Solutions for HW2

Yunhai Han

January 21, 2020

1 E2.1 Convexity

1.1 i

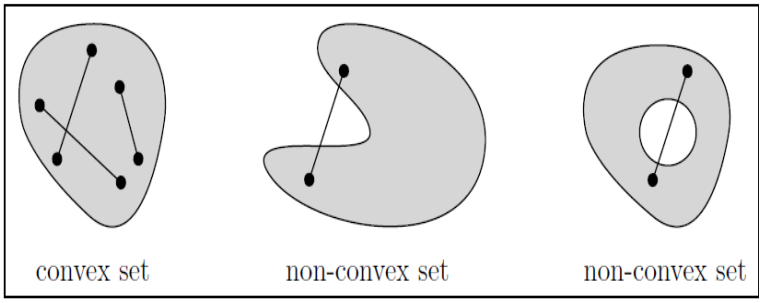

Figure 1: Convex set(from lecture notes by Bullo and Smith)

A set S is convex if for any two points p and q in S, the entire segment $\overline{pq}$ is also contained in S. Examples of convex set and non-convex set are drawn in Figure 1.

The advantage of convex sets over non-convex ones for motion planning is that if the start point and the goal point belong to the same convex set, then the segment joining the two points is an obstacle-free path, which makes the motion planning problem easier to deal with.

1.2 ii

1.2.1 a

The intersection of any two overlapping convex sets is also convex.

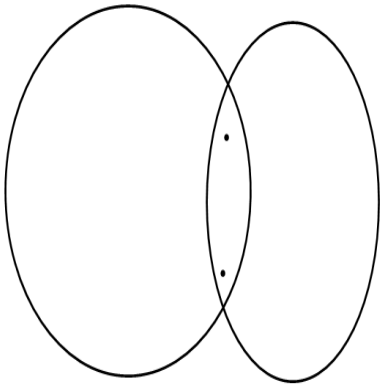

Figure 2: A sketch supporting my statement

From the above figure, the two circles represent two different convex sets. It is obvious that the segment joining any two points in the intersection of the two circles is still in the intersection.

1.2.2 b

The union of two convex sets is not convex, so the statement is false. The proof shows in the following figure.

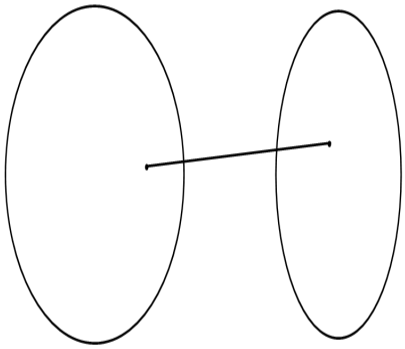

Figure 3: A sketch supporting my statement

The two points are in two different convex sets, so they are both in the union of the two sets. However, it is obvious the segment joining the two points is not wholly within the union.

1.2.3 c

This statement is true.

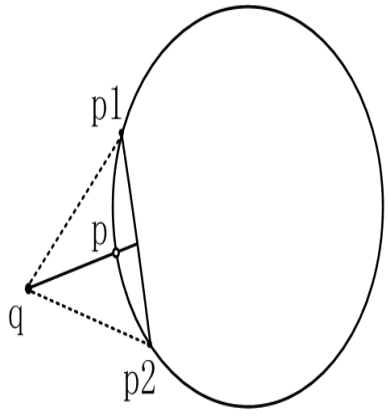

Figure 4: A sketch supporting my statement

In the above figure, if we assume p_1 and p_2 are the two points which are closest to q, we could always find a new point p which is even more closer to q and this point unique(if it is not unique, we could do it again and find a new one). I describe the reasons in the next section.

1.2.4 d

If Q is not convex, the property doesn't hold.

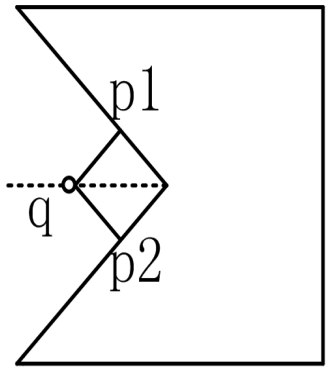

Figure 5: A sketch supporting my statement

From the above figure, it is obvious that in such a case, there are two different points in Q and the distance between point q and each point p are smaller than the distance point q and any other points in Q.

1.2.5 e

The statement is false.

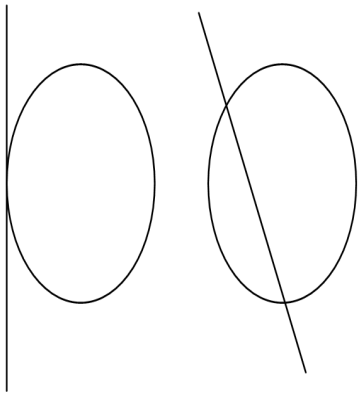

Figure 6: A sketch supporting my statement

In the above figure, I draw a line tangent to a convex set and it doesn't contain any point in Q. It is obvious that Q is not divided into two or more components. In other words, all the points in Q are either on the left or the right of the line. In the other hand, I draw a line that divides Q into two or more components and it could not be tangent to the Q because it contains points in the interior of Q.

1.3 iii

1.3.1 a

The reason is very simple: Take any two points p, q in the intersection and by definition of intersection, they are also in each convex set. The line $\overline{pq}$ joining the two points must also lie wholly within each set. Hence it must lie wholly within their intersection. Hence, the statement is true.

1.3.2 c

I could use the contradiction method to prove it:

First, I assume there exist more than one such points q that the distance between between p and q are all smaller than the distance between p and any other points in the set.

Then, I select any two points from all points p and connect them. Since the two points are in a convex set, the segment joining the two points is also in the convex set. The midpoint of the segment is more closer to the point q than the two selected points.

Finally, I draw a new segment joining the point q and the midpoint. The intersection of this segment and the boundary of the convex set is indeed what we want. If there are more than two candidates, we could implement this method iteratively(each time we need to select two different points from the group) and there would be only one point left after the final iteration. This point is what we really want to find and it is unique.

And this is done in figure 4 to find the right point.

1.3.3 e

I could also use the contradiction method to prove it: First, I assume such a tangent line divides Q into two or more components.

Then, I could select two different points from two different components. Because the two points are in the convex set, the segment joining them are also in the convex set.

The segment mush intersect with the tangent line due to the fact that the two points are on the different sides of the tangent line. The intersection point is either on the tangent line or in the convex set, which contradicts with the title(it says such a line does not contain any point in the interior of Q).

Hence, the statement is false.

2 E2.5 Computing time complexity

From the hint, it assumes that addition, multiplication and division of two numbers all have unit cost.

2.1 i

1. $a = 0$

2. for i from 1 to n:

3. $\quad a = a + 1$

For the above programs, it would take $1 + n$ operations required for execution. Hence, the time complexity is $O(n)$.

2.2 ii

1. $a = 0$

2. for i from 1 to 10:

3. $\quad a = a + 1$

For the above programs, it would take $1 + 10$ operations required for execution. Hence, the time complexity is $O(1)$, which means the running time would not increase with the scale.

2.3 iii

1. $a = 0$

2. for i from 1 to n:

3. $\quad a = a + 1$

2. for j from 1 to n:

3. $\quad a = a + 1$

For the above programs, it would take $1 + 2n$ operations required for execution. Hence, the time complexity is also $O(n)$.

2.4 iv

1. $a = 0$

2. for i from 1 to n:

3. $\quad$ for j from 1 to n:

4. $\quad\quad a = a + 1$

For the above programs, it would take $1 + n * n$ operations required for execution. Hence, the time complexity is also $O(n^2)$.

2.5 v

1. $a = 0$
2. for i from 1 to n:
3. for j from i to n:
4. $a = a + 1$

For the above programs, it would take $1 + \sum_{i=1}^{n}(n + 1 - i) = 1 + \frac{n^2+n}{2}$ operations required for execution. Hence, the time complexity is also $O(n^2)$.

2.6 vi

1. $a = n$
2. while $a \geq 1$:
3. $a = a/2$

For the above programs, it would take $1 + \lceil 2 \log n \rceil + 1$ operations required for execution. I assume the base of $\log n$ is 2 and n is larger than 1. Here, I give a brief explanation of the expression:

At first, we initialize a and it requires one step and obviously it would scale with input n. Then, we divide the variable by 2 and check whether it is less than 1 or not. This process could take $\lceil 2 \log n \rceil$ steps and the multiplier 2 means both the division and check would take 1 step for each time. After $\log n$ divisions, we could always obtain a value which is between 1 and 2. The special case is when n is a multiple of 2. In this case, the value after $\log n$ divisions is 1 and from the title, we know there is no need for more divisions. However, when n is not a multiple of 2, the value would be larger than 1 but smaller then 2 and $\log n$ could be a float. We still need to do one more check and division. Hence, I always round up $\log n$. Finally, we still have to check whether the value is not larger than 1. And this time, we know that it would be less than 1, so it just jumps out of the while-loop.

The time complexity is $O(\log n)$.

3 E2.9 Programming Project:The sweeping trapezoidation algorithm

3.1 classify each obstacle vertex

Because each polygonal obstacle is described by a counter-clockwise sequence of vertices, the sequential three vertices could represent two segments, which intersects with each other at the second vertex. Also, I think it is important to assume that no obstacle segment is vertical, which means all x-coordinates of vertices are unique.

In order to find the relative position of each vertices, we could just compare each vertex's x-coordinate, which is very easy for implementation.

In order to find the convexity of each vertex, I draw a picture to help me explain how I implement it.

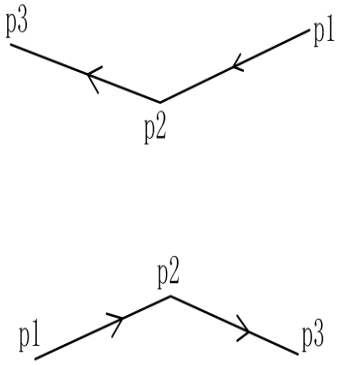

Figure 7: A sketch supporting my statement

In the above figure, the top two lines correspond to the upper boundary of any polygon and the bottom two lines correspond to the lower part. If a vertex is convex, for the bottom two lines, p_3 must be above the line passing through p_1 and p_2. Similarly, for the top two lines, p_3 must be below the line passing through p_1 and p_2. It is very quick to check whether a point is above or below a given line expression. Besides, I could use the function in the previous section to help me figure out the line equation given any two points. By comparing the x-coordinates of p_1 and p_2, I could also quickly distinguish which case it is.

I put the codes below, which looks very similar to python codes:

```
for index in all_polygons:
   polygon=all_polygons[index]
   for vertex in range(0,len(polygon)):
      if(vertex==0):
         to_vertex=polygon[1]
         from_vertex=polygon[-1]
      else if(vertex==len(polygon)-1):
         to_vertex=polygon[0]
         from_vertex=polygon[-2]
      else:
         to_vertex=polygon[vertex+1]
         from_vertex=polygon[vertex-1]
      mode_leftright=Checkleft_right(polygon[vertex],to_vertex,from_vertex)
      mode_convexity=CheckConvex(polygon[vertex],to_vertex,from_vertex)
      switch(mode_leftright):
         case 1: #right/right
            if(mode_convexity==convex):
               return mode3
            else:
               return mode4
         case 2: #left/left
            if(mode_convexity==convex):
               return mode1
            else:
               return mode2
         case 3: #left/right
            if(mode_convexity==convex):
```

```
                return mode5
            else:
                return mode6
def Checkleft_right(p_m,p_t,p_f):
    if(p_m.x<p_t.x and p_m.x<p_f.x):
        return 2
    else if(p_m.x>p_t.x and p_m.x>p_f.x):
        return 1
    else:
        return 3
def CheckConvex(p_m,,p_f):
    if(p_f.x<p_m.x): #the bottom two lines
        (a,b,c)=computeLineThroughTwoPoints(p_m,p_f): #from homework1
        y=(-a*p_t[0]-c)/b #b is not 0 since the line is not vertical
        if(y>=p_t[1]):
            return non-convex
        else:
            return convex
    else: #the top two lines
        (a,b,c)=computeLineThroughTwoPoints(p_m,p_f): #from homework1
        y=(-a*p_t[0]-c)/b #b is not 0 since the line is not vertical
        if(y=p_t[1]):
            return non-convex
        else:
            return convex
```

4 E1.7 Programming:Polygons

4.1 computeDistancePointToPolygon

First, I need to check whether the given point is inside the polygon or not. I could use a method described in the lecture before. That is if a point is inside a polygon, the number of intersection points should be an odd number, when I draw a vertical line downwards from the point. The following picture is for illustration.

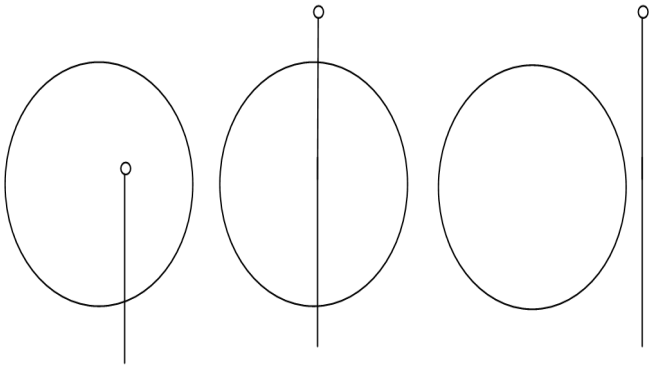

Figure 8: Illustration

The left example corresponds to the case when the point is inside the polygon and the middle one and the right one shows the other case. You can see from the figure that the number of intersection points is odd only when the point is inside the polygon. Hence, I could use this property to check it.

In order to find the smallest distance from q to the closest point in P(polygon), I could find all the smallest distances from q to each edge of the polygon. There are two possible cases:the point that makes the distance smallest is one of endpoints of or on the edge. Because the closest point in P must be on the boundary of the polygon, the smallest one among all the smallest distances from q to each edge is indeed the smallest distance from q to the polygon. When I implement the function, I could use the functions and the codes I have already built for the homework1. The special cases involve when some of the edges are vertical or horizontal because I have to use another way to obtain the distance. However, it is easier than the one I use for the general cases.

4.2 computeTangentVectorToPolygon

From the above section, I already have shown how to decide q is closest to a segment of the polygon or a vertex.

For the case(i), what I need to do next is to find the directional vector of that segment. Since u is parallel to the segment, they would have the same directional vector(up to a scale).

For the case(ii), what I need to do next is to connect q and the vertex and compute the slope of the new segment. Because u is orthogonal to the new segment and its slope is known, I could obtain the directional vector of u easily(up to a scale).

Finally, as the title requires, I need to normalize the vector and make it lie in the counter-clockwise direction(remove the effect of the scale and the sign). Normalization is very easy for implementation. And since we all know that all the vertices points are ordered in the clockwise direction of the polygen, I could compare the direction of the selected edge and the direction of the vector I obtain to make it lie in the counter-clockwise direction(the included angle should be larger than $\pi/2$).

4.3 Verification

I change the input and check whether the results are right.

```
if __name__=='__main__':
    q=[1,0.5]
    mypolygon=[[0,0],[0,2],[2,2],[1,1],[2,0]]
    if(checkPointInsidePolygon(q,mypolygon)):
        tmp=computeDistancePointToPolygon(q,mypolygon)
        computeTangentVectorToPolygon(q,tmp)
        plt.scatter(q[0],q[1],s=10)
        polygon=plt.Polygon(mypolygon,fc="r")
        plt.gca().add_patch(polygon)
        plt.axis([-5,5,-5,5])
        plt.show()
```

Figure 9: Results

```
RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework 2\python_program_hw2.py
The point is inside the polygon.
'''
```

Figure 10: Results

These two pictures show the output of my function when the given point is inside the polygon.

```
if __name__=='__main__':
    q=[1,1]
    mypolygon=[[0,0],[0,2],[2,2],[1,1],[2,0]]
    if(checkPointInsidePolygon(q,mypolygon)):
        tmp=computeDistancePointToPolygon(q,mypolygon)
        computeTangentVectorToPolygon(q,tmp)
        plt.scatter(q[0],q[1],s=10)
        polygon=plt.Polygon(mypolygon,fc="r")
        plt.gca().add_patch(polygon)
        plt.axis([-5,5,-5,5])
        plt.show()
```

Figure 11: Results

```
'''
RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework 2\python_program_hw2.py
The point is on the one of the vertices.
'''
```

Figure 12: Results

These two pictures show the output of my function when the given point is one of the vertices of the polygon.

```
if __name__=='__main__':
    q=[1,2]
    mypolygon=[[0,0],[0,2],[2,2],[1,1],[2,0]]
    if(checkPointInsidePolygon(q,mypolygon)):
        tmp=computeDistancePointToPolygon(q,mypolygon)
        computeTangentVectorToPolygon(q,tmp)
        plt.scatter(q[0],q[1],s=10)
        polygon=plt.Polygon(mypolygon,fc="r")
        plt.gca().add_patch(polygon)
        plt.axis([-5,5,-5,5])
        plt.show()
```

Figure 13: Results

```
RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework 2\python_program_hw2.py
The point is not inside the polygon.
The distance from q to the polygon is: 0
The unit-length vector u is: [-1.0, 0.0]
```

Figure 14: Results

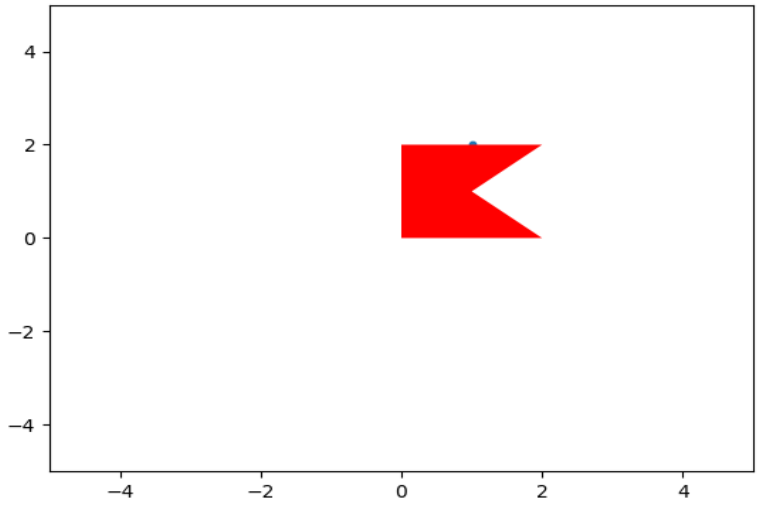

Figure 15: Results

These three pictures show the output of my function when the given point is on the boundary of the polygon.

```
if __name__=='__main__':
    q=[3,4]
    mypolygon=[[0,0],[0,2],[2,2],[1,1],[2,0]]
    if(checkPointInsidePolygon(q,mypolygon)):
        tmp=computeDistancePointToPolygon(q,mypolygon)
        computeTangentVectorToPolygon(q,tmp)
        plt.scatter(q[0],q[1],s=10)
        polygon=plt.Polygon(mypolygon,fc="r")
        plt.gca().add_patch(polygon)
        plt.axis([-5,5,-5,5])
        plt.show()
```

Figure 16: Results

```
 RESTART: C:\Users\Administrator\Desktop\winter quarter\MAE145\Homework\Homework
2\python_program_hw2.py
The point is not inside the polygon.
The distance from q to the polygon is: 2.23606797749979
The unit-length vector u is: [-0.8944271909999159, 0.4472135954999579]
```

Figure 17: Results

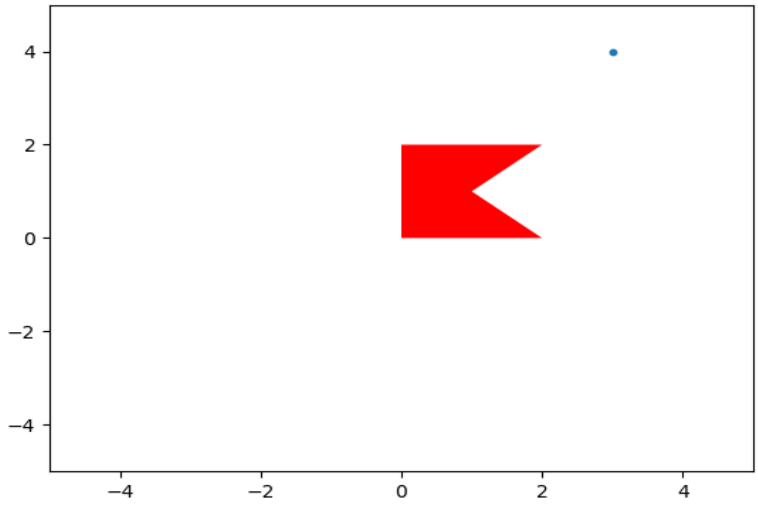

Figure 18: Results

These three pictures show the output of my function when the given point is totally outside the polygon, which are general cases.

For all the possible cases, the results are right.