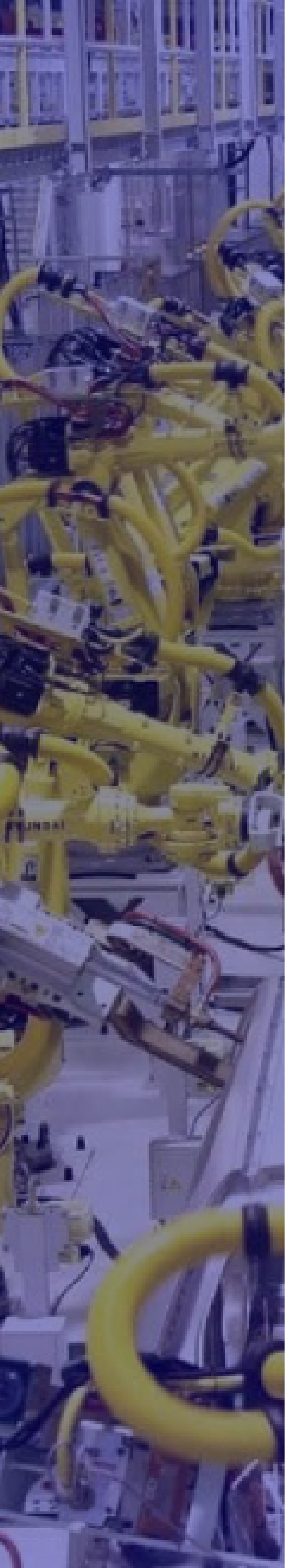


Dep. of Mechanical Eng.

Robotics

Dr. Saeed Behzadipour

Homework 6

Yashar Zafari
99106209

December 13, 2023

Robotics

Homework 6

Yashar Zafari

99106209

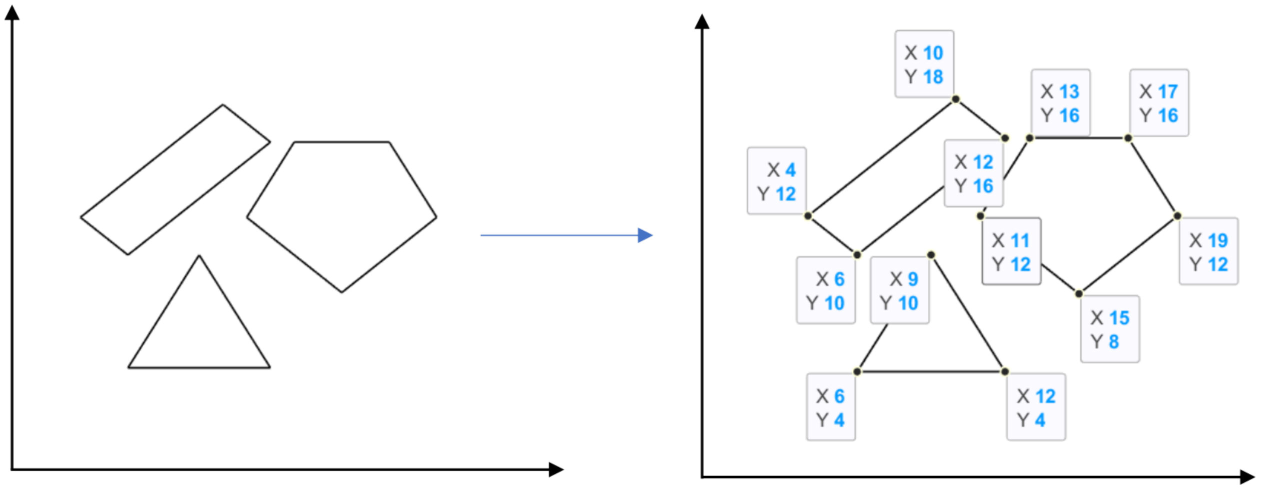
Questions

Path Generation - Obstacle Avoidance

In this assignment, you are supposed to implement the potential field method for the path planning of a point moving on a plane.

- Write a MATLAB function that generates the via points of the path. The function receives the coordinates of the starting point X_s, final destination X_f, η which is the scaling factor for the attractive field, and matrix B which stores the information of the obstacles. It has a distance of influence with a scaling factor (α) for the obstacle. The function should stop generating via points when the moving point's distance from X_f is less than 0.1. It should then return matrix P which has the coordinates of the via points starting from $X_s = \begin{bmatrix} 1 \\ 10 \end{bmatrix}$ and ending by $X_f = \begin{bmatrix} 22 \\ 12 \end{bmatrix}$.

```
        function P = Path_generator (Xs, Xf, eta, B)
```



Solution

There is a script file in order to run the function and plot the results, called "`script`".

First I start by defining the obstacles in an efficient way to give it as input to "`Path_generator`". In "**B**" matrix, the first row is dedicated to the x coordinates of the obstacle's vertices, the second row is dedicated to their y-coordinates, and finally the third row contains the tag for each obstacle, for example if obstacle "1" has 4 vertices, it will be presented as 4 columns in "**B**" matrix in which the third row is "1" for all the columns. So in conclusion, "**B**" for our problem is formed as below:

$$\mathbf{B} = \left[\begin{array}{cccc:ccc:ccccc} 4 & 10 & 12 & 6 & 9 & 12 & 6 & 11 & 13 & 17 & 19 & 15 \\ 12 & 18 & 16 & 10 & 10 & 4 & 4 & 12 & 16 & 16 & 12 & 8 \\ \hdashline 1 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 \end{array}\right]$$

Also for more convenient use of the function, it will get the values for η, α, ϵ, and ρ_o as input. For better explanation of the code, code is provided as below:

```
function P = Path_generator(Xs, Xf, eta, B, alpha,eps,p_o)
P = Xs; %Initializing P with the start point Xs
current_pos = Xs; % Initializng the current position with the
   start point Xs
obstacle_indices = unique(B(3,:)); % Indices(Number) of the
   obstacles given in B
obstacles = []; % Initializing the obstacles struct
for i = obstacle_indices
    % Vertices of the obstacle
    obstacle_points = B(1:2, B(3,:) == i);
    % Saving the vertices in "points" field of the "obstacle"
       struct
    obstacles{i}.points = obstacle_points;
    % Saving the number of the vertices or actually the number
       of the sides of the obstacle
    obstacles{i}.numPoints = size(obstacle_points, 2);
end
while norm(current_pos - Xf) >= 0.1
    if size(P,2)>2 % Local minima check
        % If in 2 iteration it goes back to the initial
           location then
        % it's in local minima
        if current_pos==P(:,end-2)
            P=P(:,1:end-2);
            disp('Local Minima at:')
            assignin("base",'local_minima',P(:,end));
            fprintf('%.4f\n',P(:,end));
            return
        end
    end
    F_attr = -eta * (current_pos - Xf); % Computing the
       attractive force.
    F_rep_total = [0; 0]; % Initializing the total repulsive
       force.
    for i = obstacle_indices
        % Initializing the repulsive force for the i-th
           obstacle.
        F_rep = [0; 0];
        % Finding the nearest side of the obstacle
        middle=mean(obstacles{i}.points,2);
        [~,~,j]=polyxpoly([obstacles{i}.points(1,:) obstacles{
           i}.points(1,1)], [obstacles{i}.points(2,:)
           obstacles{i}.points(2,1)],[current_pos(1) middle(1)
           ],[current_pos(2) middle(2)]);
        j=j(1);
        vertex1=[obstacles{i}.points(:,j);0];
        % Circular order of the vertices
        if j == obstacles{i}.numPoints
            vertex2=[obstacles{i}.points(:, 1);0];
        else
            vertex2=[obstacles{i}.points(:, j+1);0];
        end
        % Finding the shortest distance to the obstacle
        a=vertex1-vertex2;
        b=[current_pos;0]-vertex2;
        if 0<=dot(a,b) && dot(a,b)<=dot(a,a)
            rep_vector=-a*dot(a,b)/(norm(a)^2)+b;
            rep_dist=norm(cross(a,b))/norm(a);
        else
            % In this case the projection point is out of
            % line segment. Thus the shortest distance to
            % the side is from one of the vertices of the side
               .
            [rep_dist,k]=min([norm(b) norm(b-a)]);
            if k==1
                rep_vector=b;
            else
                rep_vector=b-a;
            end
        end
        if rep_dist < p_o
            rep_force_magnitude = alpha*(1/rep_dist-1/p_o) / (
               rep_dist^3);
            F_rep = rep_force_magnitude * (rep_vector /
               rep_dist);
        end
        F_rep_total = F_rep_total + F_rep(1:2);
    end
    F_net = F_attr + F_rep_total;
    new_pos = current_pos + (F_net / norm(F_net)) * eps;
    P = [P new_pos];
    current_pos = new_pos;
end
P=[P Xf];
end
```

First I initialize "**P**" and the current position with the start point "X_s". In order to sort the obstacles, by comparing their tag numbers, I cluster the points of each obstacles and the number of the vertices in "`obstacles`" structure. Later using a "while" loop with condition that the distance between current position and the final point "X_f" is bigger than 0.1, the function starts the iteration. Attraction force is easy to calculate and it's as below:

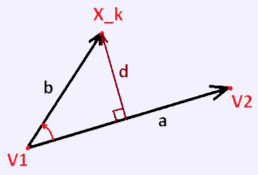
$$\vec{F}_{att} = -\eta(\vec{X}_k - \vec{X}_f)$$

in which $\vec{X}_k$ is the current position of the robot. Then in order to calculate the repulsive force, the repulsive force from each obstacle should be calculated. Thus there should be a "for" loop between the obstacle to calculate each one's effect in the total repulsive force. The repulsive force of each obstacle is calucated as below:

$$\vec{F}_{rep_i} = \begin{cases} 0 & \text{if } \rho_o \leq \rho_i \\ \alpha_i \dfrac{1}{\rho_i^3(\vec{X}_k)}(\dfrac{1}{\rho_i(\vec{X}_k)} - \dfrac{1}{\rho_o})(\vec{X}_k - B_i(\vec{X}_k)) & \text{else} \end{cases}$$

which "$\rho_i(\vec{X}_k)$" is the shortest distance from the current position to the i-th obstacle and "$B_i(\vec{X}_k)$" is the vector that represent this distance, going form the obstacle to the current position of the robot. First I use this trick to lower the calculations. Instead of checking each side of obstacle to find the shortest distance, it could be shown geometrically that if we connect the current point to the middle point of the obstacle, which is actually the mean of obstacle vertices, that line intersects with the side that has the shortest distance to the the current point. Hence in order to calculate the shortest distance I use the triangle geometry. In this manner, if we define the vector from one vertex to another as "$\vec{a}$" and the vector from the head of the previous vector to the current position as "$\vec{b}$", the following relation could be shown between these vectors and the shortest distance to the obstacle and the vector representing that distance:

$$|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}| \sin(\theta) \rightarrow$$
$$\rightarrow |\vec{d}| = \rho_i(\vec{X}_k) = |\vec{b}| \sin(\theta) = \frac{|\vec{a} \times \vec{b}|}{|\vec{a}|}$$



In order to check whether the projection of the current position on the line containing the side of the obstacle lies in between the two vertices of side, I use the dot product of two vector "$\vec{a}$" and "$\vec{b}$". If the result of the dot product is bigger than zero and less than the self dot product of "$\vec{a}$", then the point's projection lies in the side of the obstacle, else the nearest vertex is is selected as the point that has shortest distance to the current point. The vector that represents the shortest distance is found as below:

$$\vec{d} = \vec{X}_k - B_i(\vec{X}_k) \begin{cases} -\dfrac{\vec{a} \cdot \vec{b}}{|\vec{a}|^2}\vec{a} + \vec{b} & \text{if projection between the vertices} \\ min(\vec{b}, \vec{b} - \vec{a}) & \text{else} \end{cases}$$

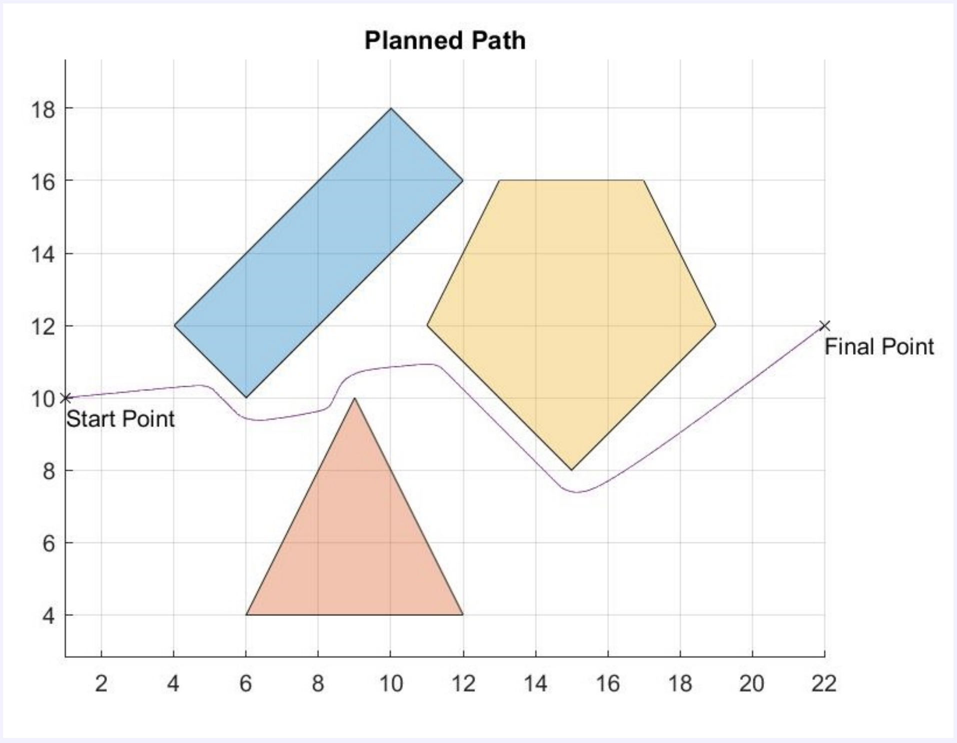
After calculating the repulsive force, by summing the attractive and repulsive forces, we find the desired direction for the movement of the robot. The direction is the normalization of gained vector and using the step size ϵ the next position is found as below:

$$\vec{F} = \vec{F}_{att} + \vec{F}_{rep} \rightarrow \ \vec{X}_{k+1} = \vec{X}_k + \epsilon \frac{\vec{F}}{|\vec{F}|}$$

- Test your code for the following example with $\eta = \alpha = 1$, $\epsilon = 0.1$ and distance of influence equal to 2 for all obstacles. The coordinates of the obstacles' end points are shown in the following figure. Plot the obstacles as well as your generated path in a MATLAB figure and submit a detailed report and your results along with your MATLAB code.

Solution

Running the code for the specified parameters in the question, here are the results:

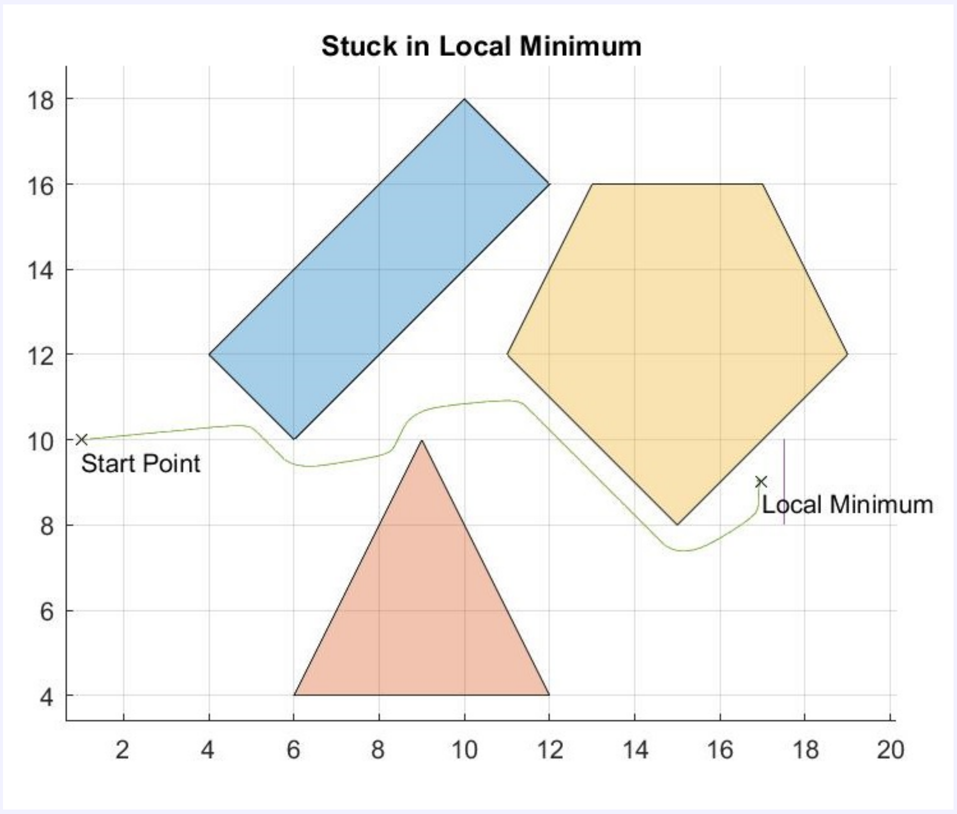


- Add a line between points (17.5,10) and (17.5,8). Try to generate a path again. Where does the path get stuck?

Solution

After adding the line, here are the results and robot gets stuck in the following local minimum:

$$X_{\text{Local Minimum}} = \begin{bmatrix} 16.97 \\ 9.028 \end{bmatrix}$$



Bonus

- Implement a random walk algorithm and see if it can solve the local minima problem of Question 3. The random walk algorithm starts when the potential field algorithm (PFA) is stuck in a local minima. It makes the robot take a random jump with a maximum length of a (let a be 6 in this example), at a random angle providing that the jump doesn't interfere with any obstacle. You can propose a different method to determine the angle of jump to increase the chance of escape. Then it runs the PFA from the new position to see if it can reach the target destination. Submit a detailed report for this question.

Solution

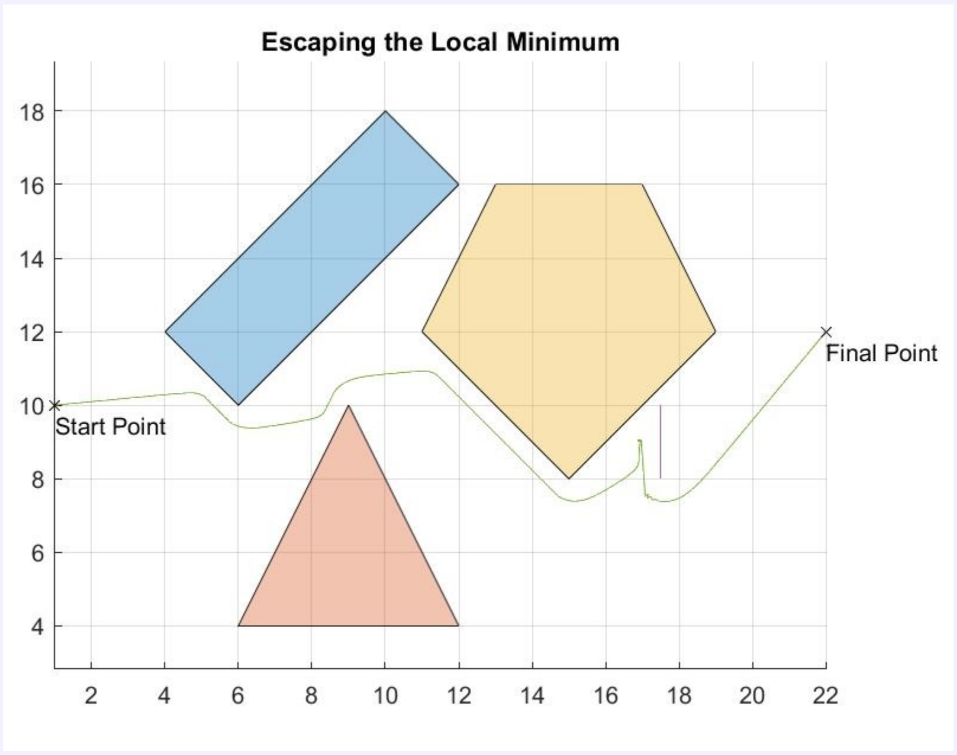
In order to escape the local minimum, I defined another function called "`potential`" that calculates the potential of the points. My implementation is such that we say that the escape from the local minimum has occurred if only the potential of the new point is at least 25% more than the potential of the local minimum.
Also for much more efficiency, I reduced the weight of the attractive force by 1/20, because our main goal for that part of path planning, is to escape the local minimum and on the other hand only the repulsive forces pushes the robot away from the local minimum, thus it's kind of logical that these forces have more weight in the direction calculation. But I didn't totally discard the attractive force since I want it have some sense of the final point.
Also for much more efficiency and speed of the code, instead of try every direction and in many different step sizes to find a direction to escape, I select the the direction calculated based on the modifications I mentioned earlier in the local minimum point. Since we already have the effects of the near obstacles in the repulsive forces, I'm less concerned of robot's collision with the obstacle by the direction decided beforehand and only the change of the step size, therefor I don't feel the obligation to calculate the forces in every iteration of the escape process. Here below is the function "`Path_generator_esc`" for our goal:

```
function P = Path_generator_esc(Xs, Xf, eta, B, alpha,eps,p_o)
P = Xs; %Initializing P with the start point Xs
current_pos = Xs; % Initializng the current position with the
   start point Xs
obstacle_indices = unique(B(3,:)); % Indices(Number) of the
   obstacles given in B
obstacles = []; % Initializing the obstacles struct
for i = obstacle_indices
    % Vertices of the obstacle
    obstacle_points = B(1:2, B(3,:) == i);
    % Saving the vertices in "points" field of the "obstacle"
       struct
    obstacles{i}.points = obstacle_points;
    % Saving the number of the vertices or actually the number
        of the sides of the obstacle
    obstacles{i}.numPoints = size(obstacle_points, 2);
end
while norm(current_pos - Xf) >= 0.1
    if size(P,2)>2 % Local minima check
```

```
        % If in 2 iteration it goes back to the initial
           location then
        % it's in local minima
        if current_pos==P(:,end-2)
            % Calculating the potential of the point
            V_current=potential(current_pos, Xf, eta,
               obstacles, alpha, p_o);
            V_local=V_current;
            % Weighting the repulsive force more than
               attractive force
            % in order to escape the local minimum
            F_net_local= F_rep_total + 0.05*F_attr;
            pos_local=current_pos;
            eps1=eps;
            P=P(:,1:end-2);
            % If the new point's potential is at least 25%
               more than the
            % potential of the local minimum, I assume that it
                has escaped
            % the local minimum.
            while 1.25*V_local>V_current
                % Instead of the testing multiple direction,
                   by weighting
                % the repulsive force more than the attractive
                    force,
                % I only use that direction for moving.
                pos_local=pos_local+F_net_local/norm(
                   F_net_local)*eps1;
                % If the step size isn't sufficient enough to
                   escape the
                % local minimum, it's increased in every
                   iteration.
                eps1=eps1*1.25;
                P=[P pos_local];
                V_current=potential(pos_local, Xf, eta,
                   obstacles, alpha, p_o);
            end
            current_pos=P(:,end);
        end
    end
    F_attr = -eta * (current_pos - Xf); % Computing the
       attractive force.
    F_rep_total = [0; 0]; % Initializing the total repulsive
       force.
    for i = obstacle_indices
        % Initializing the repulsive force for the i-th
           obstacle.
        F_rep = [0; 0];
        % Finding the nearest side of the obstacle
        middle=mean(obstacles{i}.points,2);
        [~,~,j]=polyxpoly([obstacles{i}.points(1,:) obstacles{
           i}.points(1,1)], [obstacles{i}.points(2,:)
           obstacles{i}.points(2,1)],[current_pos(1) middle(1)
           ],[current_pos(2) middle(2)]);
        j=j(1);
        vertex1=[obstacles{i}.points(:,j);0];
        % Circular order of the vertices
        if j == obstacles{i}.numPoints
            vertex2=[obstacles{i}.points(:, 1);0];
        else
            vertex2=[obstacles{i}.points(:, j+1);0];
        end
        % Finding the shortest distance to the obstacle
        a=vertex1-vertex2;
        b=[current_pos;0]-vertex2;
        if 0<=dot(a,b) && dot(a,b)<=dot(a,a)
            rep_vector=-a*dot(a,b)/(norm(a)^2)+b;
            rep_dist=norm(cross(a,b))/norm(a);
        else
            % In this case the projection point is out of
            % line segment. Thus the shortest distance to
            % the side is from one of the vertices of the side
               .
            [rep_dist,k]=min([norm(b) norm(b-a)]);
            if k==1
                rep_vector=b;
            else
                rep_vector=b-a;
            end
        end
        if rep_dist < p_o
            rep_force_magnitude = alpha*(1/rep_dist-1/p_o) / (
               rep_dist^3);
            F_rep = rep_force_magnitude * (rep_vector /
               rep_dist);
        end
        F_rep_total = F_rep_total + F_rep(1:2);
    end
    F_net = F_attr + F_rep_total;
    new_pos = current_pos + (F_net / norm(F_net)) * eps;
    P = [P new_pos];
    current_pos = new_pos;
end
P=[P Xf];
```

```
end
% Potential function
function V = potential(current_pos, Xf, eta, obstacles, alpha,
    p_o)
V_attr = 0.5*eta * (norm(current_pos - Xf)^2);
V_rep_total = 0;
obstacle_indices = size(obstacles,2);
for i = obstacle_indices
    V_rep=0;
    middle=mean(obstacles{i}.points,2);
    [~,~,j]=polyxpoly([obstacles{i}.points(1,:) obstacles{i}.
       points(1,1)], [obstacles{i}.points(2,:) obstacles{i}.
       points(2,1)],[current_pos(1) middle(1)],[current_pos(2)
        middle(2)]);
    j=j(1);
    vertex1=[obstacles{i}.points(:,j);0];
    if j == obstacles{i}.numPoints
        vertex2=[obstacles{i}.points(:, 1);0];
    else
        vertex2=[obstacles{i}.points(:, j+1);0];
    end
    a=vertex1-vertex2;
    b=[current_pos;0]-vertex2;
    if 0<=dot(a,b) && dot(a,b)<=dot(a,a)
        rep_dist=norm(cross(a,b))/norm(a);
    else
        rep_dist=min([norm(b) norm(b-a)]);
    end
    if rep_dist < p_o
        V_rep=0.5*alpha*(1/rep_dist-1/p_o)^2;
    end
    V_rep_total = V_rep_total + V_rep;
end
V = V_attr + V_rep;
end
```

And the result is as below which is satisfying:


Escaping the Local Minimum
Start Point
Final Point