49329 - Control of Mechatronic Systems

Student Name	Student Number	Contributions
Zhiye Zhao	14366494	25%
Peng Liu	13407527	25%
Kemu Lin	14450757	25%
Lirun Dai	13842101	25%

For this group project, the four group members worked together to make sure all the problems are solved correctly.

Project 1: Mobile robot formation control

When a team of robots want to keep the relative position among them (e.g. a triangle) during their motion, each robot needs to control its motion according to its relative position with other robots.

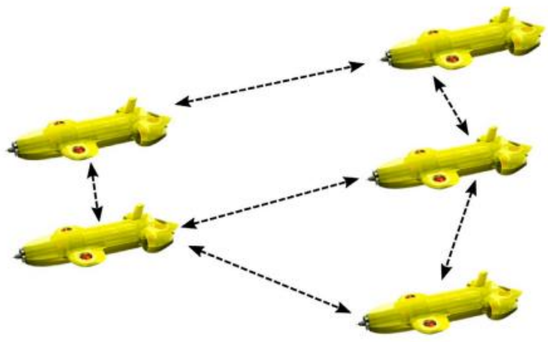
This project aims to achieve certain formation through feedback control. The final outcome will be a demonstration in MATLAB simulation to achieve the desired formation.

The group working on this project has the flexibility of defining the scenarios, the robot motion model (the model should contain some uncertainties), and the project goals.

Below are some youtube videos about robot formation:

https://www.youtube.com/watch?v=mR3cwn-r67A

https://www.youtube.com/watch?v=h4ypcDwTZpI



1. Introduction

In the field of robotics, the ability of a group of robots to maintain relative positions during movement is a challenging project. Mobile robot formation control is a fundamental concept in multi robot systems, where complex management of the relative positions of each robot is necessary to achieve predefined shapes, such as triangles and rectangles. This dynamic field combines robot technology, control theory, and computational algorithms to coordinate motion with Matlab simulation. This project begins to address the complex issue of mobile robot formation control, emphasizing the role of feedback control systems. The main goal is to develop a feedback control strategy that enables the robot team to achieve and maintain the required formation.

2. Methodology

We will use formation control and MPC (model predictive control) to solve this project. This approach will be based on ‘Bearing’, then maybe add another more efficient approach. Algorithm design and simulation are performed using MATLAB and its toolbox. Then, the set robot is controlled and tested using the code generated by MATLAB.

2.1 Path selection for random obstacles

The first part is the robot trajectory, which is the most significant part of this project. By inputting random obstacles, the usability of the control algorithm can be tested. For optimal path planning, the minimum energy consumption is used as a benchmark to achieve the robot array to automatically select the optimal path in the presence of random roadblocks.

2.1.1 MATLAB code: random_obs_map

Firstly, the code we create is used to generate a random obstacle map for simulation or control experiments. The main goal is to create a dynamic environment for testing the formation control of mobile robots.

```
function [obstacles, x_max, y_max, y_min] = random_obs_map()

    % Parameter initialisation
    x_max = 500; % Maximum width of the map
    y_max = 30; % Maximum height of the map
    y_min = -30; % Minimum height of the map
    obstacle_gap_range = [50, 50]; % Range of gaps between obstacles
    obstacle_length_range = [10, 15]; % obstacle length range
    obstacle_width_range = [10, 30];  % obstacle width range

    % Initialising an array of obstacles
    obstacles = [];

    x_current = 0; % current x-axis position


    while x_current < x_max
        % Generate starting x-axis of obstacle, add random gap
        x_start = x_current + obstacle_gap_range(1) + (obstacle_gap_range(2) - obstacle_gap_range(1)) * rand();

        % Out of bounds check
        if x_start > x_max
            break;
        end

        % Randomly generated lengths of obstacles
        obstacle_length = obstacle_length_range(1) + (obstacle_length_range(2) - obstacle_length_range(1)) * rand();

        % Double obstacle logic, triggered only when the obstacle length
        % is less than a specific value
```

The function begins by initializing various parameters for the obstacle generation. Including:

x_max: The maximum width of the map.

y_max: The maximum height of the map.
y_min: The minimum height of the map.
obstacle_gap_range: A range specifying the minimum and maximum gap between obstacles.
obstacle_length_range: A range specifying the minimum and maximum length of obstacles.
obstacle_width_range: A range specifying the minimum and maximum width of obstacles.
Initialize Obstacles Array: An empty array named obstacles is created to store information about the generated obstacles.

Obstacle Generation Loop: A while loop is used to generate obstacles along the x-axis until the x_current position reaches x_max. The loop continues until the map's full width is populated with obstacles

Generate Obstacle Starting Position: Inside the loop, a starting x-axis position (x_start) for the obstacle is generated. This position is determined by adding a random value within the specified range of obstacle_gap_range to the current x-axis position (x_current).

Out of Bounds Check: The code checks if the x_start value is beyond the maximum width of the map (x_max). If it is, the loop is terminated, ensuring that no obstacle is generated outside the map boundaries.

Generate Obstacle Length: A random length for the obstacle is generated within the specified range of obstacle_length_range.

```matlab
    if obstacle_length < 45/2
        double_obs = randi([1, 2]); % Randomly decide whether to generate a double obstacle

        if double_obs == 1

            direction = randi([0, 1]) * 2 - 1;
            y_start = direction * randi([0, y_max/2]);
            y_end = y_start + direction * obstacle_length;

            % Mirror the location of the obstacle
            y_start_mirror = -y_end;
            y_end_mirror = -y_start;

            % Restriction of obstacle locations within the map
            y_start = max(y_min, min(y_max, y_start));
            y_end = max(y_min, min(y_max, y_end));
            y_start_mirror = max(y_min, min(y_max, y_start_mirror));
            y_end_mirror = max(y_min, min(y_max, y_end_mirror));

            % Adding obstacles to an array (original and mirrored)
            obstacles = [obstacles; x_start, min(y_start, y_end), x_start + obstacle_length, max(y_start, y_end)];
            obstacles = [obstacles; x_start, min(y_start_mirror, y_end_mirror), x_start + obstacle_length, max(y_start_mirror, y_end_mirror)];

            x_current = x_start + obstacle_length;
            continue;
        end
    end

    % If no double obstacle is generated
    obstacle_width = randi(obstacle_width_range);
    direction = randi([0, 1]) * 2 - 1;
    y_base = direction * randi([0, y_max]);
    y_start = y_base;
    y_end = y_base + direction * obstacle_width;

    % Ensure that y_start is the lower value
    if y_start > y_end
        temp = y_start;
        y_start = y_end;
        y_end = temp;
    end
```

Double Obstacle Logic: If the generated obstacle's length is less than half of a certain value (45/2), a logic for creating double obstacles is applied. Double obstacles consist of two obstacles that are symmetrical along the y-axis.

Randomly Decide Double Obstacle: A random choice is made (1 or 2) to determine whether to generate a double obstacle.

Generate Double Obstacle: If a double obstacle is chosen, two identical obstacles are generated with their y-positions mirrored. The direction variable is used to determine the mirroring direction.

```

            % End of obstacle x-coordinate
            x_end = x_start + obstacle_length;

            % Add data
            obstacles = [obstacles; x_start, y_start, x_end, y_end];

            x_current = x_end;

        end
    end


```

Mirror Obstacle Locations: The y-positions of the double obstacles are mirrored, ensuring that they are within the map boundaries.

Add Obstacles to Array: The obstacle data for both the original and mirrored obstacles is added to the obstacles array.

Update Current Position: The x_current position is updated to the end of the generated obstacle to continue generating obstacles.

Single Obstacle Generation: If a double obstacle is not generated, a single obstacle is created. This obstacle has a randomly determined width (obstacle_width) and a starting y-position based on a random value within the specified height range.

Ensure Y-Start is Lower: The code ensures that the y_start position is lower than y_end for consistency.

Obstacle End Position: The end position of the obstacle along the x-axis (x_end) is determined.

Add Obstacle Data: The information about the single obstacle is added to the obstacles array, including its starting and ending positions along the x- and y-axes.

Update Current Position: The x_current position is updated to the end of the generated obstacle, and the loop continues to generate the next obstacle.

Once the loop completes, the function returns the obstacles array, as well as the x_max, y_max, and y_min values, which define the map dimensions and obstacle generation constraints.

2.1.2 MATLAB code: random_obs_choice

```
function [filtered_obstacles] = random_obs_choice(y_cart, obstacles, distance_over)
filtered_obstacles = [];
i = 1;
while i <= size(obstacles, 1)
    % Check if the current obstacle is mirrored
    if i < size(obstacles, 1) && obstacles(i, 1) == obstacles(i+1, 1)
        % Mirroring exists to determine which obstacle needs to be preserved
        if y_cart <= obstacles(i+1, 4) + distance_over ...
                && y_cart >= obstacles(i+1, 2) - distance_over
            filtered_obstacles = [filtered_obstacles; obstacles(i+1, :)];
        else
            filtered_obstacles = [filtered_obstacles; obstacles(i, :)];
        end
        % Skip the next item in the Mirror Barrier
        i = i + 1;
    else
        % No mirrors, just add
        filtered_obstacles = [filtered_obstacles; obstacles(i, :)];
    end
    % Move to the next element
    i = i + 1;
end
end
```

This function takes as input a Cartesian y-coordinate y_cart, a list of obstacles (obstacles), and a distance value (distance_over). Its purpose is to filter and choose specific obstacles from the list based on their relationship with the given y-coordinate and distance constraints. The selected obstacles are stored in the filtered_obstacles array, which is returned by the function. The purpose of this function is to filter and select obstacles that are relevant to a given vertical position (y_cart) within the context of a simulation or control problem.

The function takes three inputs including y_cart, obstacles, distance_over.

y_cart: The Cartesian y-coordinate (vertical position) used to filter obstacles.

obstacles: A matrix representing the obstacles, where each row corresponds to an obstacle and contains four values: [x_start, y_start, x_end, y_end].

distance_over: A distance value that defines a buffer zone around the y_cart coordinate, within which obstacles are considered relevant.

Initialize filtered_obstacles Array: An empty array named filtered_obstacles is initialized to store the filtered obstacle data.

While Loop: The function uses a while loop to iterate through the rows of the obstacles matrix.

Check for Mirrored Obstacles: Inside the loop, the code checks if the current obstacle is mirrored by comparing the x position of the current obstacle (obstacles(i, 1)) with the x position of the next obstacle (obstacles(i+1, 1)). If they have the same x position, it indicates that the obstacles are mirrored along the vertical axis.

Initialize filtered_obstacles Array: An empty array named filtered_obstacles is initialized to store the filtered obstacle data.

While Loop: The function uses a while loop to iterate through the rows of the obstacles matrix.

Check for Mirrored Obstacles: Inside the loop, the code checks if the current obstacle is mirrored by comparing the x position of the current obstacle (obstacles(i, 1)) with the x position of the next obstacle (obstacles(i+1, 1)). If they have the same x position, it indicates that the obstacles are mirrored along the vertical axis.

Mirrored Obstacle Handling: If the current obstacle is mirrored (i.e., i is less than the last row and the x positions match), the code evaluates whether the Cartesian y coordinate (y_cart) falls within a specific range that includes the obstacle's mirrored partner. The range is defined by the current obstacle's upper and lower bounds (plus/minus the distance_over value). If the y_cart is within this range, the mirrored obstacle is selected and added to the filtered_obstacles array. Otherwise, the original obstacle is added to the array.

Skip the Next Item in the Mirror Barrier: After handling mirrored obstacles, the code skips the next item (i.e., the mirrored partner) by incrementing i by 1 to avoid double-counting mirrored obstacles.

Non-Mirrored Obstacle Handling: If the current obstacle is not mirrored, it is simply added to the filtered_obstacles array.

Move to the Next Element: In both mirrored and non-mirrored cases, i is incremented by 1 to move to the next obstacle in the obstacles matrix.

Loop Continuation: The loop continues to iterate through the obstacles in the obstacles matrix until all obstacles have been processed.

Return Filtered Obstacles: Finally, the function returns the filtered_obstacles array, which contains only the selected obstacles that meet the criteria based on the y_cart coordinate and the specified distance constraints.

The purpose of this function is to filter and select obstacles that are relevant to a given vertical position (y_cart) within the context of a simulation or control problem.

2.1.3 MATLAB code: traj_obs_pos_act

```
function [x_start, x_end] = traj_obs_pos_act(y_cart, obstacles, distance_over)

obs_min = obstacles(1,2) - distance_over;
obs_max = obstacles(1,4) + distance_over;

del_y_min = abs(y_cart - obs_min);
del_y_max = abs(y_cart - obs_max);

% go up
if del_y_max <= del_y_min

    y_move = del_y_max;
    x_buffer = 2*y_move;

    x_start = obstacles(1,1) - x_buffer;
    x_end = obstacles(1,3) + x_buffer;

% go down
elseif del_y_max > del_y_min

    y_move = del_y_min;
    x_buffer = 2*y_move;

    x_start = obstacles(1,1) - x_buffer;
    x_end = obstacles(1,3) + x_buffer;
end

end
```

This function calculates and returns the starting and ending x-coordinates of a trajectory that avoids an obstacle, given a specified Cartesian y-coordinate (y_cart), a list of obstacles (obstacles), and a distance value (distance_over). The function determines whether the trajectory should go above or below the obstacle to safely avoid it and calculates the corresponding x-positions for avoidance.

The function also takes three inputs:

y_cart: The Cartesian y-coordinate (vertical position) used as a reference point for obstacle avoidance.

obstacles: A matrix representing the obstacles, where each row corresponds to an obstacle and contains four values: [x_start, y_start, x_end, y_end].

distance_over: A distance value that defines a buffer zone around the obstacle, which the trajectory must avoid.

Calculate Vertical Range: The code calculates the vertical range in which the obstacle is located, considering the distance_over. It calculates obs_min as the lower bound of the obstacle's vertical range (y_start - distance_over) and obs_max as the upper bound (y_end + distance_over).

Calculate Vertical Displacement: The code calculates the absolute vertical displacement (del_y_min and del_y_max) between the specified y_cart coordinate and the lower and upper bounds of the obstacle's vertical range, respectively.

Determine Trajectory Direction: The code then compares del_y_min and del_y_max to determine which way the trajectory should go to avoid the obstacle. There are two possible scenarios:

If del_y_max is less than or equal to del_y_min, it means that the trajectory should go above the obstacle to avoid it.
If del_y_max is greater than del_y_min, it means that the trajectory should go below the obstacle to avoid it.

Calculate x-buffer: A buffer distance (x_buffer) is calculated based on the y_move, which is the vertical displacement the trajectory must achieve to safely avoid the obstacle. This buffer ensures that the trajectory avoids the obstacle with a margin.

Determine x_start and x_end: Depending on whether the trajectory should go up or down, the x_start and x_end coordinates for the trajectory are calculated. These positions provide a safe corridor around the obstacle. The x_start is set to the left boundary of the obstacle minus the x_buffer, and the x_end is set to the right boundary of the obstacle plus the x_buffer.

Return x_start and x_end: The function returns the calculated x_start and x_end values, representing the starting and ending x-coordinates for the trajectory that avoids the obstacle while maintaining a buffer zone around it.

2.1.4 MATLAB code: traj_obs_pass.m

```
function [x_pos, y_pos] = traj_obs_pass(y_cart, v_cart, obstacles, distance_over, step)

obs_min = obstacles(1,2) - distance_over;
obs_max = obstacles(1,4) + distance_over;

del_down = abs(y_cart - obs_min);
del_up = abs(y_cart - obs_max);

width_obs = abs(obstacles(1,1) - obstacles(1,3));
local_x = obstacles(1,1);
local_x_2 = obstacles(1,3);

v_x = v_cart; % Set cart speed

if del_up <= del_down

    % Desired path geometry data
    y_move = del_up; % Distance the y-axis needs to be moved
    y_move_part = y_move / 2; % This process has two parts
    del_x_cto = 2*y_move; % The distance of cart starts avoiding obstacles set to be 2 times the y-axis travelling distance

    distance_x_start = local_x - del_x_cto;
    distance_x_end = local_x_2 + del_x_cto;

    % Coordinate
    y_peak = y_cart + y_move;
    x_act_1 = local_x - del_x_cto;


    % Accelerated motion data
    t_full = del_x_cto / v_x; % Total running time for obstacle avoidance
    t_part = t_full / 2; % This process has up and down, two parts
    a_y = ((y_move/2) * 2) / (t_part^2); % Acceleration in the y-axis, same acceleration and deceleration


    % Key position in the acceleration phase
    v_y_max = a_y * t_part; % Maximum speed
    x_move_part = del_x_cto/2; % x-axis displacement to reach maximum velocity

    % Coordinate
    pos_y_vy_max = y_cart + y_move_part; % Maximum speed y position
    pos_x_vy_max = local_x - del_x_cto/2; % Maximum speed x position


```

y_cart: This parameter represents the Cartesian y-coordinate of the object, indicating the vertical position concerning which obstacle avoidance is calculated.

v_cart: It represents the velocity of the object that needs to pass the obstacle. obstacles: This is a matrix containing information about the obstacle. The first row of this matrix is used to define the boundaries of the obstacle, including [x_start, y_start, x_end, y_end].

distance_over: A distance value used to define a buffer zone around the obstacle that the trajectory must avoid.

step: This parameter defines the step size for discretizing the trajectory.

Vertical Range Calculation:

It computes the minimum (obs_min) and maximum (obs_max) vertical boundaries of the obstacle, considering the buffer zone defined by distance_over.

It calculates the vertical displacements above the obstacle (del_up) and below the obstacle (del_down) based on the specified y-coordinate.

```
% Obstacle avoidance path setting
    distance = distance_x_start:step:distance_x_end;

    x_pos = zeros;
    y_pos = zeros;

    for i = 1:length(distance)

        % Not before response position
        if distance(i) <= local_x - del_x_cto

            x_pos(i) = distance(i);
            y_pos(i) = y_cart;

        % Accelerated phase of response initiated
        elseif distance(i) <= local_x - del_x_cto/2

            x_pos(i) = distance(i);
            del_t_in = (x_pos(i) - x_act_1) / v_x;
            y_pos(i) = y_cart + 0.5 * a_y * (del_t_in) ^ 2;

        % Deceleration phase of response
        elseif distance(i) <= local_x

            x_pos(i) = distance(i);
            del_t_de = (x_pos(i) - pos_x_vy_max) / v_x;
            y_pos(i) = pos_y_vy_max ...
                    + v_y_max * del_t_de - 0.5 * a_y * (del_t_de^2);

        % Straight through phase
        elseif distance(i) <= local_x + width_obs

            x_pos(i) = distance(i);
            y_pos(i) = y_peak;

        % Return to initial acceleration phase
        elseif distance(i) <= local_x + width_obs + del_x_cto/2

            x_pos(i) = distance(i);
            del_t_in = (x_pos(i) - local_x_2) / v_x;
            y_pos(i) = y_peak - 0.5 * a_y * (del_t_in) ^ 2;

        % Return to initial deceleration phase
```

```
            % Return to initial deceleration phase
            elseif distance(i) <= distance_x_end

                x_pos(i) = distance(i);
                del_t_de = (x_pos(i) - local_x_2 - x_move_part) / v_x;
                y_pos(i) = pos_y_vy_max ...
                         - v_y_max * del_t_de + 0.5 * a_y * (del_t_de^2);

            end
        end

    elseif del_up > del_down

        % Desired path geometry data
        y_move = del_down; % Distance the y-axis needs to be moved
        y_move_part = y_move / 2; % This process has two parts
        del_x_cto = 2*y_move;

        distance_x_start = local_x - del_x_cto;
        distance_x_end = local_x_2 + del_x_cto;

        % Coordinate
        y_peak = y_cart - y_move;
        x_act_1 = local_x - del_x_cto;


        % Accelerated motion data
        t_full = del_x_cto / v_x;
        t_part = t_full / 2;
        a_y = ((y_move/2) * 2) / (t_part^2);


        % Key position in the acceleration phase
        v_y_max = a_y * t_part;
        x_move_part = del_x_cto/2;

        % Coordinate
        pos_y_vy_max = y_cart - y_move_part;
        pos_x_vy_max = local_x - del_x_cto/2;

```

Obstacle Geometry Data:

The code computes the width of the obstacle (width_obs) by finding the absolute difference between the x-coordinates of the obstacle's boundaries.
It sets local x-coordinates (local_x and local_x_2) to define the positions of the obstacle.

Desired Path Geometry Data:

It calculates the distance the object needs to move vertically (y_move), and half of this distance (y_move_part) is used for specific calculations.

It defines the x-coordinates for starting (distance_x_start) and ending (distance_x_end) the trajectory around the obstacle.

The y-coordinate at the peak of the trajectory (y_peak) is computed based on whether the trajectory passes above or below the obstacle.

Accelerated Motion Data:

The function calculates the total time for the object to pass through the buffer zone (t_full) and the time for each part of the trajectory (t_part).

It computes the acceleration in the y-direction (a_y) needed to achieve the desired y-movement.

The maximum vertical speed (v_y_max) that the object can reach during the trajectory is determined.

```
    distance = distance_x_start:step:distance_x_end;

    x_pos = zeros;
    y_pos = zeros;

    for i = 1:length(distance)

        if distance(i) <= local_x - del_x_cto

            x_pos(i) = distance(i);
            y_pos(i) = y_cart;

        elseif distance(i) <= local_x - del_x_cto/2

            x_pos(i) = distance(i);
            del_t_in = (x_pos(i) - x_act_1) / v_x;
            y_pos(i) = y_cart - (0.5 * a_y * (del_t_in) ^ 2);

        elseif distance(i) <= local_x

            x_pos(i) = distance(i);
            del_t_de = (x_pos(i) - pos_x_vy_max) / v_x;
            y_pos(i) = pos_y_vy_max ...
                    - v_y_max * del_t_de + 0.5 * a_y * (del_t_de^2);

        elseif distance(i) <= local_x + width_obs

            x_pos(i) = distance(i);
            y_pos(i) = y_peak;

        elseif distance(i) <= local_x + width_obs + del_x_cto/2

            x_pos(i) = distance(i);
            del_t_in = (x_pos(i) - local_x_2) / v_x;
            y_pos(i) = y_peak + 0.5 * a_y * (del_t_in) ^ 2;

        elseif distance(i) <= distance_x_end

            x_pos(i) = distance(i);
            del_t_de = (x_pos(i) - local_x_2 - x_move_part) / v_x;
            y_pos(i) = pos_y_vy_max ...
                    + v_y_max * del_t_de - 0.5 * a_y * (del_t_de^2);

        end
```

Trajectory Calculation:

The code iterates over a range of x-coordinates (distance) defined by step.
It calculates the x and y coordinates for each x-position based on the defined trajectory phases, including the initial, accelerated, decelerated, straight-through, and return phases.

Final Output:

The function returns two arrays, x_pos and y_pos, which contain the computed x and y positions of the trajectory, respectively.

The output arrays x_pos and y_pos can be used to control the robot's movements during obstacle avoidance. The code accommodates both above and below obstacle pass scenarios and ensures that the trajectory adheres to specified acceleration and deceleration patterns for safe passage.

2.2.5 MATLAB code: traj_create.m

This MATLAB code is used to create a trajectory for an object, such as a mobile robot, to navigate through a space with obstacles. It takes into account various parameters, including the positions and shapes of obstacles, the desired position of the object, its velocity.

```
function [traj_matrix] = traj_create(obstacles, y_cart, v_cart, distance_over, step, x_max)

% Remove non-complaint barrier blocks
[filtered_obstacles] = random_obs_choice(y_cart, obstacles, distance_over);

% Initially state
x_full = x_max + 50;

x_pos = [0];
y_pos = [y_cart];

% The end position of the previous obstacle, initially set as the starting point
last_obstacle_end_x = 0;
obstacles_single_x = 0;

% Create trajectory
for i = 1:size(filtered_obstacles,1)

    obstacles_single = filtered_obstacles(i,:);

    if y_cart <= obstacles_single(1,4) + distance_over ...
    && y_cart >= obstacles_single(1,2) - distance_over % Need for obstacle avoidance

        % Get the start and end points of the obstacle avoidance process
        [x_start, x_end] = traj_obs_pos_act(y_cart, obstacles_single, distance_over);

        % Draw the path from the previous end point to the current start point
        x_between = last_obstacle_end_x:step:x_start;
        y_between = repmat(y_cart, size(x_between));

        x_pos = [x_pos, x_between];
        y_pos = [y_pos, y_between];

        % Adding a path for obstacle avoidance
        [x_pos_new, y_pos_new] = traj_obs_pass(y_cart, v_cart, ...
                                               obstacles_single, ...
                                               distance_over, step);

        x_pos = [x_pos, x_pos_new];
        y_pos = [y_pos, y_pos_new];

        last_obstacle_end_x = x_end;

        obstacles_single_x = obstacles_single(1,1);
```

```
    else % No need for obstacle avoidance

        % Draw the path from the end of the actionto the end of the current obstacle
        x_between = last_obstacle_end_x:step:obstacles_single(1,3);
        y_between = repmat(y_cart, size(x_between));

        x_pos = [x_pos, x_between];
        y_pos = [y_pos, y_between];

        % Reset the next starting position
        last_obstacle_end_x = obstacles_single(1,3);
        obstacles_single_x = obstacles_single(1,1);

    end

end

if last_obstacle_end_x < x_full

    x_end = last_obstacle_end_x:step:x_full;
    y_end = repmat(y_cart, size(x_end));

    x_pos = [x_pos, x_end];
    y_pos = [y_pos, y_end];

end

%% Return array with x position, y position and angular velocity

dt = step/v_cart;
num_points = length(x_pos);
theta = zeros(1, num_points);
omega = zeros(1, num_points);

% Calculate the angle (theta) for each point
for i = 1:(num_points - 1) % Subtract 1 to avoid going out of index range

    delta_x = x_pos(i+1) - x_pos(i);
    delta_y = y_pos(i+1) - y_pos(i);
    theta(i) = atan2(delta_y, delta_x);

end
```

Function Input:

obstacles: A matrix representing a list of obstacles, where each row contains four values - [x_start, y_start, x_end, y_end], defining the boundaries of the obstacles.

y_cart: The Cartesian y-coordinate of the object, representing the vertical position where the trajectory is calculated.

v_cart: The velocity of the object.

distance_over: A distance value used to specify a buffer zone around obstacles.
step: The step size used for discretizing the trajectory.

x_max: The maximum x-coordinate in the space.

Computation

Remove Non-Compliant Barrier Blocks:

The code calls the random_obs_choice function to filter the obstacles that need to be avoided based on their positions and the buffer zone. The result is stored in filtered_obstacles.

Initialization:

The initial states for x_pos and y_pos are set. Initially, only one point is added with the provided y_cart value.

x_full is set to a value greater than x_max, representing the end of the trajectory.

Create Trajectory:

The code iterates through the filtered obstacles.

For each obstacle:

It checks if the object needs to avoid the obstacle based on its vertical position (y_cart) and the buffer zone.

If avoidance is needed, it calls the traj_obs_pos_act function to determine the start and end positions for the obstacle avoidance.

It generates a path from the last obstacle's end point to the current obstacle's start point.

It calls the traj_obs_pass function to calculate the trajectory for obstacle avoidance and adds it to x_pos and y_pos.

The end position of the current obstacle is stored as last_obstacle_end_x.

The starting x-coordinate of the current obstacle is stored as obstacles_single_x.

If avoidance is not needed for an obstacle, a path is generated from the last obstacle's end point to the end of the current obstacle.

last_obstacle_end_x and obstacles_single_x are updated accordingly.

Handling the End Position:

If the last obstacle avoidance ends before reaching x_full, a path is generated from the last obstacle's end point to x_full.
These end-position paths are added to x_pos and y_pos.

```
% Calculate the angle (theta) for each point
for i = 1:(num_points - 1) % Subtract 1 to avoid going out of index range

    delta_x = x_pos(i+1) - x_pos(i);
    delta_y = y_pos(i+1) - y_pos(i);
    theta(i) = atan2(delta_y, delta_x);

end

% The theta of the last point can be set to be the same as the previous one
theta(num_points) = theta(num_points-1);

% Calculate angular velocity
for i = 1:(num_points - 1)
    omega(i) = (theta(i+1) - theta(i)) / dt;
end

% For the last point
omega(num_points) = omega(num_points-1);

% Merge all the data into a matrix
traj_matrix = [x_pos; y_pos; omega];

end
```

Calculating Angular Velocity:

The function calculates the angular velocity (omega) for each point in the trajectory.
It first calculates the angle (theta) at each point using the atan2 function.
Then, it computes the angular velocity by taking the difference in angles and dividing by the time step dt.

Returning the Trajectory Matrix:

Finally, all the data, including x_pos, y_pos, and omega, is merged into a matrix called traj_matrix, which is returned as the function's output.

This code is used for path planning and obstacle avoidance for mobile robots or objects moving in a defined space. It allows for the creation of trajectories that safely navigate around obstacles while considering the object's velocity and other parameters. The resulting trajectory matrix can be used to control the object's movements in a real-world or simulation environment.

\

2.2.6 MATLAB code: final.m

This MATLAB code is to simulate and visualize the trajectory and motion of multiple agents, including leaders and followers, in the context of a formation control problem with obstacles. The script uses optimization techniques to compute control inputs for the followers to maintain formation while avoiding obstacles.

It starts by initial cart positions (y_cart_1 and y_cart_2), cart speed (v_cart), obstacle avoidance distance , and the time step.

Generating Obstacles: The random_obs_map function is called to generate a random map of obstacles (obstacles) and defines the maximum and minimum boundaries for the simulation.

Creating Trajectories: Two trajectory matrices (traj_matrix_1 and traj_matrix_2) are created using the traj_create function. These matrices contain position, velocity, and angular velocity information for the leader and follower carts as they navigate the environment while avoiding obstacles.

Handling Different Trajectory Lengths: The code ensures that the trajectories for the two carts are of the same length. If the trajectories are not equal, it pads the shorter one with additional values to match the length of the longer trajectory.

Defining Parameters: Parameters such as the leader's speed (v_L) and formation control parameters (lambda_LF_d and phi_LF_d) are defined.

State Initialization: Initial positions of leaders and followers are defined, including leader_pos_1, leader_pos_2, leader_pos_3, and leader_pos_4. These positions include the x and y coordinates and the angular position (orientation).

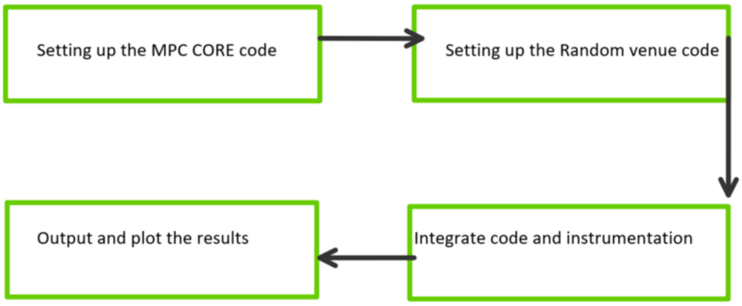
Simulation Loop 1-4: The code contains four separate simulation loops (labeled as 111, 222, 333, and 444) for each leader-follower pair.

Optimization: Within each loop, the script uses an optimization method (fmincon) to calculate optimal control inputs for the followers, specifically linear and angular velocity (v_F and omega_F) that help them follow their respective leaders while avoiding obstacles.

Updating Positions: The positions of followers (follower_pos_X) are updated based on the computed control inputs, while the leaders' positions are updated based on the current positions in the trajectory matrices.

Visualisation

Visualisation is also an essential component in this control model project. Therefore, the project team developed the following project process map.



There are two main factors that make the project successful. The first is the accuracy and logic of the MPC code. When the MPC code can face multiple complex and random paths, the project meets the basic requirements to continue moving forward. The second important part is the randomly generated setup code for the venue. Only by generating obstacles completely randomly under limited conditions can the accuracy of the MPC code be tested to meet the solution of the optimal path with minor energy consumption.

Therefore, the first step in project visualisation is the writing of MPC core code.

MPC Core Function

In this project, we set up two virtual robot identities: the leader and the pursuer. According to the theoretical setting, the relative ideal positions of the leader and the follower are fixed values, and Matlab calculates the relative positions. Matlab demonstrates the specific code in the figure below.

```
function cost = objectiveFunc_dis(u, follower_pos, leader_pos, v_L, omega_L, lambda_LF_d, phi_LF_d, Q, R, dt, N)
    v_F = u(1);
    omega_F = u(2);
    cost = 0;
    for i = 1:N
        % 计算follower在leader相对位置下的理想位置

        ideal_follower_pos = [leader_pos(1) + lambda_LF_d * cos(leader_pos(3) + phi_LF_d);

                              leader_pos(2) + lambda_LF_d * sin(leader_pos(3) + phi_LF_d);

                              leader_pos(3)];
        % 计算位置误差
        e = ideal_follower_pos - follower_pos;
        e_dot = [v_L*cos(e(3)) - v_F;
                 v_L*sin(e(3)) - v_F;
                 omega_L - omega_F];
        follower_pos = follower_pos + [v_F*cos(follower_pos(3)); v_F*sin(follower_pos(3)); omega_F] * dt;
        leader_pos = leader_pos + [v_L*cos(leader_pos(3)); v_L*sin(leader_pos(3)); omega_L] * dt;
        cost = cost + e' * Q * e + u' * R * u;
    end
end
```

To prevent errors in the robot sequence or individual robot positions, such as a collision between the leader and the follower when encountering an obstacle, the

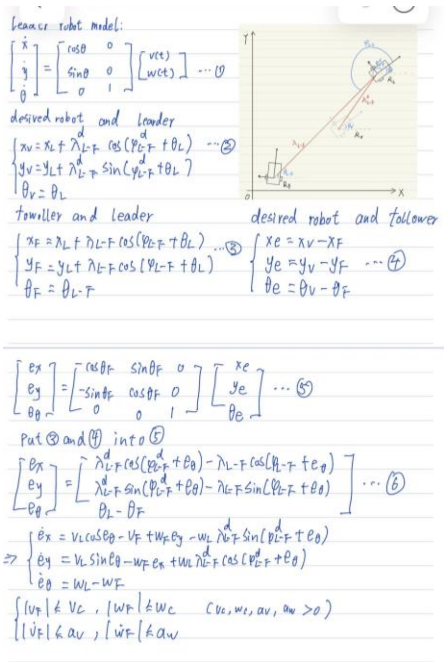
project team added an error calculation formula to the program code for the ideal position between robots to ensure that the error is maintained throughout the entire process. It is 0 when the program is running.
The following formula is the core underlying logic of MPC. It includes state space equations to express the direct distance and angle relationships of each robot.

$$\dot{e}_x = v_L cos e_\theta - v_F + \omega_F e_y - \omega_L \lambda_{L-F}^d \sin\left(\varphi_{L-F}^d + e_\theta\right)$$
$$\dot{e}_y = v_L sin e_\theta - \omega_F e_x + \omega_L \lambda_{L-F}^d \cos(\varphi_{L-F}^d + e_\theta)$$
$$\dot{e}_\theta = \omega_L - \omega_F$$

$$e_x = \lambda_{L-F}^d \cos\left(\varphi_{L-F}^d + e_0\right) - \lambda_{L-F}^d \cos(\varphi_{L-F} + e_\theta)$$
$$e_y = \lambda_{L-F}^d \sin\left(\varphi_{L-F}^d + e_0\right) - \lambda_{L-F}^d \sin(\varphi_{L-F} + e_\theta)$$
$$e_\theta = \theta_L - \theta_F$$

$$J = \sum_{k=0}^{N-1} \left(e(k)^T Q e(k) + u(k)^T R u(k)\right) + e(N)^T P e(N)$$

This is a detailed derivation process:


Leader robot model:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v(t) \\ w(t) \end{bmatrix} \cdots ①$$

desired robot and leader

$$\begin{cases} x_v = x_L + \lambda_{L-F}^d \cos(\varphi_{L-F}^d + \theta_L) \\ y_v = y_L + \lambda_{L-F}^d \sin(\varphi_{L-F}^d + \theta_L) \\ \theta_v = \theta_L \end{cases} \cdots ②$$

fowller and leader

$$\begin{cases} x_F = x_L + \lambda_{L-F} \cos(\varphi_{L-F} + \theta_L) \\ y_F = y_L + \lambda_{L-F} \cos(\varphi_{L-F} + \theta_L) \\ \theta_F = \theta_{L-F} \end{cases} \cdots ③$$

desired robot and follower

$$\begin{cases} x_e = x_v - x_F \\ y_e = y_v - y_F \\ \theta_e = \theta_v - \theta_F \end{cases} \cdots ④$$

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} \cos\theta_F & \sin\theta_F & 0 \\ -\sin\theta_F & \cos\theta_F & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ \theta_e \end{bmatrix} \cdots ⑤$$

Put ③ and ④ into ⑤

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} \lambda_{L-F}^d \cos(\varphi_{L-F}^d + e_\theta) - \lambda_{L-F} \cos(\varphi_{L-F} + e_\theta) \\ \lambda_{L-F}^d \sin(\varphi_{L-F}^d + e_\theta) - \lambda_{L-F} \sin(\varphi_{L-F} + e_\theta) \\ \theta_L - \theta_F \end{bmatrix} \cdots ⑥$$

$$\Rightarrow \begin{cases} \dot{e}_x = v_L \cos e_\theta - v_F + w_F e_y - w_L \lambda_{L-F}^d \sin(\varphi_{L-F}^d + e_\theta) \\ \dot{e}_y = v_L \sin e_\theta - w_F e_x + w_L \lambda_{L-F}^d \cos(\varphi_{L-F}^d + e_\theta) \\ \dot{e}_\theta = w_L - w_F \end{cases}$$

$$\begin{cases} |v_F| \le v_c, \ |w_F| \le w_c \\ |\dot{v}_F| \le a_v, \ |\dot{w}_F| \le a_w \end{cases} \quad (v_c, w_c, a_v, a_w > 0)$$

In the MPC code, the project team set a predetermined step size N for the entire program. After discussion and trial runs, the value of N was set to 5.

```
% time parameter
dt = step/v_cart;
numSteps = length(omega_L_1);

% MPC parameter
N = 5; % Projected time step
Q = diag([20, 25, 10]); % state weight
R = diag([0.01, 0.01]); % control weighting
```

The team used several minion functions in the MPC code to find the minimum of a constrained nonlinear multivariable function. Using fmincon can help our program discover the optimal path during operation. The project aims to find the optimal way for the robot team to satisfy the minimum energy loss and allow the follower to pursue the leader stably.

Random venue code

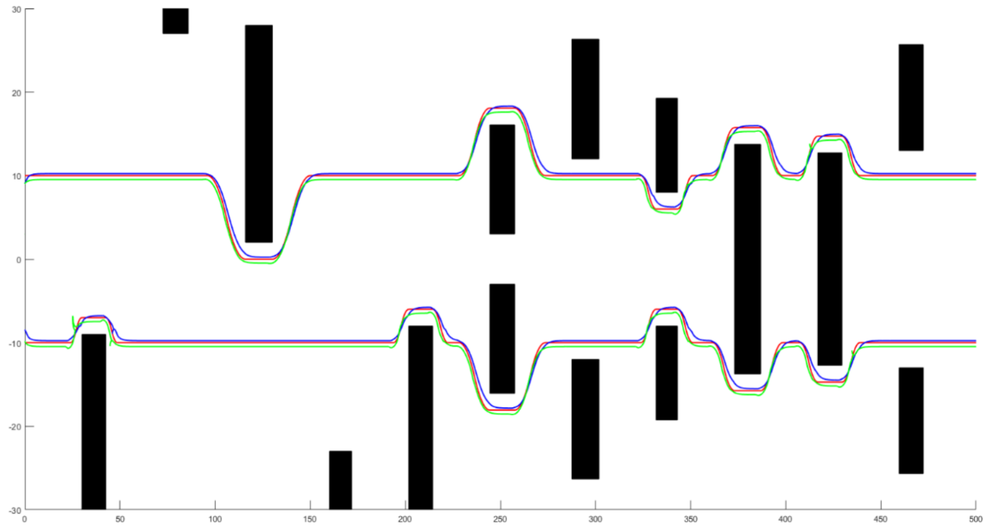
After the project theory above was presented in the form of MATLAB code, the project team combined the random paths to generate the following two sets of random tracks for subsequent MPC path control testing.
The size of the random field is a rectangle of 500*60. The position of the randomly generated obstacles will change each time, but the size is a fixed value. Specific data can be found in the figure below.

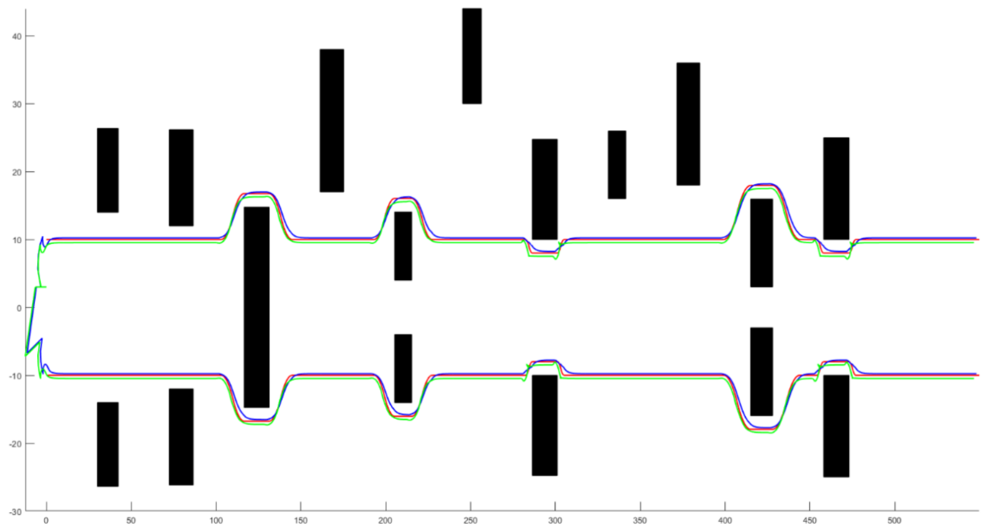
```
% Parameter initialisation
x_max = 500; % Maximum width of the map
y_max = 30; % Maximum height of the map
y_min = -30; % Minimum height of the map
obstacle_gap_range = [50, 50]; % Range of gaps between obstacles
obstacle_length_range = [10, 15]; % obstacle length range
obstacle_width_range = [10, 30];  % obstacle width range

% Initialising an array of obstacles
obstacles = [];

x_current = 0; % current x-axis position
```

```
    x_current = 0; % current x-axis position
    while x_current < x_max
        % Generate starting x-axis of obstacle, add random gap
        x_start = x_current + obstacle_gap_range(1) + (obstacle_gap_range(2) - obstacle_gap_range(1)) * rand();
        % Out of bounds check
        if x_start > x_max
            break;
        end
        % Randomly generated lengths of obstacles
        obstacle_length = obstacle_length_range(1) + (obstacle_length_range(2) - obstacle_length_range(1)) * rand();
        % Double obstacle logic, triggered only when the obstacle length
        % is less than a specific value
        if obstacle_length < 45/2
            double_obs = randi([1, 2]); % Randomly decide whether to generate a double obstacle
            if double_obs == 1
                direction = randi([0, 1]) * 2 - 1;
                y_start = direction * randi([0, y_max/2]);
                y_end = y_start + direction * obstacle_length;
                % Mirror the location of the obstacle
                y_start_mirror = -y_end;
                y_end_mirror = -y_start;
                % Restriction of obstacle locations within the map
                y_start = max(y_min, min(y_max, y_start));
                y_end = max(y_min, min(y_max, y_end));
                y_start_mirror = max(y_min, min(y_max, y_start_mirror));
                y_end_mirror = max(y_min, min(y_max, y_end_mirror));
                % Adding obstacles to an array (original and mirrored)
                obstacles = [obstacles; x_start, min(y_start, y_end), x_start + obstacle_length, max(y_start, y_end)];
                obstacles = [obstacles; x_start, min(y_start_mirror, y_end_mirror), x_start + obstacle_length, max(y_start_mirror,
y_end_mirror)];
                x_current = x_start + obstacle_length;
                continue;
            end
        end
```

Using the MATLAB code added in the appendix, various venues can be randomly generated. Each platform brings obstacles in different locations. For this reason, the collaborative robot's obstacle avoidance and pathfinding code is particularly critical. This way, collaborative tasks can be completed with the lowest energy consumption. After the random obstacle course is generated, you need to continue to use MATLAB to determine the synchronization code for subsequent robot team collaboration.

Results and discussion





In the two sets of obstacle paths randomly generated above, the blue line is the trajectory of the Leader, and the red and green line is the trajectory of the Follower. Black walls are randomly generated obstacles. When running the program, each run will create obstacles with random positions and random sizes in the X and Y directions within a given field.

The project team conducted as many as 30 tests, all passing. This further demonstrates that the program can effectively allow robot formations to pass through randomly generated fields under the MPC control code written by the team. The two sets of images shown above are randomly selected test results. The purpose of conducting multiple tests is to eliminate errors and ensure that each run passes the test rather than by chance.

Conclusion

Overall, the code for this project meets the outline requirements and is logical. The simplicity of the running program is ensured by using multiple subroutines. The completely random obstacles further add to the persuasiveness of the program. After 30 system tests, the robot formation and obstacle avoidance programs run flawlessly and accurately.

In this report, many algorithms and code-writing instructions can help groups working in the same field understand the code of this project. Each subroutine follows the MPC underlying core code and further improves the main program.

Through this project, the team members have further deepened their core understanding of mathematical control logic, and writing program code from scratch has connected with the members' coding abilities. This project is a practical attempt and proves that understanding EFK and MPC can further support robot formations.

References

Zhao, S., & Zelazo, D. (2015). Bearing-based formation stabilization with directed interaction topologies. In 2015 54th IEEE Conference on Decision and Control (CDC) (pp. 6115-6120). Osaka, Japan. doi: 10.1109/CDC.2015.7403181.

Zhao, S., & Zelazo, D. (2017). Translational and Scaling Formation Maneuver Control via a Bearing-Based Approach. IEEE Transactions on Control of Network Systems, 4(3), 429-438. https://doi.org/10.1109/TCNS.2015.2507547.

Zhao, S., & Zelazo, D. (2016). Bearing Rigidity and Almost Global Bearing-Only Formation Stabilization. IEEE Transactions on Automatic Control, 61(5), 1255-1268. https://doi.org/10.1109/TAC.2015.2459191.

Appendix

Since the program has a lot of code and a large number of subroutines, please refer to the submitted code attachment.