វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា
INSTITUTE OF TECHNOLOGY OF CAMBODIA

GRADUATE SCHOOL
MASTER'S DEGREE OF ENGINEERING
IN
MECHATRONICS, INFORMATION AND COMMUNICATION ENGINEERING

PATH PLANNING AND CONTROL OF WHEELED MOBILE ROBOT WITH OCCUPANCY GRID MAP

A THESIS SUBMITTED BY

YONRITH Phayuth

UNDER CO-SUPERVISION OF

Dr. SRANG Sarot, Mr. SAKAL Morokot

JULY 2021

PATH PLANNING AND CONTROL OF WHEELED MOBILE ROBOT WITH OCCUPANCY GRID MAP

A THESIS PRESENTED BY

YONRITH Phayuth

TO
THE INSTITUTE OF TECHNOLOGY OF CAMBODIA

IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE AWARD OF
MASTER'S DEGREE OF ENGINEERING

SPECIALIZATION: ***Mechatronics, Information and Communication Engineering***

SUPERVISOR: **Dr. SRANG Sarot**
CO-SUPERVISOR: **Mr. SAKAL Morokot**

EXAMINATION COMMITTEE

Dr. SIM Tepmony, PhD, Lecturer, ITC	*Chair*
Dr. CHRIN Phok, PhD, Lecturer, ITC	*Reviewer*
Dr. KIM Bunthern, PhD, Lecturer, ITC	*Reviewer*
Dr. PEC Rothna, PhD, Lecturer, ITC	*Reporter*
Dr. SRANG Sarot, PhD, Lecturer, ITC	*Member*

PHNOM PENH, JULY 19, 2021

ក្រសួងអប់រំ យុវជន និងកីឡា
វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

និក្ខេបបទបរិញ្ញាបត្រជាន់ខ្ពស់
របស់និស្សិត យ៉ុនរិទ្ធ ផាយុទ្ធ

ការពារថ្ងៃទី ៖ កក្កដា ២០២១

អនុញ្ញាតឱ្យការពារនិក្ខេបបទ

នាយកវិទ្យាស្ថាន ៖ _____

ថ្ងៃទី ខែ ឆ្នាំ ២០២១

ប្រធានបទ ៖ ការធ្វើផែនការស្វែងរកផ្លូវនិងការគ្រប់គ្រងបញ្ហារបស់រ៉ូបូតកង់
ចល័តដោយប្រើប្រាស់ផែនទីក្រឡាចត្រង្គ

ស្ថាប័នទទួលកម្មសិក្សា ៖ Dynamics and Control Laboratory

នាយកសាលាក្រោយបរិញ្ញាបត្រ ៖ បណ្ឌិត សុិម ទេពមុនី _____

សហសាស្ត្រាចារ្យណែនាំ ៖ ១. បណ្ឌិត ស្រង់ សារ៉ុត _____

២. លោក សាកល មរកត_____

រាជធានីភ្នំពេញ ឆ្នាំ២០២១

MINISTRY OF EDUCATION,

YOUTH AND SPORT

INSTITUTE OF TECHNOLOGY OF CAMBODIA

MASTER'S THESIS

Of Mr. YONRITH Phayuth

Defense Date: July , 2021

PERMISSION TO DEFEND THE THESIS

Director of Institute: _____

Phnom Penh, 2021

Thesis's Title: Path Planning and Control of Wheeled Mobile Robot with Occupancy Grid Map

Host Institution: Dynamics and Control Laboratory

Director of Graduate School: Dr. SIM Tepmony _____

Supervisors: 1. Dr. SRANG Sarot _____

2. Mr. SAKAL Morokot _____

PHNOM PENH, CAMBODIA

ACKNOWLEDGMENTS

First, I wish to express my faithful gratitude and deep gratitude to my advisors **Dr. Sarot SRANG** and my co-advisor **Mr. Morokot SAKAL** for their inestimable guidance, humble support, admirable advice, and unceasing encouragement.

Secondly, I would like to express my thank towards my **Family** who is supported me mentally and financially during my study.

Finally, I would like to thank my **Colleagues** and **Seniors** in **Dynamics and Control Laboratory** who always support and help in many problems.

សេចក្តីសង្ខេប

ការធ្វើផែនការស្វែងរកផ្លូវដើរនៃរ៉ូបូតគឺជាការងារមួយសំខាន់នៅក្នុងការធ្វើដំណើរ
របស់រ៉ូបូត។ការធ្វើផែនការផ្លូវដើរអាចអោយរ៉ូបូតផ្លាស់ទីពីកន្លែងមួយទៅកន្លែងមួយបាន
ដោយជៀសវាងពីឧបសគ្គផ្សេងៗដូចជា ជញ្ជាំង គ្រឿងសង្ហារឹម មនុស្ស ល។ដើម្បីស្វែងរក
ផ្លូវដើរបានរ៉ូបូតត្រូវការព័ត៌មានផ្សេងៗអំពីទីតាំងដែលវាធ្វើដំណើរជាមួយនឹងការប្រើប្រាស់
ក្បួននៃការគណនាដើម្បីទទួលបានផ្លូវមួយដែលសមស្រប។គោលបំណងនៃនិក្ខេបបទនេះ
ផ្តោតទៅលើក្បួននៃការគណនា ការគ្រប់គ្រងនិងបញ្ជា និងការប៉ាន់ស្មានទីតាំងរបស់រ៉ូបូត។
ការធ្វើពិសោធន៍បានធ្វើឡើងនៅក្នុងបរិស្ថាន SIMULATION ដោយប្រើបាស់កម្មវិធី
GAZEBO និង ROS ។ស៊ីនេទិចនិងឌីណាមិចរបស់រ៉ូបូតត្រូវបានស្រាយ។វិធីបញ្ជារ៉ូបូត
ដោយប្រើបាស់ BACKSTEPPING CONTROLLER ត្រូវបានជ្រើសរើសមកប្រើដើម្បី
គ្រប់គ្រងចលនារបស់រ៉ូបូត។ដើម្បីបង្កើតផែនទី (OCCUPANCY GRID MAP) នៃបរិស្ថានជុំ
វិញទីតាំងធ្វើដំណើរយើងប្រើវិធី SIMULTANEOUS LOCALIZATION AND MAPPING
(SLAM) ហៅ HECTOR SLAM ។វិធីស្វែងរកផ្លូវ A-STAR ត្រូវបានយកមកប្រើដើម្បីរកផ្លូវ
សម្រាប់រ៉ូបូត។វិធីគណនា EXTENDED KALMAN FILTER ជាមួយស៊ីនេទិចត្រូវប្រើជាវិធី
ប៉ាន់ស្មានទីតាំងរបស់រ៉ូបូត។យើងSIMULATEឧបករណ៍ចាប់សញ្ញា៣ប្រភេទមានដូចជា
ឧបករណ៍ WHEEL ENCODER ឧបករណ៍ INTERTIAL MEASUREMENT UNIT (IMU) និង
ឧបករណ៍ LIGHT DETECTION AND RANGING (LIDAR)។ភាពមិនច្បាស់លាស់នៃ
ទិន្នន័យនៃឧបករណ៍ទាំងនោះត្រូវបានយកជា WHITE GAUSSIAN ។ក្នុងការធ្វើពិសោធន៍រ៉ូ
បូតដើរ នៅក្នុង២ករណីផ្សេងគ្នាគឺ ”MAP1” និង “MAP2” ។ លទ្ធផលបង្ហាញពីទិន្នន័យអំពី
ផែនទី គន្លងដំណើរ និង ការគ្រប់គ្រងបញ្ជារ៉ូបូត។

ABSTRACT

Wheeled mobile robotic path planning is one of the main problem in the robot navigation task. Path planning allows the robot to navigate inside surrounding environment from point A to point B while avoiding the obstacle such as wall, furniture, human, -etc. To plan the path that it needs to take in the environment, the robot needs a right quantity of information of its surrounding and algorithm that will determine the optimal path for the navigation. This thesis presents an algorithm of path planning, control, and localization for the robot. An experiment is conducted inside the simulation environment using Gazebo and ROS software. The robot kinematic and dynamics of differential drive mobile are derived. The backstepping controller is used to control the robot motion. To crate a map of the environment (Occupancy Grid Map), we apply Simultaneous Localization and Mapping (SLAM) method called HectorSLAM. A* path planning algorithm is used to find the path for the robot to move. Extended Kalman Filter is used with the kinematic model to localize the robot position in the map. We use three simulated sensors such as: inertial measurement unit (IMU), wheel encoder, and light detection and ranging (Lidar). The noise of each sensor is assumed to be white Gaussian. In the experiment, the robot performs in two different cases: "Map1" and "Map2". The result shows the collected data from SLAM, Path, and Control of the robot.

ABBREVIATIONS AND SYMBOLS

ω	Robot's Angular velocity in Z-axis of Local Frame
ω_c	Trajectory's Angular Velocity Control
ω_l	Robot left wheel's angular velocity
ω_r	Robot right wheel's angular velocity
ω_{ref}	Trajectory's Angular Velocity Reference
θ	Heading Angle
e_1	Error in X-axis
e_2	Error in Y-axis
e_3	Error in θ Heading Angle
L	Robot base length
p_x	Position in X-axis
p_y	Position in Y-axis
r	Robot wheel radius
T_e	Transformation Matrix about Z-axis
V	Robot's Linear velocity in X-axis of Local Frame
V_c	Trajectory's Linear Velocity Control
V_l	Robot left wheel's linear velocity
V_r	Robot right wheel's linear velocity
V_{ref}	Trajectory's Linear Velocity Reference
x	Robot's pose
x_g	X-axis in Global Frame
x_l	X-axis in Local Frame
y_g	Y-axis in Global Frame
y_l	Y-axis in Local Frame
EKF	Extended Kalman Filter
ICR	Instantaneous Center of Rotation
IMU	Inertial Measurement Unit
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Background

Wheeled Mobile Robotic is a one of the robotic field that has been around for many years. The studies on wheeled mobile robot have produced many interesting results that allow many breakthroughs in the robotic field. One study subject on wheeled mobile robot is the Autonomous Robot Navigation. To achieve an autonomous navigation functionality, the robot needs a great amount of information of the surrounding environment, thus different kinds of sensors have been used and numerous algorithms have been deployed on the robot. One of the problem that attract the attention of the robotic community as well as researchers and developers is the Robotic Path Planning.

For human, moving from point A to point B is an easy task. However, for the robot, navigation is a challenging task that many researchers and developers have invested time on. A robot uses sensors to perceive the environment (up to some degree of uncertainty) and to build or update its environment map.(Klanar et al., 2017). To determine appropriate motion actions that lead to the desired goal location, it can use different decision and planning algorithms. In the process of path planning, the robots kinematic and dynamics constraints are considered.

This thesis is structured into 8 chapters. In the first chapter the **Introduction**, we show background of the study, statement of problem of why we choose this topic to study on, objectives of the thesis, and scope of the study. The second chapter is the **Literature Review** where we show past research, Differential Drive Robot mechanism, type of robot motion controller, and type of path planning algorithm. The third chapter is the **Research Methodology** where we show procedure of experiment, workflow and 3D modeling for simulation software. The fourth chapter is the **Differential Drive Mobile Robot Modeling**. This chapter shows kinematic and dynamics model for the robot. The fifth chapter is **Control Differential Drive Mobile Robot**. This chapter shows control algorithm for robot motion using kinematic and dynamics model and Sensor Fusion for robot localization using Extended Kalman Filter. The sixth chapter talk about the **Path Planning**. In the chapter, we show framework software for simulation with ROS and ROS message, Path Planning algorithm using A* and Occupancy Grid Map. In the seventh chapter, we show the **Result and Discussion** of the study. The last chapter is the **Conclussion and Recommendation** where we conclude the thesis and give future recommendation.

1.2. Statement of Problem

The navigation of wheeled mobile robot highly depends on the information that it has. For example, the information of the surrounding environment which is represented in the form of Occupancy Grid Map and the information of dynamic environment observed from sensors. To achieve autonomous navigation, a robot is required to be able to plan and move along trajectory. The trajectory is planned dynamically in accordant with moving obstacles.

Wheeled Mobile robot is a highly studied topic in today's world. Many high tech companies are deploying robot to the working environment of their company such warehouse, farm, fulfillment center -etc. That action has resulted in a significant demand of the most high performance and optimally designed robot that can accelerate the work force. Thus, high performance robot need to be equipped with highly advance sensors and devices along with the implementation of complex algorithms. However, it may induce high cost of robot production. The importance of this research is to provide a possible method that utilizes affordable sensors and robots for a commercial product. Long-term goal of this research is to construct an autonomous mobile robot that is able to serve as a transportation of objects from one place to another indoor such as a factory, a shop, -etc.

1.3. Objective

This thesis aims to:

- Determine the pathway to move the robot using known static Occupancy Grid Map from SLAM Method
- Design a controller for the robot to follow a planned pathway.

1.4. Scope

In this research, the differential drive mobile robot is selected to be the main robot platform. The robot is considered to move inside the 2D environment, thus the information of the z-axis is neglected. The experiment is conducted inside a simulation of Gazebo environment with ROS framework. The sensor and robot model are simulated using a high accurate physics software. The project produces a ROS package for robot visualization, path planning and control.

2. LITERATURE REVIEWS

2.1. Mobile Robot Modelling

Differential drive mobile robot is a commonly used and developed robot in the field. It is a very simple robot that operates in two wheels. Both wheels are attached to the common axis and usually are driven by two independent motors. To prevent the robot platform from tilting, one or more free spin caster wheel are attached to the robot platform. The differential drive wheel mobile robot model is illustrated in **Figure 2.1.**.The differential drive robot movements depend on the velocity of each individual wheel which dictate its motion and steering. For example: if both wheels move in the same direction and at the same speed, the robot will move forward. If both wheels move in the different direction of each other but in the same speed, the robot will rotate about its center etc.

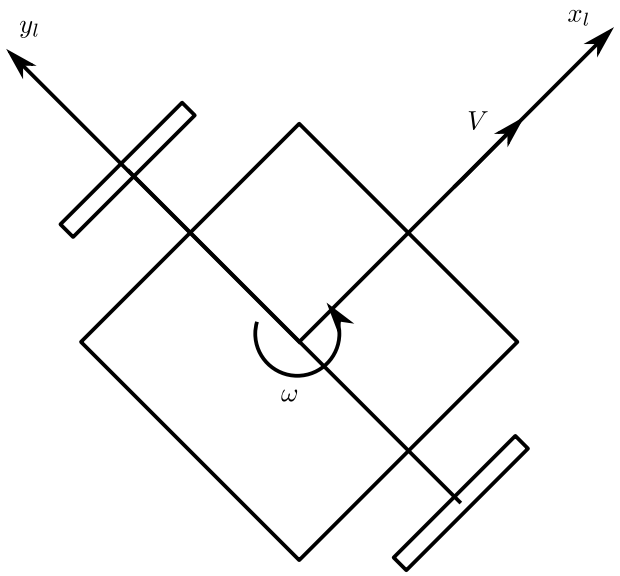

Figure 2.1. Differential Drive Wheel Mobile Robot

Differential drive mobile robot pose is represented in the 2D Cartesian coordinate system. There are two coordinate frames that the robot uses, the global frame and the local frame. In kinematic, the robot velocity is described using two variables and vice versa it is used to control input to the robot. The variables are:

- V, the linear velocity along the x-axis of the local coordinate frame
- ω, the angular velocity about the z-axis of the local coordinate frame.

2.2. Controller

The differential drive mobile robot is a nonholonomic constraint system due to the absence of the velocity along the Y-axis of the local frame. The nonholonomic constraint of the robot is:

$$-\dot{x}sin\theta + \dot{y}cos\theta = 0 \qquad \textbf{(Eq. 2.1.)}$$

Figure 2.2. shows a common navigation of mobile robot. To achieve the control of the nonholonomic constraint system, the use of a nonsmooth or time varying controller is needed because the system is nonlinear and varies in time. The robot system is divided into kinematic control and dynamics control. The Most commonly used controls for mobile robots are:

- **Control to reference pose**

  is the control approach where the reference position and orientation is predefined. From the current pose to the reference pose, the robot can drive in an arbitrary path.

- **Control to segment of line**

  is the control approach where the reference segment of the line is constructed from multiple reference poses.

- **Control to reference trajectory**

  is the control approach where the reference pose is a function of time.

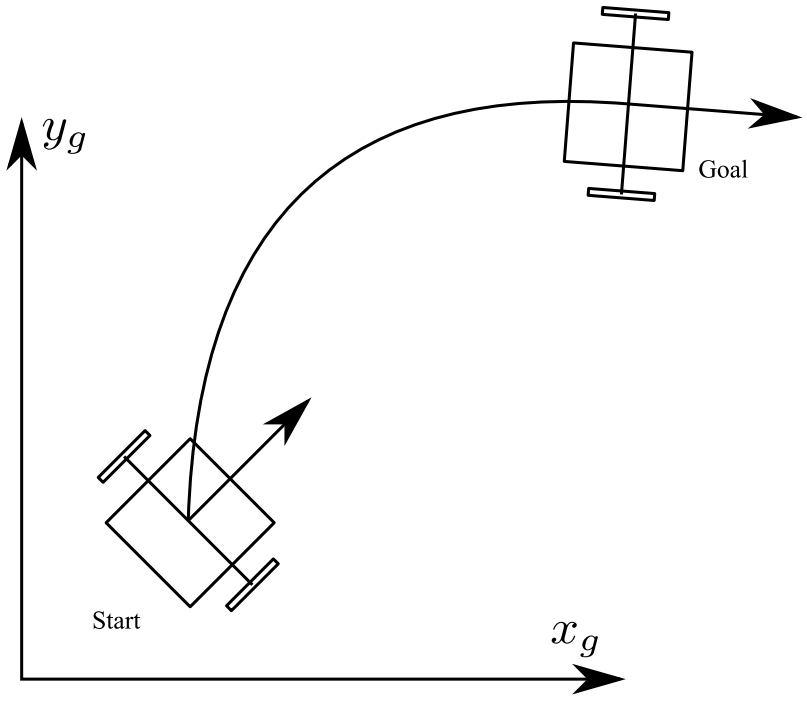

Figure 2.2. Mobile Robot Navigation

Control wheel mobile robots usually decompose into two part: the **Feedforward** control and the **Feedback** control as illustrated in **Figure 2.3.**.

- In **Feedforward** control, the input to control robot is determined from the calculation of the reference goal or trajectory. The input can then apply to the robot system to move along in the open-loop system.
- On the other hand, the **Feedback** control determine the current state of the robot in every time step and feeds back to the controller for a robust performance.

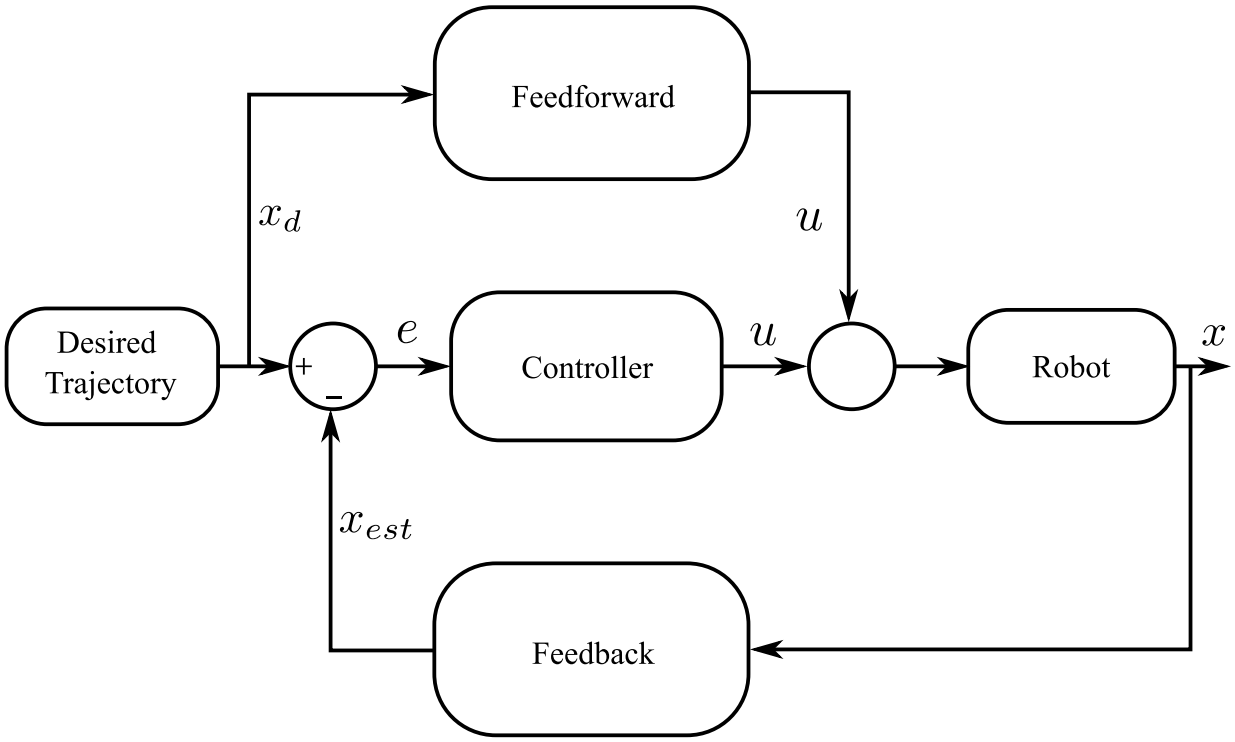

Figure 2.3. Simple Feedforward and Feedback control

In the real world, the mobile robot is a dynamics system. Thus, for robust control on the dynamics system, the dynamic properties of the system are included, such as torque, mass, moment of inertia, external force acting on the robot,-etc.

The controller of the robot dynamics system can be decomposed into two parts: the **Outer** controller and the **Inner** controller.

- For the **outer** controller, the system calculate the required system velocities to move along the reference trajectory or to move to a reference pose.
- While the **inner** control is calculating the torque or force that is required to match the required system velocity from the outer controller.

2.3. Path Planning

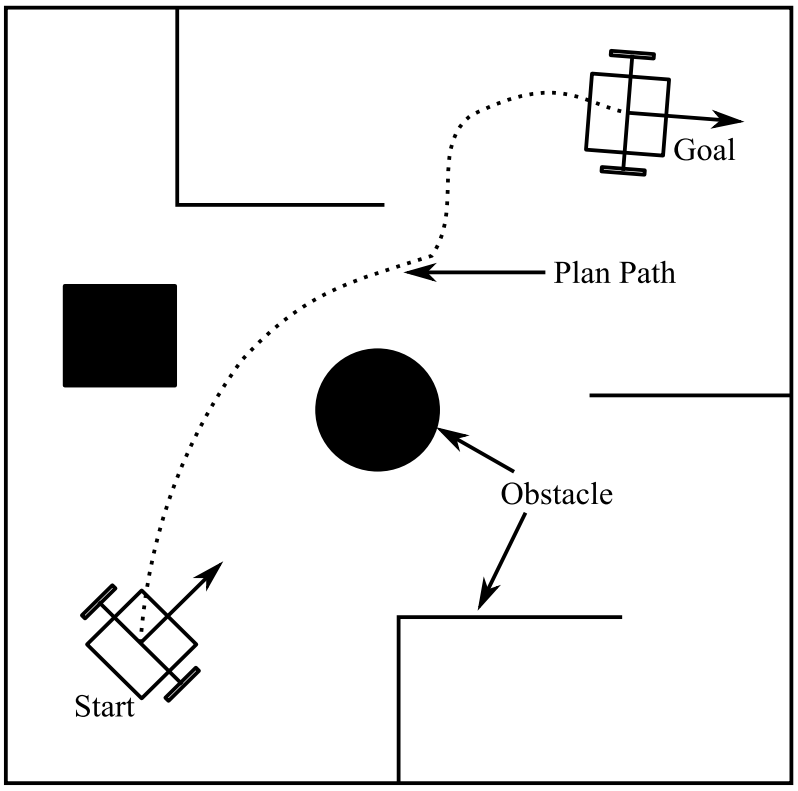

Figure 2.4. Path Planning

To reach a certain goal, the robot required a number of information and algorithms to perform. In the process of path planning, the robot computes sequence of valid actions of the robot configuration to move to the goal. In the case of a mobile robot, the problem is usually defined as the moving from start to goal while avoiding the obstacle **Figure 2.4.**. In the process of path planning of the real world scenario, the robots kinematic and dynamic constraints is considered. Since the environment is not always the same, the path planning works with the limited of the representation of the environment in the algorithm. The algorithm can perform in a fully or partly known environment.

A preliminary literature review show that there are many approaches to the problem. For example:

• **Probabilistic Road Maps (PRMs)**

- **Random Space Sampling**

  the algorithm indicates the random points in space that are selected to construct a graph, whose edges are created if two nodes abstain no more than a certain distance.

- **Space Sampling with Halton Sequence**

  a graph is created by selecting points in space using Halton sequence.

- **Uniform Space Sampling**

  the algorithm selects even points in the free space at a certain step. Each selected node is connected to eight peripheral sampling points.

• Visibility Graph

Visibility graph construction is based on the environment's convex obstacles as the algorithm treats their edges as possible nodes. A Ray Casting technique is utilized in order for the robot to perceive the obstacles' edges. As ray detection method has not been expanded as much as the others, the ending point is stored as an obstacle edge, which is the node in the graph. When one node can connect to the other without the graph's edge intersecting an obstacle, the nodes' connections in graph is established.

• Rapidly exploring Random Trees RRTs

- **Standard RRT**

  a random point in the free space is selected. Then, the current constructed tree is checked in order for its closest node to the random point in free space to be found.

- **Star RRT**

  the algorithm aim to improve the method of Standard RRT due to highly random path.

- **Multiple RRTs**

  the algorithm aim to speed the convergence of Standard RRT

• Generalized Voronoi Diagram

Generalized Voronoi Diagram is constructed by a set of points in the free space which abstain evenly from the environment's obstacle if the Manhattan distance is taken into account as metric.

• A* Planning

the algorithm determines the path based on the path cost and cost estimation required to extend the path until the goal using the cost function.

Each of the algorithms has their own advantages and disadvantages considered on the computation time and robustness.(Ducho et al., 2014) (Tsardoulias et al., 2013)

3. RESEARCH METHODOLOGY

The experiment of path planning and control for the mobile robot consists of many steps. First, the mobile robot model and sensor model are built to use inside the simulation software. Second, using the ROS packages Hector SLAM and Gmapping SLAM, the occupancy grid map is built. Third, after the occupancy grid map is built, the A* path planning algorithm is applied to get the trajectory from the start point to the goal point. Finally, the robot movement is controlled to the reference trajectory using backstepping controller algorithm and tuning parameters for smooth motion. **Figure 3.5.** presents the overall workflow of the project.

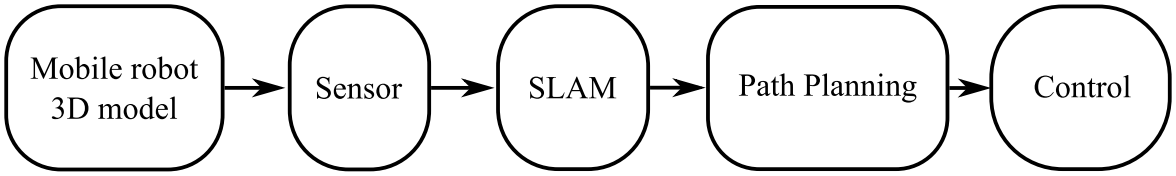

Figure 3.5. Workflow

3.1. Mobile Robot 3D Modelling

The robot model in constructed in Gazebo simulation. Gazebo uses the robot model written in SDF format. We construct the robot model with the parameters below:

• **Robot Base** (**Figure 3.6.**)

The model base dimensions are:

- Length = 0.4 *m*
- Width = 0.2 *m*
- Height = 0.1 *m*

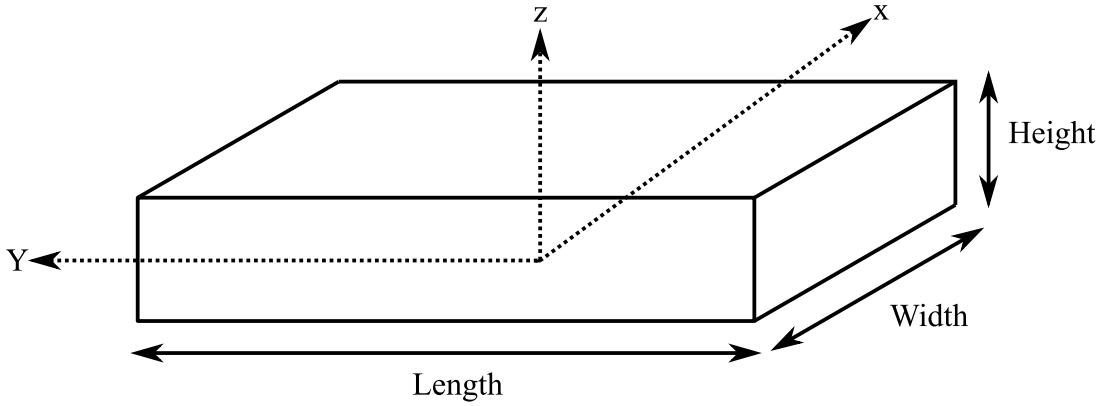

Figure 3.6. Mobile Robot Base

• **Wheels and Caster Wheels** (**Figure 3.7.**)

The robot wheels and caster wheel dimensions are:

- Wheel radius = 0.1 *m*
- Wheel width = 0.05 *m*
- Caster wheel radius = 0.05 *m*

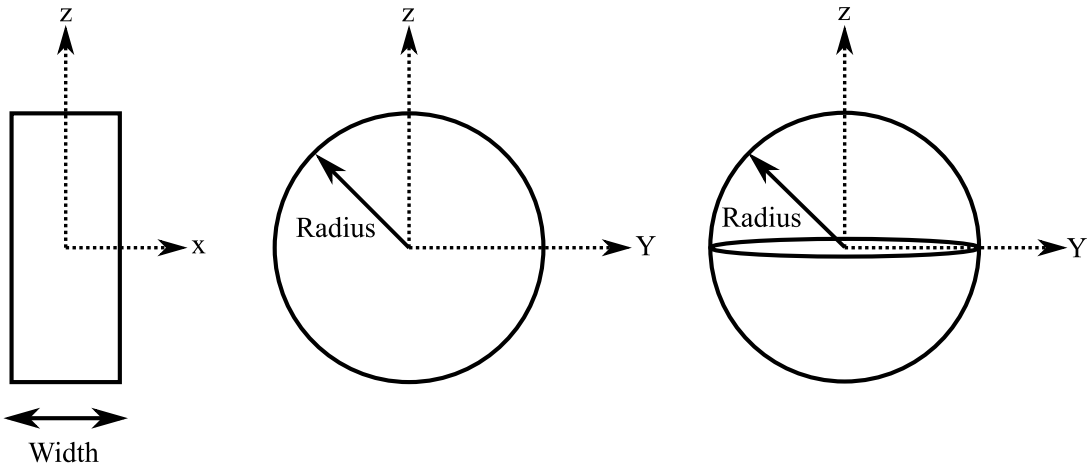

Figure 3.7. Mobile Robot Wheel and Caster Wheel

3.2. Sensor 3D Modelling

The robot is equipped with Lidar sensor, with the max range of 10 *m*, min range of 0.1 *m*, 360° field of view, and resolution of 0.1°. The IMU is attached to the robot frame with the z-axis pointing upward. The sensor noise is included in the simulation with the model of zero-mean Gaussian white. **Figure 3.10.** illustrates the mobile robot model side and top view.

• **Lidar** (**Figure 3.8.**)

The Lidar dimensions are:

- Diameter = 0.065 *m*
- Height = 0.070 *m*
- Height to Laser = 0.055 *m*

• **IMU** (**Figure 3.9.**)

The IMU dimensions are:

- Length = 0.02 *m*
- Width = 0.01 *m*
- Height = 0.01 *m*

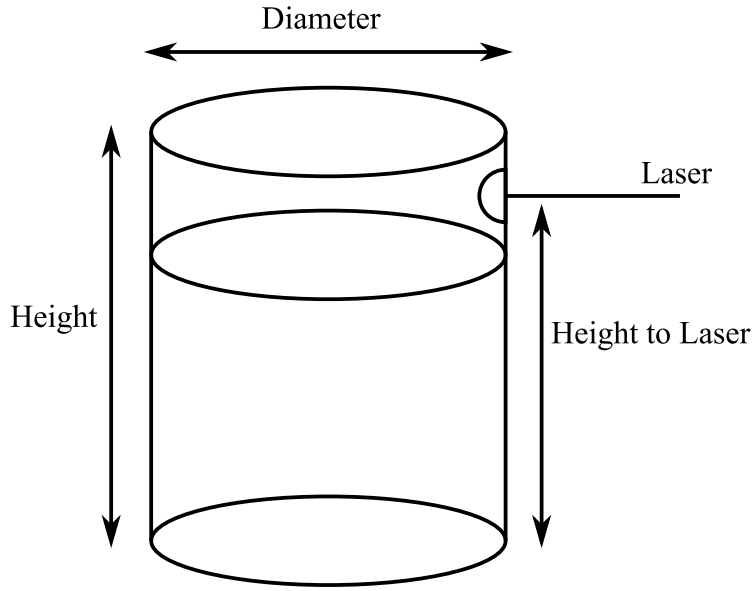

Figure 3.8. Lidar Dimension

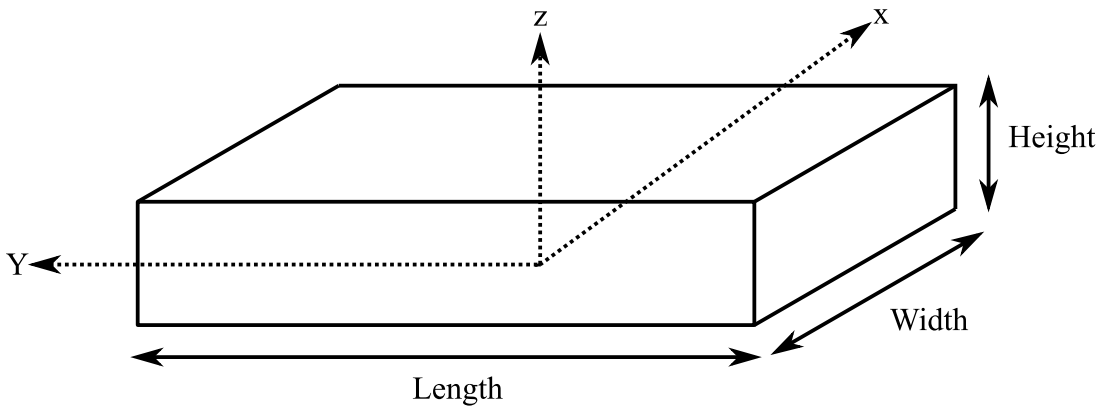

Figure 3.9. IMU Dimension

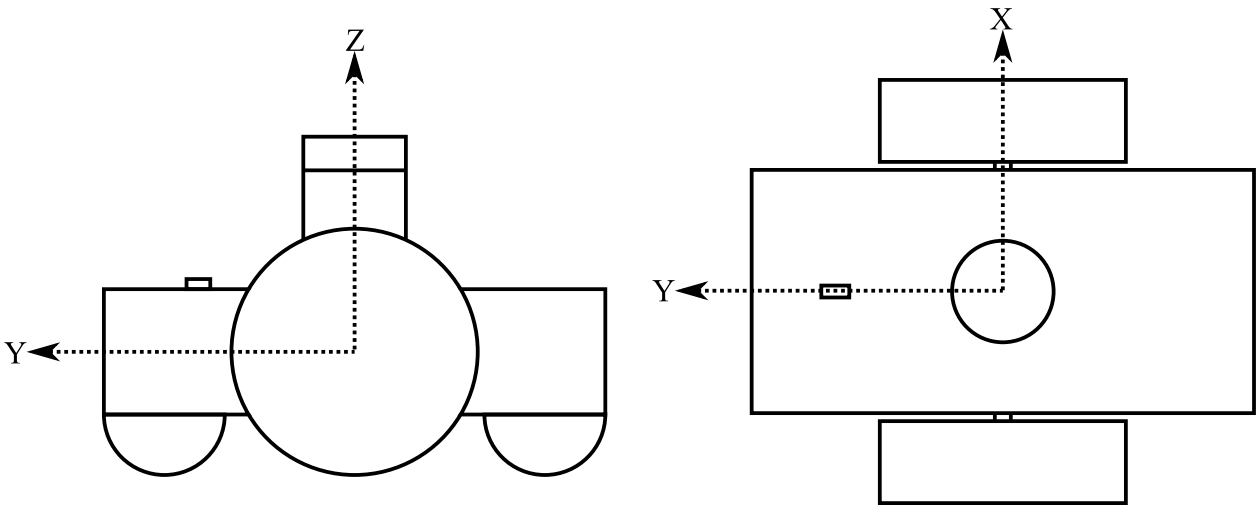

Figure 3.10. Mobile Robot Modelling Side and Top View

3.3. SLAM

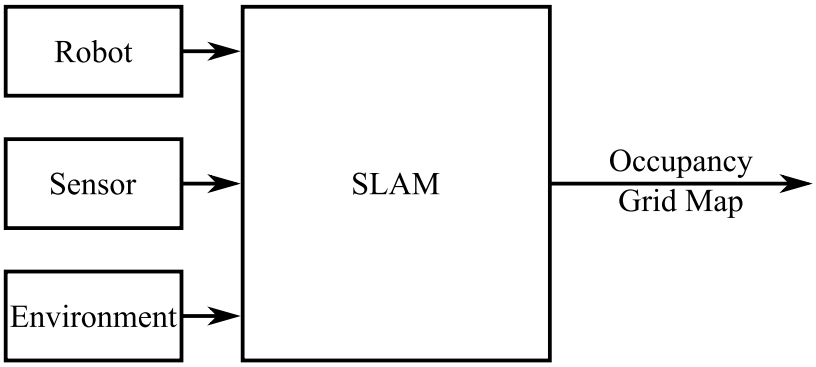

Figure 3.11. SLAM

Figure 3.11., the SLAM method is used to create the Occupancy Grid Map. In Gazebo, first we create a simulation of the indoor environment. Second, we load the robot model and sensor from the SDF file that is created in the Gazebo. Using ROS plug-in for Gazebo, the robot is controlled with the keyboard input. The robot is driven around while the SLAM is running to create the Occupancy Grid Map of the environment.

3.4. Path Planning and Control

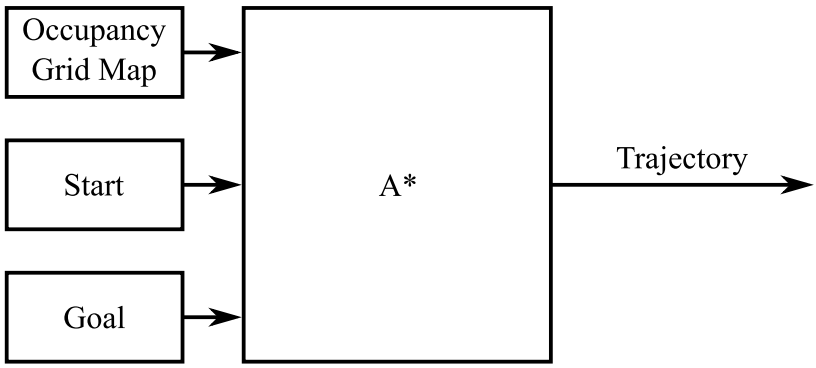

Figure 3.12. Path Planning of Astar

Figure 3.12., given the Occupancy Grid Map, the path planning algorithm is used for the calculation of the reference trajectory for the robot to move. The reference trajectory then is given to the robot which is controlled by the backstepping trajectory tracking algorithm. The control input from the backstepping is publishing on to the ROS control topic to determine the torque of each wheel of the robot to move. In every time-step, the robot pose is estimate using the extended kalman filter sensor fusion algorithm. The pose of the robot is plotted against the reference pose.

Figure 3.13. and **Figure 3.14.** are the model of simulated rooms for experiment and its dimension. In this research, we use 2 simulated room model and named it: "Map1" and "Map2". In these 2 figures, the yellow spot represents the Start location for the robot, the green spot represents the goal location where the robot will move to.

• **Indoor 3D models: "Map1"**

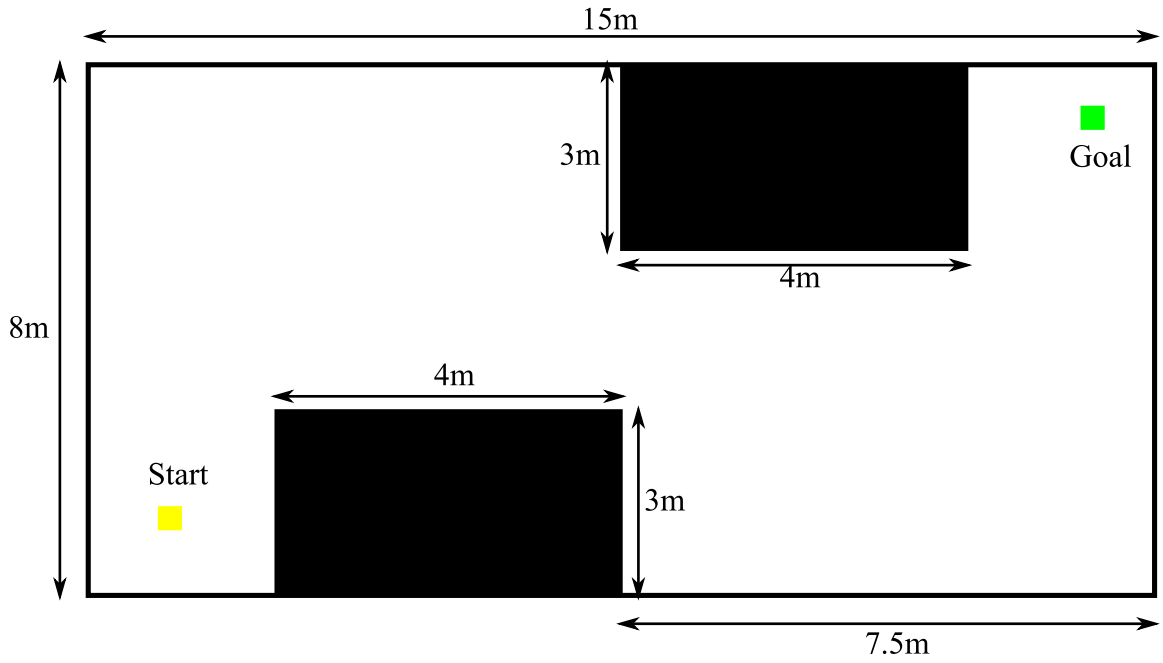

Figure 3.13. "Map1" Model

• **Indoor 3D models: "Map2"**

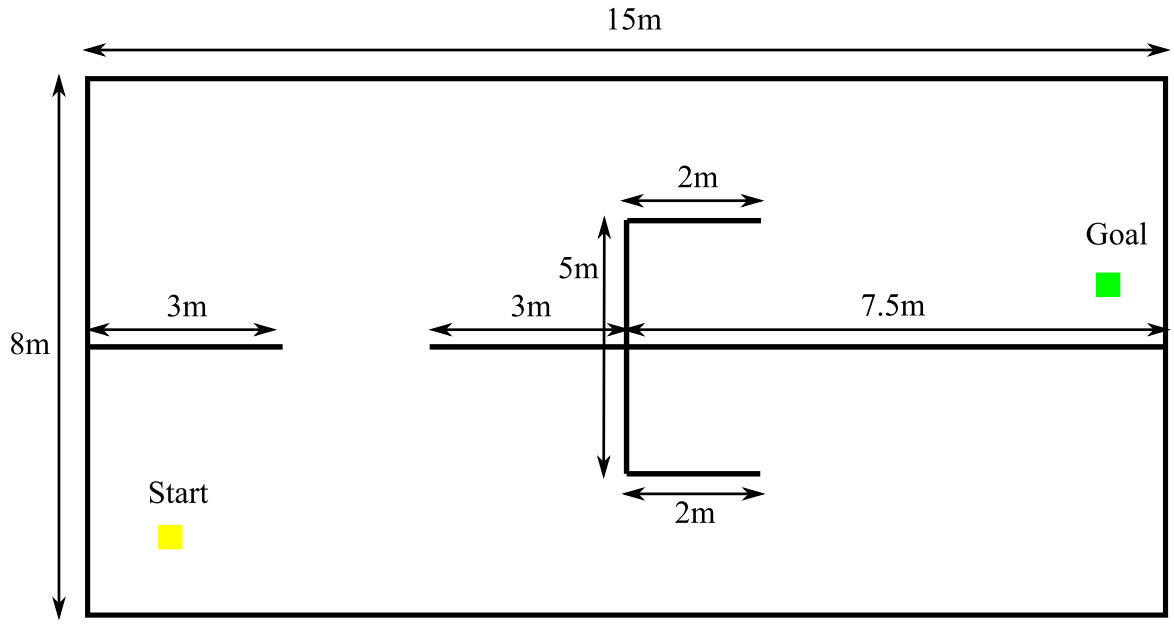

Figure 3.14. "Map2" Model

4. DIFFERENTIAL DRIVE MOBILE ROBOT MODELING

4.1. Mobile Robot Kinematic

Kinematic model of the robot describes the transformation of the robot velocities in the local frame to the velocity in global frame. Using kinematic model, the robot pose is determined and represented in the coordinate system.

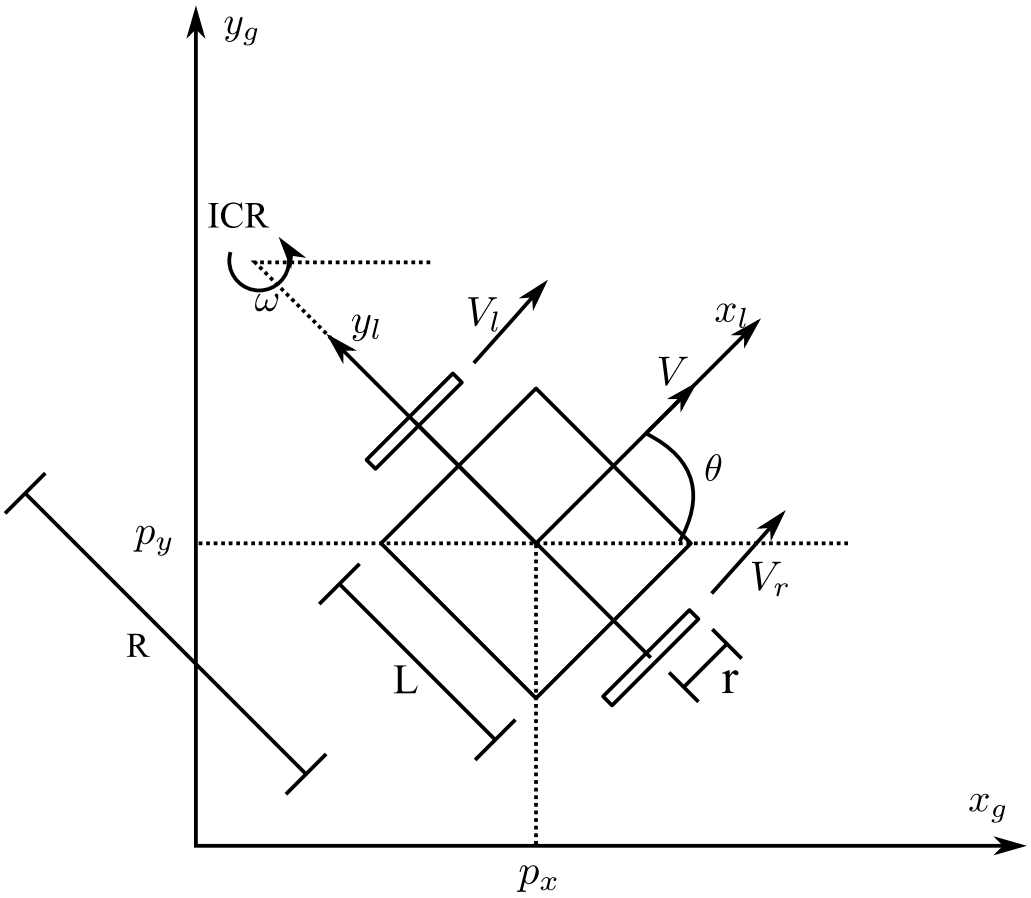

Figure 4.15. Differential Drive Kinematic Model

In **Figure 4.15.** Let:

- r is the radius of the wheel
- $R(t)$ is the instantaneous radius of the vehicle driving trajectory
- L is the length of the robot base
- V_l is the linear velocity of the left wheel
- V_r is the linear velocity of the right wheel
- V is the linear velocity of the robot base
- ω is the angular velocity of the robot base

From **Figure 4.15.**, we obtain:

$$\omega = \frac{V_l(t)}{R(t) - \frac{L}{2}} \quad \textbf{(Eq. 4.1.)}$$

$$\omega = \frac{V_r(t)}{R(t) + \frac{L}{2}} \quad \textbf{(Eq. 4.2.)}$$

From **Eq. 4.1.** and **Eq. 4.2.**, we get:

$$\omega(t) = \frac{V_r(t) - V_l(t)}{L} \quad \textbf{(Eq. 4.3.)}$$

$$R(t) = \frac{L}{2}\frac{V_r(t) + V_l(t)}{V_r(t) - V_l(t)} \quad \textbf{(Eq. 4.4.)}$$

We have:

$$V(t) = R(t)\omega(t) \quad \textbf{(Eq. 4.5.)}$$

Substitute **Eq. 4.3.** and **Eq. 4.4.** to **Eq. 4.5.**, we get:

$$V(t) = \frac{V_r(t) + V_l(t)}{2} \quad \textbf{(Eq. 4.6.)}$$

We know:

$$\begin{aligned} V_l(t) &= r\omega_l(t) \\ V_r(t) &= r\omega_r(t) \end{aligned} \quad \textbf{(Eq. 4.7.)}$$

Substitute **Eq. 4.7.** to **Eq. 4.5.** and **Eq. 4.3.**, we get:

$$V(t) = \omega_r\frac{r}{2} + \omega_l\frac{r}{2} \quad \textbf{(Eq. 4.8.)}$$

$$\omega(t) = \omega_r\frac{r}{L} - \omega_l\frac{r}{L} \quad \textbf{(Eq. 4.9.)}$$

Robot kinematic model in global frame is defined as:

$$\begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} V(t) \\ \omega(t) \end{bmatrix} \quad \textbf{(Eq. 4.10.)}$$

Using Euler integration, the equation of motion in discrete time form is:

$$\begin{bmatrix} p_{x,k} \\ p_{y,k} \\ \theta_k \end{bmatrix} = \begin{bmatrix} p_{x,k-1} + V_{k-1} T_s cos(\theta_{k-1}) \\ p_{y,k-1} + V_{k-1} T_s sin(\theta_{k-1}) \\ \theta_{k-1} + \omega_{k-1} T_s \end{bmatrix} \quad \textbf{(Eq. 4.11.)}$$

Where:

- k is the time step
- T_s is the time interval

4.2. Mobile Robot Dynamics

Differential drive mobile robot is a dynamics system. Using solely the kinematic model of the system is not enough to represent the system as a whole. For the robustness of the robot, the dynamics properties of the system such as external force, mass, inertia are considered. In our case the robot center of mass is coincides with the center geometric of the robot. Let m be the mass of the robot, J be the moment of inertia of the robot about Z-axis. In this section, we use q to describe the generalized coordinate of the system $q = [x, y, \theta]$ and τ_l is the left wheel torque, τ_r is the right wheel torque.

The dynamics model of the robot with constraint is derived using Lagrange formulation:

$$\frac{\partial}{\partial t}(\frac{\partial \mathscr{L}}{\partial \dot{q}_k}) - \frac{\partial \mathscr{L}}{\partial q_k} + \frac{\partial P}{\partial \dot{q}_k} + g_k + \tau_{dk} = f_k - \sum_{j=1}^{m} \lambda_j a_{jk} \quad \textbf{(Eq. 4.12.)}$$

Where:

- $\mathscr{L}$ is the Lagrangian
- P is the power dissipation function due to friction and damping
- g_k are the forces due to gravitation
- τ_{dk} are the system disturbances
- f_k are the general forces (external influences to the system)
- q_k general coordinate k=(1,...,n)
- m is the number of linearly independent motion constraint
- λ_j is the Lagrange multiplier associated with the jth constraint relation
- a_{jk} is coefficients of the constraints (j=1,...,n)

Assumption:

- $\mathscr{W}_p$ = 0, the robot is driven on the plane where the potential energy is constant
- gk = 0, the robot is driven on the plane where the potential energy is constant
- τ_{dk} = 0, no outside disturbances

The Lagrangian $\mathscr{L}$ is the difference between kinetic energy and potential energy. We get:

$$\mathscr{L} = \mathscr{W}_k - \mathscr{W}_p \quad \textbf{(Eq. 4.13.)}$$

Where:

- $\mathscr{W}_k$ is the kinetic energy of the system
- $\mathscr{W}_p$ is the potential energy of the system

The Kinetic energy equation is:

$$\mathscr{W}_k = \frac{1}{2}mV^2 + \frac{1}{2}J\dot{\theta}^2 \quad \textbf{(Eq. 4.14.)}$$

The velocity in the 2D plane is:

$$V^2 = \dot{x}^2 + \dot{y}^2 \quad \textbf{(Eq. 4.15.)}$$

We get the Lagrangian $\mathscr{L}$:

$$\mathscr{L} = \frac{m}{2}(\dot{x}^2 + \dot{y}^2) + \frac{J}{2}\dot{\theta}^2 \quad \textbf{(Eq. 4.16.)}$$

Substitute back to **Eq. 4.12.**, we get:

$$\frac{\partial}{\partial t}(\frac{\partial \mathscr{L}}{\partial \dot{x}}) = m\ddot{x}$$

$$\frac{\partial}{\partial t}(\frac{\partial \mathscr{L}}{\partial \dot{y}}) = m\ddot{y}$$

$$\frac{\partial}{\partial t}(\frac{\partial \mathscr{L}}{\partial \dot{\theta}}) = J\ddot{\theta}$$

$$\frac{\partial \mathscr{L}}{\partial x} = 0$$

$$\frac{\partial \mathscr{L}}{\partial y} = 0$$

$$\frac{\partial \mathscr{L}}{\partial \theta} = 0$$

The mobile robot constraint in x-axis is $-sin\theta$, in y-axis is $cos\theta$.

From **Eq. 4.12.**, we obtain:

$$\begin{aligned} m\ddot{x} &= F_x - \lambda_1(-sin\theta) \\ m\ddot{y} &= F_y - \lambda_1(cos\theta) \\ J\ddot{\theta} &= M_z \end{aligned} \qquad \textbf{(Eq. 4.17.)}$$

We get:

$$\begin{aligned} m\ddot{x} - \lambda_1 sin\theta &= F_x \\ m\ddot{y} + \lambda_1 cos\theta &= F_y \\ J\ddot{\theta} &= M_z \end{aligned} \qquad \textbf{(Eq. 4.18.)}$$

Since the assumption of no outside disturbance force, the force acting on the robot are the left wheel force F_l and the right wheel force F_r. The resultant for is:

$$F = F_r + F_l \qquad \textbf{(Eq. 4.19.)}$$

We have:

$$\begin{aligned} F_r &= \frac{\tau_r}{r} \\ F_l &= \frac{\tau_l}{r} \end{aligned} \qquad \textbf{(Eq. 4.20.)}$$

And

$$\begin{aligned} F_x &= F cos\theta \\ F_y &= F sin\theta \\ M_z &= F_r\frac{L}{2} - F_l\frac{L}{2} \end{aligned} \qquad \textbf{(Eq. 4.21.)}$$

We get:

$$\begin{aligned} F_x &= \frac{1}{r}(\tau_r + \tau_l)cos\theta \\ F_y &= \frac{1}{r}(\tau_r + \tau_l)sin\theta \\ M_z &= \frac{L}{2r}(\tau_r - \tau_l) \end{aligned} \qquad \textbf{(Eq. 4.22.)}$$

We get:

$$\begin{aligned} m\ddot{x} - \lambda sin\theta - \frac{1}{r}(\tau_r + \tau_l)cos\theta = 0 \\ m\ddot{y} + \lambda cos\theta - \frac{1}{r}(\tau_r + \tau_l)sin\theta = 0 \\ J\ddot{\theta} - \frac{L}{2r}(\tau_r - \tau_l) = 0 \end{aligned} \quad \textbf{(Eq. 4.23.)}$$

Rewrite the equation into Matrix form of:

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

We get:

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

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

$$A = \begin{bmatrix} -sin\theta & cos\theta & 0 \end{bmatrix}$$

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

Where remain are Zero

In the state-space model:

$$\tilde{M} = \begin{bmatrix} m & 0 \\ 0 & j \end{bmatrix}$$

$$\tilde{V} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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

The model is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{V} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} Vcos\theta \\ Vsin\theta \\ \omega \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{1}{mr} & \frac{1}{mr} \\ \frac{L}{2Jr} & -\frac{L}{2Jr} \end{bmatrix} \begin{bmatrix} \tau_r \\ \tau_l \end{bmatrix} \quad \textbf{(Eq. 4.24.)}$$

5. CONTROL DIFFERENTIAL DRIVE MOBILE ROBOT

There are multiple ways to control the mobile robot, namely, reference position control and reference trajectory control etc. In this project, the robot movement is control by the trajectory tracking control based on the back-stepping technique.

5.1. Kinematic Control Model

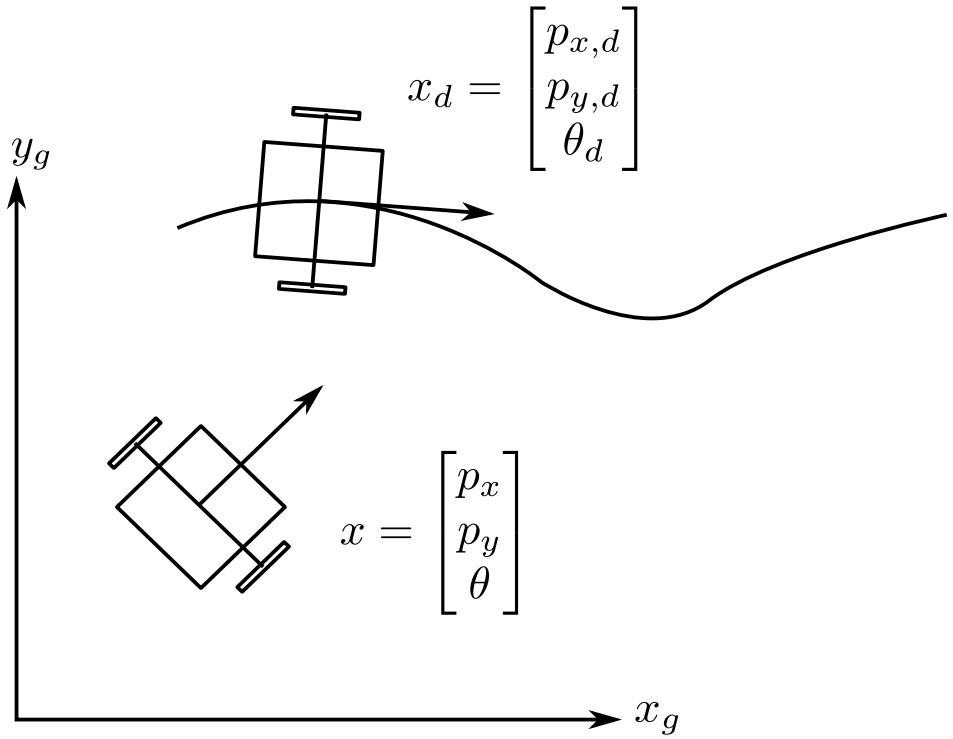

Figure 5.16. Differential Drive Kinematic control

Figure 5.16. shows the robot pose inside the environment. Let :

- x represents the current pose of the robot in a time step
- x_d represents the desired trajectory of the robot

The purpose of the back-stepping trajectory tracking control controller is to nullify the error between the x and x_d. The error between the current pose and the desired trajectory pose is defined as:

$$x_e = T\tilde{x} \qquad \textbf{(Eq. 5.1.)}$$

Where:

$$x_e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \quad \textbf{(Eq. 5.2.)}$$

$$T = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(Eq. 5.3.)}$$

$$\tilde{x} = \begin{bmatrix} p_{x,d} - p_x \\ p_{y,d} - p_y \\ \theta_d - \theta \end{bmatrix} \quad \textbf{(Eq. 5.4.)}$$

Taking the derivative of **Eq. 5.1.**, we obtain:

$$\dot{x}_e = \begin{bmatrix} \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} V + \begin{bmatrix} e_2 \\ -e_1 \\ -1 \end{bmatrix} \omega + \begin{bmatrix} V_{ref} cose_3 \\ V_{ref} sine_3 \\ \omega_{ref} \end{bmatrix} \quad \textbf{(Eq. 5.5.)}$$

Proving by the Lyapunov stability (Zidani et al., 2015), the controllers produce the control input for the robot are:

$$\begin{bmatrix} V_c \\ \omega_c \end{bmatrix} = \begin{bmatrix} V_{ref} cose_3 + k_1 e_1 \\ \omega_{ref} + k_2 V_{ref} e_2 + k_2 sine_3 \end{bmatrix} \quad \textbf{(Eq. 5.6.)}$$

Where:

- k_1, k_2, k_3 are the positive constants for tuning
- V_c is a linear velocity control
- ω_c is an angular velocity control

5.2. Dynamics Control Model

Using the dynamics model of the robot, reformulate in terms of tracking control. By using coordinate transformation. Let:

$$\begin{aligned} x_1 &= V \\ x_2 &= \omega \\ u_V &= \tau_1 + \tau_2 \\ u_\omega &= \tau_1 - \tau_2 \end{aligned} \qquad \textbf{(Eq. 5.7.)}$$

Where:

- $(x_1, x_2)^T$ is the state vector
- $(u_V, u_\omega)^T$ is the control vector

We get:

$$\dot{x_1} = \frac{1}{mr} u_V \qquad \textbf{(Eq. 5.8.)}$$

$$\dot{x_2} = \frac{L}{2rI} u_\omega \qquad \textbf{(Eq. 5.9.)}$$

There are two steps in this approach:

• **Step 1**:

Let x_{1ref} be the linear velocity of the reference trajectory. The linear velocity tracking error is defined as:

$$z_1 = x_{1ref} - x_1 \qquad \textbf{(Eq. 5.10.)}$$

Taking the derivative of z_1, we get:

$$\dot{z}_1 = \dot{x}_{1ref} - \dot{x}_1 = \dot{x}_{1ref} - \frac{1}{mr} u_V \qquad \textbf{(Eq. 5.11.)}$$

From the Lyapunov function stability, we get:

$$u_V = mr[\dot{x}_{1ref} + k_a z_1] \qquad \textbf{(Eq. 5.12.)}$$

• **Step 2**:

Let x_{2ref} be the angular velocity of the reference trajectory. The angular velocity tracking error is defined as:

$$z_2 = x_{2ref} - x_2 \quad \textbf{(Eq. 5.13.)}$$

Taking the derivative of z_2, we get:

$$\dot{z}_2 = \dot{x}_{2ref} - \dot{x}_2 = \dot{x}_{2ref} - \frac{L}{2rI} u_\omega \quad \textbf{(Eq. 5.14.)}$$

From the Lyapunov function stability, we get:

$$u_\omega = \frac{2rI}{L}[\dot{x}_{2ref} + k_b z_2] \quad \textbf{(Eq. 5.15.)}$$

We have:

$$\tau_{1c} = \frac{1}{2}[u_V + u_\omega] \quad \textbf{(Eq. 5.16.)}$$

$$\tau_{2c} = \frac{1}{2}[u_V - u_\omega] \quad \textbf{(Eq. 5.17.)}$$

Substitute **Eq. 5.12.** and **Eq. 5.15.** into **Eq. 5.16.** and **Eq. 5.17.**, we get:

$$\tau_{1c} = \frac{1}{2}[mr[\dot{x}_{1ref} + k_a z_1]] + \frac{2rI}{L}[\dot{x}_{2ref} + k_b z_2]] \quad \textbf{(Eq. 5.18.)}$$

$$\tau_{2c} = \frac{1}{2}[mr[\dot{x}_{1ref} + k_a z_1]] - \frac{2rI}{L}[\dot{x}_{2ref} + k_b z_2]] \quad \textbf{(Eq. 5.19.)}$$

Where:

- τ_{1c} is the control torque for the left wheel
- τ_{2c} is the control torque for the right wheel
- k_a is a positive constant
- k_b is a positive constant

5.3. Sensor Fusion

Sensor fusion is the method of combining the data from the sensor to determine the state of the system. One of the sensor fusion algorithms is Kalman Filter. It is an algorithm that estimates of the unknown or uncertain variable given the observation. The state of the system (in this case, the robot pose) is represented as vector with some mixed in noise (considered as ground truth). In every discrete time step, the new state is determined in the function of previous state and inputs using an operator. Then, another operator determines the measurable output (observation) from the true system. This algorithm can only apply to the linear system. Since the mobile robot is a nonlinear system, another variance of Kalman Filter is used called Extended Kalman Filter. (Al Khatib et al., 2015) (Moore and Stouch, 2016). The state transition model and the measurement model in EKF are defined as :

$$x_k = f(x_{k-1}, u_{k-1}) + w_{k-1} \quad \textbf{(Eq. 5.20.)}$$

$$y_k = h(x_k) + v_k \quad \textbf{(Eq. 5.21.)}$$

EKF is divided into two steps: the **Prediction step** and the **Correction step**.

Table 5.1. Extended Kalman Filter Algorithm (Kim and Bang, 2018)

Prediction	
Predicted State estimate	$\hat{x}_k^- = f(\hat{x}_{k-1}^-, u_{k-1})$
Predicted error co-variance	$P_k^- = JF_{k-1}P_{k-1}^+JF_{k-1}^T + Q$
Correction	
Measurement predict	$\hat{y}_k = h(\hat{x}_k^-)$
Measurement residual	$\tilde{y}_k = y_k - \hat{y}_k$
Kalman Gain	$K_k = P_k^- JH_k^T (R + JH_k P_k^- JH_k^T)^{-1}$
Updated state estimate	$\hat{x}_k^+ = \hat{x}_k^- + K_k\tilde{y}$
Updated error co-variance	$P_k^+ = (I - K_k JH_k)P_k^-$

The superscript - and + represents the variable prior and posterior to the estimate state in the time update and the measurement update, respectively.

From **Table 5.1.**,

- x is the state vector
- P is the error co-variance
- y is the measurement vector

- k is the time step
- u is the control input vector
- w is the guassian white noise
- v is the guassian white noise
- Q is the covariance matrix of prediction model
- R is the covariance matrix of measurement model
- JF_{k-1} is the Jacobian matrix the nonlinear state function

$$JF_{k-1} = \frac{\partial f}{\partial x}|_{\hat{x}_{k-1}^{+}, u_{k-1}}$$

- JH_k is the Jacobian matrix of the nonlinear measurement function

$$JH_k = \frac{\partial h}{\partial x}|_{\hat{x}_k^{-}}$$

• Overall sensor fusion localization

Using EKF, the robot pose $x = [p_x \quad p_y \quad \theta]^T$ is predicted and estimated. In the **Prediction step**, from the wheel encoder signal the V and the ω are calculated and subsequently are used to predict the robot state $\hat{x}_k^-$ and the error covariance P_k^-. In the **Correction step**, the robot pose $[p_x \quad p_y]$ is measured from the Scan Matching Lidar, and the robot orientation $[\theta]$ is measured from the continuously integrated angular velocity ω of the IMU.

• Prediction Step

In state space representation, the system is defined as:

$$x_k = Fx_{k-1} + Bu_{k-1} \qquad \textbf{(Eq. 5.22.)}$$

Where:

- F is the state transition matrix
- B is the control input matrix
- x is the state vector
- u is the control input vector

• Wheel Encoder

Wheel encoder signal ω_l and ω_r are used to calculate the u and x.

We have:

$$u_{k-1} = \begin{bmatrix} V_{k-1} \\ \omega_{k-1} \end{bmatrix} \quad \textbf{(Eq. 5.23.)}$$

And

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{bmatrix} \begin{bmatrix} \omega_r \\ \omega_l \end{bmatrix} \quad \textbf{(Eq. 5.24.)}$$

Rewrite the **Eq. 4.10.**, into the state space representation as the **Eq. 5.22.** form, we have:

$$\begin{bmatrix} p_{x,k} \\ p_{y,k} \\ \theta_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{x,k-1} \\ p_{y,k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} cos(\theta_{k-1})T_s & 0 \\ sin(\theta_{k-1})T_s & 0 \\ 0 & T_s \end{bmatrix} \begin{bmatrix} V_{k-1} \\ \omega_{k-1} \end{bmatrix} \quad \textbf{(Eq. 5.25.)}$$

We get:

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(Eq. 5.26.)}$$

$$B = \begin{bmatrix} cos(\theta_{k-1})T_s & 0 \\ sin(\theta_{k-1})T_s & 0 \\ 0 & T_s \end{bmatrix} \quad \textbf{(Eq. 5.27.)}$$

• Jacobian Matrix JF

From **Eq. 4.10.**, we have:

$$\begin{aligned} \dot{p_x} &= V cos\theta \\ \dot{p_y} &= V sin\theta \\ \dot{\theta} &= \omega \end{aligned} \quad \textbf{(Eq. 5.28.)}$$

We have:

$$Jacobian = \begin{bmatrix} \frac{\partial p_x}{\partial p_x} & \frac{\partial p_x}{\partial p_y} & \frac{\partial p_x}{\theta} \\ \frac{\partial p_y}{\partial p_x} & \frac{\partial p_y}{\partial p_y} & \frac{\partial p_y}{\theta} \\ \frac{\partial \theta}{\partial p_x} & \frac{\partial \theta}{\partial p_y} & \frac{\partial \theta}{\partial \theta} \end{bmatrix} \quad \textbf{(Eq. 5.29.)}$$

Substitute **Eq. 5.28.** into **Eq. 5.29.**, we get the Jacobian Matrix of F (JF):

$$JF = \begin{bmatrix} 1 & 0 & -V_{k-1}T_s sin(\theta_{k-1}) \\ 0 & 1 & V_{k-1}T_s cos(\theta_{k-1}) \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(Eq. 5.30.)}$$

• **Correction Step**

The measurement prediction:

$$\hat{y}_k = Hx_k \quad \textbf{(Eq. 5.31.)}$$

Where

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(Eq. 5.32.)}$$

• Sensor Correction Measurement y_k

• Lidar

Lidar measures distances and angles to the obstacle. From Scan Matching Lidar, the robot pose p_x, p_y is measured. Thus, the measurement model of the Lidar is:

$$\begin{aligned} y_{1,k} &= p_{x,k} \\ y_{2,k} &= p_{y,k} \end{aligned} \quad \textbf{(Eq. 5.33.)}$$

• IMU

IMU use Gyroscope to measure the angular velocity of the robot. From continuously integrating gyroscope data around Z-axis, the heading angle of the robot is measured. Thus, the measurement model of the IMU is :

$$y_{3,k} = \omega T_s \quad \textbf{(Eq. 5.34.)}$$

• Jacobian Matrix (JH)

$$JH = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \textbf{(Eq. 5.35.)}$$

Figure 5.17. shows the overall Sensor Fusion Localization and Control Architecture of the whole system.

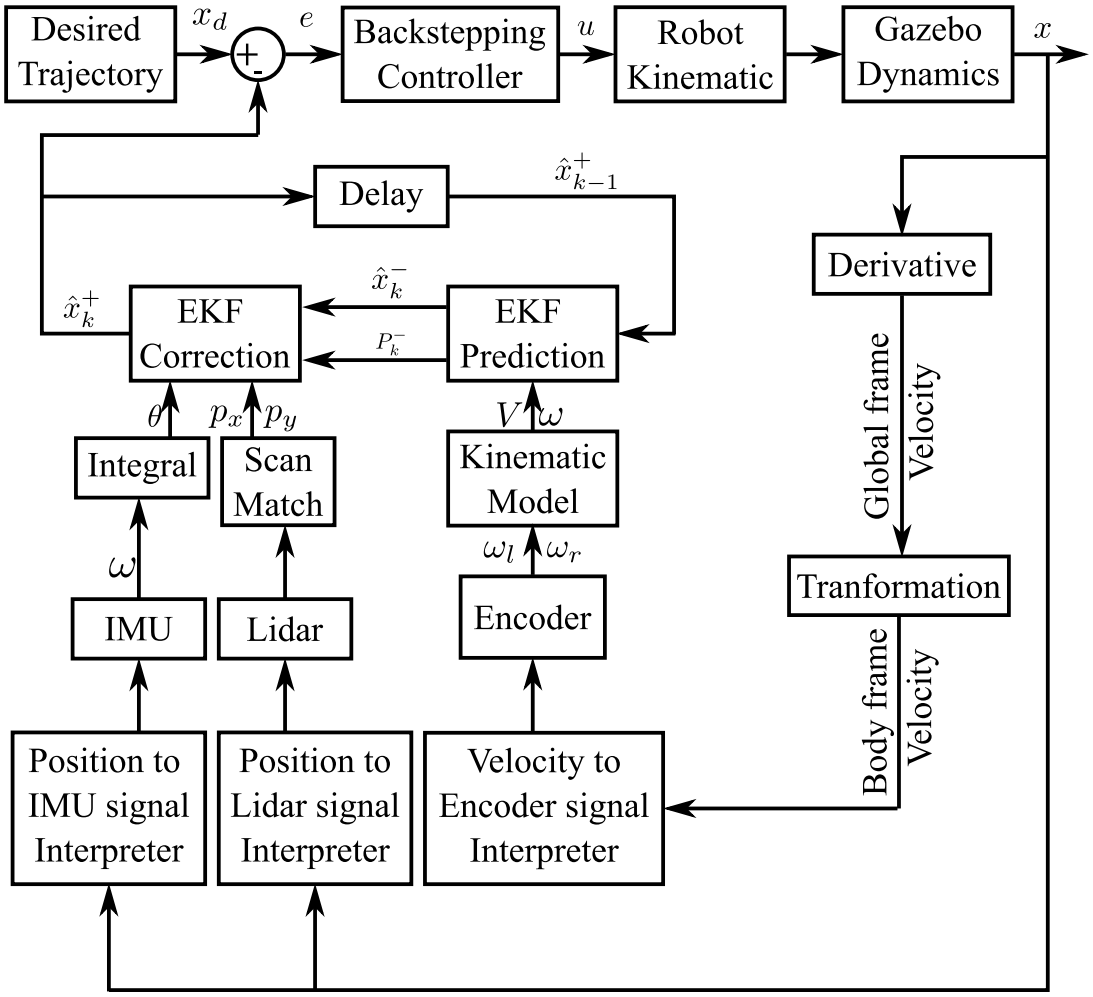

Figure 5.17. Sensor Fusion Localization and control architecture

6. PATH PLANNING

6.1. Robotic Operating System (ROS)

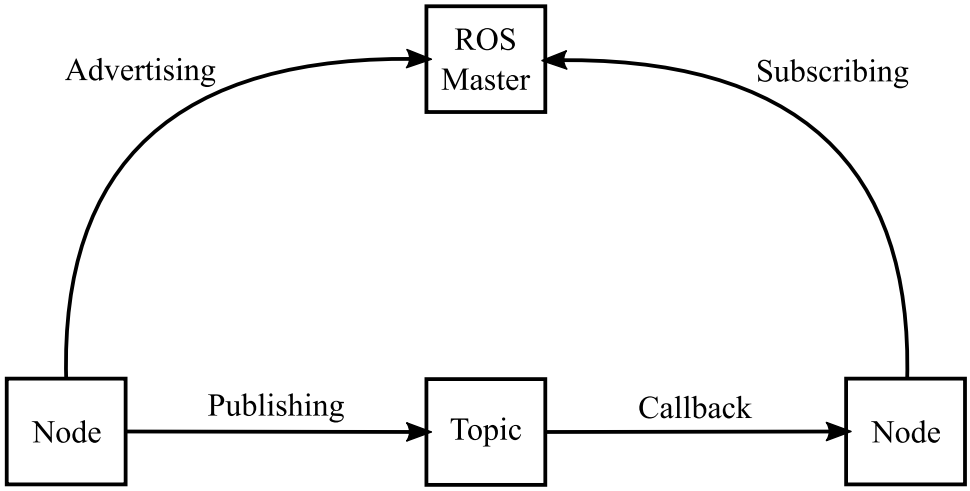

Figure 6.18. ROS Framework

Robotic operating system (ROS) is an open source framework developed for robotic purposes. It contains libraries and packages that are already built and ready to use for robots. ROS is a peer-to-peer network of processes that could run on multiple devices that are connected via network.(ROS, 2021a)

Figure 6.18., ROS center of communication is ROS Master. ROS master acts as a keeper of topics and services, registration and information of ROS nodes. ROS nodes are the process of performing the computation. It publishes or subscribes ROS messages with other nodes via ROS topics. ROS message is a data that has been simplified into a structure.

6.1.1. Odometry Message

Odometry is the information of robot position and velocity within the environment. In 3D coordinate system, the robot position is represented as $[p_x \quad p_y \quad p_z]^T$ and the orientation of the robot is represented as roll, pitch, yaw in Euler angle representation. As our robot is in 2D planar motion, thus we only interested in $[p_x \quad p_y \quad \theta]^T$. When using the ROS message, the odometry directly publishes with the position and orientation of the robot from the simulated environment. The position is expressed as $[p_x \quad p_y]$ and the orientation in the quaternion form (qw, qx, qy, qz).

In this project the Odometry Message is used to contain the data of:

- Lidar scan matching
- Wheel encoder odometry
- EKF fusion pose
- Robot true pose
- Robot current pose

Table 6.2. Odometry Properties (ROS, 2021e)

ROS message	Definition
Header	Timestamp and frame id
Child frame id	
Geometry_msgs/PoseWithCovariance	Estimation of WMR Position in free space with uncertainty
Geometry_msgs/Pose	Contain the information of the position x,y,z and orientation in quaternion form (x,y,z,w)
Geometry_msgs/TwistWithCovariance	Estimation of WMR Velocity in free space with uncertainty
Geometry_msgs/Twist	Contain the information of velocity in linear and angular

Table 6.2., In the ROS Odometry properties, In global frame,

• **Robot poses is:**

- Odometry.Pose.Pose.Position.x = p_x
- Odometry.Pose.Pose.Position.y = p_y
- Odometry.Pose.Pose.Orientation.(qw,qx,qy,qz) = θ

• **Robot velocity is:**

- Odometry.twist.twist.linear.x = $\dot{p}_x$
- Odometry.twist.twist.linear.y = $\dot{p}_y$
- Odometry.twist.twist.angular.z = ω

6.1.2. IMU Message

IMU is a module that consists of three different types of sensors (triaxial). Those sensors are accelerometer, gyroscopes, and magnetometer, which measure the acceleration, the angular velocity, and the magnetic field, respectively. The IMU ROS message are used to show the data of velocity, acceleration, and orientation of a system in quaternion form.

In this project the IMU Message is used to contain the data of:

- Robot Estimated Orientation
- Robot linear and angular velocity

Table 6.3. IMU Properties (ROS, 2021b)

ROS message	Definition
Header	Timestamp and frame id
Geometry_msgs/Quaternion	Estimation of WMR orientation in quaternion form
Geometry_msgs/Vector3	Contain the information of WMR angular velocity
Geometry_msgs/Vector3	Contain the information of WMR linear acceleration

Table 6.3., In the ROS IMU properties,

• **IMU data is:**

- IMU.Orientation.(qw,qx,qy,qz) = θ
- IMU.angular_velocity = simulated angular velocity in (x,y,z)
- IMU.linear_acceleration = simulated linear velocity in (x,y,z)

6.1.3. Twist Message

In ROS, twist is the message that carries the information of the velocity of the system in free space. The velocity of the system is decomposed into linear velocity and angular velocity. Furthermore the Twist message is used to publish the message for the mobile robot controller. In this project the IMU Message is used to contain the data of:

- Robot linear velocity along x-axis
- Robot angular velocity about z-axis

Table 6.4. Twist Properties (ROS, 2021f)

ROS message	Definition
Geometry_msgs/Vector3	Contain the information of WMR linear velocity
Geometry_msgs/Vector3	Contain the information of WMR angular velocity

Table 6.4., In the ROS Twist properties, In local frame,

• **Robot velocity is:**

- Twist.linear.x = V
- Twist.angular.z = ω

6.1.4. LIDAR Message

Lidar is a remote sensing device that uses light pulses to detect the distance from an object and has been widely used for multipurposes including navigation and mapping. Lidar usually contains a laser scanner and DC motor. The DC motor rotates the laser scanner in a 360 degree circle to obtain a full 360 degrees of the environment. **Table 6.5.**

In the project, the Lidar data is used to create the occupancy grid map and robot localization in the sensor fusion section. Using the Scan Matching algorithm, the robot pose is measured and update in the EKF fusion.

Table 6.5. Lidar Properties (ROS, 2021c)

ROS message	Definition
header	Timestamp and frame id
angle_min	Started angle of scan
angle_max	End angle of scan
angle_increment	Angular distance between scan to scan
time_increment	Time between one full scan to one full scan
scan_time	Time between scan to scan
range_min	Minimum range
range_max	Maximum range
ranges	Range data in one full scan
intensities	Intensity data

6.1.5. Simulation, Visualization and Robot Model URDF

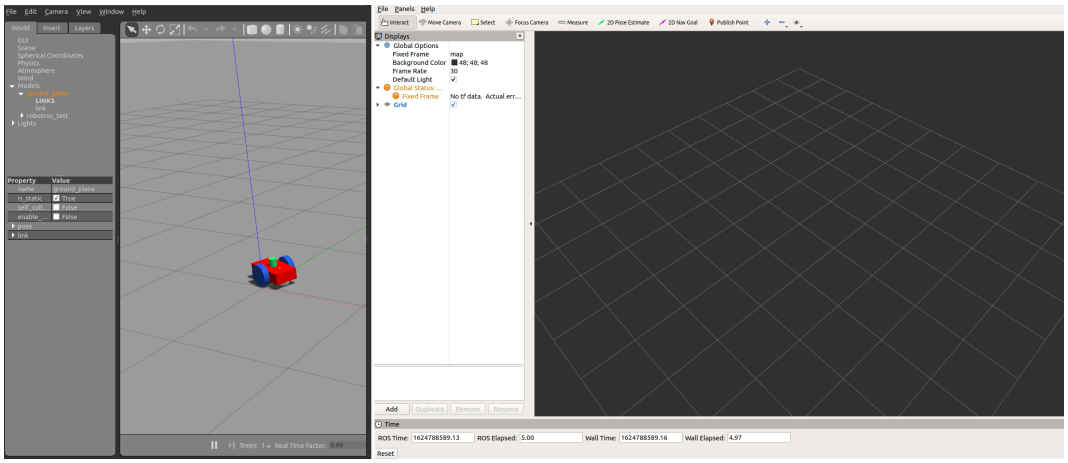

Figure 6.19. Gazebo and Rviz Window

Figure 6.19., the Gazebo robot simulation software will be used to simulate the robot, sensor and the surrounding environment. The RVIZ is used as a visualization tool. RVIZ allows

us to visualize and verify the incoming data from the ROS messages. In this project, RVIZ is used to visualize the Occupancy Grid Map, Robot Model, Robot trajectory -etc.

6.2. Path Planning A*

A* algorithm is one of the path search algorithm that is widely used in many fields. In the mobile robotic field, the algorithm is used for searching the path in a robot navigation task. In this project, the A* algorithm is used to search the path for the mobile robot. The information that is given to the algorithm are the start point (robot current position), goal point (desired point to move to), and the occupancy grid map (which contain the free space and the occupied space of the environment).

6.2.1. Occupancy Grid Map

Occupancy grid map is one of many pieces of information that are required for the mobile robot for navigation tasks such as path planning, navigation, environment map, and localization.

Occupancy grid map represents the environment in square grid cells. Each of grid cells is represented as either an occupied cell or a free cell according to the calculation of the binary probability value. **Figure 6.20.**, Occupancy Grid Map is a 2D map is a large set that contains a probability value in every cell. The cell representation as:

- **Occupied cell** by probability value of (1) with black color
- **Free cell** by a probability value of (0) with white color

With the assumption of that the cell is either occupied or free, each probability value contained in each cell are independent, and the surrounding environment is static. Occupancy grid maps are fine-grained grids defined over the continuous space of locations and often used after solving the SLAM problem by some other means and taking the resulting path estimates for granted. **Table 6.6.** is ROS properties of Occupancy Grid Map.

Table 6.6. Occupancy Grid Map Properties (ROS, 2021d)

ROS message	Definition
header	Timestamp and frame id
nav_msgs/MapMetaData	Contain the map's resolution (m/cell), width (cell), height (cell) and origin (0,0)
int8[] data	Contain the array of probability value of map

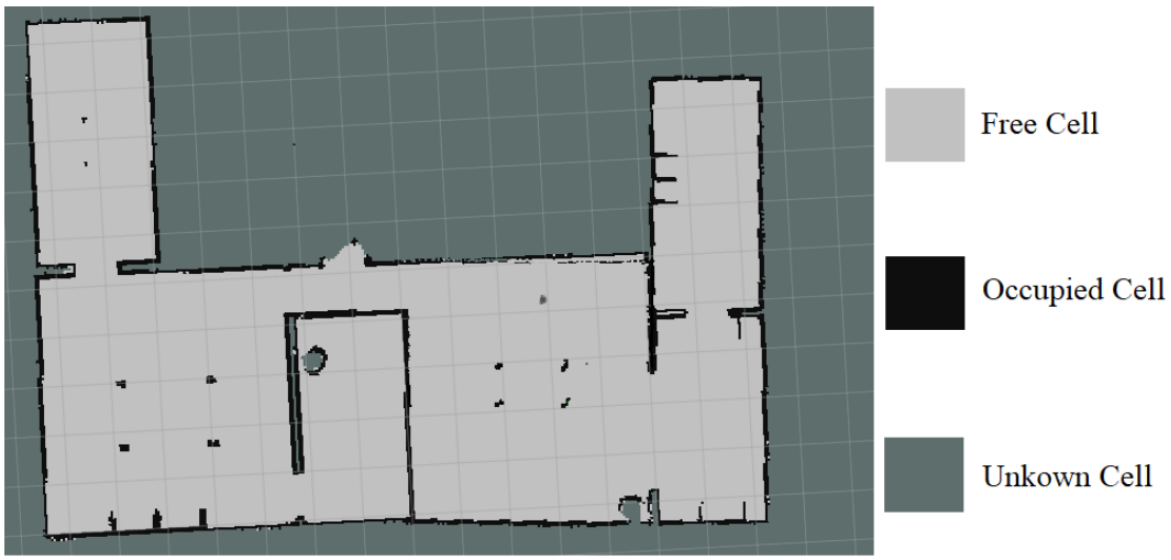

Figure 6.20. Occupancy Grid Map

Figure 6.21., SLAM start with the initialization of an empty cell grid. Each cell has an address in the grid and its size is determined by the algorithm or user assign value. Then, the empty grid is filled with probability value to represent the obstacle in the environment where the robot can not traverse on.

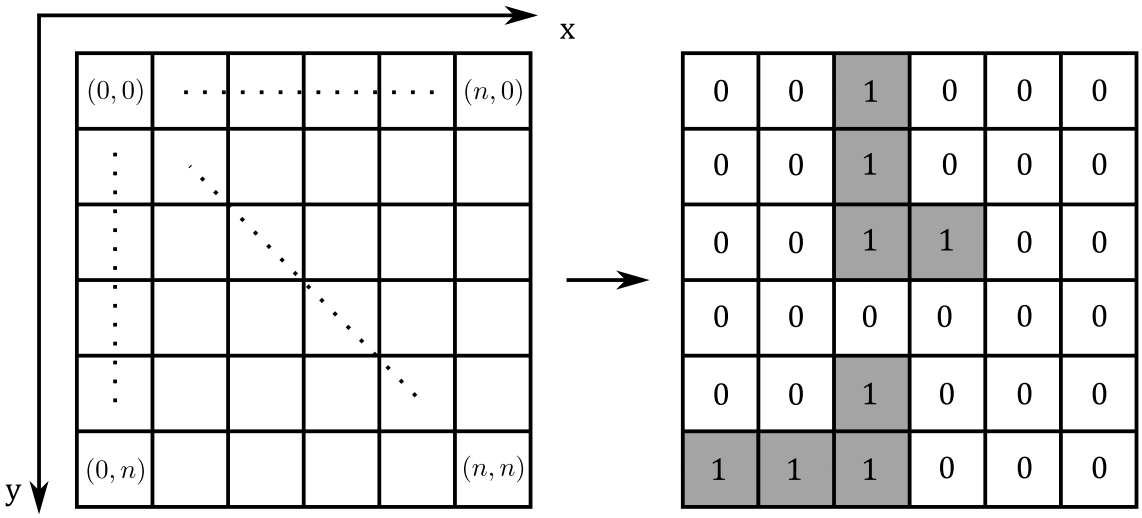

Figure 6.21. Occupancy Grid Map Cell Value

6.2.2. A* Algorithm

A* algorithm is an algorithm that based on the heuristic method. A* is the optimal best-first search algorithm. A* calculates the travel cost to the neighbor node from the current node. Node with the lowest cost to travel to is chosen.

The cost function is:

$$F(n) = G(n) + H(n) \qquad \textbf{(Eq. 6.1.)}$$

Where:

- $F(n)$ is the total cost of node path
- $G(n)$ is the exact distance from the starting node to the current node
- $H(n)$ is the estimation distance from the current node to the ending node
- n is the node

From SLAM, we obtain the occupancy grid like in figure shown below. **Figure 6.22.**, Each cell in the grid have its own address and the occupied value depending on SLAM. Let the origin address of the grid be in the top left corner, same as the occupancy grid map SLAM.

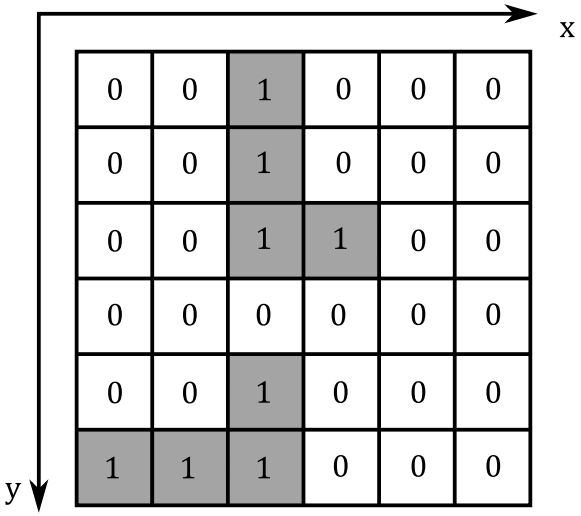

Figure 6.22. Occupancy Grid Map Example

Figure 6.23., The white cell represents the space where the robot can move and the black cell represents the space where the robot can not move. Which mean, the starting point and the ending point have to be on chosen on the white cell address.

In A* cell, the cell contain 4 information, which are:

- Address
- F value
- G value
- H value

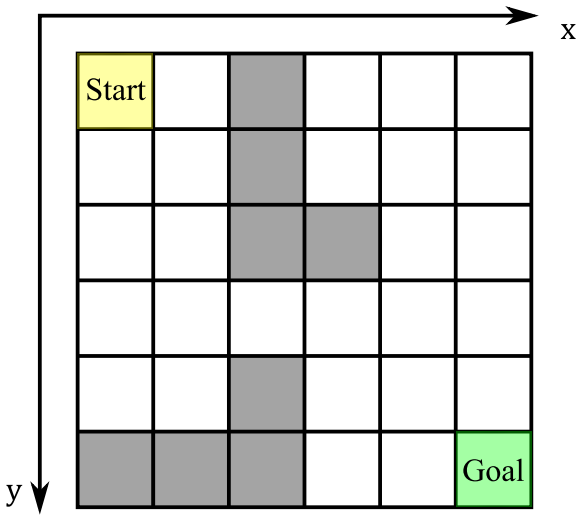

Figure 6.23. Astar starting point and ending point

• $G(n)$ **value**

In **Figure 6.24.**, to calculate the value of $G(n)$ the exact distance from the starting point to the current node, we can use the Euclidean distance formula:

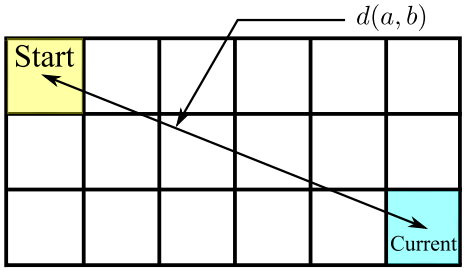

Figure 6.24. Euclidean Distance of G value

$$d(a,b)^2 = (x_b - x_a)^2 + (y_b - y_a)^2 \qquad \textbf{(Eq. 6.2.)}$$

$$d(a,b) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \qquad \textbf{(Eq. 6.3.)}$$

Where:

- $d(a,b)$ is the distance from point a to point b
- x_a, x_b are the cell address of a and b in x-axis
- y_a, y_b are the cell address of a and b in y-axis

• $H(n)$ **value**

In **Figure 6.25.**, to calculate the value of $H(n)$ the estimation distance from the current node to the ending node, we can use the Euclidean distance formula same as the **Eq. 6.3.**. The value $H(n)$ is the Heuristic part of the cost function, meaning it can be a guess value. Thus, in coding the square root can be dropped for the performance optimization.

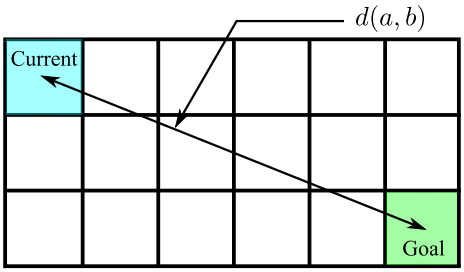

Figure 6.25. Euclidean Distance of H value

• **Parent and Child Cell**

In **Figure 6.26.**, each cell has child cell and parent cell. Parent cell is a cell where the current node state is land on. The Child cell is the 8 neighbor cells around the parent cell.

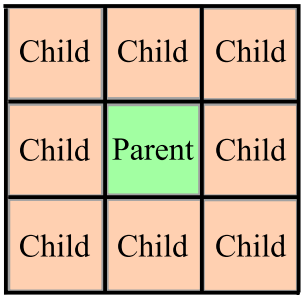

Figure 6.26. Parent and Child Cell

• **Initialize**

At start, the current node state is set to the start cell. The $H(n)$ and $G(n)$ of the cell is calculated using the **Eq. 6.3.**. The $F(n)$ is the sum of the $G(n)$ and $H(n)$ **Figure 6.27.**. Create 2 lists: Open list and Closed List.

- Open List contains the cell which the cost value is yet to be calculated or traversed pass.
- Closed List contains the cell which the cost value is calculated or traversed pass or block by the obstacle.

Starting by adding the start cell to the Open List. After calculate the cost value, move it to the Closed List.

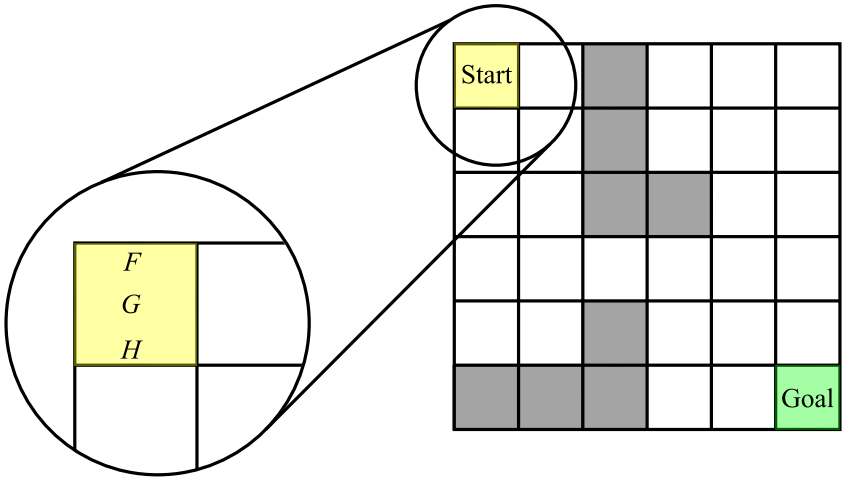

Figure 6.27. Start Point $F(n), G(n), H(n)$

• **Current state transition**

The $F(n), G(n), H(n)$ of the child cell to the parent cell is calculated (ignore the block cell) and add those to the Open List. If the $F(n)$ value of the child cell is the smallest of every cell, the current cell state is transit to that child cell, and it becomes the parent cell to the other. This action is loop until the goal point is found (the goal point is added to the Closed List). Then, backtracking the path from the goal to start as shown in **Figure 6.28.**.

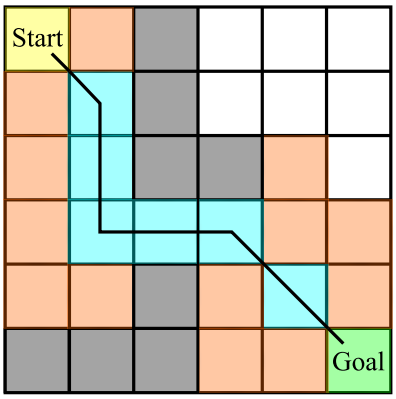

Figure 6.28. Current state transition

• **A* pseudo-code**

Table 6.7. shows the pseudo-code for the A* algorithm. **Figure 6.29.** shows the flowchart of the A* algorithm. Input data for the algorithm are: Occupancy Grid Map, Start node and Goal node. Output data for the algorithm is Pathway.

Table 6.7. A* Pseudo-code

Algorithm 1: A* Pseudo-code (Swift, 2017)

Data: Occupancy Grid Map, Start, Goal
Result: Path
INITIALIZE;
Let the *openList* equal empty list of nodes;
Let the *closedList* equal empty list of nodes;
Put the *startNode* on the *openList* (leave it's f at zero);
while *the openList is not empty* **do**
    Let the *currentNode* equal the node with the least f value;
    Remove the *currentNode* from the *openList*;
    Add the *currentNode* to the *closedList*;
    if *currentNode is the goal* **then**
        Path Found. Backtraking to the start;
    end
    Let the children of the *currentNode* equal the adjacent nodes;
    for *each child in the children* **do**
        if *child is in the closedList* **then**
            continue to beginning of for loop
        end
        child.g = *currentNode*.g + distance between child and current;
        child.h = distance from child to end;
        child.f = *child*.g + *child*.h;
        if *child.position is in the openList's nodes positions* **then**
            if *the child.g is higher than the openList node's g* **then**
                continue to beginning of for loop
            end
            Add the child to the openList
        end
    end
end

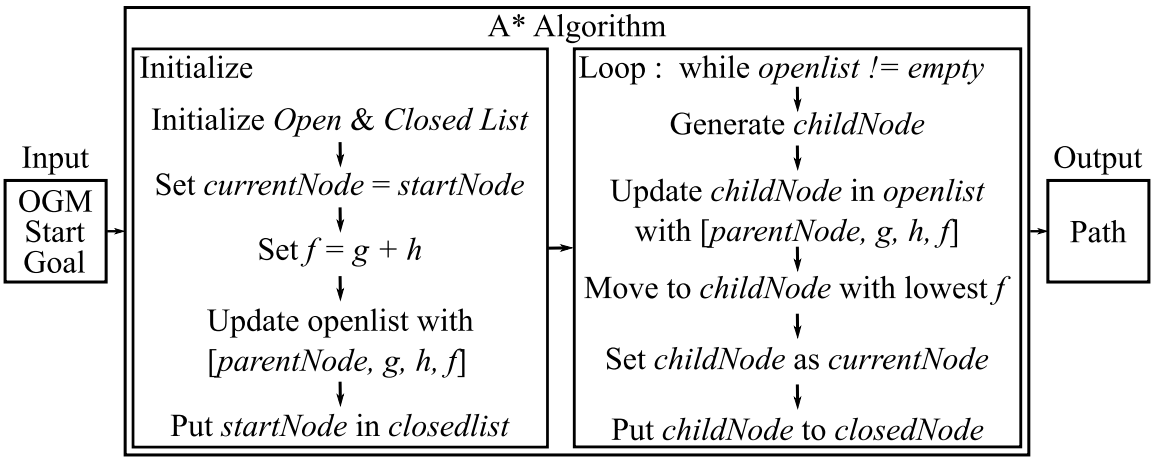

Figure 6.29. A* Flowchart

7. RESULT AND DISCUSSION

The first step of our simulation is initialization. The robot model, sensors model, and simulated room are loaded into the Gazebo simulator. For the sensor's data, it is also taking into account for the noise and the uncertainty in the simulation. For the second step, we create an occupancy grid map to represent the simulated room. To do that, we use the SLAM algorithm. After the model is loaded into the simulator, we run the SLAM algorithm on the background and drive the robot manually to explore the room until SLAM create a full map. We drive the robot using keyboard input and publish its data on ("/cmd_vel") for ROS. The result of the map obtained from SLAM is shown in **Section 7.1**. For the third step, after the map is obtained, the A* is used for finding the path from start to goal in the map. On ROS the path is published on ("/path"). The result of the path from A* is shown in **Section 7.2**. Finally, the robot controller is used to control the robot while EKF is localizing the robot pose in the map. The result of control path is shown in **Section 7.3**. **Table 7.8.** and **Table 7.9.** show the initialize parameter and the rate of data publishing.

Table 7.8. Simulation Parameters

Parameter	Value	Unit
k1	5	-
k2	10	-
k3	10	-
ka	15	-
kb	15	-
r	0.1	m
L	0.2	m
Q	$diag(1^2, 0.52^2) \times 0.01$	-
R	$diag(1^2, 1^2, 1^2) \times 0.01$	-

Table 7.9. Sensor and Algorithm Data Publication Rate for EKF

Data Publish	Rate	Unit
Encoder	20	Hz
IMU	10	Hz
Lidar Scan Match	30	Hz
EKF Fusion	20	Hz

Figure 7.30. shows the Differential Drive Wheeled Mobile Robot and Sensors (IMU, Lidar, Wheel Encoder) 3D model that is viewed inside Gazebo Simulator. **Figure 7.31.** shows the 3D models of simulated rooms.

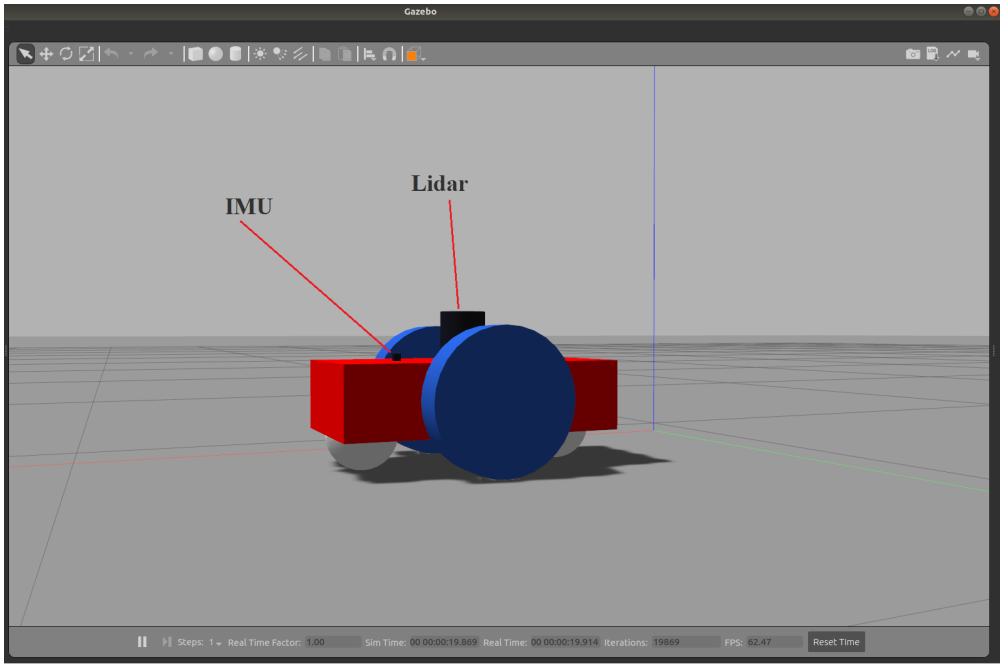

Figure 7.30. Robot Model Simulation

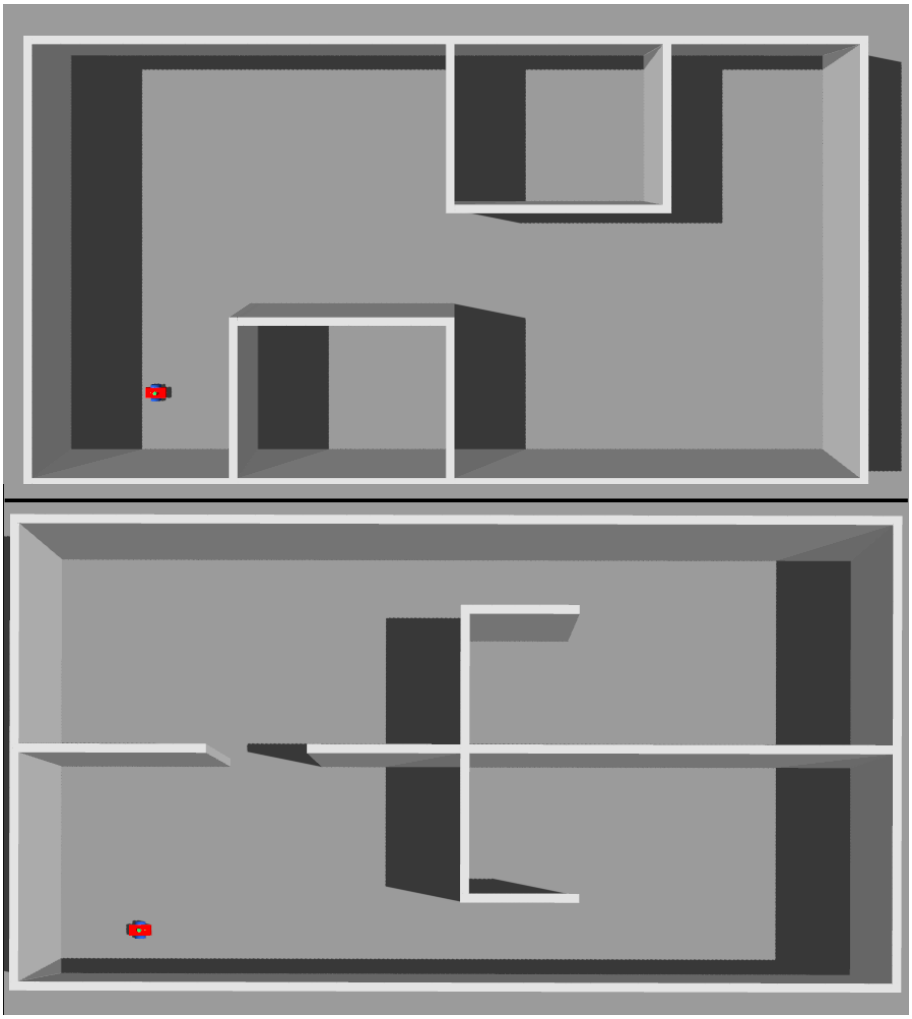

Figure 7.31. Map1 and Map2 in simulation

7.1. SLAM

In this section, we show the result from the SLAM algorithm of "Map1" and "Map2". First, the robot is placed in the room at the start location. Our goal is drive the robot manually to create the map. While the SLAM algorithm is running, we use keyboard input to move the robot along the green line from start to goal location. SLAM output the occupancy grid map and is visualized using RVIZ as shown in **Figure 7.32.** and **Figure 7.33.**. From the figures, the white space is the free space that allow the robot to move in while the black line is the obstacle.

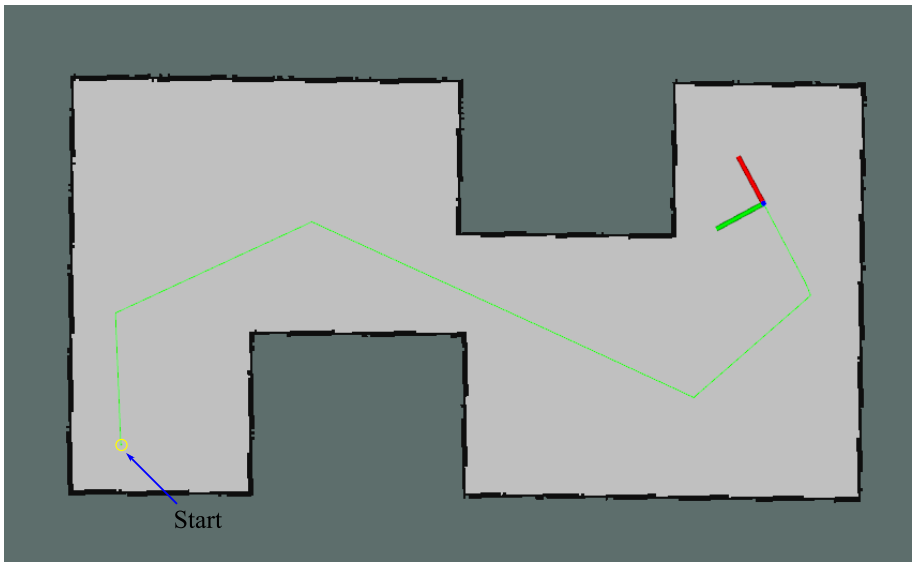

Figure 7.32. Map1 SLAM

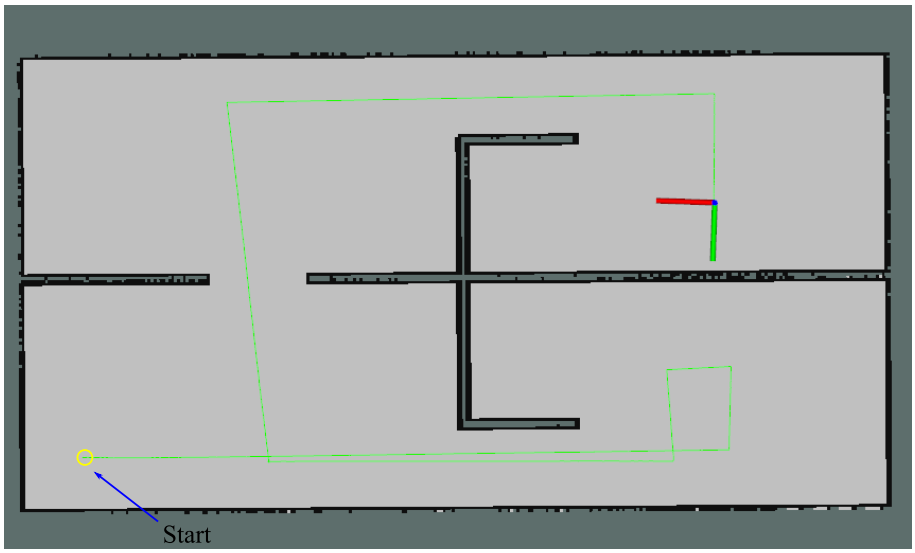

Figure 7.33. Map2 SLAM

Hector SLAM algorithm build an accurate occupancy grid map from the simulated room.

7.2. Path Planning

In this section, we show the result of the A* path planning algorithm of "Map1" and "Map2". After the occupancy grid map from the previous section is build, we put the robot at a resting state on the start location on the map. Then we chose the goal location that we want the robot to go. The A* algorithm calculates an optimal path from the map and those 2 locations and output a trajectory of the robot to follow in brown line as in **Figure 7.34.** and **Figure 7.35.**. As the algorithm is in function with Heuristic function, the H value should not be overestimated.

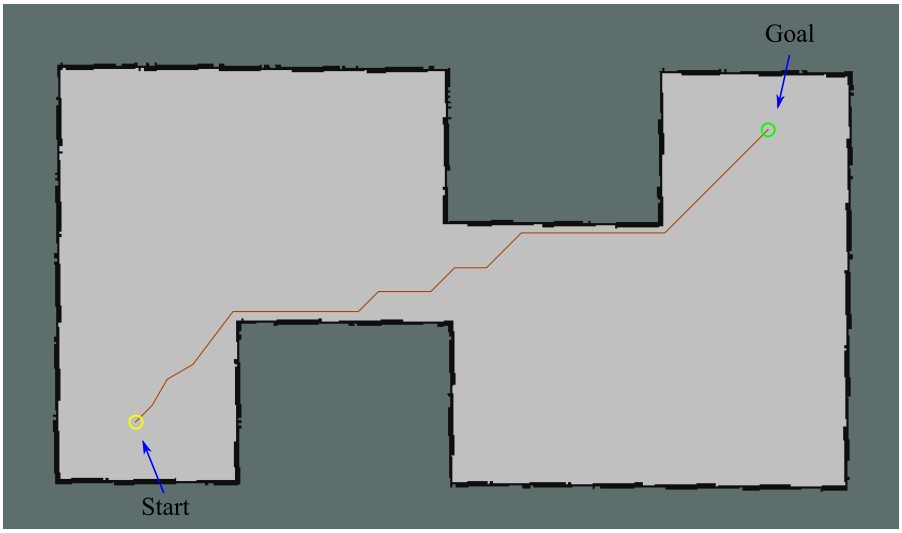

Figure 7.34. Map1 Path Planning

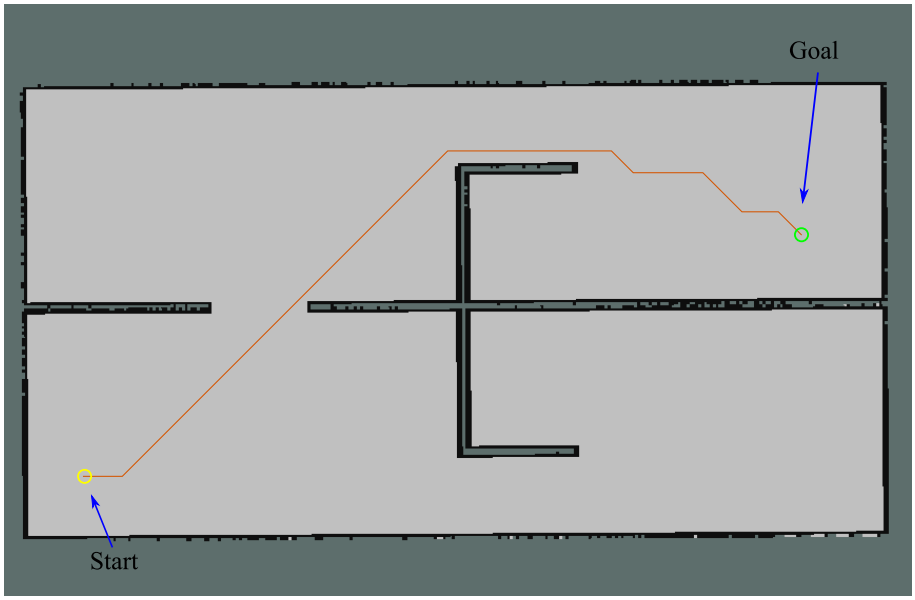

Figure 7.35. Map2 Path Planning

7.3. Control

In this section, we show the control of the robot path from start to goal. After the trajectory is generated from the A* from the last section, the controller is required to move the robot. The backstepping controller calculate the required linear velocity and angular velocity from the trajectory and the current position of the robot. To keep track of robot position, the EKF is used to estimated it location and feedback to the controller. For better performance, the constants variable in **Table 7.8.** is manually tuned. In the **Figure 7.36.** and **Figure 7.37.**, the brown line is the path that the algorithm generated. The blue line is the path that the robot currently move. As seen in the figures, the blue line follow the brown line accurately which mean the control performance is acceptable.

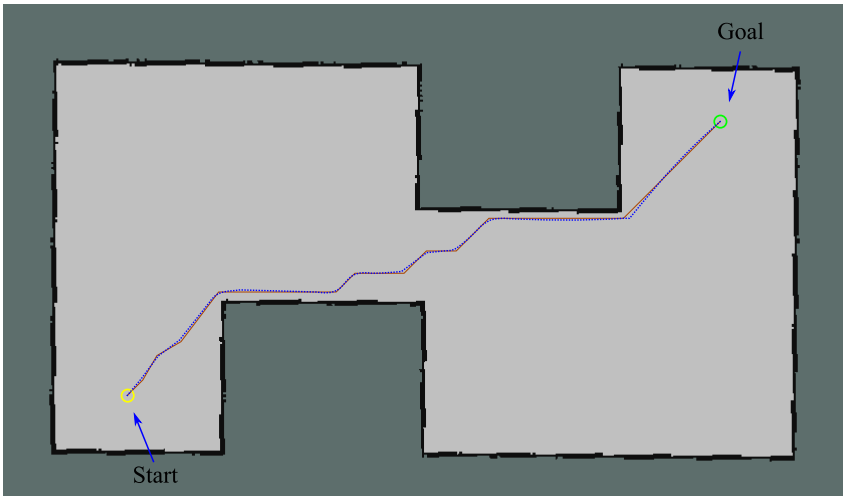

Figure 7.36. Map1 Control

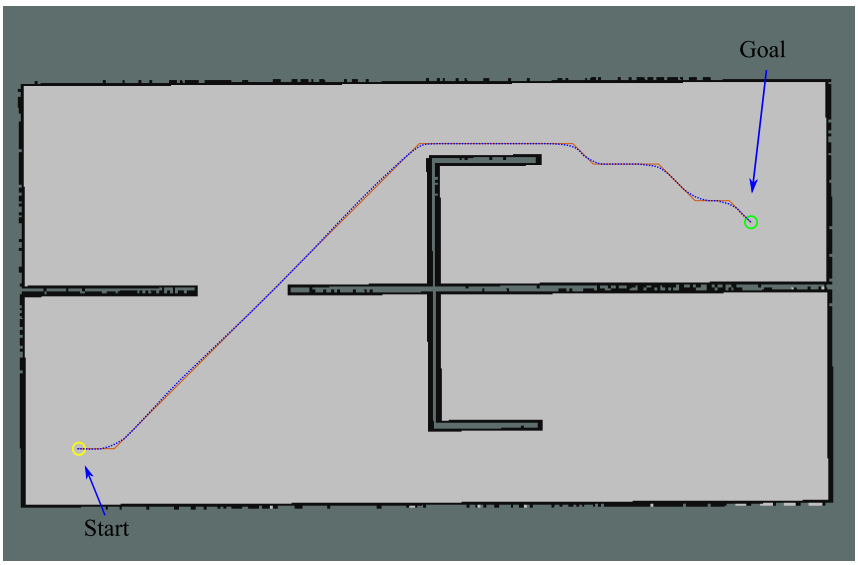

Figure 7.37. Map2 Control

8. CONCLUSION AND RECOMMENDATION

8.1. Conclusion

In conclusion, the experiment is conducted in Gazebo simulation with the robot 3D model and sensors (wheel encoders, IMU, and Lidar). The kinematic and dynamics model of differential drive mobile robot is derived. We use algorithms:

- **Extended Kalman Filter** for sensor fusion localization
- **Hector SLAM** to create the occupancy grid map
- **A* Path Planning** to determine the optimal pathway for the robot from starting point to goal point coming from user desired input
- **Backstepping** controller to control the robot motion

In static map, the A* path planning algorithm achieve a great result in generate pathway in low computational time because the cost function has the heuristic value H. The pathway is optimal as long as its value is not overestimated. The backsteppig controller results controlled output close to the generated pathway while the EKF give an accurate pose estimation.

The usage of ROS package from this project can be extended. In the robot control section, the controller can be further developed or switched between another controller. With the three sensors that is used in this project, we can add or switch with other sensors that provide a suitable data that the algorithm required. This project has produced a ROS package and framework for the differential drive mobile robot for other to test, simulate, and visualize the robot in the simulated environment. For further implementation in the real world scenario, this ROS package implement ROS topics that are ready for real devices to publish its data on.

8.2. Recommendation

In the future work, the project will be implemented in hardware. With hardware implementation, the algorithm required more accuracy and performance to cope with the hardware capability such as computational speed, data publishing rate, sensor noise and interference, unexpected failure, disturbance properties that have not been considered in the modeling.

REFERENCES

Tsardoulias, E., Petrou, L., Iliakopoulou, A., & Kargakos, A. (2013). On global path planning for occupancy grid maps.

Ducho, F., Babinec, A., Kajan, M., Beo, P., Florek, M., Fico, T., & Juriica, L. (2014). Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, *96*, 59–69.

Al Khatib, E. I., Jaradat, M. A., Abdel-Hafez, M., & Roigari, M. (2015). Multiple sensor fusion for mobile robot localization and navigation using the extended kalman filter, In *2015 10th international symposium on mechatronics and its applications (isma)*. IEEE.

Zidani, G., Drid, S., Chrifi-Alaoui, L., Benmakhlouf, A., & Chaouch, S. (2015). Backstepping controller for a wheeled mobile robot, In *2015 4th international conference on systems and control (icsc)*. IEEE.

Moore, T., & Stouch, D. (2016). A generalized extended kalman filter implementation for the robot operating system. In *Intelligent autonomous systems 13* (pp. 335–348). Springer.

Klanar, G., Zdear, A., Blai, S., & krjanc, I. (2017). Chapter 4 - path planning. In G. Klanar, A. Zdear, S. Blai, & I. krjanc (Eds.), *Wheeled mobile robotics* (pp. 161–206). Butterworth-Heinemann. https://doi.org/https://doi.org/10.1016/B978-0-12-804204-5.00004-4

Swift, N. (2017). *Easy a* (star) pathfinding*. https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 (accessed: 06.22.2021)

Kim, Y., & Bang, H. (2018). Introduction to kalman filter and its applications. *Introduction and Implementations of the Kalman Filter*, *1*, 1–16.

ROS. (2021a). *About ros*. https://www.ros.org/about-ros/ (accessed: 06.22.2021)

ROS. (2021b). *Ros imu*. http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/Imu.html (accessed: 06.22.2021)

ROS. (2021c). *Ros laserscan*. http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html (accessed: 06.22.2021)

ROS. (2021d). *Ros occupancy grid map*. http://docs.ros.org/en/melodic/api/nav_msgs/html/msg/OccupancyGrid.html (accessed: 06.22.2021)

ROS. (2021e). *Ros odometry*. http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html (accessed: 06.22.2021)

ROS. (2021f). *Ros twist*. http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html (accessed: 06.22.2021)

Appendix A Differential Drive Mobile Robot Gazebo Modeling

model.config

```
<?xml version="1.0"?>
<model>
  <name>robotros_test</name>
  <version>1.0</version>
  <sdf version='1.4'>model.sdf</sdf>

  <author>
    <name>Phayuth</name>
    <email>me@my.email</email>
  </author>

  <description>
    My robot for LILNAVI PROJECT
  </description>
</model>
```

model.sdf

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

<sdf version='1.4'>

  <model name="robotros_test">

    <static>false</static>

    <link name='chassis'>
      <pose>0 0 0.1 0 0 0</pose>

      <inertial>
        <mass>10.0</mass>
        <pose>0.0 0 0.1 0 0 0</pose>
        <inertia>
          <ixx>0.5</ixx>
          <iyy>1.0</iyy>
          <izz>0.1</izz>
        </inertia>
```

```
      </inertial>
      <collision name='collision'>
        <geometry>
          <box>
            <size>.4 .2 .1</size>
          </box>
        </geometry>
      </collision>

      <visual name='visual'>
        <pose>0 0 0.0 0 0 0</pose>
        <geometry>
          <box>
            <size>.4 .2 .1</size>
          </box>
        </geometry>
        <material> <!-- material -->
          <ambient>255 0 0 1</ambient>
          <diffuse>255 0 0 1</diffuse>
          <specular>0 0 0 0</specular>
          <emissive>0 0 0 1</emissive>
        </material> <!--material -->
      </visual>

      <collision name='caster1_collision'>
        <pose>-0.15 0 -0.05 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>.05</radius>
          </sphere>
        </geometry>

        <surface>
          <friction>
            <ode>
              <mu>0</mu>
              <mu2>0</mu2>
              <slip1>1.0</slip1>
              <slip2>1.0</slip2>
```

```
            </ode>
          </friction>
        </surface>
      </collision>

      <visual name='caster1_visual'>
        <pose>-0.15 0 -0.05 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>.05</radius>
          </sphere>
        </geometry>
      </visual>

      <collision name='caster2_collision'>
        <pose>0.15 0 -0.05 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>.05</radius>
          </sphere>
        </geometry>

        <surface>
          <friction>
            <ode>
              <mu>0</mu>
              <mu2>0</mu2>
              <slip1>1.0</slip1>
              <slip2>1.0</slip2>
            </ode>
          </friction>
        </surface>
      </collision>

      <visual name='caster2_visual'>
        <pose>0.15 0 -0.05 0 0 0</pose>
        <geometry>
          <sphere>
            <radius>.05</radius>
```

```
98          </sphere>
99        </geometry>
100     </visual>
101   </link>
102
103   <link name="left_wheel">
104     <pose>0 0.13 0.1 0 1.5707 1.5707</pose>
105     <collision name="collision">
106       <geometry>
107         <cylinder>
108           <radius>.1</radius>
109           <length>.05</length>
110         </cylinder>
111       </geometry>
112     </collision>
113     <visual name="visual">
114       <geometry>
115         <cylinder>
116           <radius>.1</radius>
117           <length>.05</length>
118         </cylinder>
119       </geometry>
120       <material> <!-- material -->
121         <ambient>37 90 201 1</ambient>
122         <diffuse>37 90 201 1</diffuse>
123         <specular>0 0 0 0</specular>
124         <emissive>0 0 0 1</emissive>
125       </material> <!--material -->
126     </visual>
127   </link>
128
129   <link name="right_wheel">
130     <pose>0 -0.13 0.1 0 1.5707 1.5707</pose>
131     <collision name="collision">
132       <geometry>
133         <cylinder>
134           <radius>.1</radius>
135           <length>.05</length>
136         </cylinder>
```

```
      </geometry>
    </collision>
    <visual name="visual">
      <geometry>
        <cylinder>
          <radius>.1</radius>
          <length>.05</length>
        </cylinder>
      </geometry>
      <material> <!-- material -->
        <ambient>37 90 201 1</ambient>
        <diffuse>37 90 201 1</diffuse>
        <specular>0 0 0 0</specular>
        <emissive>0 0 0 1</emissive>
      </material> <!--material -->
    </visual>
  </link>

  <link name="lidar_link">
    <pose>0 0 0.185 0 0 0</pose>

    <!--inertial>
      <mass>0.6</mass>
      <inertia>
        <ixx>0.00001974</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>0.00001974</iyy>
        <iyz>0</iyz>
        <izz>0.00003505</izz>
      </inertia>
    </inertial-->

    <collision name="collision">
      <geometry>
        <cylinder>
          <radius>0.0325</radius>
          <length>0.070</length>
        </cylinder>
```

```
        </geometry>
      </collision>

      <visual name="visual">
        <geometry>
          <cylinder>
            <radius>0.0325</radius>
            <length>0.070</length>
          </cylinder>
        </geometry>
        <material> <!-- material -->
          <ambient>35 230 106 1</ambient>
          <diffuse>35 230 106 1</diffuse>
          <specular>0 0 0 0</specular>
          <emissive>0 0 0 1</emissive>
        </material> <!--material -->
      </visual>

      <sensor type="gpu_ray" name="head_hokuyo_sensor">
        <pose>0 0 0.020 0 0 0</pose>
        <visualize>false</visualize>
        <update_rate>40</update_rate>
        <ray>
          <scan>
            <horizontal>
              <samples>720</samples>
              <resolution>1</resolution>
              <min_angle>-3.14159</min_angle> <!--min_angle>-1.570796</
                  min_angle-->
              <max_angle>3.14159</max_angle>  <!--max_angle>1.570796</
                  max_angle-->
            </horizontal>
          </scan>
          <range>
            <min>0.10</min>
            <max>12.0</max>
            <resolution>0.01</resolution>
          </range>
          <noise>
```

```
            <type>gaussian</type>
            <mean>0.0</mean>
            <stddev>0.01</stddev>
          </noise>
        </ray>
        <plugin name="gazebo_ros_head_hokuyo_controller" filename="
           libgazebo_ros_gpu_laser.so">
          <topicName>/laser/scan</topicName>
          <frameName>lidar_link</frameName>
        </plugin>
      </sensor>
    </link>

    <link name="imu_link">
      <pose>0.100 0 0.155 0 0 0</pose> <!--Gap of 2mm-->

      <!--inertial>
        <mass>0.05</mass>
        <inertia>
          <ixx>0.00000002</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>0.00000002</iyy>
          <iyz>0</iyz>
          <izz>0.00000002</izz>
        </inertia>
      </inertial-->

      <collision name="collision">
        <geometry>
          <box>
            <size>0.01 0.01 0.01</size>
          </box>
        </geometry>
      </collision>

      <visual name="visual">
        <geometry>
          <box>
```

```
            <size>0.01 0.01 0.01</size>
          </box>
        </geometry>
        <material> <!-- material -->
          <ambient>35 230 106 1</ambient>
          <diffuse>35 230 106 1</diffuse>
          <specular>0 0 0 0</specular>
          <emissive>0 0 0 1</emissive>
        </material> <!--material -->
      </visual>

      <gravity>true</gravity>
      <sensor name="imu_sensor" type="imu">
        <always_on>true</always_on>
        <update_rate>100</update_rate>
        <visualize>true</visualize>
        <topic>__default_topic__</topic>
        <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin
           ">
          <topicName>imu</topicName>
          <bodyName>imu_link</bodyName>
          <updateRateHZ>10</updateRateHZ><!--10-->
          <gaussianNoise>0.01</gaussianNoise><!--0.1-->
          <xyzOffset>0 0 0</xyzOffset>
          <rpyOffset>0 0 0</rpyOffset>
          <frameName>imu_link</frameName>
          <initialOrientationAsReference>false</
             initialOrientationAsReference>
        </plugin>
        <pose>0 0 0 0 0 0</pose>
      </sensor>
    </link>

    <joint type="revolute" name="left_wheel_hinge">
      <pose>0 0 -0.03 0 0 0</pose>
      <child>left_wheel</child>
      <parent>chassis</parent>
      <axis>
        <xyz>0 1 0</xyz>
```

```
288       </axis>
289     </joint>
290
291     <joint type="revolute" name="right_wheel_hinge">
292       <pose>0 0 0.03 0 0 0</pose>
293       <child>right_wheel</child>
294       <parent>chassis</parent>
295       <axis>
296         <xyz>0 1 0</xyz>
297       </axis>
298     </joint>
299
300     <joint type="fixed" name="lidar_joint">
301       <pose>0 0 -0.025 0 0 0</pose>
302       <child>lidar_link</child>
303       <parent>chassis</parent>
304       <axis>
305         <xyz>0 0 1</xyz> <!-- what axis that it going to rotate about--
              >
306       </axis>
307     </joint>
308
309     <joint type="fixed" name="imu_joint">
310       <pose>0 0 -0.005 0 0 0</pose>
311       <child>imu_link</child>
312       <parent>chassis</parent>
313       <axis>
314         <xyz>0 0 1</xyz> <!-- what axis that it going to rotate about--
              >
315       </axis>
316     </joint>
317
318     <plugin name="differential_drive_controller" filename="
        libgazebo_ros_diff_drive.so">
319       <alwaysOn>true</alwaysOn>
320
321       <!-- Plugin update rate in Hz -->
322       <updateRate>20</updateRate>
323
```

```
324         <!-- Name of left joint, defaults to `left_joint` -->
325         <leftJoint>left_wheel_hinge</leftJoint>
326 
327         <!-- Name of right joint, defaults to `right_joint` -->
328         <rightJoint>right_wheel_hinge</rightJoint>
329 
330         <!-- The distance from the center of one wheel to the other, in
                meters, defaults to 0.34 m -->
331         <wheelSeparation>0.26</wheelSeparation>
332 
333         <!-- Diameter of the wheels, in meters, defaults to 0.15 m -->
334         <wheelDiameter>0.2</wheelDiameter>
335 
336         <!-- Wheel acceleration, in rad/s^2, defaults to 0.0 rad/s^2 -->
337         <wheelAcceleration>0</wheelAcceleration>
338 
339         <!-- Maximum torque which the wheels can produce, in Nm, defaults
                to 5 Nm -->
340         <wheelTorque>5</wheelTorque>
341 
342         <!-- Topic to receive geometry_msgs/Twist message commands,
                defaults to `cmd_vel` -->
343         <commandTopic>cmd_vel</commandTopic>
344 
345         <!-- Topic to publish nav_msgs/Odometry messages, defaults to `
                odom` -->
346         <odometryTopic>odom</odometryTopic>
347 
348         <!-- Odometry frame, defaults to `odom` -->
349         <odometryFrame>odom</odometryFrame>
350 
351         <!-- Robot frame to calculate odometry from, defaults to `
                base_footprint` -->
352         <robotBaseFrame>chassis</robotBaseFrame>
353 
354         <!-- Odometry source, encoder, world, defaults to world -->
355         <odometrySource>encoder</odometrySource>
356 
357         <!-- Set to true to publish transforms for the wheel links,
```

```
        defaults to false -->
      <publishWheelTF>false</publishWheelTF><!--true-->

      <!-- Set to true to publish transforms for the odometry, defaults
          to true -->
      <publishOdometry>false</publishOdometry><!--true-->

      <publishOdomTF>false</publishOdomTF> <!--true-->
      <!-- Set to true to publish sensor_msgs/JointState on /
         joint_states for the wheel joints, defaults to false -->
      <publishWheelJointState>false</publishWheelJointState>

      <!-- Set to true to swap right and left wheels, defaults to true
         -->
      <legacyMode>false</legacyMode>

      <publishTf>1</publishTf> <!--1-->
    </plugin>

    <plugin name="p3d_base_controller" filename="libgazebo_ros_p3d.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <bodyName>chassis</bodyName>
      <topicName>ground_truth/state</topicName>
      <gaussianNoise>0</gaussianNoise><!--0.1-->
      <frameName>map</frameName>
      <xyzOffsets>0 0 0</xyzOffsets>
      <rpyOffsets>0 0 0</rpyOffsets>
    </plugin>
  </model>
</sdf>
```

Appendix B SLAM

mapping_default.launch

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

<launch>
<arg name="tf_map_scanmatch_transform_frame_name" default="
    scanmatcher_frame"/>
<!-- <arg name="base_frame" default="base_footprint"/> -->
<arg name="base_frame" default="base_frame"/>
<!-- <arg name="odom_frame" default="nav"/> -->
<arg name="odom_frame" default="base_frame"/>
<arg name="pub_map_odom_transform" default="true"/>
<arg name="scan_subscriber_queue_size" default="5"/>
<arg name="scan_topic" default="scan"/>
<arg name="map_size" default="2048"/>

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

<!-- Frame names -->
<param name="map_frame" value="map" />
<param name="base_frame" value="$(arg␣base_frame)" />
<param name="odom_frame" value="$(arg␣odom_frame)" />

<!-- Tf use -->
<param name="use_tf_scan_transformation" value="true"/>
<param name="use_tf_pose_start_estimate" value="false"/>
<param name="pub_map_odom_transform" value="$(arg␣
    pub_map_odom_transform)"/>

<!-- Map size / start point -->
<param name="map_resolution" value="0.050"/>
<param name="map_size" value="$(arg␣map_size)"/>
<param name="map_start_x" value="0.5"/>
<param name="map_start_y" value="0.5" />
<param name="map_multi_res_levels" value="2" />

<!-- Map update parameters -->
```

```
<param name="update_factor_free" value="0.4"/>
<param name="update_factor_occupied" value="0.9" />
<param name="map_update_distance_thresh" value="0.4"/>
<param name="map_update_angle_thresh" value="0.06" />
<param name="laser_z_min_value" value = "-1.0" />
<param name="laser_z_max_value" value = "1.0" />

<!-- Advertising config -->
<param name="advertise_map_service" value="true"/>

<param name="scan_subscriber_queue_size" value="$(arg␣
   scan_subscriber_queue_size)"/>
<param name="scan_topic" value="$(arg␣scan_topic)"/>

<!-- Debug parameters -->

<param name="output_timing" value="false"/>
<param name="pub_drawings" value="true"/>
<param name="pub_debug_output" value="true"/>
-->
<param name="tf_map_scanmatch_transform_frame_name" value="$(arg␣
   tf_map_scanmatch_transform_frame_name)" />
</node>

<!--<node pkg="tf" type="static_transform_publisher" name="
   map_nav_broadcaster" args="0␣0␣0␣0␣0␣0␣map␣nav␣100"/>-->
<node pkg="tf" type="static_transform_publisher" name="
   base_to_laser_broadcaster" args="0␣0␣0␣0␣0␣0␣base_frame␣
   laser_frame␣100"/>
</launch>
```

tutorial.launch

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

<launch>

<arg name="geotiff_map_file_path" default="$(find␣hector_geotiff)/
   maps"/>

```

```
7   <param name="/use_sim_time" value="false"/>
8
9   <node pkg="rviz" type="rviz" name="rviz"
10  args="-d␣$(find␣hector_slam_launch)/rviz_cfg/mapping_demo.rviz"/>
11
12  <include file="$(find␣hector_mapping)/launch/mapping_default.launch"/
        >
13
14  <include file="$(find␣hector_geotiff)/launch/geotiff_mapper.launch">
15  <arg name="trajectory_source_frame_name" value="scanmatcher_frame"/>
16  <arg name="map_file_path" value="$(arg␣geotiff_map_file_path)"/>
17  </include>
18
19  </launch>
```

Appendix C A* Path Planning

```
import numpy as np
import cv2
class Node:
"""
A node class for A* Pathfinding
parent is parent of the current Node
position is current position of the Node in the maze
g is cost from start to current Node
h is heuristic based estimated cost for current Node to end Node
f is total cost of present node i.e. : f = g + h
"""

  def __init__(self, parent=None, position=None):
    self.parent = parent
    self.position = position

    self.g = 0
    self.h = 0
    self.f = 0
    def __eq__(self, other):
    return self.position == other.position

  #This function return the path of the search
  def return_path(current_node,maze):
    path = []
    no_rows, no_columns = np.shape(maze)
    # here we create the initialized result maze with -1 in every
       position
    result = [[-1 for i in range(no_columns)] for j in range(no_rows)]
    current = current_node
    while current is not None:
      path.append(current.position)
      current = current.parent
    # Return reversed path as we need to show from start to end path
    path = path[::-1]
    start_value = 0
    # we update the path of start to end found by A-star serch with
```

```
        every step incremented by 1
    for i in range(len(path)):
      result[path[i][0]][path[i][1]] = start_value
      start_value += 1
    return result


def search(maze, cost, start, end):
"""
Returns a list of tuples as a path from the given start to the given
    end in the given maze
:param maze:
:param cost
:param start:
:param end:
:return:
"""

  # Create start and end node with initized values for g, h and f
  start_node = Node(None, tuple(start))
  start_node.g = start_node.h = start_node.f = 0
  end_node = Node(None, tuple(end))
  end_node.g = end_node.h = end_node.f = 0

  # Initialize both yet_to_visit and visited list
  # in this list we will put all node that are yet_to_visit for
      exploration.
  # From here we will find the lowest cost node to expand next
  yet_to_visit_list = []
  # in this list we will put all node those already explored so that we
      don't explore it again
  visited_list = []

  # Add the start node
  yet_to_visit_list.append(start_node)

  # Adding a stop condition. This is to avoid any infinite loop and
      stop
  # execution after some reasonable number of steps
```

```
  outer_iterations = 0
  max_iterations = (len(maze) // 2) ** 10

  # what squares do we search . serarch movement is left-right-top-
     bottom
  #(4 movements) from every positon

  move = [[-1, 0 ], # go up
  [ 0, -1], # go left
  [ 1, 0 ], # go down
  [ 0, 1 ]] # go right


"""
1) We first get the current node by comparing all f cost and selecting
   the lowest cost node for further expansion
2) Check max iteration reached or not . Set a message and stop
   execution
3) Remove the selected node from yet_to_visit list and add this node to
    visited list
4) Perofmr Goal test and return the path else perform below steps
5) For selected node find out all children (use move to find children)
a) get the current postion for the selected node (this becomes parent
   node for the children)
b) check if a valid position exist (boundary will make few nodes
   invalid)
c) if any node is a wall then ignore that
d) add to valid children node list for the selected parent

For all the children node
a) if child in visited list then ignore it and try next node
b) calculate child node g, h and f values
c) if child in yet_to_visit list then ignore it
d) else move the child to yet_to_visit list
"""
#find maze has got how many rows and columns
  no_rows, no_columns = np.shape(maze)

# Loop until you find the end
```

```
104
105 while len(yet_to_visit_list) > 0:
106
107 # Every time any node is referred from yet_to_visit list, counter of
    limit operation incremented
108   outer_iterations += 1
109
110
111 # Get the current node
112   current_node = yet_to_visit_list[0]
113   current_index = 0
114   for index, item in enumerate(yet_to_visit_list):
115     if item.f < current_node.f:
116       current_node = item
117       current_index = index
118
119 # if we hit this point return the path such as it may be no solution
    or
120 # computation cost is too high
121     if outer_iterations > max_iterations:
122       print ("giving up on pathfinding too many iterations")
123       return return_path(current_node,maze)
124
125 # Pop current node out off yet_to_visit list, add to visited list
126 yet_to_visit_list.pop(current_index)
127 visited_list.append(current_node)
128
129 # test if goal is reached or not, if yes then return the path
130 if current_node == end_node:
131   return return_path(current_node,maze)
132
133 # Generate children from all adjacent squares
134 children = []
135
136 for new_position in move:
137
138 # Get node position
139   node_position = (current_node.position[0] + new_position[0],
      current_node.position[1] + new_position[1])
```

```
140
141   # Make sure within range (check if within maze boundary)
142     if (node_position[0] > (no_rows - 1) or
143       node_position[0] < 0 or
144       node_position[1] > (no_columns -1) or
145       node_position[1] < 0):
146       continue
147
148   # Make sure walkable terrain
149     if maze[node_position[0]][node_position[1]] != 0:
150       continue
151
152   # Create new node
153     new_node = Node(current_node, node_position)
154
155   # Append
156     children.append(new_node)
157
158 # Loop through children
159   for child in children:
160
161 # Child is on the visited list (search entire visited list)
162     if len([visited_child for visited_child in visited_list if
          visited_child == child]) > 0:
163       continue
164
165     # Create the f, g, and h values
166     child.g = current_node.g + cost
167     ## Heuristic costs calculated here, this is using eucledian
          distance
168     child.h = (((child.position[0] - end_node.position[0]) ** 2) +
169     ((child.position[1] - end_node.position[1]) ** 2))
170
171     child.f = child.g + child.h
172
173     # Child is already in the yet_to_visit list and g cost is already
          lower
174     if len([i for i in yet_to_visit_list if child == i and child.g > i.
          g]) > 0:
```

```
        continue

    # Add the child to the yet_to_visit list
    yet_to_visit_list.append(child)


if __name__ == '__main__':
imgdsp = cv2.imread("map.png")
img = cv2.imread("map.png")
print("read")
img=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY) # 0 is black , 255 is white
height, width = img.shape

ls = []
maze=[]

for i in range(height):
  for j in range(width):
    if img[i,j] == 255:
      img[i,j] = 0
    elif img[i,j] == 0:
      img[i,j] = 1
    ls.append(img[i,j])
    print("added........")

f = 0
for k in range(height):
  maze.append(ls[f:f+(height-1)])
  f+=height


start = [_,_] # starting position
end = [_,_] # ending position
cost = 1 # cost per movement

print("seaching....")
path = search(maze,cost, start, end)

print('\n'.join([''.join(["{:" ">3d}".format(item) for item in row])
```

```
    for row in path]))
214
215 for a in range(len(path[0])):
216   for b in range(len(path[0])):
217     if path[a][b]>-1:
218       red = [0,0,255]
219       imgdsp[a][b]=red
220
221 fn = "new.png"
222 cv2.imwrite(fn,imgdsp)
```

Appendix D Control and EKF

- **Control**

```
#! /usr/bin/env python

import numpy as np
import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Pose,Quaternion, Twist, Vector3
from tf.transformations import euler_from_quaternion,
    quaternion_from_euler

def angnorm(theta):
  if theta>0:
    if theta>np.pi:
      theta = (2*np.pi-theta)*(-1)

  if theta<0:
    if theta*(-1)>np.pi:
      theta = (2*np.pi-(-theta))*(-1)
  return theta

class backstp_contrl:

def __init__(self,k1,k2,k3,ka,kb,m,r,b,Iner):
  self.k1  = k1
  self.k2  = k2
  self.k3  = k3
  self.ka  = ka
  self.kb  = kb
  self.m   = m
  self.r   = r
  self.b   = b
  self.Iner = Iner

def error(self,qr,qc):
  """Calculate error from the current pose to the reference pose.
     return qe"""
  theta = qc[2,0]
```

```
35
36    T = np.array([[np.cos(theta),np.sin(theta),0],
37                  [-np.sin(theta),np.cos(theta),0],
38                  [   0     ,    0     ,1]]) # (3X3)
39
40    e = qr - qc
41    return T.dot(e) # (3X1)
42
43  def controlkinematic(self,qe,vr,wr):
44    """ Control Algorithm for Kinematic, return : vc, wc"""
45    return ((vr*np.cos(qe[2,0]))+self.k1*qe[0,0]),(wr+(self.k2*vr*qe
          [1,0])*(self.k3*np.sin(qe[2,0])))
46
47  def controldynamics(self,vdotref,wdotref):
48    """ Control Algorithm for Dynamics, z1= vref-vcur , z2 = wref-wcur,
          return : tua1c, tua2c """
49    return 1/2*((self.m*self.r*(vdotref+self.ka*z1))+((2*self.r*self.Iner
          /b)*(wdotref+self.kb*z2))),1/2*((self.m*self.r*(vdotref+self.ka*z1
          ))-((2*self.r*self.Iner/b)*(wdotref+self.kb*z2)))
50
51  sq = 0
52  def odm_callback(msg):
53    global sq
54    # Find Current Pose
55    x = msg.pose.pose.position.x
56    y = msg.pose.pose.position.y
57    qq = msg.pose.pose.orientation
58    ol = [qq.x,qq.y,qq.z,qq.w]
59    (rl,pt,yw)=euler_from_quaternion(ol)
60    qc = np.array([[x],[y],[yw]])
61
62    # Find Desired Pose
63    xRef,yRef,theta_ref,vr,wr,ydot,xdot,vdotref,wdotref = ref_8(sq)
64    qr = np.array([[xRef],[yRef],[theta_ref]])
65
66    # Control init for each trajectory
67    #contl = backstp_contrl(0.7,20,20,100,3000,4,0.1,0.26,2.5) # For
          ref_circle
68    contl = backstp_contrl(0.7,20,20,100,3000,4,0.1,0.26,2.5) # For ref_8
```

```
69
70    # Find error and control
71    qe = contl.error(qr,qc)
72    vc,wc = contl.controlkinematic(qe,vr,wr)
73
74    # Print some output
75    rospy.loginfo("thetaref␣=␣"+str(theta_ref)+"    ␣thetac␣=␣"+str(qc
          [2,0]))
76
77    # Publish to ROS
78    tw_p = rospy.Publisher("/robotros_test/cmd_vel", Twist, queue_size =
          50)
79    Twm = Twist()
80    Twm.linear.x = vc
81    Twm.angular.z = wc
82    tw_p.publish(Twm)
83    sq += 1
84
85  def main():
86
87    rospy.init_node('BS_Ctrl')
88    rospy.Subscriber('/ground_truth/state',Odometry, odm_callback)
89    rospy.spin()
90
91  if __name__=='__main__':
92    main()
```

- **EKF**

```
1   #! /usr/bin/env python
2
3   import numpy as np
4   import time
5
6   import rospy
7   from nav_msgs.msg import Odometry
8   from sensor_msgs.msg import Imu
9   import tf
10  from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3,
        Pose2D
```

```
11
12
13 class EKF_node(object):
14 def __init__(self):
15
16   self.xest = np.array([[0],[0],[0]])
17   self.pest = np.eye(3)
18   self.z = np.array([[0],[0],[0]])
19
20   self.loopr = rospy.Rate(20)
21
22   rospy.Subscriber('/pose2D',Pose2D, self.read_lidr)
23
24   self.pub = rospy.Publisher("/odom_filter", Odometry, queue_size = 50)
25
26 def read_lidr(self,msg):
27   yy = msg.theta
28   if yy < 0:
29   yy = np.pi + (np.pi+yy)
30   self.z = np.array([[msg.x],[msg.y],[yy]])
31   Ts = 0.05
32
33   # input calculate
34   noise = np.diag([1.,np.deg2rad(30.)])**2
35   v = 1
36   w = 0.5
37   u = np.array([v,w]) #+ noise.dot(np.random.rand(2,1))
38
39   # jacob F
40   yaw = self.xest[2,0]
41   jF = np.array([[1.,0,-v*np.sin(yaw)*Ts],[0,1.,v*np.cos(yaw)*Ts
        ],[0,0,1.]])
42
43   # prediction
44   Q = np.diag([0.1,0.1,np.deg2rad(1.)])**2
45   F = np.array([[1.,0,0],[0,1.,0],[0,0,1.]])
46   B = np.array([[Ts*np.cos(self.xest[2,0]),0],[Ts*np.sin(self.xest
        [2,0]),0],[0,Ts]])
47   xpred = F.dot(self.xest)+B.dot(u)
```

```
48    ppred = jF.dot(self.pest.dot(jF.T)) + Q
49
50    # update
51    R = np.diag([0.1,0.1,0.1])**2
52    y = self.z - xpred
53    jH = np.diag([1.,1.,1.])
54    s = (R + jH.dot(ppred.dot(jH.T)))
55    K = ppred.dot((jH.T).dot(np.linalg.inv(s)))
56    self.xest = xpred + K.dot(y)
57    self.pest = (np.eye(3) - K.dot(jH)).dot(ppred)
58
59  if __name__ == '__main__':
60
61    rospy.init_node('ekf_locl_fusion')
62    nod = EKF_node()
63
64    while not rospy.is_shutdown():
65    odom_broadcaster = tf.TransformBroadcaster()
66    odom = Odometry()
67    odom_quat = tf.transformations.quaternion_from_euler(0,0,nod.xest
        [2,0])
68    odom_broadcaster.sendTransform((nod.xest[0,0],nod.xest[1,0],0.),
        odom_quat,rospy.Time.now(),"base_link_filter","odom_filter")
69    odom.header.frame_id = "filter_odom"
70    odom.pose.pose = Pose(Point(nod.xest[0,0],nod.xest[1,0],0.),
        Quaternion(*odom_quat))
71    odom.child_frame_id = "filter_base_link"
72    nod.pub.publish(odom)
73    nod.loopr.sleep()
```

GRADUATE SCHOOL

INSTITUTE OF TECHNOLOGY OF CAMBODIA

Russian Federation Boulevard, P.O. Box 86 Phnom Penh, Cambodia

Email: graduate@itc.edu.kh