# ME145 Robotic Planning and Kinematics: Lab 1

## Line and Segments

**Elijah Perez**

**Winter 2025, Feb 15**

# E1.6: Lines and Segments

Function(**ComputeLineThroughTwoPoints**): Code shown below.

```
function [a,b,c] = computeLineThroughTwoPoints(p1,p2)
%% Error statments

if length(p1)~=2
    error("Incorrect Dimentions for input in position 1")
elseif  length(p2)~=2
    error("Incorrect Dimentions for input in position 2")
else
%% Line Calcs

x1 = p1(1); % point 1
y1 = p1(2);

x2 = p2(1); % point 2
y2 = p2(2);


A = y2-y1;
B = x2-x1;
C = -(A/B)*x1+y1;

M = [A,B];
M_mag = sqrt(M(1)^2 + M(2)^2);
m = M./(M_mag);

if M_mag<0.1
    error("Points need to have greater than 0.1 distance")
else
end
%% outputs
a = m(1);
b = m(2);
c = C;

```

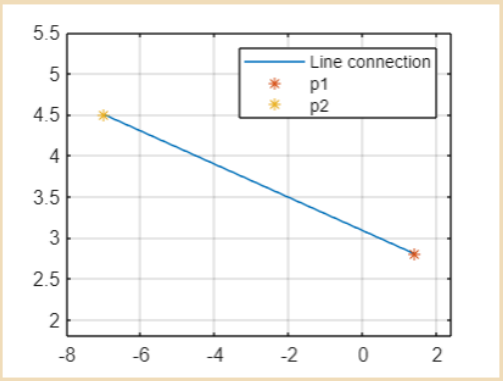
The function takes two points, p1 and p2, as input. Users need to enter both points in the format “[ x ,y ]”. The function performs three validation checks to ensure the inputs are correct. These checks include verifying the proper format for both points, ensuring the points are distinct, and allowing for a tolerance of 0.1. The output line is represented as $\{(x, y) \mid ax + by + c = 0\}$, using the point-slope equation $y - y1 = m(x - x1)$ as the basis. Finally, the parameters are normalized by dividing them by their magnitude.

**Examples:**

Ex1) Working code

Lab 1

```
p1 = [1.4,2.8];
p2 = [-7.0,4.5];

[a,b,c] = computeLineThroughTwoPoints(p1,p2)
```

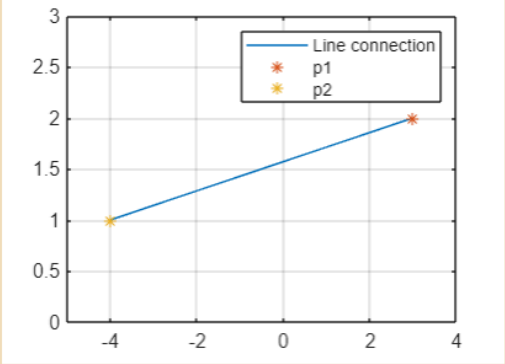



```
a = 0.1984
b = -0.9801
c = 3.0833
```

Ex2) Working code

Lab 1

```
p1 = [3,2];
p2 = [-4,1];

[a,b,c] = computeLineThroughTwoPoints(p1,p2)
```




```
a = -0.1414
b = -0.9899
c = 1.5714
```

Ex3) Incorrect Input: P1

Lab 1

```
p1 = [1,2,4];
p2 = [-4,1];

[a,b,c] = computeLineThroughTwoPoints(p1,p2)
```

```
Error using computeLineThroughTwoPoints
Incorrect Dimentions for input in position 1
```

Ex4) Incorrect Input: P2

Lab 1

```
p1 = [1,2];
p2 = -4;

[a,b,c] = computeLineThroughTwoPoints(p1,p2)
```

```
Error using computeLineThroughTwoPoints
Incorrect Dimentions for input in position 2
```

Ex5) Incorrect Input distance

Lab 1

```
p1 = [1,2];
p2 = [1,2];

[a,b,c] = computeLineThroughTwoPoints(p1,p2)
```

Error using **computeLineThroughTwoPoints**
Points need to have greater than 0.1 distance

**Function(computeDistancePointToLine)**

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

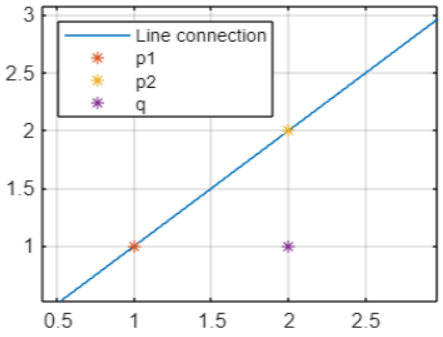
```
function D = computeDistancePointToLine(q,p1,p2)

%% Error statments
if length(p1)~=2
    error("Incorrect Dimentions for input in position 1")
elseif  length(p2)~=2
    error("Incorrect Dimentions for input in position 2")
else
%% Line Calcs

x1 = p1(1); % point 1
y1 = p1(2);
x2 = p2(1); % point 2
y2 = p2(2);


A = y2-y1;
B = x2-x1;
C = -(A/B)*x1+y1;
c = -(A * x1 + B * y1);
M = [A,B];
M_mag = sqrt(M(1)^2 + M(2)^2);
m = M./(M_mag);

if M_mag<0.1
    error("Points need to have greater than 0.1 distance")
else
end
%% outputs

D = abs((A*q(1)+B*q(2)+c))/sqrt(A^2 + B^2);

```

Three points are provided as inputs to this function: q, p1, and  p2,. The points p1 and p2 define a line, and all points are formatted as “[x,y]”. The function calculates and outputs the distance between point q and the line formed by p1 and p2. To compute this distance, the function uses the orthogonal projection of q onto the line defined by p1 and p2. The point-slope equation of a line, y−y1=m(x−x1)y - y1, is used to determine the slope of the line formed by p1 and p2. The slope of the line perpendicular to this, which passes through q, is the negative reciprocal of the slope of the line formed by p1 and p2. Using this perpendicular slope, a new line is established to measure the distance.The intersection point between the line formed by p1 and p2 and the perpendicular line passing through q is calculated. The distance from q to this intersection point is determined, which is simply the magnitude of the vector connecting q to the intersection point.

Ex1) Working Function

Function 2

```
clear
p1 = [1,1];
p2 = [2,2];
q  = [2,1];

D = computeDistancePointToLine(q,p1,p2)
```

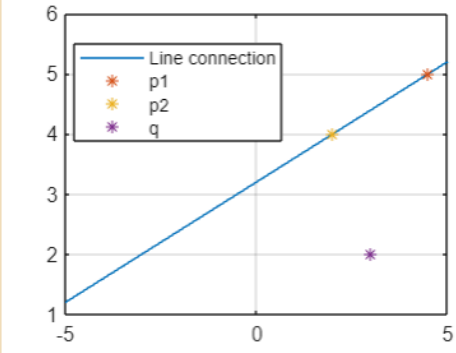



```
D = 0.7071
```

Ex2) Working Function

Function 2

```
clear
p1 = [4.5,5];
p2 = [2,4];
q  = [3,2];

D = computeDistancePointToLine(q,p1,p2)
```




```
D = 1.9312
```

Ex3) Two close to the line

Function 2

```
clear
p1 = [1,1];
p2 = [2,2];
q  = [2,2.01];

D = computeDistancePointToLine(q,p1,p2)
```

```
Error using computeDistancePointToLine
Point Q need to have greater than 0.1 distance
from line
```

Ex4) Insufficient Inputs

Function 2

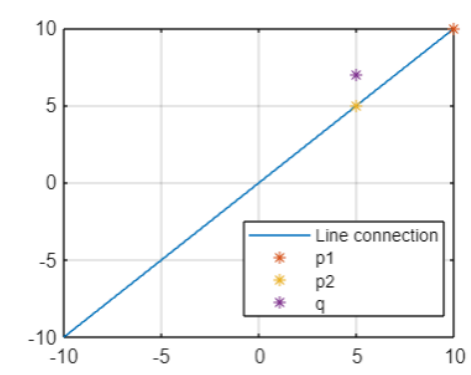
```
clear
p1 = [1,1];
p2 = [2,4];
q  = [3,5,10];

D = computeDistancePointToLine(q,p1,p2)
```

```
Error using computeDistancePointToLine
Incorrect Dimentions for input in position 1
```

Ex5) Working Function

Function 2

```
clear
p1 = [10,10];
p2 = [5,5];
q  = [5,7];
D = computeDistancePointToLine(q,p1,p2)
```




```
D = 1.4142
```

**Function(computeDistancePointToSegment)**

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

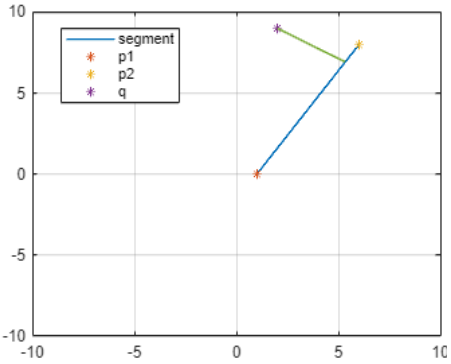
```
function D = computeDistancePointToSegment(q,p1,p2)

%% Error statments
if length(p1)~=2
    error("Incorrect Dimentions for input in position 2")
elseif  length(p2)~=2
    error("Incorrect Dimentions for input in position 3")
elseif  length(q)~=2
    error("Incorrect Dimentions for input in position 1")
else
%% Line Calcs

x1 = p1(1); % point 1
y1 = p1(2);
x2 = p2(1); % point 2
y2 = p2(2);


S = [x2,y2] - [x1,y1];
W = q-[x1,y1];
Z = (dot(W,S))/(dot(S,S)); % scalar 0-1
Zvec= p1+S*Z;

%% output
if Z>=0 && Z<=1

    point = Zvec;
    D = norm(W-Zvec);

elseif Z>1
    D = norm(q-p2);
    point = p2;
else
    D = norm(q-p1);
    point = p1;
end

%% Additional plotting
figure

plot([x1,x2],[y1,y2]);%S
hold on
plot(x1,y1,"*")
hold on
plot(x2,y2,"*")
hold on
plot(q(1),q(2),"*")
hold on
plot([point(1),q(1)],[point(2),q(2)])%Short line
grid on
legend("segment","p1","p2","q")
end
```

This function requires three inputs: p1, p2, and q. Each point is represented in the format "[# #]". The output of the function is the distance value. The function calculates the length of the line formed by p1 and p2 and identifies the orthogonal projection of q onto the line segment connecting p1 and p2. This projection represents the portion of the line segment that separates the two points. The function ensures that all inputs are correctly formatted as "[# #]" by verifying their size. It uses the point-slope form equation, y - y1 = m(x - x1), to calculate the slope of the line. The distance is determined using the formula sqrt((x2 - x1)^2 + (y2 - y1)^2). The function also plots lines from q to the segment and between p1 and p2. It calculates the intersection point of the line from q with the segment and checks if this point lies within the boundaries of the segment. Additionally, the function verifies that the inputs are distinct, the sizes are accurate, and the segment contains the orthogonal projection of q.

Ex1) Working Function:

Function 3

```
% Define the points
q =  [2,9];
p1 = [1, 0];
p2 = [6, 8];

[D,point]= computeDistancePointToSegment(q, p1, p2)
```

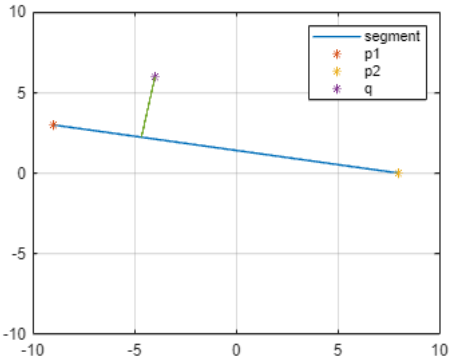



```
D = 3.9220
```

Ex2) Working Function:

Function 3

```
% Define the points
q =  [-4,6];
p1 = [-9, 3];
p2 = [8, 0];

[D,point]= computeDistancePointToSegment(q, p1, p2)
```




```
D = 3.8233
```

Ex3) Incorrect Input dimensions

```
% Define the points
q =  6;
p1 = [-9, 3];
p2 = [8, 0];

[D,point]= computeDistancePointToSegment(q, p1, p2)
```

```
Error using computeDistancePointToSegment (line 10)
Incorrect Dimentions for input in position 1
```

Ex4) Segment must be greater than 0.1 in length

```
% Define the points
q =  [5,5];
p1 = [8, 0];
p2 = [8, 0];

[D,point]= computeDistancePointToSegment(q, p1, p2)
```

```
Error using computeDistancePointToSegment (line 48)
Segment length must be greater than 0.1
```

Ex5) The point must be 0.1 distance away from the segment

```
% Define the points
q =  [8, 5];
p1 = [8, 8];
p2 = [8, 0];

[D,point]= computeDistancePointToSegment(q, p1, p2)
```

```
Error using computeDistancePointToSegment (line 46)
point must be 0.1 away from segment
```

**Function(computeDistancePointToPolygon)**

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

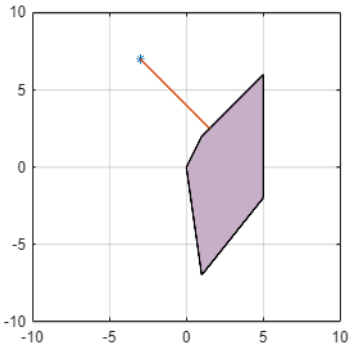
```
function D = computeDistancePointToPolygon(P,q)
%% Error checking
if length(P(1,:)) ~= 2
error ("Dimensions of P must be [x1 y1, x2 y2, ... xn yn]")

elseif length(q) ~= 2
    error ("Dimensions of q must be in the form [x,y]")
end




%% Solving all solutons
for i = 1:length(P(:,1))+1

    Counter(i) = i ;
end
Counter(end) = 1;


for i = 1:length(P(:,1))

    p1 = P(Counter(i),:);
    p2 = P(Counter(i+1),:);

    [dist(i),point(i,:)] = computeDistancePointToSegment(q,p1,p2);
end

%% output
D = min(dist);
finder = dist == D;
point = point(finder,:);


%% Plotting

figure
X = P(:,1);
Y = P(:,2);
fill(X,Y,[0.8 0.7 0.8])
hold on
plot(q(1),q(2),"*")
hold on
plot([point(1,1),q(1)],[point(1,2),q(2)])%Short line
xlim([-10,10])
ylim([-10,10])
grid on
axis normal;
axis square;


end
```

This function accepts two inputs: a polygon P and a point q. The polygon P is represented in the format "[# #; # #; # #,# #, # #]," which consists of multiple points. The point q is represented in the format "[# #]." The function calculates the distance between each vertex of the polygon and the point q separately. It identifies the nearest vertex by determining which distance is the smallest. To calculate the distance, the function considers the horizontal and vertical paths connecting each vertex to the point q and applies the distance formula. It also checks the input size to ensure that two columns of data are provided. Finally, the function plots the polygon to visualize its shape.

Ex1) Working function

Function 4

```
P = [0 0; 1 2 ; 5 6; 5 -2;1 -7];
q = [-3, 7];

D = computeDistancePointToPolygon(P,q)
```

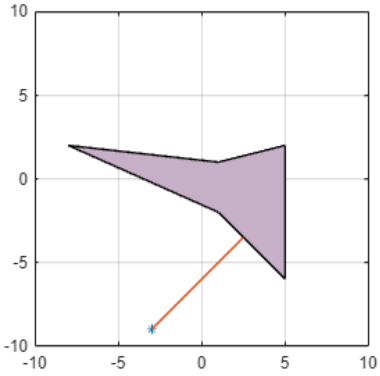



```
D = 6.3640
```

Ex2) Working function

Function 4

```
P = [-8 2; 1 -2 ; 5 -6; 5 2;1 1];
q = [-3, -9];

D = computeDistancePointToPolygon(P,q)
```




```
D = 7.7782
```

Ex3) incorrect Input for P

Function 4

```
P = [-8 2 4; 1 -2 3 ;3 5 -6; 5 2 3;1 3 3];
q = [-3, -9];

D = computeDistancePointToPolygon(P,q)
```

```
Error using computeDistancePointToPolygon (line 4)
Dimensions of P must be [x1 y1, x2 y2, ... xn yn]
```

Ex4) Incorrect Dimensions for point q

Function 4

```
P = [-8 2; 1 -2 ; 5 -6; 5 2;1 3];
q = [-3, -9, 10 , 2];

D = computeDistancePointToPolygon(P,q)
```

```
Error using computeDistancePointToPolygon (line 7)
Dimensions of q must be in the form [x,y]
```

Ex5) points are two close together

Function 4

```
P = [-8 2; -8 2 ; 5 -6; 5 2;1 3];
q = [-3, -9];

D = computeDistancePointToPolygon(P,q)
```

```
Error using computeDistancePointToSegment (line 48)
Segment length must be greater than 0.1

Error in computeDistancePointToPolygon (line 26)
    [dist(i),point(i,:)] = computeDistancePointToSegment(q,p1,p2);
```

**Function(computeTangentVectorToPolygon)**

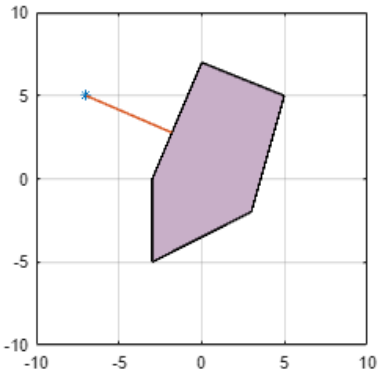
```
function u = computeTangentVectorToPolygon(P,q)

[D,Segment] = computeDistancePointToPolygon(P,q);

if length(Segment(:,2)) ==2
    x1 = Segment(1,1); % point 1
    y1 = Segment(1,2);
    x2 = Segment(2,1); % point 2
    y2 = Segment(2,2);

    S = [x2,y2] - [x1,y1];
else
    vx = Segment(1,1); % point 1
    vy = Segment(1,2);
    r = D;

end


%% Output
if length(Segment(:,2)) ==2
    u = S / (sqrt(S(1)^2+S(2)^2)); % Parellel u unit vec to segment
else % cirlce time

    phi = atan2(vy-q(2),vx-q(1));
    u = [-r*sin(phi) -r*cos(phi)]/r;
    hold on
    th = 0:pi/50:2*pi;
    xunit = r * cos(th) + vx;
    yunit = r * sin(th) + vy;
    plot(xunit, yunit);

end

end
```

The inputs for this function are the same as those for the previous function: a point q and a polygon P. The function determines whether a segment of the polygon or a vertex is closer to q. The output, u, in the format [# #], will be tangent to a circle centered at the vertex passing through q if the vertex is closer. The function rotates u counterclockwise. A segment is parallel to u if it is the closest to it. This function also uses the "computeDistancePointToPolygon" and "computeDistancePointToSegment" methods from earlier functions. It calculates the horizontal and vertical distances from point q to either a vertex or a segment, depending on which is closer, to define u. The magnitude equation, (x2 - x1)^2 + (y2 - y1)^2, is used to normalize Ux and Uy along with the "norm()" function. Lastly, the function ensures the inputs are structured to maintain two columns.

Ex1) Working function, Point is within the segment.

Function 5

```
P = [-3 -5; -3 0; 0 7; 5 5; 3 -2 ];
q = [-7, 5];

u = computeTangentVectorToPolygon(P,q)
```

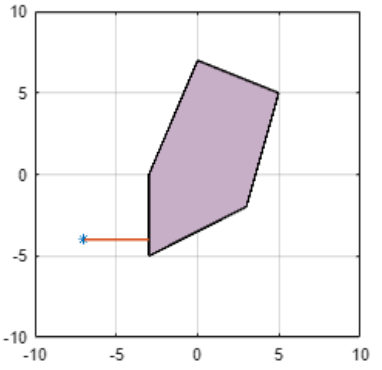



```
u = 1×2
    0.3939    0.9191
```

Ex2) Working function, Point is within the segment.

Function 5

```
P = [-3 -5; -3 0; 0 7; 5 5; 3 -2 ];
q = [-7, -4];

u = computeTangentVectorToPolygon(P,q)
```

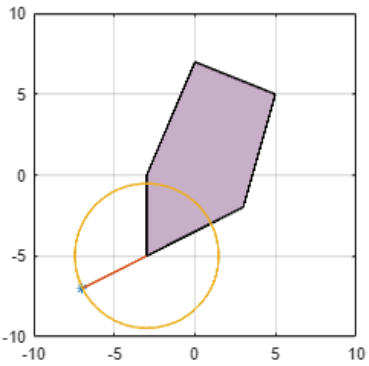



```
u = 1×2
     0     1
```

Ex3) Working function, Point to vertex

Function 5

```
P = [-3 -5; -3 0; 0 7; 5 5; 3 -2 ];
q = [-7, -7];

u = computeTangentVectorToPolygon(P,q)
```

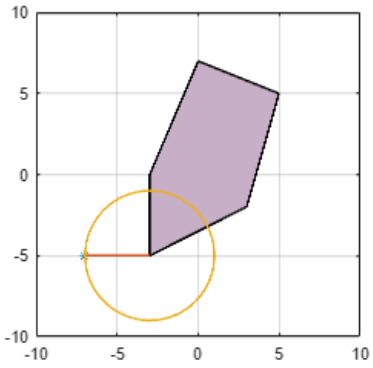



```
u = 1×2
    -0.4472   -0.8944
```

Ex4) Working function, Point to vertex

Function 5

```
P = [-3 -5; -3 0; 0 7; 5 5; 3 -2 ];
q = [-7, -5];

u = computeTangentVectorToPolygon(P,q)
```




```
u = 1×2
     0    -1
```

Ex5) Error, Incorrect polygon format

Function 5

```
P = [-3 -5 0; -3 0 0; 0 7 0; 5 5 0; 3 -2 0 ];
q = [-7 -5];

u = computeTangentVectorToPolygon(P,q)
```

```
Error using computeDistancePointToPolygon (line 4)
Dimensions of P must be [x1 y1, x2 y2, ... xn yn]

Error in computeTangentVectorToPolygon (line 3)
[D,Segment] = computeDistancePointToPolygon(P,q);
```