ក្រសួងអប់រំ យុវជន និងកីឡា

វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

ដេប៉ាតឺម៉ង់ ទេពកោសល្យ អគ្គិសនី និងថាមពល

គម្រោងសញ្ញាបត្រវិស្វករ

ការកំណត់ទីតាំង និងការដ្ឋាស់ទីដោយស្វ័យប្រវត្តិនៃរ៉ូបូតកង់បួន

(Indoor Localization and Autonomous Navigation of a Four-Wheeled Mobile Robot)

និស្សិត	: សេង ធារ៉ា
ឯកទេស	: អគ្គិសនី និងថាមពល
សាស្ត្រាចារ្យទទួលបន្ទុក	: លោក ទេព សុវិជ្ជា
ឆ្នាំសិក្សា	: ២០១៩ – ២០២០

MINISTERE DE L'EDUCATION, DE LA JEUNESSE ET DES SPORTS

INSTITUT DE TECHNOLOGIE DU CAMBODGE

DEPARTEMENT DE GENIE ELECTRIQUE ET ENERGETIQUE

MEMOIRE DE FIN D'ETUDES INGENIEUR

Localisation Intérieure et Navigation Autonome d'un Robot Mobile à Quatre Roues

Etudiant	**: SENG Theara**
Spécialité	**: Electrique et Energétique**
Maître de stage	**: M. TEP Sovichea**
Année scolaire	**: 2019 – 2020**

ក្រសួងអប់រំ យុវជន និងកីឡា

វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

ដេប៉ាតឺម៉ង់ ទេពកោសល្យ អគ្គិសនី និងជាមពល

គម្រោងសញ្ញាបត្រវិស្វករ

របស់និស្សិត សេង ធារ៉ា

ការលបរិច្ឆេទការពារនិក្ខេបបទ : ០៨ ខែ កក្កដា ឆ្នាំ ២០២០

អនុញ្ញាតឲ្យការពារគម្រោង

នាយកវិទ្យាស្ថាន: _____

ថ្ងៃទី ខែ ឆ្នាំ ២០២០

ការកំណត់ទីតាំង និងការផ្លាស់ទីដោយស្វ័យប្រវត្តិនៃរ៉ូបូតកង់បួន

សហគ្រាស : វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

ប្រធានដេប៉ាតឺម៉ង់ : បណ្ឌិត ជ្រិន ផុក_____

សាស្ត្រាចារ្យទទួលបន្ទុក : លោក ទេព សុវិជ្ជា_____

អ្នកទទួលខុសត្រូវក្នុងសហគ្រាស : លោក ទេព សុវិជ្ជា_____

រាជធានីភ្នំពេញ ឆ្នាំ២០២០

MINISTERE DE L'EDUCATION,

DE LA JEUNESSE ET DES SPORTS

INSTITUT DE TECHNOLOGIE DU CAMBODGE

DEPARTEMENT DE GENIE ELECTRIQUE ET ENERGETIQUE

MEMOIRE DE FIN D'ETUDES INGENIEUR

DE M. SENG Theara

Date de soutenance : le 08 juillet 2020

« Autorise la soutenance du mémoire »

Directeur de l'Institut : _____

Phnom Penh, le ……..…… 2020

Localisation Intérieure et Navigation Autonome d'un Robot Mobile à Quatre Roues

Etablissement du stage : Institut de Technologie du Cambodge

Chef du département : Dr. CHRIN Phok _____

Maître de stage : M. TEP Sovichea _____

Tuteur de stage : M. TEP Sovichea _____

PHNOM PENH, 2020

ACKNOWLEDGEMENTS

First of all, I would like to thank to my **My Parents** for their constant support and encouragement throughout my graduate studies. They have been a source of comfort during my difficult times and supported me mentally and financially throughout my degree.

I sincerely appreciate **H.E Dr. OM Romny**, Director General of the Institute of Technology of Cambodia (ITC) for the adequate administation and management of the Institute, especially his approval allowing me to defense this thesis for the completion of my academic study.

I wish to indicate my respect toward **Dr. CHRIN Phok**, Head of Department of Electrical and Energy Engineering (GEE) for his superior management as well as all the professors and lecturers of GEE for their precious guidance and lectures, which have greatly enlightened my knowledge.

Furthermore, I am gradeful to my thesis advisor, **Mr. TEP Sovichea**, whose teaching and guidance enabled me to better understand my thesis project which lead to a successful outcome. I sincerely thank him for his support in solving the difficult challenged of this project.

Finally, yet importantly, I am glad to denote my appreciation to all supportive people including my friends and seniors whose names were not mentioned above for spending their valuable time providing me the training samples. It has held significant impacts on the success of my final year project.

សេចក្តីសង្ខេប

គោលបំណងនៃគម្រោងនេះគឺការបង្កើតរ៉ូបូតកង់បួនMECANUMដែលអាចធ្វើការស្កេន រួមទាំងការកំណត់ទីតាំងខ្លួនរបស់វាដ៏ជាក់លាក់នៅក្នុងបរិវេនបិតជិតមួយ។ Feedback Control System គឺជាផ្នែកសំខាន់មួយនៅក្នុងរ៉ូបូតដែលកំណត់ដំណើរការនៃការផ្លាស់ទីដូចជាភាពច្បាស់លាស់ និងលំនឹង។

បញ្ហាប្រឈមដ៏សំខាន់មួយនៅក្នុងគម្រោងនេះគឺការកំណត់ទីតាំង និងលំងាកមុំច្បាស់លាស់របស់រ៉ូបូតធៀបជាមួយបរិវេន។ ដើម្បីធ្វើការប៉ាន់ស្មានទីតាំងរបស់របស់រ៉ូប៉ូតយើងមិនអាចពឹងទៅលើតែសេនស័រតែមួយនោះទេ។ ដើម្បីដោះស្រាយបញ្ហានេះយើងអាចប្រើសេនស័រច្រើនហើយរួមបញ្ចូលវាចូលគ្នាដោយការប្រើប្រាស់ Extended Kalman Filter (EKF)។ Inertial Measurement Unit(IMU)គឺជាសេនស័រមួយដែលផ្តល់ទិន្នន័យនៃលំងាក់មុំ ល្បឿនលីនេអ៊ែរល្បឿនមុំនិងសំទុះលីនេអ៊ែរហើយអ៊ុនកូដឌ័ររបស់ម៉ូតរ៍ផ្តល់អោយនូវព័តិមានទីតាំងរបស់រ៉ូបូត។ នៅក្នុងគម្រោងនេះមានIMUចំនួន២ដែលត្រូវបានប្រើ។IMUទីមួយត្រូវបានយកទៅរួមបញ្ចូលគ្នាជាមួយនិងអ៊ុនកូដឌ័ររបស់ម៉ូតរ៍ដើម្បីបង្កើតទិន្នន័យដំបូងរបស់odometry។បន្ទាប់មកទិន្នន័យនេះត្រូវបានយកទៅរួមបញ្ចូលគ្នាជាមួយIMUមួយទៀតដោយប្រើEKFដើម្បីទទួលបាននូវព័តិមានodometryរបស់រ៉ូបូតកាន់តែច្បាស់និងអាចជឿទុកចិត្តបាន។ក្រៅពីIMUនិងអ៊ុនកូដឌ័ររបស់ម៉ូតរ៍នៅមានសេនស័រLidarដែលប្រើប្រាស់ដើម្បីស្កេនផែនទីរបស់បរិវេន។

ដើម្បីទទួលបាននូវតម្រូវការដែលរ៉ូបូតអាចកំណត់ទីតាំងនិងផ្លាស់ទីដោយស្វ័យប្រវត្តិនៅក្នុងបរិវេនRobotOperatingSystem(ROS)គឺជាframeworkមួយដ៏សំខាន់ដែលផ្តល់នូវក្បួនដោះស្រាយនិងកញ្ចប់កូតដែលជួយសម្រួលដល់អ្នកអភិវឌ្ឍន៍ក្រោយទៀត។ នៅក្នុងROSមានក្បួនដោះស្រាយមួយចំនួនដូចជា Simultaneous Localization and Mapping (SLAM) ដែលប្រើដើម្បីស្កេនយកផែនទីរបស់ទីតាំងមួយហើយថែមទាំងអាចកំណត់ទីតាំងរបស់រ៉ូប៉ូតផងដែរ ហើយ Adaptive Monte Carlo Localization (AMCL) គឺប្រើដើម្បីកំណត់ទីតាំងរបស់រ៉ូបូតនៅលើផែនទីដែលមានស្រាប់ហើយធ្វើការផ្លាស់ទីដោយស្វ័យប្រវត្តិនៅលើផែនទីនោះ។

RESUME

Le but de ce projet est de construire le robot mobile à quatre roues en mecanum qui peut cartographier l'environnement intérieur ainsi que se localiser dans une position précise. Pour un robot mobile, le système de contrôle de rétroaction est la partie importante qui décide de la performance des mouvements, tels que la précision et la stabilité.

L'un des plus difficiles de ce projet est de localiser le robot dans la position et l'orientation précises par rapport à un environnement. Afin d'obtenir une estimation précise, nous ne pouvons pas compter sur une seule source de capteur. Pour surmonter ce problème, nous devons utiliser plusieurs capteurs et les fusionner ensemble à l'aide du filtre de Kalman étendu (EKF). L'unité de mesure inertielle (IMU) nous fournit l'orientation, la vitesse linéaire et angulaire et l'accélération linéaire, tandis que les encodeurs de roue donnent les informations de la position du robot. Deux IMU sont utilisées dans le projet. L'un est fusionné avec des encodeurs à roue pour construire des données d'odométrie initiales. Ces données sont ensuite fusionnées avec une autre IMU utilisant EKF afin d'obtenir des informations d'odométrie plus précises et fiables du robot. A côté des encodeurs à roue et des IMU, il y a un capteur lidar qui est utilisé pour scanner la carte de l'environnement.

Pour atteindre les exigences, que le robot mobile soit capable de localiser et de naviguer dans l'environnement de manière automatique, ROS est un cadre puissant qui fournit de nombreux algorithmes et packages pour les développeurs. Il existe plusieurs algorithmes tels que la localisation et le mappage simultanés (SLAM) sont utilisés pour cartographier l'environnement ainsi que de se localiser sur la même carte et la localisation adaptative de Monte Carlo (AMCL) qui est utilisée pour localiser le robot dans la carte existante et naviguer d'une position à une autre position de manière autonome.

ABSTRACT

The purpose of this project is to build the four-wheeled mecanum mobile robot which can map the indoor environment as well as localize itself in an accurate position. For a mobile robot, Feedback Control System is an important part which decides the performance of movements, such as precision and stability.

One of the most challenging in this project is to localize the robot in an accurate position and orientation with respect to an environment. To get the accurate estimation, we can not rely on one sensor source. To overcome this problem, we need to use multiple sensors and fuse them together using the Extended Kalman Filter (EKF). Inertial Measurement Unit (IMU) provides us orientation, linear and angular velocity, and linear acceleration, while wheel encoders give the information of the robot's position. Two IMUs are used in the project. One is fused with wheel encoders to construct initial odometry data. This data is later fused with another IMU using EKF in order to get more accurate and reliable odometry information of the robot. Beside wheel encoders and IMUs, there is a lidar sensor which is used to scan the map of the environment.

To achieve the requirements, that the mobile robot should be able to localize and navigate in the environment autonomously, ROS is a powerful framework that provides many algorithms and packages for developers. There are several algorithms such as Simultaneous Localization and Mapping (SLAM) are used for mapping the environment as well as localize its self on the same map and Adaptive Monte Carlo Localization (AMCL) which is used to localize the robot in the existing map and navigate from a position to another position autonomously.

NOMENCLATURE

AMCL Adaptive Monte Carlo Localization

DMP Digital Motion Processing

EKF Extended Kalman Filter

I2C Inter-Integrated Circuit

IMU Inertial Measurement Unit

ROS Robot Operating System

rviz Robot Operating System Visualization

SLAM Simultaneous Localization and Mapping

tf Transform Frame

TABLE OF CONTENTS

LIST OF FIQURES

LIST OF TABLES

1. Introduction

1.1. Study Background

Nowadays, an autonomous mobile robot is widely used for several purposes such as industrial transport, logistics, food serving, and mobile operation. A critical challenge in the mobile robot is navigation which is the ability to answer the question "Where am I?" and "Where am I going?".

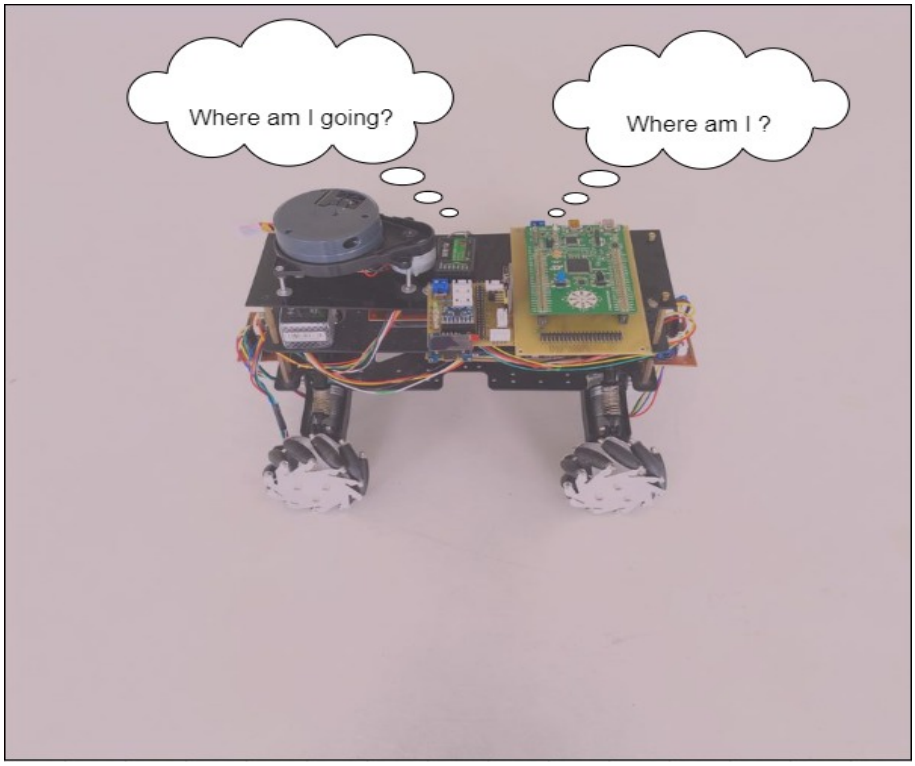

Figure 1.1: Autonomous Mobile Robot

The solution to these questions is known as localization which uses sensors to estimate the position of the robot. However, one sensor is imperfect to localize the robot in an environment with an accuracy position and orientation. Due to the noise of the sensor, the robot will lose itself in the environment or lead to the big amount of error. The wheel encoders are used to estimate the position of the robot, but with the wheel slippage or uneven floors can affect accuracy. To get an accurate state estimation, fusing multiple sensors can be a solution to this problem. Fusing multiple sensors can lead to an overall position which error will be less compared to the estimated error of a single sensor. [5]

1.2. Objective of Project

The main functionality of the robot is navigation in which the robot will have the ability to navigate from a start position to the end position without collision with its surroundings. Sensors which are used in this project are:

- **mpu6050** is used to construct the odometry message with wheel encoder.
- **android/imu** is used to get the orientation, linear velocity and acceleration.
- **Wheel Encoders** is used to measure the robot position and velocity.
- **Lidar Scanner** is used to scan map of the unknown environment.

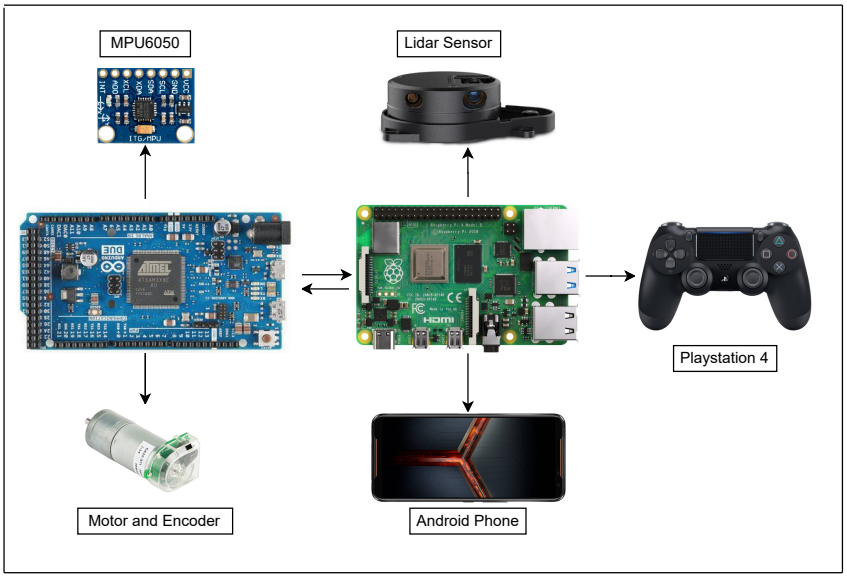

Figure 1.2: Components

1.3. Scope of Work

The scope of the project is:

- Fusing Wheel Encoders and MPU6050
- Fusing Android IMU and Odometry using EKF.
- implement foward kinematics onto the robot
- Mapping the environment using Lidar Sensor.
- Merging Odometry and Lidar Sensor to scan the map using SLAM.
- Navigate from Position A to B autonomously.

1.4. Outline Thesis

There are eight main chapters which contains in this thesis:

- **Introduction:** it includes the study background, objective of project, scope of work and thesis study.
- **Literature Review:** It introduces all relevant methods used in the project.
- **Implimentation and Verification of Forward Kinematic:** It describes the code which publishes the data from Arduino to Raspberry Pi as well as Verification of the Kinematic model Equation.
- **Position Estimation:** It describes the method of fusing Odometry and Android IMU in ROS and the estimation of the robot's position.
- **SLAM Algorithm:** It describes the method of building the map as well as localize the robot in an unknown environment.
- **AMCL Algorithm:** It introduces the navigation method from a start position to the end position autonomously in an existing map.
- **Results and Experiment:** Giving the data of test and discuss about the result and a risk.
- **Conclusion and Future Work:** It concludes the achieved progress with recommendation for future work development.

2. Literature Review

2.1. ROS Concept

ROS is a software framework used for creating robotic applications, and it has a collection of software tools, libraries, and collection of packages that makes robot software development easy.

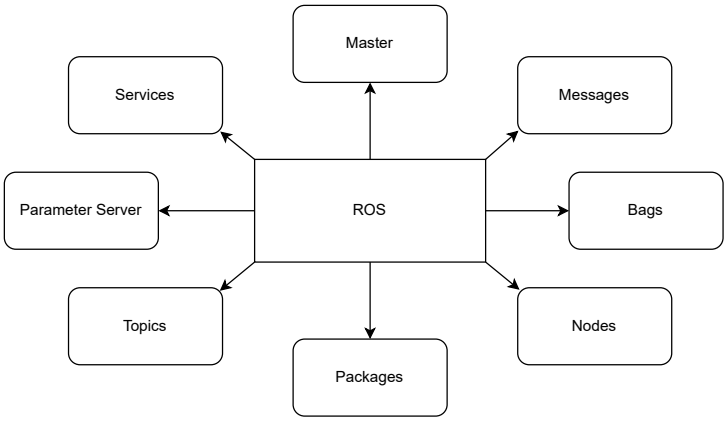

Figure 2.1: Introduction to ROS

- **Master** is a sevice that provides connection information to nodes so that they can transmit messages to one another. Every node connects to master at startup to register details of the message.
- **Messages** are simply a data structure containing the typed field, which can hold a set of data, and that can be sent to another node. Nodes communicate with each other using messages. A topic can only send or receive one type of message at a time. We can create our own message definition and send it through the topic.
- **Bags** are the formats in which to save and play back the ROS topics. ROS bags are an important tool to log the sensor data and the processed data. These bags can be used later for testing our algorithm offline.[3]
- **Nodes** are a process that perform computation using ROS client library such roscpp and rospy. One node can communicate with other nodes using ROS Topics, Services, and Parameters[3].
- **Topics** are channel where nodes can either write or read information. The data flows through the topic in the form of messages. The sending messages over a topic is called Publishing, and receiving the data through a topic is called subscribing.

- **Packages** are the most basic unit of the ROS system[6]. They contain nodes, libraries, configuration files, and so on, which are organized together as a single unit.
- **Services** work in a similar way to ROS topic in that they have a message type definition. In some cases, we need a kind of request or response interaction, in which one node can ask for the execution of a fast procedure to another node such as asking for some quick calculation. The ROS service interaction is like a remote procudure call.
- **Parameter server** is a part of ROS master which allows us to keep the data to be stored in a central location. All nodes can access and modify these values.

2.2. Mathematic Model

The Mecanum wheel is a omnidirectional wheel which is designed for the robot to move in any direction. The Kinematic equation of the four-wheels mecanum mobile robot is:

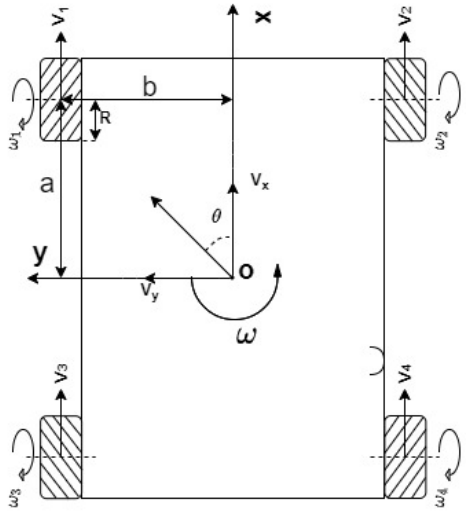

Figure 2.2: Four-wheeled Mecanum Robot Mathematic Model

$$V_1 = V_x - V_y - a\omega - b\omega$$

$$V_2 = V_x + V_y + a\omega + b\omega$$

$$V_3 = V_x + V_y - a\omega - b\omega$$

$$V_4 = V_x - V_y + a\omega + b\omega$$

So, the inverse kinematic of the robot is

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} = \begin{bmatrix} 1 & -1 & -(a+b) \\ 1 & 1 & (a+b) \\ 1 & 1 & -(a+b) \\ 1 & -1 & (a+b) \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ \omega \end{bmatrix}$$

$V = J\dot{q}$, where $V^T = [V_1 \quad V_2 \quad V_3 \quad V_4]$ is the speed of each wheels, $\dot{q}^T = [V_x \quad V_y \quad \omega]$ is the velocity and angular velocity of the robot

Then, $\dot{q} = J^+V$, where J^+ is the pseudo-inverse of J.

$$J^+ = (J^T J)^{-1} J^T$$

The Forward Kinematic of the four mecanum wheels mobile robot is

$$\begin{bmatrix} V_x \\ V_y \\ \omega \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ \frac{-1}{a+b} & \frac{1}{a+b} & \frac{-1}{a+b} & \frac{1}{a+b} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}$$

2.3. 3D Space Representation

2.3.1. Euler angle

Euler angles provide a way to represent the 3D orientation of an object using a combination of three rotations about different axes. The rotation matrix of Euler angles consist of Yaw, Pitch, and Roll as shown in **Figure 2.3**.

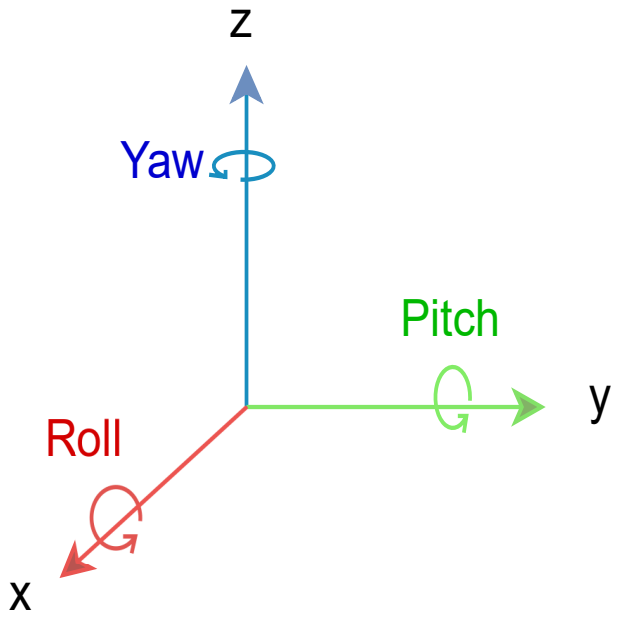

Figure 2.3: Euler Angles

- **Yaw** $R_z(\psi)$[2] represents rotation about the z-axis by an angle ψ. The yaw rotation produces a new coordinate frame where the z-axis is aligned with the initial frame and the y and z axes are rotated by the angle ψ. The rotation matrix corresponding to the yaw rotation is:

$$R_z(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Pitch** $R_y(\theta)$ represents rotation about the y-axis by an angle θ. The pitch rotation produces a new coordinate frame where the y-axis is aligned with the initial frame and the x and z axes are rotated by the angle θ. The rotation matrix corresponding to the pitch rotation is:

$$R_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

- **Roll** $R_x(\phi)$ represents rotation about the x-axis by an angle ϕ. The roll rotation produces

a new coordinate frame where the x-axis is aligned with the initial frame and the y and z axes are rotated by the angle ϕ. The rotation matrix corresponding to the roll rotation is:

$$R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix}$$

The complete rotation matrix for moving from the inertial frame to the body frame is given by:

$$\begin{aligned} R(\psi,\theta,\phi) &= R_z(\psi) \times R_y(\theta) \times R_x(\phi) \\ &= \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \end{aligned}$$

letting c represent cos and s represent sin, we get

$$R(\psi,\theta,\phi) = \begin{bmatrix} c(\psi)c(\theta) & -s(\psi)c(\phi) + c(\psi)s(\theta)s(\phi) & s(\psi)s(\phi) + c(\phi)s(\theta)c(\phi) \\ s(\psi)c(\theta) & c(\psi)c(\phi) + s(\psi)s(\theta)s(\phi) & -c(\psi)s(\phi) + s(\psi)s(\theta)c(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{bmatrix}$$

2.3.2. Quaternions

Quaternions are a four-element vector to represent of an object in 3D space which is written as $a + bi + cj + dk$ where $a, b, c \, and \, d$ are the real numbers and $i, j, \, and \, k$ are the unit vectors. Compared to quaternions, Euler angles are simple and intuitive and they lend themselves well to simple analysis and control. On the other hand, Euler Angles are limited by a phenomenon called **gimbal lock**, which prevents them from measuring orientation when the pitch angle approaches +/-90 degrees. However, quaternions provide an alternative measurement technique that does not suffer from gimbal lock[2]. If θ is the angle of rotation and $i, j, \, and \, k$

are unit vectors representing the axis of rotation, then the quaternion elements are defined as:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} cos(\frac{1}{2}\theta) \\ \mathbf{i}sin(\frac{1}{2}\theta) \\ \mathbf{j}sin(\frac{1}{2}\theta) \\ \mathbf{k}sin(\frac{1}{2}\theta) \end{pmatrix}$$

Then, the rotation matrix describes by the quaternion is given as below:

$$R(q) = \begin{bmatrix} a^2+b^2-c^2-d^2 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & a^2-b^2+c^2-d^2 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & a^2-b^2-c^2+d^2 \end{bmatrix}$$

However, Euler angles are much easier to understand than Quaternions, so to convert from quaternions to the Euler angles, we can use the following formul[2]:

$$\begin{aligned} \phi &= arctan\left(\frac{2(ab+cd)}{a^2-b^2-c^2+d^2}\right) \\ \theta &= -arcsin(2(bd-ac)) \\ \psi &= arctan\left(\frac{2(ad+bc)}{a^2+b^2-c^2-d^2}\right) \end{aligned}$$

2.4. Kalman Filter Method

A Kalman Filter is an optimal estimation algorithm which is used to estimate a system state when it cannot be measured directly and it is also used to find the best estimate of states by combining measurements from various sensors in the presence of noise. Nowadays, Kalman Filter is widely used in Robotic fields such as navigation, motion planning, and trajectory adjustment.

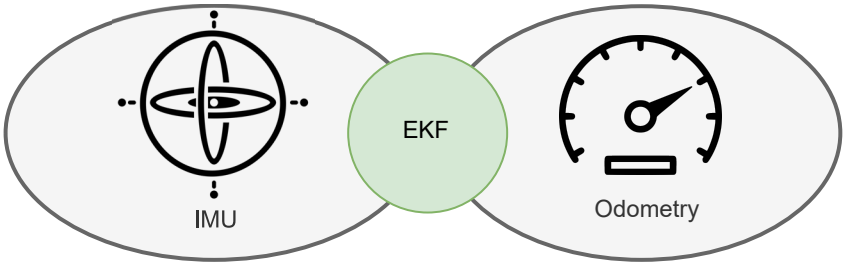

Figure 2.4: Sensor Fusion

To navigate from position A to position B, we need to measure the speed or a position in order to get there in the exact position, but one sensor is imperfect to get the robot navigates to the goal. In this case, we may need to trust the IMU reading. IMU uses accelerometers and gyroscopes to measure the robot's acceleration and angular velocity. However, IMU doesn't provide the robot's position. we need to take the double integral of the acceleration in order to get the position, but it prone to drift due to small error over time. So, to get a better position estimate, we need to use the IMU measurements along with odometer readings.

In this scenario, a Kalman Filter can be used to fuse the IMU and Odometry using EKF to find the optimal estimate of the exact position of the robot.

There are two-step processes in Kalman Filter. The first one is the Prediction step and Second is the update step.

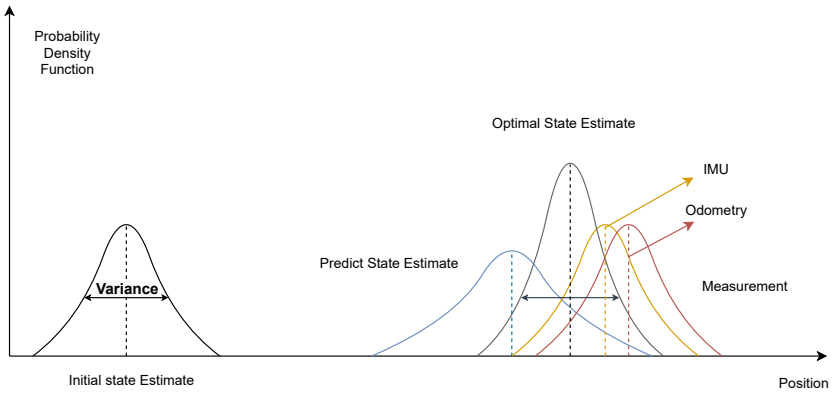

Figure 2.5: Graph of Position Estimation using EKF

- Prediction Step

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$$

$$P_k^- = AP_{k-1}A^T + Q$$

- Update Step

$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-)$$

$$P_k = (I - K_k C)P_k^-$$

The prediction step is used to calculate a prior state estimate and the error covariance P_k. It

can be thought of as a measure of uncertainty in the estimated state. The process covariance matrix represents the error in the filter estimate and is calculated based on the previous P_{k-1} and the process noise covariance Q.[5] Estimation of the kalman gain K_k determines how much emphasis to put on the current predicted state estimate and current measurements and is derived from P_k and sensor noise covariance matrix R. The second step of the algorithm uses the estimates calculation in the prediction step and updates them to find the optimal state estimate and the error covariance. The Kalman gains K is calculated such that it minimizes the error covariance.

In this project, the state estimation vector will contain $(x, y, z, \psi, \theta, \phi)$ where x, y, z represents the coordinates in 3D Cartesian frame and ψ, θ, ϕ represents Yaw, Pitch and Roll respectively.

2.5. Navigation Method

The navigation stack is a set of algorithms that use the sensors of the robot and the odometry to control the robot using a standard message. It can move the robot without any problems, such as crashing, getting stuck in a location, or getting lost to another position.

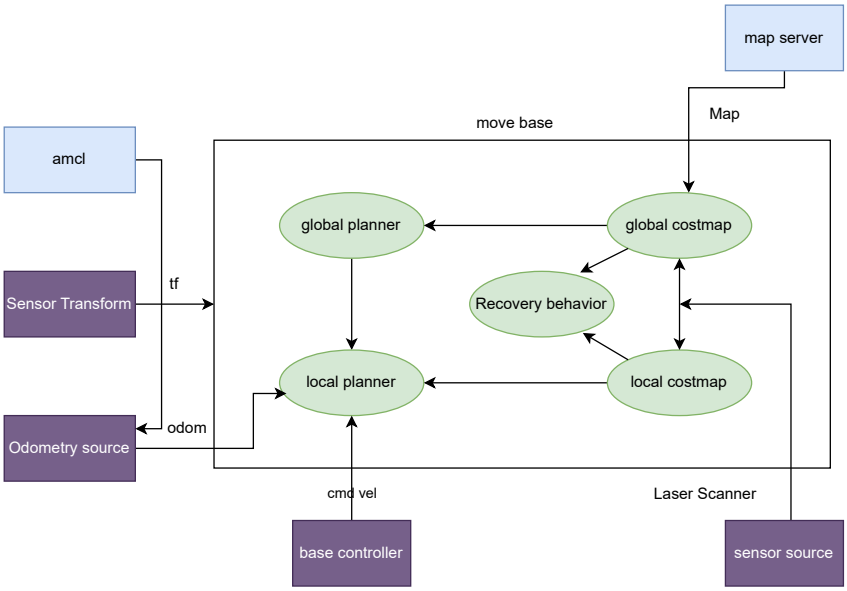

Figure 2.6: Navigation Stack in ROS

Figure 2.6 shows the diagram of navigation stacks which are organized beneficial to make the robot autonomously. In order to achieve the navigation task, the navigation stack is used to integrate the mapping, localization, and path planning together. It takes the information from the odometry, sensor streams, and goal position to produce safe velocity commands and send it to mobile robot. The odometry comes through the **nav_msgs/Odometry** message over ROS

which stores an estimate of the position and velocity of a robot in free space to determine the robot's location. The sensor information comes through **sensor_msgs/LaserScan** messages over ROS to avoid any obstacles. The goal is sent to the navigation stack by **geometry_msgs/ PoseStamped** message. The navigation stack sends the velocity command through **geometry_msgs/Twist** message on **/cmd_vel** topic[7]. The navigation stack does not require a prior static map to start with. Actually, it could be initialized with or without a map. When initialized without a prior map, the robot will know about the obstacles detected by its sensors only and will be able to avoid the seen obstacles so far. For the unknown areas, the robot will generate an optimistic global path which may hit unseen obstacles. The robot will be able to re-plan its path when it receives more information by the sensors about these unknown areas. Instead, when the navigation stack initialized with a static map for the environment, the robot will be able to generate informed plans to its goal using the map as prior obstacle information. Starting with a prior map will have significant benefits on the performance. This algorithm is known as SLAM which is used to get the prior map which helps for the navigation system in AMCL which uses this existing map to do path planning[7]. However, when the AMCL First starts up, we have to give it the initial pose (position and orientation) of the robot as this something AMCL cannot figure out on its own. To set the initial pose, there is a **2D Pose Estimate** button in rviz. Then click on the point in the map where the robot is located[4].

To build a map using ROS, ROS provides a package called Gmapping. A particle filter-mapping approach is used by the gmapping package to build an occupancy grid map. The package named **map_server** could be used to save that map. The maps are in a pair of files: YAML file and image file. The localization part is solved in the amcl package using an Adaptive Monte Carlo Localization which is also based on particle filter. It is used to track the position of a robot against a known map. The path planning part is performed in the **move_base** package, and is divided into **global and local planner** module which is a common strategy to deal with the complex planning problem[7].

As shown in **Figure 2.6** , there are three types of nodes: provided nodes, optional nodes and platform specific nodes.

2.5.1. Provided Nodes

Provided nodes are the nodes inside the box that is the core of the navigation stack and are responsible and managing the costmaps and for the path planning functionalities and these

nodes are the nodes inside the **move_base** package. **move_base** package also maintains two costmaps, **global and local costmap** to be used with the **global and local planners** respectively. The costmap used to store and maintain information in the form of an occupancy grid map about the obstacles in the environment and where the robot should navigate. The costmap initialized with the prior static map if available, then it will be updated using sensor data to maintain the information about the obstacles in the map. The local costmap use the obstacle that is detected by the Lidar scanner to compare with the global costmap in order to know the initial position of the robot in the map. Obstacle give a unique map that allows it to know where it is compared to the global map information. This algorithm does not work in an open area with no obstacle. Besides that, **move_base** also defined recovery behaviors when it fails to find a valid plan. Inside the move_base node there are[7]:

- **Global Planner:** The global planner in ROS operates on the **global_costmap**, which generally initialized from a prior static map, then it could be updated frequently based on the value of the **update_frequency** parameter. The **global planner** is responsible for generating a long-term plan from the start or current position to the goal position before the robot starts moving. It will be seeded with the costmap, and the start and the goal position.
- **Local Planner:** The local planner in ROS operates on the **local_costmap** which only uses local sensor information to build an obstacle map and updated with sensor data. It takes the generated plan from the global planner, and it will try to follow it as close as possible considering the frame of the robot as well as any moving obstacles information in the local_costmap.
- **Local and Global Costmaps:** The local and global 2D costmaps are the topics containing the information that represents the projection of the obstacles in a 2D plane, as well as a security inflation radius, an area around the obstacles that guarantee that the robot will not collide with any obstacle, no matter what it its orientation. These projections are associated with a cost, and the robot's objective is to achieve the navigation goal by creating a path with the least possible cost. While the global costmap represents the whole environment[7].

2.5.2. Optional Nodes

Optional Nodes are the **amcl and map_server** that are related to static map functions[7].

- **Map_server** contains two nodes: map_server and map_saver. The first one is a ROS node that provides static map data as a ROS service, while the second one, map_saver, saves a generated to a file.
- **amcl** does not manage the maps, it is actually a localization system that runs on a known map. It uses the base_link transformation to the map to work, therefore it needs a static map and it will only work after a map is created. This localization system is based on the Monter Carlo Localization approach. It randomly distributes particles in a known map, representing the possible robot locations, and then uses a particle filter to determine the actual robot pose.

2.5.3. Platform Specific Nodes

Platform Specific Nodes are the node related to the robot, such as sensor reading nodes and base controller nodes.

- **Sensor Transform**: The robot needs to know the position of the sensors, joints, and wheels, so the Sensor transform is the Transform Frame software library that manages a transform tree. So, we can add sensors and parts to the robot, and tf will handle all the relations for us.

  To broadcast the static TF of the sensors, we can use the **static_transform_publisher** in the launch file:

  "static_transform_publisher x y z yaw pitch roll fram_id child_frame_id period_in_ms"
- **Odometry Source** Odometry data is used to estimate the robot s position and orientation relative to its origin. The Odometry Source will subscribe the information from the Arduino (position, Orientation, and velocity) and publish the odom topic to the navigation stack, which has a message type of nav_msgs/Odometry.
- **Base Controller** ROS navigation stack drives the robot by publishing the velocity command using Twist message with /cmd_vel topic. The base controller on the Arduino subscribe to this topic through rosserial_python and convert it to velocity for the motor.
- **Sensor Source**: we need to provide the laser scan data to the navigation stack for mapping the environment. This data, along with the odometry, combines the global and local cost map of the robot.

2.5.4. SLAM Algorithm

Simultaneous Localization and Mapping (SLAM) is an algorithm which is used to build the map of the unknown environment as well as localize the robot at the same time, while the AMCL algorithms is used to localize the robot in an existing map. In order to localize the robot in the environment, we need to know the position and orientation of the robot when the robot starts and stops. The idea of SLAM node is that as the robot is moving around the environment, it will create a map of the environment using the laser scan data and the odometry data. Maps are used to plan the robot's trajectory and to navigate through this path. Using maps, the robot will get an idea about the environment. The two main challenged in mobile robot navigation are mapping and localization. Mapping involves generating a profile of obstacles around the robot. Through mapping, the robot will understand what the world looks like. Localization is the process of estimating the position of the robot relative to the map that the robot build. SLAM fetches data from the lidar scanner and uses it to build maps. The **gmapping** package provides a node to perform laser-based SLAM processing, called **hector_mapping**. This can create a 2D map from the laser and position data collected by the mobile robot.

2.5.5. AMCL Algorithm

Adaptive Monte Carlo Localization (AMCL) is an algorithm that is used to localize the robot in the existing map and navigate autonomously from the initial position to the target position. The AMCL algorithm is a probabilistic localization system for a robot moving in 2D. This system implements the Adaptive Monte Carlo Localization approach, which uses a particle filter to track the position of the robot with respect to the map. The ROS AMCL package provide a nodes for localizing the robot on a static map. The amcl node subscribes the laser scan data, laser scan based maps, and the tf information from the robot. The amcl node estimates the pose of the robot on the map and publishes its estimated position with respect to the map. When the **amcl** node is launched, it begins providing localization information on the robot based on the current 3D sensor scan (**sensor_msgs/LaserScan**), tf transforms(tf/tfMessage), and the Occupancy Grid Map. When a **2D Pose Estimate** is input by the operator, an initialpose message resets the localization parameter and reinitializes the amcl particle filter. As laser scans are reads, amcl resolves the data to the odometry frame. The amcl node provides the position estimation of the robot in the map, a particle cloud, and the tf transforms for odom.

2.6. Component Selection

2.6.1. Motor Encoder

A motor encoder is a rotary encoder mounted to an electric motor that provides closed-loop feedback signals by tracking the speed or position of a motor shaft. The encoders provide 2-channel quadrature, TTL square wave outputs which are used to determine which way the motor is rotating.

Table 2.1: Specification of Motor encoder

Parameter	Value
Motor Voltage	3-12V
Motor Current	120mA
Rated Power	2W
Rated Load	1.2kg.cm
Blocking Current	3.5A
Stall torque	8kg
No-load Speed	320rpm/min
Coding Accuracy	360CPR
Reduction Ratio	1:34
Encoder Power	3.3-5V

The motor encoder provides a 34 gear ratio with 360 counts per revolution. By 1 count per revolution equal to pulse per revolution divided by 4. So, it takes 49860 pulses to get one rotation of the motor shaft.

2.6.2. Inertial Measurement Unit

To get the information of velocity, acceleration, Euler angles and, quaternions value, I use two IMUs to publish this information to the robot. One is MPU6050 is used to construct the odometry message with the encoders of the motors and another one is the Android Sensor Driver which is used to fuse its data with an odemetry message from the encoders and MPU6050.

- **MPU6050** is an IMU that is combined with 3-Axis gyroscope and 3-Axis accelerometer. The gyroscope measures rotational velocity along the X, Y, and Z axis, and an accelerometer is used to measure the acceleration in the same way.

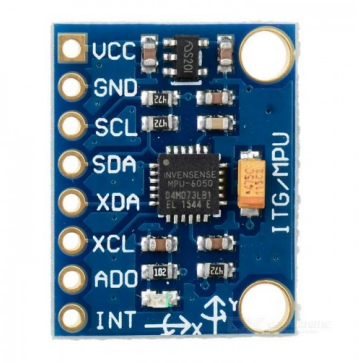

Figure 2.7: MPU6050

However, there are two ways to read the data from the MPU6050. One way is to read raw sensor data values of the gyroscope and accelerometer through the I2C bus. But the Gyroscope data has a tendency to drift with time, and accelerometer data is noisy. So, to minimize the effect of error, I use the second method is to pull the data out of the MPU's onboard Digital Motion Processor (DMP)[8]. The DMP is fused the accelerometer and gyroscope data together and computes the results in terms of quaternions.

- **Android Sensor Driver**

  As Android is one of the most operating systems in a mobile device, I find it necessary to add the android sensor driver to the robot to improve the odometry, since the odometry from wheel encoders are prone to error from voltage changes, wheel diameter, and wheel slip. I use an app called **ROS Sensor Driver** which provides the data accuracy of :

  - IMU
  - gps
  - magnetic field
  - illuminance

In this project, I subscribe only to the data of the android/Imu to fuse with the odometry from the wheel encoder and MPU6050. For the IMU in Android Sensor Driver, we can access the information of Quaternions, angular velocity, and linear acceleration which all of these data are accurate. However, there is a problem with the IMU driver where its frame_id is a string, so I cannot fuse these data directly with the odometry. So, I use ROS subscribe to the topic of IMU and publish them with the new frame_id call imu_datas. **Figure 2.8** show the frame of Android Imu in rviz.

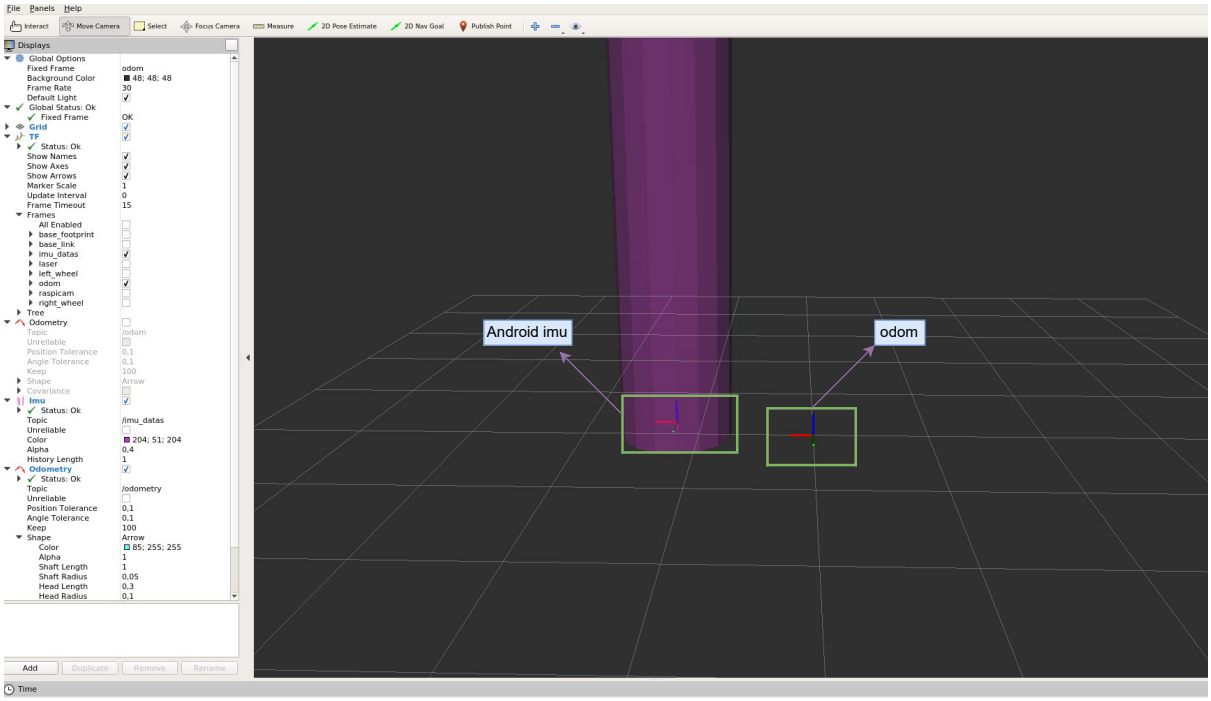

Figure 2.8: Android IMU Frame in rviz

2.6.3. Laser Scanner

To scan the map of the unknown environment, we need a sensor that can measure the distance between the robot and object. Lidar is a sensor that measures the distance to an object by sending a laser signal and receiving its reflection. It can provide accurate data of the environment, computed from each received laser signal. The main application of Lidar is mapping the environment, obstacle avoidance, object detection, and so on. In this project, I use Lidar Delta 2A which can scan perform SLAM and obstacle avoidance, but the position of the lidar is not accurate. It will go randomly anywhere in the environment. To get the accurate position, I need to merge it with the odometry of encoder and IMU.

Figure 2.9: Lidar Laser Scanner

The parameters and values of the Lidar sensor are shown as table below:

Table 2.2: Specification of Lidar Sensor

Parameter	Value
Range	0.13m-8m
Sweep Frequency	4-10Hz
Voltage	5V
Baud rate	230400
Sampling rate	5ksps
Working Current	500mA
Power consumption	1.5W
Working Mode	8 bit data, 1 stop bit, no parity check

Lidar has a type of sensor_msgs/LascerScan under the /scan topic. we can visuaulize the data of Lidar in rviz.

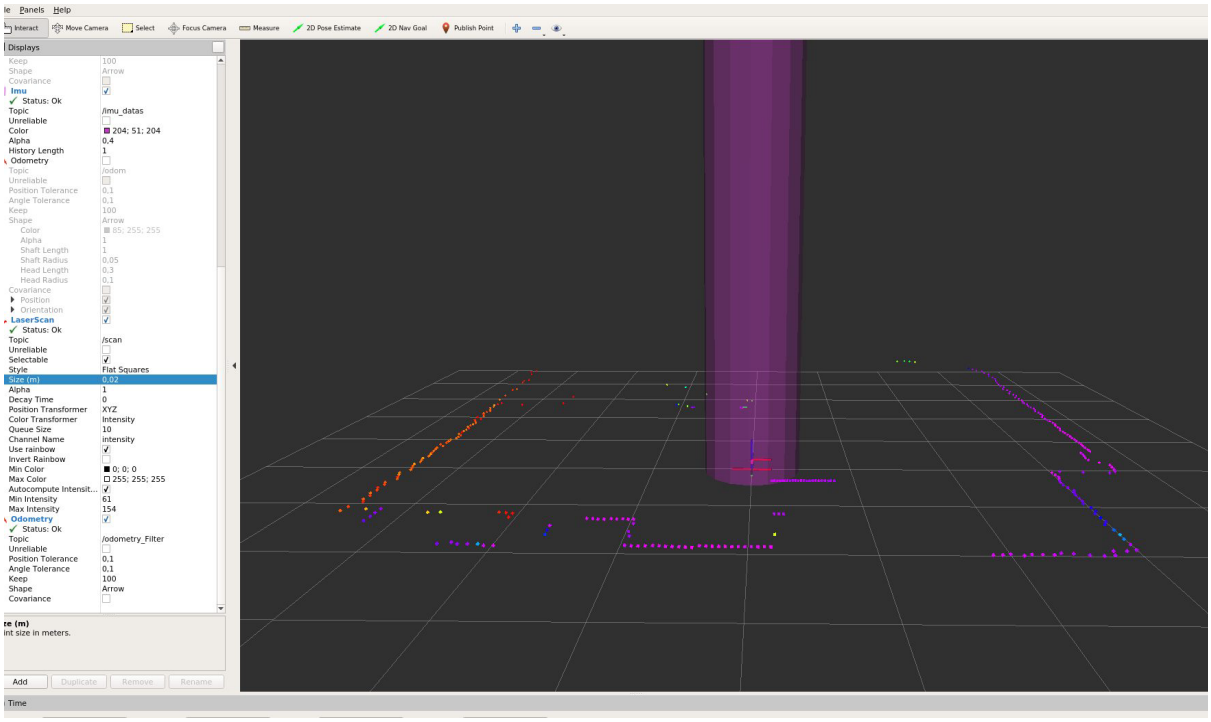

Figure 2.10: Laser Visualization in rviz

3. Implimentation and Verification of Kinematic Model

In this project, I have ROS installed on Raspberry Pi 4 with the Ubuiquity OS. All of the development is done within the Arduino IDE which includes the ROS library for Arduino plus some Arduino specific libraries. I use Arduino Due to read the value of encoders and the orientation of mpu6050 and send all of these data to Raspberry Pi 4 through Rosserial.

3.1. Implementation of Forward Kinematic Model

```
float calculateVx(float v1, float v2, float v3, float v4,
   float a, float b){
 return (1.0/4.0)*(v1+v2+v3+v4);
}
float calculateVy(float v1, float v2, float v3, float v4,
   float a, float b){
return (1.0/4.0)*(-v1+v2+v3-v4);
}
float calculateOmega(float v1, float v2, float v3, float v4,
   float a, float b){
return (1.0/((a+b)*4.0))*(-v1+v2-v3+v4);
}
```

The code above shows the implementation of the forward kinematic model into the Arduino code. When I get the speed of each motor from the motor encoders, and then I can convert it to the speed of the robot in x, y direction and angular velocity ω.

```
Vx=calcalateVx(v1, v2, v3, v4, a, b);
Vy=calculateVy(v1, v2, v3, v4, a, b);
Omega=calculateOmega(v1, v2, v3, v4, a, b);
```

I use three variables V_x, V_y, and *Omega* to represent the speed of the robot in x-direction, y-direction and angular velocity ω, respectively.

```
x += Vx*dt/1000.0;
y += Vy*dt/1000.0;
theta=orient.z;
```

When I integrate the speed of each direction, then I can get the position when the robot moves. However, for the *theta* that is the angle of the robot, I use it equal to $orient.z$ which is the yaw angle access from the MPU6050.

3.2. Implementation of the Inverse Kinematic Model

```
float calculateV1(float vx, float vy, float Omega, float a,
   float b){
return (vx-vy-(a+b)*Omega);
}
float calculateV2(float vx, float vy, float Omega, float a,
   float b){
return (vx+vy+(a+b)*Omega);
}
float calculateV3(float vx, float vy, float Omega, float a,
   float b){
return (vx+vy-(a+b)*Omega);
}
float calculateV4(float vx, float vy, float Omega, float a,
   float b){
return (vx-vy+(a+b)*Omega);
}
```

The code above shows the implementation of the inverse kinematic model into the Arduino code. With the speed of the robot, and then I can convert it into the speed of each motor.

```
ros::Subscriber <geometry_msgs::Twist> sub("/cmd_vel",
    handle_cmd);
void handle_cmd(const geometry_msg::Twist& msg){
  linear_x=msg.linear.x;
  linear_y=msg.linear.y;
  ang_z=msg.angular.z;
  v_target1=calculateV1(linear_x, linear_y, ang_z, a, b);
  v_target2=calculateV2(linear_x, linear_y, ang_z, a, b);
```

```
    v_target3=calculateV3(linear_x, linear_y, ang_z, a, b);
    v_target4=calculateV4(linear_x, linear_y, ang_z, a, b);
```

The code above, I use Arduino to subscribe the topic **cmd_vel** from ROS and return back with **handle_cmd**. Then I can get linear velocity in x and y direction and the angular velocity ω. So, I use the function of the inverse kinematic to convert them into the velocity of each motor.

3.3. Verification of Kinematic Model

When I publish the data of the speed and position from the encoder and MPU6050 then I can see the information on speed and position of the robot. In ROS, I use the **teleop** node which publish the **/cmd_vel** topic where data come from the computer keyboard.

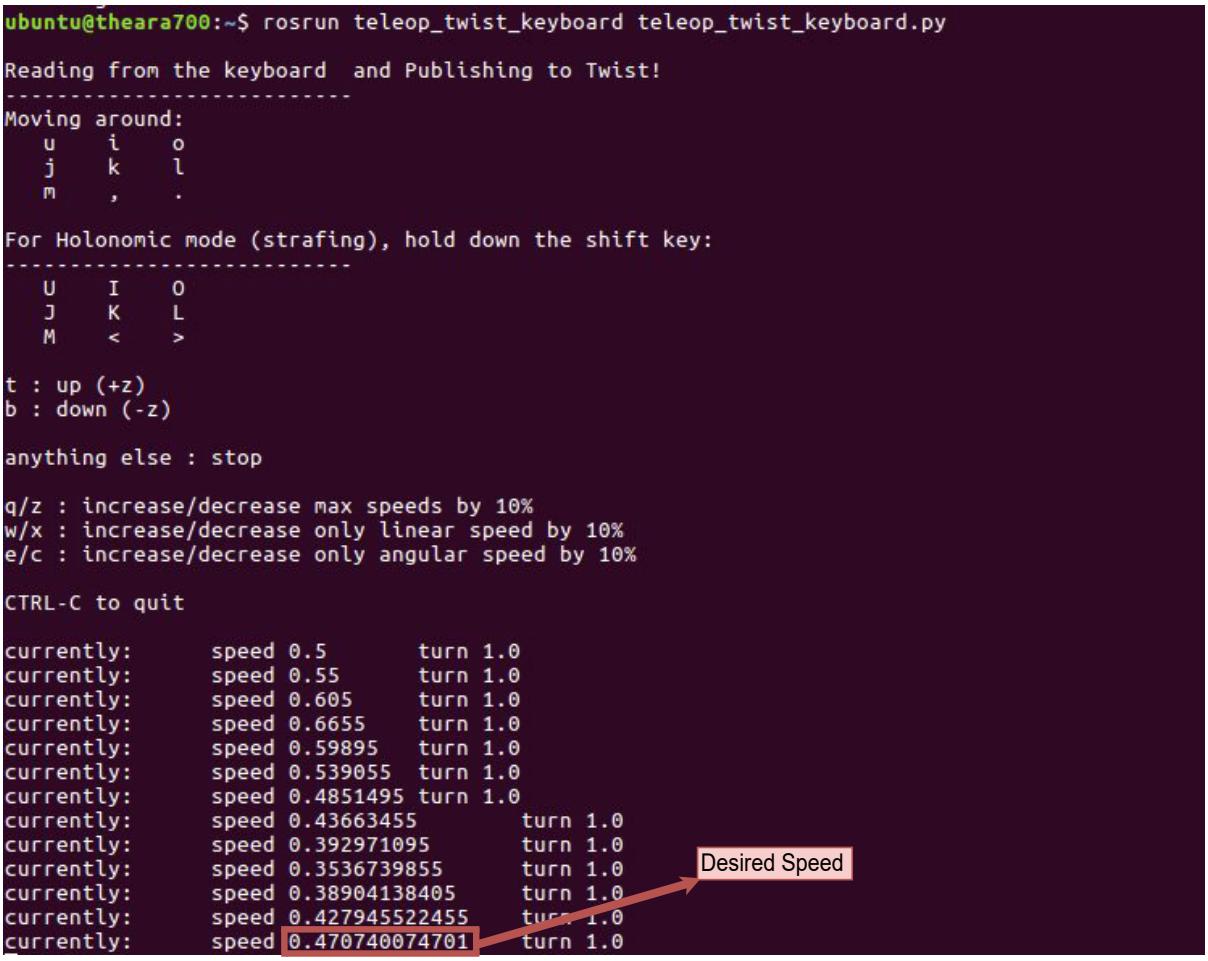

Figure 3.1: Desired Speed in x and y direction

As shown in **Figure 3.1**, I used computer keyboard to publish the speed of x and y direction. **Figure 3.2** shows the information of the robot in x-direction. In this case, I press the **i** button

and set the speed of $0.4707 m/s$, then I can see the actual speed of $0.4778 m/s$ of the robot with the position of $14.50 m$.

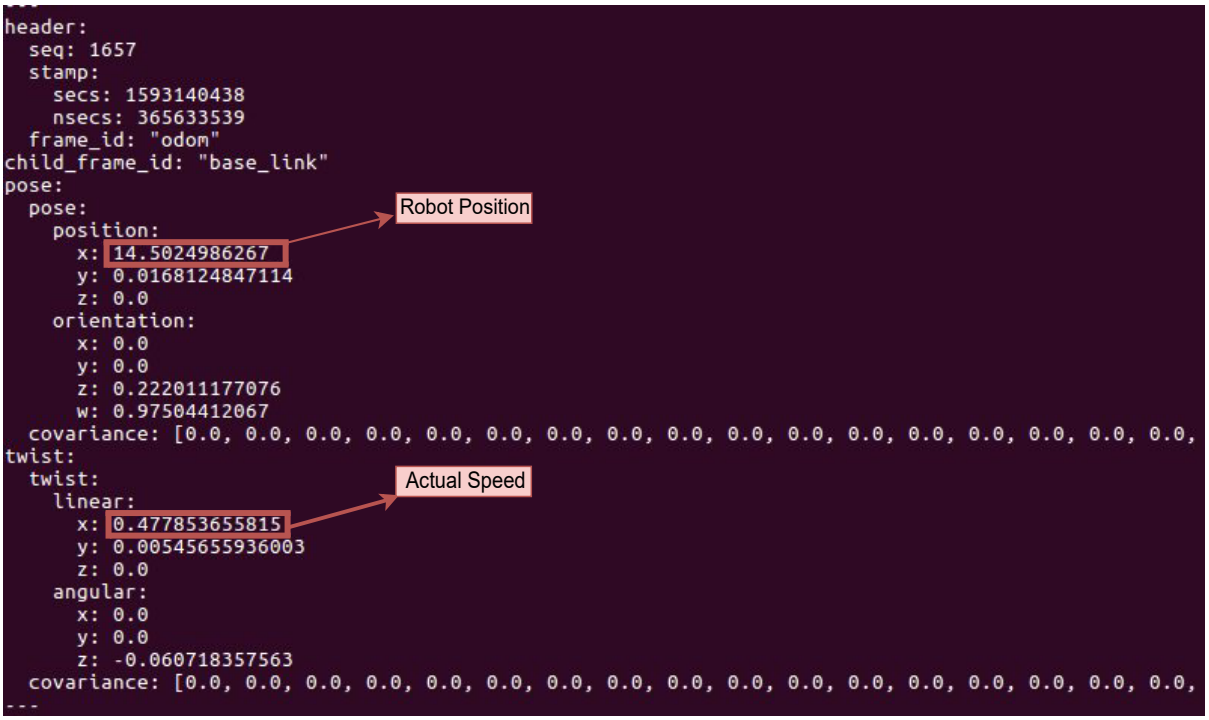

Figure 3.2: Robot in x-direction

Figure 3.3: Robot in y-derection

Figure 3.3 shows speed and position in y-direction. In this case, I press the **J** button which

desires the robot in the y-direction with the speed the same as x-direction. Then I can see the actual speed of the robot that is $0.4785 m/s$ with the position in the y-direction of $2.52m$.

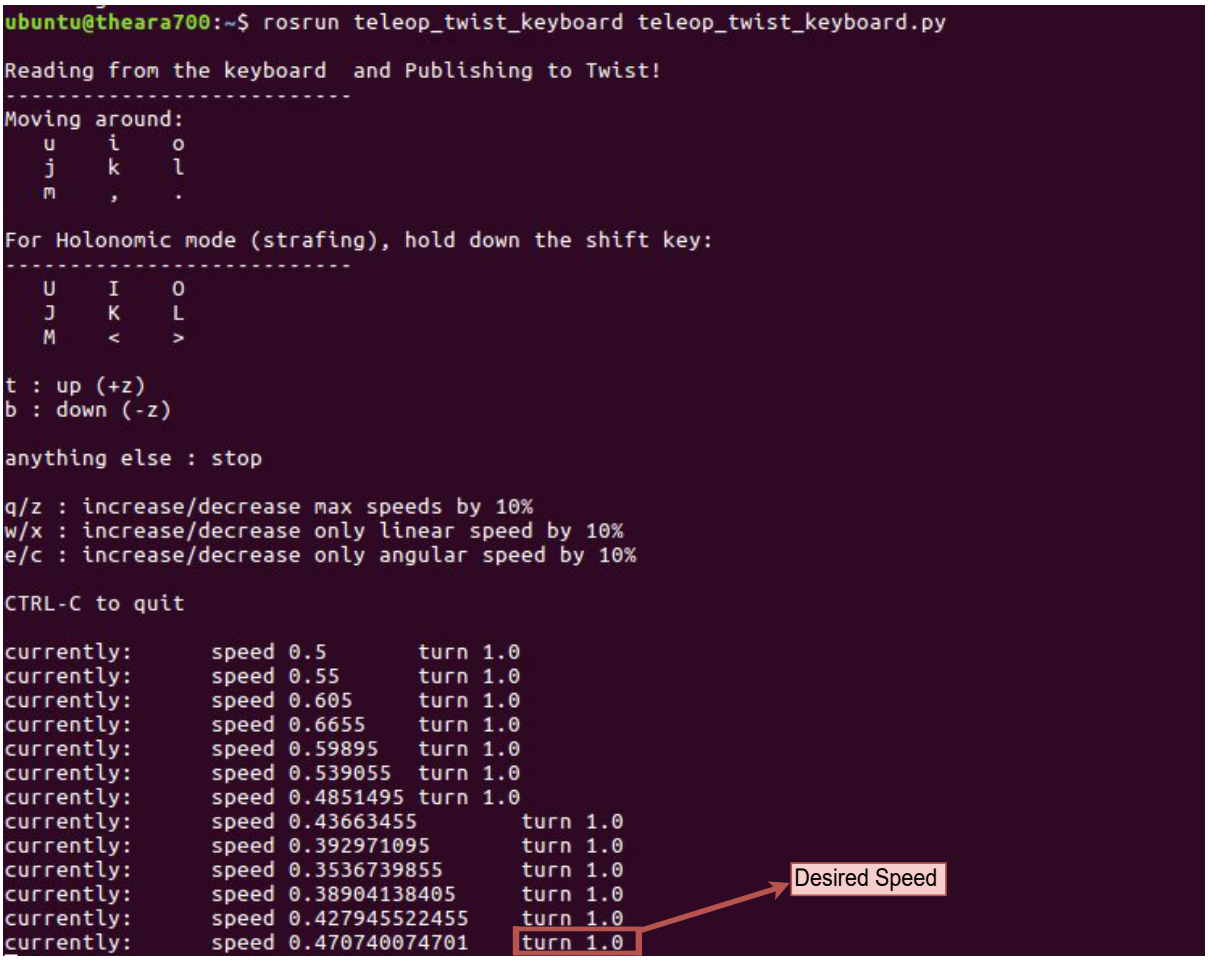

Figure 3.4: Desired Speed in θ-direction

In **Figure 3.4**, the desired speed for ω angular velocity is set by the computer keyboard. For this, I use the keyboard key with **j** letter to rotate the robot with an angle of θ and the angular velocity of $1 m/s$, and then **Figure 3.5** displays the speed and angle of the robot in θ-direction. we can see the actual speed of $0.993 m/s$ and the angle of the robot display in quaternion values. This proves that the kinematic model that is written in the mathematic model is correct.

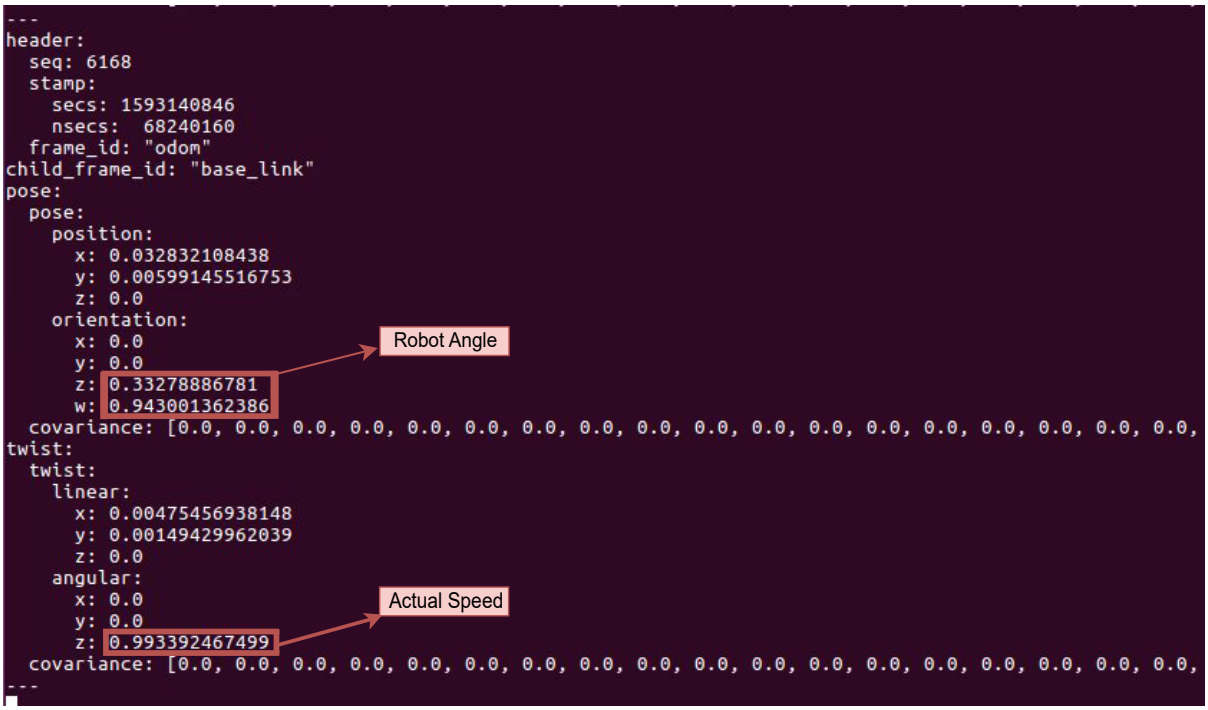

Figure 3.5: Robot in θ-direction

4. Position Estimation

To improve the robot position estimation, I am using multiple sensors to estimate them. In ROS, there is a package called **robot_localization** which is used as an Extended Kalman Filter to calculate the new estimation using data from sensors. we need to install this package beneficial to fuse these data.

To install this package, we can use the command:

$ sudo apt-get install ros-kinetic-robot-localization

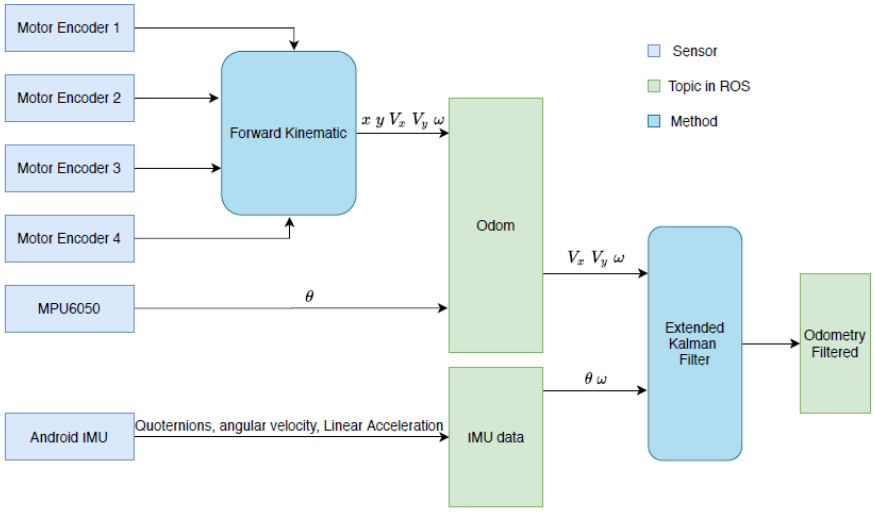

Figure 4.1: Position Estimation Method

Figure 4.1 shows the method of position estimation in this project. There are three topics such as **Odom, IMU data, and Odometry Filtered** which are used to estimate the position of the robot. The odom topic gets the data from the encoders and the MPU6050, while the IMU_data topic is the data from the Android IMU. These data fuse together using EKF to get the Odometry Filtered topic.

4.1. Odom

Odom topic has a message type of **nav_msgs/Odometry** which needs the information of position, orientation, linear velocity, and angular velocity. we need to send this information to the navigation stack. The odometry is the distance between the base_link of the robot to a fixed point in the frame odom.

Figure 4.2 show the infomation of nav_msgs/Odometry message.

$ rosmsg show nav_msgs/Odometry

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Figure 4.2: Odometry Message

We see that in the Odometry message needs the information of frame_id, child_frame_id, position, orientation, linear velocity, and angular velocity for the robot. The **frame_id** in this message is odom and **child_frame_id** is base_link which is connected to all the wheels. In this Odometry message, there are **geometry_msgs/Pose** which needs the detail of position and orientation and **geometry_msgs/Twist** which needs the information of linear and angular velocity.

So, I use Arduino to read the data from wheel encoders and MPU6050 and publish them to the Odometry message.

The wheel encoders provide the details:

- position x
- position y
- linear velocity x

- linear velocity y
- anglular velocity ω

The MPU6050 provides the infomation:

- Quaternion orientation

I write a C++ code in Raspberry Pi 4 to subscribe the data from Arduino.

```
ros::Subscriber speed=nh.subscribe("Speed", 50,
    speedCallback);
ros::Subscriber position=nh.subscribe("Position", 50,
    positionCallback);
void speedCallback(const geometry_msgs::Vector3Stamped&
    speed_msg){
vx=speed_msg.vector.x;
vy=speed_msg.vector.y;
vth=speed_msg.vector.z;
}
void positionCallback(const geometry_msgs::Vector3Stamped&
    position_msg){
x=position_msg.vector.x;
y=position_msg.vector.y;
th=position_msg.vector.z;
}
```

I use the geometry_msgs with the Vector message to subscribe to the **speed_msg** message and **position_msg** message that I publish from the Arduino. Then I use variables as shown above to represent each speed and position direction.

```
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
geometry_msgs::Quaternion odom_quat=tf::
    createQuaternionMsgFromYaw(th);
ros::Publisher odom_pub=nh.advertise<nav_msgs::Odometry>("
    odom",50);
nav_msgs::Odometry odom;
```

```
odom.header.frame_id="odom";
odom.child_frame_id="base_link"
odom.pose.pose.position.x=x;
odom.pose.pose.position.y=y;
odom.pose.pose.position.z=0;
odom.twist.twist.linear.x=vx;
odom.twist.twist.linear.y=vy;
odom.twist.twist.angular.z=vth;
odom.publish(odom);
```

To publish the information about odom, I use two ROS libraries. One is the **tf/transform_broadcaster** library which is used to handle the transform frame from odom to base_link as well as convert from yaw rotation to quaternion. Another one is the **nav_msgs/Odometry** library which is used to publish the information of odometry. We need to provide the information of **header.frame_id** and **child_frame_id** and tf will handle it for us. The odometry message access data which subscribe from the Arduino to publish them with the **odom** topic. **Figure 4.3** shows the output of the odom data

```
---
header:
  seq: 459
  stamp:
    secs: 1592808310
    nsecs: 649531259
  frame_id: "odom"
child_frame_id: "base_link"
pose:
  pose:
    position:
      x: 18.113363266
      y: 0.122152857482
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.00349470005555
      w: 0.999993893517
  covariance: [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,
twist:
  twist:
    linear:
      x: 0.511341333389
      y: 0.0029885917902
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: -0.0262285023928
  covariance: [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,
---
```

Figure 4.3: Odom ouptut data

4.2. IMU data

To get the information of the IMU, I use an Android phone to publish the data. The data which we can access from the Android IMU are:

- Quoternions value
- angular_velocity
  - angular velocity in x direction
  - angular velocity in y direction
  - angular velocity in z direction
- linear_acceleration
  - linear acceleration in x direction
  - linear acceleration in y direction
  - linear acceleration in z direction

The IMU has a message type of **sensor_msgs/Imu**. To subscribe to the data from the Android IMU, I write a C++ code in Raspberry Pi 4 as shown below.

```
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>

void imuCallback(const sensor_msgs::Imu& imu_msgs){
sensor_msgs::Imu imu;
imu.header.stamp = current_time;
imu.header.frame_id = "base_link";
imu.orientation.x = imu_msgs.orientation.x;
imu.orientation.y = imu_msgs.orientation.y;
imu.orientation.z = imu_msgs.orientation.z;
imu.orientation.w = imu_msgs.orientation.w;
imu.angular_velocity.x = imu_msgs.angular_velocity.x;
imu.angular_velocity.y = imu_msgs.angular_velocity.y;
imu.angular_velocity.z = imu_msgs.angular_velocity.z;
imu.linear_acceleration.x = imu_msgs.linear_acceleration.x;
imu.linear_acceleration.y = imu_msgs.linear_acceleration.y;
```

```
imu.linear_acceleration.z = imu_msgs.linear_acceleration.z;
imu_pub.publish(imu);
}
int main(int argc, char** argv){
ros::init(argc, argv, "IMU_publisher");
ros::NodeHandle nh;
ros::Publisher imu_pub= nh.advertise<sensor_msgs::Imu>("
   imu_datas", 50);
ros::Subscriber imu_sub = nh.subscribe("/android/imu",10,
   imu_datacallback);
ros::spin();
return 0;
}
```

The code above, I use **ros/ros** and **sensor_msgs/Imu** library to subscribe the data from the android IMU. I use a node call **IMU_publisher** to subscribe to the topic called **/android/imu** and return with the callback function called **imu_datacallback** which is used to access information from the Android IMU. Then I use an object **imu** from the library of **sensor_msgs/Imu** to get the information from the android imu such as Quaternion values, angular velocity, and linear acceleration and declare its header_frame_id with the new name called **base_link**. After that, I create another topic called **imu_datas** to publish all of the information. Then we can see the data as shown in **Figure 4.4**

```
header:
  seq: 2733
  stamp:
    secs: 1591107754
    nsecs: 894000000
  frame_id: "/imu"
orientation:
  x: 0.0077521703206
  y: -0.00464093266055
  z: 0.957070589066
  w: 0.289714187384
orientation_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
angular_velocity:
  x: -0.00242592766881
  y: -0.000936265394557
  z: -0.0013555274345
angular_velocity_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
linear_acceleration:
  x: 0.182056367397
  y: -0.0287457425147
  z: 9.5723323822
linear_acceleration_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```


Figure 4.4: Android IMU

4.3. Odometry Filtered

To get the Odometry Filtered topic, I fuse the data of IMU from the android IMU and the odom from the motor encoders and MPU6050. The **robot_lacalization** has a node called **ekf_localization_node** which is used the Extended Kalman Filter to estimate the position of the robot. We need to specify in the robot_localization node for what sensors which need to be fused. Now, from those sensors, it must be specified what variables of those messages are going to be fused to compute the final state estimate. To specify that, we must fill a 3×5 value matrix.

The matrix means the following:

$$\begin{bmatrix} X, & Y, & Z \\ roll, & pitch, & yaw \\ \frac{X}{dt}, & \frac{Y}{dt}, & \frac{Z}{dt} \\ \frac{roll}{dt}, & \frac{pitch}{dt}, & \frac{yaw}{dt} \\ \frac{X}{dt^2}, & \frac{Y}{dt^2}, & \frac{Z}{dt^2} \end{bmatrix}$$

we must specify one of those matrices for each sensor. The matrix is a matrix of **true** and **false** value. True means to be taken into account in the estimate, and false for not to be taken. In this

project, there are IMU data and the odom needs to specify the variable which needs to be fused and the topic of the sensor.

- **odom topic**

$$\begin{bmatrix} false,, & false, & false \\ false, & false, & false \\ true, & true, & false \\ false, & false, & true \\ false, & false, & false \end{bmatrix}$$

- **imu_datas topic**

$$\begin{bmatrix} false, & false, & false \\ false, & false, & true \\ false, & false, & false \\ false, & false, & true \\ false, & false, & false \end{bmatrix}$$

Figure 4.5: Testing Position Error Estimation of the Robot

The matrix above shows the data of **odom topic** and the **imu_datas** topic which I take its variable to fuse. **odom topic**, I choose the linear velocity of x and y direction and the angular velocity ω, while the topic of **imu_data**, I take only the data of yaw rotation and angular velocity ω. With that, I do a run test in the lab to see the performance of the odometry. I do it by starting the robot in a position and making move to the exact position as shown in **Figure 4.5**. I use the computer keyboard to publish the **/cmd_topic** and convert it to velocity for the robot. **Figure 4.6** shows the graph of the robot movement from the starting position and move around the table and return to the exact same position.

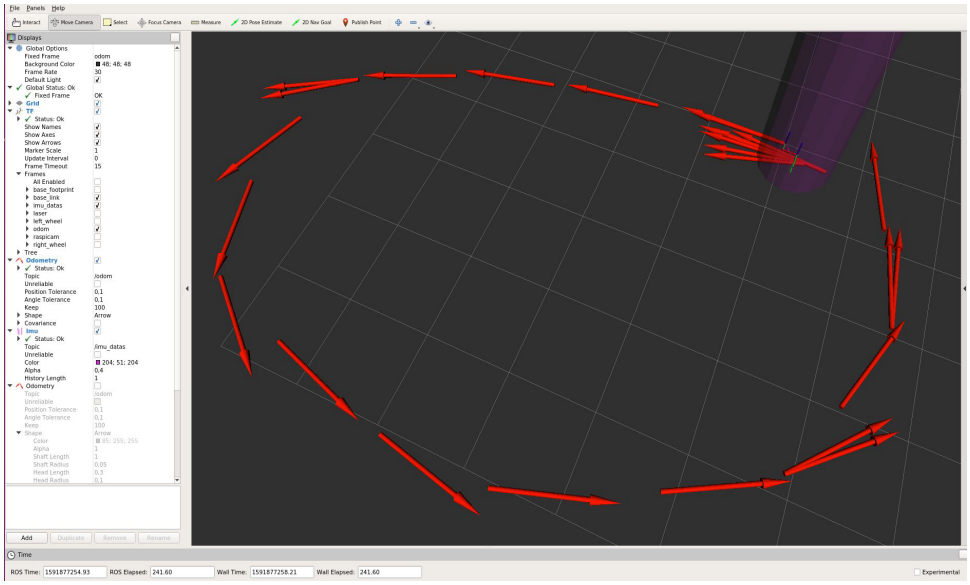

Figure 4.6: Postion Error Estimation in rviz

By observing the above figure, we can see the error in the start position and the end position for the different runs is approximately to be around 17cm. From the above error estimations, we can deduce that the EKF localization is providing a better result. The distance from the start and end position of the robot is much lower compared to the dead reckoning error.

5. SLAM Algorithm

5.1. Transform Frame of SLAM Algorithm

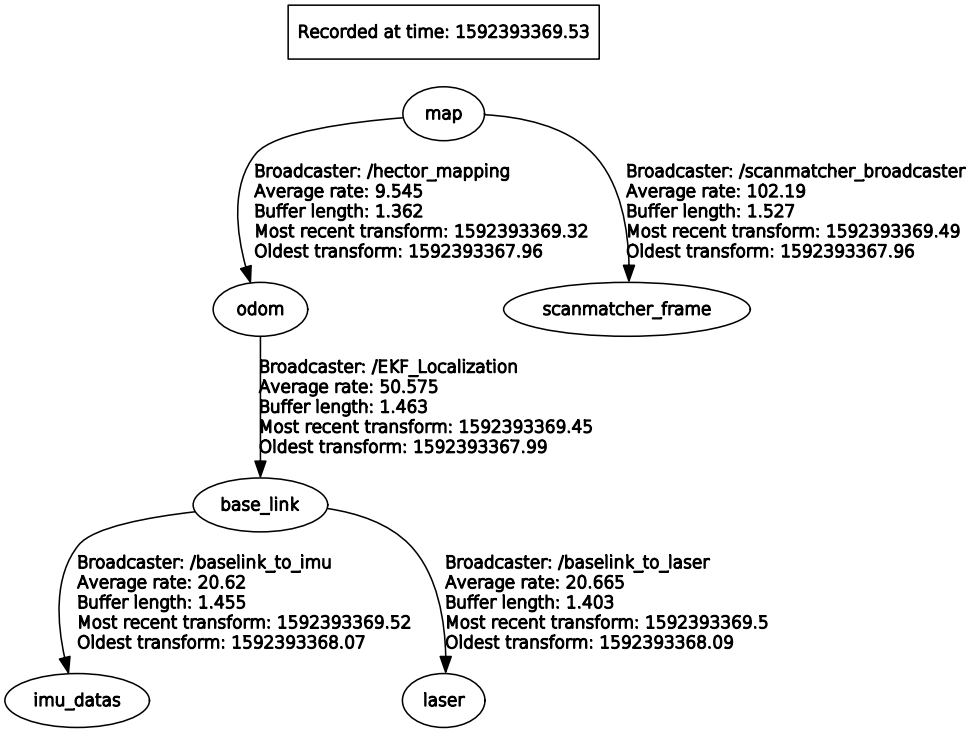

Figure 5.1: SLAM Transform Frame

Figure 5.1 shows the transform frame of the SLAM algorithm that is used to construct the map of the unknown environment. There are three transform frame to build the map which are **base_link -> sensor odom -> base_link and the map -> odom and scanmatcher_frame**

- **base_link to sensor**

  First, we need to know the position as well as the orientation of the sensors in the robot. In the project, there are two sensors that I have to call the transform frame to broadcast the information.

  – Information of the IMU: The IMU is 20cm from the base_link, so I can call the tf to broadcast the information by using:
  <node pkg="tf" type="static_transform_publisher" name="baselink_to_imu" args="0 0 0.2 0 0 0 base_link imu_datas 50"

  – Information of the Lidar Scanner: The Lidar is 20cm above, 10cm backward and -90 degree from the base_link, so the tf of the Lidar scanner is:
  <node pkg="tf" type="static_transform_publisher" name="baselink_to_laser" args="-

0.1 0 0.2 -1.57 0 0 base_link laser 50"/>

- **odom to base_link**

  base_link is the robot body coordinate system, with the center of the base as the center of rotation of the robot, while the odom is a fixed world frame system which is also the header frame_id of the base_link that is the child_frame_id.

- **map to odom**

  The laser is used to scan the map of the environment, but its odom is drift over time, so I need to merge it with the odometry of the robot. The **hector_mapping** is a node for lidar-based SLAM that helps to create a 2D occupancy grid map from the data of the laser scanner and the odometry position of the robot. The **hector_mapping** subscribe to the message data called LaserScan and the tf data, and publishes the occupancy grid map as the output.

To run the node the of the **hector_mapping**, we can the lauch file:
<node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="-screen"> The parameters which are used in the **hector_mapping** is shown in **Table 5.1**

Table 5.1: Parameters Configurations of the hector_mapping

Parameter	Value	Parameter	Value
map_frame	map	map-multi_res_levels	2
base_frame	base_link	laser_min_dist	0.4
odom_frame	odom	laser_max_dist	7.5
map_resolution	0.05	map_pub_period	2s
map_size	2048	update_factor_free	0.4
map_start_x	0.5	update_factor_occupied	0.7
map_start_y	0.5	map_update_distance_thresh	0.1
laser_z_min_value	-1.0	map_update_angle_thresh	0.025
laser_z_max_value	1.0	scan_topic	scan

5.2. rqt_graph of SLAM Algorithm

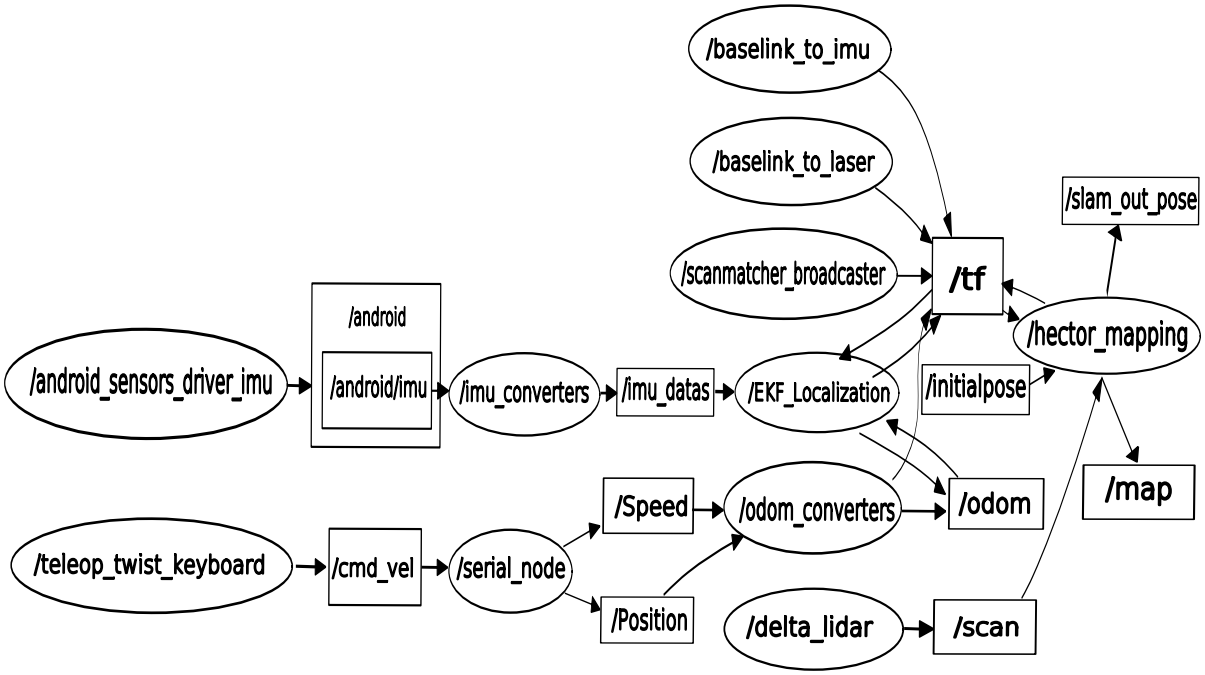

Figure 5.2: Nodes and Topics of SLAM Algorithm

Figure 5.2 shows the topics and nodes which are used to construct the map. First, we see that the **/android_sensor_driver_imu** node publishes the topic called **/android/imu**, and then **/imu_converters** node subscribes to that topic and publish new topic called **/imu_data**. The **/teleop_twist_keyboard** node is the node which uses computer keyboard to publish information. This node publishes a topic called **/cmd_vel**, and then the **/serial_node** (arduino) subscribes to this topic to control the speed of motors and publishes the information of speed and position of the robot. The **/odom_converters** node subscribes to the information of speed and position of the robot and publishes the /odom topic. This **/odom** topic fuses together with **/imu_datas** to get another /odom topic which is the odometry of the robot. The **/delta_lidar** node publishes the **/scan** topic to the **/hector_mapping** node. tf is the transform frame which handle all the relations such as **/baselink_to_imu** node, **/baselink_to_laser** node, **/scanmatche r_broadcater**, **/EKF_Localization, /odom_converter**. Then the **/hector_mapping** node access to all of these information to publish the information of the robot as well as the map. When running the node of the **hector_mapping**, we can see the frame of the odom, base_link as well the Laser Scanner in rviz as show in **Figure 5.3**.

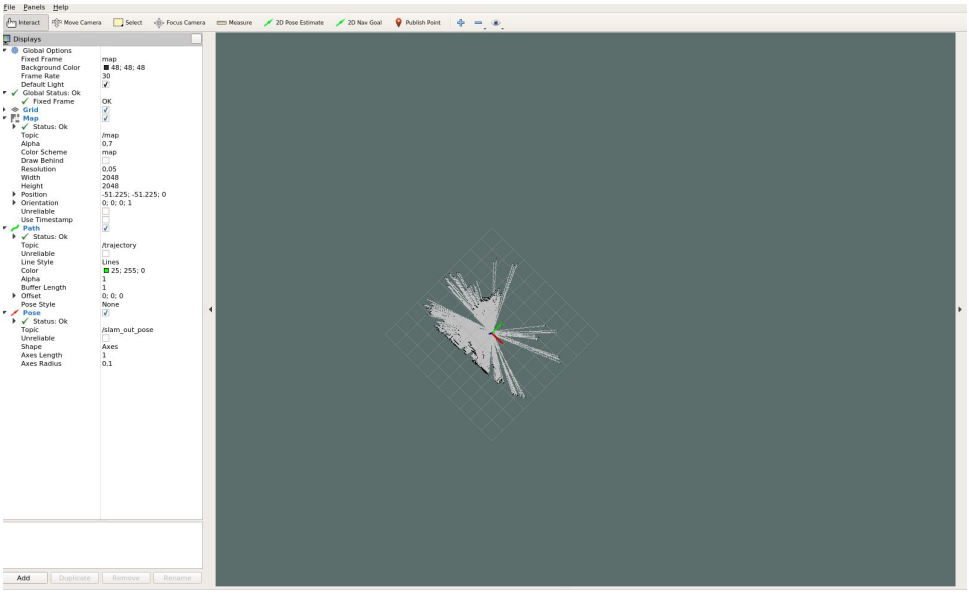

Figure 5.3: Simultaneus Localization and Mapping

The laser scanner cannot scan the whole environment at the start, we have to move the robot around to scan it. To move the robot around, I'm using the computer keyboard which publishes the **/cmd_vel** topic and the Arduino subscribes to this topic and converts it to velocity for the purpose to control the motor. When the robot move around, we will see the free and unknown space on the rviz screen, as well as the map.

After that, we can save the map using the map_server by the following command:

$ rosrun map_server map_saver -f map

This command will create two files, *map.pgm* and *map.yaml*

- **map.pgm** shows the origins of the robot in the map as well as the resolution.

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

Figure 5.4: pgm File of the Map

- **map.yaml** shows the 2D occupancy grid map of the environment.

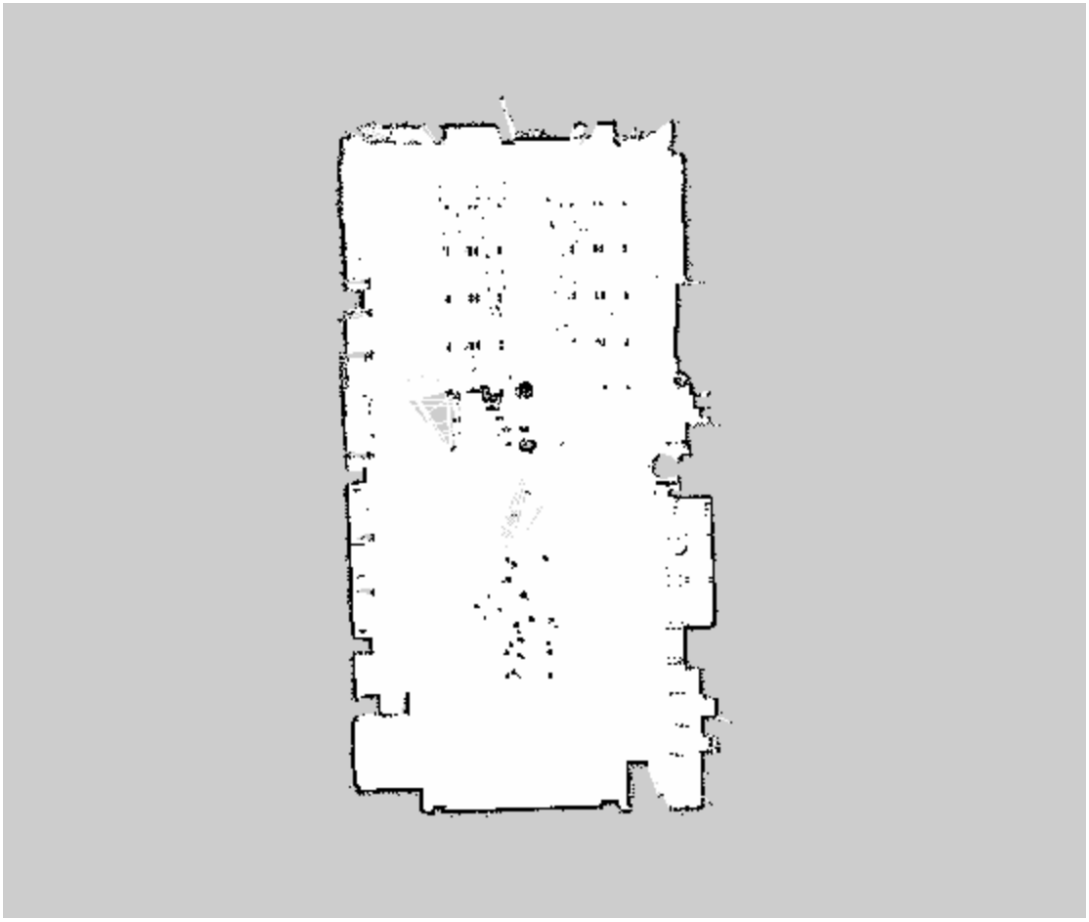

Figure 5.5: Occupancy Grid Map for Testing

6. AMCL Algorithm

6.1. Transform Frame of AMCL Algorithm

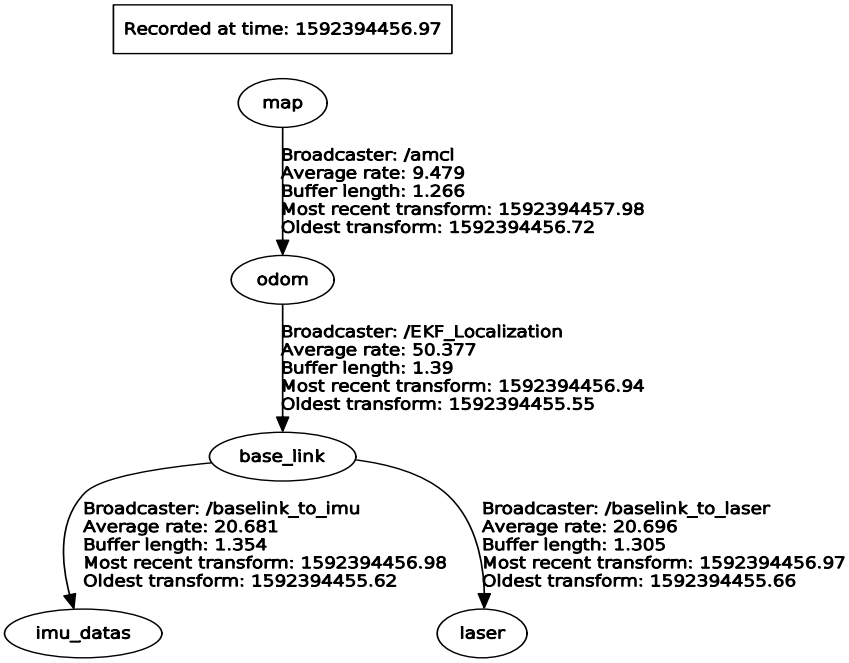

Figure 6.1: AMCL Transform Frame

We need to use the map_server package to load a static map that we saved in SLAM and publishes it into a topic called map. After loading the map, we need to configure some parameters in the amcl node and move_base node. **Figure 6.1** shows the transform frame of the AMCL which is used for autonomous navigation. The map frame is connected to the odom through amcl node which helps to localize the robot on the current map. To run the node the of the **amcl**, we can use the launch file:

<node pkg="amcl" type="amcl" name="amcl" output="screen">

The parameters which are used in the **amcl** node is shown in **Table 6.1**.

Table 6.1: Parameters Configurations of the amcl

Parameter	Value	Parameter	Value
base_frame_id	base_link	odom_frame_id	odom
kld_err	0.05	odom_model	omni
kld_z	0.99	laser_z_hit	0.5
laser_lambda_short	0.1	laser_z_short	0.05
laser_likelihood_max_dist	2.0	laser_z_max	0.05
laser_max_beams	60	laser_z_rand	0.5
laser_sigma_hit	0.2	odom_alpha1	0.25
recovery_alpha_slow	0.001	odom_alpha2	0.25
recovery_alpha_fast	0.1	odom_alpha3	0.25
resample_interval	1	odom_alpha4	0.25
transform_tolerance	1.25	odom_alpha5	0.1
max_particles	2000	update_min_a	0.2
min_particles	500	update_min_d	0.2

To get the robot move autonomously from position A to position B in the environment, the robot needs a map, a localization module, and a path planning module. The robot can safely and autonomously navigate the environment if the map completely and accurately defines the environment. To test the path planning of the robot, I use the SLAM algorithm to get the map of the environment as shown in **Figure 5.5**.

6.2. rqt_graph of the AMCL Algorithm

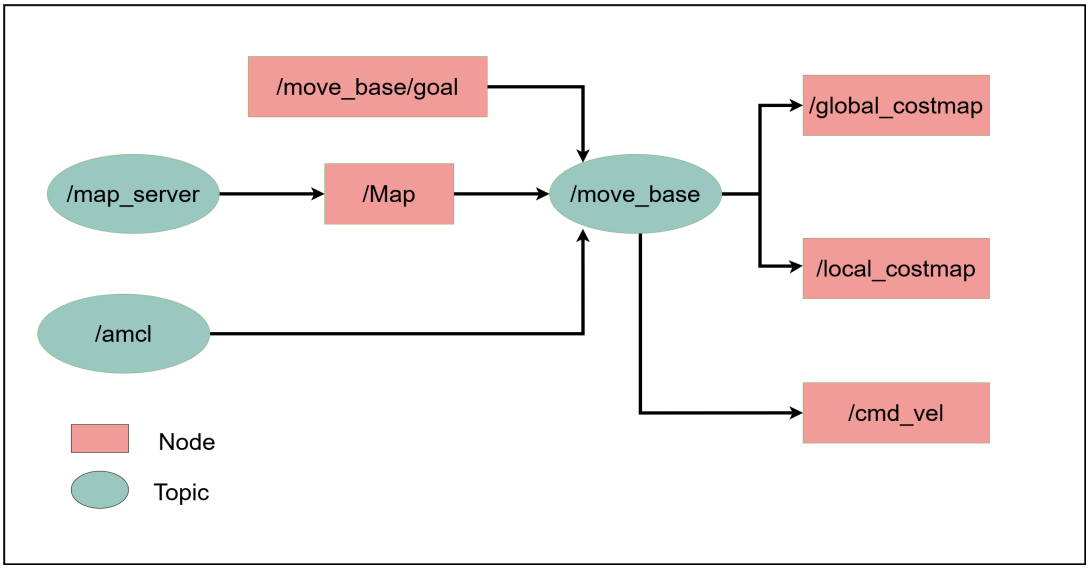

Figure 6.2: Nodes and Topics of the Navigation in ROS

Figure 6.2 shows the nodes and topic algorithm for the navigation stack of the robot. As shown in the graph, when we get the map from the SLAM algorithm, we can use the **map_server** node to publish the **map** topic to the **move_base** node. **amcl** node use **particleloud** which data come from the odometry of the robot to estimate the position of the robot. The main component of the robot navigation is the **move_base** node. This node performs the task of commanding the robot to make an attempt to reach the goal location. This task is set as a preemptable action based on its implementation as a ROS action and the robot's progress toward the goal is provided as the feedback. The **move_base** node uses a global and a local planner to accomplish the task. Two costmaps, **global_costmap** and **local_costmap** are also maintained for the planners by the **move_base** node.

- **Global costmap:** This costmap keeps information for global navigation. Global costmap parameters control the global navigation behavior. These parameters are stored in *global_costmap_params.yaml*. Parameters common to global and local costmaps are stored in *costmap_common_params.yaml*.
- **Local costmap:** This costmap keeps information for local navigation. Local costmap parameters control the local navigation behavior and are stored in *local_costmap_params.yaml*.

After that, the **move_base** node will publish the **cmd_vel** topic to control the velocity of the robot to the target position.

6.3. Configuration of the move_base node YAML file

The configuration in **move_base** basically consists of four files where we need to write our code to provide for the robot. The files are as follow:

- base_local_planner_params.yaml
- costmap_common_params.yaml
- global_costmap_params.yaml
- local_costmap_params.yaml

6.3.1. Base Local Planner Parameter

The base planner is used to generate the velocity commands to move the robot. In this planner, I create a file called **base_local_planner_params.yaml**, and add the following code:

```
TrajectoryPlannerROS:
    max_vel_x: 0.7
    min_vel_x: 0.3
    max_vel_y: 0.7
    min_vel_y: 0.3
    max_vel_theta: 1.0
    min_vel_theta: -1.0
    min_in_place_vel_theta: 0.5
    escape_vel: -0.4
    acc_lim_x: 5.0
    acc_lim_y: 5.0
    acc_lim_th: 6.0
    holonomic_robot: true
```

The above parameter, I set the minimum and maximum speed of x and y from $0.3m/s$ to $0.7m/s$, while the maximum and minimum rotational velocity is $1rad/s$ and $-1rad/s$ respectively. And the minimum rotational velocity allowed for the base while performing in place is $0.5rad/s$. I use the escape velocity when detecting an obstacle is $-0.4m/s$. In this case, the robot will move back and create another path. For the acceleration of the robot, I set the acceleration of $5m/s^2$ for the x and y direction, and $6rad/s2$ for the rotational acceleration. I set the **holnomic_robot** parameter to true because I use the mecanum base platform where the robot can move in any

direction.

6.3.2. Costmap Common Parameter

The following script is present the **costmap_common_params.yaml**

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint_range: [[-0.2, -0.2], [-0.2, 0.2], [0.2, 0.2], [0.2,
    -0.2]]
inflation_radius: 0.55
cost_scalling_factor: 10.0
observation_source: scan
scan: {sensor_frame: base_link, observation_persistence: 0.0,
   max_obstacle_height: 0.4, min_obstacle_height: 0.0,
   data_type: LaserScan, topic: /scan, marking: true, clearing:
    true}
```

The **obstacle_range** and **raytrace_range** attributes are used to indicate the maximum distance that the sensor will read and introduce new information in the global and local costmap. The first one is used for the obstacles. If the robot detects an obstacle closer than 2.5 meters, it will put the obstacle in the costmap. The other one is used to clean and clear the costmap and update the free space in it when the robot moves.

The **footprint** attribute is used to indicate the geometry of the robot to the navigation stack. It is used to keep the right distance between the obstacles and the robot, or to find out if the robot can go through a door. The **inflation_radius** attribute is the value given to keep a minimum distance between the geometry of the robot and the obstacles.

The **cost_scalling_factor** attribute modified the behavior of the robot around the obstacles. With the **observation_source** attribute, we set the sensors used by the navigation stack to get the data from the laser and calculate the path. And with the scan topic, I set the obstacle height which laser can detect by giving **min_obstacle_height** to 0.0m and **max_obstacle_height** to 40cm

6.3.3. Global Costmap Parameter

The global costmap keeps the information for the global navigation where parameters are set in the **global_costmap_parameter**.

```
global_costmap:
        global_frame: /map
        robot_base_frame: /base_link
        update_frequency: 1.0
        static_map: true
```

The **global_frame** and the **robot_base_frame** attributes define the transformation between the map and base_link of the robot. This transformation is for the global costmap. I set the **update_frequency** to $1Hz$ to update the frequency of the costmap. **static_map** attribute is used for the global costmap to see whether a map or the map server is used to initialize the costmap.

6.3.4. Local Costmap Parameter

The local costmap keeps all the information for the local navigation where parameters are set in the **local_costmap_parameter**.

```
local_costmap:
        global_frame: /map
        robot_base_frame: /base_link
        update_frequency: 1.0
        publish_frequency: 1.0
        static_map: false
        rolling_window: true
        width: 20.0
        height: 10.0
        resolution: 0.05
        planner_patience: 5.0
```

The **global_frame**, **robot_base_frame**, **update_frequency** and **static_map** parameters are the same as described in the previous section, global costmap parameter. The **publish_frequency** parameter determines the frequency for publishing the information. The **rolling_window** pa-

rameter is used to keep the costmap centered on the robot when it is moving in the map. The **width** and **height** of the map, I set them to 20m and 10m respectively, while the resolution of the map is 0.05. The **planner_patience** parameter configures how long the planner will wait in seconds in an attempt to find a valid plan, before space-clearing operation are performed.

6.4. Navigation Stack in rviz

Launching the navigation stack, we can see the static map as well as the robot with particle cloud around it in the rviz as shown in **Figure**

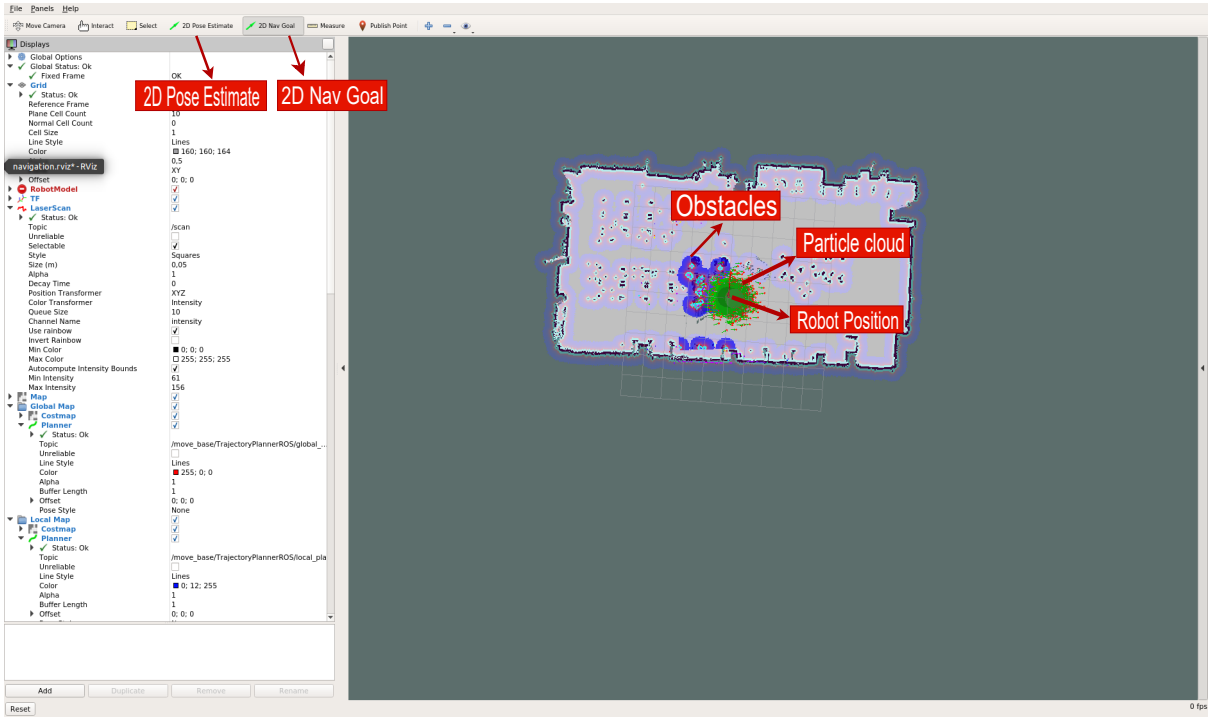

Figure 6.3: Navigation Stack in rviz

- **2D Pose Estimate** allows the user to initialize the localization system used by the navigation stack by setting the pose of the robot in the world. The navigation stack waits for the new pose of a new topic with the name initialpose. This topic is sent using the rviz windows. In order to intialize the position of the robot, we need to click on the **2D Pose Estimate** button, and click on the map. If we don't to do this in the beginning, the robot will start the auto-localization process and try to set an initial pose[3].
- **2D Nav Goal** allows the user to send a goal to the navigation by setting the desired pose for the robot to achieve. The navigation stack waits for a new goal with **move_base_simple/goal** as a topic name. Click on the 2D Nav Goal button, and select the map and the

goal for the robot, and then the robot will move to that position[3].

- **Obstacles** The obstacle that is shown in the map is the obstacle inflation which configures in the **Global and Local costmaps** for the robot to avoid the collision. For the robot to avoid the collision, the robot's footprint should never intersect with a cell that contains an obstacle.
- **Particle cloud** is used by the robot's localization system. The spread of the cloud represents the localization system's uncertainty about the robot's pose.
- **Robot Position** represents the initial position and orientation of the robot when we click on the 2D Pose Estimate, and initialize it in the map[3].

7. Results and Discussion

Launching the amcl node, we get to see the map and the robot as shown in **Figure 7.1**.

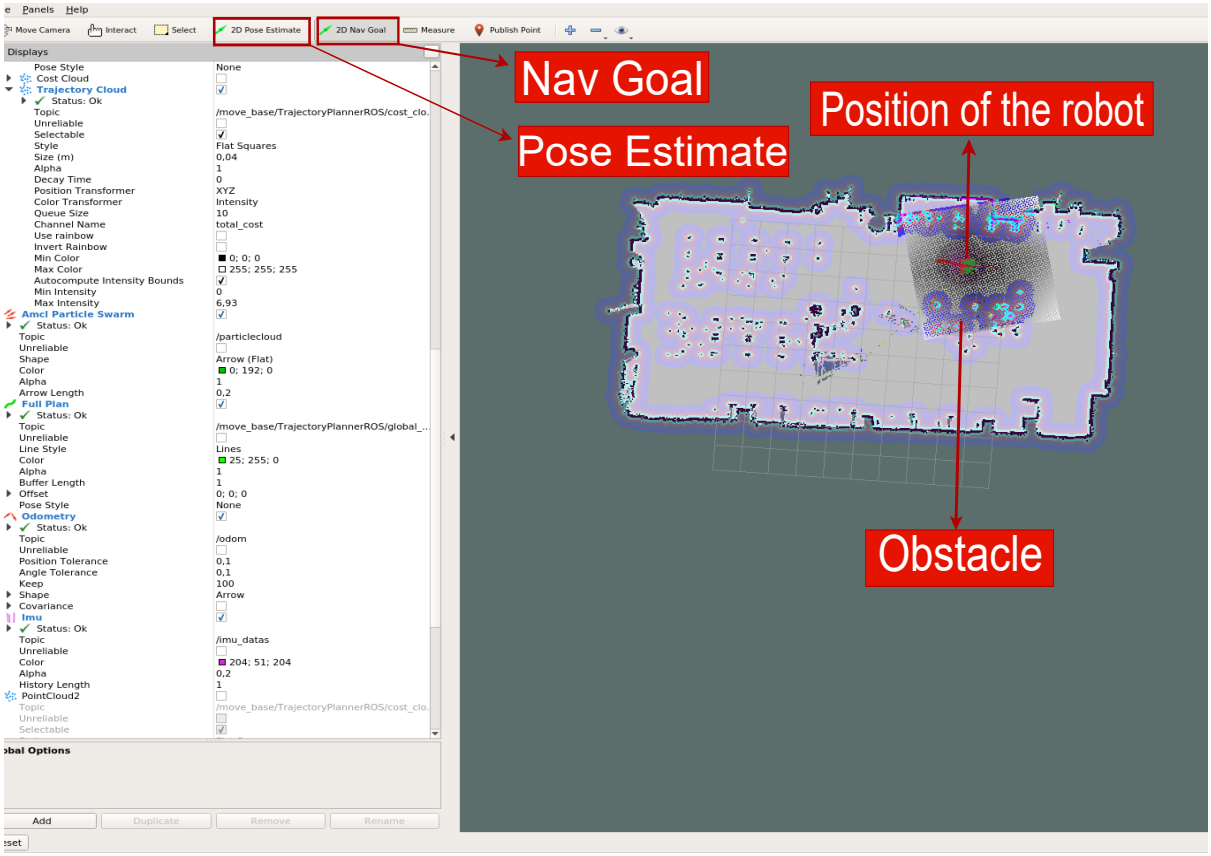

Figure 7.1: Localization the Robot in an Existing Map

There is a 2D position estimation in rviz that allows the user to initialize the localization system used by the navigation stack by setting the pose of the robot on the map. As shown in **Figure 7.1**, there is a textbox called **Pose Estimate** that is used to initialize the position and orientation of the robot in the map. So, I use that textbox to localize the initial position of the robot, and the real robot is shown in **Figure 7.2** that is the same position as the rviz.

After initializing the robot in the map, then the robot can navigate from a position to another position autonomously on the map. As shown in **Figure 7.1**, there is also a text called **Nav Goal** that allows the user to send a goal to the navigation by setting the desired pose for the robot to achieve. So, when I click on the **Nav Goal** and desire the position for the robot, and then robot will move to that position autonomously as shown in **Figure 7.3**.

Then we can see the real robot move to that position as shown in **Figure 7.4**

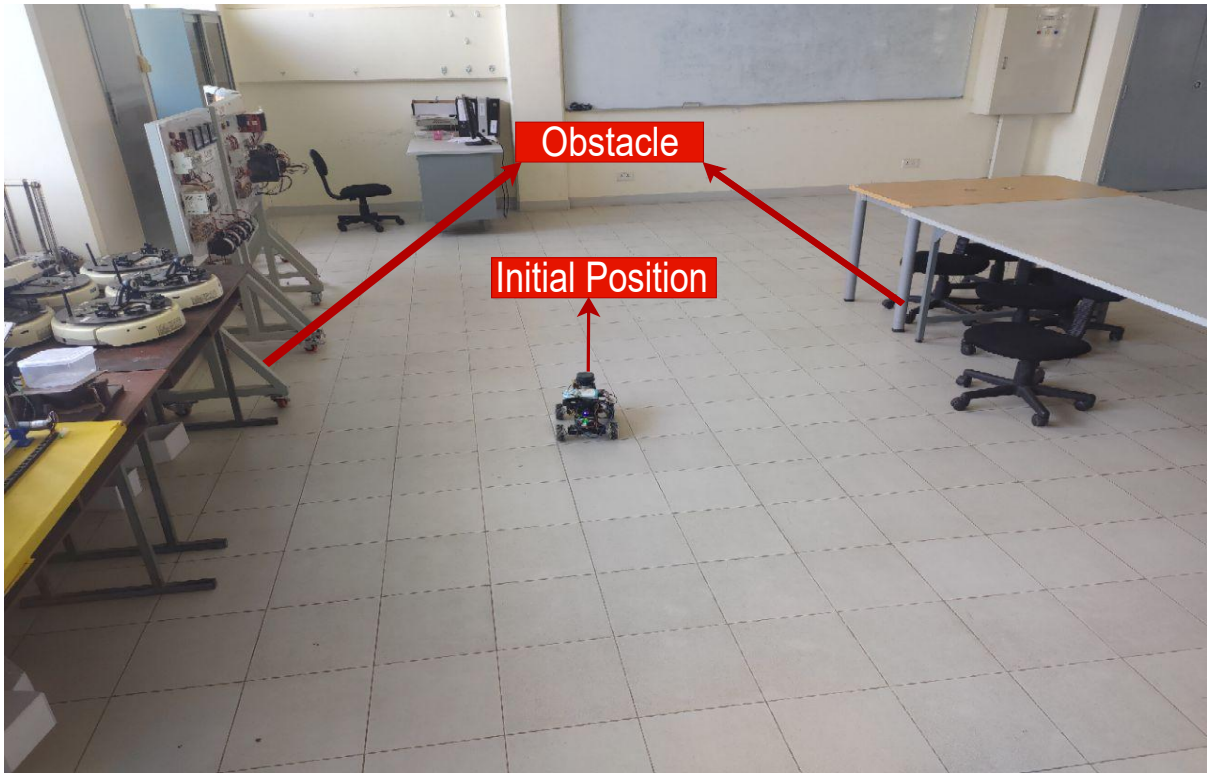

Figure 7.2: Initial Postion of Robot in the Map

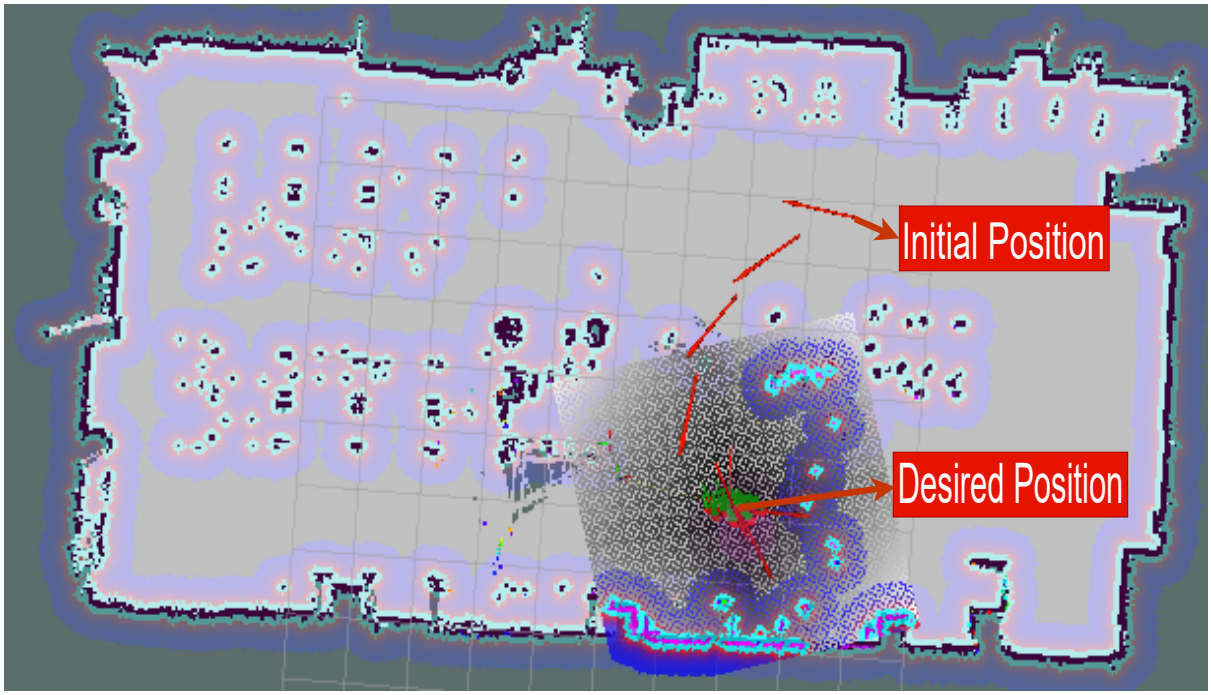

Figure 7.3: Navigation in rviz

Figure 7.4: Desired Position in Map

After testing around in the map, I get to see the one problem with the yaw angle of the robot. Sometimes, I can see the correct angle as I set it to, but sometimes it is error around 50 degree due to the configuration in base_local_planner in **move_base**. The robot can plan the trajectory to avoid the obstacle in the room, but It can not detect the object that is not shown in the map. Even if the robot is stuck in some obstacle, It will move back and plan another trajectory that I set it to in the map.

8. Conclusion and Future Work

To increase the state estimation accuracy of the robot, MPU6050 is fused with wheel encoders in the Arduino microcontroller to construct the odometry message, and then this data publishes the information to Raspberry Pi 4 through Rosserial to fuse with another IMU using the Extended Kalman Filter in order to get more accurate and reliable odometry information of the robot. Moreover, the robot can perform the Simultaneous Localization and Mapping method which is used to scan the map of an environment by fusing the data of the odometry and the lidar sensor. Besides the SLAM algorithm, the robot can do the path planning from a position to another position autonomously using Adaptive Monte Carlo Localization method in ROS.

Future Work

In future work, there are several things I want to do with the mobile robot. the first thing is the change of the components, I want to change from the Android IMU to the Adafruit IMU and fuse this data with odometry message to estimate the position of the robot. One more thing is to use the STM32 instead of Arduino to read the data of motor encoders and MPU6050 and then send these data to Raspberry Pi 4 through CAN bus. Second, I want to use the Kinect Xbox 360 along with the lidar sensor to improve the map of the environment as well as obstacle avoidance and object detection. Finally, I would like to research in Deep Learning and reinforcement learning using ROS and TensorFlow which can be used to perform SLAM and AMCL better than the conventional method of the mobile robot. Deep learning is a way of controlling the steering of the robot using a trained network in which sensor data can be fed to the network and corresponding steering control can be obtained. Reinforcement learning is a kind of machine learning technique that allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. By combining it with deep learning, robots can truly behave as truly intelligent agents that can solve tasks that are considered challenging by humans.

References

[1] Long Cheng, Cheng-dong Wu, and Yun-zhou Zhang. Indoor robot localization based on wireless sensor networks. *IEEE Transactions on Consumer Electronics - IEEE TRANS CONSUM ELECTRON*, 57:1099--1104, 08 2011.

[2] CHRobotics. Understanding euler angles and quaternions. http://www.chrobotics.com/library.

[3] Luis Sanches Crespo Aaron Martinez Enrique Fernandez, anil Mahtani. Find out everything you need to know to build powerful robots with the most up-to-date ros. In *Effective Robotics Programming with ROS*, 2016.

[4] R. Patrick Goebel. Ros by example, a do-it-yourself guide to the robot operating system. January 2015.

[5] Md Maruf lbne Hasan. Indoor and outdoor localization of a mobile robot fusing sensor data. August 2017.

[6] Lentin Joseph. Design, build, and simulate complex robot using robot operating system and master its out-of-the box functionalities. In *Mastering ROS for Robotics Programming*, 2015.

[7] Anis Koubaa. Robot operating system (ros). volume 625, 2016.

[8] Programming Robots. The mpu6050 explained. https://mjwhite8119.github.io/Robots/mpu6050.

[9] S. Zaman, W. Slany, and G. Steinbauer. Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, pages 1--5, 2011.

Appendix A Read Encoder Value, MPU6050, and Motor Speed Control

kinematic.h

```
#ifndef KINEMATIC_H
#define KINEMATIC_H

#include "Arduino.h"
#include "kinematic.h"

float calculateVx( float v1, float v2, float v3, float v4,
   float a, float b);
float calculateVy( float v1, float v2, float v3, float v4,
   float a, float b);
float calculateOmega( float v1, float v2, float v3, float v4,
   float a, float b);

float calculateV1( float vx, float vy, float Omega, float a,
   float b);
float calculateV2( float vx, float vy, float Omega, float a,
   float b);
float calculateV3( float vx, float vy, float Omega, float a,
   float b);
float calculateV4( float vx, float vy, float Omega, float a,
   float b);
#endif
```

kinematic.cpp

```
#include "kinematic.h"

float calculateVx( float v1, float v2, float v3, float v4,
   float a, float b) {
    return (1.0/4.0)*(v1+v2+v3+v4)    ;
}
```

```
float calculateVy( float v1, float v2, float v3, float v4,
    float a, float b) {
     return (1.0/4.0)*(-v1+v2+v3-v4)    ;
}
float calculateOmega( float v1, float v2, float v3, float v4,
    float a,  float b) {
     return  (1.0/((a+b)*4.0))*(-v1+v2-v3+v4)  ;
}

float calculateV1( float vx, float vy, float Omega, float a,
    float b) {
     return ( vx-vy-(a+b)*Omega);
}
float calculateV2( float vx, float vy, float Omega, float a,
    float b) {
     return (vx+vy+(a+b)*Omega );
}
float calculateV3( float vx, float vy, float Omega, float a,
    float b) {
     return ( vx+vy-(a+b)*Omega );
}
float calculateV4( float vx, float vy, float Omega, float a,
    float b ) {
     return ( vx-vy+(a+b)*Omega );
}
```

motor.h

```
#ifndef MOTOR_H
#define MOTOR_H

#include <Encoder.h>
```

```
class Motor
{
  public:
    Motor(uint8_t pinIN1, uint8_t pinIN2, uint8_t mainSpeedPin
        , uint8_t secondarySpeedPin);
    void initPins();
    void commandMotor(int command);
    Encoder enc;
  private:
    const uint8_t _pinIN1;
    const uint8_t _pinIN2;
    const uint8_t _mainSpeedPin;
    const uint8_t _secondarySpeedPin;
};
#endif
```

motor.cpp

```
#include "motor.h"

Motor::Motor(uint8_t pinIN1, uint8_t pinIN2, uint8_t
    mainSpeedPin, uint8_t secondarySpeedPin): _pinIN1(pinIN1),
    _pinIN2(pinIN2), _mainSpeedPin(mainSpeedPin),
    _secondarySpeedPin(secondarySpeedPin),enc(mainSpeedPin,
    secondarySpeedPin){
}
void Motor::initPins() {
    pinMode(_pinIN1, OUTPUT);
    pinMode(_pinIN2, OUTPUT);
    }

void Motor::commandMotor(int command) {
    command = max(-255,min(255,command));
```

```
    if (command > 0) {
      digitalWrite(_pinIN2, LOW);
      analogWrite(_pinIN1, command);
    }
    else if (command < 0) {
      digitalWrite(_pinIN1, LOW);
      analogWrite(_pinIN2, -command);
    }
    else {
      digitalWrite(_pinIN2, LOW);
      digitalWrite(_pinIN1, LOW);
    }
}
```

pid.h

```
#ifndef PID_H
#define PID_H

#include "Arduino.h"

class pid{
    public:
      pid(float kp, float ki, float kd);
      float calculate(float target, float real, long dt);
    private:
      const float _kp;
      const float _ki;
      const float _kd;
      float _error;
      float _integral;
      float _derivative;
      float _last_value;
```

```
        const float _integralBoundary;
        float _cmd;
};
#endif
```

pid.cpp

```
#include "pid.h"
pid::pid(float kp, float ki, float kd): _kp(kp), _ki(ki), _kd(
    kd), _integralBoundary(1.0) {
    _integral = 0.0;
    _last_value = 0.0;
}
float pid::calculate(float target, float real, long dt) {
    _error = target - real;
    _derivative = 1000.0* _error  / dt;
    _integral += _error * dt / 1000.0;
    _integral = max(-_integralBoundary, min(_integralBoundary,
        _integral)); // boundary for integral
    _cmd = _kp * _error + _ki * _integral + _kd * _derivative;
        // calculate PID command
    return round(_cmd);
}
```

main.ino

```
#define USE_USBCON
#if (ARDUINO >= 100)
#include <Arduino.h>
#else
#include <WProgram.h>
#endif
#include <math.h>
#include <ros.h>
#include <geometry_msgs/Vector3Stamped.h>
```

```
#include <geometry_msgs/Twist.h>
#include <ros/time.h>
#include <Encoder.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include "kinematic.h"

#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
#include "Wire.h"
#endif
#include "robot_specs.h"
#include "pid.h"
#include "motor.h"
#define LOOPTIME          100    // PID loop time(ms)
#define SMOOTH        10
#define a 0.1
#define b 0.1

MPU6050 mpu;
#define OUTPUT_READABLE_QUATERNION
#define OUTPUT_READABLE_YAWPITCHROLL

#include <sensor_msgs/Imu.h>
#include <tf/transform_broadcaster.h>
geometry_msgs::TransformStamped t;
tf::TransformBroadcaster broadcaster;

Motor Motor1(9, 10, 22, 24);
Motor Motor2(8, 7, 28, 26);
Motor Motor4(4, 3, 29, 27);
Motor Motor3(5, 6, 23, 25);
long enc_value1 = Motor1.enc.read();
```

```
long enc_value2 = Motor2.enc.read();
long enc_value3 = Motor3.enc.read();
long enc_value4 = Motor4.enc.read();

long enc_last_value1 = enc_value1;
long enc_last_value2 = enc_value2;
long enc_last_value3 = enc_value3;
long enc_last_value4 = enc_value4;

float l1 = 0.0;
float l2 = 0.0;
float l3 = 0.0;
float l4 = 0.0;
float dl1 = 0.0;
float dl2 = 0.0;
float dl3 = 0.0;
float dl4 = 0.0;
float v1 = 0.0;
float v2 = 0.0;
float v3 = 0.0;
float v4 = 0.0;
float v_target1 = 0.0;
float v_target2 = 0.0;
float v_target3 = 0.0;
float v_target4 = 0.0;
float v_motor1 = 0.0;
float v_motor2 = 0.0;
float v_motor3 = 0.0;
float v_motor4 = 0.0;
float cmd1 = 0.0;
float cmd2 = 0.0;
float cmd3 = 0.0;
```

```
float cmd4 = 0.0;
double Vx = 0.0;
double Vy = 0.0;
double Omega = 0.0;
double x = 0.0;
double y = 0.0;
double theta = 0.0;
double vmax = 40;
double rotmax = 25;
double vtarget = 0.0;
double ang_z = 0.0;
double linear_y = 0.0;
double linear_x = 0.0;
double gain=67;
unsigned long last_time = millis();
unsigned long dt = 0.0;
pid pid1(3.0, 1.0, 0.0);
pid pid2(3.0, 1.0, 0.0);
pid pid3(3.0, 1.0, 0.0);
pid pid4(3.0, 1.0, 0.0);

void handle_cmd( const geometry_msgs::Twist& msg) {
linear_x = msg.linear.x;
linear_y = msg.linear.y;
ang_z = msg.angular.z;

v_target1= gain*calculateV1(linear_x,linear_y,ang_z,a,b);
v_target2= gain*calculateV2(linear_x,linear_y,ang_z,a,b);
v_target3= gain*calculateV3(linear_x,linear_y,ang_z,a,b);
v_target4= gain*calculateV4(linear_x,linear_y,ang_z,a,b);

ros::NodeHandle nh;
```

```
#define INTERRUPT_PIN 2
// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte
   from MPU
uint8_t devStatus;      // return status after each device
   operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default
   is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in
   FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion
   container
VectorInt16 aa;         // [x, y, z]            accel sensor
   measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free
   accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame
   accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle
   container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll
    container and gravity vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = { '$', 0x02, 0, 0, 0, 0, 0, 0, 0,
   0, 0x00, 0x00, '\r', '\n' };
```

```
volatile bool mpuInterrupt = false;     // indicates whether
   MPU interrupt pin has gone high
void dmpDataReady() {
mpuInterrupt = true;
}
geometry_msgs::Vector3 orient;
sensor_msgs::Imu imu_msgs;
//Creating ROS publisher object for IMU orientation
ros::Publisher imu_pub("imu_data", &orient);
ros::Publisher imu_pubs("imu_msgs", &imu_msgs);
ros::Subscriber<geometry_msgs::Twist> sub("cmd_vel",
   handle_cmd);
geometry_msgs::Vector3Stamped speed_msg;
geometry_msgs::Vector3Stamped position_msg;
ros::Publisher speed_pub("Speed", &speed_msg);
ros::Publisher pos_pub("Position", &position_msg);
void setup() {
  for (int i = 3; i <= 10; i++) {
    pinMode(i, OUTPUT);
}
// put your setup code here, to run once:
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
  Wire.begin();
  Wire.setClock(400000); // 400kHz I2C clock. Comment this
     line if having compilation difficulties
#elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
  Fastwire::setup(400, true);
#endif
  Serial.begin(57600);
  mpu.initialize();
  pinMode(INTERRUPT_PIN, INPUT);
```

```
// load and configure the DMP

  devStatus = mpu.dmpInitialize();
  Motor1.initPins();
  Motor2.initPins();
  Motor3.initPins();
  Motor4.initPins();

  nh.initNode();
  nh.subscribe(sub);
  nh.advertise(imu_pub);
  nh.advertise(imu_pubs);
  nh.advertise(speed_pub);
  nh.advertise(pos_pub);
  if (devStatus == 0) {
// Calibration Time: generate offsets and calibrate our
   MPU6050
    mpu.CalibrateAccel(6);
    mpu.CalibrateGyro(6);
    mpu.PrintActiveOffsets();
    // turn on the DMP, now that it's ready
    Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
    Serial.print(F("Enabling interrupt detection (Arduino
       external interrupt "));
    Serial.print(digitalPinToInterrupt(INTERRUPT_PIN));
    Serial.println(F(")..."));
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN),
       dmpDataReady, RISING);
```

```
    mpuIntStatus = mpu.getIntStatus();

    // set our DMP Ready flag so the main loop() function
       knows it's okay to use it
    Serial.println(F("DMP ready! Waiting for first interrupt
       ..."));
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
// ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }
  Serial.print("theta=");
  Serial.print(theta);
}
void loop() {
// put your main code here, to run repeatedly:
  dt = millis() - last_time;
  if (dt >= 50) {
    last_time = millis();
    enc_value1 = Motor1.enc.read();
    enc_value2 = Motor2.enc.read();
    enc_value3 = Motor3.enc.read();
    enc_value4 = Motor4.enc.read();
    dl1 =3.14*0.075*(enc_value1-enc_last_value1)/49860.0;
```

```
    v1 = 1000*dl1/dt;
    enc_last_value1 =enc_value1;
    dl2 =  3.14*0.075* (enc_value2-enc_last_value2)/49860.0;
    v2 = 1000 * dl2 / dt;
    enc_last_value2 = enc_value2;
    dl3 =3.14*0.075*(enc_value3-enc_last_value3)/49860.0;
    v3 =1000*dl3/dt;
    enc_last_value3 =enc_value3;
    dl4 = 3.14*0.075*(enc_value4 - enc_last_value4)/49860.0;
    v4 =  1000*dl4/dt;
    enc_last_value4=enc_value4;
}
Vx = calculateVx(v1, v2, v3, v4, a, b);
Vy = calculateVy(v1, v2, v3, v4, a, b);
Omega = calculateOmega(v1, v2, v3, v4, a, b);
x += Vx * dt / 1000.0;
y += Vy * dt / 1000.0;
theta = orient.z;
v_motor1 = pid1.calculate(v_target1, v1, dt);
v_motor2 = pid2.calculate(v_target2, v2, dt);
v_motor3 = pid3.calculate(v_target3, v3, dt);
v_motor4 = pid4.calculate(v_target4, v4, dt);
Motor1.commandMotor(v_motor1);
Motor2.commandMotor(v_motor2);
Motor3.commandMotor(v_motor3);
Motor4.commandMotor(v_motor4);
Serial.print("v_target1=");
Serial.print(v_target1);
Serial.print(";");
Serial.print("v_target2=");
Serial.print(v_target2);
Serial.print(";");
```

```
Serial.print("v_target3=");
Serial.print(v_target3);
Serial.print(";");
Serial.print("v_target4=");
Serial.print(v_target4);
Serial.print(";");
Serial.print("vmotor1=");
Serial.print(v_motor1);
Serial.print(";");
Serial.print("vmotor2=");
Serial.print(v_motor2);
Serial.print(";");
Serial.print("vmotor3=");
Serial.print(v_motor3);
Serial.print(";");
Serial.print("vmotor4=");
Serial.print(v_motor4);
Serial.print(";");
speed_msg.vector.x = Vx;
speed_msg.vector.y = Vy;
speed_msg.vector.z = Omega;
position_msg.vector.x = x;
position_msg.vector.y = y;
position_msg.vector.z = -theta;
speed_pub.publish(&speed_msg);
pos_pub.publish(&position_msg);
#ifdef OUTPUT_READABLE_YAWPITCHROLL
// display Euler angles in degrees
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);;
orient.z = ypr[0] * 180 / M_PI;
```

```
orient.x = ypr[1] * 180 / M_PI;
orient.y = ypr[2] * 180 / M_PI;
imu_pub.publish(&orient);
#endif
}
Serial.print("x=");
Serial.print(x);
Serial.print(";");
Serial.print("y=");
Serial.print(y);
Serial.print(";");
Serial.print("theta=");
Serial.print(theta);
Serial.print(";");
Serial.print("Vx=");
Serial.print(Vx);
Serial.print(";");
Serial.print("Vy=");
Serial.print(Vy);
Serial.print(";");
Serial.print("omega=");
Serial.println(Omega);
nh.spinOnce();
delay(10);

}
```

Appendix B Subscribe and Publish data of IMU

```
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>

ros::Publisher imu_pub;
void imuCallback(const sensor_msgs::Imu& imu_msgs){
  sensor_msgs::Imu imu;
  imu.header.stamp = current_time;
  imu.header.frame_id = "base_link";
  imu.orientation.x = imu_msgs.orientation.x;
  imu.orientation.y = imu_msgs.orientation.y;
  imu.orientation.z = imu_msgs.orientation.z;
  imu.orientation.w = imu_msgs.orientation.w;
  imu.angular_velocity.x = imu_msgs.angular_velocity.x;
  imu.angular_velocity.y = imu_msgs.angular_velocity.y;
  imu.angular_velocity.z = imu_msgs.angular_velocity.z;
  imu.linear_acceleration.x = imu_msgs.linear_acceleration.x;
  imu.linear_acceleration.y = imu_msgs.linear_acceleration.y;
  imu.linear_acceleration.z = imu_msgs.linear_acceleration.z;
  imu_pub.publish(imu);
}
int main(int argc, char** argv){
    ros::init(argc, argv, "IMU_publisher");
    ros::NodeHandle nh;
    imu_pub = nh.advertise<sensor_msgs::Imu>("imu_datas", 50);
    ros::Subscriber imu_sub = nh.subscribe("/android/imu",10,
       imu_datacallback);
    ros::spin();
    return 0;
}
```

Appendix C Odometry Publisher

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Vector3.h>
#include <cmath>
#include <algorithm>
#include <sstream>
#include <geometry_msgs/Vector3Stamped.h>
double dt = 0.0
double x = 0.0;
double y = 0.0;
double th = 0.0;
double vx = 0.0;
double vy = 0.0;
double vth = 0.0;
double rate=20.0;
ros::Time current_time, last_time;
ros::Publisher odom_pub;

void speedCallback(const geometry_msgs::Vector3Stamped&
   speed_msg){
//  tf::TransformBroadcaster odom_broadcaster;

  vx=speed_msg.vector.x;
  vy=speed_msg.vector.y;
  vth=speed_msg.vector.z;
}
void positionCallback(const geometry_msgs::Vector3Stamped&
   position_msg){
```

```
  tf::TransformBroadcaster odom_broadcaster;
  current_time = ros::Time::now();
  last_time= ros::Time::now();
  x=position_msg.vector.x;
  y=position_msg.vector.y;
  th=position_msg.vector.z;
  geometry_msgs::Quaternion odom_quat =tf::
     createQuaternionMsgFromYaw(th);
  dt =(current_time-last_time).toSec(); //calc velocities

  nav_msgs::Odometry odom; //create nav_msgs::odometry
  odom.header.stamp = current_time;
  odom.header.frame_id = "odom";

  odom.pose.pose.position.x = x; //set positions
  odom.pose.pose.position.y = y;
  odom.pose.pose.position.z = 0.0;
  odom.pose.pose.orientation = odom_quat;

  odom.child_frame_id = "base_link"; // set child frame and
     set velocity in twist message
  odom.twist.twist.linear.x =vx;
  odom.twist.twist.linear.y =vy;
  odom.twist.twist.angular.z =vth;
  //ROS_INFO("%d", msg.data.c_str());
  odom_pub.publish(odom);   //publish odom message
  last_time = current_time;
  //ROS_INFO("Received [%d]",msg->data)
}

int main(int argc, char **argv)
{
```

```
    ros::init(argc, argv, "testing");
    tf::TransformBroadcaster odom_broadcaster;
    ros::NodeHandle nh;
    odom_pub = nh.advertise<nav_msgs::Odometry>("odom", 50);
    ros::Subscriber speed = nh.subscribe("Speed", 50,
       speedCallback);
    ros::Subscriber position = nh.subscribe("Position", 50,
       positionCallback);
    current_time = ros::Time::now();
    last_time= ros::Time::now();
    ros::spin();
    return 0;
}
```

Appendix D Extended Kalman Filter

```
frequency: 50
sensor_timeout: 0.1
transform_time_offset: 0.0
two_d_mode: true
print_diagnostics: true
publish_tf: true
map_frame: map
odom_frame: odom
base_link_frame: base_link
world_frame: odom

odom0: /odom
odom0_config: [ false , false , false ,
                false , false , false ,
                true , true , false ,
                false , false , true ,
                false , false , false]
odom0_differential: true
odom0_queue_size: 10

imu0: /imu_datas
imu0_config: [false , false , false ,
              false , false ,  true ,
              false , false , false ,
              false , false ,  true ,
              false , false , false]

imu0_differential: false
imu0_queue_size: 10
imu0_remove_gravitational_acceleration: true
```

Appendix E SLAM Launch File

```
<launch>
<node pkg="hector_mapping" type="hector_mapping" name="
   hector_mapping" output="screen">
 <!-- Frame names -->
<param name="pub_map_odom_transform" value="true"/>
<param name="map_frame" value="map" />
<param name="base_frame" value="base_link" />
<param name="odom_frame" value="odom" />
 <!-- Tf use -->
<param name="use_tf_scan_transformation" value="true"/>
<param name="use_tf_pose_start_estimate" value="false"/>
 <!-- Map size / start point -->
<param name="map_resolution" value="0.05"/>
<param name="map_size" value="2048"/>
<param name="map_start_x" value="0.5"/>
<param name="map_start_y" value="0.5" />
<param name="laser_z_min_value" value = "-1.0" />
<param name="laser_z_max_value" value = "1.0" />
<param name="map_multi_res_levels" value="2" />
<param name="map_pub_period" value="2" />
<param name="laser_min_dist" value="0.4" />
<param name="laser_max_dist" value="5.5" />  <!--before 5.5-->
<param name="output_timing" value="false" />
<param name="pub_map_scanmatch_transform" value="true" />
<param name="tf_map_scanmatch_transform_frame_name" value="
   scanmatcher_frame" />
 <!-- Map update parameters -->
<param name="update_factor_free" value="0.4"/>
<param name="update_factor_occupied" value="0.7" />
<param name="map_update_distance_thresh" value="0.1"/>
<param name="map_update_angle_thresh" value="0.025" />
```

```
<!-- Advertising config -->
<param name="advertise_map_service" value="true"/>
<param name="scan_subscriber_queue_size" value="5"/>
<param name="scan_topic" value="scan"/>
</node>

<node pkg="tf" type="static_transform_publisher" name="
    scanmatcher_broadcaster" args="0 0 0 0 0 0 map
    scanmatcher_frame 100"/>

<!--<node pkg="tf" type="static_transform_publisher" name="
    nav_basefoot_broadcaster" args="0 0 0 0 0 0 odom
    base_footprint 100"/>
<node pkg="tf" type="static_transform_publisher" name="
    basefootprint_baselink_broadcaster" args="0 0 0 0 0 0
    base_footprint base_link 100"/>-->

<node pkg="rviz" type="rviz" name="rviz"
args="-d $(find hector_slam_launch)/rviz_cfg/mapping_demo.rviz
    "/>
</launch>
```

Appendix F AMCL Launch File

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" output="screen">
<param name="base_frame_id" value="base_link"/>
<param name="gui_publish_rate" value="10.0"/>
<param name="kld_err" value="0.05"/>
<param name="kld_z" value="0.99"/>
<param name="laser_lambda_short" value="0.1"/>
<param name="laser_likelihood_max_dist" value="2.0"/>
<param name="laser_max_beams" value="60"/>
<param name="laser_model_type" value="likelihood_field"/>
<param name="laser_sigma_hit" value="0.2"/>
<param name="laser_z_hit" value="0.5"/>
<param name="laser_z_short" value="0.05"/>
<param name="laser_z_max" value="0.05"/>
<param name="laser_z_rand" value="0.5"/>
<param name="max_particles" value="2000"/>
<param name="min_particles" value="500"/>
<param name="odom_alpha1" value="0.25"/>
<param name="odom_alpha2" value="0.25"/>
<param name="odom_frame_id" value="odom"/>
<param name="odom_model_type" value="omni"/>
<param name="recovery_alpha_slow" value="0.001"/>
<param name="recovery_alpha_fast" value="0.1"/>
<param name="resample_interval" value="1"/>
<param name="transform_tolerance" value="1.25"/>
<param name="update_min_a" value="0.2"/>
<param name="update_min_d" value="0.2"/>
</node>
</launch>
```

Appendix G Move_base Launch File

```
<?xml version="1.0"?>

<launch>

 <!-- Run the map server -->
<node name="map_server" pkg="map_server" type="map_server"
   args="$(find testing_cpp)/maps/labITC.yaml"/>
 <!--- Run AMCL -->
<include file="$(find testing_cpp)/launch/include/amcl.launch"
    />
<node pkg="rviz" type="rviz" name="rviz"
args="-d $(find testing_cpp)/rviz/navigation.rviz" />
<node pkg="move_base" type="move_base" respawn="false" name="
   move_base" output="screen">
<param name="controller_frequency" value="1.0"/>
<param name="controller_patiente" value="2.0"/>
<rosparam file="$(find testing_cpp)/param/config/
   costmap_common_params.yaml" command="load" ns="
   global_costmap" />
<rosparam file="$(find testing_cpp)/param/config/
   costmap_common_params.yaml" command="load" ns="local_costmap
   " />
<rosparam file="$(find testing_cpp)/param/config/
   local_costmap_params.yaml" command="load" />
<rosparam file="$(find testing_cpp)/param/config/
   global_costmap_params.yaml" command="load" />
<rosparam file="$(find testing_cpp)/param/config/
   base_local_planner_params.yaml" command="load" />
</node>
</launch>
```