AmirKabir University of Technology
(Tehran Polytechnic)

Electrical Engineering Faculty
Control Department

Multi-Agent Systems
Course Project

Implementation of CAVs Flocking on Lane-Free Road

By
Mohammad Azimi

Supervisor
Dr. Atrianfar

Spring 2024

Contents

Introduction

In recent years, the study of multi-agent systems has garnered significant attention due to its wide array of applications across various fields, including robotics, computer science, and transportation. Multi-agent systems consist of multiple interacting agents, each with the ability to make decisions, communicate, and collaborate to achieve common goals. This paradigm has proven to be particularly effective in scenarios where centralized control is impractical or inefficient. One fascinating application of multi-agent systems is in the coordination of Connected and Autonomous Vehicles (CAVs) on lane-free roads, leveraging principles derived from flocking algorithms.

Flocking algorithms, inspired by the collective behavior observed in biological systems such as bird flocks and fish schools, offer a robust framework for coordinating the movements of multiple agents. Reza Olfati-Saber's seminal work on flocking algorithms for multi-agent dynamic systems has laid the groundwork for understanding and implementing these behaviors in engineered systems. His 2006 paper, "Flocking for Multi-Agent Dynamic Systems: Algorithms and Theory," provides a comprehensive analysis of the control laws and theoretical principles that underpin flocking behavior. Olfati-Saber's work demonstrates how artificial potential functions can be used to design control laws that ensure cohesion, separation, and alignment among agents, thereby enabling stable and efficient group movement.

When applied to CAVs on lane-free roads, flocking algorithms can significantly enhance traffic flow, safety, and energy efficiency. Lane-free roads represent an innovative departure from traditional traffic management, allowing vehicles to move freely within the road space rather than being confined to predefined lanes. This flexibility necessitates sophisticated coordination mechanisms to prevent collisions and ensure smooth traffic dynamics. Flocking algorithms provide a natural solution to this challenge by enabling CAVs to dynamically adjust their positions and velocities based on the relative positions and velocities of their neighbors.

In essence, the application of flocking algorithms to CAVs on lane-free roads involves adapting the principles of cohesion, separation, and alignment to the context of vehicular movement. Cohesion ensures that vehicles remain close to their neighbors, promoting group integrity; separation prevents vehicles from getting too close to avoid collisions; and alignment ensures that vehicles travel in a coordinated manner, aligning their directions and speeds. By implementing these principles, CAVs can achieve self-organized, decentralized coordination that mimics the fluid and adaptive movements seen in natural flocks.

The integration of flocking algorithms into the control systems of CAVs has the potential to revolutionize urban transportation. It promises to reduce traffic congestion, enhance safety, and improve fuel efficiency by enabling vehicles to move in harmonious formations. As research in this area continues to advance, the insights gained from Olfati-Saber's work on multi-agent flocking algorithms will undoubtedly play a crucial role in shaping the future of autonomous vehicular systems and intelligent transportation networks.

Preliminaries

Before delving into the specifics of the flocking algorithms, it is essential to understand the foundational concepts and mathematical formulations that underpin the analysis and design of these algorithms. The preliminaries cover the basic definitions, notations, and theoretical constructs necessary for a comprehensive understanding of multi-agent systems and flocking behavior.

Agents' Dynamics

The main agents of the system which in our case are vehicles are modeled as following:

$$\dot{q}_i = p_i$$

$$\dot{p}_i = u_i$$

where q_i and p_i are the position and the velocity of the i_{th} agent. Since we are considering the system as a flock of ground vehicles, both q and p state variables are of R^2 space. From now, the vehicles of the flock are recalled as α-agents since other types of agents in the system will be defined.

Proximity Nets

The set of spatial neighbors of each agent is defined as follows:

$$N_i = \{(i,j) \subset V \times V : \|q_j - q_i\| < r, i \neq j\}$$

where $r > 0$ is the interaction range between the agents and $\|.\|$ is the Euclidean norm. With the help of such definition, the adjacency matrix of the system could be constructed at any moment.

σ-Norms and Smooth Adjacency Elements

To construct a smooth collective potential of a flock and spatial adjacency matrix of a proximity net, we need to define a nonnegative map called σ-norm.

$$\|z\|_\sigma = \frac{1}{\epsilon}\left[\sqrt{1 + \epsilon\|z\|^2} - 1\right]$$

where $\epsilon > 0$.

Bumper Functions

A bump function is a scalar function that smoothly varies between 0 and 1. Here, we use bump functions for construction of smooth potential functions with finite cut-offs and smooth adjacency matrices. One possible choice is:

$$\rho_h(z) = \begin{cases} 1, & z \in [0, h) \\ \frac{1}{2}\left[1 + \cos\left(\pi \frac{z-h}{1-h}\right)\right], & z \in [h, 1] \\ 0, & otherwhere \end{cases}$$

where $h \in (0,1)$.

Adjacency Matrix

Now the adjacency matrix could be defined as:

$$a_{ij}(q) = \rho_h\left(\frac{\|q_j - q_i\|_\sigma}{r_\alpha}\right) \in [0,1]$$

where $r_\alpha = \|r\|_\sigma$ and $\alpha_{ii}(q) = 0$ for all i and q.

Potential Functions

A smooth pairwise attractive/repulsive potential with a finite cut-off at $r_\alpha = \|r\|_\sigma$ and a global minimum at $z = d_\alpha = \|d\|_\sigma$ is defined as:

$$\psi_\alpha(z) = \int_{d_\alpha}^{z} \phi_\alpha(s) ds$$

where

$$\phi_\alpha(z) = \rho_h\left(\frac{z}{r_\alpha}\right)\phi(z - d_\alpha)$$

where

$$\phi(z) = \frac{1}{2}[(a+b).\sigma_1(z+c) + (a-b)]$$

where $\sigma_1(z) = \frac{z}{\sqrt{1+z^2}}$ and $0 < a \leq b$ and $c = \frac{|a-b|}{\sqrt{4ab}}$.

Algorithm III of Olfati-Saber

While Olfati-Saber introduces three types of flocking algorithms, we discuss only Algorithm III since it covers the previous ones too. Algorithm III is capable of collision avoidance between the α-agents, following the virtual leader position and velocity as the reference values and obstacle avoidance. The control signal that achieves all these goals is defined as:

$$u_i = u_i^{\alpha} + u_i^{\beta} + u_i^{\gamma}$$

where

$$u_i^{\alpha} = c_1^{\alpha} \sum \phi_{\alpha}(\|q_j - q_i\|_{\sigma}) n_{ij} + c_2^{\alpha} \sum a_{ij}(q)(p_j - p_i)$$

$$u_i^{\beta} = c_1^{\beta} \sum \phi_{\beta}(\|\hat{q}_{i,k} - q_i\|_{\sigma}) \hat{n}_{ij} + c_2^{\beta} \sum b_{i,k}(q)(\hat{p}_{i,k} - p_i)$$

and

$$u_i^{\gamma} = -c_1^{\gamma}(q_i - q_r) - c_2^{\gamma}(p_i - p_r)$$

α-Agents Collision Avoidance and α-Lattice Structure

The first term of the control signal, u_i^{α}, is responsible for avoiding the α-agents collide with each other and meanwhile, forming them into an α-lattice structure. c_i^{α} are positive constants and $n_{i,j}$ is defined as:

$$n_{i,j} = \frac{q_j - q_i}{\sqrt{1 + \epsilon \|q_j - q_i\|^2}}$$

Obstacle Avoidance

The corresponding control signal to avoid the obstacles on the road include some terms that are not defined yet. First, the β term is mentioning to the obstacles and a β-agent is defined as the projection of each α-agent on the wall of the obstacles. Accordingly, every β-agent is represented with the same dynamics as α-agents. Assuming that the obstacle is spherical with the radius of R_k centered at y_k, the position and velocity of a β-agent are given by:

$$\hat{q}_{i,k} = \mu q_i + (1 - \mu) y_k$$

$$\hat{p}_{i,k} = \mu P p_i$$

where $\mu = \frac{R_k}{\|q_i - y_k\|}$, $P = I - a_k a_k^T$ and $a_k = \frac{q_i - y_k}{\|q_i - y_k\|}$.

Having obtained the β-agent states, the following terms are defined:

$$b_{i,k}(q) = \rho_h\left(\frac{\left\|\hat{q}_{i,k} - q_i\right\|_\sigma}{d_\beta}\right)$$

$$\phi_\beta(z) = \rho_h\left(\frac{z}{d_\beta}\right)\left[\sigma_1(z - d_\beta) - 1\right]$$

$$\hat{n}_{i,j} = \frac{\hat{q}_{i,k} - q_i}{\sqrt{1 + \epsilon\left\|\hat{q}_{i,k} - q_i\right\|^2}}$$

Virtual-Leader-Following

A virtual leader could be considered for the system such that causes the agents to converge on a reference position and velocity. Such leader could be modeled as:

$$\dot{q}_r = p_r$$

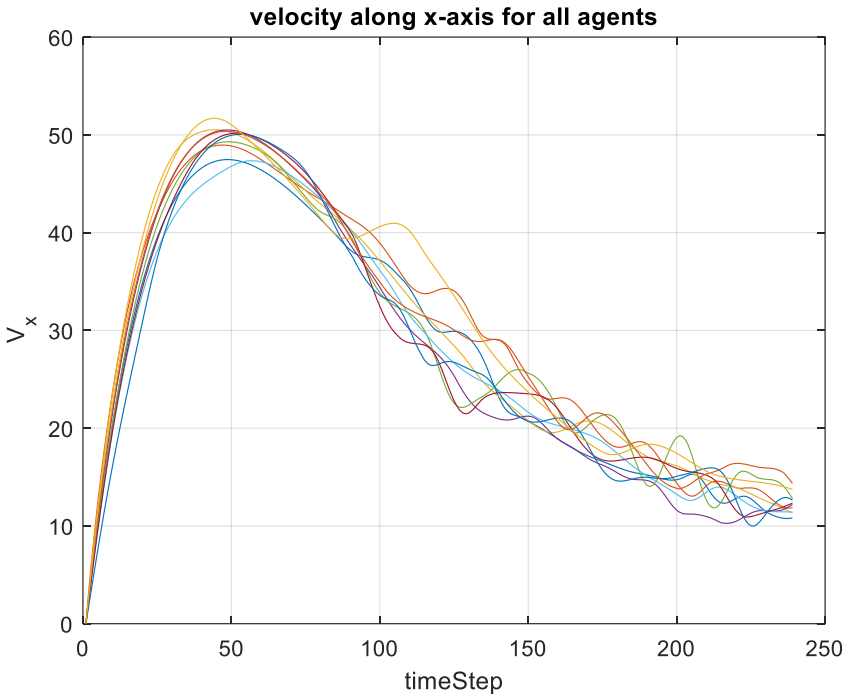
$$\dot{p}_r = \not{f}_v(q_v, q_v)$$

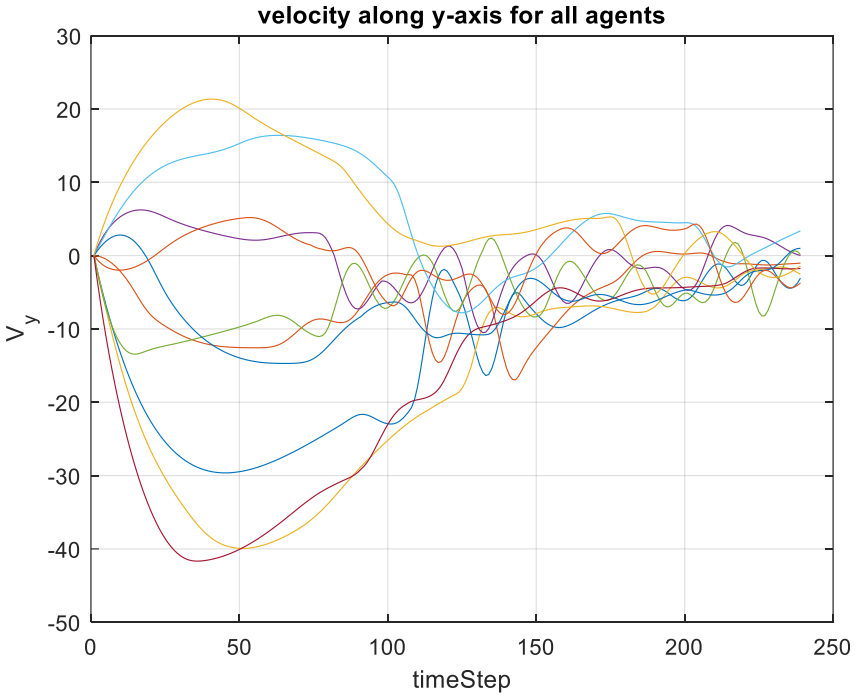
For now, it is assumed that the virtual leader is following a straight line with $\dot{p}_r = 0$. Taking such dynamics for the virtual leader, the control term u_i^γ attracts the agents to the reference values of position and velocity.

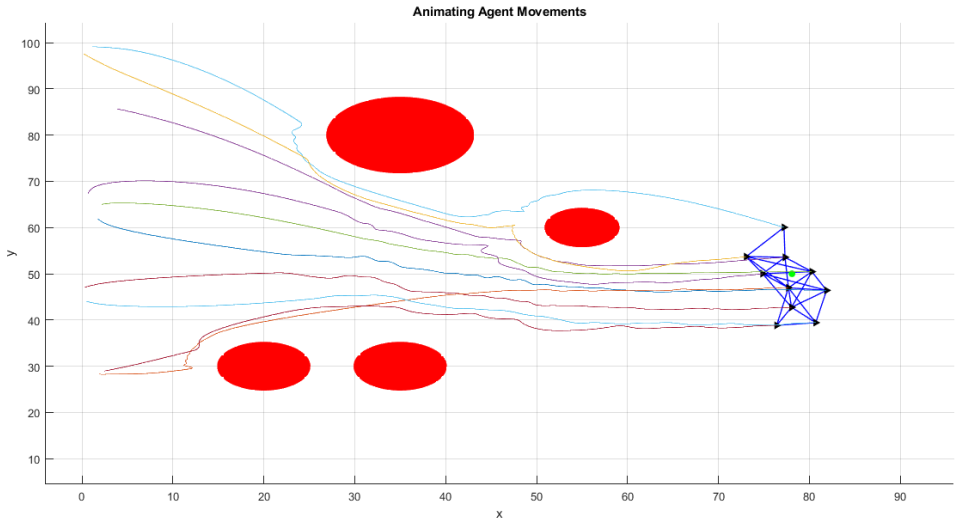
Simulation

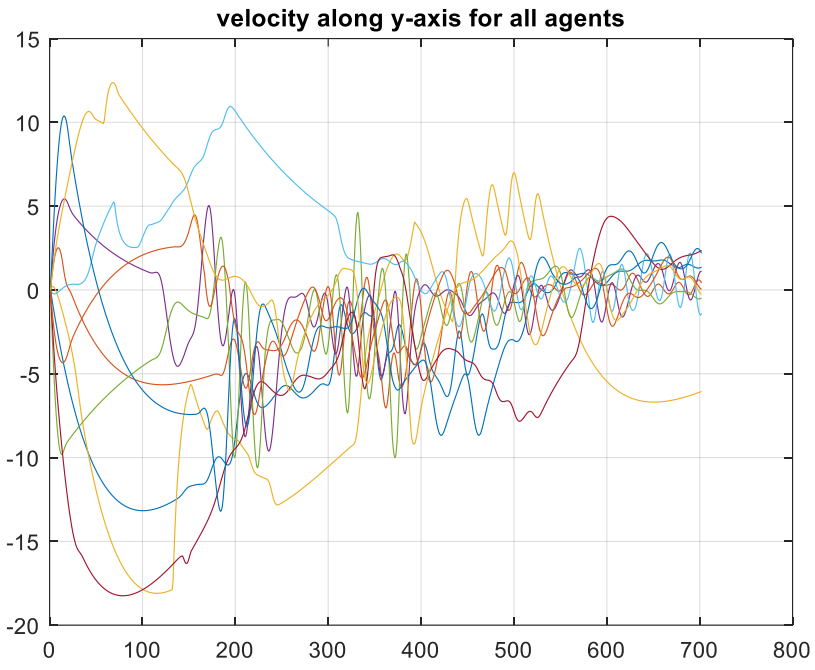
Taking the following parameters for the MAS we are having leads to the following results shown in the following images.

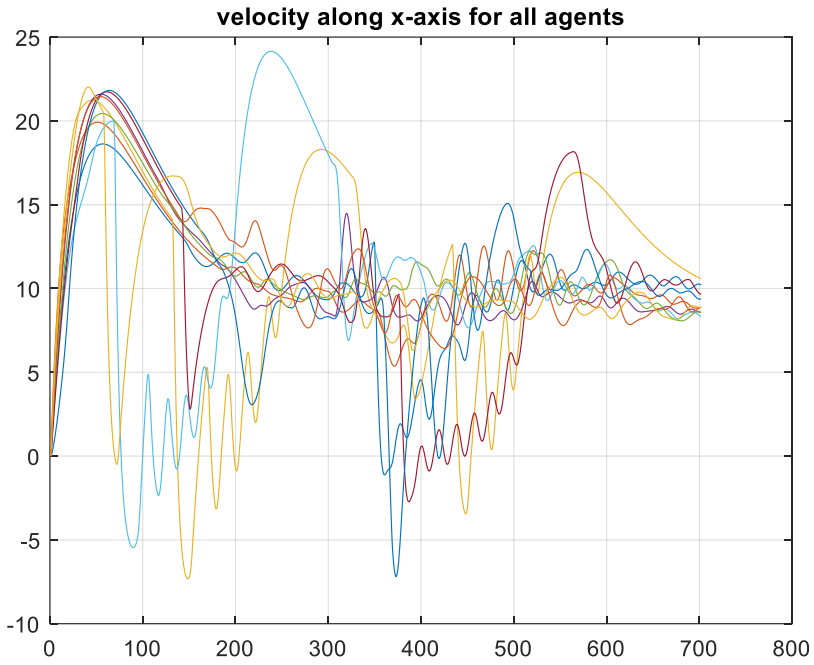
N	10
d	7
r	$1.2d$
a	5
b	5
h_a	0.2
h_b	0.9
c_1^α	10
c_1^γ	5
$c_2^{\alpha,\gamma}$	$2\sqrt{c_1^{\alpha,\gamma}}$


velocity along x-axis for all agents
60
50
40
30
20
10
0
V_x
0
50
100
150
200
250
timeStep


velocity along y-axis for all agents
30
20
10
0
-10
-20
-30
-40
-50
V_y
0
50
100
150
200
250
timeStep


Animating Agent Movements
y
x
100
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90


velocity along y-axis for all agents
15
10
5
0
-5
-10
-15
-20
0
100
200
300
400
500
600
700
800



Multi-Dimensional c_i^γ

The scalar c_i^γ is the weight of the control signal term for following the virtual leader. By choosing c_i^γ in $\mathcal{R}^2$, we can modify the agents' behavior along the x and y axis. For-example, taking $c_i^\gamma = \begin{bmatrix}5\\1\end{bmatrix}$ makes the agent follow the leader along x-axis 5 times harder than the y-axis. This relation must be chosen wisely to take the benefit of it.

Adding Road Boundaries

To step a little forward to get more similar to a road environment, road boundaries must be modeled too. Road boundaries could be considered as a new type of obstacles which is not spherical. Therefore, the previous projection of the agents on the obstacles is not useful anymore.

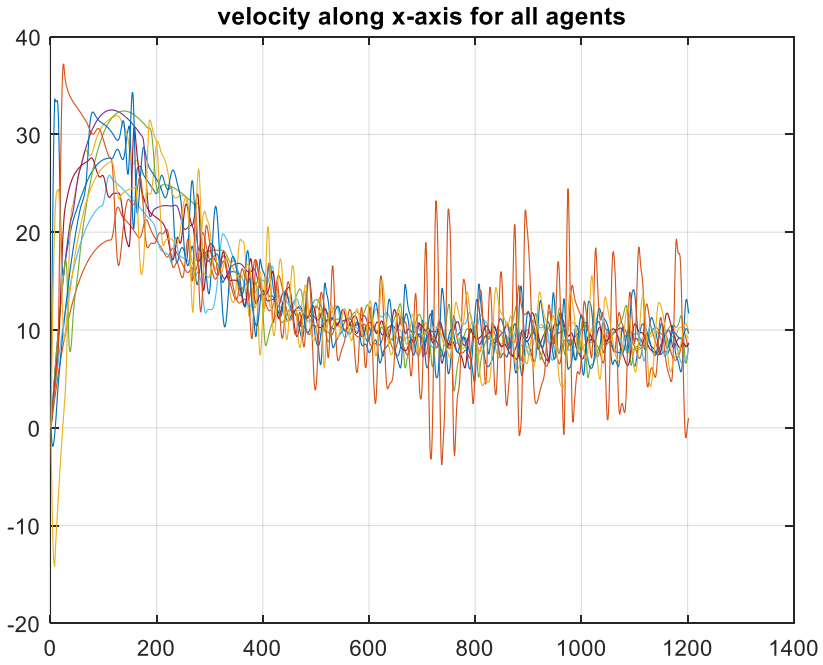
Since the road is being assumed straight, the road boundaries are like some walls on the left and right side oof the road. Hance the projection of any agent on the boundaries could be represented as:

$$\hat{q}_{i,k} = \begin{bmatrix} q_i^x \\ y_k \end{bmatrix}$$

$$\hat{p}_{i,k} = \begin{bmatrix} p_i^x \\ 0 \end{bmatrix}$$

This projection is stating that the position of the projection has the same x-axis value of the agent while its y-axis value matches with y-axis value of the boundary which is called y_k here.

Now, considering left and right road boundaries and simulating the system on a road with the width of 16 meters, leads to the following results:



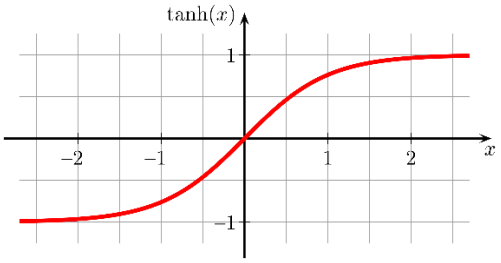
Dynamic Weights

There are some certain conditions in the system runtime that some agents are trapped in some blind spots such that control signal terms cancel each other and the agent is not able to change

in its states. To overcome this issue, one solution is to make the wight factors to vary over the conditions.

Variant Function Design

One proper function could be designed with the help of a bounded-output function as $y = \tanh(x)$. This function is shown graphically in the following figure.



Now, we could define:

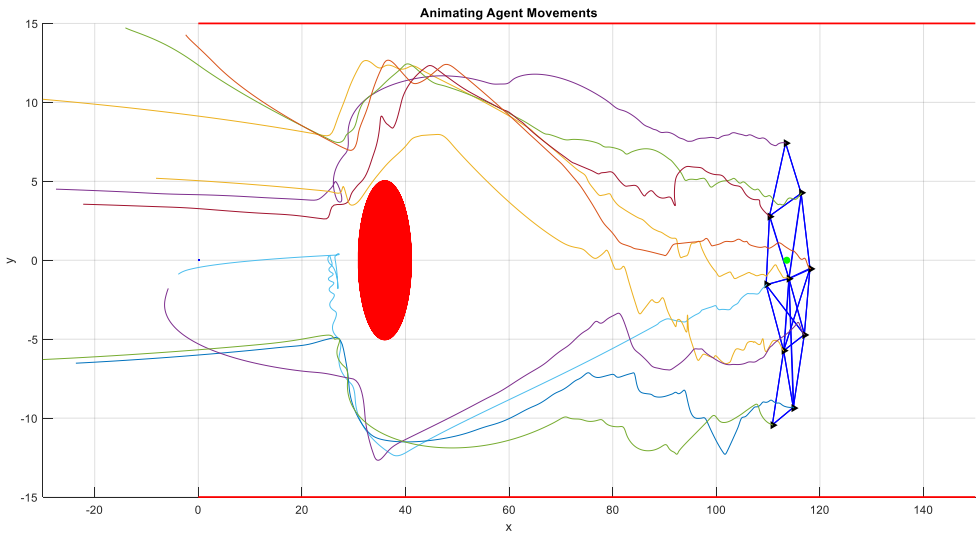
$$\boldsymbol{C(x,u) = (x)(tanh(u) + 1)}$$

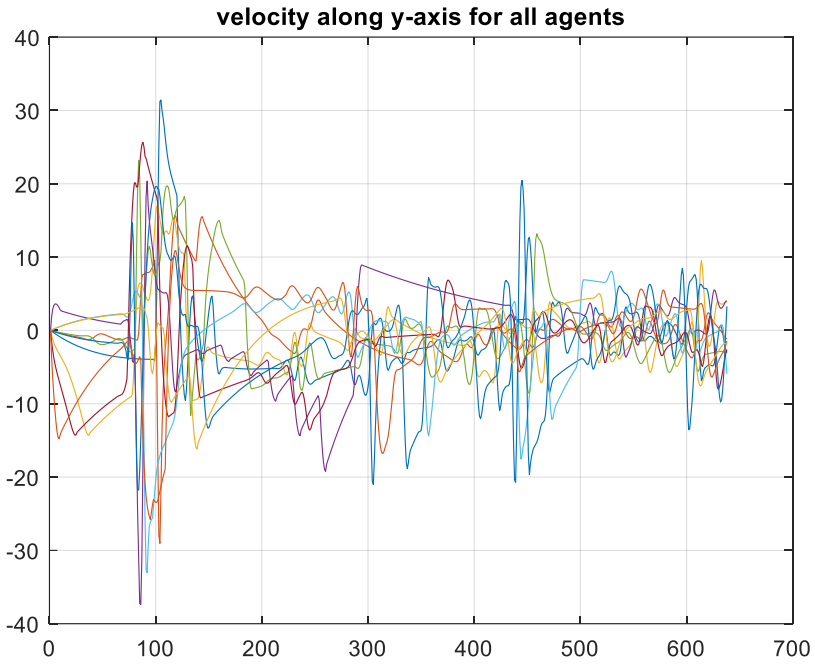
and use this function in the leader-following control signal term, we get:

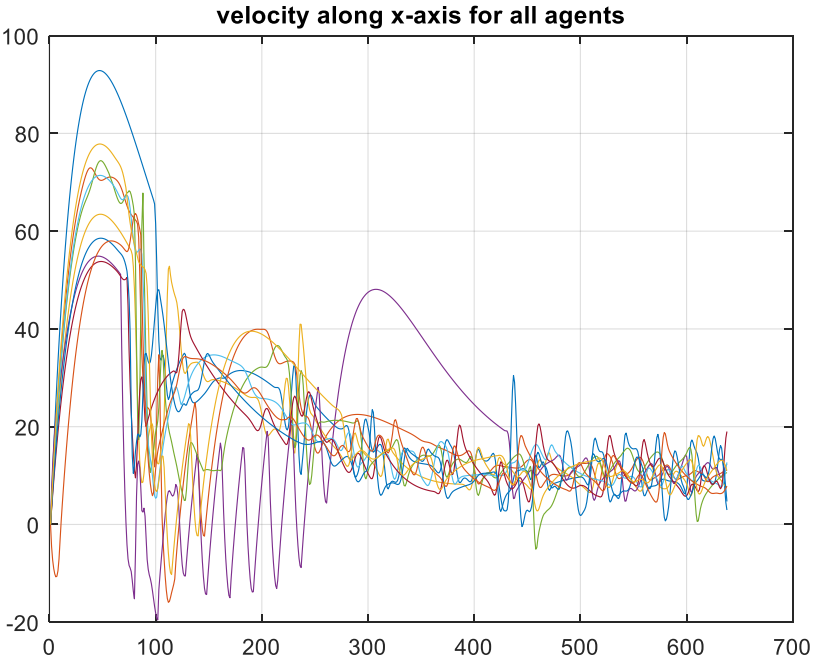
$$\boldsymbol{u_i^{\gamma} = -[C(c_1^{\gamma}, u_i^{\beta})].(q_i - q_r) - [C(c_2^{\gamma}, u_i^{\beta})].(p_i - p_r)}$$

This modification causes the agents to ignore the leader-following task whenever sensing any β-agents around. This ignorance could be helpful for the agents trapped in blind spots to escape from them.

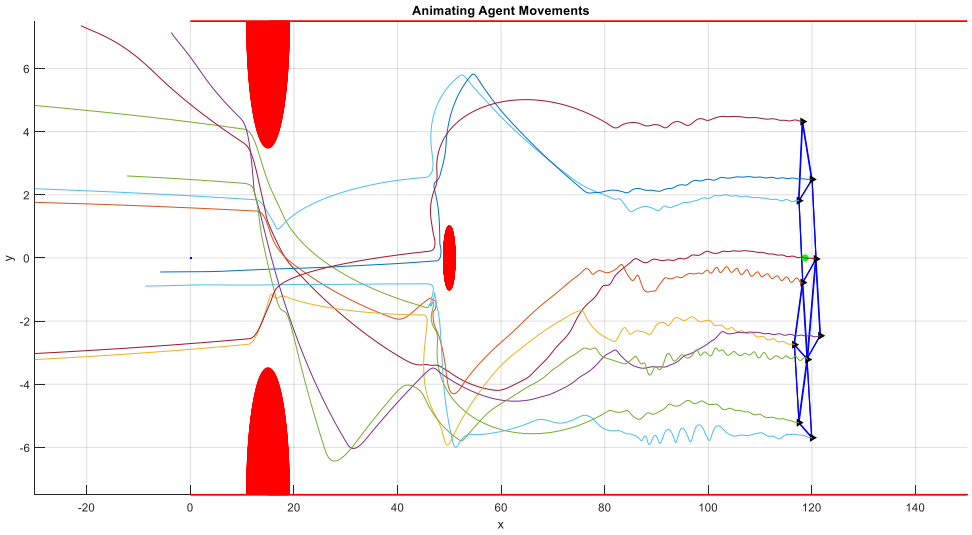
Having implemented this modification, a new simulation environment set is as follows:

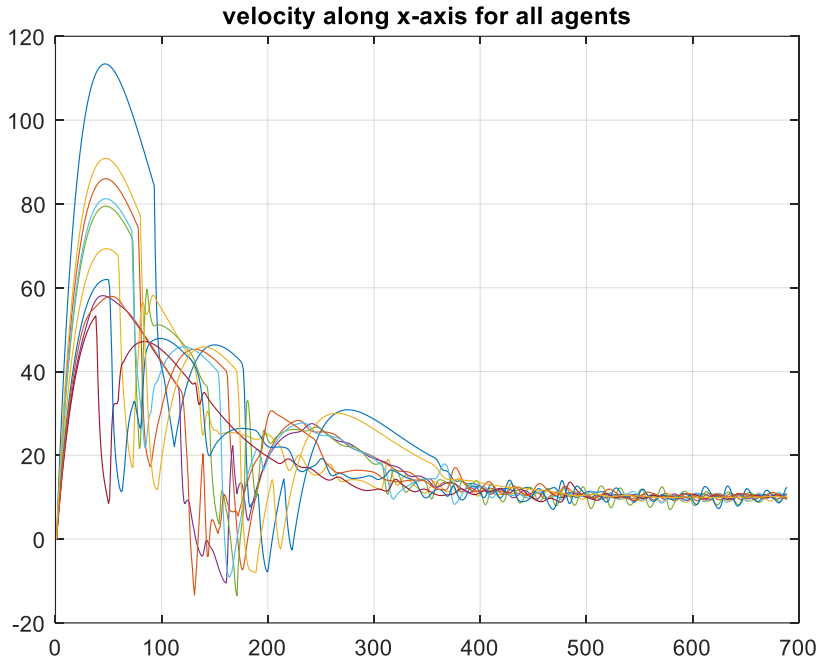


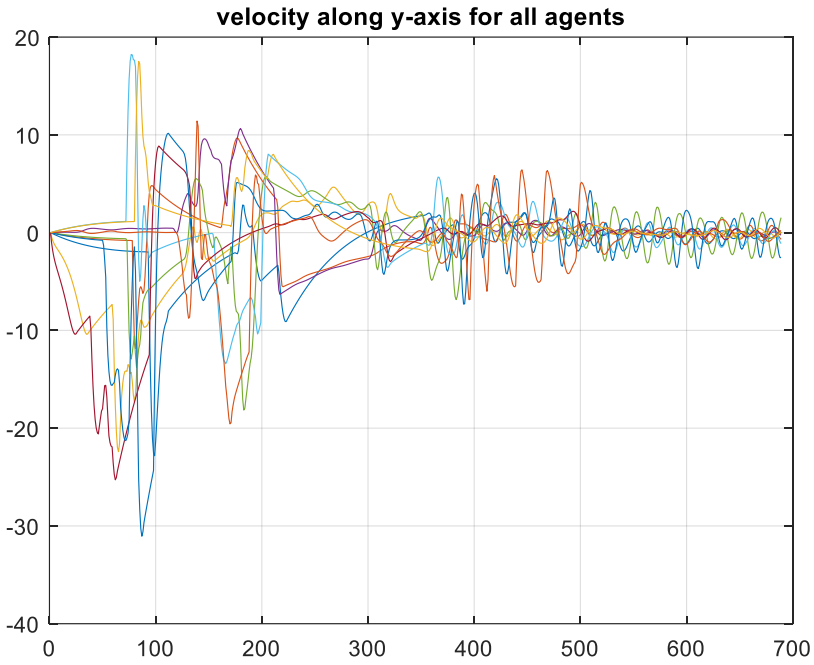




Making the environment a bit harder by adding some obstacles will lead to:

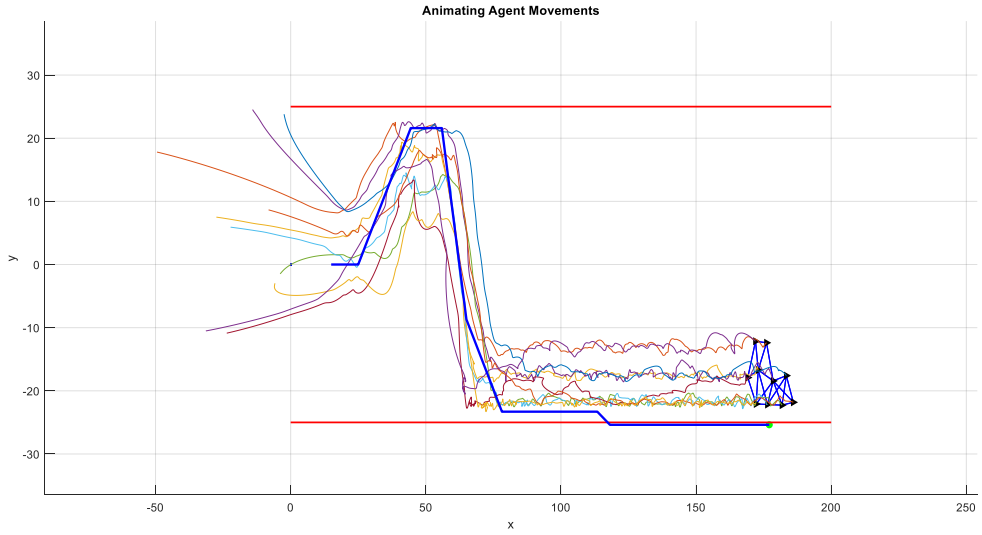

Animating Agent Movements
y
x
6
4
2
0
-2
-4
-6
-20
0
20
40
60
80
100
120
140

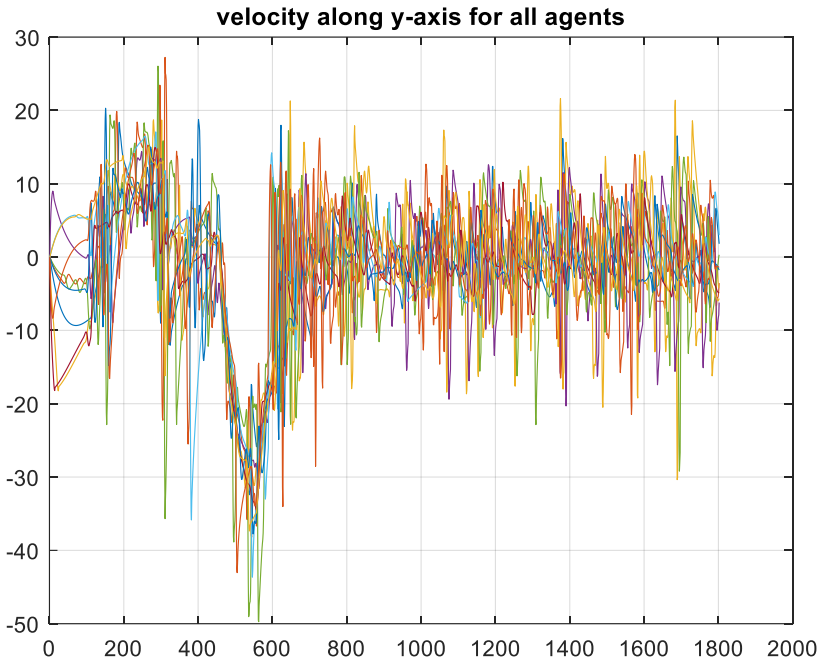

velocity along x-axis for all agents
120
100
80
60
40
20
0
-20
0
100
200
300
400
500
600
700

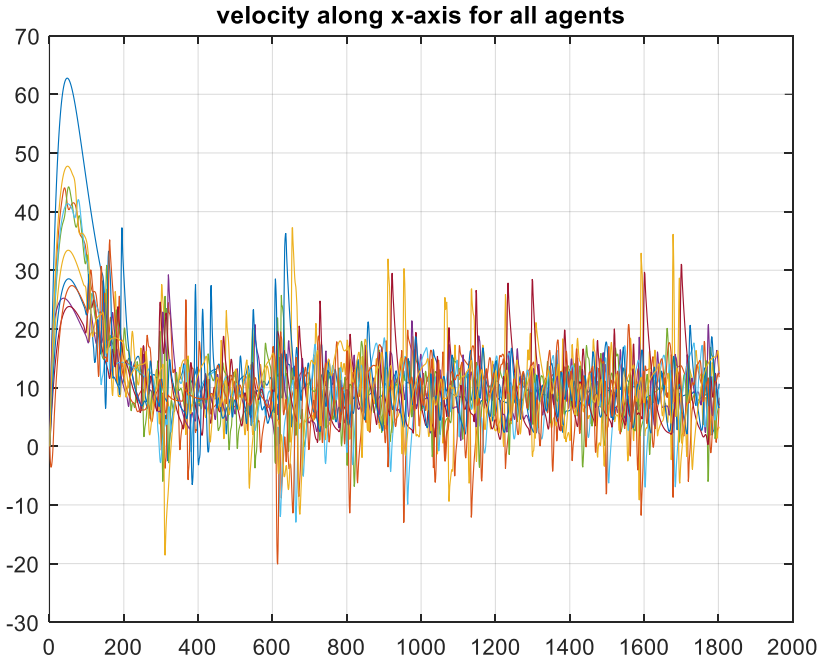


Dynamic Virtual Leader

We can modify the simulation scripts to make the virtual leader moves laterally too. In such an example with no obstacles on the road, the following results are obtained:




velocity along y-axis for all agents
30
20
10
0
-10
-20
-30
-40
-50
0
200
400
600
800
1000
1200
1400
1600
1800
2000


velocity along x-axis for all agents
70
60
50
40
30
20
10
0
-10
-20
-30
0
200
400
600
800
1000
1200
1400
1600
1800
2000

Appendix

MATLAB Scripts are as follows:

CONSTANTS.m

```
rng(45);

tInit = 0;
timeStep = 0.01;
tFinal = 15;


% MAS vars
N = 10;  % Agents Nums

% Setting the Initial Conditions for Leaders
leader_q_x = [50];
leader_q_y = [0];

leader_p_x = [10];
leader_p_y = [0];

leader_q_x_dot = [];
leader_q_y_dot = [];

leader_p_x_dot = [];
leader_p_y_dot = [];

% Setting up the Obstacles
obstacle_1 = [55;0];
obstacle_2 = [15;-10];
obstacle_3 = [75;0];
obstacle_4 = [75;5];
obstacle_5 = [44;0];
obstacle_6 = [46;0];
obstacle_7 = [48;0];
obstacle_8 = [50;0];
% obs = [obstacle_1, obstacle_2, obstacle_3, obstacle_4, obstacle_5, obstacle_6, obstacle_7];
obs = [obstacle_1, obstacle_2, obstacle_3, obstacle_4];
obs = [obstacle_1];
Rk1 = 10;
Rk2 = 8;
Rk3 = 4;
Rk4 = 1;
Rk = [Rk1;Rk2;Rk3;Rk4;Rk4;Rk4;Rk4;Rk4];

roadLeftLimit = 25;
roadRightLimit = -25;
roadWidth = roadLeftLimit - roadRightLimit;
roadLeftBoundary = [0; roadLeftLimit];
roadRightBoundary = [0; roadRightLimit];
bounds = [roadRightBoundary, roadLeftBoundary];

% Setting the Initial Conditions for Agents
q_dot_x = [];
q_dot_y = [];

p_dot_x = [];
p_dot_y = [];

q_x = [];
q_y = [];

p_x = [];
p_y = [];

q_x = [q_x, rand(N, 1)*(-75)];
q_y = [q_y, rand(N, 1)*roadWidth - 25];

p_x = [p_x, zeros(N, 1)];
p_y = [p_y, zeros(N, 1)];

ux = [];
uy = [];
```

```
% params------------------------
d = 6;
r = 1.2*d;
dPrime = 12;
rPrime = 1.2*dPrime;
dZegond = 0.5*d;
rZegond = 1.2*dZegond;
epsilon = 0.1;
a = 5;
b = 5;
h_a = 0.2;
h_b = 0.9;

c1_alpha = 50;
c1_gamma = [5; 2];
c1_beta = 100;
c1_boundary = 150;

c2_alpha = 2*sqrt(c1_alpha);
c2_gamma = 2*sqrt(c1_gamma);
c2_beta = 2*sqrt(c1_beta);
c2_boundary = 2*sqrt(c1_boundary);
```

Helper Functions

$\|z\|_\sigma$; sigma_normF.m

```
function sigma_norm = sigma_normF(z, epsilon)
    sigma_norm = (epsilon^(-1))*(sqrt(1 + epsilon*(euclidean_dist(z))^2) - 1);
end
```

$\sigma_1(z)$; sigma_1F.m

```
function sig_1 = sigma_1F(z)
    sig_1 = (z)/sqrt(1+z^2);
end
```

$\rho_h(z)$; rho_hF.m

```
function rho_h = rho_hF(z, h)
    if z>0 && z<h
        rho_h = 1;
    elseif z>h && z<1
        rho_h = 0.5*(1 + cos(pi*(z-h)/(1-h)));
    else
        rho_h = 0;
    end
end
```

$\hat{p}_{i,boundary}$; projectAgent_p_on_Boundary.m

```
function phat = projectAgent_p_on_Boundary(qi, pi, bound)
    phat = [pi(1); 0];
end
```

$\hat{p}_{i,k}$; projectAgent_p.m

```
function phat = projectAgent_p(qi, pi, obs_center, Rk)
    mu = Rk/euclidean_dist(qi-obs_center);
    a_k = (qi-obs_center)/euclidean_dist(qi-obs_center);
    P = eye(2) - a_k*transpose(a_k);
    phat = mu*P*pi;
end
```

$\widehat{q}_{i,boundary}$; projectAgent_on_Boundaries.m

```
function qhat = projectAgent_on_Boundaries(qi, boundary)
    qhat = [qi(1); boundary(2)];
end
```

$\widehat{q}_{i,k}$; projectAgent.m

```
function qhat = projectAgent(qi, obs_center, Rk)
    muu = Rk/(euclidean_dist(qi-obs_center));
%     qhat = muu*qi + (eye(2) - muu)*obs_center;
    qhat = muu*qi + (1 - muu)*obs_center;
end
```

$\phi(z)$; phiF.m

```
function phi = phiF(z,a,b)
    c = (a - b) / sqrt(4 * a * b);
    sig1 = sigma_1F(z+c);
    phi = 0.5*( (a+b)*sig1 + (a-b) );
end
```

$\phi_{\beta}(z)$; phi_betaF.m

```
function phi_beta = phi_betaF(z, r_a, d_b, h_b, a, b)
    rho_h = rho_hF(z/d_b, h_b);
    phi_beta = (rho_h)*(sigma_1F(z-d_b) - 1);
end
```

$\phi_{\alpha}(z)$; phi_alphaF.m

```
function phi_alpha = phi_alphaF(z, r_a, d_a, h_a, h_b, a, b)
    rho_h = rho_hF(z/r_a, h_a);
    phi = phiF(z-d_a, a, b);
    phi_alpha = (rho_h)*(phi);
end
```

n_{ij} ; nijF.m

```
function nij = nijF(qi, qj, epsilon)
    dist = euclidean_dist(qj-qi);
    nij = (qj-qi)/(sqrt(1 + epsilon*(dist^2)));
end
```

N_i^{β} ; formNeighborhoodObstaclesSets.m

```
function neighborsObstaclesSet = formNeighborhoodObstaclesSets(q_x, q_y, rPrime, obs, Rk)
    N = size(q_x, 1);
    neighborsObstaclesSet  = zeros(N,size(obs,2));

    for index=1:N
        x_i = q_x(index, end);
        y_i = q_y(index, end);

        qi = [x_i;y_i];

        for indey=1:size(obs,2)
            qhat = projectAgent(qi, obs(:,indey), Rk(indey));

            dist = euclidean_dist(qhat - qi);
%             fprintf(">>> Distance from Vehicle %d to Obstacle %d = %f \n", index, indey, dist)

            if dist < rPrime
                neighborsObstaclesSet(index, indey) = 1;
```

```
            end
        end
    end
end
```

$N_i^{boundary}$; formNeighborhoodBoundariesSets.m

```
function neighborsBoundariesSet = formNeighborhoodBoundariesSets(q_x, q_y, rZegond, bounds)
    N = size(q_x, 1);
    neighborsBoundariesSet  = zeros(N,size(bounds,2));

    for index=1:N
        x_i = q_x(index, end);
        y_i = q_y(index, end);

        qi = [x_i;y_i];

        for indey=1:size(bounds,2)
            qhat = projectAgent_on_Boundaries(qi, bounds(:, indey));

            dist = euclidean_dist(qhat - qi);
%             fprintf(">>> Distance from Vehicle %d to Obstacle %d = %f \n", index, indey, dist)

            if dist < rZegond
                neighborsBoundariesSet(index, indey) = 1;
            end
        end
    end
end
```

N_i^{α} ; formNeighborhoodAgntsSets.m

```
function neighborsAgentsSet = formNeighborhoodAgentsSets(q_x, q_y, r)
    N = size(q_x, 1);
    neighborsAgentsSet  = zeros(N,N);
    for index=1:N
        x_i = q_x(index, end);
        y_i = q_y(index, end);
        for indey=1:N
            if indey == index
                continue
            end
            x_j = q_x(indey, end);
            y_j = q_y(indey, end);
            dist = euclidean_dist([x_j;y_j]-[x_i;y_i]);
            if dist < r
                neighborsAgentsSet(index, indey) = 1;
            end
        end
    end
end
```

$a_{ij}(q)$; formAdjacency.m

```
function output = formAdjacency(qi, qj, r_a, h_a, epsilon)
    z = sigma_normF(qj-qi, epsilon);
    z = z/r_a;
    output = rho_hF(z, h_a);
end
```

$\|z\|$; euclidean_dist.m

```
function dist = euclidean_dist(z)
    dist = sqrt( sum( z.^2 ) );
%     dist = sqrt((q1(1) - q2(1))^2 + (q1(2) - q2(2))^2);
end
```

main.m

```
clc; clear;

% Generate constant values
run("CONSTANTS.m");

% ================================================================================
% ================================================================================
% Initialize the variable in the base workspace
myVariable = 0;
gammas = [];
% Create a figure window
fig = uifigure('Position', [100, 100, 300, 200], 'Name', 'Interactive Variable Update');

% Create a label to display the current value of the variable
lbl = uilabel(fig, 'Position', [100, 150, 100, 22], 'Text', ['Value: ', num2str(myVariable)]);

% Create a numeric edit field to input new values
editField = uieditfield(fig, 'numeric', 'Position', [100, 100, 100, 22]);

% Create a button to update the variable
btn = uibutton(fig, 'push', 'Position', [100, 50, 100, 22], 'Text', 'Update Value', ...
    'ButtonPushedFcn', @(btn, event) updateValue(editField, lbl));
% ================================================================================
% ================================================================================

temp_1_x = zeros(N,1);
temp_1_y = zeros(N,1);
temp_2_x = zeros(N,1);
temp_2_y = zeros(N,1);

% open-up simulation plot figure
figure(1);
hold on;

% graphic objects
graphics_edges = gobjects(N,N);
graphics_leader = gobjects(1,1);
graphics_obstacles = gobjects(size(obs, 2), 1);
graphics_agents = gobjects(N,1);

% plotting obstacles
theta = linspace(0, 2*pi, 100);
for i=1:size(obs,2)
    rs = Rk(i):-0.01:0;
    temp_x = [];
    temp_y = [];
    for j=1:length(rs)
        temp_x = [temp_x, obs(1,i) + rs(j) * cos(theta)];
        temp_y = [temp_y, obs(2,i) + rs(j) * sin(theta)];
    end
    graphics_obstacles(i) = plot(temp_x, temp_y, LineWidth=2, Color="red");
end

% alpha-Agents neighborhood links
for i = 1:N
    for j=1:N
        graphics_edges(i,j) = plot([0, 0], [0, 0], LineWidth=1, Color="blue");
    end
end

% gamma-Agent position
graphics_leader(end) = scatter(leader_q_x(1, end), leader_q_y(1, end),  'O', 'filled', 'green',
'DisplayName', sprintf("leader"));

% alpha-Agents positions
for i = 1:N
```

```
    graphics_agents(i) = scatter(q_x(i, end), q_y(i, end),  '>', 'black', 'filled', 'DisplayName',
sprintf("veh_%d", i));
end

% road boundaries
plot([0, 150], [roadRightLimit, roadRightLimit], Color="Red", LineWidth=1.5)
plot([0, 150], [roadLeftLimit, roadLeftLimit], Color="Red", LineWidth=1.5)

% final simulation settings
animationPauseTime = 0.01;
xlabel('x');
ylabel('y');
ylim([roadRightLimit, roadLeftLimit]);
xlim([-30,150]);
title('Animating Agent Movements');
% legend('show');
grid on;
hold off;


for tInit=0:timeStep:tFinal
    neighborsAgentsSet = formNeighborhoodAgentsSets(q_x, q_y, r);
    neighborsObstaclesSet = formNeighborhoodObstaclesSets(q_x, q_y, rPrime, obs, Rk);
    neighborsBoundariesSet = formNeighborhoodBoundariesSets(q_x, q_y, rZegond, bounds);

    for i=1:N
        qi = [q_x(i,end); q_y(i,end)];
        pi = [p_x(i,end); p_y(i,end)];
        qr = [leader_q_x(1,end); leader_q_y(1,end)];
        pr = [leader_p_x(1,end); leader_p_y(1,end)];

        phia_nij = 0;
        phib_nij = 0;
        phic_nij = 0;
        u_i_a_term01 = 0; % alpha-agents control term
        u_i_a_term02 = 0; % alpha-agents control term
        u_i_b_term01 = 0; % beta-agents control term
        u_i_b_term02 = 0; % beta-agents control term
        u_i_c_term01 = 0; % boundary control term
        u_i_c_term02 = 0; % boundary control term

        % alpha-agents collision avoidance
        for j=1:N
            if neighborsAgentsSet(i, j) == 1
                % u_i_a -----------------------------------------
                qj = [q_x(j,end); q_y(j,end)];
                pj = [p_x(j,end); p_y(j,end)];

                z = qj - qi;
                z = sigma_normF(z, epsilon);
                phia = phi_alphaF(z, r, d, h_a, h_b, a, b);
                nij = nijF(qi, qj, epsilon);
                phia_nij = phia_nij + phia*nij;

                adjacent = formAdjacency(qi, qj, r, h_a, epsilon);

                u_i_a_term01 = u_i_a_term01 + phia_nij;
                u_i_a_term02 = u_i_a_term02 + adjacent*(pj-pi);

                % -----------------------------------------------
            end
        end
        ui_a = u_i_a_term01.*c1_alpha + u_i_a_term02.*c2_alpha;

        % beta-agents
        for j=1:size(obs, 2)
            if neighborsObstaclesSet(i, j) == 1
                % u_i_b -----------------------------------------
                qhat = projectAgent(qi, obs(:,j), Rk(j));
                z = qhat - qi;
```

```
                z = sigma_normF(z, epsilon);
                phi_beta = phi_betaF(z, rPrime, dPrime, h_b, a, b);

                nij_hat = nijF(qi, qhat, epsilon);

                phib_nij = phib_nij + phi_beta*nij_hat;

                adjacent = formAdjacency(qi, qhat, dPrime, h_b, epsilon);

                phat = projectAgent_p(qi, pi, obs(:,j), Rk(j));

                u_i_b_term01 = u_i_b_term01 + phib_nij;
                u_i_b_term02 = u_i_b_term02 + adjacent*(phat-pi);

                % ------------------------------------------------
            end
        end
        ui_b = u_i_b_term01.*c1_beta + u_i_b_term02.*c2_beta;

        % following virtual leader (gamma-agent) control signal
        ui_y = -(c1_gamma.*(tanh(ui_b)+1)).*(qi-qr) - (c2_gamma.*(tanh(ui_b)+1)).*(pi-pr);
%         ui_y = -c1_gamma.*sigma_1F(qi-qr) - c2_gamma.*(pi-pr);
        gammas = [gammas , ui_y];


        % boundary-agents
        for j=1:size(bounds, 2)
            if neighborsBoundariesSet(i, j) == 1
                qhat = projectAgent_on_Boundaries(qi, bounds(:,j));
                z = qhat - qi;
                z = sigma_normF(z, epsilon);
                phi_beta = phi_betaF(z, rZegond, dZegond, h_b, a, b);

                nij_hat = nijF(qi, qhat, epsilon);

                phic_nij = phic_nij + phi_beta*nij_hat;

                adjacent = formAdjacency(qi, qhat, dZegond, h_b, epsilon);

                phat = projectAgent_p_on_Boundary(qi, pi, bounds(:,j));

                u_i_c_term01 = u_i_c_term01 + phic_nij;
                u_i_c_term02 = u_i_c_term02 + adjacent*(phat-pi);

                % ------------------------------------------------
            end
        end
        ui_c = u_i_c_term01.*c1_boundary + u_i_c_term02.*c2_boundary;

        ui = ui_a + ui_b + ui_y + ui_c;

        if i == 1
            ux(i,end+1) = ui(1);
            uy(i,end+1) = ui(2);
        else
            ux(i,end) = ui(1);
            uy(i,end) = ui(2);
        end

    end

    q_dot_x(:, end+1) = p_x(:,end);
    q_dot_y(:, end+1) = p_y(:,end);

    p_dot_x(:, end+1) = ux(:, end);
    p_dot_y(:, end+1) = uy(:, end);

    q_x(:, end+1) = q_x(:, end) + q_dot_x(:, end)*timeStep;
    q_y(:, end+1) = q_y(:, end) + q_dot_y(:, end)*timeStep;
```

```
    p_x(:, end+1) = p_x(:, end) + p_dot_x(:, end)*timeStep;
    p_y(:, end+1) = p_y(:, end) + p_dot_y(:, end)*timeStep;

    leader_q_x_dot(:, end+1) = leader_p_x(:,end);
    leader_q_y_dot(:, end+1) = leader_p_y(:,end);

    leader_p_x_dot(:, end+1) = leader_q_x(:,end)*0;
    leader_p_y_dot(:, end+1) = leader_q_x(:,end)*0;

    leader_q_x(:, end+1) = leader_q_x(:, end) + leader_q_x_dot(:, end)*timeStep;
    leader_q_y(:, end+1) = leader_q_y(:, end) + leader_q_y_dot(:, end)*timeStep;

    leader_p_x(:, end+1) = leader_p_x(:, end) + leader_p_x_dot(:, end)*timeStep;
    leader_p_y(:, end+1) = leader_p_y(:, end) + leader_p_y_dot(:, end)*timeStep;


%     cla;
%     hold on;
    for i = 1:N
        set(graphics_agents(i), 'XData', q_x(i, end), 'YData', q_y(i, end));
%         scatter(q_x(i, end), q_y(i, end), 'm>', 'red', 'DisplayName', sprintf("veh_%d", i));
    end

    for i = 1:N
        for j=1:N
            if neighborsAgentsSet(i,j) == 1
                temp_x = neighborsAgentsSet.*q_x(:,end);
                temp_y = neighborsAgentsSet.*q_y(:,end);
                set(graphics_edges(i,j), 'XData', [temp_x(i,j), temp_x(j,i)], 'YData', [temp_y(i,j),
temp_y(j,i)]);
%                   plot([q_x(i,end), q_x(j,end)], [q_y(i,end), q_y(j,end)], LineWidth=1.5,
Color="blue")
            else
                set(graphics_edges(i,j), 'XData', [0, 0], 'YData', [0, 0]);
            end

        end
    end

    set(graphics_leader(1), 'XData', leader_q_x(1, end), 'YData', leader_q_y(1, end));
    drawnow;
    pause(animationPauseTime)
end

% Callback function to update the variable
function updateValue(editField, lbl)
    global leader_p_y;
    % Get the new value from the edit field
    newValue = editField.Value;

    % Update the variable in the base workspace
    assignin('base', 'leader_p_y', [leader_p_y, newValue]);

    % Update the label text
    lbl.Text = ['Value: ', num2str(newValue)];

    % Display the updated variable in the command window
    disp(['Updated Variable: ', num2str(newValue)]);
end
```