Bilkent University

CS-478

Computational Geometry

Spring 2023/2024

Progress Report

Erkin Aydın

22002956

1) Summary of progress of my project so far

I have conducted a research on the convex hull algorithms. I have learned their history, how they work, analysis of their time and space complexities.

I have started to code my project and set up the basic functionalities needed for visualizing algorithms. I have also coded some of the functionalities needed for experimentation, such as inserting a point to a desired location, deleting a point, deleting multiple points, deleting all the points, adding points based on a 2D Gaussian Distribution in which the user gives the number of points, mean and covariance matrix as input, and adding points based on a 2D Uniform Distribution in which the user gives the number of points, minimum and maximum coordinates.

2) Algorithms to be Used & Their Explanations

I will implement 4 convex hull algorithms: Graham's Scan, Jarvis March, Quickhull and Mergehull. I will go over the pseudo codes I have prepared and some other implementation details.

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 Leftv, 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) Mergehull

Mergehull is inspired by Mergesort. 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, instead of subdividing the problem we can prefer to use another algorithm to construct the convex hull. I prefer to use Graham's Scan.

Merge step is the tricky step in Mergehull, 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 Mergehull implementation:

MERGEHULL(S)
1: if |S| < 3(any small integer would suffice as well) then
2: | return GRAHAMS_SCAN(S)
3: Construct S_1 and S_2 where $S_1 \cup S_2 = S$ and $|S_1| \approx |S_2|$
4: $H(S_1)$:= MERGEHULL(S_1)
5: $H(S_2)$:= MERGEHULL(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 := centroid of $H(S_1)$
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) Other Implementation Details

I am using Python 3.12 and the following libraries for my implementation: tkinter, matplotlib, numpy. As I progress with my implementation, the number of used libraries may increase, but all of the algorithms will be implemented from scratch.

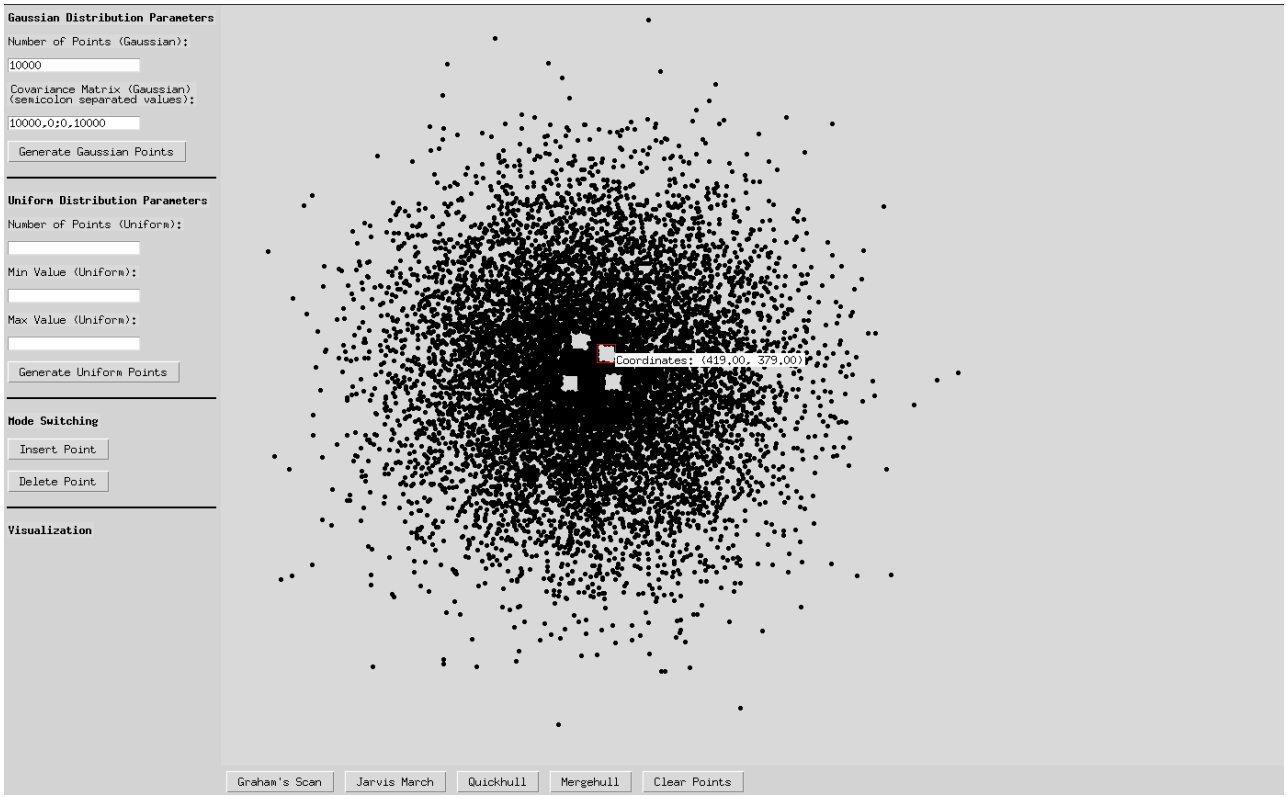
I will also present you some of the functionalities I have implemented so far.

Explanation of the Current Implementation

There are two panels and one canvas at the moment. The canvas will be used for adding and deleting points, as well as visualizing them. The left panel is used for adding and deleting points based on a 2D Gaussian or 2D Uniform distribution. You can also switch between “Insert Point” and “Delete Point” modes, in which if “Delete Points” is selected, a square eraser appears. The bottom panel will be used to select the convex hull algorithm to run. Additionally, it contains a “Clear Points” button to clear all the points on the canvas.

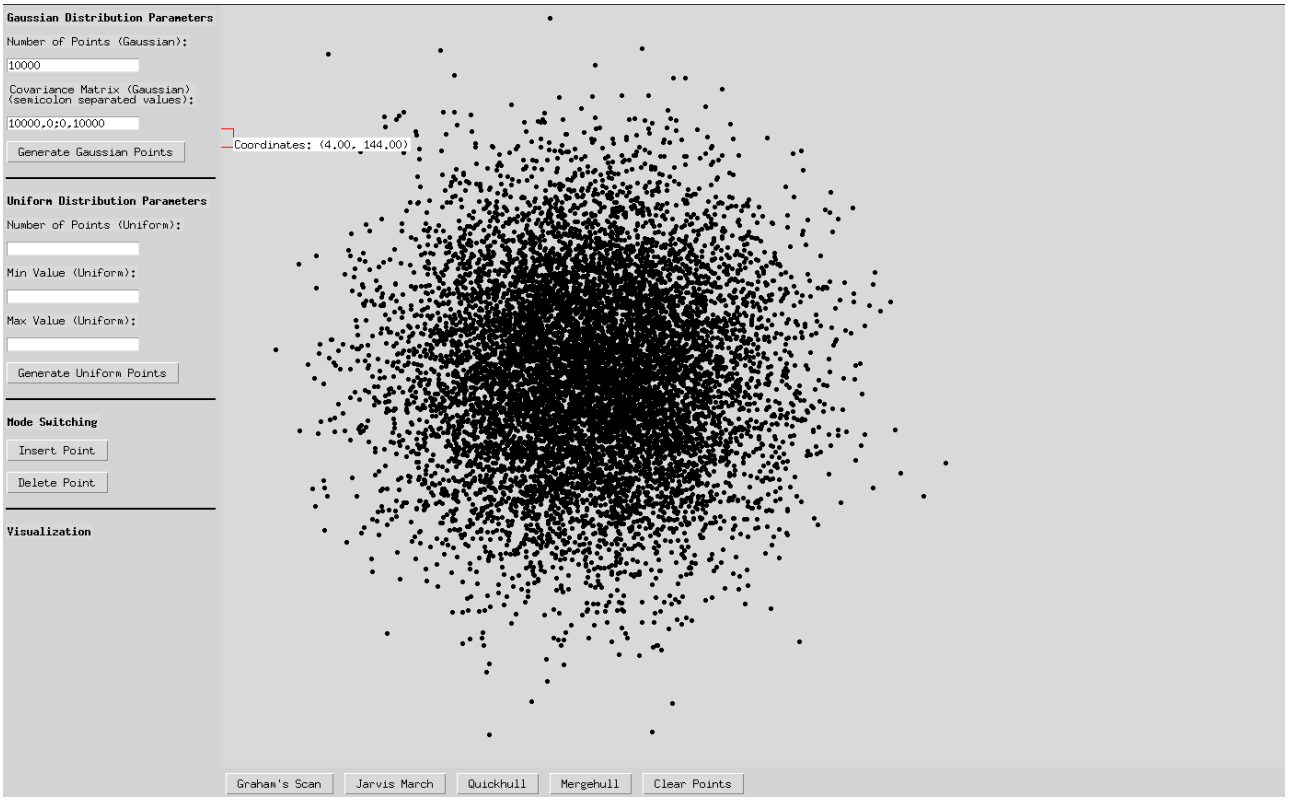
Inserting a point to a desired location

Deleting points



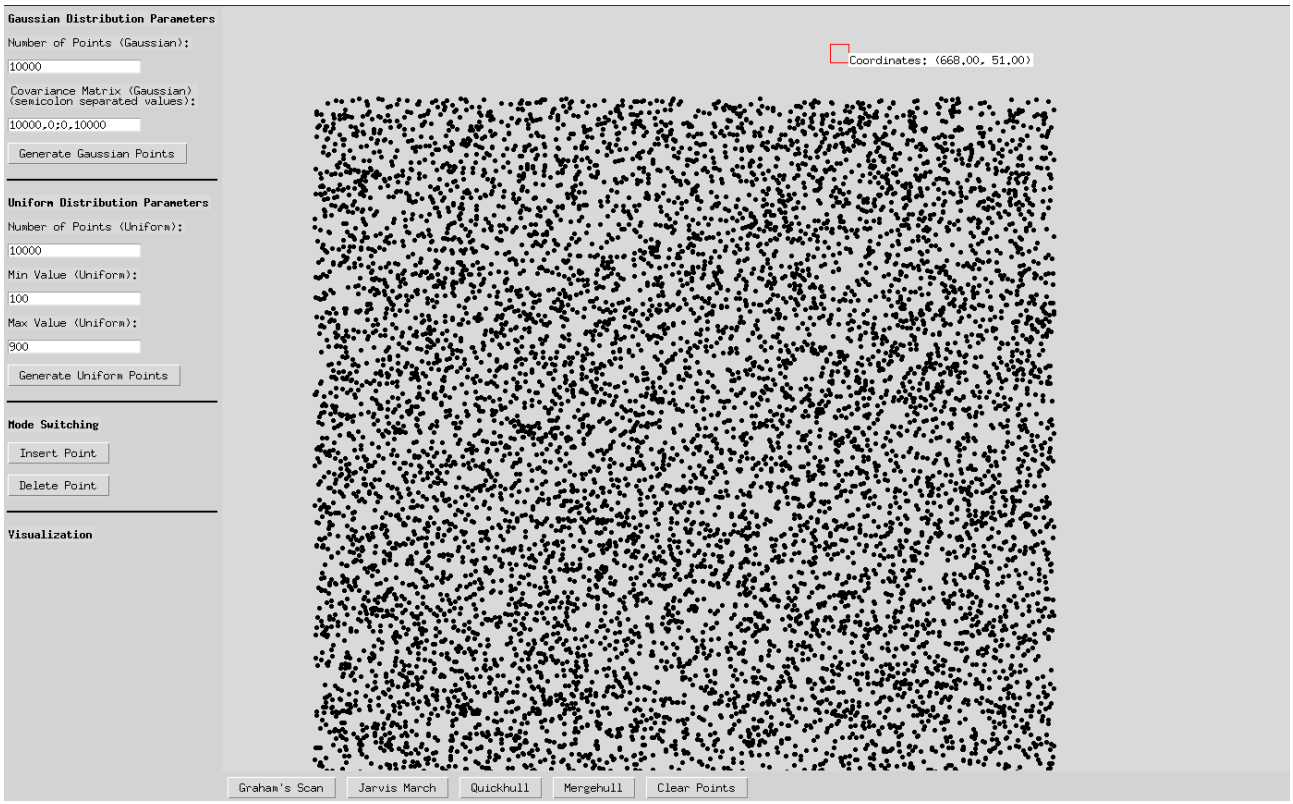
Adding points based on a 2D Gaussian Distribution

Here, the number of points is 10000, and the covariance matrix is selected as follows: [[10000, 0],[0, 10000]]



Adding points based on a 2D Uniform Distribution

Here, the number of points is 10000, min x and y coordinates are selected to be 100, and maximum x and y coordinates are selected to be 900.



4) Future Work:

I will continue to the implementation of convex hull algorithms. All will be implemented from scratch. I will add some visualization options so that the user will be able to track the progress of the algorithm step by step. Then, I will conduct performance experiments of these algorithms using huge number of points.