DAA Assignment Report

Group Details

Shubhankar Vivek Shastri – 2020A7PS2054H

Tanmay Agarwal – 2020A7PS2057H

Abhiraj Khare – 2020A7PS0161H

DCEL Implementation

The DCEL consists of three types of records:

1. Vertex records: Each vertex record stores the coordinates of a vertex and a list of incident edges.
2. Edge records: Each edge record stores the two vertices that the edge connects, a pointer to the edge that is next in clockwise order around the first vertex, a pointer to the edge that is next in clockwise order around the second vertex, and pointers to the two faces that are adjacent to the edge.
3. Face records: Each face record stores a pointer to one of the edges that bounds the face.

The DCEL data structure is efficient for a number of operations on planar subdivisions, including traversing the subdivision, finding the neighbours of a vertex or face, computing the intersection of two subdivisions, and many others.

For the given paper, we are trying to partition the given arbitrary polygon into convex partitions using DCEL. For implementation of the algorithm we do not need the information of faces. To implement vertices we have created three structs – Vertex (storing x,y and leaving Halfedge), Halfedge(storing origin vertex, twin Halfedge and next, previous Halfedge). A halfedge is basically an edge that has been divided into 2 halfedges. To store the input vertices of the polygon, we have created a class DCEL, which has two main vectors – vertices of type Vertex, halfedges of type Halfedge. Class DCEL has two primary functions – add_vertex and add_halfedge to take input and store in DCEL.

Algorithm Explanation

This algorithm is used to partition a given simple polygon with n vertices into convex polygons. The input to the algorithm is a set of vertices of the polygon, provided in clockwise order, stored in a cyclic list P. In our case we store the input points in the vertices vector of class DCEL and we push the index of each point in the vector P. We continue to work on index and access the data points vis the vertices vector whenever required.

The algorithm works by maintaining a set of vertices in l that do not contain any notches – where notch is a point that makes the polygon non convex. At each iteration, it selects (v_i-1, v_i+1), where v_i is the current vertex under consideration (In the code we have taken v_i0, v_i and v_i1 to store the indices of these vertices), and checks if any of these points are a notch by the isAcuteangle function defined above in the code. Only if these angles are all less than 180 that we move the vertices forward and add a new vertex upcoming in p to the list l.

The algorithm continues this process until there are only three vertices remaining in P. At this point, the polygon is already convex, and the algorithm terminates.

In case a notch is detected in the previous step, we then define a list lpvs which is a set of vertices in p but not in l and also consists only and only of notches.

If after the above operations the size of LPVS is greater than 0 the algorithm proceeds to remove the vertices in LPVS that are outside the rectangle formed by the x_min, x_max, y_min, y_max coordinates in L_m, and then checks if any vertex now left in LPVS lies inside the polygon generated by L_m. If there is such a vertex, the algorithm obtains a set VTR of vertices of L_m in the semi plane generated by v1 and v(where v is the first point of LPVS inside the polygon). We now pop those vertices from L_m that lie to the same side of the semi plane as that of L_last. As soon as we find a vertex in L_m that is on the opposite side of the semi plane we generate a convex polygon with the current vertices of L_m and update P and iteratively continue the algorithm.

If no vertex in LPVS lies inside the polygon generated by L_m, the algorithm sets the flag Backward to false. The algorithm then continues to the next vertex v_i+1.

If the last vertex in L_m is not equal to the vertex v_2, the algorithm writes L_m as a convex polygon of the partition and removes the vertices of L_m from P. The algorithm then proceeds to the next convex polygon, starting with a new L_m.

The output of the algorithm is a set of convex polygons that partition the input polygon.

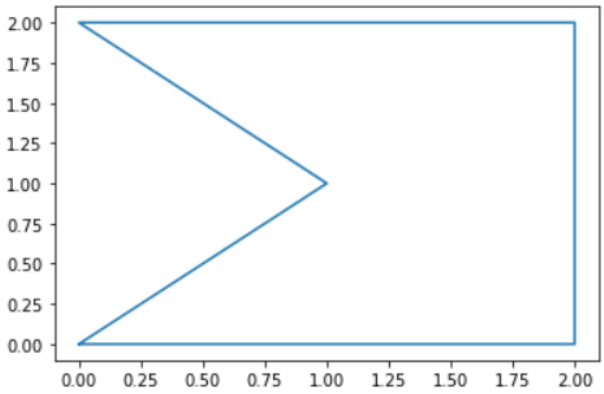
Analysis

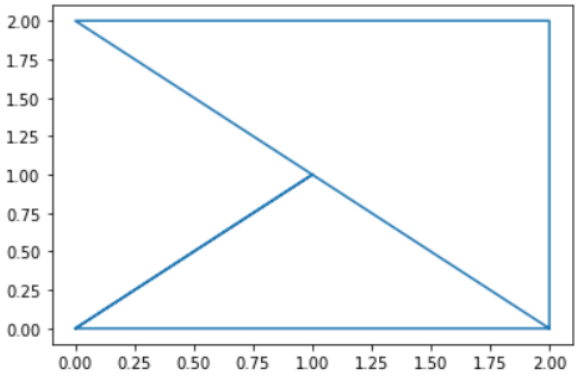
First we will generate various polygons via the python visualizer. We will run the same points of vertices on the cpp code which will generate the indices of vertices to be joined to generate the convex polygons.

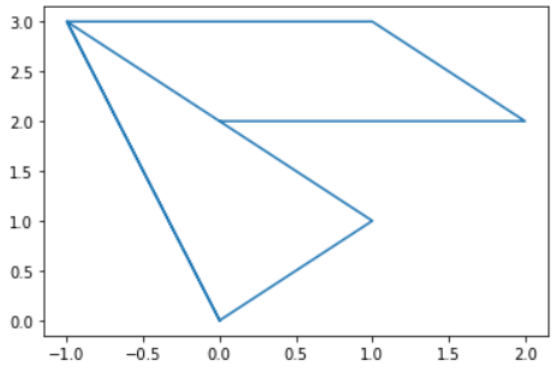
We have divided the analysis into 2 parts. In the first part we have run the code and visualizer for multiple polygons and shown the input and output images for reference. In the second part of the analysis we have run the code on a similar type of polygon by increasing its vertices every time to analyse the no of vertices vs time taken by the algorithm to run. We have then plot a time_taken vs no_of_vertices graph to show the conclusion and analysis.

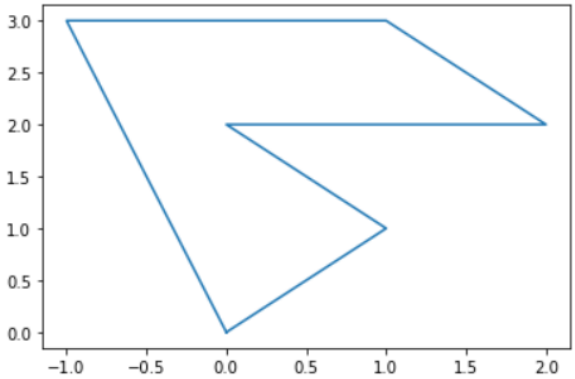
Running the code on various polygons and generating initial and final polygon diagrams via visualizer to check for correctness

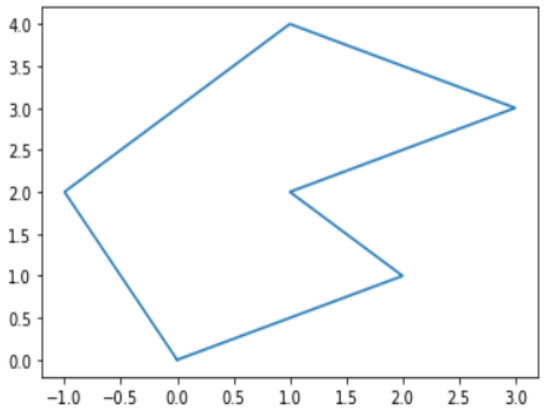
For every polygon image below there is a convex polygon decomposition which follows that image

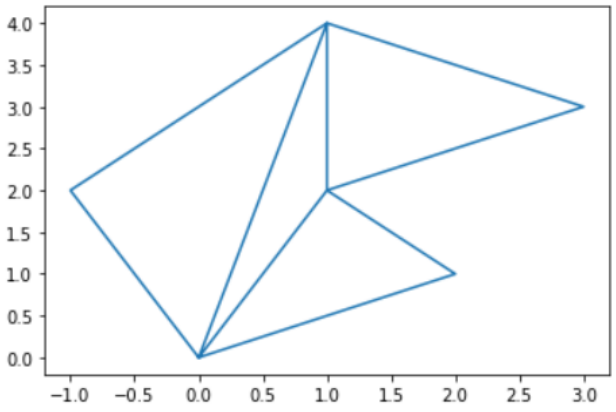


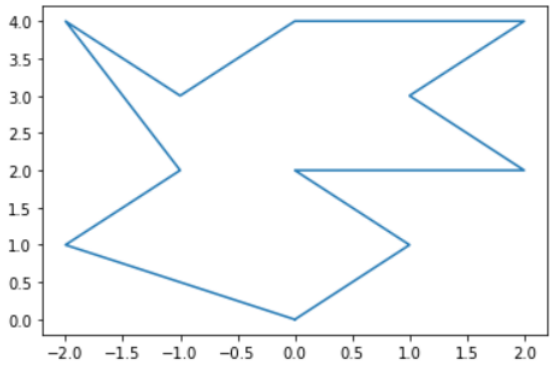


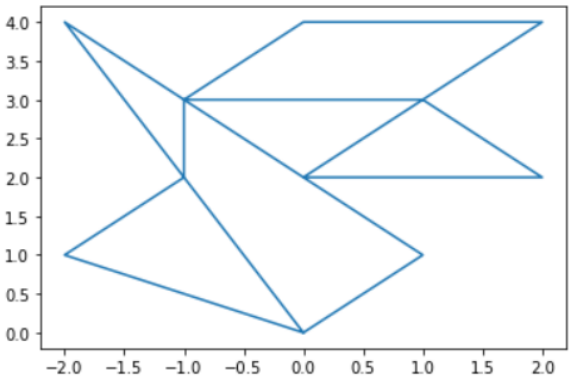


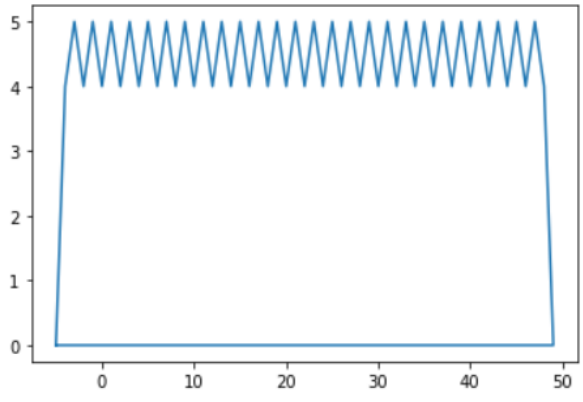


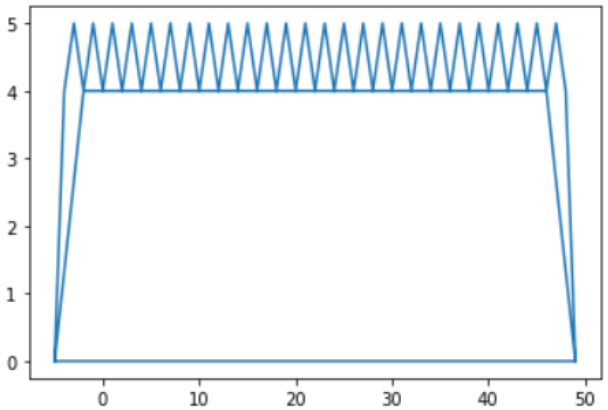

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0 −0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0

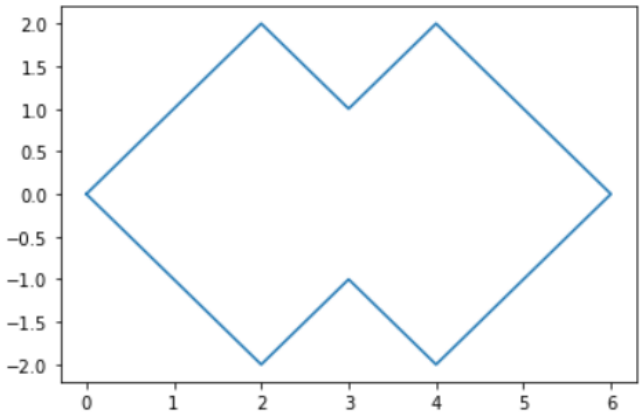
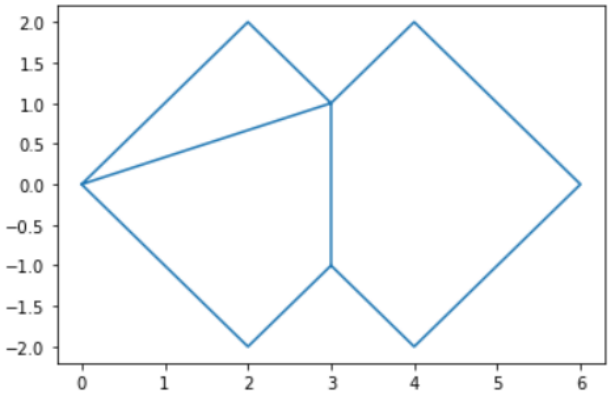
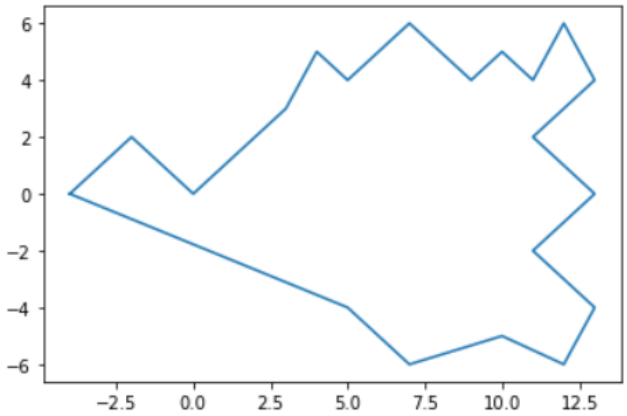
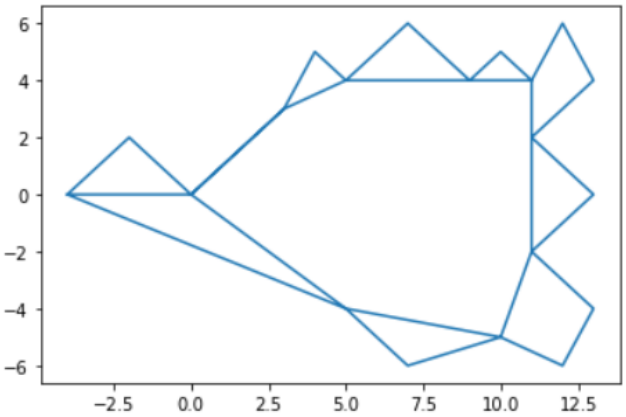
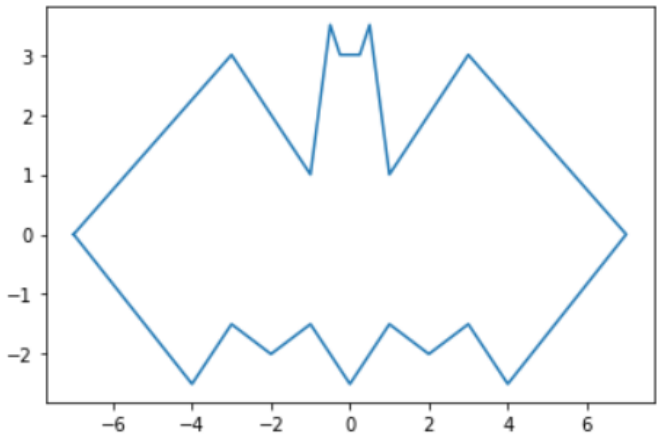
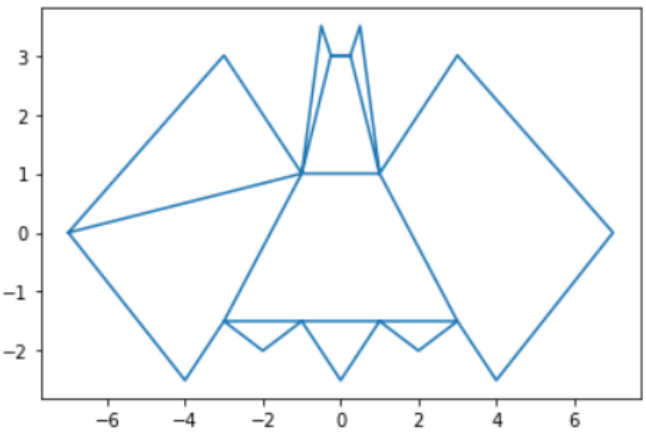

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.0 −0.5 0.0 0.5 1.0 1.5 2.0 2.5 3.0

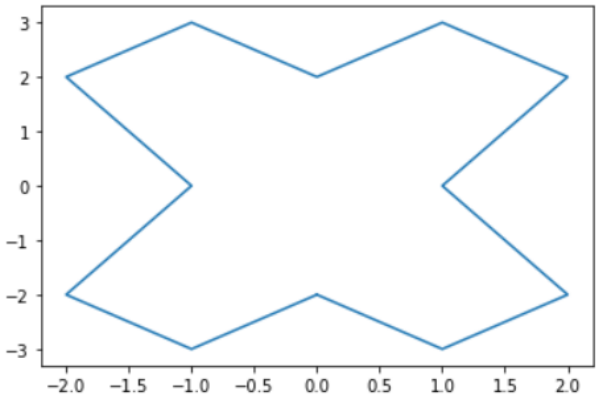

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

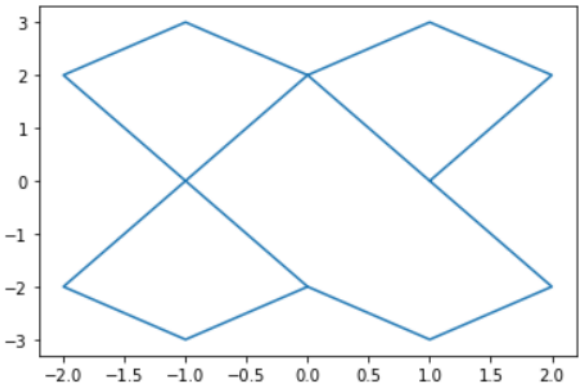

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

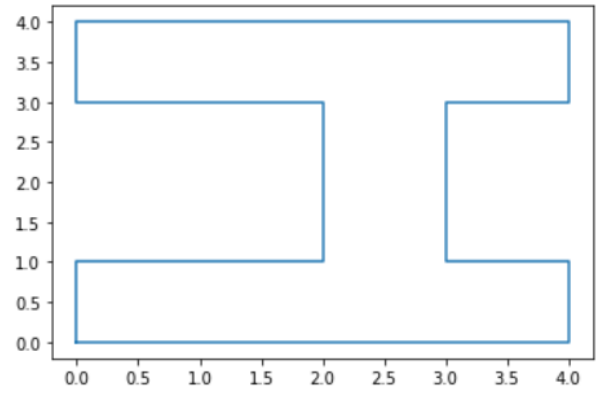

5
4
3
2
1
0
0 10 20 30 40 50

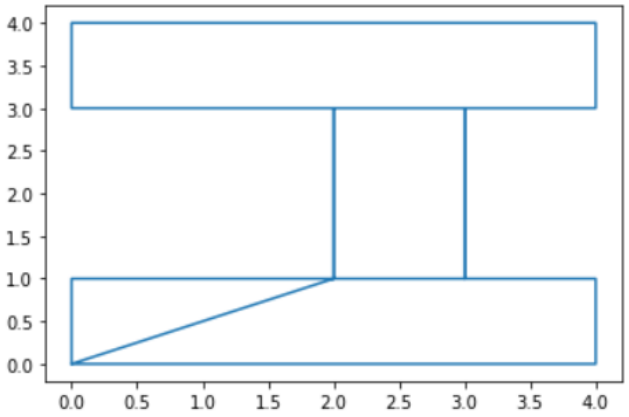

5
4
3
2
1
0
0 10 20 30 40 50

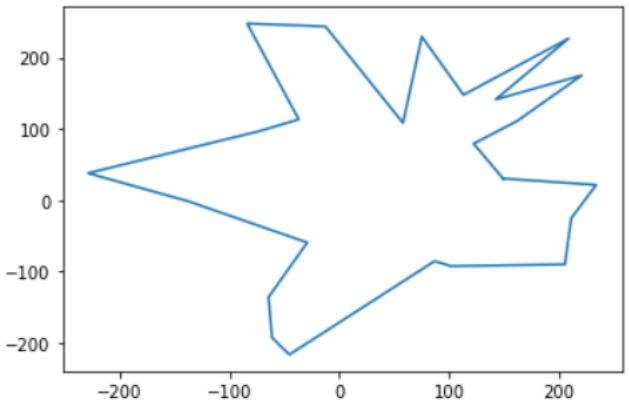







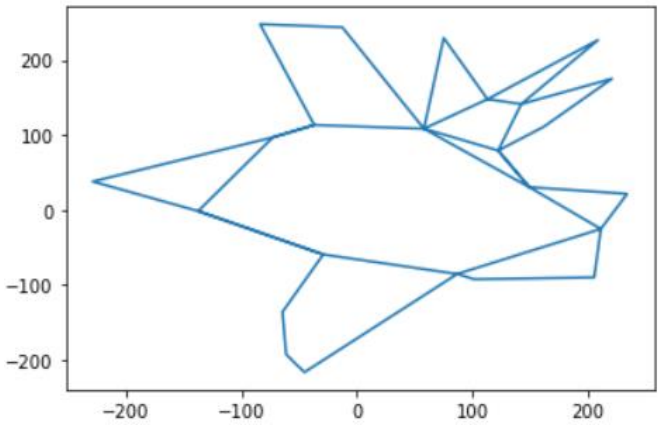

3
2
1
0
−1
−2
−3
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

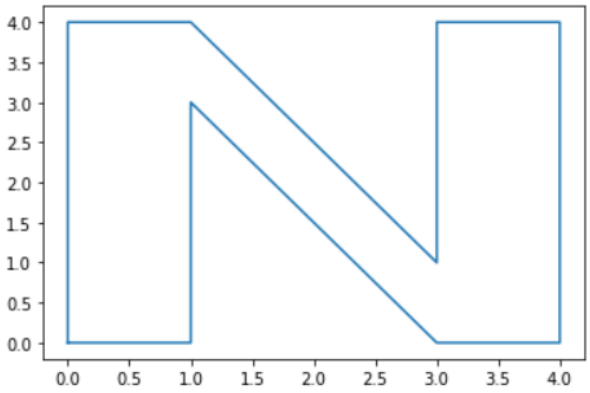

3
2
1
0
−1
−2
−3
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

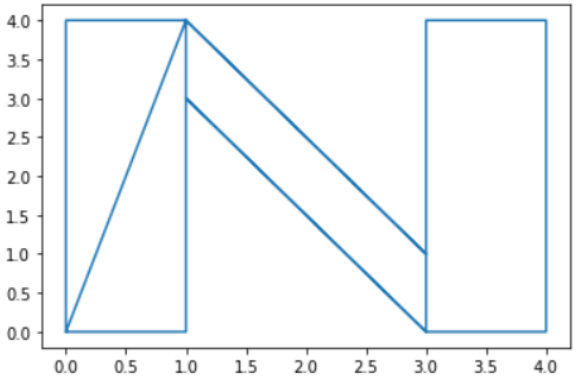

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

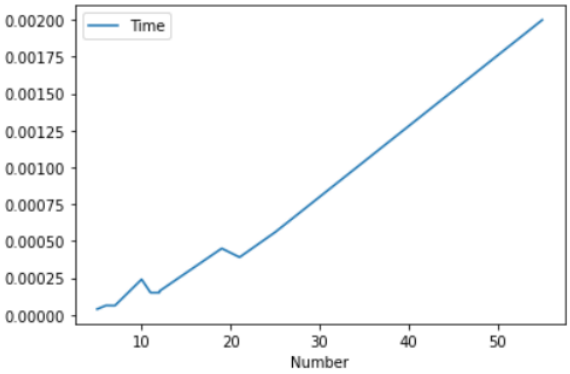

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


200
100
0
−100
−200
−200
−100
0
100
200


200
100
0
−100
−200
−200
−100
0
100
200



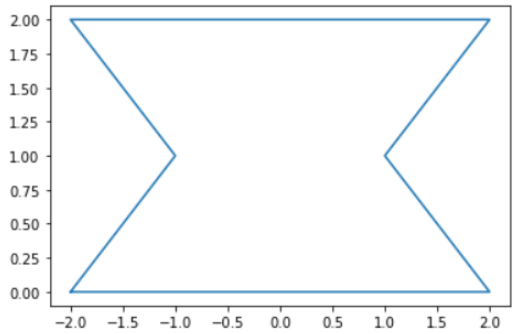


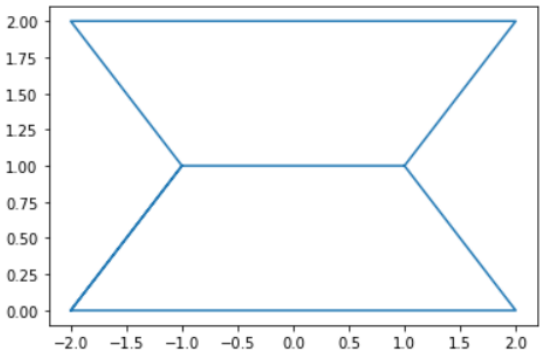


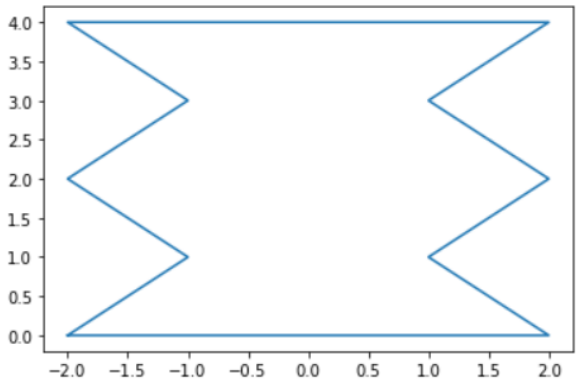
Time analysis graph (time taken vs no of vertices) for different kinds of polygons(meaning different complexity).

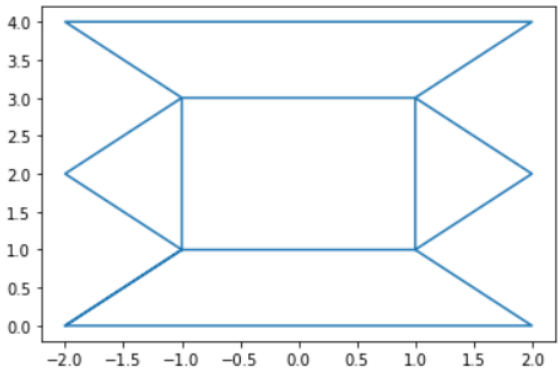
Analysis of number of vertices vs the time taken by the algorithm for the same type of polygon

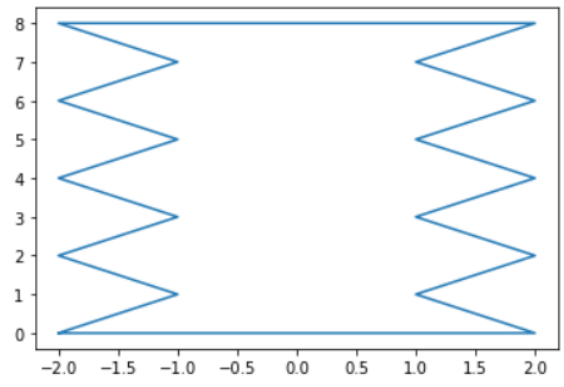
We have attached the input and output images till n=26 to show how the output looks. As similar pattern is seen in the later inputs as well, hence we have noted only the time taken by the algorithm to plot time taken vs no of vertices.

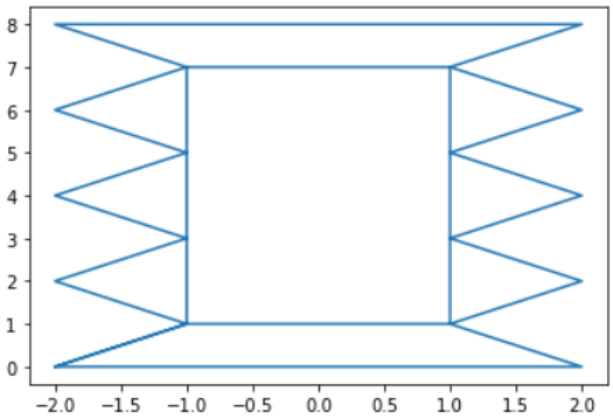

2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

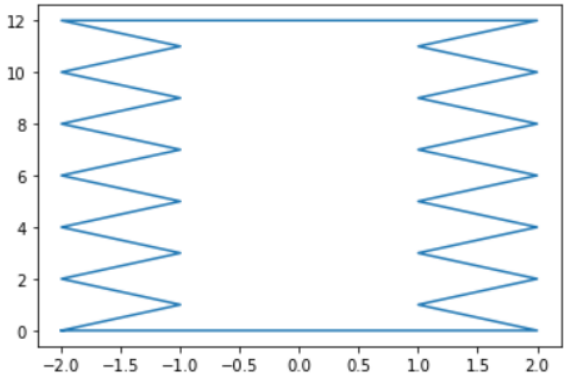

2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

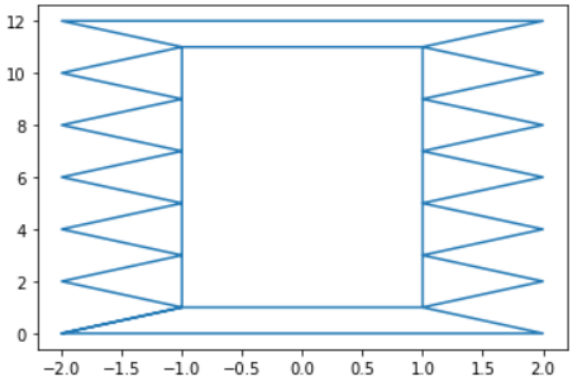

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

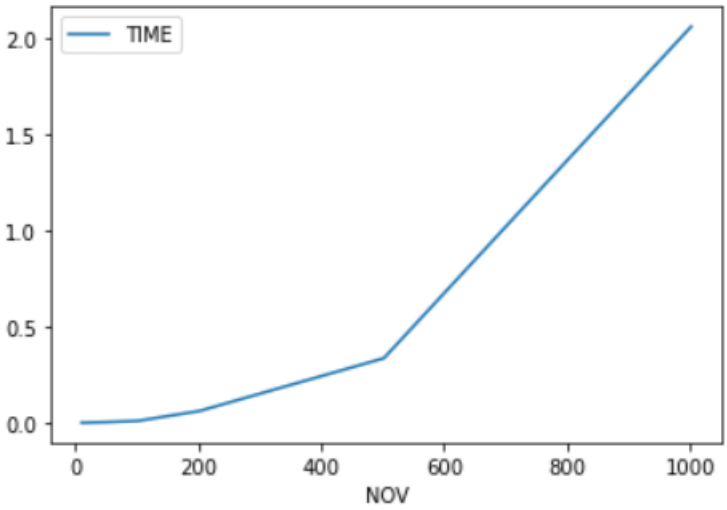

4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


8
7
6
5
4
3
2
1
0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


8
7
6
5
4
3
2
1
0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0







As can be seen seen in the final graph plotted above, as we increase the number of vertices in the initial polygon the time taken by the algorithm for decomposition increases almost linearly. Also, we had tried to use the polygon points in such a way that the number of notches also keep increasing with n(that is number of vertices) and hence we can possibly conclude that as the number of notches increase the time taken also increases linearly.

Although the algorithm works almost fine for many test cases, a major drawback of the algorithm is that if any 3 points in the polygon are collinear then either an erroneous output is generated or no output is generated because of an infinite loop being created.