1 Introduction

The objective of this project is to ensure that multiple agents, as they move together, maintain their relative positions unchanged through control methods. This allows the agents to form a stable formation of fixed shape, enabling them to work in coordination to accomplish complex tasks. In this process, MATLAB is used to simulate the movement process of Agents, Lidar scanning, Monte Carlo localization, control methods and final formation movement.

In the entire project, the most important and complex step is how to control the velocity and angular velocity of different Agents so that their positional relationship with other agents remains unchanged during movement. Therefore, this article first introduces the formation control method for agents, and then gradually introduce the relevant steps of MATLAB simulation.

2 Formation Control of Multi-Agent Systems

This chapter introduce how to maintain the relative positions of Agents in a formation using control methods, ensuring the preservation of geometric constraints between each Agent. The primary concept is inspired by the paper 'Modeling and Controller Design for Multiple Mobile Robots Formation Control' [1].

2.1 Input and Output of Control System

In order to ensure that the relative position of the Agent to other Agents remains unchanged during the movement. The Agent must have the ability to self-localization, meaning it can determine its own pose within the grid map. The Agent's pose is used as an input to the control system, which then calculates the agent's linear velocity and angular velocity. These velocities are used as feedback to maintain fixed geometric constraints between different agents. Then, the Agent moves to a new position based on these velocities, maintaining fixed geometric constraints among different agents. Therefore, the input to the control system is the agent's pose: **pos_a = [x_a, y_a, theta_a]**, and the linear velocity and angular velocity. The output of the control system is the agent's **linear velocity *v_a***nd angular **velocity *w_a***.

2.2 Establishment of Error Model

If the relative positions between Agents do not change within the formation, it is necessary to establish an error model to determine the error between the actual position of the Agent and the desired position. This error should then be minimized through control methods. The establishment of the error model is based on the principles of Leader- Follower model.

To determine the desired position for each Agent, the concept of a virtual center point is introduced in this process. The virtual center point is located at the geometric center of the Agent formation. For example, four Agents form a rectangular formation, then the virtual center point of the formation would be at the intersection of the diagonals of the rectangle. The virtual center point is not a real entity; it is a fictitious leader of the formation, and all Agents (Followers) must follow the movement of this virtual center to carry out the formation tasks. By setting constraints on the distance (l) and angle (⌀)

between each Agent and the virtual center point, it is possible to maintain a fixed relative position for each Agent within the formation.

As shown in Figure 2-1, R1 represents the virtual center point, while R2 and R3 represent two Agents. The distance constraint between R2 and the virtual center point is denoted as "L," and the angle constraint is represented as "⌀." These constraints are used to maintain a fixed geometric shape for all the Agents as they move.

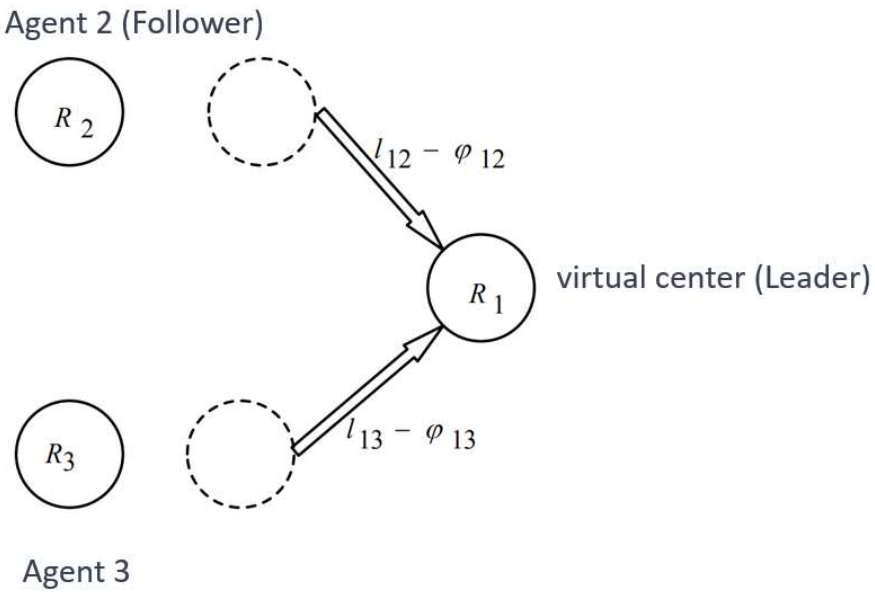

Figure 2-1: Constraints between the Virtual Center and Agents

Due to the introduction of geometric constraints with respect to the virtual center, establishing a coordinate system with the virtual center as the origin will simplify the problem. As shown in Figure 2-2, by setting the virtual center point (leader) as the coordinate system's origin, with a distance constraint of " l* " and an angle constraint of " ⌀* " ,the target position coordinates for the Agents (Followers) in the virtual center coordinate system are as follows:

$$\begin{cases} l_x^* = l^* \cos\varphi^* \\ l_y^* = l^* \sin\varphi^* \end{cases}$$

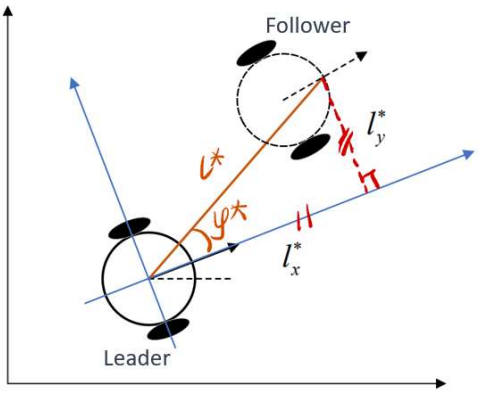

Figure 2-2: Virtual Center Coordinate System and World Coordinate System

The next step is to calculate the actual position of the Agent (Follower) in the virtual center coordinate system. Given that the Agent's pose in the world coordinate system is represented as $(\boldsymbol{xf}, \boldsymbol{yf}, \boldsymbol{\theta f})$, and the virtual center (Leader) in the world coordinate system is represented as $(\boldsymbol{xl}, \boldsymbol{yl}, \boldsymbol{\theta l})$, the geometric relationships shown in Figure 2-3 allow us to determine the actual position coordinates of the Agent in the virtual center coordinate system as follows:

$$\begin{cases} l_x = (x_f + d\cos\theta_f - x_l)\cos\theta_l + (y_f + d\sin\theta_f - y_l)\sin\theta_l \\ l_y = (y_f + d\sin\theta_f - y_l)\cos\theta_l - (x_f + d\cos\theta_f - x_l)\sin\theta_l \end{cases}$$

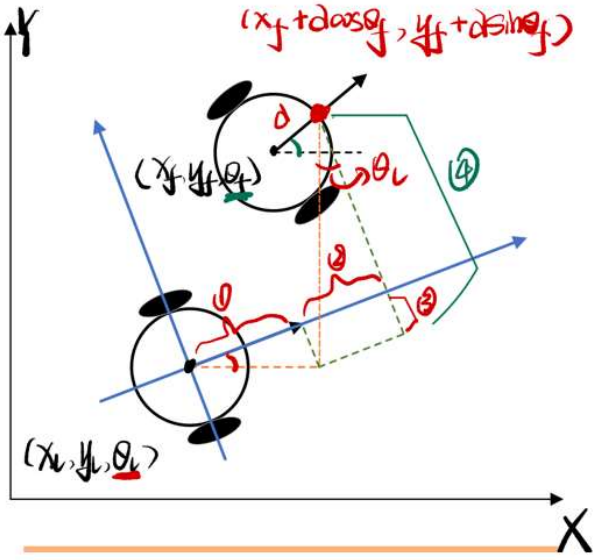

Figure 2-3: Transformation of Coordinate Points in the Virtual Center Coordinate System

Therefore, in the virtual center coordinate system, the position error of the Agent is as follows:

$$\begin{cases} e_x = l_x^* - l_x \\ e_y = l_y^* - l_y \end{cases}$$

If the errors ex and ey are both equal to 0, the Agent can ensure that its relative position within the formation remains unchanged. So, the next step is to control the Agent's velocity and angular velocity to make the error equal to 0.

2.3 Solving the Control Law

Deriving the error equations with respect to time leads to the state-space equations. This is a nonlinear equation, and the specific steps are as follows:

$$\begin{cases} \dot{e}_x = \dot{l}_x^* - \dot{l}_x \\ \dot{e}_y = \dot{l}_y^* - \dot{l}_y \end{cases} \qquad \begin{cases} l_x^* = l^* \cos\varphi^* \\ l_y^* = l^* \sin\varphi^* \end{cases}$$

$$\begin{cases} l_x = (x_f + d\cos\theta_f - x_l)\cos\theta_l + (y_f + d\sin\theta_f - y_l)\sin\theta_l \\ l_y = (y_f + d\sin\theta_f - y_l)\cos\theta_l - (x_f + d\cos\theta_f - x_l)\sin\theta_l \end{cases}$$

$$\begin{cases} \dot{l}_x^* = \dot{l}^*\cos\varphi^* - l^*\dot{\varphi}^*\sin\varphi^* \\ \dot{l}_y^* = \dot{l}^*\sin\varphi^* - l^*\dot{\varphi}^*\cos\varphi^* \end{cases}$$

$$\begin{aligned} \dot{l}_x &= (v_f\cos\theta_f - dw_f\sin\theta_f - v_l\cos\theta_l)\cos\theta_l - (x_f + d\cos\theta_f - x_l)w_l\sin\theta_l + (v_f\sin\theta_f + dw_f\cos\theta_f - v_l\sin\theta_l)\sin\theta_l + (y_f + d\sin\theta_f - y_l)w_l\cos\theta_l \\ &= v_f\cos\theta_f\cos\theta_l - dw_f\sin\theta_f\cos\theta_l - v_l\cos^2\theta_l + v_f\sin\theta_f\sin\theta_l + dw_f\cos\theta_f\sin\theta_l - v_l\sin^2\theta_l + l_y w_l \\ &= v_f\cos\delta_{fl} - dw_f\sin\delta_{fl} - v_l + l_y w_l \\ \dot{l}_y &= (v_f\sin\theta_f + dw_f\cos\theta_f - v_l\sin\theta_l)\cos\theta_l - (y_f + d\sin\theta_f - y_l)w_l\sin\theta_l - (v_f\cos\theta_f - dw_f\sin\theta_f - v_l\cos\theta_l)\sin\theta_l - (x_f + d\cos\theta_f - x_l)w_l\cos\theta_l \\ &= v_f\sin\theta_f\cos\theta_l + dw_f\cos\theta_f\cos\theta_l - v_l\sin\theta_l\cos\theta_l - v_f\cos\theta_f\sin\theta_l + dw_f\sin\theta_f\sin\theta_l + v_l\cos\theta_l\sin\theta_l - l_x w_l \\ &= v_f\sin\delta_{fl} + dw_f\cos\delta_{fl} - l_x w_l \end{aligned}$$

$$\begin{cases} \dot{e}_x = \dot{l}_x^* - \dot{l}_x = \dot{l}^*\cos\varphi^* - l^*\dot{\varphi}^*\sin\varphi^* - (v_f\cos\delta_{fl} - dw_f\sin\delta_{fl} - v_l + l_y w_l) \\ \dot{e}_y = \dot{l}_y^* - \dot{l}_y = \dot{l}^*\sin\varphi^* - l^*\dot{\varphi}^*\cos\varphi^* - (v_f\sin\delta_{fl} + dw_f\cos\delta_{fl} - l_x w_l) \end{cases}$$

Since l* and ∅* represent fixed distance and angle constraints and are constant values, the final state-space equations are as follows:

$$\begin{cases} \dot{e}_x = -v_f \cos\delta_{fl} + dw_f \sin\delta_{fl} + v_l - l_y w_l \\ \dot{e}_y = -v_f \sin\delta_{fl} - dw_f \cos\delta_{fl} + l_x w_l \end{cases}$$

If the derivative of ex and ey with respect to time is less than 0, then over time, $ex\ ey$ will eventually stabilize at 0. Set the left side of the equation equal to the following expression:

$$\begin{cases} -k_1 e_x = -v_f \cos\delta_{fl} + dw_f \sin\delta_{fl} + v_l - l_y w_l \\ -k_2 e_y = -v_f \sin\delta_{fl} - dw_f \cos\delta_{fl} + l_x w_l \end{cases}$$

In that case, the original state-space will transform into the following form:

$$\dot{e}_x = -k_1 e_x$$

To attain system stability, it is imperative to determine positive values for $k1$ and $k2$. Solving the above equation will enable the computation of the linear velocity and angular velocity of the Agent.

$$v_f = [k_1 e_x + v_l - l_y \omega_l]\cos\delta_{fl} + [k_2 e_y + l_x \omega_l]\sin\delta_{fl}$$

$$w_f = \frac{[-k_1 e_x - v_l + l_y \omega_l]\sin\delta_{fl} - [-k_2 e_y - l_x \omega_l]\cos\delta_{fl}}{d}$$

In which: $\delta_{fl} = \theta_f - \theta_l$

V_f: Linear velocity of the Follower/Agent.

W_f: Angular velocity of the Follower/Agent.

V_l: Linear velocity of the leader/ virtual center

W_l: Angular velocity of the leader/ virtual center

d: Distance from the tracking point to the center point of the agent.

3 MATLAB Simulation

The content of this chapter primarily focuses on simulating the Agents formation process using MATLAB, including lidar scanning, Monte Carlo localization, implementation of formation control algorithms, and final result visualization.

3.1 Creation of the Grid Map

To simulate Agents' navigation formation, it is necessary to create an environment in which the robots can operate first. In this process, a grid map is created using MATLAB to represent the robot's working environment.

The grid map divides the environment into regular grid cells or pixels. These cells are typically square or rectangular, forming a structured and orderly two-dimensional network. The advantage of this approach is that two-dimensional coordinates can be directly used to access the status of any location on the map. Each grid cell in the map can store the probability of occupancy by obstacles in that location, or simply use binary values of 0 or 1 to represent free or occupied states.

A binary occupancy map is created using the `map = binaryOccupancyMap(mapmatrix,1)` command in MATLAB [2], where the value of each grid cell in the map is determined by a matrix, `mapmatrix.mat`. The map's size or the occupancy state of the grid cells can be altered by modifying the binary values of each cell in the matrix. The grid resolution of the map is set to 1, indicating that each grid cell represents an area of 1 square meter. If the size of the map or the location of obstacles needs to be modified, just modify the information in “mapmatrix.mat”. Next, the map will be subject to inflation in order to prevent robots from colliding with obstacles during motion.

3.2 Creation of Agents and Virtual Center

After completing the map creation, the next step is to create Agents and virtual center and define their initial positions. Choose the differential drive robot model for the robot's motion with respect to the virtual center, using the '*differentialDriveKinematics*' function in the MATLAB Robotics Toolbox to create them[3]. The code is as follows:

```
agent_0 = differentialDriveKinematics("WheelRadius",0.1525,"TrackWidth", 0.5,
"VehicleInputs", "VehicleSpeedHeadingRate");
```

The detailed explanation of the code is as follows:

- `differentialDriveKinematics` is a function in the MATLAB Robotics Toolbox used to define the kinematic model for a differential drive robot.
- `"WheelRadius", 0.1525` defines the wheel radius of the robot as 0.1525 meters.
- `"TrackWidth", 0.5` specifies the track width of the robot, which is 0.5 meters, i.e., the distance between the wheels.
- `"VehicleInputs", "VehicleSpeedHeadingRate"` specifies the control input mode for the robot, where `"VehicleSpeedHeadingRate"` means that the robot's control inputs are specified in terms of linear speed and heading angular velocity.

Since the virtual center's motion pattern should be identical to that of the agents, the same method and parameters are used to define the motion model for the virtual center.

3.3 Define the Initial Poses of Agents and the Virtual Center

The geometric relationship between agents and the virtual center point is as shown in the diagram.

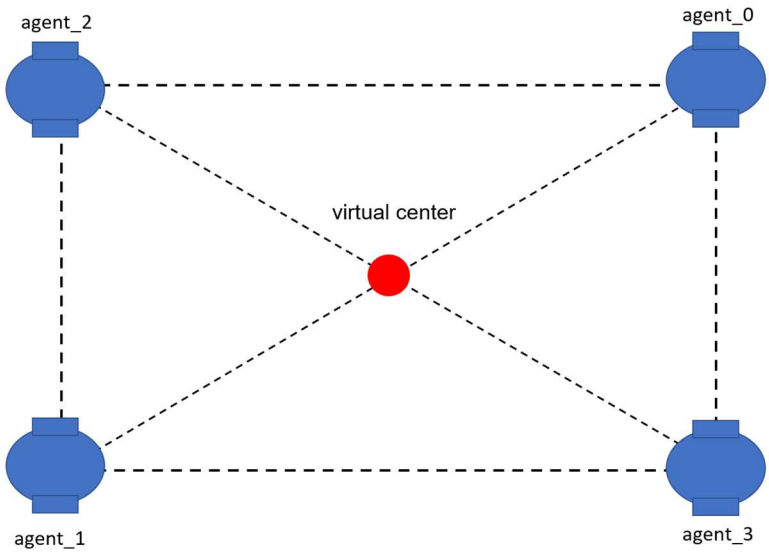

Figure 3-1: Geometric relationship between agents and the virtual center

Four agents form a rectangular formation, with their geometric center being the virtual center point. By setting the initial position of the virtual center point with x, y, and theta, the initial pose of the virtual center can be defined. Based on the pose of virtual center, the poses of all agents can also be determined using the '`Get_Init_Poses`' function. The input and output parameters for this function are as follows:

`center_x, center_y` and `center_theta` represent the x-coordinate, y-coordinate, and orientation (in radians) of the virtual center. The `width` and `length` parameters define the dimensions of the rectangular formation. Once the formation shape is defined, the geometric constraints for the agents and the virtual center: '`target_distance_a`' and '`target_angle_a`,' can be determined through basic geometric principles. Subsequently, these constraint values are returned for use in the following control algorithms. Then, the function calculates the initial poses `[x_a, y_a, theta_a]` of the four agents based on the input data and stores these poses in `pos_a0` through `pos_a3` and returns them.

3.4 Define the Path of Virtual Center

The path of the virtual center point can be determined by directly inputting its velocity and angular velocity or by setting the coordinates of the starting and ending points and using the A* algorithm from the MATLAB Robotics Toolbox to get an optimal path[4]. The code is as follows:

```
%% Astar path planing
planner = plannerAStarGrid(map);
startLocation = [center_x center_y];
endLocation = [16.0 6.0];
path = plan(planner,startLocation,endLocation);
%% define controller
% controller = controllerPurePursuit;
% The virtual center moves at a speed of 0.3 meters per second.
controller = controllerPurePursuit('DesiredLinearVelocity', 0.3);
release(controller);
controller.Waypoints = path;

%start and end point of path
vc_InitialLocation = path(1,:);
vc_Goal = path(end,:);
vc_initialOrientation = center_theta;

%initial position of the virtual center
```

```
pos_v = [vc_InitialLocation vc_initialOrientation]';

%Calculate the distance to the target position
vc_distanceToGoal = norm(vc_InitialLocation - vc_Goal);
% When the virtual center enters the range of a circle with its center at the
endpoint and a radius of 0.3 meters, it is considered that the virtual center
has reached the target.
goalRadius = 0.3;
```

3.5 Simulation of the Formation Control

Based on the control method determined in the previous chapter, the parameters k1 = 0.7, k2 = 0.7, and d = 0.05 are selected. The positions of each agent are determined through a loop that iterates to calculate the changes in their poses.

After the loop begins, the first step is to use the `Calc_Pose_Error` function to calculate the error between the current position of the agent and the desired position. The function returns the error values for the agent in the x-direction and y-direction. These error values will be used in the final result analysis.

Subsequently, the path tracking controller calculates the velocity and angular velocity of the virtual center. This velocity is then used to update the pose of the virtual center `pos_v`.

When the pose of the virtual center point changes, the poses of all agents should also change to satisfy the fixed geometric constraints. So, in next step, the `updateAgentPosition` function is used to determine the linear velocity, angular velocity, and poses of each agent.

The input and output parameters for the function are as follows:

Input:

- `pos_a = [x_a, y_a, theta_a]: previous position of the agent in a 3x1 list`
- `target_distance_a: Distance Constraint Between Agent and Virtual Center`
- `target_angle_a: Angle Constraint Between Agent and Virtual Center (radian)`
- `d, k1, k2: Parameters in the control law`
- `sampleTime: The time interval for measurement, computation, and control during discrete-time steps`
- `pos_v = [x_v, y_v, theta_v]: pose of virtual center in a 3x1 list`
- `vel_v = [vel_v_x, vel_v_y, vel_v_w]': velocity in the xy direction and angular velocity of the virtual center`

- `agent: Object describing the kinematic characteristics of the Agent`

Output:

- `pos_a = [x_a, y_a, theta_a]: new position of the agent`
- `v_a: velocity of the agent`
- `w_a: angular velocity of the agent`

The '`updateAgentPosition`' function takes the geometric constraints between the virtual center and the agent, the actual position of the virtual center, and the agent pose at the previous moment as input. Through the control method in Chapter 2, it calculates the agent's linear velocity and angular velocity, as well as the agent pose at the new moment. Given that each agent uses the same control algorithm and only with different input parameters, so the same '`updateAgentPosition`' function can be employed to 4 agents.

After obtaining the velocity and poses of each agent through the control algorithm, the results are visualized to observe the formation's effectiveness.

Finally, when the virtual center point reaches the vicinity of the endpoint, the loop ends.

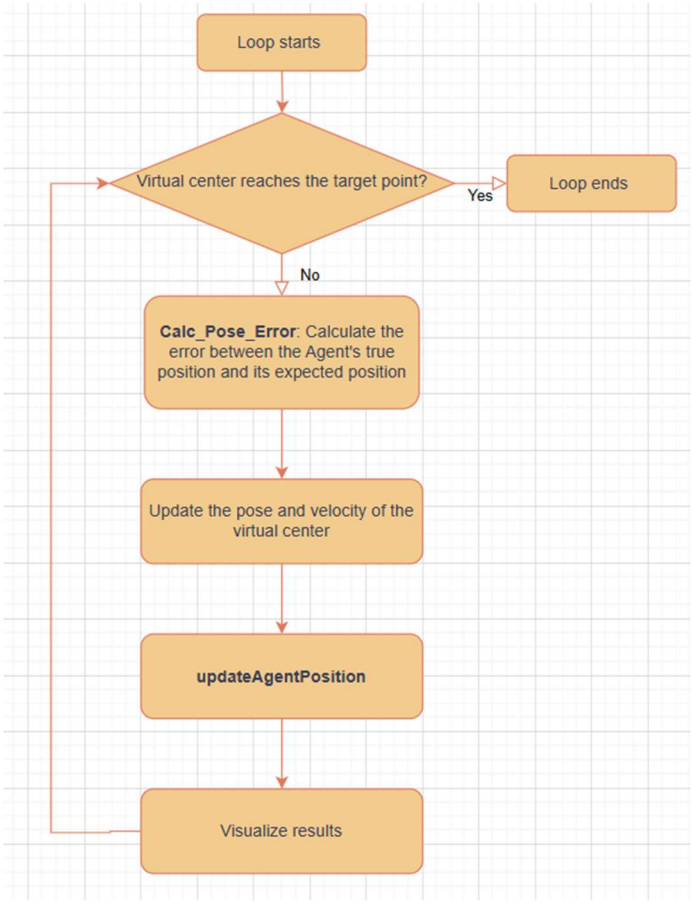

Figure 3-2: Program Flowchart of Formation Control

3.6 Simulation of Lidar Scan

In the previous MATLAB simulation phase, the agent's actual pose is calculated by integrating the velocity and angular velocity computed by the control algorithm over time, which represents only the theoretical pose. We can understand this scenario as when the robot is traveling at a set speed and angular velocity, it uses an odometer or IMU to locate the robot's pose.

However, in real-world situations, factors such as uneven terrain or tire slippage often lead to minor errors when using odometry or IMU. The accumulation of these small errors can have a significant impact on the final results. Therefore, relying solely on odometry and IMU cannot provide precise robot localization. Therefore, the agent's pose obtained through time integration in the previous simulation only represents an ideal scenario and is not realistic.

In reality, to ensure accurate localization, Monte Carlo localization methods based on radar are commonly employed. In reality, to ensure accurate localization, Monte Carlo localization methods based on LiDAR are often employed, and poses determined using the Monte Carlo method are considered to be more precise. Therefore, in this project, to make the simulation more realistic, the poses obtained through the Monte Carlo localization method should be used as inputs in the formation control algorithm. As a result, the Lidar's environmental scanning process needs to be simulated.

In order to simulate the Lidar scanning process, the '`getLidarScan`' function has been developed based on the 'lidarscan' [5] function in MATLAB.

```
scan = getLidarScan(AgentCurrentPose, map)
```

This function takes the current pose of the agent and a grid map as input and returns an object that stores Lidar scan data. The code about emitting laser beams and scanning the environment is as follows:

```
for i = 1:numel(angles)
    % Get the current angle for the laser beam.
    angle = angles(i);
    % Extract the agent's current position and orientation.
    x0 = AgentCurrentPose(1);
    y0 = AgentCurrentPose(2);
    theta0 = AgentCurrentPose(3);
    % Initialize x and y at the agent's current position.
    x = x0;
    y = y0;
```

```
        % Simulate the laser beam's interaction with the map.
        % Iterate while the map occupancy at the current position is less than 0.5
        % and the distance from the starting point does not exceed 'maxRange'.
        while map.getOccupancy([x, y]) < 0.5 && norm([x-x0, y-y0]) <= maxRange
            x = x + cos(theta0 + angle) * distance_resolution;
            y = y + sin(theta0 + angle) * distance_resolution;
        end
        ranges(i) = norm([x-x0, y-y0]);
    end
```

This program simulates the laser radar's rotational scanning process using two nested loops. The outer loop is employed to simulate different scanning angles, while the inner loop is used to model the propagation of the laser beams. When the difference between the xy coordinates of the laser beam and the obstacle's coordinates is less than 0.5, it indicates that the laser has reached the obstacle. This point is recorded and eventually returned. Running the example program "show_lidar_scan.py" allows to observe the Lidar's scan points on the map's obstacles.

3.7 Simulation of Monte Carlo Localization

In this project, MATLAB's "monteCarloLocalization" function is used to achieve precise positioning. Since the four agents use exactly the same Monte Carlo localization settings, a function "`setupMonteCarloLocalization` [6]" is used to set the specific parameters of MCL to make the whole program more concise.

The `setupMonteCarloLocalization` function primarily configures the following parameters:

- UseLidarScan: Enables the use of Lidar scan data for localization.
- GlobalLocalization: A flag to initiate global localization. In this project, global localization is not used. Instead, the initial poses of the agents are directly provided. This approach is taken because global localization may not accurately determine the agents' poses at the beginning of the program, leading to significant deviations in the control algorithms.
- InitialPose: Sets the initial pose of the agent.
- ParticleLimits: Specifies the range of the minimum and maximum number of particles for Monte Carlo Localization. In this project, the minimum number of particles for Monte Carlo Localization is set to 500, and the maximum number of particles is set to 5000.

- InitialCovariance: Defines the covariance of the initial pose, which represents the uncertainty in the initial pose estimation. In this project, these values are set relatively low because the initial poses of the agents are well-determined
- SensorModel: Creates and configures the Likelihood Field Sensor Model, which is used for sensor data likelihood calculations. Utilize the Lidar scan data from the previous subsection and the previously defined grid map.
- UpdateThresholds: Sets the minimum change in states (x, y, and theta) required to trigger a localization update.

After configuring the parameters, the program enters a loop to update the poses of the agents and the virtual center and calculate the agent's linear and angular velocities. The main structure of the loop is similar to what was described in Section 3.5 , but the function `[isUpdated_a, estimatedPose_a, covariance_a] = mcl_a(odometryPose_a, scan_a);` is used to obtain the best estimate of the agent's pose. This function is the main program for Monte Carlo Localization (MCL). It takes the agent's odometry data and Lidar data as inputs and returns the estimated agent's pose and the covariance as outputs.

The odometry pose is obtained by integrating the velocity over time, as calculated by the control algorithm. The Lidar scan data is obtained using the `getLidarScan` function, which uses the estimated pose `pos_a_mcl` from MCL as the observation pose for the Lidar sensor.

The formation control algorithm is implemented by the `updateAgentPosition_mcl` function. Unlike the `updateAgentPosition` from section 3.5 , the new function considers `pos_a_mcl` as the agent's true position and uses it as the input to the control algorithm. After the control algorithm calculates the linear and angular velocities of the agent, it decomposes them into x and y direction velocities by `vel_a = derivative(agent, pos_a_mcl, [v_a w_a]);` Subsequently, the velocity is integrated to obtain odometry data `pos_a = pos_a + vel_a * sampleTime;`. It's important to note that when decomposing the velocity, `pos_a_mcl` should be used because it is considered as the true position of the agent throughout. When calculating the odometry pose, use `pos_a` because it consistently represents the odometry data in the program.

3.8 Analysis of the Result

The error between the actual pose and the desired pose for each agent can be obtained using the `Calc_Pose_Error` function. After running the `show_error.m` program, the error over time plot, the plot of error distribution in the x and y directions, and the figure about error- range in the x and y directions can be obtained. The entire simulation process can be saved as a video for convenient viewing using `videoWriter = VideoWriter(videoFileName, 'Motion JPEG AVI');`

Four virtual center motion trajectories have been defined to observe the simulation's Performance:

Four different virtual center trajectories have been defined:

- case 1: straight path followed by a curved path.
- case 2: straight path
- case 3: circular path
- case 4: combination of straight and circular paths

Case 1 and Case 2, due to their representativeness, have been selected for analysis. The path for Case 1 is shown in the following figure.

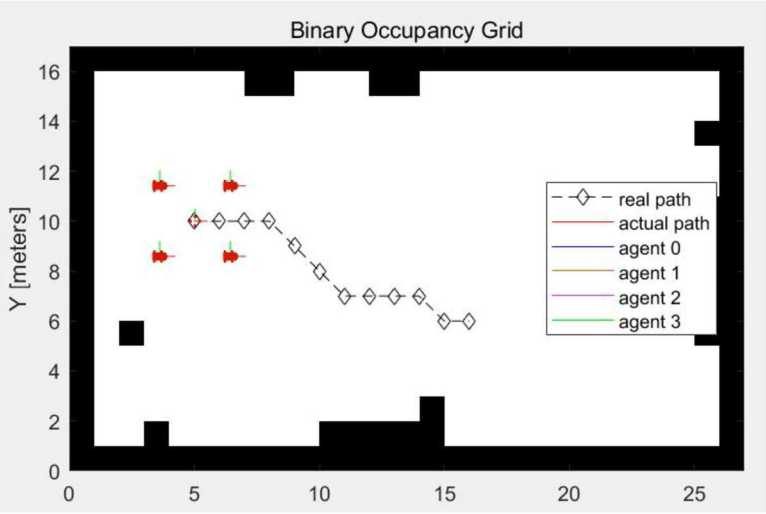

Figure 3-3: Path of case 1

Without using MCL, the errors in the x and y directions for the four agents are depicted in **Figure 3-4**. From the Figure, it can be observed that the maximum errors in the x and y directions are approximately around 0.05 meters. The formation performance appears to be quite good.

The virtual center initially moves horizontally at a fixed speed with a constant y-direction velocity of 0. As a result, the y-direction error of the agents remains at 0 during this time. The x-direction error of the agents starts at 0 when the virtual center begins to move, and error starts to accumulate. The system reaches stability at approximately the 80th iteration, but at this point, there is a steady-state error in the agent's position relative to the virtual center, which is approximately around 0.05 meters. This steady-state error is more pronounced in Case 2. When the virtual center concludes its linear motion and starts turning, the errors in the x and y directions begin to gradually change.

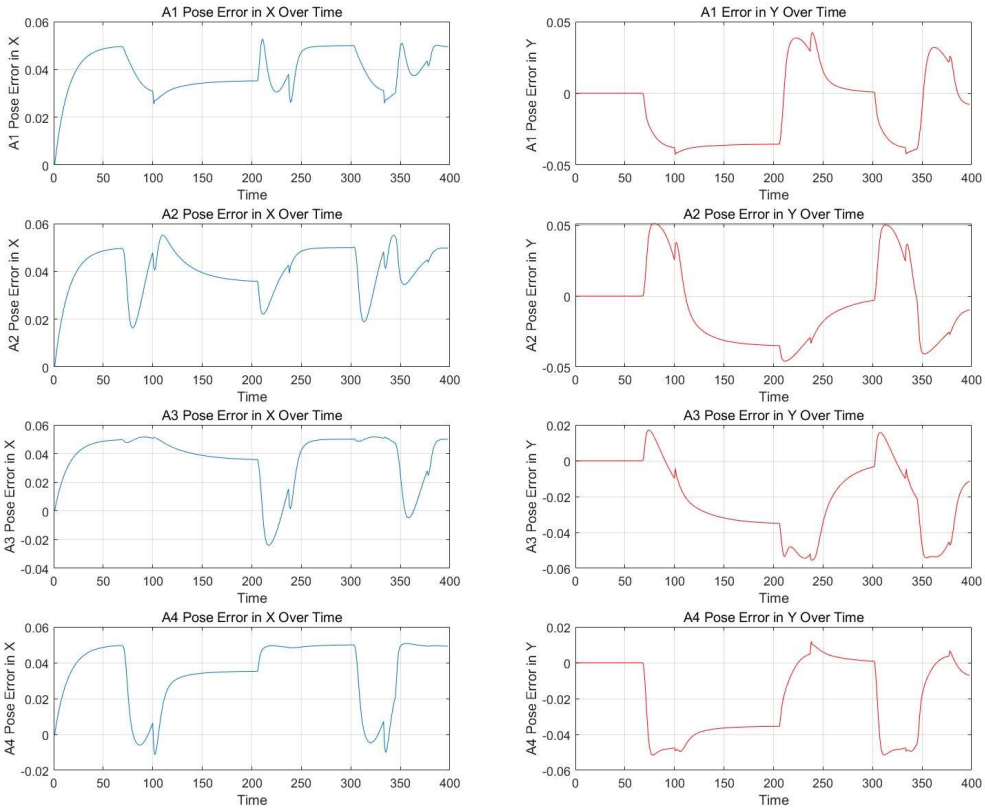

Figure 3-4: Error of the Agents overtime in case1

After introducing MCL into the simulation, the errors for CASE 1 are as shown in Figure 3-5. It can be observed that in the initial stages of the simulation, MCL struggles to accurately determine the position of the agents. As the iterations progress, MCL's estimation of the agent's position becomes more accurate, reducing the error to approximately 0.05 meters. The subsequent error evolution closely resembles the simulation without the introduction of MCL.

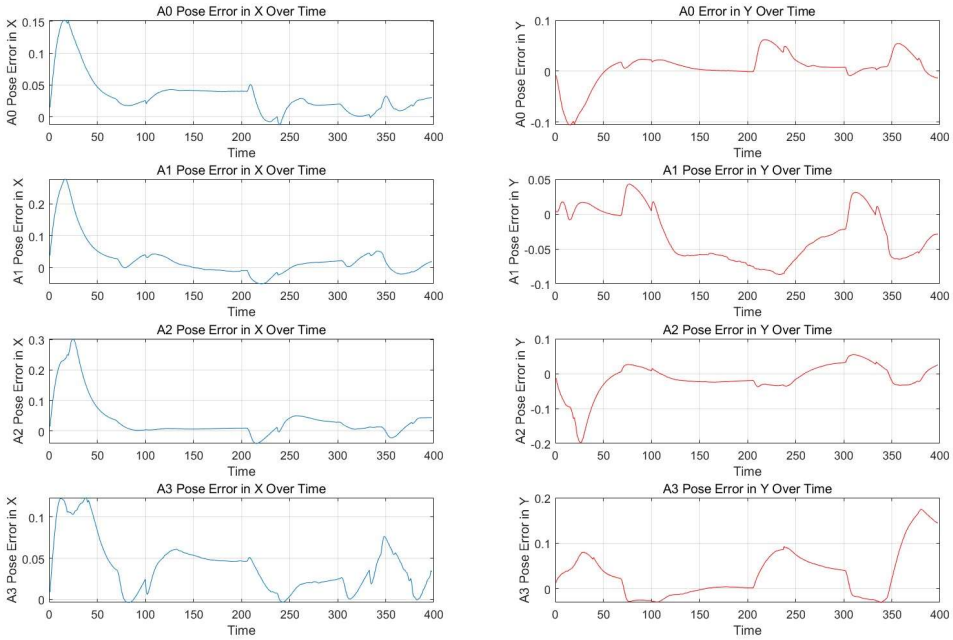

Figure 3-5: Error of the Agents overtime with MCL in case1

In Case 3, when MCL is introduced for simulation, there is a sudden and drastic change in the error when the agent reaches a certain position, causing the system to become unstable. Changing the map or increasing the number of particles in MCL does not resolve this issue. The likely scenario is that there is an issue with the LiDAR scanning process within MCL.

4 Future work

Future work will primarily involve building upon the error model used in this paper to implement more advanced control algorithms aimed at reducing steady-state errors. However, due to the nonlinearity of the error model equations, implementing it is not a straightforward task.

It has been observed in Simulation that the arrangement of agents around a virtual center point also influences the results. As shown in the figure below, a formation of four agents forms a square. When the angle constraint between an agent and the virtual center is set to 0 or 90 degrees, the error is much smaller than when the angle constraint is set to 45 degrees.

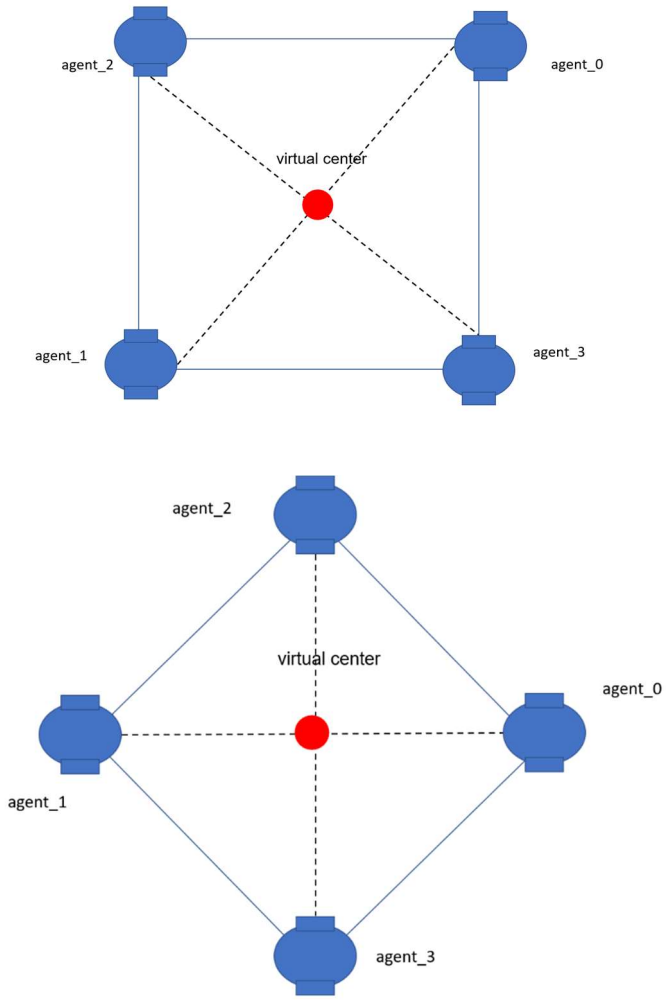

Figure 4-1: Two different formation modes

Finally, the parameters of the MCL can be adjusted to find the best parameter settings for optimal performance.

References

[1] Li X, Xiao J, Tan J. Modeling and controller design for multiple mobile robots formation control[C]//2004 IEEE International Conference on Robotics and Biomimetics. IEEE, 2004: 838-843.

[2] MATLAB: binaryOccupancyMap: https://www.mathworks.com/help/nav/ref/binaryoccupancymap.html

[3] MATLAB: differentialDriveKinematics: https://www.mathworks.com/help/robotics/ref/differentialdrivekinematics.html

[4] MATLAB: A* ,https://www.mathworks.com/help/nav/ref/plannerastargrid.html

[5] MATLAB: lidarScan, https://www.mathworks.com/help/lidar/ref/lidarscan.html

[6] MATLAB: monteCarloLocalization, https://www.mathworks.com/help/nav/ref/montecarlolocalization-system-object.html