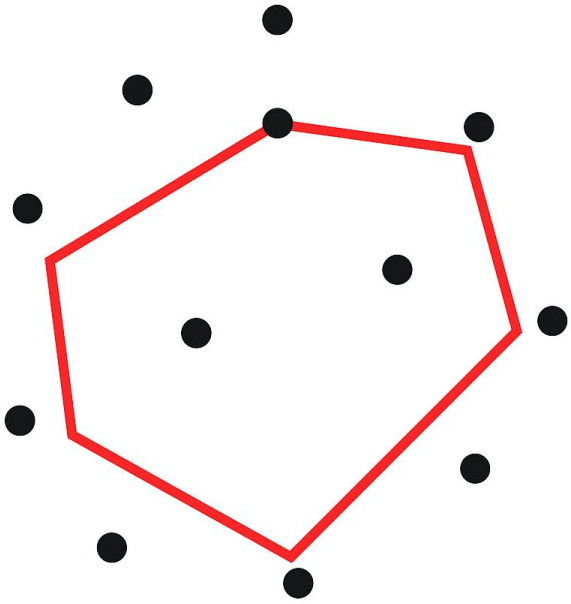
CONVEX HULL



The Convex Hull Algorithm

GRAM SCAN ALGORITHM

October 2025

#	الاسم	البرنامج
1	شهد سعيد عبداللطيف	علوم حاسب منفرد
2	سليمان وليد سليمان	علوم حاسب منفرد
3	فريده عماد ابراهيم	علوم حاسب منفرد
4	عبدالرحمن حماده السيد	علوم حاسب منفرد
5	دينا عبدالكريم حسين	بحتة حاسب
6	ساره يونان فؤاد	علوم حاسب منفرد
7	هنا اسلام عادل	بحتة حاسب
8	همس اشرف عبد الحكيم	بحتة حاسب
9	نورهان طه محمد السيد	بحتة حاسب
10	علياء عبدالسميع الشحات عبدالسميع	إحصاء حاسب

"Finding the Convex Hull of a Set of Points Using Computational Geometry Algorithms"

Group I:

- Implement and explain (Algorithm explanation in detail) Grham Scan's algorithm for computing the convex hull. Source code (Python/java/C++). **(10 Marks)**

  Input Points

  [(0,0), (1,2), (2,1), (2,4), (3,3), (4,0), (1,1)]

- **Visualization: (2 Marks)**

  Blue dots = all input points

  Red polygon = convex hull boundary

- Explain in detail the steps of computing the complexity time of the algorithm.

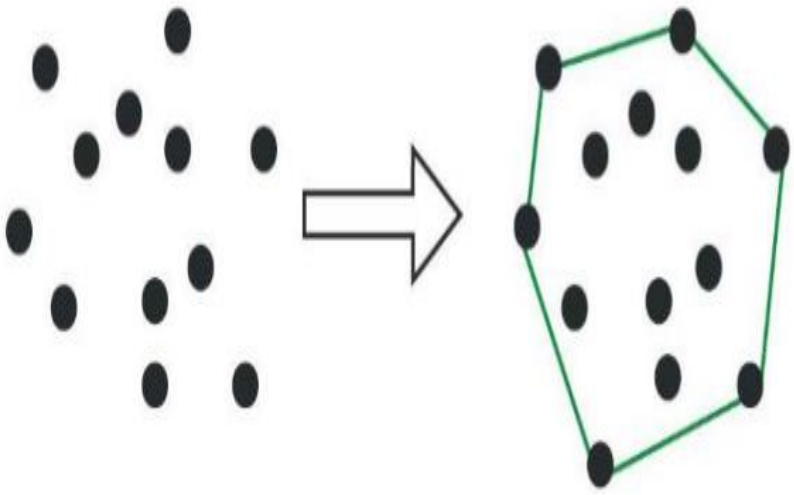
  (5 Marks)

Convex Hull Algorithm:

The Convex Hull Algorithm is used to find the convex hull of a set of points in computational geometry. The convex hull is the smallest convex set that encloses all the points, forming a convex polygon. This algorithm is important in various applications such as image processing, route planning, and object modeling.

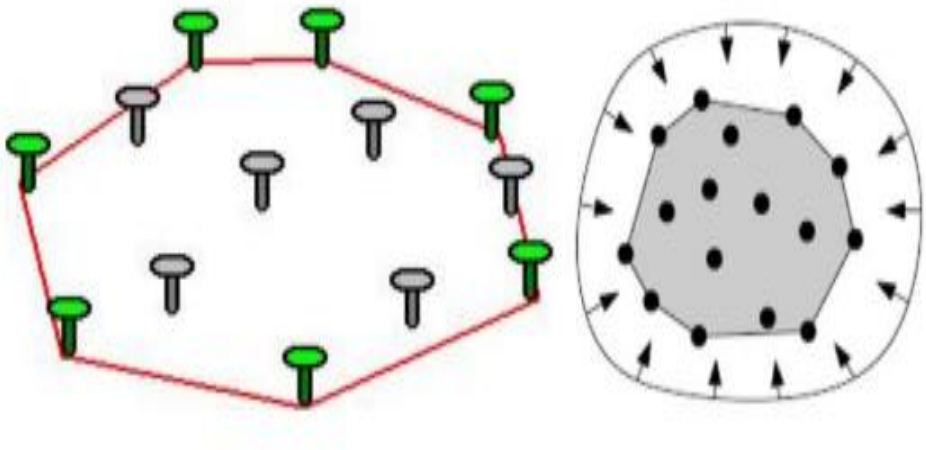
What is Convex Hull?

The convex hull of a set of points in a Euclidean space is the smallest convex polygon that encloses all of the points. In two dimensions (2D), the convex hull is a convex polygon. The below image shows a 2-D convex polygon:



Idea: A Rubber Band Analogy

Imagine you have a board with nails hammered into it at random points. Now, if you take a rubber band and stretch it around the entire set of nails and then let it go, it will snap into a specific shape, wrapping tightly around the outermost nails.



Formal Definition

The **Convex Hull** of a set of points is the smallest convex set that contains all the points.

Let's unpack the key terms:

• **Convex Set:** A shape where for any two points inside the shape, the entire straight line segment connecting them is also inside the shape. Think of a solid circle or a filled triangle—no indentations or "dents."

• **Smallest:** It's the most "efficient" or tightest-fitting convex shape that still contains all the original points. No smaller convex shape will do the job.

The Convex Hull itself is the polygon defined by the sequence of these outermost points (the nails the rubber band touches).

Key Characteristics

1. It's Always Convex: By definition, it has no dents or concave bays.

2. It's a Polygon: The hull is defined by a subset of the original points, connected by edges.

3. It Contains All Points: Every point from the original set is either **on the boundary** of the hull or **inside** it.

How is convex hull Calculated?

While the rubber band analogy is intuitive, a computer needs an algorithm. Common ones include:

• **Graham Scan:** Sorts the points by their angle around a pivot point (typically the bottom-most point) and then builds the hull by checking for "turns."

• **Jarvis March (or Gift Wrapping):** Finds the left-most point and then "wraps" around the set by repeatedly finding the point that is most to the right (from the current point's perspective), mimicking the rubber band.

In summary, the Convex Hull is a fundamental geometric concept that provides a simple, convex boundary for a complex set of points, with wide-ranging applications from robotics and gaming to data science. How to calculate the convex Hull using Graham Scan with examples

Of course! Let's dive into the **Graham Scan** algorithm for calculating the Convex Hull. It's one of the most elegant and efficient algorithms for this task.

Graham Scan Algorithm

The Graham Scan works by first finding a guaranteed starting point on the hull, then sorting all other points based on their angle around this point, and finally building the hull by traversing these sorted points and eliminating points that would create a concave turn.

And now by applying The Graham Scan Algorithm to solve our problem as follows:

Let P0=(0,0) , P1=(1,2) , P2=(2,1) , P3=(2,4) , P4=(3,3) , P5=(4,0) , P6=(1,1) .

Step 1: Find **the Anchor Point** Find the point with the **lowest y-coordinate** (and if tied, the leftmost one). This point is guaranteed to be on the convex hull.

In our problem:

• P0 = (0, 0) has the lowest y-coordinate → **Anchor Point.**

Step 2: Sort by Polar Angle Sort all remaining points by their polar angle **relative to the anchor point**.

If two points have the same angle, keep only the one **farthest** from the anchor. Calculating angles relative to P0(0, 0):

θ=tan-1 (y/x) in radiance

• P1(1, 2) → θ=63.4 °

• P2(2, 1) → θ=26.6°

• P3(2, 4) → θ=63.4 °

• P4(3, 3) → θ=45°

• P5(4, 0) → θ=0°

• P6(1, 1) → θ=45°

Sorted order: P5(4, 0), P2(2, 1), P4(3, 3), P6(1, 1), P1(1, 2) , P3(2, 4).

When building a convex hull using algorithms like Graham Scan, if two points have **the same polar angle** relative to the pivot point (typically the point with the lowest y-coordinate, or leftmost in case of ties), we need to decide which one to keep.

The Rule: Keep the point that is farther from the pivot point and discard the closer one.

Why?

• Points with the same polar angle lie on the same ray emanating from the pivot point.

• The farther point will "**dominate**" the closer one in terms of forming the convex hull.

• Keeping the closer point would create **concavities** or unnecessary interior points.

Hence, Take P3(2, 4) and discard P1(1, 2).

And, Take P4(3, 3) and discard P6(1, 1).

Now, The Sorted Order is:

P0(0, 0) , P5(4, 0), P2(2, 1) , P4(3, 3) , P3(2, 4).

Step 3: Build the Hull Use a stack to build the convex hull. Process points in sorted order, and for each point, check if moving to it creates **a clockwise turn** (which would make the hull concave).

Initial stack: [P0, P5]

Process P1(2, 1):

• Check turn from P5→P2:

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

D=(4-0)(1-0)-(0-0)(2-0)=+4 **Counter-clockwise ✓**

• Push P2 → Stack: [P0, P5, P2]

Process P4(3, 3):

• Check turn from P5→P2:

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

D=(2-4)(3-0)-(1-0)(3-4)=-5 **Clockwise ✗**

• Now **Pop** P2 from stack.

• Check turn from P5→P4:

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

D=(4-0)(3-0)-(0-0)(3-0)=+12 **Counter-clockwise ✓**

• Push P4 → Stack: [P0, P5, P4]

Process P3(2, 4):

• Check turn from P4→P3:

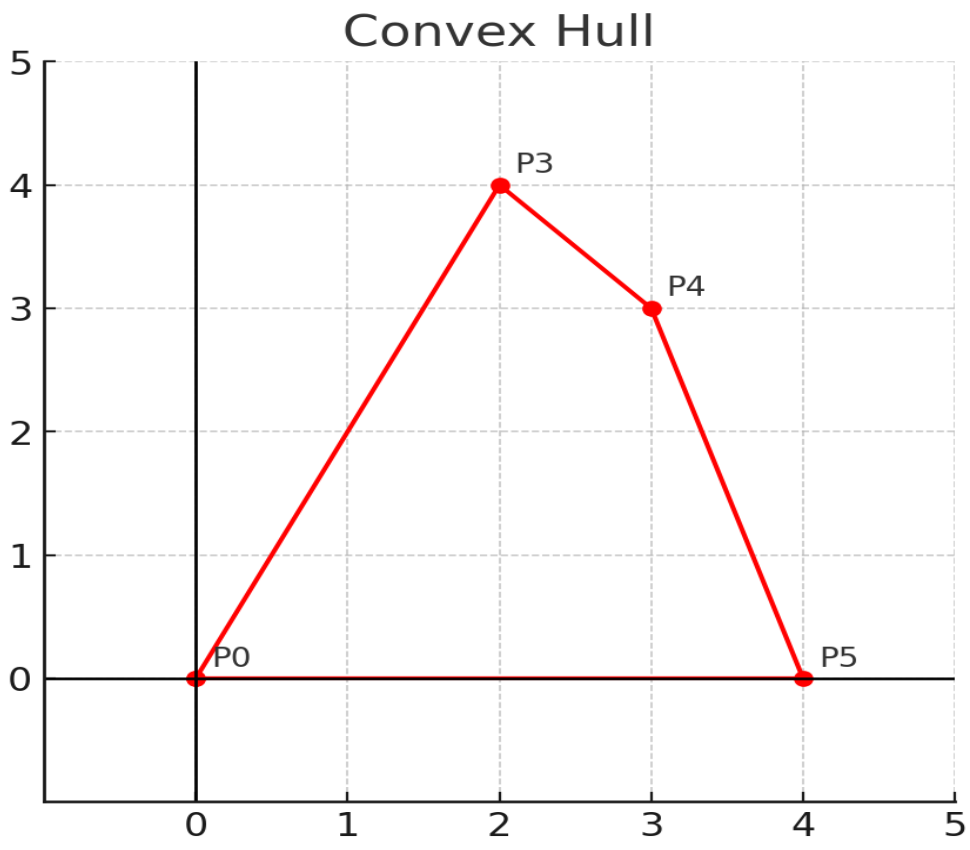
D=(x2-x1)(y3-y1)-(y2-y1)(x3-x1)

D=(3-4)(4-0)-(3-0)(2-4)=+2 **Counter-clockwise ✓**

• Push P3 → Stack: [P0, P5, P4, P3].

Final Hull: P0 → P5 → P4 → P3 → P0

The Final Hull as follows:



Time Complexity Analysis:

Let n be the number of input points.

1.Find P0 (Anchor Point):

- This requires a single loop through all **n points** to find the minimum y-coordinate (and x-coordinate for ties).
- Complexity: **O(n)**

2.Sort the Points:

- This is the most expensive step. We sort n-1 points.
- The comparison function (using orientation) takes constant time, O(1).
- Using an efficient sorting algorithm takes O(n log n) time.
- Complexity: **O(n log n)**

3.Build the Hull (Stack Scan):

- This step involves a single for loop that iterates through all **n-1** sorted points.
- Inside the for loop, a while loop pops points from the stack.
- It might seem like this nested loop structure is O(n^2), but it is not.
- Average Analysis: Each point in the sorted list is **pushed** onto the stack exactly **one time**. Each point can be **popped** from the stack at most **one time**.
- The for loop runs **n-1** times (one push per iteration).
- The while loop (popping) can run at most **n-1** times *in total* across the *entire* algorithm's execution.
- Therefore, the total number of operations (pushes and pops) in this step is proportional to n.
- Complexity: **O(n)**

Overall Complexity:

The total time T(n) is the sum of these steps:
T(n) = T(find P0) + T(sort) + T(scan)
T(n) = O(n) + O(n log n) + O(n).

Hence, The Complexity Time is : O(n log n).

Function Descriptions

Function: compare

Compares two points by their Y-coordinate.
If Y is the same, compares by X.
Used to find the **lowest point** (anchor).

Function: distances

computes the **squared Euclidean distance** between two points.
We use the squared version to avoid slow square roots.

Function: orientation

Determines the **turn direction** :

- 0 → Collinear (straight line)
- 1 → Clockwise (right turn)
- 2 → Counter-clockwise (left turn)

Function: polarAngle

calculates the **polar angle** (angle from the X-axis) between the anchor and another point.
Used to sort points around the anchor.

Function: sortByAngleAndDistance

Sorts all points by **angle**, and if two points have the same angle, keeps the closer one first.
Used to prepare points for hull construction

Function: findAnchor

Finds the **anchor point** (lowest Y, then lowest X).
This will be the first point in the convex hull

Function: sortByPolarAngle

Computes polar angles for all points relative to the anchor and sorts them.
Also removes duplicate angles (keeps the farthest point).

Function: grahamScan

Implements the **Graham Scan Algorithm**:

1. Find the anchor point.
2. Sort the other points by polar angle.
3. Use a stack to construct the convex hull

GITHUB LINK:

https://github.com/ShahdSaed/Convex-Hull-Algorithm.git