by convex hulls may not be disjoint. For this, we can carry the "checking by angle" approach from previous algorithms and use here. We can select a point internal to one of the convex hulls produced, such as the centroid of that hull. Then we can check whether this point is internal to the other convex hull or not. If so, then this means the points of both convex hulls are in sorted angular order around the selected point, therefore we can merge the lists of points of both convex hulls in linear time to construct the convex hull. If not, then we can construct the supporting lines from the selected point to the external convex hull and simply discard the chain between the vertices of the supporting lines.

With this approach, we can eliminate points internal to both of the convex hulls. Then, applying Graham's Scan to the remaining points gives us the convex hull. Using Graham's Scan is necessary here, as Mergehull algorithm runs in *O(nlogn)* time, due to dividing the finite set of points into two subset of points of approximately the same size.

Here, I give pseudo code for my future Merge Hull implementation:

MERGE_HULL(S)
1: if |S| < 4 (any small integer would suffice as well) then
2: | return S
3: Construct S_1 and S_2 where $S_1 \cup S_2 = S$ and $|S_1| \approx |S_2|$
4: $H(S_1)$:= MERGE_HULL(S_1)
5: $H(S_2)$:= MERGE_HULL(S_2)
6: H(S) := MERGE($H(S_1)$, $H(S_2)$)
7: return H(S)

MERGE($H(S_1)$, $H(S_2)$)
1: p1 := any point internal to $H(S_1)$
2: sorted_list := {}
3: if p1 is internal to $H(S_2)$ then
4: | sorted_list := Merge $H(S_1)$ and $H(S_2)$ in linear time (as they are in sorted angular order)
5: else
6: | s1 := supporting line 1 between p and $H(S_2)$
7: | s2 := supporting line 2 between p and $H(S_2)$
8: | $H(S_2)$:= $H(S_2)$ / {monotone chain between s1 and s2}
9: | sorted_list := Merge $H(S_1)$ and $H(S_2)$ in linear time (as they are in sorted angular order **now**)
10: return GRAHAMS_SCAN(sorted_list)

3) Certain Implementation Details for Algorithms

- Python has been used for this application. Python is not good when it comes to loops. Hence I tried to avoid using loops as much as I can. For instance, first 8 lines of PREPARATION_GRAHAMS_SCAN algorithm has been implemented without python loops as follows: p_min_y = min(points_copy, key=lambda p: (p[1], p[0])). Whenever I can avoid Python loops I avoid them and made sure to follow the same strategy in other algorithms too, simply to preserve the runtime relations between algorithms.

- For each algorithm, there exists three versions implemented, matching the visualization modes. “performance” versions of the algorithms are called in “Do NOT visualize” mode, as well as in benchmarking. “without_delay” versions of the algorithms are called in “Visualize without delay” mode. “with_delay” versions of the algorithms are called in “Visualize with delay” mode.

- “initial_call” methods has been implemented for all algorithms. These methods create a copy of points. This is done to ensure the algorithms do not remove any point from the canvas accidentally. These “initial_call” methods are not called in benchmarking, since creation of points would ruin the runtime analysis.

- For Merge Hull, The internal point selection for the merge step has been done in a way to create least amount of monotone chains. Naturally, this means that we tried to select a point internal to both of the hulls. However, there is no way of assuring it. One way of increasing its probability would be selecting centroid point of one of the hulls, but this is computationally heavy as we need to traverse all of the points on one of the hulls, which is in linear time. Instead, I chose the internal point as the centroid of {first point of the hull, ((# of points in the hull) / 3)’th point of the hull, (2*(# of points in the hull) / 3)’th point of the hull}. This way I approximate the biggest triangle that can be constructed that would have its corners as points from the input convex hull. This is done in constant time.

4) Benchmarks

Algorithms have been tested in eight different settings in total. Four of these used 2D Gaussian distribution and remaining four of these used 2D Uniform distribution. For each distribution, algorithms have been tested using 1) Thousands of points, ranging from 1,000 to 9,000, increasing 1,000 points at each step; 2) Tens of thousands of points, ranging from 10,000 to 90,000, increasing 10,000 at each step; 3) Hundreds of thousands of points, ranging from 100,000 to 900,000, increasing 100,000 at each step; and 4) Millions of points, ranging from 1,000,000 to 9,000,000, increasing 1,000,000 at each step.

In each step, algorithms are tested using different copies of a same set of points. Garbage collector had to be triggered after each algorithm run and before the creation of copies due to limited memory constraints and to preserve cache locality for algorithm as much as possible.

Runtimes of the algorithms are given in seconds. You can replicate the benchmarks by running benchmark.py.

4.1) Thousands of points

1. Gaussian Distribution:

Table 1: Gaussian Distribution Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
1000	0.0034	0.0039	0.0067	0.0157
2000	0.0071	0.0077	0.0113	0.0311
3000	0.0111	0.0118	0.0163	0.0385
4000	0.0149	0.0170	0.0221	0.0606
5000	0.0200	0.0265	0.0272	0.0686
6000	0.0232	0.0265	0.0314	0.0742
7000	0.0269	0.0261	0.0355	0.0954
8000	0.0305	0.0426	0.0399	0.1154
9000	0.0333	0.0398	0.0446	0.1251

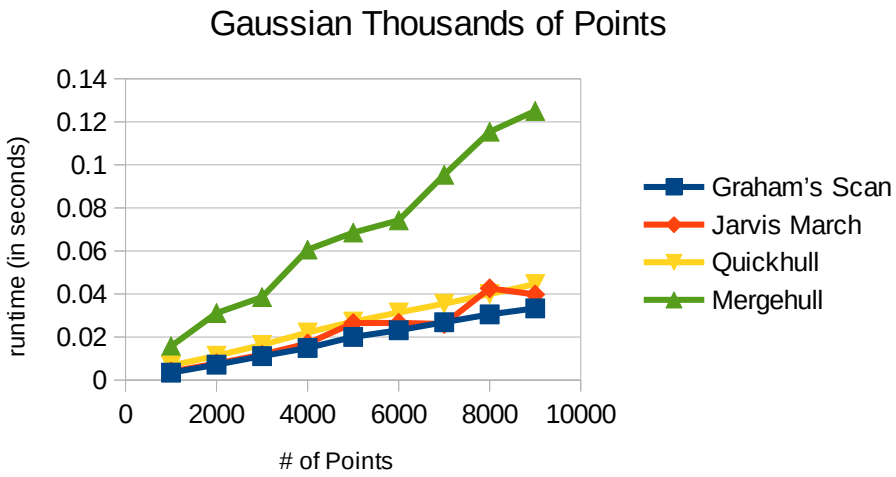

Figure 1: Gaussian Distribution Thousands of Points Results

2. Uniform Distribution:

Table 2: Uniform Distribution Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
1000	0.0034	0.0061	0.0067	0.0152
2000	0.0072	0.0213	0.0139	0.0307
3000	0.0114	0.0286	0.0292	0.0396
4000	0.0152	0.0351	0.0259	0.0628
5000	0.0200	0.0377	0.0323	0.0714
6000	0.0268	0.0427	0.0352	0.0770
7000	0.0268	0.0492	0.0427	0.0985
8000	0.0292	0.0679	0.0462	0.1203
9000	0.0333	0.0650	0.0534	0.1316

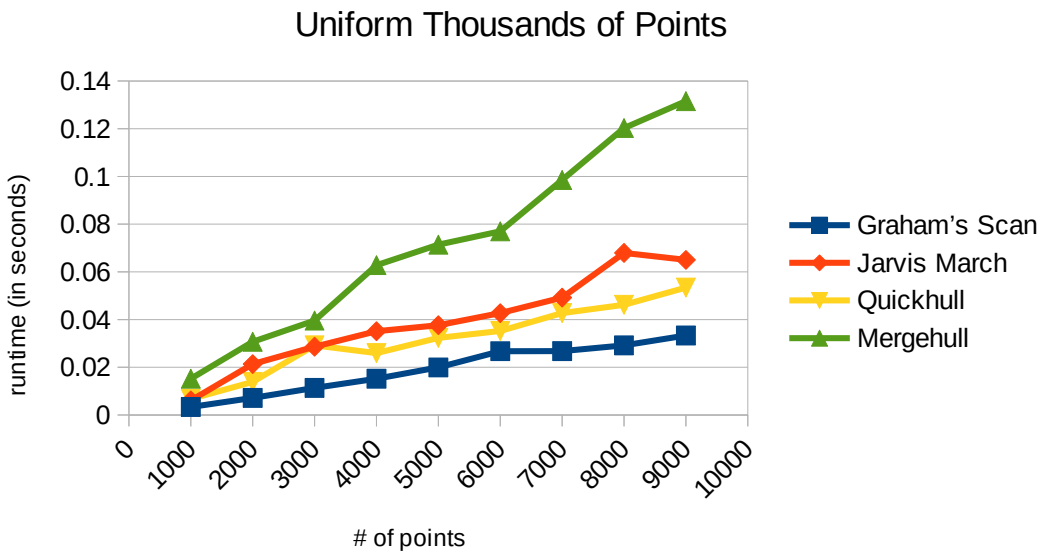

Figure 2: Uniform Distribution Thousands of Points Results

4.2) Tens of Thousands of points

1. Gaussian Distribution:

Table 3: Gaussian Distribution Tens of Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
10000	0.0546	0.0450	0.0492	0.1297
20000	0.0703	0.0976	0.0953	0.2588
30000	0.1180	0.1021	0.1383	0.4046
40000	0.1535	0.2167	0.1853	0.5170
50000	0.1838	0.3366	0.2297	0.5856
60000	0.2426	0.3060	0.2766	0.8011
70000	0.2851	0.3348	0.3214	0.9435
80000	0.3223	0.4346	0.3734	1.0054
90000	0.3611	0.3481	0.4100	1.0624

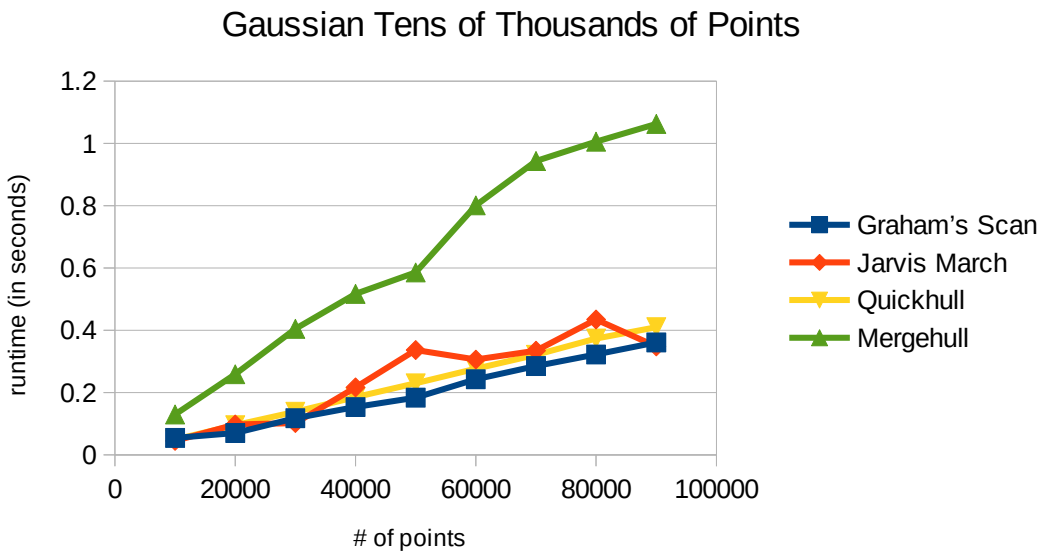

Figure 3: Gaussian Distribution Tens of Thousands of Points Results

2. Uniform Distribution:

Table 4: Uniform Distribution Tens of Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
10000	0.0351	0.0626	0.0579	0.1365
20000	0.0690	0.1947	0.1048	0.2692
30000	0.1178	0.2413	0.1512	0.4226
40000	0.1513	0.3283	0.2233	0.5349
50000	0.1860	0.3615	0.2625	0.6126
60000	0.2453	0.6182	0.3328	0.8380
70000	0.2812	0.6946	0.3809	0.9967
80000	0.3170	0.6992	0.3942	1.0710
90000	0.3586	0.9147	0.4727	1.1361

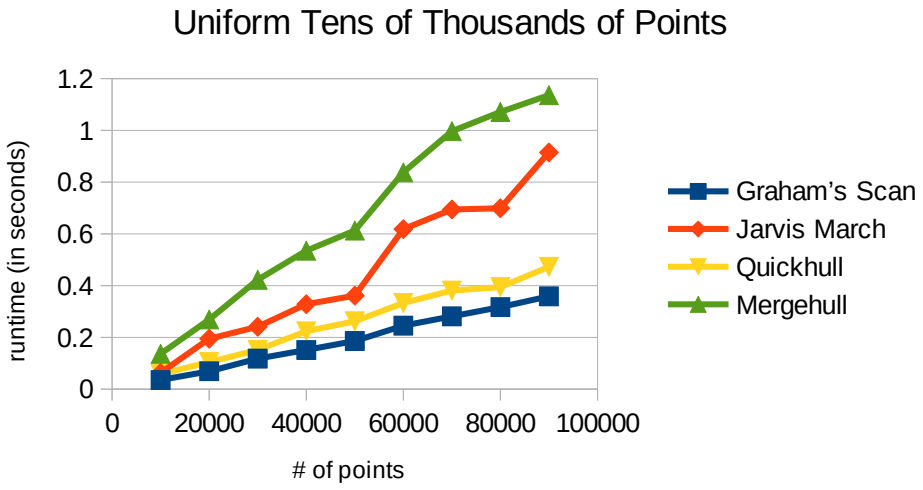

Figure 4: Uniform Distribution Tens of Thousands of Points Results

4.3) Hundreds of Thousands of points

1. Gaussian Distribution:

Table 5: Gaussian Distribution Hundreds of Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
100000	0.6350	0.3787	0.4582	1.1453
200000	1.2614	0.6669	0.9132	2.2799
300000	1.8296	1.7499	1.3648	3.9079
400000	2.4047	2.2082	1.8200	4.6113
500000	3.0305	2.5656	2.5141	6.7550
600000	3.6480	3.4682	2.7400	7.8475
700000	4.3250	4.4360	3.1913	8.4824
800000	4.8194	4.8356	3.7248	9.2754
900000	5.4997	4.8386	4.0979	11.6035

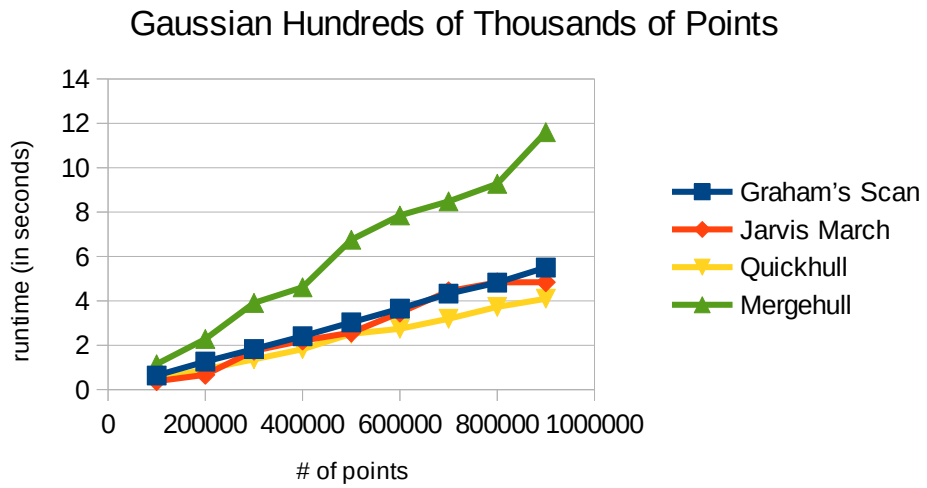

Figure 5: Gaussian Distribution Hundreds of Thousands of Points Results

2. Uniform Distribution:

Table 6: Uniform Distribution Hundreds of Thousands of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
100000	0.6336	1.0945	0.5078	1.2337
200000	1.2614	2.0359	0.9951	2.4708
300000	1.8320	3.0390	1.6802	4.1608
400000	2.4087	4.3898	2.0913	4.9900
500000	3.0253	5.8173	2.7034	7.2679
600000	3.7033	6.8304	3.0461	8.4086
700000	4.2939	6.5753	3.8485	9.2208
800000	4.8119	10.6119	4.4460	10.0980
900000	5.4933	10.6145	4.9286	12.3625

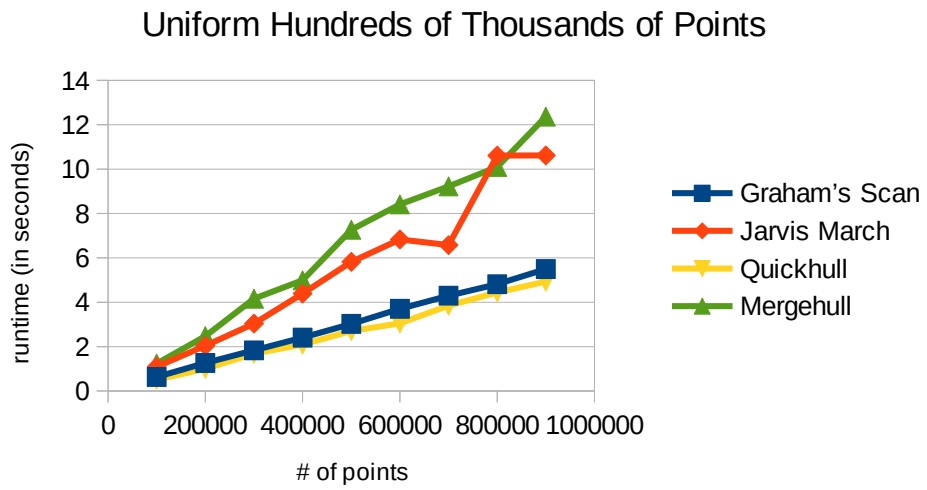

Figure 6: Uniform Distribution Hundreds of Thousands of Points Results

4.4) Millions of points

1. Gaussian Distribution

Table 7: Gaussian Distribution Millions of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
1000000	5.3303	4.4366	4.6022	13.7636
2000000	11.4630	12.6671	9.0942	27.5042
3000000	18.2356	15.1313	13.5974	35.7251
4000000	24.8451	21.2768	18.3058	55.3236
5000000	31.1413	29.3554	22.9048	66.2315
6000000	39.2451	32.3453	27.1136	72.2360
7000000	47.3976	43.9561	31.7827	90.0517
8000000	54.0594	52.4300	36.2385	112.9743
9000000	62.1922	50.6276	40.7769	126.0339

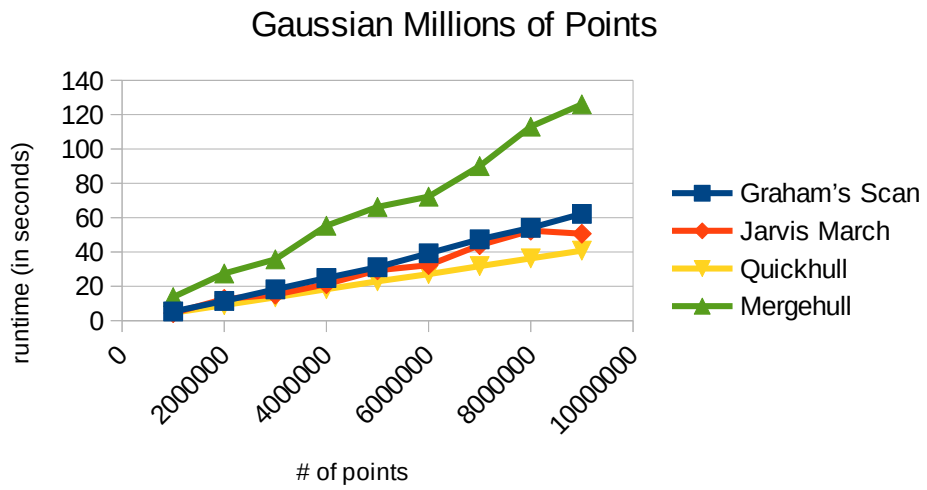

Figure 7: Gaussian Distribution Millions of Points Results

2. Uniform Distribution

Table 8: Uniform Distribution Millions of Points Results

# of Points	Graham's Scan	Jarvis March	Quickhull	Mergehull
1000000	5.2599	11.9393	4.9705	14.5706
2000000	11.2483	22.9326	11.0469	29.4271
3000000	18.2670	30.0377	14.3552	38.2512
4000000	24.9986	44.3057	22.4549	59.5555
5000000	32.4901	69.3380	24.3222	70.9150
6000000	39.6263	63.5350	30.3493	77.8194
7000000	47.3875	92.7945	37.8516	96.3750
8000000	53.3399	100.2511	42.1172	119.6141
9000000	61.4059	130.1873	45.2904	134.7874

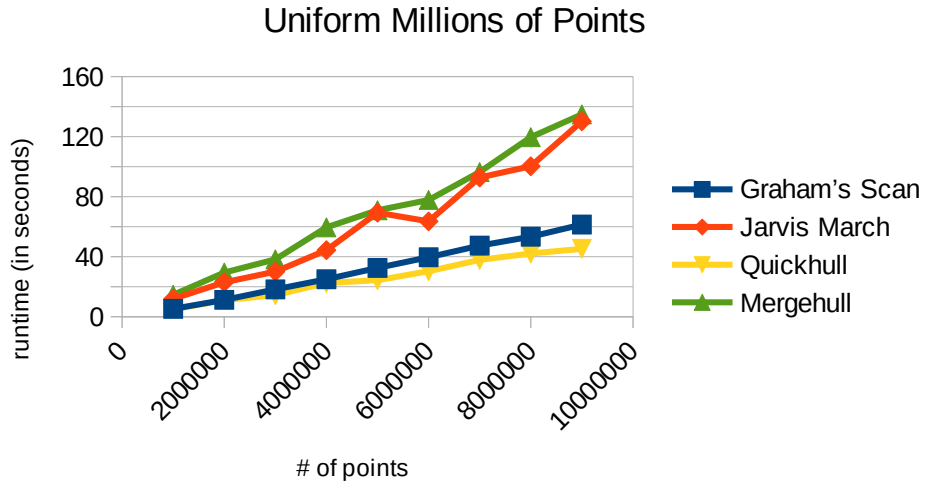

Figure 8: Uniform Distribution Millions of Points Results

Discussion about Benchmarks:

In this benchmarks, we can see the behavioral change of algorithms in different distribution of points. We also get an idea on how these algorithms behave.

- Initially Quickhull performs worse than Graham's Scan. As the number of points increase, despite it being asymptotically worse that Graham's Scan, it starts to outperform Graham's Scan. This is because as the number of points increase and as points get clustered, Quickhull eliminates huge percentage of points in recursive calls. We observe that Quickhull works best when the number of points in the convex hull is much smaller than the number of input points.

- One observation about Jarvis March is that the number of edges in the convex hull, which is equal to the number of points in the convex hull, highly affects the runtime. For instance, in Figure 8 we can see that despite the increase in the number of points the runtime of Jarvis March decreased from 5,000,000 points to 6,000,000 points. This is because, in that instance of uniform distribution of 6,000,000 points there were much less points on the convex hull and therefore the runtime decreased despite the increase in the number of points.

- Another observation about Jarvis March is that it performs much worse in 2D Uniform distribution than 2D Gaussian distribution. This performance difference can be seen clearly in Figure 7 and Figure 8. This happens because at the same number of points 2D Uniform distribution contains much more points on the convex hull than 2D Gaussian distribution. This aligns well with our previous point.

- We can observe that Merge Hull, despite being asymptotically optimal in the worst case, performs worse than other algorithms, for instance Jarvis March which is asymptotically not optimal. I attribute this to Merge Hull not preserving cache locality, as it is not an in-place algorithm. Another reason behind this might be the usage of Python programming language, in which I do not have full control on how the memory is allocated and accessed.

5) Application Implementation Details

This project has been implemented using Python 3.12.2. I used following modules for my implementation:

- math (for polar angle calculation)
- numpy (for generation of gaussian and uniform points)
- tkinter (for visualization using canvas)
- gc (for triggering garbage collector in benchmarking)
- time (for implementing delay in visualization and for measuring runtimes in benchmarking).

Please refer to README.txt for installation details.

There are certain details needed to be mentioned about this application implementation:

- Y-axis is flipped in tkinter.canvas. Therefore, algorithms may look like being executed in reverse order. For instance, where Jarvis March is marching up it will look like it is marching down.

- For visualization purposes, a scale factor has been implemented for zooming in and out to the canvas. Using this scale factor, the points are relocated into the limited screen space of

the canvas based on to which point we zoom in and out. The sizes of the points are also managed using this scale factor.

- tkinter.canvas provides coordinates for the mouse traveling on the canvas taking the top left corner of the canvas as the origin, however this does not match the real coordinates in the coordinate system since we zoom in and out to canvas. Therefore, a mechanism has been implemented for converting the canvas coordinates to real coordinates. This mechanism is used when we figure out the real coordinate of the mouse, so when we insert points, we have the real coordinates of the points.

- Again, tkinter.canvas inserts and visualizes the points based on the canvas coordinates, therefore another mechanism has been implemented for converting the real coordinates to canvas coordinates. This is used when we insert a point to the canvas: we take its real coordinates and convert it to canvas coordinates and visualize the point on the canvas using its canvas coordinates.

- When you move your mouse on the canvas, you will see coordinates of the mouse near it. Those are the real coordinates of the mouse.

A list of features is as follows:

- User can insert points using 2D Gaussian and 2D Uniform distributions. For both distributions, first you need to specify the number of points. For Gaussian distribution you need to provide a covariance matrix in the following format <val1>,<val2>;<val3>,<val4>. An example input would be 1000,0;0,1000. For Uniform distribution you need to provide minimum and maximum X-coordinate values and minimum and maximum Y-coordinate values.

- You can use the “Clear Points” button available on the bottom panel to clear all the points.

- Every point has its real coordinates displayed near it. You can disable this setting in the application by clicking “Disable Coordinates” button. This button will not disable your mouse coordinates.

- This implementation has two modes for the mouse clicking: “Insert Point” mode and “Delete Point” mode. When you are in “Insert Point” mode, your mouse click will insert a point to the coordinates of your mouse and trigger the selected algorithm to run. When you are in “Delete Point” mode, a red square shaped boundary will appear around your mouse in which every point inside that square will be deleted. The size of the square is not relative to the canvas but it is relative to the real coordinate system, its size changes by the scale factor. Thus, you can fit more points to be deleted into the square by zooming out, and less points by zooming in.

- This implementation has three modes for visualizing the convex hull algorithms: 1) Do NOT visualize, 2) Visualize without delay, 3) Visualize with delay. In all of the modes, the coordinates of the points in the convex hull are printed to the terminal. In the first mode, there is no visualization happening. In the second mode visualization happens in a fast pace without any delay. In the third mode visualization happens with a delay that you can specify in seconds, default delay is 1 second.

- Bottom panel contains buttons for switching between algorithms. When an algorithm runs in any mode, these buttons are disabled. They are re-enabled when the algorithm is done. If

you have a lot of points in the canvas and due to your selected mode the algorithm takes too much time due to visualization, there is no functionality implemented to stop it. It will eventually finish, but if you are in a hurry, simply close the application and rerun it.

Things to be careful about when using the application:

- The delay for the third visualization mode has been implemented using time.sleep, therefore, whole app freezes in a delay.

- If you try to insert a point in the middle of an algorithm running, you may get unexpected results. Thus, do not do it.

- Please be aware of the number of points you create on the canvas. After 10,000 points Running Graham's Scan, Jarvis March and Merge Hull in visual modes will take huge amount of time. Make sure coordinates are disabled before creating 10,000 points or more, as the data you see on the canvas will not make much sense and disabling coordinates after creating that many points will take a lot of time.