Convex Hull

By: Aditya Sivaram

PES1201700012-4H

Nishant Shastry

PES1201700115-4H

Convex Hull

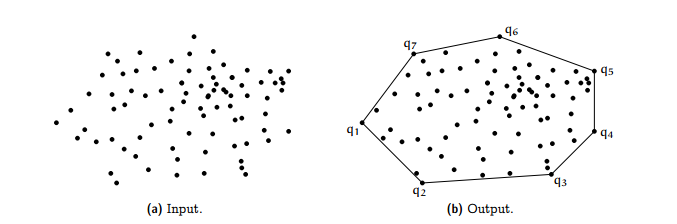
The convex hull may be defined either as the intersection of all convex sets containing *X*, or as the set of convex combinations of points in *X*.

The algorithms work by finding the outermost boundary of the shape such that all points lie either on the boundary or inside it

There are a plethora of algorithms to execute Convex Hull. We have executed solved the convex hull problem using 4 algorithms

* Monotone Chain
* Quick Hull
* Graham Scan
* Jarvis March

These algorithms vary in complexity and approach.



Monotone Chain

Pseudo Code:

Input: a list P of points in the plane.

Precondition: There must be at least 3 points.

Sort the points of P by x-coordinate (in case of a tie, sort by y-coordinate).

Initialize U and L as empty lists.

The lists will hold the vertices of upper and lower hulls respectively.

for i = 1, 2, ..., n:

while L contains at least two points and the sequence of last two points

of L and the point P[i] does not make a counter-clockwise turn:

remove the last point from L

append P[i] to L

for i = n, n-1, ..., 1:

while U contains at least two points and the sequence of last two points

of U and the point P[i] does not make a counter-clockwise turn:

remove the last point from U

append P[i] to U

Remove the last point of each list (it's the same as the first point of the other list).

Concatenate L and U to obtain the convex hull of P.

Points in the result will be listed in counter-clockwise.

Python:

def convex\_hull(points):  
    points = sorted(set(points))  
    if len(points) <= 1:  
        return points  
  
    lower = []  
    for p in points:  
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:  
            lower.pop()  
        lower.append(p)  
  
    upper = []  
    for p in reversed(points):  
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:  
            upper.pop()  
        upper.append(p)  
    print(lower[:-1] + upper[:-1])  
  
def cross(o, a, b):  
        return (a[0] - o[0]) \* (b[1] - o[1]) - (a[1] - o[1]) \* (b[0] - o[0])  
  
  
n=input("Number of points")  
n=int(n)  
points=[]  
for i in range(0,n):  
    x,y=input().split()  
    x=int(x)  
    y=int(y)  
    stu=tuple((x,y))  
    points.append(stu)  
    convex\_hull(points)

Complexity: 0(n logn)

It does so by first sorting the points lexicographically (first by *x*-coordinate, and in case of a tie, by *y*-coordinate), and then constructing upper and lower hulls of the points in 0(n) time.

Jarvis March

Pseudo Code:

jarvis(S)

// S is the set of points

pointOnHull = leftmost point in S //which is guaranteed to be part of the Hull(S)

i = 0

repeat

P[i] = pointOnHull

endpoint = S[0] // initial endpoint for a candidate edge on the hull

for j from 1 to |S|

if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint)

endpoint = S[j] // found greater left turn, update endpoint

i = i+1

pointOnHull = endpoint

until endpoint == P[0] // wrapped around to first hull point

Python:

def jarvis\_march(points):  
    a =  min(points, key = lambda point: point[0])  
    index = points.index(a)  
  
    l = index  
    result = []  
    result.append(a)  
    while (True):  
        q = (l + 1) % len(points)  
        for i in range(len(points)):  
            if i == l:  
                continue\  
            d = direction(points[l], points[i], points[q])  
            if d > 0 or (d == 0 and distance(points[i], points[l]) > distance(points[q], points[l])):  
                q = i  
        l = q  
        if l == index:  
            break  
        result.append(points[q])  
  
    return result  
  
def direction(p1, p2, p3):  
    return  cross\_product(subtract(p3, p1), subtract(p2, p1))  
  
def subtract(p1, p2):  
        return p1[0] - p2[0], p1[1] - p2[1]  
  
def cross\_product(p1, p2):  
    return p1[0] \* p2[1] - p2[0] \* p1[1]  
  
def distance(p1, p2):  
    distance = abs(p1[0] - p2[0]) \*\* 2 + abs(p1[1] - p2[1]) \*\* 2  
    return distance  
  
n=input("Number of points ")  
n=int(n)  
points = []  
for i in range(0,n):  
     x,y=input().split()  
     x=int(x)  
     y=int(y)  
     li = [x,y]  
     points.append(li)  
     hullpoints=jarvis\_march(points)  
print(hullpoints)

Complexity: 0(nh)

*n* is the number of points and *h* is the number of points on the convex hull. Its real-life performance compared with other convex hull algorithms is favorable when n is small.

Quick Hull

Pseudo Code:

Input = a set S of n points

Assume that there are at least 2 points in the input set S of points

QuickHull (S)

{

// Find convex hull from the set S of n points

Convex\_Hull := {}

Find left and right most points, say A & B

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)

}

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

S2 are points on the right side of the oriented line from C to Q

FindHull(S1, P, C)

FindHull(S2, C, Q)

}

Output = Convex Hull

Complexity:

Average Case:0(n logn)

Worst Case: 0(n2)

Python:

def convexHullPoints(points):  
    min, max = findMinMax(points)  
    hullPoints = quickHull(points, min, max)  
    hullPoints = hullPoints + quickHull(points, max, min)  
  
    return hullPoints  
  
def quickHull(points, min, max):  
        left = toLeft(min, max, points)  
        pivot = maxFromLine(min, max, left)  
        if len(pivot) < 1:  
                return [max]  
        hullPoints = quickHull(left, min, pivot)  
        hullPoints = hullPoints + quickHull(left, pivot, max)  
        return hullPoints  
  
def toLeft(start, end, points):  
        left = []  
        for point in points:  
                if isCCW(start, end, point):  
                        left.append(point)  
        return left  
  
def maxFromLine(start, end, points):  
        maxDist = 0  
        maxPoint = []  
        for point in points:  
                if point != start and point != end:  
                        dist = distance(start, end, point)  
                        if dist > maxDist:  
                                maxDist = dist  
                                maxPoint = point  
        return maxPoint  
  
def findMinMax(points):  
        minX = float('inf')  
        maxX = 0  
        minY = 0  
        maxY = 0  
        for x, y in points:  
                if x < minX:  
                        minX = x  
                        minY = y  
                if x > maxX:  
                        maxX = x  
                        maxY = y  
        return [minX, minY], [maxX, maxY]  
  
def distance(start, end, pivot):  
        xStart, yStart = start  
        xEnd, yEnd = end  
        xPivot, yPivot = pivot  
        distance = abs((yEnd - yStart) \* xPivot - (xEnd - xStart) \* yPivot + xEnd \* yStart - yEnd \* xStart) / ((yEnd - yStart) \*\* 2 + (xEnd - xStart) \*\* 2) \*\* 0.5  
        return distance  
  
def isCCW(start, end, point):  
        xStart, yStart = start  
        xEnd, yEnd = end  
        xPoint, yPoint = point  
        return (xEnd - xStart) \* (yPoint - yStart) > (yEnd - yStart) \* (xPoint - xStart)  
  
n=input("Number of points ")  
n=int(n)  
points = []  
for i in range(0,n):  
     x,y=input().split()  
     x=int(x)  
     y=int(y)  
     li = [x,y]  
     points.append(li)  
     hullpoints=convexHullPoints(points)  
print(hullpoints)

Graham Scan

|  |  |
| --- | --- |
| |  | | --- | | **Pseudo Code:**  **input:** a  set of points **S** = {P = (P.x,P.y)}      Select the rightmost lowest point P0 in **S**     Sort **S** radially (ccw) about P0 as a center {         Use isLeft() comparisons         For ties, discard the closer points     }     Let P[N] be the sorted array of points with P[0]=P0      Push P[0] and P[1] onto a stack       while i < N     {         Let PT1 = the top point on          If (PT1 == P[0]) {             Push P[i] onto              i++     // increment i         }         Let PT2 = the second top point on          If (P[i] is strictly left of the line  PT2 to PT1) {              Push P[i] onto              i++     // increment i         }         else             Pop the top point PT1 off the stack     }      **Output:** the convex hull of **S**. | |

Python:

import numpy as np

import math

n=input("Number of points")

n=int(n)

arx=[]

ary=[]

print("Points")

for i in range(0,n):

x,y=input().split()

x=int(x)

y=int(y)

arx.append(x)

ary.append(y)

ynum=min(ary)

ynumi=ary.index(ynum)

xnum=arx[ynumi]

print(xnum)

print(ynum)

arx.remove(xnum)

ary.remove(ynum)

for h in range(0,n):

for k in range(0,n-2):

ttheta1=math.atan(ary[k]-ynum)/(arx[k]-xnum)

ttheta2=math.atan(ary[k+1]-ynum)/(arx[k+1]-xnum)

d1=math.degrees(ttheta1)

d2=math.degrees(ttheta2)

if(d1<0):

d1=d1+180

elif(d2<0):

d2=d2+180

if(d1>d2):

arx[k+1]=arx[k]+arx[k+1]

arx[k]=arx[k+1]-arx[k]

arx[k+1]=arx[k+1]-arx[k]

ary[k+1]=ary[k]+ary[k+1]

ary[k]=ary[k+1]-ary[k]

ary[k+1]=ary[k+1]-ary[k]

arx.insert(0,xnum)

ary.insert(0,ynum)

#print(arx)

#print(ary)

stack=[]

stack.append(0)

stack.append(1)

print("stack")

#print(stack)

#print("rest")

for g in range(2,n):

l=len(stack)

v2=stack[l-1]

v1=stack[l-2]

ar1=[]

ar1.append(arx[g]-arx[v1])

ar1.append(ary[g]-ary[v1])

ar2=[]

ar2.append(arx[v2]-arx[v1])

ar2.append(ary[v2]-ary[v1])

v=np.cross(ar2,ar1)

#print("v")

#print(ar1)

#print(ar2)

print(v)

if(v>0):

stack.append(g)

else:

stack.pop()

stack.append(g)

print(stack)

for f in stack:

print("%s %s" % (arx[f],ary[f]))

Complexity: 0(n logn)