ME145 Robotic Planning and Kinematics: Lab 1

Line and Segments

Eric Perez

Spring 2024

April 15, 2024

E1.6 Programming: Line and segments

<u>computeLineThroughTwoPoints</u>

The function computeLineThroughTwoPoints was created and my code can be seen below.

```
% Eric Perez
% ME145 Lab 1
function [a,b,c] = computeLineThroughTwoPoints(P1,P2)
     [m1,n1] = size(P1);
     [m2,n2] = size(P2);
    if m1 ~= 1 || n1 ~= 2
        error('Wrong input format for P1');
    end
    if m2 ~= 1 || n2 ~= 2
        error('Wrong input format for P2');
    end

x1 = P1(1);
y1 = P1(2);
x2 = P2(1);
y2 = P2(2);
Mag = sqrt((x2 - x1).^2 + (y2 - y1).^2);
if Mag <= 0.1
    error('The two point are less than 0.1 away from each other')
end

a0 = (y2 - y1)/(x2-x1);
b0 = -1;
c0 = y1 - a0*x1;
s = sqrt(a0.^2 + b0.^2);
a = a0/s;
b = b0/s;
c = c0/s;
end
```

The two points "p1" and "p2" are input into the function. The user must enter both points using the syntax "[# #]". The function includes three correctness checks that check for the correct input format for both points and the distinction of both points (with a tolerance of 0.1) using if statements. The line outputted was defined as $\{(x, y) \mid ax + by + c = 0\}$ so the main equation utilized is the point slope form equation: $y - y_1 = m(x - x_1)$. The points are finally normalized by dividing the parameters by the magnitude.

Example 1:

Input: [a,b,c] = computeLineThroughTwoPoints([1,2],[5,8])

Output:

```
>> [a,b,c] = computeLineThroughTwoPoints([1,2],[5,8])

a =

    0.8321


b =

   -0.5547


c =

    0.2774
```

Example 2:

Input: [a,b,c] = computeLineThroughTwoPoints([4,2],[9,8])

Output:

```
>> [a,b,c] = computeLineThroughTwoPoints([4,2],[9,8])

a =

    0.7682


b =

   -0.6402


c =

   -1.7925
```

Example 3:

Input: [a,b,c] = computeLineThroughTwoPoints([4,2],[8]) (demonstraiting the wrong input format)

Output:

```
>> [a,b,c] = computeLineThroughTwoPoints([4,2],[8])
Error using computeLineThroughTwoPoints (line 10)
Wrong input format for P2
```

Example 4:

Input: [a,b,c] = computeLineThroughTwoPoints([5,9],[5,9]) (demonstraiting no distinction)

Output:

```
>> [a,b,c] = computeLineThroughTwoPoints([5,9],[5,9])
Error using computeLineThroughTwoPoints (line 19)
The two point are less than 0.1 away from each other
```

Example 5:

Input: [a,b,c] = computeLineThroughTwoPoints([10,9],[1,4])

Output:

```
>> [a,b,c] = computeLineThroughTwoPoints([10,9],[1,4])

a =

    0.4856


b =

   -0.8742


c =

    3.0110
```

computeDistancePointToLine

The function computeDistancePointToLine was created and my code can be seen below.

```
% Eric Perez
% ME145 lab 1

function [distance] = computeDistancePointToLine(P1,P2,q)
[m1,n1] = size(P1);
     [m2,n2] = size(P2);
    if m1 ~= 1 || n1 ~= 2
        error('Wrong input format for P1');
    end
    if m2 ~= 1 || n2 ~= 2
        error('Wrong input format for P2');
    end

x1 = P1(1);
y1 = P1(2);
x2 = P2(1);
y2 = P2(2);
q1 = q(1);
Mag = sqrt((x2 - x1).^2 + (y2 - y1).^2);
if Mag <= 0.1
    error('The two point are less than 0.1 away from each other')
end
q2 = q(2);
m = (y2 - y1)/(x2-x1);
a = m;
b = -1;
c = y1 - a*x1;
mq = (-1/m);
aq = mq;
bq = -1;
cq = q2 - aq*q1;
xpq = (c-cq)/(mq-m);
ypq = mq*xpq + cq;

distance = sqrt((ypq-q2).^2 + (q1-xpq).^2);


end
```

Three points are entered into this function: p1, p2, and q. The points p1 and p2 join a line. Each point is formatted as "[# #]". The distance value is the output. An orthogonal projection of point q onto the line is used to compute the distance between it and the line.The point slope form equation($y - y_1 = m(x - x_1)$) is used here to determine the slope. The slope of the new line (slope of q) that is being used to measure the distance has a slope that is equal to the negative inverse of the slope of the line that is joined by points p1 and p2 ($m_q = \frac{-1}{m}$). With this, a new line for measuring distance is created. The point where q intersects with the line made by p1 and p2 is found and is used the measure the distance from q to that intersection. The distance is simply the magnitude of the new line ($\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$) . This function checks if the input size is correct as "[# #]" and if the two points are distinct through if statements as well.

Example 1:
Input: [distance] = computeDistancePointToLine([6,7],[4,4],[1,2])
Output:

```
>> [distance] = computeDistancePointToLine([6,7],[4,4],[1,2])

distance =

    1.3868
```

Example 2:
Input: [distance] = computeDistancePointToLine([9,9],[5,6],[1,2])
Output:

```
>> [distance] = computeDistancePointToLine([9,9],[8,7],[1,2

distance =

    4.0249
```

Example 3:
Input: [distance] = computeDistancePointToLine([9,9],[5],[1,2]) (Demonstaring wrong input format)
Output:

```
>> [distance] = computeDistancePointToLine([9,9],[5],[1,2])
Error using computeDistancePointToLine (line 11)
Wrong input format for P2
```

Example 4:
Input: [distance] = computeDistancePointToLine([9,9],[9,9],[1,2]) (Demonstaring no distinction)
Output:

```
>> [distance] = computeDistancePointToLine([9,9],[9,9],[1,2])
Error using computeDistancePointToLine (line 21)
The two point are less than 0.1 away from each other
```

Example 5:
Input: [distance] = computeDistancePointToLine([90,90],[85,78],[2,4])
Output:

```
>> [distance] = computeDistancePointToLine([90,90],[85,78],[2,4])

distance =

   48.1538
```

computeDistancePointToSegment

The function computeDistancePointToSegment was created and my code can be seen below.

```
% Eric Perez
% ME145 lab 1
function [distance] = computeDistancePointToSegment(P1,P2,q)
[m1,n1] = size(P1);
     [m2,n2] = size(P2);
    if m1 ~= 1 || n1 ~= 2
        error('Wrong input format for P1');
    end
    if m2 ~= 1 || n2 ~= 2
        error('Wrong input format for P2');
    end

x1 = P1(1);
y1 = P1(2);
x2 = P2(1);
y2 = P2(2);
q1 = q(1);
Mag = sqrt((x2 - x1).^2 + (y2 - y1).^2);
if Mag <= 0.1
    error('The two point are less than 0.1 away from each other')
end
q2 = q(2);
m = (y2 - y1)/(x2-x1);
a = m;
```

```
b = -1;
c = y1 - a*x1;
mq = (-1/m);
aq = mq;
bq = -1;
cq = q2 - aq*q1;
xpq = (c-cq)/(mq-m);
ypq = mq*xpq + cq;
minx = min(x1,x2);
maxx = max(x1,x2);
if xpq >= minx && xpq <= maxx
distance = sqrt((ypq-q2).^2 + (q1-xpq).^2);
else
    error('orthogonal projection of q does not fall within segment')
     distance = 1/0;
end
end
```

Three inputs are required for this function: p1, p2, and q. Every point has the following format: "[# #]". The distance value is the output. The function determines the length of the line formed by the points p1 and p2, and the orthogonal projection of q that falls onto the line segment. This is the section that separates the two points. This function ensures that the inputs are [# #] by verifying their size. The point slope form equation($y - y_1 = m(x - x_1)$) is used here to determine the slope and the magnitude equation is used to determine the distance ($\sqrt{(x2\ -\ x1)^2 +\ (y2\ -\ y1)^2}$). This will draw lines from q to the segment and between p1 and p2. This function finds the intersection point of the two lines on the x axis and if it is inside the segment. This function also checks whether the inputs are distinct, the input size is accurate, and whether the segment contains the orthogonal projection of q.

Example 1:

Input: [distance] = computeDistancePointToSegment([32,15],[4,9],[8,6])

Output:

```
>> [distance] = computeDistancePointToSegment([32,15],[4,9],[8,6])

distance =

    3.7715
```

Example 2:

Input: [distance] = computeDistancePointToSegment([10,7],[1,2],[5,5])

Output:

```
>> [distance] = computeDistancePointToSegment([10,7],[1,2],[5,5])

distance =

    0.6799
```

Example 3:

Input: [distance] = computeDistancePointToSegment([10,7],[5,3],[15,9]) (Demonstrating point not in segment)

Output:

```
>> [distance] = computeDistancePointToSegment([10,7],[5,3],[15,9])
Error using computeDistancePointToSegment (line 39)
orthogonal projection of q does not fall within segment
```

Example 4:

Input: [distance] = computeDistancePointToSegment([10],[1,2],[5,5]) (Demonstrating incorrect input format)

Output:

```
>> [distance] = computeDistancePointToSegment([10],[5,3],[15,9])
Error using computeDistancePointToSegment (line 7)
Wrong input format for P1
```

Example 5:

Input: [distance] = computeDistancePointToSegment([10,10],[10,10],[5,5]) (Demonstrating no distinction)

Output:

```
>> [distance] = computeDistancePointToSegment([10,10],[10,10],[15,9])
Error using computeDistancePointToSegment (line 20)
The two point are less than 0.1 away from each other
```

E1.7 Programming: Polygons

computeDistancePointToPolygon

The function computeDistancePointToPolygon was created and my code can be seen below.

```
%Eric Perez
%Lab 1 problem 2

function [minD,MinX,MinY] = computeDistancePointToPolygon(Poly,q)
[m,n] = size(Poly);
if n ~= 2
    error('Columns of Polygon does not equal 2');
end
distance = zeros(1,m);
for i = 1:m
    distance(i) = sqrt((Poly(i,2)-q(2)).^2 + (q(1)-Poly(i,1)).^2);
end


for i = 1:m
    x(i) = Poly(i,1);
    y(i) = Poly(i,2);

end
x(end+1)= x(1);
y(end+1)=y(1);
 plot(x,y,'b');
 hold on;
 plot(q(1),q(2),'o');
 minD = min(distance);
IndexD = find(distance==minD);
MinX = x(IndexD);
MinY = y(IndexD);
end
```

A polygon P and a point q are the two inputs that this function accepts. The format of the polygon P is "[# #; # #; # #]," which is made up of several points. The format for the point q is "[# #]". This function determines the distance between each polygon vertex and the point q separately. Using the lowest distance value, it then establishes which vertex is the nearest. The process involves calculating the lengths of the horizontal and vertical paths connecting each vertex to the point. The magnitude equation is used to determine the distance from each point ($\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$). This function verifies the accuracy of the input size. This is to confirm that there are two columns entered. Additionally, this function shows the potential shape of the polygon by plotting it.

Example 1:

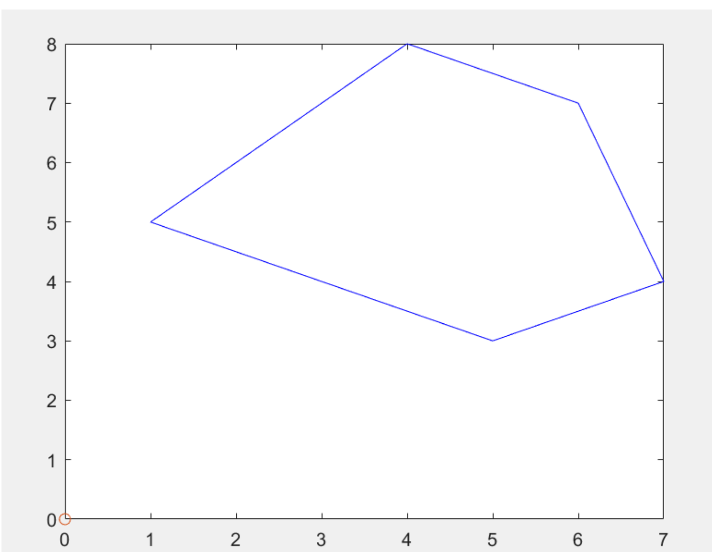
Input: [distance] = computeDistancePointToPolygon([1 5; 5 3; 7 4; 7 10; 4 8],[0,0])

Output:

```
>> [distance] = computeDistancePointToPolygon([1 5; 5 3; 7 4; 6 7; 4 8],[0,0])

distance =

    5.0990
```



Example 2:
Input: [distance] = computeDistancePointToPolygon([5 8; 10 6; 21 12; 28 21; 8 26],[3,4])

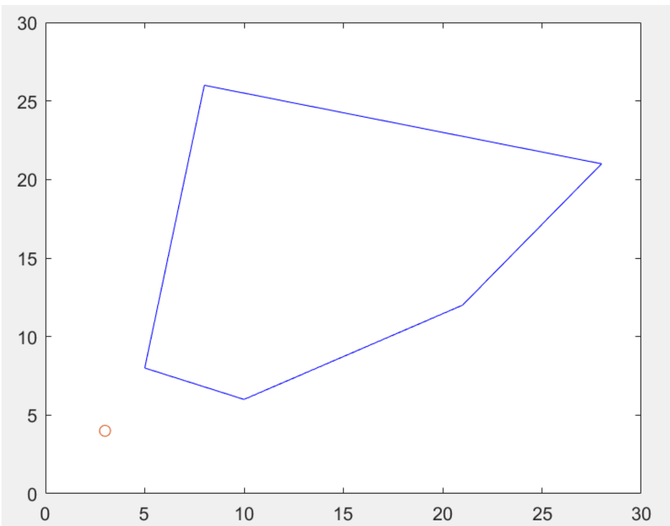
Output:

```
>> [distance] = computeDistancePointToPolygon([5 8; 10 6; 21 12; 28 21; 8 26],[3,4])

distance =

    4.4721

fx >>
```



Example 3:
Input: [distance] = computeDistancePointToPolygon([5 ; 10 ; 21 ; 28 ; 8],[0,0]) (Demonstraing wrong input format)

Output:

```
>>  [distance] = computeDistancePointToPolygon([5 ; 10 ; 21 ; 28 ; 8],[0,0])
Error using computeDistancePointToPolygon (line 7)
Columns of Polygon does not equal 2
```

computeTangentVectorToPolygon

The function computeTangentVectorToPolygon was created and my code can be seen below.

```matlab
% Eric Perez
% lab 1 Problem 2
function [U] = computeTangentVectorToPolygon(P,q)
sizeP = size(P);
FirstSeg = [0,0];
SecondSeg = [0,0];
if sizeP(2) == 2
    [MinVertexDistance,Vx,Vy] = computeDistancePointToPolygon(P,q);
     VertexPoint = [Vx,Vy];
    for i = 1:sizeP(1)
        p1 = P(i,:);
        p2 = P(mod(i,sizeP(1))+1,:); %Next Vector

        %compute distance from
        segmentDistance(i) = computeDistancePointToSegment(p1,p2,q);
        P1(i,:) = p1;
        P2(i,:) = p2;
    end
else
   error('Wrong poly size');
end
minSeg = min(segmentDistance);
IndexSeg = find(segmentDistance==minSeg);
XP1 = P1(IndexSeg,1);
YP1 = P1(IndexSeg,2);
```

```matlab
YP1 = P1(IndexSeg,2);
XP2 = P2(IndexSeg,1);
YP2 = P2(IndexSeg,2);
FirstSeg = [XP1,YP1];
SecondSeg = [XP2,YP2];

if minSeg < MinVertexDistance
    Changex = XP2 - XP1;
    Changey = YP2 - YP1;
    M = sqrt((Changex).^2 + (Changey).^2);
    Ux = Changex/M;
    Uy = Changey/M;
    U =[-Uy,Ux] ;
   disp('MinSegmentDistance')
else
    Changex = Vx - q(1);
    Changey = Vy - q(2);
    u = [-Changey, Changex]; % rotate 90 to make tangent
    Ux = u(1)/norm(u);
    Uy = u(2)/norm(u);
    U = [Ux,Uy];
    disp('MinVertextDistance')

end
```

The inputs for this function are the same as those for the last function. A point q and a polygon P. This function establishes if a polygon segment or a vertex is closer to q. The output, u, in the syntax [# #], would be tangent to a circle centered at the vertex that passes through q if the vertex is closer. U is rotated counterclockwise using this function. A segment is parallel to u if that segment is nearest to it. Additionally, this implements the "computeDistancePointToPolygon" and "computeDistancePointToSegment" from two previous functions. Depending on which one is closer, the horizontal and vertical distances are calculated from point q to the point of a vertex, or from point q to the segment for “u”. The magnitude equation ($\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$) is used to normalize U_x and U_y along with the function “norm()”. To ensure that there are two columns, this function checks the inputs.

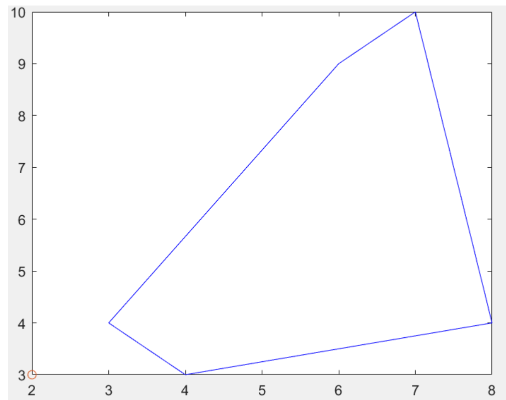
Example 1:
Input: [u] = computeTangentVectorToPolygon([3 4; 4 3; 8 4; 7 10; 6 9],[2,3])
Output:

```
MinVertexDistance =

    1.4142


u =

   -0.7071    0.7071
```



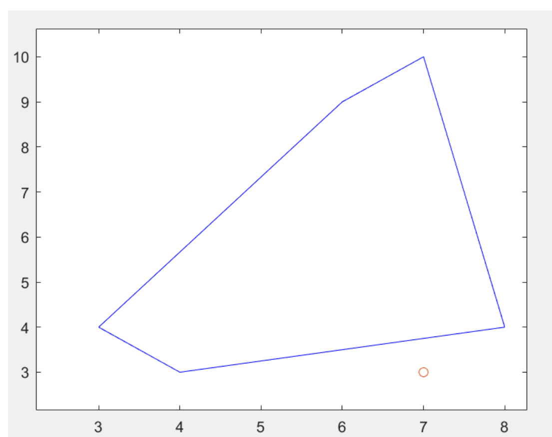
Example 2:
Input: [u] = computeTangentVectorToPolygon([3 4; 4 3; 8 4; 7 10; 6 9],[7,3])
Output:

```
minSeg =

    0.7276


u =

   -0.2425    0.9701
```



Example 3:
Input: [u] = computeTangentVectorToPolygon([3; 4; 8;10; 9],[7,3])
Output:

```
>> [u] = computeTangentVectorToPolygon([3; 4; 8;10; 9],[7,3])
Error using computeTangentVectorToPolygon (line 20)
Wrong poly size
```