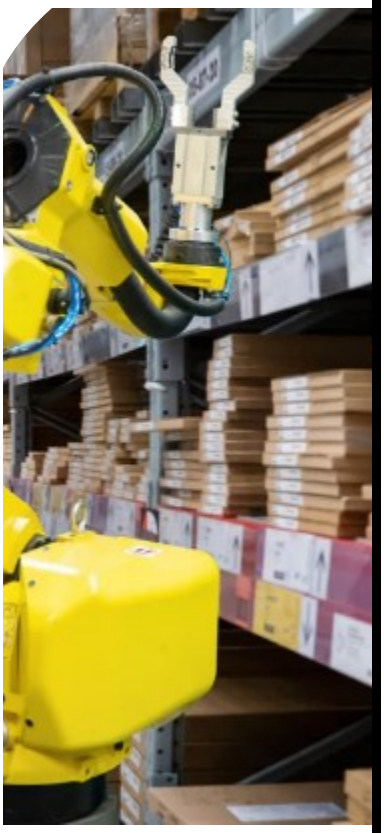
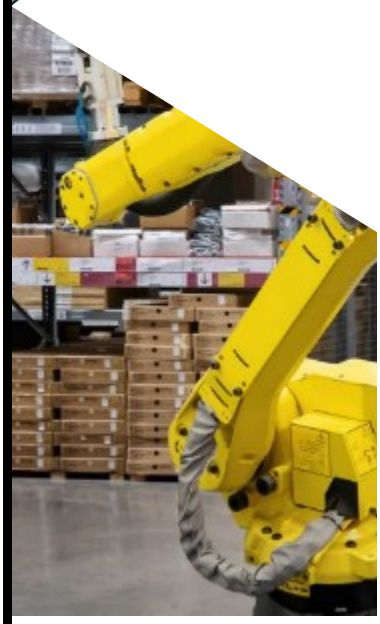
به نام خدا



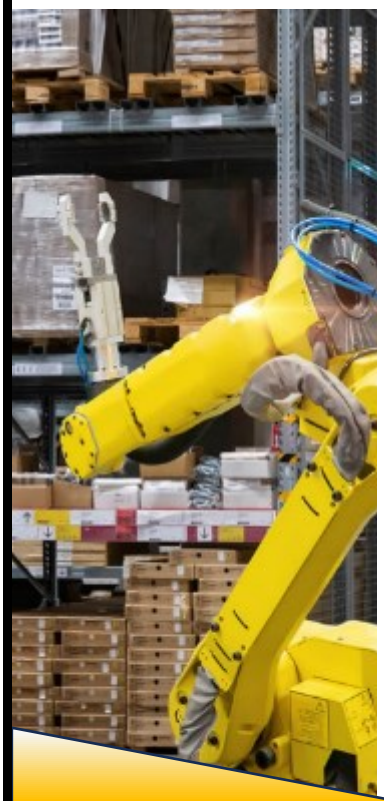


گزارش تمرین شماره 6 رباتیک

نام دانشجو : **عرفان رادفر**

شماره دانشجویی : **99109603**

استاد درس : **دکتر سعید بهزادی پور**



پاییز 1402

Erfan Radfar
99109603

Robotics

Dr. Saeed Behzadipour

Contents

2 ..... Problem 1
4 ..... Problem 2
5 ..... Problem 3
6 ..... Problem 4
6 ..... A: Continuous With No Jump
8 ..... B: Long Jump

Erfan Radfar
99109603

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.

Problem 1

1) Write a MATLAB function that generates the via points of the path. The function receives the coordinates of the starting point Xs, final destination Xf, 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 Xf is less than 0.1. It should then Return matrix P which has the coordinates of the via points starting from Xs and ending by Xf.

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

For path generation we need to define a potential field that drives our path away from obstacles. Also, it needs to move it closer to the final point which is the place we intend to be. We apply following field for obstacles: (q is vector of current position. Q^* is radius of effect. D(q) is size of vector q.)

$$U_{\text{rep}}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right)^2, & D(q) \le Q^*, \\ 0, & D(q) > Q^*, \end{cases}$$

whose gradient is

$$\nabla U_{\text{rep}}(q) = \begin{cases} \eta\left(\frac{1}{Q^*} - \frac{1}{D(q)}\right)\frac{1}{D^2(q)}\nabla D(q), & D(q) \le Q^*, \\ 0, & D(q) > Q^*, \end{cases}$$

Another field that creates attraction for final target is added on the field above which is defined as follow:

$$U_{\text{att}}(q) = \frac{1}{2}\zeta d^2(q, q_{\text{goal}}),$$

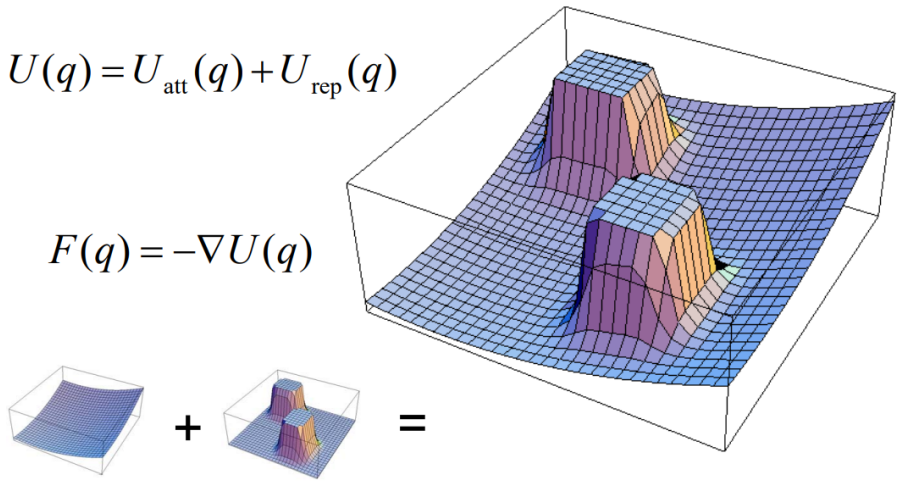
with the gradient

$$\begin{aligned} \nabla U_{\text{att}}(q) &= \nabla\left(\frac{1}{2}\zeta d^2(q, q_{\text{goal}})\right), \\ &= \frac{1}{2}\zeta \nabla d^2(q, q_{\text{goal}}), \\ &= \zeta(q - q_{\text{goal}}), \end{aligned}$$

By superposition of both fields, desired field for path planning is achieved.

$$U(q) = U_{att}(q) + U_{rep}(q)$$

$$F(q) = -\nabla U(q)$$



Now we generate the code corresponding to our fields and algorithm.

```
function  P = Path_generator_1 (Xs, Xf, eta, alpha, B)
P=[];
k=0;
epsilon=0.1;

position=Xs;
r0=2;
```

Firstly, starting point and destination are set. Also scaling factors η for attraction and α for repulsion are given. And at the end, the struct of B is given as position of obstacles' corners. Step size is set on 0.1

```
N=5000; % to set a maximum for iteration even though we don't reach the final point
while norm(position-Xf)>0.1
    k=k+1;
    if k>N

      warning('running time error');
        break
    end
    F_repulsion=[0,0];
    for i=1:length(B)
        for j=1:length(B{i})
            p1=B{i}(j,:);
            if j~=length(B{i})
                p2=B{i}(j+1,:);
            else
                p2=B{i}(1,:);
            end
            D=[(p2-p1)', -[0,-1;1,0]*(p2-p1)'];
            if det(D)~=0
                ab=D\(position-p1)';
            end
            if ab(1)<0
                H=p1;
            elseif ab(1)>1
                H=p2;
```

Here we set desired accuracy in the final point reached which is 0.1. Also, if we path the iteration limit of 5000, path planning will be stopped and following warning will be displayed.

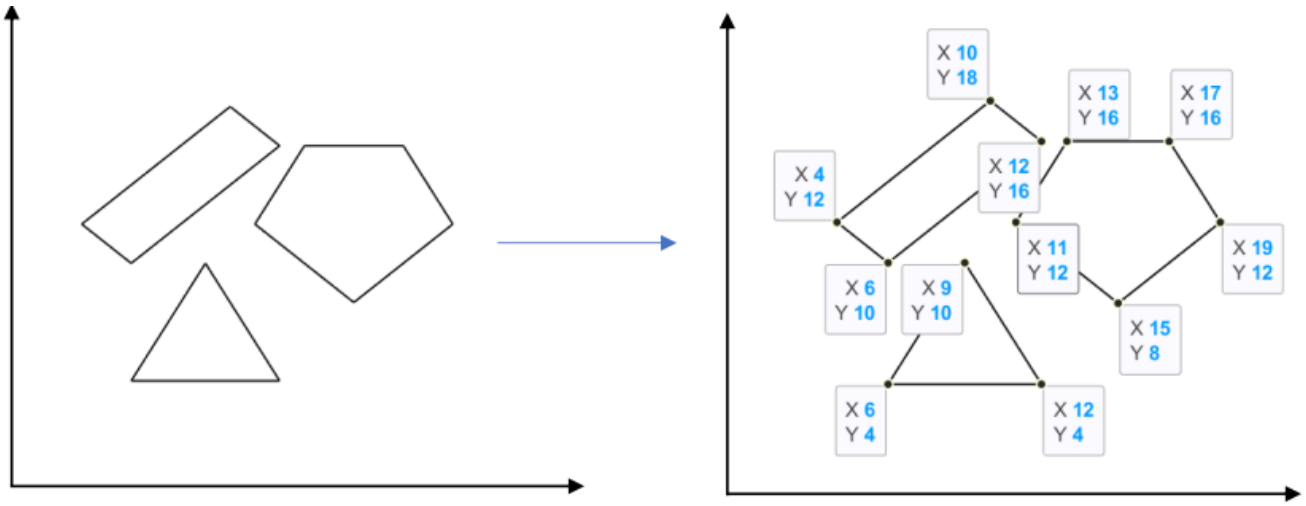
Here we enter a loop that calculates the nearest points of each obstacle and add its force to the total repulsion.

```
            else
                H=ab(1)*(p2-p1)+p1;
            end
            r=norm(position-H);
            if r<r0
                F_repulsion = alpha/r^3*(1/r - 1/r0)*(position-H) + F_repulsion;
            end
        end
    end
    F_attraction= -eta*(position-Xf);
    n=10;
    position = position +
epsilon*(F_attraction+F_repulsion)/norm((F_attraction+F_repulsion));
    P(k,:)=position;


end

end
```

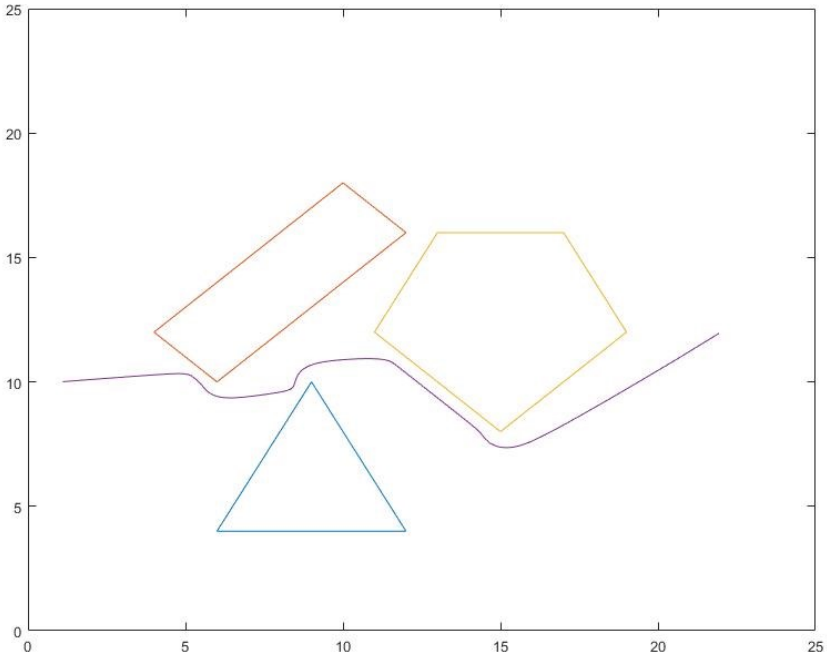
At last, we add the attraction force too and update the position with ‘epsilon’.

Problem 2

2) Test your code for the following example with $\eta = \alpha = 1$, $\varepsilon = 0.1$ 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.



The final path will be the diagram below. (Path_generator_1.mat)



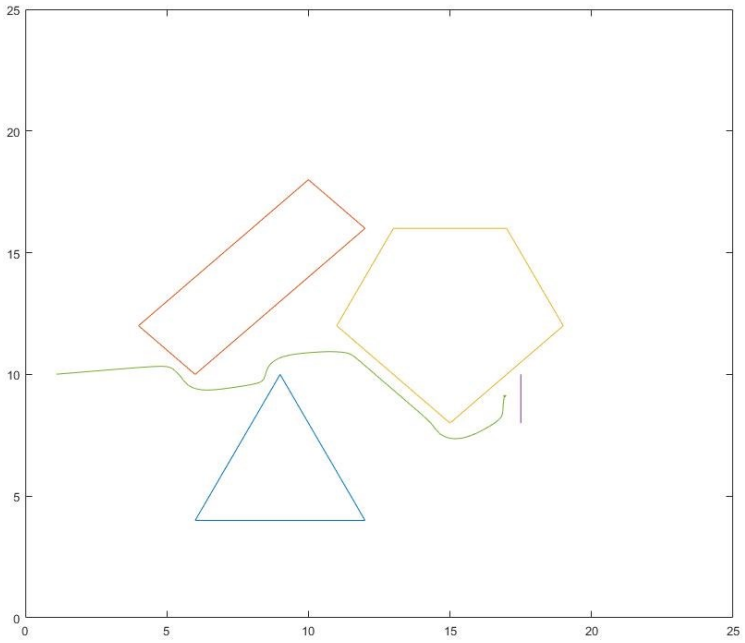
Problem 3

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

When we add the new obstacle, we get stuck in minima point with gradient zero. Thus, we reach the iteration limit of 5000 and program is terminated. (Path_generator_1: this is the same code as problem 1 and 2 but the new obstacle is included.)

```
Warning: running time error
> In Path_generator_1 (line 11)
In main (line 59)
>>
```

Path generation would be stopped (Figure below shows where it does.)



Problem 4

4) (20% Bonus Marks) Implement a random walk algorithm and see if it can solve the local minima problem of Question3. The random walk algorithm starts when the potential field algorithm (PFA) is stuck in a local minimum. It makes the robot take a random jump with a maximum length of a (let a be 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.

To avoid getting stuck, two solutions are presented:

A: Continuous With No Jump

In this algorithm (Path_generator_2) when get stuck, the attraction force will be neglected and a force that is 10 times bigger than repulsive force is applied perpendicular to original repulsive force. In this way we drive our point away from both obstacles and minimum point. All the code is the same as Path_generator_1 except the part below:

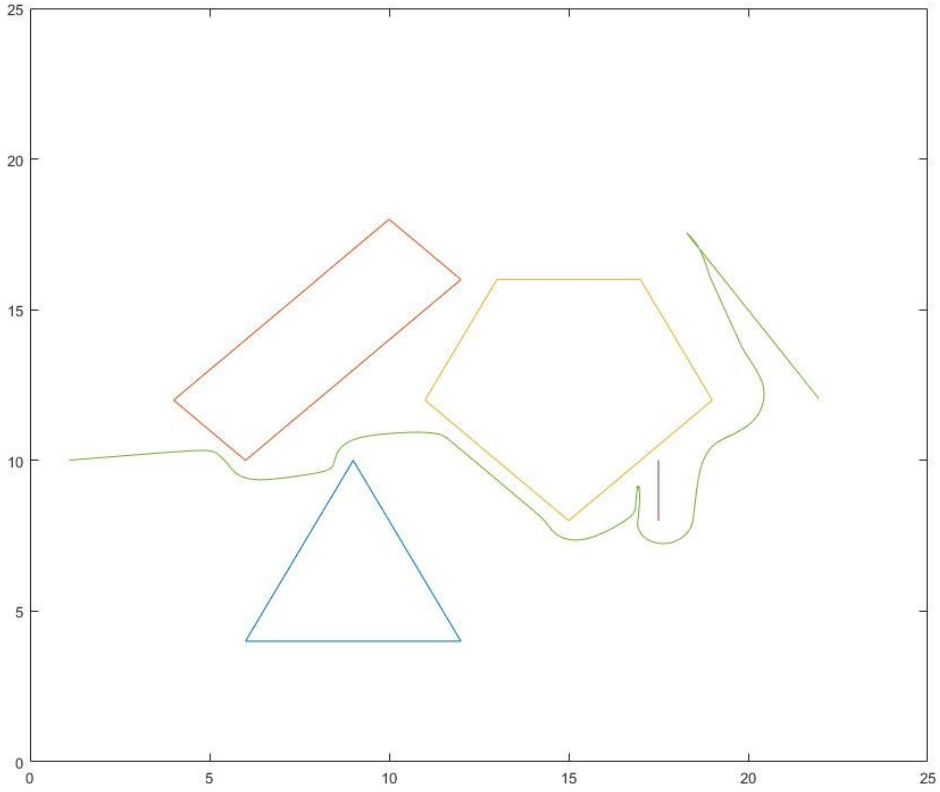
```
n=10;
    if signal(1) && norm(F_repulsion)>norm(F_attraction)*1e-6
       F_attraction =([0,-1;1,0]*F_repulsion')'*10;
       signal(2)=signal(2)+1;
    else
        signal(1)=false;
        signal(2)=0;
    end
    position = position +
epsilon*(F_attraction+F_repulsion)/norm((F_attraction+F_repulsion));
    P(k,:)=position;
    % We add next part to recognize where path gets stuck
```

We change the attraction field to a field perpendicular and 10 times bigger than attraction. The iteration is continued this way until the real attraction fades away comparing to the repulsion. Then we follow the previous algorithm.

```
if k>n
    if norm( P(k,:)-P(k-randi([1,n]),:) )<1e-6
        signal(1)=true;
    end
end
```

Here we randomly check if we get stuck or not by checking if any of previous 10 points in iteration are same as current one

The path will be the figure below:



However, the next algorithm (Long Jump) will be more efficient and have higher chance of finding a way to the target.

B: Long Jump

In this solution (Path_generator_3) we check if we get stuck or not the same way as previous algorithm. But for pulling out of local minimum, we search for closest obstacle and try to pass around it with a small margin. We pass by a jump nearly as long as distance between local minimum and obstacle's corner. We choose the corner which has smallest angular difference with target. Now we demonstrate the only differences that 'Path_generator_3' and 'Path_generator_1' have.

```
if signal
        dr_t=Xf-position;
        tar_ang=atan2(dr_t(2),dr_t(1));% This is target's angle
        dif_ang_min=pi;
        distance=norm(dr_t); % We initialize the parameter of 'distance'
        for i=1:length(B)
            ang_1=tar_ang;
            ang_2=tar_ang;
            for j=1:length(B{i})
                dr_B=B{i}(j,:)-position;
                phi=atan2(dr_B(2),dr_B(1));
                if wrapToPi(phi-ang_2)>0
                    ang_2=phi;
                    distance2=norm(dr_B);
                end
                if wrapToPi(phi-ang_1)<0
                    ang_1=phi;
                    distance1=norm(dr_B);
                end
            end

            if  distance > distance1
                dif_ang_min=ang_1-tar_ang;
                delta=-0.01;
                distance=distance1;
            end
            if  distance > distance2
                dif_ang_min=ang_2-tar_ang;
                distance=distance2;
                delta=0.01;
            end
        end
        position = position +
(distance+0.01)*[cos(tar_ang+dif_ang_min+delta),sin(tar_ang+dif_ang_min+delta)];
        k=k-3*n;
        P(k,:)=position;
        signal=false;
    else
        position = position +
epsilon*(F_attraction+F_repulsion)/norm((F_attraction+F_repulsion));
        P(k,:)=position;
    end
    %%%
    % We add next part to recognize where path gets stuck
    if k>n
        if norm( P(k,:)-P(k-randi([1,n]),:) )<1e-6
            signal=true;
        end
    end
```

The signal recognizes if it should use the algorithm of jump. Then for every obstacle, we determine the angle of view starting at ang_1 and ending at ang_2.

Then we find nearest one and try to pass around it. By this way, we can be sure that no collision with obstacles happens. At the end we delete the points belonging to when path was stuck and start from landing point.

Here we randomly check if we get stuck or not by checking if any of previous 10 points in iteration are same as current one

The path is demonstrated below: