ABSTRACT

This project shows the work on designing an intelligent autonomous mobile robot for service or rescuing mission in the indoor environment.

Mobility refers to the performance of the robot relative to the field, ability to negotiate and avoid obstacles, ability to overcome rough terrain and crossing streams.

This thesis illustrates based on a practical example how maps that result from simultaneous localization and mapping (SLAM) methods can be used for path planning and path optimization. Special attention will be paid to RRT* path planning algorithm which is used to create a feasible and optimized trajectory for the mobile robot. A Self-tuning Fuzzy PID Controller also is introduced for driving the robot follow the trajectory precisely.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, we would like to thank our family and friends for their support. A special thank to our advisor, Dr. Mac Thi Thoa, for her guidance through each stage of our thesis. Her guidance and advice shaped my career and help me to complete my graduation thesis at Hanoi University of Science and Technology.

ABBREVIATION

ROS	Robot Operation System
SLAM	Simultaneous Localization and Mapping
RRT	Rapidly-Exploring Random Tree
URDF	Unified Robot Description Format
OGM	Occupancy grid map
PGM	Portable Gray Map
XML	Extensible Markup Language
IMU	Inertial Measurement Units
PGM	Portable Graymap Format
DWA	Dynamic Window Approach
AMCL	Adaptive Monte Carlo Localization

CHAPTER 1 INTRODUCTION

1.1 Historical development of rescues Robots

A rescue robot is a robot that has been designed for purpose of rescuing people. Common situations that employ rescue robots are mining accidents, urban disasters, hostage situations, and explosions. The benefits of rescue robots to these operations include reduced personnel requirements, reduced fatigue, and access to otherwise unreachable areas.

Rescue robots in development are being made with abilities such as searching, reconnaissance and mapping, removing rubble and delivery of supplies, medical treatment. Even though, there are still some technical challenges that remain.

There are three main levels of challenges. First, the information processing of the robot. Second, the mobility of the robot. Third, the manipulation of the robot.

Rescue robots were used in the search for victims and survivors after the September 11 attacks in New York. During September 11 disasters rescue robots were first really tested. The robots had trouble working in the rubble of the World Trade Center and were constantly getting stuck or broken. Since then many new ideas have been formed about rescue robots.

Figure 1-1: Recuse mobile robots

1.2 The task of Thesis

The main request to a mobile robot obviously is mobility. The robot has to be capable of moving from its current pose, defined by its position and orientation, to a desired target configuration. In order to achieve this, it needs to find a valid collision-free path connecting the corresponding configurations. More over there may be other demands to the path like to be as short, as fast or as safe as possible.

1.2.1 Objective

Our system consist of some advance technology method includes:

- Using Simultaneous localization and mapping (SLAM) to define the surrounding environment and locating the pose of Robot.
- Using RRT* path planning algorithm to find a feasible and optimal path for the mobile robot
- Implementing Fuzzy-PID controller to control the Robot follow the planned path.

In this thesis, we had covered some tasks:

- Designing the mechanical and controller for the Mobile robot.
- Understanding and analyzing the communication techniques between "Robot Operating System" and the real robot system.
- Understanding the dynamics of the movements of the mobile robot via simulation.
- Understand how sensors such as 3D Depth camera, Laser scanner, encoders work.
- Optimize the path planning algorithm and embedded it into ROS and real robot for informing the mobile robot to reach the target autonomously.
- From experiment, choose the most suitable controller for applying in system to get the precise motion based on the path already planned.
- Applying the decided controller in to the system, use an communication method connecting between Arduino and Raspberry to get motor's control signal

1.2.2 Methods

Robot Operating System (ROS)

At the beginning, we installed ROS to be able to program the mobile robot. ROS is running on Linux machines and it supports C++, python and a few more programming languages. Along with the installation of ROS we also had the Gazebo simulator. Then, we investigated ROS in detail, its features and utilities. After, we started to work on Gazebo for simulation before testing real experiment on the real mobile robot.

1.3 Introduction to ROS

ROS is an open-source, meta-operating system for the robot to help software developers produce robot programs. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

The goal of ROS is to "build the development environment that allows robotic software development to collaborate on a global level. ROS is focused on maximizing code reuse in the robotics research and development.

1.3.1 ROS Ecosystem

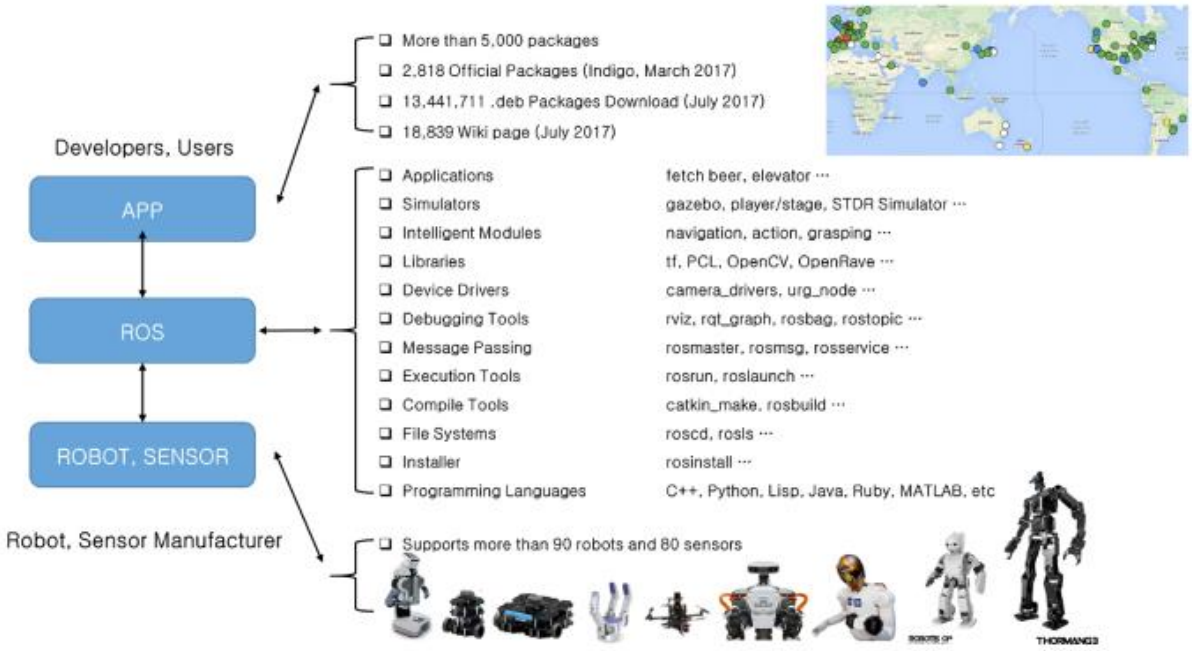


Figure 1-2: ROS Eco system

1.3.2 Message Communication

ROS data communication is supported not only by one operating system, but also by multiple operating systems, hardware, and programs, making it highly suitable for robot development where various hardware are combined.

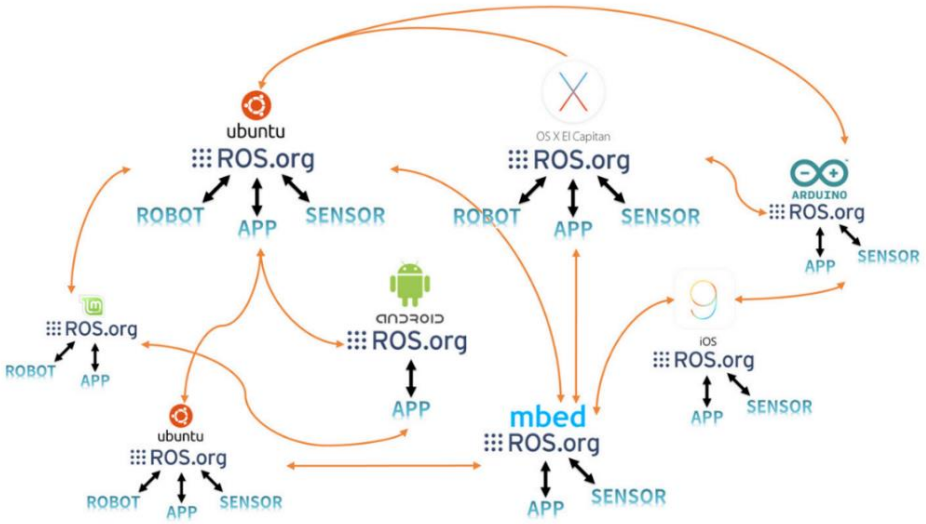


Figure 1-3: ROS Multi-Communication

ROS is developed in unit of nodes, which is the minimum unit of executable program that has broken down for the maximum reusability. The node exchanges data with other nodes through messages forming a large program. There are three different methods of exchanging messages: a topic which provides a unidirectional message transmission/reception, a service which provides a bidirectional message request/response and an action which provides a bidirectional message goal/result/feedback. In addition, the parameters used in the node can be modified from the outside of node. This can also be considered as a type of message communication in the larger context.

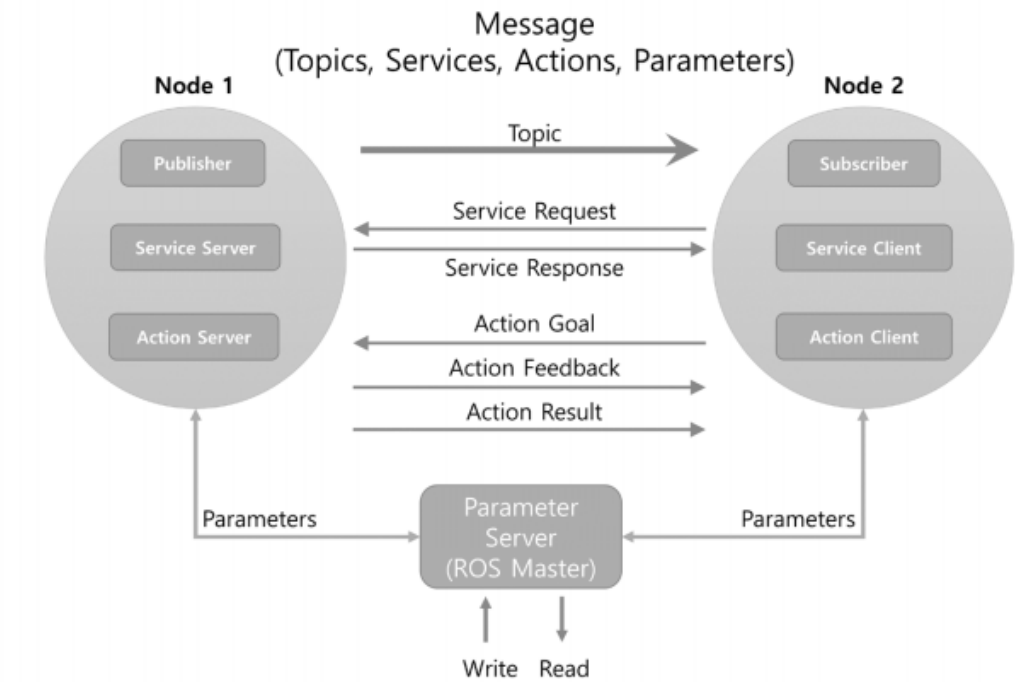


Figure 1-4: Message Communication between Nodes

Table 1.1: Comparison of the Topic, Server, and Action

Type	Features		Description
Topic	Asynchronous	Unidirectional	Used when exchanging data continuously
Service	Synchronous	Bi-directional	Used when request processing requests and responds current states
Action	Asynchronous	Bi-directional	Used when it is difcult to use the service due to long response times after the request or when an intermediate feedback value is needed

1.3.3 Requirement

Ubuntu is a popular Linux distribution. It is freely available for use, and it is open source, so it can be modified according to applications. It supports architectures such as Intel x86, AMD-64, ARMv7, and ARMv8 (ARM64).

A robot application can be run on an operating system that provide functionalities to communicate with robot actuators and sensors. A Linux-based operating system can provide great flexibility to interact with low-level hardware and customize the operating system according to the robot application.

The advantages of Ubuntu in this context are its responsiveness, lightweight and high degree of security. Beyond these factors, Ubuntu has great community support. These factors have led the ROS developers to stick to Ubuntu, and it is the only operating that is fully supported by ROS.

CHAPTER 2 MECHANICAL DESIGN AND CALCULATION

2.1 Mobile Robot design scheme selection

In this project we will use the Navigation Meta-package of ROS for navigating the mobile robot. There are three main hardware requirements that restrict its use:

- It is meant for both differential drive and holonomic wheeled robots only. The mobile base is controlled by sending desired velocity commands to achieve in the form of: x velocity, y velocity, theta velocity.
- It requires a planar laser mounted somewhere on the mobile base. This laser is used for map building and localization.
- The Navigation Meta-package's performance will be best on robots that are nearly square or circular.

Composition and specifications of the mobile robot

- The chassis,
- Parts: motors with encoders...
- The control: control Circuits, sensors, computers, camera ...
- Connector: wire connection and wireless connector
- Power source: battery, electric.

Specifications of the mobile robot

- Size: 220x220x155 mm
- Vehicle weight: 2.2 kg
- Sensor: LIDAR, 3D Depth camera, encoders
- Maximum speed: 1.5 m/s

2.2: Designing the mechanical part of mobile robot

2.2.1: Chassis

Chassis is for fixing the entire drivetrain, mounted sensors, the control circuit boards, batteries. The chassis must be designed compact enough stiffness to withstand the load above and the proportion of goods, in accordance with the conditions of the narrow working environment.

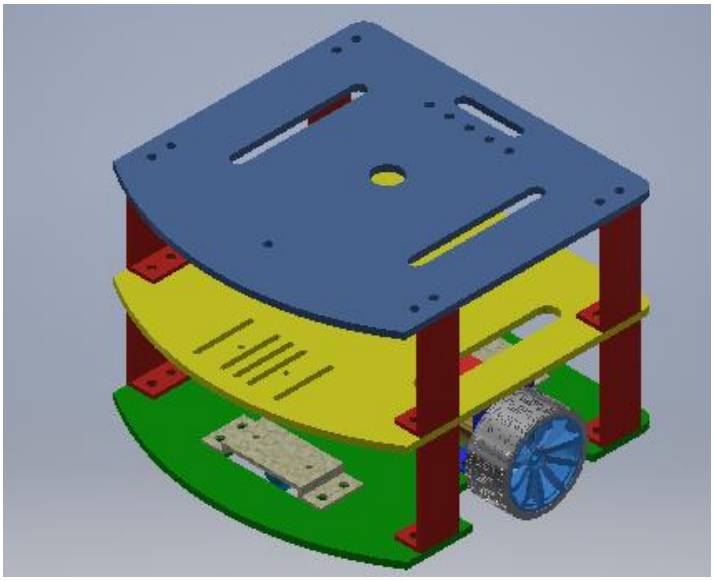


Figure 2-1: CAD model

Figure 2-2: Real chassis after manufacturing and assembly

2.2.2: Wheels

Wheels make the entire vehicle structure can move smoothly. They also support the structural parts of the car such as transmission parts and rotation. It includes driving wheels and casters.

Figure 2-3: The CAD design and the real driving wheels

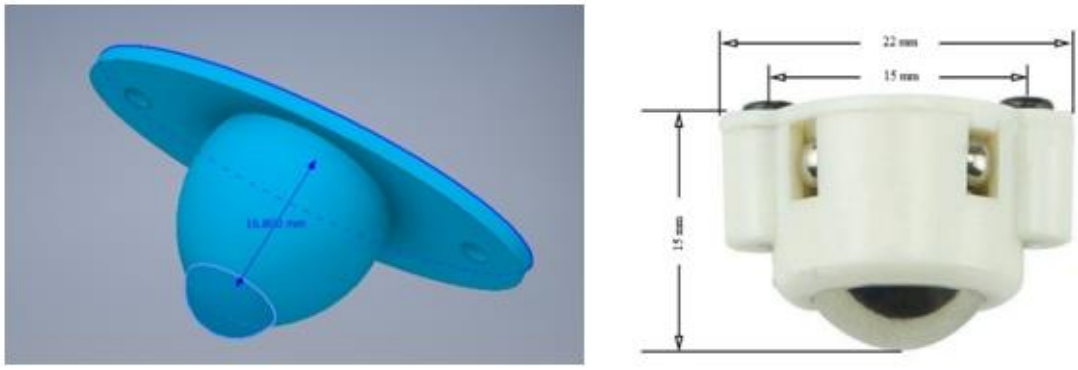


Figure 2-4: Caster wheels in CAD design and market available

2.2.3: Calculate the main engine for the selected transmission system.

For the structure of the vehicle moving, static resistance depends on mass of the vehicle, state road (curved, straight, potholes, ramps ...). Therefore, resistance force calculated according to the following formula:

$$F = M \times g \times \mu_{ms}\ (N) \tag{2.1}$$

Where:

M: the total mass (kg)

g: gravity acceleration: $9.81(m/s^2)$

μ_{ms}: Rotational friction coefficient between the wheel and the road (0.010 – 0.015)

To start moving, the robot has to win against the drag motion. Requirement torque that is generated by the engine to win the drag motion is calculated by:

$$M = \frac{F \times R_b}{i \times \eta}\ (Nm) \tag{2.2}$$

Where:

F: The resistance force (N)

i: The transmission ratio

ɳ: The efficiency of the structure.

R_b: The radius of the wheel (m)

The power of the engine when moving loads is calculated by:

$$P = \frac{F \times v}{2}(W) \quad (2.3)$$

Table 2.1: Data from the real mobile robot

M	the total mass (kg)	2.2
i	the transmission ratio	1
R_b	the radius of the wheel (m)	0.031
v	maximum velocity of the mobile robot (m/s)	1.5
ɳ	the efficiency of the structure	0.9

Therefore, the resistance force is:

$$F = M \times g \times \mu_{ms} \quad (2.4)$$

$$F = 2.2 \times 9.81 \times 0.015 = 0.32(N)$$

Torque generated by the engine to win the drag motion by:

$$M = \frac{F \times R_b}{i \times ɳ} = \frac{32.47 \times 0.031}{1 \times 0.9} = 1.11(Nm)$$

The power of the engine when moving loads by:

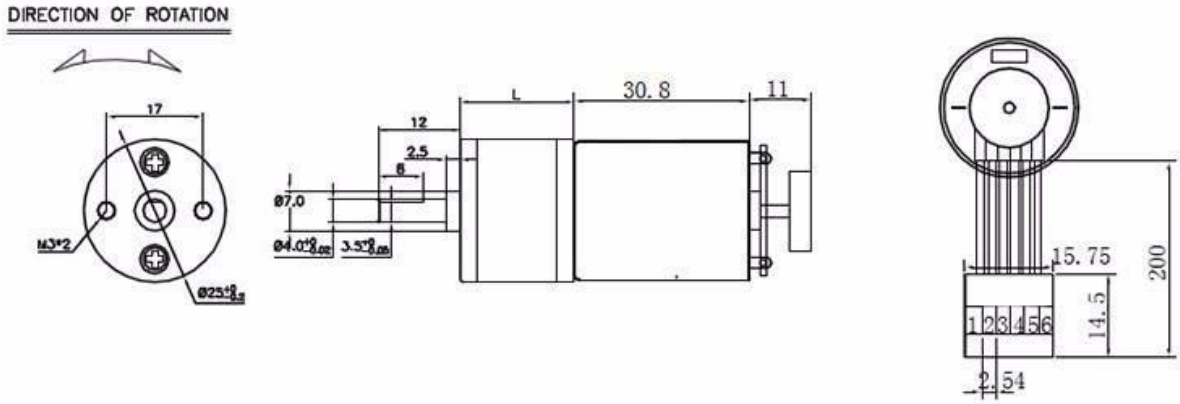
$$P = \frac{F.v}{2} = \frac{0.32 \times 1.5}{2} = 0.24(W)$$

The selection of the engine is important in engineering, reliability in the process of work and the economics of the robot. Electric DC motors for the mobile robot must meet the following requirements:

- The engine has enough power to pull.
- Speed matching and meets required speed adjustment, in the process of working to change the speed.
- Satisfying requirements for acceleration and braking speeds.
- Fit electrical energy source used (type of current, supply voltage).
- Suitable for working conditions.
- Price matching while ensuring set requirements

The power of the engine is selected based on the required parameters. To ensure capacity for the mobile robot we've chosen engines with the model JGA25-371 DC Gearmotor with Encoder.

Figure 2-5: The DC motor



Parameter:

Operating voltage: Between 6 V and 24 V DC

Nominal voltage:12 V DC

Operating temperature:-40℃~120℃

Connection of the encoder:

1.Red :Motor+

2.Black :Motor-

3.Green :Hall Sensor GND

4.Blue :Hall Sensor Vcc

5.Yellow:Hall Sensor A Vout

6.White: Hall Sensor B Vout

Figure 2-6: Motor's parameter

Table 2.2: JGA25-371 DC Gearmotor with Encoder

Operating voltage	6-24 VDC
Rated voltage	12 VDC
Power	1.25 W
Encoder voltage	5 VDC
Free load run speed at 12 V	126 RPM
Free-run current at 12 V	46 mA
Stall current at 12 V	1 A
Stall torque at 12 V	4.2 kg.cm
Gear ratio	1:34
Weight	99 g

2.3 The components in the control system.

Our mobile robot will be controlled by microprocessor connected to the computer: this method has advantages such as low cost, lightweight, observation system by the computer screen and can be programmed on demand easily.

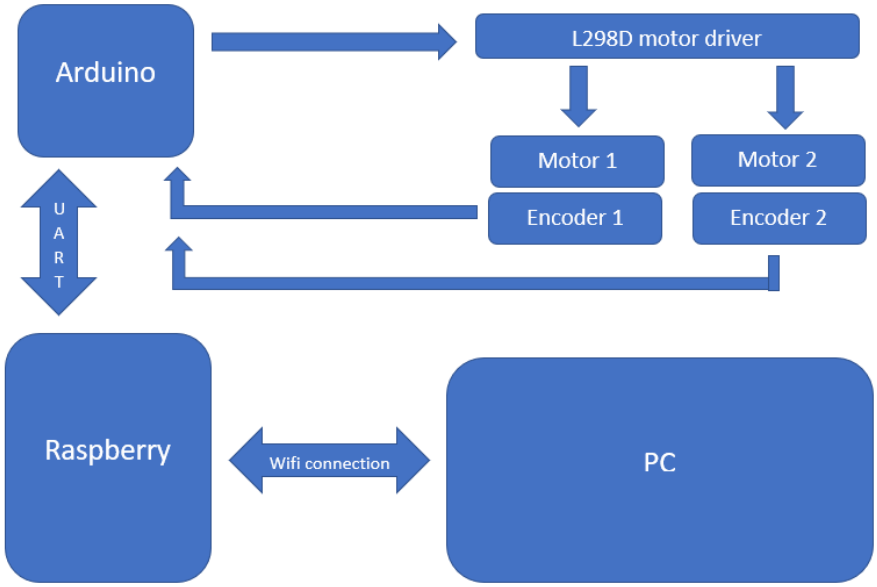


Figure 2-7: Block diagram of control system

The two motors are connected to an L-298 H-bridge. The main connections are the enable pin and two input pins. The enable pin activates the current H-bridge and two IN pins determine the motor's rotation direction. There is a total of six pins controlling the two motors. The Arduino sends the proper signals to these pins to control motor movement. There are three pins in wheel encoders: VCC, GND, and output. VCC and GND are connected to the Arduino VCC and GND, and the output of both encoders can be connected to the Arduino pins.

Arduino

Arduino is a microprocessor circuit board, to build applications that interact with each other or the environment was more favorable. The hardware includes an open-source circuit board is designed on the basis of atmel AVR 8 bit microprocessor, or 32-bit Atmel ARM.

Figure 2-8: Arduino Mega 2560 R3

Table 2.3: Arduino Mega Specification

Microcontroller	Atmega2560
Operating voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Pin (Digital I/O)	54 feet of which 15 pins PWM
Pin (analog)	16
Field DC input / output	20mA
3.3V DC pin	50mA
SRAM	8KB
EEPROM	4KB
Heartbeat	16MHx
Length	101.52 mm
Width	53.3 mm
Weight	37 g

In the Arduino programming language, there is also a library for interfacing with ROS. Using this library, the Arduino can send/receive messages to the PC. These messages are converted to topics on the PC side. Arduino can publish data and subscribe data, similar to a ROS node.

Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer. It uses Raspbian Operating System or Linux Operating System with chip ARM. The Pi can process multiple ROS nodes at a time and can take advantage of many features of ROS. In this project, the Raspberry Pi is used to transfer control data and environment data wirelessly from Arduino, sensors and camera to monitor.

Figure 2-9: Raspberry Pi 3+

Table 2.4: Raspberry Pi Specification

Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
1GB LPDDR2 SDRAM
2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)
Extended 40-pin GPIO header
Full-size HDMI
4 USB 2.0 ports
CSI camera port for connecting a Raspberry Pi camera
DSI display port for connecting a Raspberry Pi touchscreen display
4-pole stereo output and composite video port
Micro SD port for loading your operating system and storing data
5V/2.5A DC power input
Power-over-Ethernet (PoE) support (requires separate PoE HAT)

Motor Driver

The motor driver is an electronic circuit board that adjusts the speed of the motor by feeding a pulse-width modulated (PWM) signal as input. We are using the motor driver shown in Figure 2.10 for our robot.

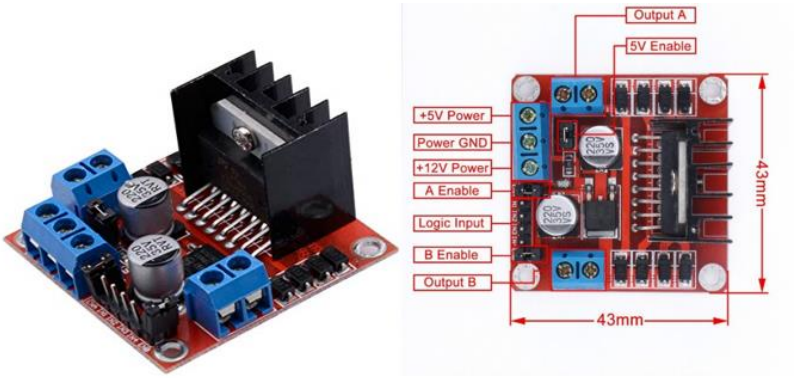


Figure 2-10: Motor driver LM928

This motor driver board uses a 298N chip with input voltage in the range of 5 volts to 35 volts, and a maximum drive current is up to 2A. One motor driver controls the speed of two motors, so we only need a single motor driver for this robot.

Encoder

Encoder is an electromechanical device that generate a digital signal that is correspond to speed/or position of measured device. It turns mechanic motion to read-able signal, which can be simply used by the microcontroller.

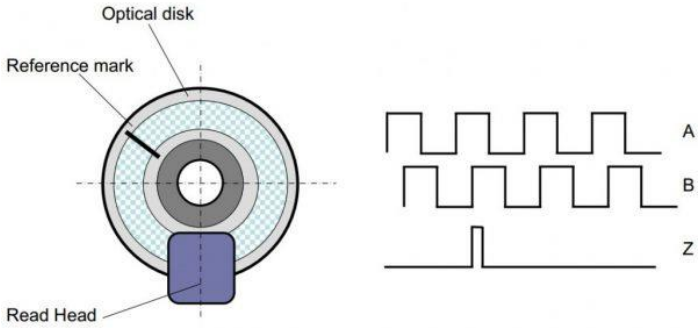


Figure 2-11: Motor's Encoder

Figure 2-12: Mobile robot's Optical encoder

2.4 External Sensors and Devices

2.4.1 3D Depth Camera

The camera corresponds to the eyes in the robot. The images obtained from the camera are very useful for recognizing the environment around the robot. Depth cameras can be divided into various types such as ToF (Time of Flight), Structured Light, and Stereo method:

ToF (Time of Flight)

The ToF method radiates infrared rays and measures the distance by the time it returns. In general, the IR transmission unit and the receiving unit are a pair and the distance measured by each pixel is read.

Figure 2-13: From left, D-Imager, SwissRnager, CamBoard, Kinect2

Structured Light

The structured light using the coherent radiation pattern method.

Figure 2-14 From left Kinect, Xtion, Carmine, Structure Sensor

Stereo

A stereo camera is one of the Depth Camera types. Its distance is calculated using binocular parallax like the left and right eyes of a person. The stereo camera is equipped with two image sensors at specifc distance and calculates the grid value using the difference between the two image images captured by these two image sensors.

Figure 2-15 From left Bumblebee, OjOcamStereo, RealSence

This 3D sensor technology is primarily for use indoors and does not typically work well outdoors. Infrared from the sunlight has a negative effect on the quality of readings from the depth camera. Objects that are shiny or curved also present a challenge for the depth camera.

2.4.2 Laser Distance Sensor

Laser Distance Sensors (LDS) are referred to various names such as Light Detection and Ranging (LiDAR), Laser Range Finder (LRF) and Laser Scanner. LDS is a sensor used to measure the distance to an object using a laser as its source. The LDS sensor has the advantage of high performance, high speed, real time data acquisition. This is a sensor widely used in the field of robots for recognition of objects and people, and SLAM (distance-based sensor).

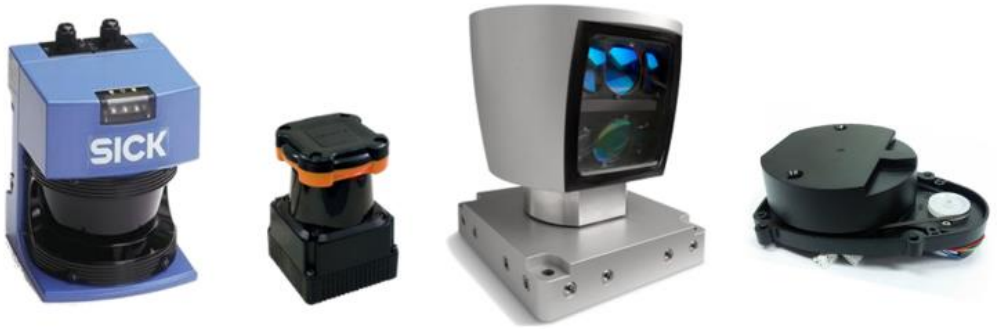


Figure 2-16: From left SICK LMS, Hokuyo UTM, Velodyne HDL, HLS-LFCD LDS

Principle of LDS Sensor's Distance Measurement

The LDS sensor calculates the difference of the wavelength when the laser source is reflected by the object. A typical LDS consists of a single laser source, a reflective mirror, and a motor. When we drive the LDS the rotating motor rotates the inner mirror and scans the laser in a horizontal plane, in other words, it is 2D data.

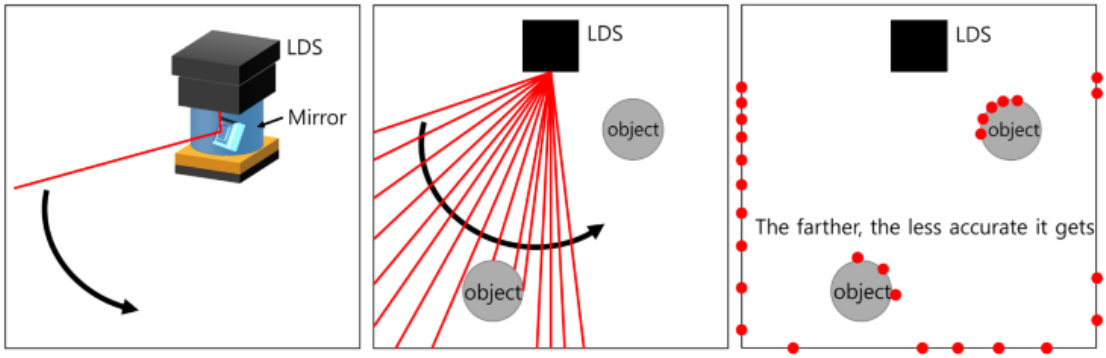


Figure 2-17: Distance measurement using LDS

Our robot is equipped with a 360 Laser Distance Sensor (Rplidar-A1) that is a 2D laser scanner capable of scanning 360 degrees with the aid of a DC motor that is attached to the sensor by a rubber belt drive.

Figure 2.2-18: Rplidar-A1 sensor

CHAPTER 3 SLAM AND NAVIGATION

3.1 Navigation of Mobile Robot

Navigation is the movement of the robot to a defined destination. It is important to know where the robot itself is and to have a map of the given environment. It is also important to find the optimized route among the various routing options, and to avoid obstacles such as walls and furniture.

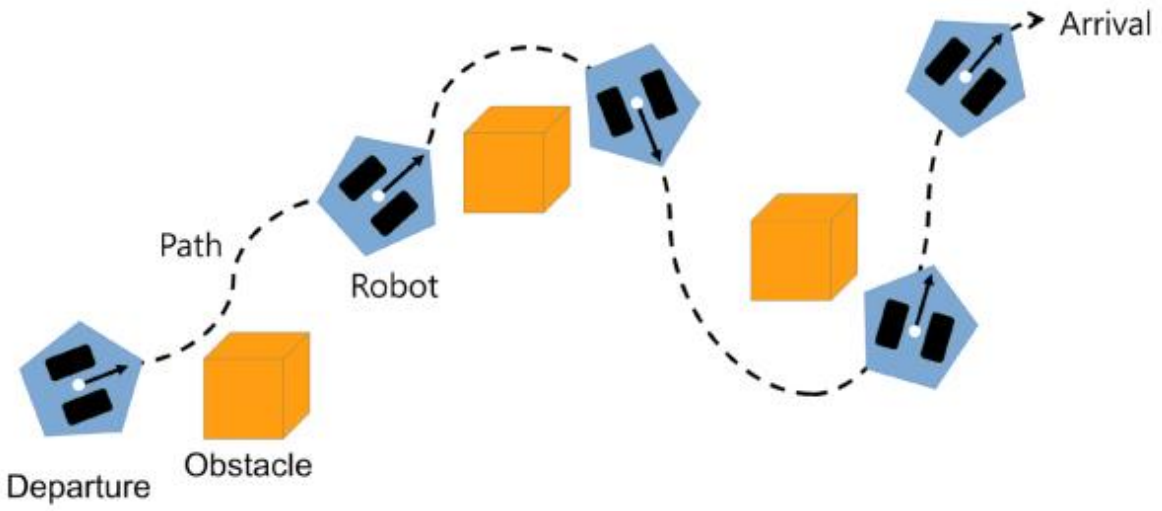


Figure 3-1: Mobile Robot Navigation

The followings are required as basic features for navigation:

- Map
- Pose of Robot
- Sensing
- Path Calculation and Driving

3.1.1 Map

The navigation system is equipped with a very accurate map and the modified map can be downloaded periodically so that it can be guided to the destination based on the map. We need to create a map and give it to the robot, or the robot should be able to create a map by itself.

3.1.2 Pose of Robot

A robot must be able to measure and estimate its pose (position + orientation). Currently, the most widely used indoor pose estimation method for service robots is dead reckoning. The

amount of movement of the robot is measured with the rotation of the wheel. ROS defines pose as the combination of the robot's position (x, y, z) and orientation (x, y, z, w).

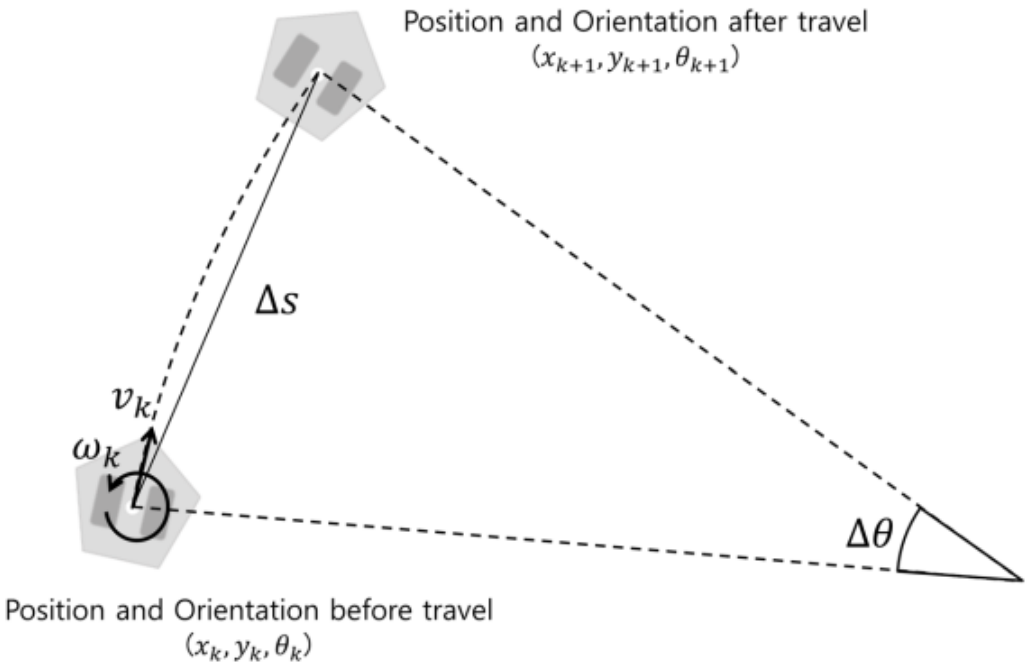


Figure 3-2 Dead Reckoning

Let D be the distance between the wheels and r be the radius of the wheel. Assuming the robot traveled a very short distance during time T_e, the rotational speed (ω_l, ω_r) of the left and right wheels are calculated as shown in Eqs. (3.1) and (3.2) with the amount of left and right motor rotation (current encoder value $E_{lc}, E_{rc}$and the previous encoder valueE_{lp}, E_{rp}).

$$\omega_l = \frac{E_{lc} - E_{lp}}{T_e} \times \frac{\pi}{180}(rad/s) \tag{3.1}$$

$$\omega_r = \frac{E_{rc} - E_{rp}}{T_e} \times \frac{\pi}{180}(rad/s) \tag{3.2}$$

The Equations 3.3 and 3.4 calculates the velocity of the left and right wheel (v_l, v_r). From the left and right wheel velocity, linear velocity v_k and the angular velocity $\omega_k$of the robot can be obtained as shown in Equations 3.5 and 3.6.

$$v_l = \omega_l \times r(m/s) \tag{3.3}$$

$$v_r = \omega_r \times r(m/s) \tag{3.4}$$

$$v_k = \frac{v_l + v_r}{2}(m/s) \quad (3.5)$$

$$\omega_k = \frac{v_r - v_l}{D}(rad/s) \quad (3.6)$$

Finally, using these values, we can obtain the position x_{k+1}, y_{k+1} and the orientation $\theta_{k+1}$of the robot from Equation 3.7 to 3.10.

$$\Delta s = v_k T_e \Delta\theta = \omega_k T_e \quad (4.7)$$

$$x_{k+1} = x_k + \Delta s cos\left(\theta_k + \frac{\Delta\theta}{2}\right) \quad (4.8)$$

$$y = y + \Delta s sin\left(\theta_k + \frac{\Delta\theta}{2}\right) \quad (4.9)$$

$$\theta_{k+1} = \theta_k + \Delta\theta \quad (4.10)$$

In this project, our robot has the configuration is:

Table 3.1: Robot's configuration

Parameter	Symbol	Value
Distance between two wheels (mm)	D	207
Radius of wheels (mm)	r	31
Encode resolution (Pulses/rev)	E	440

3.1.3 Sensing

Figuring out whether there are obstacles such as walls and objects requires sensors. Various types of sensors such as distance sensors and vision sensors are used.

3.1.4 Path Calculation and Driving

The last essential feature for navigation is to calculate and travel the optimal route to the destination. There are many algorithms that perform this such as A* algorithm, Dijkstra's algorithm, potential field, particle filter, and RRT (Rapidly-exploring Random Tree).

3.2 The navigation meta-package in ROS

The main objective of the Navigation meta-package is to move a robot from a position A to a position B, assuring it won't crash against obstacles, or get lost in the process. The following diagram shows how the navigation meta-package are organized. The plain white boxes indicate the Meta-packages that are provided by ROS, and they have all the nodes to make the robot autonomous. The parts marked in gray depend on the platform used. It is necessary to write code to adapt the platform to be used in ROS and to be used by the navigation Meta-package.

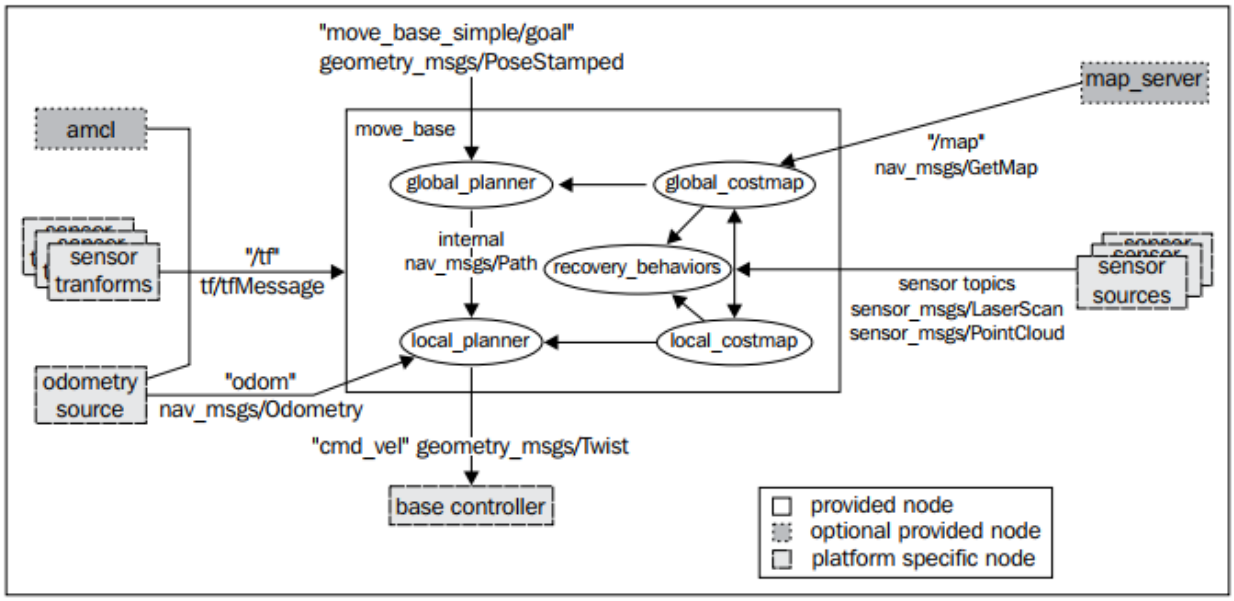


Figure 3-3: Navigation Meta-Package

The Navigation Stack will take as input the current location of the robot, the desired location the robot wants to go, the Odometry data of the Robot (wheel encoders, IMU, GPS. . .) and data from a sensor such as a Laser. In exchange, it will output the necessary velocity commands and send them to the mobile base in order to move the robot to the specified goal position.

3.3 Navigation core term

The core terms that are used in robot navigation are as follows:

- **Odometry:** This data is used to estimate the current position of the robot relative to its starting location.
- **Occupancy Grid Map (OGM):** An OGM is a map generated from the 3D sensor measurement data and the known pose of the robot. The environment is divided into an evenly-spaced grid in which the presence of obstacles is identified as a probabilistic value in each cell on the grid.

- **Localization:** Localization determines the present position of the robot with respect to a known map. The robot uses features in the map to determine where its current position is on the map.

3.3.1 Costmap

The pose of robot and obstacle information, and the occupancy grid map obtained as a result from SLAM are used to load the static map and utilize the occupied area, free area, and unknown area for the navigation.

Depending on the type of navigation, costmap can be divided into two. One is the '*global_costmap*', which sets up a path plan for navigating in the global area of the fixed map. The other is '*local_costmap*' which is used for path planning and obstacle avoidance in the limited area around the robot.

Marking and Clearing

The costmap automatically subscribes to sensors topics over ROS and updates itself accordingly. Each sensor is used to either *mark* (insert obstacle information into the costmap), *clear* (remove obstacle information from the costmap), or both.

Occupied, Free, and Unknown Space

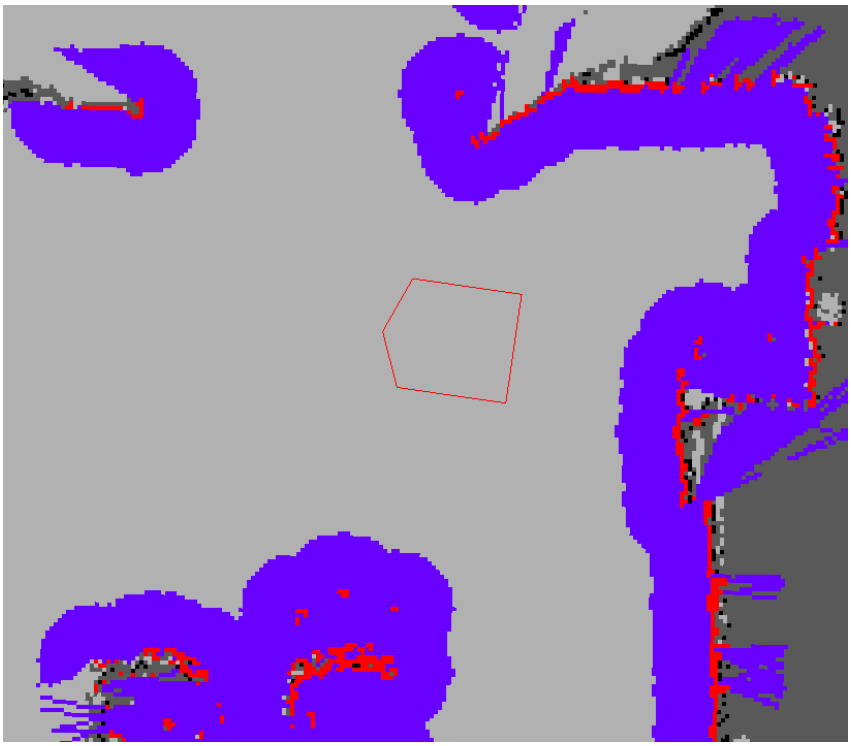


Figure 3-4: Costmap display

In the picture above, the red cells represent obstacles in the *costmap*, the blue cells represent obstacles inflated by the inscribed radius of the robot, and the red polygon represents the footprint of the robot. For the robot to avoid collision, the footprint of the robot should never intersect a red cell and the center point of the robot should never cross a blue cell.

Map Updates

The costmap performs map update cycles. Each cycle, sensor data comes in, marking and clearing operations are performed in the underlying occupancy structure of the costmap.

Inflation

Inflation is the process of propagating cost values out from occupied cells that decrease with distance. The costmap is expressed as a value between '0' and '255'. The value is used to identify whether the robot is movable or colliding with an obstacle. The calculation of each area depends on the costmap configuration parameters specified

- 000: Free area where robot can move freely
- 001~127: Areas of low collision probability
- 128~252: Areas of high collision probability
- 253~254: Collision area
- 255: Occupied area where robot can not move

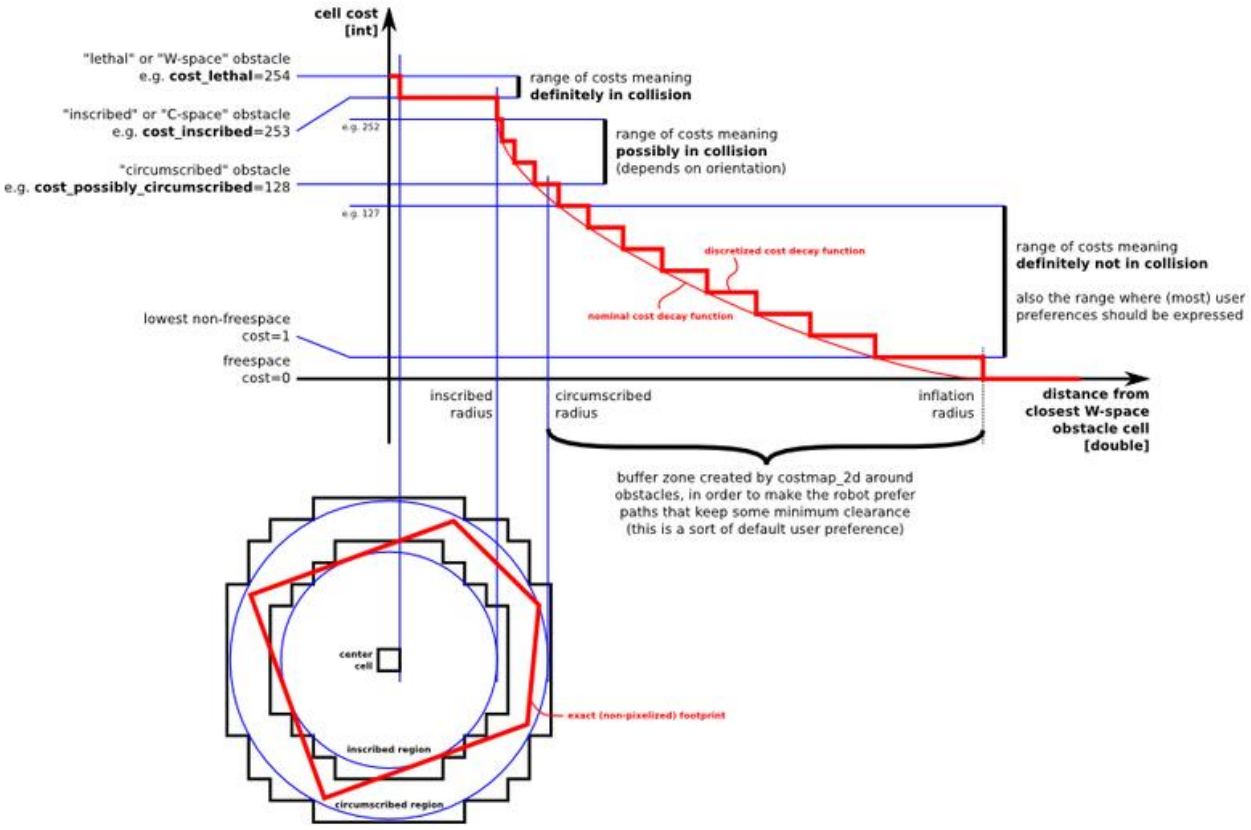


Figure 3-5 Relationship between distance to obstacle and costmap value

- "*Lethal*" cost means there is an actual (workspace) obstacle in a cell. If the robot's center were in that cell, the robot would obviously be in a collision.
- "*Inscribed*" cost means a cell is less than the robot's inscribed radius away from an actual obstacle so the robot is certainly in a collision with some obstacle if the robot center is in a cell that is at or above the inscribed cost.
- "*Possibly circumscribed*" cost is similar to "*inscribed*" but using the robot's circumscribed radius as cutoff distance.
- "*Freespace*" cost is assumed to be zero, and it means that there is nothing that should keep the robot from going there.
- "*Unknown*" cost means there is no information about a given cell. The user of the costmap can interpret this as they see fit.
- All other costs are assigned a value between "*Freespace*" and "*Possibly circumscribed*" depending on their distance from a "*Lethal*" cell and the decay function provided by the user.

3.3.2 AMCL

The Monte Carlo localization (MCL) pose estimation algorithm is widely used in the field of pose estimation. AMCL (Adaptive Monte Carlo Localization) can be regarded as an improved version of Monte Carlo pose estimation.

The goal of MCL is to determine where the robot is located in a given environment. That is, we must get x, y, and θ of the robot on the map. For this purpose, MCL calculates the probability that the robot can be located base on the position and orientation (x, y, θ) of the robot at time t and the distance information obtained from the distance sensor up to time t and the movement information obtained from the encoder up to time t.

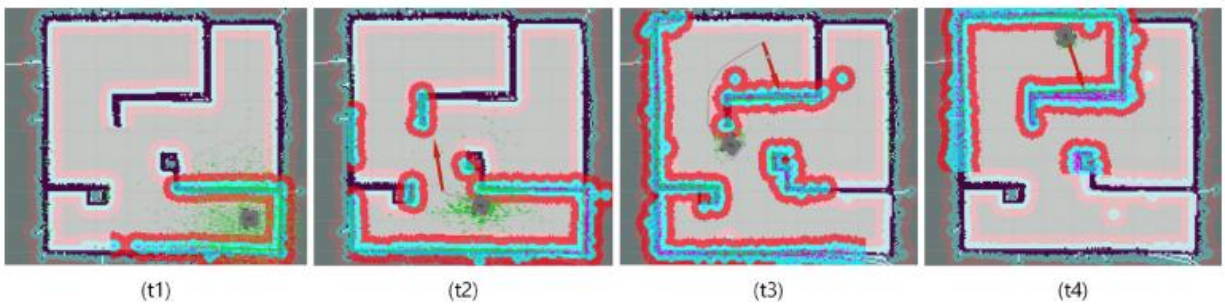


Figure 3-6: AMCL process for robot pose estimation

3.3.3 Dynamic Window Approach (DWA)

Dynamic Window Approach (DWA) is a popular method for obstacle avoidance planning and avoiding obstacles. This is a method of selecting a speed that can quickly reach a target point while avoiding obstacles that can possibly collide with the robot.

First, the robot is not in the x and y-axis coordinates, but in the velocity search space with the translational velocity v and the rotational velocity ω as axes, as shown in Figure 30. In this space, the robot has a maximum allowable speed due to hardware limitations, and this is called Dynamic Window.

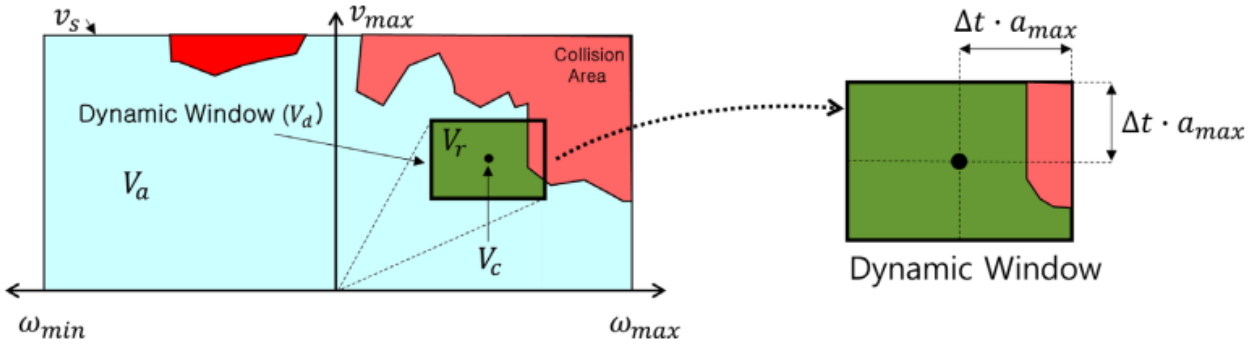


Figure 3-7: Robot's velocity search space and dynamixel window

v: Translational velocity (meter/sec)
ω: Rotational velocity (radian/sec)
V_s: Maximum velocity area
V_a: Permissible velocity area
V_c: Current velocity
V_r: Speed area in Dynamic Window
a_{max}: Maximum acceleration / deceleration rate
$G_{(v,\omega)} = \sigma(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega))$: Objective function
$heading(v,\omega)$: 180 − (difference between the direction of the robot and the direction of the target point)
$dist(v,\omega)$: Distance to the obstacle
$velocity(v,\omega)$: Selected velocity
α, β, γ: Weighting constant
$\sigma(x)$: Smooth Function

Figure 3-8: DWA's parameters

In this dynamic window, the objective function $G(v, \omega)$ is used to calculate the translational velocity v and the rotational velocity ω that maximizes the objective function which considers the direction, velocity and collision of the robot.

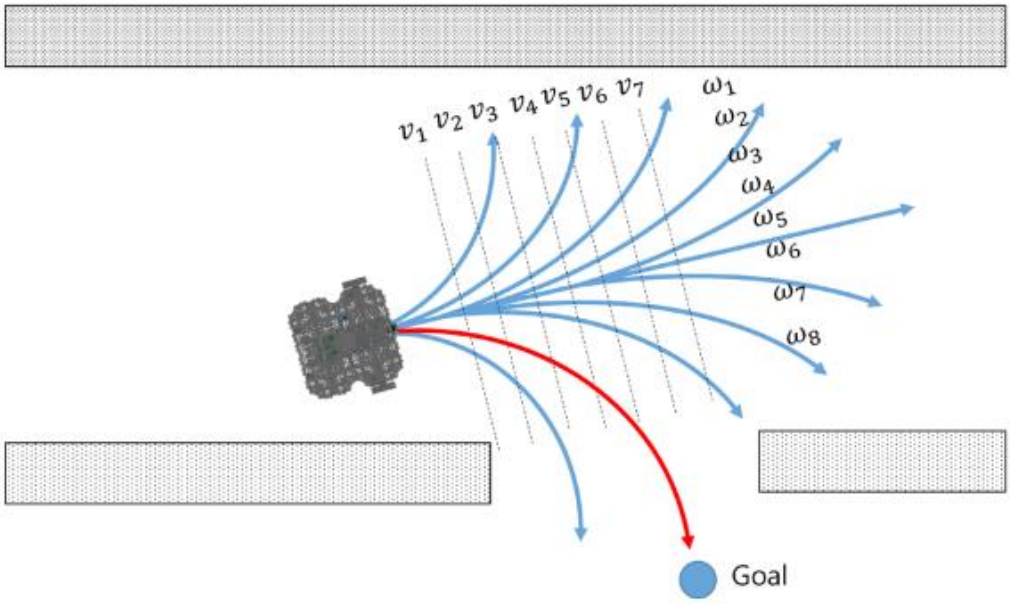


Figure 3-9: Translational velocity v and rotational velocity ω

3.4 SLAM theory

SLAM (Simultaneous Localization and Mapping) means to explore and map the unknown environment while estimating the pose of the robot itself by using the mounted sensors on the robot.

Encoders and IMU are typically used for pose estimation. This estimated pose can be corrected once again with the surrounding environmental information obtained through the distance sensor or the camera used when creating the map. This pose estimation methodology includes Kalman filter, Markov localization, Monte Carlo localization using particle filter, and so on.

Distance sensors such as ultrasonic sensors, light detectors, radio detectors, laser range finders, and infrared scanners are often used for mapping. In addition to the distance sensor, cameras are also used to measure the distance such as a stereo camera.

3.4.2. Various Localization Methodologies

SLAM can be done easily if the pose of the robot can be properly estimated. However, there are many problems such as uncertainty of the sensor observation information, and the real-time

property must be secured in order to operate in the actual environment. Kalman filter and Particle filter methodology are discussed as general examples of pose estimation.

Particle filter

Particle Filter is the most popular algorithm in object tracking. In the particle filter method, the uncertain pose is described by a bunch of particles. We move the particles to a new estimated position and orientation based on the robot's motion model and probabilities and measure the weight of each particle according to the actual measurement value, and gradually reduce the noise to estimate a precise pose.

This particle filter goes through the following 5 procedures. Except for the initialization in step 1, steps 2~5 are repeatedly performed to estimate the robot's pose. It is a method to estimate the pose of the robot by updating the distribution of the particles that shows the probability of the robot on the X, Y coordinate plane based on the measured sensor value.

- Step 1: Initialization

  Since the robot's initial pose is unknown, the particles are randomly arranged within the range where the pose can be obtained with N particles. Each of the initial particle weighs 1/N, and the sum of the weight of particles is 1. N is empirically determined, usually in the hundreds. If the initial position is known, particles are placed near the robot.

- Step 2: Prediction

  Based on the system model describing the motion of the robot, it moves each particle as the amount of observed movement with odometry information and noise.

- Step 3: Update

  Based on the measured sensor information, the probability of each particle is calculated and the weight value of each particle is updated based on the calculated probability.

- Step 4: Pose estimation

  The position, orientation, and weight of all particles are used to calculate the average weight, median value, and the maximum weight value for estimating pose of the robot.

- Step 5: Resampling

  The step of generating new particles is to remove the less weighed particles and to create new particles that inherit the pose information of the weighted particles. Here, the number of particles N must be maintained.

3.4.3 Registration

Iterative closest point

Iterative Closest Point (ICP) is an algorithm, which minimizes the difference between two points clouds by iteratively finding correspondences between the two sets of points. In the algorithm, one cloud, the Target, is fixed while the other cloud, the Source, is transformed. In each iteration, the closest neighbor of each point in the source is found by using a search algorithm and the rigid body transformation between the target points and their closest neighbor. The entire target point cloud is then transformed using the rigid body transformation estimation and a new closest neighbor search is performed. This process is iterated until convergence.

Algorithm Iterative Closest Point
Require: Point clouds: $A = \{a_i\}$, $B = \{b_i\}$ Initial transformation $M0$ 1: $M = M_0$ 2: **while** not converged **do** 3: **for** i = 1 to N **do** 4: c_i = FindClosestPointInA($M \cdot b_i$) 5: **if** $kc_i - M \cdot b_i k \leq dmax$ **then** **6:** $w_i = 1$ 7: else 8: $w_i = 0$ 9: **end if** 10: **end for** 11: M=$argmin\{\sum_i w_i \, \| M \cdot b_i - c_i \|^2\}$ 12: **end while** 13: **return** M

The target A and source B is necessary but the initial transformation M0 is optional. If no initial transform is known, it is set to identity, which corresponds to no transformation at all. In line 4, the set of target points which are closest to the source set is found. For some applications, it might be beneficial to down-sample the point clouds before registration. This reduces the necessary computational power at the expense of accuracy.

To account for the fact that some points will not have any correspondence, a threshold dmax is used to define the weights in line 5 through 9. This threshold represents a trade-off between accuracy and convergence in most implementations of ICP.

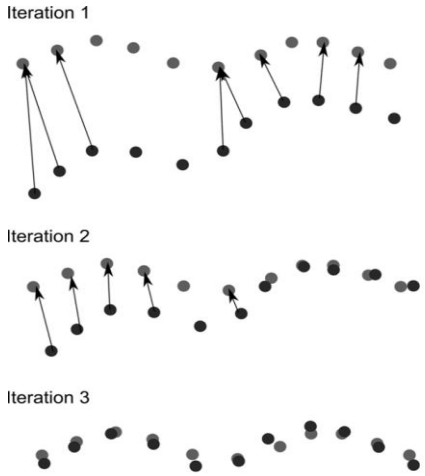


Figure 3-10: Iterative Closest Point Process

Feature-Matching

Instead of taking whole point clouds into consideration, a few features can be used to register. This greatly reduces the computational complexity as well as the amount of information to store. Since the point clouds often do not have complete point-point correspondence, which can happen when the point clouds only partially overlap, a lot of effort has been made into feature selection and description extraction to improve correspondence selection. Two sets of features from two different scans are matched and a set of matching feature pairs is selected. For each matching pair, a homogeneous transformation can be calculated through the least-squares formulation.

A feature is often generated in basically two steps, detection and description. In the detection step, the image is searched for points of interest or key-points. Commonly used algorithms for 3D point clouds include NARF and 3D-SIFT. In the description step, the areas around the key-points are described. Commonly used descriptors for images include SIFT, SURF and BRISK.

SIFT(scale-invariant feature transform): a feature detection algorithm to detect and described the local feature in images. It uses gray-scale gradient vector as descriptors.

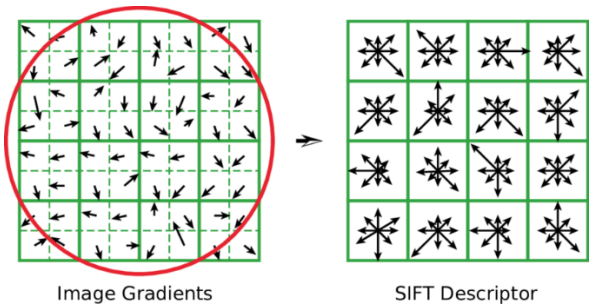


Figure 3-11: SIFT working principle

FLANN (Fast Library for Approximate Nearest Neighbors) is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

Figure 3-12: Feature-Matching with FLANN

RANSAC (Random sample consensus) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it capable of finding inliers and outliers of desired model.

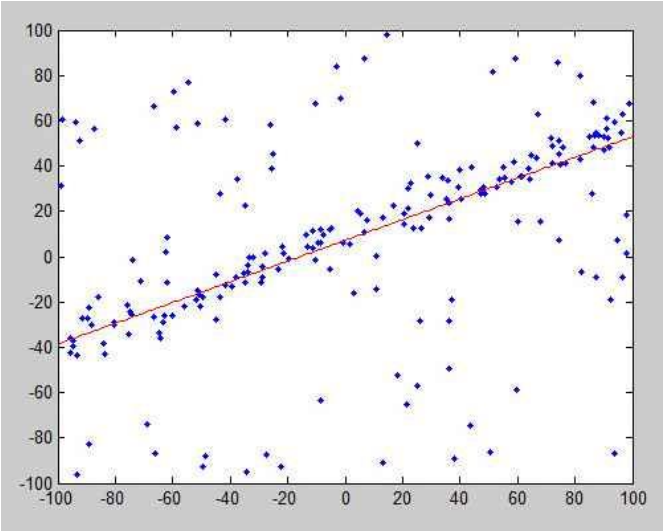


Figure 3-13: RANSAC fitting in a set of data

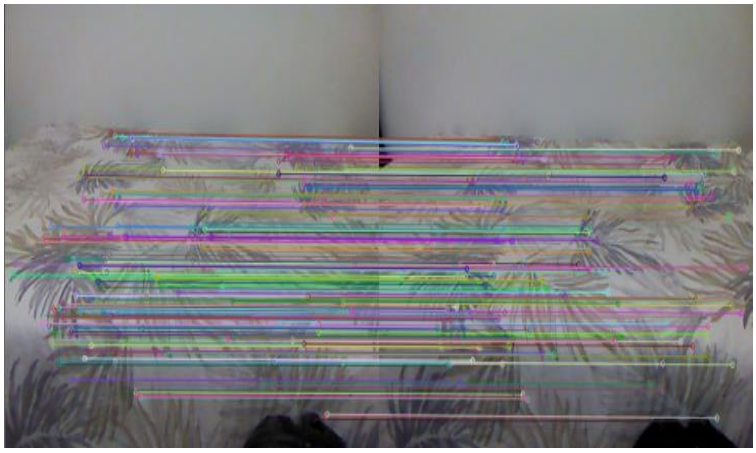


Figure 3-14: Feature-Matching after applying RANSAC

3.5 SLAM Variation

3.5.1 Fast SLAM

The Fast SLAM algorithm is based on the particle filter approach. Since particle filters suffer from the curse of dimensionality, using particles to estimate all landmarks would be infeasible. In these, the particles represent the posterior over some variables while other probability density functions represent the other variables.

3.5.2 Graph SLAM

The basic idea behind the Graph SLAM algorithm is to model the posterior as a graph where poses and landmarks are represented by nodes, and motions and measurements as edges between the nodes. Edges can be viewed as soft constraints between the nodes. As the motion and measurements are subject to noise the constraints will be contradictory.

The Graph SLAM algorithm is generally split into two separate problems, the Front-End which constructs the graph from the measurements and the Back-End which solves the optimization of the state vector.

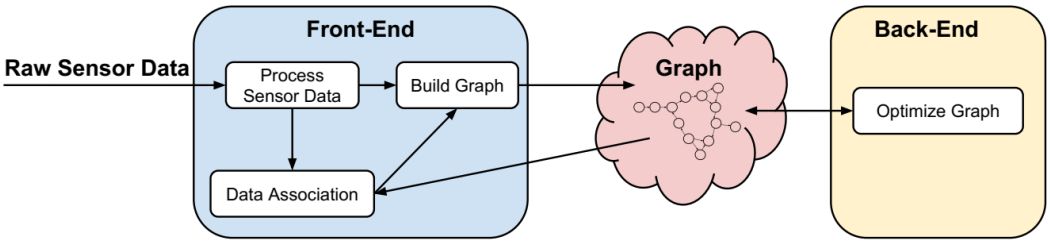


Figure 3-15: Division of the Graph SLAM algorithm into Front-End and Back-End.

Front-end

The Front-End builds the graph of nodes and edges from sensor measurements. While the robot is moving, edges between poses are created. New landmarks are added as a node with an edge to the pose from which it was first observed. When the robot sees a previously observed landmark, a new edge is formed between the landmark and the pose node currently observing the landmark. In the pose-Graph SLAM algorithm, since no landmarks are stored, loop closures are formed as constraints between the current pose and a previously observed pose.

The global ambiguity problem is solve by clustering loop closure hypotheses in local groups, and the local ambiguity problem is solve by finding the most pairwise consistent set of hypotheses. When the robot returns to an area it has previously visited and is able to recognize where it is in the environment relative to its previous pose.

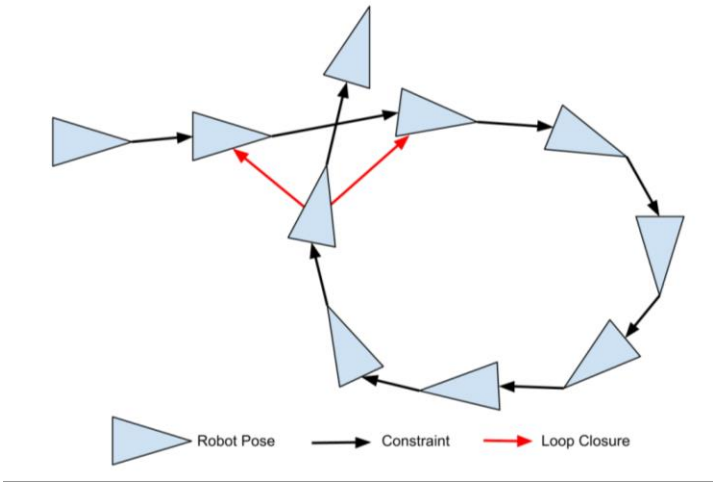


Figure 3-16: Illustration of a loop closure

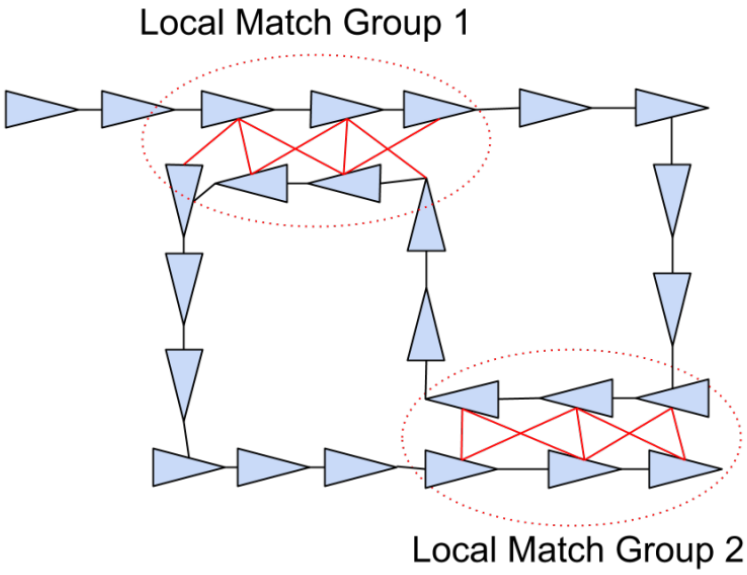


Figure 3-17: Drawing illustrating two local groups of loop closures

Each element is the probability that the hypotheses and agree. By finding the shortest path between the hypotheses, a loop can be constructed as seen in Figure 3.18 and the probability of hypothesis agreement can be estimated with the information stored in the edges. A pair of hypotheses *hi* and *hj* are said to be consistent if the transformations around the loop returns to the origin.

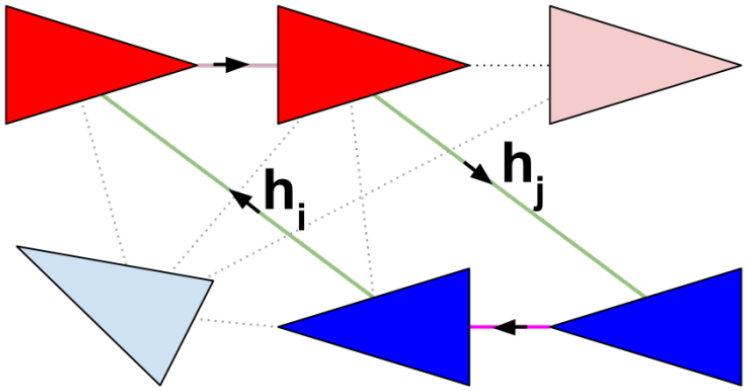


Figure 3-18: Drawing illustrating a set of hypotheses

Back-end

The Back-End optimizes the poses and landmarks based on the information stored in the graph. This is done by finding the minimum of the least-squares objective function by using a solver. The arrows denote the direction of the transformations. *M* denotes transformations and **e***ij* the error in vector form.

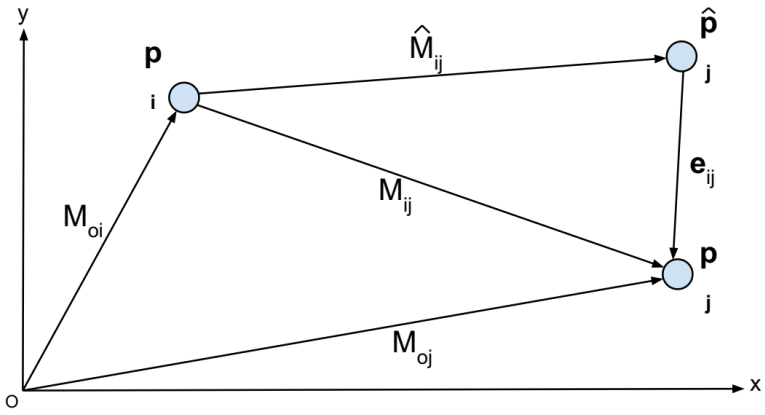


Figure 3-19: Illustration of the GraphSLAM error function

The optimization of the graph is very sensitive to outliers, e.g. incorrect loop closures. These can contort the graph beyond recognition. It is therefore important to remove outliers before the optimization (Front-End) or compensate for them during the optimization (Back-End). For increased robustness, outliers should be handled in both the Front-End and Back-End.

3.5.3 Comparison between variations

After having done some research, FastSLAM and GraphSLAM algorithms were identified. A comparison of the algorithms can be seen in Table 3.2. In the complexity row, *n* is the number of landmarks, *k* the number of particles and *e* is the number of edges.

Table 3.2: Comparison of the two main categories of SLAM algorithms.

	Fast SLAM	**Graph SLAM**
Complexity	O(k*log(n))	O(*e*)
Distribution	Any	Gaussian + outliner rejection
Linearization	Not need	Re-linearize
Flexibility	+	++
Large scale	+	++
Parallelizability	+	++
Pros	Can use negative information	Scales well, robust
Cons	Hard to recover, need many particles to be robust	Harder to implement

Since SLAM and navigation usually work with medium to large map, Graph SLAM would be the best algorithms. In addition, the algorithm scales well and is robust so it fits for online or real-time job. Further, only pose nodes are used to reduce the size of the graph along with the optimization complexity.

3.6 SLAM implementation

There are a great variety of open-source SLAM approaches available through ROS. In this section, we review the most popular ones to outline their characteristics in terms of inputs and outputs.

3.6.1 2D SLAM

Hector SLAM can create fast 2D occupancy grid maps from a 2D LIDAR with low computation resources. It has proven to generate very low-drift localization while mapping in real-world

autonomous navigation scenarios. However, Hector SLAM is not exactly a full SLAM approach as it does not detect loop closures, and thus the map cannot be corrected when visiting back a previous localization. Hector SLAM does not need external odometry, which can be an advantage when the robot does not have one but can be a disadvantage when operating in an environment without a lot of geometry constraints, limiting laser scan matching performance. Hector SLAM also can be used with AMCL for localization and navigation.

The laser scans can be used to build a map by employing a probabilistic approach. For a robot at a given pose, each range measurement in the scan determines the coordinates of a cell that contains an obstacle. Cells that are behind the detected obstacles are registered as unknown cells whereas the cells that are between the sensor and the detected obstacles are registered as obstacle-free cells.

Figure 3-20: Rplidar-A1 sensor

The maximum distance of Rplidar-A1 rangefinder (500cm) plays a key role in the measurements, if there is no returning laser pulse within 23.33ns then the robot registers all the cells within 500cm range in the direction of the emitted laser pulse as obstacle-free cells.

Figure 3.21 is provided as an example of how the laser rangefinder operates. In this case, the grid cell size of 5cm is chosen, the sensor is located at cell (0, 0). The unexplored and the unknown cells that are behind the obstacle are shown by gray whereas white cells are the obstacle-free cells and the black cells are the occupied cells.

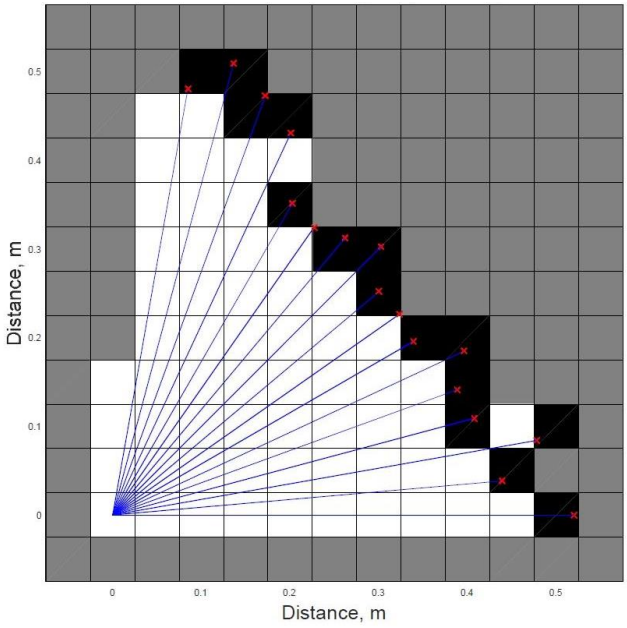


Figure 3-21: LIDAR Data point occupancy grid map

Now that a diagonal obstacle is detected, a line can be fitted to analytically express the obstacle. Figure 3.22 shows the result of the line fitting procedure. The estimated wall is defined by the green line

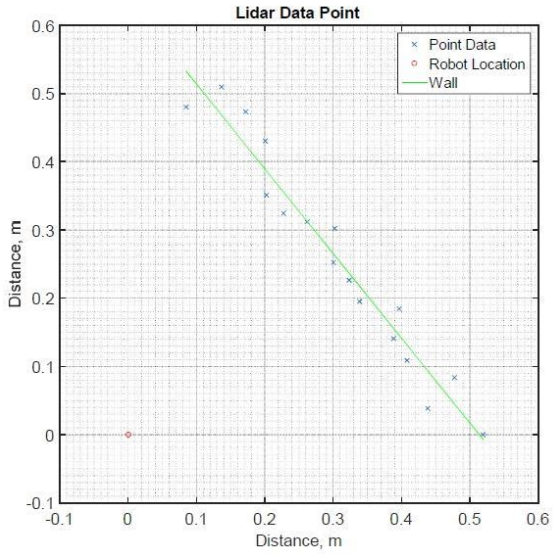


Figure 3-22: Line fitting using LSE

However, the line that is fitted to the obstacle is not useful unless we relate it to the robot. To relate the line to the robot the normal line needs to be calculated. Let us define the line in polar coordinates at the point which is normal to the line. The normal line is shown by Figure 3.23.

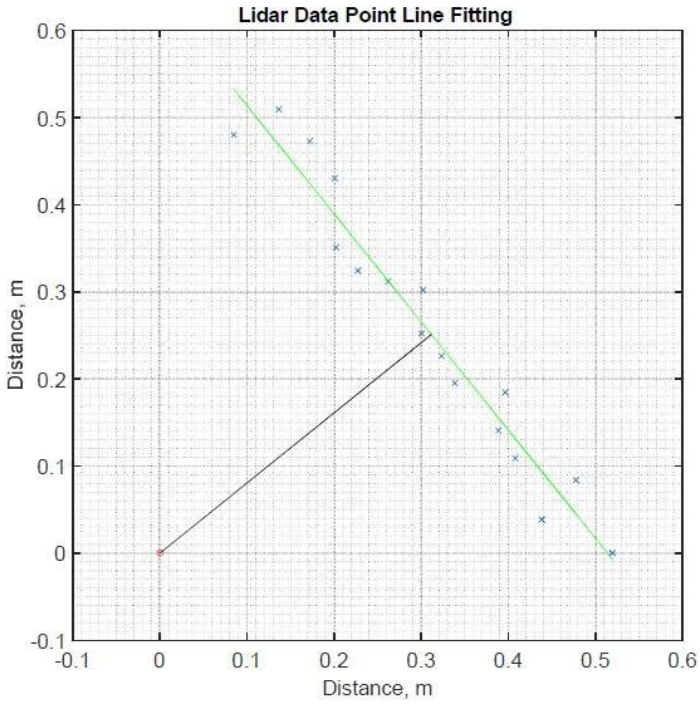


Figure 3-23: Normal line and the fitted line

The split-and-merge algorithm is used in Hector SLAM package as line extraction algorithm. The procedure of the algorithm is illustrated by Figure 3.24. The line estimation is done using LSE where the first and last data points are connected by a straight-line. Then a normal line is found that connects the furthest data point to the straight line between the first and the last data points. If the length of the normal line is greater than a given threshold, the line splits into two, connecting the first and last data points with the furthest data point. This procedure is repeated until the length of the normal line is less than the pre-defined threshold value.

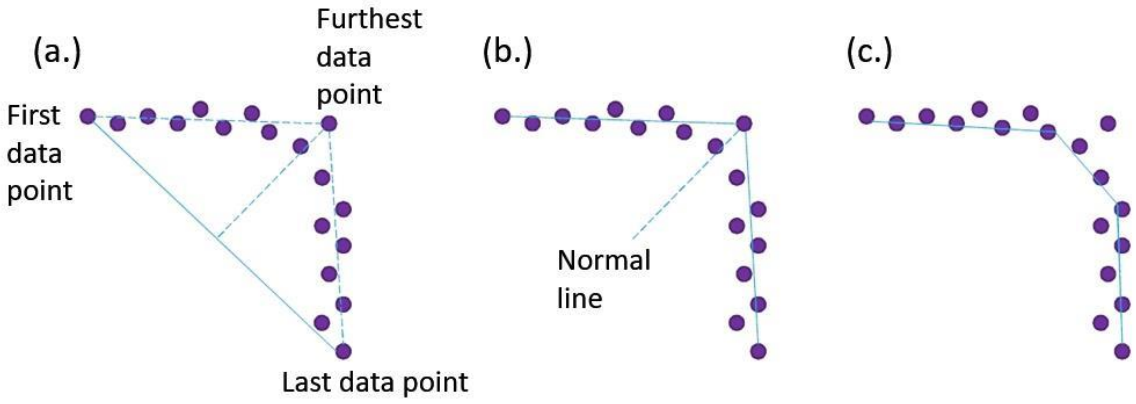


Figure 3-24: Split-and-merge algorithm procedure

3.6.2 3D SLAM

RTAB-Map (Real-Time Appearance-Based Mapping) is an RGB-D graph-based SLAM approach based on an incremental appearance-based loop closure detector. Loop closure is the problem of recognizing a previously-visited location and updating beliefs accordingly.

A graph-based SLAM system such as *RTAB-Map* consists of three stages: sensor measurement, frontend, and backend. In the frontend stage, the sensor data is processed and the geometric constraints between the successive RGB-D frames are extracted. While the odometry was estimated in the frontend stage, the backend stage is focused on solving the accumulated drift problem and on detecting the loop closure in Loop Detection. Finally, the 3D map is generated by using the *g2o* algorithm in Global Optimization. Figure 3.25 illustrates the flowchart of the proposed method.

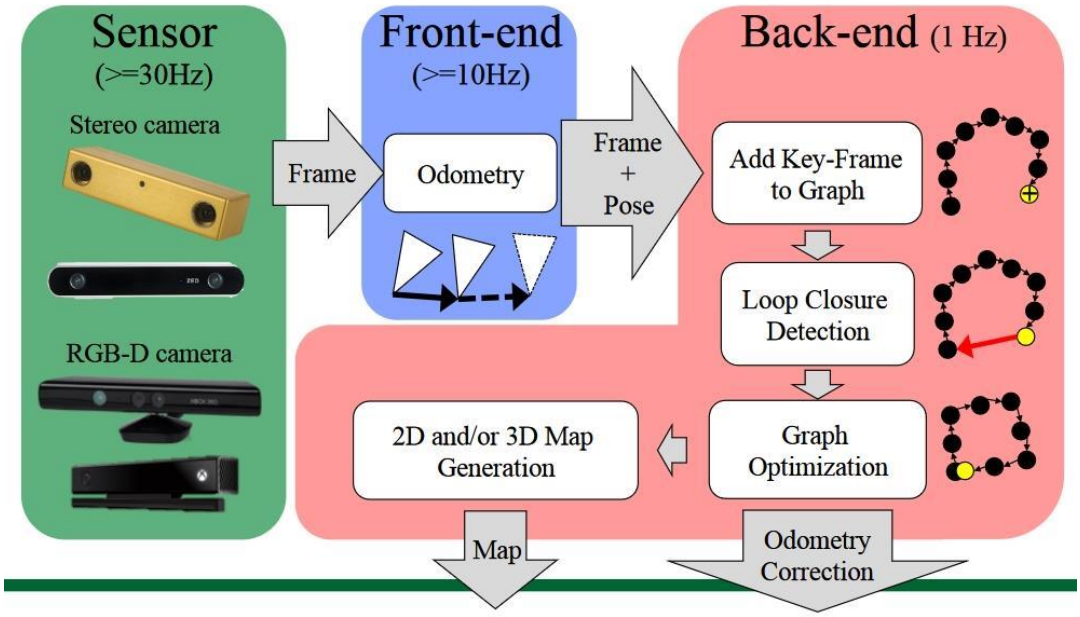


Figure 3-25: Stages of the Real-Time Appearance-based Mapping (RTAB-Map)

3.6.3 Sensor fusion

Sensor fusion networks can be categorized according to the type of sensor configuration.

Complementary: A sensor configuration is called complementary if the sensors do not directly depend on each other but can be combined to give a more complete image of the phenomenon under observation. This resolves the incompleteness of sensor data.

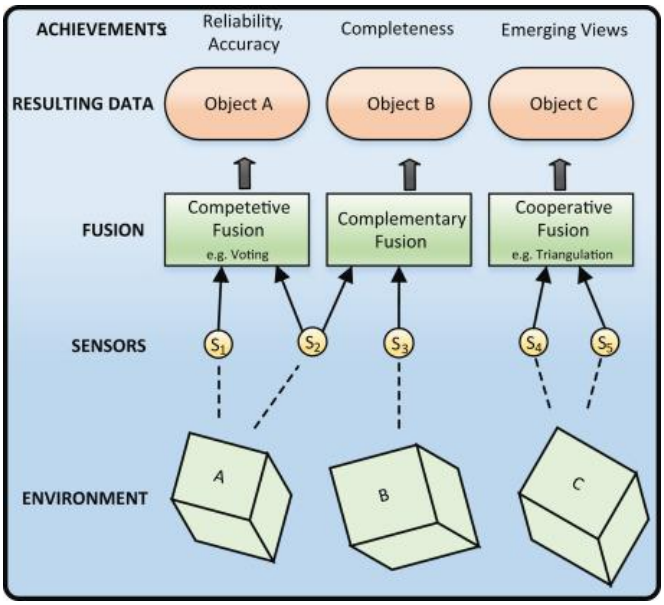


Figure 3-26: Competitive, complementary, and cooperative fusion

Competitive: Sensors are configured competitively if each sensor delivers independent measurements of the same property. There are two possible competitive configurations: the fusion of data from different sensors or the fusion of measurements from a single sensor taken at different instants.

Cooperative: A cooperative sensor network uses the information provided by two independent sensors to derive information that would not be available from the single sensors. An example of a cooperative sensor configuration is stereoscopic vision—by combining two-dimensional images from two cameras at slightly different viewpoints, a three-dimensional image of the observed scene is derived.

3.7 SLAM Application in ROS

In this section, we are going to look into the ROS packages used in SLAM. The Hector SLAM is used for 2D SLAM, RTAB-Map is used for 3D SLAM and AMCL is used for localization.

3.7.1 Occupancy Grid Map

The OGM is shown in Figure 3.27, white is the free area in which the robot can move, black is the occupied area in which the robot can’t move, and gray is the unknown area.

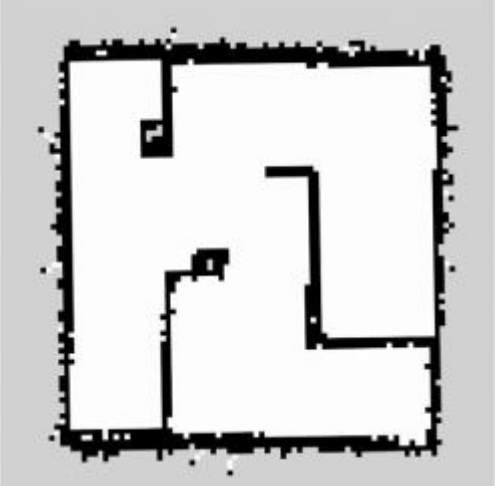


Figure 3-27: Occupancy Grid Map

The area in the map is represented by grayscale values that range from '0' to '255'. This value is obtained through the posterior probability of the Bayes' theorem, which calculates the occupancy probability that represents the occupancy state.

$$occ = \frac{255 - color\_avg}{255.0} \tag{4.11}$$

The closer this 'occ' is to 1, the higher the probability that it is occupied, and the closer to '0', it is the less likely to be occupied.

When the occupancy probability is published as ROS message (*nav_msgs/OccupancyGrid*), it is redefined to an integer [0 ~ 100]. An area close to '0' is the free area defined as an unoccupied area whereas '100' is defined as an occupied area, and '-1' is especially defined for unknown area.

In ROS, actual map is stored in *'*.pgm'* file format and the *'*.yaml'* file contains map information about the image parameter defines the map file name and resolution defines the map resolution in meters/pixel.

```
image: map.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figure 3-28: map.yaml

Each pixel can be converted to 5cm. Origin is the origin of the map. The lower left corner of the map represents x yaw respectively. Negate inverts black and white color. The color of each pixel is determined by the occupancy probability. If the occupancy probability exceeds occupied threshold (*occupied_thresh*), the pixel is expressed as an occupied area in black color. Otherwise, the pixel will be expressed as a free area in white color.

3.7.2 Information required in SLAM

The first thing we need is the distance value. The robot should be able to obtain the distance value from certain objects. Second is the pose value which stands for the pose information of the sensor that is attached to the robot. Thus, the pose value of the sensor depends on the odometry of the robot, it is necessary to provide the odometry information to calculate the pose value.

In the below Figure 3.29, the distance measured with LDS is called *'scan'* in ROS and the pose information is affected by the relative coordinate, so it is called *'tf'* (transform). We run SLAM based on two pieces of information, 'scan' and 'tf' and create the map we want.

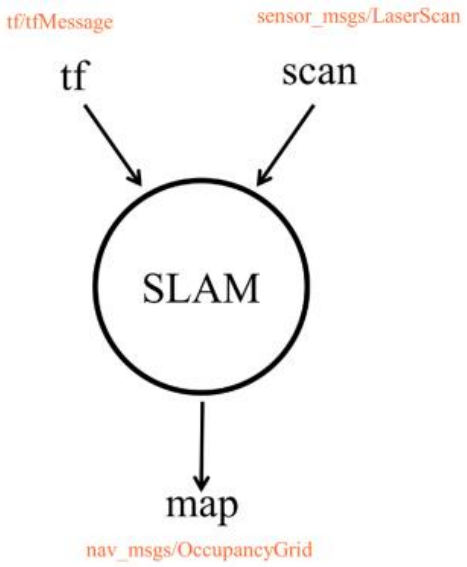


Figure 3-29: Scan and tf data required in SLAM and the relation to the map.

Transform configuration

A transform tree defines offsets in terms of both translation and rotation between different coordinate frames. Consider our mobile robot of a simple robot that has a mobile base with a

single laser mounted on top of it. We'll call the coordinate frame attached to the mobile base *"base_link"* and we'll call the coordinate frame attached to the laser *"base_laser"*.

When we have some data from the laser in the form of distances from the laser's center point, we want to take this data and use it to help the mobile base avoid obstacles in the world. To do this we need to define a relationship between the *"base_laser"* and *"base_link"* coordinate frames.

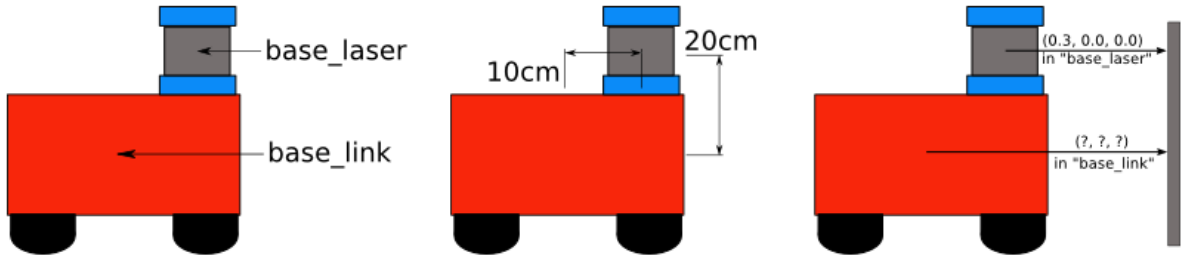


Figure 3-30: Relationship between the "base_laser" and "base_link" coordinate frames

We'll define the relationship between *"base_link"* and *"base_laser"* once using *tf* and let it manage the transformation between the two coordinate frames for us.

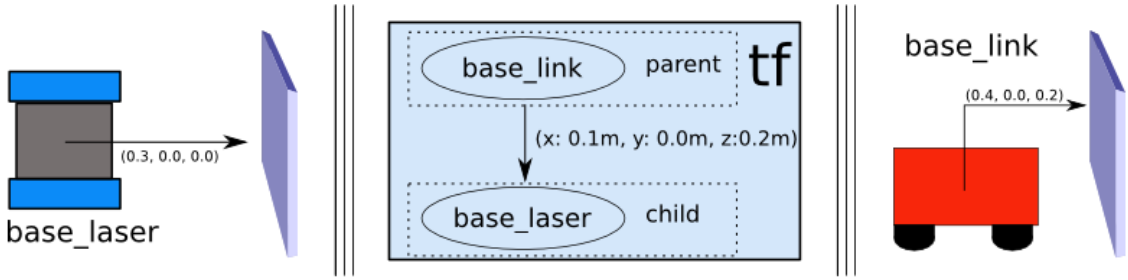


Figure 3-31: relationship between "base_link" and "base_laser" using tf

Publish sensor streams over ROS

There are many sensors that can be used to provide information to the Navigation Stack: lasers, cameras, sonar, infrared, bump sensors, etc. However, the current navigation stack only accepts sensor data published using either the *sensor_msgs/LaserScan* Message type or the *sensor_msgs/PointCloud* Message type.

3.8 Experiment result

3.8.1 SLAM building 2D map

The laser-based SLAM packages that are used in this project have several characteristic constraints. As the Rplidar-A1 emits infrared light, the objects that are matte-black, glasses that do not reflect infrared light, and mirrors that scatter light degraded the performance of the laser-based SLAM packages. It is also well known that long corridors without any distinctive obstacles, square-shaped rooms and open wide areas where no obstacle information can be acquired make the laser-based SLAM algorithms non-operational.

The map below is the result of experiment. Maps provide only a cross-section of the room, considering that the robot height is only about 23cm.

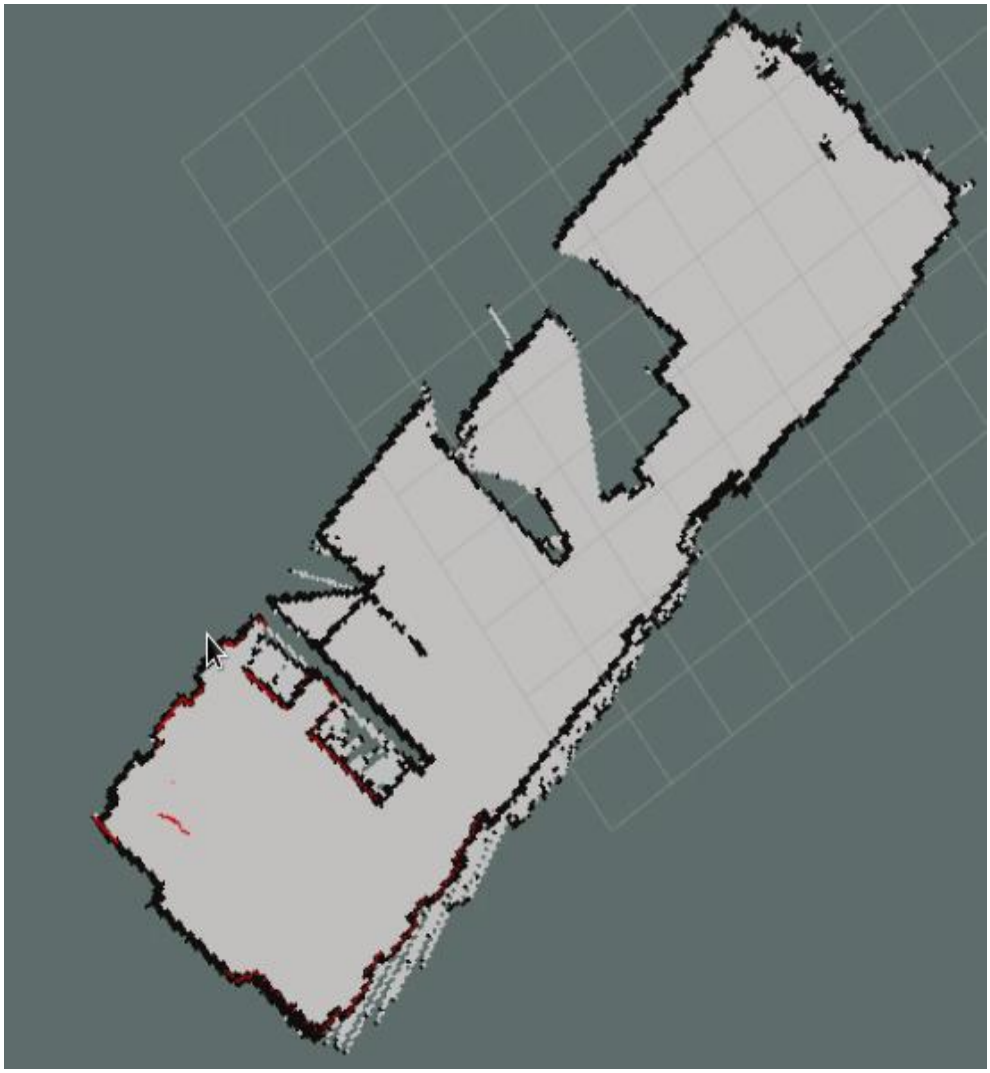


Figure 3-32: Using hector SLAM to generate 2D map

3.8.2 SLAM building 3D map

The RTAB-Map package provides a graphical user interface named as rtabmapviz, which visualizes visual odometry, output of the loop closure detector, and a point cloud that is a 3D dense map.

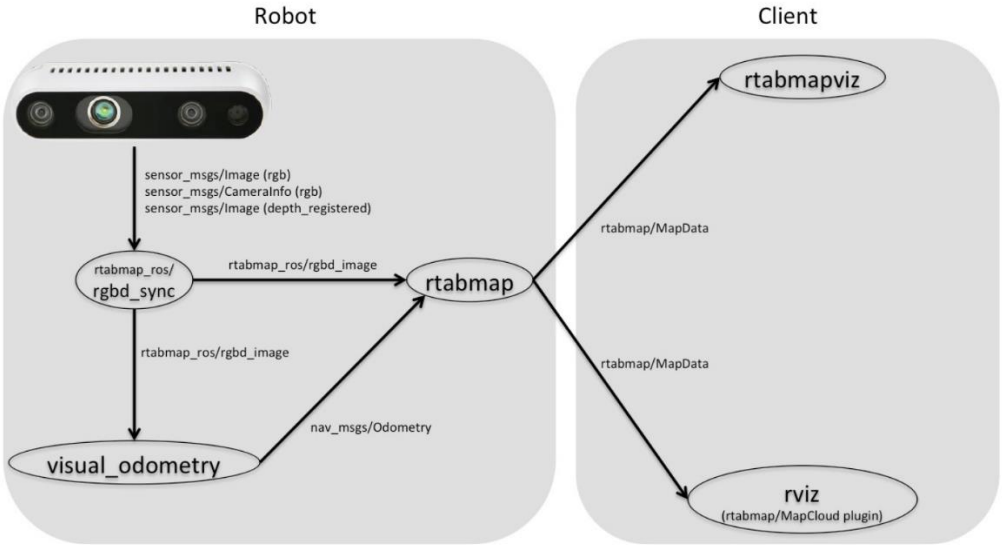


Figure 3-33: 3D mapping process

Using the Intel Realsense D435 sensor, the generated 3D map has good quality and the point cloud is dense. The following results are done by connect the camera directly to the laptop.

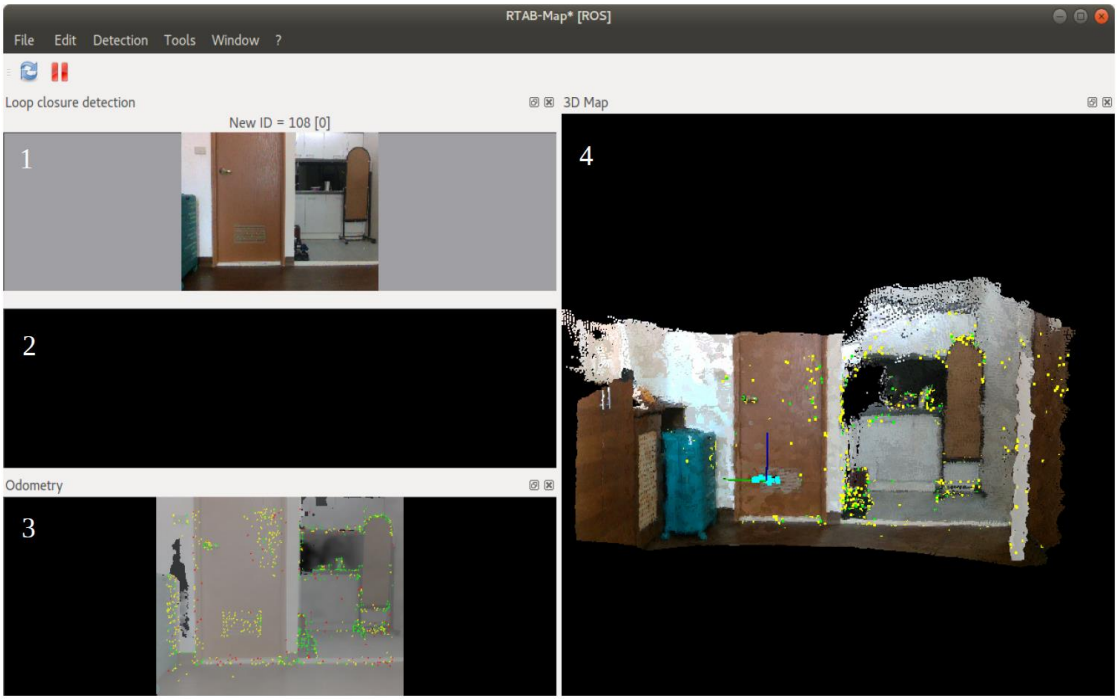


Figure 3-34: Rtabrviz interface

The rtabrviz interface gives several information while making 3D map figure 3.34. Window 1 is the RGB image that camera received. Window 2 shows when there is loop closure detection or not. Window 3 gives the image after applying SIFT algorithm and highlight points that will be used for feature matching. And window 4 show 3D point cloud map.

3.8.3 Sensor fusion

In this project, sensors are configured competitively to improve the output odometry, since both sensors give distance information of surrounding environment. Using Kalman filter as base, information will be weighted and only the one with reliability high enough will be chosen.

Both sensors will give point cloud data with the distance information of surrounding environment. Then feature-matching algorithm will be applied to 3D point cloud to update the map; 2D point cloud will be used to deduce the odometry information. After combining the calculated information, we will receive the pose information of the robot.

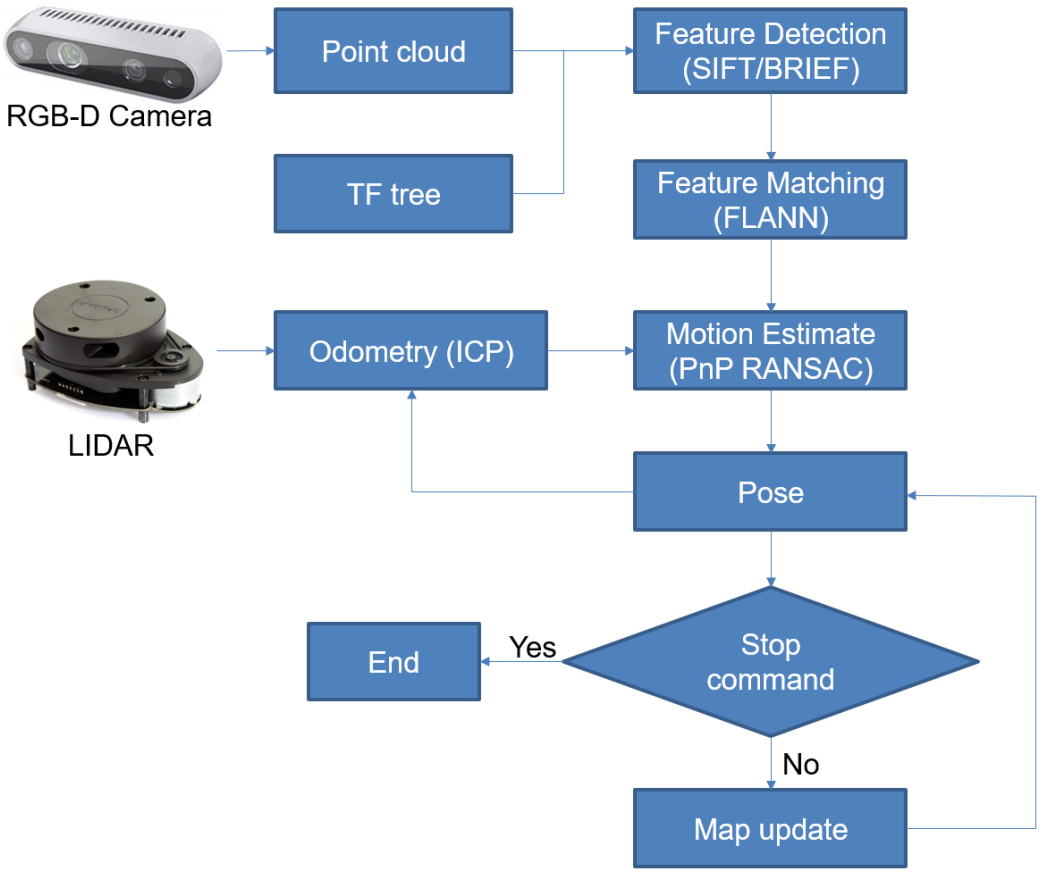


Figure 3-35: Sensor fusion flow chat

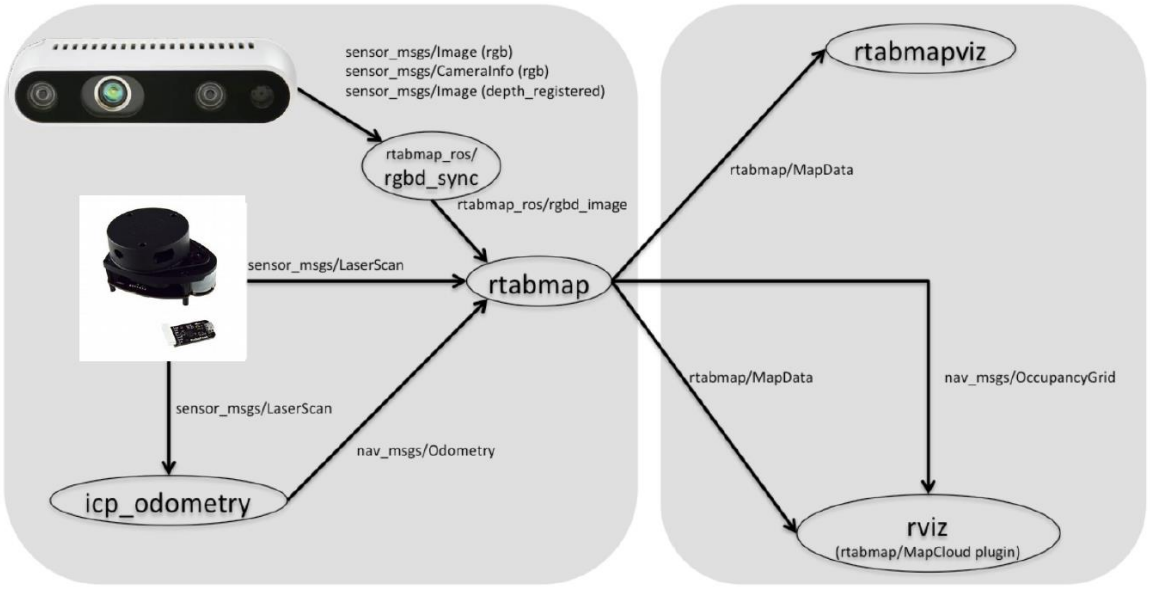


Figure 3-36: Mapping process after apply sensor fusion

The following figure show mapping process after apply sensor fusion.

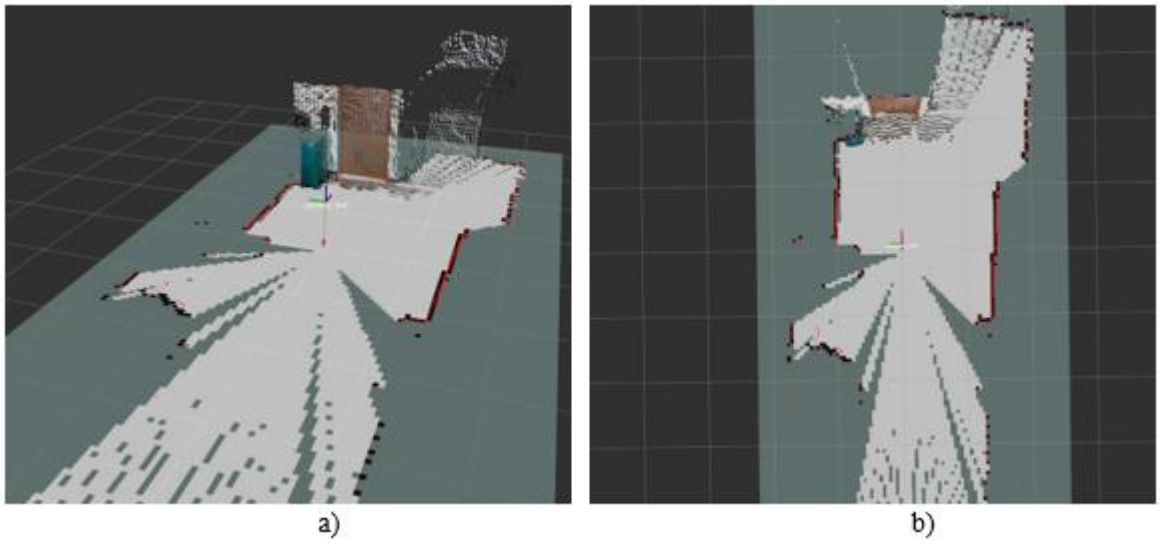


Figure 3-37: Combining 3D point cloud with 2D grid map. a) 3D view; b) top view

CHAPTER 4 PATH PLANNING

In the context of mobile robots, the path planning problem refers to finding a path (reference trajectory) from a given initial position to the desired goal position. This reference path must avoid any obstacle in the environment.

4.1 Introduction to RRT algorithm

A rapidly-exploring random tree (RRT) is an algorithm designed to efficiently search high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem. They easily handle problems with obstacles and differential constraints (nonholonomic) and high degrees of freedom. It has been widely used in autonomous robotic motion planning.

Given the start and goal position of the robot, the RRT algorithm is implemented or the path-finding purpose. A tree is grown from the start point (tree root), and it is strictly biased to the unexplored area of the working space till a certain node/branch on the tree reaches the goal point.

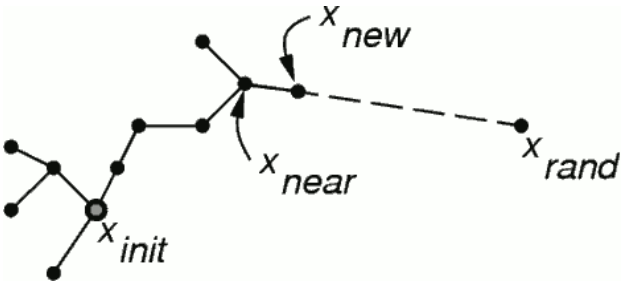


Figure 4-1: Illustrating the RRT algorithm

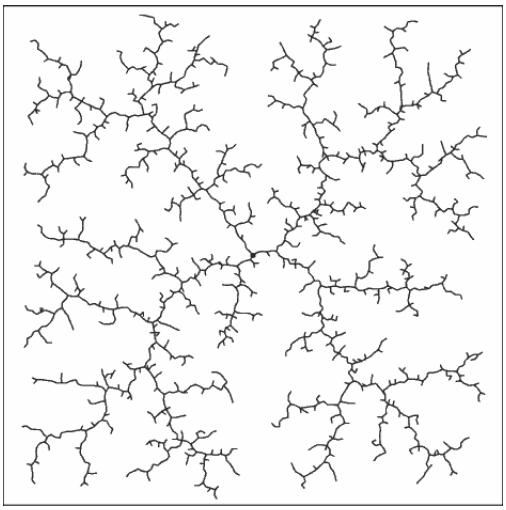


Figure 4-2: A Rapidly-exploring Random Tree searching a 2D space.

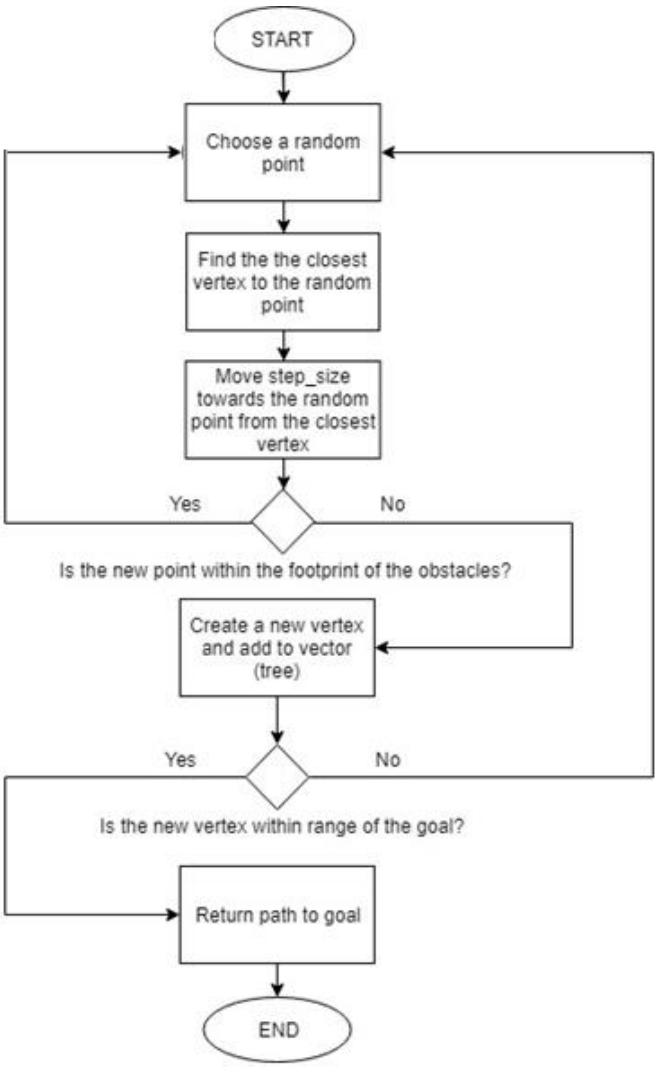


Figure 4-3: RRT algorithm flowchart

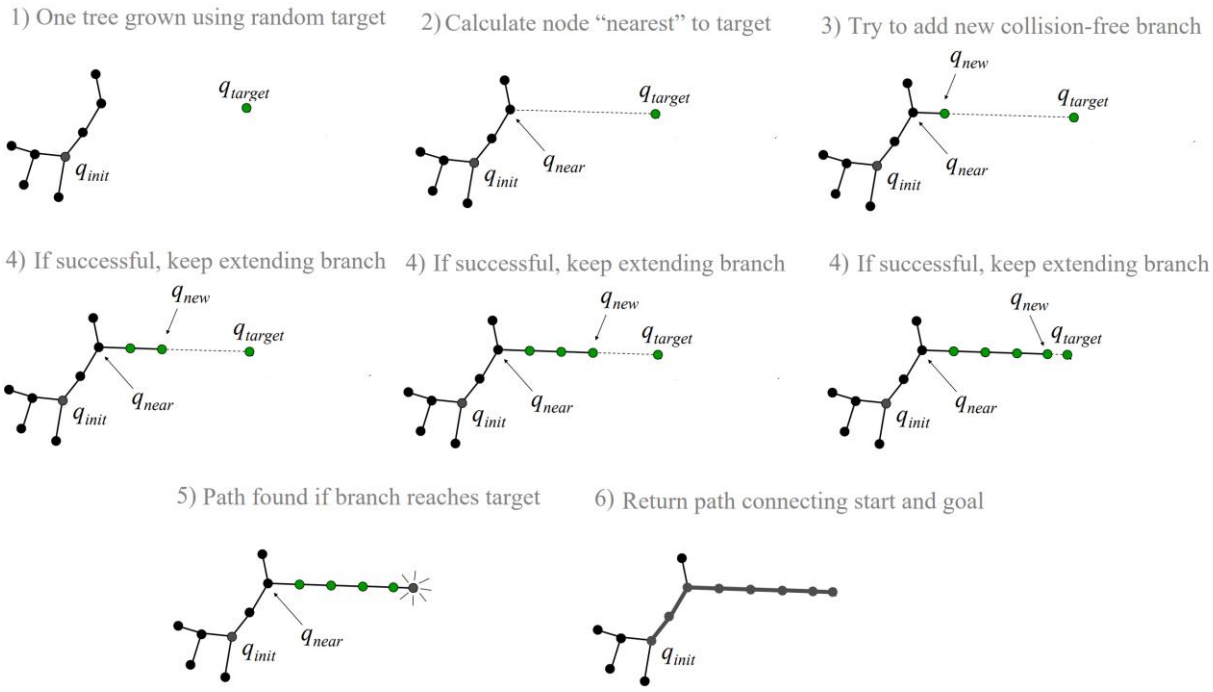


Figure 4-4: RRT-Connect iteration

4.2 Introduction to RRT* algorithm Bi-directional RRT algorithm

4.2.1 RRT* algorithm

RRT can find a path to a given goal fast but the optimality of the final path is not acceptable. The final path looks like a tree, so a post smooth method is required to reduce the cost to goal. For the purpose of solving the critical problem, the sub-optimality, we are going to use RRT*.

The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results. First, RRT* records the distance each vertex has traveled relative to its parent vertex. This is referred to as the cost of the vertex. After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined. If a node with a cheaper cost than the proximal node is found, the cheaper node replaces the proximal node. The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbor, the neighbors are again examined. Neighbors are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does indeed decrease, the neighbor is rewired to the newly added vertex. This feature makes the path smoother.

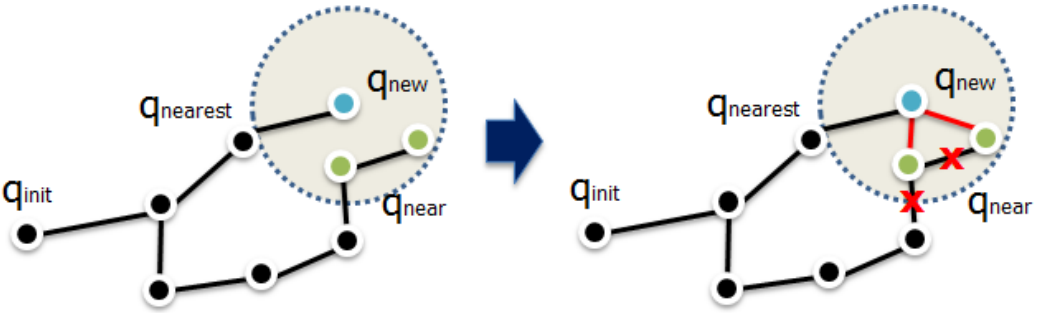


*Figure 4-5: The rewire procedure of RRT**

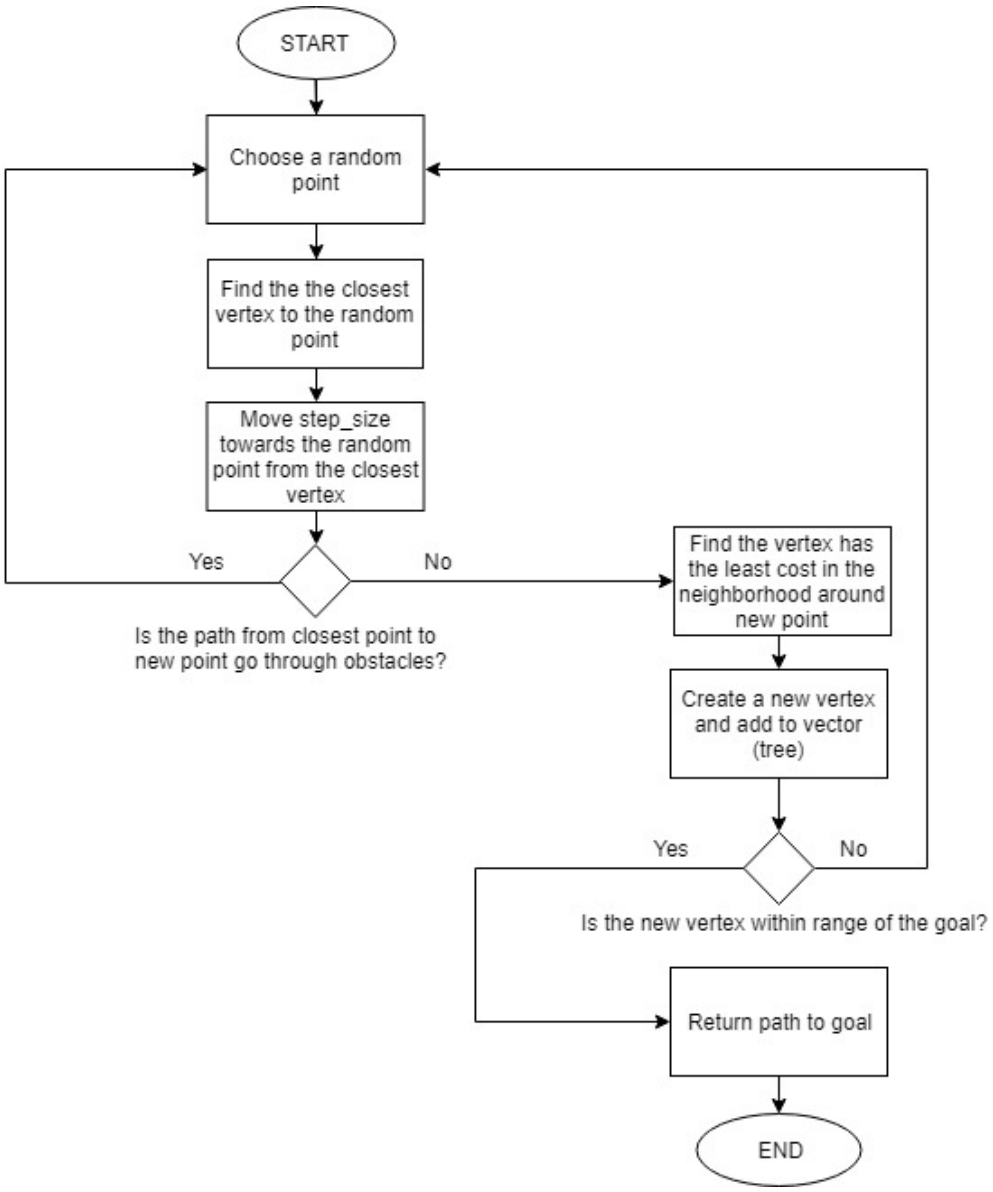


Figure 4-6: RRT algorithm flowchart*

5.2.2 Bi-directional RRT algorithm

For increasing the speed of finding a feasible path, Bi-directional RRT was introduced. Bi-directional RRT simply RRT algorithm in which the tree grows from both the starting point and the ending point. In other world, there will be two trees grow in the space. When two trees' nodes meet or close enough, a path is generated.

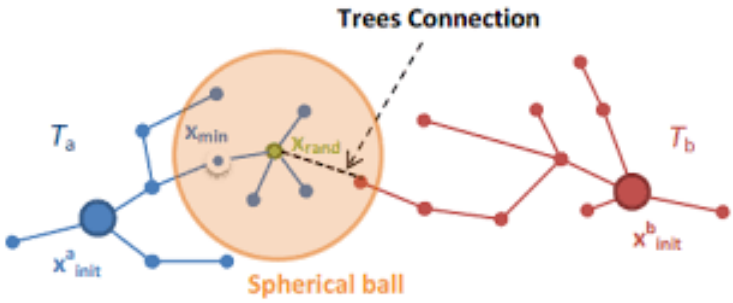


Figure 4-7: Illustrating the Bi-directional RRT algorithm

4.3 Experiment the RRT star algorithms and compare

Our robot will run in the indoor environment so we focus on the optimization more than the speed. That is the reason we will experiment and compare the RRT star and bi-directional RRT star only. The Matlab software is used for this purpose.

4.3.1 Executing RRT* algorithm

Parameters	Value
Step size (mm)	200
Goal radius (mm)	500
Maximum sampling point	800
Starting point (x, y)	(0, 0)
Goal point (x,y)	(2500, 4000)
Testing area (mm x mm)	5000 x 5000
Neighborhood (mm)	500

Table 4.1: RRT Parameters*

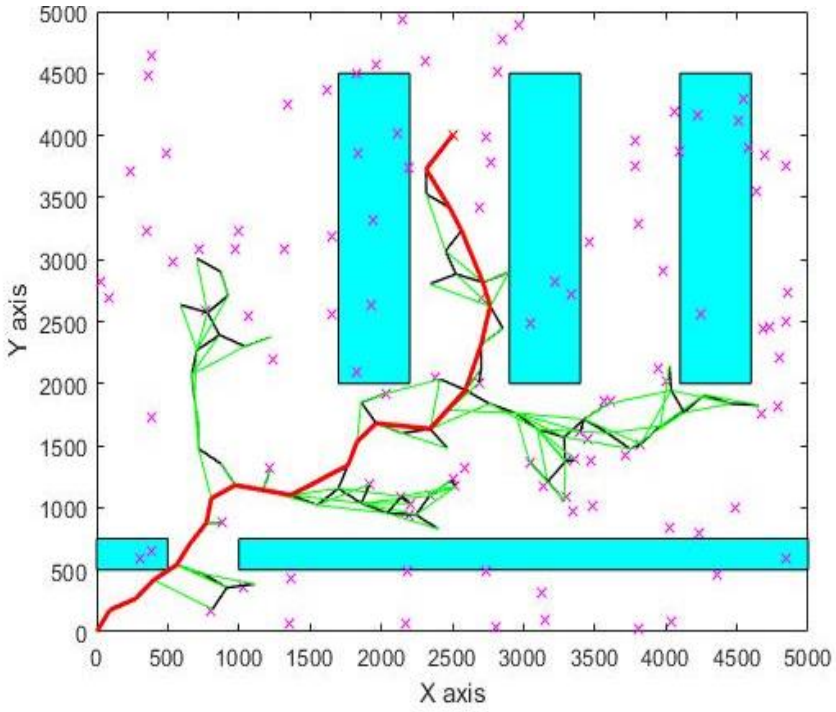


*Figure 4-8: Path planning in 2D using RRT**

Table 4.2: RRT Graph explanation*

Graphical objects	Meaning
Red line	Final found path
Black line	Branch of the searching tree
Green line	Rewired of branch for cost-optimal path
Blue rectangles	Obstacles
Red point	Goal
Violet points	Random sampling points

Comment: The path still looking bad with zig-zag form.

4.3.2 Executing bi-directional RRT star algorithm

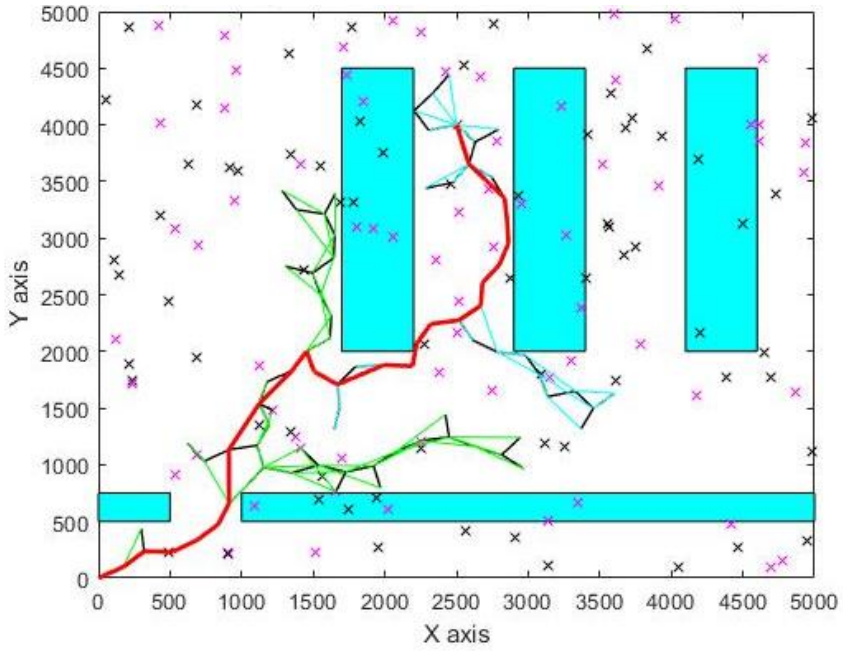


*Figure 4-9: Path planning in 2D using bi-directional RRT**

Table 4.3: Bi-directional RRT* graph explanation

Graphical objects	Meaning
Red line	Final found path
Black line	Branch of the searching tree
Green line	Rewired of branch for cost-optimal path for tree from starting point
Blue line	Rewired of branch for cost-optimal path for tree from goal point
Blue rectangles	Obstacles
Red point	Goal
Black points	Random sampling points from tree from goal point
Violet points	Random sampling points from tree from starting point

The parameters are the same with the single tree case.

Comment: The path still has the bad shape.

4.3.3 Path planning with neighborhood extended

In experiment process, we find a way to make the path more beautiful or in other word, including less but longer straight lines.

We made it by setting the neighborhood parameter significantly larger. This make the computation load becoming heavier but the path is now very straight forward. All the parameter are the same except the neighborhood now became 3000mm

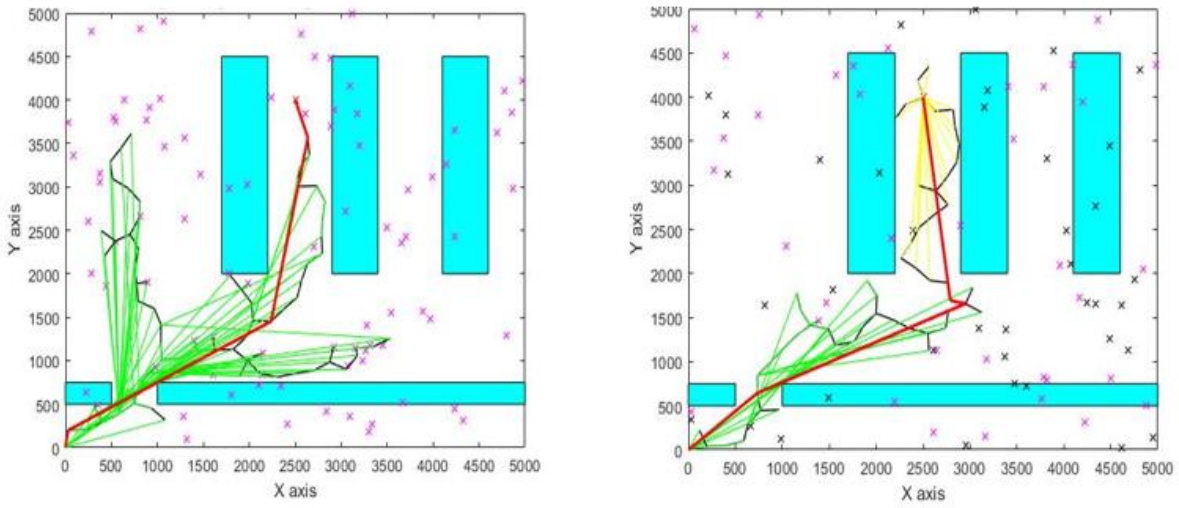
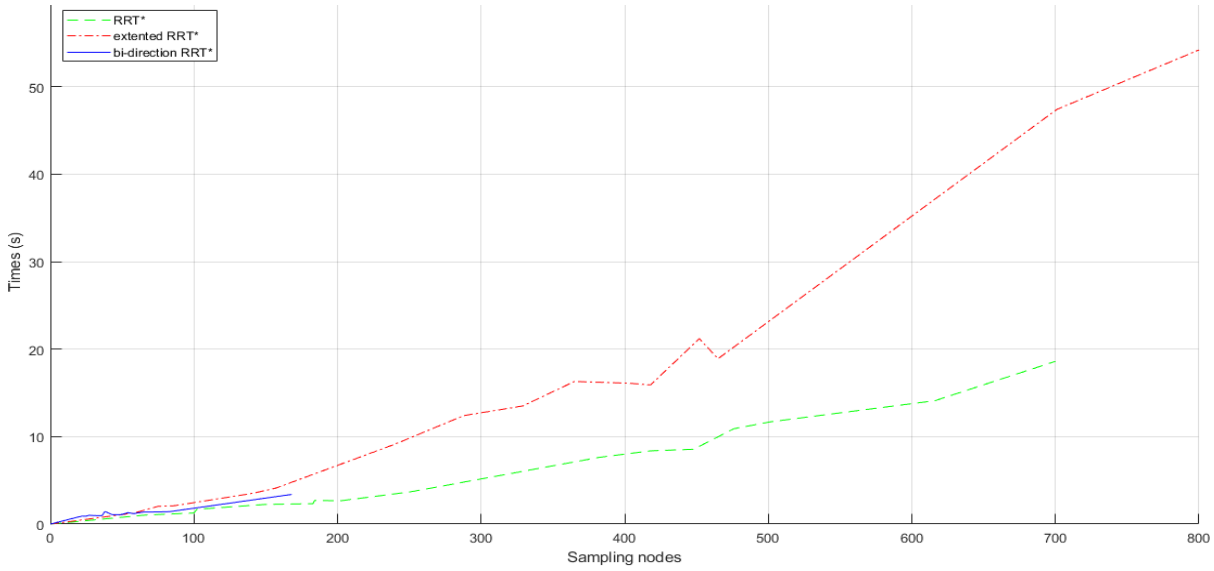


Figure 4-10: RRT and bi-directional RRT* with neighborhood extended*

Comment: The path is now more beautiful and shorter in both cases

4.4 Comparing each variation of RRT* algorithm



Comment:

1. With bi-directional RRT*, a feasible path can always be found and the maximum sampling nodes never exceed over 200.
2. The time consuming of extending neighborhood will increase dramatically with the increasing of sampling nodes.
3. Sometimes single direction RRT* can not return back a feasible path within the allowed number of sampling points

CHAPTER 5 MOTOR CONTROLLER

The navigation package on ROS will take control the AGV by sending data include “linear velocity” and “angular velocity”. Controller had to modify the motion of mobile robot base on these data, which means bring it to reality.

Our robot have two wheels connect directly to 2 motor with two free-rotating wheel – a common type of mobile robot:

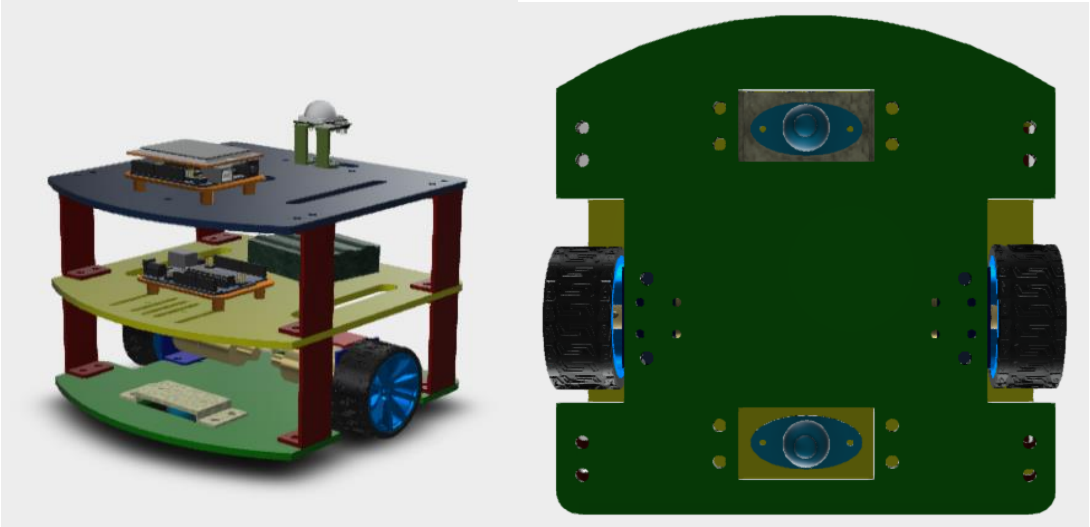


Figure 5-1 Two wheel and lead constructor

5.1 Analyzing control signal from ROS

5.1.1 Controlling method

The popular method controlling this kind of robot is converting the requirement motivation vectors in to two linear velocity vectors of each wheels. This will bring the problem to simpler question: controlling velocity of motors. Here is a simple example of this method:

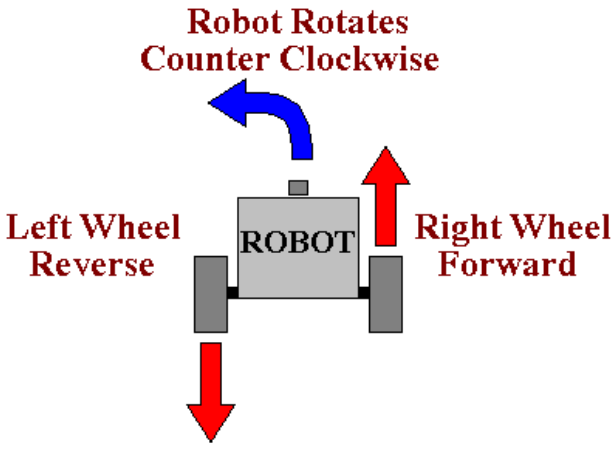


Figure 5-2 Example about controlling method

It is necessary building a dynamic model of the robot in order to have better imagination about motion of robot and wheels. It's can be hard to deal with the displacement and velocities of two wheels robot. So that we prefer using a simpler model in which we can think of the mobile robot as having only one wheel with velocity (v) and specified heading (∅). Having the equations to translate between the unicycle model and our two wheels model.

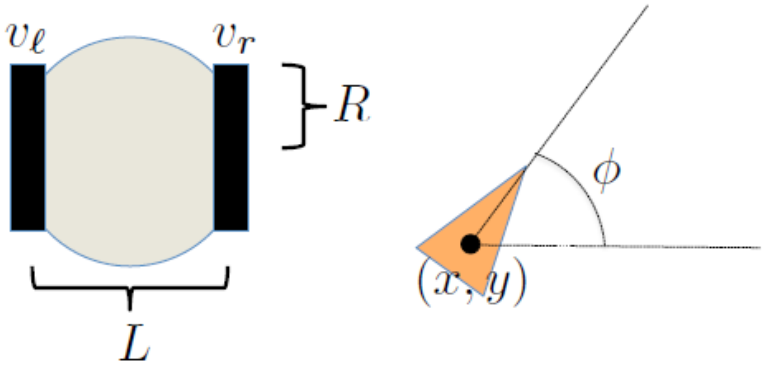


Figure 5-3 Unicycle model of mobile robot

Here: $\begin{cases} v_l, v_r: Wheels's\ velocity \\ R: Wheel's\ radius \\ L: Distance\ between\ two\ main\ wheels \\ x, y, \emptyset: Robot's\ position\ parameters \end{cases}$

From the unicycle model, we build the equations for translational and angular velocities. The forward velocity is calculated as average of wheels velocities:

$$V = \frac{R}{2}(v_r + v_l)$$

The rotation velocity is the different of wheels velocities divide by the radius of rotation. Here, the wheels velocities are v_l, v_r, which is velocity of each wheels and radius is L that show the distance between two wheels.

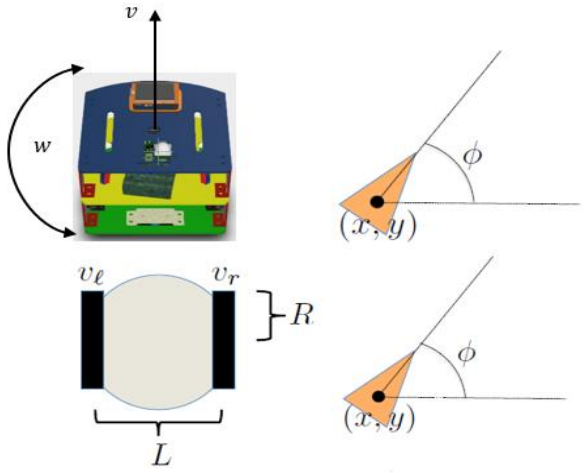


Figure 5-4 Robot's dynamic model

Thus:

$$w = \frac{R}{L}(v_r + v_l)$$

We generate the motion equation depend on those models:

Dynamic equations:

$$\begin{cases} \dot{x} = v \times cos\emptyset \\ \dot{y} = v \times sin\emptyset \\ \dot{\emptyset} = w \end{cases} (1) \; ; \; \begin{cases} \dot{x} = \frac{R}{2}(v_l + v_r) \times cos\emptyset \\ \dot{y} = \frac{R}{2}(v_l + v_r) \times sin\emptyset \\ \dot{\emptyset} = \frac{R}{L} \times (v_r - v_l) \end{cases} (2)$$

From eq. (1) and eq. (2):

$$\begin{cases} \frac{R}{2}(v_l + v_r) \times cos\emptyset = v \times cos\emptyset \\ \frac{R}{2}(v_l + v_r) \times sin\emptyset = v \times sin\emptyset \\ \frac{R}{L} \times (v_r - v_l) = w \end{cases} \quad (3)$$

We have the final equation:

$$\begin{cases} v_r = \frac{2 \times v + w \times L}{2R} \\ v_l = \frac{2 \times v - w \times L}{2R} \end{cases} \quad (4)$$

Here:

$$\begin{cases} v_l, v_r: velocity\ of\ left\ wheel\ and\ right\ wheel\ (mm/s) \\ v: linear\ velocity\ (mm/s) \\ w: angular\ velocity\ (rad/sec) \\ L: Distance\ between\ two\ wheels\ (mm) \\ R: Radius\ of\ wheels\ (mm) \end{cases}$$

5.1.2 Convert from linear to rotation speed

Now, the requirements is taking control rotation speed and direction of each wheel follow the requirements from the previous method. From linear velocity values getting from equation (3) in previous part, we convert it to the rotation velocity by the equation:

$$w = v * \frac{E_r}{2\pi R} \qquad (4)$$

Here: $\begin{cases} w: Rotation\ velocity\ (rad/s) \\ v: Linear\ velocity\ (mm/s) \\ E_r: Encoder\ resolution\ \left(\frac{pulses}{evolution}\right) \\ R: Radius\ of\ wheels \end{cases}$

Base on measurement and calibrations, we define the parameters of robot as:

- Distance between two wheels : L = 207 (mm)
- Radius of wheels : R = 31 (mm)
- Encode resolution : E_r = 440 (pulses/revolution)

Block diagram of controlling function:

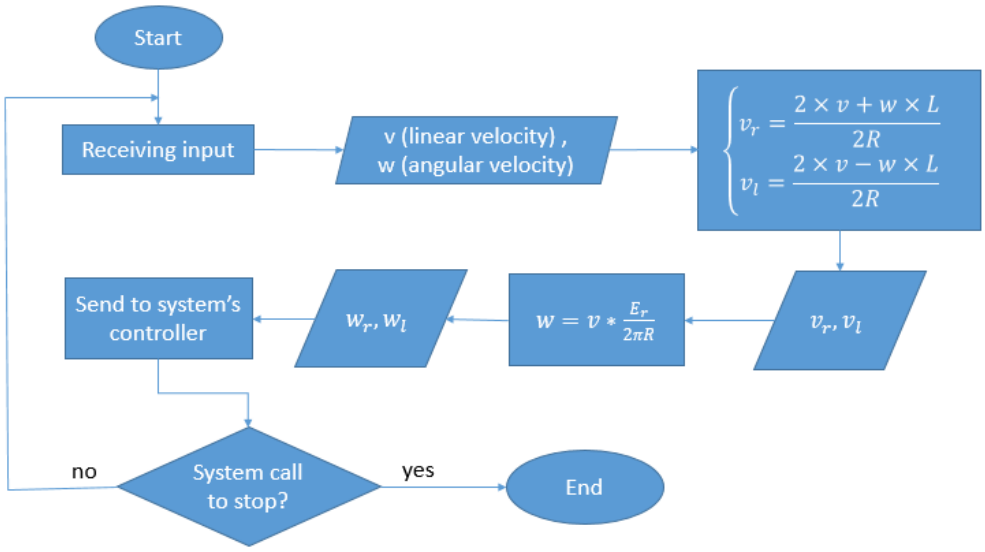


Figure 5-5 Controlling function's block diagram

5.1.3 Arduino – Raspberry communication

Because of using two separately microcontroller Arduino and Raspberry, we have to create a connection to help Arduino receiving motion control signal from ROS in Raspberry.

Rosserial is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics and services over a character device such as a serial port or network socket. This package supplies communicating methods for different hardware to ROS.

From rosserial's library, we use *rosserial_arduino* for the requirement from project. This package contains Arduino-specific extensions required to run rosserial_client on an Arduino. It is meant to demonstrate how easy it is to integrate custom hardware and cheap sensors into ROS project using an Arduino.

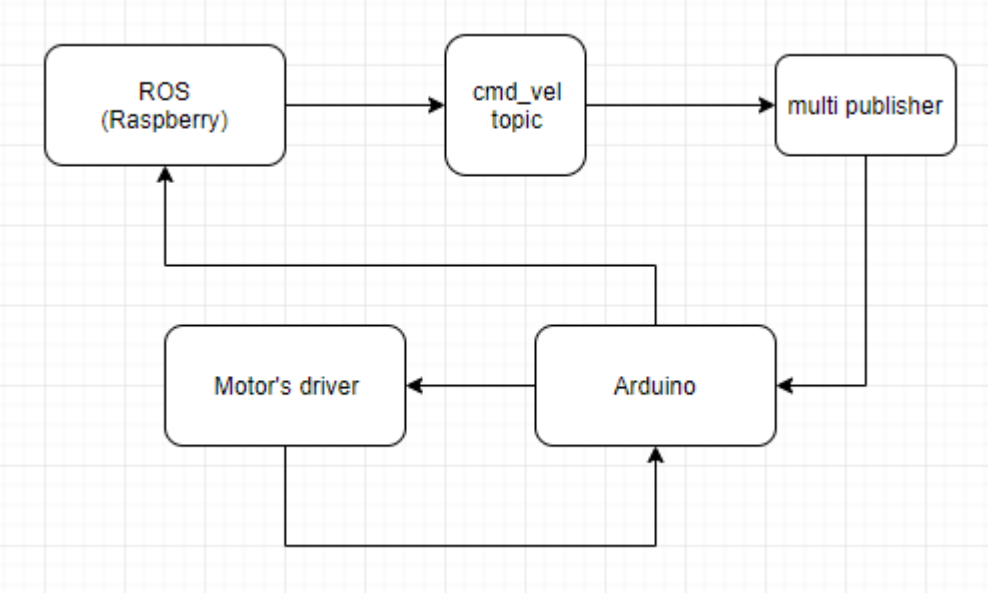


Figure 5-6 Publishing and subscribing velocity data with rosserial_arduino

Velocity data is publish by ROS to *cmd_vel* topic, which is subscribed by another topic named *multi_publisher*. This topic will continuously subscribe and publish data from *cmd_vel* to Arduino. The connection method base on UART communication with baud rate is 115200 (mean that maximum transferring speed is 115200 bits per second). From these data. Arduino's program will convert to input signal to control motors via driver.

5.2 Designing controller

In working environment, there are many things can affect to the working process of robot, such as friction between road and wheels, load, air resistant force, energy… To keep robot performing as close as much with our expected, we have to add a "close-loop controller"

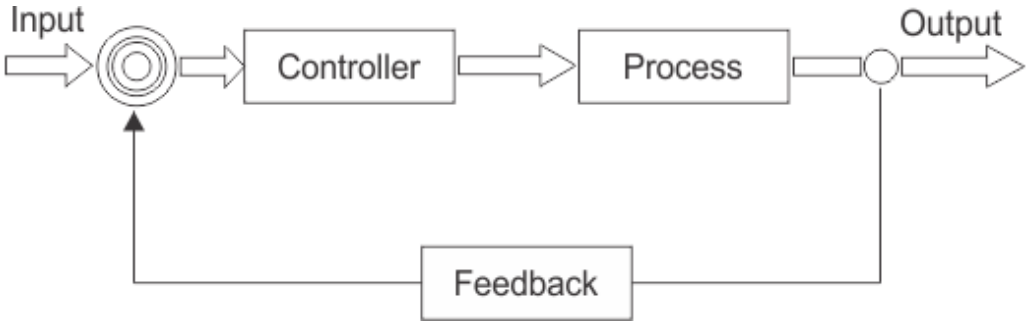


Figure 5-7: Basic Closed-loop controller system

In this project, we're going to test the system with some basic controller: PID controller and Fuzzy controller. Finally, based on the result we decide which controller is the most suitable for robot

5.3 PID controller

PID is a controller that is widely used in industrial control system. This base on the three important parts that make the controller: Proportional part, Integral part and Derivative part to create an algorithm, which will modify the control signal to make output signal as closed as much with set point.

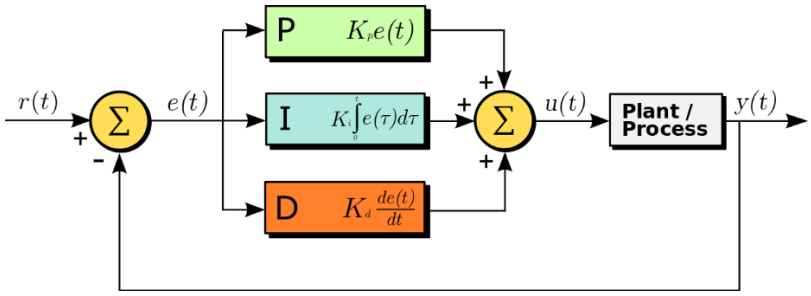


Figure 5-8: Basic PID controller's blog diagram

5.3.1 Characteristic of PID

The mathematical form of PID controller:

$$u(t) = Kp \times e(t) + Ki \int_0^t e(\tau)d\tau + Kd \times \frac{de(t)}{dt} \quad (6.1)$$

Where:

Kp, Ki, Kd: PID's parameters

$e(t)$: Error between control signal and feedback signal

$u(t)$: Output signal of PID controller

Proportions part

A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable. In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller.

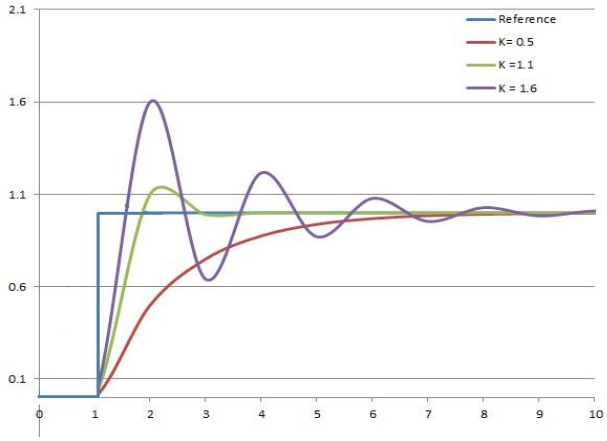


Figure 5-9: The response to step change of SP vs time, for three values of Kp

Integral part

The integral in a PID controller is the sum of the instantaneous error over time and gives the accumulated offset that should have been corrected previously. Since the integral term responds to accumulated errors from the past, it can cause the present value to overshoot the setpoint value.

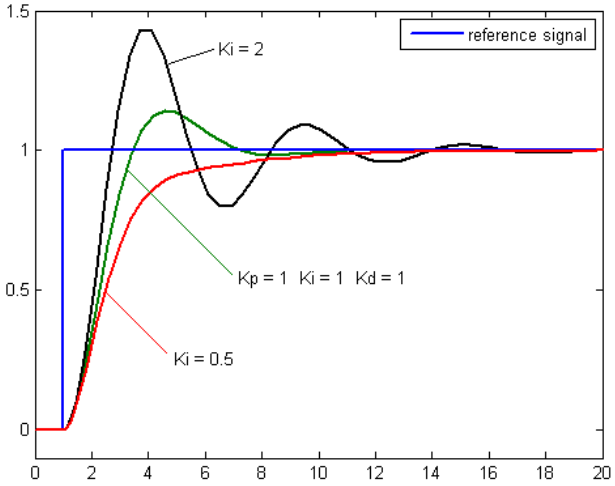


Figure 5-10: The response to step change of SP vs time, for three values of Ki

Derivative part

The derivative of the process error is calculated by determining the slope of the error over time and multiplying this rate of change by the derivative gain Kd. The magnitude of the contribution of the derivative term to the overall control action is termed the derivative gain, Kd.

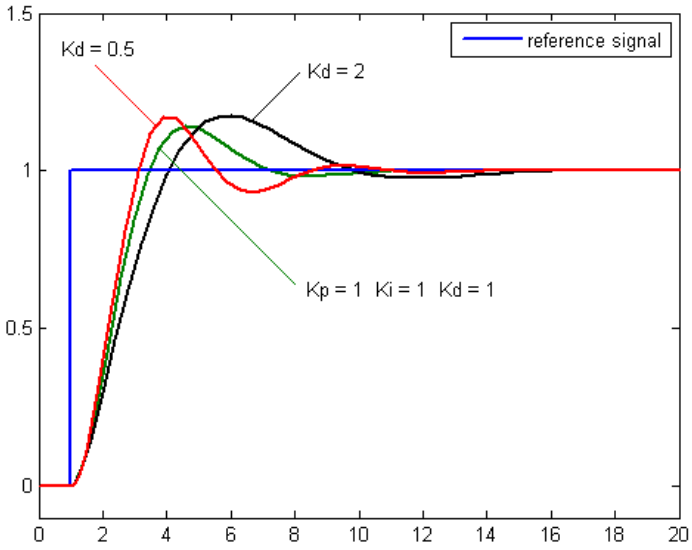


Figure 5-11: The response of to step change of SP vs time, for three values of Kd

5.3.2 PID turning method

PID parameters Ki, Kp, Kd can be turned by some process like “trial-and-error”, using Mathlab with already known transfer function, or “auto-tune process” in which our calculating depends on system’s reaction with control signal.

In this project, we choose “Relay-based auto tuning” as the turning method of our PID control base on its automation, quickly generating result and easily modifying. It also minimizes the possibility of operating the plant close to the stability limit.

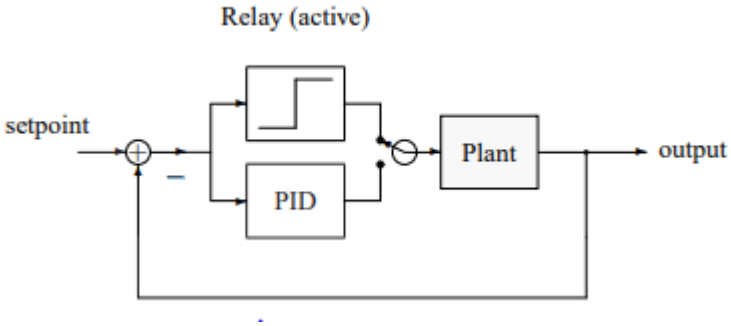


Figure 5-12: Blog diagram of relay auto-tune method

Relay-based auto tuning process's key ideal is to temporarily swap a simple relay for the PID controller in the feedback loop. As it turns out, under relay feedback, most plants oscillate with a modest amplitude fortuitously at the critical frequency:

Step 1: Using a random constant input, u_0 to determine the corresponding steady state output y_0.

Step 2: Setting the setpoint as y_0

Step 3: Running controller in relay mode, which have the input signal as:

$$u = \begin{cases} u_0 + h \; if \; e \geq 0 \\ u_0 - h \; if \; e < 0 \end{cases} \quad (6.2)$$

Where:

- u: Control signal
- e: Difference between y_0 (setpoint) and y_t (output signal)
- h: Amplitude of input signal

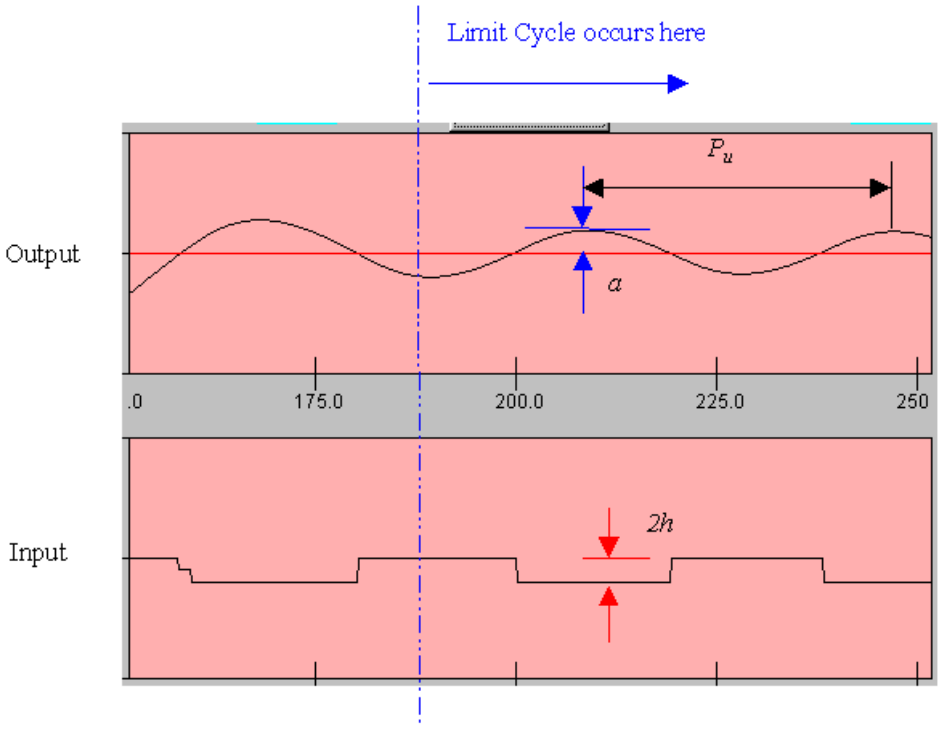


Figure 5-13: Relay-based method signal

Step 4: From the signals, define the output amplitude a of reaction signal and ultimate period P_u. Then we can approximate the ultimate gain K_u as:

$$Ku = \frac{4h}{\pi a} \quad (6.3)$$

After having ultimate gain Kp and ultimate period $P_{u,}$ applying them with Ziegler-Nichols tuning method to turning PID parameters:

Table 5.1: Ziegler-Nichols tuning parameters:

Specification	K_p	K_i	K_d
Original	$0.60\times K_u$	$2\times K_p / T_u$	$K_p\times T_u/8$
Little overshoot	$0.33\times K_u$	$2\times K_p / T_u$	$K_p\times T_u/3$
No overshoot	$0.2\times K_u$	$2\times K_p / T_u$	$K_p\times T_u/8$

5.3.3 Result of "Original Ziegler-Nichols tuning method" process:

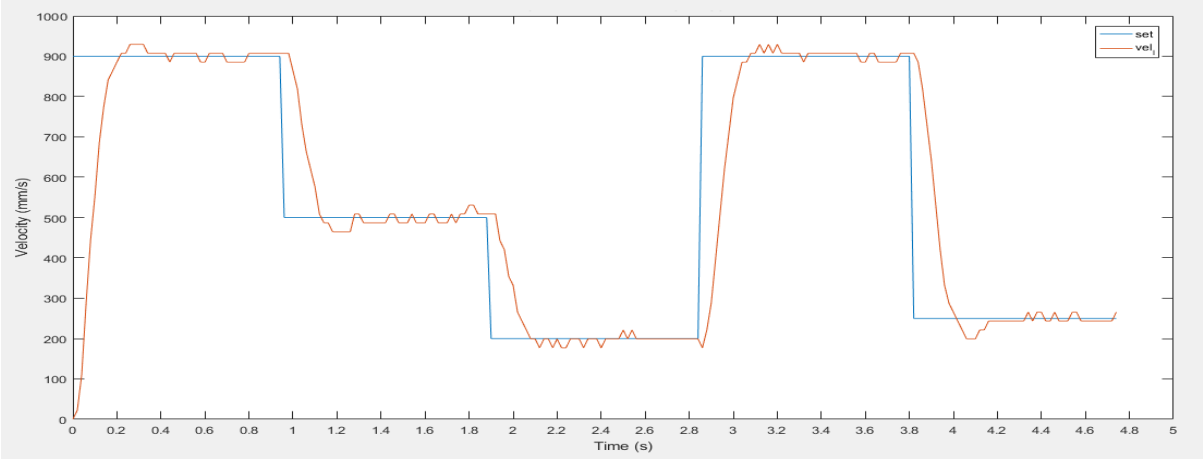


Figure 5-14: Original Ziegler-Nichols: Left wheel's output signal

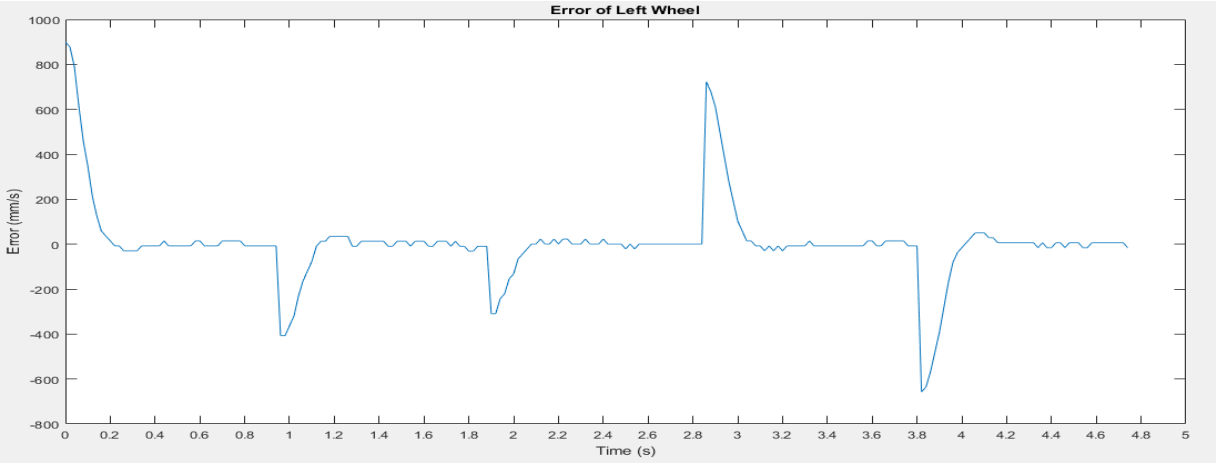


Figure 5-15: Original Ziegler-Nichols: Left wheel's error

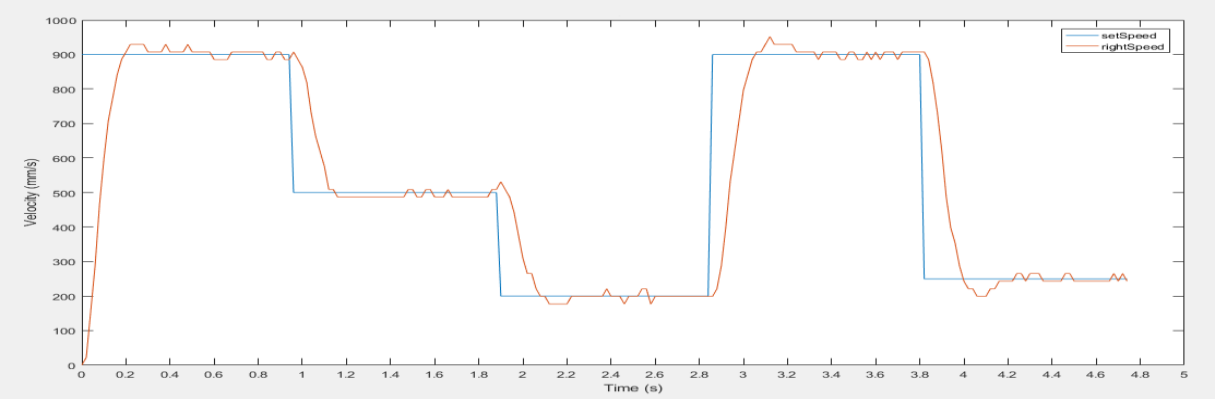


Figure 5-16: Original Ziegler-Nichols: Right wheel's output signal

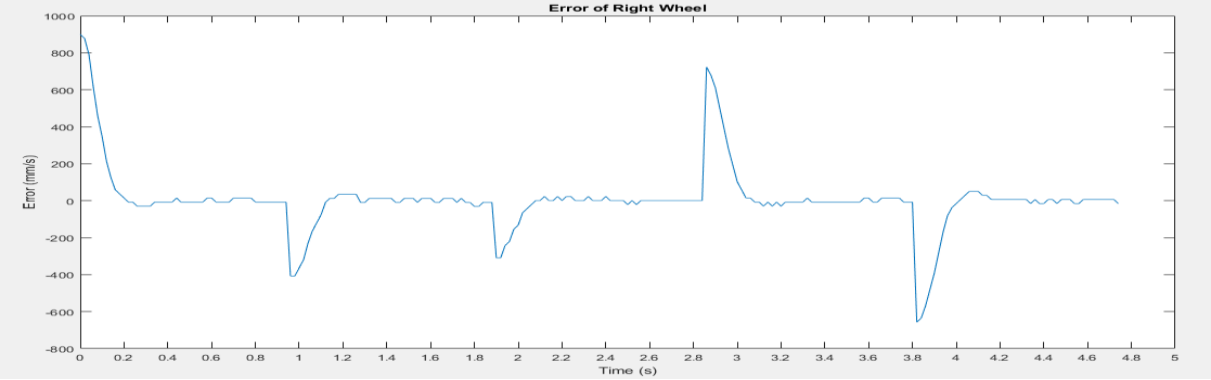


Figure 5-17: Original Ziegler-Nichols: Left wheel's error

Conclusion: This type of controller really fit with the system. The overshoot is low (about 3%) when raising the set point, but when decreasing speed set point it has some over shoot (about 8-9%). Settling time is around 0.25 second, peek time is about 0.2 second.

5.3.4 Result of "None overshoot Ziegler-Nichols tuning method" process:

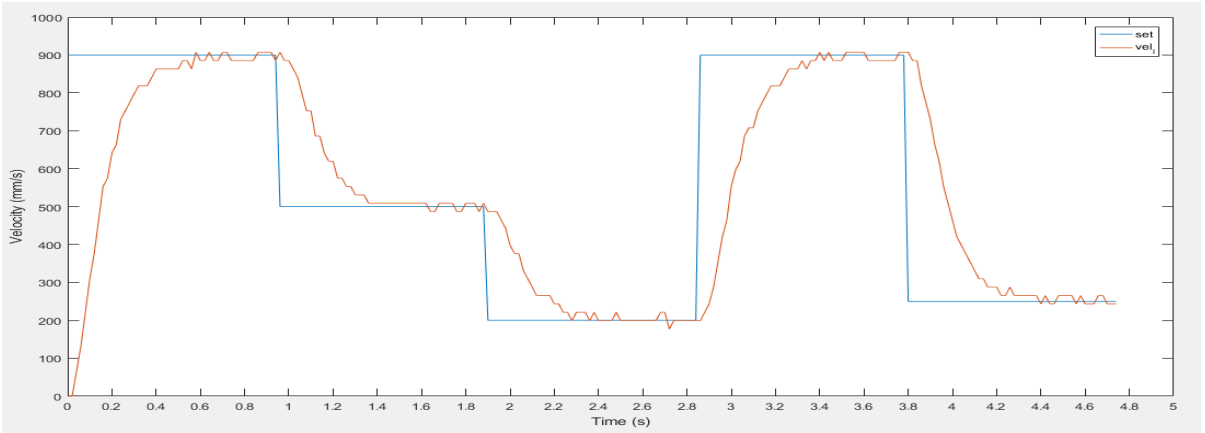


Figure 5-18: None overshoot method: Left wheel's output signal

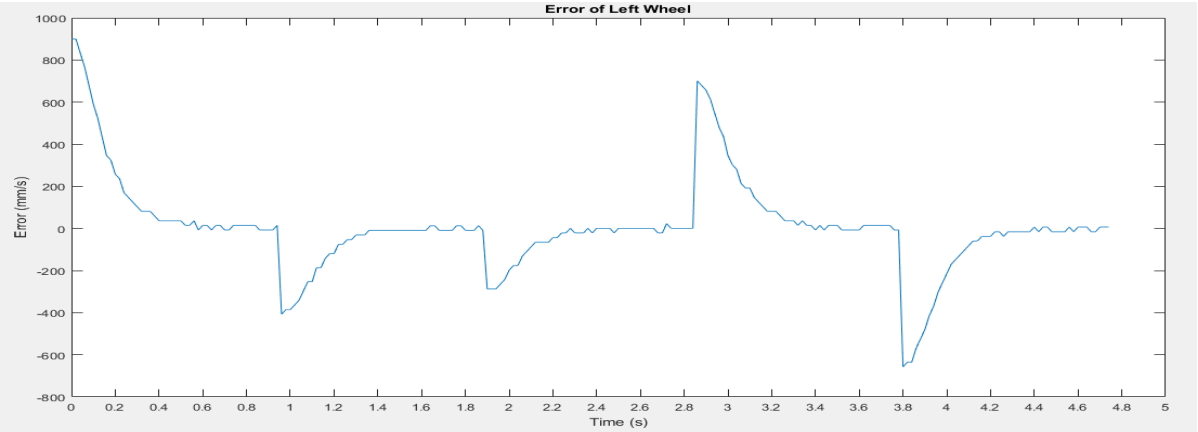


Figure 5-19: None overshoot method: Left wheel's error

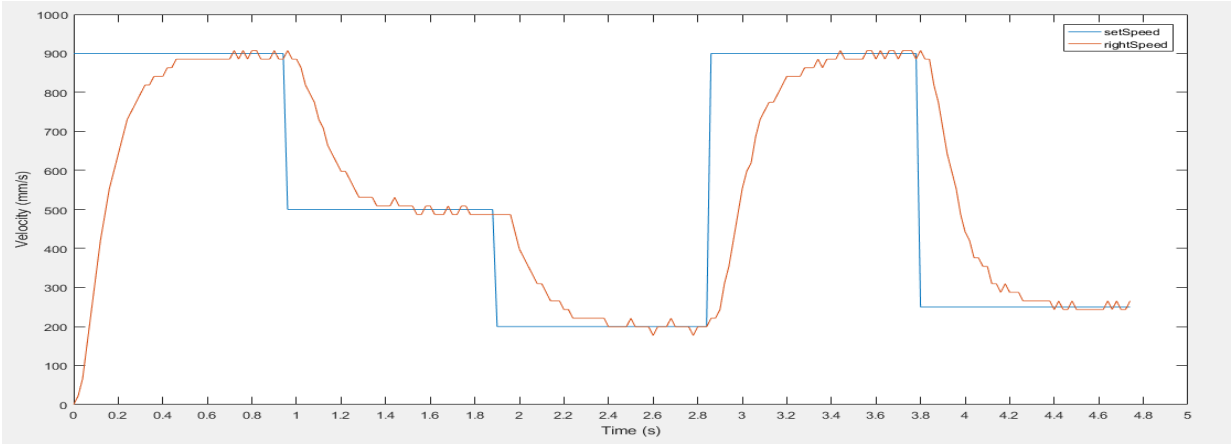


Figure 5-20: None overshoot method: Right wheel's output signal

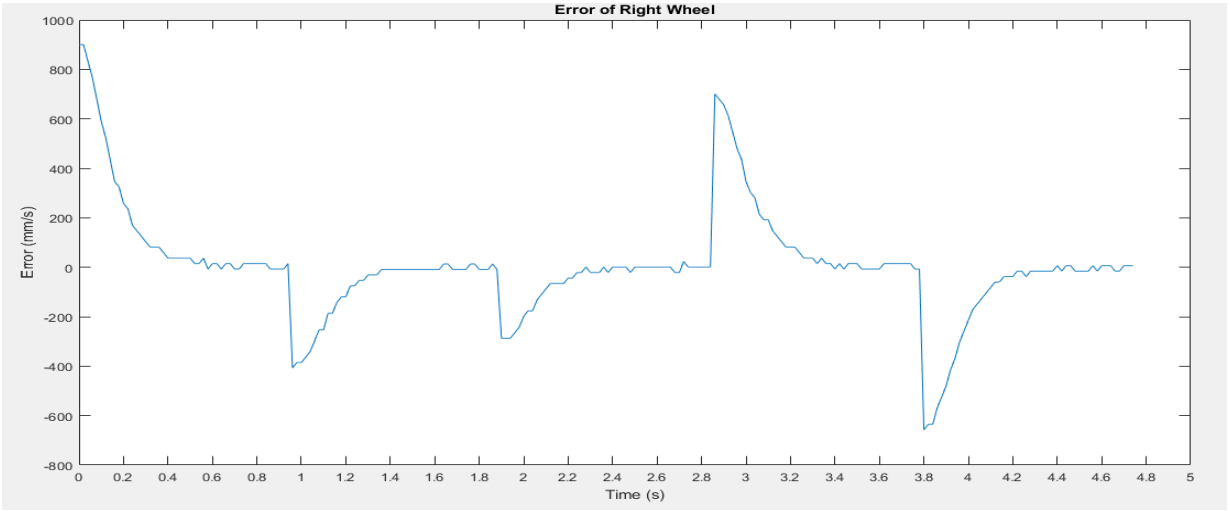


Figure 5-21: None overshoot method: Right wheel's error

Conclusion: The overshoot value is almost eliminated. Settling time is around 0.4 second, which is too slow compare with basic Ziegler – Nichols method.

5.4 Fuzzy controller

The Fuzzy controller base on Fuzzy logic. The word “Fuzzy” refers to the fact that the logic involved can deal with concepts that cannot be expressed as the "true" or "false" but rather as "partially true". The advantage of this logic is that the solution for the problem can be shown in the term that understandable with human operators so that their experience can be applied in the design of the controller.

Fuzzy controller consists of an input stage, a processing stage, and an output stage. The input stage is a group of inputs from sensors, switches, thumbwheels… The processing stage links data from input stage with its rules then generate results for each case. Finally, the output stage converts the combined result back into a specific control output value.

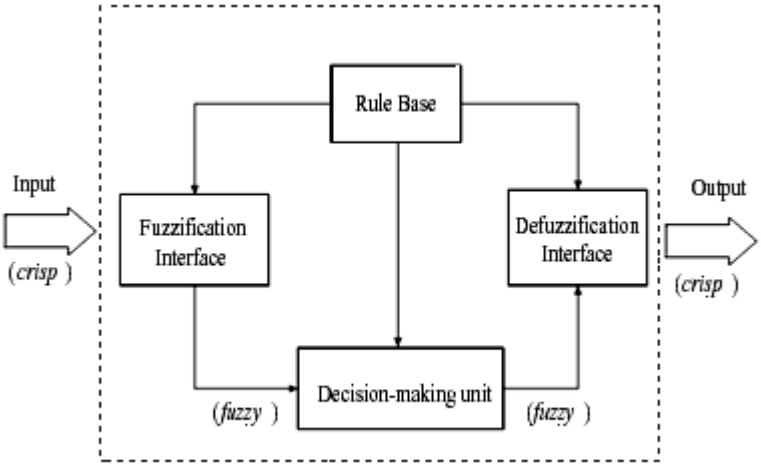


Figure 5-22: Basic Configuration of Fuzzy logic system

5.4.1 Designing Fuzzy logic controller

We follow these steps:

1. Identification of variables: Identified the input, output and state variables
2. Fuzzy subset configuration: The universe of information is divided into the number of fuzzy subsets and each subset is assigned a linguistic label.
3. Obtaining a membership function: Obtain the membership function for each fuzzy subset that we get in the previous step

4. Fuzzy rule base configuration: Create the fuzzy rules by assigning the relationship between the input stage and output stage.
5. Fuzzification: Initiating fuzzification process.
6. Combining fuzzy outputs: By applying fuzzy approximate reasoning, locate the fuzzy output and merge them.
7. Initiating defuzzification: Initiate the defuzzification process to form a crisp output.

Based on our experience about the mobile robot, we designed a fuzzy controller with one input and one output as shown bellowed:

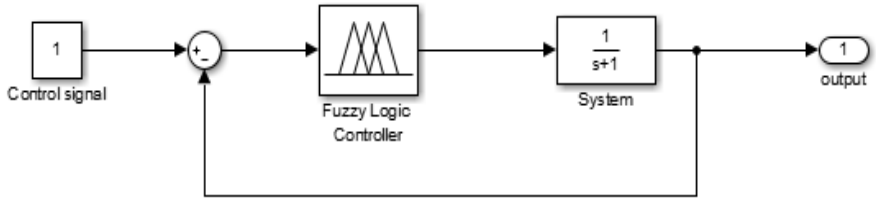


Figure 5-23: Fuzzy controller in motor system

In this Fuzzy controller, the input signal is the error between controlling speed and the realize speed of the motor. The output signal is extra value to add in the PWM control signal. From the feedback signal from the encoder, the Fuzzy controller will modify the PWM signal to get the output to reach the required signal.

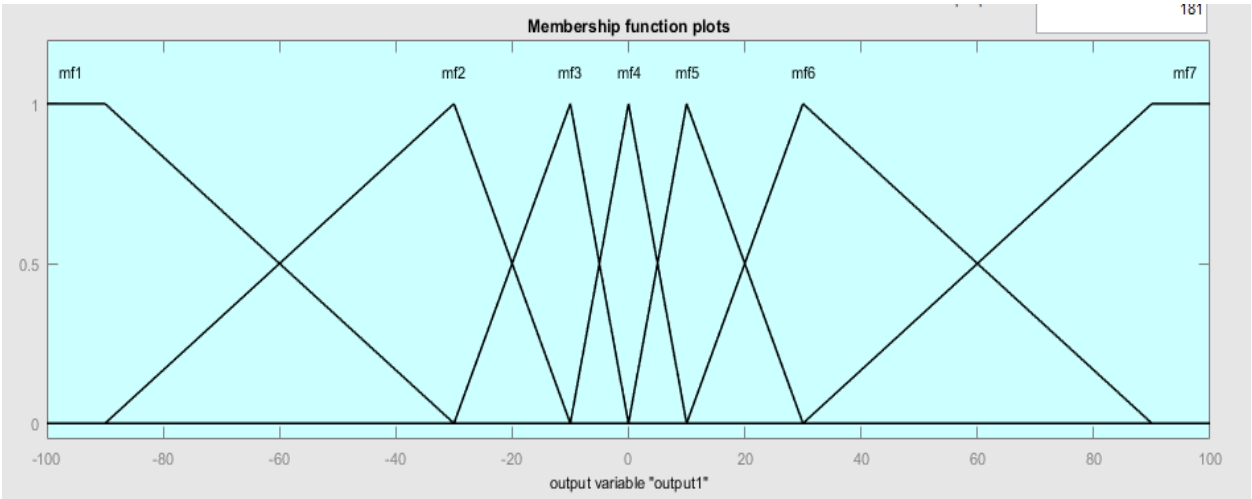


Figure 5-24: Fuzzy controller's output membership function

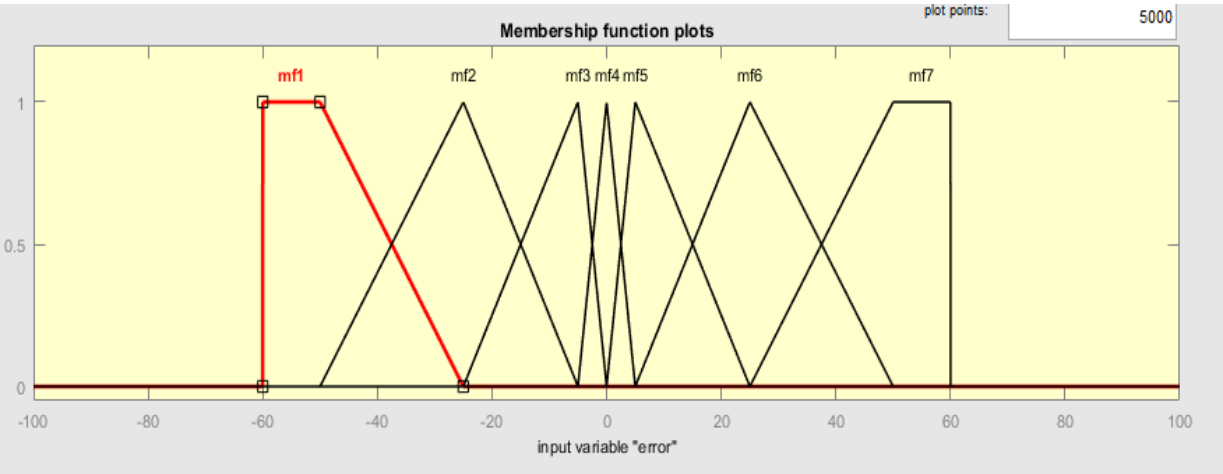


Figure 5-25: Fuzzy controller's input membership function

Fuzzy rule base configuration:

- **If** error is *negative large* **then** ouput is *add negative much*
- **If** error is *negative medium* **then** ouput is *add negative medium*
- **If** error is *negative small* **then** ouput is *add negative small*
- **If** error is *zero* **then** ouput is *add zero*
- **If** error is *positive small* **then** ouput is *add positive small*
- **If** error is *positive medium* **then** ouput is *add positive medium*
- **If** error is *positive large* **then** ouput is *add positive large*

5.4.2 Fuzzy controller's performance

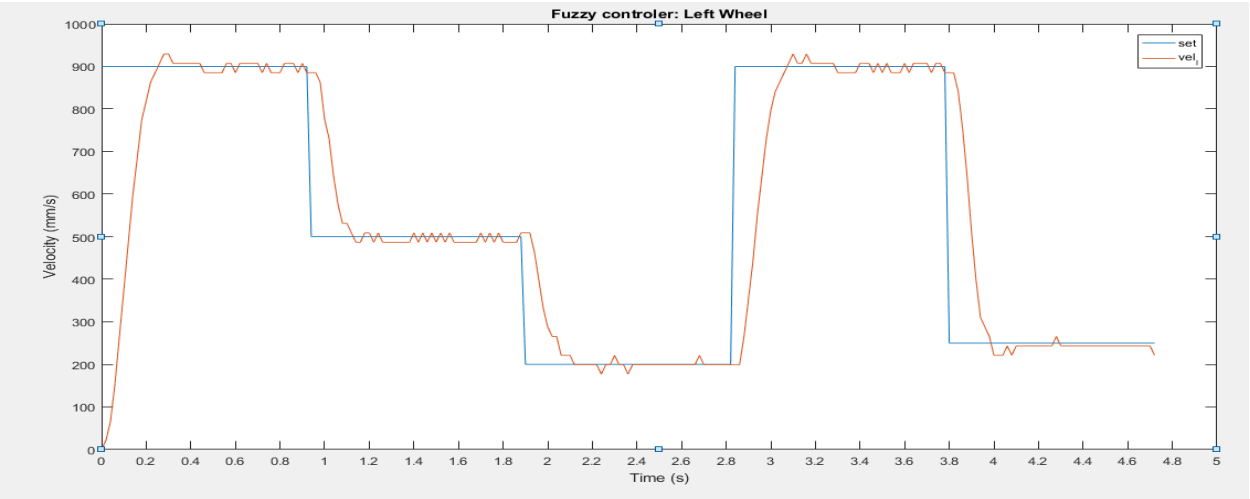


Figure 5-26: Left wheel's speed

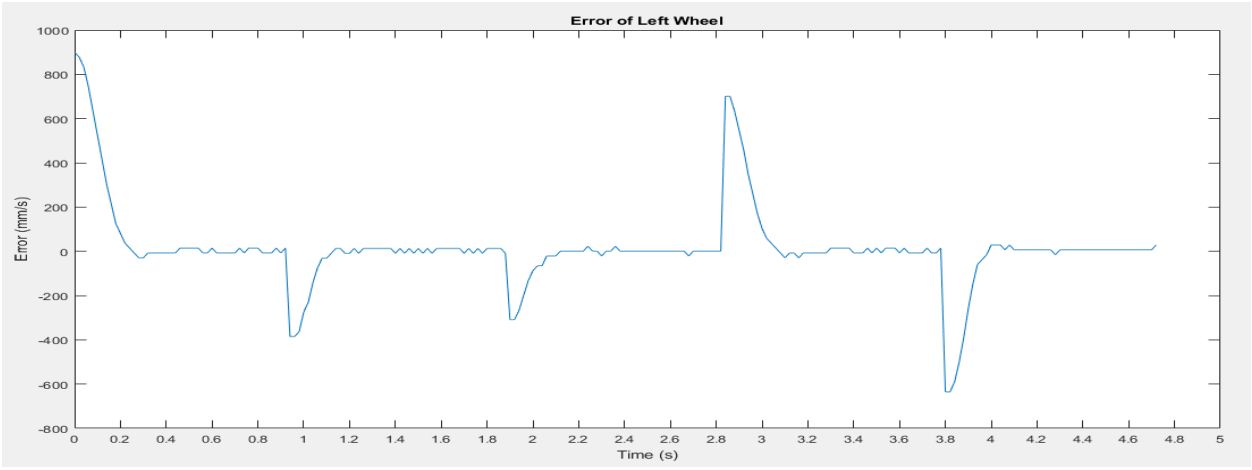


Figure 5-27: Left wheel's error

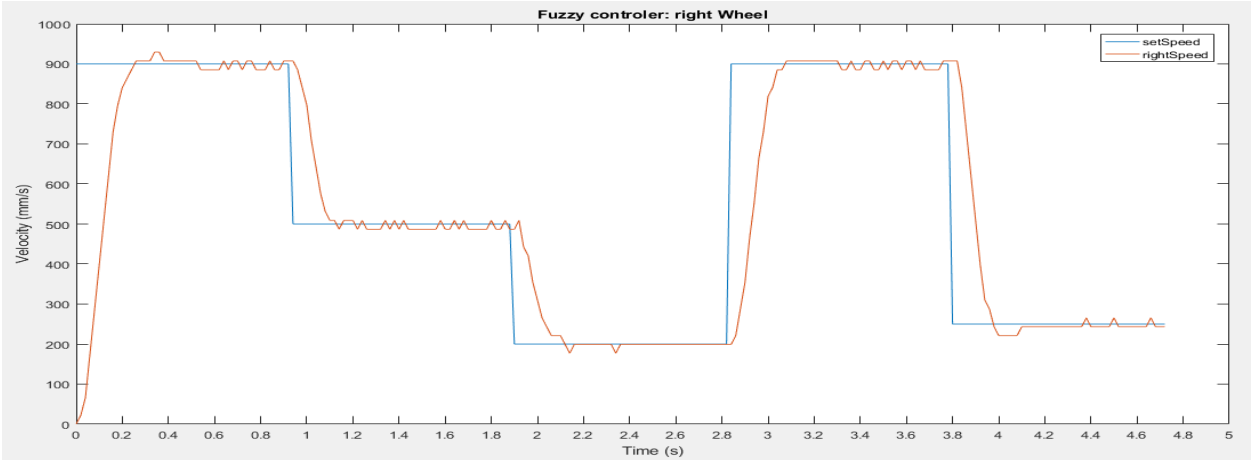


Figure 5-28: Right wheel's speed

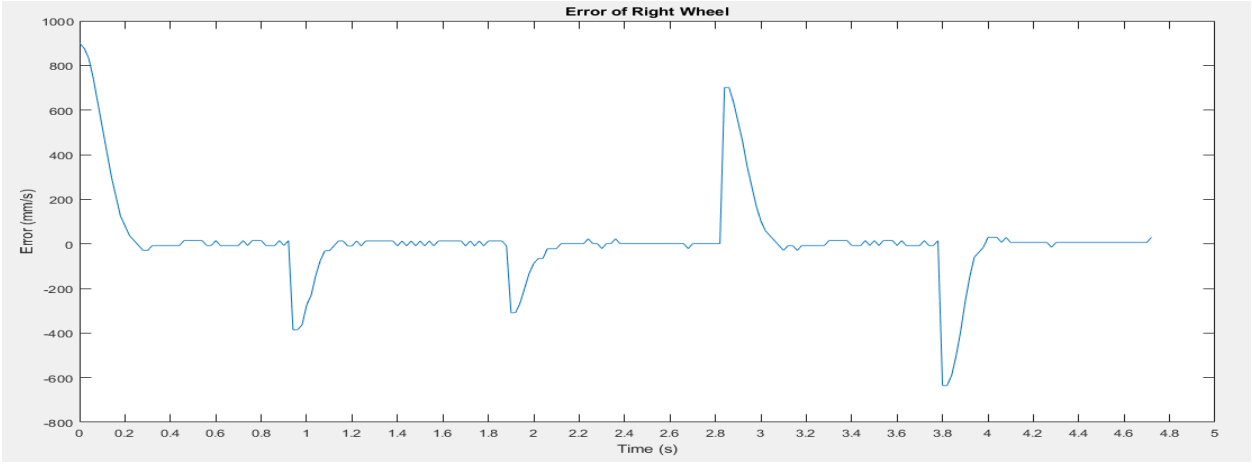


Figure 5-29: Right wheel's error

Conclusion: The Fuzzy controller have smaller overshoot than PID controller, especially in the decreasing speed process, also the stability is very good. However, the peak time and settling time (around 0.2 second) that is faster than PID's ones.

5.5 Self-tuning Fuzzy PID Controller

From the ideal about creating a PID self-tuning controller, we apply the Fuzzy logic into the PID Controller to design a Self-tuning Fuzzy PID Controller.

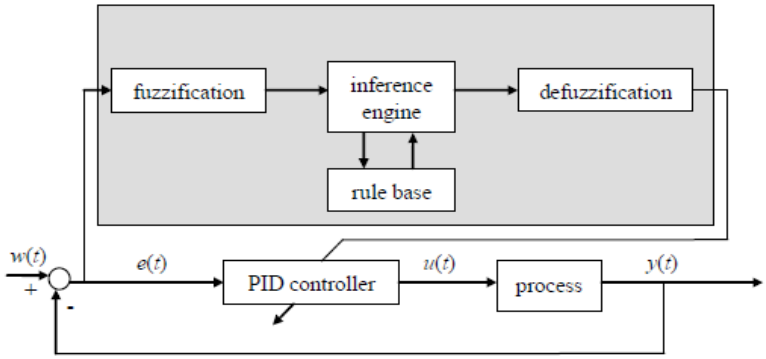


Figure 5-30: Self-tuning Fuzzy PID Controller

Basically, our controller is a PID controller, but has the proportional parameter and integral parameter are continuously tuned by Fuzzy logic, based on feedback signal:

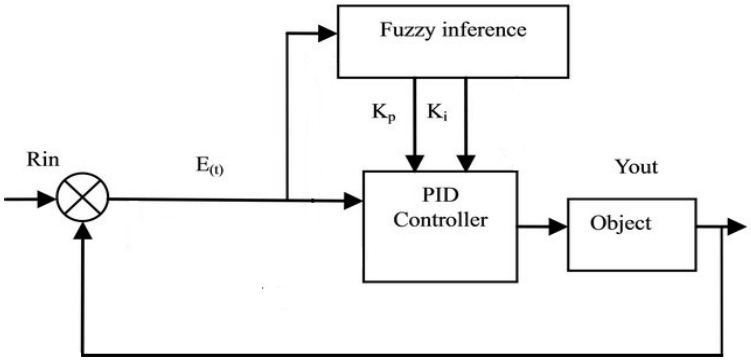


Figure 5-31: Self-tuning Fuzzy PID Controller

Based on the result of previous experience about Fuzzy and PID controller, we choose range of Fuzzy logic input and output as:

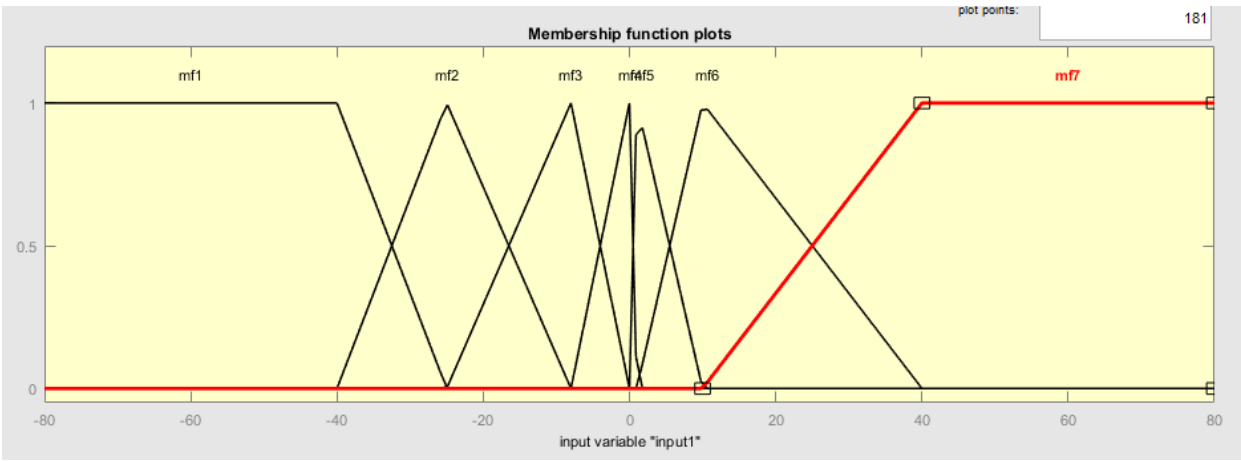


Figure 5-32: Fuzzy's input: Error

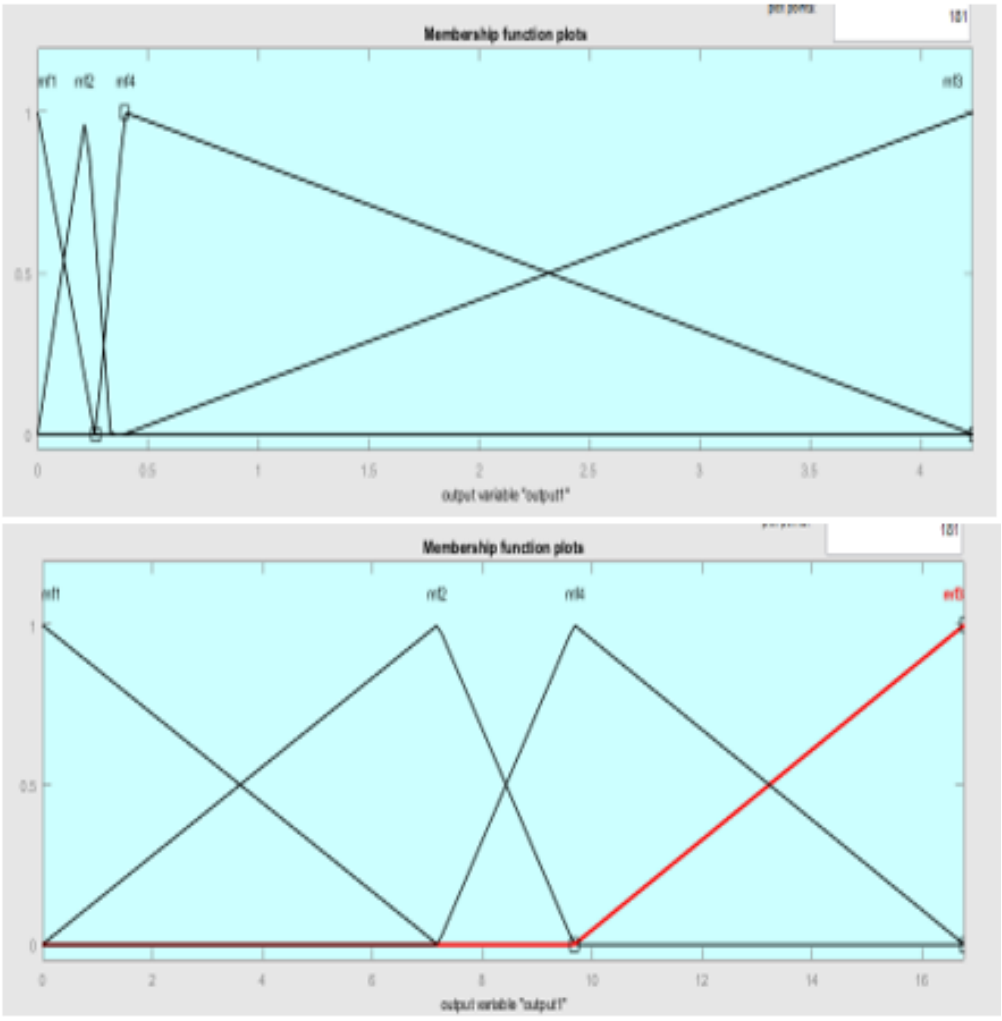


Figure 5-33: Fuzzy logic's Output member function: Kp (left) and Ki (right)

Add rules:

- **If** error is *negative large* **then** Kp is *medium* and Ki is *medium*
- **If** error is *negative medium* **then** Kp is *medium* and Ki is *medium*
- **If** error is *negative small* **then** Kp is *small* and Ki is *small*
- **If** error is *zero* **then** Kp and Ki is *zero*
- **If** error is *positive small* **then** Kp is *small* and Ki is *small*
- **If** error is *positive medium* **then** Kp is *medium* and Ki is *medium*
- **If** error is *positive large* **then** Kp is *large* and Ki is *large*

Applying to the mobile robot, we get the results:

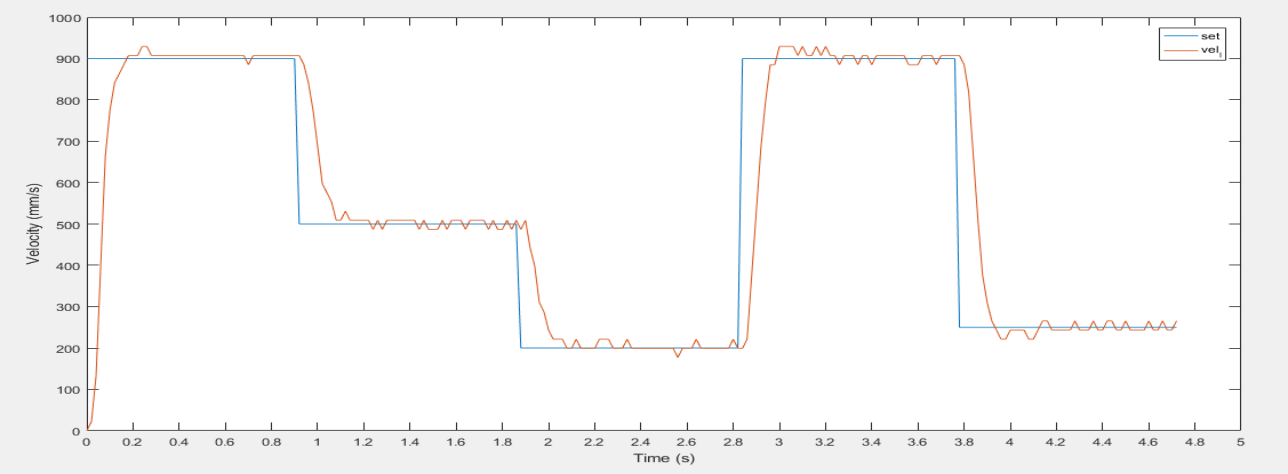


Figure 5-34: Self-tuning Fuzzy PID Controller: Left wheel's speed

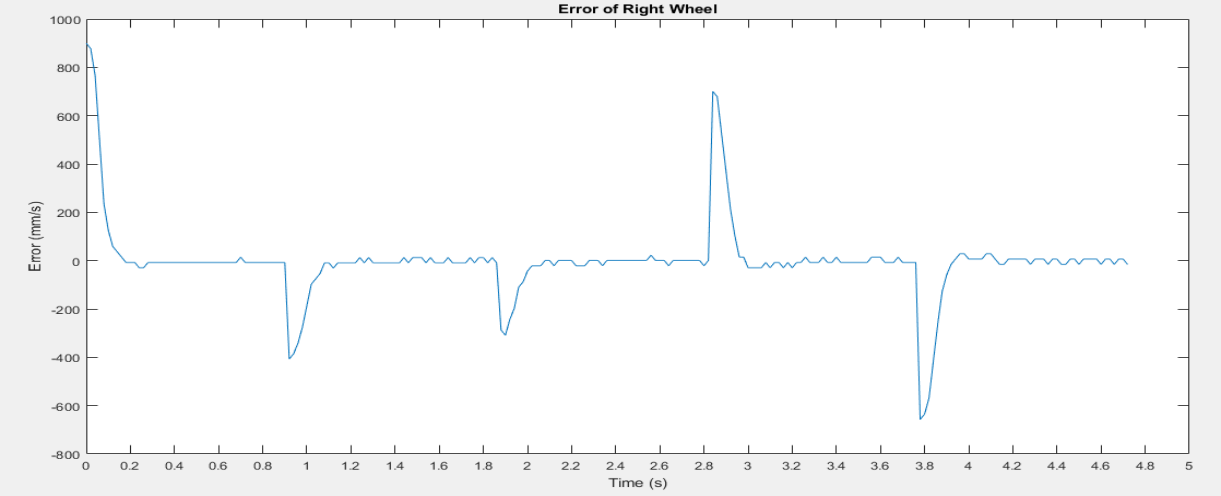


Figure 5-35: Self-tuning Fuzzy PID Controller: Left wheel's error

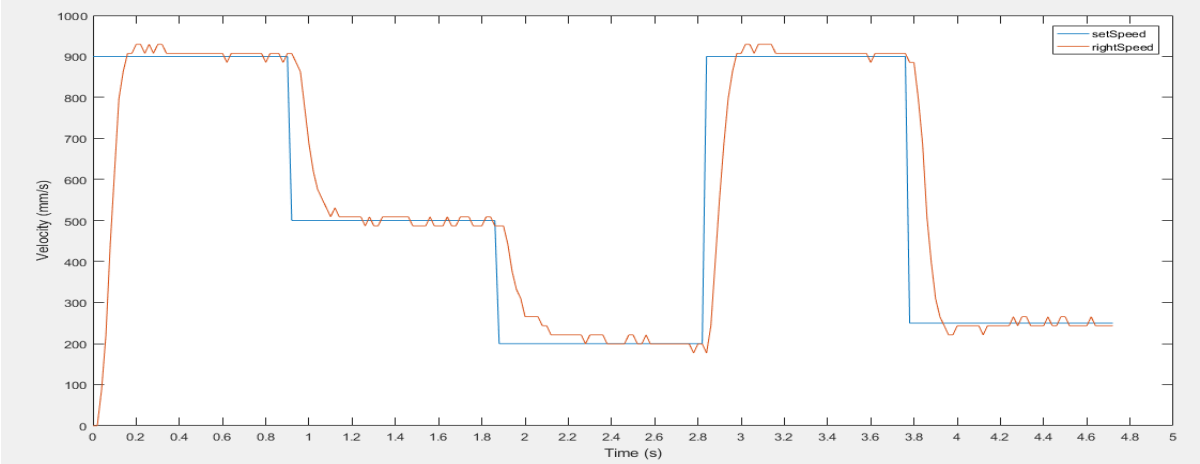


Figure 5-36: Self-tuning Fuzzy PID Controller: Left wheel's error

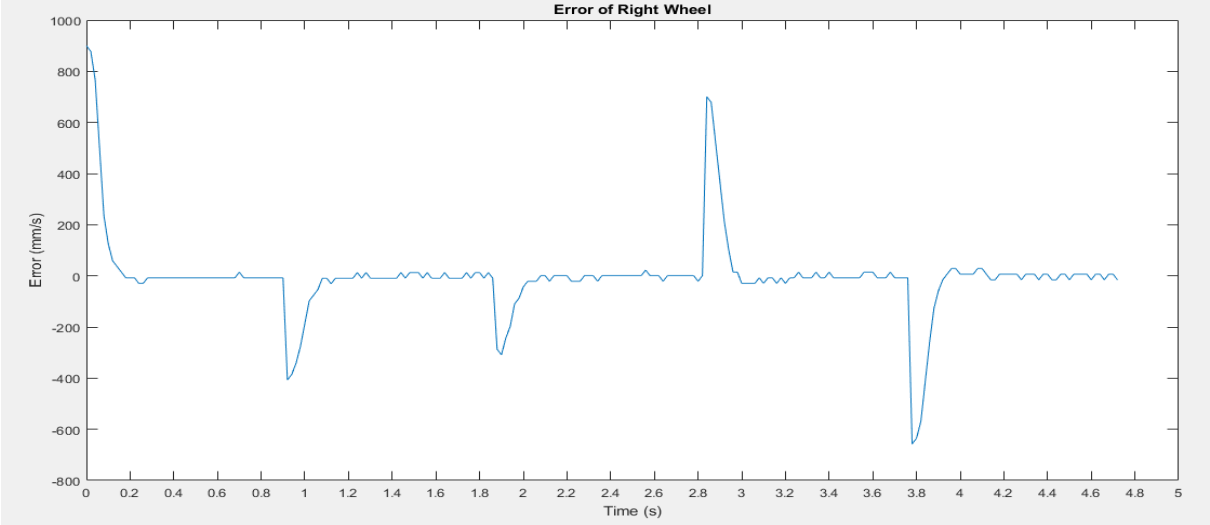


Figure 5-37: Self-tuning Fuzzy PID Controller: Right wheel's error

Conclusion: There is a little overshoot signal (about 2%), the peak time and settling time are slightly decrease comparing with Fuzzy controller.

5.6 Comparing controllers

Finally, we will compare these controllers to decide, which is the best controller for our mobile robot.

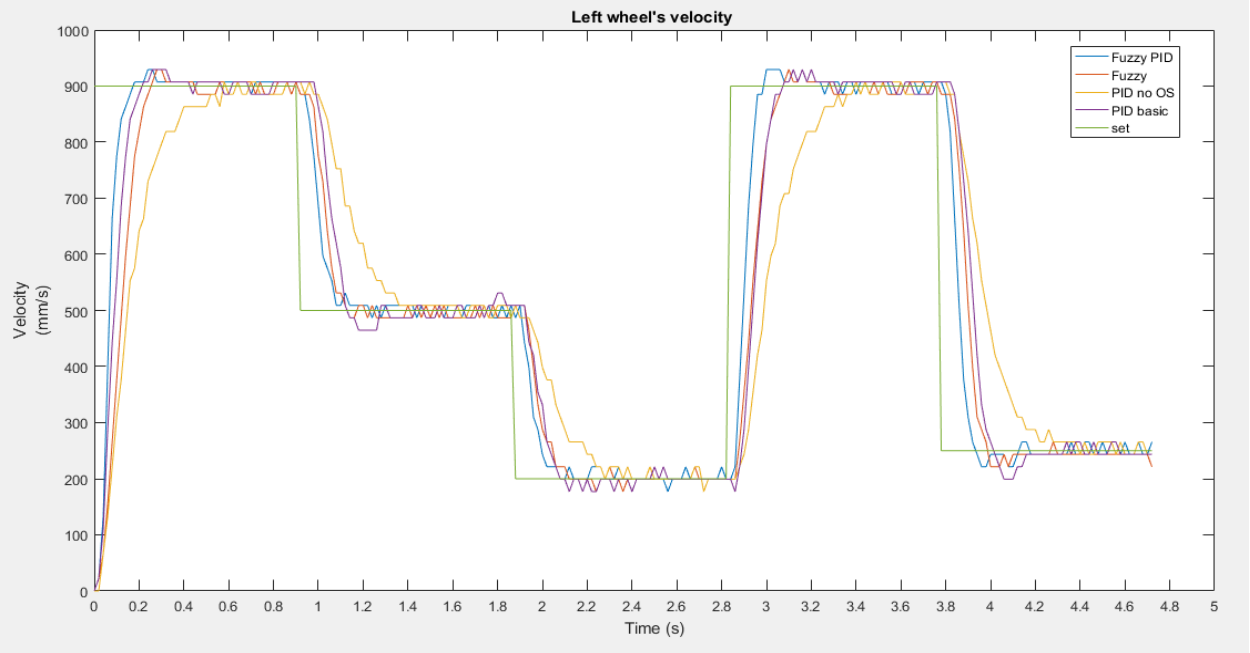


Figure 5-38: Left wheel's velocity

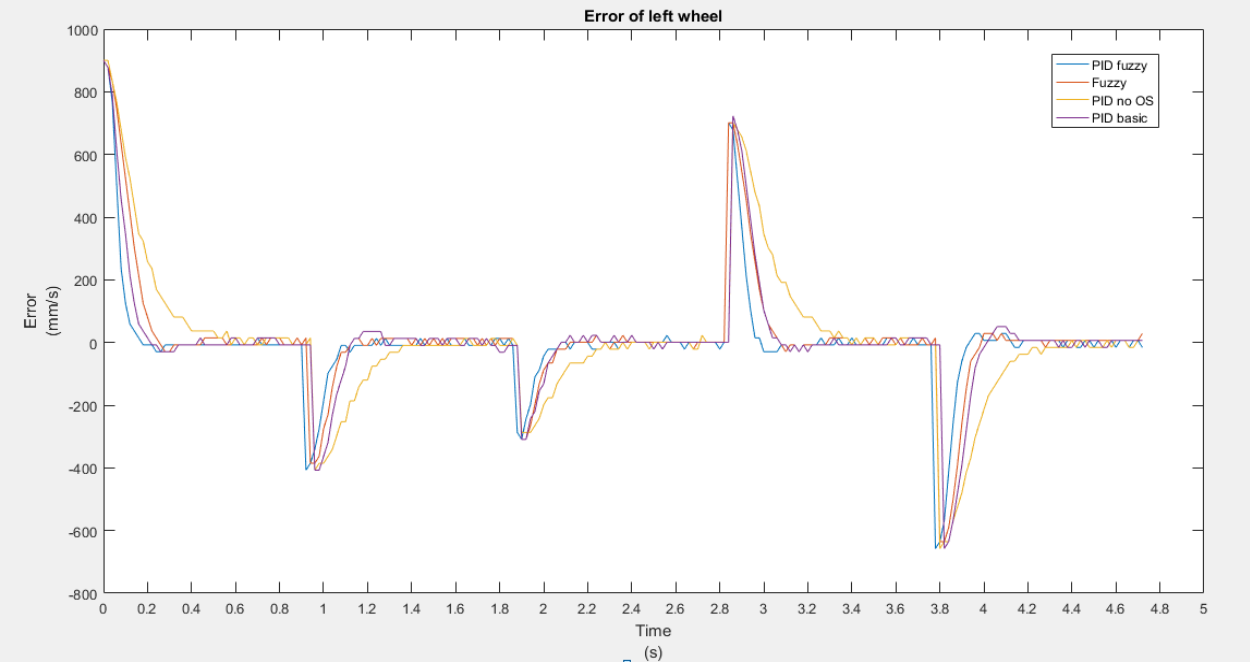


Figure 5-39: Left wheel's error

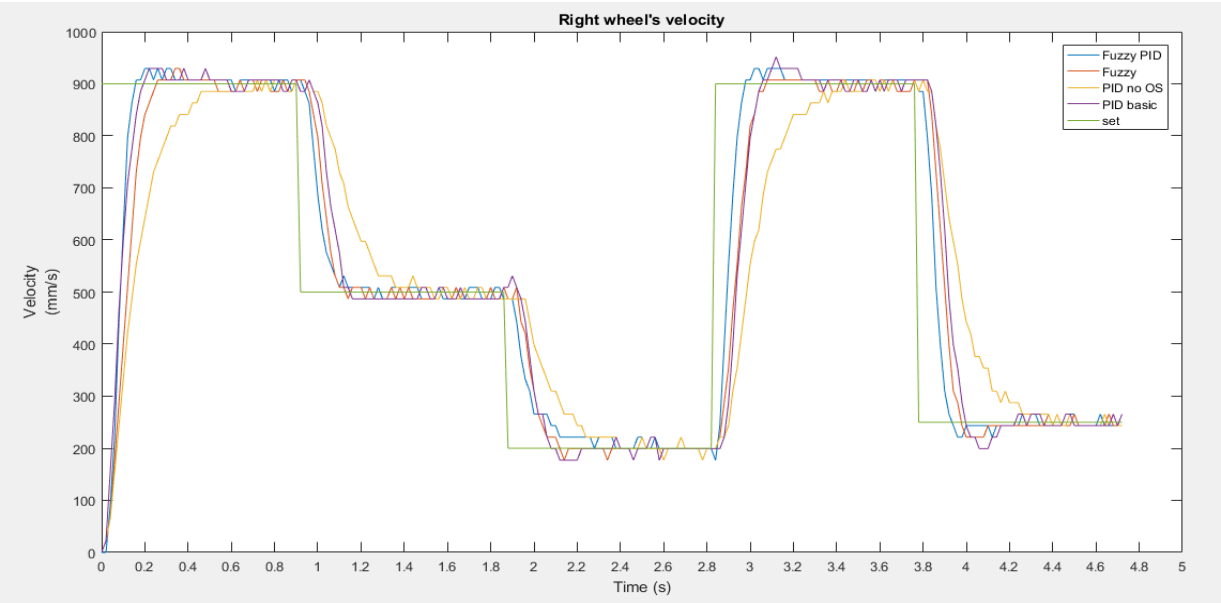


Figure 5-40: Right wheel's velocity

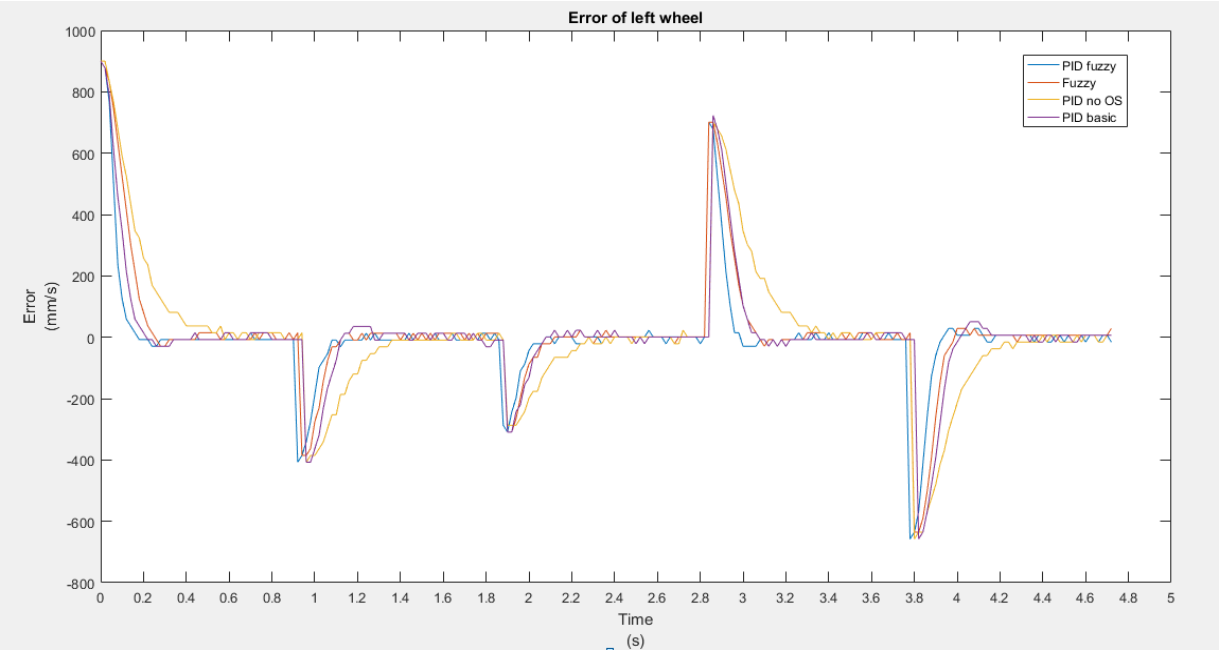


Figure 5-41: Right wheel's error

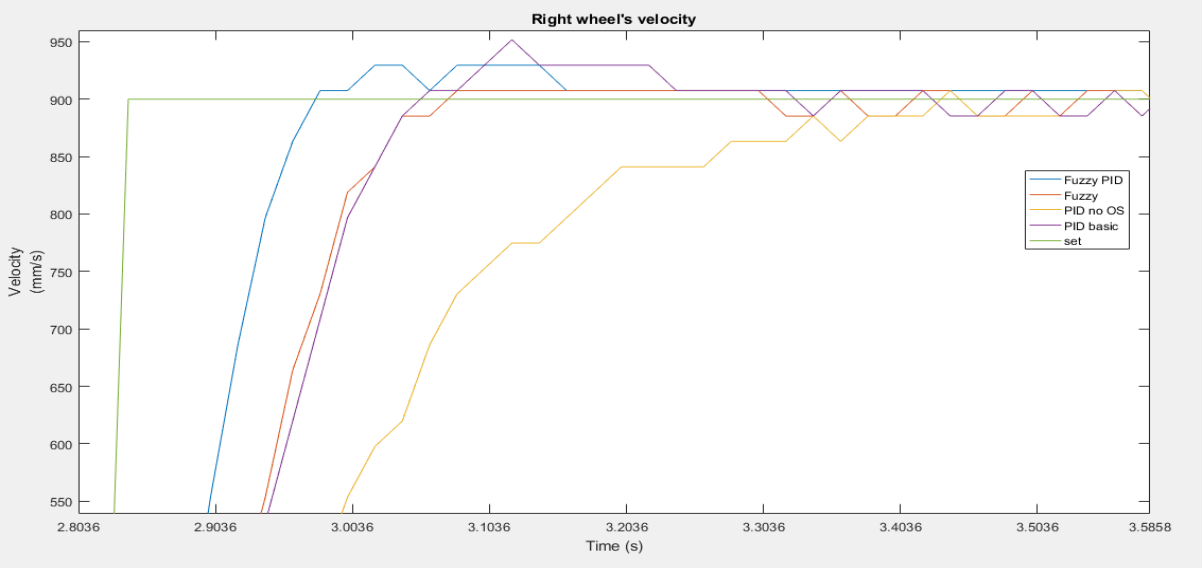


Figure 5-42: Velocity of left wheel (zoom in)

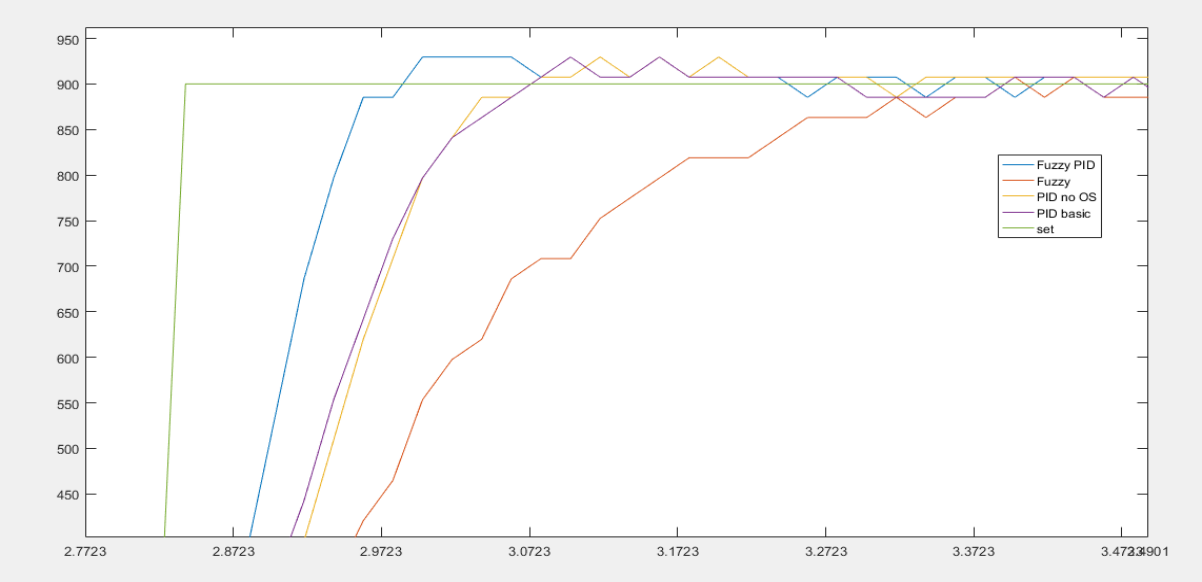


Figure 5-43: Velocity of right wheel (zoom in)

Conclusion:

Self-tuning Fuzzy PID Controller is the most suitable controller comparing with Fuzzy controller and PID controller. Following the comparison, it has the lower peak time and settling time than the other. Overshoot also less than PID, and stability are similar with Fuzzy controller. The wind-up problem is also minimized, and the system working process is smoothly. We decided to apply this controller to our mobile robot in this project.

CHAPTER 6 SIMULATION ON GAZEBO

We will build a Unified Robot Description Format (URDF) file for the robot that will describe the main components of our robot and enable it to be visualized and controlled by ROS tools, such as rviz and Gazebo.

6.1 Introduction to Gazebo

Gazebo is a multi-robot simulation tool which has the capability of accurate and efficient simulation of a population of robots, sensors and objects in a 3-dimensional world. Gazebo generates realistic sensor feedback and has a robust physics engine to generate interactions between objects, robots and environment.

Gazebo enables simulation of the world environment, physical model, sensors and control system through the URDF file. ROS is interfaced with Gazebo which allows utilization and implementation of different robotic software and tools on the simulated robot. This presented approach allows development, testing and validation of our mobile robot and required software before implementation on the real system.

We can also use it to test new code or physical systems that might be too dangerous or expensive to test in the real world.

Our task involved in taking a CAD model in Inventor and bringing it through the exportation process into Gazebo for testing.

6.2 Introduction to Rviz

Rviz is a powerful 3D visualization tool for ROS. It allows the user to view the simulated robot model, log sensor information from the robot's sensors, and replay the logged sensor information. By visualizing what the robot is seeing, thinking, and doing, the user can debug a robot application from sensor inputs to planned (or unplanned) actions.

6.3 Model description and simulation

The Gazebo simulator provides dynamics simulation by considering gravity, friction, and contact forces provided by the included physics engines. Different types of Gazebo plugins enable the development of control interfaces and sensing systems for the simulated robots. The main parts required to model a controllable robotic system in general are:

1. World environment model description

2. Physical model description

    • Kinematic and dynamic modelling of the robot links

    • Kinematic and dynamic modelling of the robot joints

3. ROS/Gazebo Plugins to model the sensors and control/hardware interfaces

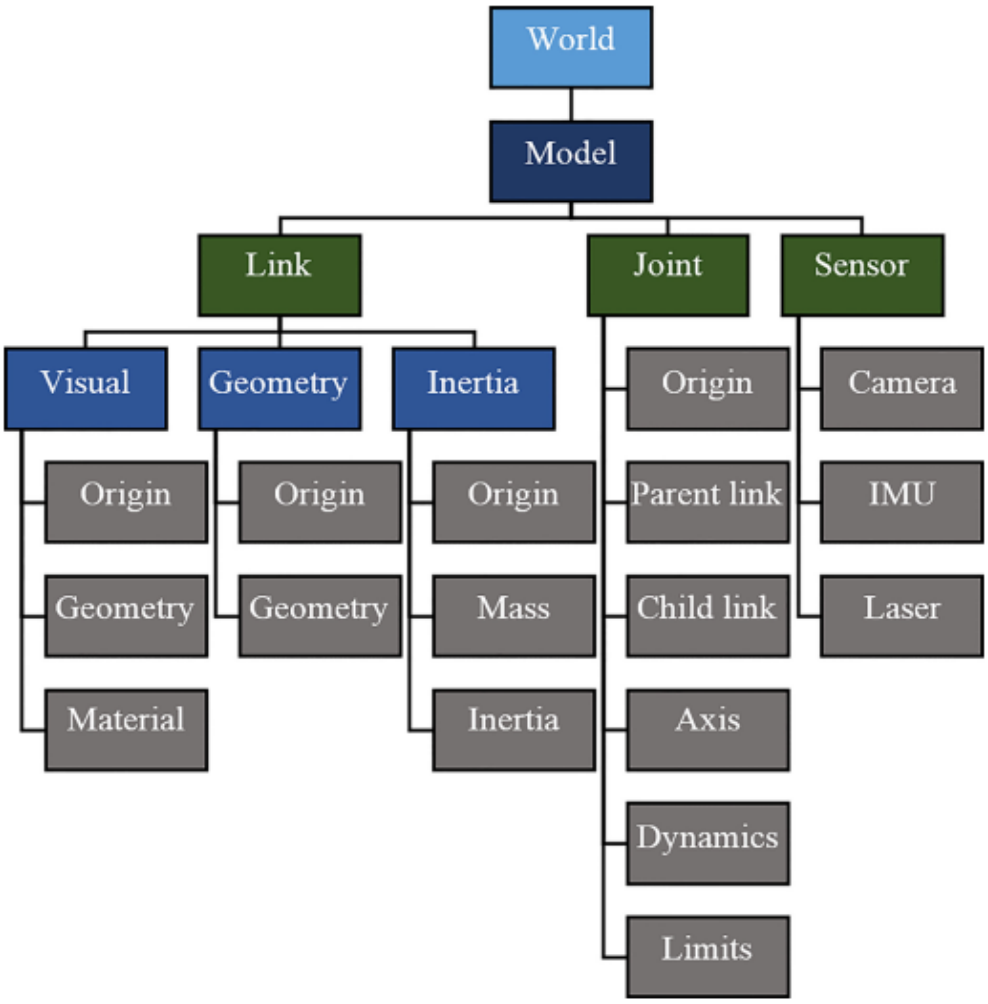


Figure 6-1 General structure of required components to model a robotic system in Gazebo

6.3.1 World description

For simulation, we use the a *".world"* file which will build an environment with walls and obstacle. For simulating, we used the *"corridor.world"* file from gazebo's library.

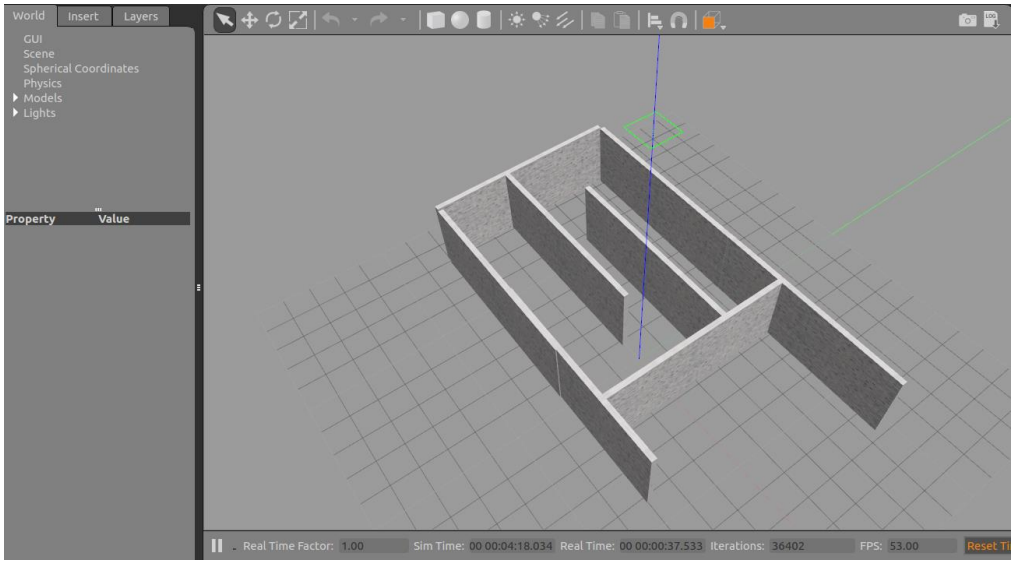


Figure 6-2: Simulation environment

6.3.2 Physical model description

Mobile robot physical platform description

The mobile robot as a differential drive robot platform consists of 3 wheels attached to a chassis body that holds all the electronics. Two driving wheels with the modules are equipped with a DC motor to drive the wheels. It is also equipped with equipped with the RealSense™ Depth Camera D435. The platform base dimension is 0.22 m × 0.22m × 0.155 m. The figure shows the physical developed model with the onboard sensors and electronics on the field and the 3D CAD model of the Mobile robot base developed by Autodesk Inventor.

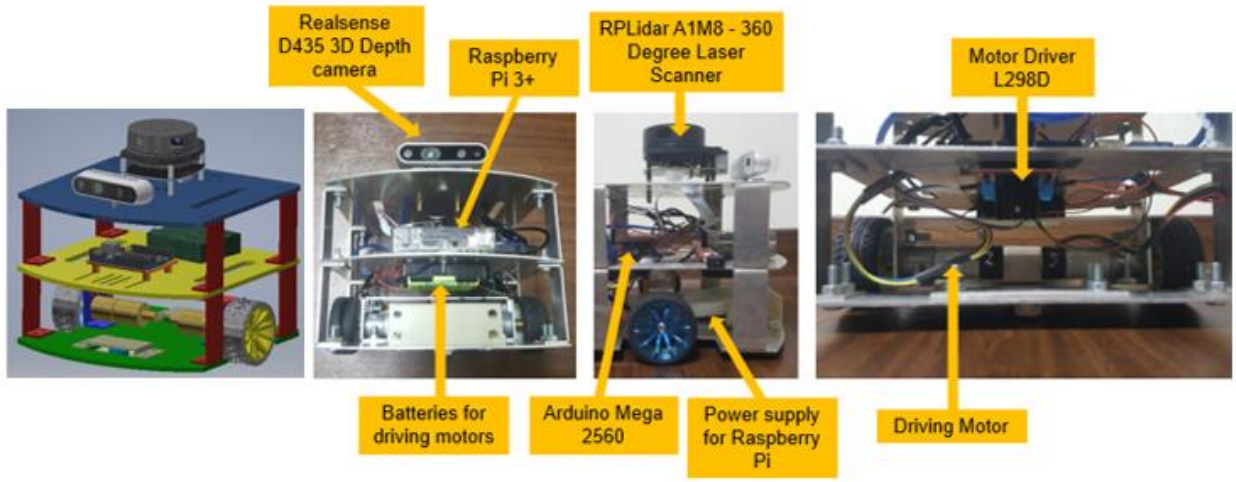


Figure 6-3: Mobile robot prototype and the on-board electronics and sensors

6.3.3 Mobile robot simulated model

A robot, as a dynamic object in the world, consists of links that are connected to each other by joints. A link in Gazebo describes the kinematic and dynamic properties of a physical link in the form of visual/collision geometry and inertia information. A joint models kinematic and dynamic properties of a joint such as joint type, motion axes, and joint safety limits. All information is described in the Universal Robotic Description Format - URDF file format.

File Types: URDF

The file is a tree structure of child links (like wheels) connected to parent links (like the chassis) by a series of joints. This file is the description of how system moves internally, and this will determine how it can interact with the environment. URDF files can become long and cumbersome on complex robot systems. Xacro (XML Macros) is an XML macro language created to make these robot description files easier to read and maintain.

6.4 Building the differential drive mobile robot URDF

Two basic URDF components are used to define a tree structure that describes a robot model. The link component describes a rigid body by its physical properties (dimensions, the position of its origin, color, and so on). Links are connected together by joint components. Joint components describe the kinematic and dynamic properties of the connection (that is, links connected, types of joint, the axis of rotation, amount of friction and damping, and so on). The URDF description is a set of these link elements and a set of the joint elements connecting the links together. The first component of our robot is a simple chassis box.

6.4.1 Creating a robot chassis

The physical geometry of each link needs to be defined in the form of visual and collision geometry. The mobile robot can be modelled using a cubic box as the chassis and 2 cylindrical shapes as the wheels. Using continuous joints, wheels can be attached to the chassis.

Inertial parameters define the mass, centre of the mass, and the moment of inertia tensor matrix for each modelled link. These information are obtained from Inventor. Below is the example of URDF code to model the chassis.

```
<link name='chassis'>
  <pose>0 0 0 0 0 0</pose>

  <inertial>
    <mass value="3.0"/>
    <origin xyz="0 0 0" rpy=" 0 0 0"/>
    <inertia
        ixx="0.018" ixy="0" ixz="0"
        iyy="0.018" iyz="0"
        izz="0.024"
    />
  </inertial>

  <collision name='collision'>
    <geometry>
      <box size="0.22 0.22 0.155"/>
    </geometry>
  </collision>

  <visual name='chassis_visual'>
    <origin xyz="0 0 -0.07525" rpy=" 0 0 0"/>
    <geometry>
      <mesh filename="package://mobile_description/meshes/mobile.dae"/>
    </geometry>
  </visual>
```

Figure 6-4: Chassis URDF description

```
<link name="left_wheel">
  <collision name="collision">
    <origin xyz="0 0 0" rpy="1.5707 0 0"/>
    <geometry>
      <cylinder radius="0.03" length="0.0325"/>
    </geometry>
  </collision>

  <visual name="left_wheel_visual">
    <origin xyz="0 0 0" rpy="1.5707 0 0"/>
    <geometry>
      <cylinder radius="0.03" length="0.0325"/>
    </geometry>
  </visual>

  <inertial>
    <origin xyz="0 0 0" rpy="1.5707 0 0"/>
    <mass value="1"/>
    <cylinder_inertia m="0.5" r="0.03" h="0.0325"/>
    <inertia
      ixx="0.00016" ixy="0.0" ixz="0.0"
      iyy="0.00016" iyz="0.0"
      izz="0.000225"/>
  </inertial>
</link>
```

Figure 6-5: Wheel URDF description

```
<joint type="continuous" name="left_wheel_hinge">
  <origin xyz="0 0.1 -0.06" rpy="0 0 0"/>
  <child link="left_wheel"/>
  <parent link="chassis"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
  <limit effort="100" velocity="100"/>
  <joint_properties damping="0.0" friction="0.0"/>
</joint>

<joint type="continuous" name="right_wheel_hinge">
  <origin xyz="0 -0.1 -0.06" rpy="0 0 0"/>
  <child link="right_wheel"/>
  <parent link="chassis"/>
  <axis xyz="0 1 0" rpy="0 0 0"/>
  <limit effort="100" velocity="100"/>
  <joint_properties damping="0.0" friction="0.0"/>
</joint>
```

Figure 6-6: Joints URDF description

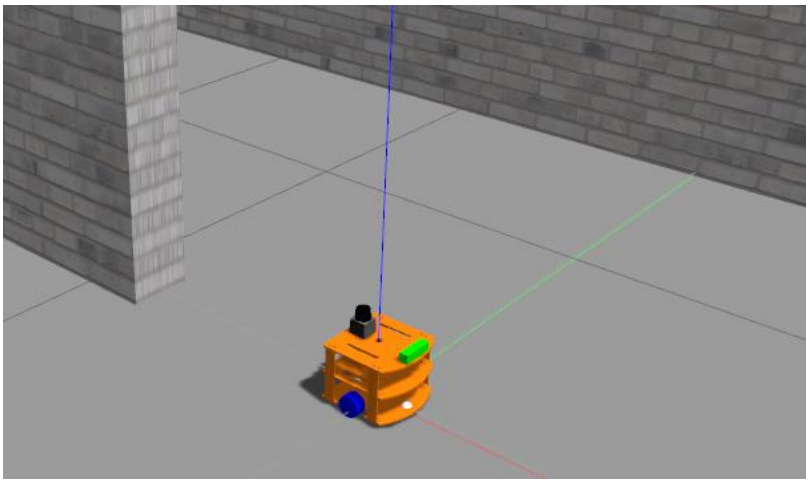


Figure 6-7: Simulation model of robot

6.4.2 Sensor modelling

With many different sensors on the robot, each sensor can be simulated independently as a Gazebo plugin and must be physically attached to the robot model as a link. These plugins are included in the URDF file. These plugins output sensor information in form of standard ROS messages and services. Common parameters, such as error characteristics and transformation frame of each sensor, can be defined as well as other parameters related to each sensor. By default, sensors in Gazebo have no noise and they sense the simulated environment perfectly. To make them more realistic, noise can be added.

```
<gazebo reference="camera">
  <material>Gazebo/Green</material>
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>mybot/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

Figure 6-8: Camera plugin

```
<!-- hokuyo -->
<gazebo reference="hokuyo">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-3.141592</min_angle>
          <max_angle>3.141592</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <!-- Noise parameters based on published spec for Hokuyo laser
             achieving "+-30mm" accuracy at range < 10m.  A mean of 0.0m and
             stddev of 0.01m will put 99.7% of samples within 0.03m of the true
             reading. -->
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
      <topicName>/mybot/laser/scan</topicName>
      <frameName>hokuyo</frameName>
    </plugin>
  </sensor>
</gazebo>
```

Figure 6-9: Laser scanner plugin

6.5 Using the ROS navigation meta-package

We have created a simulated differential drive model that satisfies the necessary pre-requisites for autonomous navigation: the transform tree, odometry, laser scanner, and base controller. We are now making our simulated robot move around on its own by using ROS's navigation Meta-package. In this part, we are going to create a map to navigate the robot with the map

1. Plug in the ROS navigation Meta-package
2. Create a map
3. Navigation with the map

6.5.1 Mapping

There are basically two steps: creating the map, saving the map.

Use the *teleop* node to move the robot around to create an accurate and thorough map

In Terminal 1, launch the *Gazebo* world

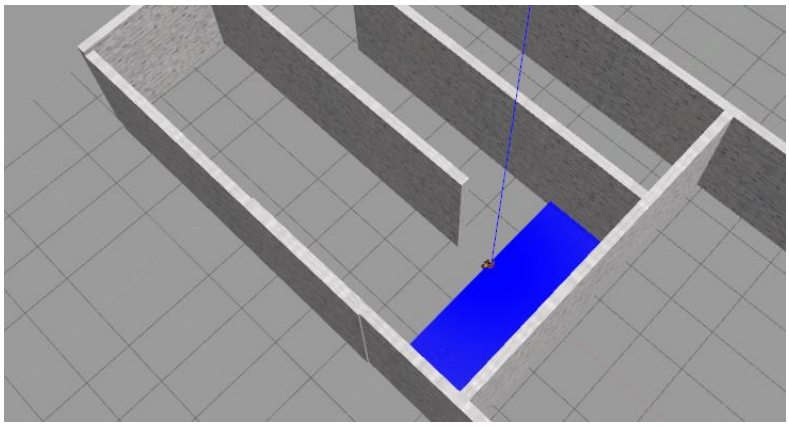


Figure 6-10: Robot in Gazebo simulation environment

In Terminal 2, start map building

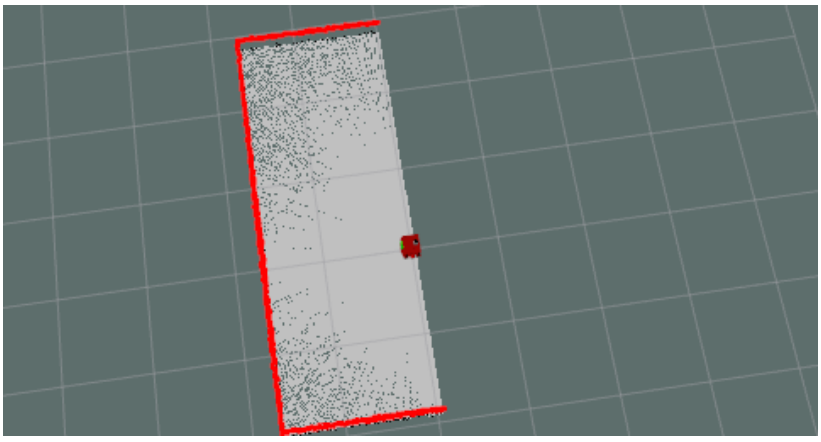


Figure 6-11: Starting of SLAM, map building

In Terminal 3, launch *rviz* and set the following parameters:

In Terminal 4, start *teleop to move the robot around environment*

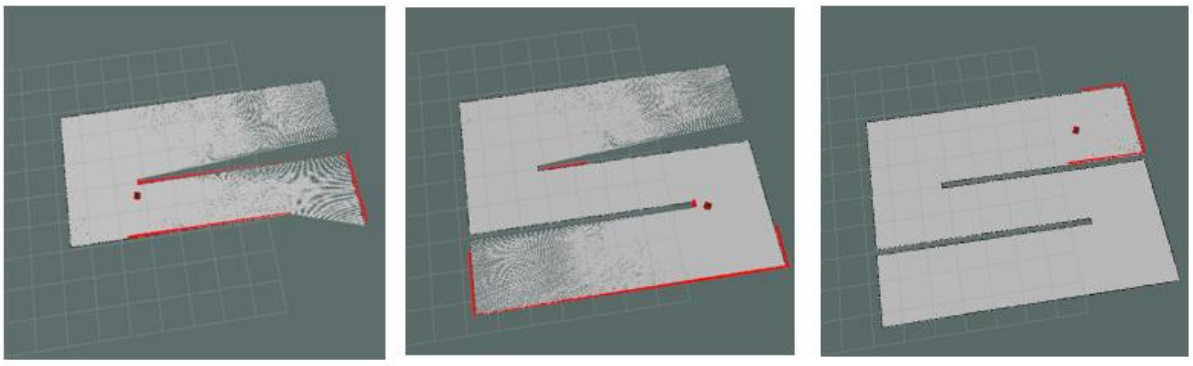


Figure 6-12: Map is being built by laser scanner

In Terminal 5, save the map to some file path.

After saving we can obtain two files with **pgm* and **yaml* extension. The test_map.pgm contains the image of the map while the test_map.yaml contains informations of the map include the path to the image, resolution, origin, etc.

```
image: /home/nhatluong/catkin_ws/src/mybot_ws/src/mybot_navigation/maps/
test_map.pgm
resolution: 0.010000
origin: [-20.000000, -20.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

*Figure 6-13: Content of file *yaml*

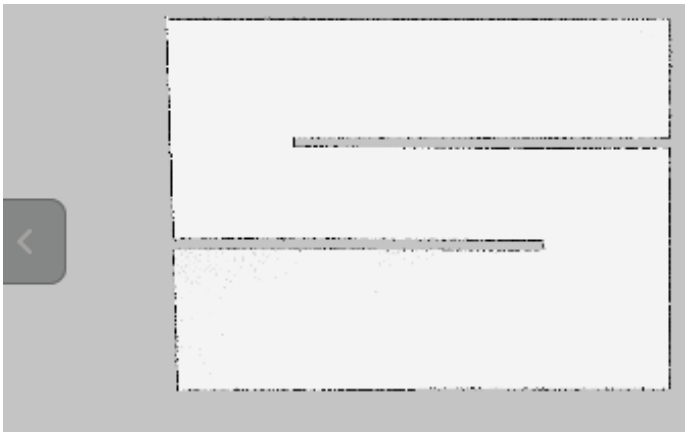


*Figure 6-14: Contents of file *pgm*

6.5.2 Implement the RRT* into ROS with navigation meta-package

The navigation stack of the Robot Operating System (ROS) open-source middleware incorporates both global and local path planners to support ROS-enabled robot navigation. However, only basic algorithms are defined for the global path planner including Dijkstra, A*, and carrot planners. In this part, we present RRT* global path planner into the ROS navigation system. Also, we compared the performance of the RRT* with ROS default planner.

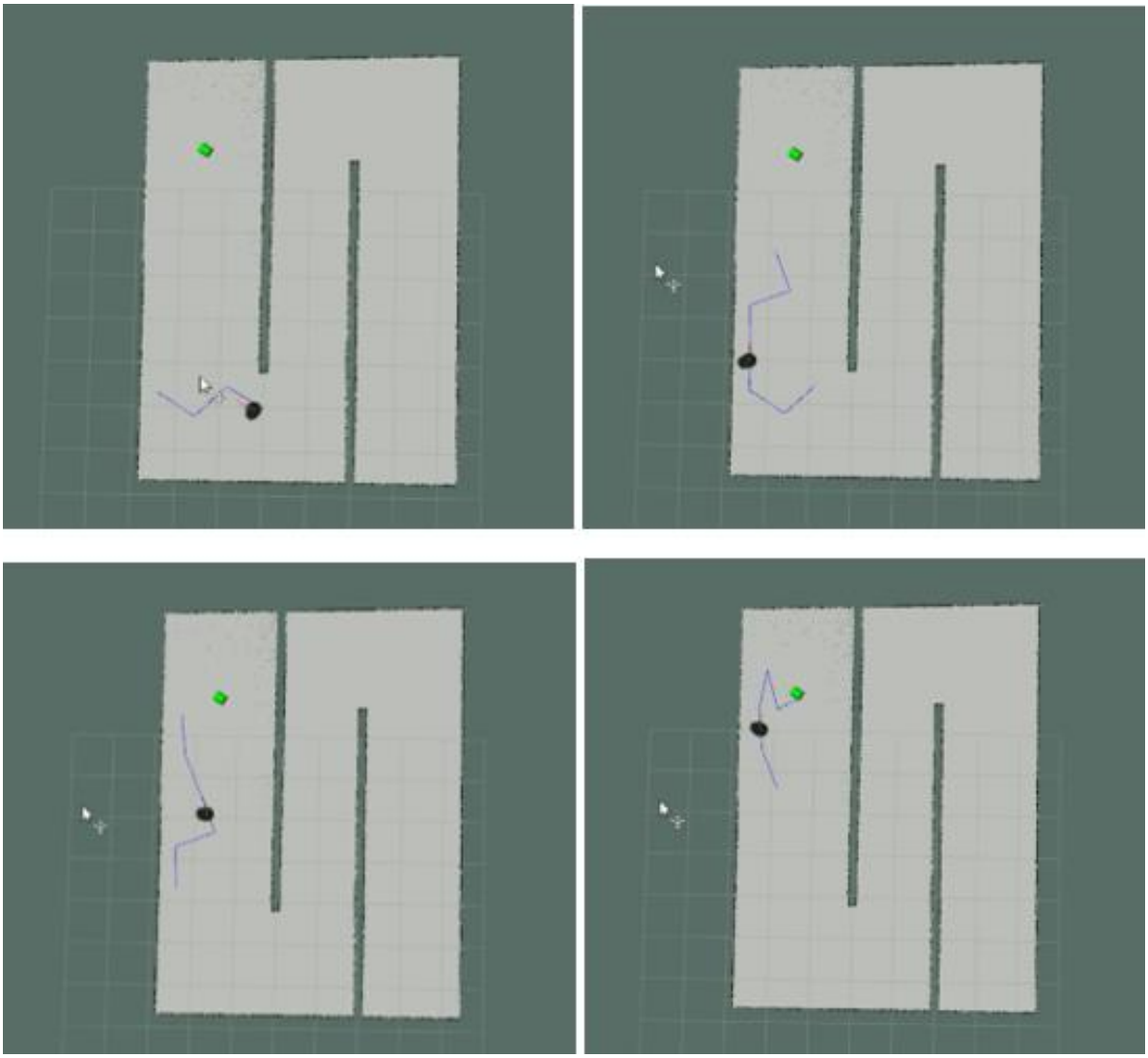


Figure 6-15: Testing RRT algorithm on turtlebot*

The local trajectory planned by *dwa local planner* is colored red. The blue line is the global path created by RRT* algorithm. The path has a zig-zag shape so we extend the neighborhood value.

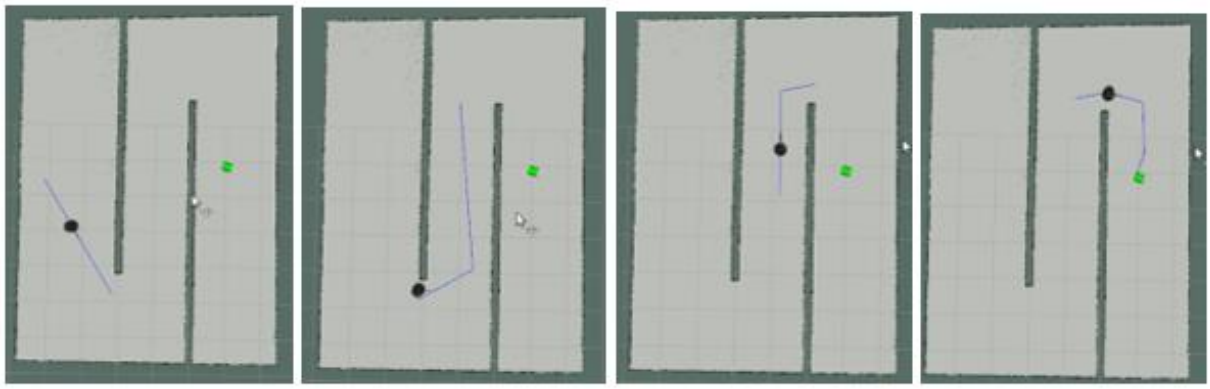


Figure 6-16: Planned path after extending the neighborhood

After succeeding in using RRT* path planner in ROS, we now used it with our robot model in Gazebo. But now we added obstacles include static obstacle (gray) and dynamic obstacle (white).

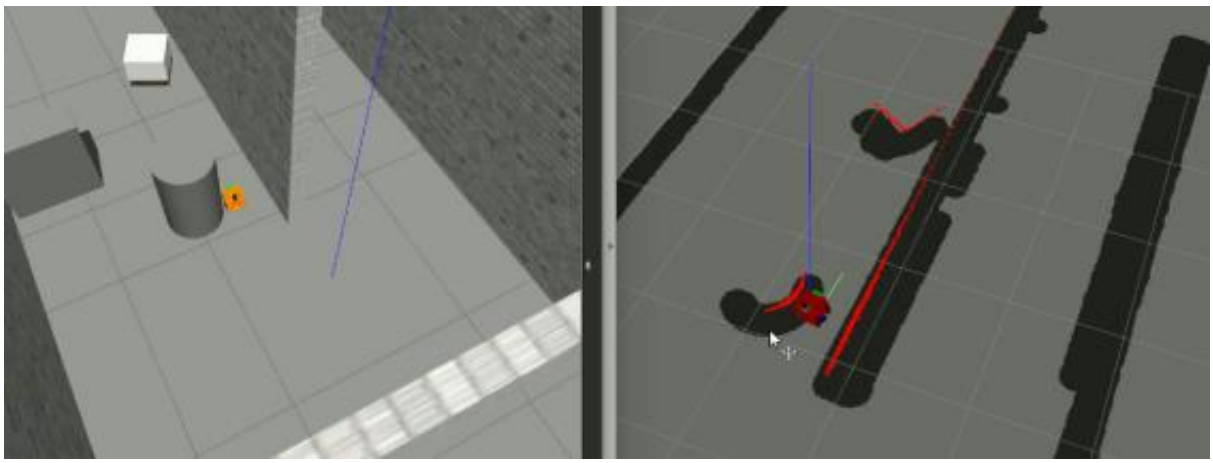


Figure 6-17: Robot is avoiding static obstacle

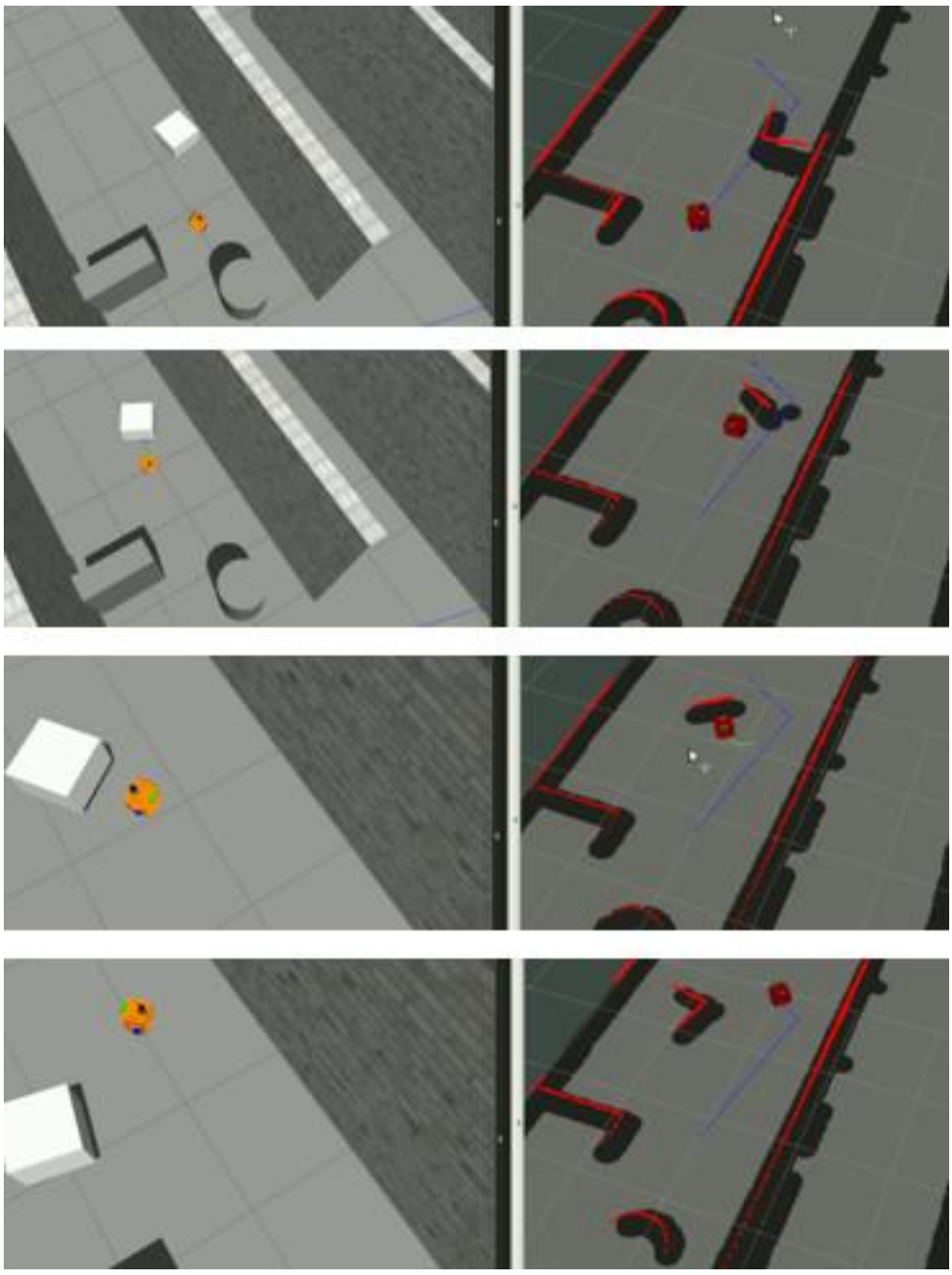


Figure 6-18: Robot avoid obstacles in simulation

CHAPTER 7 FINAL RESULTS

7.1 Configuring the ROS Network

This is the configuration for ROS_MASTER_URI and ROS_HOSTNAME. ROS uses a network to communicate messages between nodes, so the network configuration is very important. The PC is the master and the robot uses Raspberry Pi as a host PC, the network has to be configured on both PC and Pi in order to allow multiple computers to communicate with each other.

7.2 Mapping

7.2.1 2D mapping

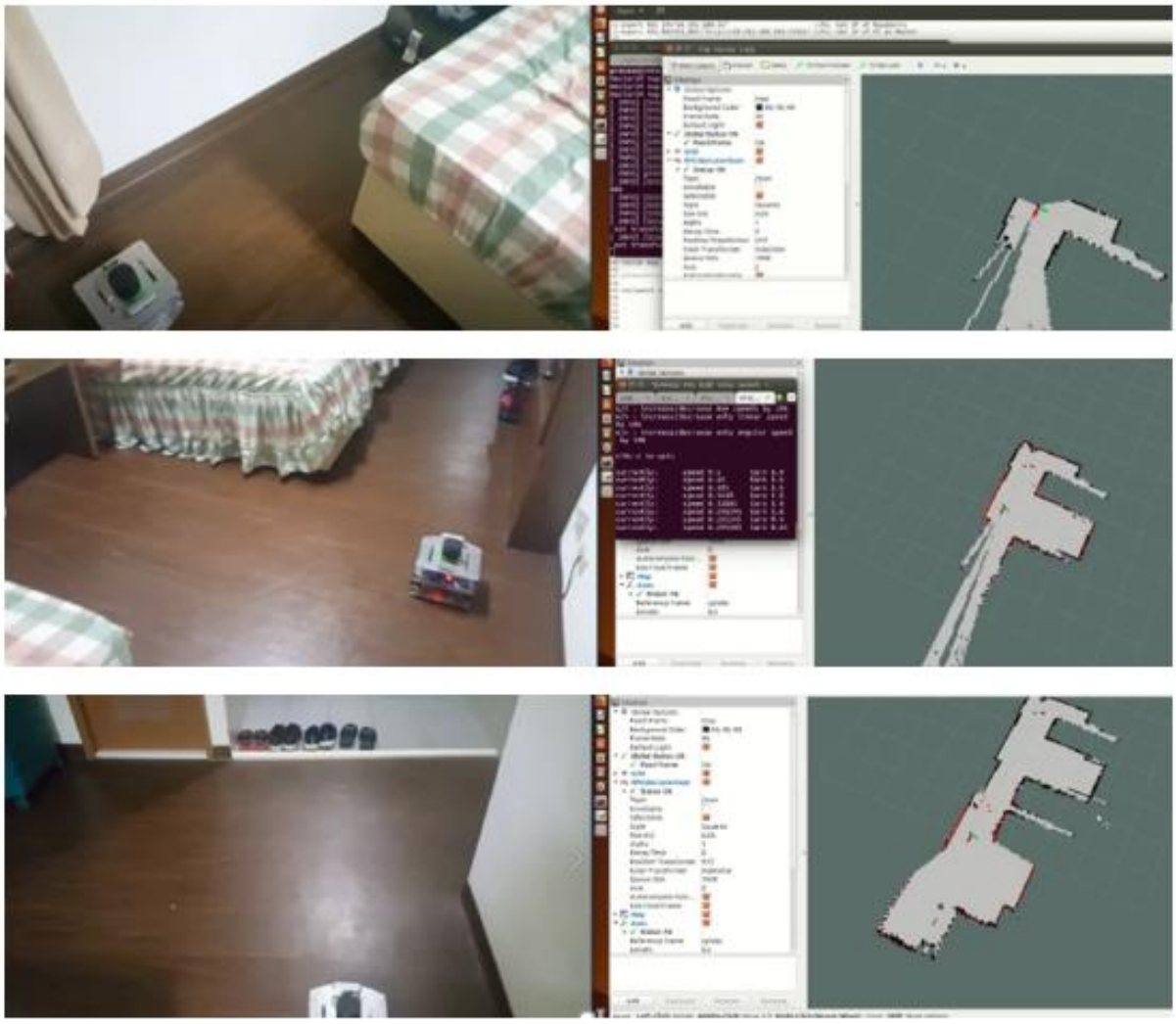


Figure 7-1: Robot goes around the room and build the 2D map

7.2.2 3D fusion mapping

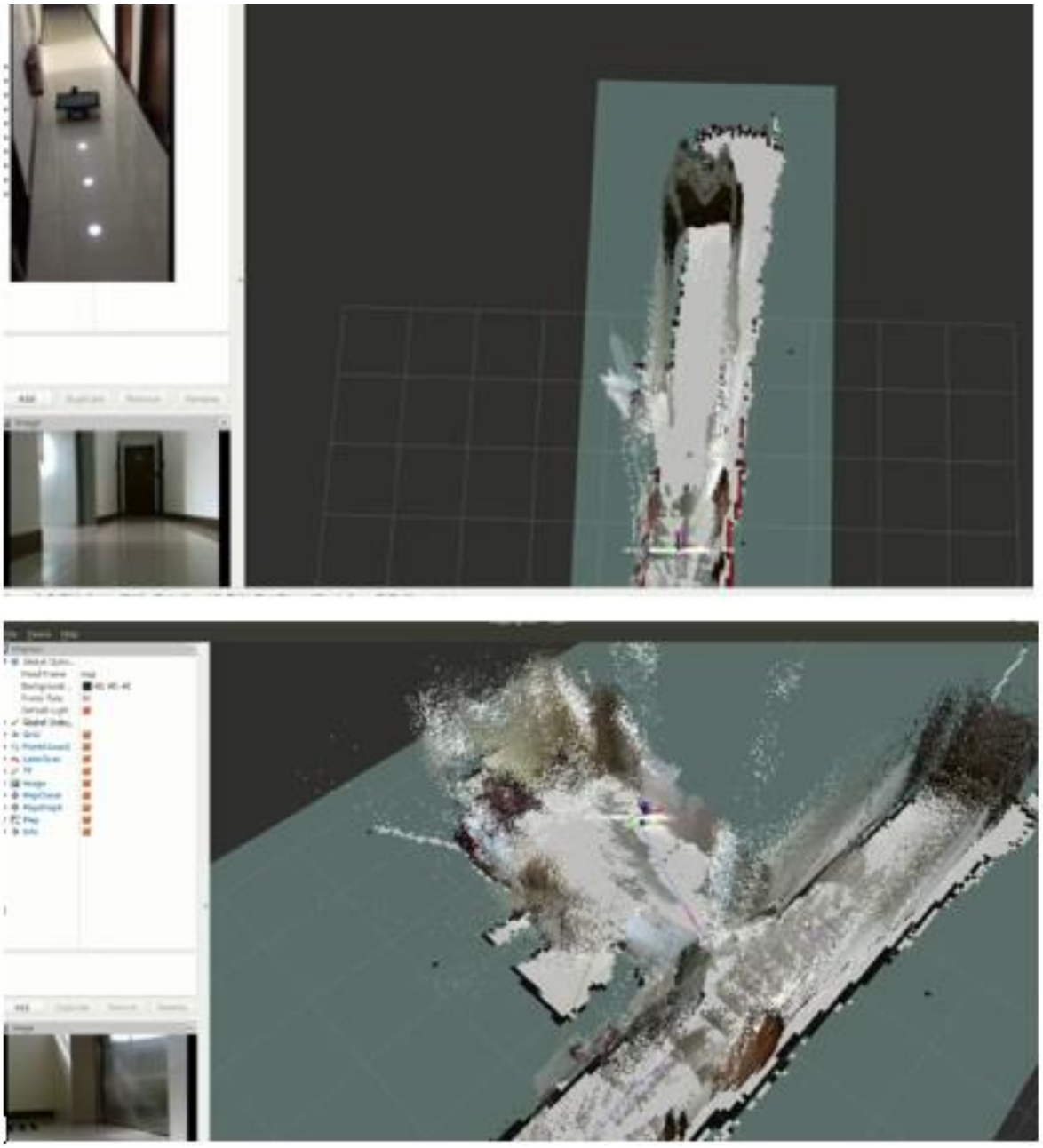


Figure 7-2: Robot goes around the lobby and build the 3D map

7.3 Localization

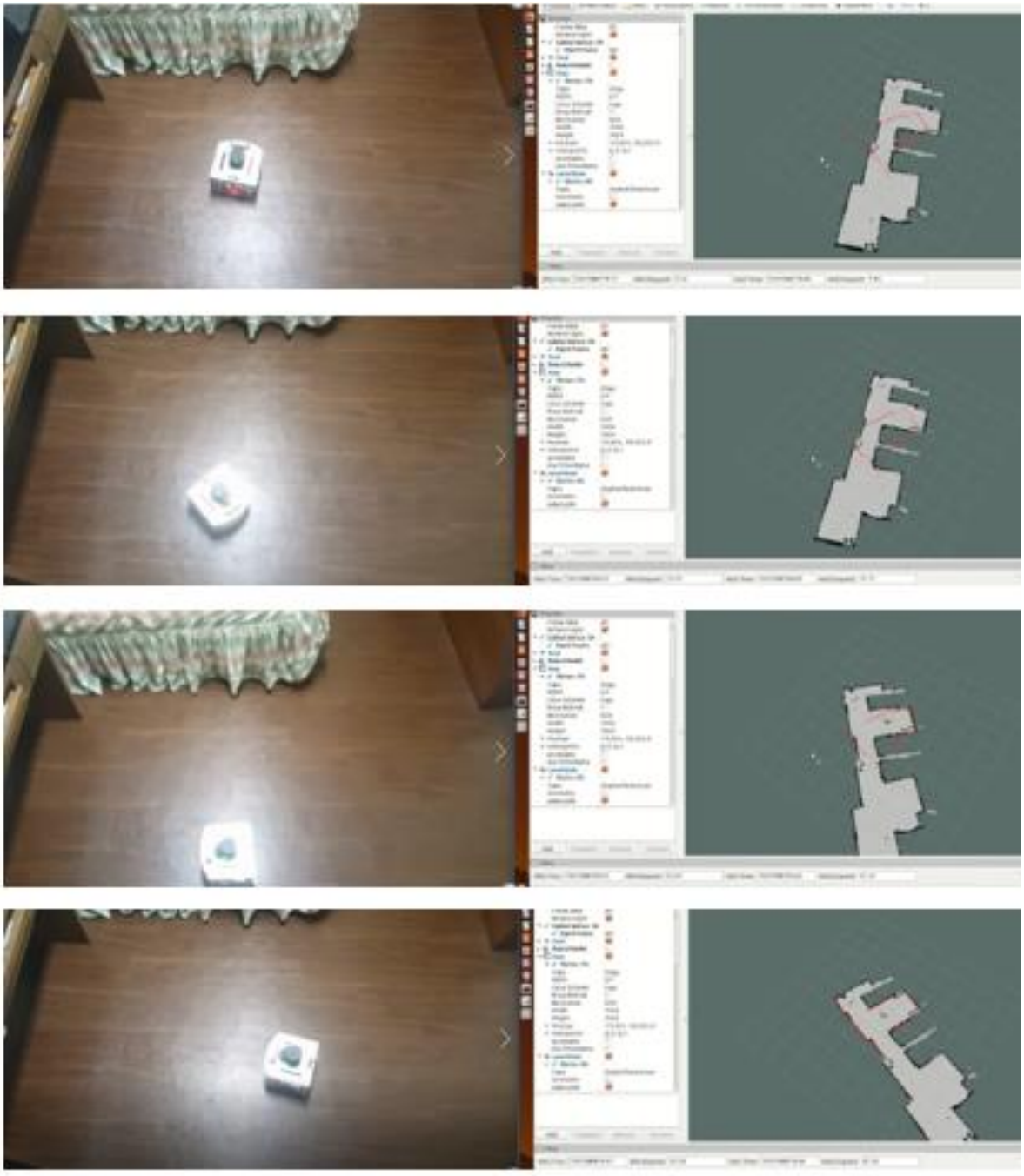


Figure 7-3: The pose of robot is determined after moving robot around its position

7.4 Navigation, detecting and avoiding obstacle

Figure 7-4: Setting the goal point for robot

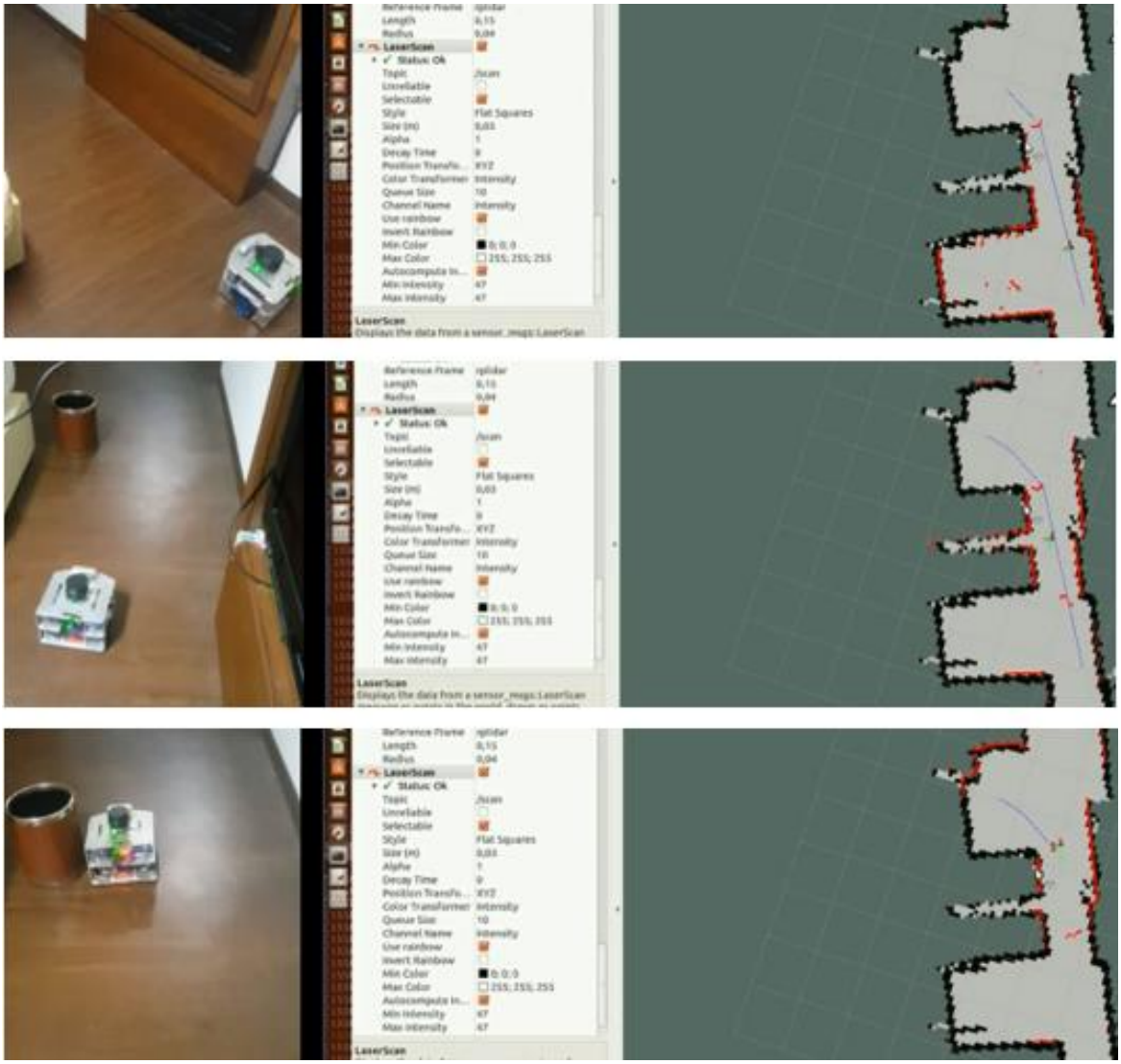


Figure 7-5: Robot avoiding obstacle

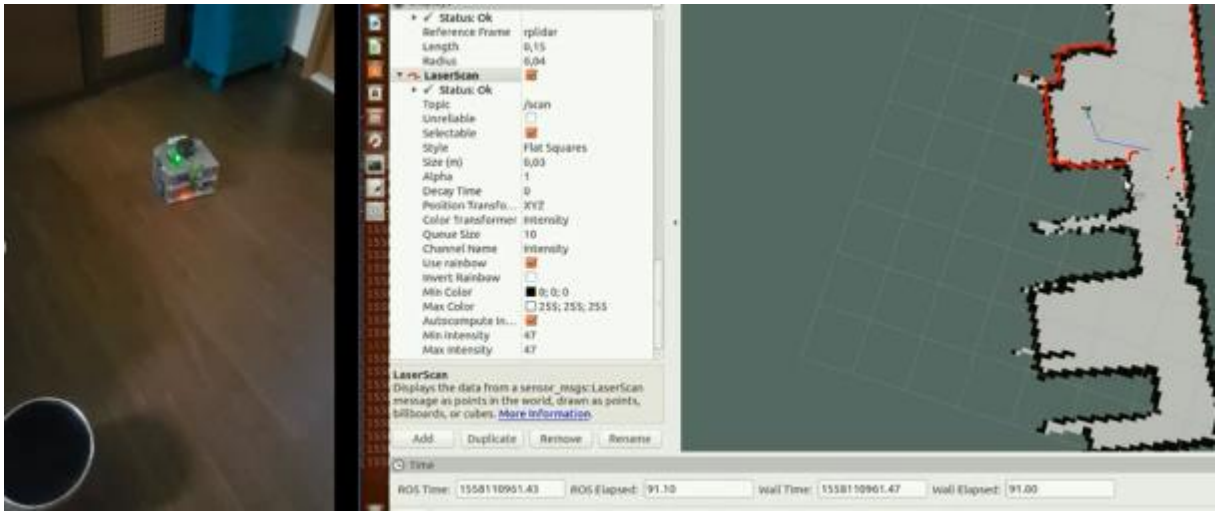


Figure 7-6: Robot reached goal point

CONCLUSION

In this thesis, we have presented a full solution for SLAM with LIDAR sensors and RGB-D camera. The solution can either work on only LIDAR data or with a combination of LIDAR, RGB-D camera data. Hector SLAM can generate 2D quite fast with a little computational burden. But the Hector SLAM algorithm doesn't have close loop detection so in some cases the map is not accurate. On the other hand, visual SLAM with Rtabmap package comes with feature-matching and a 3D map that will give more information on the surrounding environment.

The path found by RRT* is also nearly optimal for robot moving in shortest distance. There will be a delay about 1 second after setting goal point to robot. However, it is still considered a good result.

After obtaining the path, the Fuzzy-PID controller made the movement of robot follow tightly with the planned path and the robot finally reach the goal within a small allowed tolerance.

Our group has also succeeded in building 3D map and also applied RRT* algorithm in 3D. Therefore, in the future, our group want to develop the system into the collaborations of a hybrid mobile robot UGV-UAV (Unmanned Ground Vehicle – Unmanned Aerial Vehicle) because they are vastly different in skills. The flying robot can quickly cover big areas while a smaller mobile robot carries substantial payload so by combining mobile robot with flying robots, we leverage the capabilities of these two system covering for example an unknown disaster scenario where drone needs to go quickly go in and then deliver goods or find people.

REFERENCES

[1] Roland SIEGWART, Illah R. NOURBAKHSH Introduction to Autonomous Mobile Robots. A Bradford Book The MIT Press Cambridge, Massachusetts London, England.

[2] Sterven M. La Valle Rapidly-Exploring Random Trees: A New Tool for Path Planning. Iowa State University Ames, IA 50011 USA.

[3] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, TaeHoon Lim - ROS Robot Programming (2018)

[4] Carol Fairchild, Dr. Thomas L. Harman - ROS Robotics by Example (2016)

[5] M. Labbé and F. Michaud, "RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation," in Journal of Field Robotics, vol. 36, no. 2, pp. 416–446, 2019

[6] Fuzzy PID Controllers Using 8-Bit Microcontroller for U-Board Speed Control - Sereyvatha Sarin, Hilwadi Hindersah, Ary Setijadi Prihatmanto in 2012 International Conference on System Engineering and Technology

[7] Implementation of a New Self-Tuning Fuzzy PID Controller on PLC - Onur KARASAKAL, Engin YEŞİL, Müjde GÜZELKAYA, İbrahim EKSİN

[8] Evolutionary auto-tuning algorithm for PID controllers - Gilberto Reynoso-Meza, Javier Sanchis, Juan M. Herrero, César Ramos

[9] www.teamhector.de/resources/32-hector-slam2

[10] https://en.wikipedia.org/wiki/Rescue_robot

[11] http://wiki.ros.org/

All links were last followed on May 20, 2019.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature