HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

PROJECT II
Wheeled Mobile Robot Design

STUDENT GROUP

Tran Tuan Minh	-	20181906
Tran Minh Duc	-	20181875
Ly Duc Trung	-	20181930

Supervisor: Le Minh Thuy, Ph.D

Department: Instrumentation and industrial informatics
School: Electrical Engineering

HA NOI, 01/2022

TABLE OF CONTENTS

LISTS OF FIGURES

LIST OF TABLES

CHAPTER 1. GENERAL INTRODUCTION

1.1 About mobile robot

Mobile robot is a robot that have the capability to move around in their environment and are not fixed to one physical location. A mobile robot is called "autonomous" when it can navigate an uncontrolled environment without the need for physical or electro-mechanical guidance devices.

Autonomy has to be guaranteed by the following:

+ The robot should carry some source of energy.

+ The robot should be capable of taking certain decisions and performing appropriate actions.

Based on the level of the robot's autonomy, the following typical commands can be taken from the operator:

+ Desired wheel velocity

+ Desired robot longitudinal and angular velocities

+ Desired robot path or robot trajectory

+ Desired operation inside of the known environment with potential obstacles

+ Desired missions

The main mechanical and electronic parts of an autonomous mobile robot are the following:

+ Mechanical parts: body, wheel, etc.

+ Actuators: Electrical motors

+ Sensors: Rotation encoders, Lidar,

+ Computers: Micro-controller, Embedded system, etc.

+ Power unit: Batteries

+ Electronics: actuator drive, telecommunication electronics.

Nowadays, there are various applications of wheeled mobile robots. They are expected become an integral part of our daily lives such as: medical services, operation support, cleaning applications, forest maintenance and logging, consumer goods store, etc.

Besides, using robot as a servant is becoming more and more popular in the world. The purpose of the Serving Robot is to deliver meals and drinks to guests and customers at hotels and airport lounges quickly and efficiently. Once the delivery is confirmed, the Serving Robot makes its way back on its own.

To understand more about robotic techniques and apply our theoretical knowledge into reality, our project's purpose is to design an autonomous mobile robot for serving at the restaurant.

1.2 Serving mobile robot

Below are some design models of serving robot in the market:

"Servi" is a robot that autonomously travels and carries food when you select a table to serve and tap it. It can move while detecting and avoiding obstacles such as people and objects.

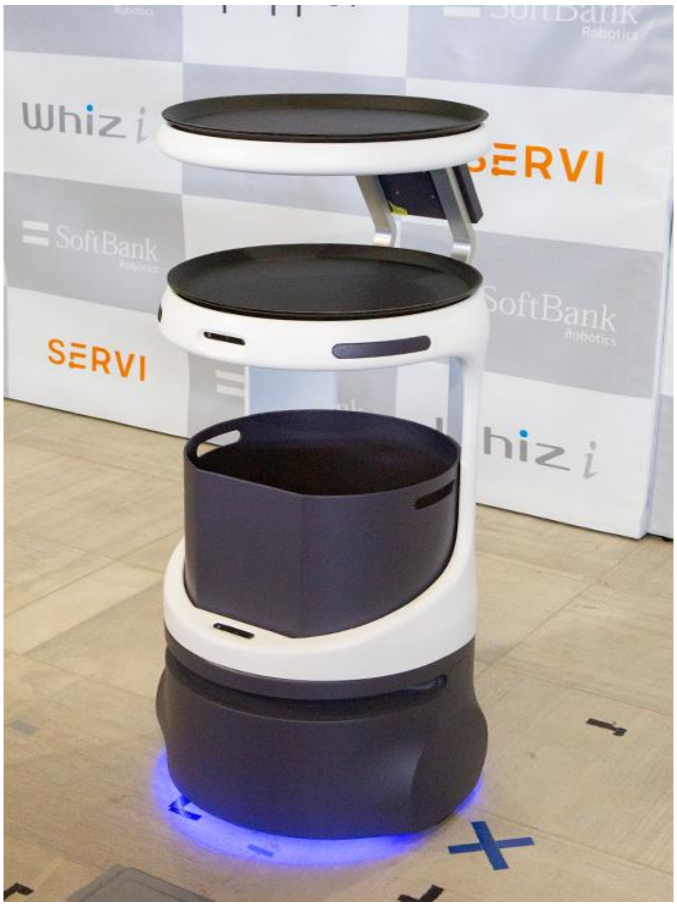


Figure 1.4.1.1 Softbank Robotics automated serving robot "Servi" panoramic view

Table 1.4.1.1 "Servi" Robot specification

Main specs	***Value***
Maximum Load Weight	35kg
Navigation Method	SLAM
Sensor	LiDAR, 3D Camera
Communication	Wi-Fi 2.4/5 GHz
Maximum speed	0.6 m/s

Beside "Servi" robot, there are many other robotics company introduced their invention of serving robot. These robot designs share the same techniques such as: motion obstacle avoidance, map construction, autonomous navigation, and communication. With the modern technology applied inside, serving robot can totally replace human in performing simple tasks.

Figure 1.4.1.2 Matradee Robot from Richtech robotics

Figure 1.4.1.3 DSR02-A Open Type from YZ Robot

1.3 Serving mobile robot design planning

Our project’s purpose is to design a service mobile robot, which can be used for delivering at restaurants or even in hospitals.

The robot is designed with three trays to place food and the base to hold drink.

The driving mechanism we selected is differential drive with one castor wheel to support the vehicle and prevent tilting. Both main wheels are placed on a common axis. The velocity of each wheel is controlled by a separated motor.

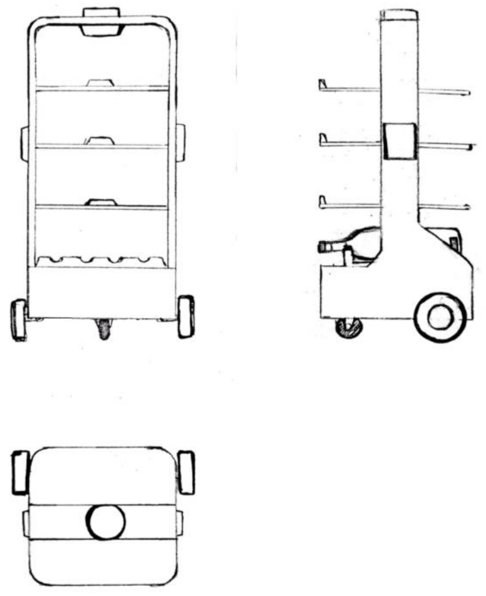


Figure 1.4.1.1 2D design view

Almost all the hardware components include embedded computer, motor driver, sensor, etc are put inside the robot base. A lidar is placed on the top for localization. Screen and speaker to help user to interface with the robot.

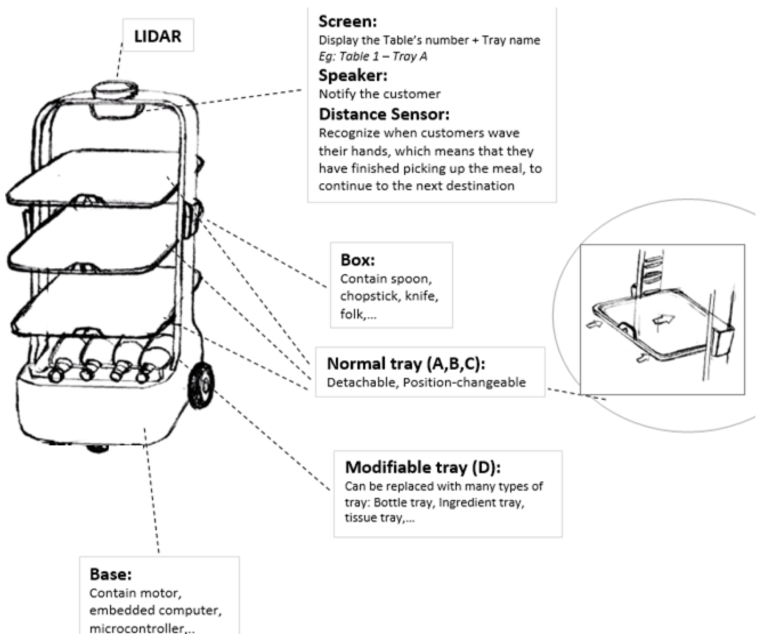

LIDAR
Screen:
Display the Table's number + Tray name
Eg: Table 1 – Tray A
Speaker:
Notify the customer
Distance Sensor:
Recognize when customers wave their hands, which means that they have finished picking up the meal, to continue to the next destination
Box:
Contain spoon, chopstick, knife, folk,...
Normal tray (A,B,C):
Detachable, Position-changeable
Modifiable tray (D):
Can be replaced with many types of tray: Bottle tray, Ingredient tray, tissue tray,...
Base:
Contain motor, embedded computer, microcontroller,..

Figure 1.4.1.2 Hardware components placement

Detailed specification:

+ Sensor: LiDAR, Ultrasonic sensor, IMU

+ Navigation method: SLAM

+ Body weight: 7 kg

+ Maximum load weight: 15 kg

+ Maximum speed: 0.5 m/s

1.4 Mathematical motion modeling

1.4.1 Differential drive kinematics

Kinematic model describes geometric relationships that are presented in the system. It describes the relationship between input (control) parameters and behavior of a system given by state-space representation.

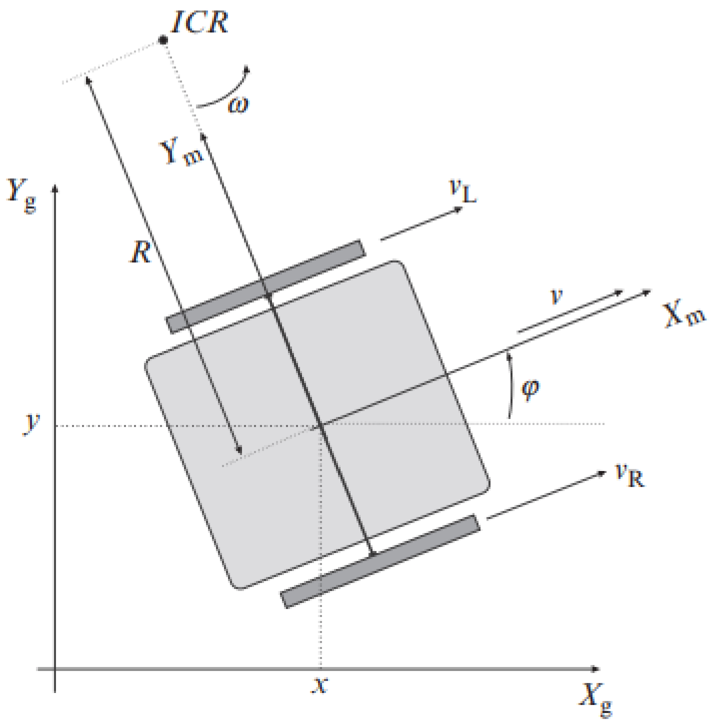


Figure 1.4.1.1 Differential drive kinematics

For differential drive mechanism, the input (control) variables are the velocity of the right wheel $v_R(t)$ and the velocity of the left wheel $v_L(t)$. The meanings of other variables:

+ r: wheel radius

+ L: the distance between the wheels

+ $R(t)$: the instantaneous radius of the vehicle driving trajectory (the distance between the vehicle center (middle point between the wheels) and ICR point)

In each instance of time both wheels have the same angular velocity ω(t) around the ICR:

$$\omega = \frac{v_L}{R(t) - \frac{L}{2}} = \frac{v_R}{R(t) + L/2} \qquad \textit{Equation 1}$$

From where $\omega(t)$ and $R(t)$ are computed:

$$\omega(t) = \frac{v_R(t) - v_L(t)}{L} \qquad \textit{Equation 2}$$

$$R(t) = \frac{L}{2}\frac{v_R(t) + v_L(t)}{v_R(t) - v_L(t)} \qquad \textit{Equation 3}$$

And tangential velocity is calculated as:

$$v(t) = \omega(t)R(t) = \frac{v_R(t) + v_L(t)}{L} \qquad \textit{Equation 4}$$

Considering the above relations, the internal kinematics (in local coordinates) can be expressed as:

$$\begin{bmatrix} \dot{x}_m(t) \\ \dot{y}_m(t) \\ \dot{\varphi}(t) \end{bmatrix} = \begin{bmatrix} v_{Xm}(t) \\ v_{Ym}(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ 0 & 0 \\ -\frac{r}{L} & \frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_L(t) \\ \omega_R(t) \end{bmatrix} \qquad \textit{Equation 5}$$

And robot external kinematics (in global coordinates) is given by

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\varphi}(t) \end{bmatrix} = \begin{bmatrix} \cos(\varphi(t)) & 0 \\ \sin(\varphi(t)) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} \qquad \textit{Equation 6}$$

In discrete form, by using Euler integration and evaluated at discrete time instants $t = kT_s, k = 0, 1, 2, \ldots$ where T_s is the following sampling interval, equation 6 can become:

$$\begin{aligned} x(k+1) &= x(k) + v(k)T_s \cos(\varphi(k)) \qquad \textit{Equation 7} \\ y(k+1) &= y(k) + v(k)T_s \sin(\varphi(k)) \\ \varphi(k+1) &= \varphi(k) + \omega(k)T_s \end{aligned}$$

1.4.2 Dynamic model of differential drive robot

For the equation 6, we get the kinematic model of the vehicle is:

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

The nonholonomic motion constraint is:

$$-\dot{x}\sin(\varphi) + \dot{y}\cos(\varphi) = 0 \qquad \textit{Equation 8}$$

Dynamic Model:

The obtained dynamic model written in matrix form is:

$$\mathrm{M(q)\ddot{q} + V(q,\dot{q}) + F(\dot{q}) = E(q)u - A^T(q)\lambda} \qquad \textit{Equation 9}$$

Table 1.4.2.1 Meaning of matrices in the dynamic model

$\boldsymbol{q}$	Vector of generalized coordinates (dimension n × 1)
$\boldsymbol{M(q)}$	Positive-definite matrix of masses and inertia (dimension n × n)
$\mathrm{V(q,\dot{q})}$	Vector of Coriolis and centrifugal forces (dimension n × 1)
$\mathrm{F(\dot{q})}$	Vector of Coriolis and centrifugal forces (dimension n × 1)
$\mathrm{E(q)}$	Transformation matrix from actuator space to generalized coordinate space (dimension n × r)
u	Input vector (dimension r × 1)
$\mathrm{A^T(q)}$	Matrix of kinematic constraint coefficients (dimension m × n)
λ	Vector of constraint forces (Lagrange multipliers) (Dimension m × 1)

Where matrices are:

$$\boldsymbol{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & J \end{bmatrix}$$

$$\boldsymbol{E} = \frac{1}{r}\begin{bmatrix} cos\varphi & cos\varphi \\ sin\varphi & sin\varphi \\ \frac{L}{2} & -\frac{L}{2} \end{bmatrix}$$

$$\boldsymbol{A} = [-sin\varphi \cos\varphi \;\; 0]$$

$$\boldsymbol{u} = \begin{bmatrix} \tau_r \\ \tau_l \end{bmatrix}$$

And the remaining matrices are zero.

With m is the mass, J is the inertia, τ_r and τ_l is the torque on the left and the right wheel.

The common state-space model that include the kinematic and dynamic model is determined by matrices:

$$\widetilde{\boldsymbol{M}} = \begin{bmatrix} m & 0 \\ 0 & J \end{bmatrix}$$

$$\widetilde{\boldsymbol{V}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\widetilde{\boldsymbol{E}} = \frac{1}{r}\begin{bmatrix} 1 & 1 \\ \frac{L}{2} & -\frac{L}{2} \end{bmatrix}$$

Then the system can be written in the state-space form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{g}(\boldsymbol{x}).\boldsymbol{u}$, where the state vector is $\boldsymbol{x} = [\boldsymbol{q^T}, \boldsymbol{v^T}]^{\boldsymbol{T}}$

The resulting model is

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\varphi}\\ \dot{v}\\ \dot{\omega}\end{bmatrix}=\begin{bmatrix}v\cos\varphi\\ v\sin\varphi\\ \omega\\ 0\\ 0\end{bmatrix}+\begin{bmatrix}0 & 0\\ 0 & 0\\ 0 & 0\\ \frac{1}{mr} & \frac{1}{mr}\\ \frac{L}{2Jr} & -\frac{L}{2Jr}\end{bmatrix}\begin{bmatrix}\tau_r\\ \tau_l\end{bmatrix} \qquad \textit{Equation 10}$$

CHAPTER 2. SERVING MOBILE ROBOT ANALYSIS AND DESIGN

Table 2.1. Working task and Contribution

Name	Main Tasks	Contribution
Tran Minh Duc	ROS system setup; Navigation Stack implementation, Robot model design; Writing report	10/10
Ly Duc Trung	Setup Ubuntu Operating system and ROS system. Programming Kalman Filter, PID controller, differential driving, ultrasonic sensor publisher. Writing report.	10/10
Tran Tuan Minh	Hardware design; Motor speed controller research; Writing report & slide design	10/10

2.1 SERVING MOBILE ROBOT HARDWARE DESIGN

2.1.1 Mobile robot function blocks

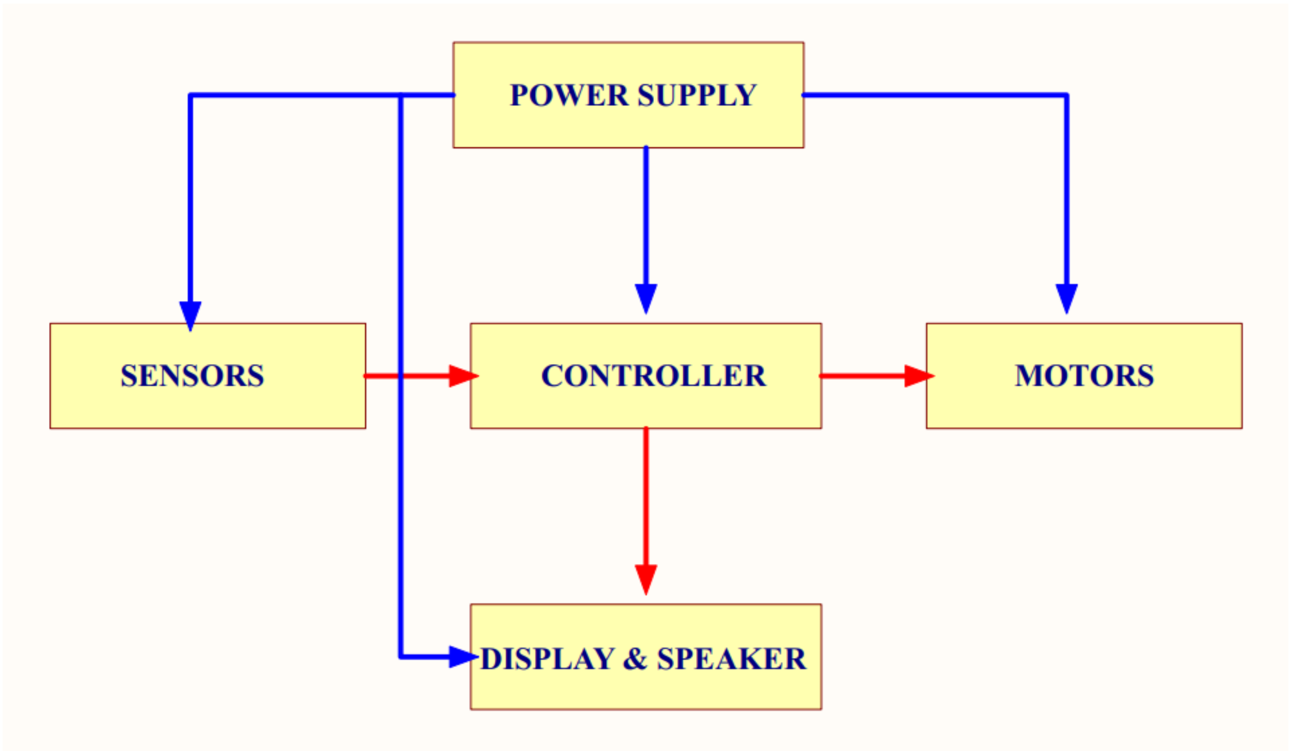


Figure 2.1.1.1 Serving Robot Function Block Diagram

2.1.2 Blocks selection

a) Sensor block

For localizing and mapping the surrounding, a scanning LiDAR sensor is applied.

LiDAR follows a simple principle — throw laser light at an object on the earth surface and calculate the time it takes to return to the LiDAR source. Given the speed at which the light travels (approximately 186,000 miles per second), the process of measuring the exact distance through LiDAR appears to be incredibly fast. The formula that analysts use to arrive at the precise distance of the object is as follows:

$$The\ distance\ of\ the\ object = \frac{Speed\ of\ Light\ x\ Time\ of\ Flight}{2}$$

The data collected from LiDAR is then analyzed to build 2D map.

In this project, we use LiDAR Scanse Sweep:

+ Range: 40m

+ Resolution: 1cm

+ Sample Rate: Up to 1075Hz (500Hz default)

+ Frequency: 1-10Hz (Adj)

+ Interface: 3.3V (5V tolerant) UART TTL

+ Power: 5V @ 450mA to 650ma USB Plug and Play

Figure 2.1.2.1 LiDAR Scanse Sweep

To detect the obstacles having lower height than LiDAR’s position placement, ultrasonic sensors are taken into account. The working principle of ultrasonic sensor in measuring distance of objects is somehow the same as LiDAR, except that it sends an ultrasonic pulse out and if there is an obstacle or object, it will bounce back to the sensor.

In our project, ultrasonic sensors HC-SR04 is selected:

+ Operating voltage: 5V

+ Working current: 15mA

+ Ranging distance: 2cm-4m

+ Effectual angle: <15°

+ Measuring angle: 30°

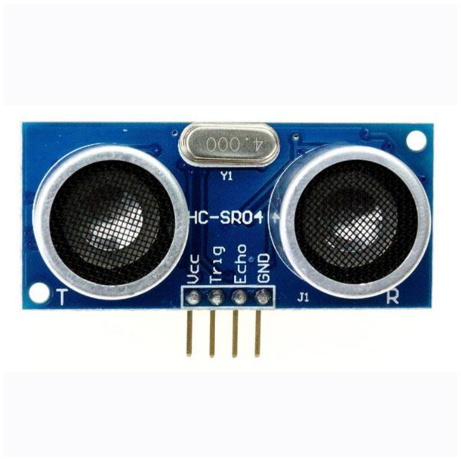


Figure 2.1.2.2 HC-SR04 Ultrasonic sensor

Beside LiDAR and ultrasonic sensor for mapping and distance measuring, some other sensors are also implemented in this project.

To measuring speed and control motors, rotary encoders are placed in motors shaft. The rotary encoder can detect the speed and direction of motors by detecting output signal generated from two channels A and B. About rotary encoder specification, we will discuss further in motor block selection.

For better measurement of motion processing, a MEMS (Micro Electro-mechanical system) MPU6050 is used. It consists of three-axis accelerometer and three-axis gyroscope to calculate velocity, orientation, acceleration, displacement, and other motion. The module also supports I2C interface to communicate with microcontroller.

MPU6050 specifications:

+ Operating voltage: 3-5 VDC.

+ Acceleration range: $\pm 2g, \pm 4g, \pm 8g, \pm 16g$.

+ Operating current: 4mA.

+ Gyroscope range: $\pm 250, \pm 500, \pm 1000, \pm 2000\ °/s$

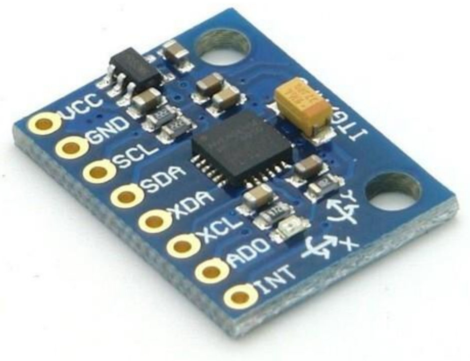


Figure 2.1.2.3 MPU 6050 module

Beside measuring robot states such as velocity and position, robot power capability is also taken into consideration. By referring to voltage on the battery, the system can send alarm to user on an occasion when the battery is nearly using up. The voltage after being divided by resistors is read by an ADC converter. Here, an 8-bit A/D PCF8591 converter is used.

PCF8591 specifications:

+ Operating voltage range: 2.5V – 6V.

+ Communication via I2C interface.

+ 8-bit successive approximation A/D converter.

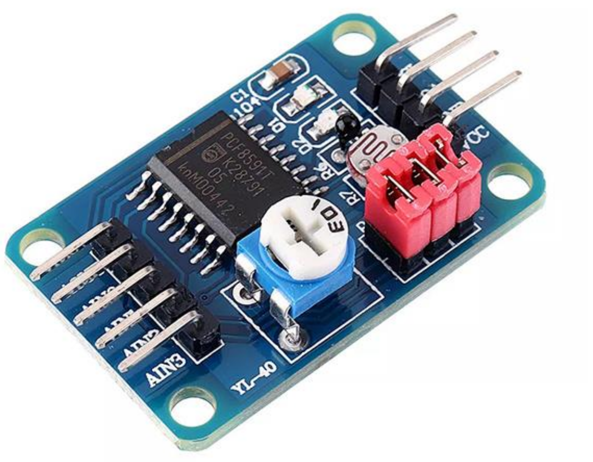


Figure 2.1.2.4 PCF 8591 AD/DA converter module

b) Motors block

We want to control the speed and increase the torque of the motor, so a DC gearhead motor with encoder is selected.

To calculate and select appropriated motors for this project, some variables of the robot must be considered:

+ Wheel radius: r =0.045m.

+ Maximum load weight: $m = 15kg$

+ Maximum speed: $v = 0.5\ m/s$

+ Acceleration: $a = 0.2m/s^2$

+ Maximum inclined angle: $\theta = 5°$

Assume that there was enough friction between the wheel and the surface so there is no slip.

To properly size the motor, we focus on the situation where the robot is accelerating from rest to full speed. The torque required to get the robot moving can be much greater than keeping it in motion.

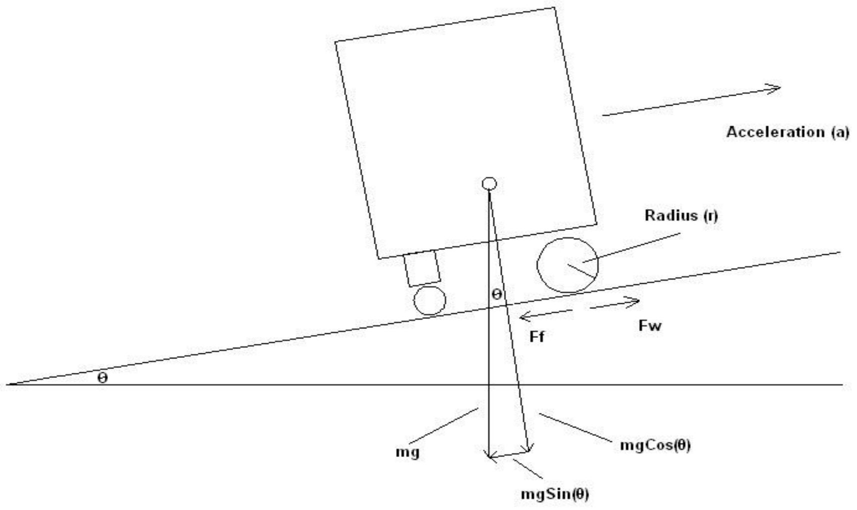


Figure 2.1.2.5 Forces act on the robot

We have:

$$\sum Force = f_{total} = f_w - f_g = ma$$

$$f_w = ma + f_g$$

$$\frac{T}{r} = ma + msin\theta$$

$$\boldsymbol{T = m(a + gsin\theta)r} \qquad \textit{Equation 11}$$

With given value, we can calculate the maximum torque required for the motor:

$$T_{max} = 15.(0.2 + 9.8\sin(5)).0.045 = 0.71(Nm) \qquad \textit{Equation 12}$$

This is the total torque required to drive the robot. Since we are using two drive motors: $T_{permotor} = 0.35Nm$

Next, we will determine how fast in rpms the motor need to turn:

$v.\frac{60}{2\pi r} = 106.1\ (RPM)$

Finally, we calculate power one motor is required to supply:

$$P_{mech} = T_{permotor}.\omega = T_{permotor}.\frac{v}{r} = \frac{0.35 \times 0.5}{0.045} = 3.89\ (W)$$

Based on the calculated value: $T_{permotor(max)}$, $RPM\ value,$ and P_{mech}, we can choose suitable DC motors.

Here, we choose DC motor JGB37-545:

Specifications:

+ Rated voltage: 12V

+ No load speed: 208 RPM.

+ No load current: I_O =200 mA.

+ Load Torque speed: 176 RPM

+ Rated Load torque: 0.22 Nm

+ Stall torque: 1.5Nm

+ Stall current: I_S =5 A

+ Reducer ratio: 1:30

Motor encoder specifications:

+ Operating voltage: 3-5 VDC

+ Hall sensor with 2 channel A & B for determining direction and speed.

+ 16 pulses/channel/round encoder. For one rotation of motor shaft, the pulse output is 480 pulse/channel.

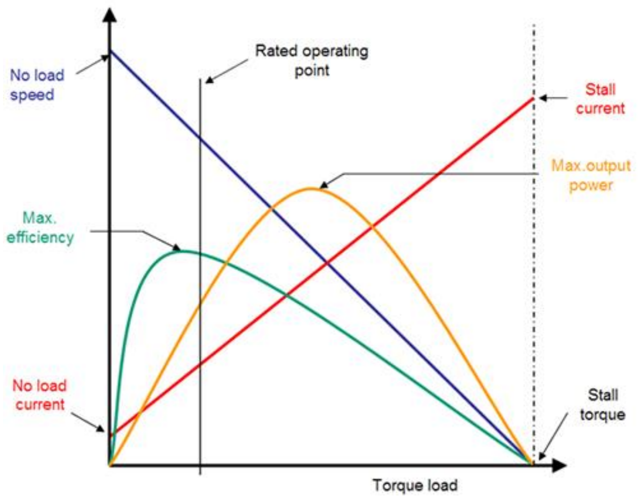


Figure 2.1.2.6 DC motor characteristic curve for speed, current, torque and efficiency

From the mechanical characteristic of DC motor, the maximum torque calculated is 0.35Nm/ motor, so the maximum current & maximum power supply to the motor is:

$$I_{max} = I_O + (I_S - I_O).\frac{0.35}{1.5} = 1.32(A)$$

$$P_{max} = 12 \times 1.32 = 15.84(W)$$

After selecting motors, a motor driver is also needed to receive control signal from the controller. The motor driver is required to handle the motor current, high efficiency with low voltage drop in H-bridge.

In this project, we decide to use motor driver TB6612FNG:

Specifications for each of the two H-bridges:

+ Motor supply voltage: 2.5 - 13.5 VDC.

+ Logic supply voltage: 2.7 - 5.5 VDC.

+ Output current continuous: 1.2 amperes.

+ Output current peak: 3.2 amperes.

+ Voltage drop: 0.15 – 0.8 VDC

+ Built-in thermal shutdown.

+ Standby power mode.

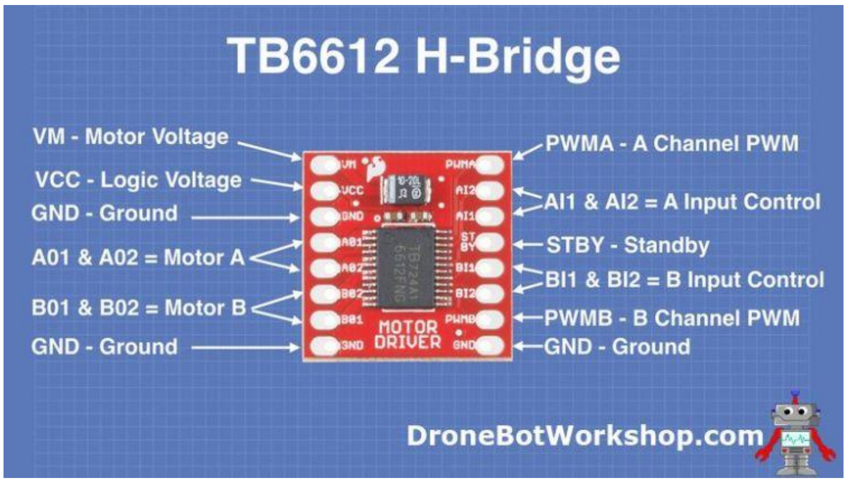


Figure 2.1.2.7 TB6612FNG motor driver (Source: DroneBotWorkshop.com)

c) Output display & Speaker

To communicate with guest, a small OLED is implemented in front of the robot:

The robot is also equipped with a speaker to interact with guest when it approaches the desired table.

Here, 5W analog speaker is applied.

d) Controller

In this project, the controller plays a vital part of controlling system and communication. The chosen controller must be able to read data from sensors, control motors and do the navigation tasks, then communicate with the computer.

An embedded computer is suitable for managing all above-mentioned problems.

In this project, we choose embedded computer Raspberry Pi 4B+:

Hardware:

- Quad core 64-bit ARM-Cortex A72 running at 1.5GHz
- 1, 2 and 4 Gigabyte LPDDR4 RAM options
- Video Core VI 3D Graphics
- Supports dual HDMI display output up to 4Kp60

Software:

- 802.11 b/g/n/ac Wireless LAN
- Bluetooth 5.0 with BLE
- 2x USB2 ports & 2x USB3 ports
- 28x user GPIO supporting various interface options:

+ Up to 6x UART
+ Up to 6x I2C.
+ Up to 5x SPI
+ 1x SDIO interface.
+ 1x DPI (Parallel RGB Display).
+ 1x PCM–Up to 2x PWM channels.

+ Up to 3x GPCLK outputs.

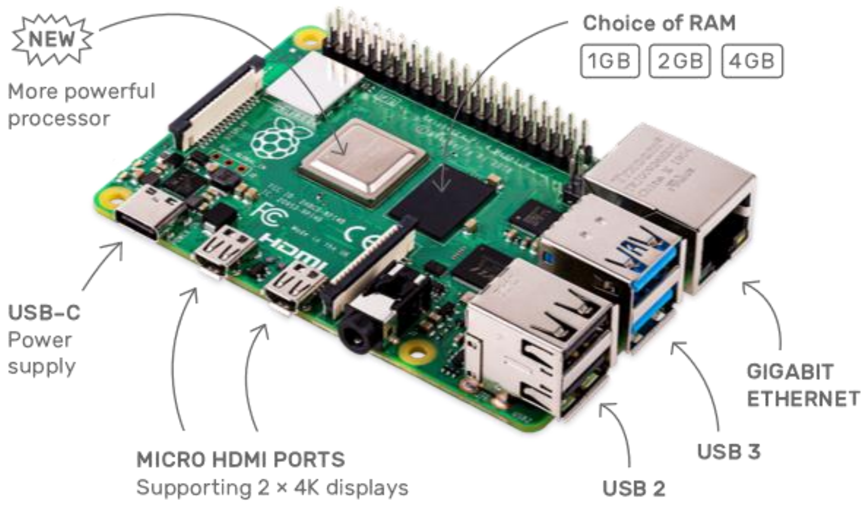


Figure 2.1.2.8 Raspberry Pi 4B+ model

d) Power supply block

To get enough power for other blocks: controller, sensors, motors drive, a LiPo battery is used.

Block	Name	Power (W)	Detailed specs
Motor Driver	TB6612FNG	1.36	
	Motor JGB37-545DC x 2	31.68	12V-1.32A
Controller	Raspberry Pi 4B+	15	5V-3A
Sensor	LiDAR Scanse	3.25 (Powered by Raspberry Pi)	5V-650mA
	MPU6050	0.013	3.3V-3.9mA
	HC-SR04 x5	0.3	5V-15mA x 5
	IR Sensor	0.5	5V-20mA
	PCF8591	0.005	5V-1mA
Screen & Display	Speaker	5	
	I2C OLED	0.08	
	Total	54W	

The power calculated for the system is about 54W & the estimated continuous working hours is about 3 hours.

LiPo battery specifications:

+ Maximum voltage: 12.6 VDC

+ Capacity: 14Ah

+ Continuous discharge current: 39A (3C)

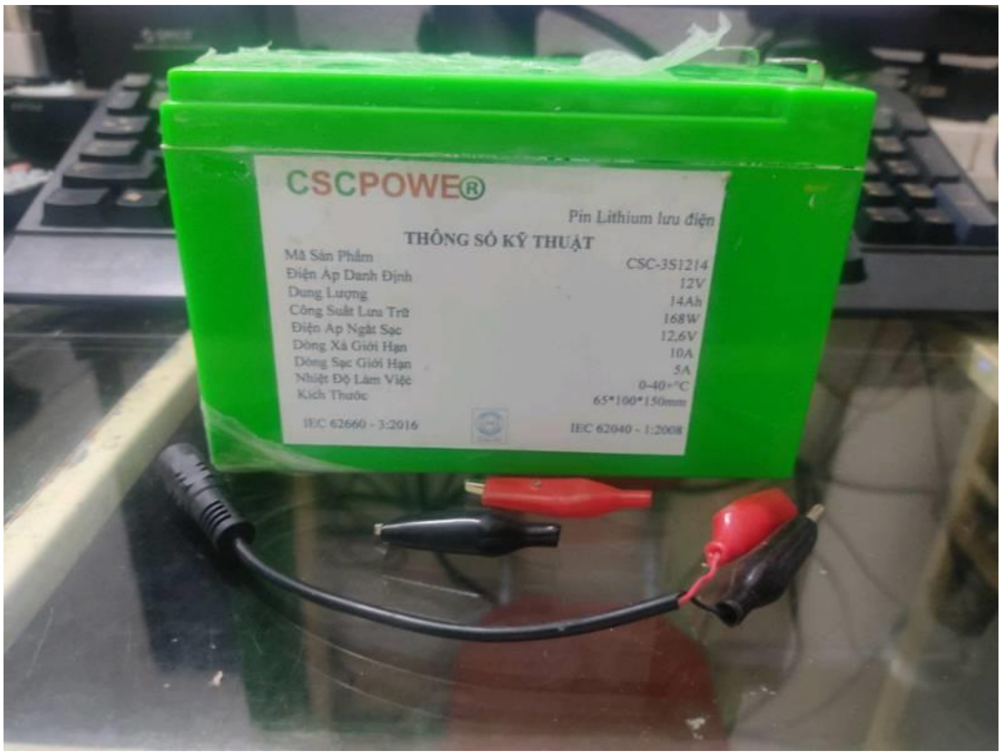


Figure 2.1.2.9 12V 14Ah LiPo battery

The battery can directly supply the motor driver, and also is regulated to 5 VDC and 3.3VDC to supply the controller and other sensors by DC/DC buck converters.

XY-3606 DC/DC Buck Converter:

+ Input: 9-24 VDC.

+ Output: 5.2VDC/ 5A/ 25W

LM2596 DC/DC Buck Converter:

+ Input: 3VDC – 30VDC.

+ Output: 1.5VDC – 35VDC.

+ Maximum current: 3A.

Combining all blocks together, we get the detailed hardware system diagram.

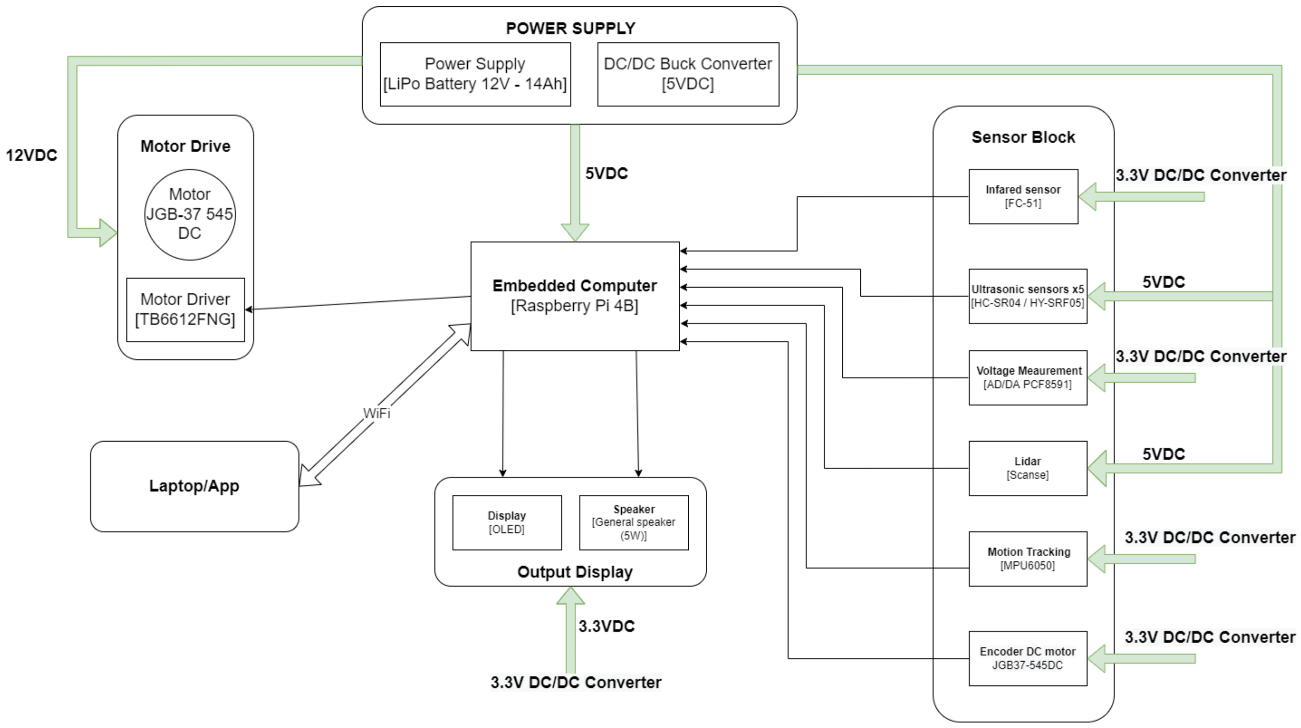


Figure 2.1.2.10 Hardware system design

2.1.3 Hardware implementation

From the planning idea, we move to the detailed design for robot frame.

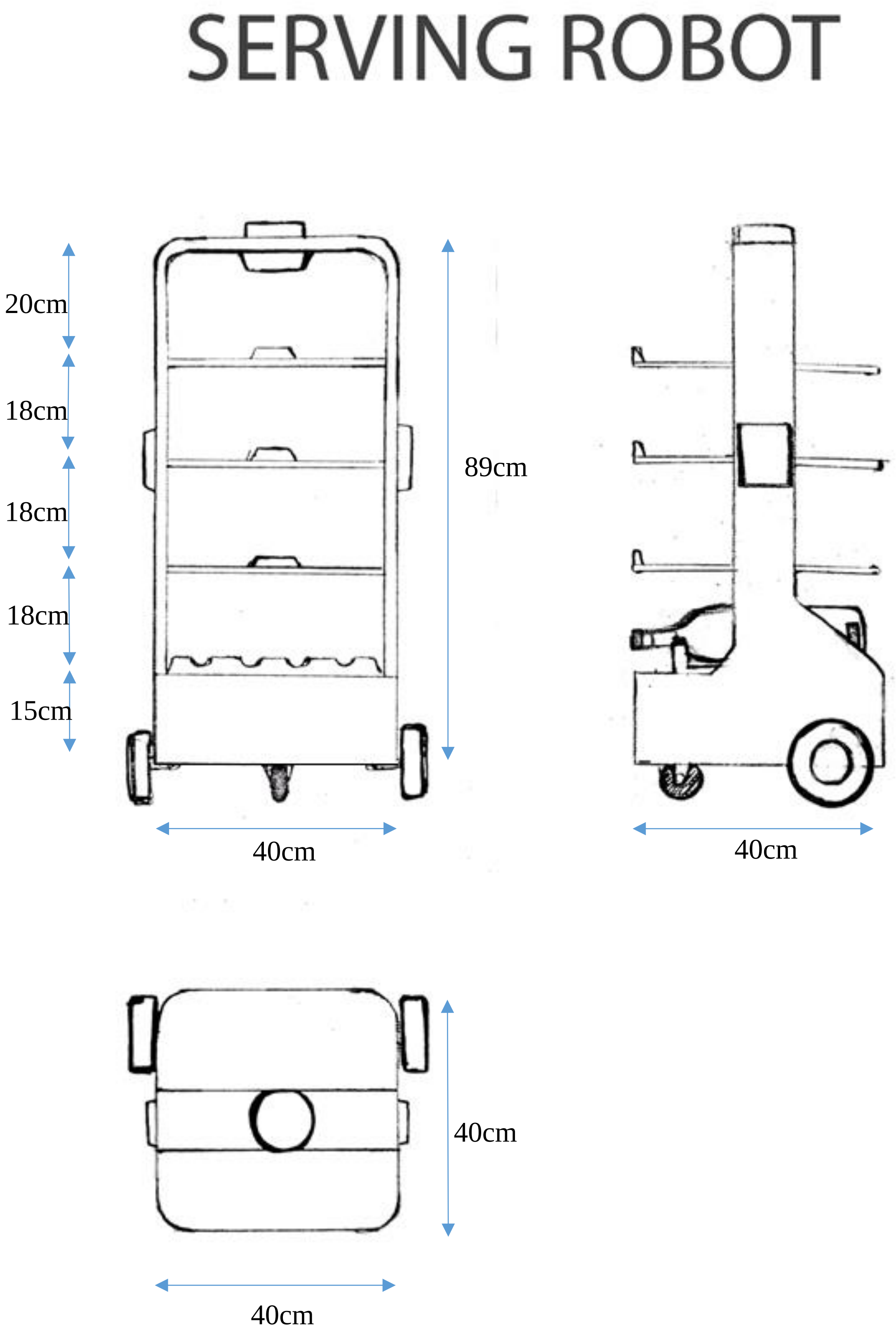
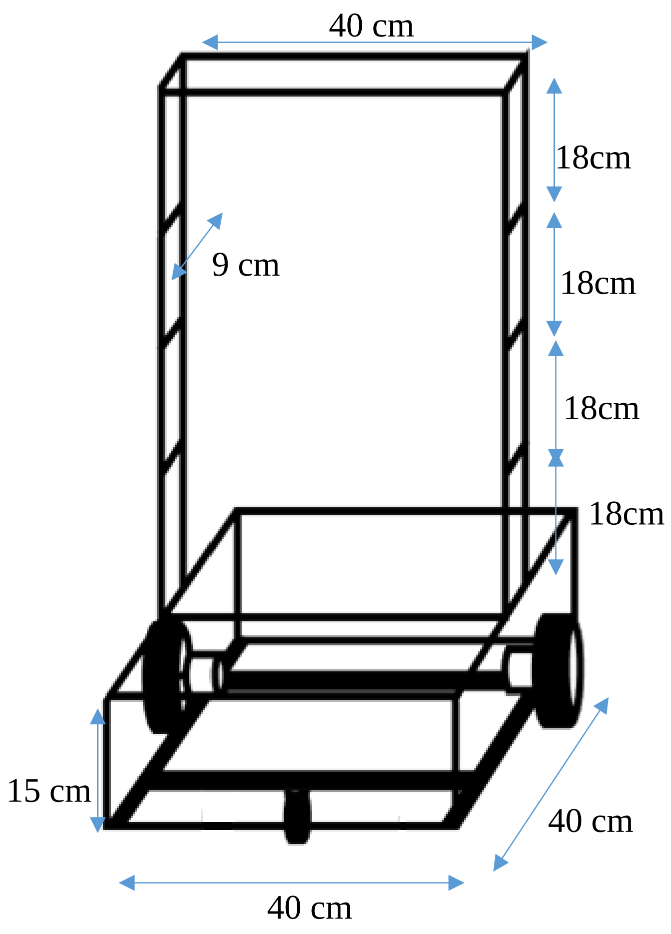


Figure 2.1.3.1 2D detailed design parameter

The material for building robot frame is stainless steel. After building robot frame, it was shielded by mica material.



S

Figure 2.1.3.2 Robot frame design parameters

The mass of frame is estimated about 3.5kg and the other 3.5kg is approximated for the shield, hardware components and trays.

Some pictures below show the design of serving robot in reality:

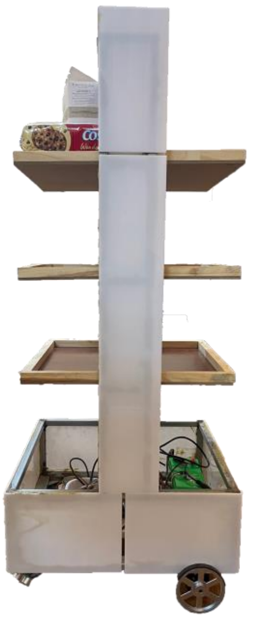


Figure 2.1.3.3 Side View

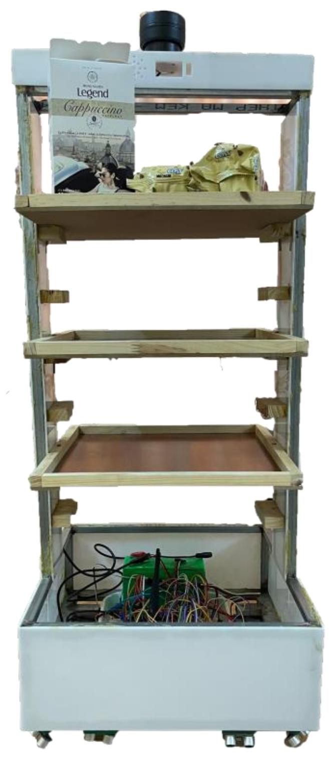


Figure 2.1.3.4 Front View

2.2 SERVING ROBOT FIRMWARE DESIGN

2.2.1 ROS and important concepts

ROS provides standard operating system services such as hardware abstraction, device drivers, implementation of commonly used features including sensing, recognizing, mapping, motion planning, message passing between processes, package management, visualizers and libraries for development as well as debugging tools.

Important concept of ROS:

2.2.1.1. Master

The master acts as a name server for node-to-node connections and message communication. The command roscore is used to run the master, and if you run the master, you can register the name of each node and get information when needed. The connection between nodes and message communication such as topics and services are impossible without the master.

The master communicates with slaves using XMLRPC (XML-Remote Procedure Call), which is an HTTP-based protocol that does not maintain connectivity. In other words, the slave nodes can access only when they need to register their own information or request information of other nodes. The connection status of each other is not checked regularly. Due to this feature, ROS can be used in very large and complex environments. XMLRPC is very lightweight and supports a variety of programming languages, making it well suited for ROS, which supports variety of hardware and programming languages.

When you execute ROS, the master will be configured with the URI address and port configured in the ROS_MASTER_URI. By default, the URI address uses the IP address of local PC, and port number 11311, unless otherwise modified.

2.2.1.2. Node

ROS recommends creating one single node for each purpose, and it is recommended to develop for easy reusability. For example, in case of mobile robots, the program to operate the robot is broken down into specialized functions. Specialized node is used for each function such as sensor drive, sensor data conversion, obstacle recognition, motor drive, encoder input, and navigation.

Upon startup, a node registers information such as name, message type, URI address and port number of the node. The registered node can act as a publisher, subscriber, service server or service client based on the registered information, and nodes can exchange messages using topics and services.

The node uses XMLRPC for communicating with the master and uses XMLRPC or TCPROS of the TCP/IP protocols when communicating between nodes. Connection request and response between nodes use XMLRPC, and message communication uses TCPROS because it is a direct communication between nodes independent from the master. As for the URI address and port number, a variable called ROS_HOSTNAME, which is stored on the computer where the node is running, is used as the URI address, and the port is set to an arbitrary unique value.

2.2.1.3. Package

A package is the basic unit of ROS. The ROS application is developed on a package basis, and the package contains either a configuration file to launch other packages or nodes. The package also contains all the files necessary for running the package, including ROS dependency libraries for running various processes, datasets, and configuration file. The number of official packages is about 2,500 for ROS Indigo as of July 2017 (http://repositories.ros.org/status_page/ ros_indigo_ default.html) and about 1,600 packages for ROS Kinetic (http://repositories.ros.org/status_page/ ros_kinetic_default.html). In addition, although there could be some redundancies, there are about 4,600 packages developed and released by users (http://rosindex.github.io/stats/).

2.2.1.4. Metapackage

A metapackage is a set of packages that have a common purpose. For example, the Navigation metapackage consists of 10 packages including AMCL, DWA, EKF, and map_server.

2.2.1.5. Message

A node sends or receives data between nodes via a message. Messages are variables such as integer, floating point, and Boolean. Nested message structure that contains other messages or an array of messages can be used in the message.

TCPROS and UDPROS communication protocol is used for message delivery. Topic is used in unidirectional message delivery while service is used in bidirectional message delivery that request and response are involved.

2.2.1.6. Topic

The topic is literally like a topic in a conversation. The publisher node first registers its topic with the master and then starts publishing messages on a topic. Subscriber nodes that want to receive the topic request information of the publisher node corresponding to the name of the topic registered in the master. Based on this information, the subscriber node directly connects to the publisher node to exchange messages as a topic.

2.2.1.7. Publish and Publisher

The term ‘publish’ stands for the action of transmitting relative messages corresponding to the topic. The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic. The publisher is declared in the node and can be declared multiple times in one node.

2.2.1.8. Subscribe and Subscriber

The term ‘subscribe’ stands for the action of receiving relative messages corresponding to the topic. The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topic from the master. Based on received publisher information, the subscriber node directly requests connection to the publisher node and receives messages from the connected publisher node. A subscriber is declared in the node and can be declared multiple times in one node.

The topic communication is an asynchronous communication which is based on publisher and subscriber, and it is useful to transfer certain data. Since the topic continuously transmits and receives stream of messages once connected, it is often used for sensors that must periodically transmit data. On the other hands, there is a need for synchronous communication with which request and response are used. Therefore, ROS provides a message synchronization method called ‘service’. A service consists of the service server that responds to requests and the service client that requests to respond. Unlike the topic, the service is a one-time message communication. When the request and response of the service is completed, the connection between two nodes is disconnected.

2.2.1.9. Service

The service is synchronous bidirectional communication between the service client that requests a service regarding a particular task and the service server that is responsible for responding to requests.

2.2.1.10. Service Server

The ‘service server’ is a server in the service message communication that receives a request as an input and transmits a response as an output. Both request and response are in the form of messages. Upon the service request, the server performs the designated service and delivers the result to the service client as a response. The service server is implemented in the node that receives and executes a given request.

2.2.1.11. Service Client

The ‘service client’ is a client in the service message communication that requests service to the server and receives a response as an input. Both request and response are in the form of message. The client sends a request to the service server and receives the response.

The service client is implemented in the node which requests specified command and receives results.

2.2.1.12. Action

The action is another message communication method used for an asynchronous bidirectional communication. Action is used where it takes longer time to respond after receiving a request and intermediate responses are required until the result is returned. The structure of action file is also similar to that of service. However, feedback data section for intermediate response is added along with goal and result data section which are represented as request and response in service respectively. There are action clients that set the goal of the action and action server that performs the action specified by the goal and returns feedback and result to the action client.

2.2.1.13. Action Server

The 'action server' is in charge of receiving goal from the client and responding with feedback and result. Once the server receives goal from the client, it performs predefined process.

2.2.1.14. Action Client

The 'action client' is in charge of transmitting the goal to the server and receives result or feedback data as inputs from the action server. The client delivers the goal to the action server, then receives corresponding result or feedback, and transmits follow up instructions or cancel instruction.

2.2.1.15. Parameter

The parameter in ROS refers to parameters used in the node. Think of it as *.ini configuration files in Windows program. Default values are set in the parameter and can be read or written if necessary. In particular, it is very useful when configured values can be modified in real-time. For example, you can specify settings such as USB port number, camera calibration parameters, maximum and minimum values of the motor speed.

2.2.1.16. Parameter Server

When parameters are called in the package, they are registered with the parameter server which is loaded in the master.

2.2.1.17. Catkin

The catkin refers to the build system of ROS. The build system basically uses CMake (Cross Platform Make), and the build environment is described in the 'CMakeLists.txt' file in the package folder. CMake was modified in ROS to create a ROS-specific build system. Catkin started the alpha test from ROS Fuerte and the core packages began to switch to Catkin in the ROS Groovy version. Catkin has been applied to most packages in the ROS Hydro version. The Catkin build system makes it easy to use ROS-related builds, package management, and dependencies among packages. If you are going to use ROS at this point, you should use Catkin instead of ROS build (rosbuild).

2.2.1.18. ROS Build

The ROS build (rosbuild) is the build system that was used before the Catkin build system. Although there are some users who still use it, this is reserved for compatibility of

ROS, therefore, it is officially not recommended to use. If an old package that only supports the rosbuild must be used, we recommend using it after converting rosbuild to catkin.

2.2.1.19. roscore

roscore is the command that runs the ROS master. If multiple computers are within the same network, it can be run from another computer in the network. However, except for special case that supports multiple roscore, only one roscore should be running in the network. When ROS master is running, the URI address and port number assigned for ROS_MASTER_URI environment variables are used. If the user has not set the environment variable, the current local IP address is used as the URI address and port number 11311 is used which is a default port number for the master.

2.2.1.20. rosrun

rosrun is the basic execution command of ROS. It is used to run a single node in the package. The node uses the ROS_HOSTNAME environment variable stored in the computer on which the node is running as the URI address, and the port is set to an arbitrary unique value.

2.2.1.21. roslaunch

While rosrun is a command to execute a single node, roslaunch in contrast executes multiple nodes. It is a ROS command specialized in node execution with additional functions such as changing package parameters or node names, configuring namespace of nodes, setting ROS_ ROOT and ROS_PACKAGE_PATH, and changing environment variables when executing nodes.

roslaunch uses the '*.launch' file to define which nodes to be executed. The file is based on XML (Extensible Markup Language) and offers a variety of options in the form of XML tags.

2.2.1.22. bag

The data from the ROS messages can be recorded. The file format used is called bag, and '*.bag' is used as the file extension. In ROS, bag can be used to record messages and play them back when necessary to reproduce the environment when messages are recorded. For example, when performing a robot experiment using a sensor, sensor values are stored in the message form using the bag. This recorded message can be repeatedly loaded without performing the same test by playing the saved bag file. Record and play functions of rosbag are especially useful when developing an algorithm with frequent program modifications.

2.2.1.23. ROS Wiki

ROS Wiki is a basic description of ROS based on Wiki (http://wiki.ros.org/) that explains each package and the features provided by ROS. This Wiki page describes the basic usage of ROS, a brief description of each package, parameters used, author, license, homepage, repository, and tutorial. The ROS Wiki currently has more than 18,800 pages of content.

2.2.1.24. Repository

An open package specifies repository in the Wiki page. The repository is a URL address on the web where the package is saved. The repository manages issues, development, downloads, and other features using version control systems such as svn, hg, and git. Many of currently available ROS packages are using GitHub as repositories for source code. In order to view the contents of the source code for each package, check the corresponding repository.

2.2.1.25. Graph

The relationship between nodes, topics, publishers, and subscribers introduced above can be visualized as a graph. The graphical representation of message communication does not include the service as it only happens one time. The graph can be displayed by running the ‘rqt_graph’ node in the ‘rqt_graph’ package. There are two execution commands, ‘rqt_graph’ and ‘rosrun rqt_graph rqt_graph’.

2.2.1.26. Name

Nodes, parameters, topics, and services all have names. These names are registered on the master and searched by the name to transfer messages when using the parameters, topics, and services of each node. Names are flexible because they can be changed when being executed, and different names can be assigned when executing identical nodes, parameters, topics, and services multiple times. Use of names makes ROS suitable for large-scale projects and complex systems.

2.2.1.27. Client Library

ROS provides development environments for various languages by using client library in order to reduce the dependency on the language used. The main client libraries are C++, Python, Lisp, and other languages such as Java, Lua, .NET, EusLisp, and R are also supported. For this purpose, client libraries such as roscpp, rospy, roslisp, rosjava, roslua, roscs, roseus, PhaROS, and rosR have been developed.

2.2.1.28. URI

A URI (Uniform Resource Identifier) is a unique address that represents a resource on the Internet. The URI is one of basic components that enables interaction with Internet and is used as an identifier in the Internet protocol.

2.2.1.29. MD5

MD5 (Message-Digest algorithm 5) is a 128-bit cryptographic hash function. It is used primarily to verify data integrity, such as checking whether programs or files are in its unmodified original form. The integrity of the message transmission/reception in ROS is verified with MD5.

2.2.1.30. RPC

RPC (Remote Procedure Call) stands for the function that calls a sub procedure on a remote computer from another computer in the network. RPC uses protocols such as TCP/IP and IPX, and allows execution of functions or procedures without having the developer to write a program for remote control.

2.2.1.31. XML

XML (Extensible Markup Language) is a broad and versatile markup language that W3C recommends for creating other special purpose markup languages. XML utilizes tags in order to describe the structure of data. In ROS, it is used in various components such as *.launch, *.urdf, and package.xml.

2.2.1.32. XMLRPC

XMLRPC (XML-Remote Procedure Call) is a type of RPC protocol that uses XML as the encoding format and uses the request and response method of the HTTP protocol which does not maintain nor check the connection. XMLRPC is a very simple protocol, used only to define small data types or commands. As a result, XMLRPC is very lightweight and supports a variety of programming languages, making it well suited for ROS, which supports a variety of hardware and languages.

2.2.1.33. TCP/IP

TCP stands for Transmission Control Protocol. It is often called TCP/IP. The Internet protocol layer guarantees data transmission using TCP, which is based on the IP (Internet Protocol) layer in the Internet Protocol Layers. It guarantees the sequential transmission and reception of data. TCPROS is a message format based on TCP/IP and UDPROS is a message format based on UDP. TCPROS is more frequently used in ROS.

2.2.1.34. CMakeLists.txt

Catkin, which is the build system of ROS, uses CMake by default. The build environment is specified in the ‘CMakeLists.txt’ file in each package folder.

2.2.1.35. package.xml

An XML file contains package information that describes the package name, author, license, and dependent packages.

2.2.2 Servebot Software Architecture

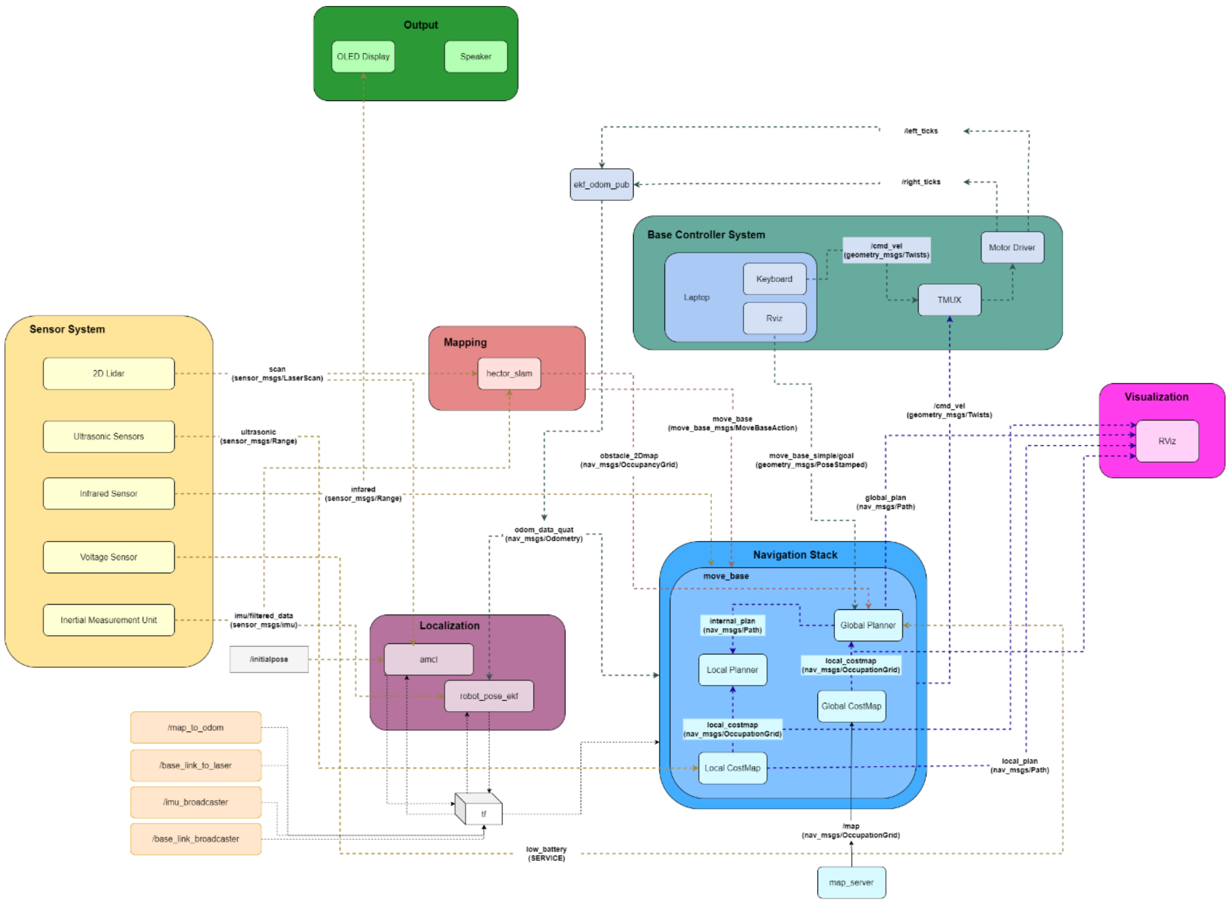

Output
OLED Display
Speaker
/left_ticks
/right_ticks
ekf_odom_pub
Base Controller System
Laptop
Keyboard
Rviz
/cmd_vel (geometry_msgs/Twists)
TMUX
Motor Driver
Sensor System
2D Lidar
Ultrasonic Sensors
Infrared Sensor
Voltage Sensor
Inertial Measurement Unit
scan (sensor_msgs/LaserScan)
ultrasonic (sensor_msgs/Range)
infared (sensor_msgs/Range)
imu/filtered_data (sensor_msgs/imu)
/initialpose
Mapping
hector_slam
move_base (move_base_msgs/MoveBaseAction)
obstacle_2Dmap (nav_msgs/OccupancyGrid)
move_base_simple/goal (geometry_msgs/PoseStamped)
/cmd_vel (geometry_msgs/Twists)
global_plan (nav_msgs/Path)
odom_data_quat (nav_msgs/Odometry)
Visualization
RViz
Localization
amcl
robot_pose_ekf
Navigation Stack
move_base
internal_plan (nav_msgs/Path)
Global Planner
Local Planner
local_costmap (nav_msgs/OccupationGrid)
Global CostMap
local_costmap (nav_msgs/OccupationGrid)
Local CostMap
local_plan (nav_msgs/Path)
/map_to_odom
/base_link_to_laser
/imu_broadcaster
/base_link_broadcaster
tf
low_battery (SERVICE)
/map (nav_msgs/OccupationGrid)
map_server

Figure 2.2.2.1 Servebot's software architecture

2.2.2.1. Sensor System Block:

Sensor System Block is used for making maps, localization, obstacles avoidance, interaction with customers and safety issues.

a. 2D lidar

- Position of hardware: Upper frame of ServeBot.
- Function: 2D lidar node is used to build global map and obstacle (local) map.

b. Ultrasonic Sensors

- Position of hardware: Use 5 sensors in the bottom level, placed around the front face of the ServeBot.
- Function: This node helps to detect near and moving obstacles, even lying objects on the floor like shoes,... due to their low placements.

c. Infrared Sensor

- Position of hardware: Below the top frame of robot

- Function: This node receives the hand-waving of the customer, which implies that food has been served at that table so that the robot can move to the next customer.

d. Voltage Sensor

- Position of hardware: In the base of ServeBot.
- Function: This node measures battery voltage for analysing the state of the battery.

e. Inertial Measurement Unit

- Position of hardware: In the base of ServeBot.
- Function: This node processes datas from an IMU sensor that can be used for localization besides the signal from motors's encoders.

2.2.2.2. Mapping Block

Mapping Block is used for building maps which includes 2D ('hector mapping' or 'gmapping') and 3D maps ('octomap'). For ServeBot, we use 2D maps with the 'hector mapping' method.

2.2.2.3. Localization Block:

a. Robot_pose_ekf:

The Robot Pose EKF package is used to estimate the 3D pose of a robot, based on (partial) pose measurements coming from different sources. It uses an extended Kalman filter with a 6D model (3D position and 3D orientation) to combine measurements from wheel odometry, IMU sensor and visual odometry. The basic idea is to offer loosely coupled integration with different sensors, where sensor signals are received as ROS messages.

odom -> **base_footprint** transform is not static because the robot moves around the world. The base_footprint coordinate frame will constantly change as the wheels turn. This non-static transform is provided by robot_pose_ekf package.

b. AMCL:

The ROS Navigation Stack requires the use of AMCL (Adaptive Monte Carlo Localization), a probabilistic localization system for a robot. AMCL is used to track the pose of a robot against a known map. It takes as input a map, LIDAR scans, and transform messages, and outputs an estimated pose.

2.2.2.4. Navigation Stack Block

move_base is a package that contains the local and global planners and is responsible for linking them to achieve the nav goal.

a. Global Planner

- The global planner builds a map of the environment. It gathers all the information ever received and then plans a path that reaches to the goal. This happens at a much lower frequency than in the local planner and much more costly on the computational side.

b. Local Planner

- The local planner works only with the information it currently gets from the sensors and plans a path that is around a meter long. When the next set of information comes in it plans a new piece of the path. The obstacle avoidance works well with the local planner by building a histogram where the free and blocked cells are marked. The local planner is responsible for creating a trajectory rollout over the global trajectory that is able to return to the original trajectory with fewer costs.

c. Global Costmap

- The global costmap is everything the robot knows from previous visits and stored knowledge. Global planner uses the global costmap to generate a long-term plan.
- While the global costmap represents the whole environment (or a huge portion of it), the local costmap is, in general, a scrolling window that moves in the global costmap in relation to the robot's current position.

d. Local Costmap

- In the local costmap is everything that can be known from the current position with the data from sensors. E.g. walking people and other moving objects, as well as every wall etc. that can be seen. Local planner uses the local costmap to generate a short-term plan.

2.2.2.5. Base Controller System Block:

This block contains methods for driving ServeBot. TMUX node is programmed to select the control source. ServeBot can be manually controlled by Laptop using either keyboard or Rviz. On the other hand, ServeBot can work automatically by Navigation Stack Block and send_goals node. Send_goals node helps to set a goal location for robot with specific pre-defined coordinate (customer table, charge zone, cook zone).

2.2.2.6. Visualization Block:

a. RViz

- rviz is a 3d visualization tool for ROS applications. It provides a view of the robot model, captures sensor information from robot sensors, and replay captured data. It can display data from cameras, lasers, from 3D and 2D devices including pictures and point clouds. In short, It helps to visualize what the robot is seeing and doing.

2.2.2.7. Output Block:

Output informations to customer's side which include visual and sound notifications:

a. OLED

- Position of hardware: Below the top frame of ServeBot.
- Function: Display information of the table number corresponding to the food tray and basic interactions with customers.

b. Speaker

- Position of hardware: Below the top frame of ServeBot.
- Function: Make sound for drawing the attention of customers. On the other hand, it can play a warning sound for safety issues.

2.2.2.8. TF Transform Nodes:

ROS Transformer (as known as "tf") is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

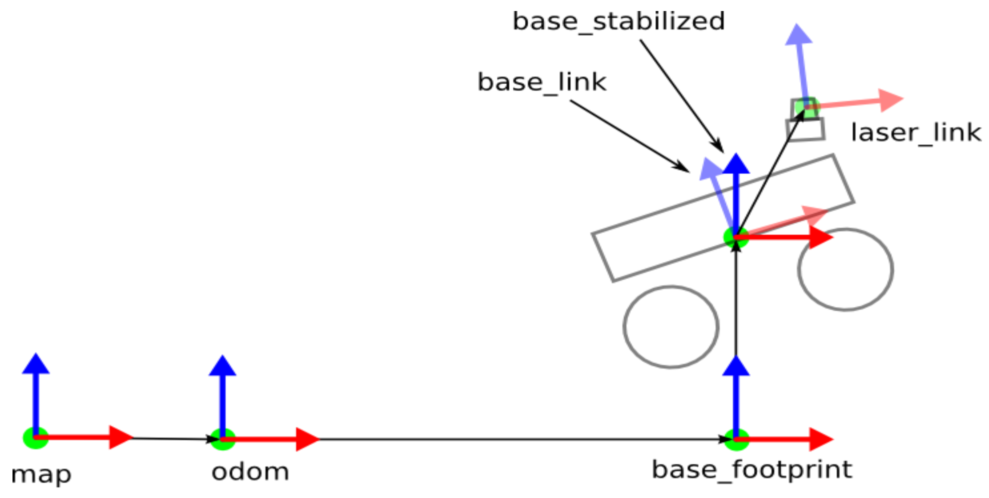


Figure 2.2.2.2 Multiple coordinate frames of a typical robot

- map frame has its origin at some arbitrarily chosen point in the world. This coordinate frame is fixed in the world.

- odom frame has its origin at the point where the robot is initialized. This coordinate frame is fixed in the world.

- base_footprint has its origin directly under the center of the robot. It is the 2D pose of the robot. This coordinate frame moves as the robot moves.

- base_link has its origin directly at the pivot point or center of the robot. This coordinate frame moves as the robot moves.

- **laser_link** has its origin at the center of the laser sensor (i.e. LIDAR).

2.2.3 Fundamentals of robot's operation:

2.2.3.1. How the robot moves:

The robot constructs a two-dimensional geometric representation of its environment using the laser scanner, this belongs to building-map state. In the moving process, it utilizes a combination of laser scan data, odometry information supplied through the wheel encoders with data from imu sensor to determine its current location on the map. It creates a point cloud in which each point refers to a possible location depending on the data supplied. As the robot advances, it eliminates potential places, reducing the number of points in the cloud. Its number of belief states quickly converges on its real location in this way. As a result, the robot uses probabilistic inference to locate itself. Once the robot has successfully localized, it may be given destination coordinates and a global planner can be used to create a course to those locations. Internally, the environment is represented as a two-dimensional lossless picture. Unexplored space, known clean space, obstructions, the inflating radius, and the robot itself are all present in the environment. The inflating radius is a protective barrier that extends outward from all obstructions. The robot sees the inflating radius as a barrier and is unable to get around it. The inflating radius is the same as the robot's radius, ensuring that the robot always constructs pathways with adequate room to avoid obstacles. For the sake of simplicity, it is implemented as a buffer around obstacles rather than on the robot itself, and it accomplishes the same result. The global planner in this paradigm generates a path to the objective using Dijkstra's algorithm or A* Algorithm. The local planner converts the global plan into velocity directives for the robot's motors once it has been created. It accomplishes this by defining a value function around the robot, sampling and simulating trajectories within that space, scoring each simulated trajectory based on its expected outcome, sending the highest-scoring trajectory to the robot as a velocity command, and repeating until the goal is reached. The reason for this is that the local planner was designed to be very general, allowing it to be used by robots with irregular footprints, Ackerman steering geometry, appendages, and other configurations that would not work with an algorithm designed for a robot with a simple shape, differential drive system, and no appendages. It is extremely difficult for the robot to reach its goal coordinates precisely because there is always a little degree of inaccuracy in the robot's physical motions. As a result, the robot may oscillate about the target in an attempt to reach the precise place.

2.2.3.2. Mapping:

The map building problem for an unknown environment with use of onboard sensors while solving the localization problem at the same time is known as Simultaneous Localization and Mapping (SLAM).

The main sensor for indoor robot navigation and SLAM is usually lidar. 2D lidar SLAM systems are currently presented in different packages like GMapping and Hector SLAM.

- **Hector Slam:**

Hector SLAM1 incorporates with 2D LIDAR sensor to generate a map from the laser scan. In contrast to other SLAM techniques (e.g. Gmapping), Hector SLAM does not require any auxiliary odometry sensor (e.g. wheel encoders) which directly measures the travel distance of a land-based robot, but only relies on the information from the laser scan matching approaches. Therefore, the Hector SLAM is more suitable for aerial vehicles. The Hector SLAM takes advantage of the low distance measurement noise and high sampling rates of LIDAR for a fast scan-matching method. Another advantage of the Hector SLAM is its capability to generate multi-resolution grid maps to avoid singularity during scan matching. A map can be generated by the Hector SLAM according to the endpoints of the laser beams hit onto the walls. Then, the transformation of the current scan is determined by the Gauss-Newton approach, which finds the best alignment of the current scan to the map generated previously.

- **GMapping:**

Gmapping is a laser-based SLAM algorithm, which uses a Rao-Blackwellized Particle Filter SLAM approach, which is a technique for model-based estimation. Each particle in the PF can be considered as a candidate solution to the problem and the set of particles together will approximate the true probability distribution. In general, the particle filter family of the algorithm requires high sampling particles to obtain accurate results, therefore it might have relatively increased computational complexity. Also, the depletion problem associated with this method decreases the algorithm accuracy. This arises when the elimination of a large number of particles from a sample set during the resampling step. This approach takes into account not only the movement of the mobile robot but also the most recent sensor observation with odometry information; therefore decreasing the uncertainty for the robots pose in the particle filter’s prediction step. As a result, the number of particles required is significantly reduced since the uncertainty is lower, due to the quality of the laser scan matching process.

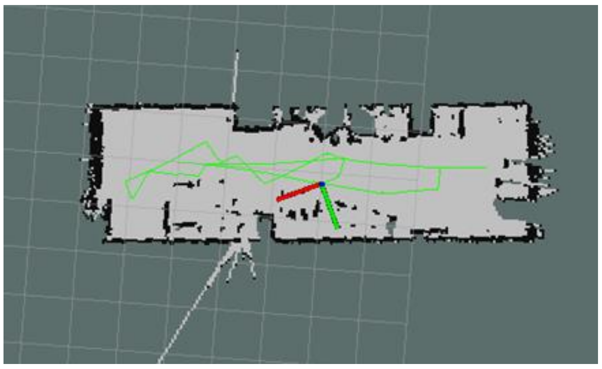


Figure 2.2.3.1Hector_Slam Map Result

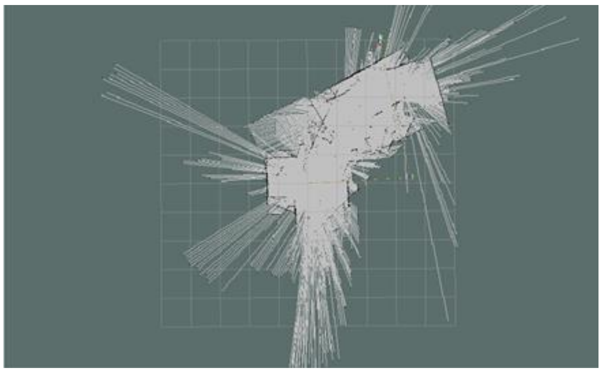


Figure 2.2.3.2Gmapping Map Result

Mapping Result comparison of Hector slam vs. Gmapping

Practical result of mapping via Hector slam method gives a much better look over Gmapping so this method is chosen for the map building step.

2.2.3.3. Localization:

For localization task of Servebot, both data from encoder and IMU sensor will be used because using encoder alone will lead to larger error accumulated the more robot moves. These two kinds of data will be fused together by robot_pose_ekf package. The fused data will be published through /robot_pose_ekf/odom_combined topic but this topic won’t be subscribed by any node. The main task of the robot_pose_ekf is creating the transform between odom -> base_footprint as the robot moves. This transform then will be used by AMCL package, which takes in the data from transform message, LiDAR scan and map and outputs estimated pose.

The method Adaptive Monte Carlo Localization uses to localize the robot is call Adaptive Particle Filter Localization. This method predicts the location and orientation of a robot as it travels and senses the environment using a map of the environment. The Particle Filter Localization algorithm represents the distribution of potential states using a particle filter, with each particle representing a conceivable state, i.e., a guess as to where the robot is. The procedure usually starts with a uniform random distribution of particles over the configuration space, implying that the robot has no idea where it is and is equally likely to be anywhere. When the robot moves, the particles shift in order to predict the robot's new condition following the movement. Whenever the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the actual sensed data correlate with the predicted state. Ultimately, the particles should converge towards the actual position of the robot.

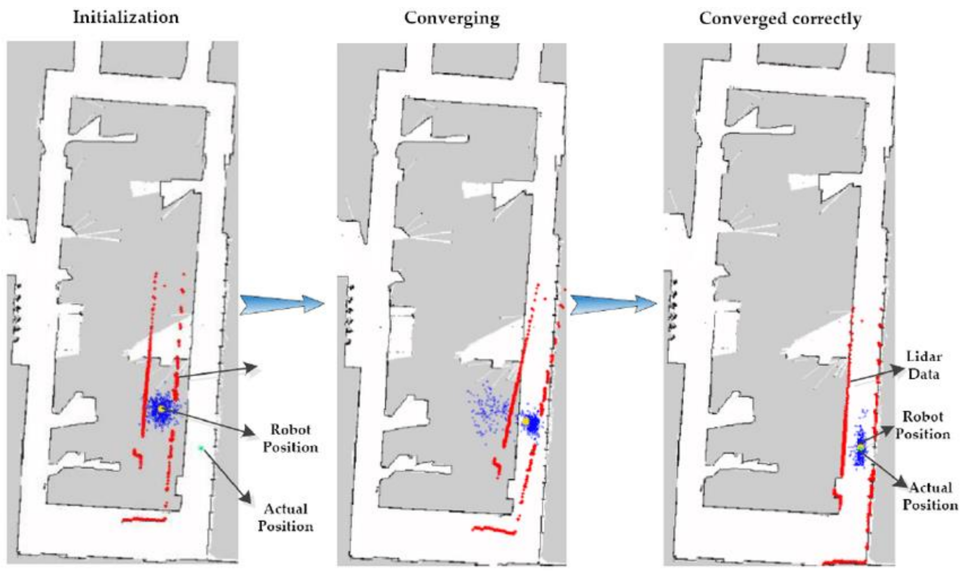


Figure 2.2.3.3Particle Filter Localization algorithm

A key problem with particle filter is maintaining the random distribution of particles throughout the state space, which goes out of hand if the problem is high dimensional. Therefore, Adaptive Particle Filter, which converges much faster and is computationally much more efficient than a basic Particle Filter, is used. The key idea is to bound the error introduced by the sample-based representation of the particle filter. To derive this bound, it is assumed that the true posterior is given by a discrete, piece-wise constant distribution such as a discrete density tree or a multidimensional histogram. For such a representation we can determine the number of samples so that the distance between the maximum likelihood estimate (MLE) based on the samples and the true posterior does not exceed a pre-specified threshold.

2.2.3.4. Navigation Algorithm:

The navigation algorithms are divided into two kinds of control: global path planning and local motion control.

Global path planning considers owning a priori model or a map of the environment, on which the robot wants to move, using this information calculates the shortest path that allows the motion from a start position to the goal.

Whereas local motion is more related to the real-time motion of the robot inside in unknown terrain, where monitoring the environment with the sensors, it can distinguish which and where are the obstacles and generate a motion that will avoid the collision.

A complete robot navigation system should integrate the local and global navigation systems: the global system pre-plan a global path and incrementally search the best new paths when discrepancy with the map occurs; instead, the local system uses onboard sensors to define a path when the information of the map is not yet available, and detect and avoid unpredictable obstacles. Using the information connected to its geometric points that are matched with the map, the mobile robot may conduct an ideal path from a beginning region to an arriving place. If an obstruction blocks the planned path, the local navigation algorithm is responsible for avoiding a collision with the obstacle, such as by allowing it to go around the perimeter until the obstacle is overcome, or by planning another optimum global path to reach the destination ahead of time. The global route planner, on the other hand, generates an appropriate path based on an environment map, while the Obstacle Avoidance algorithm determines a suitable motion direction based on incoming sensor data (real-time events).

a) Global path planning:

global_planner package provides an implementation of a fast, interpolated global planner for navigation. This class adheres to the nav_core::BaseGlobalPlanner interface specified in the nav_core package. Two options for navigation algorithm are provided by the package: Dijkstra's algorithm (default) and A* algorithm. For Servebot, Dijkstra's algorithm is used for global planner and the theory of it is introduced as below.

- **Dijkstra's algorithm:**

  Time Complexity: $O(n^2)$

Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

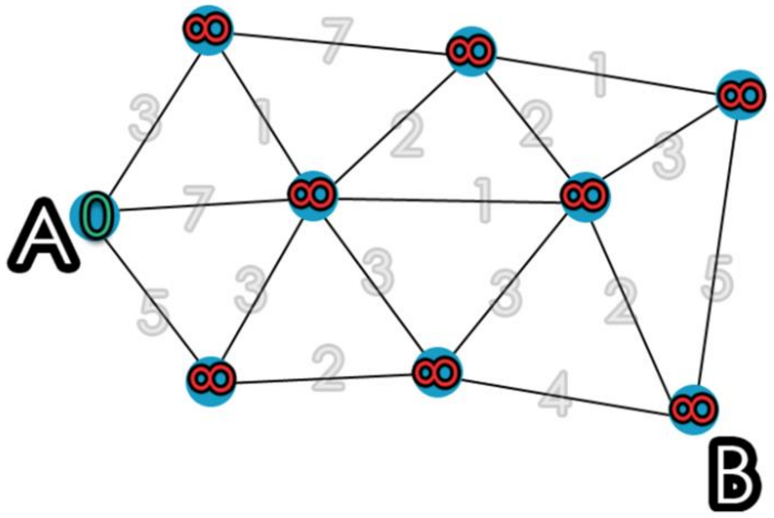


Figure 2.2.3.4 Dijkstra's algorithm

The graph has the following:

- vertices, or nodes, denoted in the algorithm by *v* or *u*;
- weighted edges that connect two nodes: (*u*,*v*) denotes an edge, and *w*(*u*,*v*) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- dist: an array of distances from the source node ss to each node in the graph, initialized the following way: *dist*(*s*) = 0; and for all other nodes *v*, dist (*v*) = ∞. This is done at the beginning because as the algorithm proceeds, the *dist* from the source to each node *v* in the graph will be recalculated and finalized when the shortest distance to *v* is found.

- *Q*: a queue of all nodes in the graph. At the end of the algorithm's progress, *Q* will be empty.
- *S*: an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, *S* will contain all the nodes of the graph.

The algorithm proceeds as follows:

1. While *Q* is not empty, pop the node *v*, that is not already in *S*, from *Q* with the smallest *dist* (*v*). In the first run, source node *s* will be chosen because *dist*(*s*) was initialized to 0. In the next run, the next node with the smallest *dist* value is chosen.
2. Add node *v* to *S*, to indicate that *v* has been visited
3. Update *dist* values of adjacent nodes of the current node *v* as follows: for each new adjacent node *u*.

- if *dist* (*v*) + *weight*(*u*,*v*) < *dist* (*u*), there is a new minimal distance found for *u*, so update *dist* (*u*) to the new minimal distance value;
- otherwise, no updates are made to *dist*(*u*).

The algorithm has visited all nodes in the graph and found the smallest distance to each node. *dist* now contains the shortest path tree from source *s*.

- **A* Algorithm:**

  Time complexity: $O\ (n\ log\ n)$

  The A* algorithm is based on heuristics for navigating the search.

  For the A* algorithm, the evaluation function has a specific form:

$$f(n) = g(n) + h(n)$$

$g(n)$ – shortest cost path from start node to node n

$h(n)$ – represents heuristic approximation of the value of node n

The algorithm proceeds as follows:

1. There are two lists:

- Closed list: keep the visited nodes whose neighbors are all also visited
- Open list: keep the visited nodes whose neighbors are not necessarily all visited

At the beginning, the open list only contains the start node. The closed state list is empty.

2. Choose node n with the best value of f(n). If node n is also final node, process is done. If not, go over its direct neighbors.
3. For each neighbor m of the node n, check whether it's in one of the two lists. If not, put it in the open list. We mark n as the parent of m. Then, calculate g(m) and f(m). However, if the neighbor is in one of the two lists, check whether the path from the start node to node m over state n is shorter than the current existing path to m. If this is true, mark n as the parent of m and update g(m) and f(m). If the node m was in the closed list before, place it in the open list instead.
4. Finally, put the current node in the closed state list.
5. As long as there are elements in the open states list, we repeat the steps 2, 3, and 4.
6. If can’t get to the final state, but open list is empty, the path to the final state doesn't exist

- Comparison of Dijkstra’s algorithm and A* Algorithm:

These 2 picture below show how different Dijkstra’s algorithm and A* Algorithm plan a path with the same initial point and goal point. For a complex path, while Dijkstra’s algorithm always come with the path in middle of the way that gives the robot a lot of empty space to move, A* Algorithm tries to make the path along the wall so the inflation layer will be taken into account to make the robot moves smoothly without hitting the wall. For simple path, I.e the path from intial point to goal point is a straight line without obstacle, the path of 2 algorithm are the same.

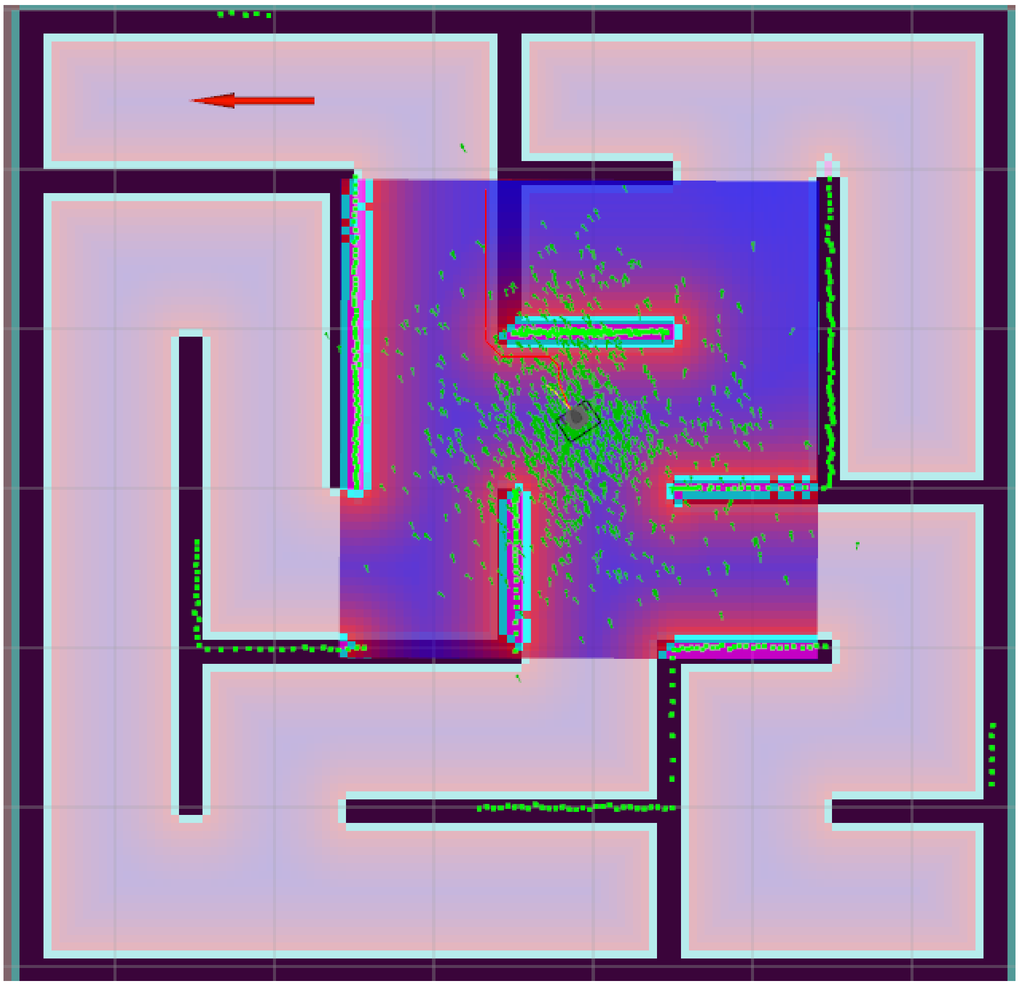


Figure 2.2.3.5 *Path created by A* Algorithm*

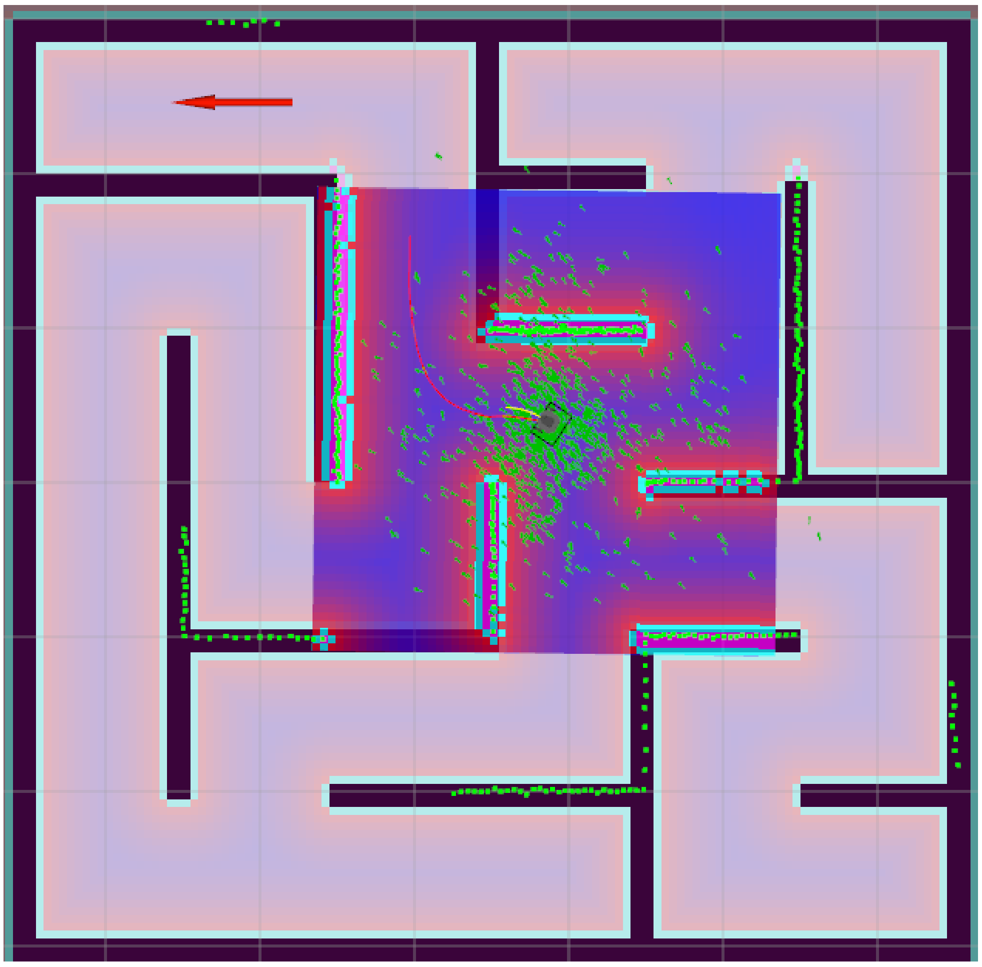


Figure 2.2.3.6 *Path created by Dijkstra Algorithm*

Many test case of suddenly change goal point were also tried. The time that both algoritm took to create a new path plan was about 0.4 -0.5s for this maze map, and there is nearly no differences in how long each algorithm took to plan path.

In theory, Dijkstra's always result more optimal distance path in the enviroment with many obstacle. However, the disadvantages of Dijkstra’s algorithm are performs blind searches. Thus, a lot of time will be wasted to search for useless resource. The A* algorithm is much faster than Dijkstra's algorithm. However, the efficiency of the A* algorithm is highly dependent on its evaluation function, and with the wrong function, the results could be even worse than Dijkstra. The distance heuristic function (eg: Euclidean, Manhattan, Octile,…) generates optimal paths in real time. Nevertheless, as the size and the number of obstacles increases, A* not only spends more time on searching but also requires more memory for the nodes as it needs more nodes to find a path.

In our tests, Dijkstra's always give a clearer path so robot is easier to move, the time each algorithm takes are nearly the same in all of the cases. That’s our main reason to choose Dijkstra's for the current project.

b) Local path planning:

The base_local_planner package provides a controller that drives a mobile base in the plane. This controller serves to connect the path planner to the robot. Using a map, the planner creates a kinematic trajectory for the robot to get from a start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx, dy, dtheta velocities to send to the robot.

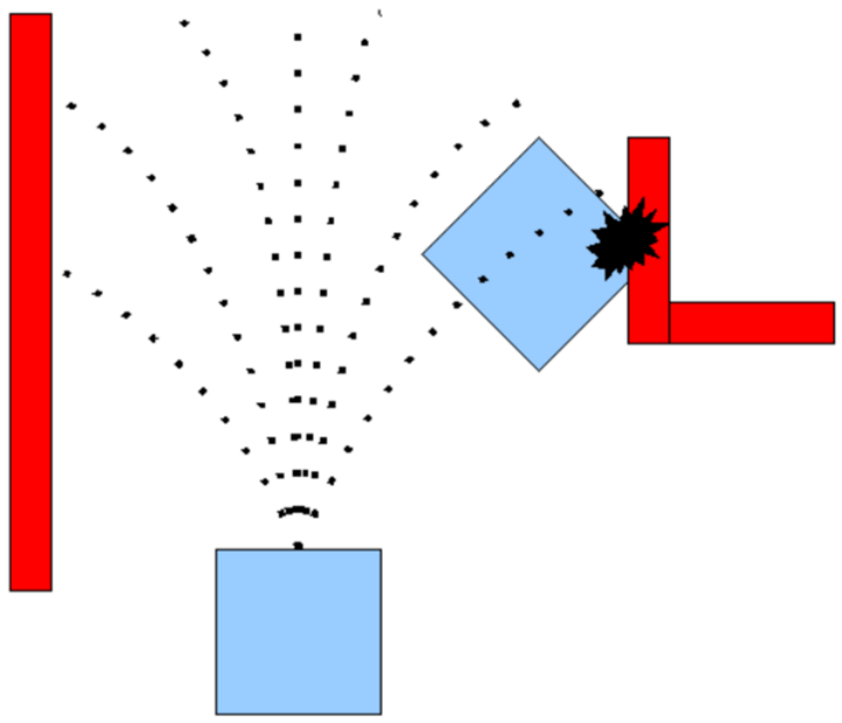


Figure 2.2.3.7Local path planner of robot

There are two options for choosing local path planning algorithm provided by base_local_planner: Trajectory Rollout and Dynamic Window Approach (DWA).

The basic idea of both the Trajectory Rollout and Dynamic Window Approach (DWA) algorithms is as follows:

1. Discretely sample in the robot's control space (dx,dy,dtheta)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

2.2.3.5. TF Graph:

The coordinate frames of Servebot are placed in positions as described in this picture.

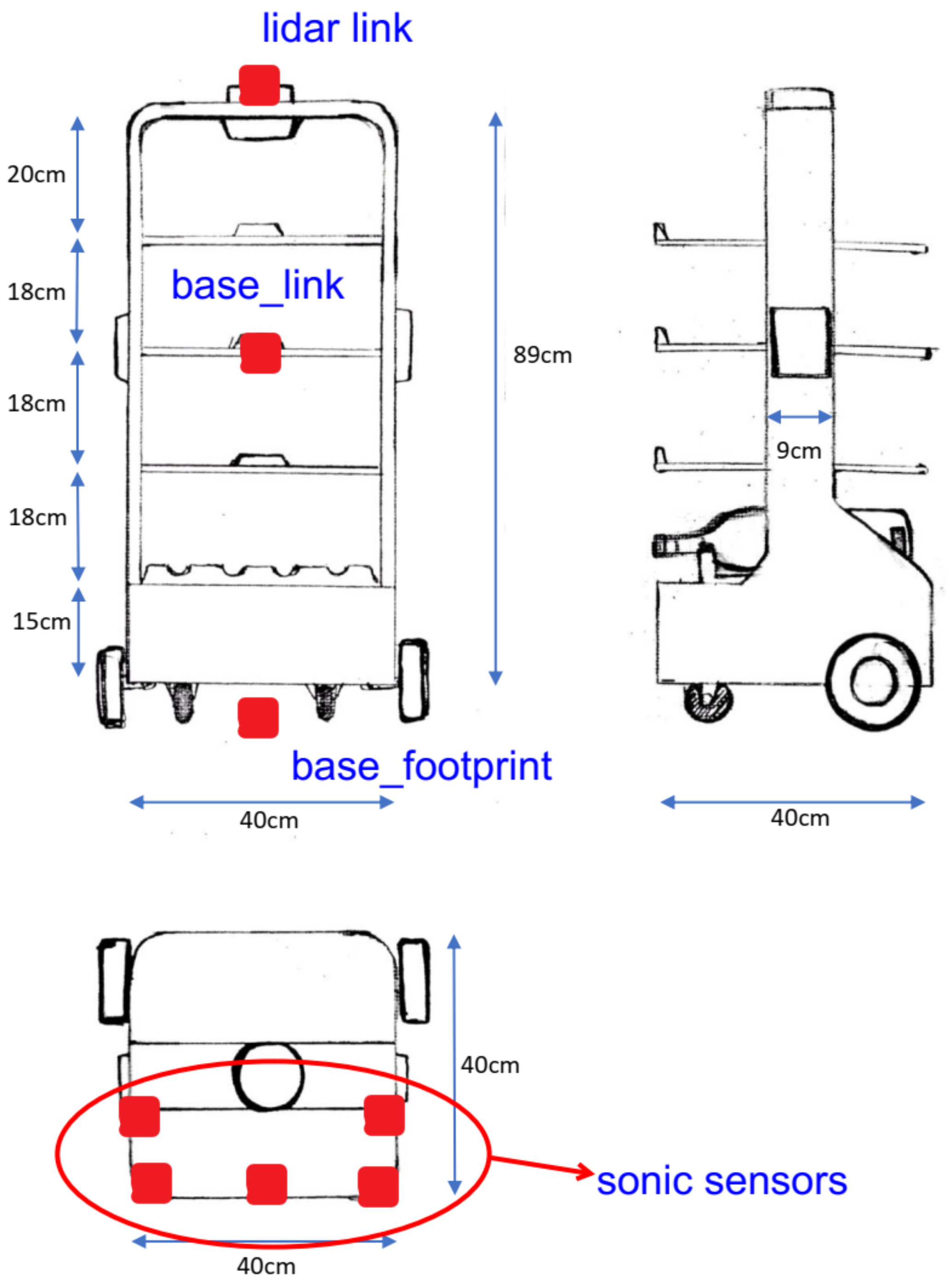


Figure 2.2.3.8 Coordinate frames of Servebot

From the picture, the tf tree of the whole Serbot system is set up as below:

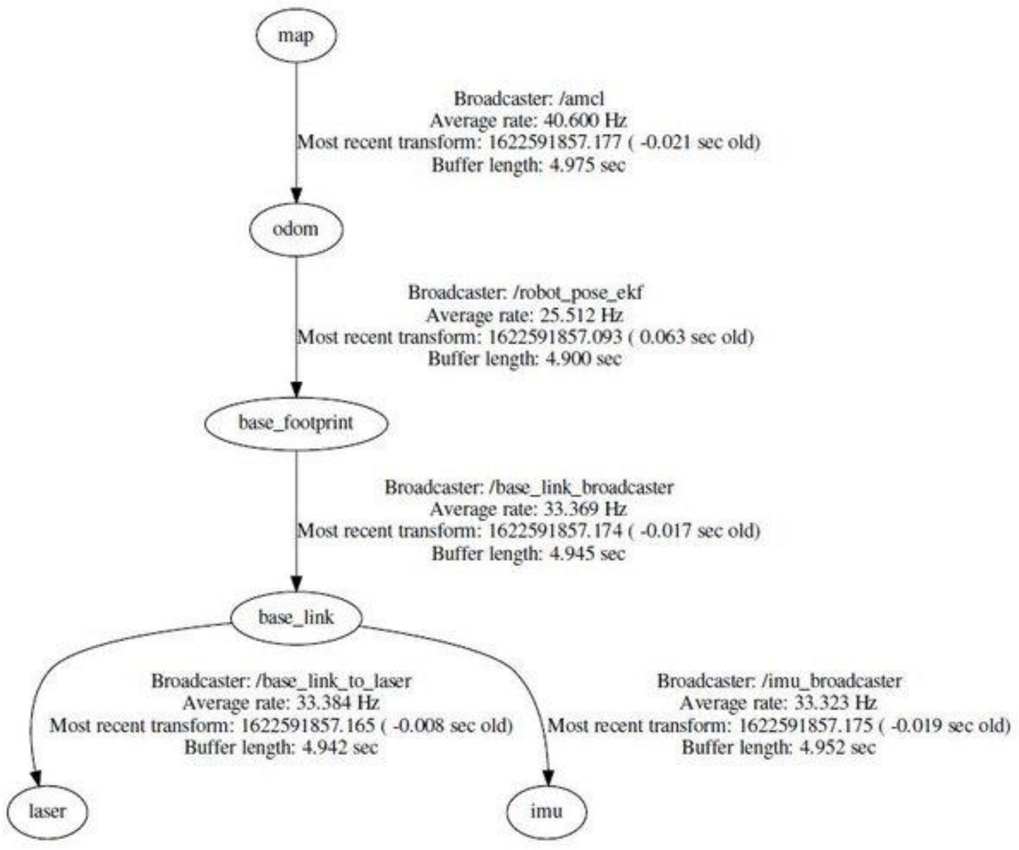

map
Broadcaster: /amcl
Average rate: 40.600 Hz
Most recent transform: 1622591857.177 (-0.021 sec old)
Buffer length: 4.975 sec
odom
Broadcaster: /robot_pose_ekf
Average rate: 25.512 Hz
Most recent transform: 1622591857.093 (0.063 sec old)
Buffer length: 4.900 sec
base_footprint
Broadcaster: /base_link_broadcaster
Average rate: 33.369 Hz
Most recent transform: 1622591857.174 (-0.017 sec old)
Buffer length: 4.945 sec
base_link
Broadcaster: /base_link_to_laser
Average rate: 33.384 Hz
Most recent transform: 1622591857.165 (-0.008 sec old)
Buffer length: 4.942 sec
Broadcaster: /imu_broadcaster
Average rate: 33.323 Hz
Most recent transform: 1622591857.175 (-0.019 sec old)
Buffer length: 4.952 sec
laser
imu

Figure 2.2.3.9 Complete TF graph of Servebot

For coordinate frames that don't change relative to each other through time (e.g. laser_link to base_link stays static because the laser is attached to the robot), we use the Static Transform Publisher.

For coordinate frames that do change relative to each other through time (e.g. map to base_link), we use tf broadcasters and listeners.

a) Static Transform:

+ **map -> odom:** transform tells us the position and orientation of the starting point of the robot (i.e. odom coordinate frame) inside the map's coordinate frame.

+ **base_footprint -> base_link:** The coordinate frame of robot "footprint" will move as the robot moves. In this case above, we assume that the origin of the base_link coordinate frame (i.e. center of the robot) is located 0.41 meters above its footprint.

+ base_link -> laser transform gives us the position and orientation of the laser inside the base_link's coordinate frame. The laser is located 0.38 meters above the center of the robot.

+ base_link -> imu transform gives us the position and orientation of the IMU mpu6050.

b) Non-static transform:

+ **odom -> base_footprint:** transform is not static because the robot moves around the world. The base_footprint coordinate frame will constantly change as the wheels turn. This non-static transform is provided by robot_pose_ekf package.

2.2.3.6. Motor speed controller

One of the most important parts of controlling a robot is the velocity. To control the robot position and path planning, a robot should be able to run at precise defined speed generated from the controller.

a) Read motor speed by rotary encoder

For the purpose of controlling DC motor, speed measuring is an essential task.

We can measure the speed of motor by reading number of ticks counted by the encoder over a sampling time T_s.

For the DC motor, the number of ticks at one channel when motor shaft finishes one cycle is 480 pulses.

The motor speed n (round/sec) is calculated by the equation:

$$n = \frac{Countedpulse}{480 \times T_s} \qquad \textit{Equation 13}$$

To convert to RPM, we multiply n by 60.

Flow chart for reading motor encoder:

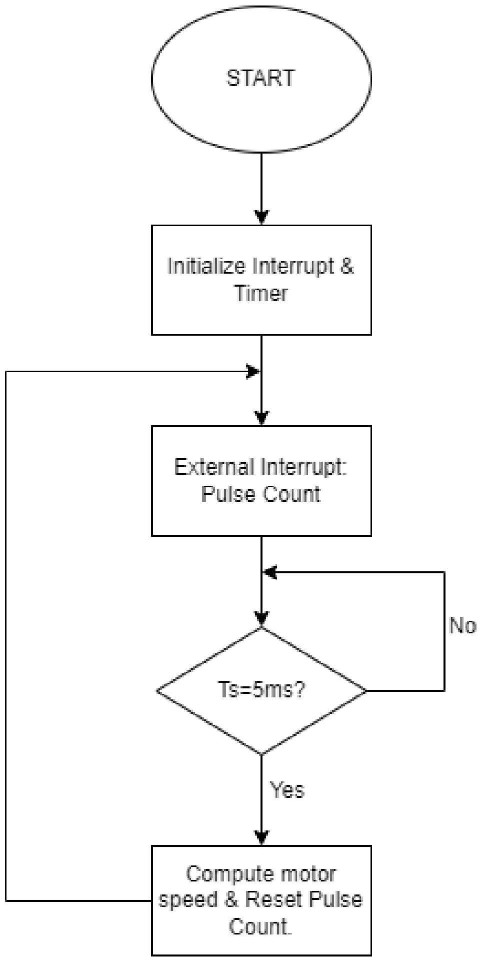

START
Initialize Interrupt & Timer
External Interrupt: Pulse Count
Ts=5ms?
No
Yes
Compute motor speed & Reset Pulse Count.

Figure 2.2.3.10 Reading motor encoder flow chart

Method of motor direction detection by encoder:

There exist 2 channel A and B with 90 degrees out of phase from each other. If the encoder is rotating clockwise the output A will be ahead of output B and vice versa for counterclockwise. By considering this, we can easily program our controller to read the rotation direction of the motor.

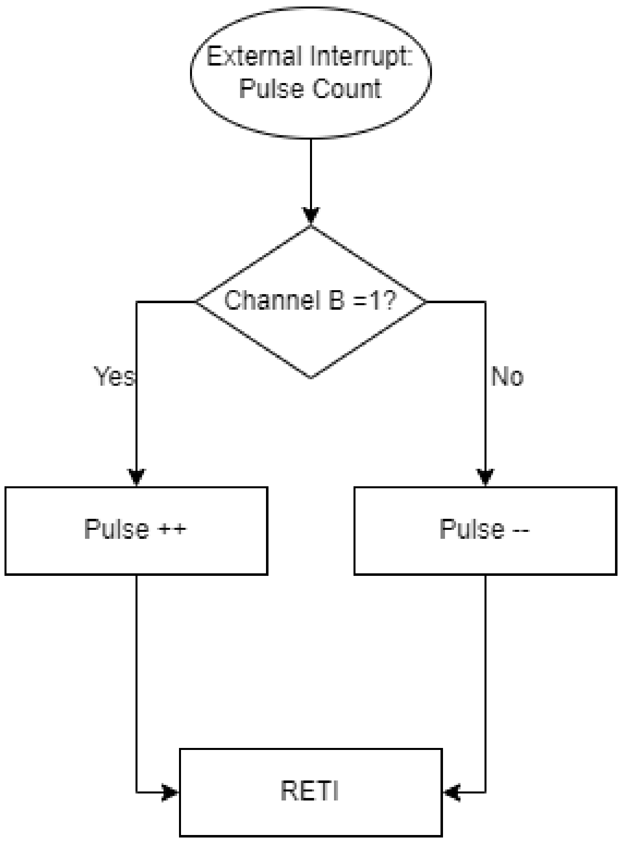


Figure 2.2.3.11 External Interrupt for checking motor speed and direction

There exists only one problem for measuring speed by motor encoder is the noise from motor when running at low speed can affect to computed result.

Therefore, we need a filter to remove the noise

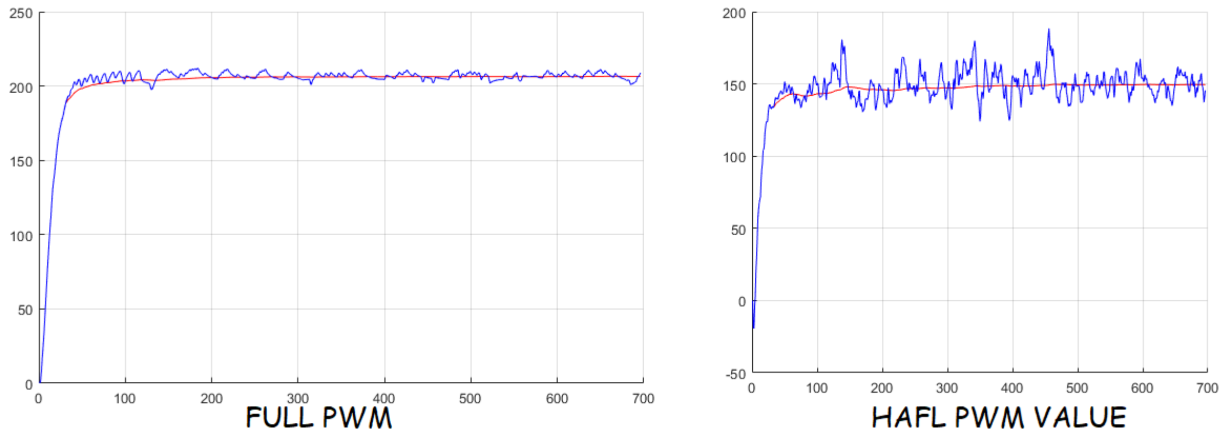


Figure 2.2.3.12 Measured speed value without filter

b) Kalman filter for speed measurement

About Kalman filter:

One of the biggest challenges of tracking and control systems is providing accurate and precise estimation of the hidden variables in the presence of uncertainty. The Kalman Filter is one of the most important and common estimation algorithms. The Kalman Filter produces estimates of hidden variables based on inaccurate and uncertain measurements. Also, the Kalman Filter provides a prediction of the future system state based on past estimations.

The filter inputs are:

+ Initialization: The initialization is performed by only once with two parameters:

- Initial system state ($\hat{x}_{1,0}$)
- Initial state uncertainty ($p_{1,0}$)

The initialization parameters can be provided by another system, another process (for instance, a search process in radar), or an educated guess based on experience or theoretical knowledge. Even if the initialization parameters are not precise, the Kalman filter will be able to converge close to the real value.

+ Measurement: performed for every filter cycle, provides two parameters:

- Measured system state (z_n)
- Measured uncertainty (r_n)

The filter outputs are:

- System state estimate ($\hat{x}_{n,n}$)
- Estimate uncertainty ($p_{n,n}$)

Kalman filter in one dimension for measuring constant velocity:

+ Update:

$$\hat{x}_{n,n} = \hat{x}_{n,n-1} + K_n\left(z_n - \hat{x}_{n,n-1}\right) \qquad \textit{Equation 14}$$

$$K_n = \frac{p_{n,n-1}}{p_{n,n-1} + r_n} \qquad \textit{Equation 15}$$

$$p_{n,n} = (1 - K_n)p_{n,n-1} \qquad \textit{Equation 16}$$

+ Predict:

$$\hat{x}_{n+1,n} = \hat{x}_{n,n} \qquad \textit{Equation 17}$$

$$p_{n+1,n} = p_{n,n} \qquad \textit{Equation 18}$$

Where: K_n is Kalman gain

z_n is the velocity measured at state n.

$\hat{x}_{n,n}$ is the estimated velocity at state n.

$p_{n,n}$ is the estimated uncertainty at state n.

r_n is measurement uncertainty.

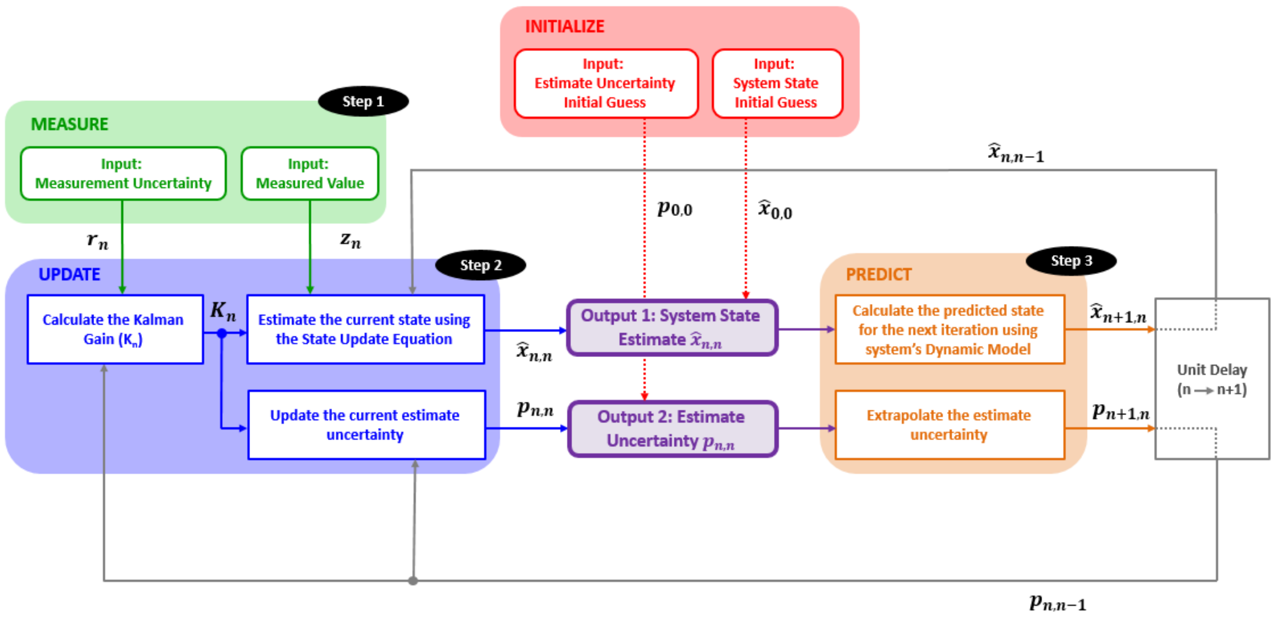

INITIALIZE
Input: Estimate Uncertainty Initial Guess
Input: System State Initial Guess
MEASURE
Step 1
Input: Measurement Uncertainty
Input: Measured Value
r_n
z_n
$p_{0,0}$
$\hat{x}_{0,0}$
$\hat{x}_{n,n-1}$
UPDATE
Step 2
Calculate the Kalman Gain (K_n)
K_n
Estimate the current state using the State Update Equation
Update the current estimate uncertainty
$\hat{x}_{n,n}$
$p_{n,n}$
Output 1: System State Estimate $\hat{x}_{n,n}$
Output 2: Estimate Uncertainty $p_{n,n}$
PREDICT
Step 3
Calculate the predicted state for the next iteration using system's Dynamic Model
Extrapolate the estimate uncertainty
$\hat{x}_{n+1,n}$
$p_{n+1,n}$
Unit Delay (n → n+1)
$p_{n,n-1}$

Figure 2.2.3.13A detailed description of Kalman Filter's block diagram

Setting parameter for filtering:

+ System state initial guess: $\hat{x}_{0,0}$ equals the first constant of speed value.

+ A human's estimation error (standard deviation) is set to be 100RPM because our guess is very imprecise. The Estimate Uncertainty of the initialization is the error variance: $\sigma^2 = p_{0,0} = 10000$.

+ Measurement uncertainty of encoder: $r_n = 273$ (equal the variance of encoder speed measurement error).

The simulation result on Matlab:

Blue line: Raw speed value

Red line: Filtered value

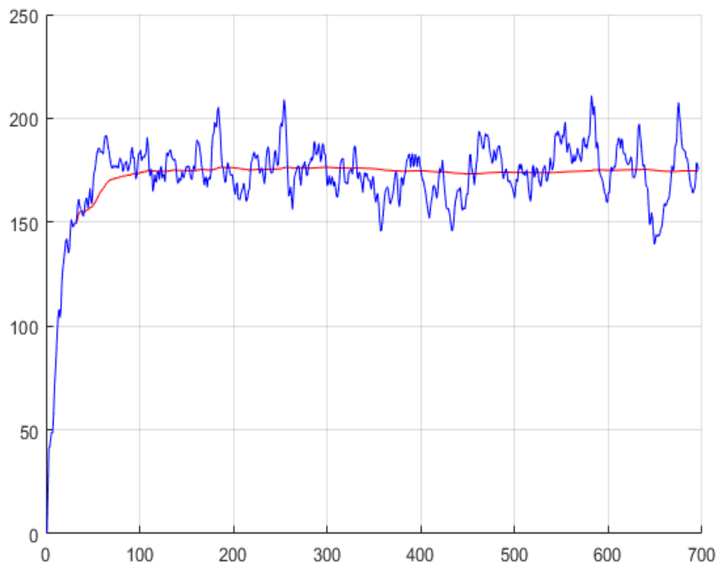


Figure 2.2.3.14 Raw speed vs filtered speed at 70% PWM

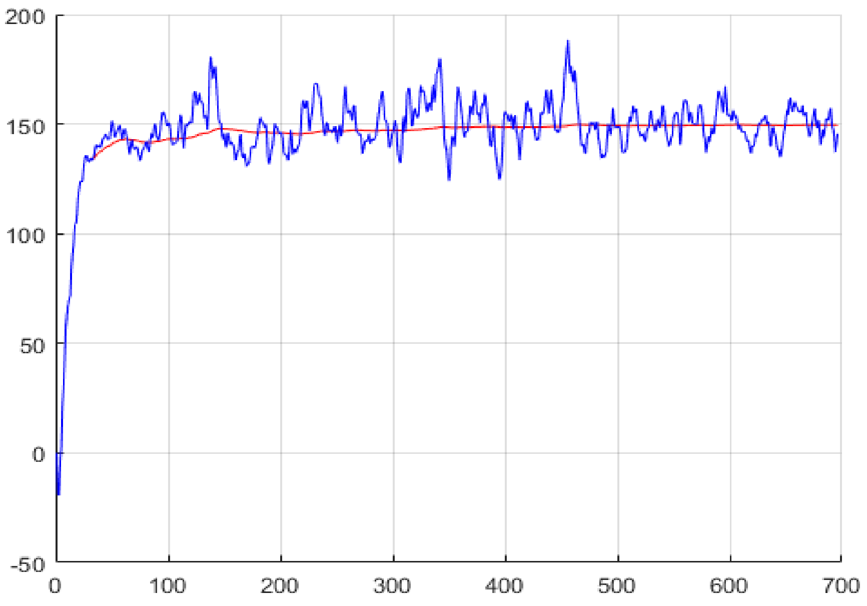


Figure 2.2.3.15 Raw speed vs filtered speed at 50% PWM

c) PID motor speed controller

In this project, we decided to use a simple PID controller to control the speed of two motors.

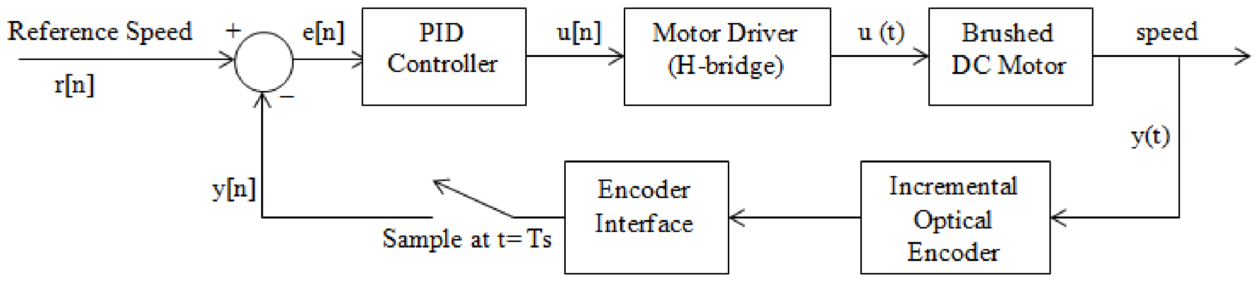


Figure 2.2.3.16 PID speed controller example

About PID controller:

PID controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. A PID controller continuously calculates an error e(t) as the difference between a desired setpoint (SP) and a measured process variable (PV) and applies a correction based on proportional, integral, and derivative terms (denoted P, I, and D respectively).

By measuring no load motor speed and using Matlab System Tdentification Toolbox, we can obtain the simple transfer function of the DC motor:

$$G(s) = \frac{18.02}{s + 1.156}$$

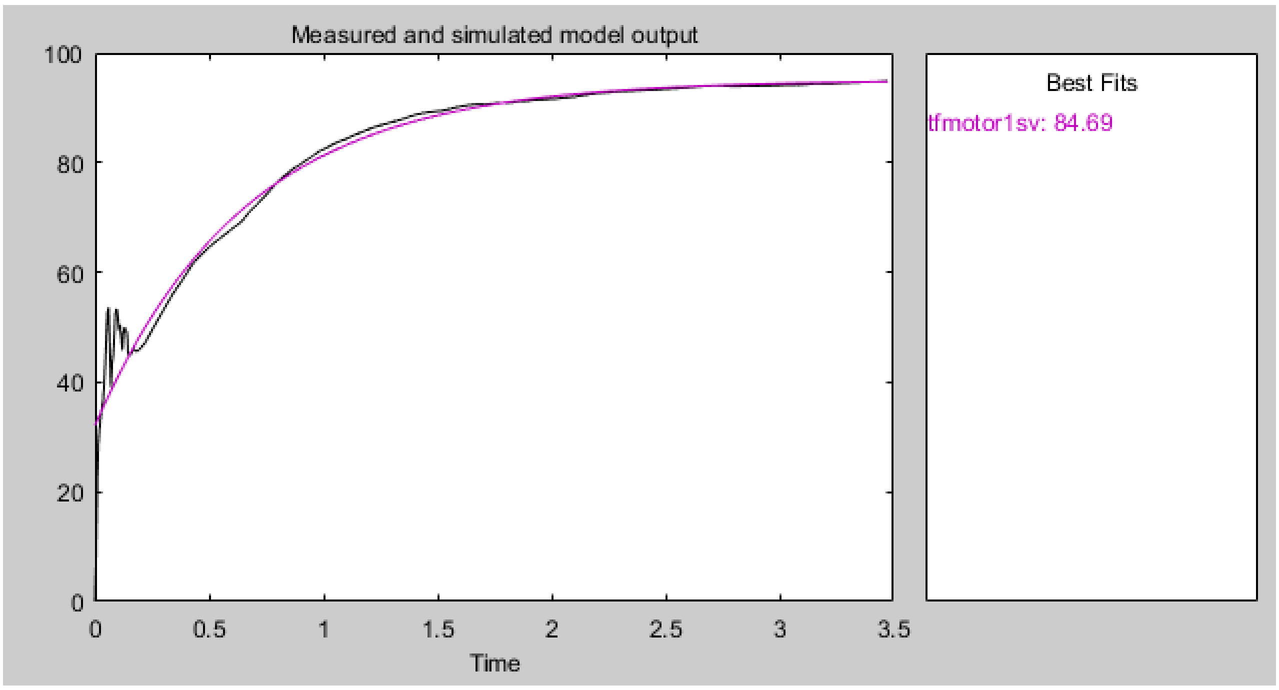


Figure 2.2.3.17 Measured and simulated model of DC motor speed

PID tuning method:

+ Ziegler-Nichols 2 method: This method starts by zeroing the integral and differential gains and then raising the proportional gain until the system is unstable. The value of k_p at the point of instability is called k_{th}; the period of oscillation is T_{th}. The method then backs off the proportional gain a predetermined amount and sets the integral and differential gains as a function of T_{th}.

Table 2.2.3.1 Settings for K_p, T_i and T_d according to the Ziegler-Nichols method

Controller type	K_P	T_i	T_d
P	$\frac{1}{2}k_{th}$		
PI	$0.45k_{th}$	$0.85T_{th}$	
PID	$0.6k_{th}$	$0.5T_{th}$	$0.12T_{th}$

By simulating the motor transfer function in Matlab, we obtain $k_{th} = 7$ and $T_{th} = 0.13s$.

From there we obtain the coefficients $\mathrm{K_p}, \mathrm{T_i}$ for PI motor controller $K_p = 3.15$ and $T_I = 0.11$ $(I = \frac{K_P}{T_I} = 28.65)$

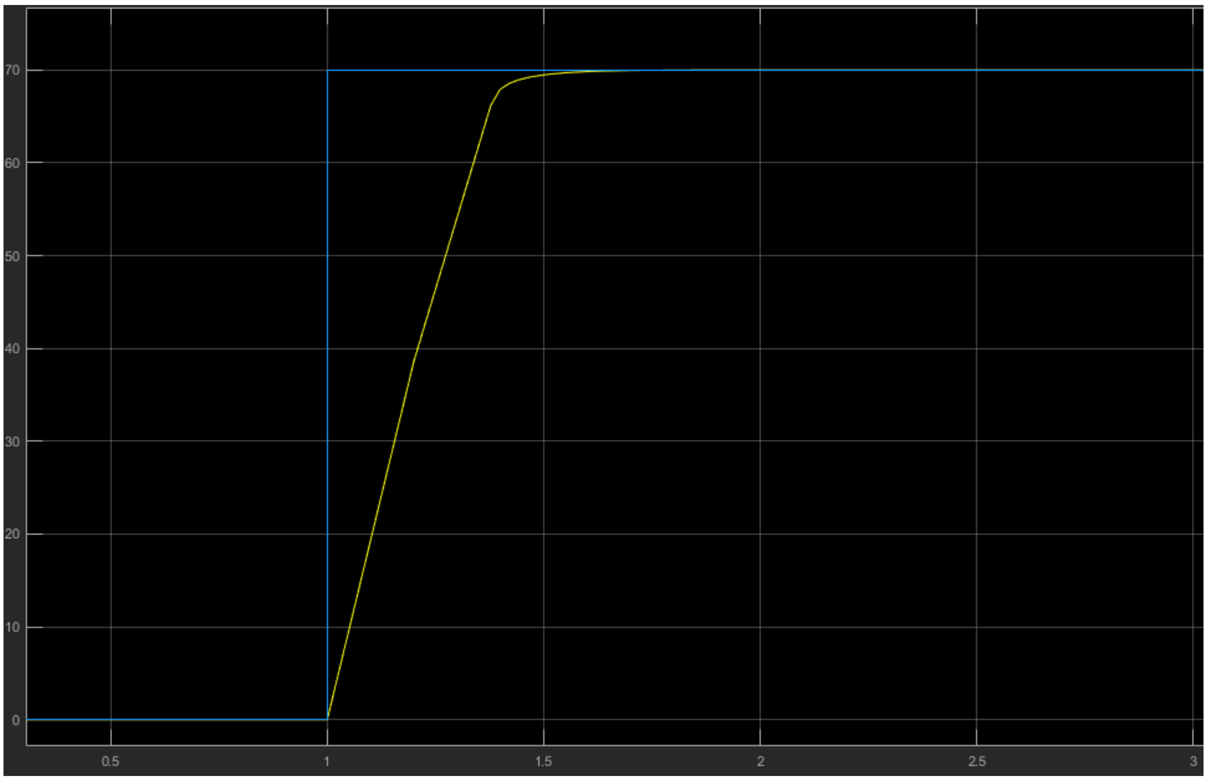


Figure 2.2.3.18 Simulation result with the setpoint of 70RPM

After obtaining the coefficient by Ziegler Nichols, the next step is to apply it in reality and using manual tuning method to modify the coefficient value.

+ Manual tuning method: First, we set K_i and K_d values to zero and increase the K_p until the output of the loop oscillates; then set $K_p$to approximately half that value for a "quarter amplitude decay"-type response. Then increase K_i until any offset is corrected in sufficient time for the process, but not until too great a value causes instability. Finally, increase K_d, if required, until the loop is acceptably quick to reach its reference after a load disturbance. Too much K_d causes excessive response and overshoot. A fast PID loop tuning usually overshoots slightly to reach the setpoint more quickly.

Finally, by manual tuning method, we can get the parameters for continuous PID controller controlling 2 motors speed.

+ For the motor 1: $K_{P1} = 3; K_{I1} = 3$

+ For the motor 2: $K_{P2} = 3; K_{I2} = 3.1; K_{D2} = 0.07$

Dicrete PID controller implementation:

The analysis for designing a digital implementation of a PID controller in a microcontroller (MCU) requires the standard form of the PID controller to be discretized.

We have the convert equation from continuous PID controller to discrete PID controller with control signal:

$$u(k) = \frac{\alpha e(k) + \beta e(k-1) + \gamma e(k-2) + \Delta * u(k-1)}{\Delta} \qquad \textit{Equation 19}$$

Where:

$\alpha = 2TK_P + K_I T^2 + 2K_D$

$\beta = K_I T^2 - 4K_D - 2TK_P$

$\gamma = 2K_D$

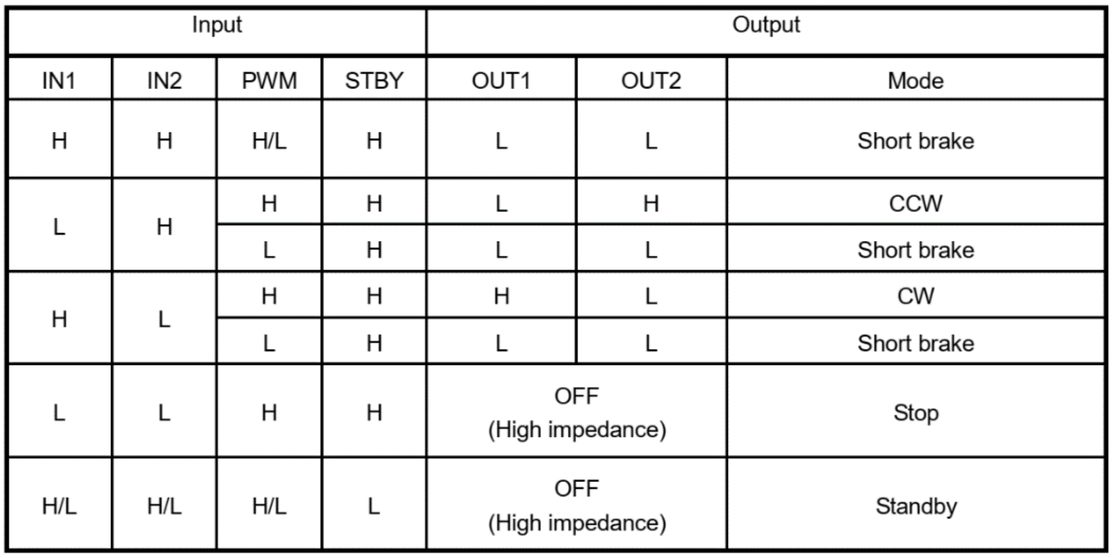
$\Delta = 2T$

From equation 19, we get the control signal for motor 1 and motor 2:

$$u_1(k) = \frac{0.03e(k) - 0.03e(k-1) + 0.01u(k-1)}{0.01}$$

$$u_2(k) = \frac{0.17e(k) - 0.3e(k-1) + 0.14e(k-2) + 0.01u(k-1)}{0.01}$$

Synchronizing two motors:



Input				Output		
IN1	IN2	PWM	STBY	OUT1	OUT2	Mode
H	H	H/L	H	L	L	Short brake
L	H	H	H	L	H	CCW
		L	H	L	L	Short brake
H	L	H	H	H	L	CW
		L	H	L	L	Short brake
L	L	H	H	OFF (High impedance)		Stop
H/L	H/L	H/L	L	OFF (High impedance)		Standby

Figure 2.2.3.19 TB6612FNG Control function with STANDBY pin

According to the table, the STBY (standby) pin is set to LOW to put OUT1 (output pin 1) and OUT2 (output pin 2) to High impedance mode. The STBY pin is set to HIGH to put OUT1 and OUT2 into operating mode. To synchronize two motors, we set STBY pin to LOW before setting PWM pulses to OUT1 and OUT2 then set STBY pin to HIGH to trigger the PWM pulses.

PID motor speed controller flow chart for one motor:

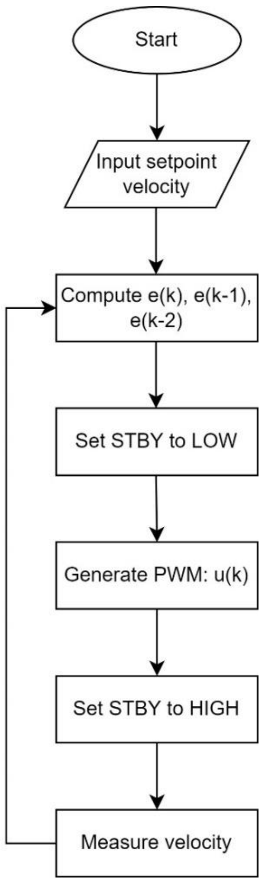


Figure 2.2.3.20 PID motor speed controller

2.3 SERVING ROBOT WORKING EVALUATION

2.3.1 Motor speed controller

a) Motor speed measurement

After being filtered by Kalman filter, the result is shown below:

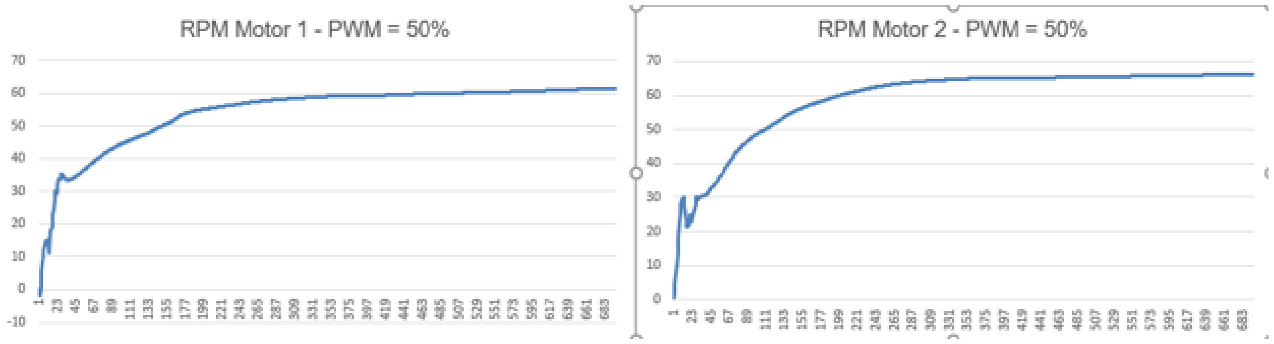


Figure 2.3.1.1 Motor speed measurement with Kalman filter in reality no load (PWM=50%)

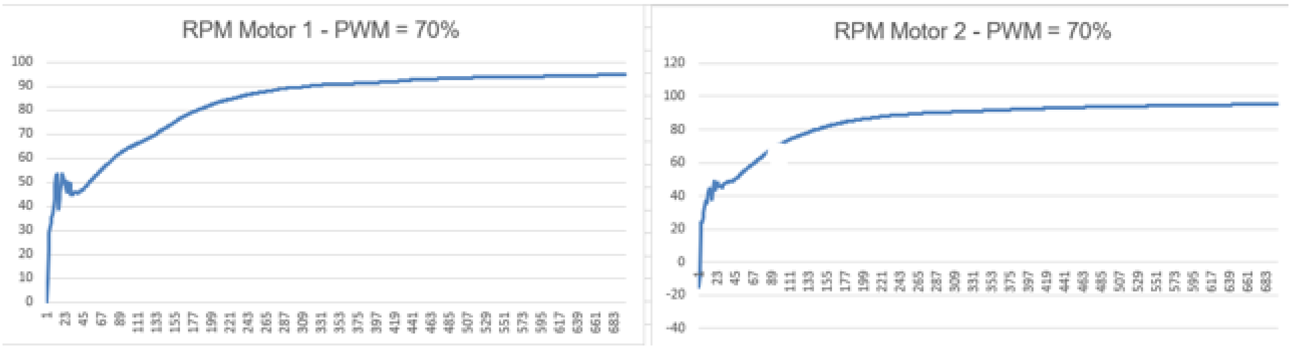


Figure 2.3.1.2 Motor speed measurement with Kalman filter in reality no load (PWM=70%)

b) PID controller for controlling motor speed

Result analysis:

+ Settling time: $121 \times 0.005 = 0.6\ (s) < 1\ (s)$

+ Small overshoot < 20%.

+ Steady state error nearly 0.

The control result is measured by setpoint of speed at 70 and 120RPM respectively.

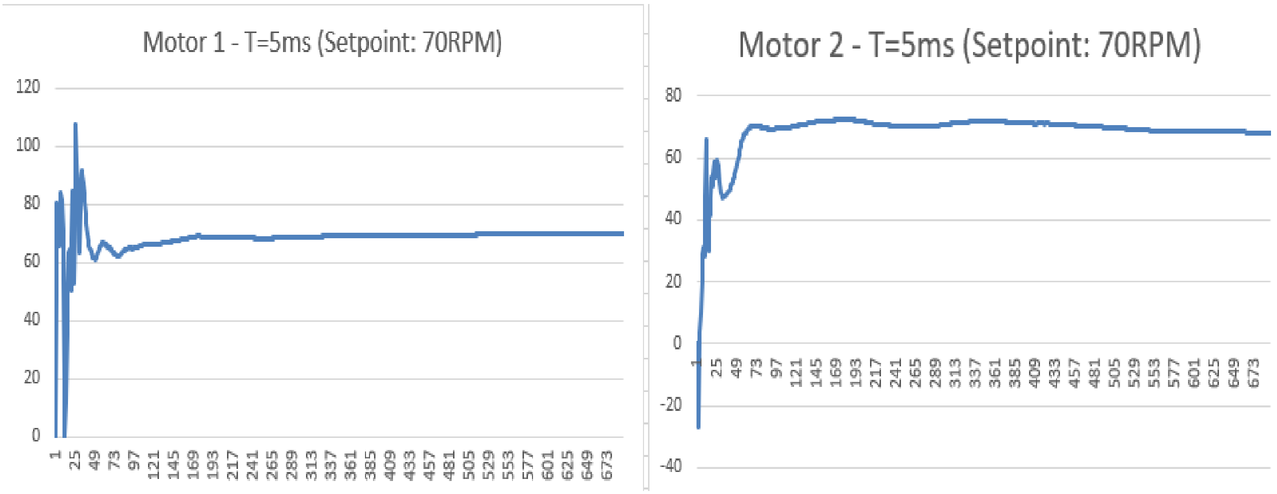


Figure 2.3.1.3 Measured velocity at setpoint 70 RPM

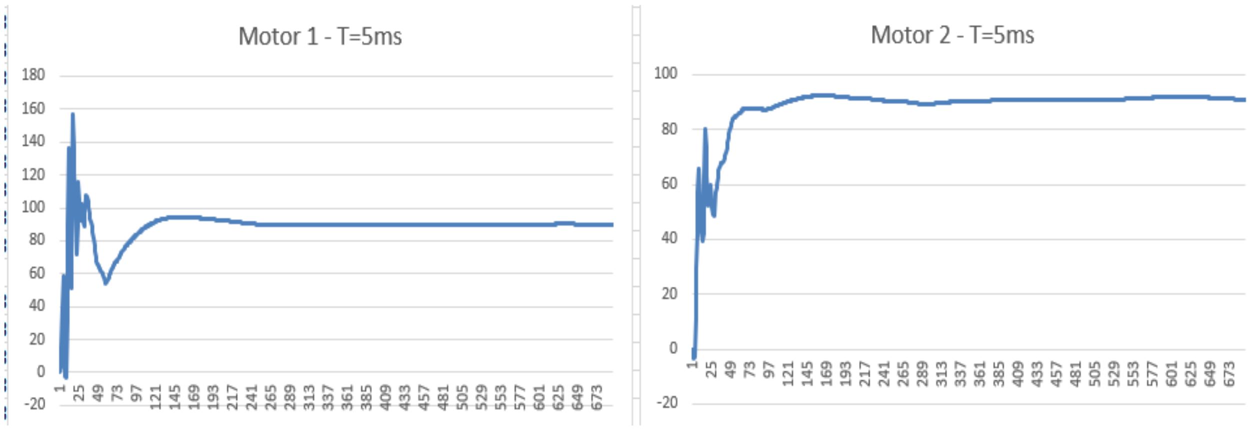
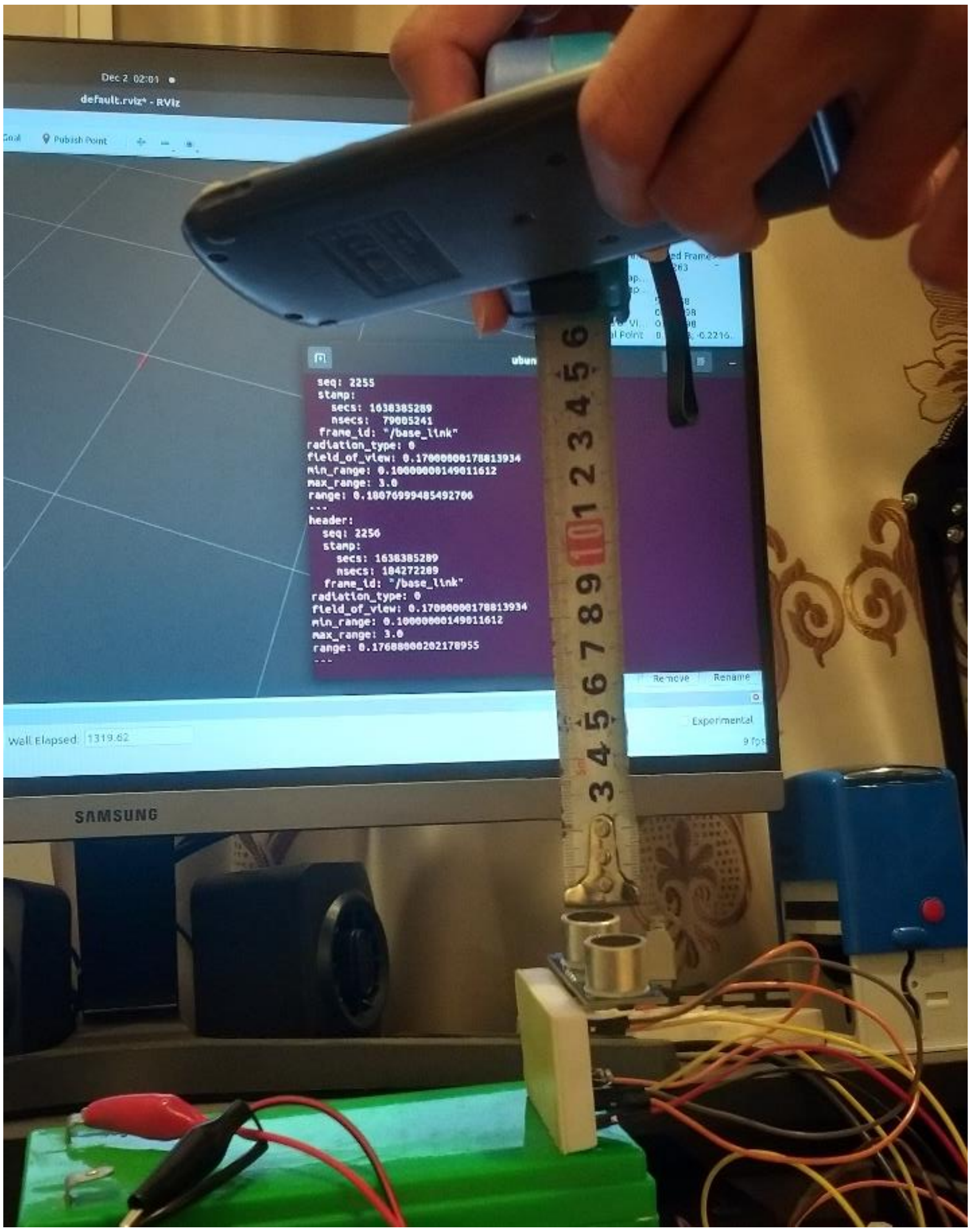


Figure 2.3.1.4 Measured velocity at setpoint 90RPM

2.3.2 Ultrasonic sensor



```
  seq: 2255
  stamp:
    secs: 1638385289
    nsecs:  79005241
  frame_id: "/base_link"
radiation_type: 0
field_of_view: 0.17000000178813934
min_range: 0.10000000149011612
max_range: 3.0
range: 0.18076999485492706
---
header:
  seq: 2256
  stamp:
    secs: 1638385289
    nsecs: 184272289
  frame_id: "/base_link"
radiation_type: 0
field_of_view: 0.17000000178813934
min_range: 0.10000000149011612
max_range: 3.0
range: 0.17680000202178955
---
```

Figure 2.3.2.1 Distance measured by ultrasonic sensor

Explanation:

- "seq": "seq" is stand for "sequence", indicates the order number of a message. The order number of the message in picture is 2256.

- "stamp": means timestamps, includes second and nanosecond passed from Epoch time (start from 00:00:00 UTC on 1 January 1970). Timestamp in picture contains secs: 1638385289 and nsecs: 184272289. (02:01 on 2 December 2021).

- "frame_id": is the name of the reference frame in the system which this ultrasonic sensor belongs to. The reference frame of the sensor in picture: "/base_link".

- "radiation_type": is the number representing the type of ranging sensor (0: ultrasound, 1: infrared).

- “field_of_view”: is the size of the arc that the distance reading is valid for [rad] the object causing the range reading may have been anywhere within -field_of_view/2 and field_of_view/2 at the measured range. Zero angle corresponds to the x-axis of the sensor. For ultrasonic sensor HC-SR04, its field_of_view is 0.26 radidans (approximately 17 degrees).

- “min_range”: is minimum range value of sensor HC-SR04 is 0.02 [m].

- “max_range”: is maximum range value of sensor HC-SR04 is 3.00 [m].

- “range”: is range data of sensor in meter. It is the value greater than “min_range” and smaller than “max_range”. HC-SR04 sensor indicates the value of 0.1768 [m] which is an acceptable value relative to the actual distance of 0.171 [m].

2.3.3 Map building

The map was built by Hector slam method via Hector_slam node and teleop_key to control robot moving slowly around the room. After the shape of room is completely displayed via Rviz, map_server was used to save data into two files: my_map.pgm and my_map.yaml

my_map.pgm: Show the shape of the map

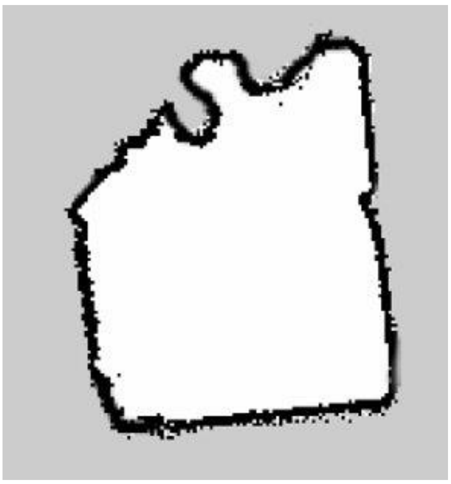
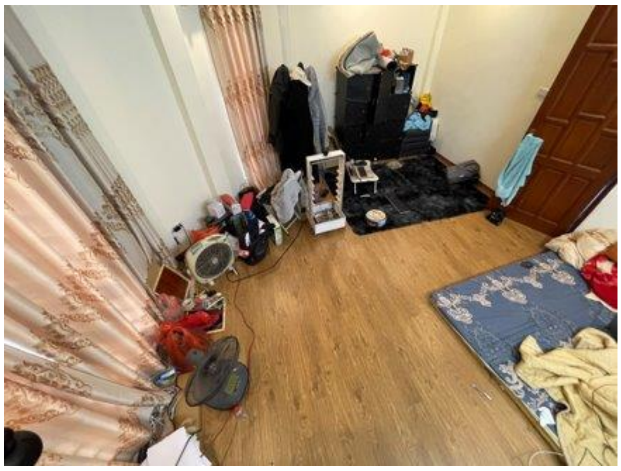
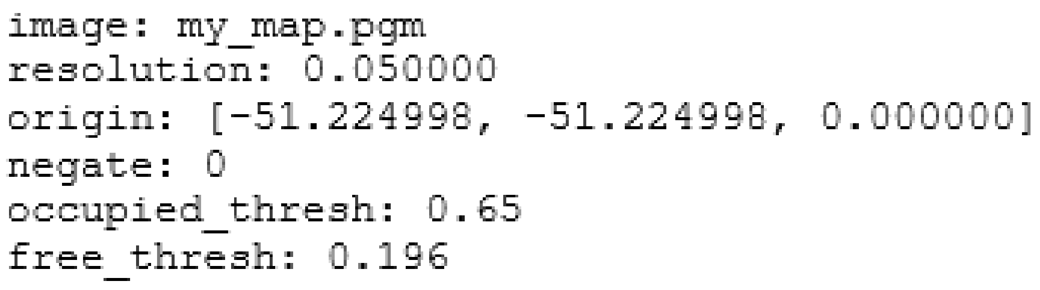



Figure 2.3.3.1 Map generated by Hector_slam

my_map.yaml: Show the metadata of map



```
image: my_map.pgm
resolution: 0.050000
origin: [-51.224998, -51.224998, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figure 2.3.3.2 Metadata of the map

Required fields:

- **image** : Path to the image file containing the occupancy data; can be absolute, or relative to the location of the YAML file
- **resolution** : Resolution of the map, meters / pixel

- **origin** : The 2-D pose of the lower-left pixel in the map, as (x, y, yaw), with yaw as counterclockwise rotation (yaw=0 means no rotation). Many parts of the system currently ignore yaw.
- **occupied_thresh** : Pixels with occupancy probability greater than this threshold are considered completely occupied.
- **free_thresh** : Pixels with occupancy probability less than this threshold are considered completely free.
- **negate** : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

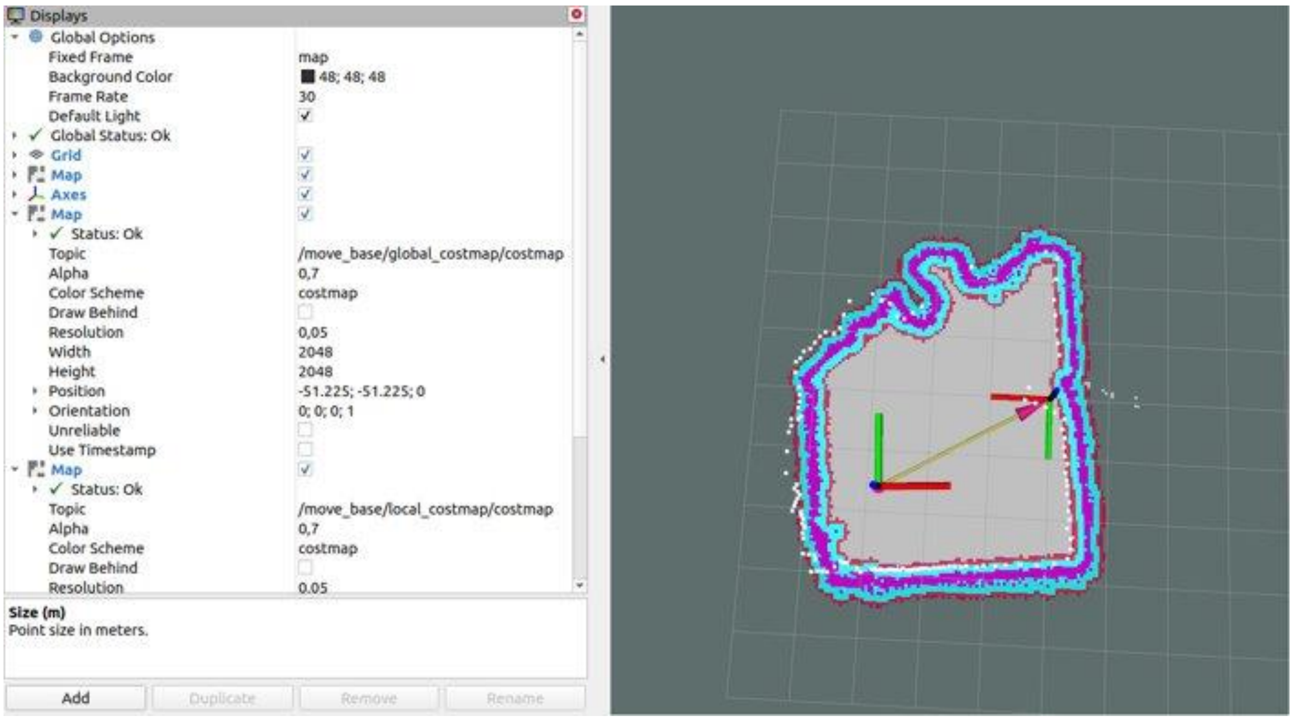
Optional parameter:

- **mode** : Can have one of three values: trinary, scale, or raw. Trinary is the default. More information on how this changes the value interpretation is in the next section.

2.2.4 Auto-navigation:

After doing command **roslaunch navstack_pub servebot.launch**, the whole system starts and rviz screen appears. Add topic /scan and /map to display map and current scan data from Lidar. Now the robot needs to be set its initial pose on map in RVIZ by **2D Pose Estimate** button corresponding to its real pose.

Adjust with the **2D Pose Estimate** until the data from lidar scan is matched with the map, which mean the initial pose of robot is right.



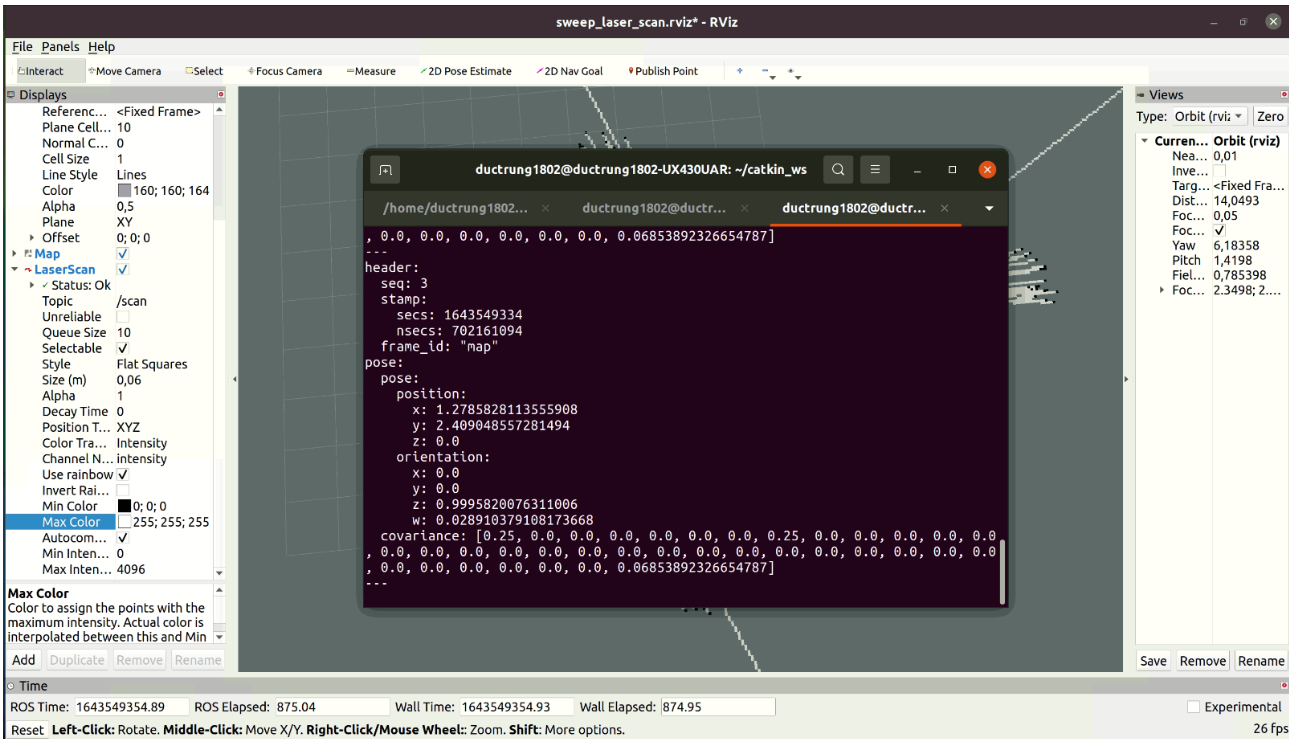
Displays

Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
Grid	✓
Map	✓
Axes	✓
Map	✓
✓ Status: Ok	
Topic	/move_base/global_costmap/costmap
Alpha	0,7
Color Scheme	costmap
Draw Behind	☐
Resolution	0,05
Width	2048
Height	2048
Position	-51.225; -51.225; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Use Timestamp	☐
Map	✓
✓ Status: Ok	
Topic	/move_base/local_costmap/costmap
Alpha	0,7
Color Scheme	costmap
Draw Behind	☐
Resolution	0.05

Size (m)
Point size in meters.

Figure 2.3.3.3 Setting Initial Pose

When the pose is set, information about robot’s initial position and orientation will be published into /initalpose topic. This information includes position and orientation of the robot.


sweep_laser_scan.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Views
ductrung1802@ductrung1802-UX430UAR: ~/catkin_ws
/home/ductrung1802... ductrung1802@ductr... ductrung1802@ductr...
Add Duplicate Remove Rename
Save Remove Rename
Time
Reset

```
Referenc...   <Fixed Frame>
Plane Cell... 10
Normal C...   0
Cell Size     1
Line Style    Lines
Color         160; 160; 164
Alpha         0,5
Plane         XY
Offset        0; 0; 0
Map           ✓
LaserScan     ✓
  Status: Ok
  Topic         /scan
  Unreliable
  Queue Size    10
  Selectable    ✓
  Style         Flat Squares
  Size (m)      0,06
  Alpha         1
  Decay Time    0
  Position T... XYZ
  Color Tra...  Intensity
  Channel N...  intensity
  Use rainbow   ✓
  Invert Rai...
  Min Color     0; 0; 0
  Max Color     255; 255; 255
  Autocom...    ✓
  Min Inten...  0
  Max Inten...  4096
Max Color
Color to assign the points with the maximum intensity. Actual color is interpolated between this and Min
```

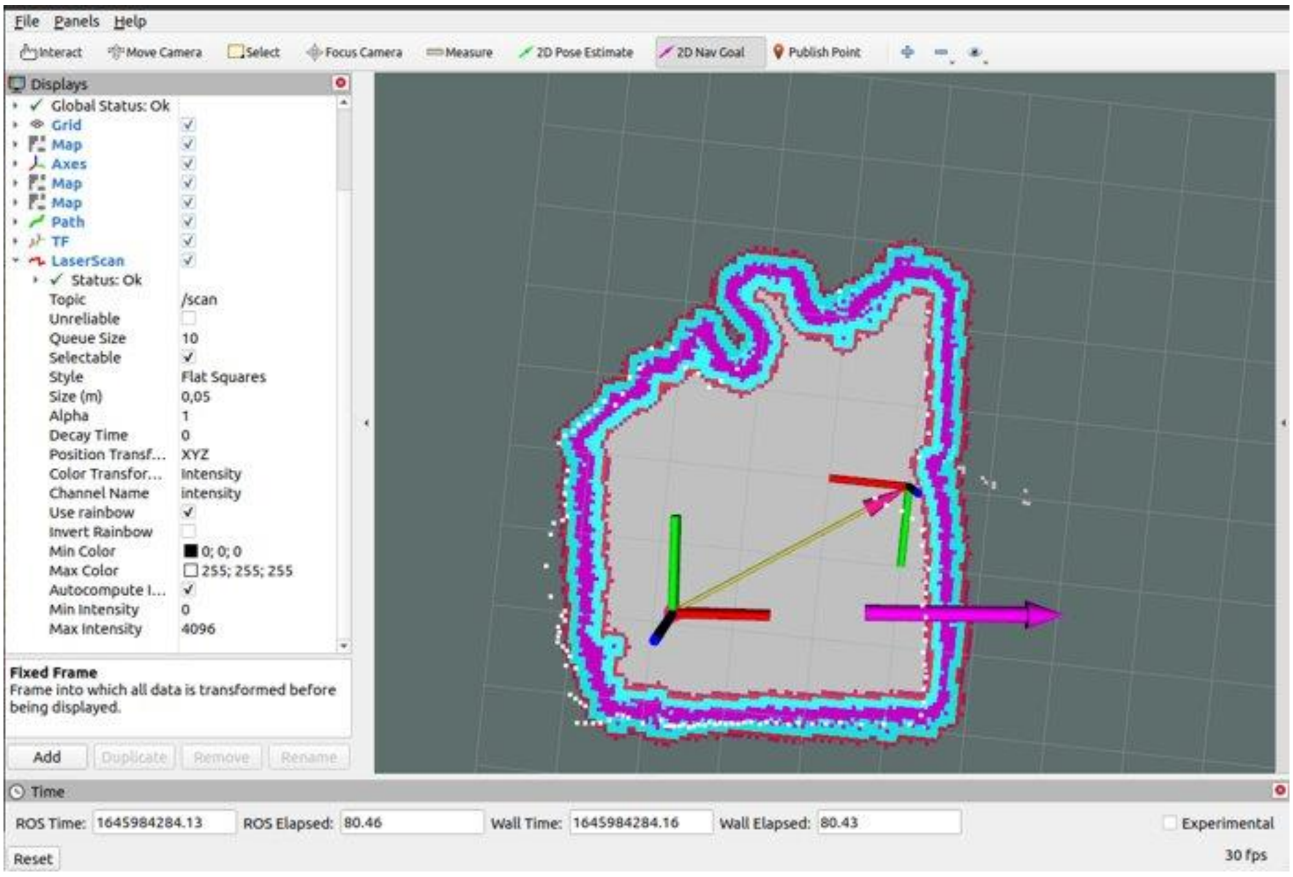
```
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06853892326654787]
---
header:
  seq: 3
  stamp:
    secs: 1643549334
    nsecs: 702161094
  frame_id: "map"
pose:
  pose:
    position:
      x: 1.2785828113555908
      y: 2.409048557281494
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.9995820076311006
      w: 0.028910379108173668
  covariance: [0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06853892326654787]
---
```

```
Type: Orbit (rviz)   Zero
Curren... Orbit (rviz)
  Nea...  0,01
  Inve...
  Targ... <Fixed Fra...
  Dist... 14,0493
  Foc...  0,05
  Foc...  ✓
  Yaw     6,18358
  Pitch   1,4198
  Fiel... 0,785398
  Foc...  2.3498; 2....
```

```
ROS Time: 1643549354.89   ROS Elapsed: 875.04   Wall Time: 1643549354.93   Wall Elapsed: 874.95   Experimental
Reset  Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   26 fps
```

Figure 2.3.3.4 Data published into /initialpose topic

The next step is setting a goal for robot to reach by use **2D Nav Goal.**

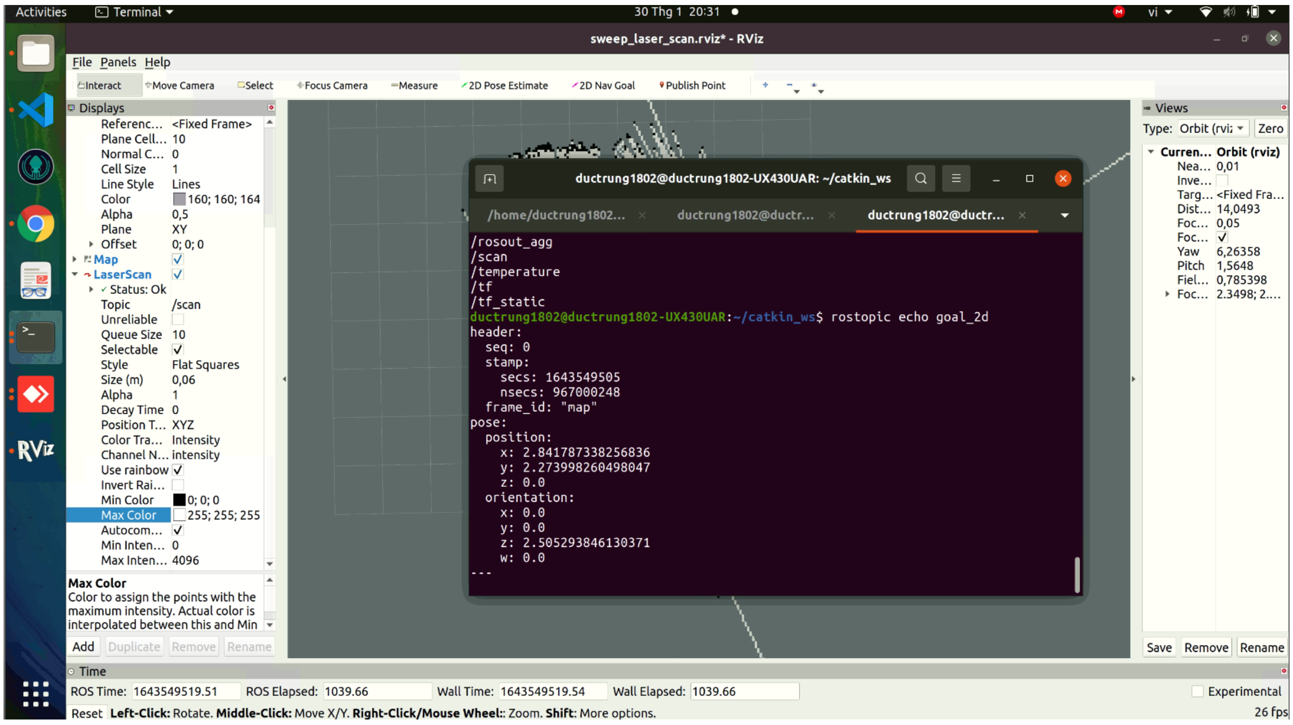

File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Displays
Add Duplicate Remove Rename
Time
Reset

```
Global Status: Ok
Grid                ✓
Map                 ✓
Axes                ✓
Map                 ✓
Map                 ✓
Path                ✓
TF                  ✓
LaserScan           ✓
  Status: Ok
  Topic               /scan
  Unreliable
  Queue Size          10
  Selectable          ✓
  Style               Flat Squares
  Size (m)            0,05
  Alpha               1
  Decay Time          0
  Position Transf...  XYZ
  Color Transfor...   Intensity
  Channel Name        intensity
  Use rainbow         ✓
  Invert Rainbow
  Min Color           0; 0; 0
  Max Color           255; 255; 255
  Autocompute I...    ✓
  Min Intensity       0
  Max Intensity       4096
Fixed Frame
Frame into which all data is transformed before being displayed.
```

```
ROS Time: 1645984284.13   ROS Elapsed: 80.46   Wall Time: 1645984284.16   Wall Elapsed: 80.43   Experimental
Reset                                                                                            30 fps
```

Figure 2.3.3.5 Setting Goal Point

After the goal is placed on map, the information of goal including position and orientation will be published into /goal_2d topic. This also includes the information of the position and orientation of the robot.



Displays	
Referenc...	<Fixed Frame>
Plane Cell...	10
Normal C...	0
Cell Size	1
Line Style	Lines
Color	▨ 160; 160; 164
Alpha	0,5
Plane	XY
Offset	0; 0; 0
Map	✓
LaserScan	✓
Status: Ok	
Topic	/scan
Unreliable	
Queue Size	10
Selectable	✓
Style	Flat Squares
Size (m)	0,06
Alpha	1
Decay Time	0
Position T...	XYZ
Color Tra...	Intensity
Channel N...	intensity
Use rainbow	✓
Invert Rai...	
Min Color	■ 0; 0; 0
Max Color	□ 255; 255; 255
Autocom...	✓
Min Inten...	0
Max Inten...	4096

Max Color
Color to assign the points with the maximum intensity. Actual color is interpolated between this and Min

```
ductrung1802@ductrung1802-UX430UAR: ~/catkin_ws
/home/ductrung1802...   ductrung1802@ductr...   ductrung1802@ductr...
/rosout_agg
/scan
/temperature
/tf
/tf_static
ductrung1802@ductrung1802-UX430UAR:~/catkin_ws$ rostopic echo goal_2d
header:
  seq: 0
  stamp:
    secs: 1643549505
    nsecs: 967000248
  frame_id: "map"
pose:
  position:
    x: 2.841787338256836
    y: 2.273998260498047
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 2.505293846130371
    w: 0.0
---
```

Views	
Type:	Orbit (rviz) Zero
Curren...	Orbit (rviz)
Nea...	0,01
Inve...	
Targ...	<Fixed Fra...
Dist...	14,0493
Foc...	0,05
Foc...	✓
Yaw	6,26358
Pitch	1,5648
Fiel...	0,785398
Foc...	2.3498; 2....

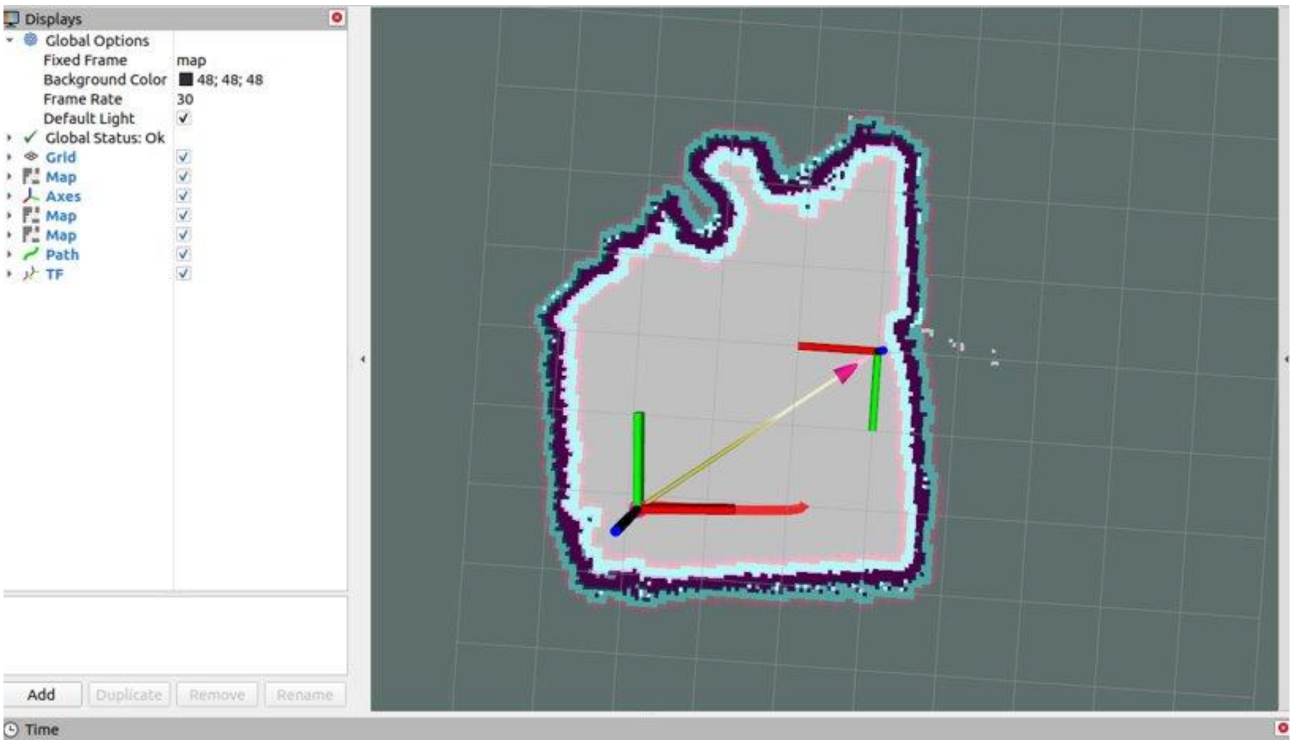
Time

ROS Time: 1643549519.51 ROS Elapsed: 1039.66 Wall Time: 1643549519.54 Wall Elapsed: 1039.66 Experimental

Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options. 26 fps

Figure 2.3.3.6 The information of set goal

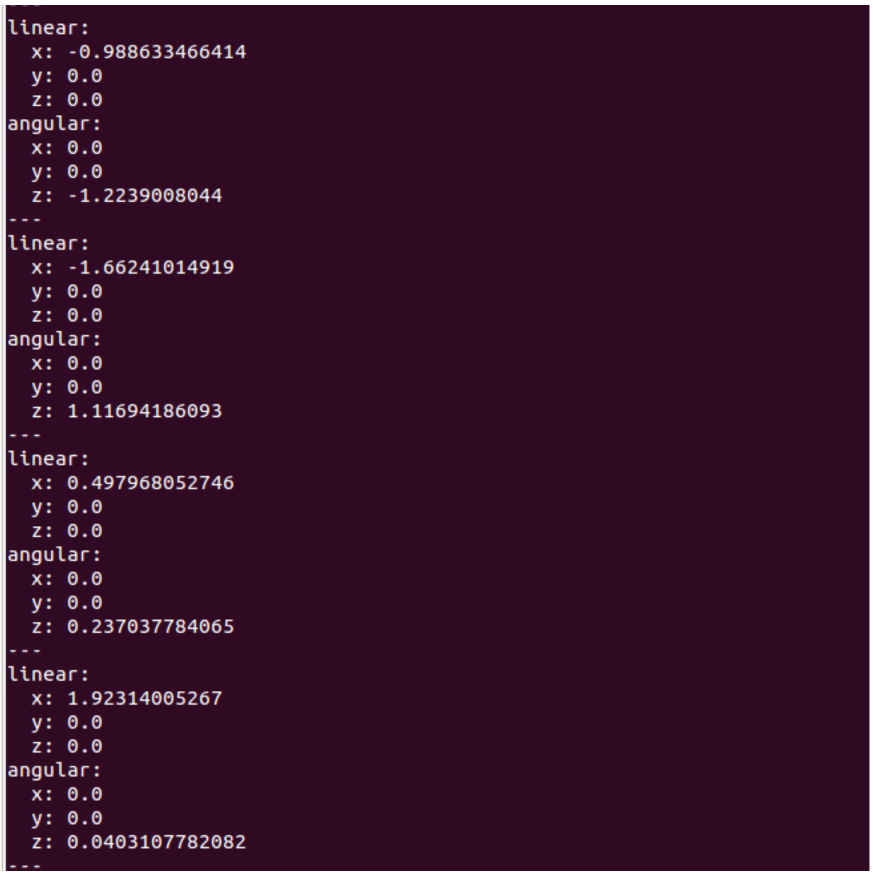
Robot starts to plan the path from Initial Point to the Goal Point.



Displays	
Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Grid	✓
Map	✓
Axes	✓
Map	✓
Map	✓
Path	✓
TF	✓

Figure 2.3.3.7 Path planned by global planner

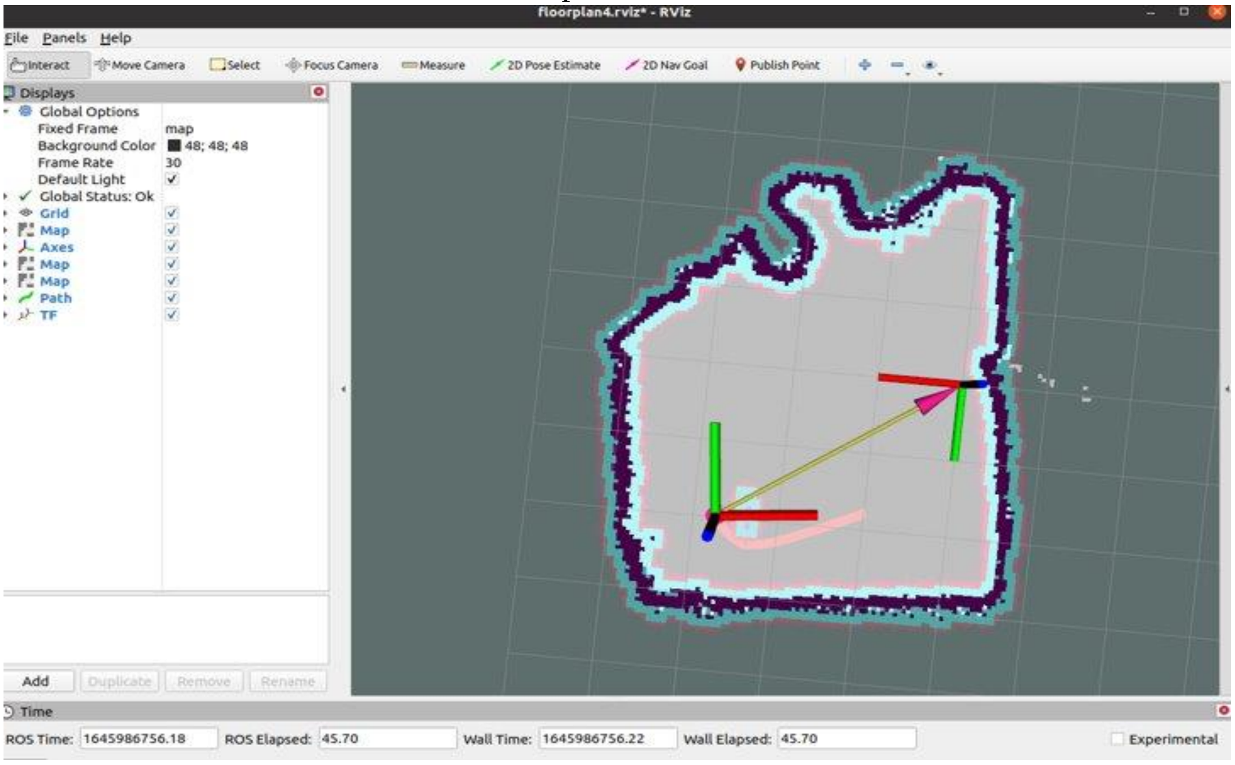
Immediately this data will be processed by move_base package to plan path to goal then publish the required velocity to control the movement of robot. This velocity information is published into topic cmd_vel, which is the input of the motor node. This information includes the linear and angular velocity of the robot.



```
linear:
  x: -0.988633466414
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -1.2239008044
---
linear:
  x: -1.66241014919
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.11694186093
---
linear:
  x: 0.497968052746
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.237037784065
---
linear:
  x: 1.92314005267
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0403107782082
---
```

Figure 2.3.3.8 Published velocity information

Robot suddenly realizes a obstacle and changes to alternative route. This is handled by local planner.

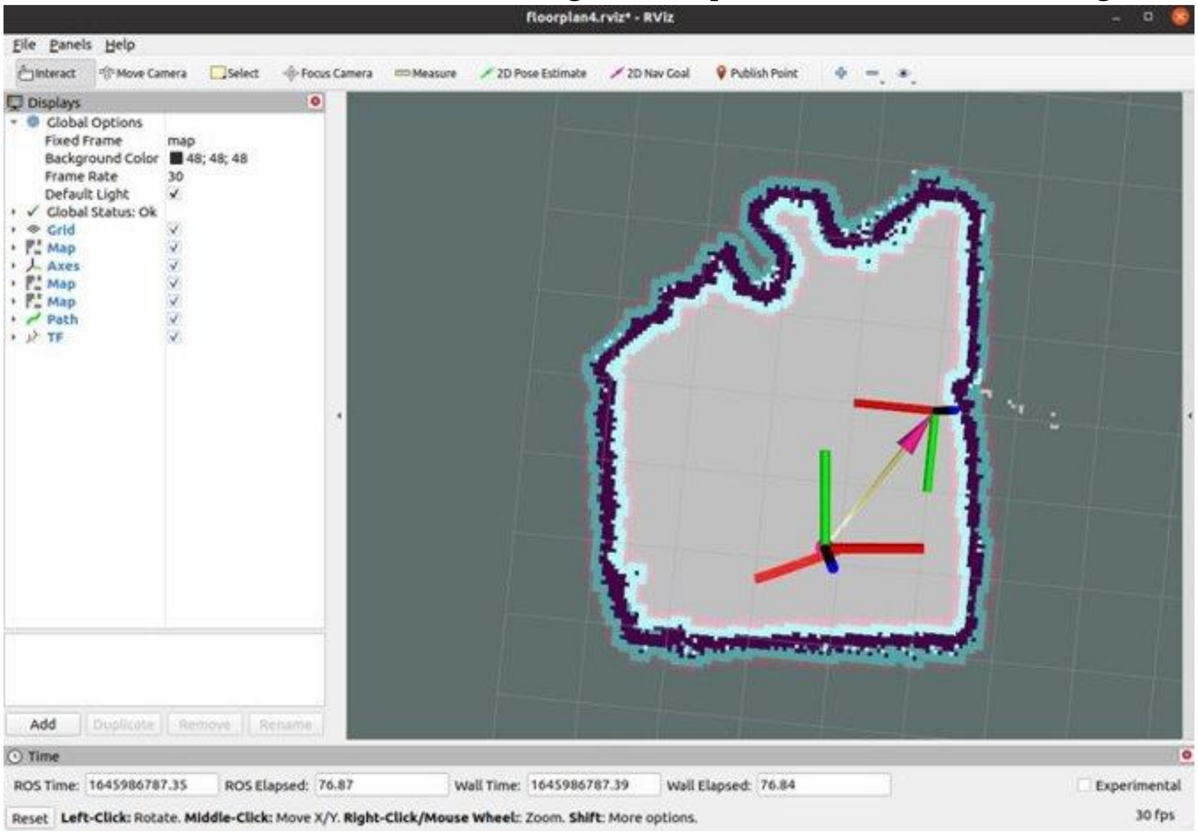


Fixed Frame: map
Background Color: 48; 48; 48
Frame Rate: 30
Default Light: ✓
Global Status: Ok

ROS Time: 1645986756.18 ROS Elapsed: 45.70 Wall Time: 1645986756.22 Wall Elapsed: 45.70

Figure 2.3.3.9 Alternative path to avoid obstacle

The robot continues to move along the new path until it reached it final goal .



Global Options
Fixed Frame map
Background Color ■ 48; 48; 48
Frame Rate 30
Default Light ✓
Global Status: Ok

ROS Time: 1645986787.35 ROS Elapsed: 76.87 Wall Time: 1645986787.39 Wall Elapsed: 76.84

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.

Figure 2.3.3.10 Robot reached the goal point

CONCLUSION

After over 4 months of implementing serving mobile robot, many problems we had to face up with. We also finished some parts of controlling mobile robot:

+ Controlling motor speed by PID controller.

+ Applying Kalman filter in measuring motor speed.

+ Building reality map by LiDAR for the future path planning.

+ Gaining more knowledge about ROS in robot controlling.

+ Make the robot able to auto-navigate to desired goals.

Because of the complexity of project and the expansion of Covid-19 pandemic, several parts of designing serving mobile robot are planned to be covered in the future:

+ Combining Ultrasonic sensors in detecting obstacle.

+ Combining differential drive with PID controller to move the robot to desired goal.

+ Finding a method for robot self-localization.

+ Building app on mobile phone to display the map and to control the robot.

+ Programming the customer-interacting part which includes an infrared sensor, a speaker and an OLED display.

+ Develop HMI interaction via Bluetooth speaker by implementing text to speech and speech to text API.

+ Building database for customers on the mobile app.

REFERENCES

1. Alex. 2022. "Online Kalman Filter Tutorial". Kalmanfilter.Net. https://www.kalmanfilter.net/kalman1d.html?fbclid=IwAR2uqxPT3S0IbQCmbph2uwVYKZzlZUEWz0xJqr1MD6-RCUHPkRVKMkgjTXk.

2. Gregor Klancar, Andrej Zdesar, Saso Blazic, and Igor Skrjanc. 2017. Wheeled Mobile Robotics: From Fundamentals Towards Autonomous Systems (1st. ed.). Butterworth-Heinemann, USA.

3. Yoonseok Pyo, Hancheol Cho, Leon Jung, Darby Lim. (2017). ROS Robot Programming (English). ROBOTIS. http://community.robotsource.org/t/download-the-ros-robot-programming-book-for-free/51

4. How to Set Up the ROS Navigation Stack on a Robot – Automatic Addison (2021). Available at: https://automaticaddison.com/how-to-set-up-the-ros-navigation-stack-on-a-robot/?fbclid=IwAR2qgAFNx3z4L5yfuFqnDPJt4bel7VG0ni9Knc5zfe22tqy9Mbi8cu41LdQ (Accessed: 30 January 2022).

5. Carol Fairchild and Thomas L. Harman. 2016. ROS Robotics By Example. Packt Publishing.

6. Raveendran, Rajesh & Ariram, Siva & Tikanmäki, Antti & Röning, Juha. (2020). Development of task-oriented ROS-based Autonomous UGV with 3D Object Detection. 427-432. 10.1109/RCAR49640.2020.9303034.