UiA University of Agder

Master

Development of a mobile manipulator robot kit for educational purposes

Conceptual Design, Prototyping, Programming, Software Implementation, and Testing of an Autonomous Robot

HUSAM ZAIN HAIJ

SUPERVISOR
Daniel Hagen, PhD
Emil Mühlbradt Sveen, MSc

University of Agder, 2022
Faculty of Engineering and Science
Department of Engineering Sciences

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	NA

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

I want to acknowledge and give my warmest thanks to my supervisor Daniel Hagen, who made this work possible. His guidance and advice carried me through all the stages of my thesis. I would also like to thank my co-supervisor, Emil Muhlbradt Sveen, who has been extremely patient through the writing stage and of great help during this master thesis.

Abstract

This thesis focuses on designing, building, and developing a prototype of a mobile robot base for a manipulator mobile robot using the Robot Operating System (ROS) 2 as a part of a larger project conducted by the mechatronics department at the University of Agder. This thesis has four main goals: 1) Design an affordable mobile robot base that fits the uArm manipulator. 2) Develop the required ROS2 packages for the selected hardware components in order to perform mobile robot localization, mapping, and autonomous navigation using the SLAM toolbox and Nav2 framework. 3) Construct a prototype of the designed mobile robot base. 4) Test the developed ROS2 packages by performing autonomous navigation in a dynamic environment. The test results showed that the Uiabot prototype succeeded in building a 2D representation of the environment, navigating successfully in this dynamic environment, and avoiding static and dynamic objects.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Robots are electro - mechanical programmable machines i.e. mechatronic systems, that performs various tasks and operations, and they can be fully automated or semi-automated. They are classified in many different ways based on their movement, kinematic structure, degrees of freedom, work-space geometry etc.

Currently, mobile and manipulator robots are the most used robots in the automated warehouses and industries, due to the tasks and operations that they can perform. Mobile robots, as shown in Fig. 1.1a, have the ability of moving around in the surroundings (locomotion). They are not fixed physically to one location, and can be fully autonomous.

Mobile robots are capable to navigate in their environment autonomously without any guidance, or can be guided and moved on pre-defined routes. They are also classified into several categories based on how it moves and type of environment where they are deployed, e.g. land wheeled mobile robots which are the most used in automated warehouses.

(a) Mobile robot [17].

(b) Manipulator robot [16].

Figure 1.1: Industrial robots.

Whereas manipulators Fig. 1.1b, are fixed to one physical location, and perform different operations like pick and place of objects within their work-space and reach limitations.

Combining these two kinds of robots provided a new ability and made them more efficient and convenient i.e. allow the manipulator to move around instead of being fixed to one location. Manipulator mobile robots are simply a robot manipulator on-board a mobile robot that can be used in warehouses and industries to perform mobile and manipulator robots tasks simultaneously. This improves the functionality and effectiveness of robots. The mobile manipulator robots have in the last few years attracted significant interest for use in several fields such as marine, undersea, agriculture, space, rescue, and goods transport in warehouses. Consequently, there is an urgent need for engineers in these fields to have the necessary skills related to these types of autonomous systems.

Recently, the manipulator mobile robots have become a popular research field, due to the rapid development in several fields such as robotics, instrumentation, and AI. Furthermore, integrating these fields open up new aspects of robot utilization and enhances the efficiency and effectiveness of robots. As a result, collaborative robots like the manipulator mobile robots, are highly demanded and represent the backbone of the labor force of nowadays automated warehouses and industries. These advantages made robotics become an area of interest for giant worldwide companies Amazon for instance, and locally as well such as Autostore.

The need for well-educated and skilled engineers in the robotics field is higher than ever, and because of that there is also a need for appropriate robotic kits and tools to assist the educational process. Unfortunately, existing commercially available robots, such as the ¨KUKA KMR IIWA¨ available at the Mechatronics Innovation Lab (MIL), are not suitable and convenient for educational purposes since they use proprietary software and are very expensive. Furthermore, current open-source robot kits are neither affordable, nor easy to adapt for the academic purposes because of a lack of flexibility in changing components and adjusting configurations, such as the *Turtlebot* and *SMARTmBOT* which are ROS-based mobile robots

The robot operating system (ROS) [15] have recently become on of the most utilized software for robot programming purposes. ROS2, the newest version of ROS, is used widely for robotic developments and implementations in the industrial sector since it overcame the communication and safety issues that ROS1 had.

Therefore, a manipulator mobile robotic kit is needed. This robotic kit is required to be affordable, configurable, and can be developed and programmed using ROS2. Providing such a robotic kit will have a great impact on the students and engineers who are interested in fields such as robotics, instrumentation, and computer vision.

1.2 State of the Art

Building and developing a mobile robotic kit for educational purposes can be challenging, and there are many consideration should be taken into account e.g. robot design, hardware selection, tasks that the robot expected to perform, and which software to use in order to program the robot. Furthermore, low-cost and flexibility should be the prime features of the robotic platform, since further developments will be conducted by the students.

Developing a robotic platform has been carried out in many contexts, and different approaches have been employed for the purposes of the robotic kit developments. These approaches can differ in the utilized hardware of the robots and/or the robot programming and software implementation. e.g. T.Gue [9] and J.filho et al [8] used a micro controller and i/o interface as the main control module and processing unit. Whereas J.Zhu et al [38] utilized an industrial computer. And in order for the mobile robot to perceive it's surroundings, [9] used US-100 ultrasonic sensors and A.Lenskiy [13] used analog GPS and RpLiDAR, whereas [38] utilized a set of sensors such as RpLiDAR, IMU and wheel encoders. In addition to the utilized hardware, several software and different approaches can be used for mobile robot programming purposes e.g. [9] and [8] used C and assembly programming language respectively. Whereas, [13] and [38] used ROS.

There are a variety of robot software available, used for the aim of robot programming and implementation. In term of accessibility, robot software can be categorized into two main categories :

- Open source software.
- Closed source software (commercial software)

Commercial software are limited for specific robots and not adaptable for a wide variety of robots and tasks. On other hand, open source software are more flexible and allow the users to adapt and build their own applications based on the type of the robot and tasks to complete. Therefore, open source programs are more popular and have large communities, where developers and users can exchange knowledge and enrich program development e.g. ROS community.

One remarkable state of the art, that is also ROS-based, is the *Turtlebot* created by M.Wise and T.Foote in 2010 [29]. The Turtlebot is an open-source hardware and software robotic kit. It uses ROS for programming and software implementation purposes. The third generation of the *Turtlebot* "*Turtlebot3*" is very relevant for this thesis and constitutes a benchmark for the building and design stages.

Turtlebot3 [30], as seen in Fig.1.2, is a programmable ROS-based mobile robot that constructed for development purposes. It is equipped with the required hardware to performs several tasks e.g. SLAM and autonomous navigation. The *Turtlebot3* has a large community and can be used in the educational process. However, it has two main drawbacks; drive system module and its relatively high cost. The drive system of the *Turtlebot3* consists of DYNAMIXEL module [31], which is one compact module that contains the wheel encoder, gearbox and motor, which not suitable for educational process, where students need to learn and work on these parts. The *Turtlebot3 Waffle* which is the largest in the *Turtlebot3* family costs around $ 1500 for the mobile robot base alone, and around $ 2800 with the OpenManipulator [20]. The OpenManipulator is a manipulator robot compatible with the *Turtlebot3*, and it used with the *Turtlebot3* as a manipulator mobile robot [32], as seen in Fig.1.4.

The next generation of the *Turtlebot* is the *Turtlebot4* [33], seen in Fig.1.3. The *Turtlebot4* is not relevant for this Thesis since it's not flexible and uses Create 3 [5] as the base platform

which is not fully configurable and accessible.

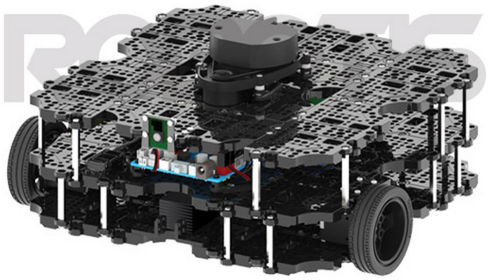


Figure 1.2: Turtlebot3 Waffle [30].

Figure 1.3: Turtlebot4 [33].

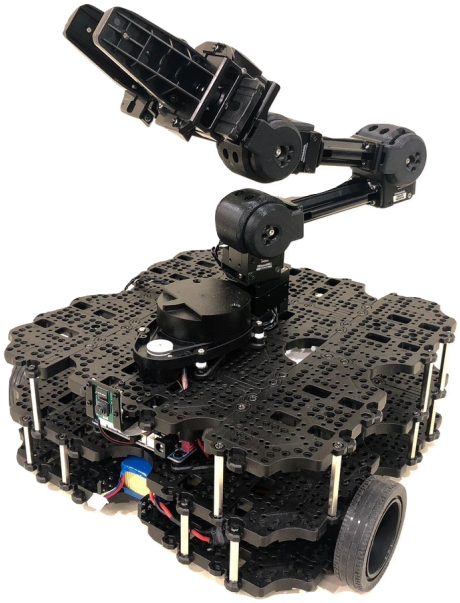


Figure 1.4: Turtlebot3 with OpenManipulator [32].

Figure 1.5: uArm Swift Pro [36].

1.3 Problem Statement and Research Objectives

The department of Engineering Sciences at UiA possess multiple components e.g. Jetson Nano's and uArm's, seen in Fig.1.5, that can be utilized and reused in order to develop the manipulator mobile robots.

The following research question was formulated for this thesis:

What are the current possibilities and strengths of a mobile robotic platform for educational purposes, based on readily available and low-cost hardware (uArm and Jetson Nano), in combination with a currently developing and not fully supported ROS2 framework?

In order to answer the research question, the following objectives will be carried out:

1. Design an affordable mobile robot base that fits the existing manipulator uArm. The mobile base should be flexible and suitable for the education purposes.
2. Develop the necessary ROS2 packages for the selected hardware of the mobile robot in order to perform several operations such as SLAM and autonomous navigation using SLAM toolbox and Nav2 frameworks.
3. Build a prototype of the designed mobile robot using the selected hardware.
4. Test the developed ROS2 packages of the mobile robot prototype.

1.4 Thesis Structure

To achieve the thesis objectives, the report is divided into these chapters:

Chapter 1 This chapter presents an introduction about the importance of the manipulator mobile robots generally, problem statement, the objectives of the project and the related work.

Chapter 2 This chapter discusses the mobile robot *see-think-act* cycle, theory behind the implementation of the ROS2 packages of a mobile robot and the information that the mobile robot required to perform autonomous navigation. Furthermore, it's discusses the structure of the ROS2 frameworks e.g. SLAM toolbox and Nav2.

Chapter 3 Describe the design of the mobile robot base, the selected components, the ROS2 packages development of the selected components, SLAM toolbox and Nav2 implementation, and the performed test on the mobile robot prototype.

Chapter 4 This chapter discusses the robot development issues and the shown results of the ROS2 packages test, SLAM toolbox test, and the autonomous navigation test.

Chapter 5 This chapter present Project contributions in relation to the objectives of the project that stated in chapter 1.

Chapter 2

Theory

This chapter discusses the theory behind the implementation of the ROS2 packages of the mobile robot and data that the mobile robot required to perform autonomous navigation. Furthermore, it's discusses the structure of the ROS2 frameworks e.g. SLAM toolbox and Nav2.

2.1 Autonomous Navigation

The main goal of a mobile robot is to navigate autonomously. Autonomous navigation can be achieved by implementing a *see-think-act* cycle. The *see-think-act* cycle is a control cycle and work order as shown in Fig. 2.1. The mobile robot has to perform sub-tasks and fulfill several requirements in order to navigate autonomously. The mobile robot has to perceive it's environment (perception), localize itself within this environment (localization and mapping), find the optimal path to a given goal (path planning), and then be able to move in it's environment and generate the wheel actions to get from A to B (motion control).

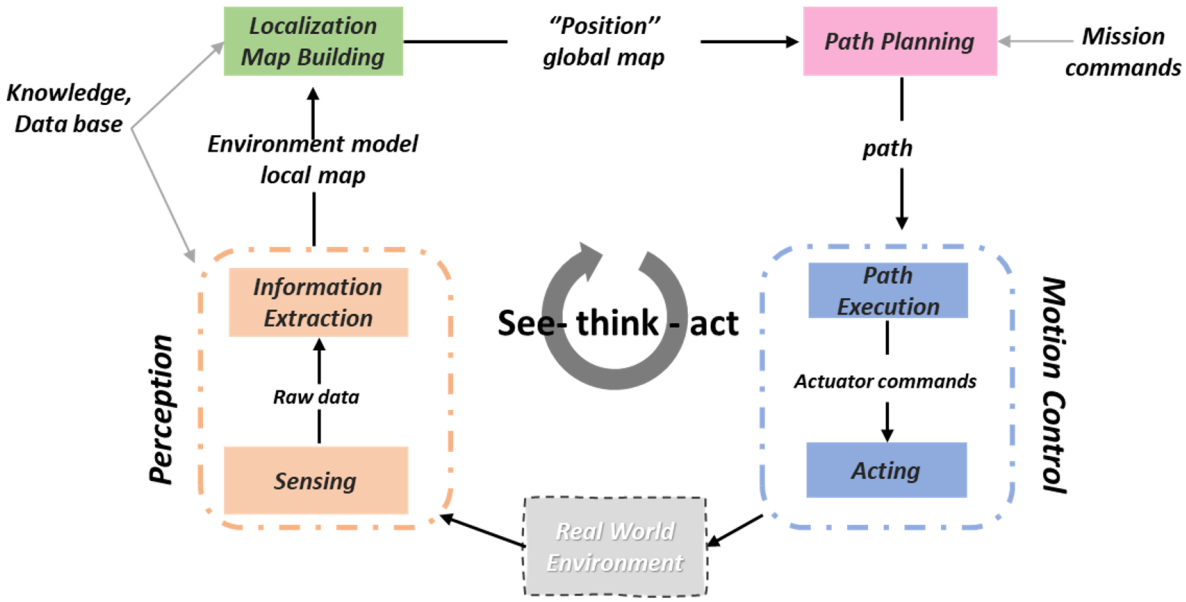


Figure 2.1: *See-Think-Act* cycle for autonomous mobile robots [26].

2.1.1 Motion Control

The first consideration in motion control is how to drive the robot. For wheeled robots there are different types of wheels which impose a number of constraints on the robot motion e.g. rolling constraint and no-sliding constraint. e.g. standard wheel can roll in one direction and should ideally not slip in the other direction. In order to derive the kinematic model of a mobile robot, all the constraints of the mobile robot's selected wheels should be stacked together with rolling and no-sliding constraints.

The kinematic model provides the needed equations to compute the speed of the mobile robot body $\dot{\xi} = [\dot{x},\ \dot{y},\ \dot{\theta}]^T$, from the speed of the individual wheel $\dot{\phi}_i$ and this is called the forward kinematic model, as shown in eq 2.1. Whereas, the inverse kinematic model is used to compute the speed of the individual wheel $\dot{\phi}_i$ from the speed of the mobile robot body $\dot{\xi}$, as shown in eq 2.2.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = f(\dot{\phi}_1, ..., \dot{\phi}_n, geometry) \tag{2.1}$$

$$\begin{bmatrix} \dot{\phi}_1 \\ \vdots \\ \vdots \\ \dot{\phi}_n \end{bmatrix} = f(\dot{x}, \dot{y}, \dot{\theta}) \tag{2.2}$$

Differential Drive Kinematics

A differential drive mobile robot has two drivable wheels, which can be rotated at different angular velocities in order to move the robot. Changing the velocity of the wheels changes the velocity and direction of the mobile robot accordingly. Therefore, the kinematics of the mobile robot can be derived by describing each wheel contribution to the mobile robot motion, as well as the constraints that are imposed by the wheels.

The resultant velocity of the mobile robot $\dot{\zeta}_I = [\dot{x}\ \dot{y}\ \dot{\theta}]^T$ is a result of the locomotion of each individual wheel $\dot{\phi}_i$. The relationship between each wheel angular velocity and the velocity of the mobile robot is the differential drive kinematics. Equation 2.3 and 2.4, are the general stacked equations for a differential drive mobile robot. For more details about these equations refer to [26].

- Rolling constraint equation:

$$\begin{bmatrix} -sin(\alpha+\beta) & cos(\alpha+\beta) & lcos(\beta) \end{bmatrix} R(\theta)\dot{\xi}_I - \dot{\phi}r = 0 \tag{2.3}$$

- No-slide constraint equation:

$$\begin{bmatrix} cos(\alpha+\beta) & sin(\alpha+\beta) & -lsin(\beta) \end{bmatrix} R(\theta)\dot{\xi}_I = 0 \tag{2.4}$$

where, α is the angle of the steering frame from the robot frame, β is the steering angle, l is the distance from the robot frame to the wheel center, and $\dot{\xi}$ is the robot state velocity expressed in the inertial frame, as seen in Fig. 2.2.

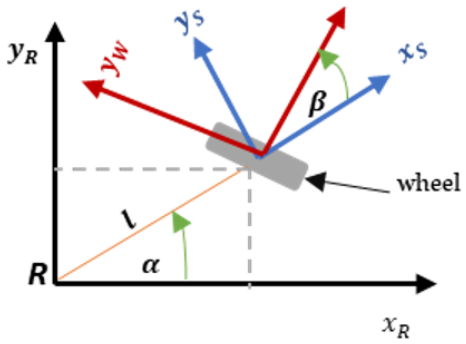


Figure 2.2: Standard wheel coordinate frames.

The stacked eq 2.3 and eq 2.4 can be modified for this differential drive configuration

- Two fixed standard wheels.
- The robot frame (R) in between the wheels, as seen in Fig. 2.3.
- Two passive castor wheels act as contact points to stabilize the robot on the ground.

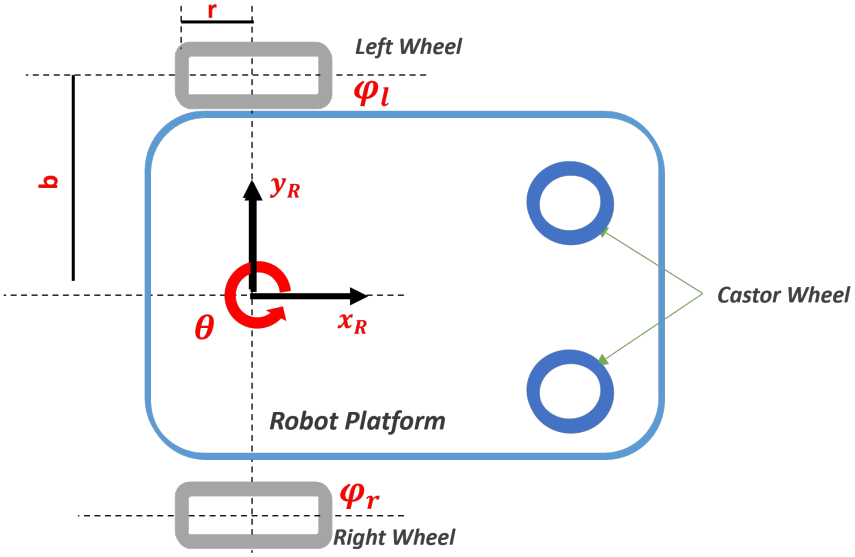


Figure 2.3: Differential drive configuration scheme.

Fig. 2.3 is a top-down schematic view of the built differential drive robot in this project, where

- **b** is the distance from the robot frame to the wheels center.
- **r** is the radius of the wheels.

The two equations 2.3 and 2.4 that were derived for the standard wheel in the inertial frame can also be expressed in the robot frame as,

- Rolling constraint:

$$\begin{bmatrix} -sin(\alpha+\beta) & cos(\alpha+\beta) & lcos(\beta) \end{bmatrix} \dot{\xi}_R = \dot{\phi} r \tag{2.5}$$

or

$$J_1(B_s)R(\theta)\dot{\xi}_R = \dot{\phi} r \tag{2.6}$$

- No-slide constraint:

$$\begin{bmatrix} cos(\alpha+\beta) & sin(\alpha+\beta) & -lsin(\beta) \end{bmatrix} \dot{\xi}_R = 0 \tag{2.7}$$

or

$$C_1(B_s)R(\theta)\dot{\xi}_R = 0 \tag{2.8}$$

For the particular robot configuration in Fig. 2.3,

- Right wheel:

$$\alpha = -\frac{\pi}{2}\ ,\ \beta = 0\ ,\ l = b \tag{2.9}$$

- Left wheel:

$$\alpha = -\frac{\pi}{2}\ ,\ \beta = 0\ ,\ l = -b \tag{2.10}$$

Substituting eq 2.9 and 2.10 in eq 2.5 and 2.7,

- Rolling constraint:

$$\begin{bmatrix} 1 & 0 & b \\ 1 & 0 & -b \end{bmatrix} \dot{\xi}_R = \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix} \begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} \tag{2.11}$$

- No-sliding constraint:

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \dot{\xi}_R = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{2.12}$$

Which means J_1 and C_1 in eq 2.6 and 2.8 is equal to,

$$J_1 = \begin{bmatrix} 1 & 0 & b \\ 1 & 0 & -b \end{bmatrix} \quad , \quad C_1 = \begin{bmatrix} 0 & -1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.13}$$

Equations 2.11 and equation 2.12 can also be stacked to one equation,

$$\begin{bmatrix} 1 & 0 & b \\ 1 & 0 & -b \\ 0 & -1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \dot{\xi}_R = \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} \tag{2.14}$$

or simply,

$$\mathbf{A}\dot{\xi}_R = \mathbf{B}\dot{\phi} \tag{2.15}$$

Where,

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & b \\ 1 & 0 & -b \\ 0 & -1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \quad , \quad \mathbf{B} = \begin{bmatrix} r & 0 \\ 0 & r \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \quad , \quad \dot{\phi} = \begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} \tag{2.16}$$

Now, the differential kinematics can either be found by solving for the velocity state of the mobile robot $\dot{\xi}_R$, or the velocities of the wheels $\dot{\phi}$. Matrices A and B are not square matrices, so standard inverse can't be applied, instead a pseudo inverse is used.

Forward Differential Drive Kinematics

A differential drive mobile robot can't be controlled directly. Deriving the mobile robot can be achieved by controlling the velocity of the individual wheel and using the kinematic model. Computing the velocity of the mobile robot $\dot{\xi}_R$ from the the velocity of the individual wheels $\dot{\phi}$, is the forward differential kinematics.

$$\dot{\xi}_R = (A^T A)^{-1} A^T B \, \dot{\phi} \tag{2.17}$$

Using the matrices A and B and pseudo inverse,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ 0 & 0 \\ \frac{r}{2b} & \frac{-r}{2b} \end{bmatrix} \begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} \tag{2.18}$$

In equation 2.18, the forward velocity, $\dot{x} = r\frac{\dot{\phi}_r + \dot{\phi}_l}{2}$, is the average speed of the two wheels. The no-sliding constraint, $\dot{y} = 0$, is satisfied. The angular velocity, $\dot{\theta} = r\frac{\dot{\phi}_r - \dot{\phi}_l}{2b}$, means the robot will turn to the right and θ is positive when $\dot{\phi}_r > \dot{\phi}_l$.

Inverse differential drive kinematics

Inverse kinematics is used to compute the velocity of each wheel $\dot{\phi}$, from the the velocity of the mobile robot $\dot{\xi}_R$.

$$\dot{\phi} = (B^T B)^{-1} B^T A \, \dot{\xi}_R \tag{2.19}$$

Using matrices A and B and pseudo inverse,

$$\begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} = \begin{bmatrix} \frac{1}{r} & 0 & \frac{b}{r} \\ \frac{1}{r} & 0 & \frac{-b}{r} \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{2.20}$$

Equation 2.20 is the the inverse kinematics. In the inverse kinematics, the rotation of each wheel is explained by two parts. The first one is explained from the motion in x-direction of the mobile robot, $\frac{1}{r}$. And the second part is explained by the angular velocity of the mobile robot, $\dot{\theta}$.

These forward and inverse kinematic models will be used later on in Uiabot implementation for different purposes such as controlling the motion of the Uiabot, and calculating the odometry information.

2.1.2 Perception

Mobile robots can recognize it's environment and know where its in this environment by sensing the surroundings. Mobile robots generally have proprioceptive and exteroceptive sensors. Proprioceptive sensors measure values internally to the mobile robot e.g. motor speed and heading of the robot, and exteroceptive sensors measure information from the robots environment e.g. distance to objects unique features. Exteroceptive sensors such as laser scanner and cameras are used for mobile robot localization and mapping.

Laser scanners, as shown in Fig. 2.4, send a leaser beams to measure the distance between the sensor and the objects that the laser beams hit using time of flight measurements. Information about the elements of the environment e.g. wall, corner etc, can be extracted from these distance raw data in a way the mobile robot can recognize it's surroundings, and localize itself.

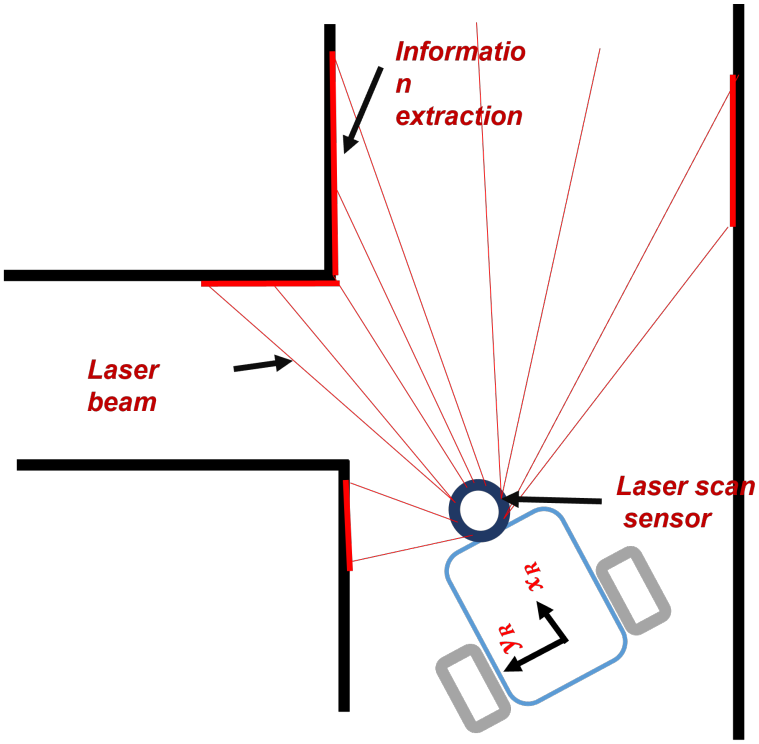


Figure 2.4: Information extraction from laser scanner data.

Apart from Laser scanners and cameras that used to perceive the environment, mobile robot might have other sensors e.g. IMU and wheel encoder to measure it's pose, velocity and acceleration.

2.1.3 Localization and Mapping

Once the mobile robot has perceived the environment and extracted features, it tries to localize it self in this environment by comparing what it has perceived of the environment with a stored map of the environment.

As shown in Fig. 2.5, the mobile robot is in an environment that contain three pillars. The three pillars represent a stored and known map for the mobile robot. The robot starts up from a given point and has no prior information, so it's mathematical prediction about it's own position is equally distributed along the axis as shown in Fig. 2.5.

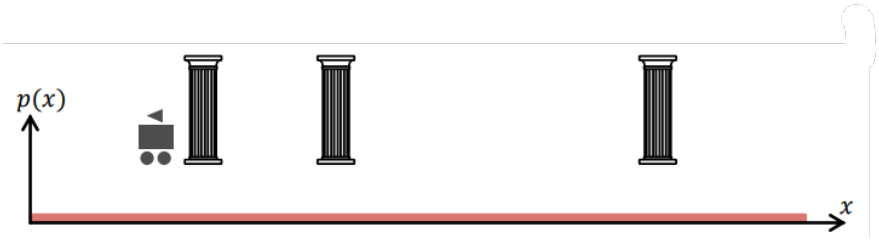


Figure 2.5: Robot starts up with no prior belief [26].

The robot detects a pillar and since the laser scaner has limited range it can estimate from this that it is in front of one of the three pillars that stored in the map, thus the mathematical prediction distribution changes, and there are three peaks as seen in Fig. 2.6.

Figure 2.6: First step: robot perceives environment for the first time [26].

Then, the robot typically perform an action and moves forward. The certainty about the environment becomes less due to the robot's movement. As a result, the mathematical prediction about the robot position is reduced as all the peaks moved forward with the motion of the robot as seen in Fig. 2.7.

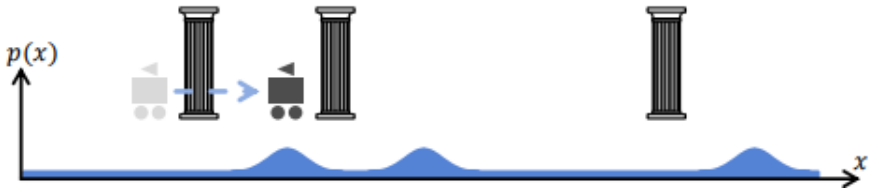


Figure 2.7: Second step: robot moves forward (action step) [26].

Finally, the robot capture a second view of the environment as seen in Fig. 2.8, and again tries to recognize where it is in the environment. As the first detection, if it's only considering the perception it would has three peaks; however, combining this result with the mathematical prediction from the prior action step (blue), then there is a clear strong peak at one point which is the most probable position of the robot, as seen in Fig. 2.9, so it localizes it self and can also start use this information for planning.

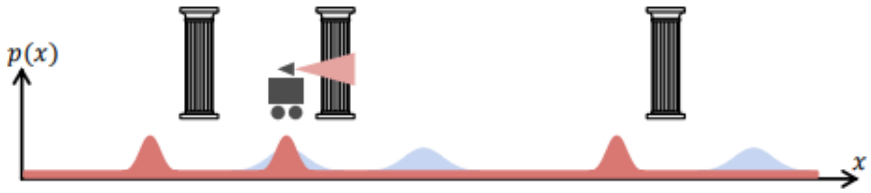


Figure 2.8: Third step: robot takes a second view of the environment [26].

Figure 2.9: Information fusion for localization [26].

2.1.4 Path Planning

The last part in the see-think-act cycle is planning. Planning means how the robot computes a path to travel from A to B based on a mission command provided by a robot operator. The robot starts at a certain point with some knowledge about it's environment. For example, it knows that there are obstacles in the environment, as seen in Fig. 2.10, and it has to move to a certain goal. There could be many feasible paths that robot can follow, so the planning and cognition is to find the optimal path from the current point of the robot to the goal point avoiding the different obstacles. Planing through an environment that contains static and dynamic obstacles can be in some situations a very complex planning process. This planning process is divided into two parts, global and local planning.

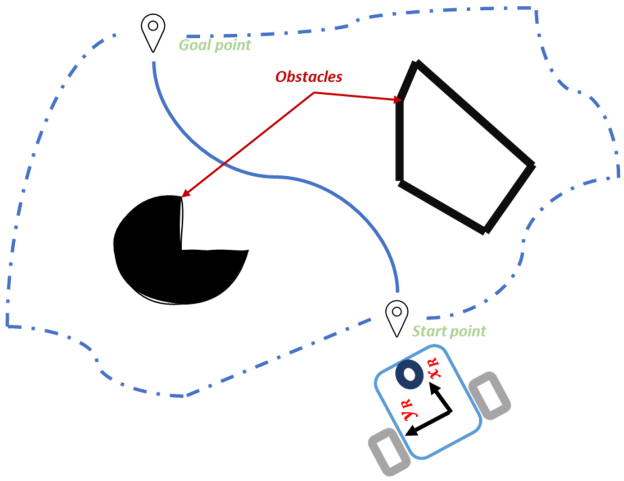


Figure 2.10: Robot path planning.

In most cases of global planning, the environment is divided into smaller parts defined as cells and this method called grid-based representation. Each cell gets a distance value such that the robot select the optimal feasible path from the start to the goal point depending on the cells values. The obstacles are also represented in order to prevent robot collision, as seen in Fig. 2.11. Finding the feasible path to the goal can be achieved by graph search algorithm e.g. A* [10], and Dijkstra's [7].

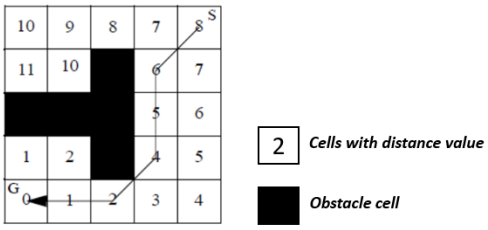


Figure 2.11: Robot global path planning [26].

The local planning ensures that the robot doesn't collide with the obstacles in the environment that are not represented in the map, or if there are dynamic obstacles like pedestrians. As seen in Fig. 2.12, this can be achieved by detecting the obstacles using a sensor e.g. laser scanner, and free space as well, then controlling the motion of the robot to avoid collision and find new obstacle free trajectory which allow the robot maneuver and move forward by selecting the optimal path as defined by the task assignment and perceived environment.

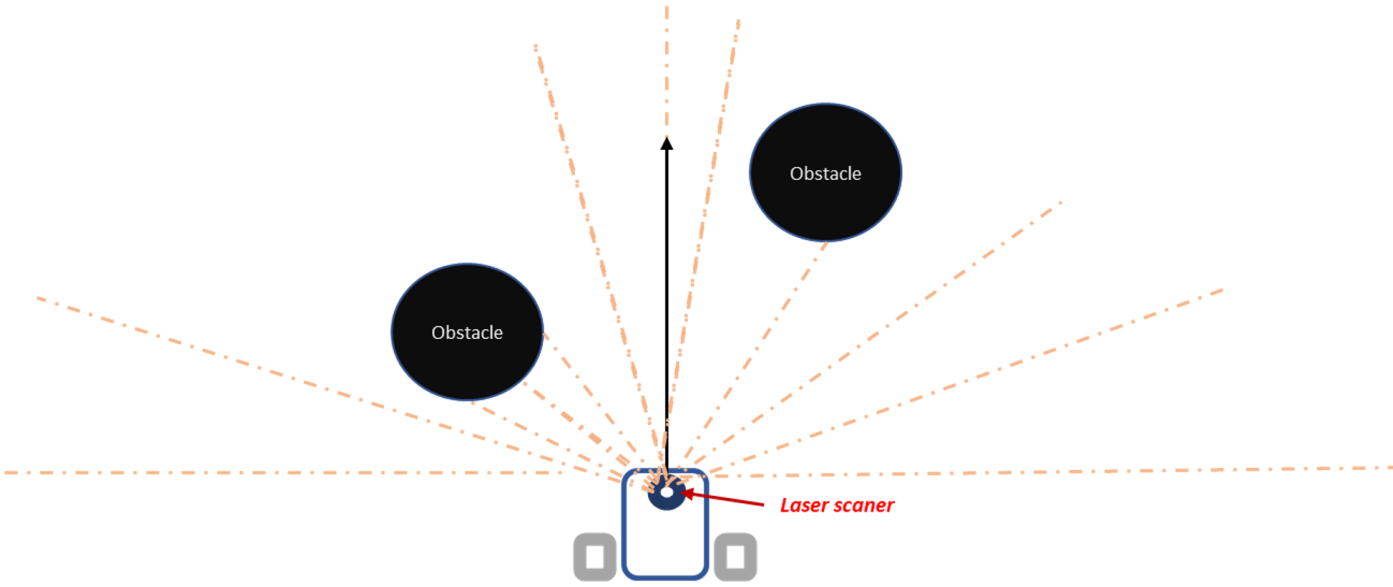


Figure 2.12: Robot local path planning.

2.2 Robot Operating System

ROS is a meta operating system that can be built on top of an operating system to allow different processes (nodes) to communicate with each other at runtime.

ROS provides the expected services from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

ROS can be used with almost any drivers, actuators and sensors as long it has a software interface, which can be ROS-wrapped in order to make these components communicate together via ROS messages. Furthermore, ROS has a powerfull tools like Rviz (robot visualization) and Gazebo, which allow to build a digital twin of the robot and visualize it without the need for the actual robot in order to perform testing and see how the system will work. Currently, ROS is the most popular robot software and has a large community where users and developers can interact and publish different libraries and packages for a variety of components.

ROS Packages

ROS uses packages to organize code and control applications. A ROS package might consist of nodes, data-set, configuration file, or anything else that logically constitutes a useful module. In another words it can be considered as a container for the ROS code. In order to be able to use, install and share codes, it should be kept in order in ROS packages.

ROS Nodes

A ROS node is the smallest programmable unit in a ROS application, it's an executable program which contains a part of robot code. Normally, ROS application has more than one node. A ROS application for a mobile robot, can have a wide range of nodes, which are built for different purposes. e.g. node for computing robot inverse kinematics, node to send a control command to the actuators of the mobile robot, node to get joints position and velocity, and many other nodes for many different purposes.

Fig. 2.13 shows the structure of a ROS application for sending commands to the Uiabot using a keyboard. As shown in the figure, keyboard teleport node is used to wrap the keyboard driver (python driver, c++ driver etc), so it can communicate with the other nodes and send the robot's velocity command in form of ROS message.

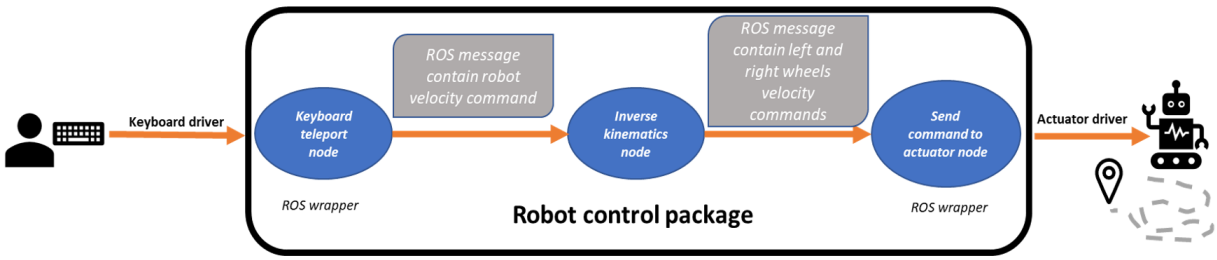


Figure 2.13: ROS application structure.

ROS Nodes Communication Methods

There are three methods in which nodes can communicate and exchange data, which are topics, services and actions.

Topics are named buses over which nodes exchange messages. This is the most used method for nodes communication in ROS, and it's one-way communication. A node that broadcast a message to a topic name called a publisher, whereas a node that receives the message is called a subscriber node, and it has to subscribe to the topic. A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

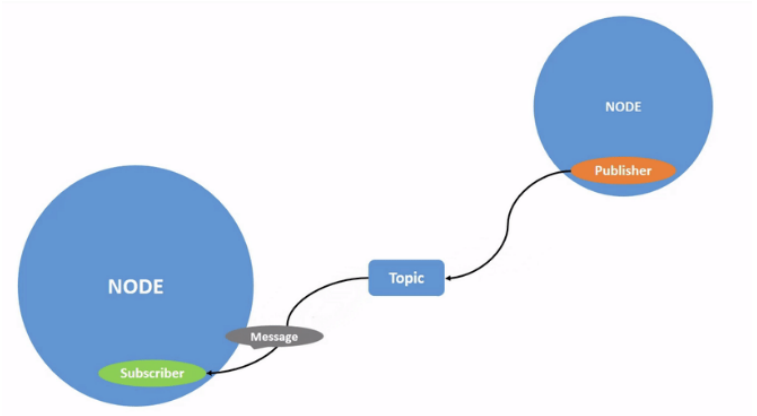


Figure 2.14: ROS topic method [24].

Topic is one-way communication, thus its not always sufficient. A node might need to execute/perform task for a short time or provides data when asked. Service node is the preferable method to do so. It's based on a call-and-response model. Unlike topic's publisher-subscriber model where the data is published continually, it only provide data when they are specifically requested by a client.

As shown in Fig. 2.15, the service client node send a request to the service, this request will be sent to the service server which will response to the request and send a response message to the service client.

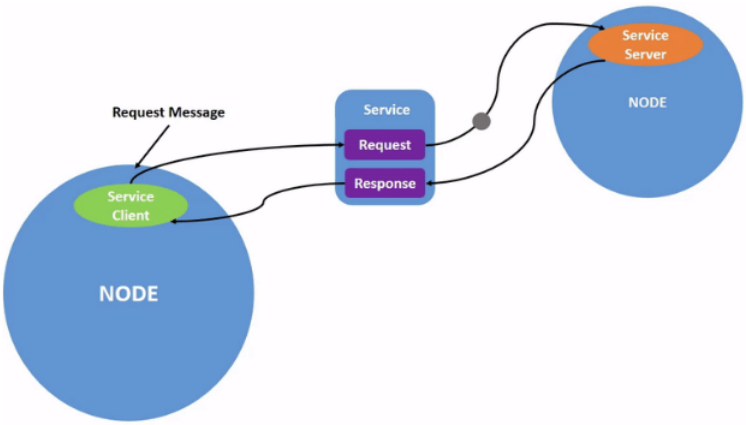


Figure 2.15: ROS service method [24].

However, service node suffers from one draw back. If the service for any reason takes a long time to complete the task i.e. a request is sent and it took a lot of time to preform the task, there is no indicator of the state of the task. This might effect the other nodes that depend on the response of the service and there is no way to cancel the execution. Therefore, services are not used for long-running processes. For such cases there is a third method called "Action".

Action is used for long-running tasks, and contain three parts; a goal, feedback and a result. It's a combination of topics and services, its functionality is similar to services, but unlike the services it can be canceled. Furthermore, it provides a steady feedback where a user can know the state of the task.

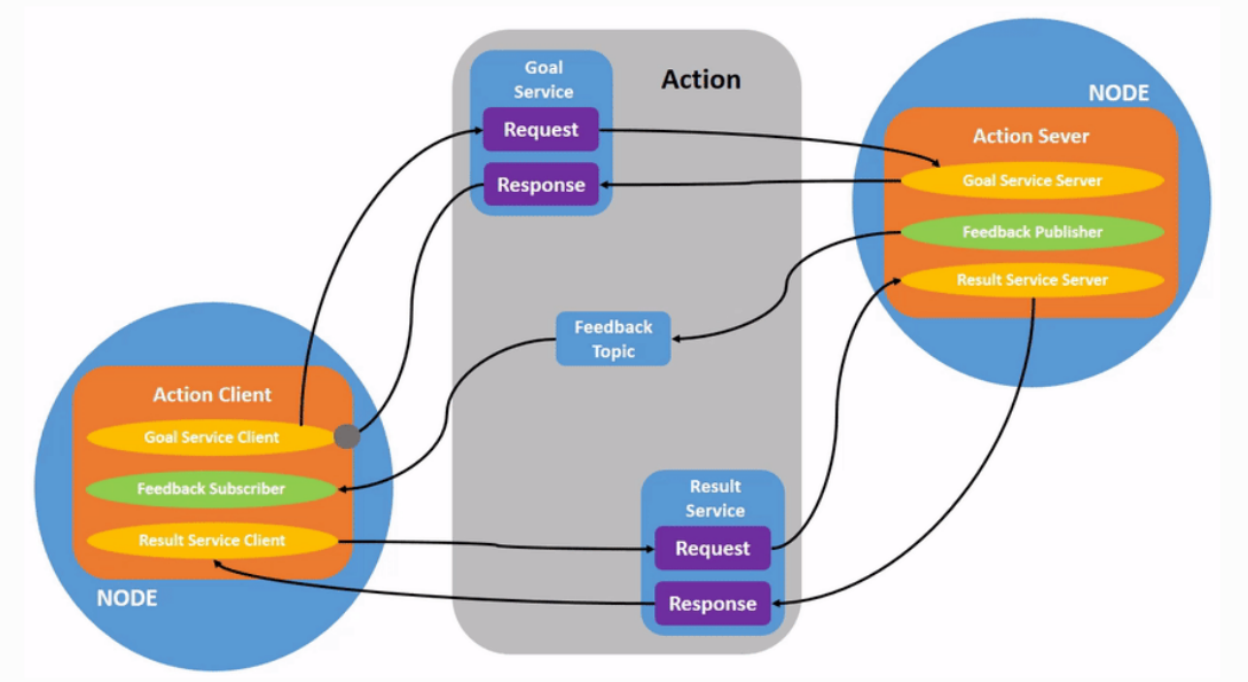


Figure 2.16: ROS action method [24].

2.2.1 Simultaneous Localization and Mapping Toolbox

In order for a mobile robot to navigate autonomously, it has to achieve the localization and mapping part successfully. Localization is the estimation of the mobile robot's pose and path when a map of the environment is provided. On the contrary, building a representation of the environment knowing the mobile robot's path is mapping.

Simultaneously localization and mapping (SLAM) technique is to build a model of the environment while estimation the pose of the mobile robot. SLAM uses the information collected from the proprioceptive and exteroceptive sensors. The collected information are the displacement of the mobile robot estimated from the odometry, and the extracted feature from the laser scanner.

SLAM toolbox [28], is a ROS package that provides a solution for the SLAM problem. The SLAM toolbox uses graph-based algorithms which estimate the full trajectory of the mobile robot using the complete set of the measurements.

In order to use the SLAM toolbox package, there are two requirements:

1. The laser scanner information, published using the laser scan message on the */scan* topic.
2. The $odom \Rightarrow base\_link$ transform.

SLAM methods that uses laser scanner data are the most robust SLAM technique [14].

2.2.2 Navigation Stack

Autonomous navigation is an implementation of the *see-think-act* cycle. The mobile robot needs to perceive the surroundings, localize itself in the environment and plan a path to certain goal.

ROS2 provides a solution and framework called "Nav2". This framework is designed to be as general purposes as possible , however it can be easily adapted for a wide variety of mobile robots. In order for Nav2 framework to work properly, it requires some data from different sensors and setups to be processed first.

Nav2 uses a behavior tree for navigational task orchestration, and employs new methods designed for dynamic environments applicable to a wider variety of modern sensors.

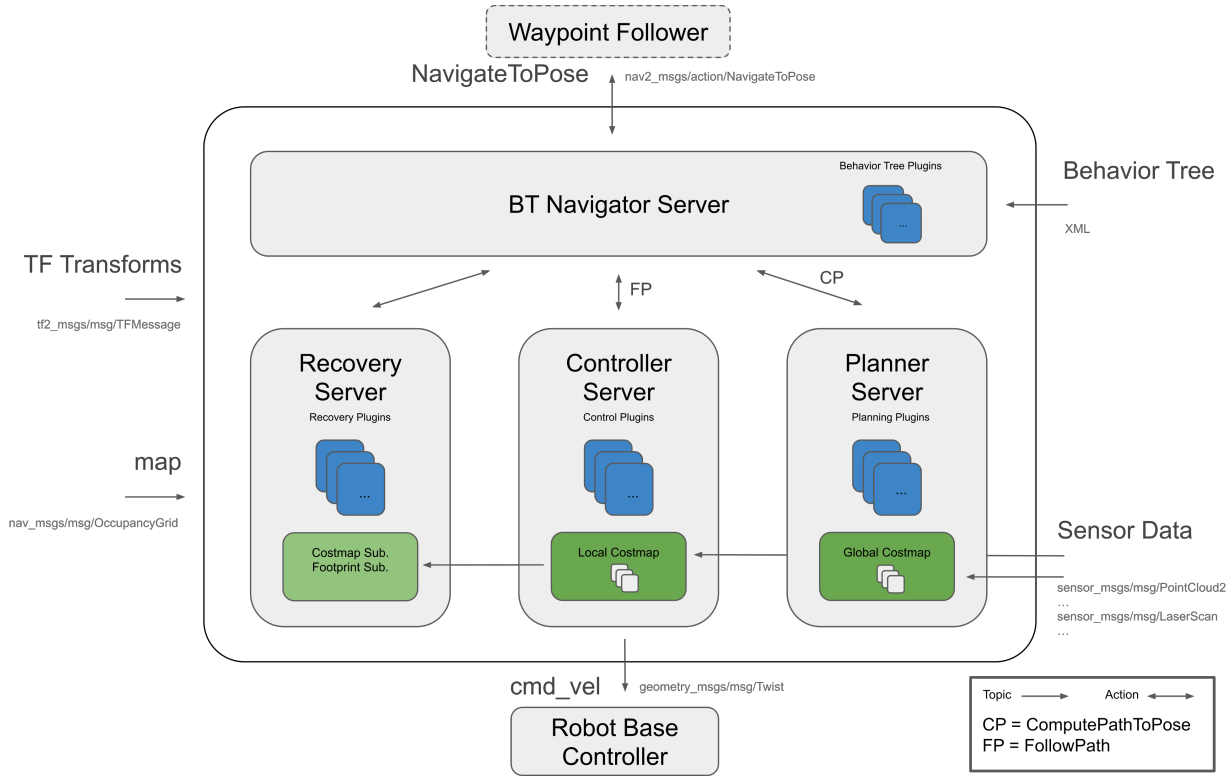


Figure 2.17: NAV2 setup. Foto: navigation.ROS.org

Nav2 uses three type of servers; planner, controller and recovery. Lately, a smoother server is added in Nav2 to improve the planned path. Controller and planner are the backbone of the navigation task, recovery server assist the mobile robot to overcome different issues and handle problems in a way that the mobile robot fault-tolerant.

The planner server is responsible for the planning and cognition part in the *see-think-act* cycle. The main function of the planner server is to determine a feasible optimal path to the goal. There are different type of path planning algorithms and plugins in Nav2 that can be used e.g. *NavFn* Planner, *SmacPlannerHybrid*, and *ThetaStarPlanner* in the planner server for path planning.

There are two requirements for the planner server in order to plan a path:

1. Proprioceptive and exteroceptive sensors information e.g. odometry and laser scan.
2. Global environment representation.

The controller server is responsible of motion control along with the drive system of the mobile robot in the *see- think - act* cycle, by providing the control signal to the mobile robot to follow the planned global path. The controller server require access to the representation of the local environment. In return, it publishes the velocity signals of the mobile robot $\dot{\xi} = [\dot{x} \ , \ \dot{y} \ , \ \dot{\theta}]$ as a twist message on the */cmd_vel* topic. These velocity signals of the mobile robot then will be converted to velocity signal for each wheel using the inverse kinematics. The goal of the recovery server is to handle unknown and failure situations and to ensure that the mobile robot is a fault-tolerant system.

As shown in Fig. 2.17, different requirements should be processed first, in order to accomplish an autonomous navigation task. The following data have to be obtained:

- Transforms data.
- Sensors data.
- Behavior tree data.
- Map.
- Navigation goal.

Transformations Data

ROS has two important REP's, which are ROS documents that detail different standards set by the ROS community. This project is also adheres to these standards and conventions,

- REP 105 - coordinate frames for mobile platform.
- REP 103 - standard unit of measure and coordinate conventions.

REP 105, defines naming conventions of the different coordinate frames e.g. *map*, *odom* and *base_link* coordinate frames used for mobile robots in ROS. The *map* coordinate frame represent the world fixed frame that is used for globally-constant representation of distances. The *odom* coordinate frame is the starting position of the mobile robot. It's fixed and mainly utilized for locally-consistent representation of distances. The *base_link* coordinate frame is a frame attached normally to the center of the robot chassis and rotational center.
REP 103, is defining the standard measurement units and other conventions to minimize integrating issues between the various ROS packages.

Nav2 requires the following transformations to be published in order to work properly:

- *map* $\Rightarrow$ *odom*.
- *odom* $\Rightarrow$ *base_link*.
- *base_link* $\Rightarrow$ *laser_link*, and other sensors base frames.

The *map* $\Rightarrow$ *odom* transformation is usually provided by the localization and mapping packages in ROS e.g. Adaptive Monte-carlo Localizer (AMCL) in Nav2 framework and SLAM toolbox.

The *odom* $\Rightarrow$ *base_link* transformation is normally published either by the mobile robot odometry system, or by using the robot localization package that fuses odometry data of wheel encoders and IMU sensor.

The rest of transformations e.g. *base_link* $\Rightarrow$ *laser_link*, are static transformations and can be published either by using the TF2 static ROS package, or using robot state publisher and the unified robot description format (URDF).

URDF is a xml file that describes the shape of the physical robot in Rviz. It contains the entire physical description of a robot. Writing a URDF file to describe a robot body demands a knowledge of the robot construction. The robot body consist of:

- Links- are the rigid parts of the robot e.g. robot chassis, wheels, sensors, etc. These links are connected and interact to each other by joints.
- Joints- define the motion and relationship between the links.

For a differential drive mobile robot, there are two revolute joints (wheels joint) between the wheels and chassis (*base_link*), the rest are fixed joints (no motion) like the one between the *laser_link* and *base_link*.

Joint state publisher is a ROS package that keeps track position and velocity of all robot joints and publish these data to ROS as sensor message. Whereas, the robot state publisher take the messages and data that are published by the joint state publisher and robot description in URDF, and provides the position and orientation of the robot coordinate frames, and publish this data to the TF2 package. The robot state publisher handles all the static transformations that are required in Nav2 as mentioned in sec 2.2.2 such as the *base_link* ⇒ *laser_link* transformation.

2.4.5.2 Sensors Data

Many different sensors can be used in a mobile robot. The proprioceptive and exteroceptive sensors are both essential in the navigation task. They provides the needed data that is used to compute the odometry information and to extract features from the environment.

Odometry Data

There are a wide range of sensors that can be used to obtain odometry information, such as encoder, radar, lidar, IMU, etc, or a sensor fusion can be used to fuse different type of sensors to obtain more accurate odometry information. Normally, wheel encoders which drift over distance, are fused with IMU which drift over time.

Odometry information are required by Nav2 to be published on the */odom* topic using the odometry message. The odometry message contains:

- Header message: used to publish timestamped data in a specific coordinate frame e.g. *odom* frame.
- Pose message: used to publish the robot position and orientation to a relative frame defined in the header message.
- Twist message: used to publish the linear and angular velocity relative to a frame specified in child frame_id e.g. *base_link* frame.

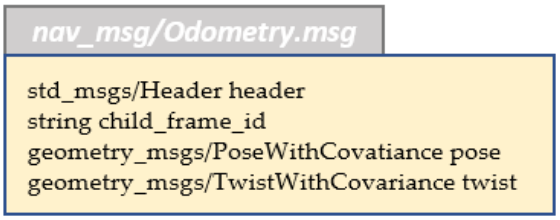


Figure 2.18: Compact message definition.

In addition to the odometry information, Nav2 requires the *odom* coordinate frame to be attached to the other frames in the system, through a dynamic transform which is the *odom* $\Rightarrow$ *base_link* transform. Unlike the static transforms that are published using the URDF file and the robot state publisher package, this transform can be published using TF2 broadcaster using the same node that publishes the odometry information, or using other frameworks such as robot localization package.

Laser Scan Data

The laser scan data in Nav2 have to be published using the the laser scan ROS message on the *scan* topic. The laser scan message contains the following :

- Header message: is used to publish timestamped data in a specific coordinate frame e.g. *laser_link*.
- Angle min: start angle of the scan [rad].
- Angle max: end angle of the scan [rad].
- Angle increment: angular distance between measurements [rad].
- Time increment: time between measurements [seconds] - if the scanner is moving, this will be used in interpolating position of 3D points.
- Scan time: time between scans [seconds].
- Range min: minimum range value [m].
- Range max: maximum range value [m].
- Ranges: range data [m].
- Intensities: intensity data.

Map, Behavior Tree and Navigation Goal

The map data can be obtained either by a static map as *yaml* file or by using the SLAM toolbox package. The behavior tree is obtained by the Nav2 configuration file. The goal of the navigation is provided by Rviz or by publishing a *nav2_msgs/actions/NavigateToPose* message from the command line.

Chapter 3

Methods

3.1 Conceptual Design

Uiabot is a manipulator robot fixed on a mobile platform to perform specific tasks and operations such as localization, mapping, navigation, pick and place, etc. These tasks define the selection of the components that are needed to design and construct the robot.

The Swift Pro uArm [36], was selected as the manipulator robot for the Uiabot. However, the manipulator ROS2 package implementation is not within the scope of this project.

Although the Uiabot components have to be affordable, they should provide the required accurate data in order to accomplish the *see-think-act* cycle. Based on the following provided hardware, a conceptual design was created for the Uiabot :

- Manipulator.
- Drive system (motors, gearboxes, wheel encoders and motor controller).
- RpLiDAR.
- Inertial Measurement Unit (IMU).
- Single board computer.

Fig. 3.1 and 3.2, show the CAD model of the conceptual design.

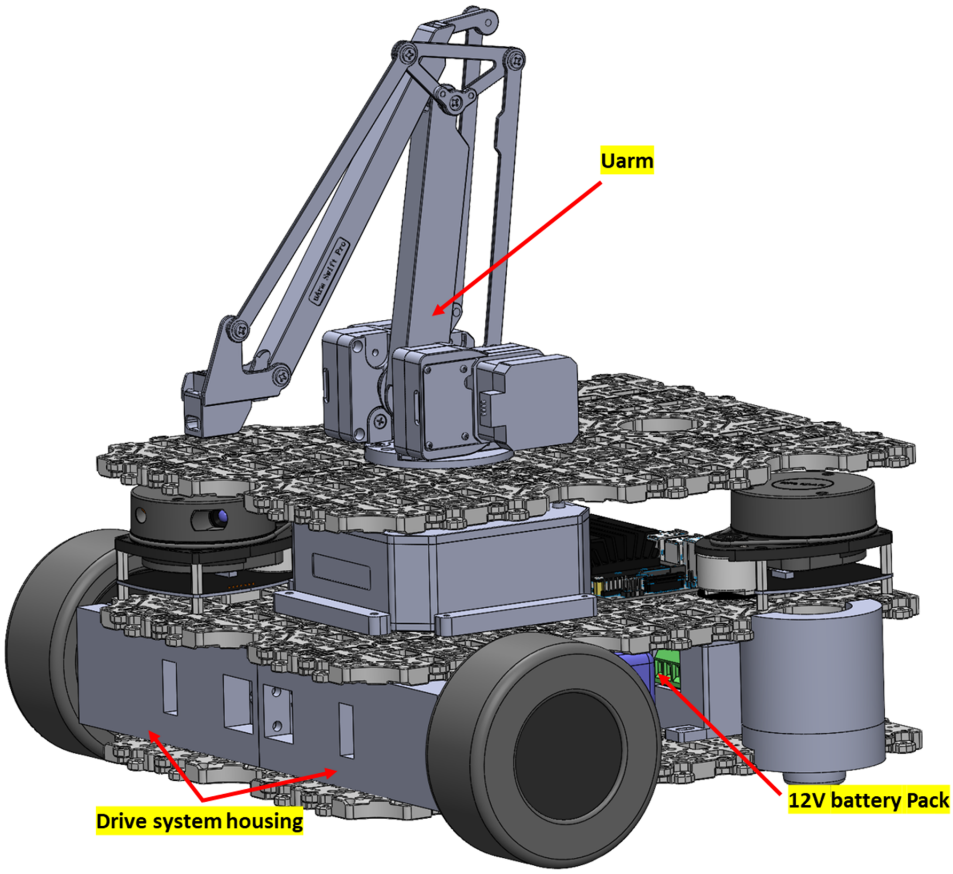


Figure 3.1: Uiabot front-left view.

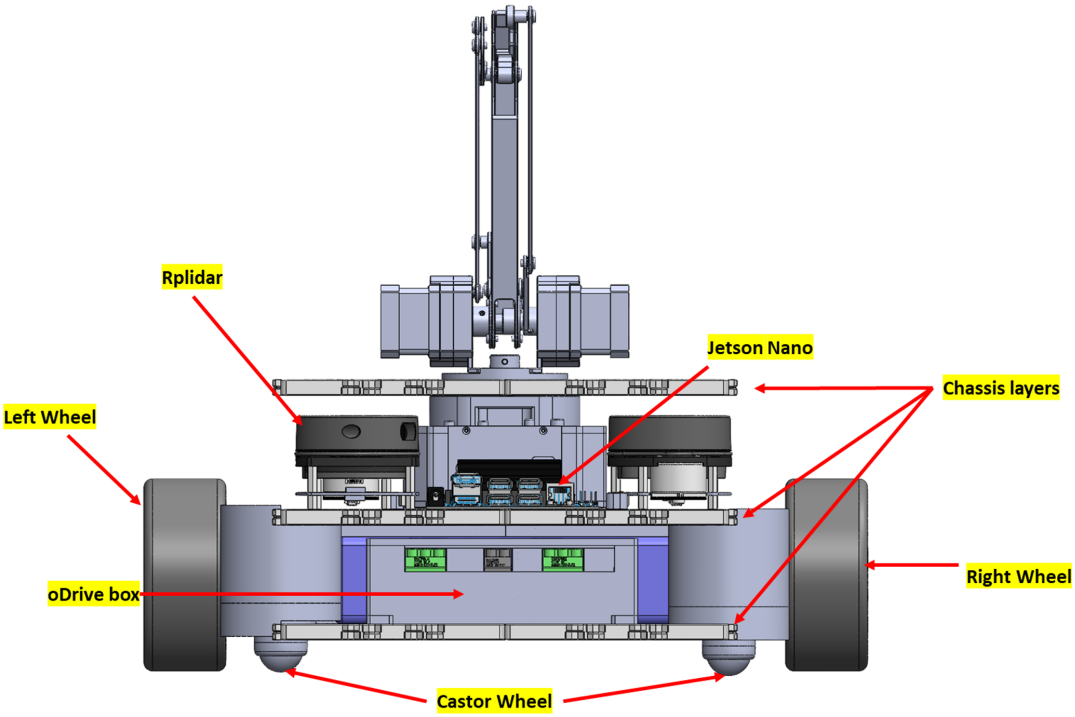


Figure 3.2: Uiabot back view.

3.2 Prototyping

Based on the conceptual design, a prototype of the mobile platform was designed first, as seen in Fig. 3.3, then built using the provided hardware and the 3d printed chassis, as seen in Fig. 3.5 and 3.6.

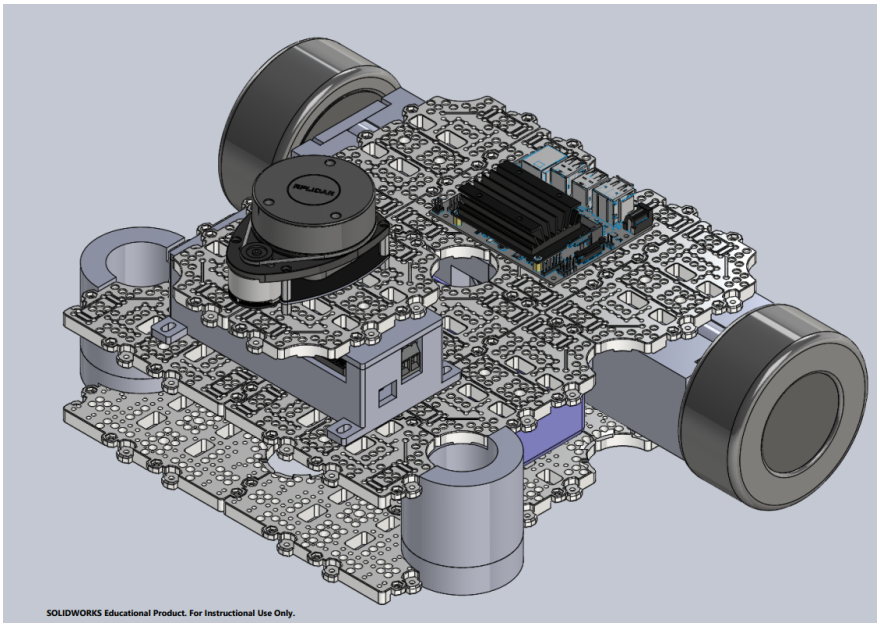


Figure 3.3: Uiabot CAD model

The hardware wiring of the mobile platform shown in Fig. 3.4. A buck converter is used to convert the power supply voltage from 12V to 5V.

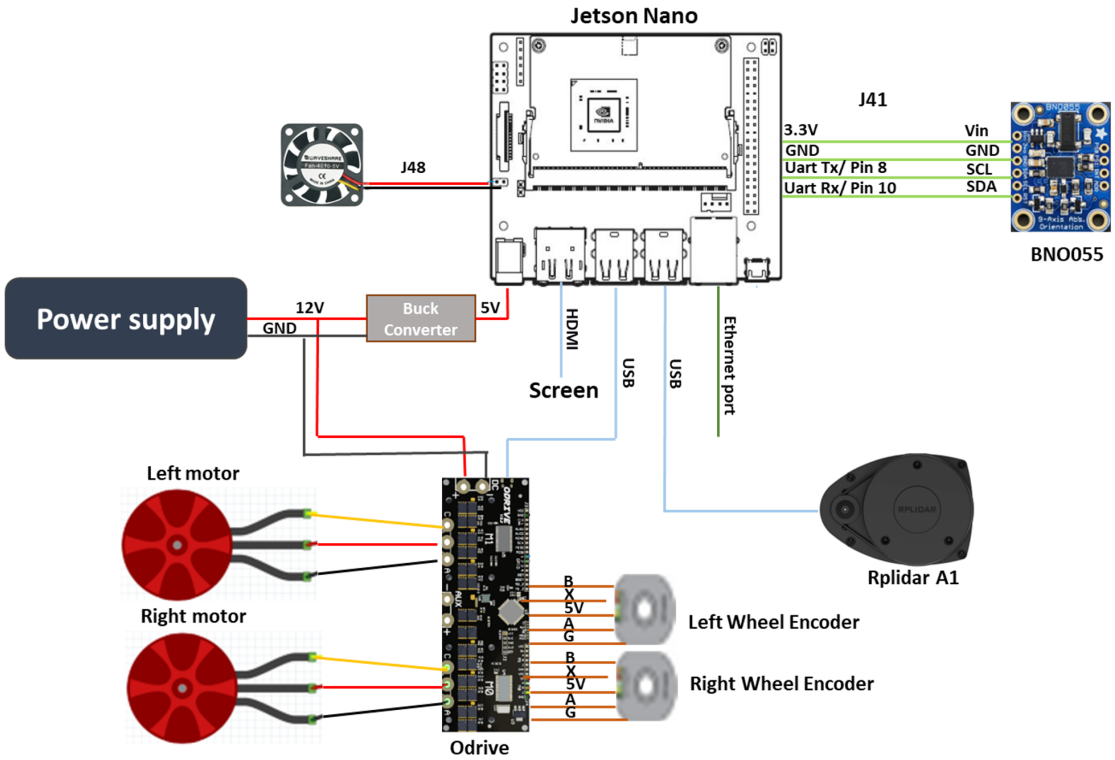


Figure 3.4: wiring diagram

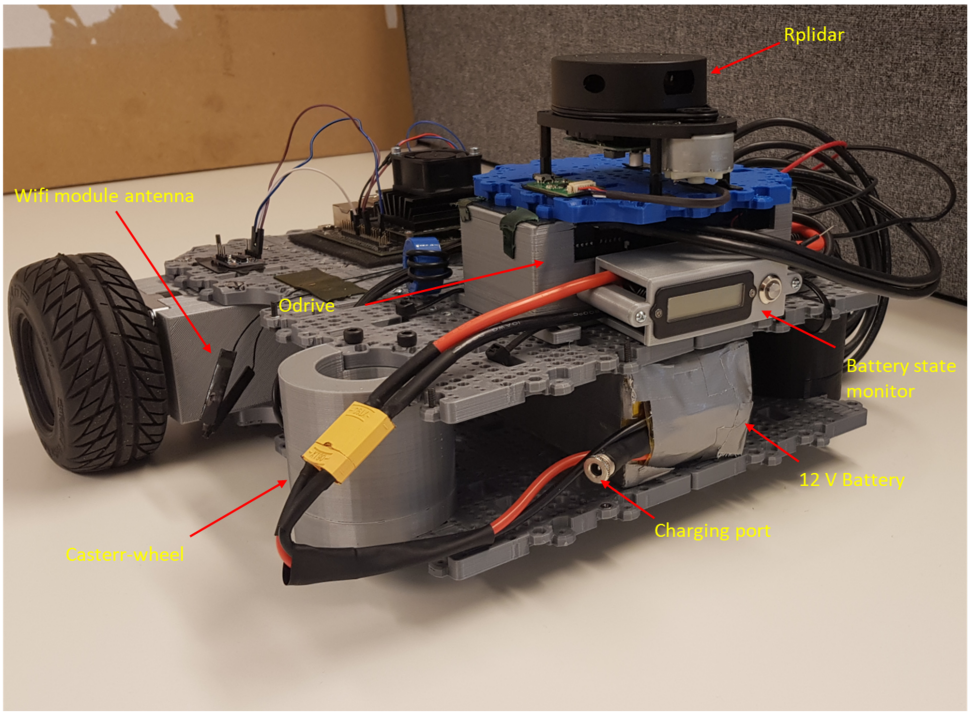


Figure 3.5: Mobile platform prototype front-right view.

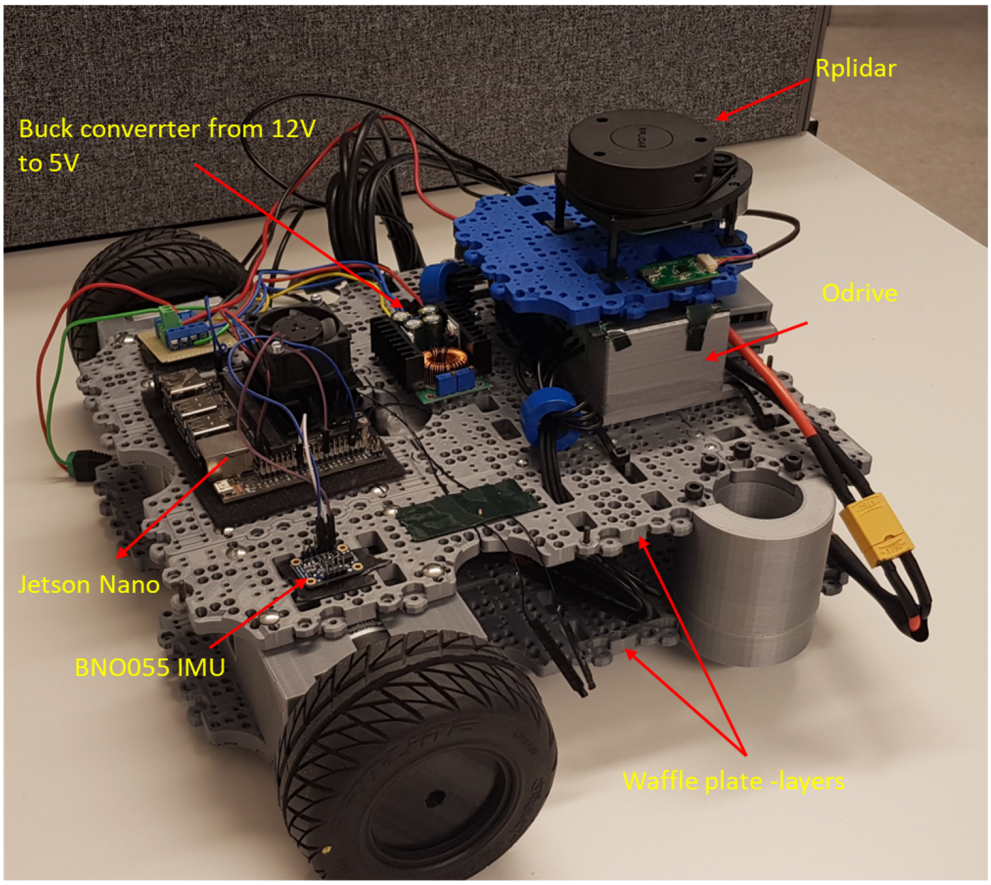


Figure 3.6: Mobile platform prototype back-right view

3.2.1 Robot Chassis

Further development will be carried out on the Uiabot such as adding new sensors, integrating the manipulator and changing the robot shape and body. Therefore, a configurable chassis is required. *Turtlebot3* provides a smart solution, "Turtlebot3 waffle plate" as seen in 3.7. The Turtlebot3 waffle plate was adopted in the Uiabot. These plates, as seen in Fig. 3.8, are configurable plates, that can be assembled in different shapes to form a layer, which can be modified so the the robot can be built in different shapes and sizes.

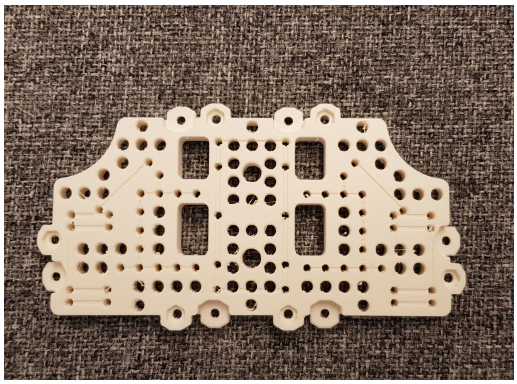
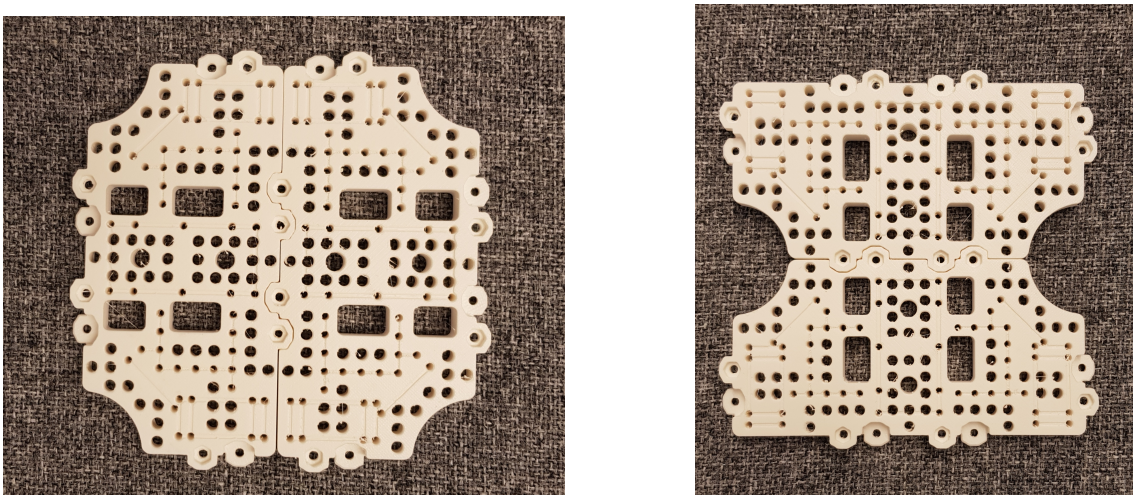


Figure 3.7: 3D printed Turtlebot3 waffle plate.



(a) Plates configuration 1

(b) Plates configuration 2

Figure 3.8: Different plates configurations

The chassis of Uiabot base consists of two castor wheels, two driven wheels, and 20 plates that form 2 layers (10 plates each) . These plates were 3D-printed at UiA lab, and various components were mounted on them using bolts and rivets.

3.2.2 Single Board Computer

Single board computer (SBC) is the brain of the mobile robot that contains the microprocessor which holds the main operating system (Linux) and the meta operating system ROS2.
There are many different types of SBC's in different features and costs. Raspberry Pi and Jetson Nano are the most used SBC's for these kinds of projects and applications. Table 3.1 shows a comparison between the primary features of Raspberry Pi4 and the Jetson Nano.

Rapberry Pi4 Vs Jetson Nano		
Feature	**Jetson Nano**	**Raspberry Pi 4**
CPU	Quad-Core ARM Cortex-A57 64-bit @ 1.42 Ghz	Quad-core ARM Cortex-A72 64-bit @ 1.5 Ghz
Memory	4 GB LPDDR4[1]	4 GB LPDDR4 [2]
GPU	NVIDIA Maxwell w/ 128 CUDA cores @ 921 Mhz	Broadcom VideoCore VI (32-bit)
USB	4x USB 3.0, USB 2.0 Micro-B	2x USB 3.0, 2x USB 2.0
GPU Pin	40-pin GPIO	40-pin GPIO
Video encode	H.264/H.265 (4Kp30)	H264(1080p30)
Video decode	H.264/H.265 (4Kp60, 2x 4Kp30)	H.265(4Kp60), H.264(1080p60
Price	$99 USD	$55 USD

Table 3.1: Jetson Nano Vs the Raspberry Pi 4.

The one remarkable feature that Jetson Nano has over the Raspberry Pi, that it has more capable GPU (Graphical Processing Unit). Jetson Nano GPU's are advantageous when training deep learning algorithm, or during processing of image data.

Uiabot will be employed for different tasks and applications in the robotics and computer vision courses. In these courses, sensors e.g. L515 Intel Realsense lidar and RGB cameras, are used. Therefore, the Jetson Nano A02 was chosen. In addition, several Jetson Nano's were available at UiA, and using them reduced the total cost of Uiabot implementation.

[1]The NVIDIA Jetson Nano Developer Kit available in two options including 2 GB or 4 GB RAM.
[2]The Raspberry Pi 4 available in several options including 1 GB, 2 GB, 4 GB or 8GB RAM.

3.2.3 Drive System

Since the mobile platform of the Uiabot is a differential drive robot, two brushless motors [3] and oDrive motor controller were provided by the supervisor.
Uiabot contains heavy hardware and can be used for goods transportation, which implies relatively high torque requirements to provide the required acceleration and the ability to overcome the high friction caused by the heavy weights. In general, autonomous mobile robots don't require high translational velocity and because of that a gearbox with gear ratio equal to 20:1 is used to gear down the wheel's angular velocity.

The oDrive has ability to control two motors simultaneously. It has variety of control modes e.g. position and velocity control modes. The oDrive can be interfaced using different types of communication protocols such as USB, CAN, UART, etc. Two planetary gearboxes [21], were coupled with the two brushless motors. Also, two capacitive encoders [1] mounted to the shaft of the motors to measure motors shaft angular position and velocity.

Drive System Wiring and Configuration

The Uiabot left motor is connected to M1 (axis 1) of the oDrive as shown in Fig. 3.9, whereas right motor is connected to M0 (axis 0). The oDrive is connected with the Jetson Nano via USB cable. This drive system is powered up with voltage of 12V for the 24V board variant that is used in this project.

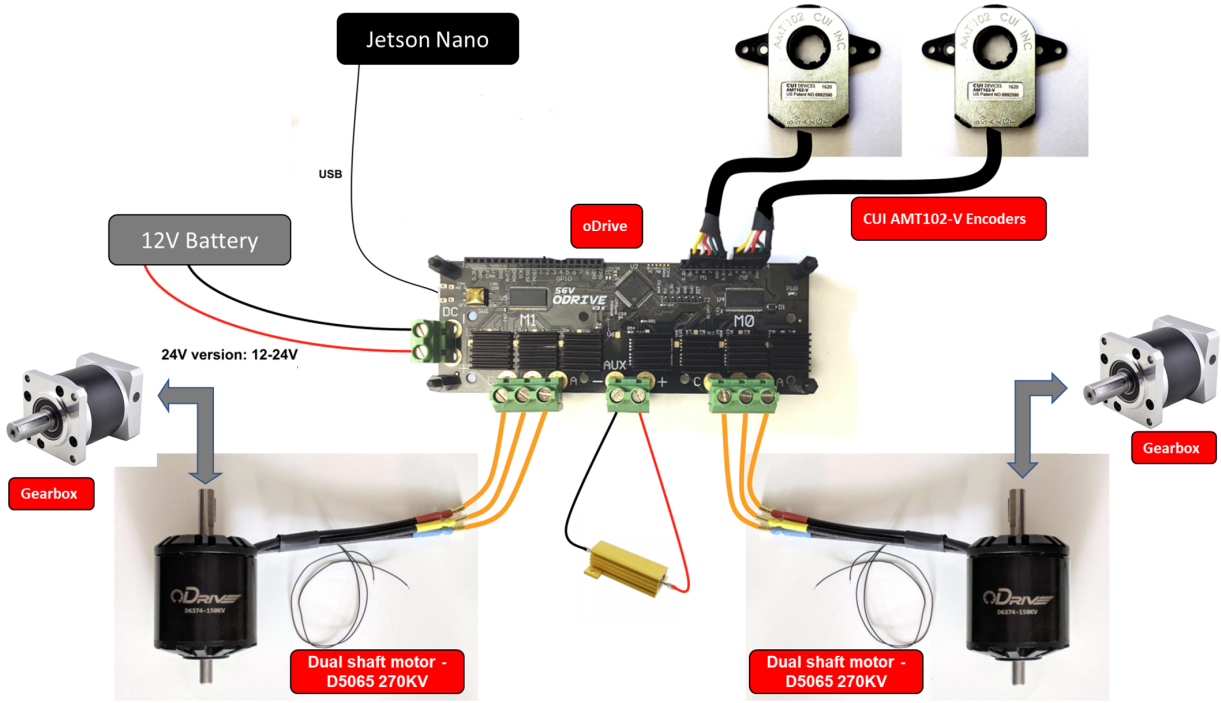


Figure 3.9: Drive system wiring

[3] dual shaft motor - D5065 270KV

oDrive is configured using it's own utility called "oDrivetool". It's a python language based tool, so python commands are used in order to configure a wide range of oDrive and motor parameters. These configuration is saved and loaded into the oDrive memory. The primary configured parameters are shown in table 3.2.

Configuration Parameters	
Parameter	**Value**
Current limit	10 [A]
Velocity limit	30 [rev/sec]
Enable Brake Resistor	Yes
Brake Resistor Value	0.05 [Ω]
Negative Current	default
Pole Pairs	7
Torque constant	0.038

Table 3.2: oDrive configuration parameters.

3.2.4 Perception System

In order for Uiabot to perform *see-think-act* cycle and navigate autonomously, it has to perceive it's environment and estimate it's pose in this environment. Therefore, the Uiabot is equipped with set of sensors (wheel encoders, IMU and RpLiDAR). Furthermore sensor fusion is used to fuse the wheel encoder and IMU data in order to obtain a better estimation of the Uiabot position and orientation.

IMU Sensor

BNO055 is 9-Axis Absolute Orientation Sensor. Uiabot is equipped with BNO055 [27], which provides an estimation of the Uiabot position and velocity. BNO055 provides several output data, such as:

- Absolute Orientation(Euler Vector).
- Absolute Orientation (Quaterion).
- Angular Velocity Vector.
- Acceleration Vector.
- Magnetic Field Strength Vector.
- Linear Acceleration Vector.
- Gravity Vector.
- Temperature.

BNO055 communicates to the Jetson Nano through serial communication. Two types of protocol are used in order to establish a serial communication between the Jetson Nano and the BNO055, I2C and UART protocols. These two protocols were used, and two different ROS2 packages were developed in order to obtain the IMU data as shown in sec 3.3.

BNO055 and Jetson Nano Connection via I2C

The Jetson Nano has I2C port on the J41 header that can be used for serial communication between the Jetson Nano and the BNO055. Wiring between the Jetson Nano and the BNO055 shown in Fig. 3.10 and table 3.3.

I2C connection	
BNO055 Pin's	**Jetson Nano Pin's**
SCL pin	J41 pin 5 (SCL)
SDA pin	J41 pin 3 (SDA)
GND Pin	J41 Pin 25 (GND)
Vin Pin	J41 Pin 17 (3.3V)

Table 3.3: Wiring table.

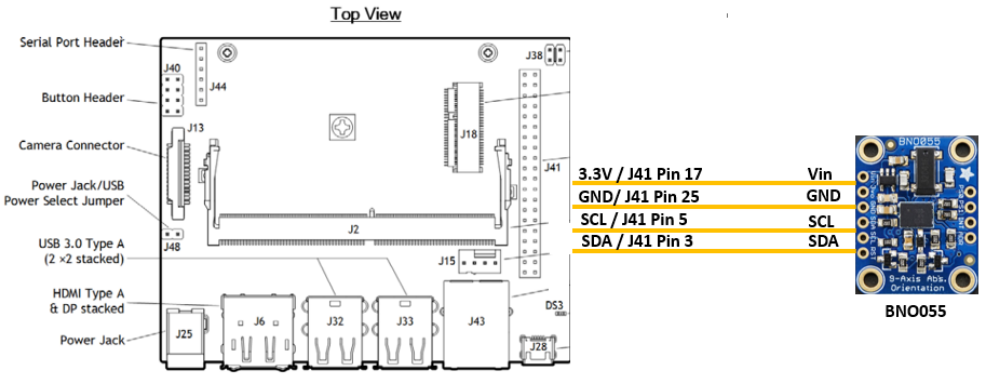


Figure 3.10: Jetson Nano and BNO055 I2C connection

BNO055 and Jetson Nano Connection via UART

The Jetson Nano contains two UART connections. One used for the serial debug console on the J44 header which used for start up issues. The second UART is on the J41 header which is a general purpose UART.

Wiring between the BNO055 and the Jetson Nano shown in table 3.4 and Fig. 3.11, to use BNO055 in UART mode, BNO055 PS1 Pin should be connected to 3.3V Pin.

UART connection	
BNO055 Pin's	**Jetson Nano Pin's**
SCL pin (Rx)	J41 pin 8(Tx)
SDA pin (Tx)	J41 pin 10(Rx)
GND Pin	J41 GND
Vin Pin	J41 3.3V

Table 3.4: Wiring table.

Also the *nvgetty* service in the Jetson Nano is disabled to establish UART connection since the serial console is not used, by executing following commands.

```
$ systemctl stop nvgetty
$ systemctl disable nvgetty
$ udevadm trigger
```

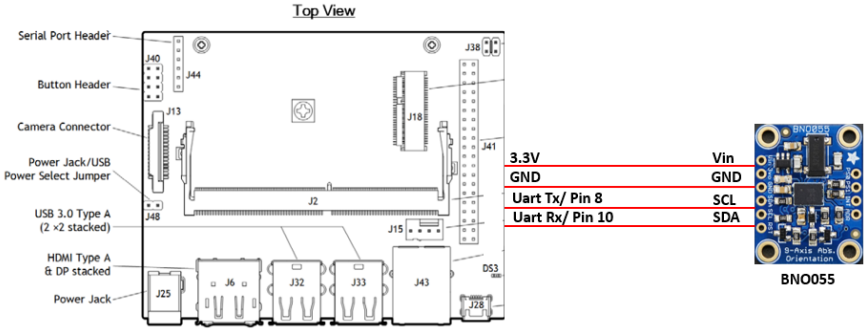


Figure 3.11: Jetson Nano and BNO055 UART connection

Laser Range Finder

LiDARs provide measurements of the robot environment by using time of flight method. Uiabot equipped with RpLiDAR A1 type, which is a rotating, low cost lidar that gives a 360 degree scan field. Table 3.5 shows it's primary features.

A1 RpLiDAR features	
Parameter	**Value**
Range	12 [m]
Rotating angle	360 degree omnidirectional
Number of samples	8000
Frequency	5.5[Hz]

Table 3.5: RpLiDAR primary features.

The RpLiDAR connected to the Jetson nano via USB cable, and mounted on the top of the oDrive's box in order to get a better scan field, and also the laser beams doesn't get blocked by the other components of the Uiabot.

3.2.5 Cost

One of the main objectives of this project is to offer an affordable educational robotic kit. Therefore, the cost of the Uiabot is crucial in this project. Table 3.6 shows the cost of the components that is used to build the Uiabot.

Mobile platform cost in $		
Component	**Quantity**	**Unit cost**
Brushless motor	2	79
oDrive	1	179
Encoder	2	39
Gearbox	2	43
Tires	2	31
Castor wheels	2	4
Jetson Nano	1	100
RpLiDAR	1	90
IMU	1	45
Transportation and Taxes		50
3D printing of the waffle plates	20	30
12V battery	1	120
Total		1002

Table 3.6: Mobile platform cost in $.

3.3 Programming and Software Implementation

This section discusses the software aspects and the implementation of the Uiabot's *see-think-act* cycle using the selected hardware. The objective of this section is to provide a clear method and user guide for the future master students to set up and further develop the required software packages for the Uiabot.

Fig. 3.12, shows the structure of the Uiabot *see-think-act* cycle and how ROS2 packages, SLAM toolbox and Nav2 framework are used in order to achieve autonomous navigation.

As mentioned in chapter 2, ROS2 galactic is used for programming. ROS2 Galactic requires Ubuntu 20.04 which haven't been released officially by Nvidia on the Jetson Nano yet, however there are two different images released that can be installed on the Jetson Nano.

The first image is provided by Q-engineering [34] and it's a suitable Ubuntu 20.04 image that contains pre-installed software such as Tensorflow, OpenCV, ect. The second image provided by *Mr.Chrismitchells* [37] and based on Xubunut. Both images were tested, and the first image was selected due to the fact it has pre-installed software that are needed later on for computer vision purposes, and it's not based on the Xubuntu.

After installing Ubuntu 20.04 on the Jetson Nano, ROS2 galactic was installed from source by following the official ROS2 website [24]. The last step is to set up ROS2 drivers (wrappers) for Uiabot components and devices.

The next subsections show ROS2 packages implementation, and the needed ROS2 packages to exchange data via ROS2 messages for the *see-think-act* cycle implementation using Nav2 framework.

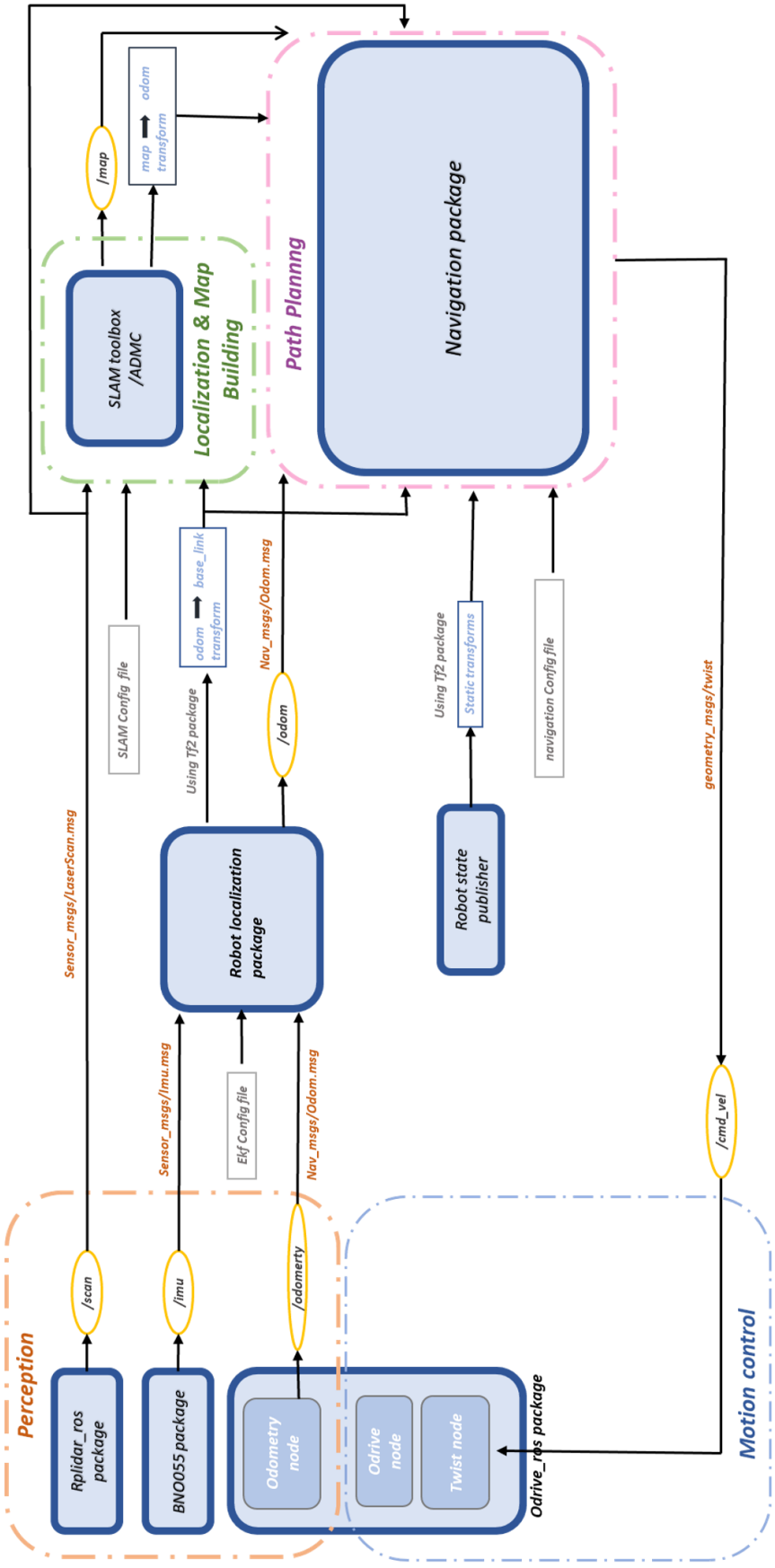


Figure 3.12: See-Think-Act cycle of the Uiabot.

3.3.1 Motion Control

As mentioned in sec 2.2.2, the controller server of Nav2 sends the Uiabot velocity control signals. These velocity signals are converted and sent to the wheels by the drive system ROS2 package.
Fig. 3.13 shows Uiabot motion control, and the structure of the drive system package.

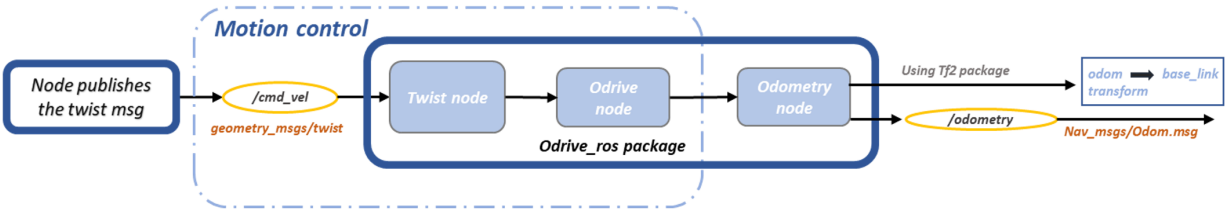


Figure 3.13: Uiabot motion control.

Drive System Package

In addition to the wheel's angular velocity control, the Uiabot drive system package calculates:

- Desired wheel velocity: by obtaining the desired velocity of the Uiabot $\dot{\xi}_d = [\dot{x}\ , \dot{y}\ , \dot{\theta}]^T$ and using the inverse differential kinematics in eq 2.20 to compute the desired velocity of each wheel $\dot{\phi}_d = [\dot{\phi}_r\ , \dot{\phi}_l]^T$.
- Odometry information: using the forward differential kinematics in eq 2.18 to compute Uiabot velocity $\dot{\xi} = [\dot{x}\ , \dot{y}\ , \dot{\theta}]^T$ from the encoders measurements of the wheels $\dot{\phi} = [\dot{\phi}_r\ , \dot{\phi}_l]^T$.

In order to accomplish that, three nodes were created:

- oDrive node.
- Odometry node.
- Velocity command node.

oDrive Node
This is the main node that initialize connection between the oDrive and Jetson Nano. This node contains the ROS2 wrapper. It responsible for publishing encoder data and calculating the desired velocities of the right and left motor.

The oDrive node provides these ROS2 services:

- connect_oDrive: in order to connect the oDrive to the Jetson Nano, a call has to be sent to this service as a trigger request. If the connection is successful the message "oDrive is connected" will be printed as a response. Otherwise oDrive timeout message will be printed.
- request_state: after connecting successfully to the oDrive, this service is used to choose one of the different modes of the motor's state, e.g. motors and encoders calibration, closed loop control, etc. Table 3.7 shows each state and the assigned value that should be written as STD request contains the number of the motors (0 for the right motor and 1 for the left motor) and the state number (from table 3.7).
- position_cmd: used to control the position of the motor's shaft by sending a request to this service.
- velocity_cmd: used to control the velocity of the motor's shaft by sending a request to this service.

Request State Service	
Axis State	**Value**
AXIS_STATE_UNDEFINED	0
AXIS_STATE_IDLE	1
AXIS_STATE_STARTUP_SEQUENCE	2
AXIS_STATE_FULL_CALIBRATION_SEQUENCE	3
AXIS_STATE_MOTOR_CALIBRATION	4
AXIS_STATE_SENSORLESS_CONTROL	5
AXIS_STATE_ENCODER_INDEX_SEARCH	6
AXIS_STATE_ENCODER_OFFSET_CALIBRATION	7
AXIS_STATE_CLOSED_LOOP_CONTROL	8
AXIS_STATE_LOCKIN_SPIN	9
AXIS_STATE_ENCODER_DIR_FIND	10

Table 3.7: Assigned values for requesting a state.

In order to drive the motors in closed loop control, three sequential steps should be performed:

1. Connect the oDrive to the Jetson Nano.
2. Set the right and left motor in full calibration sequence.
3. Set the right and left motor in closed loop control.

The oDrive node also publishes these topics:

- /right_angle: right motor shaft angle, ϕ_r in rev, that is acquired from right wheel encoder.
- /left_angle: left motor shaft angle, ϕ_l in rev, that is acquired from left wheel encoder.
- /right_omega: right motor shaft angular speed, $\dot{\phi}_r$ in rev/sec, that is acquired from right wheel encoder.
- /left_omega: left motor shaft angular speed, $\dot{\phi}_l$ in rev/sec, that is acquired from left wheel encoder.
- /joint_state: position and velocity of the wheels using the joint state message. The joint state message is used by the joint state publisher and robot state publisher packages.

And subscribes to the following topics:

- /rwheel_vtarget: to set the desired angular velocity of the right wheel, $\dot{\phi}_r$.
- /lwheel_vtarget: to set the desired angular velocity of the left wheel, $\dot{\phi}_l$.

Fig. 3.14 shows the oDrive node structure and the utilized services and topics.

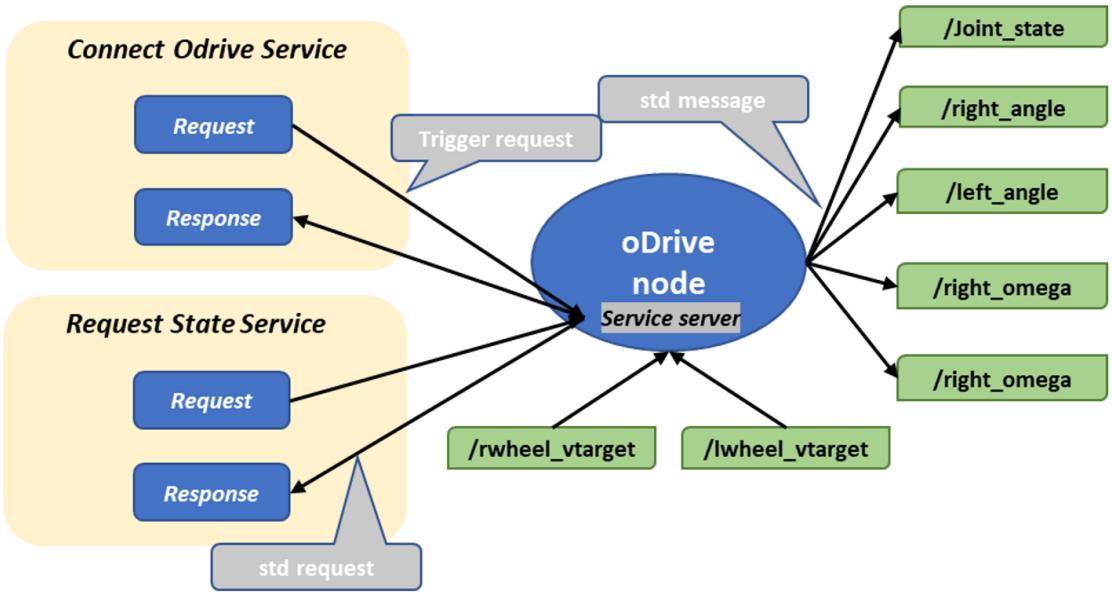


Figure 3.14: oDrive node structure.

Odometry Node

The objectives of this node is to publish the odometry information and the *odom* ⇒ *base_link* transform required by the Nav2 framework. It subscribes to the following topics, that are published by the oDrive node:

- /right_angle.
- /left_angle.
- /right_omega.
- /left_omega.

The odometry node also publishes the odometry information on the "*/odometry*" topic using odometry message. The odometry message contains the pose $\xi = [x\ , y\ , \theta]^T$ and velocity $\dot{\xi} = [\dot{x}\ , \dot{y}\ , \dot{\theta}]^T$ of the Uiabot. Whereas the information that the odometry node subscribed to is the individual wheel position $\phi = [\phi_r\ , \phi_l]^T$ and velocity $\dot{\phi} = [\dot{\phi}_r\ , \dot{\phi}_l]$ which obtained by the encoders. Therefore, the differential drive forward kinematic is required and implemented in the odometry node as shown in eq 3.1 and 3.2. Table 3.8, shows Uiabot parameters.

$$\dot{x} = \frac{r}{2 \times b} \times (\dot{\phi}_r - \dot{\phi}_l) \times \frac{2\pi}{i} \tag{3.1}$$

$$\dot{\theta} = 0.5 \times r \times (\dot{\phi}_r + \dot{\phi}_l) \times \frac{2\pi}{i} \tag{3.2}$$

$\dot{x}$ and $\dot{\theta}$ are the transnational and angular velocities of the Uiabot respectively.
$\dot{y} = 0$, due to the no-sliding constraint, as shown in Fig. 3.15.

Uiabot parameters		
Symbol	**Description**	**Value**
r	Wheel's radius	0.05 [m]
b	Half of the distance between the wheels	0.032[m]
i	Gearbox ratio	20

Table 3.8: Uiabot parameters.

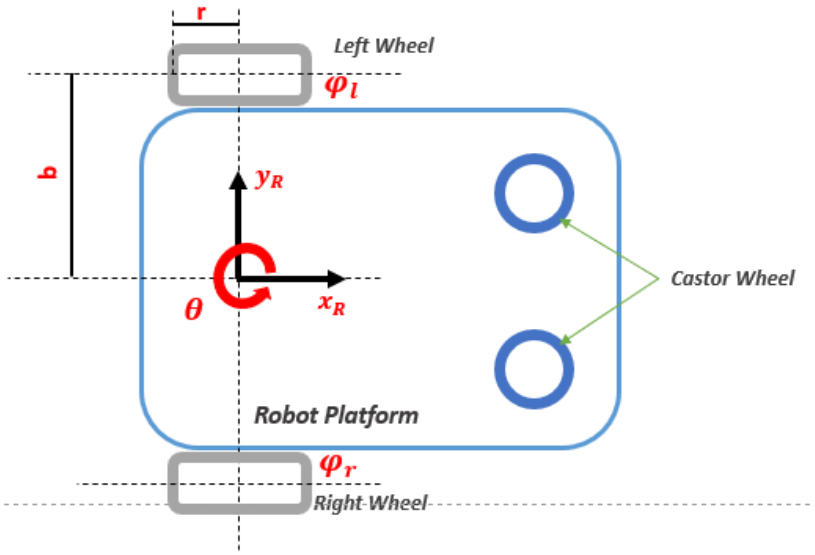


Figure 3.15: Differential drive configuration scheme

Messages in ROS2 adhere to the ROS-103 standard. The oDrive sends and receives data in rev and rev/sec, so a unit conversion of 2π is added to convert from [rev] into [rad] as shown in eq 3.1 and eq3.2. It is also divided by the gearbox ratio, since the encoders measure the position of the motor shafts, not the gearbox shafts.

The pose of the Uiabot $\xi = [x\ , y\ , \theta]^T$, computed as shown in eq. 3.3, 3.4 and 3.5.

$$\theta = \dot{\theta} \times \Delta t \tag{3.3}$$

$$x = \dot{x} \times \cos\theta \times \Delta t \tag{3.4}$$

$$y = \dot{x} \times \sin\theta \times \Delta t \tag{3.5}$$

All the required information for the odometry message is computed and published on the */odometry* topic.

In addition to the odometry message, the *odom* $\Rightarrow$ *base_link* transform is published, as shown in the code snippet, listing. 3.1. The quaternion angles calculated from the Uiabot orientation angle θ, then the transform stamped message published using TF2 package.

Listing 3.1: *odom*$\Rightarrow$ *base_link TF*

```
# publish the odom  base link TF
quaternion = Quaternion()
quaternion.x = 0.0
quaternion.y = 0.0
quaternion.z = sin(self.th / 2)
quaternion.w = cos(self.th / 2)

transform_stamped_msg = TransformStamped()
transform_stamped_msg.header.stamp = ...
   self.get_clock().now().to_msg()
transform_stamped_msg.header.frame_id = self.odom_frame_id
transform_stamped_msg.child_frame_id = self.base_frame_id
transform_stamped_msg.transform.translation.x = self.x
transform_stamped_msg.transform.translation.y = self.y
transform_stamped_msg.transform.translation.z = 0.0
transform_stamped_msg.transform.rotation.x = quaternion.x
transform_stamped_msg.transform.rotation.y = quaternion.y
transform_stamped_msg.transform.rotation.z = quaternion.z
transform_stamped_msg.transform.rotation.w = quaternion.w

self.odom_broadcaster.sendTransform(transform_stamped_msg)
```

Fig. 3.16 shows the node structure and which topics it connected with.

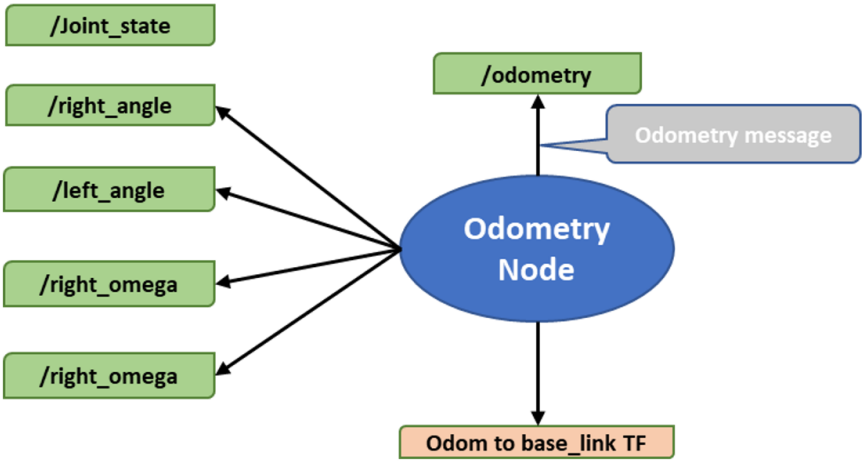


Figure 3.16: Odom node.

Twist Node

This node is responsible for converting the Uiabot desired velocity commands $\dot{\xi}_d = [\dot{x}\ , \dot{y}\ , \dot{\theta}]^T$ into individual wheel velocity commands $\dot{\phi}_d = [\dot{\phi}_r\ , \dot{\phi}_l]^T$, using the inverse kinematics of the differential drive.

In order to perform the velocity conversion, the twist node subscribes to "*/cmd_vel*" topic that holds the twist message. The twist message is published either using a keyboard teleportation ROS2 package to move the Uiabot around, or by any other package such as the navigation packages. It contains the desired linear and angular velocities of the Uiabot $\dot{\xi}_d = [\dot{x}\ , \dot{y}\ , \dot{\theta}]^T$.

Eq. 3.6 and 3.7 is used to obtain the desired velocity of each individual wheel. The packages that publishes "*/cmd*" topic adhere ROS 105 standard. Therefore, it is required to convert from SI-unit to the unit system that oDrive use. In addition, the gearbox ratio have to be included. The equations of the desired velocity of each individual wheel are:

$$\dot{\phi}_r = \frac{\dot{x} + (\dot{\theta} \times b)}{r} \times \frac{i}{2\pi} \tag{3.6}$$

$$\dot{\phi}_l = \frac{\dot{x} - (\dot{\theta} \times b)}{r} \times \frac{i}{2\pi} \tag{3.7}$$

The twist node publishes $\dot{\phi}_d = [\dot{\phi}_r\ , \dot{\phi}_l]^T$ on the following topics,

- /rwheel_vtarget.
- /lwheel_vtarget.

Which are the same topics that oDrive node subscribe to. Fig. 3.17 shows the twist node structure.

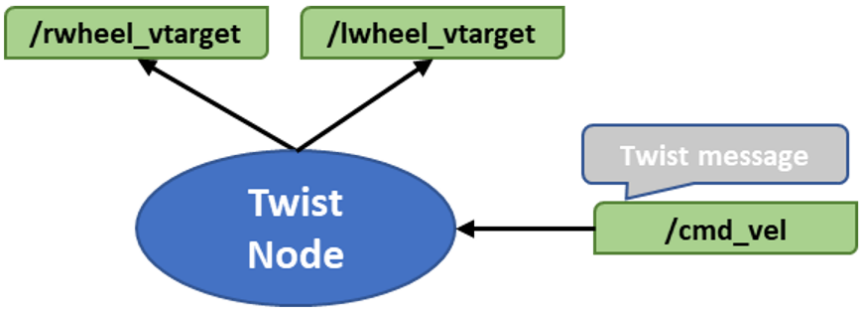


Figure 3.17: Twist node structure.

The oDrive odometry and twist nodes are created in ROS2 package named "oDrive_ROS2". Fig. 3.18 shows structure of the oDrive_ROS2 package and how these nodes are linked together.

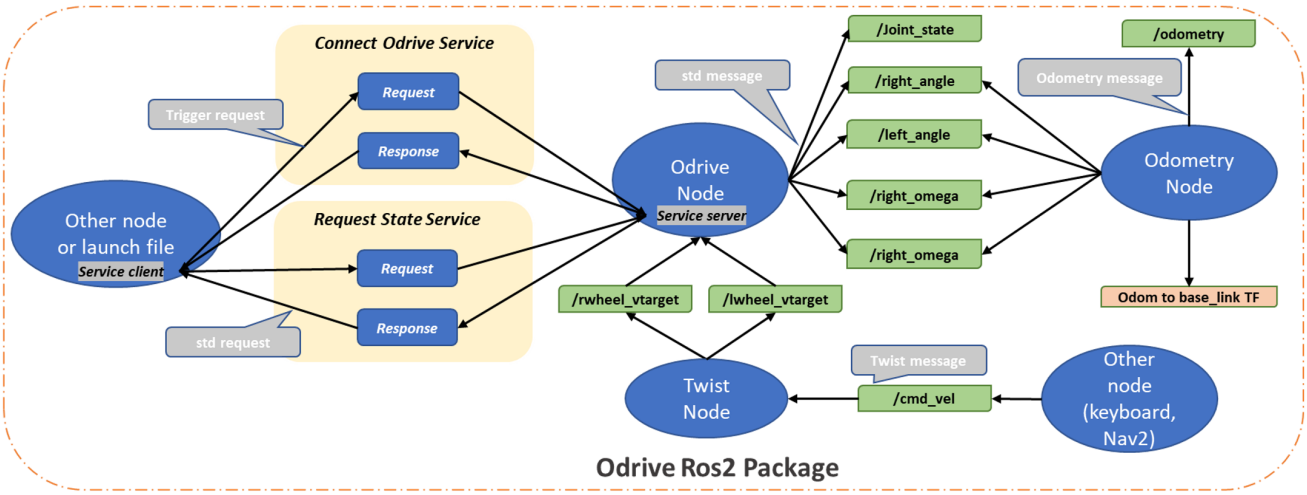


Figure 3.18: oDrive ROS2 package structure

3.3.2 Perception

The selected sensors is utilized to obtain data from the Uiabot e.g. heading and orientation, and data from the surroundings e.g. distance between objects and the Uiabot. *RpLiDAR_ROS* and *bno055_ROS* ROS2 packages were created in order to publish the laser scan and IMU messages. Furthermore, a sensor fusion package was used to fuse the data from the wheel odometry message and the IMU message to obtain better estimation of the robot odometry.

Fig. 3.19 shows the Uiabot perception structure. Notice that odometry node is also included in the perception structure of the Uiabot.

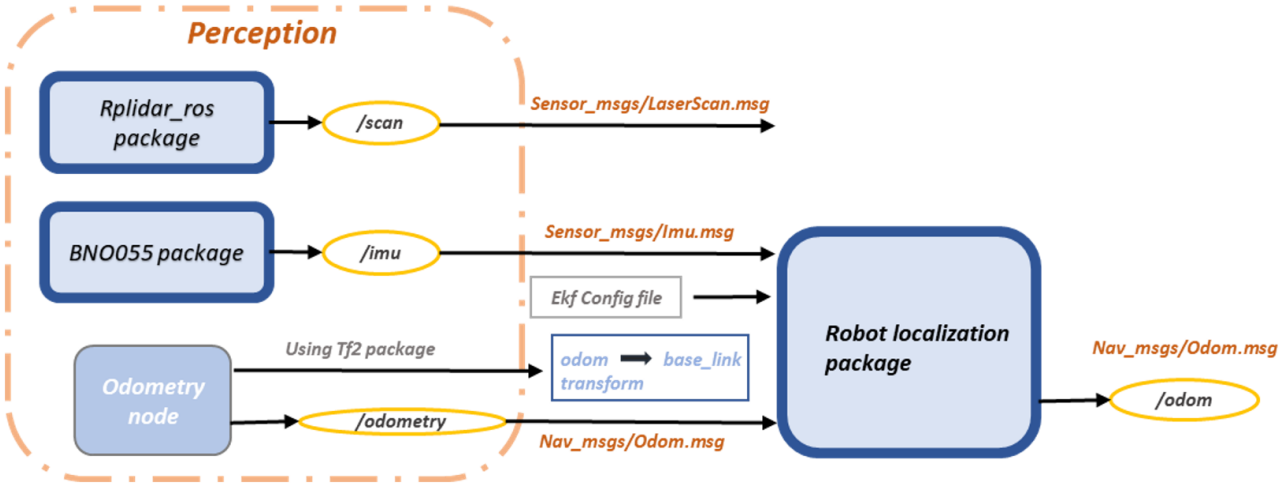


Figure 3.19: Uiabot perception

Laser Scan Package

Slamtech provides ROS2 RpLiDAR packages called "RpLiDAR_ROS" [25]. This package contains the ROS2 wrapper and a node that publishes laser scan data on the "*/scan*" topic using the laser scan message. The parameters of the RpLiDAR package is configured and changed to fit the project requirements, as seen in table 3.9.

RpLiDAR ROS2 package		
Parameter	**Value**	**Description**
frame_id	*lidar_link*	All the data reading is with respect to this frame.
baud rate	11580	Rate of communication in bps for RpLiDAR type A1.
serial port	USB0	USB port number that is connected with Jetson Nano.

Table 3.9: RpLiDAR package parameters

The laser scan data is published using the Laser scan message over */scan* topic as required by Nav2 framework. The frame_id parameter have the same name as the static coordinate frame of the *laser_link* in the URDF file in sec 3.3.2. The baud rate is set to 11580 as mentioned in Slamtec ROS2 [25] for the type A1 RpLiDAR's.

Connection between the RpLiDAR and the Jetson Nano requires port permission which can be provided by executing the following command in the Jetson Nano terminal:

```
$ ls -l /dev | grep ttyUSB # list USB0 ports.
$ sudo chmod 666 ttyUSB0  #permission for USB0 port
```

Fig. 3.20 shows the structure of the RpLiDAR package.

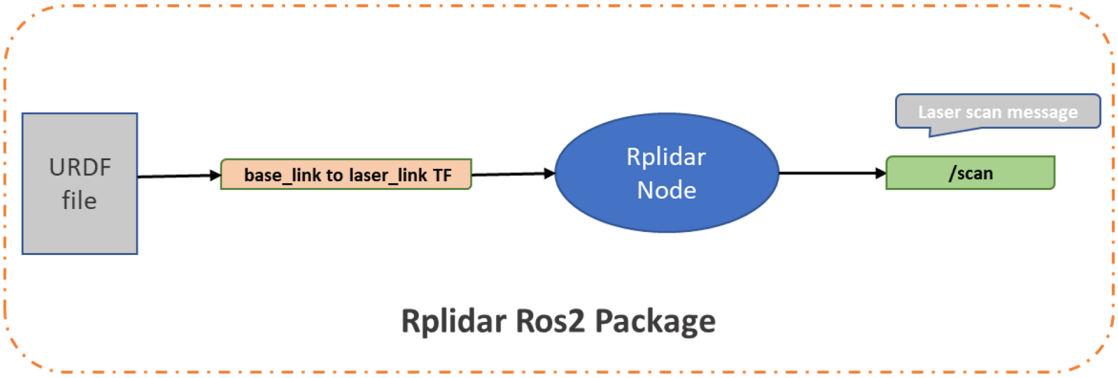


Figure 3.20: RpLiDAR ROS2 package structure

BNO055 Packages

As mentioned earlier, communication between BNO055 and Uiabot Jetson Nano can be established in ROS2 using UART and I2C preotocols. And each protocol has it's own ROS2 package that can is used to publish BNO055 data.

[4] is a BNO055 ROS2 package that uses UART serial communication. This package publishes the following different topics:

- bno055/imu: publishes accelerometers and gyROScopes fused data using IMU message.
- bno055/imu_raw: publishes accelerometers and gyROScopes raw data using IMU message.
- bno055/temp: publishes the ambient temperature using temperature message.
- bno055/mag: publishes magnetometer reading using magnetic field message.
- bno055/calib_status: publishes calibration status of the magnetometer, accelerometers and gyROScopes as JSON string using STD string message.

It also contains a variety of parameters that can be configured to fit the user requirements. For Uiabot the same parameters are used except uart port is configured as "/dev/ttyTHS1".

Fig. 3.21, shows the package structure and the published topics.

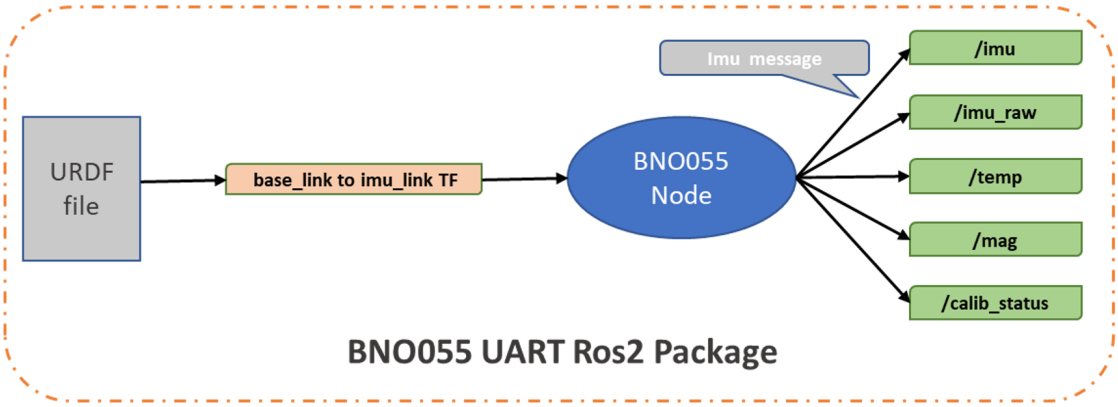


Figure 3.21: BNO055 ROS2 package structure using UART connection

Whereas, [3] package uses I2C serial communication to transfer data, and it publish the following topics :

- /imu/raw: publishes accelerometers and gyROScopes raw data using IMU message.
- /imu/data: publishes accelerometers and gyROScopes fused data using IMU message, required the BNO055 to be in NDOF mode, and this was achieved by rotating the BNO055 sensor around, as seen here [2], in order to calibrate it.
- /mag: publishes magnetometer reading using magnetic field message.
- /temp: publishes the ambient temperature using temperature message.

Furthermore, the package parameters was configured as shown in table 3.10 in order to communicate with Jetson Nano.

BNO055 ROS2 package		
Parameter	**Description**	**Value**
Device port	path to the i2c device	/dev/i2c-1
Address	i2c address of the BNO055	0x28

Table 3.10: BNO055 package parameters

Fig 3.22 shows BNO055 package structure and the published topics.

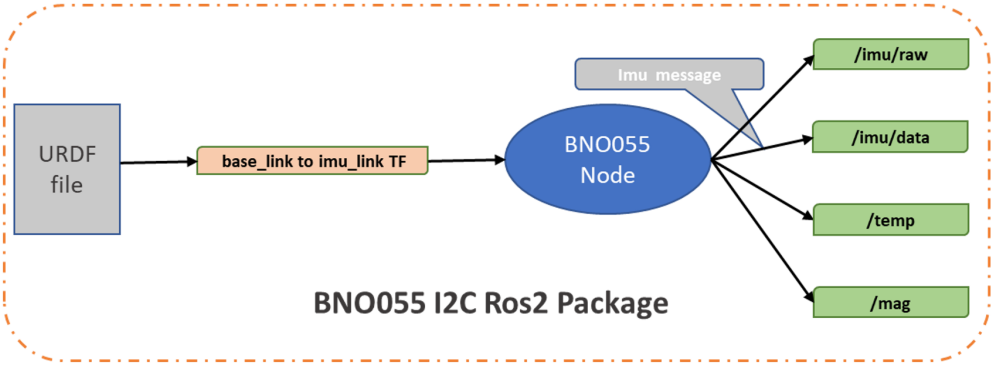


Figure 3.22: BNO055 ROS2 package structure using I2C connection

BNO055 UART package was used in this project since it built using C++ (faster). However, the BNO055 I2C package is also built in the Jetson Nano as an alternative that can be used in case UART communication is needed for another device or application.

Sensor Fusion

The robot localization package [22] was installed from source which is required for ROS2 galactic and built in the Uiabot work space. It subscribes to the */odometry* topic that is published by the odometry node, and to the */imu* topic that is published by the BNO055 node. In return, it uses the Extended Kalman Filter to publish the fused odometry information with topic name */odom* that adheres ROS-105 standard.

The *odom* ⇒ *base_ link* transform can be either published by this package or by the odometry node. Since the odometry node already publishes this transform, the Publish_tf option is set to false in the configuration file of the robot localization package as seen in snippet 3.2.

Listing 3.2: ekf configuration file

```
ekf_filter_node:
    ROS__parameters:
        frequency: 30.0
        two_d_mode: false
        publish_acceleration: true
        publish_tf: false
        map_frame: map              # Defaults to "map" if unspecified
        odom_frame: odom            # Defaults to "odom" if unspecified
        base_link_frame: base_link  # Defaults to "base_link" if unspecified
        world_frame: odom
        odom0: odometry
        odom0_config: [true,  true,  true,
                       false, false, false,
                       false, false, false,
                       false, false, true,
                       false, false, false]
        imu0: bno055/imu
        imu0_config: [false, false, false,
                      true,  true,  true,
                      false, false, false,
                      false, false, false,
                      false, false, false]
```

Uiabot URDF File

URDF file is created for Uiabot in order to publish the various static transforms between the links of the Uiabot e.g. *base_link* ⇒ *lidar_link*, the joints state of Uiabot joints e.g. right wheel joint and left wheel joint, and to visualize Uiabot in Rviz2 and Gazebo simulator.

The URDF file contains all the parameters of the geometry of Uiabot links (chassis, wheels, castor wheel, RpLiDAR, and IMU).

Uiabot consists of the following links:

- *base_link*: main chassis of the Uiabot and holds the *base_link* coordinate frame.
- *drivewhl_r_link*: right wheel and holds the *drivewhl_r_link* coordinate frame.
- *drivewhl_l_link*: left wheel of Uiabot and holds the *drivewhl_l_link* coordinate frame.
- *front_castor*: front caster wheel of the Uiabot and holds the *caster_wheel* coordinate frame.
- *imu_link*: BNO055 sensor link and holds the *imu_link* coordinate frame.
- *lidar_link*: RpLiDAR sensor link and holds the *lidar_link* coordinate frame.
- *base_footprint*: a virtual link (non-physical link). The *footprint_link* is needed in Nav2 framework, to compute the center of the robot projected on the ground. It also holds the footprint coordinate frame.

The URDF file also consists of the following continuous (revolute) joints:

- *rwheel_joint*: continuous joint between the *base_link* and the *rwheel_link*.
- *lwheel_joint*: continuous joint between the *base_link* and the *lwheel_link*.

The other links are attached to the *base_link* by fixed joints. The snippet 3.3 of the URDF file show how the castor wheel is implemented.

Listing 3.3: Caster Wheel in the URDF

```
<!-- Caster Wheel -->
 <link name="front_caster">
   <visual>
     <geometry>
       <sphere radius="${(wheel_radius+wheel_zoff-(base_height/2))}"/>
     </geometry>
     <material name="Cyan">
       <color rgba="0 1.0 1.0 1.0"/>
     </material>
   </visual>
 </link>

 <joint name="caster_joint" type="fixed">
   <parent link="base_link"/>
   <child link="front_caster"/>
   <origin xyz="${caster_xoff} 0.0 ${-(base_height/2)}" rpy="0 0 0"/>

 </joint>
```

The URDF file is executed by the robot state publisher package. Fig 3.23, shows in Rviz2 the TF tree of the static transforms included in the URDF file, while table 3.11 shows the distance in (x, y ,z) between the *base_ link* and *lidar_ link*, *imu_ link*, *drivewhl_ r_ link*, *drivewhl_ l_ link*, *front_ castor* and *base_ footprint* frames.

Distance to the *base_ link* frame in cm			
Frame	**x**	**y**	**z**
lidar_ link	10	0.0	13
imu_ link	-9.5	-9.5	4.75
rwheel_ link	-10	18.05	0.0
lwheel_ link	-10	- 18.05	0.0
front_ castor	10	0	
footprint_ link	-10	18.05	0.0

Table 3.11: Static transform in Uiabot

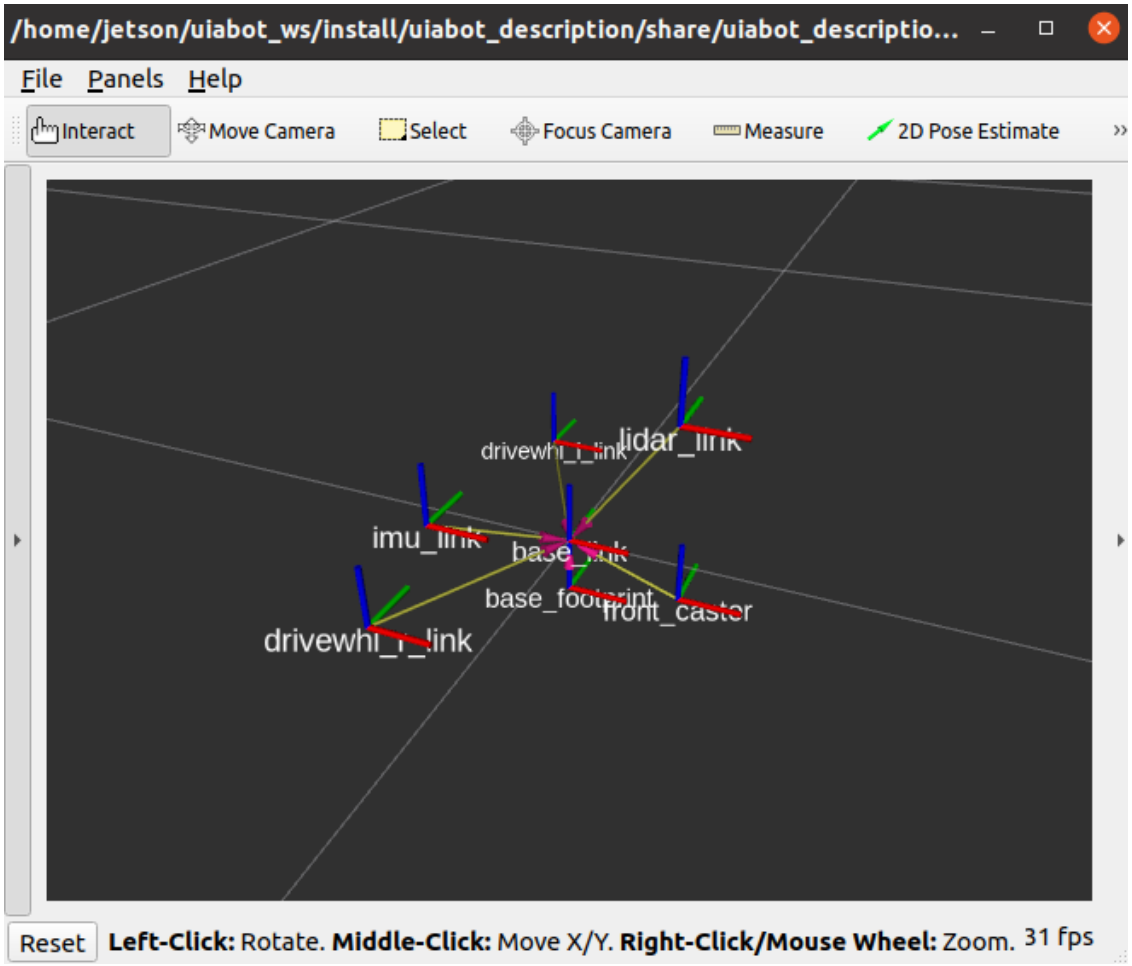


Figure 3.23: Uiabot static TF tree in Rviz2

Uiabot Bring Up Launch File

Different sensors need different configurations and set ups, and running these nodes one by one can be tedious. The Uiabot *bring up* launch file is implemented in order to configure and run Uiabot nodes when motion control and perception is required, the *bring up* launch file is divided into four main steps.

Step 1. Import the required Uiabot files and folders e.g. packages path and URDF file path.

Step 2. Launch arguments and node configurations to configure the sensors, and to set the launch arguments that define how to launch the Uiabot, as seen in snippet 3.4. This step allows the user to set different variables and options from the Jetson Nano terminal.

Listing 3.4: Launch arguments config

```
    return launch.LaunchDescription([

        launch.actions.DeclareLaunchArgument(name='rvizconfig', ...
            default_value=default_rviz_config_path,
                                             description='Absolute path to ...
                                                rviz config file'),
        launch.actions.DeclareLaunchArgument(name='serial_port', ...
            default_value=serial_port,
                                             description='Specifying usb ...
                                                port to connected lidar'),
        launch.actions.DeclareLaunchArgument(name='serial_baudrate', ...
            default_value=serial_baudrate,
                                             description='Specifying usb ...
                                                port baudrate to connected ...
                                                lidar'),
        launch.actions.DeclareLaunchArgument(name='frame_id', ...
            default_value=frame_id,
                                             description='Specifying ...
                                                frame_id of lidar'),
```

Step 3. Running the nodes that will be used for Uiabot motion control and perception. The following nodes will be launched using *bring up* launch file:

1. oDrive node.
2. Odometry node.
3. Twist node.
4. Robot state publisher node.
5. BNO055 node.
6. Robot localization node.
7. RpLiDAR node.
8. Rviz node.

Step 4. oDrive services request, to send different requests to the oDrive services.

The following requests is performed by *bring up* launch file to engage and launch the drive system:

- Trigger request: sent to the "connect_oDrive" server in oDrive node to connect the oDrive.
- STD request: contains the axis (motor) number 0 and requested state 3. This request is sent to the "axis_state" server in the oDrive node to put the right motor in full calibration sequence.
- STD request: contains the axis (motor) number 1 and requested state 3. This request is sent to the "axis_state" server in the oDrive node to put the left motor in full calibration sequence.
- STD request: contains the axis (motor) number 0 and requested state 8. This request is sent to the "axis_state" server in the oDrive node to put the right motor in closed loop control.
- STD request: contains the axis (motor) number 1 and requested state 8. This request is sent to the "axis_state" server in the oDrive node to put the left motor in closed loop control.

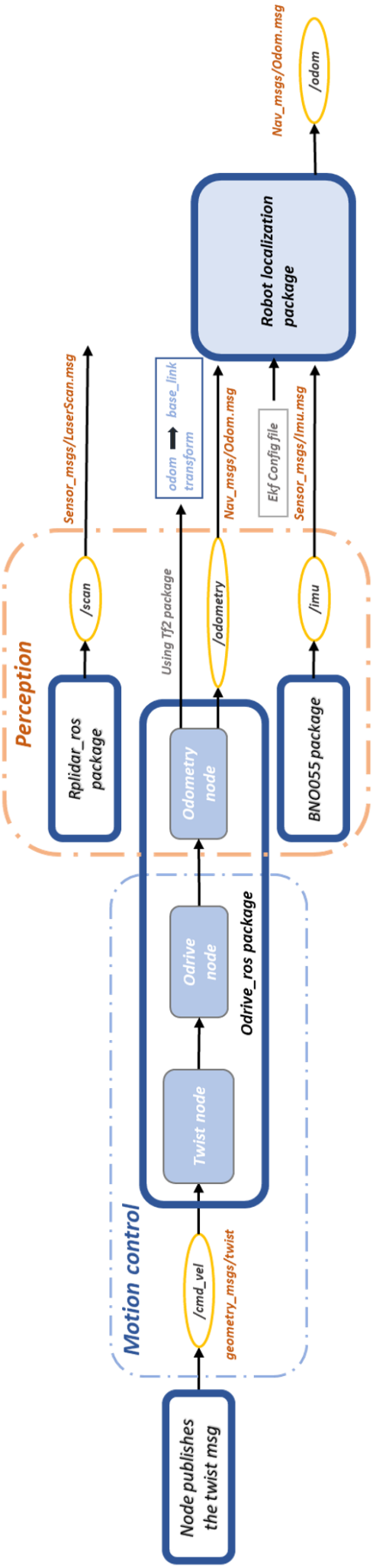
These requests have to be sent sequentially in order to control Uiabot motors properly. A waiting time is set between the requests using *Event Handler* in the launch file, as seen in the snippet 3.5.

Listing 3.5: Launch arguments config

```
RegisterEventHandler(
    OnProcessStart(
        target_action= od_node,
        on_start=[
        LogInfo(msg='Looking for oDrive device ... '),
        oDrive_connection,  # Connect the oDrive
        TimerAction(
            period= 10.0,
            actions=[
            LogInfo(msg=' Starting axis sequence '),
            axis0_state3, axis1_state3], #put right and left motor in ...
                calibration sequential mode

                ),
        TimerAction(
            period= 30.0,
            actions=[
            LogInfo(msg=' Starting axis closed loop control '),
            axis0_state8, axis1_state8], #put right and left motor in ...
                closed loop control mode
```

Launching the *bring up* file initializes the Uiabot motion control and perception packages. Uiabot can now be driven around by a velocity command, and publishes various sensor data e.g. wheel encoders, IMU and RpLiDAR. Also, odometry and joint state information are available. Furthermore, *odom* $\Rightarrow$ *base_link*, *base_link* $\Rightarrow$ *footprint*, *base_link* $\Rightarrow$ *lidar_link* and *base_link* $\Rightarrow$ *imu_link* are published. Figure. 3.24, shows motion control and perception packages and structure in the Uiabot.


Nav_msgs/Odom.msg
/odom
Robot localization package
Sensor_msgs/LaserScan.msg
odom → base_link transform
Nav_msgs/Odom.msg
Ekf Config file
Sensor_msgs/Imu.msg
Perception
/scan
Using Tf2 package
/odometry
/imu
Rplidar_ros package
Odometry node
BNO055 package
Odrive_ros package
Odrive node
Twist node
Motion control
/cmd_vel
geometry_msgs/twist
Node publishes the twist msg

3.3.3 Localization and Mapping

This section present how localization was implemented in Uiabot, and what tools and data Uiabot needs in order to build a map.

As mentioned in sec 2.2.1, SLAM toolbox is a ROS2 package that provides a solution for the localization and mapping problem. This package is installed from APT [28] on the Jetson Nano. SLAM toolbox utilizes the data obtained from the laser scan sensor in order to perceive Uiabot's environment and build the map. To do so, SLAM toolbox must subscribe to the same topic that publishes the laser scan message, which is published by the *RpLiDAR_ROS* package on the */scan* topic.
In addition to the perception information, it requires the *odom* ⇒ *base_link* transform to be published, which is already achieved by the odometry node in the drive system package.

As mentioned in chapter 2, the Uiabot should take an action-step in addition to the perception-step to localize itself. Therefore, a teleportation package is installed and used to give command to the Uiabot using the keyboard. This package publishes the twist message over the */cmd_vel* topic, which is the same topic that the twist node subscribes to in the drive system package.

Lastly, the SLAM toolbox requires a configuration file in *yaml* format to be uploaded. This file contains the slam configurations, the primary parameters are shown in the code snippet and the rest kept to default:

```
# ROS Parameters
odom_frame: odom
map_frame: map
base_frame: base_link
scan_topic: /scan
mode: mapping #localization
```

SLAM Launch File

A launch file was created to start *keyboard-teleportation* node, SLAM toolbox, and to load the SLAM configuration file. Launching these nodes will start the map building process, publish the "*/map*" topic, and publish the *map* ⇒ *odom* transform that is needed by Nav2. Fig. 3.25, shows how the localization and mapping in the *see-think-act* cycle is implemented in Uiabot using ROS2 packages.

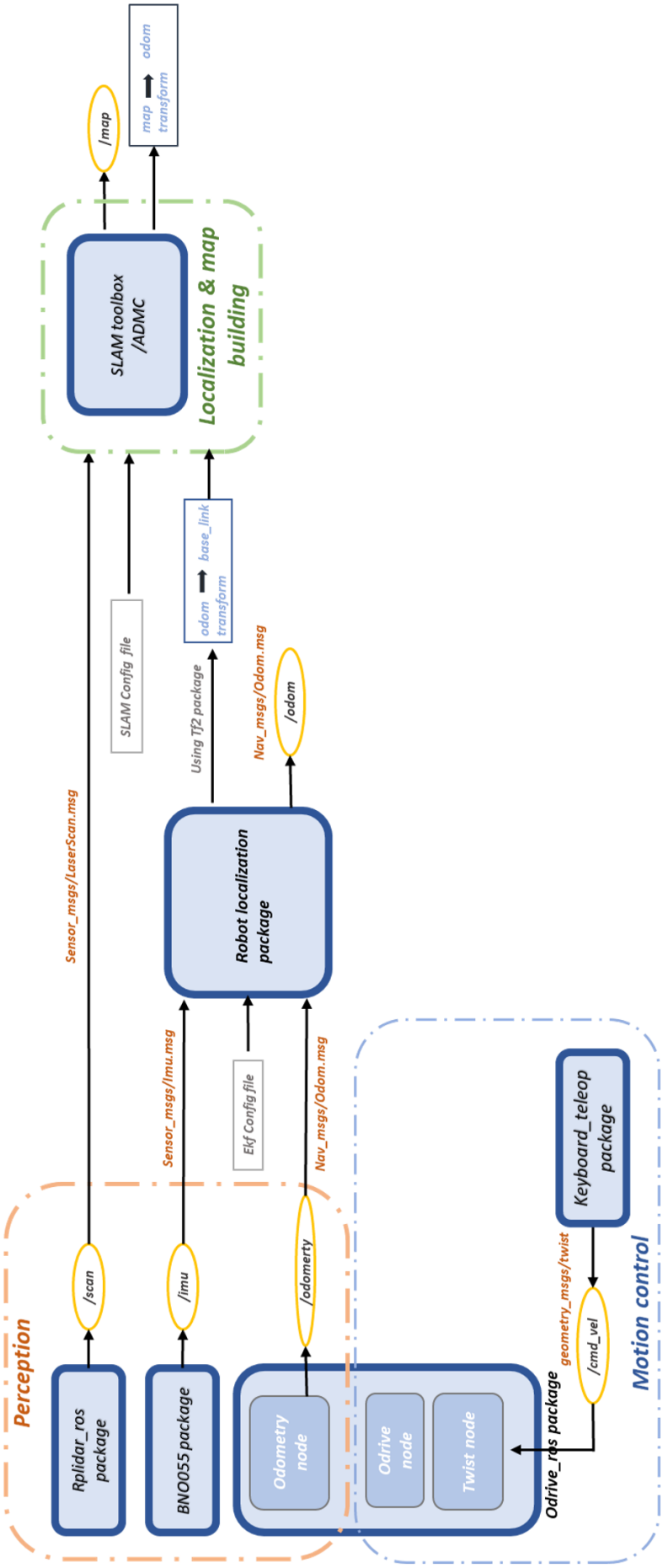


Figure 3.25: Uiabot motion control, perception and localization in SLAM task.

3.3.4 Path Planning

In order for Uiabot to navigate autonomously it has to interpret the obtained information and plan a feasible path to the goal point. Nav2 framework is utilized in the Uiabot to perform the path planning process. Nav2 requires different information and configuration in order to work properly, as discussed in chapter 2.

Launching the Uiabot *bring up* launch file publishes all the sensor data needed by Nav2, and all the required frames transforms except the $map \Rightarrow odom$ transform, that can be published either by static map or using the SLAM toolbox. Therefore, there are two methods to navigate:

1. Navigation without SLAM: in this method the static map is provided as *yaml* file, which was achieved by using the SLAM toolbox to build the map. The Adaptive Monte-Carlo Localizer (AMCL) of Nav2 is used to localize the Uiabot in a static map.
2. Navigation with SLAM: the mobile robot navigates and performs SLAM simultaneously.

In both methods, the Nav2 framework publishes the twist message on the */cmd_vel* topic to move the Uiabot along the planned global path and to avoid any detected obstacles.

Uiabot Navigation Launch File

Nav2 servers and the costmap are required to be configured. Uiabot navigation launch file was created to load the configuration file of the navigation package and to provide the ability to select the navigation method by enabling/dis-enabling the SLAM argument from the Jetson Nano terminal.

Most of the configuration parameters of Nav2 were kept to the default values, since topics and messages adhere the ROS-103 and 105 standards. However, the parameters that rely on the robot dimensions and the size of the map, such as inflation radius and footprint dimensions, was adjusted to fit the Uiabot.

3.4 Testing

Validation tests were performed to verify that the Uiabot can execute different assessments, and to demonstrate Uiabot abilities.

3.4.1 Simulation

Gazebo simulation was used to create a model of the Uiabot and to create a virtual environment (work-space) where the Uiabot can navigate and perform all the tasks were performed by the real robot.

The simulation package was built in three steps:

1. Creating Uiabot URDF file.
2. Odometry and sensor setup.
3. SLAM toolbox and Nav2 setup.

Creating Uiabot URDF File

The same URDF file for the real Uiabot was utilized to build the in Rviz2 and Gazebo environment. However, collision parameters were added to the robot links, e.g. inertial and weight, in order to resemble the motion and dynamic response of the real robot when it moves or collide with objects in Gazebo.

Odometry Data

The odometry information was determined by Gazebo using differential drive plugin [6], IMU plugin [11], and Joint state publisher plugin [12].

Utilizing the differential drive plugin instead of the wheel encoder plugin is convenient. Since it's built for differential drive robot such as the Uiabot, it doesn't require to calculate neither the odometry information nor the wheel velocity command for the wheels using the differential kinematics.

The differential drive plugin publishes the twist message over the */cmd_vel* topic, and subscribe to the */odometry* topic in order to obtain the wheel odometry message. The IMU and joint state plugins publish the IMU data, and the angles of the drive wheels respectively. The robot localization package was also used to fuse the information from the differential drive and IMU plugin. Furthermore, the laser scan data was obtained by using Gazebo sensor plugin to model the laser scan data in the virtual environment.

SLAM Toolbox and Nav2 Setup

The same configuration files for SLAM toolbox and Nav2 of the real robot were used for the simulation in Gazebo, however the simulation parameters were enabled to launch the Gazebo simulator.

Fig.3.26 and Fig. 3.27, shows The Uiabot in Rviz and in virtual environment that was used in Gazebo simulation.

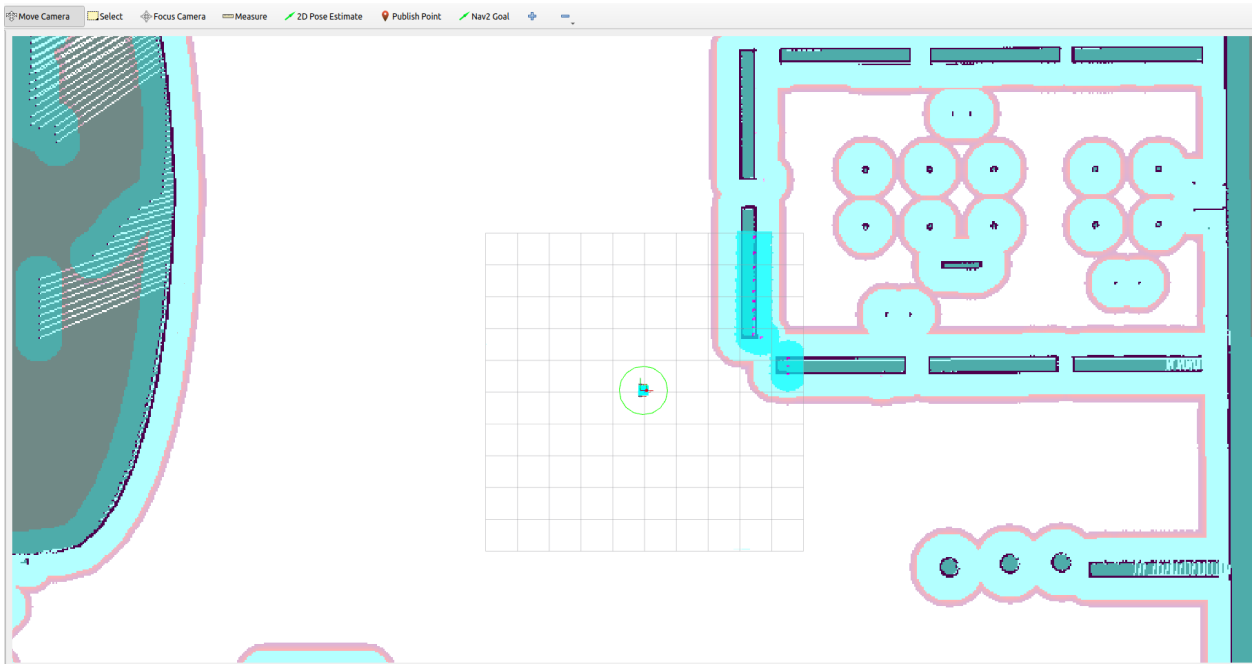


Figure 3.26: Uiabot in Rviz.

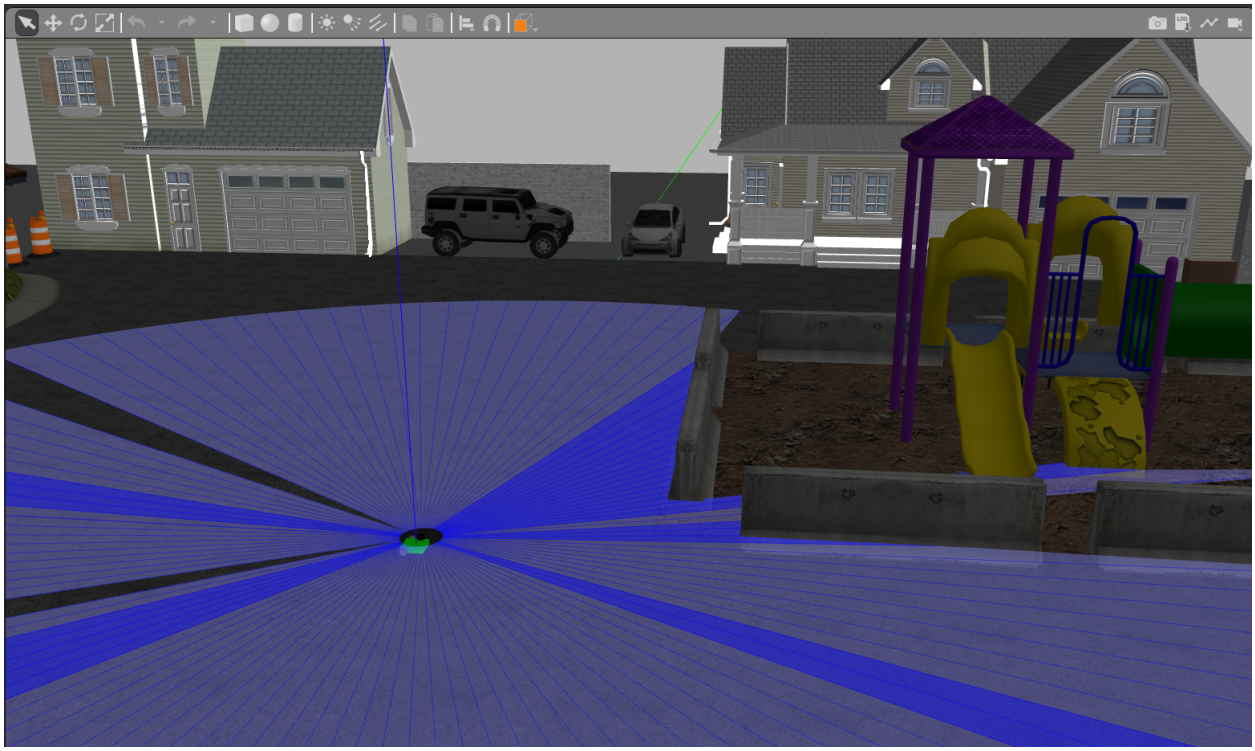


Figure 3.27: Uiabot in Gazebo.

3.4.2 Experimental Test Scene

The tests were conducted in the mechatronics department at the university of Agder. The aisle of the mechatronics department about 60 meters long and 1.5 meter wide, and an image of the area is shown in Fig. 3.28.

(a) Aisle section 1.

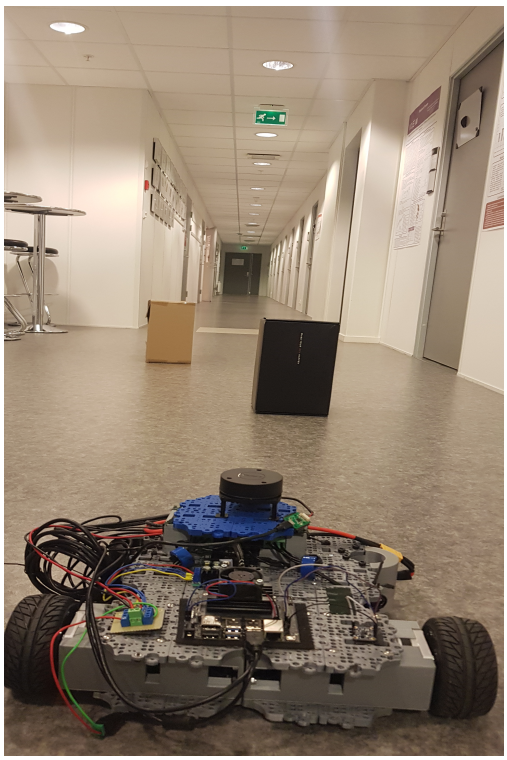
(b) Aisle section 2.

Figure 3.28: Mechatronics department aisle.

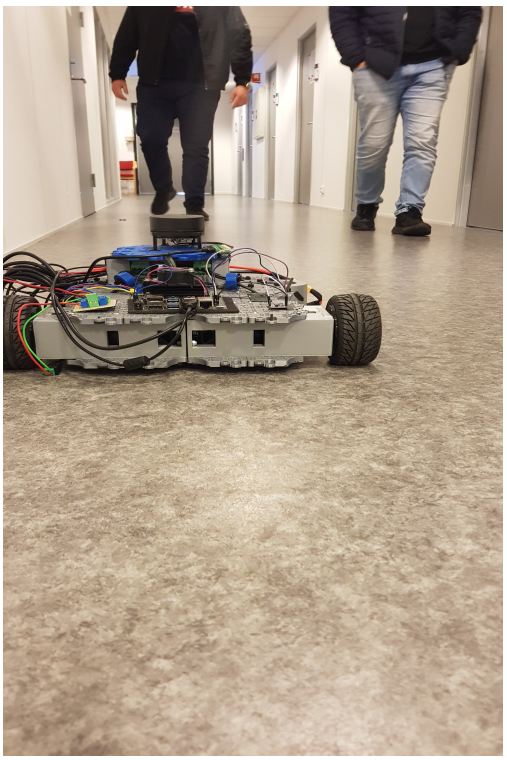
Four tests where performed:

1. Initial: test the mobile platform prototype, and the ROS2 packages of the hardware components.
2. SLAM: localize the Uiabot mobile platform in the environment and build 2D map of the department.
3. Autonomous navigation: navigate in a pre-loaded map using the AMCL.
4. Autonomous navigation with SLAM: build a map of the department and navigate to a goal using SLAM toolbox localization.

During the autonomous navigation test, two static obstacles and pedestrians (dynamic obstacles) were placed in the path of the robot, as seen in Fig. 3.29, to demonstrate the behavior and response in a dynamic environment.



(a) Static obstacles.



(b) dynamic obstacles (pedestrians).

Figure 3.29: Static and dynamic obstacles.

Chapter 4

Results and Discussion

4.1 Uiabot Prototype

4.1.1 Uiabot vs Turtlebot

One of the main objectives of this project is to develop a low-cost educational robotic kit by building a mobile platform for the available Uarm. Table 4.1, is a comparison between the Uiabot and the Turtlebot, which is used widely for educational purposes.

Turtlebot waffle vs Uiabot		
Components	**Turtlebot3 Waffle**	**Uiabot**
SBC	Raspberry Pi 4	Jetson Nano A02
Battery	LIPO Battery 11.1V 1,800mAh	3s10p with 18650 cells,100A BMS
Drive system	DYNAMIXEL (XL430-W250-T), Contactless absolute encoder AS5045	DUAL SHAFT MOTOR - D5065 270KV, 20480 CPR ENCODER, PLM Series Planetary Gearbox
Rplidar	LDS-01	slamtec A1
Manipulator	OpenManipulator	Uarm
Price	1500 $	1002 $

Table 4.1: Turtlebot3 Waffle vs Uiabot.

The Turtlebot3 Waffle and Uiabot both can perform the same tasks and operation such as SLAM and autonomous navigation. Although Uiabot has relatively low cost, yet it's equipped with the all needed components. The mobile platform of the Uiabot costs around 1000 $ and 1800 $ with the Uarm manipulator, whereas the Turtlebot3 Waffle costs 1500 $ and 2500 $ with OpenManipulator.

4.1.2 Uiabot Chassis

20 waffle plates were used to build the Uiabot chassis. This number of plates can be inconvenient in 3d-printing and building process of multiple Uiabot's as planned. It's time consuming and requires a lot of assembling work (many bolts and rivets), which can be solved by designing one rigid plate that shaped as two waffle plates with less holes. Furthermore, it can be resized to be more suitable for the Uiabot since the width of the motor housing is equal to 0.32 [cm], whereas The width of the current chassis is equal to 0.28 [cm] (the length of two waffle plates), as seen in Fig. 4.1.

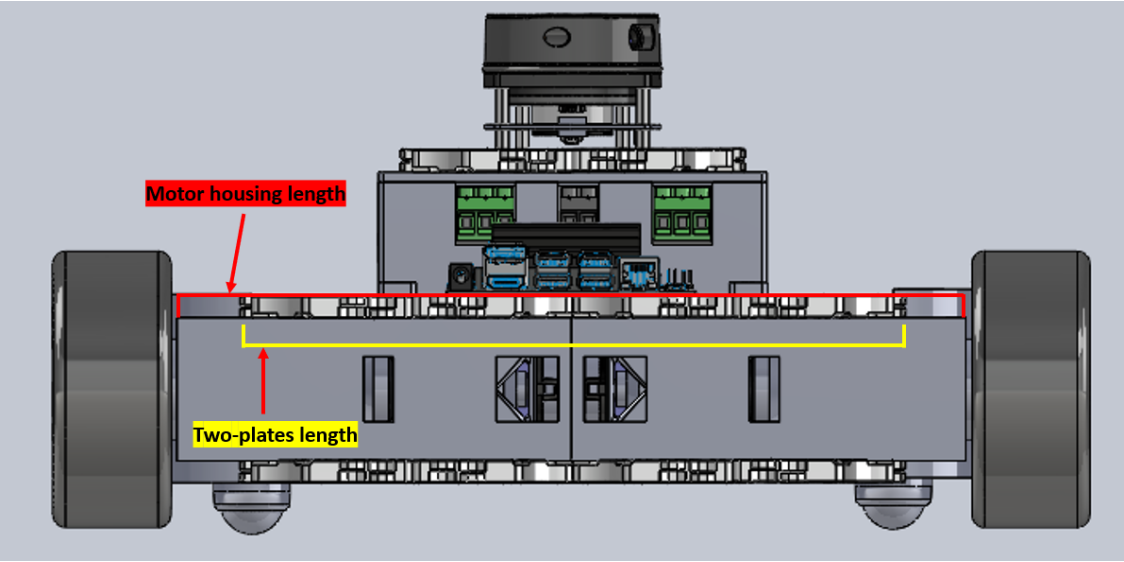


Figure 4.1: Uiabot back view.

4.1.3 Jetson Nano Software

The used 20.04 Ubuntu image is not totally reliable since it's not an official release from Jetson Nano manufacturer "Nvidia". It's relatively slower than the official 18.04 Ubuntu release, and it hanged couple of times while updating and upgrading the software. Nvidia is planning to release 20.04 Ubuntu for the Jetson Nano, however it's not schedule yet [35].

4.2 Simulation

Fig. 4.2, shows the Uiabot model in Rviz navigating towards the first waypoint using AMCL, whereas Fig.4.3 shows the Uiabot in Gazebo environment.

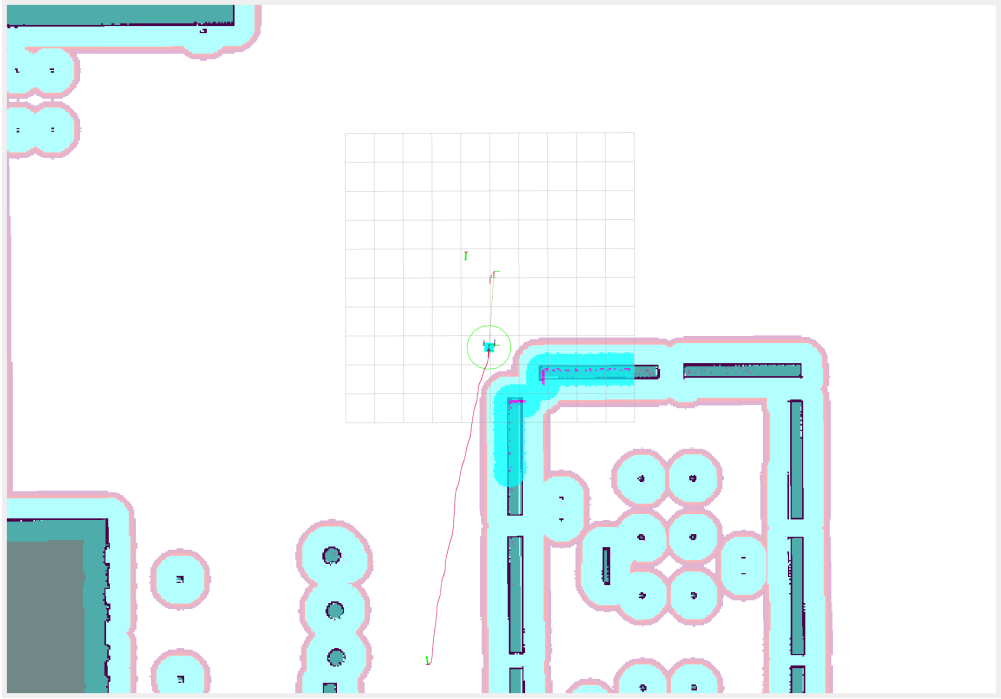


Figure 4.2: Uiabot model in Rviz.

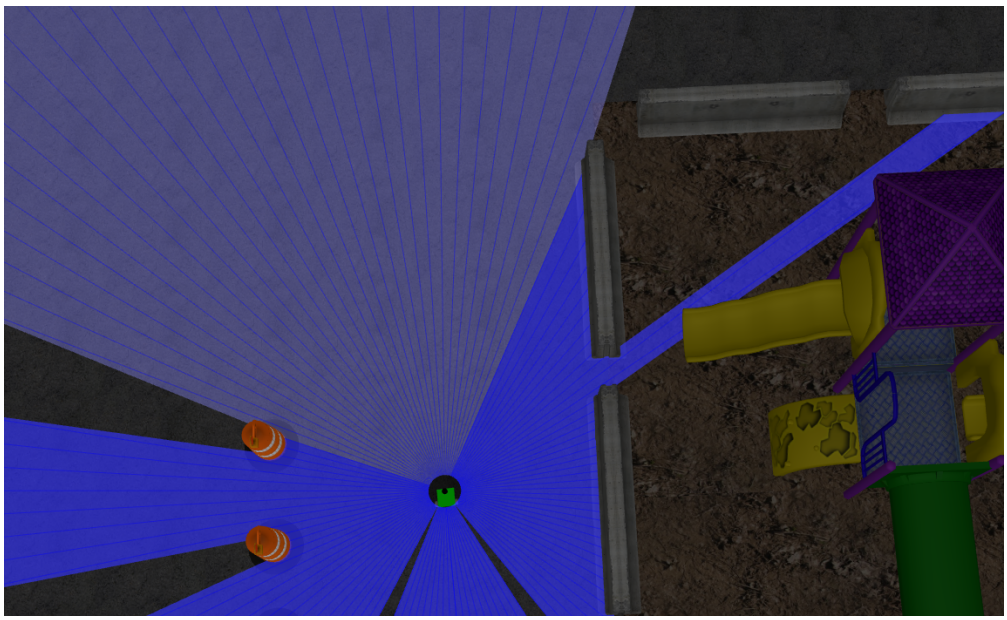


Figure 4.3: Uiabot model navigate in Gazebo using AMCL.

The visualization of the Uiabot in Rviz2 and Gazebo was built using XML in the URDF file. This can be improved by using mesh files e.g. STL and DAE files in order to be more realistic. The Uiabot's STL files was exported from the CAD model; however, it didn't work which is known problem in ROS2, and it is preferable to use DAE files.

4.3 Experimental

4.3.1 Initial Testing

Drive System

Fig. 4.4 shows a comparison between the desired velocity that is publishes on the */cmd_vel* topic as twist message versus the actual velocity measured by encoders and published on the */rwheel_vtarget* and *lwheel_vtarget* topics. The twist message contains the desired velocity of the Uiabot $\dot{\xi}$ which converted to motor's desired velocity $\dot{\phi}_r$ and $\dot{\phi}_l$ using the forward differential drive kinematics equation in the twist node. Fig. 4.6 shows the wheel odometry visualization in Rviz2.

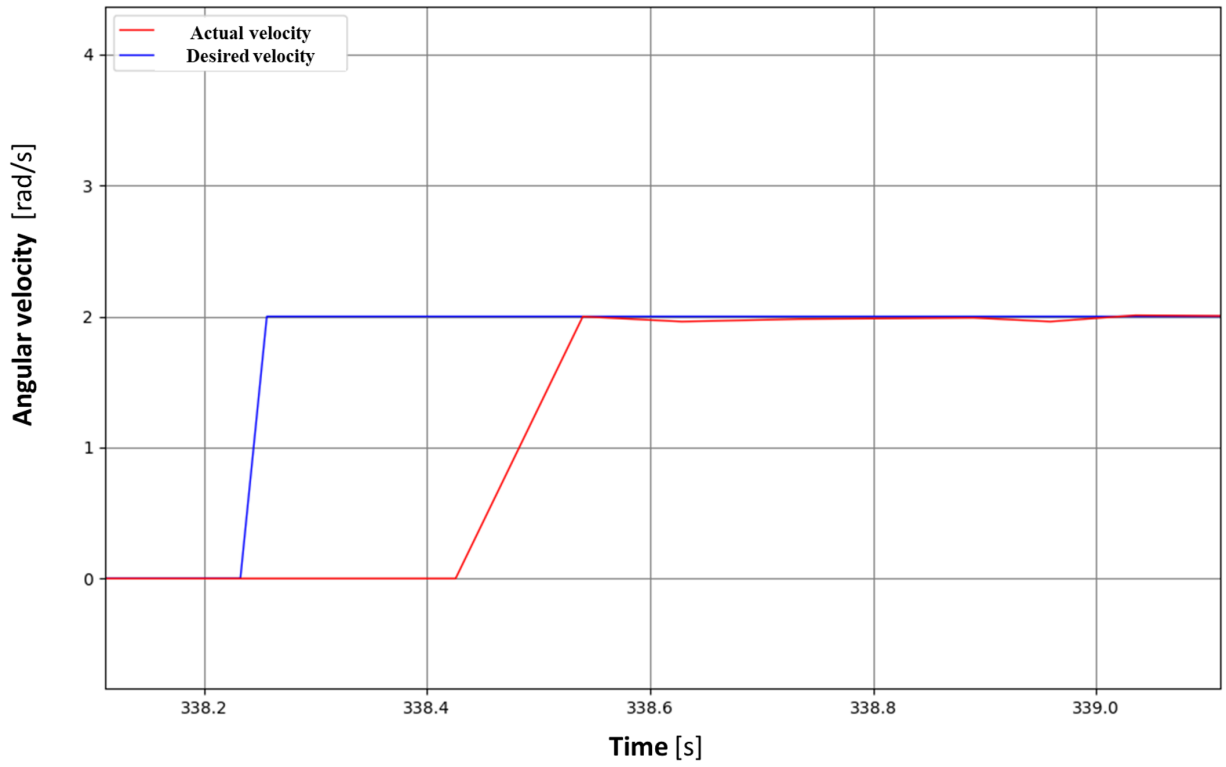


Figure 4.4: Dynamic response of Uiabot right motor.

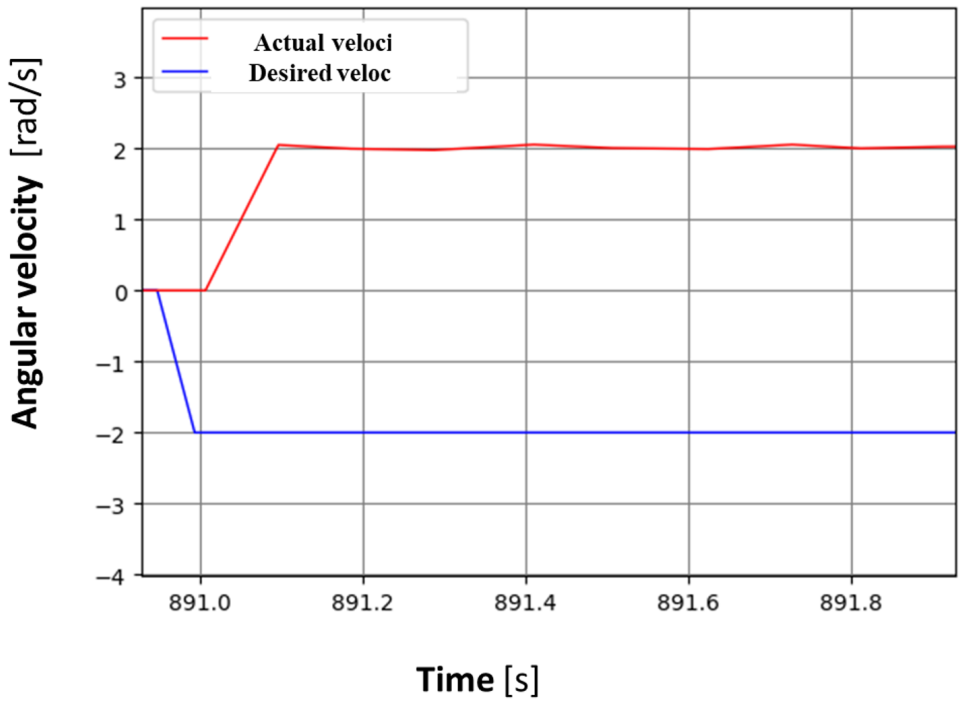


Figure 4.5: Dynamic response of Uiabot left motor.

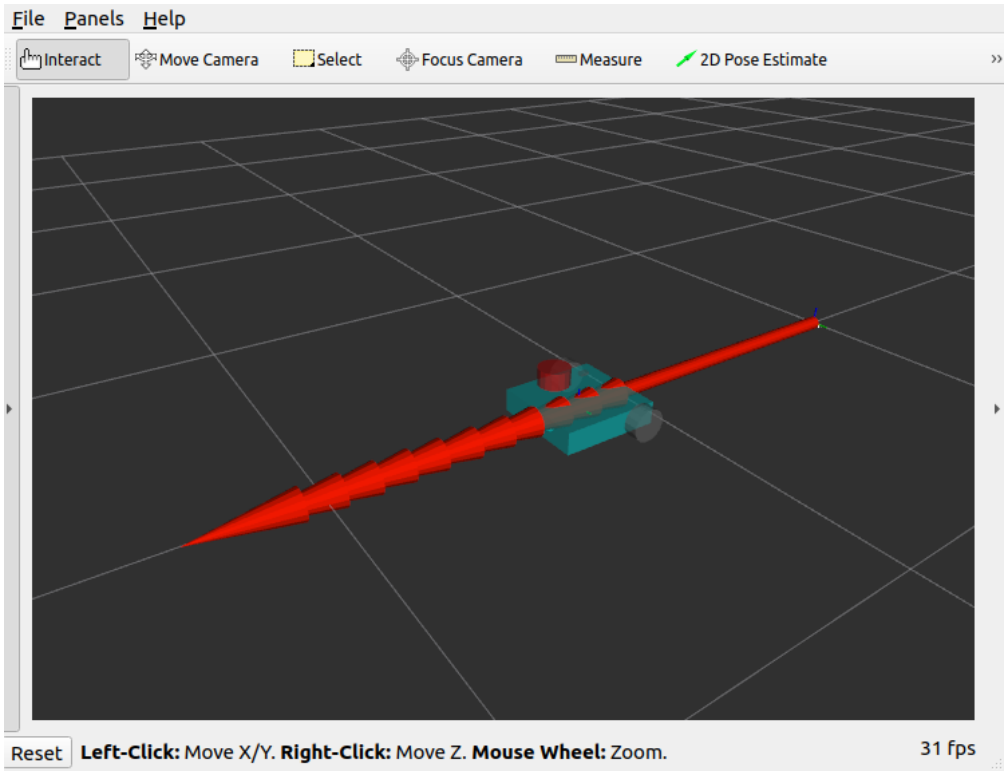


Figure 4.6: Visualization of the odometry in Rviz2.

Left Motor Velocity Signal
As seen in Fig. 4.5, the desired velocity of the left motor is negative. This is due to the fact that the right and left motor rotate in the same direction (clockwise) when a positive signal is applied. Therefore, the desired velocity signal of the left motor multiplied by a negative sign in the inverse kinematic equation to ensure that the Uiabot moves forward with positive signal from the oDrive.

Drive System ROS2 Package
Using ROS2-control framework [23] that provides different plugins could be beneficial and more reliable instead of building the odometry and twist nodes form the kinematic model. The differential drive control plugin in the ROS2-control framework requires the ROS2 wrapper (interface) of the component, and in returns it publishes the odometry message on the */odom* topic and subscribe to the */cmd_vel* topic.

However, it has one drawback which is The ROS2 interface must to be written in certain structure which could change from one ROS version to another. [19] is an oDrive package for ROS2-control, it is built for ROS2 foxy and can not be directly used for ROS2 galactic.

Inertial Measurement Unit

Fig. 4.7 shows the IMU axes in Rviz2. When the BNO055 is rotated, the axes rotate accordingly.

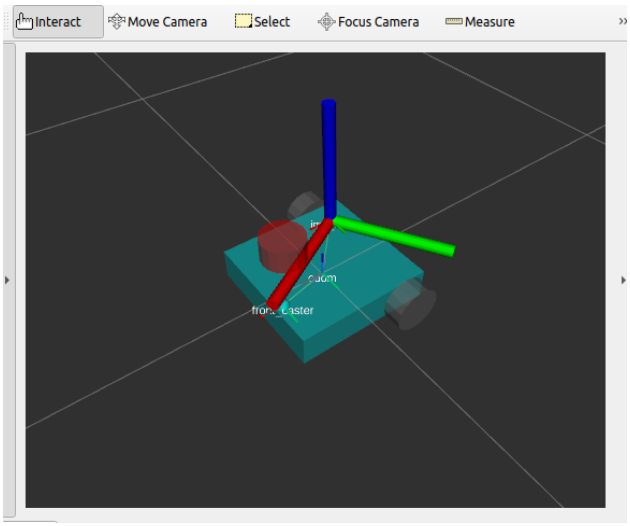
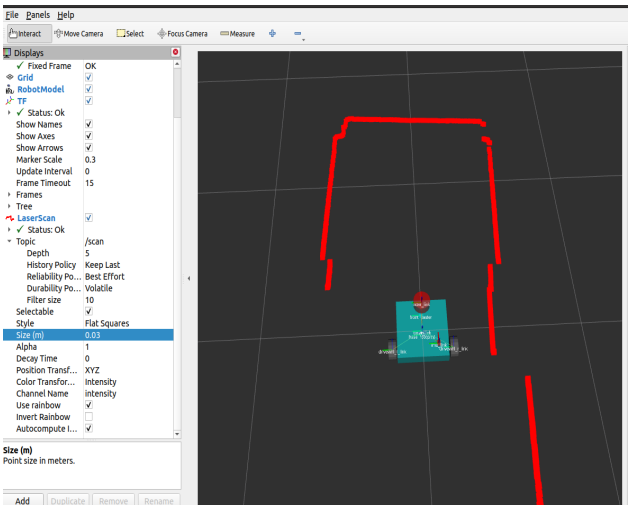


Figure 4.7: IMU axes visualization in Rviz2.

RpLiDAR

Laser scan data is published as *sensor.msgs* message on the */scan* topic by the rplidar node. Fig 4.8 shows the laser scan points in Rviz2 and rplidar surroundings.

(a) Rplidar surroundings.



(b) Visualization of the laser scan data in Rviz2.

Figure 4.8: Laser scan information.

Bring Up Launch File

bring up file is used to launch all the drive system and sensor nodes of the Uiabot. In addition URDF file and robot state publisher to publish the static transforms of the Uiabot. Fig. 4.9 shows the visualization of the Uiabot in Rviz2 and Uiabot links, after launching the *bring up* file.

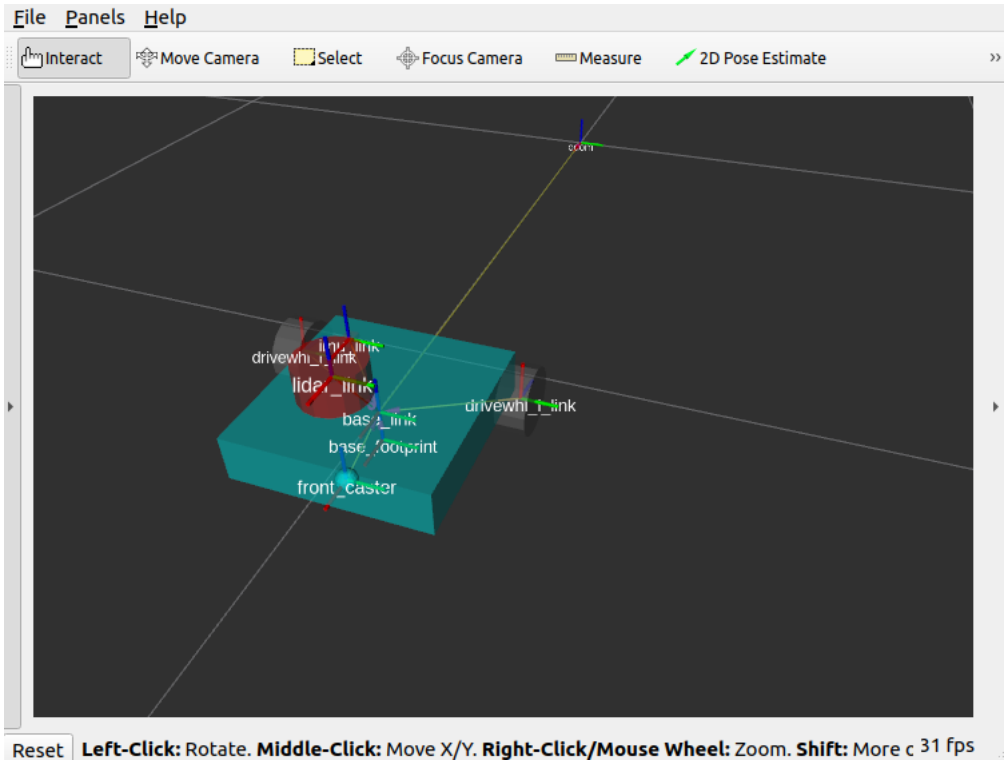


Figure 4.9: Visualization of Uiabot in Rviz2.

4.3.2 SLAM Test

SLAM toolbox is used for localization the Uiabot in a the environment and to build a 2D-map of Uiabot environment. Uiabot *bring up* launch file is used to publish the laser scan data and the *odom* $\Rightarrow$ *base_link* transform. In return, it publishes the *map* topic and the *map* $\Rightarrow$ *odom*. Fig. 4.10 and 4.11 shows a comparison between the real environment and SLAM generated environment.

Figure 4.10: Real environment.

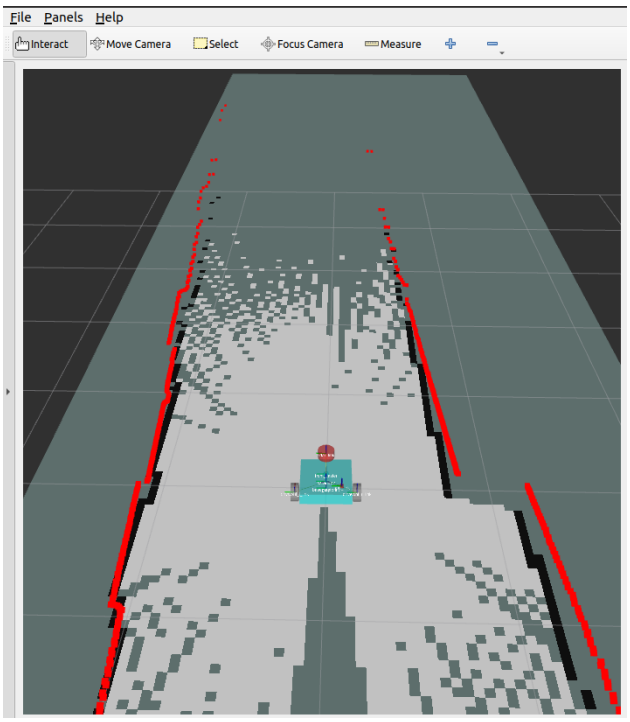


Figure 4.11: SLAM generated environment.

Fig. 4.12 shows a map of the mechatronics department at UiA that is generated using Uiabot. This map are also used for the autonomous navigation.

Figure 4.12: Map of the mechatronics department at UiA using Uiabot and SLAM toolbox.

A larger map also was created to validate SLAM toolbox mapping tool in larger scale, as seen in Fig. 4.13.

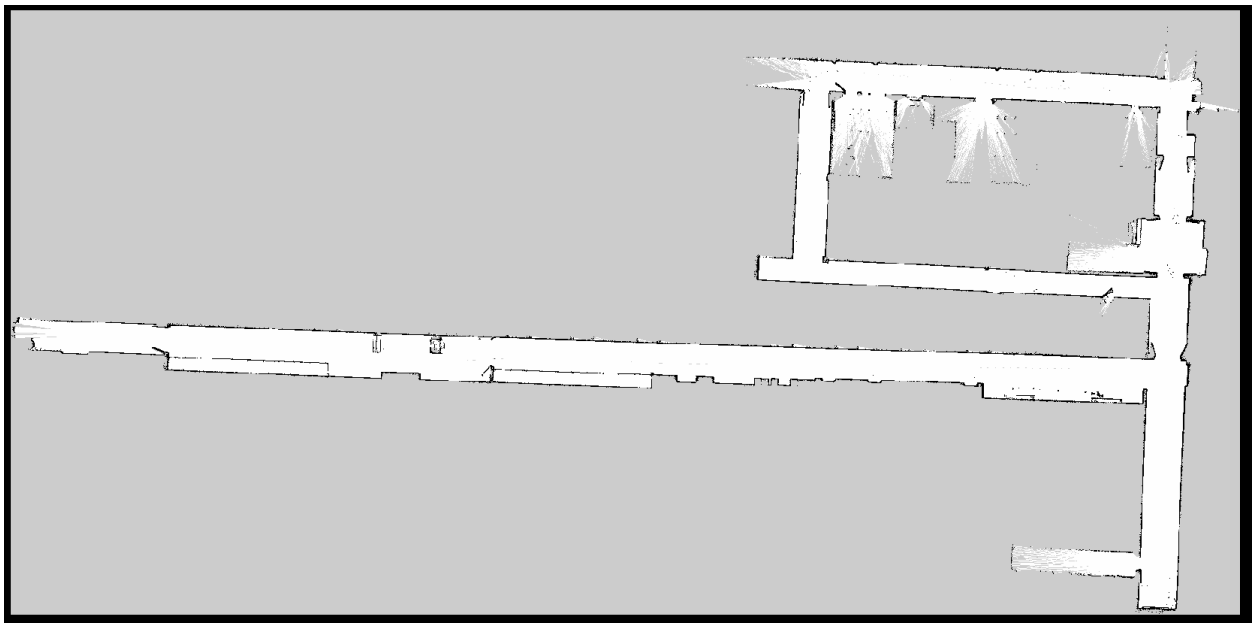


Figure 4.13: Map of C building third floor at UiA using Uiabot and SLAM toolbox.

As seen from the map in Fig. 4.14, the meeting room with glass walls were detected as a free space. The Uiabot would try to plan a path through it (magenta trajectory) during autonomous navigation, as seen in Fig. 4.15. This is due to the laser beam penetrating through the glass.

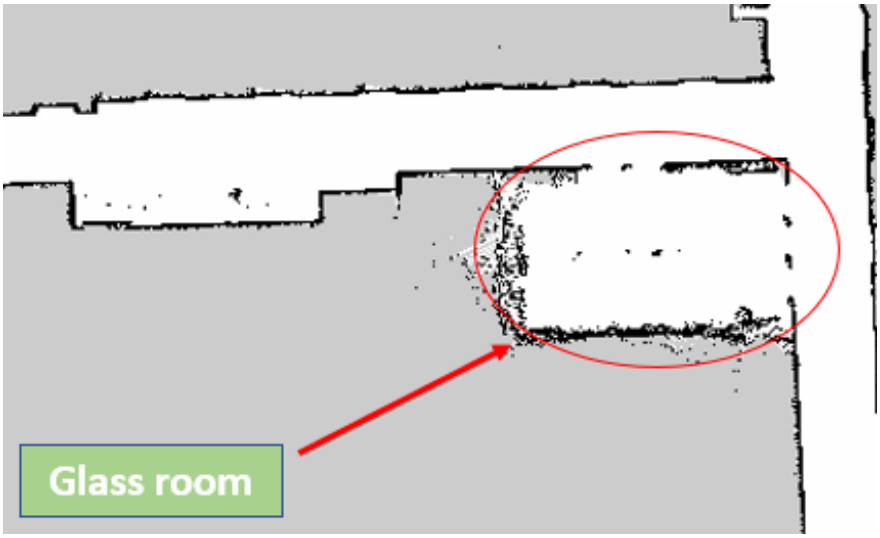


Figure 4.14: Meeting room with glass walls.

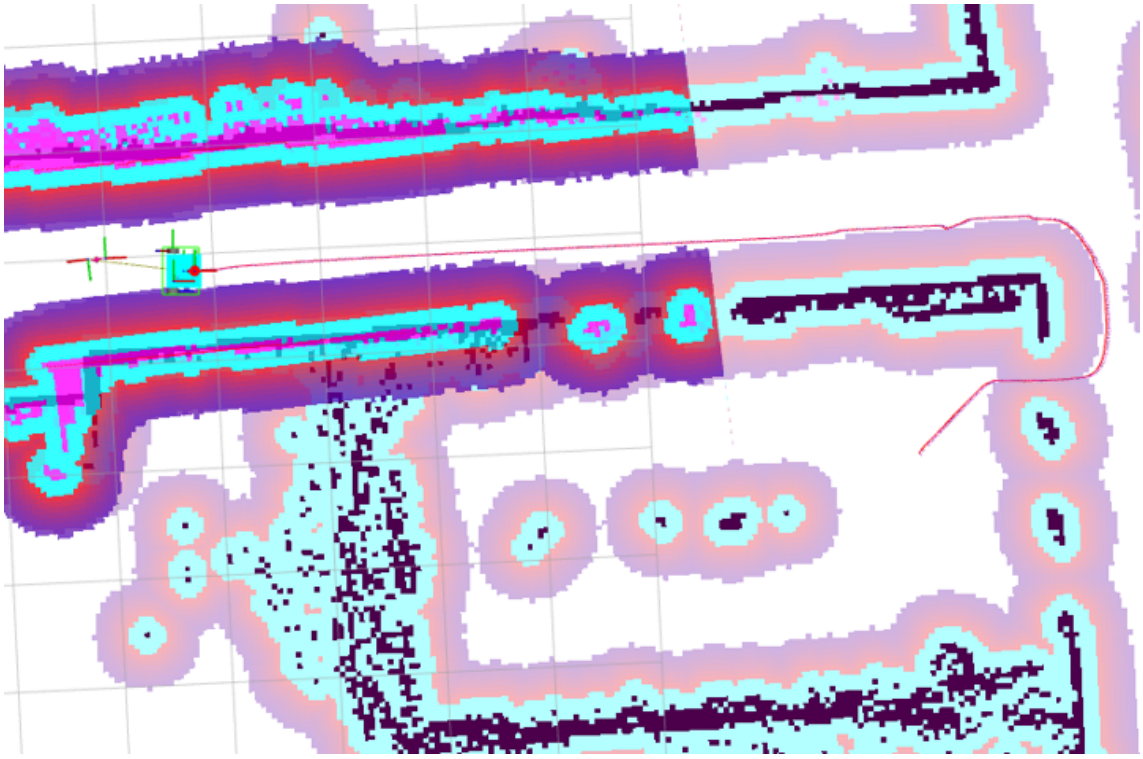


Figure 4.15: Uiabot plans the path through the glass walls.

4.3.3 Autonomous Navigation without SLAM Test

Launching the Uiabot *bring up* and the Uiabot navigation files will show a visualization of Uiabot and it's environment (2D-map) in Rviz2, as seen in Fig. 4.16. In this method Nav2 uses AMCL.

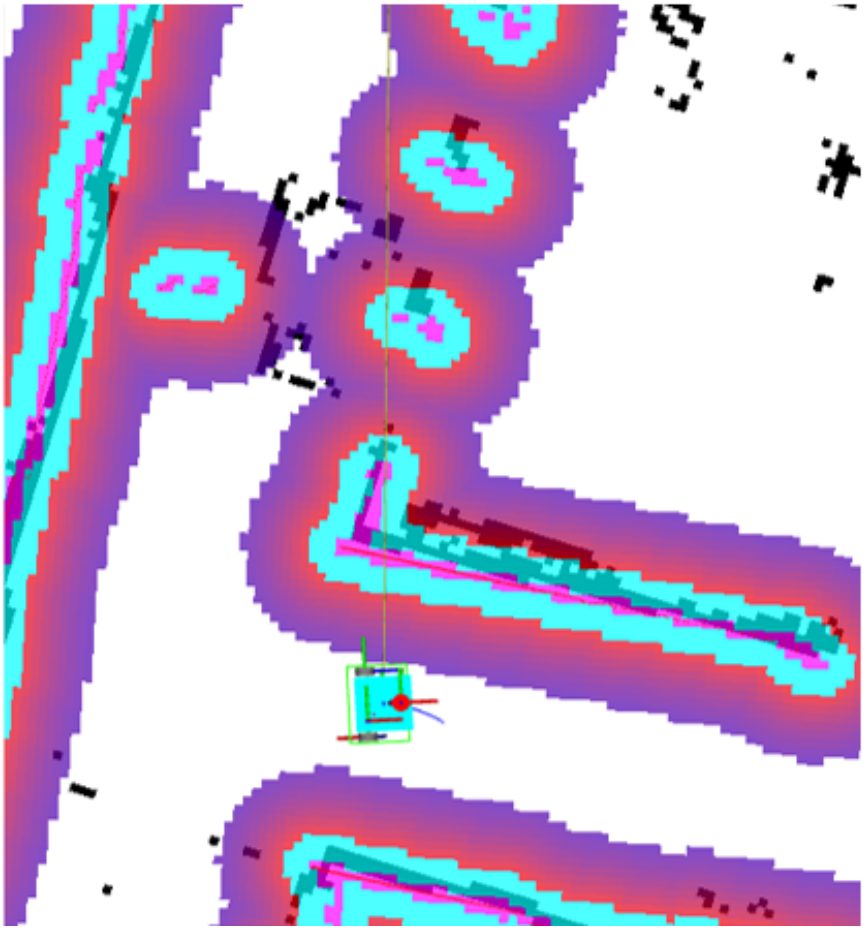


Figure 4.16: Uiabot navigation in static map.

Before a goal is sent for the Uiabot to navigate towards, the initial pose of the Uiabot is defined using the "2D Pose Estimate" in Rviz2 to align the map and the Uiabot pose.

Two waypoints were selected (green arrows) on the map using the navigation panel for the Uiabot to follow, as seen in Fig. 4.17.

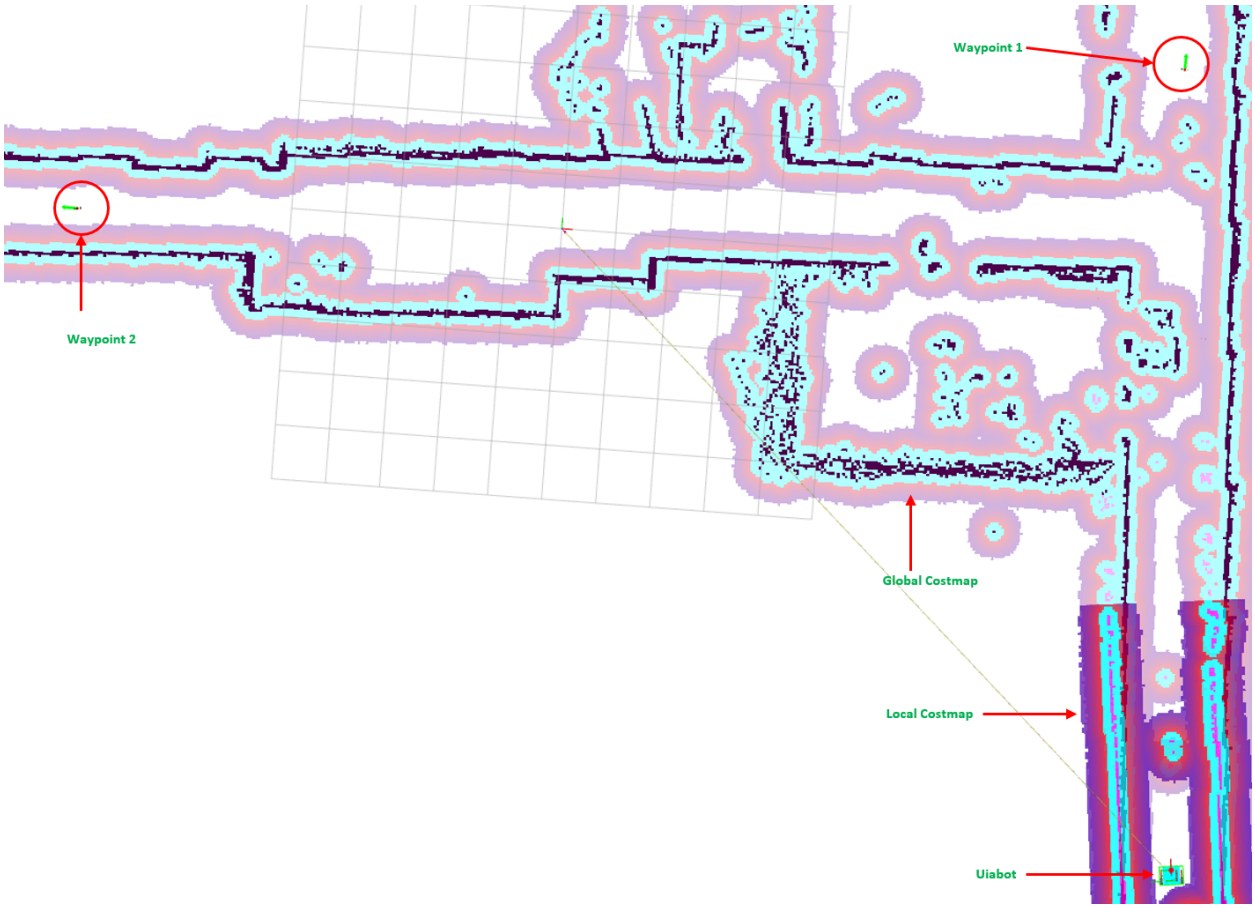


Figure 4.17: Set up two waypoints for the Uiabot.

First waypoint were at the end of section 1 of the aisle, and the second waypoint was at the end of section two of the aisle. Fig. 4.17 shows also the local and global Costmaps.

The Uiabot planned a feasible global path (red trajectory) through these waypoints, as seen in Fig. 4.18, and managed to avoid static and dynamic obstacles along it's path by planning a local path (blue trajectory), as seen in Fig. 4.19.

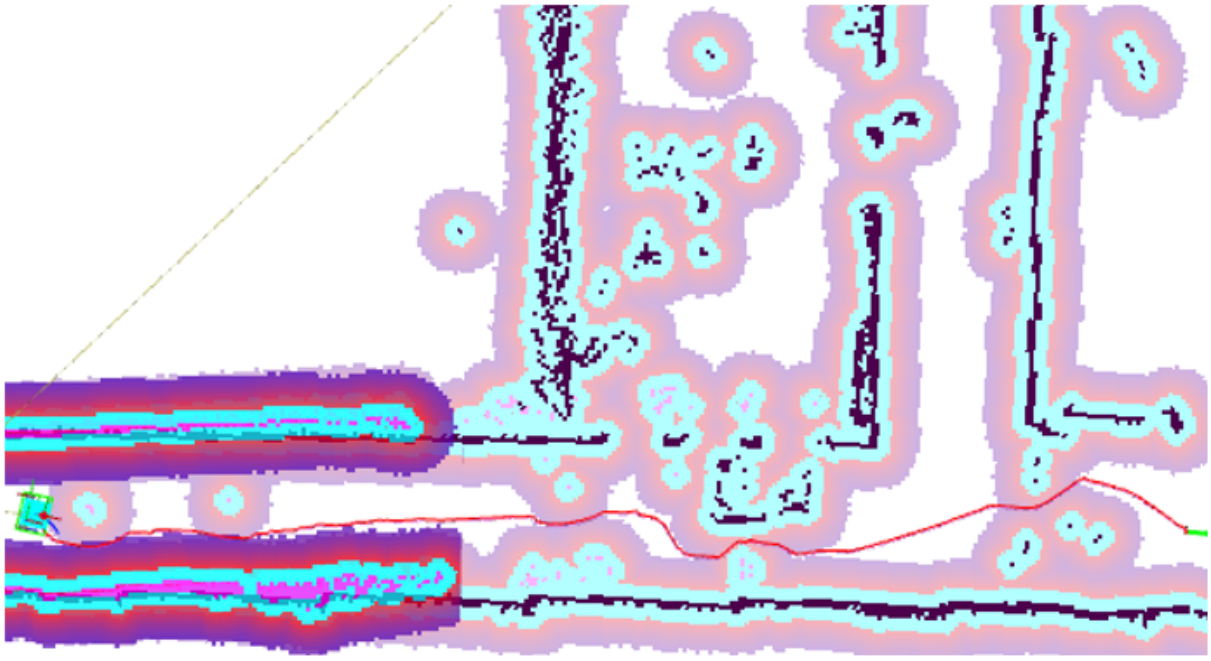


Figure 4.18: Global path planning through the first waypoint.

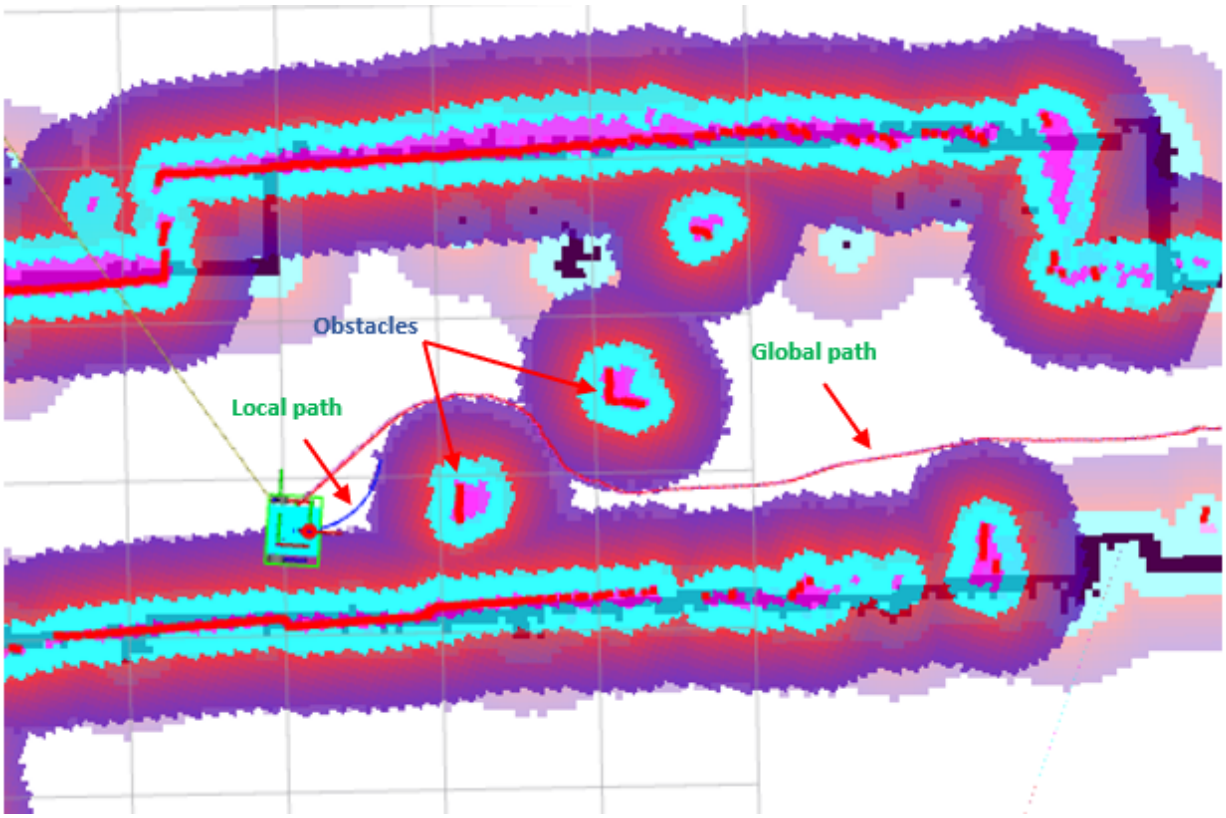


Figure 4.19: local path planning to avoid the obstacles.

4.3.4 Autonomous Navigation with SLAM Test

In this method SLAM toolbox is used for localization purposes. As shown in Fig. 4.20, Uiabot starts to localize itself and map the surroundings at same time using the laser scan data.

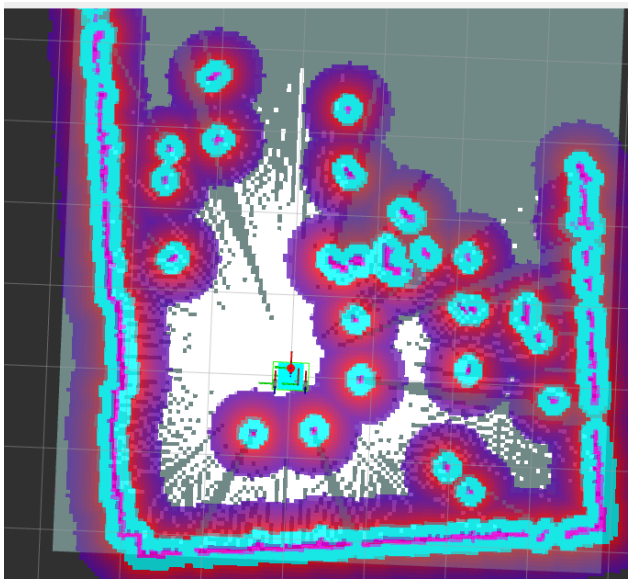


Figure 4.20: Uiabot starts to generate a map of the surroundings.

Fig. 4.21, shows the robot navigate from waypoint 1 to wards waypoint 2.

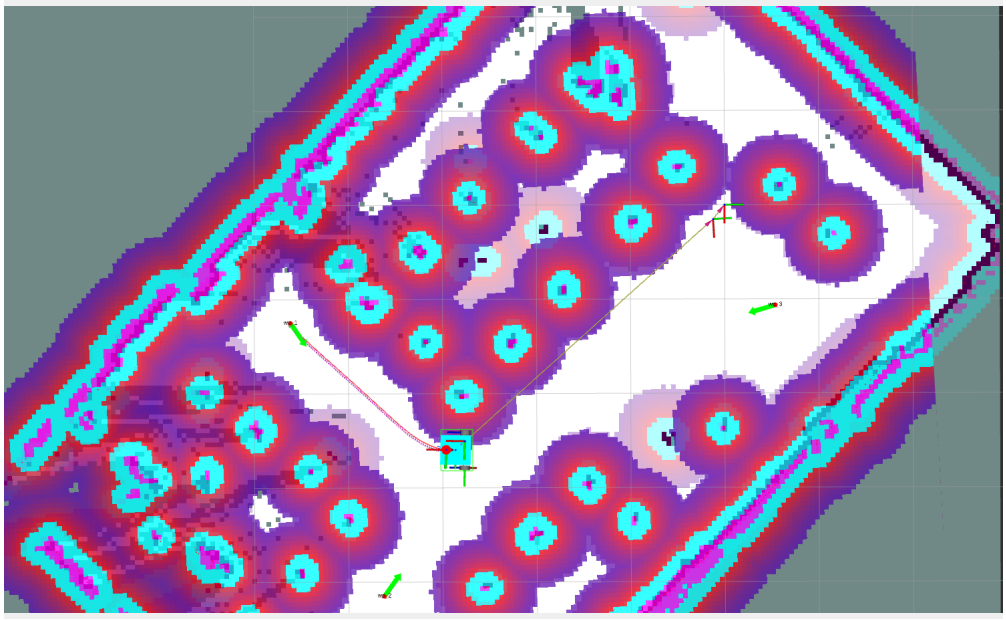


Figure 4.21: Uiabot navigates from waypoint 1 to waypoint 2.

Chapter 5

Conclusions

One of the main goals of this thesis was to design an affordable mobile platform that fits the existing uArm manipulator. The designed mobile platform adopted the concept of the waffle plates used in the state of the art *Turtlebot3*. The chassis of the mobile robot can be resized and reshaped to fit the uArm manipulator and the user requirements, which adds more flexibility to the structure of the mobile robot and can be beneficial in future works. The designed mobile platform prototype has a cost of about $1000. Compared to the *Turtlebot3* platform, the cost savings are $500 (33.33%).

The second goal was to develop the required ROS2 packages for the selected hardware of the mobile robot to perform SLAM and autonomous navigation using the SLAM toolbox and Nav2. This goal was achieved by programming and configuring the necessary ROS2 packages for the oDrive, IMU, and RpLiDAR using ROS2. The structure of the developed ROS2 packages was documented in the methods chapter as a user guide to assist the students who will conduct further development on the mobile robot in robotics and computer vision courses.

Also, a prototype of the designed mobile robot was built and constructed using the modified and printed waffle plates and the provided hardware in order to perform SLAM and autonomous navigation tests using the developed ROS2 packages. First, a gazebo simulation model of the mobile platform was built and examined by using the SLAM toolbox and Nav2 framework. The result of the software implementation showed that the packages of the hardware components were appropriate and acceptable to be used for SLAM and autonomous navigation, and all the required information for Nav2 was accomplished and available. Secondly, experiments using the prototype showed that the mobile robot was able to build a precise map of the test scene and was successfully able to navigate autonomously with and without the existence of a static map and avoid the static and dynamic obstacles.

5.1 Contributions

1. A conceptual design of a manipulator mobile robot.
2. A complete prototype of mobile robot base designed for carrying the manipulator.
3. Development of ROS2 packages for the specific selected hardware of the Uiabot, ready for SLAM toolbox and Nav2 usage.

5.2 Future Work

As stated earlier, this project focuses on the software development of the mobile platform. Therefore, the next step is to mount the uArm manipulator on the developed mobile platform. Since the uArm officially only supports ROS1, the ROS2 package for the uArm must be developed, integrated, and tested with the existing packages.
The SLAM toolbox provides different modes for mapping in addition to the modes that were implemented in this project that should be further explored. The serialized and life-long mapping in the SLAM toolbox allows the mobile robot to build a map on a large scale in several sessions [28].

Furthermore, the Nav2 framework provides many plugins and tools to perform different tasks e.g. boarding an elevator and docking with a charging station. Also, safety zones function and speed restriction areas function to keep the mobile robot out of a specific zone and restrict the velocity of the robot in certain areas can be implemented using the Nav2 [18].

The capabilities of the developed autonomous robot platform are promising, and further developments can be performed in several fields such as instrumentation, SLAM, autonomous navigation, and manipulation. This project will provide future students with the foundation and needed tools to accomplish their ideas and projects on mobile manipulator robots.

Bibliography

[1] *8192 CPR ENCODER WITH ODRIVE V3 CABLE*. URL: https://odriverobotics.com/shop/cui-amt-102.

[2] *BNO055 Calibration*. URL: https://www.youtube.com/watch?v=Bw0WuAyGsnY&ab_channel=BoschSensortec.

[3] *BNO055 ros2 package using I2C*. URL: https://github.com/bdholt1/ros2_bno055_sensor.

[4] *BNO055 ros2 package using UART*. URL: https://github.com/flynneva/bno055.

[5] *Create 3*. URL: https://edu.irobot.com/what-we-offer/create3.

[6] *Differential drive plugin*. URL: https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#DifferentialDrive.

[7] Edsger W Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[8] José Ahirton Batista Lopes Filho, Will Ribamar Mendes Almeida, and Sérgio Gomes Martins. "Development of a multitasking mobile robot for the construction of educational robotics kits." In: *2011 International Conference on Electronic Devices, Systems and Applications (ICEDSA)*. 2011, pp. 213–216. DOI: 10.1109/ICEDSA.2011.5959090.

[9] Tinghang Guo et al. "Design of Educational Mobile Robot." In: *2020 Chinese Automation Congress (CAC)*. 2020, pp. 3234–3238. DOI: 10.1109/CAC51589.2020.9326549.

[10] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "uCorrection/u to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"." In: *ACM SIGART Bulletin* 37 (Dec. 1972), pp. 28–29. DOI: 10.1145/1056777.1056779. URL: https://doi.org/10.1145/1056777.1056779.

[11] *IMU plugin*. URL: https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#IMUsensor(GazeboRosImuSensor).

[12] *Joint State publisher plugin*. URL: http://docs.ros.org/en/jade/api/gazebo_plugins/html/gazebo__ros__joint__state__publisher_8h_source.html.

[13] Artem Lenskiy et al. "Educational platform for learning programming via controlling mobile robots." In: *2014 International Conference on Data and Software Engineering (ICODSE)*. 2014, pp. 1–4. DOI: 10.1109/ICODSE.2014.7062695.

[14] Steve Macenski and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world." In: *Journal of Open Source Software* 6.61 (2021), p. 2783. DOI: 10.21105/joss.02783. URL: https://doi.org/10.21105/joss.02783.

[15] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild." In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[16] *Manipulator robot*. URL: https://www.gobankingrates.com/money/jobs/richard-branson-work-culture/.

[17] *Mobile robot*. URL: photo:https://mobilerobotguide.com/2019/06/13/mir-announces-mir-financing/.

[18] *Nav2 capabilities*. URL: https://navigation.ros.org/concepts/index.html.

[19] *Odrive package for ros2-control*. URL: https://github.com/Factor-Robotics/odrive_ros2_control.git.

[20] *Open manipulator-x*. URL: https://www.robotis.us/openmanipulator-x-rm-x52-tnm/.

[21] *PLM Series Planetary Gearbox Gear*. URL: https://www.omc-stepperonline.com/plm-series-planetary-gearbox-gear-ratio-20-1-backlash-50-arc-min-for-8mm-shaft-nema-23-stepper-motor-plm23-g20-d8.

[22] *Robot Localization Package*. URL: http://docs.ros.org/en/api/robot_localization/html/index.html.

[23] *Ros2 Control Framework*. URL: https://control.ros.org/master/index.html.

[24] *ROS2 Galactic Installation*. URL: https://docs.ros.org/en/galactic/Installation.html.

[25] *RpLiDAR ROS2 Package*. URL: https://github.com/Slamtec/rplidar_ros.git.

[26] Roland Siegwart and R.Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Massachusetts Institute of Technology, 2004. ISBN: 0-262-19502-X.

[27] *Smart sensor : BNO055*. URL: https://www.bosch-sensortec.com/products/smart-sensors/bno055/.

[28] SteveMacenski. *SLAM toolbox*. URL: https://github.com/SteveMacenski/slam_toolbox.git.

[29] *Turtlebot*. URL: https://spectrum.ieee.org/interview-turtlebot-inventors-tell-us-everything-about-the-robot.

[30] *Turtlebot3*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications.

[31] *Turtlebot3 Motor*. URL: https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/.

[32] *Turtlebot3 Open manipulator-x*. URL: https://emanual.robotis.com/docs/en/platform/turtlebot3/manipulation/.

[33] *Turtlebot4*. URL: https://turtlebot.github.io/turtlebot4-user-manual/overview/features.html.

[34] *Ubuntu 20.04 image Q-engineering*. URL: https://github.com/Qengineering/Jetson-Nano-Ubuntu-20-image.

[35] *Ubuntu 20.04 released for Jetson Nano*. URL: https://developer.nvidia.com/jetpack-sdk-501dp.

[36] Ufactoryi. *Uarm Swift Pro*. URL: http://download.ufactory.cc/docs/en/uArm%5C%20Swift%5C%20Pro_Developer%5C%20Guide%5C%20v1.0.6.pdf.

[37] *Xubuntu 20.04 image*. URL: https://forums.developer.nvidia.com/t/xubuntu-20-04-focal-fossa-l4t-r32-3-1-custom-image-for-the-jetson-nano/121768.

[38] Jianjun Zhu and Li Xu. "Design and Implementation of ROS-Based Autonomous Mobile Robot Positioning and Navigation System." In: *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 2019, pp. 214–217. DOI: 10.1109/DCABES48411.2019.00060.