AI and Intelligent Robots

Final Lab Report

Name: **MD MAHFUZUR RAHMAN**

Student ID: **3123999081**

Instructor: **Prof. Dr. Xiaodong Zhang**

Submission Date: **26.07.2024**

Table of Contents

1 Introduction..... 3
1.1 Four types of mobile robots..... 4
1.2 Internal and External Sensors..... 5
2 PID Controller for Robot Navigation..... 7
2.1 PID Algorithm Flow Chart..... 8
2.2 Graphs for PID Controller..... 10
2.2.1 Trajectory Graph..... 10
2.2.2 Velocity Graph..... 11
2.2.3 Laser Mapping Graph..... 11
3 Fuzzy Controller for Robot Navigation..... 12
3.1 Fuzzy Algorithm Flow Chart..... 13
3.2 Graphs for Fuzzy Controller..... 14
3.2.1 Trajectory Graph..... 15
3.2.2 Velocity Graph..... 16
3.2.3 Laser Mapping Graph..... 17
4 Appendix..... 18
4.1 Code Snippet for PID Controller..... 18
4.2 Code Snippet for Fuzzy Controller..... 20
4.3 Overview Result of others Lab..... 22
4.3.1 Lab1..... 22
4.3.2 Lab2..... 22
4.3.3 Lab3..... 23
4.3.4 Lab4..... 23
4.4.5 Lab5..... 23
4.3 Fuzzy and PID Code..... 24
4.3.1 PID Code..... 32
4.3.2 Fuzzy Code..... 32

1 Introduction

Mobile robotics is a dynamic and rapidly evolving field that combines elements of mechanical engineering, electronics, computer science, and artificial intelligence to design and implement autonomous robots capable of navigating various environments. These robots have the potential to transform industries such as logistics, healthcare, surveillance, and exploration by performing tasks that are repetitive, dangerous, or beyond human capabilities.

In the realm of mobile robotics, there are four primary types of robots, each distinguished by their modes of locomotion and application domains. Understanding the fundamental differences and functionalities of these types is crucial for the development of effective navigation and control algorithms. The four types of mobile robots typically include wheeled robots, legged robots, aerial robots, and aquatic robots. Each type presents unique challenges and opportunities in terms of mobility, stability, and environmental interaction.

To enable mobile robots to interact effectively with their surroundings, a sophisticated array of internal and external sensors is employed. Internal sensors, such as encoders and gyroscopes, provide critical information about the robot's state, including its position, orientation, and speed. External sensors, such as LiDAR, cameras, and sonar, offer valuable data about the environment, allowing the robot to perceive obstacles, map its surroundings, and make informed navigation decisions.

A key aspect of mobile robot navigation is the implementation of robust control systems that ensure accurate and efficient movement. The Proportional-Integral-Derivative (PID) controller is one of the most widely used control mechanisms in robotics. The PID controller adjusts the robot's movements by continuously calculating the error between a desired setpoint and the actual position, and then applying corrective actions based on proportional, integral, and derivative terms. This report delves into the PID algorithm, presenting its flow-chart and illustrating its effectiveness through trajectory and velocity graphs, as well as laser mapping data.

In addition to the PID controller, fuzzy logic controllers offer an alternative approach to robot navigation, particularly in complex and uncertain environments. Fuzzy logic controllers mimic human reasoning by handling imprecise inputs and applying a set of rules to determine the appropriate output actions. This report explores the fuzzy algorithm, providing a flow-chart and visual representations of its performance in terms of trajectory, velocity, and laser mapping.

The appendix of this report includes detailed code snippets for both the PID and fuzzy controllers, other lab screenshot, offering practical insights into their implementation. By comparing these two control strategies, the report aims to highlight their respective strengths and limitations, providing a comprehensive understanding of their roles in mobile robot navigation.

In summary, this report on AI and Intelligent Robotics focuses on the design, implementation, and comparison of control algorithms for mobile robots. Through the examination of internal and external sensors, PID and fuzzy controllers, and their respective performance metrics, the report seeks to contribute to the ongoing

development and refinement of autonomous robotic systems capable of navigating complex environments with precision and reliability.

1.1 Four Types of Mobile Robots

Mobile robots are autonomous or semi-autonomous machines capable of navigating through and interacting with their environments. They can be broadly classified into four main types based on their locomotion mechanisms and the environments in which they operate: wheeled robots, legged robots, flying robots, and underwater robots. Each type of mobile robot has distinct characteristics, advantages, and challenges.

Wheeled Robots

Wheeled robots are the most common type of mobile robots, primarily due to their simplicity, efficiency, and versatility. They utilize wheels for movement, which makes them highly suitable for flat, hard surfaces such as floors, roads, and pavements. Wheeled robots can be further categorized based on their wheel configurations and mobility mechanisms.

- **Tricycle Mobile Robots**: These robots use a three-wheel configuration, typically with one steering wheel and two driving wheels. They are simple and efficient for many applications.
- **Car-Like Mobile Robots**: These robots mimic the steering and driving mechanism of a car, with front-wheel steering and rear-wheel drive. They are used in applications requiring smooth and controlled turns.
- **Differential Drive Mobile Robots**: These robots use two independently driven wheels on either side of the robot, allowing for precise control over direction and speed. This configuration is popular in many indoor and outdoor applications.
- **Synchros Drive Mobile Robots**: All wheels can be driven and steered independently, allowing the robot to move in any direction. This high degree of maneuverability is useful in complex environments.
- **Tracked Drive Mobile Robots**: Using continuous tracks instead of wheels, these robots can traverse rough terrain and obstacles that would be challenging for wheeled robots.
- **Multi DOFs Mobile Robots**: Robots with multiple degrees of freedom in their wheels or joints, enabling complex movements and adaptability to various terrains.

Legged Robots

Legged robots mimic the movement of animals and humans, using legs to traverse a variety of terrains, including rough and uneven surfaces. The complexity of legged robots makes them more challenging to design and control, but their ability to handle diverse environments makes them invaluable for certain applications.

- **One-Legged Hopping Robots**: These robots use a single leg to hop around. Though seemingly simple, they require sophisticated control algorithms to maintain balance and direction.

- **Bipedal Robots**: These robots have two legs and are designed to walk in a manner similar to humans. Balancing and stability are significant challenges, but advancements in control algorithms and sensors are making bipedal robots increasingly viable.
- **Quadrupedal Robots**: With four legs, these robots are more stable and can move efficiently over uneven terrain. They are used in applications such as search and rescue, military operations, and exploration.
- **Hexapods and Other Multi-Legged Robots**: Robots with six or more legs offer even greater stability and are capable of traversing very challenging environments. They are often used in research and specialized industrial applications.

Flying Robots

Flying robots, or drones, are capable of flight and are used in a wide range of applications, from aerial photography and surveillance to delivery services and environmental monitoring. Flying robots can vary significantly in design, from fixed-wing aircraft to multi-rotor helicopters.

- **Fixed-Wing Drones**: These drones have a wing structure similar to an airplane and are highly efficient for covering large distances and long-duration flights. However, they require runways or catapults for takeoff and landing.
- **Rotary-Wing Drones**: These include multi-rotor designs such as quadcopters, which are highly maneuverable and capable of vertical takeoff and landing (VTOL). They are widely used due to their ease of control and versatility in various applications.

Underwater Robots

Underwater robots operate in water environments, including oceans, rivers, and lakes. They are used for tasks such as underwater exploration, environmental monitoring, and infrastructure inspection.

- **Underwater Autonomous Vehicles (UAVs)**: These robots can operate independently underwater, navigating and performing tasks without real-time human control. They are used for deep-sea exploration, pipeline inspection, and marine biology studies.
- **Remotely Operated Vehicles (ROVs)**: Controlled by operators from the surface, ROVs are equipped with cameras and tools for tasks such as underwater construction, repair, and scientific research.
- **Surface Robots**: These robots operate on the surface of the water and are used for tasks like environmental monitoring, search and rescue, and surface mapping.

1.2 Internal and External Sensors

In the field of mobile robotics, sensors play a critical role in enabling robots to perceive their environment and make informed decisions. Sensors can be broadly categorized into two types: internal sensors and external sensors. Both types are

essential for different aspects of robot functionality, from navigation and obstacle avoidance to state estimation and control.

Internal Sensors

Internal sensors provide information about the robot's own state, including its position, orientation, and motion. These sensors are crucial for maintaining stability, executing precise movements, and ensuring the robot operates correctly according to its control algorithms.

- **Encoders**: Encoders measure the rotation of the robot's wheels or joints, providing data on speed, distance traveled, and angular position. This information is vital for odometry calculations, which help the robot estimate its position relative to a starting point.
- **Gyroscopes**: Gyroscopes measure the rate of rotation around the robot's axes. They are used to determine the robot's orientation and maintain balance, especially in dynamic environments or for robots with complex movements like legged robots.
- **Accelerometers**: Accelerometers measure linear acceleration along the robot's axes. This data can be used to detect changes in motion and orientation, helping to improve the accuracy of the robot's state estimation.
- **Inertial Measurement Units (IMUs)**: IMUs combine gyroscopes and accelerometers to provide comprehensive data on the robot's orientation, acceleration, and velocity. They are essential for advanced motion tracking and stability control in mobile robots.

External Sensors

External sensors allow the robot to perceive and interact with its environment. They provide critical information for tasks such as navigation, obstacle avoidance, mapping, and environment recognition.

- **LiDAR (Light Detection and Ranging)**: LiDAR sensors emit laser beams and measure the time it takes for them to return after hitting an object. This information is used to create precise 3D maps of the environment, detect obstacles, and assist in navigation.
- **Cameras**: Cameras capture visual information about the environment. They can be used for tasks such as object recognition, terrain mapping, and visual navigation. Cameras can be monocular (single lens) or stereo (dual lens) to provide depth perception.
- **Sonar (Ultrasonic Sensors)**: Sonar sensors emit sound waves and measure the time it takes for the echo to return. They are commonly used for detecting obstacles and measuring distances in environments where light-based sensors may be less effective.
- **Infrared Sensors**: Infrared sensors detect infrared radiation and are used for proximity sensing and obstacle detection. They are particularly useful in low-light conditions or for detecting heat signatures.
- **GPS (Global Positioning System)**: GPS sensors provide the robot's global position using satellite signals. They are essential for outdoor navigation, where precise localization over large areas is required.

- **Proximity Sensors**: These sensors detect the presence of nearby objects without physical contact. They are often used for collision avoidance and short-range navigation.

Why Sensors are Required

Sensors are indispensable for mobile robots as they provide the necessary data for the robot to understand its environment and operate autonomously. Without sensors, a robot would be unable to perceive obstacles, navigate through unknown terrains, or interact effectively with its surroundings. Sensors enable the robot to:

- **Navigate Safely**: By detecting and avoiding obstacles, robots can move safely and efficiently within their environment.
- **Maintain Stability**: Internal sensors help the robot maintain balance and execute precise movements, which is especially important for legged robots.
- **Map and Localize**: External sensors like LiDAR and cameras allow the robot to create maps of its environment and determine its location within those maps.
- **Interact with Objects**: Sensors provide the data needed for robots to recognize and manipulate objects, making them capable of performing complex tasks.

How Sensors Differ

Internal and external sensors differ in their placement and the type of information they provide. Internal sensors are located within the robot and focus on the robot's own state, while external sensors are mounted on the robot's exterior and gather information about the environment.

- **Purpose**: Internal sensors monitor the robot's own state (e.g., position, orientation), whereas external sensors perceive the external environment (e.g., obstacles, terrain).
- **Data Type**: Internal sensors provide data related to motion and stability (e.g., gyroscopes, accelerometers), while external sensors offer environmental data (e.g., LiDAR, cameras).
- **Placement**: Internal sensors are embedded within the robot's body, while external sensors are typically mounted on the robot's exterior to maximize their field of view.

2 PID Controller for Robot Navigation

The Proportional-Integral-Derivative (PID) controller is a widely used control algorithm in robotics for achieving precise and stable navigation. It continuously calculates the error between a desired setpoint and the actual position of the robot and applies corrective actions to minimize this error. The PID controller combines three components – proportional, integral, and derivative – to adjust the robot's movements effectively.

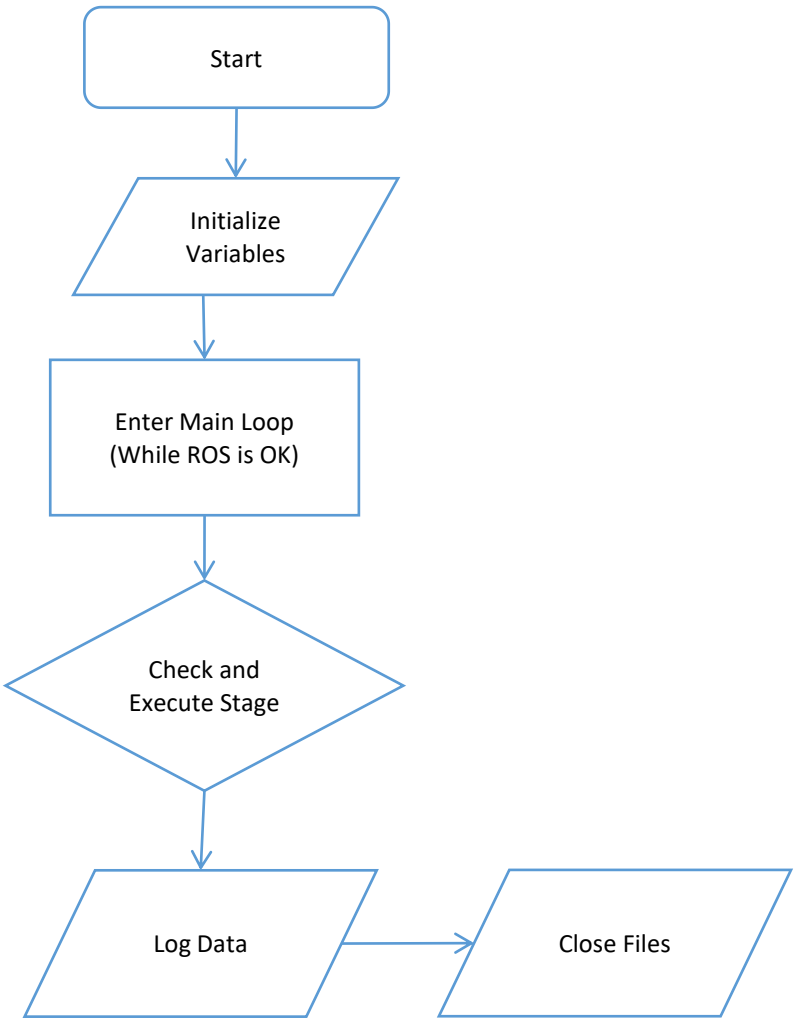
- **Proportional Component (P)**: The proportional component produces an output that is proportional to the current error value. It is responsible for

reducing the overall error and bringing the robot closer to the desired setpoint. The proportional gain (Kp) determines the response strength.

- **Integral Component (I)**: The integral component addresses the accumulated error over time. It eliminates the residual steady-state error that may persist after the proportional control is applied. The integral gain (Ki) determines the influence of this component.
- **Derivative Component (D)**: The derivative component predicts future error based on the current rate of change. It helps in damping the system response and reducing overshoot. The derivative gain (Kd) determines the effect of this component.

By tuning these three gains (Kp, Ki, Kd), the PID controller can be optimized for different robot behaviors and environmental conditions. This section presents the PID algorithm flow chart and its application in robot navigation through trajectory graphs, velocity graphs, and laser mapping.

2.1 PID Algorithm Flowchart



Detailed Breakdown of Each Step

Start

1. Begin the process.

Initialize Variables

1. Initialize ROS node and publishers/subscribers.
2. Initialize PID parameters and stage variables.
3. Open output files (`laserData.csv`, `laserMapData.csv`, `trajectory.csv`, `velocity.csv`).

Enter Main Loop (While ROS is OK)

1. Continue looping as long as ROS is running (`ros::ok()`).

Check and Execute Stage

1. Depending on the current stage, execute the corresponding PID control logic:

Stage 1: Wall Following

1. If `PositionY` < `landmark1`, perform `PID_wallFollowing`.
2. Else, set `stage` to 2.

Stage 2: Move Toward 1st Gap

1. If `PositionX` < `landmark2`, perform `PID_to1stGap`.
2. Else, set `stage` to 3.

Stage 3: Move Toward 2nd Gap

1. If `PositionX` < `landmark3`, perform `PID_to2ndGap`.
2. Else, set `stage` to 4.

Stage 4: Go Through 2nd Gap

1. If `PositionX` < `landmark4`, perform `PID_to2ndGap`.
2. Else, set `stage` to 5.

Stage 5: Move Towards Charger

1. If `PositionX` < `landmark5`, perform `PID_reachGoal`.
2. Else, set `stage` to 6.

Stage 6: Stop Robot

1. Call `moveStop`.

Log Data

1. For each of the 360 laser data points, log to `laserData.csv`.
2. Call `transformMapPoint` to log transformed laser data to `laserMapData.csv`.
3. Log trajectory (PositionX, PositionY) to `trajectory.csv`.
4. Log velocity (linear, angular) to `velocity.csv`.

Close Files

1. Close all opened files (`laserData.csv`, `laserMapData.csv`, `trajectory.csv`, `velocity.csv`).

End

1. End the process.

2.2 Graphs for PID Controller

2.2.1 Trajectory Graph

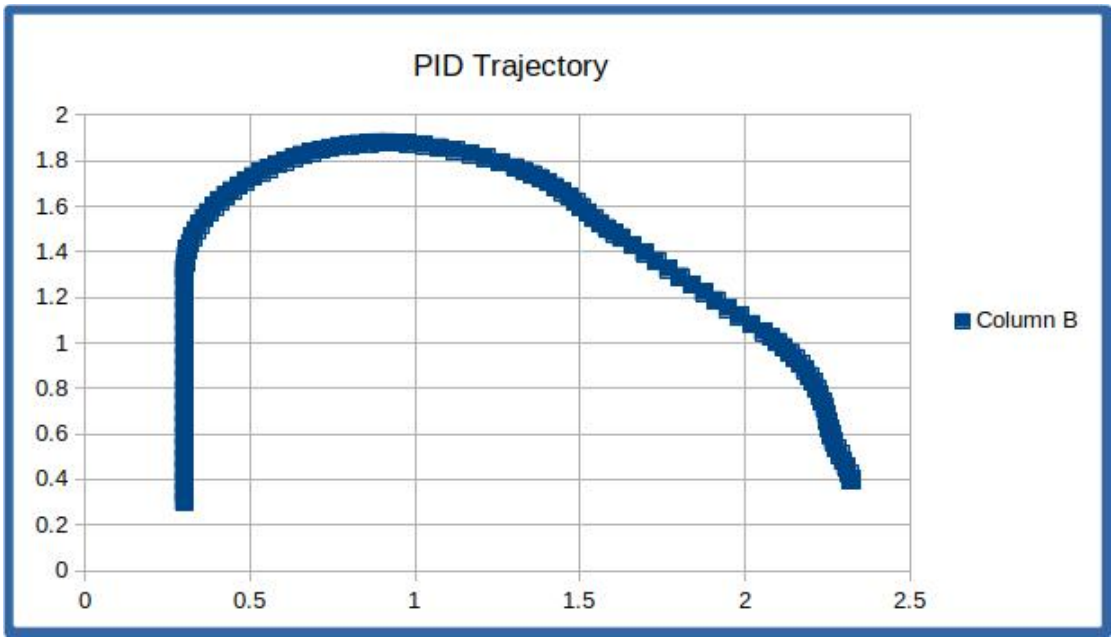


Figure 1. Trajectory Graph for PID Controller

In this figure we see the robot's path using a PID controller for navigation. The X-axis and Y-axis represent the robot's positions in two dimensions, showing a smooth trajectory. Initially, the robot starts at (0.3, 0.3), ascends to approximately (1.0, 2.0), and then descends to (2.3, 0.5). This indicates a steady rise to a peak before a gradual descent, suggesting effective control by the PID controller, which adjusted the robot's movements in real-time to maintain a balanced path.

2.2.2 Velocity Graph

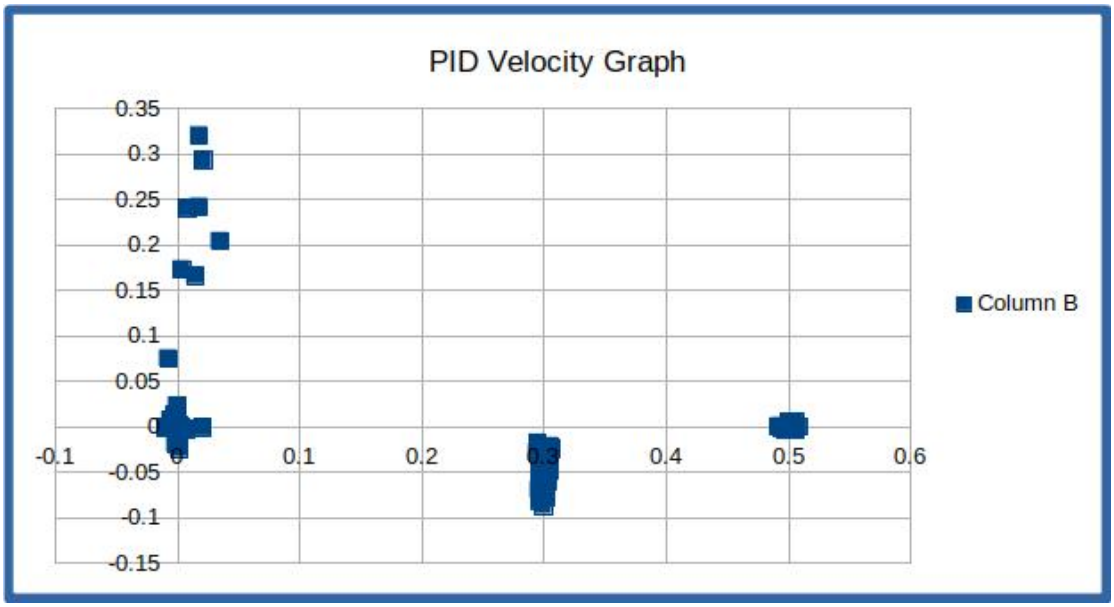


Figure 2. Velocity Graph for PID Controller

This figure represents the robot's velocity over time as it navigates using a PID controller. The X-axis shows the time steps or indices at which the velocity was recorded, and the Y-axis shows the corresponding velocity values. Initially, the robot's velocity starts at 0, indicating a stationary position. The graph displays fluctuations in velocity, with some points showing higher velocities and others showing lower velocities.

2.2.3 Laser Mapping Graph

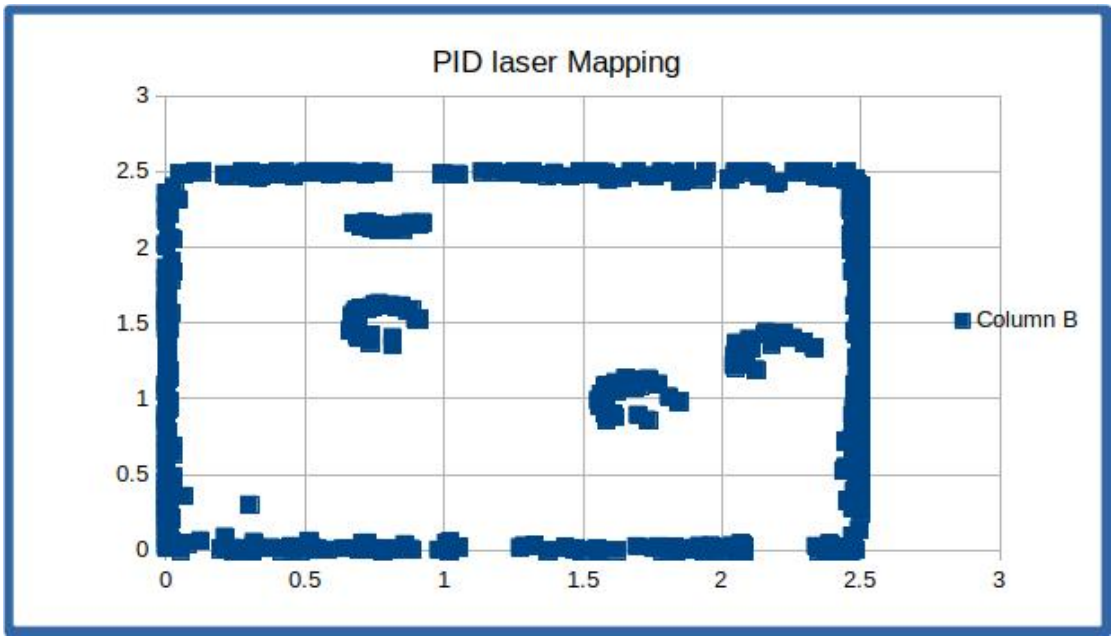


Figure 3. Laer Mapping Graph for PID Controller

Figure 3 represents the map generated by the robot using its laser sensors while navigating an environment with the help of a PID logic controller. The X-axis and Y-axis represent the coordinates of the detected points. The graph shows a collection of points forming a map of the robot's environment, with a clearly defined outer boundary indicating the edges of the mapped area. Various clusters of points within the boundary suggest the presence of obstacles or features, likely representing objects detected by the robot's laser sensors.

3 Fuzzy Controller for Robot Navigation

Fuzzy logic control is another popular approach in robotic navigation, particularly effective in dealing with uncertainties and imprecise information. Unlike traditional control methods that rely on precise mathematical models, fuzzy logic controllers use linguistic variables and a set of rules to determine the control actions. This method is well-suited for environments where precise modeling is difficult or impossible. Below component of fuzzy controller.

Fuzzification: Converts crisp input values into fuzzy values based on predefined membership functions. These input values often include variables such as distance to obstacles, desired direction, and current speed.

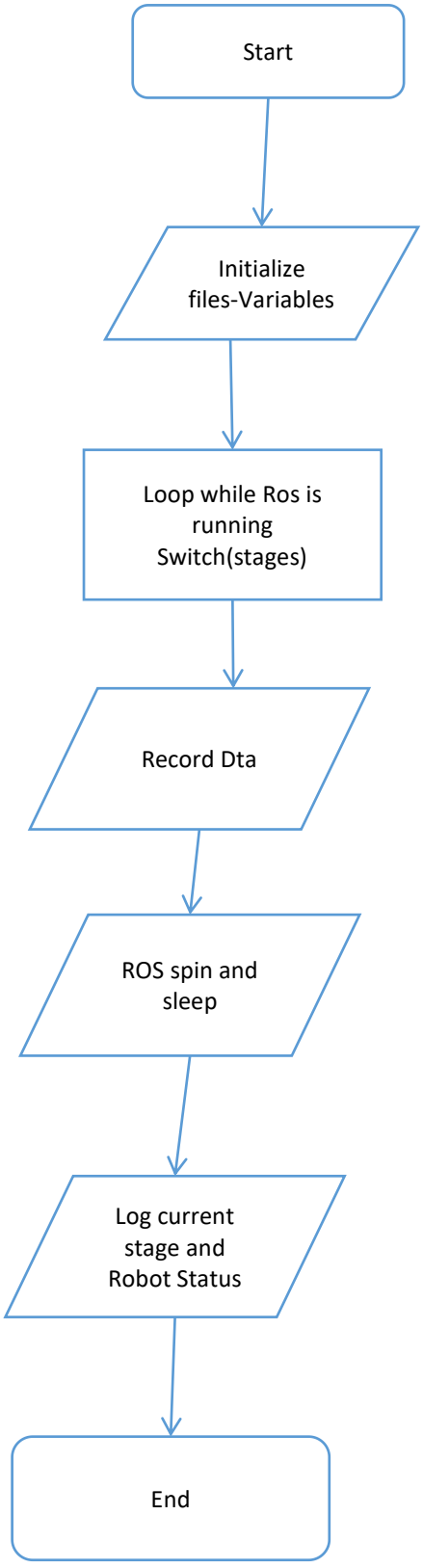
Fuzzy Inference: Uses a set of fuzzy rules stored in a rule base to evaluate the fuzzy input values. These rules are typically expressed in the form of "IF-THEN" statements, such as:

1. IF distance to obstacle is small THEN reduce speed.
2. IF desired direction is left AND distance to obstacle is moderate THEN turn slightly left.

Rule Base: A collection of fuzzy rules that define the control strategy. These rules are derived from expert knowledge or through learning processes.

Defuzzification: Converts the fuzzy output values back into crisp control actions. This step involves selecting a specific value from the fuzzy set that represents the output variable.

3.1 Fuzzy Algorithm Flow Chart



Details Breakdown of Flowchart

- **Start**
- **Initialize Files and Variables**

  - Open `odomVelFile`
  - Open `odomTrajFile`
  - Open `laserFile`
  - Open `laserMapFile`
  - Initialize `stage = 1`

- **Loop while ROS is running**

1. **Switch (stage)**

Case 1: Move Forward from Home

- If `PositionY < landmark1`
  - Call `Fuzzy_wallFollowing(leftRange, mleftRange)`
- Else
  - Set `stage = 2`

Case 2: Turn Right Toward 1st Gap

- If `PositionX < landmark2`
  - Call `Fuzzy_to1stGap(leftRange, mleftRange)`
- Else
  - Set `stage = 3`

Case 3: Move Forward Fast

- If `PositionX < landmark3`
  - Call `Fuzzy_toTrajectory(leftRange, mleftRange)`
- Else
  - Set `stage = 4`

Case 4: Move and Turn Right Slowly

- If `PositionX < landmark4`
  - Call `Fuzzy_to2stGap(leftRange, mleftRange)`
- Else

- Set `stage = 5`

Case 5: Move Towards the Charger

- If `PositionY > landmark5`
  - Call `Fuzzy_toEnd(leftRange, mleftRange)`
- Else
  - Set `stage = 6`

Case 6: Stop at Charger Position

- Call `moveStop()`
- Set `stage = 7`

2. **Record Data**
   - Calculate `runTime` since `startTime`
   - Write `runTime` and `robVelocity` to `odomVelFile`
   - Write `PositionX` and `PositionY` to `odomTrajFile`
   - Write laser data to `laserFile`
   - Transform and write laser map points to `laserMapFile`
3. **ROS Spin and Sleep**
   - Call `ros::spinOnce()`
   - Sleep for the defined rate
4. **Log Current Stage and Robot Status**
   - Log stage, `PositionX`, `PositionY`, `robotHeadAngle`, and `robVelocity`

➢ **End**

3.2 Graphs for Fuzzy Controller

3.2.1 Trajectory Graph

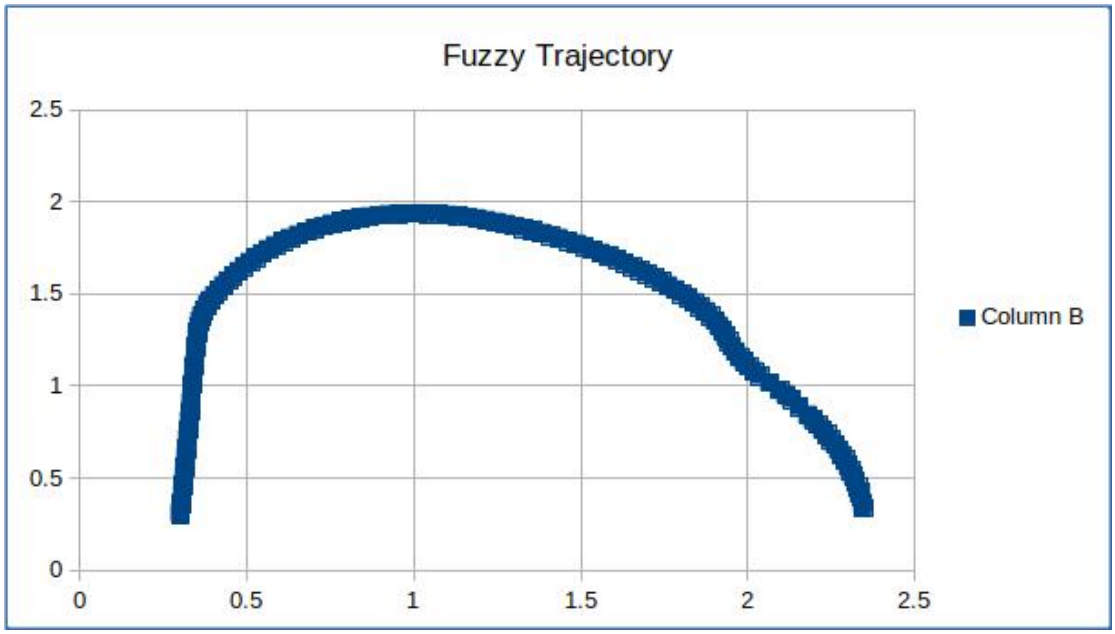


Figure 4. Trajectory Graph for Fuzzy Controller

We can see in this figure the robot's path using a fuzzy controller for navigation. The X-axis and Y-axis represent the robot's positions in two dimensions, showing a smooth trajectory. Initially, the robot starts at (0.3, 0.3), ascends to approximately (1.0, 2.0), and then descends to (2.3, 0.5). This indicates a steady rise to a peak before a gradual descent, suggesting effective control by the fuzzy controller, which adjusted the robot's movements in real-time to maintain a balanced path.

3.2.2 Velocity Graph

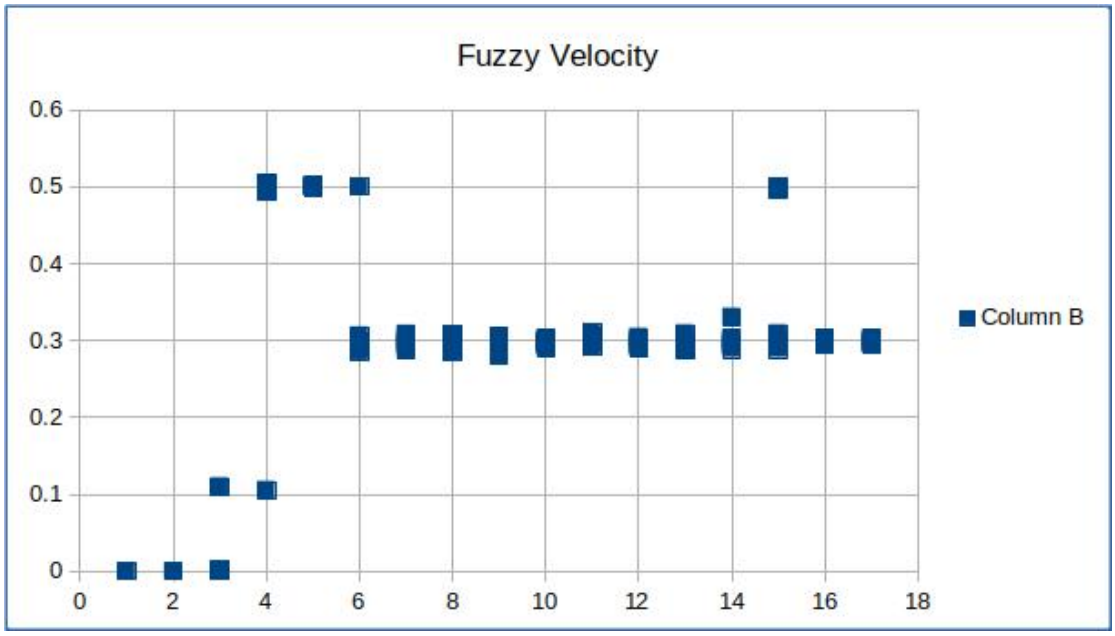


Figure 5. Velocity Graph for Fuzzy Controller

This figure represents the robot's velocity over time as it navigates using a fuzzy controller. The X-axis shows the time steps or indices at which the velocity was recorded, and the Y-axis shows the corresponding velocity values. Initially, the robot's velocity starts at 0, indicating a stationary position. The graph displays fluctuations in velocity, with some points showing higher velocities (e.g., around time steps 4 and 14) and others showing lower velocities. The legend "Column B" refers to the second column of the CSV file, which corresponds to the recorded velocity values. Each row in the CSV file represents the robot's velocity at a particular time step, with initial rows showing zero velocity, indicating the stationary start.

3.2.3 Laser Mapping Graph

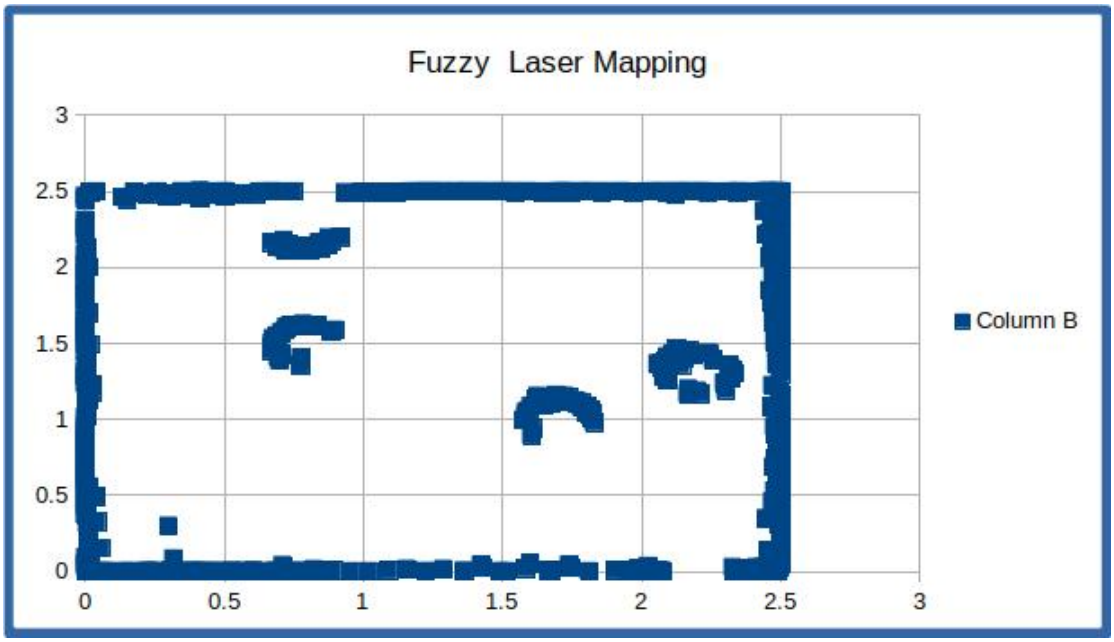


Figure 6. Laser Mapping Graph for Fuzzy Controller

This graph represents the map generated by the robot using its laser sensors while navigating an environment with the help of a fuzzy logic controller. The X-axis and Y-axis represent the coordinates of the detected points. The graph shows a collection of points forming a map of the robot's environment, with a clearly defined outer boundary indicating the edges of the mapped area. Various clusters of points within the boundary suggest the presence of obstacles or features, likely representing objects detected by the robot's laser sensors.

4 Appendix

4.1 Code Snippet for PID Controller

Wall Following PID Controller

```
void RobotMove::PID_wallFollowing(double moveSpeed, double laserData) {
    double ei, ed, err, output;
    err = landmark1_toWall - laserData;
    ei = ei_pre1 + err;
    ed = err - ed_pre1;
    ei_pre1 = ei;
    ed_pre1 = ed;
    output = kp1 * err + ki1 * ei + kd1 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}
```

Move Toward the Middle of the 1st Gap PID Controller

```
void RobotMove::PID_to1stGap(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark2_heading - robotHeading;
    ei = ei_pre2 + err;
    ed = err - ed_pre2;
    ei_pre2 = ei;
    ed_pre2 = ed;
    output = kp2 * err + ki2 * ei + kd2 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}
```

LibreOffice Writer

Move Toward the 2nd Gap PID Controller

```
void RobotMove::PID_to2ndGap(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark3_heading - robotHeading;
    ei = ei_pre3 + err;
    ed = err - ed_pre3;
    ei_pre3 = ei;
    ed_pre3 = ed;
    output = kp3 * err + ki3 * ei + kd3 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}
```

Reach Goal PID Controller

```
void RobotMove::PID_reachGoal(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark5_heading - robotHeading;
    ei = ei_pre4 + err;
    ed = err - ed_pre4;
    ei_pre4 = ei;
    ed_pre4 = ed;
    output = kp4 * err + ki4 * ei + kd4 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}
```

Complete PID Controllers

```
void RobotMove::startMoving() {
    std::ofstream laserFile("/home/mahfuz/ros_workspace/src/tutorial_pkg/laserData.csv");
    std::ofstream laserMapFile("/home/mahfuz/ros_workspace/src/tutorial_pkg/laserMapData.csv");
    trajectoryFile.open("/home/mahfuz/ros_workspace/src/tutorial_pkg/trajectory.csv");
    velocityFile.open("/home/mahfuz/ros_workspace/src/tutorial_pkg/velocity.csv");

    if (!laserFile.is_open() || !laserMapFile.is_open() || !trajectoryFile.is_open() || !velocityFile.is_open()) {
        ROS_ERROR("Failed to open CSV files for writing");
        return;
    }

    ros::Rate rate(20); // Define rate for repeatable operations.
    ROS_INFO("Start moving");

    while (ros::ok()) { // keep spinning loop until user presses Ctrl+C
        switch (stage) {
            case 1: // wall following
                if (PositionY < landmark1) {
                    PID_wallFollowing(FORWARD_SPEED_HIGH, leftRange);
                } else {
                    stage = 2;
                }
                break;
            case 2: // move toward the middle of the 1st gap
                if (PositionX < landmark2) {
                    PID_to1stGap(FORWARD_SPEED_MIDDLE, robotHeadAngle);
                } else {
                    stage = 3;
                }
                break;
            case 3: // move toward the 2nd gap
                if (PositionX < landmark3) {
                    PID_to2ndGap(FORWARD_SPEED_MIDDLE, robotHeadAngle);
                } else {
                    stage = 4;
                }
                break;
            case 4: // go through the 2nd gap
                if (PositionX < landmark4) {
                    PID_to2ndGap(FORWARD_SPEED_LOW, robotHeadAngle);
                } else {
                    stage = 5;
                }
                break;
            case 5: // move towards the charger
                if (PositionX < landmark5) {
                    PID_reachGoal(FORWARD_SPEED_LOW, robotHeadAngle);
```

4.2 Code Sippet for Fuzzy Controller

Fuzzy Controller for moving along the wall

```
//  Fuzzy controller for moving along the wall
void RobotMove::Fuzzy_wallFollowing(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (laserData1 < 0.2)   fuzzySensor1 = 1;       // The robot is near to the wall
    else if (laserData1 < 0.5)  fuzzySensor1 = 2;       // The robot is on the right distance
    else    fuzzySensor1 = 3;       // The robot is far from the wall;

    if (laserData2 < 0.3)   fuzzySensor2 = 1;   // The robot is near to the wall
    else if (laserData2 < 0.6)  fuzzySensor2 = 2;   // The robot at the right distance;
    else    fuzzySensor2 = 3;   // The robot is far from the wall;

    // Fuzzy rule base and control output
    if (fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_SPEED_ZERO);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_LEFT_SPEED_LOW);
    else ROS_INFO("Following the left wall");
}
```

Fuzzy Controller Going through 1st gap

```
//  Fuzzy controller for going through the gap 1
void RobotMove::Fuzzy_to1stGap(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.2)  fuzzySensor1 = 1;
    else if (laserData1 < 0.5)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.2)   fuzzySensor2 = 1;
    else if (laserData2 < 0.9)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if(fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_HIGH);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_RIGHT_SPEED_LOW);
    else ROS_INFO("Going through the 1st gap");
}
```

Going through 2nd gap

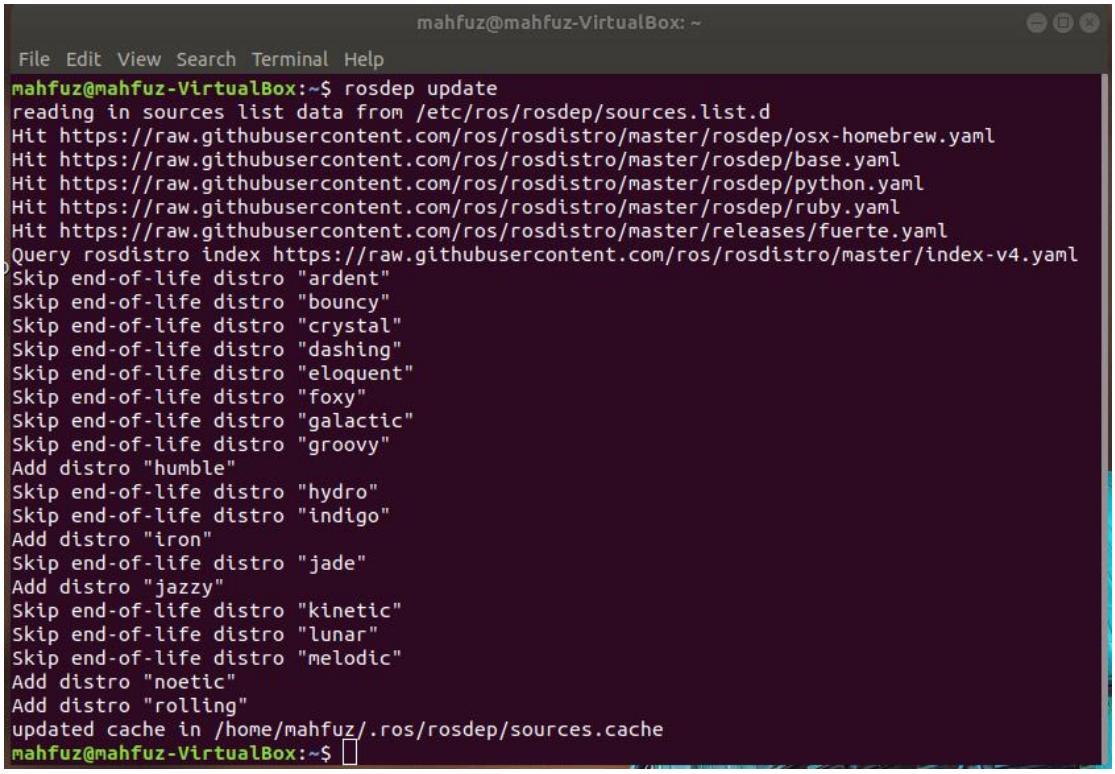
```
//  Fuzzy controller for going through the gap 2
void RobotMove::Fuzzy_to2stGap(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.52)  fuzzySensor1 = 1;
    else if (laserData1 < 0.82)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.5)   fuzzySensor2 = 1;
    else if (laserData2 < 0.7)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if(fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_HIGH);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_RIGHT_SPEED_LOW);
    else ROS_INFO("Going through the 2st gap");
}
```

Going to end point

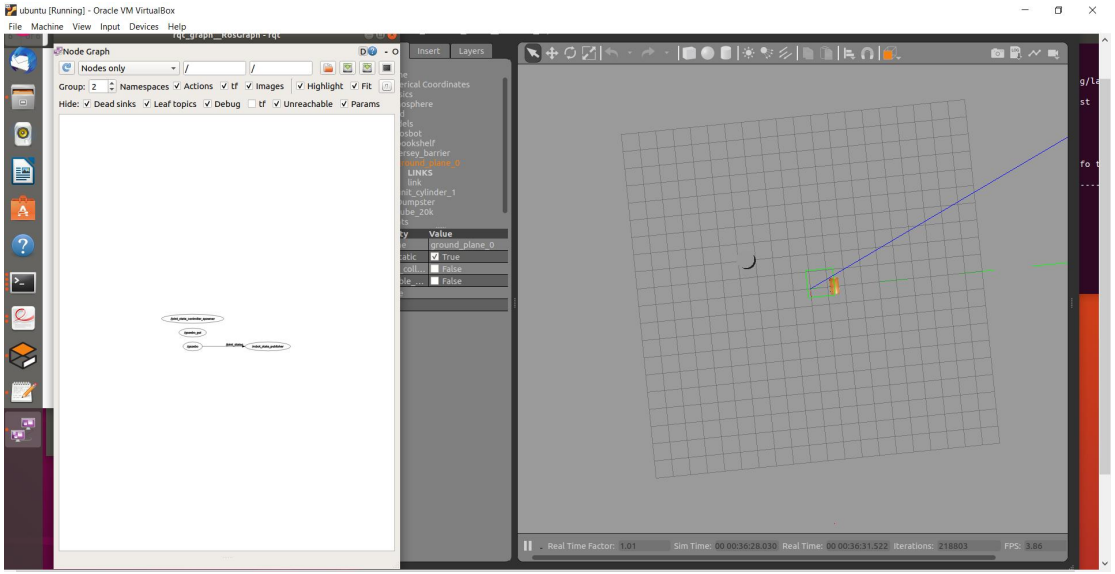
```
//  Fuzzy controller for going to the end point
void RobotMove::Fuzzy_toEnd(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.38)  fuzzySensor1 = 1;
    else if (laserData1 < 0.52)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.38)   fuzzySensor2 = 1;
    else if (laserData2 < 0.5)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if (fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_SPEED_ZERO);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_LEFT_SPEED_LOW);
    else ROS_INFO("Going through the End");
}

// add the following function
```

4.3 Overview Result of other Lab

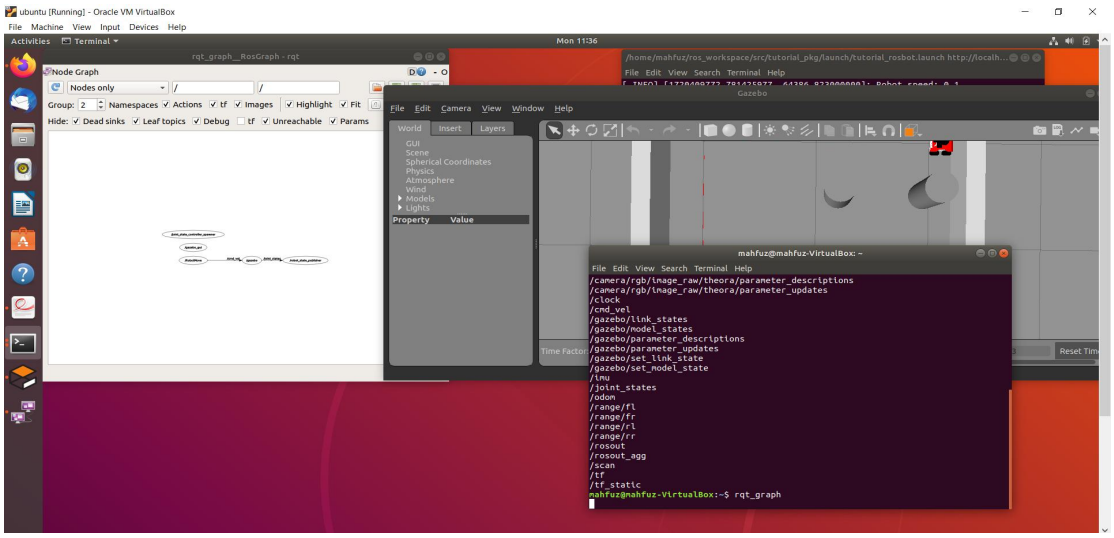
4.3.1 Lab1



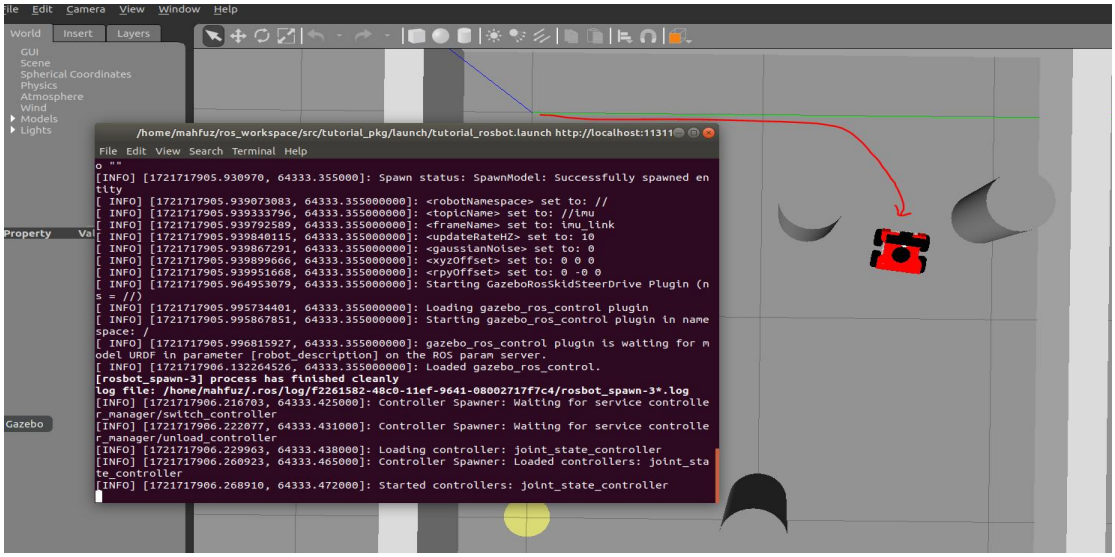
4.3.2 Lab2



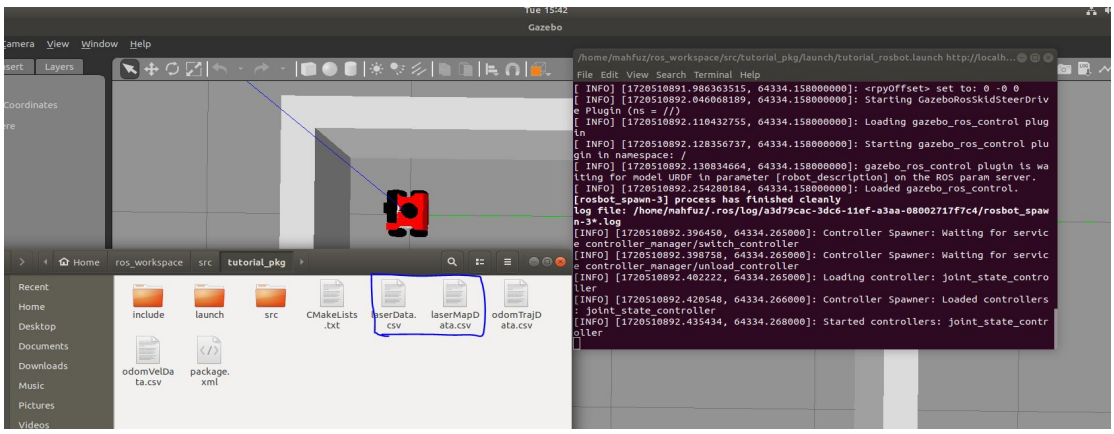
4.3.3 Lab3



4.3.4 Lab4



4.3.5 Lab5



4.3 Fuzzy and PID code

For all the code checkout my github.

4. 4. 1 PID code

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <chrono>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <nav_msgs/Odometry.h>
#include <sensor_msgs/LaserScan.h>

using namespace std;

struct EulerAngles {
    double roll, pitch, yaw; // yaw is what you want, i.e. Th
};

struct Quaternion {
    double w, x, y, z;
};

EulerAngles ToEulerAngles(Quaternion q) {
    EulerAngles angles;
    // roll (x-axis rotation)
    double sinr_cosp = +2.0 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = +1.0 - 2.0 * (q.x * q.x + q.y * q.y);
    angles.roll = atan2(sinr_cosp, cosr_cosp);
    // pitch (y-axis rotation)
    double sinp = +2.0 * (q.w * q.y - q.z * q.x);
    if (fabs(sinp) >= 1)
        angles.pitch = copysign(M_PI / 2, sinp); // use 90 degrees if out of range
    else
        angles.pitch = asin(sinp);
    // yaw (z-axis rotation)
    double siny_cosp = +2.0 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = +1.0 - 2.0 * (q.y * q.y + q.z * q.z);
    angles.yaw = atan2(siny_cosp, cosy_cosp);
```

```
    return angles;
}

class RobotMove {
public:
    RobotMove();
    void laserCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
    void odomCallback(const nav_msgs::Odometry::ConstPtr& odomMsg);
    void startMoving();
    void transformMapPoint(ofstream& fp, double laserRange, double laserTh);
    void PID_wallFollowing(double moveSpeed, double laserData);
    void PID_to1stGap(double moveSpeed, double robotHeading);
    void PID_to2ndGap(double moveSpeed, double robotHeading);
    void PID_reachGoal(double moveSpeed, double robotHeading);

private:
    ros::NodeHandle node;
    ros::Publisher commandPub;
    ros::Subscriber laserSub;
    ros::Subscriber odomSub;

    double frontRange, leftRange, rightRange;
    double mleftRange, mrightRange;
    double frontAngle, leftAngle, rightAngle;
    double mleftAngle, mrightAngle;

    Quaternion robotQuat;
    EulerAngles robotAngles;
    double robotHeadAngle;

    double PositionX = 0.0; // Initialize PositionX
    double PositionY = 0.0; // Initialize PositionY
    int stage = 1; // Initialize stage

    double laserData[360];

    double linearVelocity = 0.0;
    double angularVelocity = 0.0;

    // PID parameters for wall following
    double kp1 = 0.01, ki1 = 0.001, kd1 = 0.001, ei_pre1 = 0;
```

```
    double ed_pre1 = 0, landmark1_toWall = 0.3, Max_PID_output = 0.6;

    // PID parameters for 1st gap
    double kp2 = 0.01, ki2 = 0.001, kd2 = 0.001, ei_pre2 = 0;
    double ed_pre2 = 0, landmark2_heading = 0;

    // PID parameters for 2nd gap
    double kp3 = 0.01, ki3 = 0.001, kd3 = 0.001, ei_pre3 = 0;
    double ed_pre3 = 0, landmark3_heading = 0;

    // PID parameters for reaching goal
    double kp4 = 0.01, ki4 = 0.001, kd4 = 0.001, ei_pre4 = 0;
    double ed_pre4 = 0, landmark5_heading = 0;

    // Movement parameters
    const double FORWARD_SPEED_HIGH = 0.5;
    const double FORWARD_SPEED_MIDDLE = 0.3;
    const double FORWARD_SPEED_LOW = 0.1;

    // Landmark positions
    double landmark1 = 1.0;
    double landmark2 = 2.0;
    double landmark3 = 3.0;
    double landmark4 = 4.0;
    double landmark5 = 5.0;

    std::ofstream trajectoryFile;
    std::ofstream velocityFile;

    void moveStop();
    std::pair<double, double> getRobotVelocity();
    void logTrajectoryAndVelocity();
};

RobotMove::RobotMove() {
    commandPub = node.advertise<geometry_msgs::Twist>("/cmd_vel", 10);
    laserSub = node.subscribe("/scan", 10, &RobotMove::laserCallback, this);
    odomSub = node.subscribe("/odom", 20, &RobotMove::odomCallback, this);

    // Initialize angles
    frontAngle = 0;
```

```
    leftAngle = M_PI / 2;
    rightAngle = -M_PI / 2;
    mleftAngle = 3 * M_PI / 4;
    mrightAngle = -3 * M_PI / 4;
}

void RobotMove::laserCallback(const sensor_msgs::LaserScan::ConstPtr& scan) {
    frontRange = scan->ranges[0];
    leftRange = scan->ranges[90];
    rightRange = scan->ranges[270];
    mleftRange = scan->ranges[45];
    mrightRange = scan->ranges[315];

    for (int i = 0; i < 360; ++i) {
        laserData[i] = scan->ranges[i];
    }
}

void RobotMove::odomCallback(const nav_msgs::Odometry::ConstPtr& odomMsg) {
    robotQuat.x = odomMsg->pose.pose.orientation.x;
    robotQuat.y = odomMsg->pose.pose.orientation.y;
    robotQuat.z = odomMsg->pose.pose.orientation.z;
    robotQuat.w = odomMsg->pose.pose.orientation.w;

    robotAngles = ToEulerAngles(robotQuat);
    robotHeadAngle = robotAngles.yaw;

    PositionX = odomMsg->pose.pose.position.x;
    PositionY = odomMsg->pose.pose.position.y;

    // Store linear and angular velocities
    linearVelocity = odomMsg->twist.twist.linear.x;
    angularVelocity = odomMsg->twist.twist.angular.z;
}

void RobotMove::transformMapPoint(ofstream& fp, double laserRange, double laserTh)
{
    double localX, localY, globalX, globalY;
    localX = laserRange * cos(laserTh);
    localY = laserRange * sin(laserTh);
    globalX = (localX * cos(robotHeadAngle) - localY * sin(robotHeadAngle)) +
```

```
PositionX;
    globalY = (localX * sin(robotHeadAngle) + localY * cos(robotHeadAngle)) +
PositionY;

    if (globalX < 0) globalX = 0;
    else if (globalX > 2.5) globalX = 2.5;

    if (globalY < 0) globalY = 0;
    else if (globalY > 2.5) globalY = 2.5;

    fp << globalX << "," << globalY << endl;
}

void RobotMove::PID_wallFollowing(double moveSpeed, double laserData) {
    double ei, ed, err, output;
    err = landmark1_toWall - laserData;
    ei = ei_pre1 + err;
    ed = err - ed_pre1;
    ei_pre1 = ei;
    ed_pre1 = ed;
    output = kp1 * err + ki1 * ei + kd1 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}

void RobotMove::PID_to1stGap(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark2_heading - robotHeading;
    ei = ei_pre2 + err;
    ed = err - ed_pre2;
    ei_pre2 = ei;
    ed_pre2 = ed;
    output = kp2 * err + ki2 * ei + kd2 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
```

```
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}

void RobotMove::PID_to2ndGap(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark3_heading - robotHeading;
    ei = ei_pre3 + err;
    ed = err - ed_pre3;
    ei_pre3 = ei;
    ed_pre3 = ed;
    output = kp3 * err + ki3 * ei + kd3 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
    commandPub.publish(msg);
}

void RobotMove::PID_reachGoal(double moveSpeed, double robotHeading) {
    double ei, ed, err, output;
    err = landmark5_heading - robotHeading;
    ei = ei_pre4 + err;
    ed = err - ed_pre4;
    ei_pre4 = ei;
    ed_pre4 = ed;
    output = kp4 * err + ki4 * ei + kd4 * ed;
    if (output > Max_PID_output)
        output = Max_PID_output;
    else if (output < -Max_PID_output)
        output = -Max_PID_output;
    geometry_msgs::Twist msg;
    msg.linear.x = moveSpeed;
    msg.angular.z = output;
```

```
    commandPub.publish(msg);
}

void RobotMove::startMoving() {
    std::ofstream
laserFile("/home/mahfuz/ros_workspace/src/tutorial_pkg/laserData.csv");
    std::ofstream
laserMapFile("/home/mahfuz/ros_workspace/src/tutorial_pkg/laserMapData.csv");
    
trajectoryFile.open("/home/mahfuz/ros_workspace/src/tutorial_pkg/trajectory.csv");
    velocityFile.open("/home/mahfuz/ros_workspace/src/tutorial_pkg/velocity.csv");

    if (!laserFile.is_open() || !laserMapFile.is_open() || !trajectoryFile.is_open()
|| !velocityFile.is_open()) {
        ROS_ERROR("Failed to open CSV files for writing");
        return;
    }

    ros::Rate rate(20); // Define rate for repeatable operations.
    ROS_INFO("Start moving");

    while (ros::ok()) { // keep spinning loop until user presses Ctrl+C
        switch (stage) {
            case 1: // wall following
                if (PositionY < landmark1) {
                    PID_wallFollowing(FORWARD_SPEED_HIGH, leftRange);
                } else {
                    stage = 2;
                }
                break;
            case 2: // move toward the middle of the 1st gap
                if (PositionX < landmark2) {
                    PID_to1stGap(FORWARD_SPEED_MIDDLE, robotHeadAngle);
                } else {
                    stage = 3;
                }
                break;
            case 3: // move toward the 2nd gap
                if (PositionX < landmark3) {
                    PID_to2ndGap(FORWARD_SPEED_MIDDLE, robotHeadAngle);
                } else {
```

```
                stage = 4;
            }
            break;
        case 4: // go through the 2nd gap
            if (PositionX < landmark4) {
                PID_to2ndGap(FORWARD_SPEED_LOW, robotHeadAngle);
            } else {
                stage = 5;
            }
            break;
        case 5: // move towards the charger
            if (PositionX < landmark5) {
                PID_reachGoal(FORWARD_SPEED_LOW, robotHeadAngle);
            } else {
                stage = 6;
            }
            break;
        case 6:
            moveStop();
            break;
    }

    for (int n = 0; n < 360; ++n) { // save laser data for view and check
        laserFile << n << " " << laserData[n] << std::endl;
    }

    transformMapPoint(laserMapFile, frontRange, frontAngle);
    transformMapPoint(laserMapFile, leftRange, leftAngle);
    transformMapPoint(laserMapFile, rightRange, rightAngle);
    transformMapPoint(laserMapFile, mleftRange, mleftAngle);
    transformMapPoint(laserMapFile, mrightRange, mrightAngle);

    // Log trajectory and velocity
    logTrajectoryAndVelocity();

    ros::spinOnce();
    rate.sleep();
  }

  laserFile.close();
  laserMapFile.close();
```

```
    trajectoryFile.close();
    velocityFile.close();
}

void RobotMove::moveStop() {
    // Define the logic to stop the robot
    geometry_msgs::Twist msg;
    msg.linear.x = 0;
    msg.angular.z = 0;
    commandPub.publish(msg);
}

std::pair<double, double> RobotMove::getRobotVelocity() {
    return {linearVelocity, angularVelocity};
}

void RobotMove::logTrajectoryAndVelocity() {
    trajectoryFile << PositionX << ", " << PositionY << std::endl;
    auto [linearVel, angularVel] = getRobotVelocity();
    velocityFile << linearVel << ", " << angularVel << std::endl;
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "tutorial_pkg_node");
    RobotMove move;
    move.startMoving();
    return 0;
}
```

4.4.2 Fuzzy Code

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include "sensor_msgs/LaserScan.h"
#include <fstream>
#include <time.h>
#include <iomanip>
#include <iostream>
#include <cmath>
#include <chrono>
using namespace std;
```

```
using namespace std::chrono;

struct EulerAngles{
    double roll, pitch, yaw;};          // yaw is what you want, i.e. Th

struct Quaternion{double w, x, y, z;};

EulerAngles ToEulerAngles(Quaternion q){
    EulerAngles angles;
    // roll (x-axis rotation)
    double sinr_cosp = +2.0 * (q.w * q.x + q.y * q.z);
    double cosr_cosp = +1.0 - 2.0 * (q.x * q.x + q.y * q.y);
    angles.roll = atan2(sinr_cosp, cosr_cosp);
    // pitch (y-axis rotation)
    double sinp = +2.0 * (q.w * q.y - q.z * q.x);
    if (fabs(sinp) >= 1)
        angles.pitch = copysign(M_PI/2, sinp);  //use 90 degrees if out of range
    else
        angles.pitch = asin(sinp);
    // yaw (z-axis rotation)
    double siny_cosp = +2.0 * (q.w * q.z + q.x * q.y);
    double cosy_cosp = +1.0 - 2.0 * (q.y * q.y + q.z * q.z);
    angles.yaw = atan2(siny_cosp, cosy_cosp);

    return angles;
}

class RobotMove { //main class
    public:
        // Tunable parameters
        constexpr const static double FORWARD_SPEED_LOW = 0.1;
        constexpr const static double FORWARD_SPEED_MIDDLE = 0.3;
        constexpr const static double FORWARD_SPEED_HIGH = 0.5;
        constexpr const static double FORWARD_SPEED_STOP = 0;
        constexpr const static double TURN_LEFT_SPEED_HIGH = 1.0;
        constexpr const static double TURN_LEFT_SPEED_MIDDLE = 0.8;
        constexpr const static double TURN_LEFT_SPEED_LOW = 0.6;
        constexpr const static double TURN_RIGHT_SPEED_HIGH = -1.0;
        constexpr const static double TURN_RIGHT_SPEED_LOW = -0.3;
        constexpr const static double TURN_RIGHT_SPEED_MIDDLE = -0.6;
        constexpr const static double TURN_SPEED_ZERO = 0;
```

```
        RobotMove();
        void startMoving();
        void moveForward(double forwardSpeed);
        void moveStop();
        void moveRight(double turn_right_speed = TURN_RIGHT_SPEED_HIGH);
        void moveForwardRight(double forwardSpeed, double turn_right_speed);
        void odomCallback(const nav_msgs::Odometry::ConstPtr& odomMsg);
        void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
        void transformMapPoint(ofstream& fp, double laserRange, double laserTh);
        void Fuzzy_wallFollowing(double Data1,double Data2);
        void Fuzzy_to1stGap(double laserData1, double laserData2);
        void Fuzzy_toTrajectory(double laserData1, double laserData2);
        void Fuzzy_to2stGap(double laserData1, double laserData2);
        void Fuzzy_toEnd(double laserData1, double laserData2);
    private:
        ros::Time current_time;
        ros::Duration real_time;
        ros::NodeHandle node;
        ros::Publisher commandPub;          // Publisher to the robot's velocity
command topic
        ros::Subscriber odomSub;           //Subscriber to robot's odometry topic
        ros::Subscriber laserSub;          // Subscriber to robot's laser topic

        Quaternion robotQuat;
        EulerAngles robotAngles;
        double robotHeadAngle;

        //  parameter for landmark and robot position
        double PositionX = 0.3, PositionY = 0.3, landmark1 = 1.15, landmark2 = 0.9;
        double homeX = 0.3, homeY = 0.3;
        double landmark3 = 1.4, landmark4 = 1.88, landmark5 = 0.35;
        double robVelocity = 0;
        //  parameter for laser
        double frontRange=0,mleftRange=0,leftRange=0,rightRange=0,mrightRange=0;
        double backRange=0, backleftRange=0, backrightRange=0, laserData[36];
        double frontAngle=0, mleftAngle=M_PI/4, leftAngle=M_PI/2;
        double rightAngle=-M_PI/2, mrightAngle=-M_PI/4;
        double backAngle=M_PI, backleftAngle=3*M_PI/4, backrightAngle=-3*M_PI/4;

        double Max_Fuzzy_output = 0.6;
```

```
};

void RobotMove::transformMapPoint(ofstream& fp, double laserRange, double
laserTh){
    double localX, localY, globalX, globalY;
    localX = laserRange * cos(laserTh);
    localY = laserRange * sin(laserTh);
    globalX =(localX*cos(robotHeadAngle)-localY*sin(robotHeadAngle))+ PositionX;
    globalY = (localX*sin(robotHeadAngle)+localY*cos(robotHeadAngle))+ PositionY;
    if (globalX < 0) globalX = 0; else if (globalX > 2.5) globalX = 2.5;
    if (globalY < 0) globalY = 0; else if (globalY > 2.5) globalY = 2.5;
    fp << globalX << " " << globalY << endl;
}

RobotMove::RobotMove(){
    //Advertise a new publisher for the simulated robot's velocity command topic at
10Hz
    commandPub = node.advertise<geometry_msgs::Twist>("cmd_vel",  10);
    // subscribe to the odom topic
    odomSub = node.subscribe("odom",  20,  &RobotMove::odomCallback,  this);
    laserSub = node.subscribe("scan", 1, &RobotMove::scanCallback, this);
}

//send a velocity command
void RobotMove::moveForward(double forwardSpeed){
    geometry_msgs::Twist msg;          //The default constructor to set all commands
to 0
    msg.linear.x = forwardSpeed;           //Drive forward at a given speed along the
x-axis.
    commandPub.publish(msg);
}

void RobotMove::moveStop(){
    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_STOP;
    commandPub.publish(msg);
}

void RobotMove::moveRight(double turn_right_speed){
    geometry_msgs::Twist msg;
    commandPub.publish(msg);
```

```
}

void RobotMove::moveForwardRight(double forwardSpeed,  double turn_right_speed){
    //move forward and right at the same time
    geometry_msgs::Twist msg;
    msg.linear.x = forwardSpeed;
    msg.angular.z = turn_right_speed;
    commandPub.publish(msg);
}

// add the callback function to determine the robot position.
void RobotMove::odomCallback(const nav_msgs::Odometry::ConstPtr& odomMsg){
    PositionX = odomMsg->pose.pose.position.x + homeX;
    PositionY = odomMsg->pose.pose.position.y + homeY;
    robVelocity = odomMsg->twist.twist.linear.x;
    robotAngles = ToEulerAngles(robotQuat);
    robotHeadAngle = robotAngles.yaw;
    robotQuat.x = odomMsg->pose.pose.orientation.x;
    robotQuat.y = odomMsg->pose.pose.orientation.y;
    robotQuat.z = odomMsg->pose.pose.orientation.z;
    robotQuat.w = odomMsg->pose.pose.orientation.w;
}

void RobotMove::scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan){
    // collect 36 laser readings every 360 degrees scan
    for(int i=0; i<36; i++) // to get 36 laser readings over 360 degrees
        laserData[i] = scan->ranges[i*10]; // to get laser readings every 10 degrees
    // the following code for the control purpose
    frontRange = scan->ranges[0]; // get the range reading at 0 radians
    mleftRange = scan->ranges[89]; // get the range reading at -π/4 radians
    leftRange = scan->ranges[179]; // get the range reading at -π/2 radians
    rightRange = scan->ranges[539]; // get the range reading at π/2 radians
    mrightRange = scan->ranges[629]; // get the range reading at π/4 radians
    backRange = scan->ranges[359]; // get the range reading at π radians
    backleftRange = scan->ranges[269]; // get the range reading at π/2 radians
    backrightRange = scan->ranges[449]; // get the range reading at π/4 radians
}

ofstream openFile(const string& name){          // open files for data storage
    string homedir = getenv("HOME");
    ostringstream path;
```

```
    ofstream file;

    //  change the path here
    //  ex:  if your pkg path is
/home/wang/Ros/workspace/ai_course/src/tutorial_pkg/
    //          hrer only need /ROS/workspace/ai_course/src/tutorial_pkg/
    path << homedir <<  "/ros_workspace/src/tutorial_pkg/" << name;
    ROS_INFO("File address: %s", path.str().c_str());
    file.open(path.str().c_str());
    if (file.is_open())      ROS_INFO("File opened");

    return file;
}

//  Fuzzy controller for moving along the wall
void RobotMove::Fuzzy_wallFollowing(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (laserData1 < 0.2)   fuzzySensor1 = 1;       // The robot is near to the wall
    else if (laserData1 < 0.5)  fuzzySensor1 = 2;       // The robot is on the right
distance
    else    fuzzySensor1 = 3;       // The robot is far from the wall;

    if (laserData2 < 0.3)   fuzzySensor2 = 1;   // The robot is near to the wall
    else if (laserData2 < 0.6)  fuzzySensor2 = 2;   // The robot at the right distance;
    else    fuzzySensor2 = 3;   // The robot is far from the wall;

    // Fuzzy rule base and control output
    if (fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_SPEED_ZERO);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_LOW);
```

```
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_LEFT_SPEED_LOW);
    else ROS_INFO("Following the left wall");
}

//  Fuzzy controller for going through the gap 1
void RobotMove::Fuzzy_to1stGap(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.2)  fuzzySensor1 = 1;
    else if (laserData1 < 0.5)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.2)   fuzzySensor2 = 1;
    else if (laserData2 < 0.9)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if(fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_HIGH);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_RIGHT_SPEED_LOW);
```

```
    else ROS_INFO("Going through the 1st gap");
}

//  Fuzzy controller for going through the Relay point
void RobotMove::Fuzzy_toTrajectory(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.7)  fuzzySensor1 = 1;
    else if (laserData1 < 1)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.7)   fuzzySensor2 = 1;
    else if (laserData2 < 1)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if(fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_HIGH);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_RIGHT_SPEED_LOW);
    else ROS_INFO("Going through the Trajectory");
}

//  Fuzzy controller for going through the gap 2
void RobotMove::Fuzzy_to2stGap(double laserData1, double laserData2)
{
```

```
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.52)  fuzzySensor1 = 1;
    else if (laserData1 < 0.82)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;

    if (laserData2 < 0.5)   fuzzySensor2 = 1;
    else if (laserData2 < 0.7)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if(fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_HIGH);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_MIDDLE);
    else if(fuzzySensor1 = 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_RIGHT_SPEED_LOW);
    else ROS_INFO("Going through the 2st gap");
}

//  Fuzzy controller for going to the end point
void RobotMove::Fuzzy_toEnd(double laserData1, double laserData2)
{
    int fuzzySensor1, fuzzySensor2;
    // sensor data fuzzification
    if (0 <= laserData1 < 0.38)  fuzzySensor1 = 1;
    else if (laserData1 < 0.52)  fuzzySensor1 = 2;
    else    fuzzySensor1 = 3;
```

```
    if (laserData2 < 0.38)   fuzzySensor2 = 1;
    else if (laserData2 < 0.5)  fuzzySensor2 = 2;
    else    fuzzySensor2 = 3;

    // Fuzzy rule base and control output
    if (fuzzySensor1 == 1 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 1 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_LOW, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_LOW);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_SPEED_ZERO);
    else if(fuzzySensor1 == 2 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_LEFT_SPEED_LOW);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 1)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 2)
        moveForwardRight(FORWARD_SPEED_MIDDLE, TURN_RIGHT_SPEED_MIDDLE);
    else if(fuzzySensor1 == 3 && fuzzySensor2 == 3)
        moveForwardRight(FORWARD_SPEED_HIGH, TURN_LEFT_SPEED_LOW);
    else ROS_INFO("Going through the End");
}

// add the following function
void RobotMove::startMoving(){
    auto currentTime = high_resolution_clock::now();
    auto startTime = currentTime;
    ofstream odomVelFile = openFile("fuzzyVelData.csv");
    ofstream odomTrajFile = openFile("fuzzyTrajData.csv");
    ofstream laserFile = openFile("fuzzylaserData.csv");
    ofstream laserMapFile = openFile("fuzzylaserMapData.csv");

    int stage = 1;
    ros::Rate rate(20);        //Define rate for repeatable operations.
    ROS_INFO("Start moving");
    while (ros::ok()){         // keep spinning loop until user presses Ctrl+C
        switch(stage){
            case 1:        // the robot move forward from home
```

```
            if (PositionY < landmark1)
                Fuzzy_wallFollowing(leftRange, mleftRange);
            else stage = 2;
            break;
        case 2:        // the robot turns right toward the 1st gap
            if (PositionX < landmark2)
                Fuzzy_to1stGap(leftRange, mleftRange);
            else stage = 3;
            break;
        case 3:        // the robot moves forward fast
            if (PositionX < landmark3)
                Fuzzy_toTrajectory(leftRange, mleftRange);
            else stage = 4;
            break;
        case 4:        // the robot moves and turns right slowly
            if (PositionX < landmark4)
                Fuzzy_to2stGap(leftRange, mleftRange);
            else stage = 5;
            break;
        case 5:        // the robot moves towards the charger
            if (PositionY > landmark5)
                Fuzzy_toEnd(leftRange, mleftRange);
            else stage = 6;
            break;
        case 6:        // stop at the charger position
            moveStop();
            stage = 7;  //  used to exit the while
            break;
      }
      auto currentTime = high_resolution_clock::now();
      duration<double,std::deca> runTime = currentTime  - startTime;
      runTime = runTime * 10;            // convert time to seconds
      odomVelFile << ceil(runTime.count()) << " " << robVelocity << endl;
      odomTrajFile << PositionX << " " << PositionY << endl;

      for(int i=0; i<36; i++) // save laser data for view and check
         laserFile << i << " "<< laserData[i]<<" ";
      laserFile << endl;

      transformMapPoint(laserMapFile, frontRange, frontAngle);
      transformMapPoint(laserMapFile, leftRange, leftAngle);
```

```
        transformMapPoint(laserMapFile, rightRange, rightAngle);
        transformMapPoint(laserMapFile, mleftRange, mleftAngle);
        transformMapPoint(laserMapFile, mrightRange, mrightAngle);

        ros::spinOnce();               // Allow ROS to process incoming messages
        rate.sleep();              // Wait until defined time passes.
        ROS_INFO("\nCurrent stage:%d, Robot at [X: %0.2f, Y: %0.2f, Theta: %0.2f],
Speed: %0.2f", stage, PositionX, PositionY, robotHeadAngle, robVelocity);
        if (stage == 7)
            break;
    }
    //  save all files
    odomTrajFile.close();
    odomVelFile.close();
    laserFile.close();
    laserMapFile.close();
    ROS_INFO("Finish ALL");
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "RobotMove");        //Initiate new ROS node named
"RobotMove"
    RobotMove RobotMove;           // Create new RobotMove object
    RobotMove.startMoving();           // Start the movement

    return 0;
}
```