PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

University of Amar Telidji - Laghouat

Faculty of Technology

Department of Electronics

MASTER THESIS

DOMAIN: Science & Technology

FIELD: Automation

SPECIALTY: Automation & Industrial Computing

SACI Zhour & **BENLAHBIB Serine**

Theme

Navigation and Control System of Mobile Robot using ROS

Jury members:

BELKHIRI Mohammed	Prof	President
REGUIEGUE Mourad	MCB	Examiner
OUBBATI Brahim Khalil	MCB	Supervisor
AHMINE Yacine	MCB	Co-Supervisor

2022 / 2023

Abstract

Mobile robots have become one of the most used technologies recently, due to its efficiency, accuracy and speed of executing orders. This work aims to build a mobile robot with two wheels and the application of simultaneous localization and mapping algorithms, such as Hector SLAM and G-mapping to construct maps of unknown environments. In addition, the project is based on using sensors such as camera, Lidar, and the JETSON NANO card for data acquisition and processing. Experimental tests include implementing Hector SLAM algorithms, controlling the robot via a keyboard, and performing autonomous navigation and collision avoidance using a camera and deep learning techniques. The objective is to develop a versatile mobile robot capable of mapping and autonomous operation.
Keywords: Mobile Robot, Robotic Operating System ROS, Simultaneous Localization and Mapping SLAM, Hector SLAM.

Résumé

Les robots mobiles sont devenus l'une des technologies les plus utilisées récemment, en raison de leur efficacité, de leur précision et de leur rapidité d'exécution. Ce travail vise à construire un robot mobile à deux roues et à appliquer des algorithmes de localisation et de cartographie simultanées, tels que Hector SLAM et G-mapping, pour construire des cartes d'environnements inconnus. De plus, le projet est basé sur l'utilisation de capteurs tels que la caméra, le Lidar, et la carte JETSON NANO pour l'acquisition et le traitement des données. Les tests expérimentaux comprennent l'implémentation des algorithmes SLAM d'Hector, le contrôle du robot via un clavier,

et la navigation autonome et l'évitement des collisions à l'aide d'une caméra et de techniques d'apprentissage profond. L'objectif est de développer un robot mobile polyvalent capable de cartographier et de fonctionner de manière autonome.

mots-clés:

Robot mobile,Système d'exploitation robotique ROS, localisation et cartographie simultanées SLAM, Hector SLAM.

ملخص

اصبحت الروبوتات المتنقلة من اكثر التكنولوجيات استعمالا حديثا ، نظرا لكفائتها و دقتها و سرعة تنفيذها للأوامر ، يهدف هذا العمل الى بناء روبوت متنقل ذو عجلتين مع تطبيق خوارزميات تحديد الموقع المتزامن و رسم الخرائط ، تتمثل هذه الخوارزميات في هكتور سلام و جي مابينغ التي تسمح بأنشاء خرائط في بيئات غير معروفة ، بالاضافة الى ذلك يشمل المشروع استخدام حساسات مثل كاميرا و الليدار و بطاقة جاتسون نانو للحصول على البيانات ومعالجتها ،تشمل الاختبارات التجريبية تنفيذ خوارزميات هيكتور سلام ، والتحكم في الروبوت عبر لوحة مفاتيح ، وأداء التنقل المستقل وتجنب الاصطدام باستخدام الة التصوير وتقنيات التعلم العميق.

الكلمات المفتاحية : الروبوت المتنقل ، نظام التشغيل الألي روس ، تحديد الموقع المتزامن و رسم الخرائط سلام، هكتور سلام

Acknowledgements

We would like to express our sincere appreciation and gratitude to our esteemed supervisors, Mr. OUBBATI Brahim Khalil and Co-supervisor Mr. AHMINE Yacine.Their guidance, encouragement and continuous support were instrumental in the successful completion of our Master's thesis.

We extend our sincere thanks to the, Prof. BELKHEIRI Mohammed, for his valuable inputs and thoughtful suggestions throughout the thesis process. We are also grateful to the laboratory LTSS for providing us with the necessary resources and facilities to conduct our research. Their contributions were indispensable in achieving our research goals.

We would like to express our deepest gratitude to our families. Their endless love, understanding, and unwavering support have been our pillar of strength throughout this academic endeavor. We are forever grateful for the sacrifices they have made to ensure our success.

SACI zhour & BENLAHBIB serine
Laghouat University
July 2023

Acronyms

ROS Robot Operating System

ROVs Remotely operated vehicles

LTS Long-Term Support

GPS Global Positioning System

IMUs Inertial measurement units

SLAM Simultaneous Localization and Mapping

LIDAR Light Detection and Ranging

AUVs Autonomous underwater vehicles

SIR Sampling Importance Resampling

AI Artificial intelligence

SCMD Serial Controlled Motor Driver

PTH Plated Through-Hole

DSP Digital Signal Processor

ToF Time of-Flight

PSD Position Sensitive Device

FOV Field of View

Rvis ROS visualization

Contents

List of Figures

General Introduction

The progression of humanity during the Industrial Revolution led to remarkable advancements in the creation of mechanisms and tools that made life easier and facilitated progress. Among these inventions, robots emerged as one of the most influential and versatile technologies[1]. With their ability to perform tasks autonomously or under human guidance, robots have found applications across diverse fields, serving and assisting people in countless ways, robots have become indispensable companions, contributing to increased efficiency, enhanced productivity, and improved quality of life [1].
The main objective of this Master thesis is to build a mobile robot with two wheels, using jetson NANO card and different sensors, and implement a perception algorithm for a mobile robot, specifically focusing on Simultaneous Localization and Mapping (SLAM). This project utilizes a classical SLAM algorithm to construct 2-D maps and estimate the position of the mobile robot.

This master thesis is divided into four chapters besides the introduction and the conclusion:

- **In chapter one**, we will present the mobile robots and their application in real life, and we will talk about Robot Operating System (ROS) , and principle architecture of ROS that provides a set of tools and libraries permit developers to create a software application for robot quickly and easily .
- **The second chapter** is devoted to mathematical modeling of mobile robot with two wheels, and explain two important techniques in SLAM (Hector SLAM and Gmapping), and their ability to construct a map of unknown environment.
- **The third chapter** is dedicated to the process of building a robot using various

components, such as sensors (Light Detection and Ranging (LIDAR) and camera) to collect data in unknown environment, and JETSON NANO card supported by software libraries for deep learning, computer vision, and much more.

- **In the fourth chapter**, we test experimental, we will implementing hector slam algorithms to build a map and we will controlling mobile robot via keyboard, and navigate autonomously and avoid collisions using camera sensor and deep learning techniques.

Chapter 1

State of art: mobile robot and ROS

1.1 Introduction

Mobile robots have become more popular in various fields, from industrial automation to healthcare and exploration. These robots are designed to operate in dynamic environments, navigate through unknown surroundings, and perform specific tasks, navigation, and control are fundamental aspects of mobile robots. ROS provides a comprehensive set of tools and algorithms for navigation and control, including mapping, localization, path planning, and motion control. These capabilities enable mobile robots to navigate autonomously, avoid obstacles, plan optimal paths, and execute precise motions. The integration of ROS with mobile robots empowers them with advanced navigation and control capabilities, allowing them to operate efficiently in diverse and difficult environments.

In this chapter we take a look about the types of mobile robots and their application in real life, and we discuss the definition and history of ROS, and also we explore their architecture such as ROS master, ROS nodes, topics, files launch, ... etc, and we will explain how the Nodes communicate with each other.

1.2 The mobile robot

1.2.1 Definition

A mobile robot is an autonomous or semi-autonomous robotic system designed to move and operate in various environments. It is equipped with mechanical components, such as wheels, legs, or propellers, that enable mobility and the ability to navigate through its surroundings.

Mobile robots are equipped with sensors to perceive the environment, such as cameras, lidars, or sonars, which provide information about obstacles, landmarks, or other relevant data. They also possess actuators, such as motors or manipulators, to interact with the environment and perform specific tasks.

1.2.2 Classification of Mobile Robots

Mobile robots come in various types, each designed for specific tasks and operating in different environments. Here are some common types of mobile robots along with their applications:

Wheeled Robots:

Figure 1.1 shows the wheeled robots. These are versatile and are commonly used in indoor environments for tasks such as material handling, transportation, surveillance, and inspection. are also used in industries such as logistics, manufacturing, and health care.

Figure 1.1: Wheeled Robots

Legged Robots

Legged robots, inspired by biological creatures like humans and animals, as shown in the below figure 1.2, are ideal for traversing challenging terrains where wheeled robots may struggle. They find applications in search and rescue operations, exploration of rough terrain, and assistance in environments with uneven surfaces.

Figure 1.2: Legged Robots

Aerial Robots

Figure 1.3 represents Aerial robots , commonly known as drones, which are used in various applications such as aerial photography, surveillance, agricultural monitoring, infrastructure inspection, and delivery services. They provide a unique perspective and are particularly useful in areas that are difficult to access.

Figure 1.3: Aerial Robots

Underwater robots

Underwater robots, also known as Remotely operated vehicles (ROVs), are used for exploration, research, and inspection tasks in underwater environments. They are employed in marine biology, offshore industries, underwater archaeology, and oil and gas exploration..

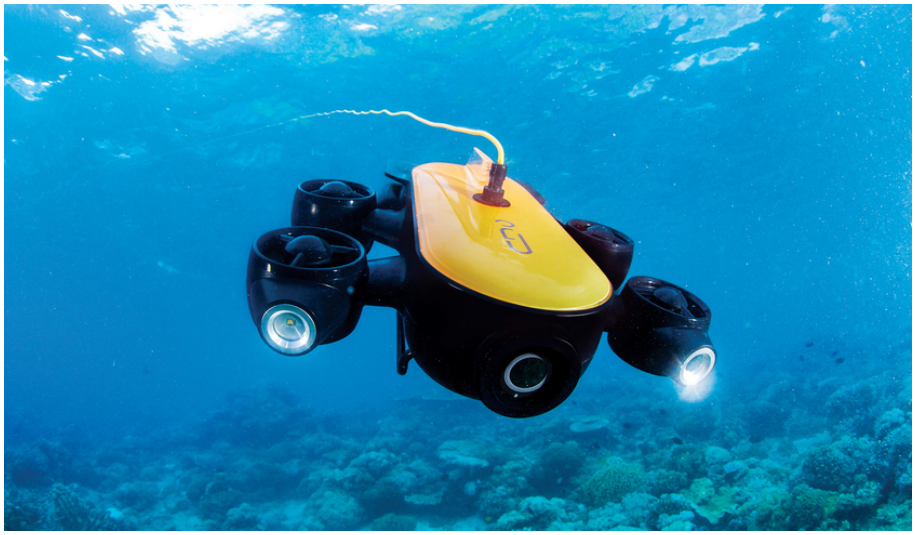


Figure 1.4: Underwater robots

1.3 Robot Operating System (ROS)

1.3.1 definition

ROS is a powerful and open-source software framework for programming robots. It is used to control robots in a variety of ways, from navigation and object recognition to motion planning and manipulation. ROS provides a set of tools and libraries that enable developers to quickly and easily create software applications for robots. ROS also provides access to a wide range of sensors, actuators, and other hardware components. This makes it possible for robots to interact with the environment in ways that were not previously possible.

1.3.2 History of ROS

- The ROS project was started in 2007 with the name SWITCHYRAD as part of the Stanford robot project.
- ROS was developed at Willow Garage for around 6 year, until Willow Garage shut down, back in 2014.

Figure 1.5: Willow Garage

- In 2013 under the open source robotics foundation ROS continued to develop and release new distribution .

Figure 1.6: Open source Robotics Foundation

1.3.3 Versions of ROS

ROS has several major versions, with each version having its own set of features and compatibility. Some notable versions of ROS include:

ROS Noetic

ROS Noetic, released in 2020, is another Long-Term Support (LTS) version of ROS. It supports Ubuntu 20.04 and newer versions. Noetic incorporates significant improvements, bug fixes, and new features, making it an excellent choice for robot development[2].

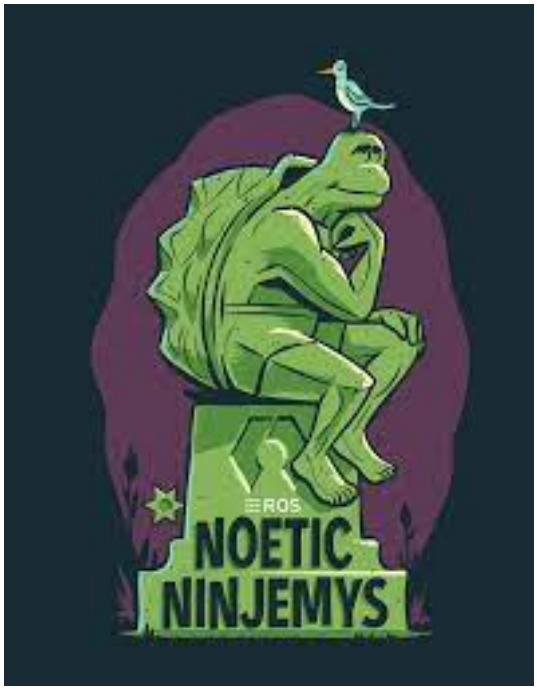


Figure 1.7: ROS Noetic NINJEMYS release

ROS Melodic

ROS Melodic is one of the LTS versions of ROS. It was released in 2018 and is compatible with Ubuntu 18.04 and newer versions. Melodic provides stability, reliability, and extensive community support[2].

Figure 1.8: ROS Melodic Morenie release

ROS Kinetic

ROS Kinetic, released in 2016, is an LTS version that is compatible with Ubuntu 16.04. It offers a stable and mature platform for robotic applications[2].

Figure 1.9: ROS Kinetic Kame release

1.3.4 Architecture of ROS

The basic architecture of ROS is shown in the following figure 1.10

ROS Master

The ROS master is the central component of the ROS architecture. It is responsible for managing the nodes and topics in the system. It keeps track of the nodes in the system, which topics they are publishing and subscribing to, and which services they are providing. The ROS master is the hub of the ROS architecture and is essential for the nodes to be able to communicate with each other.

ROS Nodes

- Its any robot software file built to work under theROS MASTER.
- Smallest unit of a processor running in ROS.
- Registers the information and specialized in each purpose.
- Acts as publisher, subscriber, service server, and service client.

ROS Topics

ROS topics are the communication channels used by nodes to communicate with each other. Each topic has a name and a data type. Nodes can subscribe to topics to receive data and publish data to topics to send data to other nodes. Topics can be used to communicate sensor data, control commands, or other data. They are the primary way that nodes communicate with each other in the ROS architecture.

Messages

Messages define the structure and content of data exchanged between nodes. ROS provides a wide range of predefined message types, and custom message types can be defined as per the robot's requirements.

Launch Files

A launch file simplifies the process of starting multiple ROS nodes together and configuring them for a specific task. Instead of starting each node separately and configuring them manually, it allows for an automated and more efficient approach. Launch files can be created and edited using a text editor, and they are typically saved with the '.launch' extension. They

can also be executed using the '$ roslaunch' command-line tool, which provides a simple way to start ROS nodes and configure their settings[2].

communication between nodes

ROS provides two primary communication mechanisms: publisher/subscriber and client/server. These mechanisms facilitate the exchange of data and allow different nodes in a ROS system to interact with each other. as shown in the figure 1.10 below

a. Publisher and subscriber

If a node wants to share information, it uses a publisher to send data to a topic. A node that wants to receive that information uses a subscriber to that same topic , Topics are used for many-to-many communication. Many publishers can send messages to the same topic and many subscribers can receive them [2].

b. services

are another way for nodes to communicate with each other. A service is defined by a pair of messages, a request message, and a response message. The request message is sent by the client node to the server node and contains the data that the server needs to perform a specific task. The server node processes the request, performs the task, and sends back a response message to the client node.

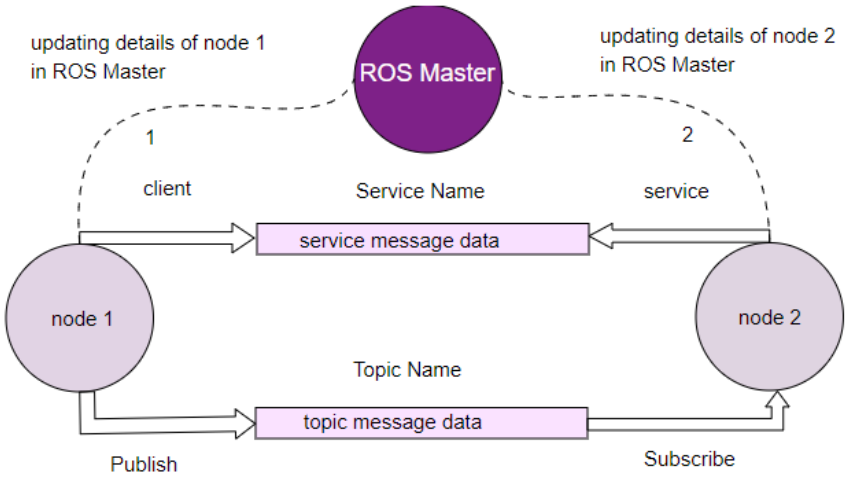


Figure 1.10: ROS Communication block diagram

1.4 conclusion

As mentioned in the chapter, mobile robots are designed for specific tasks or environments, and they may not be easily adaptable to new or changing situations. They often require reprogramming or hardware modifications to accommodate different tasks or operating conditions, which can be time-consuming and costly. Moreover, mobile robots rely on sensors to perceive and interact with their environment. However, these sensors have limitations. For example, vision sensors may struggle in low-light conditions or with occlusions, while range sensors like Global Positioning System (GPS) or ultrasonic sensors may have difficulty accurately perceiving certain materials or transparent objects. These limitations can affect the robot's ability to navigate and interact effectively in complex or challenging environments. To mitigate these drawbacks, mobile robots often employ a combination of localization techniques, including integrating depth camera with other sensors like Inertial measurement units (IMUs), odometry, computer vision, or SLAM algorithms to improve accuracy, robustness, and reliability. Furthermore, employing a robust framework ROS that offers modularity and collaboration enhances the development and control of robotic systems. This promotes code reuse, simplifies the development process, and fosters a collaborative environment.

Chapter 2

Autonomous Navigation System of Mobile Robot

2.1 Introduction

Mobile robots have gained significant prominence across diverse fields, including exploration, search and rescue operations, and transportation. Their ability to proficiently traverse unfamiliar terrains, evade obstacles, and precisely map their surroundings has become crucial. SLAM emerges as a valuable technique to attain these objectives, enabling robots to construct an environment map while concurrently establishing their own position within that environment.

There are different types of mobile robots designed to meet specific needs in various application fields like robot wheeled, tracked robots, Arial robots, and marine robots... Etc. In this chapter, we will touch on the model mathematics of mobile robots with two wheels, and will examine the different types of sensors used in mobile robots such as cameras and LIDAR, and will talk about the different applications of SLAM like, indoor and outdoor navigation, exploration underwater, and in agriculture ... etc. By the end of this chapter, we will explore how algorithms, such as hectorSLAM and Gmapping and loop closure, can be used in mobile robots to navigate and create a map of their environment in real-time.

2.2 Model mathematic of mobile robot

In order to carry out our practical study on mobile robot, it's necessary first discussing about mathematical model of a mobile robot with two wheels. A mathematical model is a representation of a real-world system using mathematical equations and principles. In the case of a mobile robot, the mathematical model describes its motion, dynamics, and control algorithms. By using mathematical modeling, we can simulate and predict the behavior of the robot under different conditions, optimize its performance. The mathematical model of a mobile robot typically involves concepts from kinematics, dynamics, control theory, and optimization.

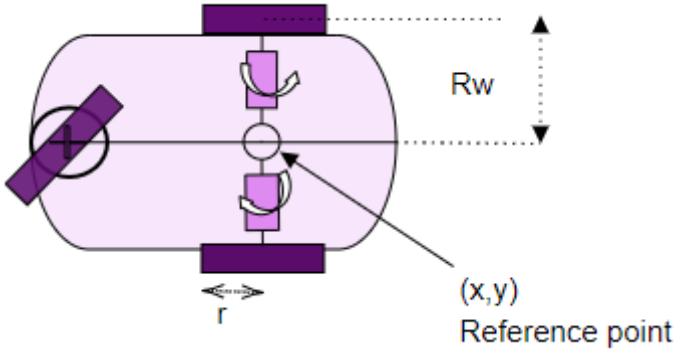


Figure 2.1: Chassis scheme of mobile robot with two wheeled.

We'll take look at how we actually model differential drive robots ,so that we can you actually compute odometry information .

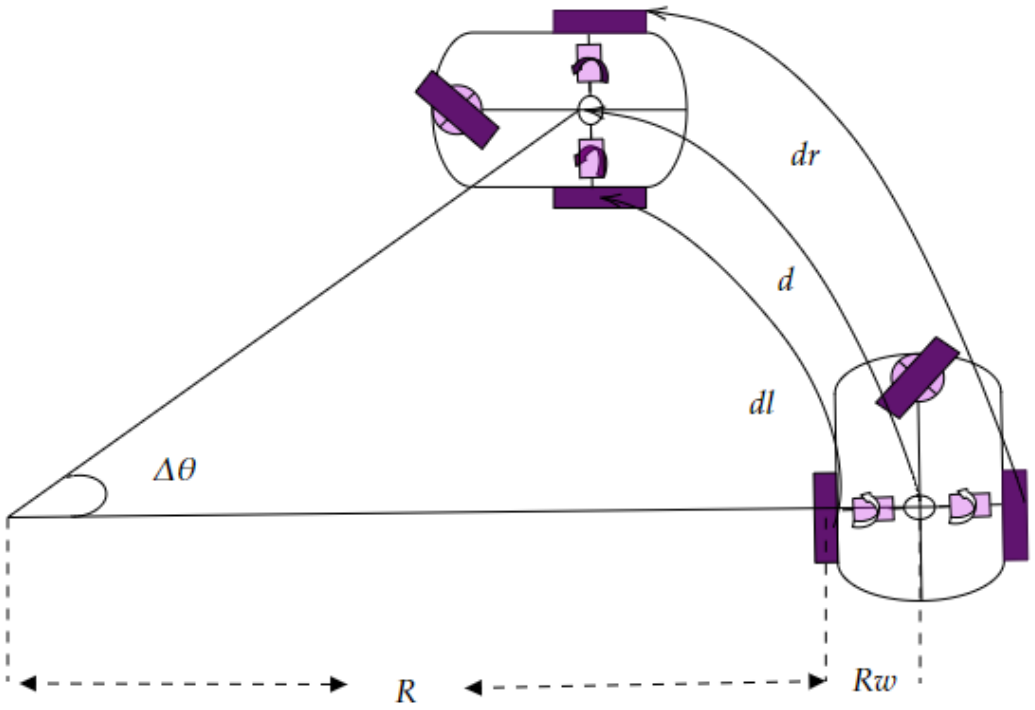


Figure 2.2: Distance between each wheel in the center of the robot.

mathematical model :

By referring to Figure 2.2, we can derive the subsequent equations:

$$\begin{aligned} d_l &= R\Delta\theta \\ d &= (R + R_w)\,\Delta\theta \\ d_r &= (R + 2R_w)\,\Delta\theta \end{aligned} \tag{2.1}$$

Where:

R_w: Distance between wheel and center

R: radius of curved path

$\Delta\theta$: change in orientation

d_l: distance travelled by left wheel

d_r: distance travelled by right wheel

d: distance travelled by robot

Usually, we don't know the change in orientation $\Delta\theta$ and distance travelled by robot d

mathematical model :

Let's solve for $\Delta\theta$ first:

$$\begin{aligned} d_l &= R\Delta\theta \\ d_r &= (R + 2R_w)\,\Delta\theta \\ \Delta\theta R_w &= \frac{d_r - R\Delta\theta}{2} \\ \Delta\theta &= \frac{d_r - d_l}{2R_w} \end{aligned} \tag{2.2}$$

solve for d:

$$\begin{aligned} d &= (R + R_w)\,\Delta\theta \\ d &= R\Delta\theta + R_w\Delta\theta \\ d &= d_l + \frac{d_r - d_l}{2} \\ d &= \frac{d_l}{2} + \frac{d_r}{2} \\ d &= \frac{d_l + d_r}{2} \end{aligned} \tag{2.3}$$

The inputs required for the motion of a mobile robot are the linear velocity (V) and orientation θ ,The rate of change of position of robot along the x-direction is $\dot{x}$ and that in the y-direction is $\dot{y}$ are stated by.

mathematical model :

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \end{aligned} \tag{2.4}$$

Known:

v_l : left wheel velocity

v_r : right wheel velocity

r : wheel radius

The velocity V of the robot in a fixed reference coordinate given by

$$v = \frac{v_l + v_r}{2} \tag{2.5}$$

and the angular velocity of the robot is given by

$$\dot{\theta} = \frac{r}{2R_w}(v_r - v_l) \tag{2.6}$$

Two wheels for mobile robot work together. When the velocity of left wheel is less than the velocity of right wheel, the mobile robot takes left turn. Similarly, when velocity of right wheel is less than the velocity of left wheel, the mobile robot takes right turn. With the equal velocities of the two wheels the mobile robot moves straight.

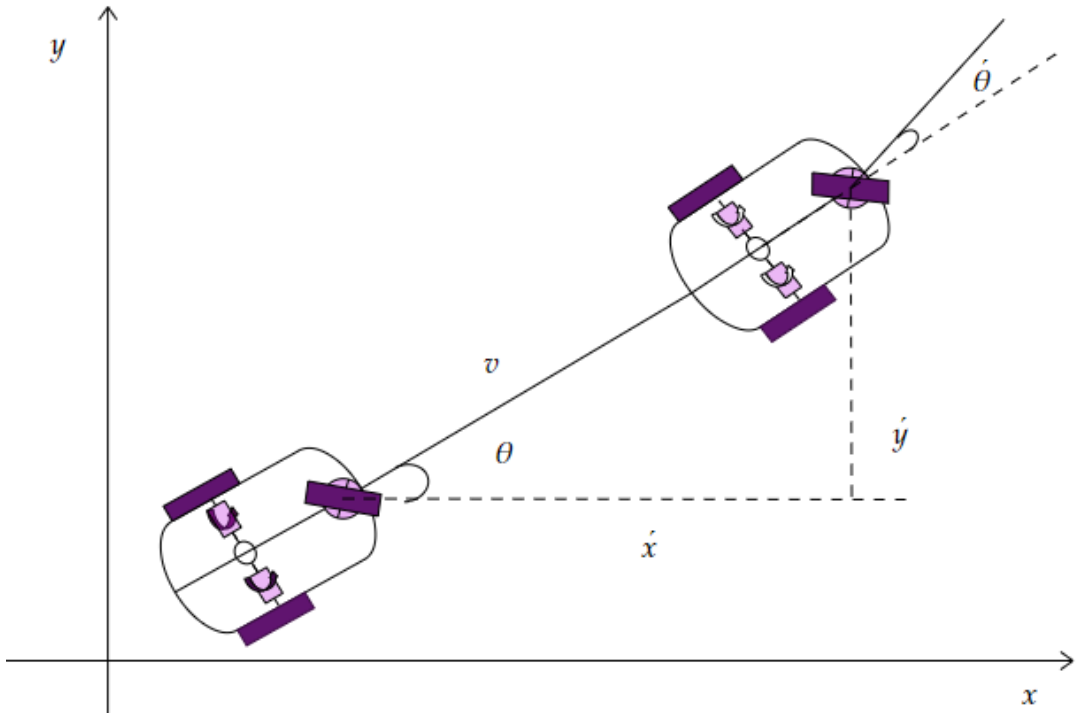


Figure 2.3: Linear and angular speeds.

mathematical model :

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \frac{r(v_r - v_l)}{2R_w} \end{bmatrix} = \begin{bmatrix} \frac{(v_l + v_r)\cos\theta}{2} \\ \frac{(v_l + v_r)\sin\theta}{2} \\ \frac{r(v_r - v_l)}{2R_w} \end{bmatrix} \tag{2.7}$$

2.3 Sensors (Lidar, cameras)

Sensors, such as LIDAR and cameras play a vital role in mobile robotics by providing essential information about the robot's surroundings. LIDAR sensors use laser beams to measure distances and create detailed 2D maps of the environment, while cameras capture visual data for object detection and recognition. These sensors enable the robot to perceive its surroundings, avoid obstacles, and navigate safely. Moreover, LIDAR and cameras are essential for SLAM techniques, allowing the robot to build accurate maps of its environment and determine its own position within the map.

Figure 2.4: Sensors (depth camera)

2.4 Simultaneous Localization and Mapping :

2.4.1 Definition:

Simultaneous Localization and Mapping (SLAM) is a technique used in robotics to enable a mobile robot to navigate and create a map of its environment in real-time. It involves the simultaneous estimation of the robot's position (localization) and the construction of an accurate representation of the environment (mapping). SLAM integrates sensor data from cameras, LIDARs, with motion estimation algorithms to iteratively update the robot's position and orientation while constructing the map. By utilizing SLAM, robots can autonomously operate in unknown or dynamic environments, making it a vital technology for navigation and mapping applications.

2.4.2 SLAM application:

Indoor and Outdoor Navigation :

SLAM enables mapping and navigation functionalities in both outdoor and indoor applications, with outdoor applications SLAM can be applied to navigation systems like Google Maps or GPS-based navigation devices. SLAM algorithms can help build and update road maps by integrating GPS data, while indoor applications SLAM can be utilized for mapping and navigation purposes. For example, a vacuum robot can use SLAM to construct a 2D map of the indoor space it operates in. The robot captures sensor data, such as lidar to build the map and determine its own location within it. This allows the robot to navigate efficiently, clean unexplored areas, and avoid obstacles.

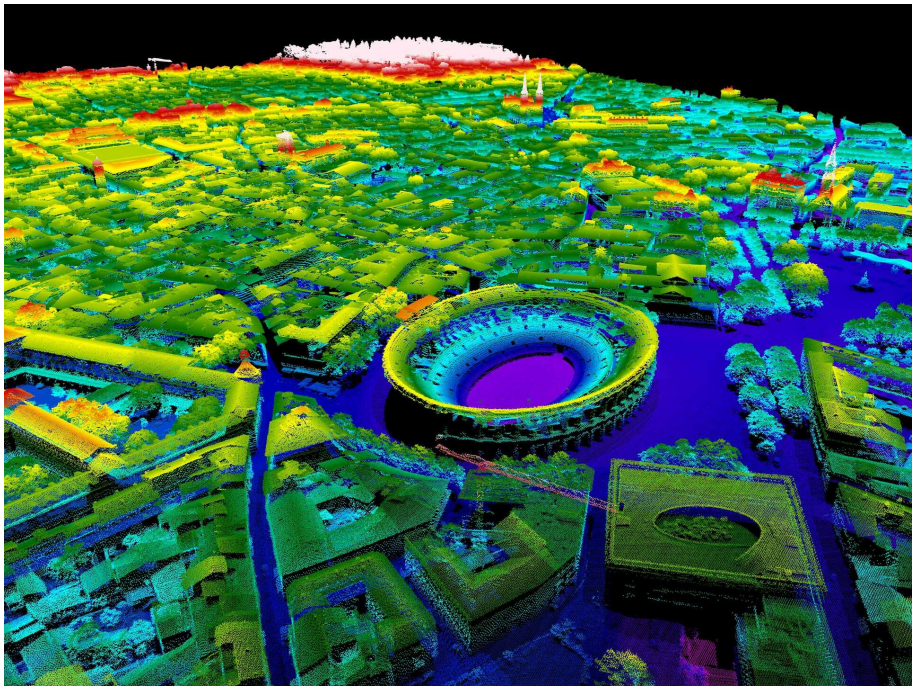


Figure 2.5: SLAM in Outdoor Application.

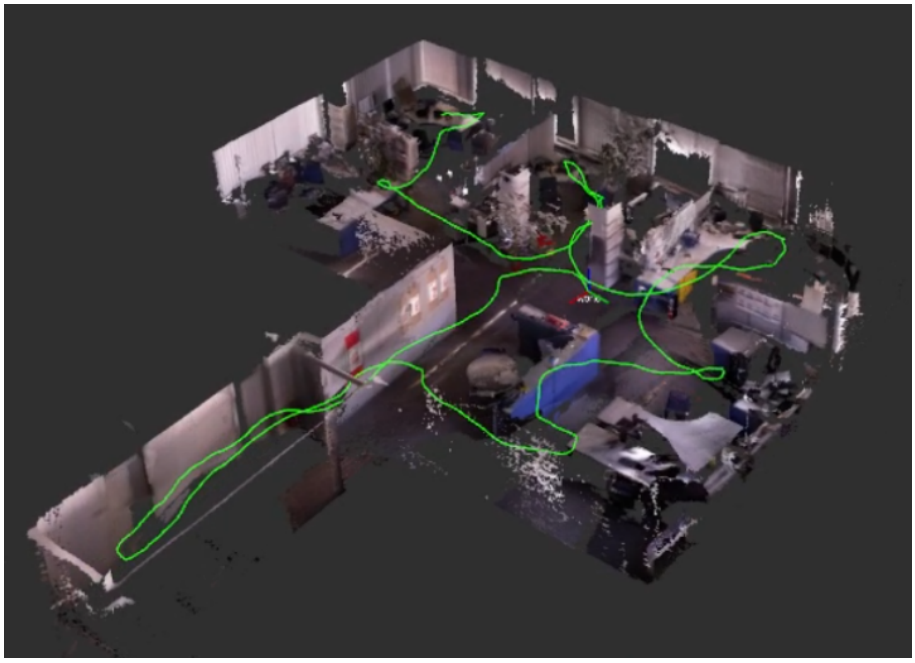


Figure 2.6: SLAM in indoor Application.

Underwater Exploration

SLAM can be used in underwater applications as well. due to the high pressure, the human can't get deeper under the water, the widely known application can be found in the Autonomous underwater vehicles (AUVs) that use SLAM to scan the underwater terrain and collect data to give a 3D map and gives its current altitude under the water[3].

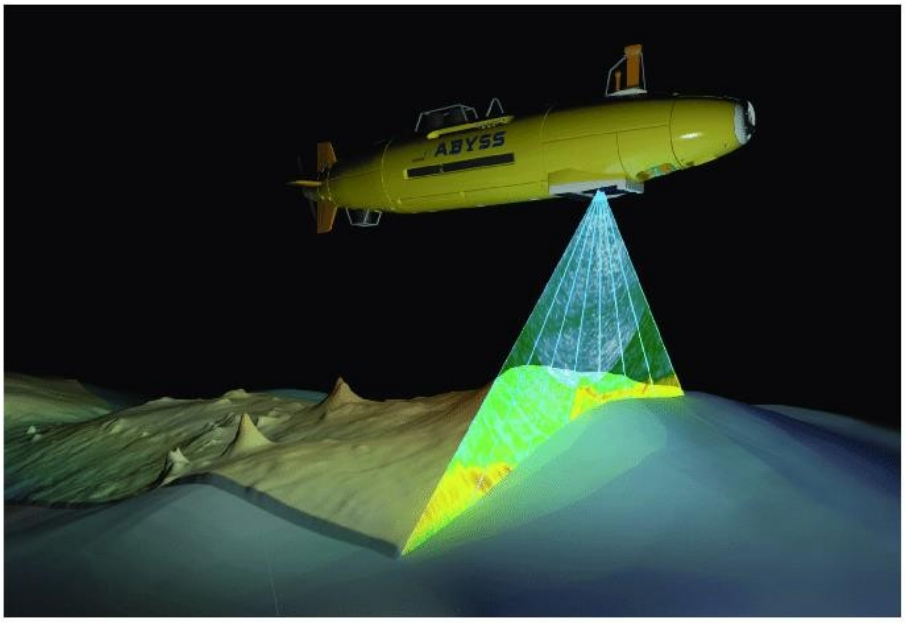


Figure 2.7: SLAM in Underwater application.

Agriculture Technology

SLAM technology has transformative applications in agriculture, revolutionizing various aspects of farming. It enables detailed crop monitoring by creating maps of fields, allowing farmers to assess crop health and growth stages. Autonomous farming vehicles equipped with SLAM capabilities can navigate fields, optimize paths, and avoid obstacles, improving efficiency and productivity. Soil mapping and analysis using SLAM algorithms provide valuable insights into soil characteristics, optimizing irrigation and fertilizer usage.

Figure 2.8: Aerial Drones for Agriculture.

Spatial Exploration:

SLAM (Simultaneous Localization and Mapping) technology finds valuable applications in space exploration. It enables autonomous navigation for robots and rovers, allowing them to explore planetary surfaces with precision. By utilizing sensor data and SLAM algorithms, detailed maps of the terrain and geological features can be created. These maps aid in path planning and localization, ensuring accurate navigation and safe maneuvering around obstacles.

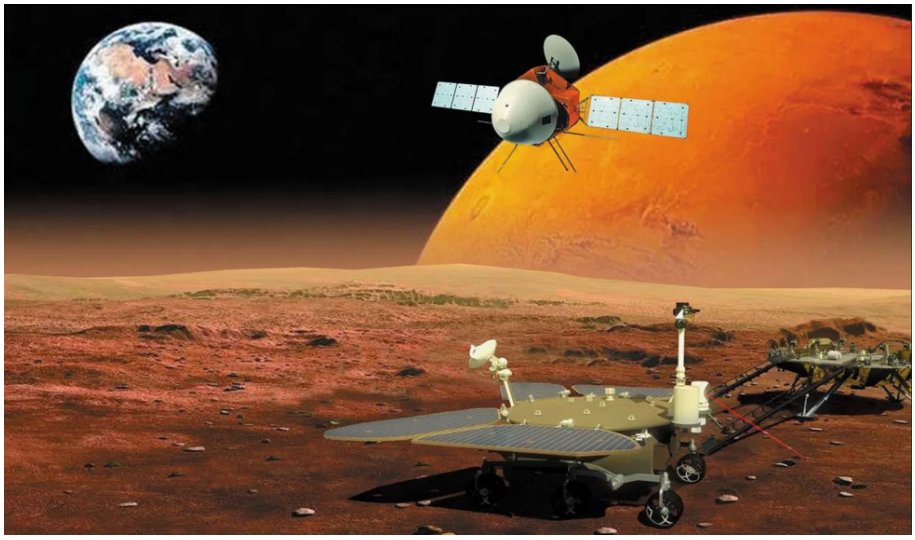


Figure 2.9: Robots for planetary exploration.

2.4.3 SLAM techniques:

The mobile robot (two wheels) needs to know its environment (localization) and must be able to navigate without colliding with either dynamic or static obstacles in the surroundings (mapping). This is accomplished with the help of SLAM algorithms, For this project, the most popular SLAM algorithm we are using is (GMapping and Hector SLAM).

Hector SLAM:

Hector SLAM is an algorithm used for building a 2D grid map for the surrounding environment based on a laser scan sensor (LIDAR), which work by employing an occupancy grid map representation, where each cell in the grid represents the probability of occupancy.
Hector SLAM divides the environment into a grid and incrementally builds the map by combining the current laser scans with previous scans and estimated robot poses. It applies scan matching techniques to align consecutive scans and estimate the robot's movement between them.

a. Occupancy grid mapping

Hector SLAM employs an occupancy grid mapping technique to construct a map of the environment. The occupancy grid divides the environment into a grid of cells, where each cell represents a small portion of the area. Initially, all cells are considered unknown. As the robot moves and collects sensor data, Hector SLAM updates the occupancy grid by marking cells as occupied or free based on the sensor measurements and the estimated robot pose.

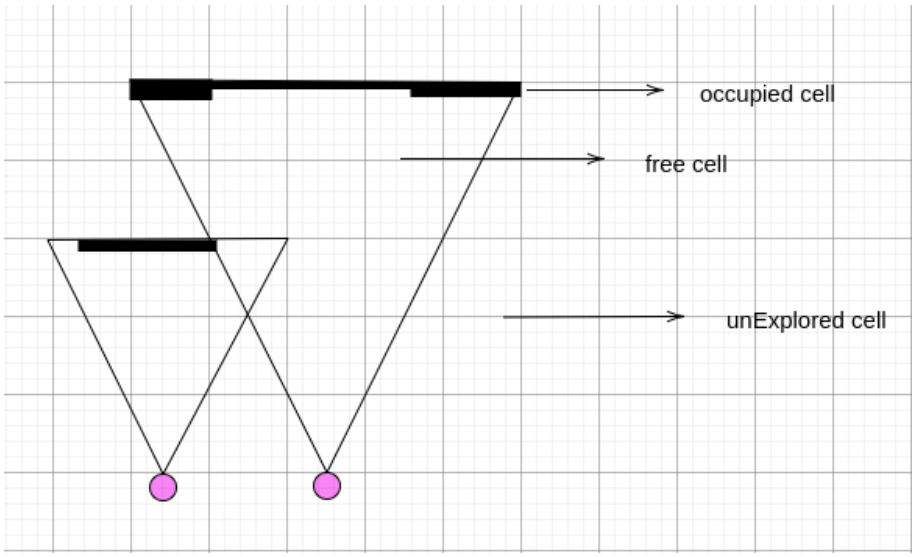


Figure 2.10: Occupancy grid mapping .

Each cell is a binary random variable that models the occupancy
Cell is occupied: $p(m_i) = 1$
Cell is not occupied: $p(m_i) = 0$

b. Scan Matching

The optimization process in scan matching aims to find the best alignment of beam endpoints with the map. It seeks to minimize the discrepancies between the observed laser measurements and the expected measurements based on the current pose estimate. By iteratively adjusting the incremental motion $\Delta\boldsymbol{\xi}$ using an optimization algorithm, such as Gauss-Newton, the algorithm refines the alignment and improves the accuracy of the estimated pose.
Laser scanner utilizes the transformed maps with the relation $\boldsymbol{\xi} = (p_x, p_y, \psi)^{\mathrm{T}}$ [4]. to extract the minimal error value $\boldsymbol{\xi}^*$ into the estimative end-points set by 2.8:

Robot position :

$$\boldsymbol{\xi}^* = \underset{\boldsymbol{\xi}}{\operatorname{argmin}} \sum_{i=1}^{n} \left[1 - M\left(\mathbf{S}_i(\boldsymbol{\xi})\right)\right]^2 \tag{2.8}$$

where $\boldsymbol{\xi}^*$ is the minimal value through the operation of arg min. The $S_i(\boldsymbol{\xi})$ is settled as the approximated function of $\boldsymbol{\xi}$ to reach the best position at the ray end of $S_i \left(S_{i,x}, S_{i,y}\right)^{\mathrm{T}}$ in the world coordinates spaces [5].

Therefore, robot position is described by formulas 2.9 of the world coordinates spaces:

$$\mathbf{S}_i(\boldsymbol{\xi}) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{pmatrix} \begin{pmatrix} s_{i,x} \\ s_{i,y} \end{pmatrix} + \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{2.9}$$

The function $M\left(\mathbf{S}_i(\boldsymbol{\xi})\right)$ returns the map value at the coordinates given by $\mathbf{S}_i(\boldsymbol{\xi})$. Given some starting estimate of $\boldsymbol{\xi}$, we want to estimate $\Delta\boldsymbol{\xi}$ which optimizes the error measure according to [5]

Taylor expansion :

$$\sum_{i=1}^{n} \left[1 - M\left(\mathbf{S}_i(\boldsymbol{\xi} + \Delta\boldsymbol{\xi})\right)\right]^2 \to 0 \tag{2.10}$$

In this study, the first order of Taylor series is selected and expanded its value at $M\left(\mathbf{S}_i(\boldsymbol{\xi} + \Delta\boldsymbol{\xi})\right)$ by the following equation[4]:

$$\sum_{i=1}^{n} \left[1 - M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) - \nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \Delta\boldsymbol{\xi}\right)\right]^2 \to 0 \tag{2.11}$$

The partial differentiation is directly taken into the previous formulas to approximate the desired zero purpose.

$$\begin{aligned} 2 \sum_{i=1}^{n} & \left[\nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right]^T \\ & \left[1 - M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) - \nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \Delta\boldsymbol{\xi}\right)\right] = 0 \end{aligned} \tag{2.12}$$

The Gauss-network approximates the minimal function of $\Delta\boldsymbol{\xi}$ by the formula 2.13:

The Gauss-network approximates :

$$\Delta\boldsymbol{\xi} = \mathbf{H}^{-1} \sum^{n} \left[\nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \right]^T \left[1 - M\left(\mathbf{S}_i(\boldsymbol{\xi})\right)\right] \tag{2.13}$$

Hessian matrix H is defined by 2.14

$$\mathbf{H} = \left[\nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \right]^T \left[\nabla M\left(\mathbf{S}_i(\boldsymbol{\xi})\right) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \right] \tag{2.14}$$

GMAPPING:

Gmapping is an algorithm commonly used for simultaneous localization and mapping (SLAM) in mobile robotics. It is designed to enable a robot to navigate and build a map of its environment simultaneously, The main objective of Gmapping is to use the Rao-Blackwellized particle filter to estimate the robot's pose (position and orientation) in an unknown environment while constructing a map of the surroundings.

a. RAO-BLACKWELLIZED MAPPING

The key idea of the Rao-Blackwellized particle filter for SLAM is to estimate a posterior $p\left(x_{1:t} \mid z_{1:t}, u_{0:t}\right)$ about potential trajectories $x_{1:t}$ of the robot given its observations $z_{1:t}$ and its odometry measurements $u_{0:t}$ and to use this posterior to compute a posterior over maps and trajectories [6]:

The posterior probability :

$$p\left(x_{1:t}, m \mid z_{1:t}, u_{0:t}\right) = p\left(m \mid x_{1:t}, z_{1:t}\right) p\left(x_{1:t} \mid z_{1:t}, u_{0:t}\right). \tag{2.15}$$

where:

$p\left(x_{1:t}, m \mid z_{1:t}, u_{0:t}\right)$: the posterior probability of the robot's pose trajectory and the map.

$p\left(m \mid x_{1:t}, z_{1:t}\right)$: the conditional probability of the map given the pose trajectory and measurements.

$p\left(x_{1:t} \mid z_{1:t}, u_{0:t}\right)$: the conditional probability of the pose trajectory given the measurements and control inputs .

This equation forms the basis for estimating the robot's pose trajectory and mapping in SLAM algorithms.

One of the most common particle filtering algorithms is the Sampling Importance Resampling (SIR) filter, which follows a sequential process of importance sampling, resampling, and Map Estimating. It estimates the posterior distribution by sampling particles from a proposal distribution based on the importance weights. Then it resamples particles according to their weights to generate a new set of particles for the next time step.
Here's a brief overview of the steps involved in the SIR filter:

- Sampling: In the sampling step of particle filtering, new particles for the current time step are generated from the existing particles from the previous time step. Each particle represents a possible state of the system. This is done by sampling from a proposal distribution, $\pi\left(x_t \mid z_{1:t}, u_{0:t}\right)$., which represents our best estimate of the true distribution of the state given the available measurements and control inputs up to the current time step.

- Importance Weighting: Once the particles are generated, each particle is assigned an importance weight, denoted as w(i), based on its likelihood given the available measurements and control inputs. The weight is calculated by evaluating the posterior probability of the particle's state,$p\left(x_t^{(i)} \mid z_{1:t}, u_{0:t}\right)$, and scaling it by the proposal distribution used for sampling, $\pi\left(x_t^{(i)} \mid z_{1:t}, u_{0:t}\right)$ [6]. The importance weight represents the relative importance or likelihood of each particle being the true state of the system. These weights play a crucial role in subsequent steps, such as re-sampling, as they determine the probability of particles being selected for the next iteration based on their importance.

$$w^{(i)} = \frac{p\left(x_t^{(i)} \mid z_{1:t}, u_{0:t}\right)}{\pi\left(x_t^{(i)} \mid z_{1:t}, u_{0:t}\right)}$$

- Re-sampling: After the weights are assigned to the particles, a re-sampling step is performed to select a new set of particles for the next time step. Re-sampling is done with replacement, where particles with higher weights have a higher chance of being selected, while particles with lower weights are less likely to be chosen This step helps to maintain diversity among the particles and ensures that particles representing the most probable states are retained.

- Map estimating: Map estimation typically involves using the particles to update the map representation based on sensor measurements. The specific method for map estimation can vary depending on the algorithm and the mapping technique used (e.g.,

grid-based mapping,scan matching).

Loop closure

In SLAM , loop closure refers to the process of detecting and closing loops in a robot's trajectory or map. It is a very important step in SLAM algorithms to improve the accuracy and consistency of the estimated robot pose and map. When a robot explores an environment, it collects sensor measurements, to estimate its pose and create a map of the surroundings, errors can accumulate over time due to sensor noise, These errors can cause the estimated trajectory or map to deviate from the true values.
Loop closure aims to correct these errors by recognizing previously visited locations when the robot revisits them. This is done by comparing the current sensor measurements with the previously stored data, If a loop closure is detected, the estimated trajectory and map can be adjusted to minimize the accumulated errors and align the newly observed data with the existing map. as shown in the following figure 2.11
The process of closing the loop typically consists of the following steps:

- **Loop detection:** The algorithm analyzes the robot's sensor data or odometry information to identify potential loop closures. This can be done by comparing the current sensor data with previously recorded sensor data to find similar samples or landmarks.
- **Loop Validation:** Once potential loop closures are detected, the algorithm evaluates the likelihood of each potential closure.
- **loop correction :**When a loop closure is detected and validated, the loop correction step aims to align the newly observed data with the existing map and refine the robot's estimated pose.

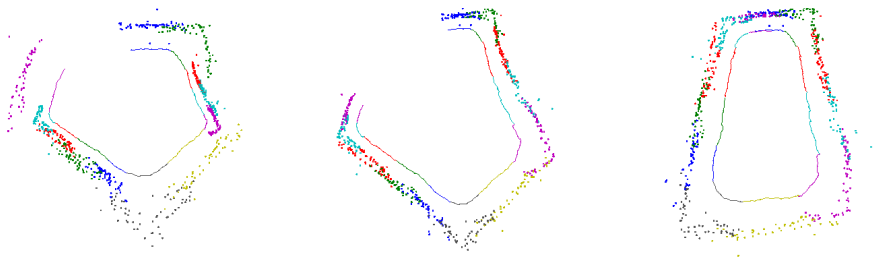


Figure 2.11: Loop closing steps

2.5 conclusion

In this chapter, we discussed about the mathematics model of mobile robots with two wheels and the two sensors commonly used (cameras and LIDAR). In addition, we touched upon the applications of SLAM, and we learned about algorithms of SLAM such as hector slam and Gmapping, which permeate the mobile robot to navigate and create a map of their environment.

Hence, in the next chapter, we will delve into the process of building a robot using different components such as wheels, motors, sensors, motor driver, and the Jetson Nano card... etc. We will explore how to install JetPack and ROS on the Jetson Nano card, which will serve as the foundation for our robot's capabilities. Moreover, we will focus on utilizing Hector SLAM and Gmapping algorithms to construct maps with our mobile robot. This will involve understanding these algorithms and implementing them effectively for real-time mapping.

Chapter 3

Build robot and framework configuration

3.1 Introduction

In this chapter, we will explore the essential components required to build our mobile robot and delve into the process of launching the Hector SLAM (Simultaneous Localization and Mapping) algorithm to generate a map based on the LiDAR sensor. Our robot requires specific components such as motors, wheels, sensors, and Nvidia jetson nano embedded card to function effectively. Additionally, we will focus on the LiDAR sensor, which utilizes laser beams to measure distances and create a detailed 2D map of the robot's surroundings. By integrating the Hector SLAM algorithm, the robot can simultaneously localize itself within the environment while constructing an accurate map. Through this chapter, we will gain insights into the key components and processes necessary for building the mobile robot and using LiDAR and Hector SLAM algorithm to achieve comprehensive mapping capabilities.

3.2 Robot Components and System Framework

The robot comes equipped with a range of components carefully selected to provide an immersive learning experience. These components are presented in the following table 3.1 [7]:

Part	Qty
Circular Robotics Chassis Kit (Two-Layer)	1
Lithium Ion Battery Pack - 10Ah (3A/1A USB Ports)	1
Ball Caster Metal - 3/8”	1
Edimax 2-in-1 WiFi and Bluetooth 4.0 Adapter	1
Header - male - PTH - 40 pin – straight	1
2 in - 22 gauge solid core hookup wire (red)	1
Shadow Chassis Motor (pair)	1
Jetson Dev Kit (Optional)	1
SparkFun JetBot Acrylic Mounting Plate	1
SparkFun Jetbot image (Pre Flashed)	1
Leopard Imaging 145 FOV Camera	1
Screw Terminals 2.54mm Pitch (2-Pin)	2
SparkFun Micro OLED Breakout (Qwiic)	1
SparkFun microB USB Breakout	1
SparkFun Serial Controlled Motor Driver	1
Breadboard Mini Self-Adhesive Red	1
SparkFun Qwiic HAT for Raspberry Pi	1
SparkFun JetBot Acrylic sidewall for camera mount	2
SparkFun JetBot Acrylic Camera mount & 4x nylon mounting hardware	1
Qwiic Cable - 100mm	1
Qwiic Cable - Female Jumper (4-pin)	1
Wheels & Tires - included as part of circular robotics chassis	2
USB Micro-B Cable - 6”	2
Dual Lock Velcro	1
RP Lidar a1	1

Table 3.1: The different components used for assembly

3.3 NVIDIA JETSON NANO kit

3.3.1 Introduction

The NVIDIA Jetson Nano Developer Kit is the compute performance to run modern Artificial intelligence (AI) workloads at unprecedented size, power, and cost. It can be used for AI frameworks and model applications like image classification, object detection, segmentation, and speech processing. The developer kit can be powered by micro-USB and comes with extensive I/Os, This makes it simple for developers to connect a diverse set of new sensors to enable a variety of AI applications. It's incredibly power-efficient, consuming as little as 5 watts. Jetson Nano is also supported by software libraries for deep learning, computer vision, GPU computing, multimedia processing, and much more. The software is even available using an easy-to-flash SD card image, making it fast and easy to get started. make it shorter

Figure 3.1: The NVIDIA Jetson Nano Developer Kit.

3.3.2 Key features

The NVIDIA Jetson Nano Developer Kit is a powerful and compact AI development platform that enables developers, researchers, and hobbyists to create and deploy AI applications and projects. It combines the performance of an NVIDIA GPU with a low-power ARM processor, making it suitable for edge computing and AI tasks on embedded systems. Here are the key features of the NVIDIA Jetson Nano Developer Kit:

Jetson Nano Module :

- 128-Core NVIDIA Maxwell™ GPU
- Quad-Core ARM® A57 CPU
- 10/100/1000BASE-T Ethernet

Power Options :

- Micro-USB 5V 2A
- DC Power Adapter 5V 4A

I/O :

- USB 3.0 Type A
- USB 2.0 Micro-B
- HDMI/DisplayPort
- M.2 Key E
- Gigabit Ethernet
- GPIOs, I2C, I2S, SPI, UART
- MIPI-CSI Camera Connector
- Fan Connector
- PoE Connector

Kit Contents :

- NVIDIA Jetson Nano Module with Heatsink and Reference Carrier Board
- Quick Start Guide and Support Guide

3.4 The Serial Controlled Motor Driver

3.4.1 Introduction

The Serial Controlled Motor Driver (SCMD) is a specialized driver for small DC motors, offering convenient control options. It supports UART, I2C, or SPI communication protocols and is capable of driving small DC motors with a continuous load of 1.2A per motor (up to 1.5A peak) at 11V. If you require additional motors, multiple SCMDs can be connected together and controlled through a single serial interface. Moreover, by bridging the output of each board, you can double the current capacity for increased power requirements.

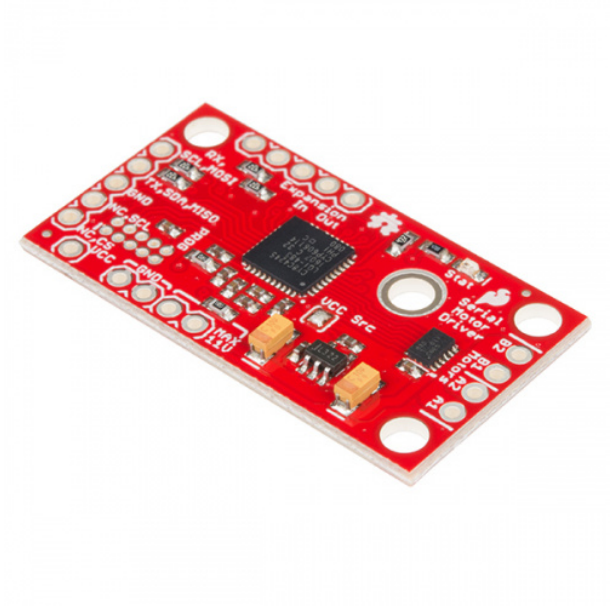


Figure 3.2: The completed Serial Controlled motor drive.

3.4.2 Motor Driver Assembly and Configuration

- The 2-pin screw terminals are soldered to the "Motor Connections."
- Break off 4 Male Plated Through-Hole (PTH) straight headers and solder into the "Power (VIN) connection" points.
- Break off 5 Male PTH straight headers and solder into the "Expansion port" points. These will not be used but will provide additional board stability when installed into the mini breadboard.
- Break off 5 Male PTH straight headers and solder into the "User port" points for connection into the included Female Jumper Qwiic cables.

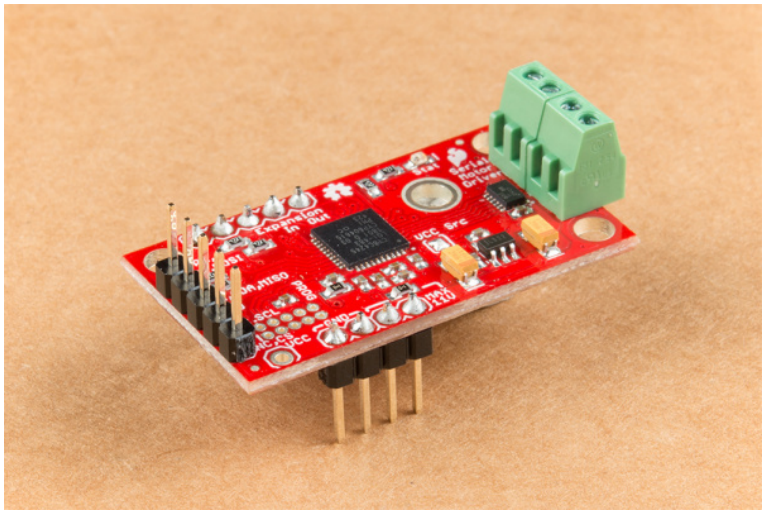


Figure 3.3: The Serial Controlled Motor Driver.

3.4.3 The assembly of the micro-USB breakout and Serial Controlled Motor Driver on the breadboard

VCC to PWR:

The red wire connects Vcc(5” ” V) from the microUSB jack to the PWR input for the SCMD.

GND to GND:

Ground loop or connect the ground reference between the two devices.

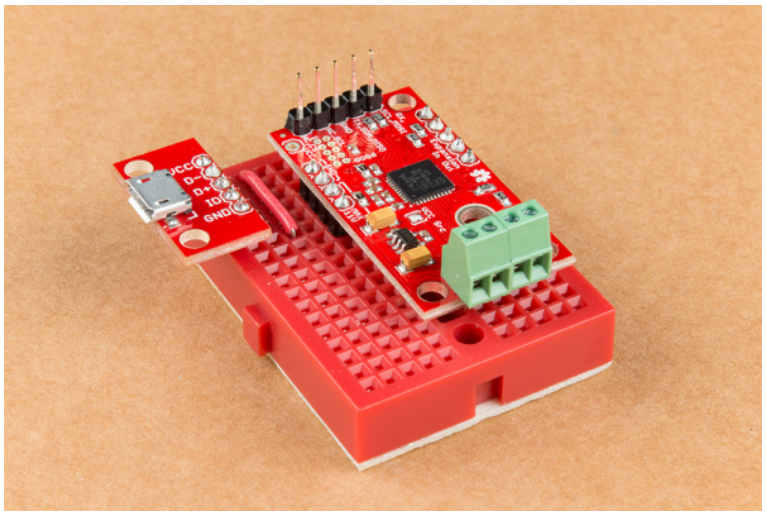


Figure 3.4: The assembly of the micro-USB breakout and SCMD on the breadboard.

3.5 Data Acquisition

3.5.1 RPLIDAR A1

RPLIDAR A1 360-degree Laser Scanning Range finder is a low-cost 2D laser radar (LIDAR) solution developed by SLAMTEC Corporation with sensor resolution up to millimeters. The system can perform 360-degree scans within a 12-meter range. and produces flat point cloud map information of the space in which it is located. This flat-point cloud information can be used in practical applications such as map mapping, robot positioning and navigation, object and environment modeling. With the sampling period set to 360 samples per week, the RPLIDAR sweep frequency reaches 5.5hz and sweeps up to 10hz.[8]

Figure 3.5: LIDAR 360 degree 2D laser scanner.

```
header:
  seq: 365
  stamp:
    secs: 1685437062
    nsecs:  87334061
  frame_id: "laser"
angle_min: -3.12413907051
angle_max: 3.14159274101
angle_increment: 0.00871450919658
time_increment: 0.000183839176316
scan_time: 0.13218036294
range_min: 0.15000000596
range_max: 12.0
```

Figure 3.6: Key characteristics of RPLidar a1.

Mechanism

It has the capability to measure distance data more than 8000 times per second, providing highly accurate distance outputs with less than 1% error. By emitting a modulated infrared laser signal, RPLIDAR detects objects by analyzing the reflected laser signal. The vision acquisition system in RPLIDAR A1 samples the returning signal, while the embedded Digital Signal Processor (DSP) processes the data to calculate the distance and angle values between the object and RPLIDAR A1. This information is then communicated through the interface. The RPLIDAR system is designed to be mounted on a spinning rotator with an angular encoding system, enabling it to perform a full 360-degree scan of the surrounding environment during rotation.[9]

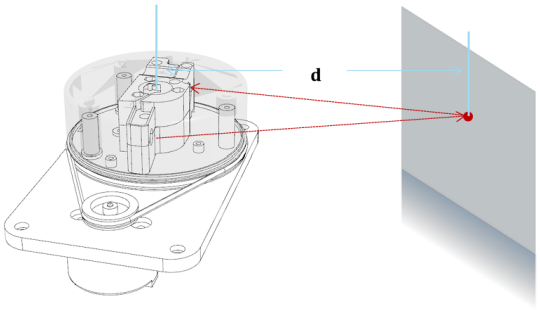


Figure 3.7: The RPLIDAR A1 Working Schematic .

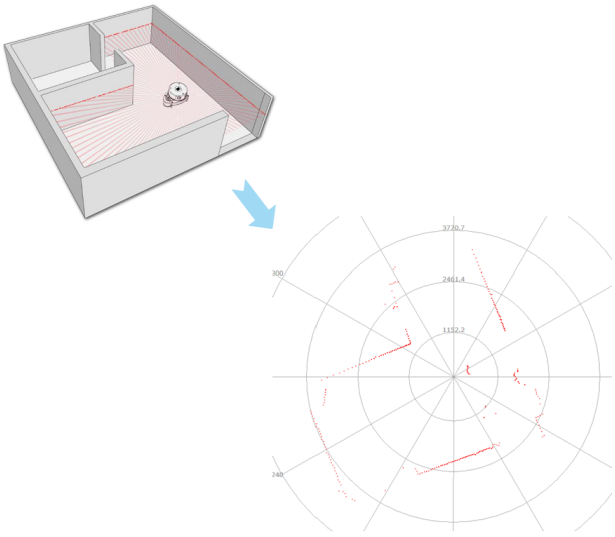


Figure 3.8: The Obtained Environment Map from RPLIDAR A1 Scanning

Theoretical Framework

the principle of laser triangulation is a typical alternative to Time of-Flight (ToF) measurement in low-cost laser sensors. A lens with a focal distance d', placed at a distance b from the laser source, localizes the returned ray on a CCD (CMOS)-based Position Sensitive Device (PSD) at distance d' from the lens. The triangles defined by (b, d) and (b', d') are similar, and the distance to the object is non-linearly proportional to the angle of the reflected light (i.e., the laser acceptance angle). The perpendicular distance to the object (d) is given by:

$$d = \frac{bd'}{b'}$$

if β is the angle of the laser beam, the slant range (distance along the geometric ray) is:

$$D = \frac{d}{\sin \beta}$$

From (2) and (3), the range sensitivity (or resolution) $\Delta d/\Delta b'$ is equal to:

$$\frac{\Delta d}{\Delta b'} = -\frac{d^2}{bd'}$$

and grows quadratically with the distance from the object. Small values of bd' allow measurement of small distances, whereas the sensor resolution is enhanced by large bd'.[10]

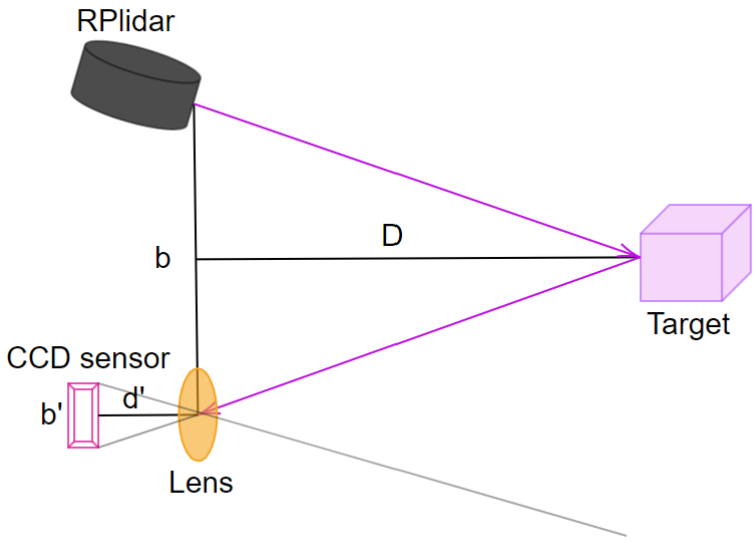


Figure 3.9: Laser triangulation

3.5.2 The Leopard Imaging 145 FOV Camera

The Leopard Imaging 145 Field of View (FOV) Camera is a cutting-edge imaging device that offers a wide FOV for capturing a broader visual perspective. Produced by Leopard Imaging Inc. this camera is designed specifically for compatibility with the NVIDIA Jetson Nano Developer Kit. It combines advanced optics and image sensor technology to deliver high-quality and immersive imaging experiences. With its 145-degree FOV, the camera can capture a wide-angle view, allowing for more comprehensive coverage of the surrounding environment. Whether used in robotics, surveillance systems, autonomous vehicles, or other applications[11].

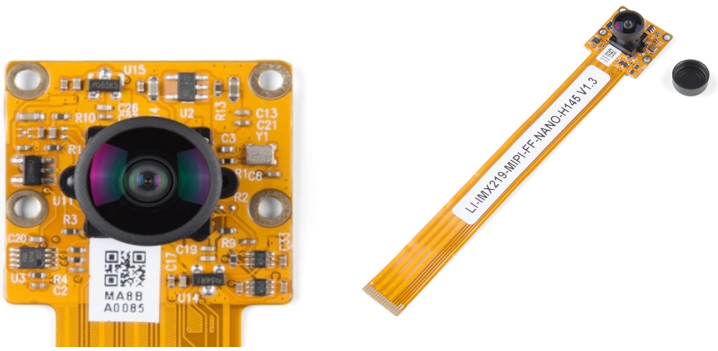


Figure 3.10: The Leopard Imaging 145 FOV Camera.

3.5.3 The Fully Assembled Robot

The figure 3.11 illustrates the final result of assembling the components of the robot. It shows a complete robot with all its parts put together. with sensors to sense its surroundings and wheels for smooth movement. It also has a powerful onboard computer (the Jetson NANO embedded card) and a camera module combined with RPlidar a1 for depth and visual analysis. The figure represents the culmination of hard work, resulting in a smart and capable robot that is ready to explore and discover new things.

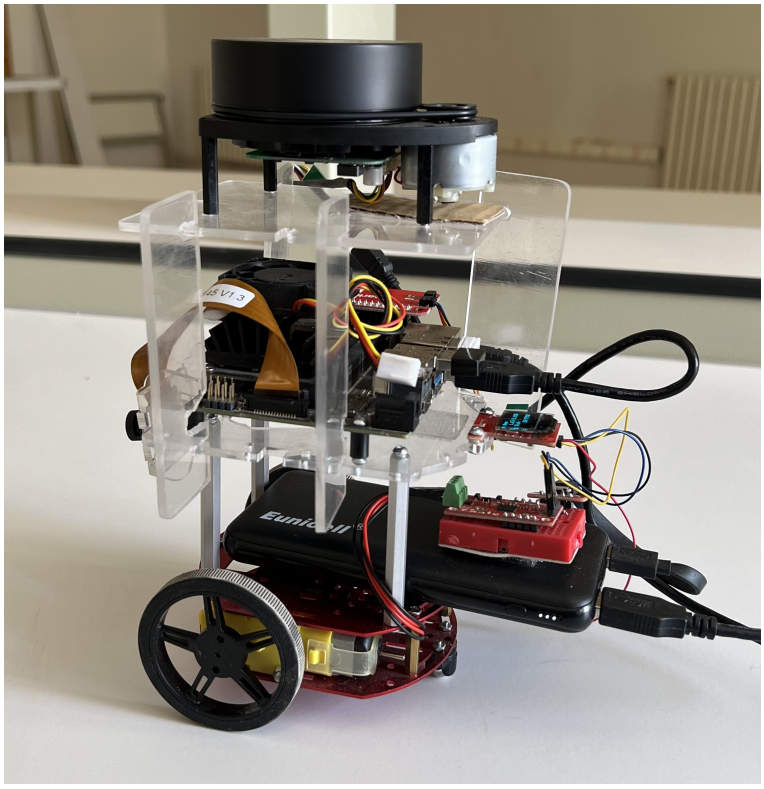


Figure 3.11: The completed robot

3.5.4 Table of Costs:

The provided table displays the expenses associated with the primary elements.

The component	The cost
Lithium Ion Battery Pack - 10Ah (3A/1A USB Ports)	$21.50
NVIDIA Jetson Nano Developer Kit (V3)	$149.00
Leopard Imaging Camera - 136 Degree FOV	$32.50
SparkFun Qwiic pHAT v2.0 for Raspberry Pi	$6.95
SparkFun (PID 13911) Serial Controlled Motor Driver	$37.47
Slamtec RPLIDAR A1M8	$139.79
SparkFun Micro OLED Breakout (Qwiic)	$18.50
Hobby Gearmotor - 140 RPM (Pair)	$5.50

Table 3.2: Table of Costs

3.6 Preparation of the Jetson Nano developer kit

3.6.1 installing JetPack

- First of all, we must download the Jetson Nano Developer Kit SD Card Image and extract the files.

Figure 3.12: The Jetson Nano Developer Kit SD Card Image

-Then, we Format the microSD card using 'SD Card Formatter' from the SD Association [12].

Figure 3.13: SD Card Formatter

-After that, we download "Etcher" to be able to write the Jetson Nano Developer Kit SD Card Image to the microSD card [13].

Figure 3.14: Balena Etcher application

After some time, the microSD card is prepared, and we can now proceed with the setup of the developer kit.

3.6.2 Dev Kit Setup and First Boot

1. We Insert the microSD card (with the system image already written on it) in ① and we Connect the USB keyboard and mouse via ② and use ③for the screen in order to be able to operate on the card .
Now, before powering the card using ④ the most important thing that we have to consider is :

- If we're using a Micro-USB power supply rated at a voltage of 5V and 2A current remove the J48 jumper ⑤.
- If we're using a DC barrel jack power supply rated at a voltage of 5V and 4A current, keep the J48 jumper ⑤ connected.

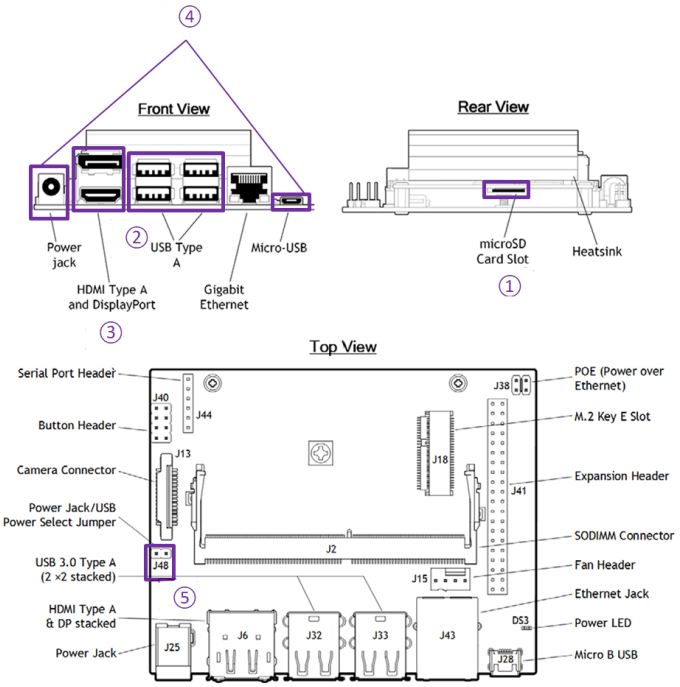


Figure 3.15: The interfaces of the jetson nano

2. Upon powering on the developer kit, a green LED located near the Micro-USB connector will illuminate. Subsequently, the Jetpack will guide us through additional configuration

steps during the initial boot. The configuration process is menu-based and relatively simple. We will be prompted to review and agree to the NVIDIA Jetson software EULA, select our preferred language, accept the license agreement, choose a time zone, set up the keyboard layout, and perform other standard system-setup tasks.

3. As we know, OpenCV is a widely used open-source computer vision library that provides a collection of algorithms and functions for image and video processing, object detection, and more. CUDA, is a parallel computing platform developed by NVIDIA for utilizing the power of GPUs but The pre-installed OpenCV library on Jetpack does not include the CUDA accelerated versions of the OpenCV algorithms.so in order to maintain that we have to: -Open the Terminal and check the OpenCV version using the commands:

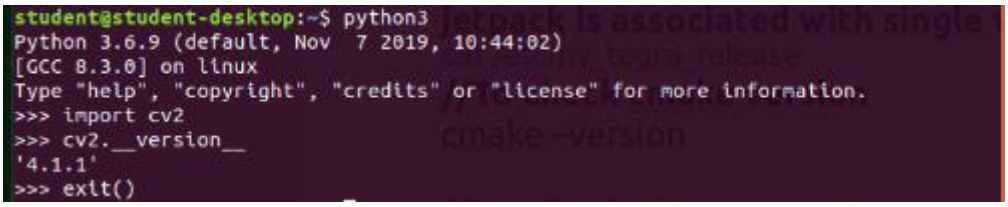


Figure 3.16: OpenCV installed version

```
 -OPENCV location /usr/local/include/opencv4/opencv2
-OPENCV samples folder:
/usr/share/opencv4/samples/
/usr/local/include/opencv4
-OPENCV header files:  /usr/local/cuda/include
```

3.6.3 The Edimax EW7811-UN USB wireless adapter

The Edimax EW7811-UN is a USB wireless adapter that can be used with the Jetson Nano. The Jetson Nano does not have built-in Wi-Fi capabilities, so using a USB wireless adapter like the Edimax EW7811-UN allows you to connect the Jetson Nano to Wi-Fi networks.in order to install it we have to Plug in the WI-FI USB on the Jetson Nano, Connect it to the internet, Power it and open the terminal and follow these commands by order:

- First of all, We must Update and upgrade our file system:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

-Then Download the driver:

```
$ git clone https://github.com/lwfinger/rtl8723bu.git
```

-Now, we will head to the downloaded directory using the command:

```
$ cd rtl8723bu
```

and keep working in this directory until the end of the installation .

-After that, we will use DKMS package to install the driver by typing the commands:

```
$ sudo apt-get install dkms
$ sudo apt-get install git linux-headers-generic build-essential dkms
```

-When the installation is done, we type :

```
$ source dkms.conf
```

-The next step is to create working project directory using

```
$ sudo mkdir /usr/src/$PACKAGE_NAME-$PACKAGE_VERSION
```

And copy the files to it by typing:

```
$ sudo cp -r core hal include os_dep platform dkms.conf Makefile
rtl8723b_fw.bin /usr/src/$PACKAGE_NAME$PACKAGE_VERSION
```

-Last but not least, We will add those files to DKMS using:

```
$ sudo dkms add $PACKAGE_NAME/$PACKAGE_VERSION
```

-Finally, we can install the driver using the command:

```
$ sudo dkms autoinstall $PACKAGE_NAME/$PACKAGE_VERSION
```

-When all of this is done, We have to reboot our board, and it is ready to go with the installation of the ROS part [14].

3.6.4 Installing ROS on the JETSON NANO

The installation is completely straightforward, to get it done successfully we need to open a new Terminal and follow these commands: [15]
1-in the beginning, we will start with the command :

```
$ git clone https://github.com/JetsonHacksNano/installROS.git
```

2-After that, we switch to the directory that has been downloaded using:

```
$ cd installROS
```

2-Now we will use help instruction to see the packages using :

```
$ ./installROS.sh -h
```

```
Usage: ./installROS.sh [[-p package] | [-h]]
Install ROS Melodic
Installs ros-melodic-ros-base as default base package; Use -p to override
-p | --package <packagename>  ROS package to install
                              Multiple usage allowed
                              Must include one of the following:
                               ros-melodic-ros-base
                               ros-melodic-desktop
                               ros-melodic-desktop-full
-h | --help  This message
```

Figure 3.17: The prerequisites and ROS packages

3- If no specific packages are specified, the default option for installation is `ros-melodic-ros-base` Usually, individuals will choose to install the `ros-base` option if they don't intend to run any desktop applications on their robot. in our case we will install `ros-melodic-desktop` by typing:

```
$ ./installROS.sh -p ros-melodic-desktop -p ros-melodic-rgbd-launch
```

4-Comming here, All we have to do is to create a default catkin workspace and we can do that using the command:

```
$./setupCatkinWorkspace.sh
```

3.7 Preparation of the RPLIDAR A1

To initiate, we connect the RPLidar A1 to the USB port of your NVIDIA Jetson Nano using a USB adapter and the provided communication cable. Once connected, open a terminal window and execute the given command to change the permission [16]

```
$ sudo chmod 666 /dev/ttyUSB0
```

Now we are able to read and write with this device using the port. and we can Verify it via this command

```
$ ls -l /dev | grep ttyUSB
```

the output should be something similar to this:

```
crw-rw-rw- 1 root dialout 188, 0 Dec 31 20:33 ttyUSB0
```

2-What comes is to make and configure a catkin workspace. to be able to create or modify existing catkin packages.to Create the catkin root and source folders we type he following command :

```
$ mkdir -p ~ /catkin_ws/src
```

Now, we switch to the source folder using:

```
$ cd ~/catkin_ws/src
```

and download in it the RPLIDAR ROS package by typing:

```
$ git clone https://github.com/robopeak/rplidar_ros.git
```

we head back by running:

```
$ cd ..
```

and we use this command to compile our catkin workspace.

```
$ catkin_make
```

Then we run to source the environment with our current terminal by using the command below and leave the Terminal, Without closing it.

```
$ source devel/setup.bash
```

Last but not least, we open a new Terminal and run the roscore command which is:

```
$ roscore
```

leave it open and head back to the terminal where we sourced the environment, and run below command.

```
$ roslaunch rplidar_ros view_rplidar.launch
```

An instance of **Rviz!** (**Rviz!**) will then open with a map of the RPLIDAR a1 surroundings. as it is shown below.see Figure 3.19.

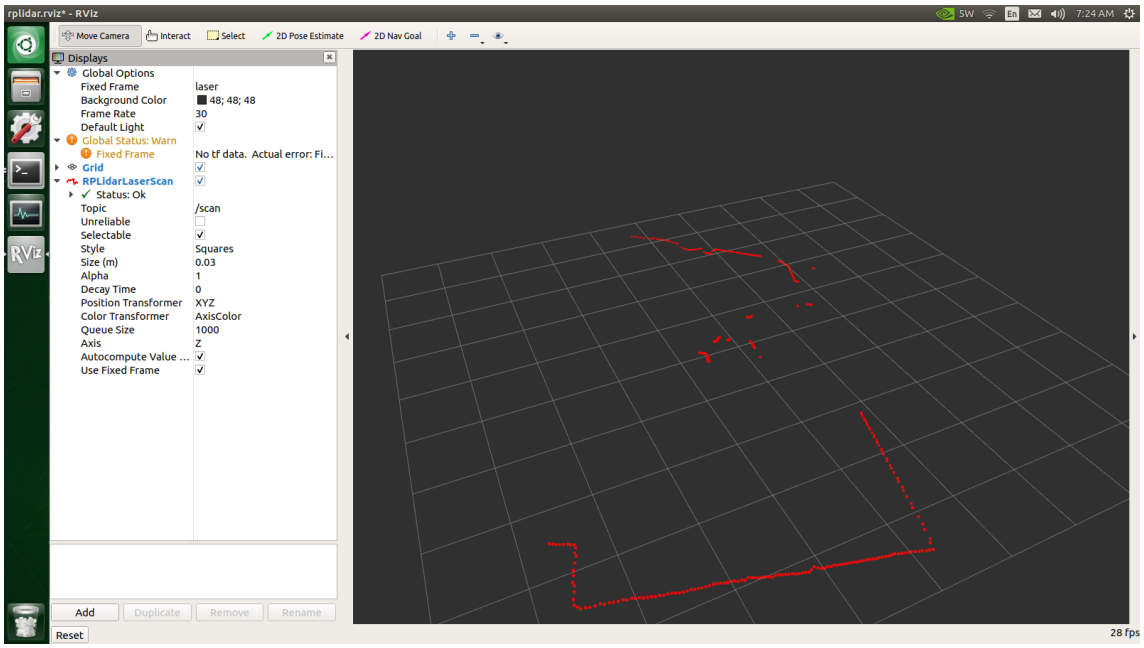


Figure 3.18: Visualization of LiDAR sensor in the RVIZ

So now the next step will be to create a map out of it.

3.8 Launching Hector SLAM

Incorporating mapping functionality into our system involves utilizing the Hector-SLAM package, which allows us to generate maps. These maps serve as visual representations of the environment in which the LIDAR operates. It only needs the data from Lidar and relies on scan matching approach to construct a complete map [17].

- hector_mapping The SLAM node.
- hector_geotiff Saving of map and robot trajectory to geotiff images files.
- To install the Hector Slam package we

need to use the following commands:
1-In the beginning, The hector-mapping nodes depend on Qt4, so we need to install it first.

```
$ sudo apt-get install qt4-qmake qt4-dev-tools
```

2-After that we have to move into catkin_w/src folder using:

```
$ cd ~/catkin_ws/src
```

3-Next step is to clone the Hector SLAM source files:

```
$ git clone https://github.com/tu-darmstadt-ros-pkg/hector_slam.git
```

-As we don't have a "base_footprint" frame, we need to make modifications to two files. In this case, we will designate our "base_link" frame as our odometry frame. firstly,we need to switch Hector SLAM's launch file which can be found at.using the commands:

```
$cd~/catkin_ws/src/hector_slam/hector_mapping/launch/mapping_default.launch
```

getting here, what follows is to modify these lines :

```
<arg name="base_frame"default="base_footprint"/ >
<arg name="odom_frame" default="nav"/>
```

and replace it with:

```
<arg name="base_frame" default="base_link"/>
<arg name="odom_frame" default="base_link"/>
```

Then, at the end of this file this line :

```
<!-- <node pkg="tf" type="static_transform_publisher" name="map_nav"
args="0 0 0 0 0 0 map nav 100"/> -->
```

must be changed to:

```
<node pkg="tf" type="static_transform_publisher" name="base_to_laser"
args="0 0 0 0 0 0 base_link laser 100"/>
```

Hereafter is to switch to `/catkin_ws/src/hector_slam/hector_slam_launch/launch` folder and open `tutorial.launch` file:

```
$ sudo nano tutorial.launch
```

And must change the line presented below:

```
<param name="/use_sim_time" value="true"/>
```

To:

```
<param name="/use_sim_time" value="false"/>
```

What comes is to open a terminal window and execute the subsequent command.

```
$ cd ~ /catkin_ws/
$ catkin_make_isolated
```

well, now we run to source the environment as usual :

```
$ source devel/setup.bash
```

after that we run this command:

```
$ roslaunch rplidar_ros rplidar.launch
```

leave the Terminal running .and launch the above file and see the result in rviz and start the mapping process by using:

```
$ roslaunch hector_slam_launch tutorial.launch
```

Now are able to see the map of the first scan in **Rviz** like below:

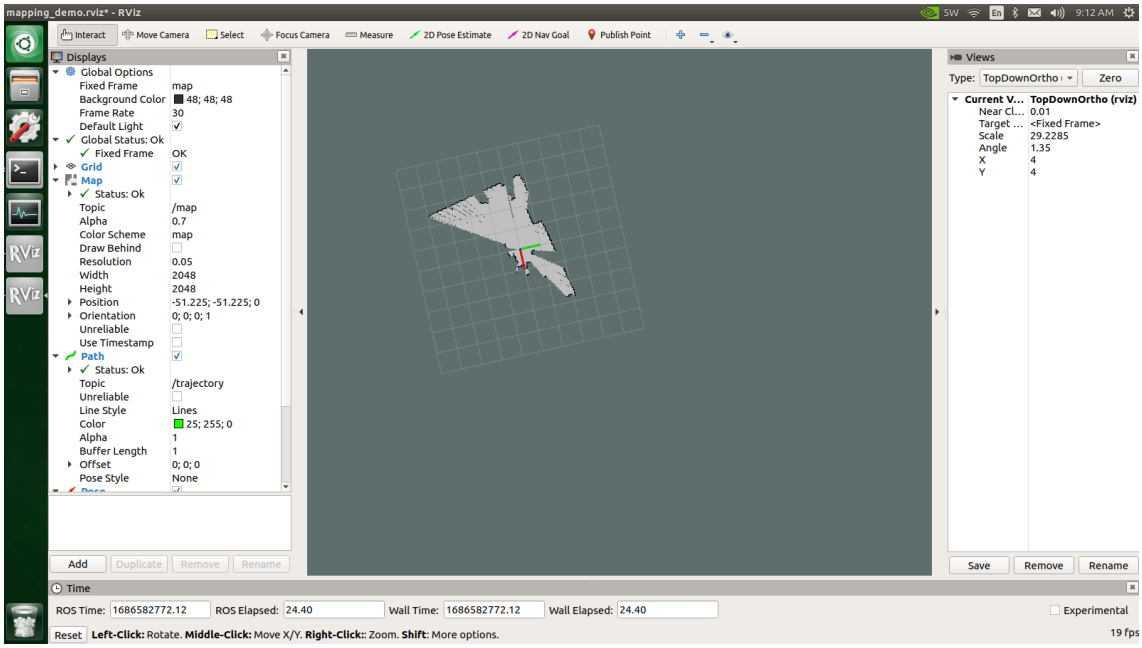


Figure 3.19: The map of the environment surrounding the robot in the RVIZ

3.9 Conclusion

This chapter has provided an overview of the components needed to build our mobile robot and introduced the concepts of launching the LiDAR sensor and implementing the Hector SLAM algorithm for mapping. In the next chapter we'll embark on practical experiments with the robot, we'll start by placing the robot in different places and environments, in addition, we'll use the Hector SLAM algorithm to build a map by implementing two important applications (control via keyboard and collision avoidance) that will allow the robot to navigate in multiple scenarios, we are going to compare between the results achieved and derive a concept about the absence of the closing loop technique in the Hector SLAM algorithm. These experiments will allow us to witness first hand the robot's navigation, mapping and

localization capabilities, further deepening our understanding of its capabilities and potential applications.

Chapter 4

Experimental validation

4.1 Inrtoduction

In this chapter, we will explore the process of connecting our robot, to Jupyter Notebook for seamless control via a keyboard interface. Also, We will get into the field of Collision Avoidance, where we aim to teach the robot how to navigate its environment safely. To achieve this, we will engage in data collection and model training using the powerful ResNet18 neural network architecture. By leveraging the robot's camera and the LiDAR sensor, we will gather visual and depth information to create a comprehensive map of the surroundings. Through live demonstrations, we will witness the robot's newfound ability to intelligently maneuver and avoid collisions in real-time, showcasing the potential of this integration between Jupyter Notebook and our robot's platform.

4.2 Connecting the robot to the Jupyter notebook

Getting to this point, now all that we have to do is to connect the IP indicated on the robot's OLED to the `Jupyter` interface and log in as shown below :

Figure 4.1: The Robot's IP address

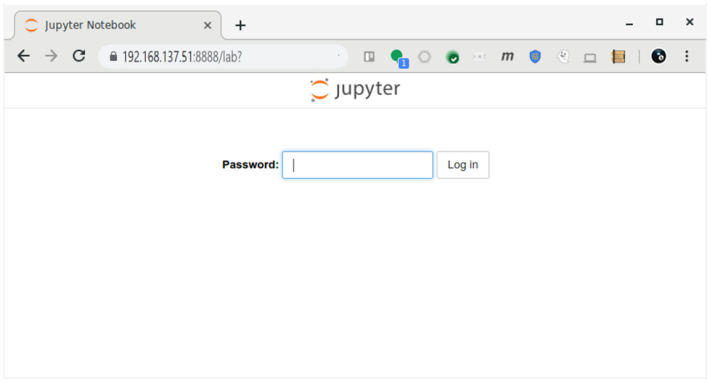


Figure 4.2: Connecting to the JUPYTER interface

the **Jupyter** notebook is a powerful platform that houses the Python 3 code which we are going to execute on our highly advanced robotic system. This notebook acts as a comprehensive environment, providing an interactive interface where we can seamlessly write, edit, and run our code. By utilizing this cutting-edge technology, we are able to harness the full potential of Python 3, enabling us to develop and implement intricate algorithms and instructions that will drive the functionalities of our sophisticated robot. Through the execution of the code within the **Jupyter** notebook, we can effectively control and guide our robot, empowering it to perform a diverse range of tasks, from autonomous navigation and object recognition to complex decision-making processes.

4.3 Controlling the mobile robot via KEYPAD

The first application to control the robot using a keypad. The initial code that we are going to utilize in our project is designed to provide us with the ability to control our robot using keypad inputs. This keypad-controlled mechanism establishes a direct and responsive channel of communication between us and the robot, facilitating a fluid and efficient control scheme. To commence we must import the Robot class, which provides convenient motor control capabilities for the robot. This essential class is part of the robot package.
Import the Robot class:

```
from jetbot import Robot
```

After successfully importing the Robot class, we can proceed to initialize an instance of the class using the following syntax.

```
robot = Robot()
```

Now, we will try to command our robot by running a test.

4.3.1 Commanding the robot

Having created our Robot instance, named "robot," we can now utilize this instance for controlling the robot. To achieve a counterclockwise spin at 40% of the robot's maximum speed, we can invoke the following command:

```
robot.left(speed=0.4)
```

In order to stop the robot you can call the stop method as follow:

```
robot.stop()
```

In some cases, we may desire to operate the robot for a specific duration. In such instances, we can utilize the time package in Python.

```
import time
```

After testing our robot we must link the motors to traitlets

4.3.2 Link motors to traitlets

An intriguing aspect of these traitlets is their ability to be linked with other traitlets. This feature proves highly useful as **Jupyter** Notebooks enable us to create graphical widgets that utilize traitlets internally. Consequently, we can connect our motors to widgets, facilitating their control from the browser or visualizing their values.

Note : the traitlets powers the configuration system of IPython and Jupyter and the declarative API of IPython interactive widgets. To demonstrate this functionality, we will generate and showcase two sliders that will serve as controls for our motors. The sliders that control the motors are shown in Figure 4.3

```
import ipywidgets.widgets as widgets
from IPython.display import display
# create two sliders with range [-1.0, 1.0]
left_slider = widgets.FloatSlider(description='left', min=-1.0,
max=1.0, step=0.01, orientation='vertical')
right_slider = widgets.FloatSlider(description='right', min=-1.0,
max=1.0, step=0.01, orientation='vertical')
# Create a horizontal box container to place the sliders next to each
other
slider_container = widgets.HBox([left_slider, right_slider])
# display the container in this cell's output
display(slider_container)
```

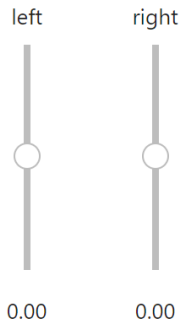


Figure 4.3: The sliders

We will not be able to use those sliders until we connect the motors, We'll do that by

using the link function from the traitlets package.

```
import traitlets
left_link = traitlets.link((left_slider, 'value'), (robot.left_motor,
'value'))
right_link = traitlets.link((right_slider, 'value'), (robot.right_motor,
'value'))
```

The preceding link function we have implemented actually establishes a bidirectional link. This implies that if we modify the motor values from another source, the sliders will automatically update accordingly. we will execute the code block provided below to observe this behavior:

```
robot.forward(0.3)
time.sleep(1.0)
robot.stop()
left_link.unlink()
right_link.unlink()
```

4.3.3 Attach functions to events

Traitlets can also be utilized by associating functions, such as "forward," with events (see Figure 4.4). These functions are triggered whenever a modification occurs in the object, and they receive relevant information about the change, such as the previous value and the new value. So here, to generate and showcase a set of buttons that will serve as controls for the robot:

```
# create buttons
button_layout = widgets.Layout(width='100px', height='80px',
align_self='center')
stop_button = widgets.Button(description='stop', button_style='danger',
layout=button_layout)
forward_button = widgets.Button(description='forward',
layout=button_layout)
backward_button = widgets.Button(description='backward',
layout=button_layout)
left_button = widgets.Button(description='left', layout=button_layout)
right_button = widgets.Button(description='right',
layout=button_layout)
# display buttons
middle_box = widgets.HBox([left_button, stop_button,
right_button],layout=widgets.Layout(align_self='center'))
controls_box = widgets.VBox([forward_button, middle_box,
backward_button])
display(controls_box)
```

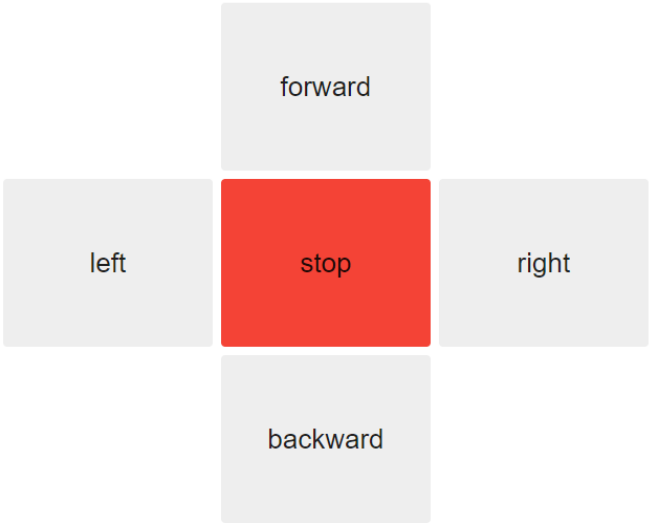


Figure 4.4: The keypad buttons

To do that we'll need to create some functions that we'll attach to the button's **on_click**

event.

```
def stop(change):
robot.stop()
def step_forward(change):
robot.forward(0.4)
time.sleep(0.5)
robot.stop()
def step_backward(change):
robot.backward(0.4)
time.sleep(0.5)
robot.stop()
def step_left(change):
robot.left(0.3)
time.sleep(0.5)
robot.stop()
def step_right(change):
robot.right(0.3)
time.sleep(0.5)
robot.stop()
```

Now that we've defined the functions, let's attach them to the on-click events of each button.

```
stop_button.on_click(stop)
forward_button.on_click(step_forward)
backward_button.on_click(step_backward)
left_button.on_click(step_left)
right_button.on_click(step_right)
```

By running the code mentioned earlier, incorporating LIDAR and **Hector SLAM**, we can generate a map that displays the environment, surroundings, and the robot's movement trajectory. This map provides a visual representation of the physical layout and captures the path the robot has followed.

Now, when we start moving the robot randomly using the keyboard we got the map shown in figure 4.5 as you can see the map has problems. it did not fit together correctly. It slowly became less accurate over time, some parts did not fit together well, and faraway things did not represent accurately. and that is due to the absence of the **loop closer** technique, which we have explained previously in Chapter 2.

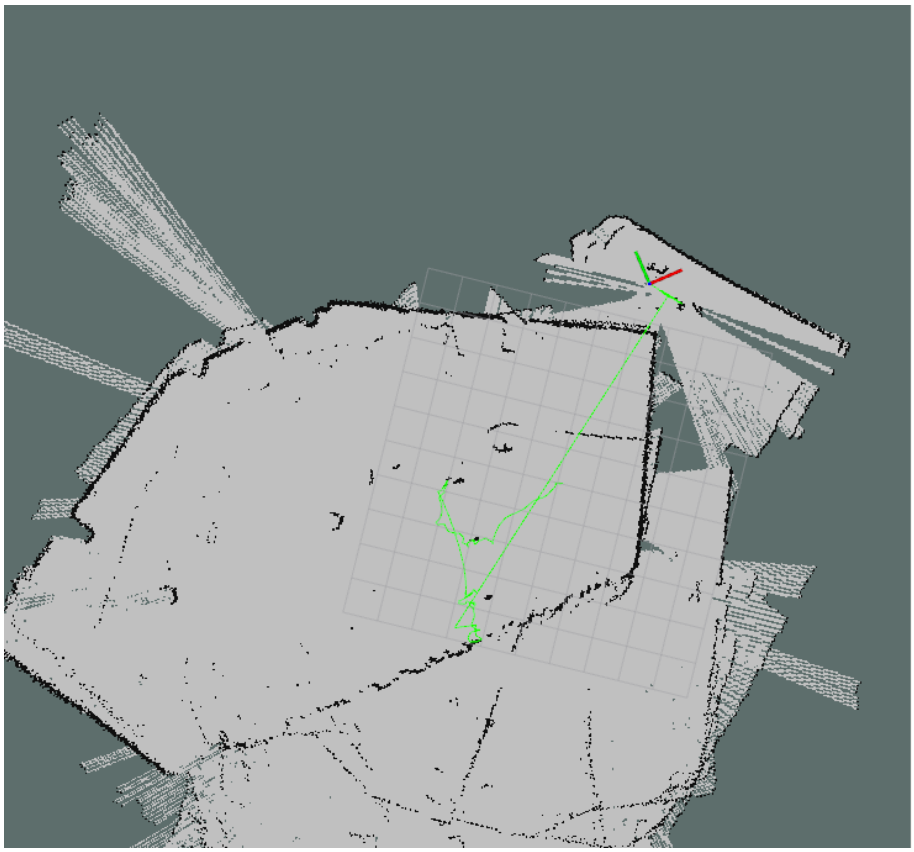


Figure 4.5: The Map and the path obtained using Hector SLAM technique

The Figure 4.6 presents the environment facing the robot and the map that has been generated when we moved it in a kind of direct path, The straightforward trajectory enables the robot to avoid scanning conflicts as what happened in Figure 4.6, as you can see it gives an excellent result, the map is clear and accurate.

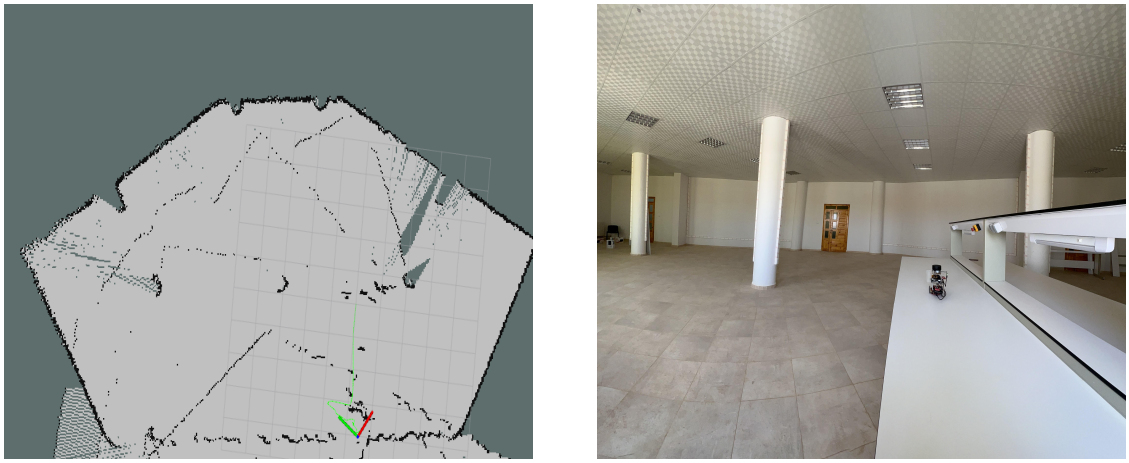


Figure 4.6: The Map and the path result obtained using Hector SLAM technique

4.4 Collision Avoidance

4.4.1 Data Collection

Our objective is to enable the robot to autonomously navigate and avoid collisions using a single camera sensor and deep learning techniques. By creating a virtual "safety bubble" around the robot, it can maneuver without hitting objects or encountering dangerous situations. The camera captures visual data, which is processed by a neural network running on the NVIDIA Jetson Nano platform. This allows the robot to recognize obstacles within its field of vision and react accordingly. While the robot's awareness is limited to what it sees, this approach enhances its ability to navigate safely and protect itself within its visual range. The way we'll do this is super simple:

First, we'll manually place the robot in scenarios where its "safety bubble" is violated, and label these scenarios **blocked**. We save a snapshot of what the robot sees along with this label.

Second, we'll manually place the robot in scenarios where it's safe to move forward a bit and label these scenarios **free**. Likewise, we save a snapshot along with this label.

Once we have lots of images and labels, we'll upload this data to a GPU enabled machine where we'll **train** a neural network to predict whether the robot's safety bubble is being

violated based on the image it sees. We'll use this to implement a simple collision avoidance behavior in the end. First, let's initialize and display our camera, our neural network takes a **224x224** pixel image as input. We'll set our camera to that size to minimize the file size of our dataset. The code below allows us to do that:

```
import traitlets
import ipywidgets.widgets as widgets
from IPython.display import display
from jetbot import Camera, bgr8_to_jpeg
camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
camera_link = traitlets.dlink((camera, 'value'), (image, 'value'),
transform=bgr8_to_jpeg)
display(image)
```

Next, we will create a few directories where we'll store all our data. We'll create a folder **dataset** that will contain two sub-folders **free** and **blocked**, where we'll place the images for each scenario.

```
import os
blocked_dir = 'dataset/blocked'
free_dir = 'dataset/free'
```

Those directories should appear when we refresh the browser.
after that, we will create and display some buttons that we'll use to save snapshots for each class label. We'll also add some text boxes that will display how many images of each category that we've collected so far. This is useful because we want to make sure we collect about as many free images as blocked images. It also helps to know how many images we've collected overall.see Figure 4.7

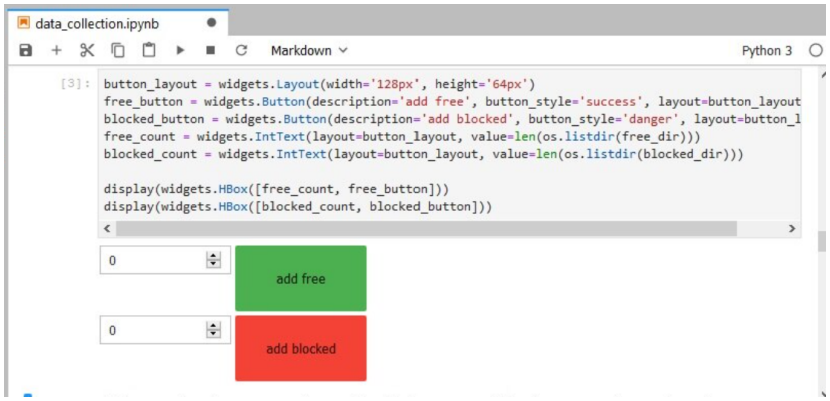


Figure 4.7: Free and blocked buttons

```
button_layout = widgets.Layout(width='128px', height='64px')
free_button = widgets.Button(description='add free',
button_style='success', layout=button_layout)
blocked_button = widgets.Button(description='add blocked',
button_style='danger', layout=button_layout)
free_count = widgets.IntText(layout=button_layout,
value=len(os.listdir(free_dir)))
blocked_count = widgets.IntText(layout=button_layout,
value=len(os.listdir(blocked_dir)))
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

Right now, these buttons wont do anything. We have to attach functions to save images for each category to the buttons' on_click event. We'll save the value of the Image widget (rather than the camera), because it's already in compressed JPEG format.

To make sure we don't repeat any file names (even across different machines) we'll use the **uuid** package in python, which defines the **uuid1** method to generate a unique identifier. This unique identifier is generated from information like the current time and the machine address.

```
from uuid import uuid1
def save_snapshot(directory):
image_path = os.path.join(directory, str(uuid1())  '.jpg')
with open(image_path, 'wb') as f:
f.write(image.value)
def save_free():
global free_dir, free_count
save_snapshot(free_dir)
free_count.value = len(os.listdir(free_dir))
def save_blocked():
global blocked_dir, blocked_count
save_snapshot(blocked_dir)
blocked_count.value = len(os.listdir(blocked_dir))
free_button.on_click(lambda x:  save_free())
blocked_button.on_click(lambda x:  save_blocked())
```

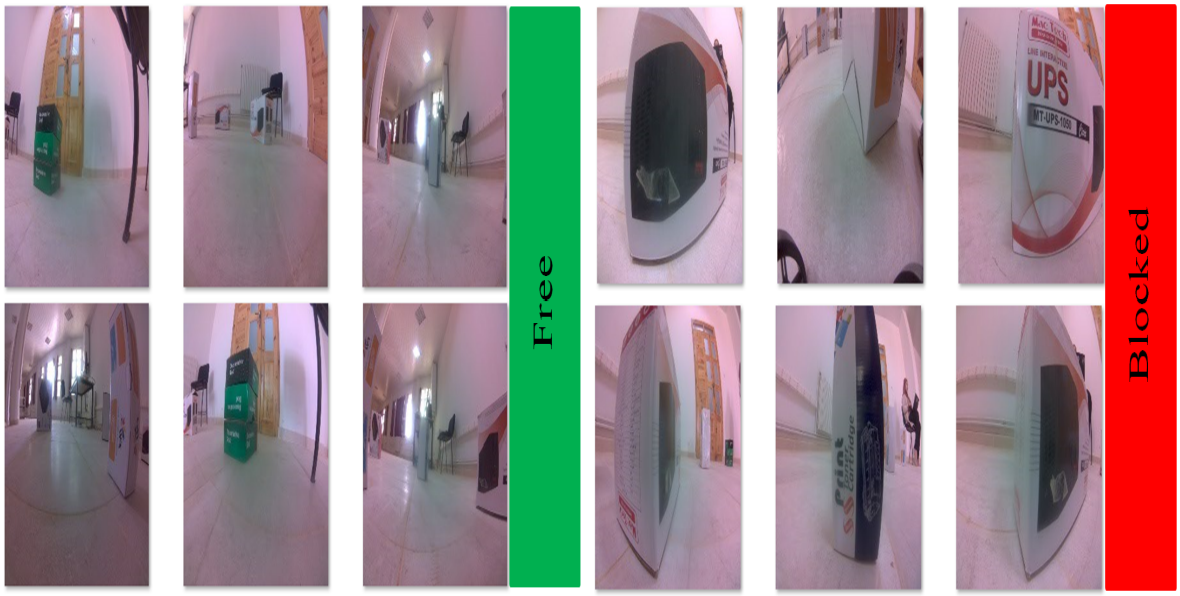
Now the buttons above shown in Figure 4.7 should save images to the free and blocked directories. what comes next is :

1. Placing the robot in a scenario where it's blocked and press add blocked
2. Placing the robot in a scenario where it's free and press add free

The code below helps us to combine the robot's view with the free & blocked buttons.

```
display(image)
display(widgets.HBox([free_count, free_button]))
display(widgets.HBox([blocked_count, blocked_button]))
```

The following Figure 4.8 represents samples from the blocked and free images that we have obtained during the process of collecting the data:



(a) free images (b) blocked images

Figure 4.8: Samples from the dataset folder

Once we've collected enough data, we'll need to copy that data to our GPU desktop or cloud machine for training. First, we can call the following terminal command to compress our dataset folder into a single zip file. we have to run the commande bellow:

```
!zip -r -q dataset.zip dataset
```

The ! prefix indicates that we want to run the cell as a **shell** (or **terminal**) command. The **-r** flag in the zip command below indicates **recursive** so that we include all nested files, the **-q** flag indicates **quiet** so that the zip command doesn't print any output.
Now a file named **dataset.zip** in the **Jupyter** Lab file browser should appear, all we have to do is download it our desktop and upload it in our robot's GPU to be able to train it.and that training is known as **Train model**, it's what's coming next.

4.4.2 Train Model (ResNet18)

In this section, we will train our image classifier to detect two classes free and blocked, which we'll use for avoiding collisions. For this, we will use ResNet18 which is a popular pre-defined convolutional neural network architecture commonly used for image classification. and a popular deep learning library **PyTorch**.as shown in the code below:

```
import torch
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms
```

The code above snippet provided imports essential modules from the PyTorch library and torchvision package. These modules include **torch** for core functionality, **optim** for optimization algorithms, **F** for neural network functions, **datasets** for pre-defined datasets, **models** for pre-trained models, and **transforms** for image transformations. These modules enable tasks such as building and training neural networks, using popular optimization algorithms, applying activation and loss functions, loading pre-defined datasets, utilizing pre-trained models, and performing image transformations. Together, they provide a comprehensive toolkit for training and working with deep learning models in computer vision applications.
the next step is to upload and extract dataset.zip file that we have created previously in **Data Collection**. we can do that by calling the command below :

```
!unzip -q dataset.zip
```

a folder named dataset appear in the file browser.as its shown in Figure—4.9:

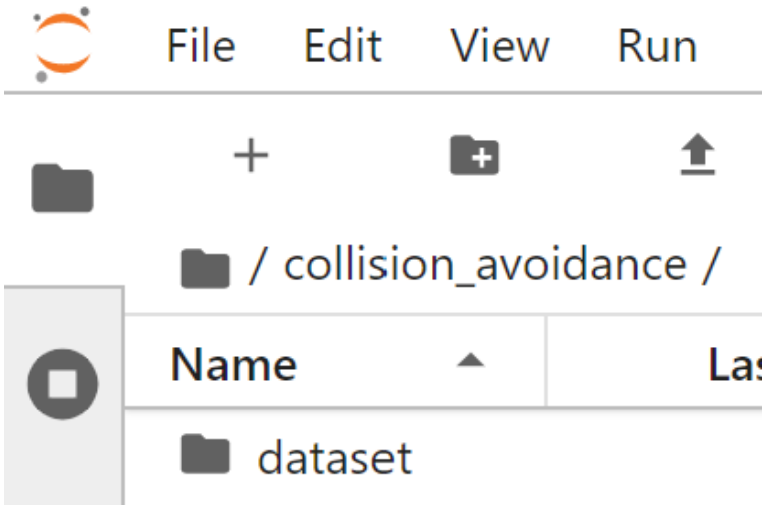


Figure 4.9: The dataset folder

Now we use the ImageFolder dataset class available with the torchvision.datasets package. We attach transforms from the torchvision.transforms package to prepare the data for training.

```
dataset = datasets.ImageFolder( 'dataset', transforms.Compose([
transforms.ColorJitter(0.1, 0.1, 0.1, 0.1), transforms.Resize((224,
224)), transforms.ToTensor(), transforms.Normalize([0.485, 0.456,
0.406], [0.229, 0.224, 0.225]) ]) )
```

Next, we split the dataset into **training** and test sets. The test set will be used to verify the accuracy of the model we train.

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset,
[len(dataset) - 50, 50])
```

After that,We'll create two DataLoader instances, which provide utilities for shuffling data, producing batches of images, and loading the samples in parallel with multiple workers.

```
train_loader = torch.utils.data.DataLoader( train_dataset,
batch_size=8, shuffle=True, num_workers=0, ) test_loader =
torch.utils.data.DataLoader( test_dataset, batch_size=8, shuffle=True,
num_workers=0,)
```

Right now, we define the neural network we'll be training. The torchvision package provides a collection of pre-trained models that we can use.
In a process called transfer learning, we can repurpose a pre-trained model (trained on millions of images) for a new task that has possibly much less data available.
Important features that were learned in the original training of the pre-trained model are re-usable for the new task. We'll use the resnet18 model.

```
model = models.resnet18(pretrained=True)
```

The resnet18 model was originally trained for a dataset that had 1000 class labels, but our dataset only has two class labels! We'll replace the final layer with a new, untrained layer that has only two outputs. as shown below:

```
model.fc = torch.nn.Linear(512, 2)
```

Finally, we transfer our model for execution on the GPU

```
device = torch.device('cuda') model = model.to(device)
```

Coming to the most important code that is used to train the neural network for 30 epochs, saving the best performing model after each epoch. An epoch is a full run through our data.

```
NUM_EPOCHS = 30
BEST_MODEL_PATH = 'best_model_resnet18.pth' best_accuracy = 0.0
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
for epoch in range(NUM_EPOCHS):
for images, labels in iter(train_loader):
images = images.to(device)
labels = labels.to(device)
optimizer.zero_grad()
outputs = model(images)
loss = F.cross_entropy(outputs, labels)
loss.backward()
optimizer.step()
test_error_count = 0.0
for images, labels in iter(test_loader):
images = images.to(device)
labels = labels.to(device)
outputs = model(images)
test_error_count += float(torch.sum(torch.abs(labels -
outputs.argmax(1))))
test_accuracy = 1.0 - float(test_error_count) / float(len(test_dataset))
print('%d:  %f' % (epoch, test_accuracy))
if test_accuracy  best_accuracy:
torch.save(model.state_dict(), BEST_MODEL_PATH)
best_accuracy = test_accuracy
```

This training loop continues for the specified number of epochs which in our case is 30 , optimizing the model's parameters using SGD and evaluating its performance on the test dataset. At the end, the best model based on test accuracy is saved to the file specified by **BEST_MODEL_PATH**. as its displayed in the Figure 4.10.

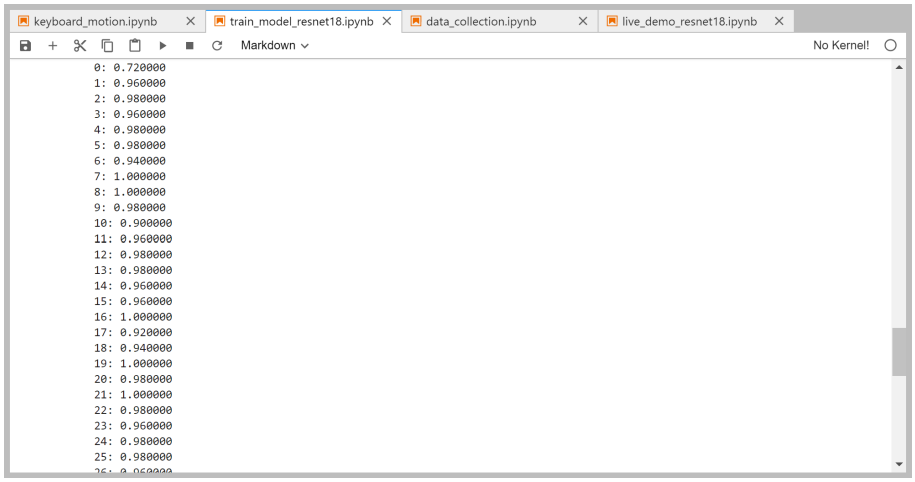


Figure 4.10: Choosing the best model basing on accuracy

Now all we have to do is download the file called **best_model_resnet18.pth** that appeared in our **Jupyter** browser, because we are going to need it in the Real-time Implementation of the model (Resnet18) which we are going to see in the upcoming.

4.4.3 Real-time Implementation of the model (Resnet18)

Approaching to our main objective which is to enable the robot to autonomously navigate and evade obstacles. Thereby, we are going to use **Collision Avoidance - real-time implementation (Resnet18)** and that refers to a real-time demonstration of a collision avoidance system using the ResNet18 neural network that we've talked about previously . In order to do that, we'll use the model we trained to detect whether the robot is free or blocked.

To intiate, we need to upload the **best_model.pth** file that we've already downloaded into our notebook's directory .

The first code is to initialize the PyTorch model :

```
import torch
import torchvision
model = torchvision.models.resnet18(pretrained=False)
model.fc = torch.nn.Linear(512, 2)
```

The provided code utilizes the PyTorch library and the torchvision package for computer vision tasks. It creates a ResNet-18 model, a popular architecture for image classification. By setting pretrained=False, the model is initialized with random weights. The last fully connected layer of the model is then replaced with a new layer that performs binary classification, with an input size of 512 (the output size of the previous layer) and an output size of 2, representing the two classes. This modification adapts the model to the specific binary classification task at hand.
Next step is to load the trained weights from the **best_model_resnet18.pth** file that we've uploaded and making sur that the keys match up correctly and there are no problems to address.

```
model.load_state_dict(torch.load('best_model_resnet18.pth'))
```

Currently, the model weights are located on the CPU memory we will execute the code below to transfer to the GPU device.

```
device = torch.device('cuda')
model = model.to(device)
model = model.eval().half()
```

In the following we will fix a slight issue,The format that we trained our model doesn't exactly match the format of the camera. once our model is loaded, we need to do some pre-processing. This involves the code below,this code will allow as to defined our pre-processing function which can convert images from the camera format to the neural network input format.

```
import torchvision.transforms as transforms
import torch.nn.functional as F
import cv2
import PIL.Image
import numpy as np
mean = torch.Tensor([0.485, 0.456, 0.406]).cuda().half()
std = torch.Tensor([0.229, 0.224, 0.225]).cuda().half()
normalize = torchvision.transforms.Normalize(mean, std)
def preprocess(image):
image = PIL.Image.fromarray(image)
image = transforms.functional.to_tensor(image).to(device).half()
image.sub_(mean[:, None, None]).div_(std[:, None, None])
return image[None, ...]
```

Now, we will display our camera, and We'll also create a slider that will display the probability that the robot is blocked. We'll also display a slider that allows us to control the robot's base speed.they are femilliar to the ones in the Figure 4.3.

```
import traitlets
from IPython.display import display
import ipywidgets.widgets as widgets
from jetbot import Camera, bgr8_to_jpeg
camera = Camera.instance(width=224, height=224)
image = widgets.Image(format='jpeg', width=224, height=224)
blocked_slider = widgets.FloatSlider(description='blocked', min=0.0,
max=1.0, orientation='vertical')
speed_slider = widgets.FloatSlider(description='speed', min=0.0,
max=0.5, value=0.0, step=0.01, orientation='horizontal')
camera_link = traitlets.dlink((camera, 'value'), (image, 'value'),
transform=bgr8_to_jpeg)
display(widgets.VBox([widgets.HBox([image, blocked_slider]),
speed_slider]))
```

We'll also create our robot instance which we'll need to drive the motors.

```
from jetbot import Robot robot = Robot()
```

Next, we'll create a function that will get called whenever the camera's value changes. This function will do the following steps:

1. Pre-process the camera image
2. Execute the neural network
3. While the neural network output indicates we're blocked, we'll turn left, otherwise we go forward.

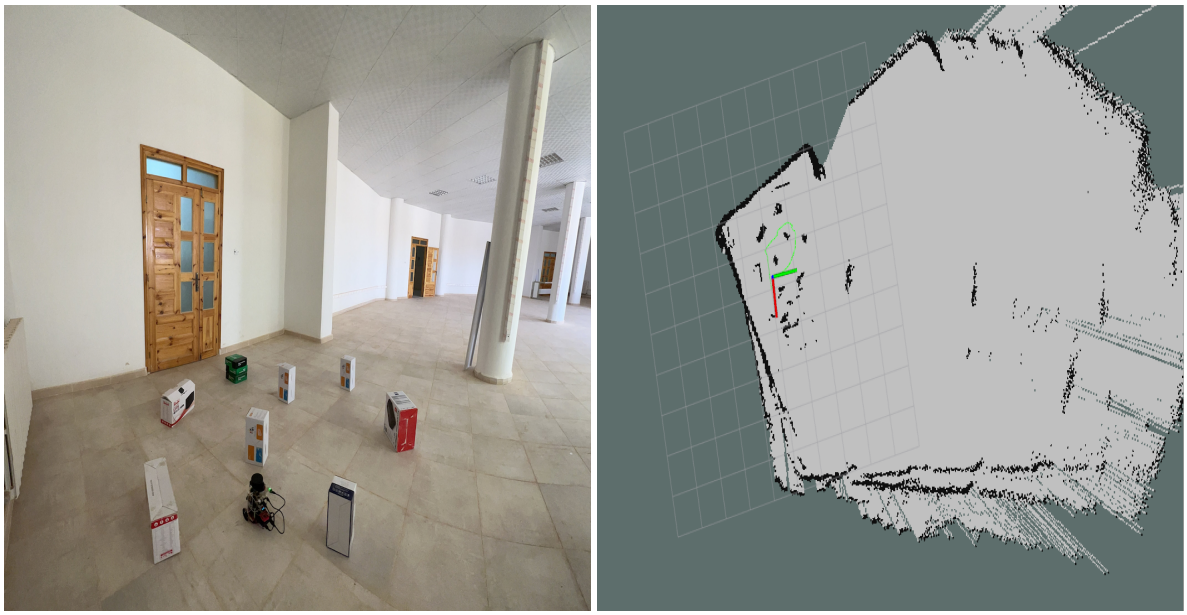
```
import torch.nn.functional as F
import time
def update(change):  global blocked_slider, robot
x = change['new']
x = preprocess(x)
y = model(x)
y = F.softmax(y, dim=1)
prob_blocked = float(y.flatten()[0])
blocked_slider.value = prob_blocked
if prob_blocked  0.5:
robot.backward(0.5)
else:
robot.left(0.5)
time.sleep(0.001)
update('new':  camera.value)
```

What's above means that We've created our neural network execution function, but now we need to attach it to the camera for processing.
We accomplish that with the observe function.that will attaches the **update** function to the **value** traitlet of our camera

```
camera.observe(update, names='value')
```

Reaching this point , all we have to do is giving our robot an initial speed value using the sliders , once we did that our robot start moving around and identifying the obstacles and

trying to avoid theme, as you can see Figure 4.11a is showing the environment that we put our robot in it and the Figure 4.11b presents the map the it generated and the path i took to be able to avoid the obstacles.



(a) (b)

Figure 4.11: The real environment and The map generated by using Collision Avoidance

If we ever want to stop this behavior all we have to do is executing the code below.

```
import time
camera.unobserve(update, names='value')
time.sleep(0.1)
robot.stop()
```

4.5 Conclusion

In conclusion, this chapter focused on the process of connecting the robot to Jupyter Notebook in order to establish control over the robot through keyboard commands. Additionally, it discussed the implementation of Collision Avoidance, which involved data collection, model training, and a live demonstration. The utilization of ResNet18, a popular deep learning model, enabled the integration of the camera and LIDAR systems to create a comprehensive

map. And in analyse results, we conclude that the maps obtained when the robot moves and returns in its initial pose were inaccurate because the Hector SLAM algorithm failed to compute accurate pose estimates, this is due to the lack of loop closure in the Hector SLAM algorithm, loop closure aims at correcting these errors by recognizing previously visited locations when the robot revisits them, resulting in precise maps.

General Conclusion

The work presented in this master thesis is devoted to the navigation and Mapping mobile robot using ROS and SLAM .

First chapter, we have explored certain generalizations about mobile robots and their importance and their applications in various fields.and we have dealt with the ROS platform and discussed its main concepts, which provide helpful tools such as packages, nodes, topics, and more.these tools help to build and develop software programs and applications for the robot.

Then we presented the mathematical model of a two-wheeled mobile robot using odometry information. And we explored how we use two SLAM algorithms for building a 2D map and navigating in unknown environment, in Hector SLAM algorithm we explained scan matching technique based in laser scan sensor data for estimating robot poses. And in Gmapping technique we used Rao-Blackwellized particle filter to estimate the robot's (position and orientation).

After, we presented in detail the different components used to build mobile robot, such as RPLIDAR A1 360-degree laser, we discussed the theoretical framework of laser triangulation is ToF (Time of Flight) also, we learned how we install JetPack and ROS in JETSON NANO card.Then we take a look at how we launch RPLIDAR A1 and Hector SLAM.

and in the last section, we presented different experimental results, we used in hector SLAM algorithm two applications, the first application is controlling robot via keyboard,and the second application is Collision Avoidance, by using cameras sensor and deep learning techniques that enable the robot navigate autonomously and avoid obstacles.the maps obtained in two applications were inaccurate because the algorithm failed to compute accurate pose estimates.This was due to the fact that Hector SLAM does not have a loop closing capability, and the mapping errors due to incorrect pose estimates accumulated over time.
In the future work we look at, we will use other techniques and algorithms more developed

that will allow the robot to navigate and create maps in an exalted and practical way.

Business Model Canvas

Our project focuses on the development of a mobile navigation and mapping robot for unknown environments. The robot uses Simultaneous Localization and Mapping (SLAM) techniques and utilizes the Robot Operating System (ROS) framework. Equipped with a combination of sensor cameras and LiDAR, our mobile robot is designed to collect data and navigate autonomously in unknown environments. Utilizing advanced algorithms and software, our robot provides accurate mapping and navigation capabilities, saving time and cost compared to manual methods. Our goal is to provide research institutions, industrial automation companies, and robotics enthusiasts with a reliable and efficient solution for mapping and navigating in unknown environments.


key Partners
Sensor manufacturers
Robot hardware manufacturers
Open-source community
Local universities or research institutions
Key Activities
Research and development
Hardware integration
Software development
Testing and validation
Key Resources
Robotics experts
Development tools and software
Manufacturing facilities
Value Proposition
Accurate mapping and navigation in unknown environments
Time and cost savings
Customizability
User-friendly interface
Customer Relationship
Personalized support
Continuous improvement
Community engagement
Channels
Online platforms and websites.
Robotics conferences and events.
Partnerships with robotics suppliers or distributors.
Customer Segments
Industrial automation Companies
Research institutions and universities
Robotics enthusiasts and hobbyists
Cost Structure
Research and development costs
Hardware and sensor costs
Operational costs
Revenue Streams
Robot sales
Subscription model
Consulting and customization

Bibliography

[1] Mohd Javaid, Abid Haleem, Ravi Pratap Singh, and Rajiv Suman. Substantial capabilities of robotics in enhancing industry 4.0 implementation. Cognitive Robotics, 1:58–75, 2021.

[2] RyuWoon Jung TaeHoon Lim YoonSeok Pyo, HanCheol Cho. ROS Robot Programming. ROBOTIS Co.,Ltd, Dec 22, 2017.

[3] SAIED Boufateh Lahcen and HASSASA Mohamed Badredinne. Implementation of a Perception Algorithm for a Mobile Robot Using ROS. Master thesis, University of Amar Telidji - Laghouat, 2022.

[4] Stefan Kohlbrecher, Oskar Von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable slam system with full 3d motion estimation. In 2011 IEEE international symposium on safety, security, and rescue robotics, pages 155–160. IEEE, 2011.

[5] Mustafa Eliwa, Ahmed Adham, Islam Sami, and Mahmoud Eldeeb. A critical comparison between fast and hector slam algorithms. REST Journal on Emerging trends in Modelling and Manufacturing, 3(2):44–49, 2017.

[6] Zhaohua Liu, Zhi Cui, Yong Li, and Wei Wang. Parameter optimization analysis of gmapping algorithm based on improved rbpf particle filter. In Journal of Physics: Conference Series, volume 1646, page 012004. IOP Publishing, 2020.

[7] Assembly guide for SparkFun JetBot AI kit - SparkFun learn.

[8] Yi Cheng and Gong Ye Wang. Mobile robot navigation based on lidar. In 2018 Chinese Control And Decision Conference (CCDC), pages 1243–1246, Shenyang, June 2018. IEEE.

[9] Maggie Peng. RPlidar a1 – SLAMTEC global network.

[10] Umberto Papa, Gennaro Ariante, Salvatore Ponte, and Alberto Greco. Ground control system for UAS safe landing area determination (SLAD) in urban air mobility operations.

[11] Leopard imaging camera - 145 degree FOV - DEV-15430 - SparkFun electronics.

[12] SD memory card formatter for windows download | SD association.

[13] balenaEtcher - flash OS images to SD cards & USB drives.

[14] Kamal Sehairi. Getting started with Jetson Nano developer kit V.4. Technical report, December 2021.

[15] kangalow. Install ROS on jetson nano.

[16] Shakhizat Nurgaliyev. Getting started with the low-cost RPLIDAR using jetson nano | linux. Section: Linux.

[17] Building a map using LiDAR with ROS melodic on jetson nano - hackster.io.