Anthony Hernandez
COT4521

Assignment 1 Report

Pseudo code

Algorithm ConvexHull
Input: list of points S of form (x,y) of size n
Output: Sequences of edges CH that form a convex hull around the given points

```
ConvexHull(S)

    Remove duplicates(s)

    sort(s) by increasing x-coord, if equal x's then increasing y-coord

    return recHull(S)
```

Algorithm: RecHull
Input: sorted list of points S of form (x,y) of size n
Output: Sequences of edges CH that form a convex hull around the given points

```
 RecHull(S)

    if len(S) == 3

        #this is done through calculating determinants of 3 points
        draw clockwise triangle

    else if len(S) == 2
        draw line

    else

        leftHull = RecHull(S(0…n/2))

        lighthull = RecHull(S(n/2…n))

    return Hull = Merge(leftHull, rightHull)
```

Algorithm: Merge
Input: lH, rH where lH and rH are left convex hull and right convex hull respectively

```
    # for any given edge between lH and rH, an edge is considered extreme if the adjacent vertices
    of the vertices making up the edge are all to the rightOf the given edge assuming these edges
    are drawn clockwise
    #determining leftof/rightof is done through representing points as vectors and calculating
    determinants
    upper = Find upper extreme edge of lH and rH by looking at adjacent vertices for any given edge
    drawn between lH and rH.
```

lower = Find lower extreme edge of lH and rH by looking at adjacent vertices for any given edge drawn between lH and rH.

#in the actual implementation, this involves rather complex slicing of lists in python, however the general idea is rather simple and should suffice in pseudocode

return newHull where newHull is lH and rH rewritten by joining the two hulls through upper and lower

NOTE: Algorithm: ConvexHull() serves as a wrapper function to pre-process a given list of points, the actual heart of the program lies within the algorithm: RecHull()

Example Cases

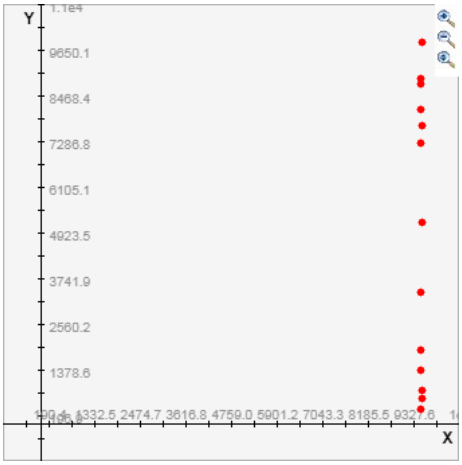
The following example cases were for the most part done on coordinates that ranged from 1 to 100, that is, for any given point, the x-coordinate and y-coordinate was 1 to 100. However, the first example case to be discussed lies far outside this range and was actually created as a response to a challenge proposed to a classmate wherein the challenge was to break my implementation by any means.

Case 1: extreme edge case

Assume the following Hulls when attempting a merge:

leftHull= [(9497, 1364), (9499, 8913), (9506, 8773), (9505, 3404), (9499, 354), (9498, 1911)]
rightHull= [(9507, 7236), (9507, 8129), (9516, 9847), (9518, 7705), (9517, 5197), (9514, 849), (9513, 653)]

When attempting to merge these two hulls, my implementation entered an infinite loop. The cause of which was quickly determined to be finding an extreme edge. These sets of points create an instance where an upper extreme edge could never be found between the hulls, that is, for any given edge, between leftHull and rightHull, there was always an adjacent vertex to the edge that was leftof the edge.



Attempts were made to recreate this circumstance on a more manageable range of coordinates (1 to 100) to debug my logic but no matter how many points were introduced or how many instances of coordinates were created and tested, the infinite loop never occurred.

The image to the left shows what these points plotted appear as in a coordinate system. Looking at the graph, it's safe to say these points are an extreme edge case and will occur very rarely. Nonetheless, this case proves that my implementation is not entirely correct and so cannot be dismissed.

Anthony Hernandez
COT4521

Figure 1: Plot of points that caused an infinite loop

Case 2: extreme collinearity

The next case is another edge case involving collinearity. A square was drawn to see how the convex hull would be built.

Given the set of points: [(0,0), (0,1), (0,5), (2,5), (4,5), (4,4), (4,2), (4,0), (3,0), (2,0)]
Returned convex hull: [(0,0), (0,1), (0,5), (4,5), (4,0), (2,0)]

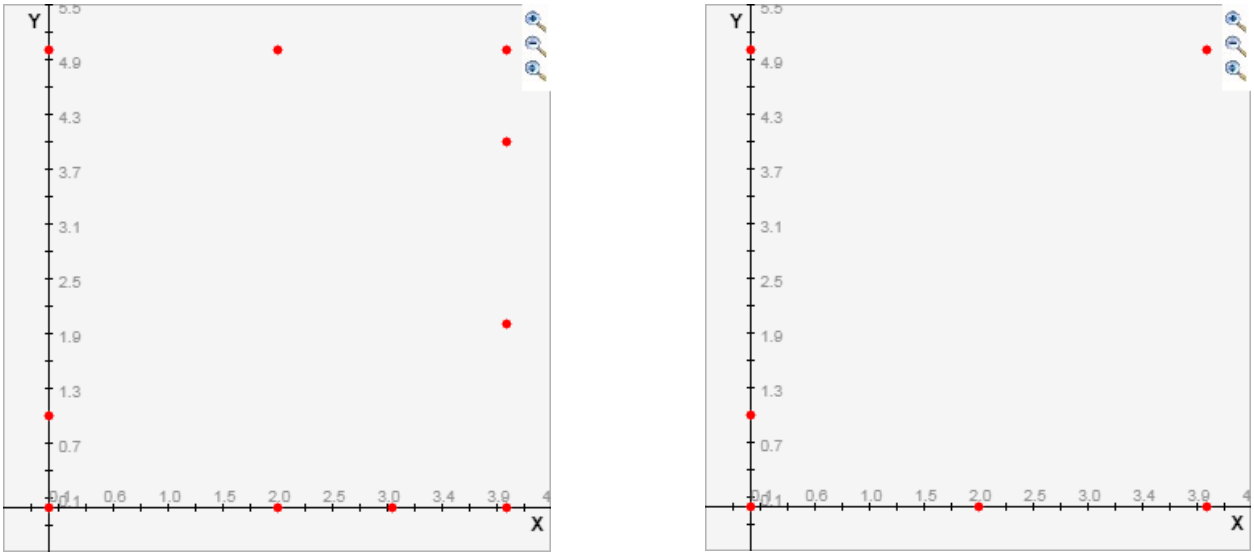


Figure 2: plot of points forming a square(left) and plot depicting the hull formed from the points on a square (right)

Looking at the plots it's immediately clear that there is inconsistency in how the edges for the hull are created. When creating a convex hull, edges that are collinear should either be merged into a singular edge made up of the furthest possible vertices along the edges or every edge should be recognized. My implementation does neither, and instead seems to almost choose at random which edges will make up the convex hull. However, technically speaking, the list of edges returned still creates a convex hull as all points in the original set of coordinates still lie on the returned list of edges. I would make the argument that this returned hull is "good enough" in the sense that it still a correct convex hull despite inconsistencies in naming edges.

Furthermore, my implementation of calculating a convex hull still works well for most cases that don't exhibit extreme behaviors as can be seen in case 3.

Case 3: typical set of points

Anthony Hernandez
COT4521

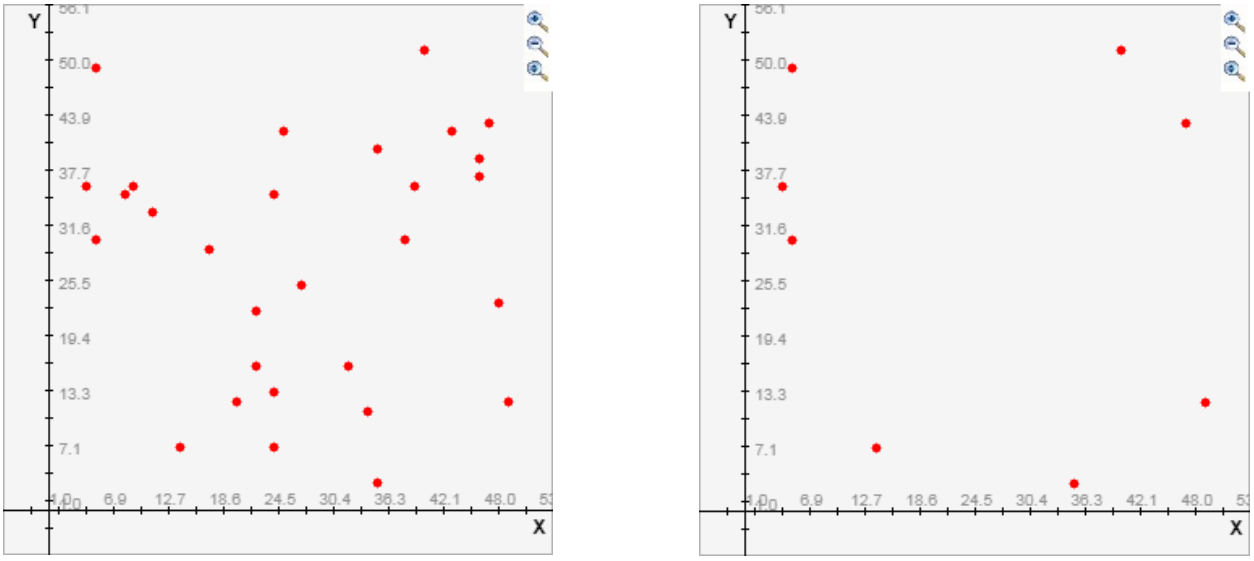


Figure 3: Plot of points depicting average scenario of scattered points(left) and resulting convex hull(right)

Returned convex hull: [(4, 36), (5, 49), (40, 51), (47, 43), (49, 12), (35, 3), (14, 7), (5, 30)]

Case 3 focuses on the average case I tested my implementation on. These cases were sets of points typically between 20-40 total points where the value of coordinates could range from 30-100. These conditions were chosen as they could lead to easy verification of returned hulls.

As can be seen by the left plotting of points, there several instances of pairs of points that are horizontal or vertical, yet these cause no trouble when calculating the convex hull. Furthermore, the list of points that were returned are in clockwise order, with no duplicates and cyclical, that is, the last element forms an edge to the first element.

Lastly, as was mentioned these cases were the kind my implementation was tested on the most. The implementation was considered satisfactory when the convex hull of over a dozen consecutive sets of points was correctly calculated.

Case 4: connected extreme edges

Given the set of points: [(0, 5), (3, 6), (1,4), (4, 3), (2, 2)]
Returned convex hull: [(0, 5), (3, 6), (4, 3), (2, 2)]

Anthony Hernandez
COT4521

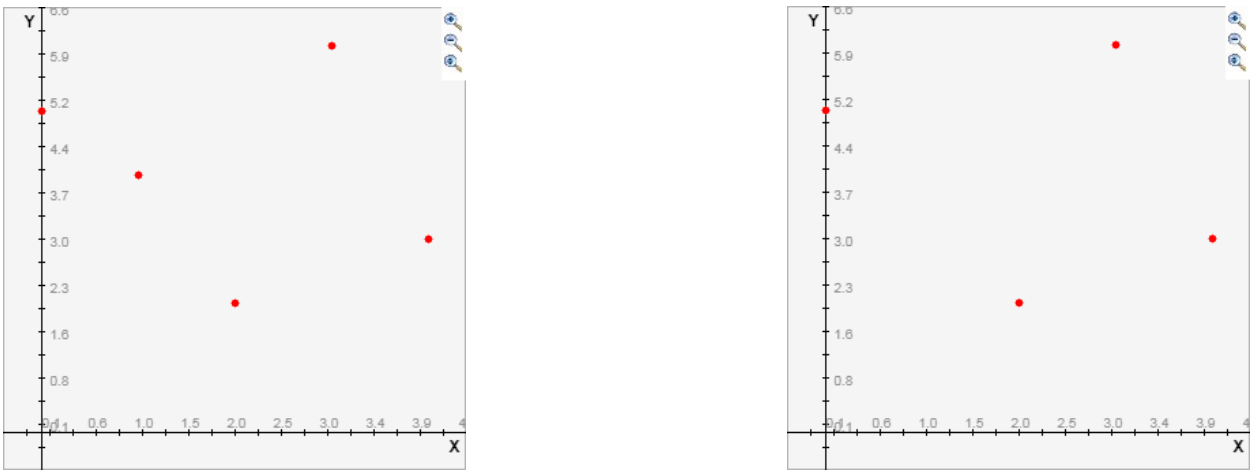
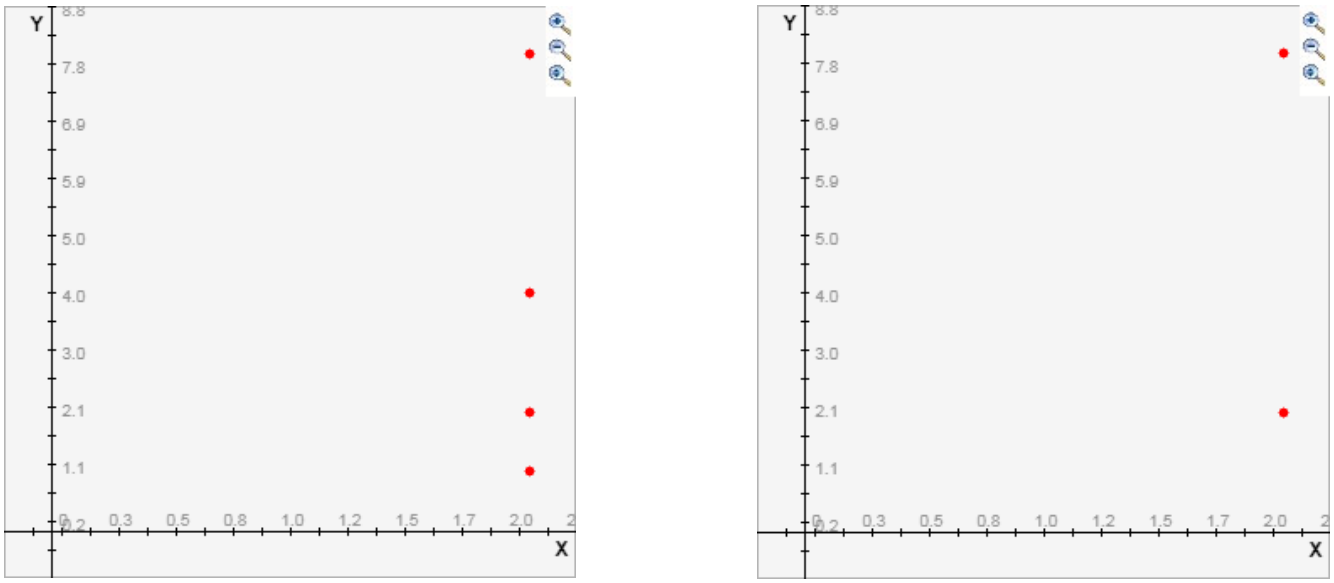


Figure 4: Plot of a line and triangle (left) and resulting hull from the shapes (right)

Case 4 resembles one of the first unexpected issues ran into during testing. It reflects when an upper extreme edge shares a vertex with a lower extreme edge. In the left plot, the leftmost 2 points formed a line and the right most 3 points formed a triangle. The upper extreme edge is [(0,5), (3,6)] and the lower is [(2,2), (0,5)] and it can be seen then that the point shared is (0,5).

Although the correct hull is returned, this case is what started careful review of how the list of edges was rewritten when combining a leftHull and rightHull. My implementation was done in python and relied heavily on lists which can be sliced. The method to rewriting the lists that makeup the hulls involved slicing by hull lists by indices of the extreme edges' vertices. When the indices were shared by the edges then incorrect lists were returned. For this reason, the most complex part of my implementation revolves around rewriting the lists as can be seen by the many conditions written in the Merge() function.

Case 5: Vertical lines
Given the set of points: [(2, 1), (2, 2), (2, 4), (2, 8)]
Returned convex hull: [(2,2), (2,8)]



Lastly, case 5 may not ever be an expected case when calculating convex hulls however it is still worth mentioning as it further demonstrates behavior exhibited in case 2. Once again, the collinearity causes strange behavior in the implementation, however this sort of case was not accounted for as it was presumed that a vertical line should never be given as input.

However, one possible convex for this line could be the edge formed by the furthest 2 vertices, or all the edges listed in connected order, neither of which does my implementation produce. Instead 2 points seem to be chosen at random and returned as a convex hull. On a larger set points that aligned vertically, the returned hull was still random in appearance but with more returned edges

Plot of time

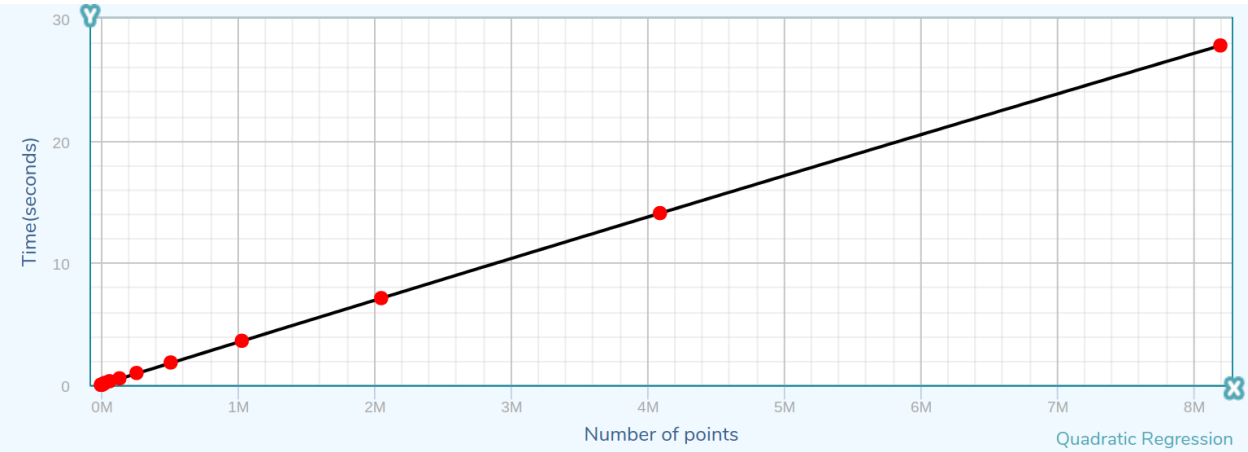


Figure 5: Plot of time(seconds) versus number of points used to calculate convex hull

The figure above was represents the time taken to generate random points as well as compute the convex hull of the points. The starting number of points was 1024 and was doubled each time until the time taken to finish execution was over a minute. The approximation is quadratic polynomial with linear scaling of points. It was seen that when changing the x-coordinates scaling to logarithmic, the resulting curve was exponential, and at the moment, the scaling for x-coordinates is linear and the resulting approximation is a linear curve.

One thing to note that seemed off was testing the time taken to computer the convex hull and not include time taken to generate points. At 10million points, the time taken for a convex hull was roughly 2 seconds.

T