23TTP409 Autonomous Vehicles
Coursework 1 - Path Planning and Path Following

Gabriel Wendel

First assignment in module 23TTP409 - Autonomous Vehicles

Aeronautical and Automotive Engineering
Erasmus Exchange Semester at Loughborough University
United Kingdom

Contents

Problem Formulation

The task is to design and implement a path-planning and path-following algorithm to guide the autonomous vehicle from start to goal in a square area. The vehicle must avoid four obstacles along the path: a circle, a pentagon, and a hexagon and a triangle.

1 Path Planning - Potential Field Algorithm

In order to plan a path that avoids all shape obstacles Potential Field Algorithm was implemented. The following section describes the design of the algorithm as well as the derivation process.

1.1 Theory

Potential field algorithm is gradient-based, i.e. an algorithm that utilize the gradient of a cost or objective function to guide the vehicle to the optimal path. Objective function in this case are attractive- and repulsive potentials. Together these functions form a potential field:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{1}$$

Attractive potential, $U_{att}(q)$, is defined as:

$$U_{att}(q) = \frac{1}{2}\eta\rho^2(q, q_g) \rightarrow U_{att}(x, y) = \frac{1}{2}\eta\left[(x - x_g)^2 + (y - y_g)^2\right] \tag{2}$$

where:

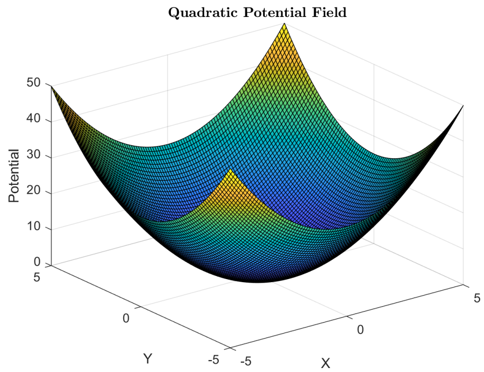
$$\begin{aligned} (x, y) &= \text{vehicle position} \\ (x_g, y_g) &= \text{obstacles position} \\ \eta &= \text{attractive potential coefficient} \\ \rho^2(q, q_g) &= \text{Euclidean distance between vehicle and goal} \end{aligned}$$

This is a quadratic potential (ρ squared) and defines the attractive potential to the goal. Repulsive potential is defined as:

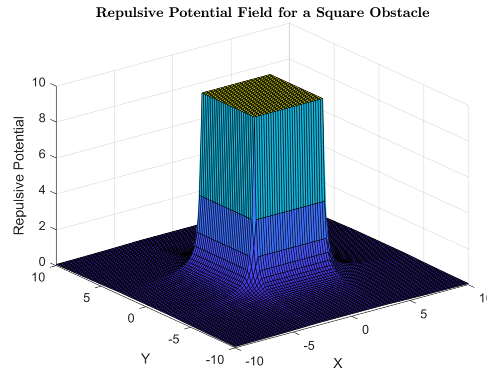
$$U_{rep}(q) = \frac{1}{2}k\left(\frac{1}{\rho(q, q_0)} - \frac{1}{\rho_0}\right) \rightarrow U_{rep}(x, y) = \frac{1}{2}k\left[\frac{1}{\sqrt{(x - x_0)^2 + (y - y_0)^2}} - \frac{1}{\rho_0}\right]^2 \tag{3}$$

Where k is the repulsive coefficient. Equation 3 governs the repulsive force exerted by all obstacles on the vehicle. Figure 1a and Figure 1b illustrates the gradient field generated by attractive- and repulsive potentials. By combining these plots, we obtain a potential field that guides the vehicle toward the goal while simultaneously avoiding obstacles. Total potential function can be expressed according to Equation 4 [2].

$$F(q) = -\nabla U(q) \tag{4}$$



(a) Quadratic Attractive Potential.



(b) Repulsive potential for a square obstacle.

Figure 1: Attractive- and repulsive potential examples.

1.2 MATLAB - Implementation

Before running the algorithm, a *radius of influence* was set. As the name suggests, it represents the distance within which obstacles will exert repulsive potential on the vehicle. Table 1 list parameters used in the algorithm.

Table 1: Potential Field Parameters

Parameter	**Name in script**	**Value**
Maximum number of steps taken by vehicle	`nMaxSeps`	800
Goal point	`xGoal`	[195, 195]m
Start point	`xStart`	$5 \leq x \leq 15$m, $10 \leq y \leq 20$m
Radius of Influence	`RadiusOfInfluence`	50m

How the algorithm works can be summarised by the following steps:

- Compute the error vector, `GoalError` between the current vehicle position and the goal.
- Determine obstacles within the influence range `RadiusOfInfluence` and calculate repulsive forces `FObjects` exerted by these obstacles, based on Equation 3.
- Calculate attractive forces `FGoal` towards the goal based on the error vector.
- Combine attractive and repulsive forces to get the total force `FTotal` acting on the vehicle.
- Limit the magnitude of the total force to achieve smooth movement.
- Update the vehicle position based on the total force.
- Compute the angle, `Theta`, of the resultant force vector `FTotal`, i.e. calculate the angle orientation that the vehicle should have based on the direction of the total force acting on it.
- The error between the goal and vehicle position is recalculated.

The algorithm runs within a while-loop, iterating as long as the magnitude of the error vector is larger than one and the maximum number of steps has not been reached. Vehicle's position is stored in vector `pos` for each iteration.

1.3 Results

The code was run three times to verify its robustness. Figure 2 - 4 illustrates the planned paths for the three runs.

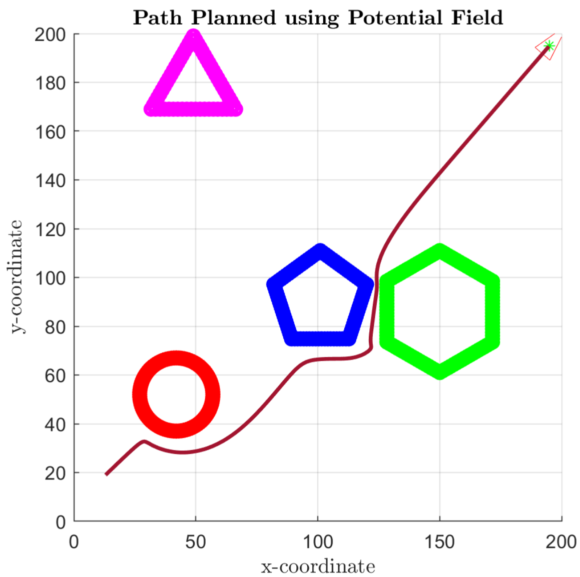


Figure 2: Planned Path first run.

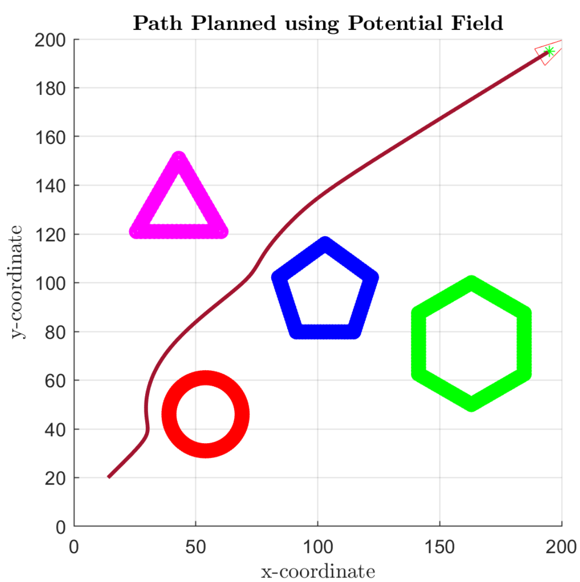


Figure 3: Planned Path second run.

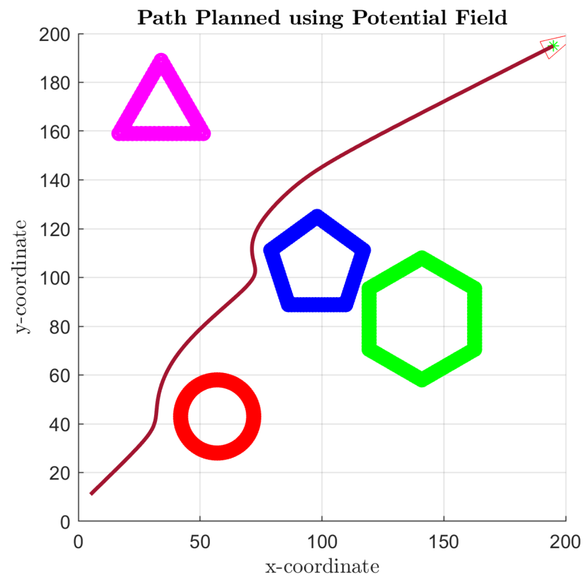


Figure 4: Planned Path third run.

1.4 Advantages, Disadvantages and Tuning Parameters

One of the main advantages of potential field is its simplicity, its quick and easy to implement and provide decent results. Y. Koren et al. have discussed the use of potential field methods for mobile robot navigation, highlighting both their advantages and inherent limitations. Some of the main disadvantages are listed below [1]:

1. Local minima can lead to situations where the vehicle becomes trapped. This situation may arise when the vehicle encounters a dead end. After multiple tests of the algorithm, this issue didn't arise, and the potential field didn't create any local minima. However, it could become problematic when encountering U-shaped obstacles [1].

2. Oscillations in Narrow Passages due to simultaneous opposing repulsive forces. This phenomena can be seen in the waypoint example plot in Figure 5 aswell as the planned path in Figure 2. At the midpoint of its journey towards the goal, the vehicle encounters two obstacles in its path, resulting in oscillations and the generation of significantly more waypoint data within a short distance.

The main tuning paramter is the radius of influence, attractive- and repulsive coefficients. A larger radius allows the algorithm to consider a broader range of nearby obstacles and goals, potentially leading to more thorough exploration of the environment. It also resulted in smoother paths. However, the vehicle might be more prone to overshooting its target with increased radius of influence.

Increasing the strength of repulsive forces makes the agent more reactive to nearby obstacles, resulting in more avoidance. If the repulsive forces are too strong, the agent may deviate too much from the optimal path and become overly cautious. Conversely, a smaller range may result in the agent reacting only when very close to obstacles, potentially leading to abrupt manoeuvres. Increasing the strength of attractive forces accelerates the agent towards the goal more rapidly, facilitating faster convergence to the target. However, excessively strong attraction may cause the agent to overshoot the goal or ignore nearby obstacles.

2 Path Following - Pure Pursuit Algorithm

To ensure that the vehicle follows the planned path Pure Pursuit Algorithm was implemented. The following section covers the theory behind the algorithm, derivation process and how it was used to solve the task at hand.

2.1 Theory

Pure Pursuit algorithm is a a geometric path tracking controller that determines the angular velocity that moves the vehicle from its current position to some look ahead-point. It tracks the previosly planned path using only the geometry of the vehicle kinematics. It uses a look ahead-point which lies on a fixed look ahead-distance in front of the vehicle. The goal is for the vehicle to move to that point using a steering angle computed by the algorithm.

Waypoints, based on the planned path, are used to compute the vehicle velocity, see Figure 5 below. The position of the vehicle is defined as x- and y-coordinate as well as angular orientation, θ.

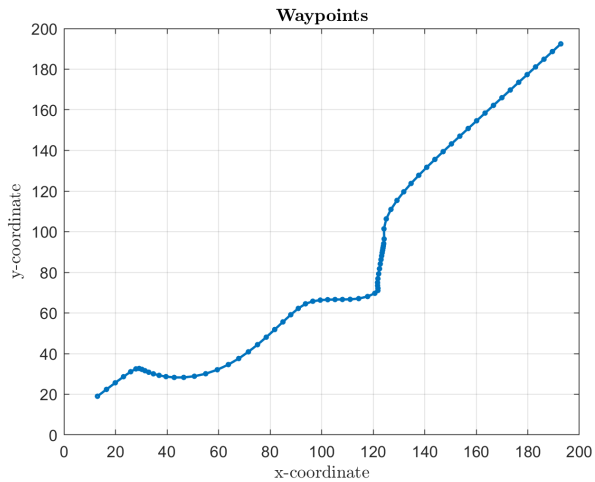


Figure 5: Example of waypoints obtained from Path Planning Algorithm.

The goal is to find steering angle, δ, such that the vehicle is aligned with the look ahead point along the planned path. We can express the distance from centre of rotation to the target point as R. The distance form centre of rear wheel to the look ahead point is the look ahead-distance, l_d. Angle, α is the target direction angle [3]. Using the law of sines we obtain:

$$\begin{aligned}
\frac{l_d}{\sin 2\alpha} &= \frac{R}{\sin\left(\frac{\pi}{2} - \alpha\right)} \\
\frac{l_d}{2\sin\alpha\cos\alpha} &= \frac{R}{\cos\alpha} \\
\frac{l_d}{\sin\alpha} &= 2R \\
k = \frac{1}{R} &= \frac{2\sin\alpha}{l_d}
\end{aligned}$$

Where k is the curvature. From the bicycle model we know that:

$$R = \frac{L}{\tan\delta}$$

From this we can express steering angle, δ, as:

$$\delta = \arctan\left(\frac{2L\sin\alpha}{l_d}\right) \tag{5}$$

2.2 MATLAB - Implementation

The algorithm was implemented using Simulink, see Figure 6.

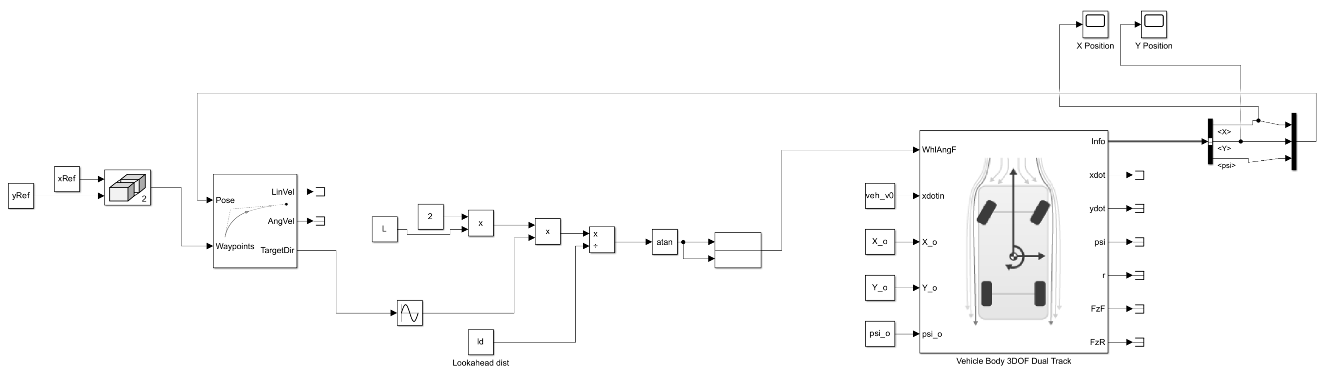


Figure 6: Pure Pursuit Simulink model.

To determine vehicle direction (TargetDir in Figure 6) Simulink's builtin *Pure Pursuit* was used. Inputs are concatenated waypoints and `pose` which represents vehicles actual position (x- and y-coordinates and angular position, θ) [4]. Steering angle is then calculated based on Equation 5 and given parameters L and l_d. Parameters implemented in the Simulink model can be found in Table 2. Simulink's *Vehicle Body 3DOF Dual Track* block was used to simulate the vehicle. It implements a rigid two-axle vehicle body model to determine longitudinal, lateral and yaw motion [5]. Inputs are x- and y-position, yaw angle and vehicle velocity. From this the actual path of the vehicle can be obtained and plotted.

Table 2: Pure Pursuit parameters.

Parameter	**Name in script**	**Value**
x-position	`X_o`	384x1 vector
y-position	`Y_o`	384x1 vector
Bicycle length	`L`	3m
Look ahead-distance	`ld`	5m
Yaw angle	`psi_o`	random value between 0°and 90 °
Initial vehicle velocity (constant)	`veh_v0`	8km/h

2.3 Results

Performance of the pure pursuit algorithm can be seen in Figure 7 - 9. The initial deviation from the planned path occurs due to the random yaw angle, represented by `psi_o = 90*rand(1)`. It also experienced some difficulty during the first run when the planned path required sudden sharp steering when the vehicle took the narrow passage between the hexagon and pentagon, see Figure 2 and 7. However, overall pure pursuit successfully followed the planned path quite effectively. Future improvement would be to make vehicle go around one of the obstacles instead of though the narrow path in order to avoid the sudden change is steering angle.

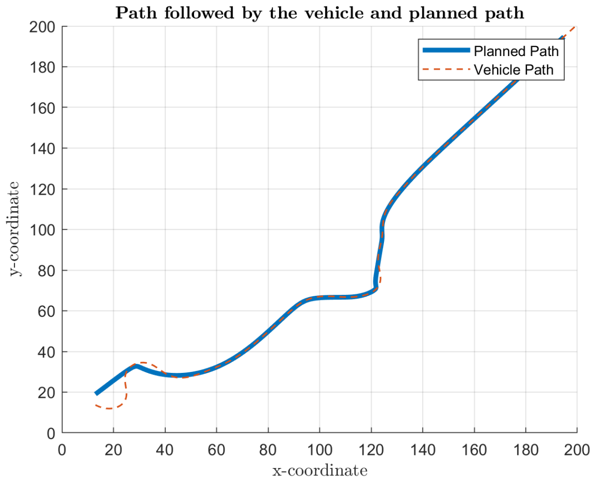


Figure 7: Path following performance first run.

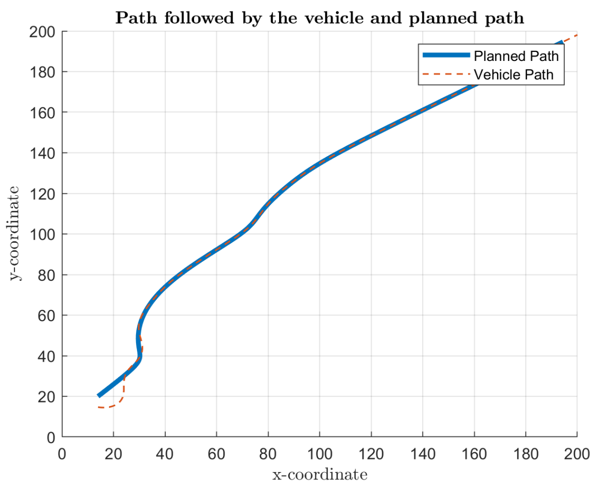


Figure 8: Path following performance second run.

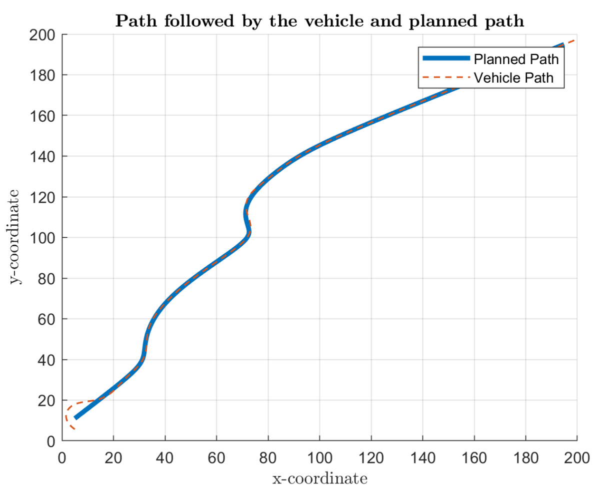


Figure 9: Path following performance third run.

2.4 Advantages, Disadvantages and Tuning Parameters

The main tuning parameter in Pure Pursuit is the look ahead-distance. It determines how far from current position along the path the vehicle should look in order to compute angular velocity. Changing this parameter affects how the vehicle tracks the path, influencing two primary objectives: path recovery and maintaining the path. To swiftly recover the path between waypoints, using a small look ahead-distance will cause the vehilce to rapidly approach and realign with the desired trajectory. However, this can result in overshooting and the vehicle starts to oscillate along the path. A longer look ahead-distance can reduce these oscillations. But a longer look ahead distance could result in larger curvatures at the planned path corners [6]. After some testing it was determined that a look ahead-distance of 5m yielded a satisfactory result.

Key limitations of Pure Pursuit are related to dynamics. The algorithm assumes perfect response to path curvatures. This means that a sudden sharp change in curvature can be requested, which would cause a real vehicle to skid. Additionally the vehicle will not close on the path as quickly as desired because of the first order lag in steering [7].

Appendix

Listing 1: MATLAB code Potential Field

```

%% Course Work 1 Autonomous Vehicles 23TTP409 Loughborough University
% Potential field Path Planning
% Gabriel Wendel
close all;
clear all;
clc;

%% Parameters
% Vehicle starting position
x_vmin=5;
x_vmax=15;
y_vmin=10;
y_vmax=20;
x_veh=randi([x_vmin x_vmax]);
y_veh=randi([y_vmin y_vmax]);

%% Circle obstacle
xcmin=40;
xcmax=60;
ycmin=15;
ycmax=55;
xc=randi([xcmin xcmax]);
yc=randi([ycmin ycmax]);
centers_circle=[xc yc]; % coordinate of the circle centre
radius=15; % radius

th = 0:pi/50:2*pi;
x_circle = (radius * cos(th) + xc);
y_circle = (radius * sin(th) + yc);
%% Pentagon obstacle
r_outer_pent=20;%outer radius covering the pentagon
t_pent=2*r_outer_pent*sind(36); % Side of the pentagon
xpcmax=105;
xpcmin=95;
ypcmax=105;
ypcmin=90;
xpc=randi([xpcmin xpcmax]); % center points for the pentagon
ypc=randi([ypcmin ypcmax]); %
%Coordinates for the pentagon
xp1=xpc;
yp1=r_outer_pent+ypc;
xp2=xpc+(r_outer_pent*cosd(18));
yp2=ypc+(r_outer_pent*sind(18));
xp3=xpc+(r_outer_pent*cosd(-54));
yp3=ypc+(r_outer_pent*sind(-54));
xp4=(xp3-t_pent);
yp4=yp3;
xp5=xpc-(r_outer_pent*cosd(18));
yp5=yp2;

xp = [xp1 xp2 xp3 xp4 xp5];
yp = [yp1 yp2 yp3 yp4 yp5];
```

```
% Generate coordinates along each side of the pentagon
N_points_per_side = 20; % Adjust the number of points

% Initialize arrays to store x and y coordinates
x_pentagon = zeros(1, N_points_per_side * 5);
y_pentagon = zeros(1, N_points_per_side * 5);

% Generate coordinates for each side using a loop
for i = 1:5
    x_pentagon((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(xp(i), xp(mod(i,5)+1), N_points_per_side);
    y_pentagon((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(yp(i), yp(mod(i,5)+1), N_points_per_side);
end
%% Hexagon obstacle
r_outer_hex=25;%outer radius covering the hexagon
% Hexagon outer circle radius= eaxh side of the hexagon
xhcmax=175;
xhcmin=140;
yhcmax=100;
yhcmin=70;
xhc=randi([xhcmin xhcmax]); % center points for hexagon
yhc=randi([yhcmin yhcmax]);
% Hexagon coordinates
xh1=xhc;
yh1=r_outer_hex+yhc;
xh2=xhc+(r_outer_hex*cosd(30));
yh2=yhc+(r_outer_hex*sind(30));
xh3=xh2;
yh3=(yh2-r_outer_hex);
xh4=xhc+(r_outer_hex*cosd(-90));
yh4=yhc+(r_outer_hex*sind(-90));
xh5=-(-xhc+(r_outer_hex*cosd(-30)));
yh5=(yhc+(r_outer_hex*sind(-30)));
xh6=xh5;
yh6=yh5+r_outer_hex;

xh = [xh1 xh2 xh3 xh4 xh5 xh6];
yh = [yh1 yh2 yh3 yh4 yh5 yh6];

N_points_per_side = 20;

x_hexagon = zeros(1, N_points_per_side * 6);
y_hexagon = zeros(1, N_points_per_side * 6);

for i = 1:6
    x_hexagon((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(xh(i), xh(mod(i,6)+1), N_points_per_side);
    y_hexagon((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(yh(i), yh(mod(i,6)+1), N_points_per_side);
end
%% Triangle obstacle
radius_t= 20; % outer radius covering the triangle
xtcmax=50;
xtcmin=25;
```

```
ytcmax=180;
ytcmin=130;
xtc=randi([xtcmin xtcmax]);
ytc=randi([ytcmin ytcmax]); % triangle center points
% Coordinates of the triangle
xt1=xtc;
yt1=radius_t+ytc;
xt2=radius_t*cosd(30)+xtc;
yt2=(-radius_t*sind(30)+ytc);
xt3=-radius_t*cosd(30)+xtc;
yt3=yt2;

xt = [xt1 xt2 xt3];
yt = [yt1 yt2 yt3];

N_points_per_side = 20;

x_triangle = zeros(1, N_points_per_side * 3);
y_triangle = zeros(1, N_points_per_side * 3);

for i = 1:3
    x_triangle((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(xt(i), xt(mod(i,3)+1), N_points_per_side);
    y_triangle((i-1)*N_points_per_side + 1 : i*N_points_per_side)...
        = linspace(yt(i), yt(mod(i,3)+1), N_points_per_side);
end

Obs_x = [x_circle x_pentagon x_hexagon x_triangle]';
Obs_y = [y_circle y_pentagon y_hexagon y_triangle]';
Obs = [Obs_x, Obs_y];

%% Map
MapSize = 200; % 200 X 200 square
nObs = length(Obs);
Map = Obs';
hold on
% illustrare shape obstacles in different colors
plot(Obs(1:length(x_circle), 1), Obs(1:length(x_circle), 2), 'ro', 'LineWidth', 2, 'MarkerSize', 6);
plot(Obs(length(x_circle)+1:length(x_circle)+length(x_pentagon), 1), ...
    Obs(length(x_circle)+1:length(x_circle)+length(x_pentagon), 2),...
    'bo', 'LineWidth', 2, 'MarkerSize', 6);
plot(Obs(length(x_circle)+length(x_pentagon)+1:length(x_circle)+length(x_pentagon)+ ...
    length(x_hexagon), 1), Obs(length(x_circle)+length(x_pentagon)+1:length(x_circle)+ ...
    length(x_pentagon)+length(x_hexagon), 2), 'go', 'LineWidth', 2, 'MarkerSize', 6);
plot(Obs(length(x_circle)+length(x_pentagon)+length(x_hexagon)+1:end, 1), Obs(length(x_circle) ...
    +length(x_pentagon)+length(x_hexagon)+1:end, 2), 'mo', 'LineWidth', 2, 'MarkerSize', 6);

axis equal
```

```
xlim([0 200])
ylim([0 200])

%% Potential Field Algorithm
% Based on code provided during tutorial

nMaxSteps = 800;

xGoal = [195; 195];
xStart = [x_veh; y_veh];
xVehicle = xStart;

RadiusOfInfluence = 100; % obstacle influence range

KGoal= 1; % attractive potential coefficient to the goal
KObj = 50; % repulsive potential coefficient to the goal

GoalError =  xGoal - xVehicle; % error vector

plot(xGoal(1),xGoal(2),'g*','MarkerSize', 6);


Hr = DrawRobot([xVehicle;0],'r',[]); % draw robot

k = 0;

pos = xStart; % store the trajectory in this array

while(norm(GoalError)>1 && k<nMaxSteps)


    % find distance to all obstacle entities
    % error vector between obstacles and the vehicle: q_obs - q
    Dp = Map-repmat(xVehicle,1,nObs);
    Distance = sqrt(sum(Dp.^2));
    % determine which obstacles that influence vehicle
    iInfluencial = find(Distance<RadiusOfInfluence);

    % if there are obstacles within influence range
    if(~isempty(iInfluencial))
        % vector sum of repulsions:
        rho = repmat(Distance(iInfluencial),2,1); %

        V = Dp(:,iInfluencial);

        DrhoDx = -V./rho;

        %          DrhoDx = -V;

        F = (1./rho-1./RadiusOfInfluence)*1./(rho.^2).*DrhoDx;

        FObjects = KObj*sum(F,2);

    else
        % nothing close
        FObjects = [0;0];
```

```
    end

    % the gradient of the attractive potential is
    FGoal = KGoal*(GoalError)/norm(GoalError);
    % normalised FGoal = KGoal*(GoalError);

    % Combine attractive and repulsive forces to get the total force acting on
        the vehicle.
    FTotal = FGoal+FObjects;

    Magnitude = min(1,norm(FTotal));

    % Limit the magnitude of the total force to achieve smooth movement
    FTotal = FTotal/norm(FTotal)*Magnitude;

    % Update the vehicle position based on the total force
    xVehicle = xVehicle+FTotal;

    k = k+1;

    % compute the angle of the resultant force vector (FTotal),
    % i.e. calculate the orientation that the vehicle should have based on the
        direction of
    % the total force acting on it.
    Theta = atan2(FTotal(2),FTotal(1));
    DrawRobot([xVehicle;Theta],'k',Hr);
    pause(0.0);
    drawnow;

    % Update error vector based on new position
    GoalError =  xGoal - xVehicle;

    % Save position in vector
    pos = [pos, xVehicle];

end

figure(1)
plot(pos(1,:),pos(2,:),'LineWidth', 2);
title('\textbf{Path Planned using Potential Field}','Interpreter','latex')
xlabel('x-coordinate','Interpreter','latex')
ylabel('y-coordinate','Interpreter','latex')
axis equal
grid on
xlim([0 200])
ylim([0 200])


%-------- Drawing Vehicle -----%
function H = DrawRobot(Xr,color,H)

p=0.02; % percentage of axes size
a=axis;
l1=(a(2)-a(1))*p;
l2=(a(4)-a(3))*p;
P=[-1 1 0 -1; -1 -1 3 -1];%basic triangle
```

```
theta = Xr(3)-pi/2;%rotate to point along x axis (theta = 0)
c=cos(theta);
s=sin(theta);
P=[c -s; s c]*P; % rotate by theta
P(1,:)=P(1,:)*l1+Xr(1); % scale and shift to x
P(2,:)=P(2,:)*l2+Xr(2);
if(isempty(H))
    H = plot(P(1,:),P(2,:),color,'LineWidth',0.1);
else
    set(H,'XData',P(1,:));
    set(H,'YData',P(2,:));
end
end
```

Listing 2: MATLAB code Pure Pursuit

```

%% Course Work 1 Autonomous Vehicles 23TTP409 Loughborough University
% Pure Pursuit Path Planning
% Gabriel Wendel

% reference points
refPose = pos;
xRef = refPose(1,:)';
yRef = refPose(2,:)';

Ts = 40; % simulation time
L = 3; % bicycle length
ld = 5; % lookahead distance
X_o = refPose(1,1); % initial vehicle position
Y_o = refPose(1,2); % initial vehicle position
psi_o = 90*rand(1); % initial yaw angle
veh_v0= 8; % initial vehicle velocity m/s;

% paramters from Simulink
SimOut = sim('CW1_Pure_Pursuit_sim.slx');

figure(2)
hold on

% Plot the planned path
plot(xRef, yRef, 'LineWidth', 3)

% Plot the vehicle path with red circles
plot(SimOut.real_x_pos(:,2), SimOut.real_y_pos(:,2), '--', 'LineWidth', 1)

legend('Planned Path', 'Vehicle Path')
title('\textbf{Path followed by the vehicle and planned path}','Interpreter','
    latex');
xlabel('x-coordinate','Interpreter','latex')
ylabel('y-coordinate','Interpreter','latex')
xlim([0 200])
ylim([0 200])
grid on
hold off
```

References

[1] Y. Koren, J. Borenstein. *Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation*. The University of Michigan, Ann Arbor, Michigan. 07-04-1991. Accessed: 15-03-2014. [Online]. Available: https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated1/borenstein_potential_field_limitations.pdf

[2] Choset Howie, Lee Ji Yeong, G.D. Hager, Z.Dodds. *Robotic Motion Planning Potential Functions*. Carnegie Mellon's School of Computer Science, Pittsburgh, Pennsylvania. Accessed: 14-03-2024. [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap4-Potential-Field_howie.pdf

[3] Ding Yan.*Three Methods of Vehicle Lateral Control: Pure Pursuit, Stanley and MPC*. Medium. 06-03-2020. Accessed: 13-03-2024. [Online]. Available: https://dingyan89.medium.com/three-methods-of-vehicle-lateral-control-pure-pursuit-stanley-and-mpc-db8cc1d32081

[4] *Pure Pursuit*. Mathworks. 2024. Accessed: 12-03-2024. [Online]. Available: https://se.mathworks.com/help/robotics/ref/purepursuit.html.

[5] *Vehicle Body 3DOF*. Mathworks. 2024. Accessed: 12-03-2024. [Online]. Available:https://se.mathworks.com/help/vdynblks/ref/vehiclebody3dof.html.

[6] *Pure Pursuit Controller*. Mathworks. 2024. Accessed: 12-03-2024. [Online]. Available: https://se.mathworks.com/help/nav/ug/pure-pursuit-controller.html.

[7] Coulter, R. *Implementation of the Pure Pursuit Path Tracking Algorithm.*. Carnegie Mellon University, Pittsburgh, Pennsylvania. 01-1990. Accessed: 13-03-2024. [Online]. Available: https://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf