ME570 Assignment 1

Zili Wang
zw2445@bu.edu

September 29, 2022

Contents

1 Question 1.1

Code reference:

- polygon_plot.m
- polygon_isFilled.m

Figure 1a shows the counter-clockwise square starting from the origin, and Figure 1b show its plot after the filled-in check. Similarly, Figure 1c shows the clockwise square started from the origin, which has an opposite flowing direction to Figure 1a, and Figure 1d show its plot after the filled-in check. Gray color indicates the blocked area and white color is the open area.

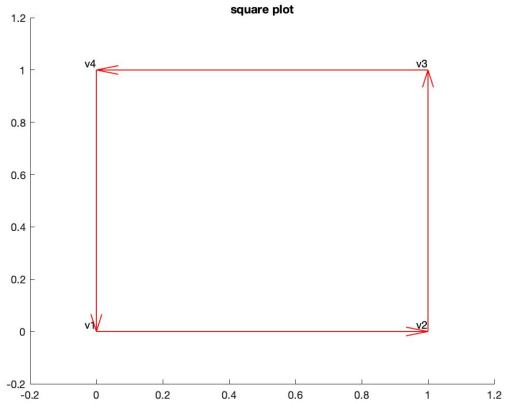
In the function polygon_isFilled() , Schoelace formula is applied based on the description in:

https://www.element84.com/blog/determining-the-winding-of-a-polygon-given-as-a-set-of-ordered-points

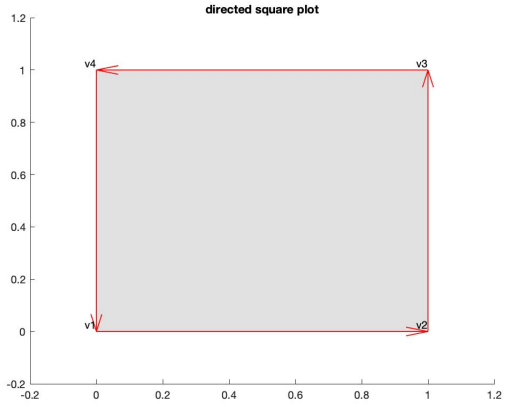
Basically, the Schoelace formula is used to calculate the area inside the polygon and can be expressed as

$$
\begin{aligned}
A &= \frac{1}{2}\left|\sum_{i=1}^{n} \det\begin{pmatrix} x_i & x_{i+1} \\ y_i & y_{i+1} \end{pmatrix}\right| \\
&= \frac{1}{2}\left|\sum_{i=1}^{n} (x_{i+1} + x_i)(y_{i+1} - y_i)\right|
\end{aligned}
$$

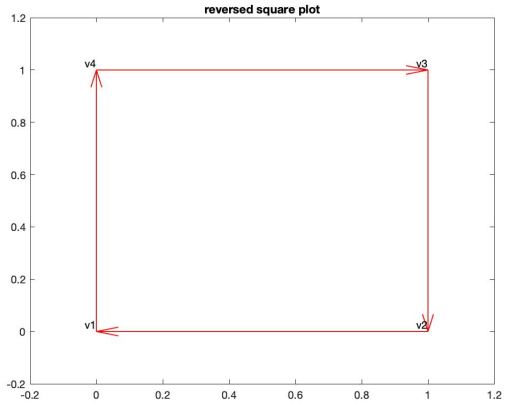
where the sum of the determinants gives the direction of polygon. If the sum is positive, the points are labelled sequentially in counter-clockwise direction, and vice-versa.



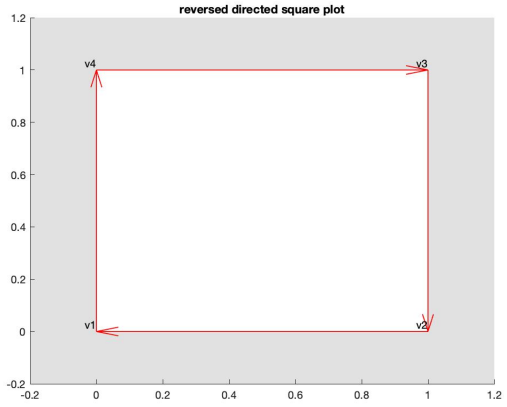
(a) Counter-clockwise Square



(b) Filled-in Square



(c) Clockwise Square



(d) Hollow Square

Figure 1: Polygon Filled-in and Plot Demo

2 Question 1.2

Code reference:

- edge_isCollision.m
- edge_angle.m

Figure 2 shows how edge_angle() works.

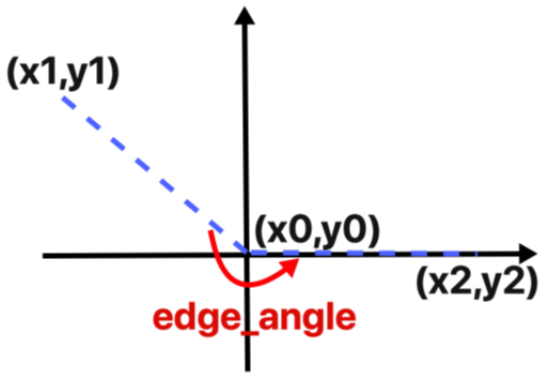


Figure 2: edge angle demo

sAngle gives the sine value of the edge angle, and cAngle gives the cosine value of the edge angle, only the tangent value can give the complete information of magnitude and direction of the angle. To examine the function,

- edge_angle($[0;0]$, $[-1;1]$, $[0;1]$) return -0.7854
- edge_angle($[0;0]$, $[-1;1]$, $[0;1]$, 'unsigned') return 5.4978

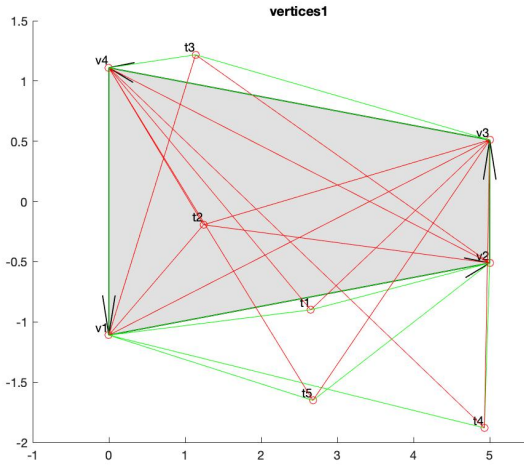
3 Question 1.3

Code reference:

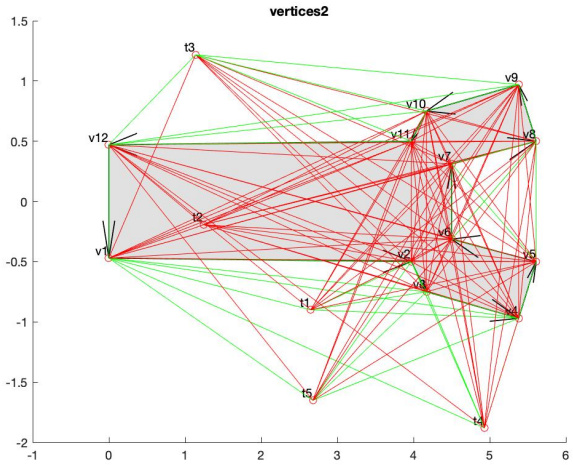
- polygon_isSelfOccluded.m
- polygon_isVisible.m
- twolink_polygon.m
- plotLinesFlag.m

- polygon_isVisible_test.m

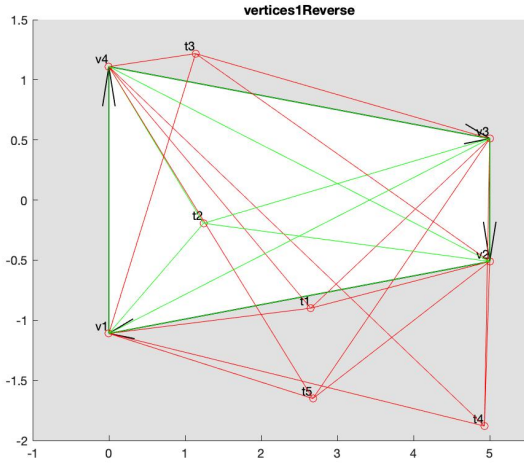
Figure 3 illustrates the results from polygon_isVisible_test(). For each polygon, each vertex together with five random points are selected as the test points. For each polygon vertex, the visibility of each test point is checked. The visibility is defined such that there is no self-occlusion and self-occlusion no edge intersection. A green line means the test point is visible from the corresponding vertex, and red line means invisible.



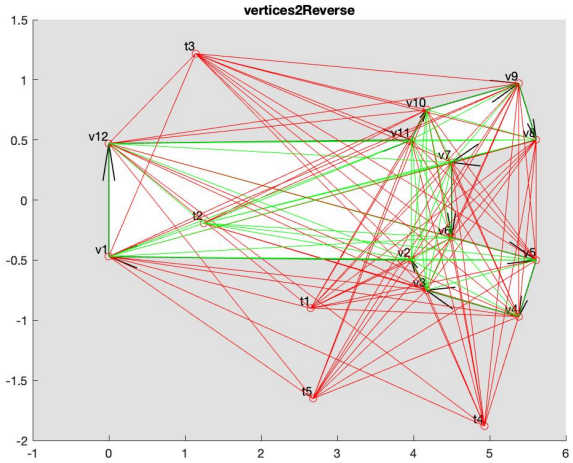
(a) vertices1



(b) vertices2



(c) reversed vertices1



(d) reversed vertices2

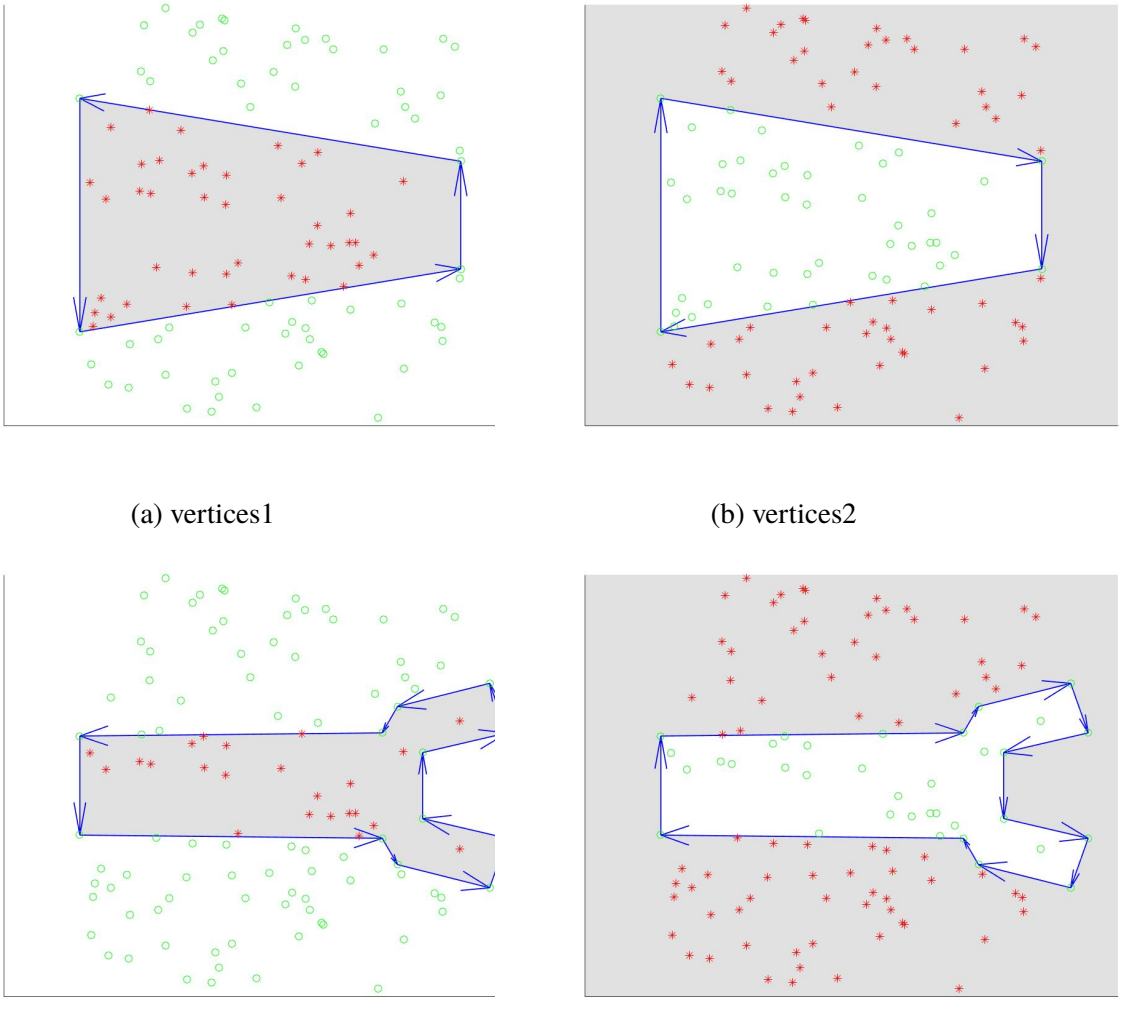
Figure 3: Test Point Visibility Demo

4 Question 1.4

Code reference:

- polygon_isCollision.m
- plotPointsFlag.m
- polygon_isCollision_test.m

Figure 4 illustrates the results from polygon_isCollision_test(). For each polygon, the vertices together with 100 random points are selected as the test points. The collision of a test point with a polygon is defined such that it is not visible from any polygon vertex. A green circle point means the test point collides with the polygon, and the red point means it collides.



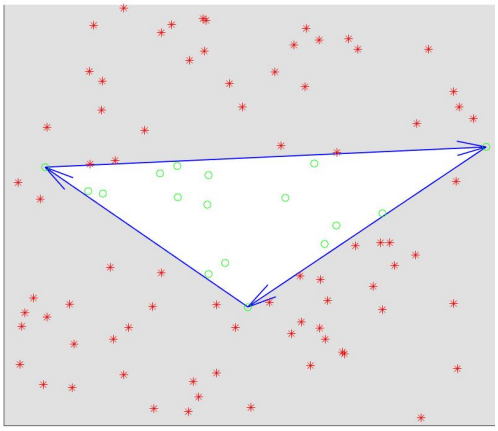
(a) vertices1

(b) vertices2

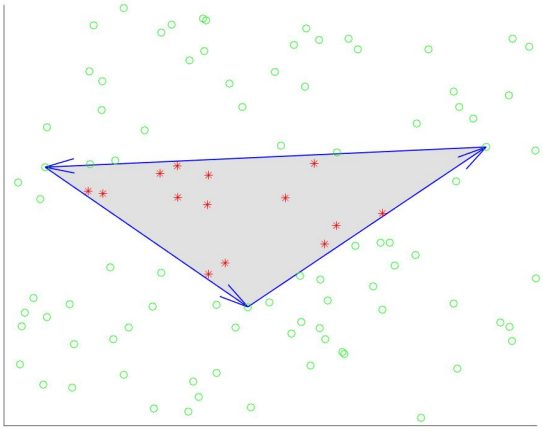
(c) reversed vertices1

(d) reversed vertices2

Figure 4: 100 Test Points Collision demo



(e) reversed vertices1



(f) reversed vertices2

Figure 4: 100 Test Points Collision demo (cont.)

5 Question 2.1

Code reference:

- priority_prepare.m
- priority_insert.m
- priority_minExtract.m
- priority_isMember.m
- priority_test.m

Below is the diary file (matlabDiary.txt) recorded from priority_test.m . The queue is first initialized to be empty, then three elements are added, after that, the elements with minimum cost is sorted out, and another new element is added. Then the the element key is checked and all elements are cleared.

```
empty queue with key:  and cost:
new element with key: a and cost: 5
new element with key: true and cost: 10
new element with key: c and cost: 3
minimum element is with key: c and cost: 3
new element with key: true and cost: 10
is element with key: c present?  no
is element with key: true present?  yes
all elements are removed!
empty queue with key:  and cost:
```

A diary of command lines is saved in 2_1.txt and can be found in the code files directory.

6 Question 2.2

The pseudo code can be:

Data: 2-D array M with size (k,j), each element indicating the cost at that coordinate

Result: 1-D array pQueueNew with ascending cost

either keep 2-D M or flatten it to 1-D

pQueue = priority_prepare()

pQueueNew = priority_prepare()

for *iRow = 1:k* **do**

  for *iColumn=1:j* **do**

    pQueue = priority_insert()

  end

end

for *i = 1:k*j* **do**

  pQueue = priority_minExtract()

  add the returned minimum key and cost to pQueueNew

end

Algorithm 1: Change Grid to structured Array with Orderings

In this case, it requires $O(n^2)$ operations. Instead of using priority_minExtract(), using QuickSort requires $O(n\log(n))$ operations.

7 Question 3.1

Code reference:

- qp_minEffort.m
- qp_test.m

Below is the diary of running qp_test.m, from which four set of parameters are tested and the results are printed.

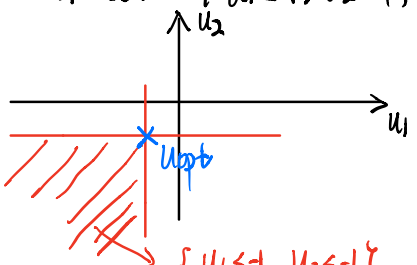
```
qp_test
Case 1: the optimal u is [-1.00,-1.00] and delta is 0.00
Case 2: the optimal u is [-0.33,-0.33] and delta is -0.33
Case 3: the optimal u is [0.00,0.00] and delta is -0.00
Case 4: the optimal u is [-0.00,-0.00] and delta is -0.00
```

$u_{opt}, b_{opt} = \underset{u \in \mathbb{R}^2, b \in \mathbb{R}}{\arg\min} \|u\|^2 + mb^2 \quad \Rightarrow u_1^2 + u_2^2 + mb^2$

$$s.t \begin{cases} A_{attr}\, u + b_{attr} + b \le 0 \\ A_{barrier}\, u + b_{barrier} \le 0 \end{cases}$$

(1) constraints: $\begin{cases} u_1 + u_2 + 1 + b \le 0 \\ u_1 \le -1 \\ u_2 \le -1 \end{cases}$

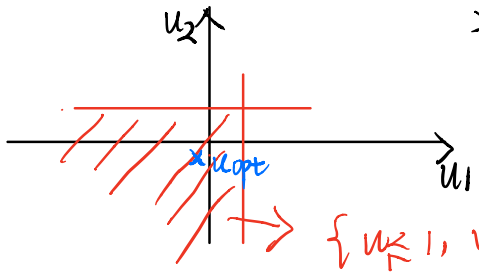
feasible set: $\{ u_1 \le -1, u_2 \le -1, b \le \underbrace{-u_1}_{\ge 1} \underbrace{-u_2}_{\ge 1} - 1 \}$



since both u_1, u_2 are negative, $b=0$ will give the minimum of $(u_1^2 + u_2^2 + b^2)$, and $u_1 = u_2 = 1 \Rightarrow u_{opt} = (-1,-1)$ $b_{opt} = 0$

(2) Constraints $\begin{cases} u_1 + u_2 + 1 + b \le 0 \\ u_1 \le 1 \\ u_2 \le 1 \end{cases}$

feasible set: $\{ u_1 \le 1, u_2 \le 1, b \le \underbrace{-u_1}_{\ge -1} \underbrace{-u_2}_{\ge -1} - 1 \}$



in this case, u_1, u_2, b are simultaneously moving to 0

$u_1 = u_2 = b = -\frac{1}{3}$

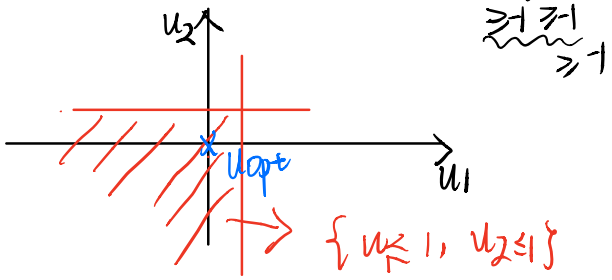
$u_{opt} = (-\frac{1}{3}, -\frac{1}{3})$ $\quad b_{opt} = 0$

(3) constraints

$$\begin{cases} u_1 + u_2 + b - 1 \le 0 \\ u_1 \le 1 \\ u_2 \le 1 \end{cases}$$

feasible set: $\{u_1 \le 1, u_2 \le 1, b \le -u_1 - u_2 + 1\}$

$-u_1 - u_2 \ge -1 - 1 = -1$



the minimum of $u_1^2 + u_2^2 + b^2$ is at $u = (0, 0)$ $b = 0$, which is feasible in the case

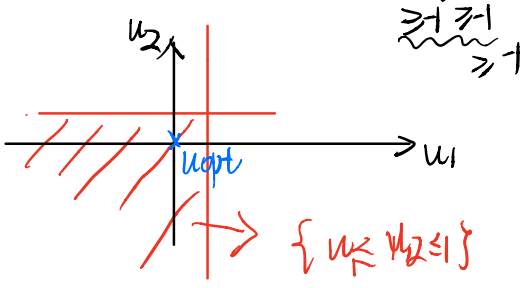
$\therefore$ $u_{opt} = (0, 0)$ $b_{opt} = 0$

(4) constraints

$$\begin{cases} u_1 + u_2 + b - 1 \le 0 \\ u_1 \le 1 \\ u_2 \le 1 \end{cases}$$

feasible set: $\{u_1 \le 1, u_2 \le 1, b \le -u_1 - u_2 + 1\}$

$-u_1 - u_2 \ge -1 - 1 = -1$



objective: $u_1^2 + u_2^2 + 1000b^2$

more effort should be taken to minimize $b^2 \Rightarrow b \to 0$

$\therefore$ $b_{opt} = 0$

$$\begin{cases} u_1 + u_2 \le 1 \\ u_1 \le 1 \\ u_2 \le 1 \end{cases}$$

$u_{opt} = (0, 0)$

OR

the constraints are the same as (3)

$\therefore$ if the minimal of (3) and (4) is the same, the optimal value is the same

8 Question 4.1

Coding time allocations:

- Section 1: 3 hours + 1 hour debug
- Section 2: 40 minutes
- Section 3: 30 minutes

Writing time: 1 hour
In total: approximately 6 hours

The hard part of this homework is that there are many concepts, and it is not easy to follow at the start. Sometimes the question is not straightforward to understand.