Development of an Autonomous Navigation System for a Wheeled Mobile Robot

Miguel André Gaspar Cordeiro Ferreira

Master's Dissertation

Supervisor at FEUP: Prof. António Mendes Lopes
Supervisor at INEGI: Eng. Luís Carlos Moreira

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado em Engenharia Mecânica

October 30, 2022

Desenvolvimento de um Sistema de Navegação Autónoma para um Robot Móvel

Resumo

A robótica móvel é um campo de investigação em rápida expansão devido ao seu grande número de aplicações. Os robots móveis autónomos (AMRs) são utilizados em múltiplas áreas, tais como automação industrial, logística e gestão de armazéns, cuidados médicos, guias de museus, exploração de ambientes com condições extremas como o espaço e os oceanos, e muitas outras aplicações industriais e não-industriais.

Nesta dissertação, foi desenvolvido um sistema de navegação autónoma para um robot móvel. O seu sistema de locomoção utiliza duas rodas motrizes movidas por um motor Beckhoff cada, e o seu controlo de movimento foi desenvolvido utilizando o software de automação TwinCAT 3 da Beckhoff num PC industrial. Um computador Raspberry Pi de dimensões reduzidas tratou da percepção do ambiente utilizando um módulo Lidar e uma webcam, e os sistemas de localização e planeamento através do Robot Operating System (ROS). A comunicação foi estabelecida entre estes dois computadores, resultando num sistema de navegação completo.

Este sistema foi testado, e subsequentemente afinado de forma a obter um melhor desempenho. Um teste final mostrou que o sistema de navegação funciona com sucesso, e algumas limitações relacionadas com o design do robot e o seu sistema de navegação foram observadas e discutidas.

Os métodos utilizados nesta dissertação podem ser alargados, ajustados, e utilizados noutros robots móveis.

Development of an Autonomous Navigation System for a Wheeled Mobile Robot

Abstract

Mobile robotics is a rapidly expanding field of research due to its large number of applications. Autonomous mobile robots (AMRs) are used in multiple areas, such as industrial automation, logistics and warehouse management, medical care, museum guides and restaurant waiters, exploring extreme environments such as space and oceans, and many other industrial and non-industrial applications.

In this dissertation, a navigation system for a low-cost mobile robot was developed. Its locomotion system used two driven wheels powered by a Beckhoff motor each, and their motion control was developed using Beckhoff's TwinCAT 3 automation software in an industrial PC. A separate small sized Raspberry Pi computer handled the perception of the environment using a Lidar module and a webcam, and the localization and planning systems through Robot Operating System (ROS). Communication was set up between these two computers, resulting in a complete navigation system.

This system was tested, and subsequently tuned to achieve a better performance. A final test showed that the navigation system was functioning successfully, and some limitations related to the robot's design and its navigation system were discussed.

The methods used in this dissertation can be extended, adjusted, and used in other mobile robots.

Acknowledgments

This dissertation marks the end of my academic journey of the past five years, studying mechanical engineering and specializing in automation.

I would like to thank my parents for giving me the opportunity to spend these years studying in Porto and supporting me both financially and psychologically.

Regarding the development of this dissertation, there are multiple people I would like to express my gratitude towards.

I am extremely grateful for my supervisor Eng. Luís Carlos Moreira's help. His guidance and suggestions were of extreme importance, allowing me to explore the subjects related to this project and always have an idea of how to proceed.

I am also very thankful to Prof. António Mendes Lopes, whose knowledge and opinions as a supervisor were crucial.

I'd also like to acknowledge other INEGI employees for their help. I'd like to thank Filipe Pinto for his assistance in 3D printing components and their assembly. Lastly, I'd like to recognize Marco Rodrigues for his availability to help and friendliness.

Table of Contents

List of Figures

List of Tables

Acronyms and Symbols

AMR – Autonomous Mobile Robot

FBD – Function Block Diagram

GPIO – General Purpose Input/Output

GPS – Global Positioning System

GVL – Global Variable List

IL – Instruction List

IPC – Industrial PC

LD – Ladder Diagram

Lidar – Light detection and ranging

OpenCV – Open Source Computer Vision Library

PLC – Programmable Logic Controller

QR Code – Quick Response Code.

RAM – Random Access Memory

RDP – Remote Desktop Protocol

ROS – Robot Operating System

SFC – Sequential Function Chart

Sonar – Sound navigation and ranging

SSH – Secure Shell

ST – Structured Text

UART – Universal Asynchronous Receiver/Transmitter

1 Introduction

Automation solutions incorporating mobile robots in the industry have grown in popularity in the last decades. This is due to an increasing demand for more flexible solutions. AMRs (autonomous mobile robots) are used in multiple areas, such as industrial automation, logistics and warehouse management, medical care, museum guides and restaurant waiters, exploring extreme environments such as space and oceans, and many other industrial and non-industrial applications.

The present dissertation focuses on the development of a navigation system for a low-cost mobile robot.

1.1 Scope of the project at INEGI

INEGI – Institute of Science and Innovation in Mechanical and Industrial Engineering is a Research and Technology Organisation (RTO), founded in 1986, focused on research and technology-based innovation activities, technology transfer, consulting, and technological services, oriented to the development of industry and economy in general.

INEGI is working in multiple areas of research and development. Some, that are relevant to this dissertation, are the areas of robotics and flexible automation.

1.2 Objectives of the project

The objective of this project is to develop a navigation system that is compatible with a specific pre-existing mobile robot that allows it to autonomously navigate its workspace and avoid obstacles. To achieve this, multiple subsystems need to be developed, set up, and configured to communicate successfully. The robot in question is a low-cost robot developed previously in INEGI that includes an anthropomorphic robot arm.

1.3 Methodology followed in the project

In an early stage of the dissertation, the work performed was based on learning information about the robot used in this dissertation and researching information about mobile robotics. their navigation systems and sensors used for perception.

A decision was made on utilizing a Lidar module and a webcam as exteroceptive sensors connected to a Raspberry Pi. As such, these components were assembled in the robot and given a protective bumper. Methods of utilizing the Lidar module and the webcam were researched and ROS and OpenCV were chosen respectively.

The motion control system was developed on Beckhoff's TwinCAT 3 due to all the motors being from Beckhoff. The Raspberry Pi was set up to be used headless, to be ROS compatible, and to be compatible with the Lidar module. Information from all sensors needed to be published to the Raspberry Pi's ROS environment. This included sending odometry information from the robot's industrial PC's TwinCAT 3 to the Raspberry Pi and subsequently publishing it in ROS. To aid this communication, *pyads* was utilized. The localization and trajectory planning systems were developed using open-source ROS packages and were tuned to provide a better performance. The localization system was built upon by adding localization updates using QR codes as landmarks. An option to feed goals to the robot through QR codes detected by the webcam was also developed.

Multiple tests were performed to tune parameters of the navigation system and improve its performance.

The final version of the navigation system was tested, and the results were analysed and discussed.

1.4 Structure of the dissertation

This dissertation is divided into 6 different chapters. These chapters are based on the order of development mentioned in the methodology and on their dependency of information included in previous chapters. In **Mobile Robotics**, the importance of robotics and mobile robots are discussed. The basics of mobile robots and each subsystem required are also analysed in some detail. **Robot Overview** presents information about the robot used, including hardware upgrades it received during the development of this dissertation. In the chapter **Motion Control System**, the kinematics of a differential drive locomotion system are detailed. Software used for control of the robot's motors as well as the programming used to implement movement commands and odometry calculation are also included. Subsequently, the process of testing and correcting the odometry is specified. **Navigation System** explores the process of setting up a Raspberry Pi computer to be accessible remotely and to be compatible with every other piece of hardware and software used in this dissertation. It also discusses the open source navigation configuration used in this dissertation, including the fulfilment of its pre-requisites, setting up its different subsystems, the tuning of its parameters, and the implementation of computer vision to enhance the robot's navigation. The navigation system as a whole is also tested, and various behaviours exhibited by the robot are observed. In **Conclusions and Future Work**, conclusions regarding the robot and the navigation system's performance and limitations are presented. Solutions to these limitations are proposed as future work.

2 Mobile Robotics

Robotics is an essential area of industrial development and automation. It allows large increases in efficiency and productivity, precision, and quality of manufacturing processes, while increasing safety when performing tasks that would be dangerous for a human operator to perform.

Robot arms, or manipulators, are a prime example of this. They can move with great speed and accuracy to perform repetitive tasks such as welding, painting, and assembly. However, robot arms suffer from a large disadvantage which is their lack of mobility [1]. Consequently, mobile robots incorporating robot arms aim to overcome this disadvantage, and as such, have been the target of intensive research in the last decades.

Mobile robotics is one of the fastest expanding fields of research due to its large number of applications. Autonomous mobile robots (AMRs) are used in multiple areas, such as industrial automation, logistics and warehouse management, medical care, museum guides and restaurant waiters, exploring extreme environments such as space and oceans, and many other industrial and non-industrial applications [1, 2, 3].

Many mobile robots can move autonomously through the environment they are designed to travel in, without any assistance from an external human operator apart from being given certain goals. The robot can perform these tasks due to its perception capabilities, trajectory planning, and navigation systems working together to find possible solutions. The basics of mobile robots can be divided in the following areas: locomotion, perception, localization, and planning and navigation [1, 2, 4].

2.1 Locomotion

A mobile robot requires locomotion mechanisms to be able to move in its working environment and perform its tasks. Even though most mobile robots operate in known environments with flat ground where the use of wheels excels because of their extremely high efficiency and simple kinematics and odometry, there is also a need for robots to operate in other more unpredictable or dangerous environments, where other locomotion solutions are used [1].

The locomotion mechanism used in mobile robots needs to take multiple characteristics into consideration, such as [1]:

- type of environment (land, air, water);
- characteristics of contact:
  - contact points and their shape;

  - friction;
- stability:
  - centre of gravity;
  - number of contact points;
  - inclination of the terrain.

Mobile robots can be categorized as land-based, water-based, air-based or hybrid mobile robots. For land-based mobile robots, wheeled systems (Figure 1) are by far the most common locomotion mechanisms. Tracked systems (Figure 2) can be a good solution for rugged terrains. As for legged systems (Figure 3), they have only more recently become a mature technology and are more complex to implement [2].

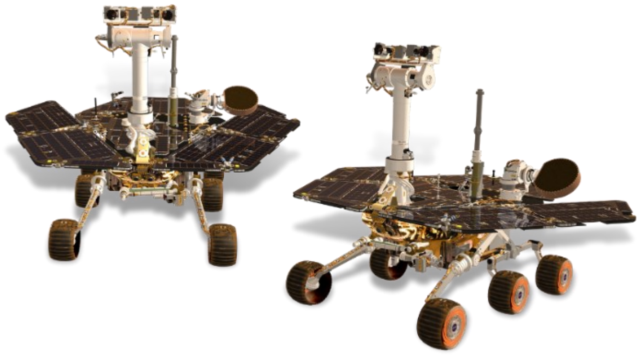


Figure 1: Mars Opportunity Rover, a wheeled robot by NASA [5].

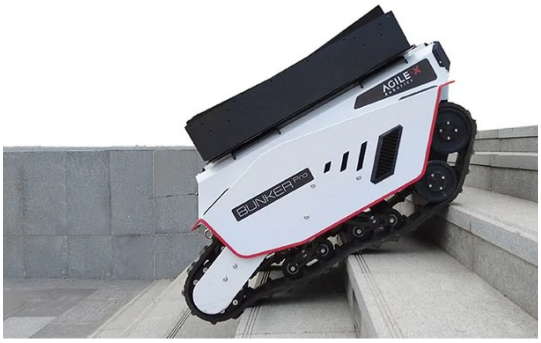


Figure 2: BUNKER Pro, a tracked robot by AgileX [6].

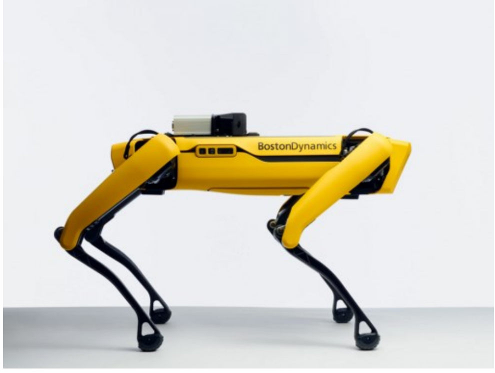


Figure 3: Spot, a legged robot by Boston Dynamics [7].

2.1.1 *Wheeled mobile robots*

Wheels are one of the most common systems for robotic locomotion. When designing a robot for use in flat non-rugged terrain, wheeled locomotion brings the advantages of being simpler, in terms of design and kinematics, and having a lower economical cost. The use of wheels also contributes to a great stability of the robot [1, 2]. These factors make wheeled mobile robots the most popular choice for most industrial environments.

- **Type of Wheels:**

There are four basic wheel types [2]:

1) Fixed wheels (Figure 4). These only have one degree of freedom in the axis that the wheels rotate on.
2) Castor wheels (Figure 5). These wheels have two degrees of freedom. One in the axis that the wheels rotate on, and another in the vertical axis where the wheels can also rotate.
3) Omnidirectional wheels (Figure 6). These wheels have three degrees of freedom. They are commonly used for omni-directional robots.
4) Spherical wheels (Figure 7). These have more degrees of freedom but are harder and more complex to implement.

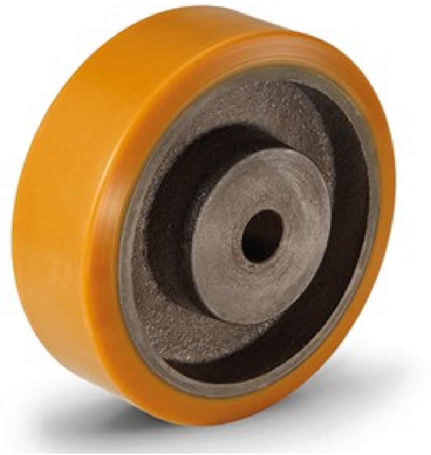


Figure 4: Fixed wheel by ELESA+GANTER [8].

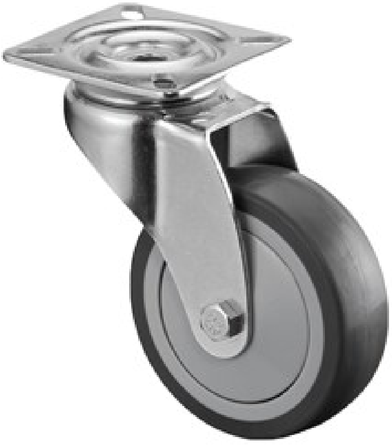


Figure 5: Castor wheel by MÄDLER [9].

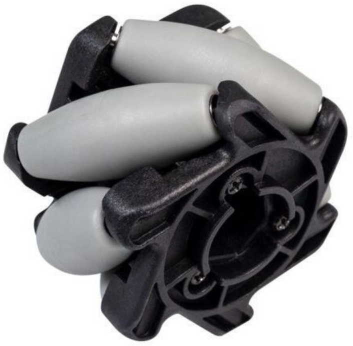


Figure 6: Mecanum wheel by VEX Robotics [10].

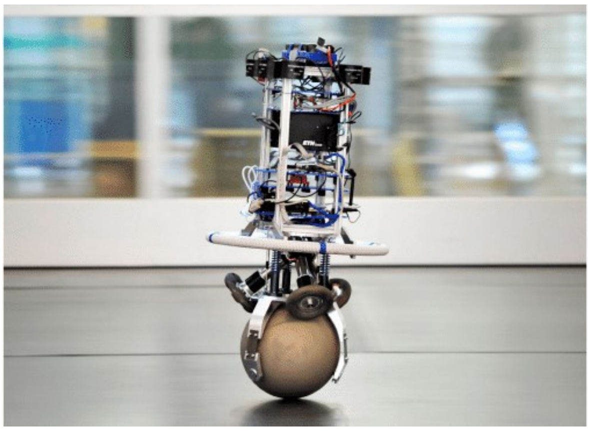


Figure 7: Rezero, a Ballbot that uses a single spherical wheel developed in ETH Zurich [11].

To ensure that the robot has the stability and manoeuvrability that are necessary for its use, there are multiple different wheel configurations that can be used. Typically, the minimum number of wheels necessary to guarantee static stability is three, however some models only use one or two wheels [1]. The use of extra wheels can increase the robot’s stability, but it will also result in an increased economical cost.

The different types of wheels can be configured together to achieve different drive types. These influence the kinematic freedom of the robot, stability, compatible terrain and economical cost of the robot.

- **Steer drive:**

The steer drive is a configuration where one or more steered traction caster wheels have the function of steering and locomotion. It also utilizes free fixed wheels for stability. These configurations have high stability specially at higher speeds and on irregular terrain, however their manoeuvrability is quite limited as they cannot rotate in place [1, 12]. An example of this configuration is provided in Figure 8.

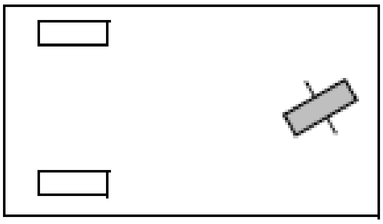


Figure 8: Steer drive with two free wheels in rear and one steered traction wheel in front [1].

- **Differential drive:**

This configuration utilizes fixed traction wheels positioned symmetrically. Steering is controlled by the velocity differential of the traction wheels. Additional free castor wheels or omnidirectional wheels can be added for additional stability. This configuration provides good manoeuvrability and the ability to steer in a circle with a null radius [1, 12]. An example of this configuration is provided in Figure 9. In industrial environments this is one of the most common drive configurations for locomotion, typically with four free castor wheels for increased stability and to have the rotational centre coincide with the geometrical centre of the robot (Figure 10).

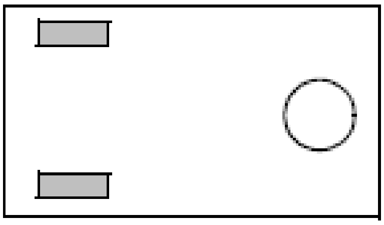


Figure 9: Differential drive with two independently driven wheels and one unpowered omnidirectional wheel [1].

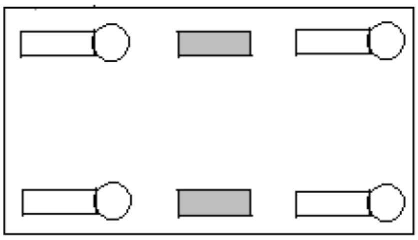


Figure 10: Differential drive with two independently driven wheels and four free castor wheels.

- **Omnidirectional drives:**

Omnidirectional drives have the highest manoeuvrability. However, they are more complex, involve greater costs (more motors) and can be unstable on uneven terrain. Common configurations are the use of quad drives (Figure 11) where all wheels are castor wheels, and some or all are steered traction castor wheels. Other common configurations are using omnidirectional wheels (Figure 12 and Figure 13) [1].

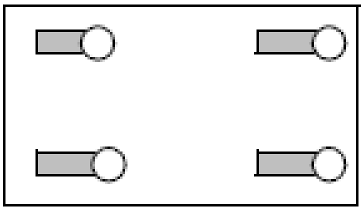


Figure 11: Quad drive with four motorized and steered castor wheels [1].

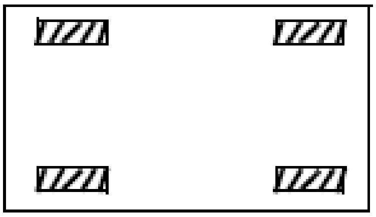


Figure 12: Omnidirectional drive using four omnidirectional wheels [1].

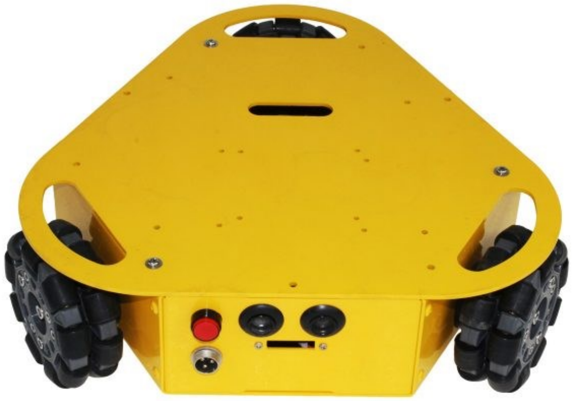


Figure 13: Omnidirectional drive robot using three omnidirectional wheels [13].

2.2 Perception

It is extremely important for an autonomous mobile robot to obtain information about its environment. This information can then be used for the robot to locate itself, avoid obstacles and map its surroundings.

It is also very important for safety reasons, as in many cases the robot will share a workplace with humans, and it is extremely important to make sure that the robot will not collide with any human or harm them. As humans are not static and will move around, they will also not be included in any maps created to assist the robot’s navigation. This is also the case for many objects that are regularly moved around in the environment. These non static obstacles make it extremely important for the robot to always be sensing its surroundings and using that information to create flexible navigation plans.

2.2.1 Sensors

Sensors are the main form of obtaining information about the robot and about its surroundings. They are proprioceptive if they measure information internal to the robot, such as motor speed and position in the case of encoders. They are exteroceptive if they acquire information external to the robot, such as distance to obstacles or camera vision [1].

Sensors can also be characterized as passive or active. Passive sensors read external energy that enters the sensor. Active sensors emit energy and then measure the reaction [1, 2].

Some sensors relevant to the perception systems of AMRs are addressed here.

- **Motor/wheel sensors:**

Motor/wheel sensors are extremely important for robots to obtain information about their position and velocities. They can be used for odometry, which is the use of the information from these types of sensors to estimate the variation of the system's position over time. An encoder is a sensor which transforms a position into an electronic signal. Encoders can be mechanical, magnetic, resistive, or optical (Figure 14); optical being the most common option [14].

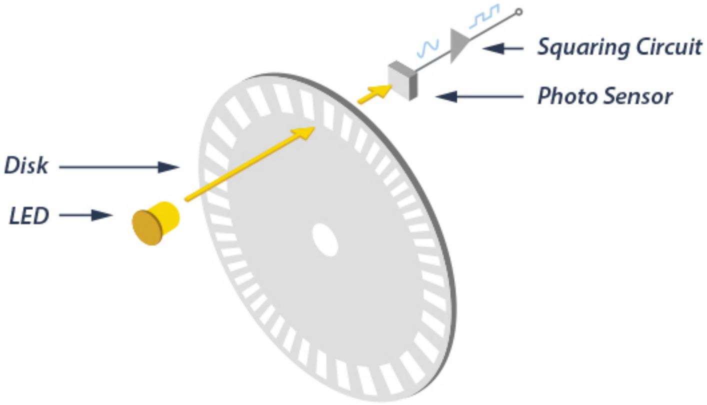


Figure 14: Optical encoder [14].

- **Tactile sensors:**

Tactile sensors are used to sense objects at close distances. They can detect these objects either by direct physical contact or by closeness. Tactile sensors are commonly used as safety measures. Some examples are contact switches, bumpers (Figure 15), and optical barriers [2].

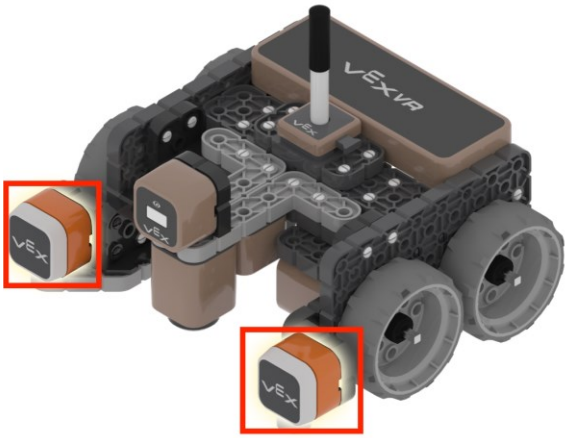


Figure 15: Mobile robot with bumper sensors [15].

- **Heading sensors:**

Heading sensors are used to obtain the orientation or position of the robot over time in relation to a fixed frame. Some examples are compasses, gyroscopes, and accelerometers (Figure 16) [1].

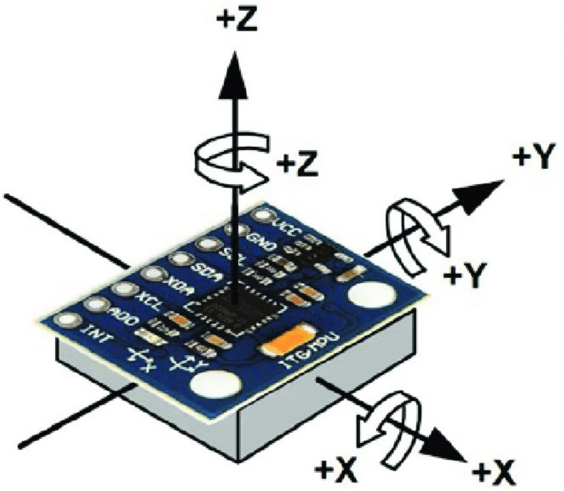


Figure 16: Gyroscope and accelerometer module [16].

- **Active ranging sensors:**

These types of sensors acquire information about distances to certain obstacles to help the robot navigate, locate itself and evade collisions. Some examples are Sonar (Sound Navigation and Ranging) and Lidar (Light Detection and Ranging). Lidar modules can be 2D or 3D. A 2D Lidar module with 360° ranging that works using a motorized rotating low power infrared laser light source and measuring the environment's reaction can be seen in Figure 17.

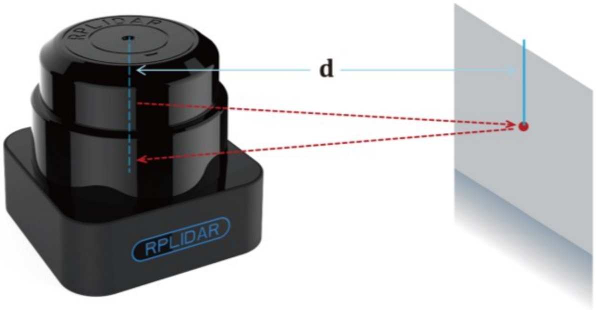


Figure 17: 2D Lidar module by Slamtec [17].

- **Vision-based sensors:**

These sensors process visual data. Normal cameras (Figure 18), infrared cameras, 3D cameras and others can be used. 3D cameras can also be used compute the distance to objects that are captured. Alternatively, multiple normal cameras can be used to simulate a 3D camera. These types of sensors can be used for object detection and many others such as evaluating if a zone is passable or not by the robot. They are of extreme importance in autonomous road vehicles, as they can be used to identify traffic markings and signs [18].

Figure 18: USB Webcam [19].

The sensors discussed previously are the most common in mobile robots, but there are also many others that can be used to obtain information relevant to the robot and help in its navigation or other functions.

2.3 Localization

While it is not necessary for a mobile robot to have a localization system, there are certain advantages in implementing it in its navigation system. As such, there are two very different approaches to the localization issue. One is to avoid having a localization system, and the other is performing map-based localization. For industrial use, GPS based localization is not currently viable since GPS systems are not accurate enough and have problems working correctly indoors or in enclosed areas [1].

When using non-mapped navigation, it is not possible to give the robot goals relative to a map or known location. Instead, any goals given to the robot must be relative to its current position or the position the robot's odometry started in.

Localization systems allow the robot to determine its position in the environment. This localization can be done with reference to a pre-built map of the robot's working environment.

In industrial settings, it is necessary for the robot to be able to travel to and from specific positions in the facilities' layout. This requires a model (map) of the environment the robot is in and knowing the position of the robot relative to that map. For the robot to be able to locate itself in the map, information from sensors is extremely important. This is known as map-based localization and navigation. The mapping of the environment can be done using the information from the robot's sensors [2]. It is possible to create a map using recorded data from sensors that read the distance to obstacles, such as Lidars, together with the robot's odometry data. To record this data, a user can drive the robot manually with a remote control, such as a joystick, around the environment to be mapped.

There are multiple techniques developed for creating a map representation of the environment. A map representation can be continuous or decomposed. One of the most common types of map representation is using an occupancy grid representation, which is a form of fixed decomposition. In an occupancy grid, the environment is represented by a discrete grid where each cell can be either filled (representing an obstacle) or empty (representing clear space). They can also have a third value representing unknown space [1]. Figure 19 shows an example of an occupancy grid map where each cell contains one of the three mentioned values.

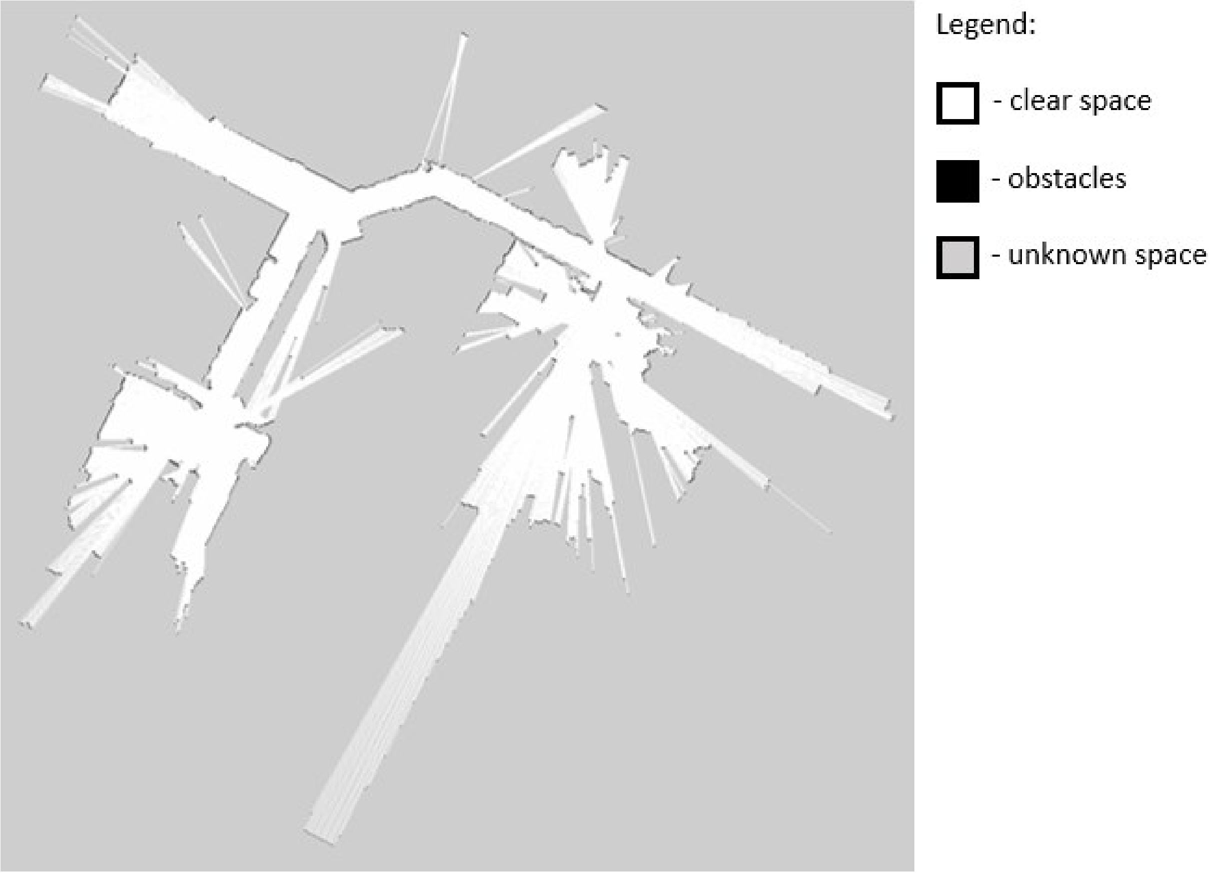


Figure 19: Occupancy grid recorded using sensor data and odometry information [20].

2.4 Planning and navigation

The objective of path planning is to find the best path possible for the robot to reach its goal while avoiding collisions, allowing the robot to navigate through obstacles. However, it does not take into consideration the evolution of the robot's motion over time. A more complete approach is trajectory planning, which tries to compute a plan that includes the robot's accelerations and velocities over time necessary to reach the desired goal [2].

There are multiple motion planning techniques and algorithms, that can fall in different categories [2]:

- Classical methods;
- Probabilistic methods;
- Heuristic planners;
- Evolutionary algorithms.

Heuristic planners are popular because they can compute possible paths more efficiently and faster than the classical methods [2].

It is also important for the robot check for obstacles in real-time while travelling through a planned trajectory since its environment may not be static. Most environments have objects that are not always in the same position. Humans, other mobile robots, and movable furniture, such as chairs, can also share an environment, making the space perceived by the robot change dynamically. Because of these factors, it is important for the trajectory planner to be able to react to the environment and create new motion plans incorporating new information gained during the execution of its path [1, 2].

Obstacles in the map can also be inflated by a certain radius to account for the robot's radius and given extra padding (inflation) to encourage the robot to not drive too close to obstacles. This padding is given a higher cost (value used in planning algorithms) the closer it is to an obstacle, creating a costmap. The planner then searches for the path with the total lowest cost considering distance travelled and the cost of the terrain the robot would travel in. Figure 20 shows an example of a costmap containing this inflation and the planned path [21].

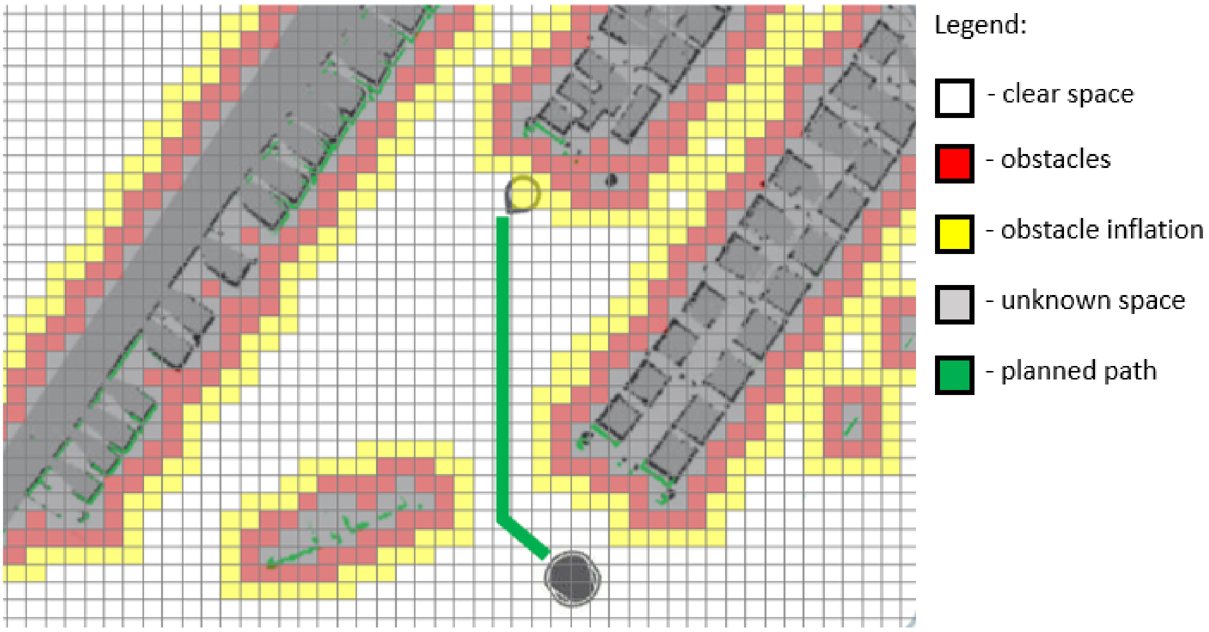


Figure 20: Grid costmap with planned path [21].

A complete and effective autonomous navigation system is achieved by ensuring that the robot has all mentioned subsystems implemented. It requires reliable means of locomotion, being capable of gathering sufficient information about itself and its surroundings through its perception system, locating itself and planning trajectories to navigate around its environment. Additionally, it is also necessary to implement motion control to convert the velocity and acceleration commands to motor inputs.

3 Robot Overview

In [22], a mobile robot with an anthropomorphic arm was developed. The work in this dissertation uses the same robot and builds on top of it.

The robot used was a differential drive mobile robot with two independently driven wheels and one free castor wheel. Its wheel distribution is represented by Figure 21. The driven wheels are equipped with an AS1050-0120 Beckhoff motor with incorporated incremental encoders each. The resolution of the encoders is 1.8° or 200 increments per revolution. The motors are coupled with a gearbox with a gear ratio of 5. The robot is also equipped with an anthropomorphic robot arm, but the use of it is not relevant to the work developed in this dissertation. Pictures of the robot can be seen in Figure 22 and Figure 23. The two independently driven wheels are shown on Figure 22 along with their respective motors, gearboxes, and encoders. The free castor wheel can be seen on Figure 23.

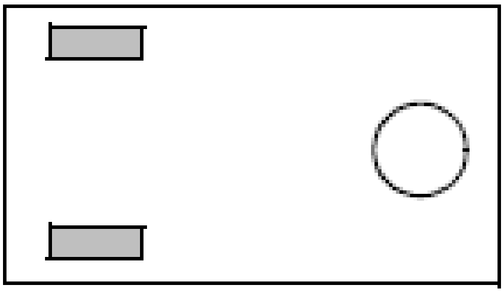


Figure 21: Wheel distribution of the robot.

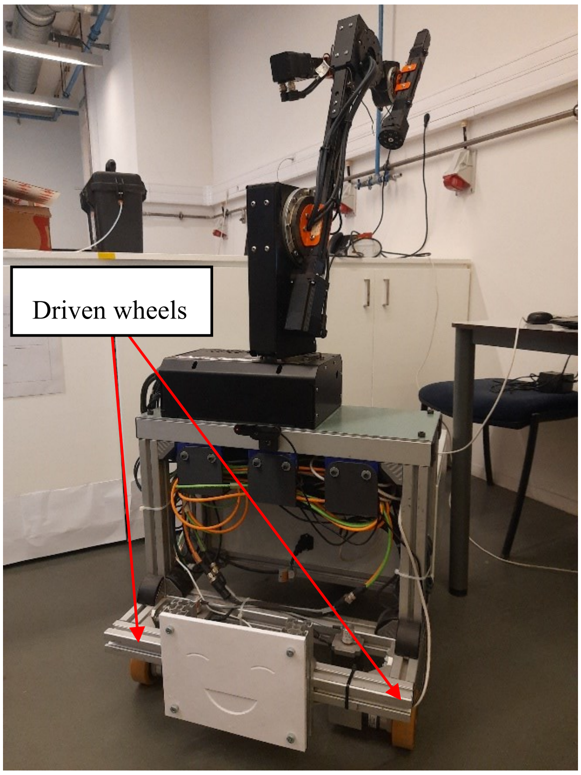


Figure 22: Front picture of the robot.

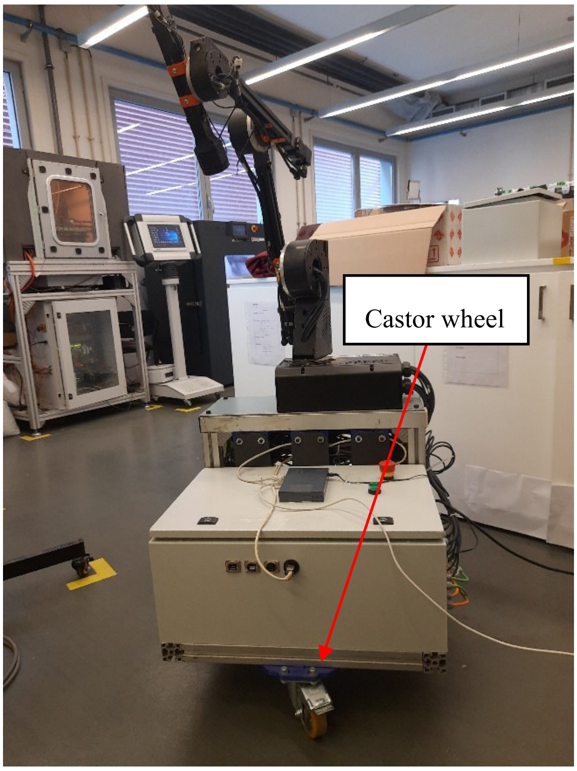


Figure 23: Back picture of the robot.

All the mechanical and electrical installation were already complete. The electrical cabinet can be seen in Figure 25. The robot does not have a battery. Instead, it is powered by connecting it to the domestic power grid using a large cord.

It contains three different power supplies, one with 48 V, one with 24 V and one with 5 V.

- The 48 V power supply is used to power 5 EtherCAT terminals (3 EL7047 terminals and 2 EL7041) that connect to 3 stepper motors used in the robot arm and the 2 stepper motors used for the robot's locomotion respectively.
- The 24 V power supply powers the EtherCAT EL9800 terminal, that powers 2 EtherCAT EL7037 modules that connect to 2 other stepper motors used in the robot arm. It also powers their encoders, the robot's industrial PC (IPC), the EtherCAT coupler EK1100, and the input (EL1008) and output terminals (EL2008).
- The 5 V power supply is used for three of the encoders in the robot arm, and the two encoders used for locomotion.

Figure 24 represents the schema of electrical power in the robot.

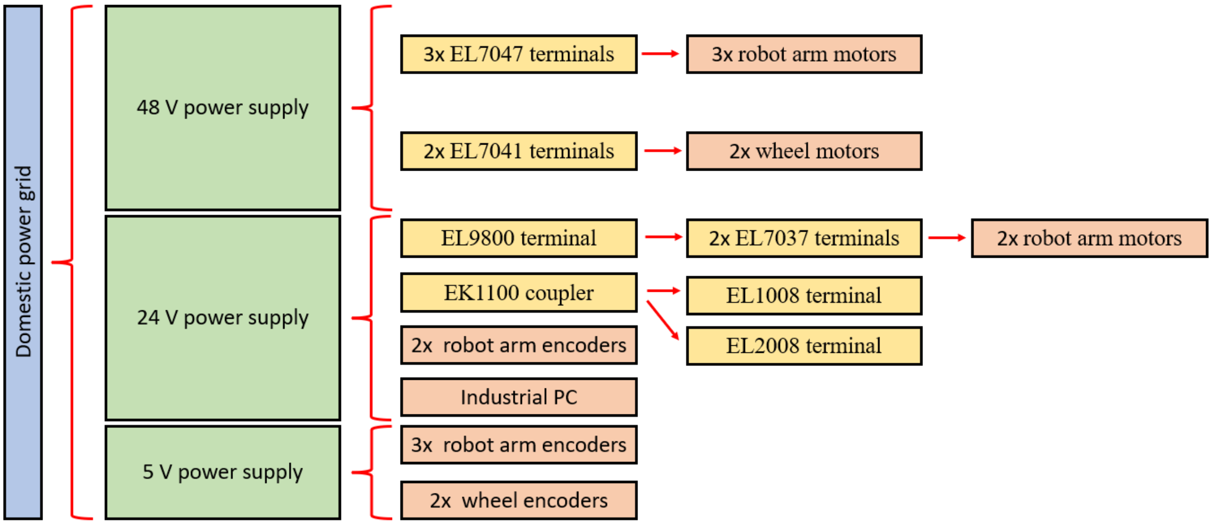


Figure 24: Electrical power schema of the robot.

The IPC used is a C6015-0010 from Beckhoff. It uses windows 10 as its operating system and is used to run TwinCAT 3, Beckhoff's software solution for PC-based control of automation systems.

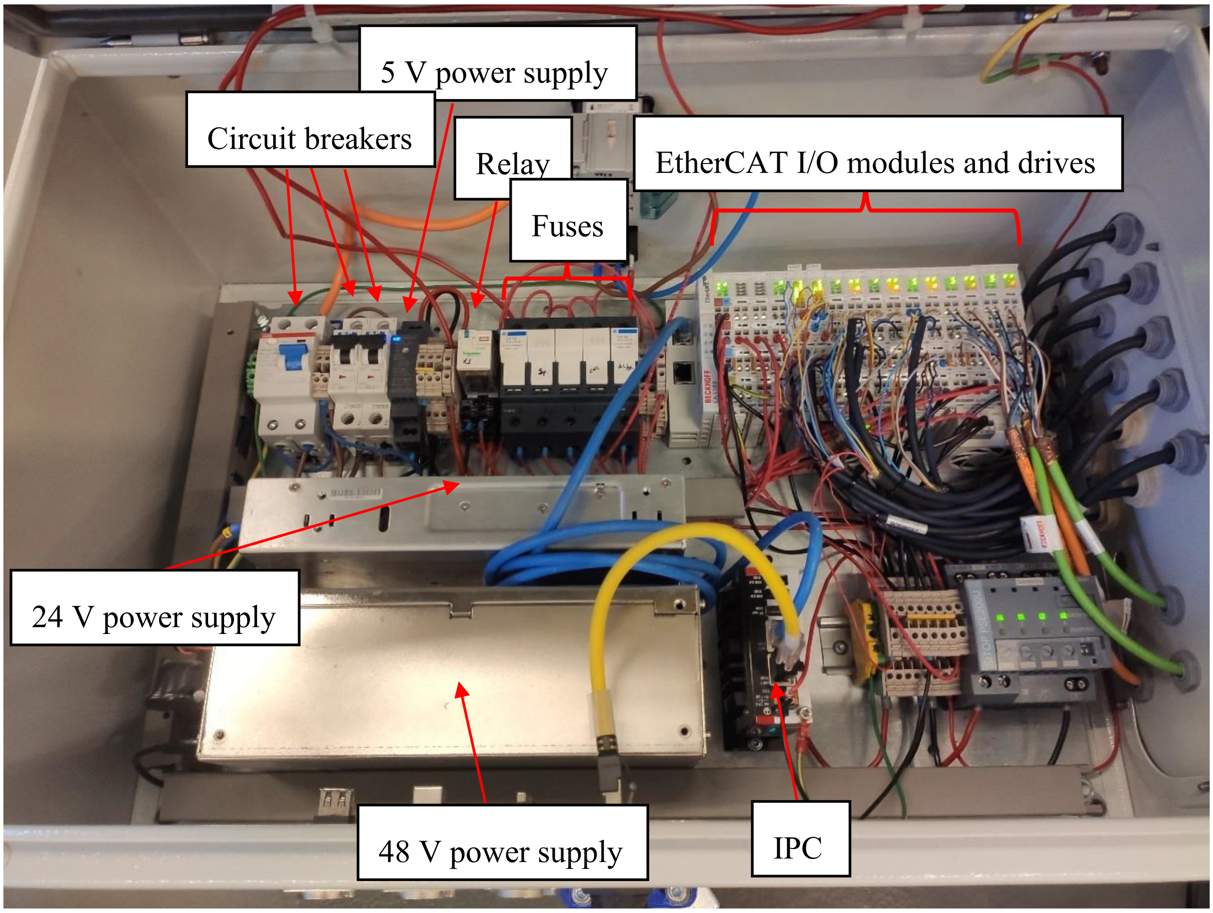


Figure 25: Electrical cabinet of the robot.

The robot was also equipped with an emergency button (Figure 26) that, when activated, turns off all circuits that send power to the motors, while keeping the command circuits powered, including the IPC and the EtherCAT communication.

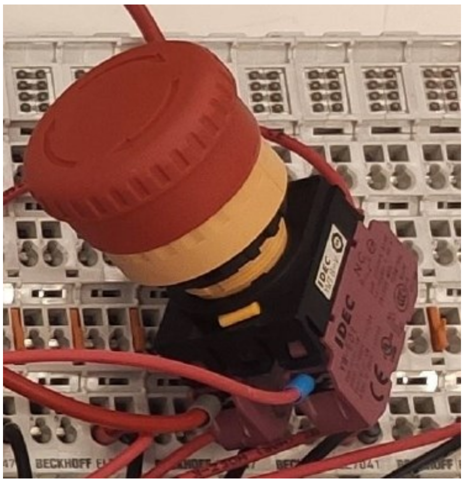


Figure 26: Emergency button used [22].

The command and control system was already partially developed utilizing Beckhoff's software TwinCAT 3. TwinCAT 3 is an automation software solution for PC-based control that can turn a PC with EtherCAT communication into a real time controller with multiple subsystems running simultaneously. It consists of a XAR (eXtended Automation Runtime) that runs the already programmed control modules and the XAE (eXtended Automation Engineering) that, using Microsoft Visual Studio, allows the user to setup, modify and program the control system to be used [23].

TwinCAT 3 utilizes multiple modules that communicate with each other to create a complete controller. There is a module for the overall configuration of the system (*SYSTEM*), a module for movement control (*MOTION*), a module for the PLC programming (*PLC*), a safety module (*SAFETY*), a module for C++ programming (*C++*), a module for system analytics (*ANALYTICS*), and a module for I/O configuration (*I/O*). These modules can be seen in Figure 27. These are the default modules and additional ones may be added and configured.

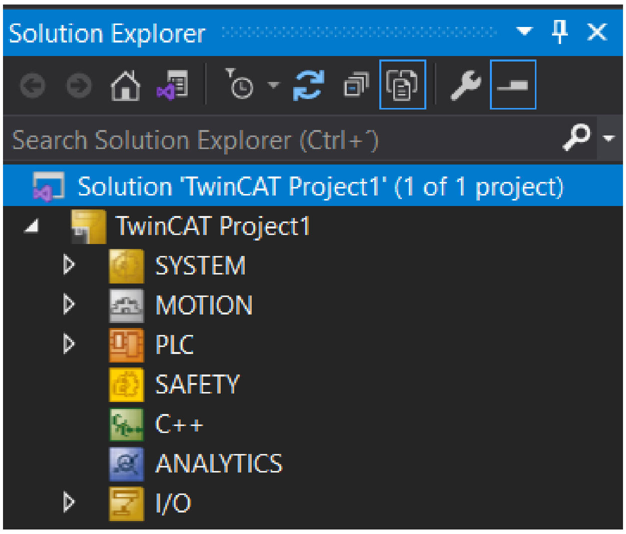
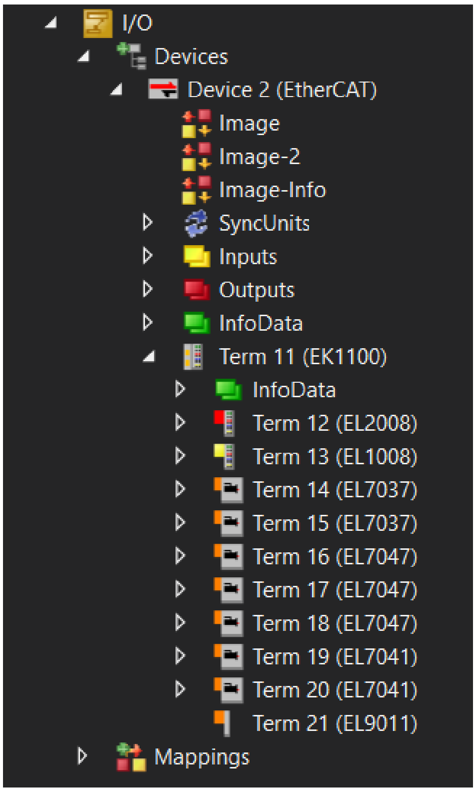

Solution Explorer
Search Solution Explorer (Ctrl+´)
Solution 'TwinCAT Project1' (1 of 1 project)
TwinCAT Project1
SYSTEM
MOTION
PLC
SAFETY
C++
ANALYTICS
I/O

Figure 27: The modules in TwinCAT 3.

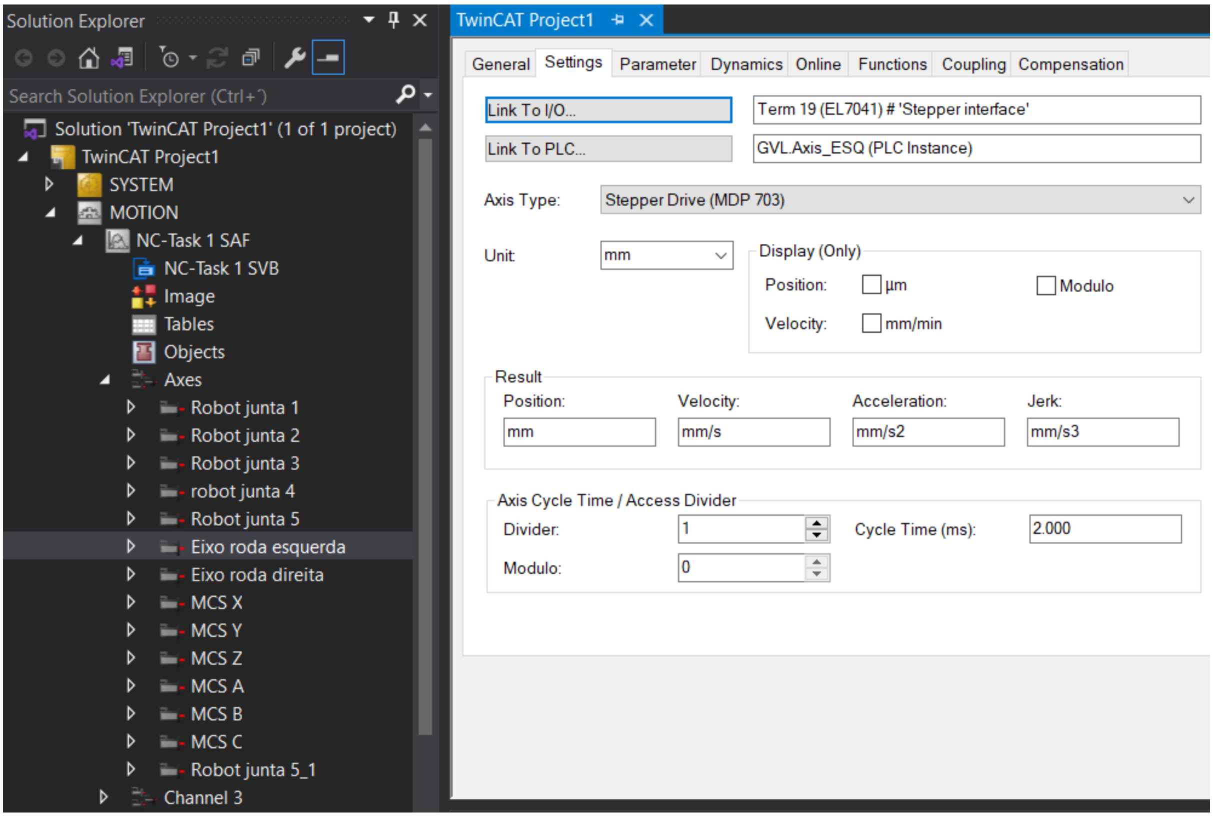
The *C++* and *ANALYTICS* modules were not utilized in this dissertation. The *MOTION* and *I/O* modules were already completely configured [22]. In the *I/O* module, the software recognized all connected EtherCAT devices (Figure 28). Additionally, the motor's drivers were configured, which includes inputting multiple motor properties, such as, maximum current, nominal voltage, number of steps per revolution, and the encoder's number of pulses per revolution [22].



```
◢ I/O
  ◢ Devices
    ◢ Device 2 (EtherCAT)
        Image
        Image-2
        Image-Info
      ▷ SyncUnits
      ▷ Inputs
      ▷ Outputs
      ▷ InfoData
      ◢ Term 11 (EK1100)
        ▷ InfoData
        ▷ Term 12 (EL2008)
        ▷ Term 13 (EL1008)
        ▷ Term 14 (EL7037)
        ▷ Term 15 (EL7037)
        ▷ Term 16 (EL7047)
        ▷ Term 17 (EL7047)
        ▷ Term 18 (EL7047)
        ▷ Term 19 (EL7041)
        ▷ Term 20 (EL7041)
          Term 21 (EL9011)
  ▷ Mappings
```

Figure 28: EtherCAT devices recognized by the I/O module.

In the *MOTION* module, the different movement axes can be configured. This module is responsible for the numerical control of the movement axes. It can use point-to-point control and generate the necessary signals to send to the respective I/O device. Figure 29 shows the configuration menu for a movement axis. It includes multiple tabs where different fields need to be configured correctly [22].



Link To I/O... Term 19 (EL7041) # 'Stepper interface'

Link To PLC... GVL.Axis_ESQ (PLC Instance)

Axis Type: Stepper Drive (MDP 703)

Unit: mm

Display (Only)

Position: □ µm □ Modulo

Velocity: □ mm/min

Result

Position:	Velocity:	Acceleration:	Jerk:
mm	mm/s	mm/s2	mm/s3

Axis Cycle Time / Access Divider

Divider: 1 Cycle Time (ms): 2.000

Modulo: 0

Figure 29: Configuration of an axis and association with I/O module.

The *SAFETY* module was also not utilized since its setup is very expensive and unnecessarily complex for this application, as this is merely a laboratory test robot and not a commercial product. Instead, the safety measures used were the previously mentioned emergency button, plus a programmed safety measure to guarantee that communication between the IPC and the Raspberry Pi is working successfully (described in Section 4.2). The *PLC* module already had software to ensure basic movements implemented. It allowed the robot to perform movements at specific velocities and for specific distances. However, these movements could not be changed flexibly once they started, the inputs used for movement were different from the outputs given by path planning algorithms, and no odometry was implemented. As such, it was not compatible with the work done in this dissertation to create an autonomous navigation system. Its changes are described in Chapter 4.

3.1 Hardware upgrades

As mentioned in Section 2.2, it is extremely important for a robot to be able to collect information about its surroundings for it to be able to navigate autonomously, safely, and successfully. Initially, the only sensors the robot was equipped with were encoders. Encoders are extremely important for the robot's odometry, allowing it to keep track of its position over time relative to an initial position by reading how much each wheel has rotated over time. However, it is still necessary for the robot to be able to obtain information about its environment, so that it can create a map, locate itself, avoid obstacles and reach specific goals.

A decision was made to utilize a Lidar module and a camera as exteroceptive sensors in this dissertation. Software to use the information obtained from these sensors and to aid in designing the navigation system was also researched. Various approaches to design autonomous navigation systems have been studied. Robot Operating System (ROS) is a software framework commonly used in the last 12 years [24, 25, 26, 27, 28]. However, most studies focus on the implementation of the systems, providing limited information about both the implementation process and the results, and not performing any parameter tuning in order to optimize performance. A small number of works address parameters' tuning of the ROS Navigation Stack, which depends heavily on the hardware used on the robot [29, 30, 31]. As such, ROS was selected to be used in this dissertation as it is a powerful and commonly used open source set of software libraries and tools to facilitate building robot applications.

Because of these factors, for extra computational power and to not overburden the IPC, a Raspberry Pi 4 Model B (Figure 30) with 2GB of RAM was also installed on the robot. In this robot, the Raspberry Pi has the function of utilizing the Lidar module and camera, receiving odometry information from the IPC's TwinCAT, running the navigation and localization algorithms through ROS, and sending velocity commands to the IPC's TwinCAT 3 software. This communication is done via an Ethernet cable connecting the Raspberry Pi and the IPC.

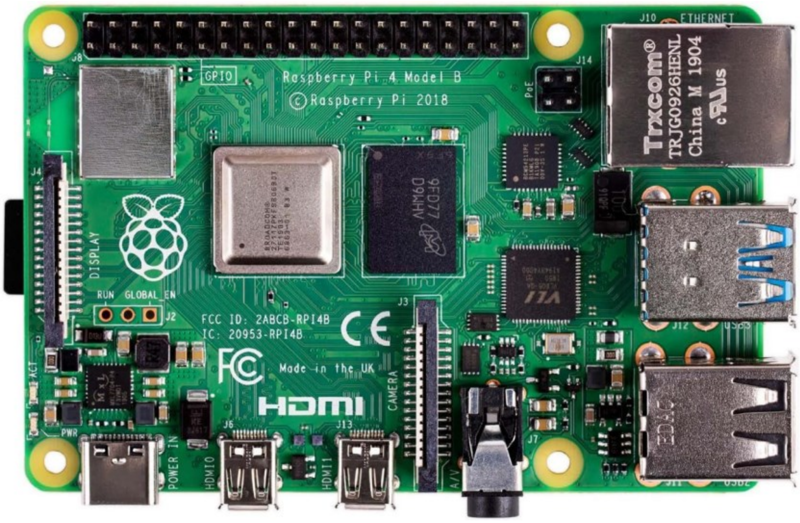


Figure 30: Raspberry Pi 4 Model B [32].

For the robot to sense its surroundings it was equipped with a Lidar module, as this is an efficient method of sensing distances to obstacles around the robot in a 2D plane, which can then be utilized for localization and navigation when paired with the odometry data. The Lidar module used was an OKdo Lidar LD06, which is a low-cost and small Lidar module made to be used with a Raspberry Pi and ROS. A picture of the module can be seen in Figure 31. Its properties are summarized in Table 1. Assuming the Lidar module is used at the default reading frequency of 10 Hz, it can detect 450 points over a 360° angle per revolution.

Figure 31: OKdo Lidar LD06 [33].

Table 1: Properties of the OKdo Lidar LD06.

Reading frequency	Ranging frequency	Scanning angle	Scanning range	Error
5-13 Hz	4500 Hz	360°	0.02-12 m	± 45 mm

A webcam was also installed with the intent of feeding visual information to the robot and use it to aid navigation. The webcam used was an AutoFocus 4Mpx 1440p with USB connection (Figure 32).

Figure 32: AutoFocus 4Mpx 1440p webcam [34].

To protect this extra equipment and attach it to the robot, additional 45 mm x 45 mm aluminium strut profiles were used and secured by T-nuts, collar screws and brackets, as used previously in the robot for its structure.

In [35], a 3D printable case model was obtained (Figure 33) and was used to protect and hold the Raspberry Pi. It was modified to include a mount for the Lidar module, a surface to mount to the strut profile, and a large gap underneath to increase air flow and reduce overheating (Figure 34 and Figure 35). Other air flow cuts were also changed.

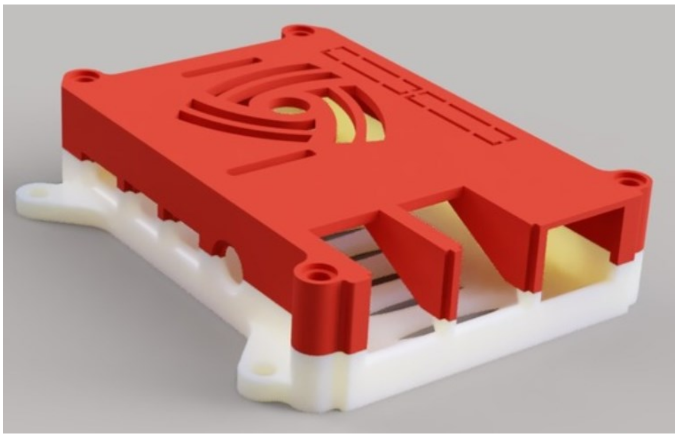


Figure 33: Raspberry Pi 4 case from Thingiverse.com [35].

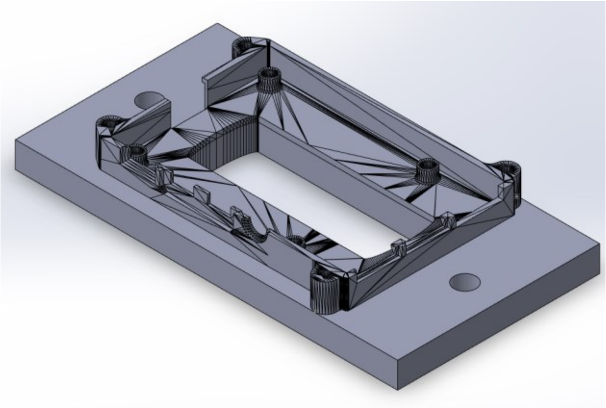


Figure 34: Bottom part of modified Raspberry Pi case.

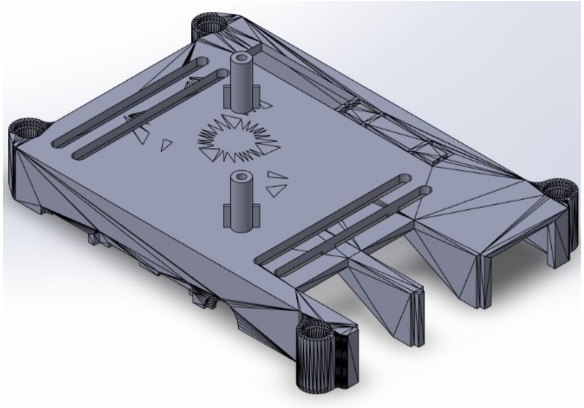


Figure 35: Top part of modified Raspberry Pi case.

The pins on top of the case in Figure 35 used to secure the Lidar module in place were printed separately and screwed on the case due to a much greater ease of 3D printing those shapes separately.

An additional 3D printed part was also made to act as a bumper and protect both the Lidar module and the Raspberry Pi against possible collisions (Figure 36). It also has holes to secure the part to the strut profiles using T-nuts and collar screws.

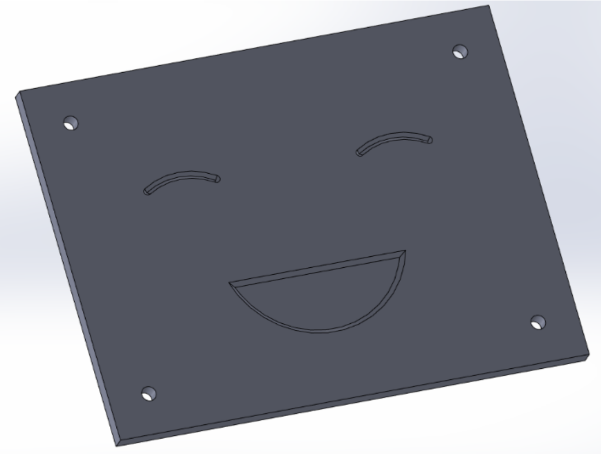


Figure 36: Bumper part to protect the Lidar and the Raspberry.

The full installation can be seen 3D modelled using Solidworks on Figure 37 and Figure 38 and in a live photograph in Figure 39. Note that the Lidar module is located slightly below the bumper mechanism so it can have a larger field of vision and not be obstructed by the bumper. Due to the robot having an anthropomorphic arm on top of it; and wheels, motors, and gearboxes under it, it was not possible for the Lidar module to have access to full 360° ranging. To eliminate the dead zones caused by the motors, gearboxes, and wheels, it would be necessary to install an additional Lidar module on the robot on a different location that would have full vision of the current dead zones.

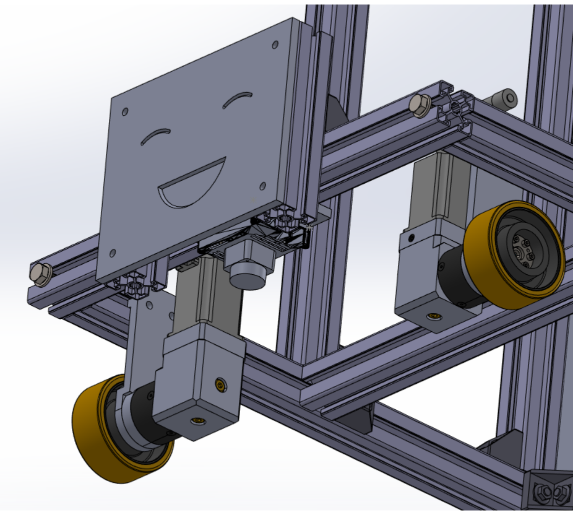


Figure 37: Installation of the new parts. View one.

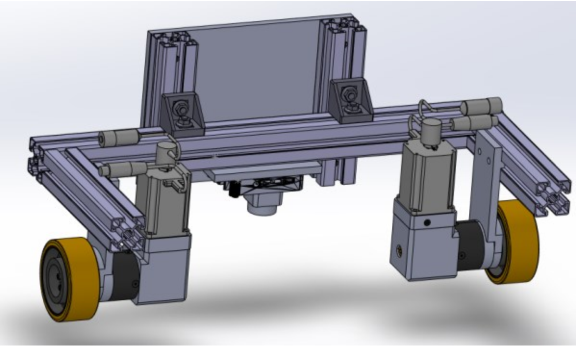


Figure 38: Installation of the new parts. View two.

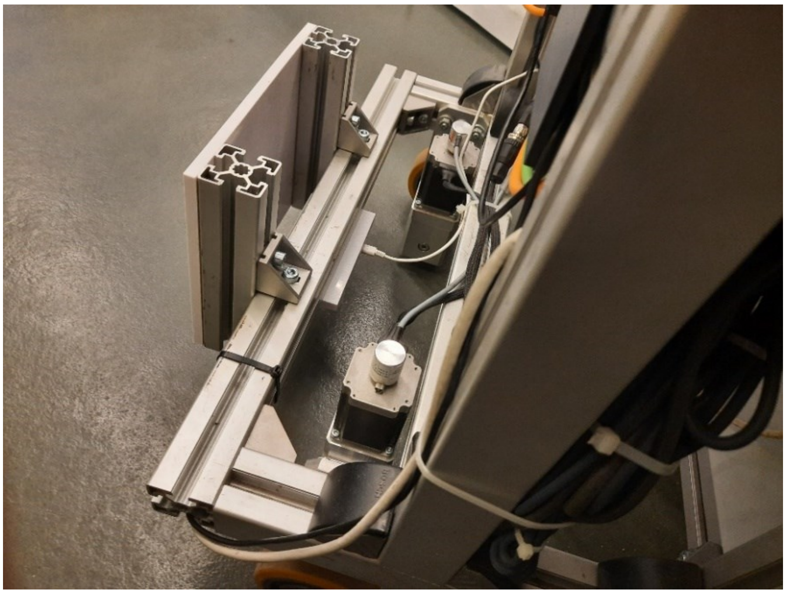


Figure 39: Photograph of the installation of the new parts.

Figure 40 represents a simplified diagram including the robot's processing units, sensors, and actuators and how they are connected to each other.

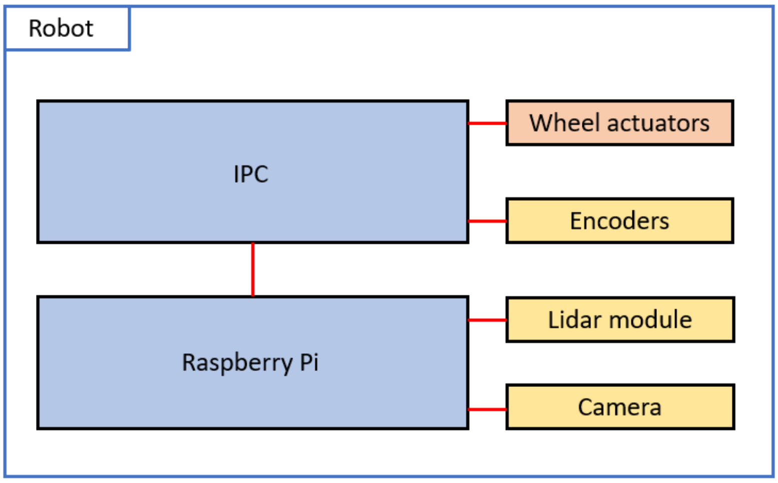


Figure 40: Robot's simplified hardware diagram.

4 Motion Control System

The software used for the control and command of the system is TwinCAT 3. The *PLC* module is where the system's logic is implemented, making it responsible for the overall control of the motion system of the robot.

The *PLC* module interconnects the other TwinCAT 3 modules, managing the flow of information exchanged between them. This module is programmable according to the IEC 61131-3 international standard for programmable PLCs, which supports the following programming languages:

- Ladder diagram (LD);
- Function block diagram (FBD);
- Sequential function chart (SFC);
- Structured text (ST);
- Instruction list (IL).

In the case of this dissertation's robot, every program in the PLC module was written in structured text (ST).

Aside from creating programs, the PLC module can also create function blocks, lists of global variables (GVLs), and visual interfaces that can interact with global variables and programs.

The objective is for TwinCAT to receive velocity commands from the ROS system in the Raspberry Pi while simultaneously sending odometry information to the Raspberry Pi's ROS system. This flow of information is shown in Figure 41.

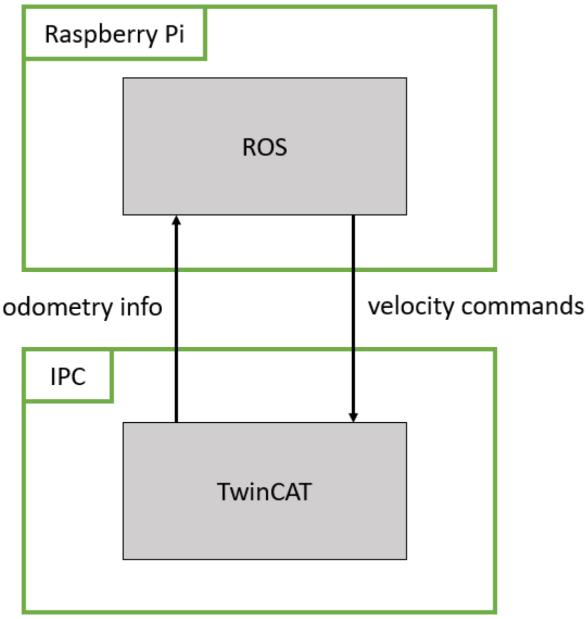


Figure 41: Flow of information between the IPC and the Raspberry Pi.

Therefore, it is necessary to create a program in the PLC module that can take these velocity commands and make the robot move accordingly. The program should also calculate the robot's odometry over time. The ROS Navigation Stack uses the *move_base* package, which generates the following velocity commands: velocity in x (V_x), velocity in y (V_y), and velocity in θ (V_θ). These velocities are relative to the robot's current position and in accordance with Figure 42.

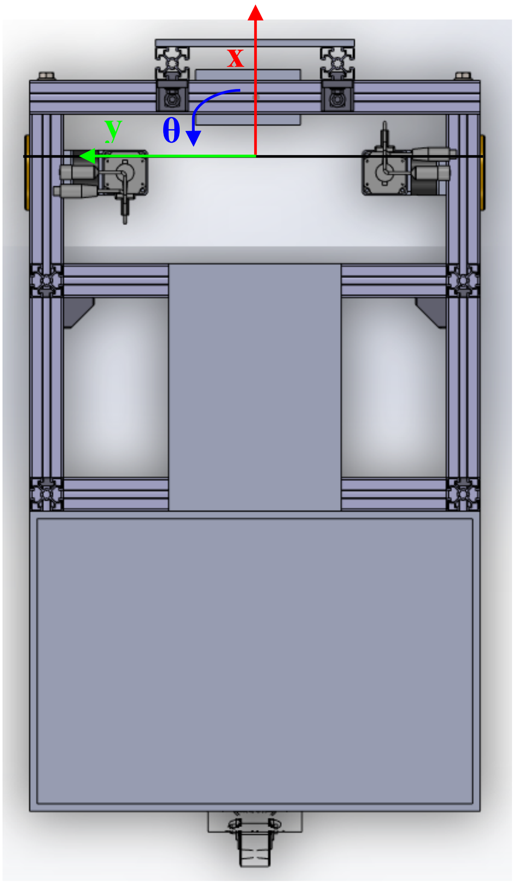


Figure 42: Robot's movement axes.

In the case of this specific robot, since it is a differential drive robot, it can only have velocities in the x and θ directions, but not in the y direction. Because there are no other possible velocities that a differential drive robot can take, V_x and V_θ can also be called linear and angular velocity respectively.

It was necessary to create a program in TwinCAT that can take these commands and make the wheels move at the appropriate velocities until a new velocity command is sent. To achieve this, the *MC_MoveVelocity* function block shown in Figure 43 was used. This function block starts a continuous movement with specified velocity and direction [36].

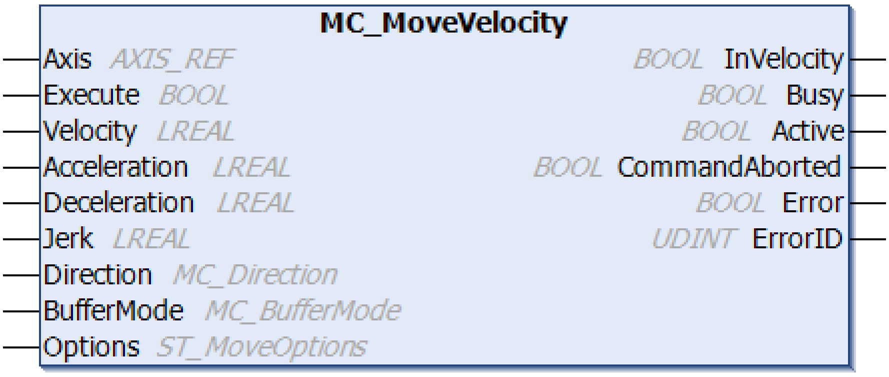


Figure 43: TwinCAT's *MC_MoveVelocity* function block [36].

However, this function block can only receive a positive velocity as an input, which means an additional function block must be used for when the velocity of any of the wheels is zero. The *MC_Halt* (Figure 44) function block stops an axis' movement with a defined breaking ramp [37].

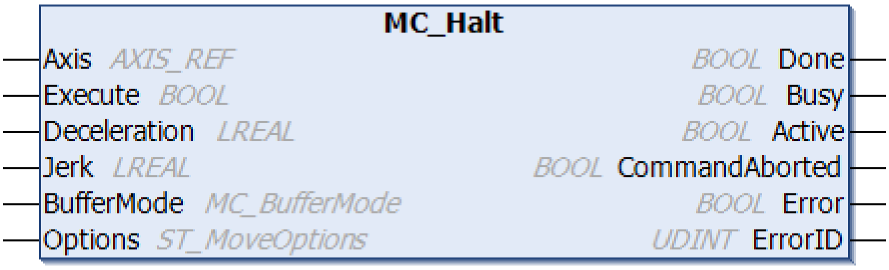


Figure 44: TwinCAT's *MC_Halt* function block [37].

4.1 Differential drive kinematics

Since TwinCAT 3 will control the movement of both wheel axes independently, it is necessary to transform the linear velocity and angular velocity that it receives in velocity for the left wheel, and velocity for the right wheel. It is also necessary to obtain odometry information about the robot using information from the encoders in each of the axes.

TwinCAT already had its *MOTION* module configured with information about the motors and transmission systems used. Because of this it can use wheel velocities in mm/s and read information from the encoders in mm automatically. Therefore, it is only necessary to transform V_x and V_θ in V_R (velocity of right wheel) and V_L (velocity of left wheel); and δ_R and δ_L in δ_x, δ_y and δ_θ; δ meaning the variation of position between each cycle.

In differential drives, linear and angular velocities are given as a function of the velocities of each wheel by Equations (4.1) and (4.2) [38].

$$V_x = \frac{V_R + V_L}{2} \tag{4.1}$$

$$V_\theta = \frac{V_R - V_L}{B} \tag{4.2}$$

Where: B, is the axial distance between the driven wheels.

By adding and rearranging these two equations, Equations (4.3) and (4.4) can be obtained.

$$V_R = V_x + \frac{V_\theta \cdot B}{2} \tag{4.3}$$

$$V_L = (2 \cdot V_x) - V_R \tag{4.4}$$

Since we can obtain the increments in position of each wheel in mm from the encoder data in TwinCAT, linear and angular displacements of the robot are given by Equations (4.5) and (4.6).

$$\delta_x = \frac{\delta_R + \delta_L}{2} \tag{4.5}$$

$$\delta_\theta = \frac{\delta_R - \delta_L}{B} \tag{4.6}$$

The length of the red arc in Figure 45 is given by δ_x. The size of the arc in the figure is exaggerated for purposes of easier understanding. The linear distance, *d*, between the two points (green line), can be obtained with the trigonometric relations given in Equations (4.7) and (4.8).

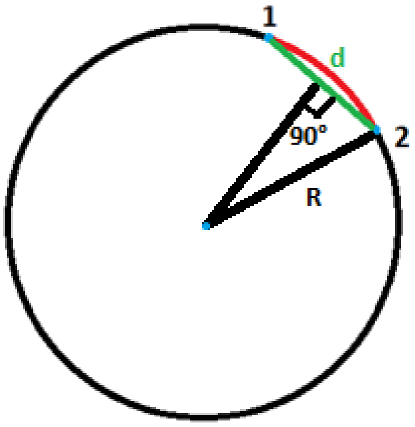


Figure 45: Arc travelled by the robot between two points.

$$d = 2 \cdot R \cdot \sin\left(\frac{\delta_\theta}{2}\right) \tag{4.7}$$

$$R = \frac{\delta_x}{\delta_\theta} \tag{4.8}$$

Equation (5.9) can be obtained by adding the previous two equations.

$$d = 2 \cdot \frac{\delta_x}{\delta_\theta} \cdot \sin\left(\frac{\delta_\theta}{2}\right) \tag{4.9}$$

If between two cycles, the robot moved in a straight line, Equation (4.10) is used instead of Equation (4.9), since δ_θ is 0.

$$d = \delta_x \tag{4.10}$$

The position of the robot in relation to its starting frame can be obtained over time by adding the linear and angular displacements between the last two cycles (k+1 and k) to the last calculated position of the robot, using Equations (4.11), (4.12) and (4.13). The angle used to calculate *x* and *y* can be the average between the angle in each of the last two cycles to reduce errors since the movement happened between those two positions. Initial position is (0, 0, 0).

$$\theta_{k+1} = \theta_k + \delta_\theta \tag{4.11}$$

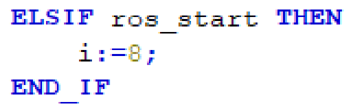
$$x_{k+1} = x_k + d \cdot \cos\left(\frac{\theta_k + \theta_{k+1}}{2}\right) \tag{4.12}$$

$$y_{k+1} = y_k + d \cdot sin\left(\frac{\theta_k + \theta_{k+1}}{2}\right) \qquad (4.13)$$

Since usually these cycles are calculated at very high frequencies, the distance between two points in an arc given by Equation (4.9) can be approximated by the length of the arc itself with negligible error. Lower cycle frequency, and higher linear and angular velocities result in a higher error when using this approximation. In the case of the robot used in this dissertation, the cycle time of the program is 10 ms. At the robot's maximum linear velocity (V_x) of 0.3 m/s and maximum angular velocity (V_x) of 0.7 rad/s, the length of the arc travelled between each cycle (δ_x) is 3 mm and the angular displacement (δ_θ) is 0.007 rad. Using these values in Equation (4.9) results in a linear distance travelled (d) of approximately 2.999994 mm. Which means that in this case, the maximum error obtainable by using this approximation is around 0.0002 %.

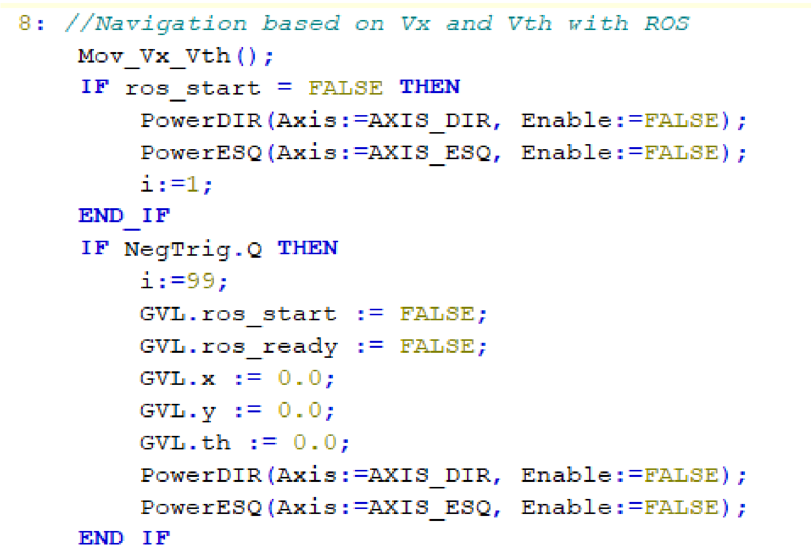
4.2 Implementation in TwinCAT 3

Multiple different functions of the robot can be called from the program *MAIN* which is always running by default. The program with the function of sending movement commands to the motors and calculating the odometry of the robot (*Mov_Vx_Vth*) is called in *MAIN* when communication with ROS happens. To achieve this, the ROS program will set the global variable *ros_start* to true, which causes *MAIN* to run what is in the "Case" function for i=8 (Figure 46). This makes TwinCAT run the program *Mov_Vx_Vth* until *ros_start* is set to false or the emergency button is pressed (Figure 47). Both the *MAIN* and *Mov_Vx_Vth* programs are written in Structured Text.



```
ELSIF ros_start THEN
    i:=8;
END_IF
```

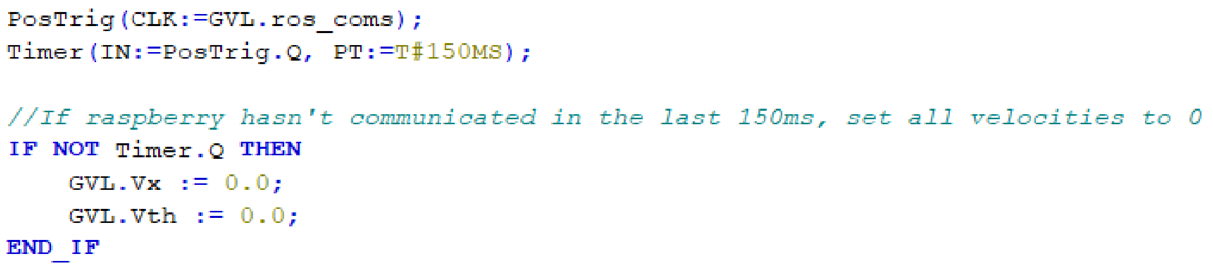
Figure 46: Condition to call the movement command and odometry program.



```
8: //Navigation based on Vx and Vth with ROS
    Mov_Vx_Vth();
    IF ros_start = FALSE THEN
        PowerDIR(Axis:=AXIS_DIR, Enable:=FALSE);
        PowerESQ(Axis:=AXIS_ESQ, Enable:=FALSE);
        i:=1;
    END_IF
    IF NegTrig.Q THEN
        i:=99;
        GVL.ros_start := FALSE;
        GVL.ros_ready := FALSE;
        GVL.x := 0.0;
        GVL.y := 0.0;
        GVL.th := 0.0;
        PowerDIR(Axis:=AXIS_DIR, Enable:=FALSE);
        PowerESQ(Axis:=AXIS_ESQ, Enable:=FALSE);
    END IF
```

Figure 47: Case function for i=8 in MAIN.

The *Mov_Vx_Vth* program is included in Appendix A.1. It utilizes the *MC_Halt* and *MC_MoveVelocity* function blocks mentioned previously and implements movement command and odometry according to the kinematics discussed in Section 4.1. It also adds an additional safety measure, to make sure that TwinCAT 3 and ROS are communicating successfully (Figure 48). Every time ROS sends information to TwinCAT, it will set *ros_coms* to true and subsequently false; this activates a rising edge in TwinCAT that is the input to an off delay timer of 150 ms. In the case of no communication happening for 150 ms between ROS and TwinCAT, all velocities will be set to 0.



```
PosTrig(CLK:=GVL.ros_coms);
Timer(IN:=PosTrig.Q, PT:=T#150MS);

//If raspberry hasn't communicated in the last 150ms, set all velocities to 0
IF NOT Timer.Q THEN
    GVL.Vx  := 0.0;
    GVL.Vth  := 0.0;
END_IF
```

Figure 48: Safety measure against communication failure.

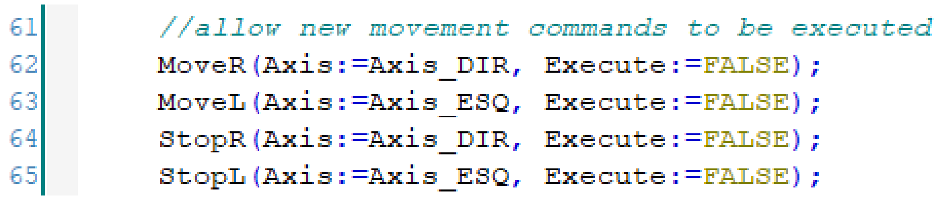
Since the *MC_MoveVelocity* function block only takes velocity inputs larger than 0 and a forward or negative direction value, it is necessary to have an IF condition for all nine possible combinations of positive, null, and negative velocity of each wheel.

The odometry is calculated using the difference between the absolute values of the encoders in the current and last cycle.

The velocity command variables *Vx* and *Vth*, and the odometry variables *x*, *y* and *th*, are all global variables to allow them to be written and read by ROS. The program also reads the current velocity of each wheel and calculates the linear and angular velocities the robot is moving at currently, as they are also necessary for ROS.

Additionally, during the first cycle of the program, odometry values are set to (0, 0, 0), the motors are powered, and their axes are reset.

Because the *MC_MoveVelocity* and *MC_Halt* function blocks require a rising edge signal of the *Execute* boolean input, it is also necessary to add the following lines present in Figure 49. These lines feed the mentioned function blocks an input of "Execute=FALSE". This allows the robot's movement to be flexible and able to change velocities at any moment.



```
//allow new movement commands to be executed
MoveR(Axis:=Axis_DIR, Execute:=FALSE);
MoveL(Axis:=Axis_ESQ, Execute:=FALSE);
StopR(Axis:=Axis_DIR, Execute:=FALSE);
StopL(Axis:=Axis_ESQ, Execute:=FALSE);
```

Figure 49: Necessary lines to allow new movement commands.

4.3 Odometry quality testing and correction

The quality and errors of the implemented odometry were tested using the following methods:

- Comparing the odometry values with the geometry of the trajectory of the robot recorded by attaching a pen to it in its rotation axis;
- Utilizing a visual method comparing the points the Lidar detects in relation to the odometry frame while the robot moves.

A part to attach the pen to the robot in the position of its frame was modelled and 3D printed. It can be seen 3D modelled in Figure 50 and installed on the robot in Figure 51. It has a geometry that only allows the pen to move in the vertical axis. A spring was inserted inside to guarantee that there is tension between the pen's ballpoint and the ground.

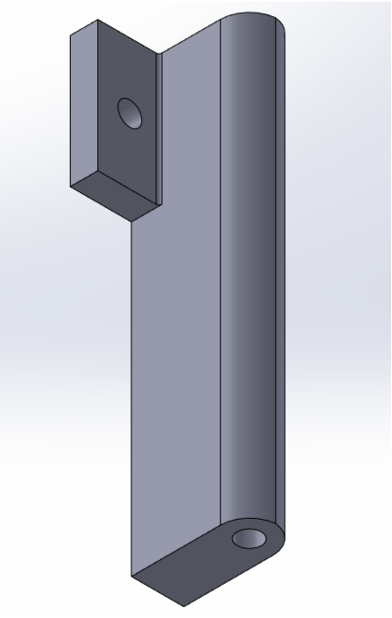


Figure 50: 3D model of the pen gripper.

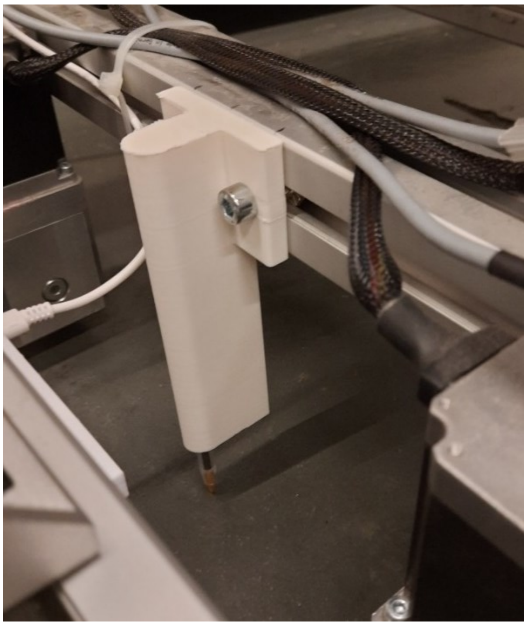


Figure 51: Pen gripper assembled in the robot.

To make the robot perform specific movements that can be repeated, another program in TwinCAT was also written in ST. This program includes the odometry functionalities implemented in Section 4.2, but the movements it performs are done for a set distance and relative to the position at the start of the movement. To achieve this, it utilizes the *MC_MoveRelative* function block shown in Figure 52. It can be used to perform linear, circular, or rotational trajectories. The program is available in Appendix A.2.

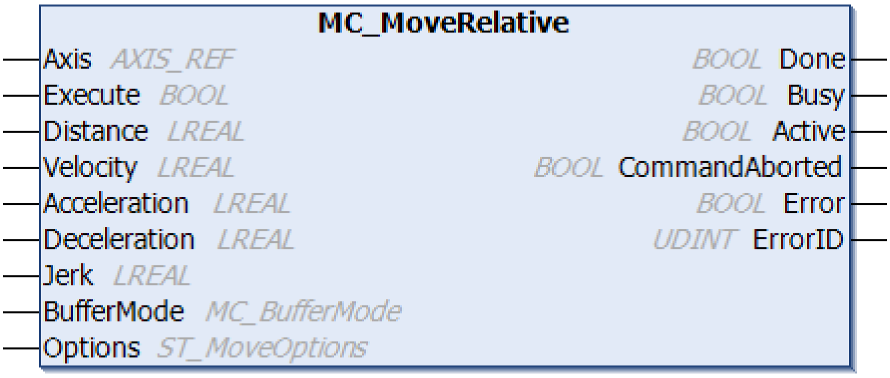


Figure 52: TwinCAT’s *MC_MoveRelative* function block [39].

The quality of the linear odometry was tested by attaching the pen to the robot’s axis and measuring the length of the resulting line after instructing the robot to move 1 m forward (Figure 53). This test was done at a linear velocity of 0.025 m/s and repeated three times. The results of the linear odometry tests are shown in Table 2.

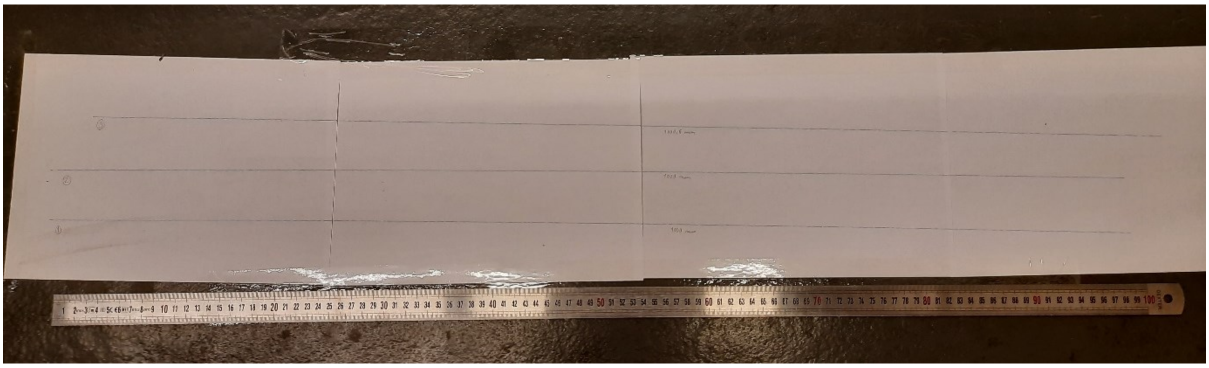


Figure 53: Lines produced by the robot and ruler used to measure them.

Table 2: Results of the linear odometry test.

	x **given by the odometry**	**Length of line**	**Error (x_{odom}-$x_{measured}$)**	**%Error (Error / $x_{measured}$)**
Test 1	1000.0 mm	1009.5 mm	-9.5 mm	-0.94 %
Test 2	1000.0 mm	1011.0 mm	-11.0 mm	-1.09 %
Test 3	1000.0 mm	1010.5 mm	-10-5 mm	-1.04 %
Average	1000.0 mm	1010.3 mm	-10.3 mm	-1.02 %

The results of the linear odometry test show that the robot’s odometry was giving values lower than the measured values by around 1.03 %. To counteract this error, it is necessary to change the scaling factor parameter of the motor’s encoder. This scaling factor gives how much the wheels move in mm per increment of the encoder. The scaling factor is given by Equation 4.13.

$$SF = \frac{\pi \cdot D}{4 \cdot e} \cdot i \tag{4.13}$$

Where:

SF, is the scaling factor;

D, is the diameter of the wheel;

i, is the gear ratio of the transmission system;

e, is the number of encoder increments per rotation.

To correct the linear odometry of the robot, it is necessary to divide the scaling factor by (1+%Error) (Equation 4.14). Since e and i are constant, this is the same as dividing the wheel diameter by the same denominator. The original SF used in this robot was 0,015340 mm/Inc. This was obtained using a wheel diameter of 100 mm.

$$SF_{corrected} = \frac{SF}{1 + \%Error} \tag{4.14}$$

In this case, a corrected *SF* of 0.015498 mm/Inc is obtained. This equates to considering the wheel diameter to be around 101.0 mm. The same linear odometry tests were repeated using the corrected scaling factor. The results are shown in Table 3.

Table 3: Results of the linear odometry tests using the corrected scaling factor.

	x given by the odometry	**Length of line**	**Error (x_{odom}-x_{measured})**	**%Error (Error / x_{measured})**
Test 1	1000.0 mm	1000.0 mm	0.0 mm	0.0 %
Test 2	1000.0 mm	1000.0 mm	0.0 mm	0.0 %
Test 3	1000.0 mm	1000.5 mm	-0.5 mm	-0.05 %
Average	1000.0 mm	1000.2 mm	-0.2 mm	-0.02 %

The results now show an average error of around 0.02 % or 0.2 mm, which are within the reading error of the ruler used in these tests to measure the line.

An angular odometry test was then performed. This test utilized the already corrected scaling factor. The robot was programmed to perform an in-place rotation of 360°. A line parallel to the wheel's initial and final positions was drawn on a piece of paper on the floor (Figure 54 and Figure 55). The angle between these two lines was measured and compared to the angle given by the robot's odometry. This test was done at an angular velocity of 0.182 rad/s and repeated three times. Its results are shown in Table 4.

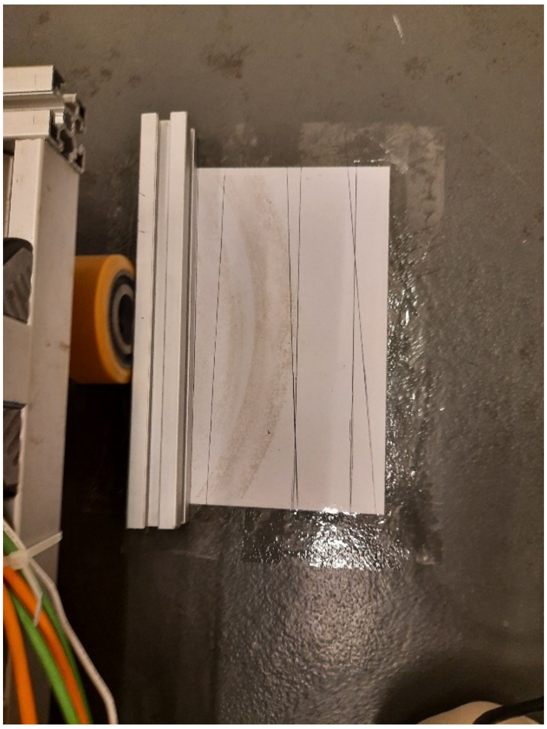


Figure 54: Methodology used to draw the lines parallel to the robot's wheels.

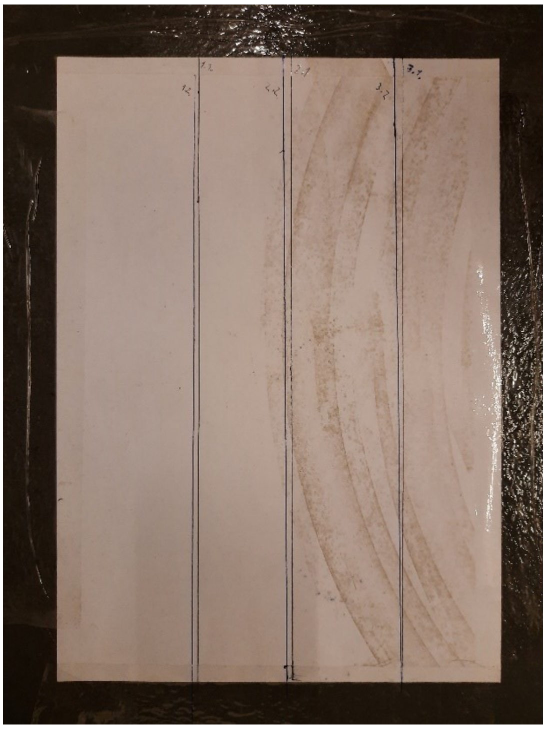


Figure 55: Lines drawn during three tests.

Table 4: Results of the first series of angular odometry tests.

	θ given by the odometry	**Angle measured**	**Error (θ_{odom} – θ_{measured})**	**%Error (Error / θ_{measured})**
Test 1	360.0°	363.1°	-3.1°	-0.85 %
Test 2	360.0°	362.9°	-2.9°	-0.80 %
Test 3	360.0°	361.8°	-1.8°	-0.50 %
Average	360.0°	362.6°	-2.6°	-0.72 %

Angular displacement is given by Equation 4.6 in Section 4.1. Considering that the linear odometry is already corrected, angular odometry can only be corrected by changing the value of the distance between wheels (B), which is inversely proportional to the angular displacement. It can be corrected by utilizing Equation 4.15.

$$B_{corrected} = B \cdot (1 + \%Error) \tag{4.15}$$

In this case, the measuring error was considered too large in comparison with the odometry error to take any conclusions. Because of this, a new series of tests was conducted by commanding the robot to perform a 1080° in-place rotation with the same angular velocity of 0.182 rad/s. This larger movement results in an increased odometry error while maintaining similar measurement errors, which allows for a more accurate percentual error to be calculated. Results of these tests are shown in Table 5.

Table 5: Results of the second series of angular odometry tests.

	θ given by the odometry	**Angle measured**	**Error ($\theta_{\text{odom}} - \theta_{\text{measured}}$)**	**%Error (Error / θ_{measured})**
Test 1	1080.0°	1089.1°	-9.1°	-0.84 %
Test 2	1080.0°	1090.0°	-10.0°	-0.92 %
Test 3	1080.0°	1088.5°	-8.5°	-0.78 %
Average	1080.0°	1089.2°	-9.2°	-0.84 %

By applying the average error obtained from these tests to Equation 4.15, a corrected distance between wheels of 545.38 mm is calculated. The same angular odometry tests were repeated using the corrected value of B. The results are shown in Table 6.

Table 6: Results of the angular odometry tests using the corrected value of the distance between the wheels.

	θ given by the odometry	**Angle measured**	**Error ($\theta_{\text{odom}} - \theta_{\text{measured}}$)**	**%Error (Error / θ_{measured})**
Test 1	1080.0°	1079.2°	+0.8°	+0.07 %
Test 2	1080.0°	1079.8°	+0.2°	+0.02 %
Test 3	1080.0°	1080.1°	-0.1°	-0.01 %
Average	1080.0°	1079.7°	+0.3°	+0.03 %

In a 1080° trajectory, the results now indicate an average error of 0.03 percent or 0.03°, which is within the repeatability errors of the tests. As a result, the value of 545.38 mm for the corrected distance between wheels is accepted.

An additional test was performed to evaluate the overall accuracy of the odometry, including both the linear and angular odometry. This test consisted in the robot performing a 90° arc with a radius of $\frac{\sqrt{2}}{2}$ m. The robot is expected to move from the position (0 m, 0 m, 0 rad) to ($\frac{\sqrt{2}}{2}$ m, $\frac{\sqrt{2}}{2}$ m, $\frac{\pi}{2}$ rad), which is approximately (707.1 mm, 707.1 mm, 1.571 rad) and for the distance between the two positions to be 1 m. As it follows this trajectory, the robot uses the previously mentioned pen positioned in its rotational axis to record its trajectory on paper taped to the floor. The circular arcs and auxiliary lines used to measure these distances can be seen in Figure 56. The results of the tests are shown in Table 7.

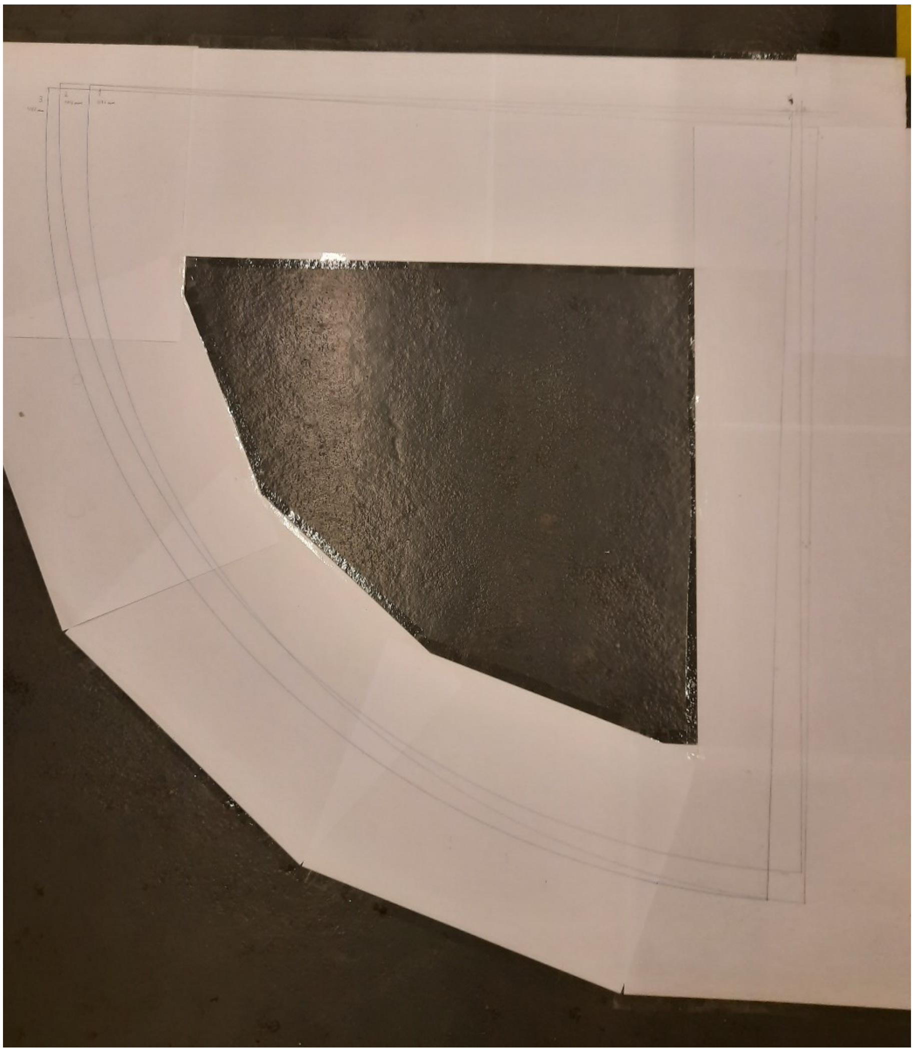


Figure 56: Odometry tests when performing a circular arc.

Table 7: Result of the odometry tests when performing a circular arc.

	Odometry vector	Distance between initial and final point	*x*	*y*	*θ*
Test 1	(706.9 mm, 707.4 mm, 1.571 rad)	1006 mm	706.5 mm	712.5 mm	90.9°
Test 2	(707.4 mm, 706.9 mm, 1.571 rad)	1006 mm	702 mm	720 mm	90.7°
Test 3	(706.9 mm, 707.4 mm, 1.571 rad)	1007 mm	723 mm	721 mm	88.5°
Average	(707.1 mm, 707.2 mm, 1.571 rad)	1006.3 mm	710.5 mm	717.8 mm	90.0°

This test's results show much larger errors than the previous tests. It is suspected that this is caused by the following factors:

- The pen that recorded the trajectory is misaligned with the robot's rotational axis by around 7 millimetres, as shown in a test where the robot performed a purely rotational movement;
- There is a large error when tracing the auxiliary lines perpendicular to the arc's initial and final positions. These lines were used to measure the distance in x and y, and the angle θ.

As such, this test was considered too faulty to draw conclusions regarding the quality of the robot's odometry.

4.3.1 Additional linear odometry visual test

Visual tests were also conducted by visualizing the points of obstacles seen by the Lidar module over time in relation to the odometry frame using *RViz* (a visualization tool for ROS) while driving the robot. The larger the odometry errors, the thicker these obstacles should become in the visualization the more the robot drives around.

A visual test of the linear odometry was made by analysing the thickness of the set of points of a wall seen by the Lidar as the robot moves closer to it in a straight line at a velocity of 25 mm/s. The test was conducted by starting recording at an initial time where only 1 cycles of points measured by the Lidar could be observed. Then, the robot stayed in place for 15 s to accumulate more points measured by the Lidar module and to be able to observe the Lidar's measurement error. At 15 s, the robot's movement started. At 55 s the robot reached the goal and stopped moving.

Results at t=0 s, t=15 s, t=16 s, t=40 s and t=60 s can be observed in Figure 57, Figure 58, Figure 59, Figure 60 and Figure 61 respectively. The squares of the grid have a width of 1 m.

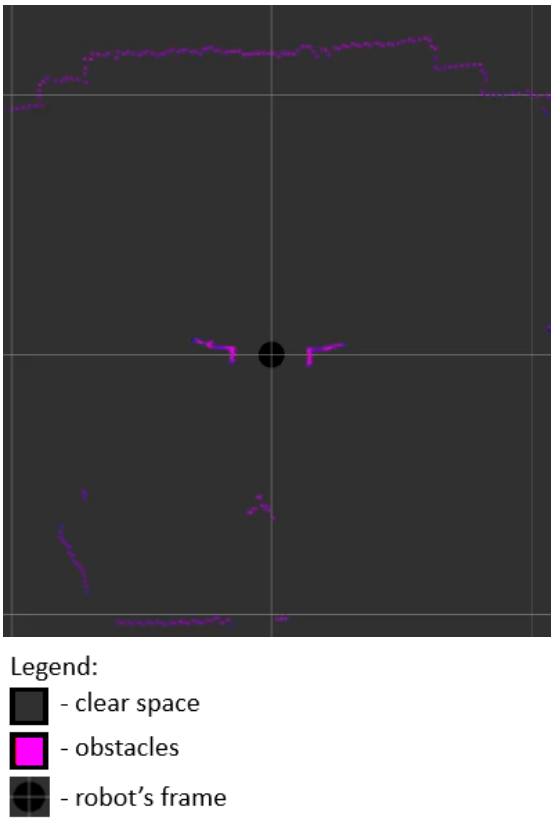


Figure 57: Visual linear odometry test at t=0 s.

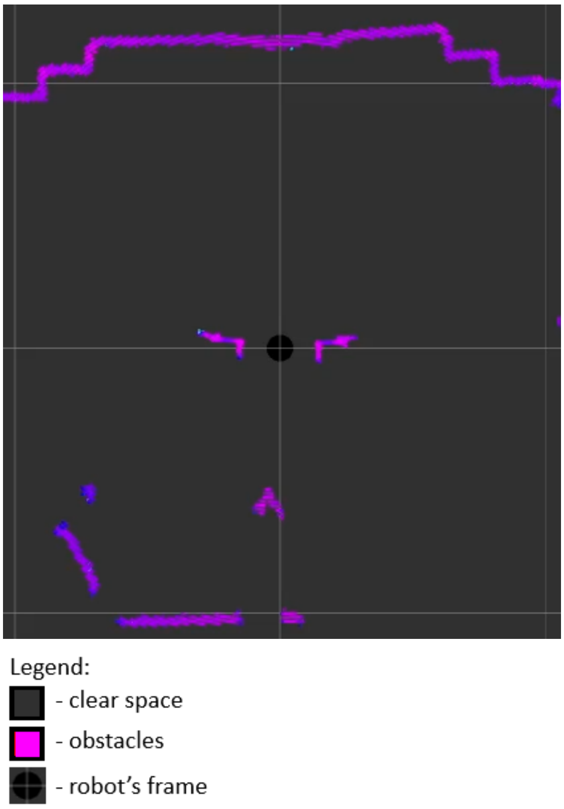


Figure 58: Visual linear odometry test at t=15 s.

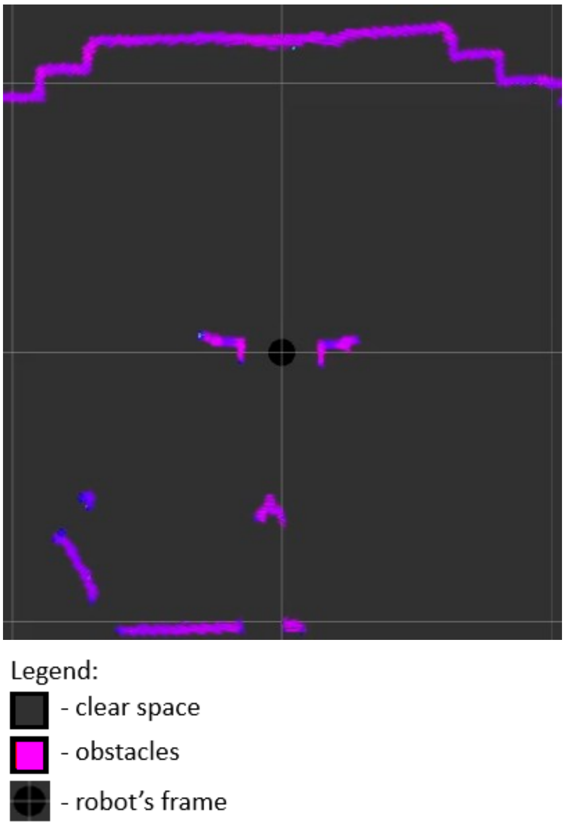


Figure 59: Visual linear odometry test at t=16 s.

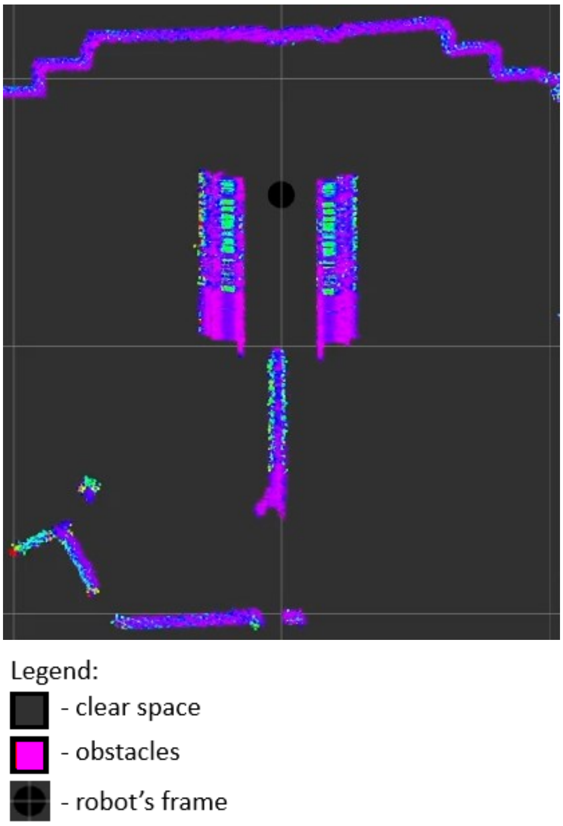


Figure 60: Visual linear odometry test at t=40 s.

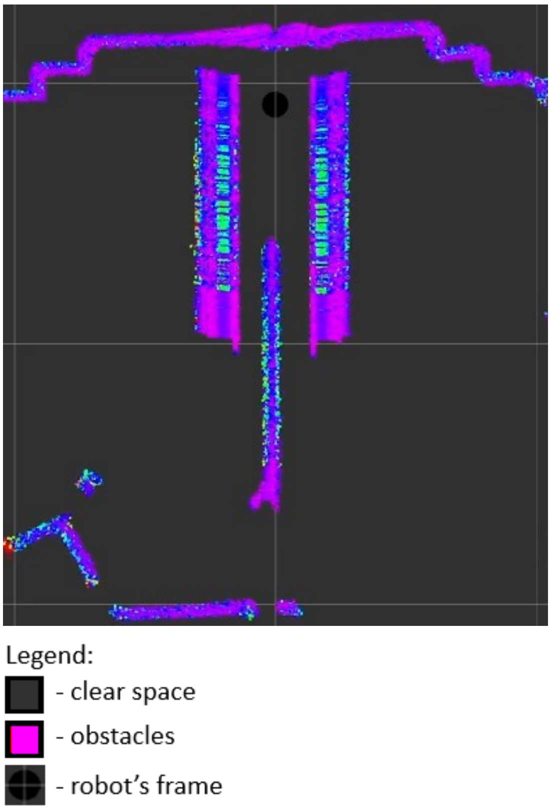


Figure 61: Visual linear odometry test at t=60 s.

The following conclusions were taken by analysing the figures:

- By analysing the difference in thickness of obstacles in Figure 57 and Figure 58 it is possible to observe that the Lidar module's measurement deviation was around 40 mm.
- Observing the difference in thickness of obstacles in Figure 58 and Figure 59, should represent the error resulting mostly from desynchronization of odometry information and the Lidar's measurements. In this case, the error was considered negligible.
- Towards the end of the test, as the robot became very close to the wall, the points of the wall detected by the Lidar module started deviating more, as it can be observed by comparing the top part of Figure 61.
- The odometry error was measured by comparing the thickness of the wall in Figure 59 and Figure 60. This represents the difference between 1 second after the robot started moving, and 25 s after when the robot had travelled 625 mm. The error measured was of around 5 mm, which represents an error of 0.8 %.

5 Navigation System

For extra computational power and to not overburden the IPC, a Raspberry Pi 4 Model B (Figure 30) with 2GB of RAM was installed on the robot. In this robot, the Raspberry Pi has the function of utilizing the Lidar module and camera, receiving odometry information from the IPC's TwinCAT, running the navigation and localization algorithms through Robot Operating System (ROS), and sending velocity commands to the IPC's TwinCAT 3 software. This communication is done via an Ethernet cable connecting the Raspberry Pi and the IPC.

A Raspberry Pi 4 is a low cost, credit-card sized computer. It is possible to connect additional devices to it through USB, Ethernet, and micro-HDMI ports. It can plug into a computer monitor or TV and be used with keyboard and mouse. It also has GPIO (General Purpose Input/Output) pins (Figure 62). Sensors can be connected to the raspberry pi using the GPIO pins or USB ports. Information regarding the Raspberry Pi was obtained from https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/.

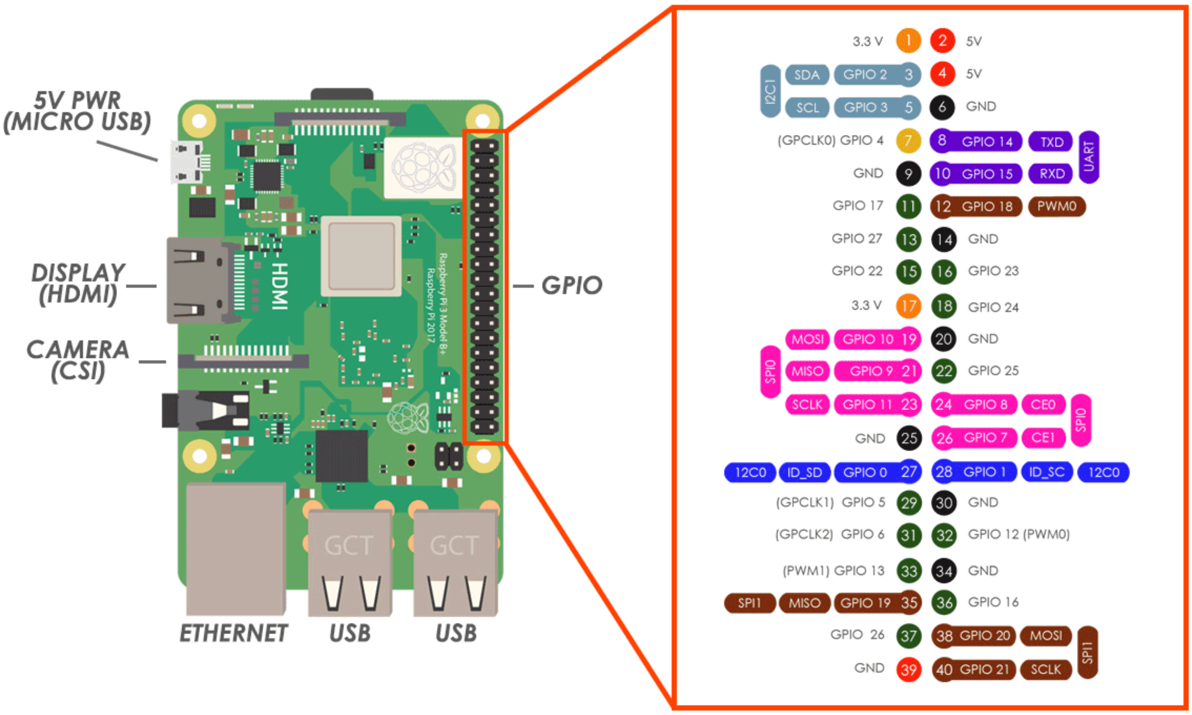

5V PWR (MICRO USB)
DISPLAY (HDMI)
CAMERA (CSI)
HDMI
Raspberry Pi 3 Model B+
Raspberry Pi 2017
GPIO
ETHERNET
USB
USB
GCT
GCT
3.3 V 1 2 5V
I2C1 SDA GPIO 2 3 4 5V
SCL GPIO 3 5 6 GND
(GPCLK0) GPIO 4 7 8 GPIO 14 TXD UART
GND 9 10 GPIO 15 RXD
GPIO 17 11 12 GPIO 18 PWM0
GPIO 27 13 14 GND
GPIO 22 15 16 GPIO 23
3.3 V 17 18 GPIO 24
SPI0 MOSI GPIO 10 19 20 GND
MISO GPIO 9 21 22 GPIO 25
SCLK GPIO 11 23 24 GPIO 8 CE0 SPI0
GND 25 26 GPIO 7 CE1
I2C0 ID_SD GPIO 0 27 28 GPIO 1 ID_SC I2C0
(GPCLK1) GPIO 5 29 30 GND
(GPCLK2) GPIO 6 31 32 GPIO 12 (PWM0)
(PWM1) GPIO 13 33 34 GND
SPI1 MISO GPIO 19 35 36 GPIO 16
GPIO 26 37 38 GPIO 20 MOSI SPI1
GND 39 40 GPIO 21 SCLK

Figure 62: Raspberry Pi 4's ports and GPIO pins [40].

Robot Operating System (ROS) is an open source set of software libraries and tools to facilitate building robot applications. It includes drivers, algorithms, and developer tools. Unlike the name suggests, ROS is not an operating system. ROS started in 2007 and since then has received many contributions from multiple different sources, including its creators, community developers and users. Information regarding ROS was obtained from its website at https://www.ros.org/ and its wiki page at http://wiki.ros.org/.

ROS is a distributed framework of processes (each process is called a Node in ROS), that enables executables to be individually designed and then coupled at runtime, communicating via standardized ROS messages. These messages are published in ROS topics with the appropriate name, and each Node can access these topics either to read or publish messages. Figure 63 shows an example of this communication between nodes used in this dissertation. These processes can be grouped into Packages and Stacks, which can be shared and distributed. This design promotes sharing and collaboration, as well as having reusable code across multiple robot projects.

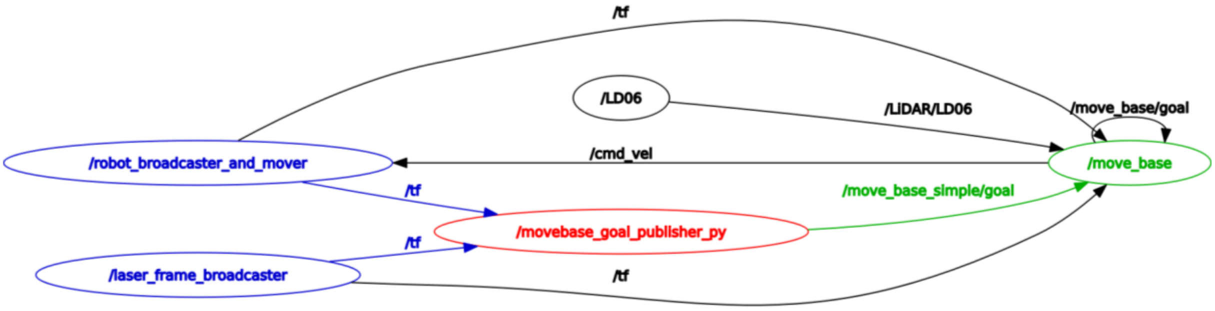


Figure 63: ROS nodes and their communications using ROS topics. The node in red reads messages in the */tf* topic published by the nodes in blue, and it publishes messages in the */move_base_simple/goal* topic that are read by the green node.

Currently, there are two major versions: ROS (also known as ROS 1), and ROS 2. ROS 2 is intended to be a more modern version of ROS, with additional functionalities. ROS 2 is available for a wider variety of operating systems, including Ubuntu, Windows, macOS and RHEL. ROS 1 supports Ubuntu and Debian.

Since the drivers of the Lidar used in this dissertation were developed for use in ROS 1, the Raspberry Pi was set up to work with ROS 1. At the time of writing this dissertation, 13 different distributions of ROS 1 have been released since ROS was created. Currently, the recommended and most recent long-term support (LTS) ROS 1 distribution is *Noetic Ninjemys*. It primarily targets Ubuntu Focal 20.04 and supports Debian Buster 10. It is possible to install it on Raspberry Pi OS, which is based on Debian; however, it can cause some complications. Because of these factors, in this dissertation, Ubuntu 20.04 and ROS Noetic were installed on the Raspberry Pi.

5.1 Headless Ubuntu configuration for Raspberry Pi

In the case of a mobile robot, requiring a computer monitor, keyboard and mouse connected to the robot’s Raspberry Pi to access it can be inconvenient. As such, using it remotely, can be easier. A computer that can be used without a monitor or keyboard connected to it is known as a headless computer.

Ubuntu Server 20.04 64-bit was installed on the Raspberry Pi according to the guide available in Ubuntu's website in https://ubuntu.com/tutorials/how-to-install-ubuntu-on-your-raspberry-pi. Additionally, Xubuntu, a light-weight desktop environment for Ubuntu, was installed to allow visual representation of the robot's navigation system in ROS.

Initially, the Raspberry Pi was accessible using another computer via SSH (Secure Shell); however, to have a graphical representation of the desktop environment, XRDP was also installed, which made it possible to access the Raspberry Pi via Remote Desktop Protocol (RDP) using Microsoft's *Remote Desktop Connection* program.

To facilitate connections and to make sure that they will keep working in the future, the Raspberry Pi was also given a static IP address and a known hostname.

ROS Noetic was installed following the steps in the guide from the ROS Wiki.

The Lidar's drivers were installed following the manufacturer's guide in https://www.okdo.com/getting-started/get-started-with-lidar-hat-for-raspberry-pi/#h-4-lidar-driver-toc.

To be able to use the Lidar module on the Raspberry Pi it is necessary to enable the Raspberry Pi's serial port and disable its serial console. When using Raspberry Pi OS this only requires selecting certain checkboxes in a menu. On the other hand, when using Ubuntu, it is not as user friendly. A solution to this problem was found by following the steps provided in Section 5 of the guide in https://github.com/nasa-jpl/osr-rover-code/blob/foxy-devel/setup/rpi.md#5-setting-up-serial-communication-on-the-rpi. This solution makes the UART (Universal Asynchronous Receiver/Transmitter) serial port */dev/ttyAMA0* usable, but not the mini-UART */dev/ttyS0*, which is the default port that the OKdo LD06 Lidar uses. Because of this, it was also necessary to edit the launch file of the Lidar's drivers and replace */dev/ttyS0* with */dev/ttyAMA0*.

Additionally, when booting up the Raspberry Pi with the Lidar module connected, the Lidar's serial messages interrupted U-Boot from loading Ubuntu. A solution to this issue was found by following the steps provided in the user comment in https://askubuntu.com/questions/1215848/how-to-disable-ttyama0-console-on-boot-raspberry-pi/1287847#1287847, which resulted in the removal of the initial boot delay where it can be interrupted, as well as silencing the booting process, preventing U-Boot from sending messages to the Lidar module.

5.2 ROS Navigation Stack

The ROS Navigation Stack is designed to be a general purpose navigation configuration. It takes in information from odometry and sensor streams and outputs velocity commands to send to the mobile robot. It can only be used for differential drive and holonomic (omnidirectional) wheeled robots. It also requires a sensor that can sense obstacles in the world. Figure 64 shows a diagram with an overview of the navigation stack. The white components are required components that are already implemented in the *move_base* package, the grey components are optional components to use map-based navigation that are already implemented, and the blue components must be created for each specific mobile robot. The arrows represent ROS topics and messages published and read by each node.

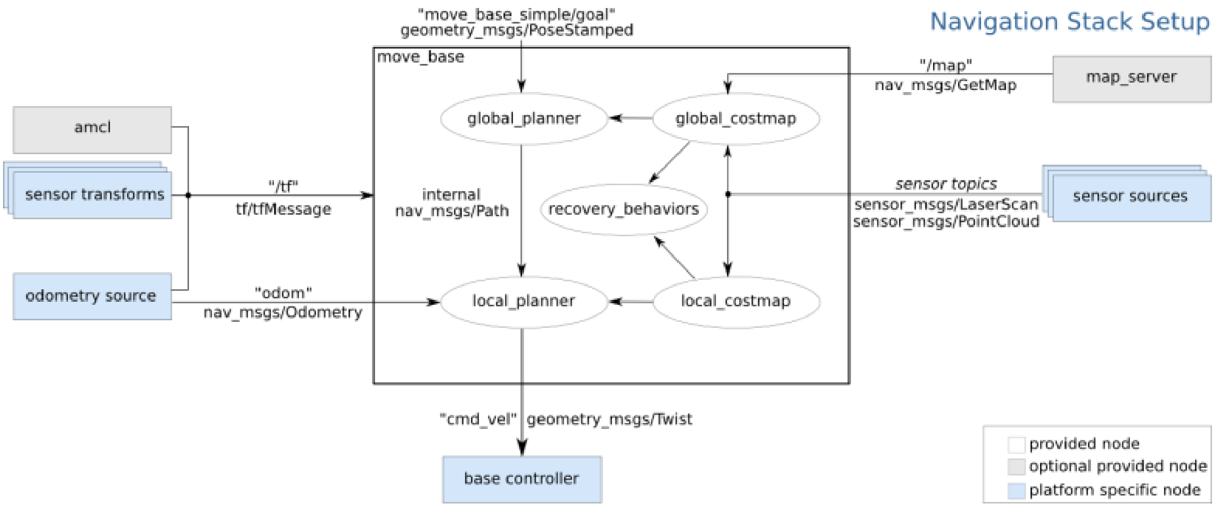


Figure 64: Overview of the Navigation Stack [41].

5.2.1 Navigation Stack pre-requisites

Apart from ROS, pre-requisites to utilize the navigation stack are:

- Transform configuration;
- Sensor information;
- Odometry information;
- Base controller;
- Mapping (for map-based navigation only).

After fulfilling all these pre-requisites, the Navigation Stack can be setup and configured.

- **Transform configuration:**

The Navigation Stack requires the robot to publish information about the relations between coordinate frames. This can be done using either *tf* or *tf2*, which are ROS packages that let users keep track of relations between multiple coordinate frames in a tree structure over time. They also allow users to transform points and vectors between any two coordinate frames at any desired point in time if a relation between them exists. *tf2* is the more recent package, is more efficient and contains additional features. As such, *tf2* was used in this dissertation.

To create relations between coordinate frames, it is necessary to name the existing frames. The relevant frames for the robot used in this dissertation are the Lidar’s frame, the robot’s frame, and the frame where the odometry started; these will be called in ROS by the names *lidar_frame*, *base_link* and *odom* respectively. If a map is used, it also has its own frame, which is commonly called */map* in ROS.

For the specific robot in this dissertation, it is necessary to configure two different transforms. One between the Lidar (*lidar_frame*) and the robot’s frame (*base_link*), and one between the robot’s frame (*base_link*) and the frame where the odometry started (*odom*). The first transform is static since the relation between the Lidar’s position and the robot’s frame position is always the same. Figure 65 shows both frames and the fixed distances between them.

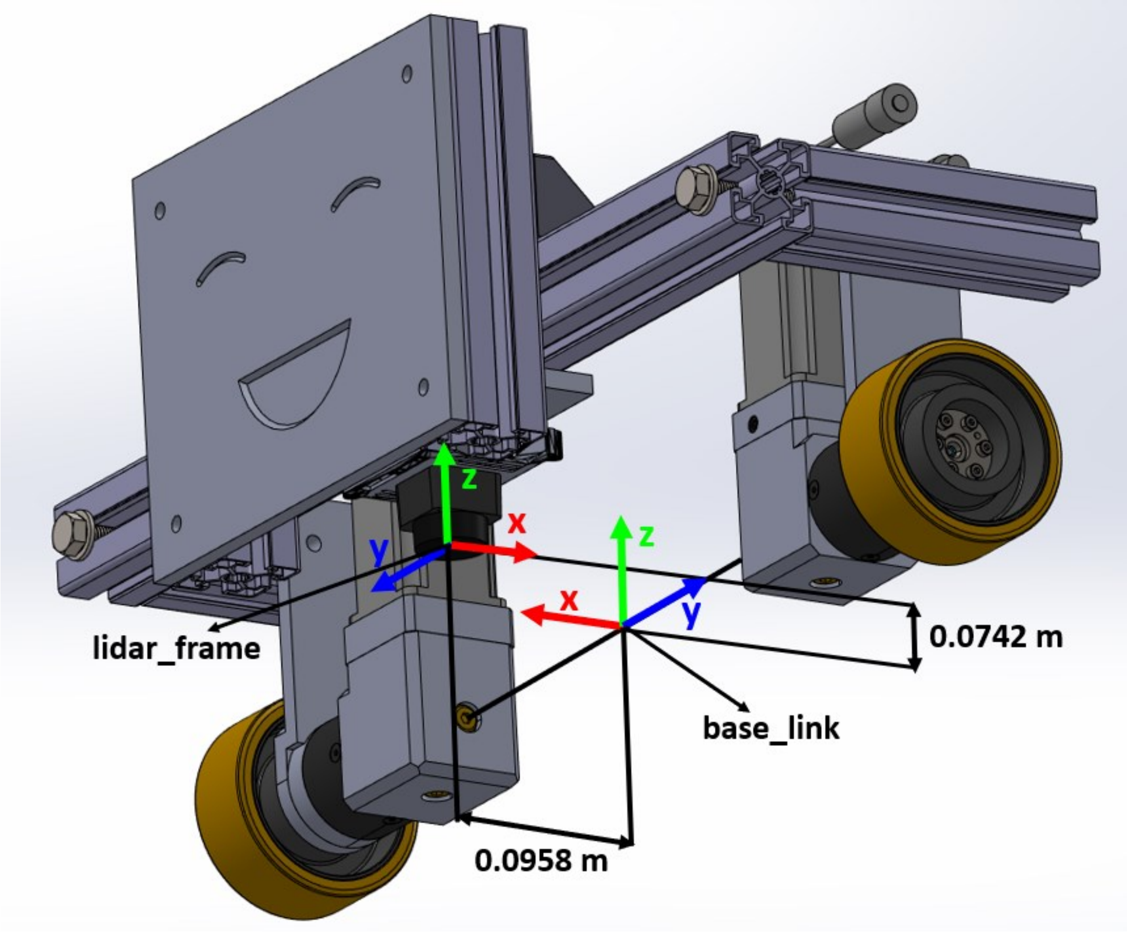


Figure 65: *base_link* and *lidar_frame* frames and their relations.

The Lidar's frame is oriented as shown in Figure 65, because of these factors:

- The Lidar module is upside down and with the *x* axis oriented to the back of the robot, which is the direction where it points when it is reading information at an angle of 0. This is equal to a rotation of 180° around the *y* axis of the robot's frame;
- The Lidar module sweeps its surroundings while rotating in a clockwise direction. Since in the robot's kinematics a positive angle is in the counterclockwise direction, it is necessary to rotate its frame by 180° around the *x* axis to counteract this;
- A rotation of 180° around the *y* and 180° around the *x* axes has the same effect as a rotation of 180° around the *z* axis, which is what is shown in Figure 65.

In ROS' *tf2*, transforms are broadcast with information about which frame is being transformed into which child frame, a timestamp, a translation vector, and a rotation quaternion. In this case, the original frame is *base_link*, the child frame is *lidar_frame*, the translation vector is (0.0958, 0, 0.0742) and the rotation quaternion is (0, 0, 1, 0).

The ROS node that broadcasts this transform is included in Appendix B.1 and is written in Python. It publishes the transform once every second.

The transform between the robot's frame (*base_link*) and the odometry frame (*odom*) is dynamic and is given by the odometry information.

An additional transform can exist if the robot uses map-based navigation. This transform gives the relation between the odometry frame and the map frame. It is not configured by the user, instead it is obtained using a localization algorithm that compares the sensor streams of the robot and the map used.

- **Sensor information:**

The navigation stack uses information from sensors to avoid obstacles in the world. Since the Lidar module used in this dissertation already had drivers for ROS, they already publish the sensor information using ROS messages. This Lidar publishes *sensor_msgs/LaserScan* messages which include a timestamp, a coordinate frame, a scanning angle, angle, and time increment between each point, scanning range, a vector with the distance of each point per scan, and a vector with the intensity of each point. It is only necessary to guarantee that the coordinate frame these messages are published on has the same name as the *lidar_frame*.

- **Odometry information and base controller**

The navigation stack also requires odometry information to be published. This needs to be published as *nav_msgs/Odometry* messages which include the robot's position and velocities in relation to the odometry frame, and as a *tf* or *tf2* transform between the odometry frame and the robot's frame.

The base controller's job is to receive velocity commands computed by the navigation stack and published in the ROS topic *cmd_vel* and convert them into the respective motor commands for the robot.

The robot's TwinCAT 3 system already possesses all the odometry information needed by the navigation stack, and the ability to transform the linear and angular velocity commands in real movement of the robot. To integrate this TwinCAT system with ROS it is necessary to create ROS nodes for it. One that can read the odometry information from the IPC's TwinCAT and publish it in ROS messages, and one to read the messages in the ROS topic *cmd_vel* and send the velocity commands to TwinCAT.

To facilitate communication with TwinCAT, the *pyads* Python library was used. This is a python wrapper for TwinCAT's ADS library that provides python functions for communicating with TwinCAT devices. Information regarding *pyads* was obtained from https://pypi.org/project/pyads/.

It is recommended to have a ROS node that can read the odometry data at a determined frequency; and one ROS node that is always running and subscribed to the *cmd_vel* topic and sends the velocity commands to TwinCAT whenever a new message is published on that topic (Figure 66). However, during the development of this dissertation, there were issues communicating with TwinCAT's ADS using different programs simultaneously in Ubuntu. Because of these issues, a single ROS node was written with the functions of both nodes previously mentioned and running at a desired frequency of 20 Hz (Figure 67). This node was written in Python and is available in Appendix B.2. It also writes the safety variable in TwinCAT mentioned in Section 4.2.

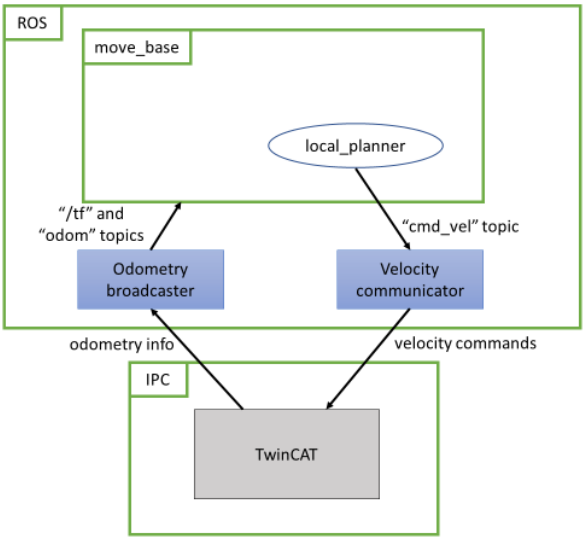


Figure 66: Recommended organization of nodes and communication between them.

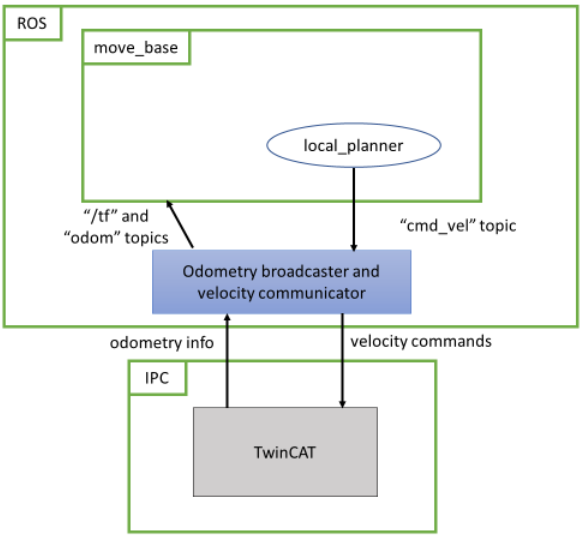


Figure 67: Implemented organization of nodes and communication between them.

- **Mapping the environment:**

To create a map using the ROS package *slam_gmapping*, it is necessary to make sure that the robot is publishing its laser scans and transform data between the laser, the robot and the odometry frames to their respective ROS topics. An option to drive the robot around its environment is also necessary. For this purpose, a program was written to teleoperate the robot using a keyboard's arrow keys or "WASD" keys. This program was written in Python and is available in Appendix B.3. It also utilizes *pyads* to communicate with TwinCAT.

The first step of creating a map using the *slam_gmapping* package is to record a "bag" file, which is a file format in ROS for storing message data. This bag file should contain the laser scan and transform messages published while driving the robot around the environment to be mapped. To record this bag file, the ROS package *rosbag* can be used.

General advice while making the bag file is to:

- Limit the robot's velocity to lower values, especially its angular velocity;
- Visualize the message data using a program such as *RViz* while recording. This allows the user to possibly spot zones that the robot missed and that should be mapped;
- Drive the robot back to its initial position to create a closed loop. It is also recommended to drive for a few more meters after closing the loop.

After making the bag file it is necessary to open *slam_gmapping* configured to take in information from the ROS topic where the Lidar module publishes its messages, and the name of the odometry's frame and play back the bag file to feed data to *slam_gmapping*. This will result in a map to be broadcast in the ROS environment. When *rosbag* has finished playing the bag file, the tool *map_saver* from the *map_server* ROS package can be used to save the map as a ".pgm" file and save a ".yaml" configuration file that includes its resolution, origin point, and more.

Figure 68 shows the map created for the robot's environment in this dissertation. It is necessary to note that the points highlighted by the red shape do not represent real obstacles and appeared there probably due to the robot's large dead zones. As such, the map was edited using an image editing software, resulting in the map shown in Figure 69.

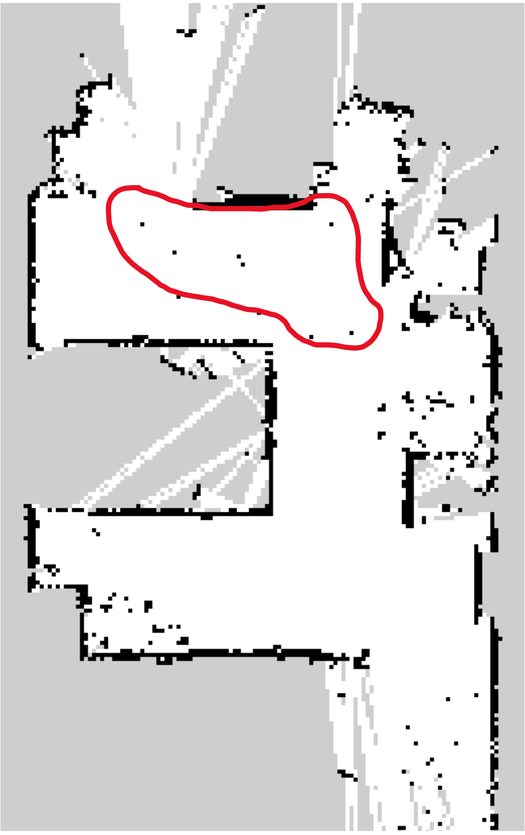


Figure 68: Map of the robot's environment created by *slam_gmapping*.

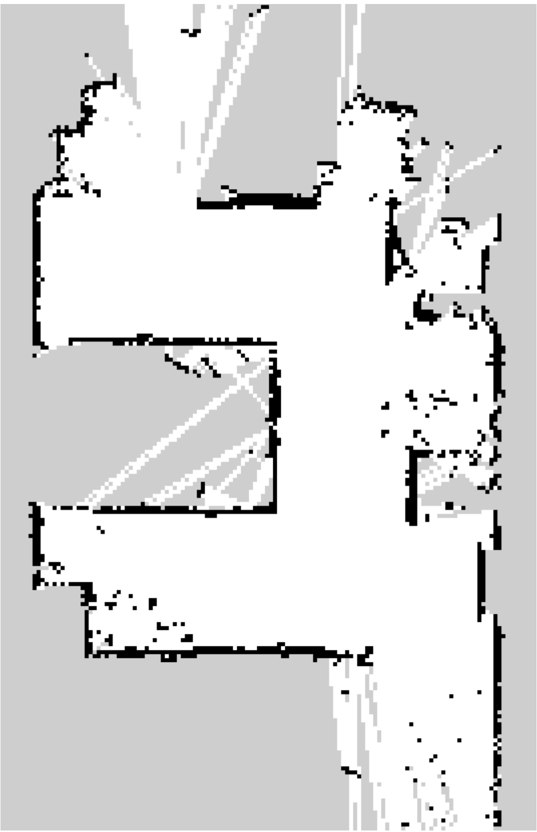


Figure 69: Map of the robot's environment edited to not include points in free space.

5.2.2 Setting up the Navigation Stack

After the above-mentioned parameters were satisfied, a ROS package was created to store all the configuration and launch files for both the navigation stack and its pre-requisites. A launch file that runs all the configured pre-requisites for the navigation stack (publishing transforms, sensor streams, and odometry info, as well as sending velocity commands to TwinCAT) was created. This launch file is available in Appendix B.4.

Configuration files for various navigation stack components are also necessary to configure. These components are: local and global costmaps; local and global planners; and *AMCL*, a ROS package for localization (only necessary for map-based navigation).

- **Costmaps:**

Costmaps are maps that represent the cost of navigating through different areas. These costs can depend on different parameters such as presence of obstacles, closeness to obstacles, having or not having knowledge about an area, flatness of the terrain, and others.

By default, the ROS navigation stack utilizes the *costmap_2d* package. This is an implementation of a 2D costmap that takes in sensor data from the world and can take information from a map to create an initial costmap. It builds an occupancy grid of the data, and inflates costs, resulting in a 2D costmap based on the occupancy grid and user specified inflation parameters. It can represent occupied space, free space, and unknown space in the environment that the robot is in. These three different statuses are given different cost values. Obstacles can also be inflated. Inflation is the process of propagating cost values out from occupied cells that decrease with distance. Figure 70 shows a costmap created by the *costmap_2d* package. Obstacles are represented by the colour red, obstacle inflation in blue, free space in light grey, and unknown space in dark grey. The robot's footprint is represented by the red polygon.

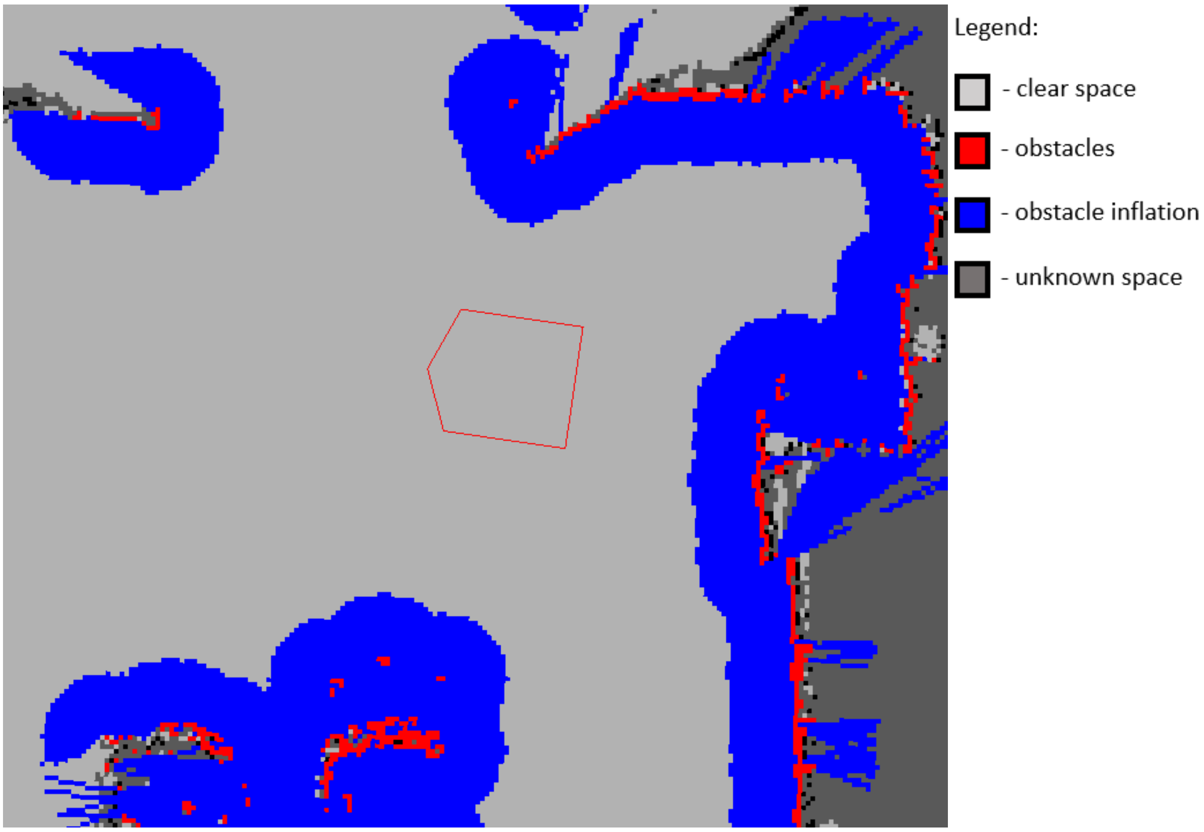


Figure 70: Costmap created by the *costmap_2d* package [42].

The navigation stack requires two different costmaps, one local and one global. The global costmap contains everything the robot knows from a pre-built map, and it is updated over time with sensor information. It is used for global planning, which is creating long-term plans for navigating over the environment. The local costmap contains only information given by the sensors and is used for obstacle avoidance and local planning, which is trajectory planning including velocity commands to try and follow the global path plan closely.

The robot's footprint is also configured in the costmap package. In this case, it was given the list of points that corresponds to the robot's vertices in meters, that, when connected, result in an approximation of the robot's shape when viewed from above. The resulting footprint by connecting each point in the list in the order they are written is shown in Figure 71. The larger right side is due to the cables located outside of the robot in that area. For safety and to guarantee that the points the Lidar measures that are situated in its dead zones are located inside the footprint, the footprint was slightly enlarged.

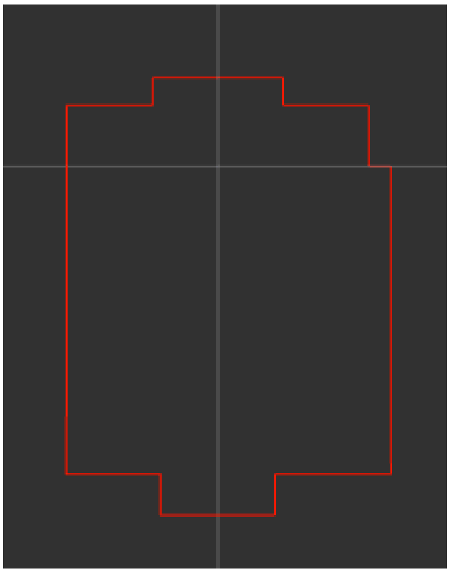


Figure 71: Robot's footprint.

This configuration can be stored in 3 different files. One for parameters that are common to both the local and global costmap, one for global costmap specific parameters, and one for local costmap specific parameters. Optimization of these parameters for the robot used in this dissertation and its environment is analysed in Section 5.2.3. Appendix B.5 contains the configuration files used.

- **Planners:**

The global and local planners are responsible for generating path and trajectory plans required to reach a goal. The global planner creates long-term plans for navigating over the environment. The local planner does trajectory planning including computing velocity commands to try and follow the global path closely while avoiding obstacles.

The global planner chosen for this dissertation was the ROS package *global_planner*, which computes the minimum cost plan from the start point to the end point using Dijkstra's algorithm by default. It also includes customization options to utilize the A* algorithm instead.

The local planner chosen was the *dwa_local_planner*. This planner is a ROS implementation of the Dynamic Window Approach to local robot navigation on a plane. When given a global plan to follow and a costmap of the environment, the local planner will produce the necessary velocity commands.

Both these planners include customizable parameters that can be adjusted to achieve a better performance for a specific robot. Configuration files set up for the robot used in this dissertation were written for both planners and are included in Appendix B.6.

In the case of the local planner, it is necessary to include information regarding the robot's minimum and maximum velocities and accelerations for both the linear and angular components. In the case of the robot used in this dissertation, it was setup with the following limits for each wheel in TwinCAT present in Table 8. These values are smaller than the real limits imposed by the robot's locomotion system to guarantee that they can always be achieved.

Table 8: Maximum velocity, and default and maximum accelerations for each wheel.

Maximum velocity	**Default acceleration**	**Maximum acceleration**
0.5 m/s	0.5 m/s^2	1.0 m/s^2

For the local planner's parameters, the default acceleration was used as the planner only computes velocity commands and not acceleration commands. It was decided to allocate 60 % of the motor's power exclusively to linear velocity and acceleration, and 40 % exclusively to angular velocity and acceleration. This results in Equations 5.1, 5.2, 5.3 and 5.4 for the maximum linear velocity, angular velocity, linear acceleration, and angular acceleration respectively.

$$v_{x,\ MAX} = 0.6 \cdot v_{MAX} \tag{5.1}$$

$$v_{\theta,\ MAX} = \frac{0.4 \cdot (2 \cdot v_{MAX})}{B} \tag{5.2}$$

$$a_{x,\ MAX} = 0.6 \cdot a_{MAX} \tag{5.3}$$

$$a_{\theta,\ MAX} = \frac{0.4 \cdot (2 \cdot a_{MAX})}{B} \tag{5.4}$$

Where: B, is the distance between the driven wheels.

The resulting values rounded down to the decimal place and implemented in the local planner are present in Table 9.

Table 9: Maximum linear and angular velocities, and accelerations for the local planner.

$\boldsymbol{v_{x,\ MAX}}$	$\boldsymbol{v_{\theta,\ MAX}}$	$\boldsymbol{a_{x,\ MAX}}$	$\boldsymbol{a_{\theta,\ MAX}}$
0.3 m/s	0.7 rad/s	0.3 m/s^2	0.7 rad/s^2

In terms of minimum velocities, the values of 0.01 m/s and 0.01 rad/s were chosen arbitrarily for the linear and angular velocity respectively.

- **Localization system:**

AMCL is a probabilistic localization system for a robot moving in a 2D plane implemented in ROS. It is based on the adaptive Monte Carlo localization approach which uses a particle filter to track the position of a robot against a known map described in [43] and [44].

It can localize the robot's position in a map by comparing the robot's laser sensor streams and its odometry information with a previously built map of the environment. *AMCL* estimates the transform that gives the relation between the robot's frame in relation to the map's frame and publishes the transform between the map's frame and the odometry frame to the ROS environment, which can then be used for navigation. This transform accounts for drift and other errors accumulated in the robot's odometry by recalculating itself over the course of the robot's trajectory.

The localization algorithm requires an initial probability distribution of the robot's pose, represented by particles distributed around a set initial pose according to a gaussian distribution with a certain covariance. Both the initial pose and covariance values can be configured, and new "initial poses" can be fed to the algorithm after its initial localization. These particles are propagated following the robot's odometry model as the robot moves around in its environment. When receiving new sensor readings, each particle evaluates its accuracy as the robot's location by checking how likely it would receive those sensor readings at its current position. The algorithm then redistributes new particles according to a probability distribution around the particles that were considered more likely. By iterating these steps, the localization given by the algorithm should become more accurate as the robot moves around its mapped environment more [44].

The initial pose used by *AMCL* is by default the point (0, 0, 0) of the map frame. This point can be changed in the *AMCL* configuration file. An initial pose can also be given by selecting it in *Rviz* using the *2D Pose Estimate* button.

Figure 72 shows a visualization in *RViz* of the robot's navigation system using *AMCL*. The obstacles in magenta and their respective inflation are given by the local costmap. It is possible to see that the obstacles in the costmap are lined up with the obstacles in the pre-recorded map from Figure 69 due to *AMCL*'s localization algorithm estimating the robot's position in the map.

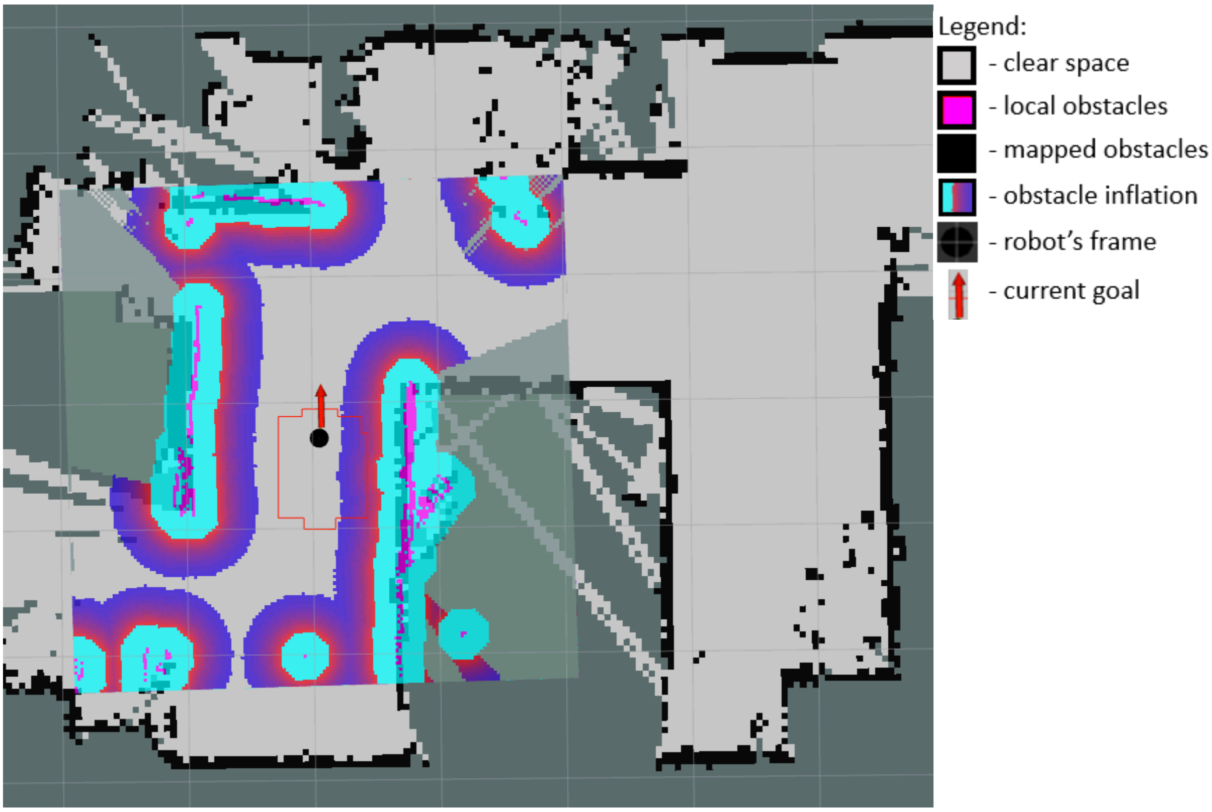


Figure 72: Visualization of the robot's location and local costmap in the recorded map.

The *AMCL* launch file configured for the robot used in this dissertation is included in Appendix B.7.

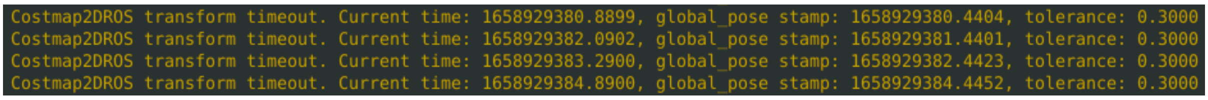
5.2.3 Tuning the Navigation Stack parameters

The parameters of each component of the Navigation Stack can be configured to optimize the robot's behaviour. Different values of these parameters can prove to have a better performance for different robots and environments, depending on the robot's computational power, dynamics, shape, and size, and depending on the size and geometry of the robot's environment.

- **Localization system:**

It is necessary to guarantee that *AMCL* can provide the transforms between the robot's odometry frame and the map's frame at a rate that can be used by the other nodes of the Navigation Stack that need it, specifically the global costmap and the global planner.

The *transform_tolerance* parameter gives the time with which to post-date the transform that is published, indicating how long into the future it should be considered valid. Testing with the default *transform_tolerance* parameter of 0.1 seconds showed that the other components were consistently having warnings due to not having a recent enough transform by margins of up to 1 second (Figure 73). This occurs because the computational power of the Raspberry Pi is not enough to run the *AMCL* process at higher frequencies while running the others at their required frequencies. The *transform_tolerance* parameter was changed to 1 second and the existing warnings stopped occurring. A tolerance of 1 second should not result in significant errors due to the high accuracy of the odometry.



```
Costmap2DROS transform timeout. Current time: 1658929380.8899, global_pose stamp: 1658929380.4404, tolerance: 0.3000
Costmap2DROS transform timeout. Current time: 1658929382.0902, global_pose stamp: 1658929381.4401, tolerance: 0.3000
Costmap2DROS transform timeout. Current time: 1658929383.2900, global_pose stamp: 1658929382.4423, tolerance: 0.3000
Costmap2DROS transform timeout. Current time: 1658929384.8900, global_pose stamp: 1658929384.4452, tolerance: 0.3000
```

Figure 73: Warnings caused by the low transform tolerance.

The odometry model of *AMCL* was changed from the default *diff* to *diff-corrected*. While they are both models for differential drive robots, the *diff-corrected* model is a newer version with a certain bug fix. This model also requires lower values than the defaults of the *odom_alpha1*, *odom_alpha2*, *odom_alpha3*, and *odom_alpha4* parameters that specify the expected noise in the odometry values. Since the robot's odometry proved to have a very high accuracy, they were tuned to values of 0.005, 0.005, 0.010 and 0.005 respectively with positive results.

The laser model used was the default *likelihood_field* model because it requires less computational resources than the others. It uses two parameters that represent a weighted average of the type of measurements the laser sensor gives: *z_hit* and *z_rand*. The *z_hit* parameter represents the weighted average of the measurements that represent real obstacles and include a certain measurement noise characterized by the standard deviation *laser_sigma_hit*. The *z_rand* parameter represents the weighted average of the random measurements laser sensors occasionally produce. Because these parameters represent a weighted average, their sum should be equal to 1 [44]. Their default values and the values used in this dissertation are shown in Table 10. Because the value of the measurement error provided by the Lidar module's manufacturer is of ± 45 mm, the *laser_sigma_hit* parameter was reduced to 0.045 m.

Table 10: Values of the likelihood_field model parameters.

	z_hit	*z_rand*	*laser_sigma_hit*
Default value	0.95	0.05	0.2 m
Value used	0.95	0.05	0.045 m

The *AMCL* launch file configured for the robot used in this dissertation is included in Appendix B.7.

A program for the robot's camera pointing at the floor to search for QR codes that feed an "initial position" used by *AMCL* for its localization was also developed. It is discussed in Section 5.3.3.

- **Global planner:**

The global planner contains three parameters that are very important when computing the lowest cost global path. They are *lethal_cost*, *neutral_cost*, and *cost_factor*. *cost_factor* is the factor to multiply each cost from the costmap by. *neutral_cost* is the cost value of neutral terrain, meaning free space not given any cost by the costmap. *lethal_cost* should remain at the value of 253 as it represents the cost value of obstacles, and the robot should never collide with them.

The default and used values of these parameters are shown in Table 11.

Table 11: Default values of the global planner's cost parameters.

	lethal_cost	*neutral_cost*	*cost_factor*
Default value	253	50	3
Used value	253	55	0.8

Different values of the *cost_factor* parameter were experimented. Increasing the value of the *cost_factor* showed a larger tendency to avoid inflation zones, while decreasing the value showed shorter paths that travelled closer to obstacles. When using a *cost_factor* of 0.8 the planned paths had a satisfactory smoothness while avoiding obstacles successfully. Planned paths using different *cost_factor* values can be seen in Figure 74.

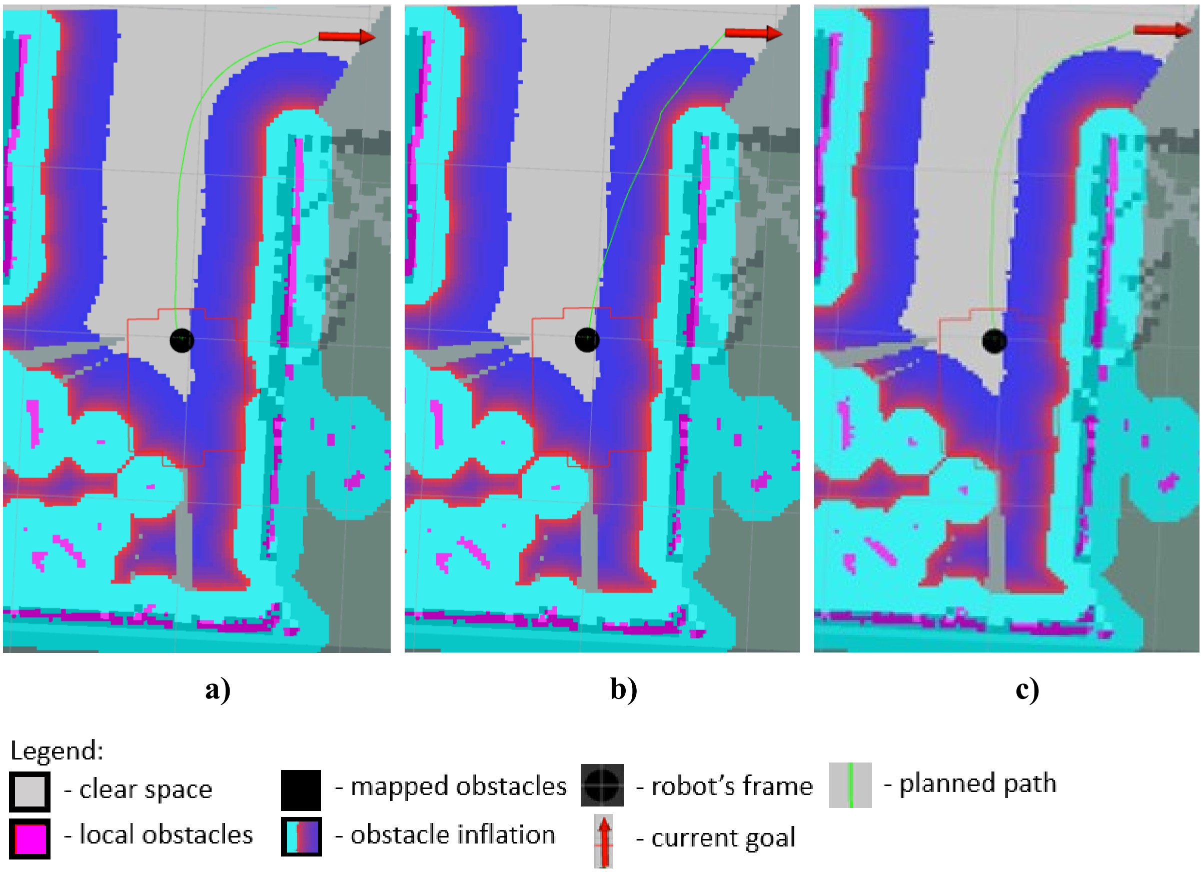


Figure 74: Paths with respective cost_factor and neutral_cost of: a) 3.0 and 50; b) 0.01 and 50; c) 0.8 and 50.

Different values of the *neutral_cost* parameter were also experimented. Decreasing its value resulted in straighter paths that heavily prioritized travelling in neutral areas, while increasing its value resulted in smoother paths that would travel very close obstacles. When using a *neutral_cost* of 55, the calculated paths showed a satisfactory smoothness while avoiding obstacles successfully. Planned paths using different *neutral_cost* values can be seen in Figure 75.

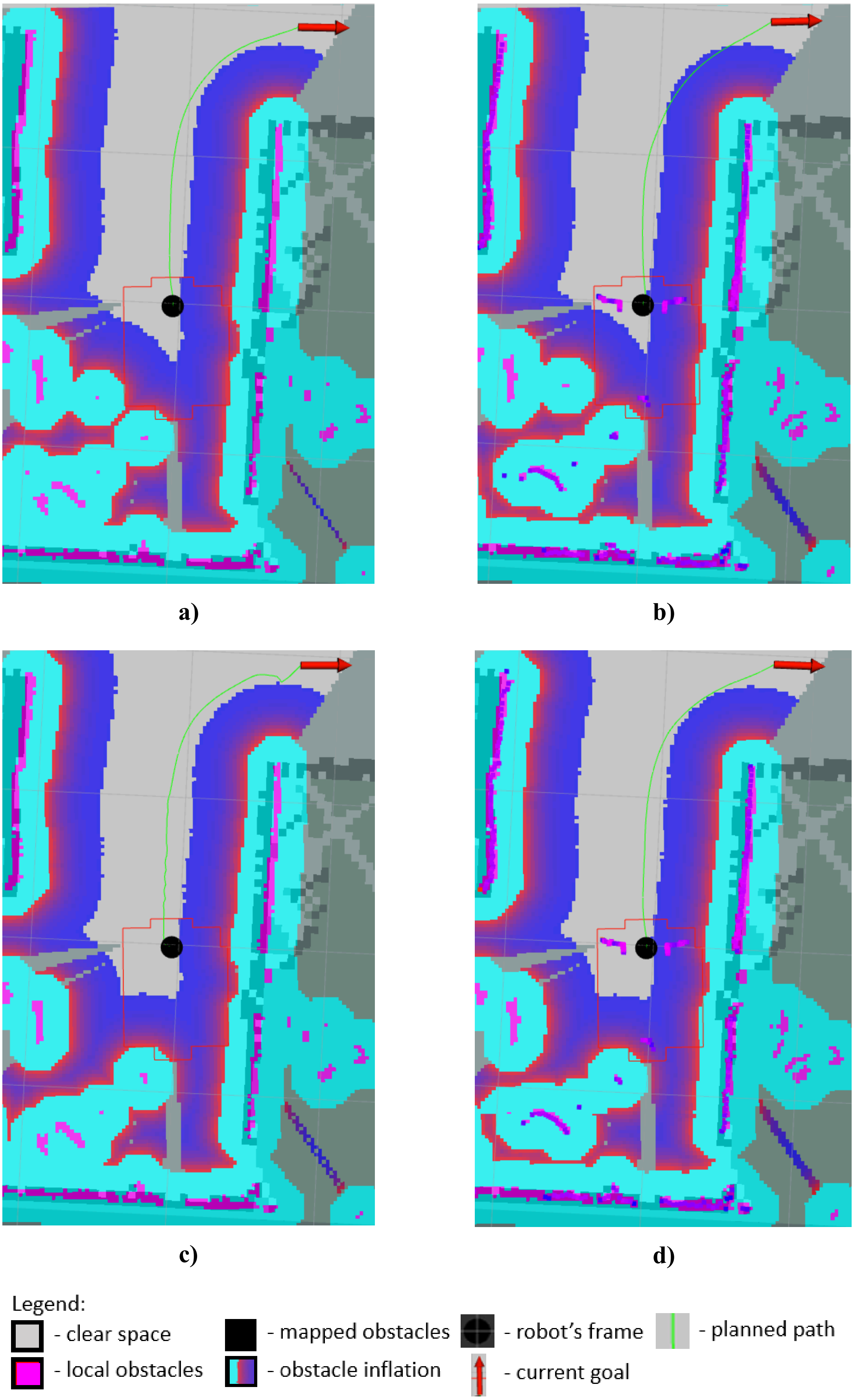
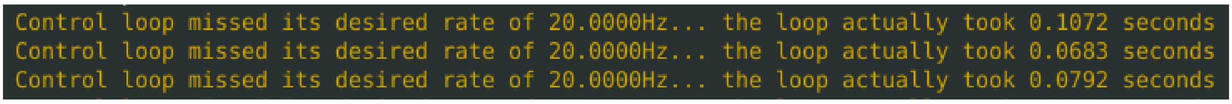


Figure 75: Paths with respective neutral_cost and cost_factor of: a) 50 and 0.8; b) 120 and 0.8; c) 1 and 0.8; d) 55 and 0.8.

The global planner configuration file used in this dissertation is included in Appendix B.6.1.

- **Local planner:**

With the default settings of the local planner, the robot was producing very unreliable trajectories with a lot of oscillating movements. Experimenting with the forward simulation parameters of *dwa_local_planner* resulted in smoother trajectories that reached the goal faster. The parameter that defines the amount of time to forward-simulate trajectories in seconds, *sim_time*, was changed from the default of 1 second to 2 seconds. The parameter that defines the number of samples to use when exploring the x velocity space, *vx_samples*, was changed from the default of 3 to 10. The parameter that defines the number of samples to use when exploring the theta velocity space, *vtheta_samples*, was changed from the default of 20 to 25. However, with these changes, the robot started to give more warning messages related to the control loop missing its desired rate of 20 Hz (Figure 76) as each loop required a higher computational load. As such, the controller's frequency was changed to 10 Hz. At 10 Hz the robot showed to still have a fast enough response to moving obstacles, as this is also the measuring frequency of the Lidar module used.



```
Control loop missed its desired rate of 20.0000Hz... the loop actually took 0.1072 seconds
Control loop missed its desired rate of 20.0000Hz... the loop actually took 0.0683 seconds
Control loop missed its desired rate of 20.0000Hz... the loop actually took 0.0792 seconds
```

Figure 76: Warnings about the control loop rate.

Both the default and the used forward simulation parameters are shown in Table 12.

Table 12: Default and used values for the forward simulation parameters.

	sim_time	***vx_samples***	***vtheta_samples***	***controller_frequency***
Default value	1.0 s	3	20	20 Hz
Used value	2.0 s	10	25	10 Hz

When approaching the goal position, a robot can start to show large oscillatory movements and difficulty reaching a pose similar enough to the goal. This behaviour can be controlled by editing the goal tolerance parameters. These parameters are: *yaw_goal_tolerance* (tolerance in radians for the controller in yaw/rotation when achieving its goal), *xy_goal_tolerance* (tolerance in meters for the controller in the *x* and *y* distance when achieving a goal), and *latch_xy_goal_tolerance* (if enabled, when the robot reaches the goal *xy* location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so). Since the robot used in this dissertation showed to be able to achieve the goal position easily, the *xy_goal_tolerance* parameter was reduced to 0.05 m to increase the accuracy of the robot when reaching a goal. The default and used values of the goal tolerance parameters are shown in Table 13.

Table 13: Default and used values of the goal tolerance parameters.

	yaw_goal_tolerance	***xy_goal_tolerance***	***latch_xy_goal_tolerance***
Default value	0.1 m	0.05 rad	false
Used value	0.05 m	0.05 rad	false

The local planner configuration file used in this dissertation is included in Appendix B.6.2.

- **Costmaps:**

The local and global costmaps are essential for the planners to compute appropriate paths and velocities. In the ROS Navigation Stack, the robot's footprint is configured in the costmap package. This footprint was already defined in Section 5.2.2.

The costmap's inflation layer contains two parameters that change the inflation of cost values around obstacles. These are:

- *inflation_radius*. This defines the radius in meters to which the map inflates obstacle cost values;
- *cost_scaling_factor*. This is a scaling factor to apply to cost values during inflation. The farther from an obstacle a point inside the inflation radius is, the lower its cost value is. The higher the *cost_scaling_factor*, the stronger this cost decay is. This cost decay follows an exponential distribution with a negative exponent.

The smaller the inflation radius is, the less the robot will prioritize navigating far from obstacles. By increasing the inflation radius and decreasing the *cost_scaling_factor*, the robot will prioritize navigating further from the obstacles. If the inflation radius of two obstacles meets, the robot will have a bias towards navigating in the middle of them. This behaviour can be observed in Figure 77.

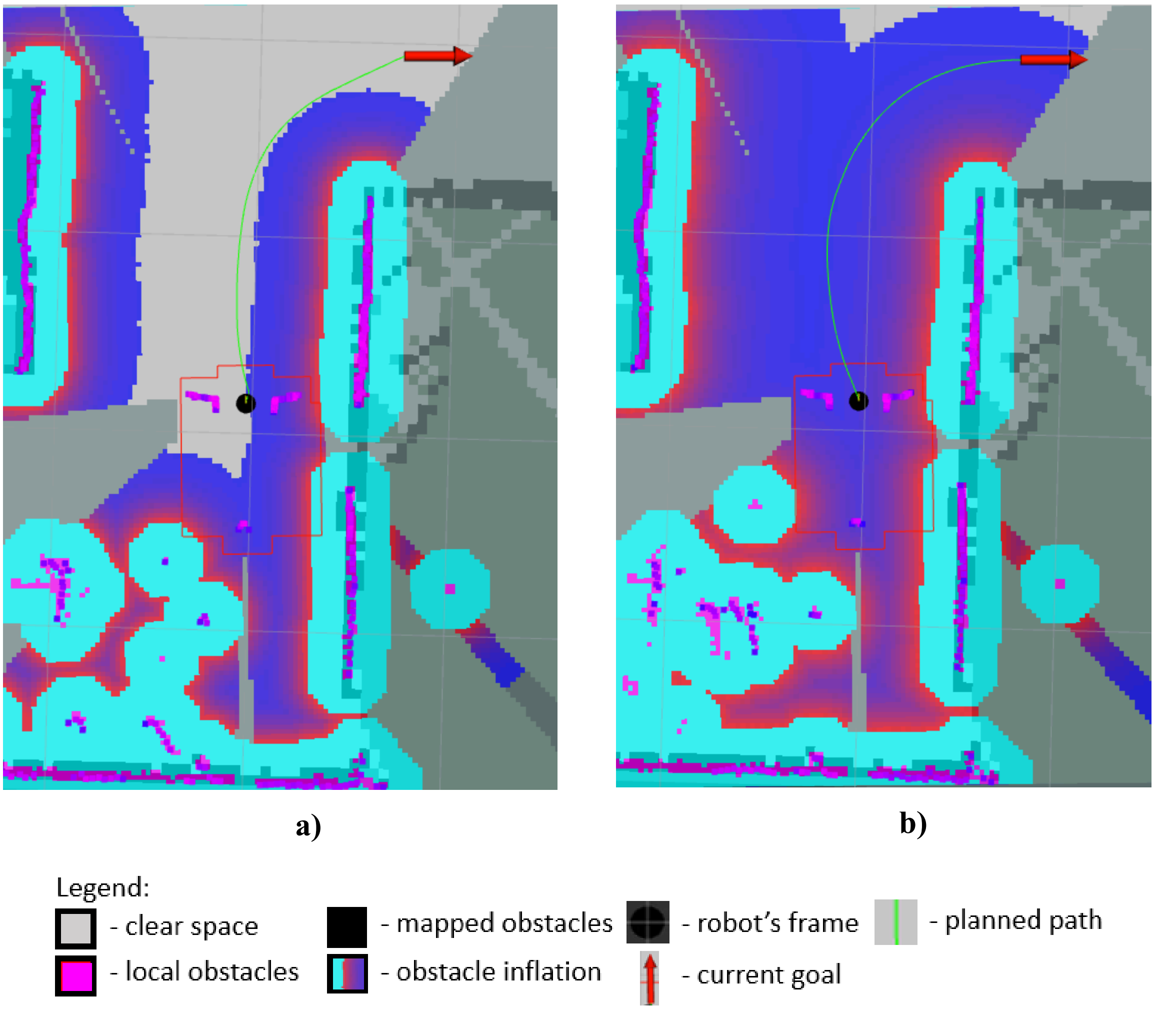


Figure 77: Global paths with respective inflation_radius and cost_scaling_factor of: a) 0.55 and 10.0; b) 1.0 and 6.0.

Therefore, if the priority is for the robot to reach its goal by travelling through a shorter path, a lower inflation radius and higher *cost_scaling_factor* should be used. On the contrary, if navigating farther away from obstacles is more important, a higher inflation radius and lower *cost_scaling_factor* should be used. In the case of this dissertation, the extra distance from obstacles was considered more important. The default and used values of these parameters are included in Table 14.

Table 14: Default and used values of the inflation layer parameters of the *costmap_2d* package.

	inflation_radius	***cost_scaling_factor***
Default value	0.55 m	10.0
Used value	1.0 m	6.0

A costmap's resolution is the size of each cell in the grid map containing a specific value and representing either occupied, free or unknown space. The lower the resolution, the more accurately obstacles and free space are represented in the costmap, however it will require a higher computational cost. In this case the global costmap was set to the default resolution of

0.05 m. This is the same resolution of the map recorded in Section 5.2.1. The local costmap however, was configured to use a lower resolution of 0.025 m. This allows the robot to navigate more easily through narrow places such as doors.

The *update_frequency* parameter specifies the desired rate at which to perform map updates in Hz. The local costmap's *update_frequency* was set to 10 Hz, to match the frequency of the laser scans and the local planner. The global costmap's *update_frequency* was set to 5 Hz to reduce the computational load.

The costmap configuration files used in this dissertation are included in Appendix B.5.

5.3 Implementation of Computer Vision

Computer vision can be important for mobile robots that require certain functions. They are of extreme importance in autonomous road vehicles, as they need to be able to adapt to different types of roads, acquire information from traffic markings and signs, and have a 3D view of their surroundings. For other types of mobile robots, computer vision can be used to extract information from the environment and use it for navigation or localization. An example of an autonomous vehicle incorporating all the above is Tesla's Autopilot system whose cameras have the vision coverage shown in Figure 78. Computer vision can also be used for identification of objects for robots with anthropomorphic arms to handle. These can be done utilizing various types of cameras [18, 45].

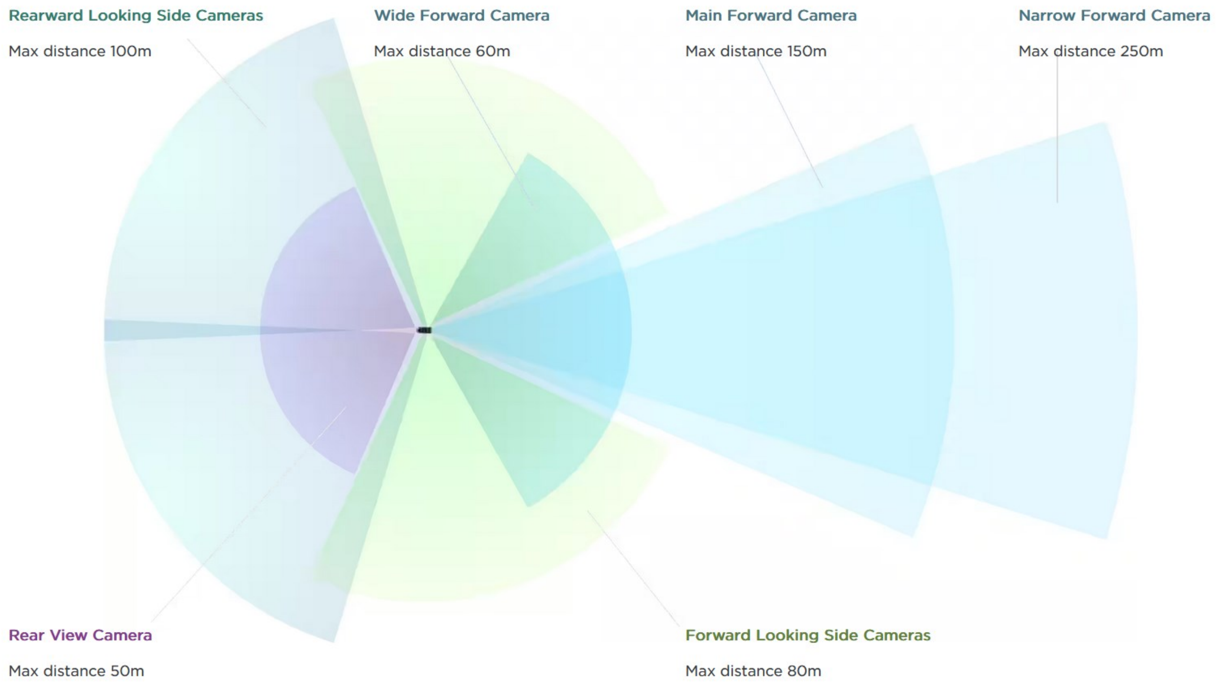


Figure 78: Tesla's Autopilot's sensor coverage [46].

A simpler use of computer vision in mobile robotics, is the use of markings with position coordinates to aid the robot's localization, to adjust errors obtained from odometry over time, or to feed certain information to the robot, such as goals to follow.

5.3.1 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. It is available in C++, Python, Java, and MATLAB; and supports Windows, Linux, Android, and MacOS. It contains a multitude of algorithms that can be used to detect and identify faces, follow eye movements, identify objects, track movements and moving objects, compute 3D point clouds from stereo (3D) cameras, manipulate images, among other uses. OpenCV also includes a statistical machine learning library with multiple different algorithms and neural network systems. OpenCV is also the primary vision package used in ROS. Information regarding OpenCV was obtained from https://opencv.org/.

Libraries like OpenCV can facilitate the use of computer vision data for mobile robots. Algorithms for production of 3D point clouds from stereo cameras and object identification algorithms are especially useful for AMRs.

5.3.2 Feeding goals via QR codes

To obtain information about a goal using computer vision, multiple approaches can be taken. Information that can be read by people can be used, such as presenting written position vectors to the camera, or arrows pointing in a specific direction with a number representing the distance. Popular methods of visual representation of data can also be used, such as barcodes or QR codes (quick response codes) containing the relevant information.

A ROS node was developed to read a QR code and publish the respective goal in the topic that the navigation stack uses for navigation goals. It utilizes the OpenCV library's algorithm for identifying and reading QR codes. It checks the camera's images for a QR code, reads it and takes in information containing the goal's coordinates and the appropriate frame. It then converts those coordinates in their original frame to the robot's odometry frame and publishes the goal message to the topic */move_base_simple/goal* that is used by the navigation stack.

Because of this, the QR codes need to have their information structured in a certain way. To work with this specific node, they need to contain the following information in the following order separated by "spaces": position in x, y and θ, and name of the respective coordinate frame in ROS. In Figure 79, an example of a QR code that follows this information structure is presented. Its information is the string "goal 1 0 0 base_link" shown above it, which gives the robot a goal of 1 meter in x in relation to the robot's own frame at the time of reading the QR code.

goal 1 0 0 base_link

Figure 79: QR code to give the robot a goal of 1 meter forward.

5.3.3 Landmark localization

Another program using OpenCV was developed to obtain information about the location of a QR code in each image frame given by the webcam. This includes the position of the vertices, centre, and orientation of the QR code in the camera's frame. This information is then transformed into position relative to the robot's axis. The QR code also contains information about its position in the map frame. All this information is added resulting in the robot's position in the map frame. This is then used to give an initial position to *AMCL* or to adjust odometry errors obtained over time and recalibrate the localization. The second use should not be necessary as while testing the robot's navigation using *AMCL*, the localization seemed to always improve the more the robot travelled until it reached a very accurate position.

The camera was positioned in a fixed position and parallel to the floor. A frame of the camera's vision (640 by 480 pixels) with a ruler on the floor was taken and can be seen in Figure 80. By analysing the image, it is possible to find that 640 pixels in the camera's vision corresponds to around 52 cm or 0.52 m in the floor's plane. With this, the value of 0.0008125 m/pixel can be used to calculate an approximation of the QR code's pose in relation to the robot's frame. Figure 81 shows a ruler with its bottom part positioned along the axis of both wheels. By analysing the image, the position of the robot's rotational axis and its frame in pixels can be obtained. In this case, the robot frame's position is (320 pixels, 258 pixels).

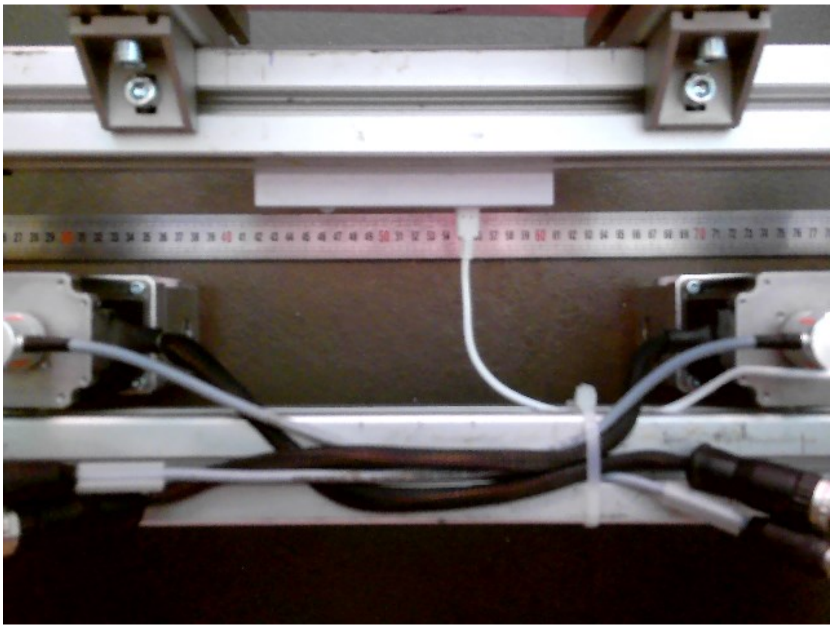


Figure 80: Ruler on the floor as seen by the robot's camera.

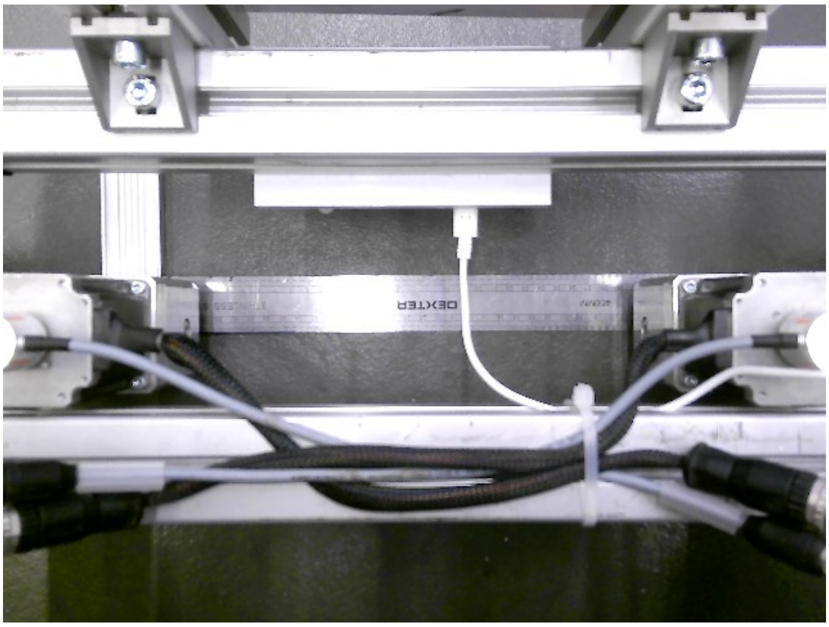


Figure 81: Ruler placed along the wheels' axis as seen by the robot's camera.

OpenCV's algorithm to detect and decode QR codes also gives the position of its vertices. By knowing the position of the vertices, it is possible to calculate the position of the QR code's centre and its angular orientation. We can calculate the robot's position in the frame relative to the QR code by subtracting the robot's frame position at (320 pixels, 258 pixels) with the QR code's centre. Multiplying the result by 0.0008125 m/pixel, the robot's position relative to the QR code in meters is obtained. By adding it and subtracting the QR code's angular orientation in radians to the position in the map frame given by the code, the robot's position in the map frame is obtained. It is then published in the */initialpose* ROS topic as a *PoseWithCovarianceStamped* message with null covariance.

Both this functionality and the functionality of feeding goals to the robot were implemented in the same program as they both utilize the same webcam and the same OpenCV algorithm. This program is present in Appendix B.8.

An example of a QR code used for feeding an initial position to the robot can be seen in Figure 82. The code's information is the string "initial_pos 1 0 0.5", which gives the robot an initial pose of 1 meter in x, 0 meters in y and 0.5 radians in θ, plus the offset between the robot's frame and the QR code's centre. This is in relation to the map frame.

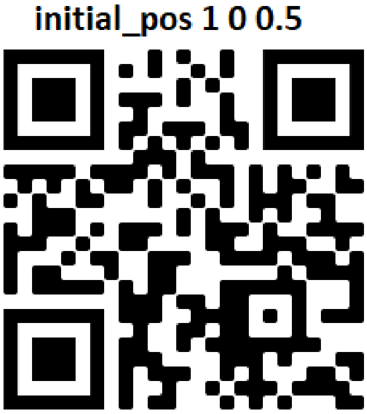


Figure 82: QR code that indicates that it is located at a position of (1, 0, 0.5) in the map frame.

5.4 Testing the Navigation System

To assess the quality, performance, and limitations of the navigation system and the robot, a test was set up. After performing the test three times, conclusions were taken.

5.4.1 *Methodology of the test*

This test consisted in the following steps:

- Feeding an accurate initial position to the robot using a QR code located on the floor;
- Using a QR code, feed the robot a position goal where another QR code with a different goal is located. This repeats four times until the final goal is reached. This final goal's position is the same as the initial position.

While navigating the environment of this test, the robot also needs to avoid static obstacles and moving people. The environment used was the same environment as the one mapped in Section 5.2.1. Figure 83 shows the map of the environment and red points with the order of the QR codes the robot should read.

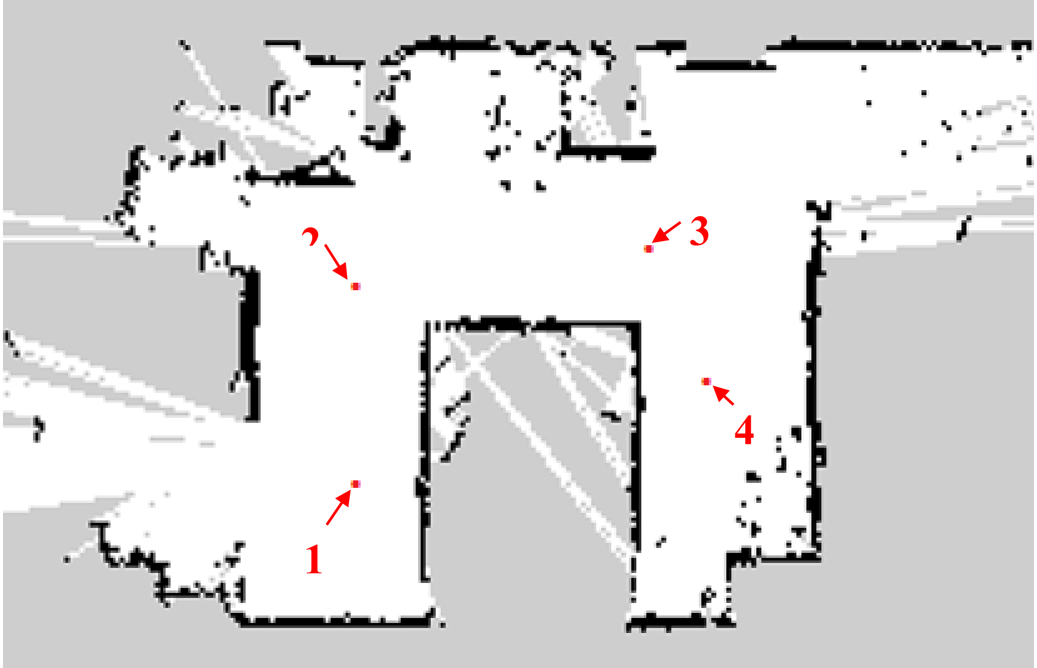


Figure 83: Test environment and QR code order.

The test starts with the robot in the position given by point 1 but with the webcam program that reads QR codes not running. After starting the test, the webcam program is turned on. This is to observe the difference between where the localization system initially thinks the robot is, and where it thinks it is after it reads the QR code located in point 1 containing its location on the map. After this localization, a user moves a QR code with the position goal of point 2 in front of the robot's camera. After navigating to point 2, the robot should read the QR code on the ground that gives it a position goal equivalent to point 3, and subsequently do the same from point 3 to point 4, and from point 4 directly to point 1. After navigating back to point 1, the test ends.

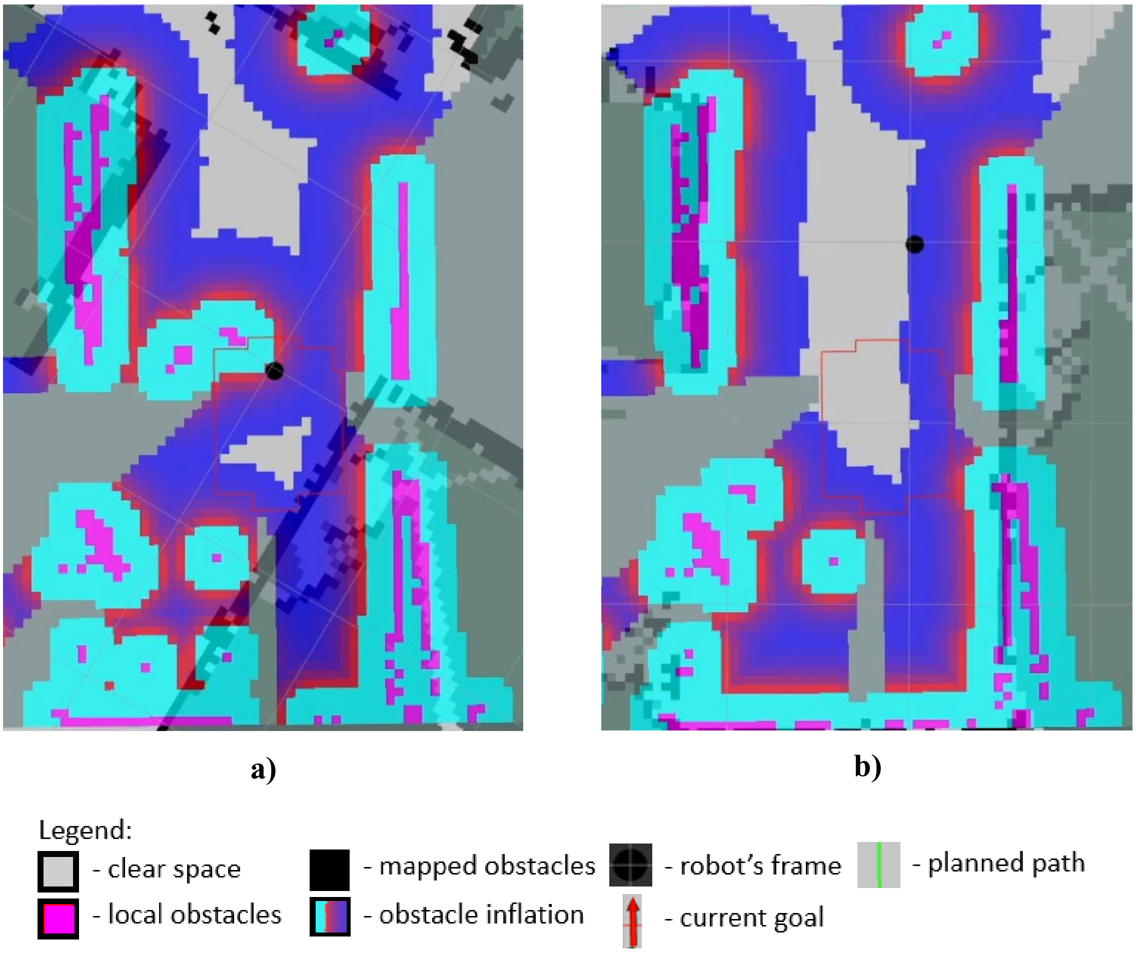
The last goal given to the robot to travel from point 4 to point 1 requires a longer and more complex path than the other goals. This is used to test if there are any differences in behaviour when performing trajectories of different lengths.

5.4.2 Results and observations

The test was performed three times in total. During these tests the following behaviours were observed:

- The initial position given by the first QR code proved to be extremely accurate even if the robot started at different linear and angular positions, as long as the QR code was visible to the camera. Figure 84 shows the robot's localization before and after reading the QR code in one of the tests respectively;
- Sometimes, when navigating to the next QR code, the robot showed oscillatory movements while progressing very slowly;
- When the robot reached a goal, sometimes the next QR code would not be completely visible to the camera. This required a user to move the robot slightly so that the QR code would be completely visible, and the robot could proceed to the next goal. This behaviour is shown in Figure 85 and Figure 86;

- During one of the three tests, the robot was not able to find a possible trajectory from point 4 back to point 1, and the test ended there;
- The robot did not hit any obstacle or person. When a person moved in front of the robot's trajectory, the robot would stop and try to calculate a new possible trajectory.


a)

b)

Figure 84: Robot's localization before and after being given an initial position: a) before; b) after.

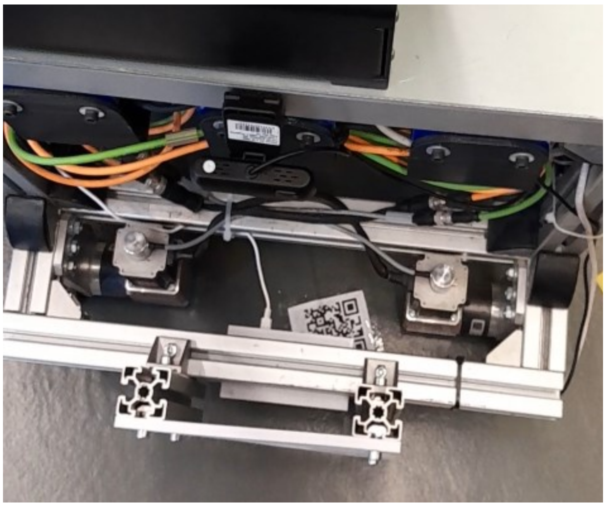


Figure 85: Goal achieved where the QR code is not fully visible.

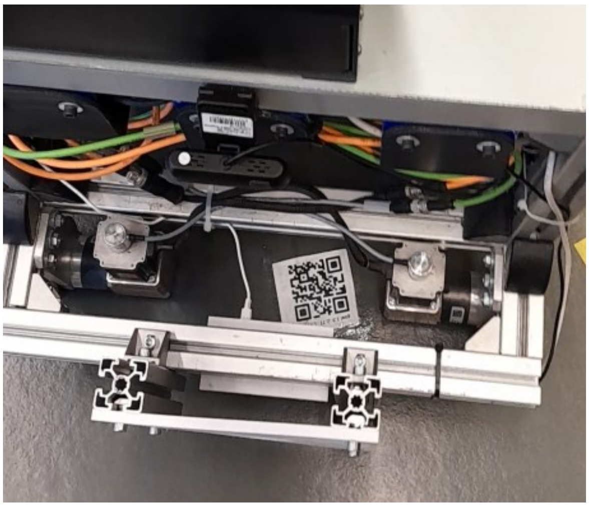


Figure 86: QR code is fully visible after the user manually adjusts the robot's position.

6 Conclusions and Future Work

All the different subsystems developed showed to be able to function and communicate with each other successfully, resulting in a complete navigation system.

The navigation system proved to be able to successfully navigate the environment, avoid both static and moving obstacles and reach specific goals given to it in multiple different coordinate frames (robot's frame, odometry frame, and the map's frame). The localization also proved to be very accurate.

The tuning of the odometry and the navigation stack parameters also showed significant improvement in the robot's navigation performance by reducing positional errors and allowing for a faster and more accurate navigation of the environment.

However, some limitations of the navigation system were also noted. These limitations are:

1) Sometimes, occurring often in narrow spaces, the robot would exhibit oscillatory movements while progressing towards its goal very slowly;
2) Sometimes, the robot gave up on reaching a certain goal because with the information it currently has, there is no possible trajectory it can take. However, this could be due to moving obstacles that are no longer in the Lidar's field of view;
3) The lidar module's field of view possessed large dead zones due to it being obstructed by the wheels, gearboxes, and motors of the robot. This caused problems in not detecting certain obstacles, and in not clearing moving obstacles that the robot thinks are now located in its dead zones, but already moved away;
4) The camera has a very low useful field of view in the position it was used. Since it is a normal 2D camera, it was installed in a parallel plane to the floor to be able to calculate the distance to QR codes. However, this resulted in its vision being cluttered by the robot's bumper, wheels, motors, gearboxes, the Raspberry Pi's case, and other components, as it can be seen in Figure 85 and Figure 86 in Section 5.4.2.
5) The robot is not equipped with a battery. It was used by connecting it to the domestic power grid. This resulted in navigation issues where the robot could not travel away too much, and a user had to walk around the environment handling the robot's cables. Sometimes, the cables would be in the Lidar's field of view and make the navigation system assume there were obstacles in the cables' position.

To improve the performance of the robot and its navigation system, certain solutions are suggested to be explored as future work:

- Limitation number 1 can be due to several issues. One of those are performance issues with the navigation system on the Raspberry Pi. It is suggested that to solve this, a more powerful PC should be used. This would also allow for a different tuning of the ROS navigation stack parameters, allowing for each subsystem to process at faster frequencies, and for each algorithm to be more accurate and complete. Another issue is with the robot's geometry. A lot of these algorithms are made to work in circular or square robot's that have their rotational axis on their geometrical centre. The robot used in this dissertation is very large, with an elongated shape, and its rotational axis is positioned close to one of its edges. This results in the global and local planner having a worse performance. The current geometry of the robot and a suggested improved geometry and wheel disposition can be seen in Figure 87 and Figure 88 respectively;

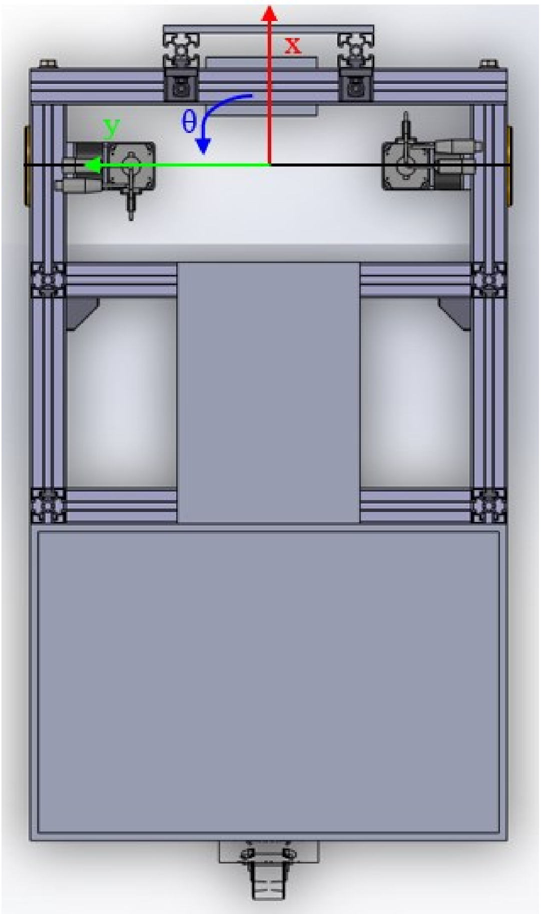


Figure 87: Geometry of the robot used in this dissertation.

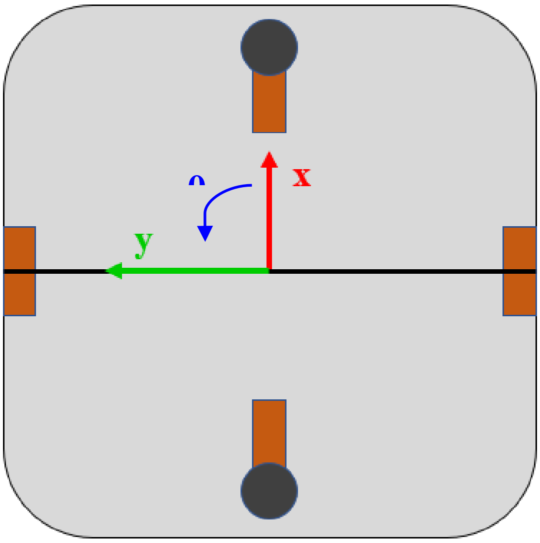


Figure 88: Suggested improved geometry and wheel disposition for the robot

- Limitations number 2 and 3 could be solved by installing additional Lidar modules that can cover the original blind spots. Additionally, 3D cameras or multiple normal cameras simulating a 3D camera, can also be installed and used to observe the robot's surrounding and publish points corresponding to obstacles in ROS. For limitation number 2 specifically, an additional program could be developed that would make the robot travel to a closer possible goal and rotate in place before giving up. This would give more time and an increased field of view for the robot to observe if there is still no possible trajectory to the final goal.
- Limitation number 4 could be solved by either installing more cameras, or 3D cameras, allowing it to not require the cameras to be parallel to the floor and be able to observe much more of the robot's surroundings.
- Limitation number 5 only requires an appropriate battery to be installed on the robot. A charging interface would also need to be designed.

Bibliography

[1] R. Siegwart and I. R. Nourbakhsh, Introduction to Autonomous Mobile Robots, Cambridge, Massachusetts: The MIT Press, 2004.

[2] F. Rubio, F. Valero and C. Llopis-Albert, A review of mobile robots: Concepts, methods, theoretical framework, and applications, Valencia, Spain: International Journal of Advanced Robotic Systems, 2019.

[3] L. Jaulin, Mobile Robotics, 2nd ed., London: ISTE Ltd; John Wiley & Sons, 2019.

[4] U. Nehmzow, Mobile robotics: a practical introduction, 2nd ed., Springer, 2012, pp. 1-21.

[5] NASA, "Mars Exploration Rovers Overview," [Online]. Available: https://mars.nasa.gov/mer/mission/overview/. [Accessed 2 June 2022].

[6] AgileX, "AgileX," [Online]. Available: https://global.agilex.ai/products/bunker-pro. [Accessed 2 June 2022].

[7] B. Dynamics, "Boston Dynamics," [Online]. Available: https://www.bostondynamics.com/products/spot. [Accessed 2 June 2022].

[8] ELESA+GANTER, "ELESA+GANTER," [Online]. Available: https://www.elesa-ganter.com/en/www/products/castors-and-wheels--1/Castors-and-wheels--Mould-on-polyurethane-wheels--REF4-RBL#sortby=0&facetvalue=. [Accessed 2 June 2022].

[9] MÄDLER, "MÄDLER," [Online]. Available: https://www.maedler.de/product/1643/12021/apparaterollen-lenkrollen-mit-lochplatte-leichte-ausfuehrung. [Accessed 2 June 2022].

[10] "VEX Robotics," VEX Robotics, [Online]. Available: https://www.vexrobotics.com/mecanum-wheels.html. [Accessed 2 June 2022].

[11] L. Hertig, D. Schindler, M. Bloesch, C. D. Remy and R. Siegwart, "Unified State Estimation for a Ballbot," *Proceedings - IEEE International Conference on Robotics and Automation,* 2013.

[12] A. Z. S. B. I. Š. Gregor Klančar, Wheeled Mobile Robotics, Butterworth-Heinemann, 2017.

[13] Oz Robotics, "Oz Robotics," [Online]. Available: https://ozrobotics.com/shop/3wd-triangular-100mm-omni-wheel-mobile-robotics-car-10003/. [Accessed 29 August 2022].

[14] Encoder Products Company, "The Basics of How an Encoder Works," 2019. [Online]. Available: https://www.encoder.com/wp2011-basics-how-an-encoder-works. [Accessed 13 June 2022].

[15] Vex Robotics, "Vex Education," [Online]. Available: https://education.vex.com/stemlabs/cs/computer-science-level-1-blocks/navigating-a-maze/lesson-1-what-is-a-bumper-sensor. [Accessed 29 August 2022].

[16] F. Mirelez-Delgado, J. Díaz-Paredes and M. Gallardo-Carreón, "Stewart-Gough Platform: Design and Construction with a Digital PID Controller Implementation.," in *Automation and Control*, 2020.

[17] Slamtec, "RPLIDAR S1," [Online]. Available: https://www.slamtec.com/en/Lidar/S1. [Accessed 13 June 2022].

[18] N. Gryaznov and A. Lopota, "Computer Vision for Mobile On-Ground Robotics," *25th DAAAM International Symposium on Intelligent Manufacturing and Automation, DAAAM 2014,* 2015.

[19] Logitech, "Logitech," [Online]. Available: https://www.logitech.com/en-us/products/webcams/c922-pro-stream-webcam.960-001087.html. [Accessed 29 August 2022].

[20] ROS, "ROS Wiki," [Online]. Available: http://wiki.ros.org/slam_gmapping/Tutorials/MappingFromLoggedData.

[21] M. Wise, *Understanding the Basics of AMR Technology: What You Need to Know,* Association for Advancing Automation.

[22] T. d. R. Anjo, "Implementação e controlo de um robô móvel com braço antropomórfico," *Faculdade de Engenharia da Universidade do Porto,* 2021.

[23] Beckhoff, "TwinCAT automation software," [Online]. Available: https://www.beckhoff.com/en-en/products/automation/twincat/. [Accessed 7 June 2022].

[24] M. Köseoğlu, O. M. Çelik and Ö. Pektaş, "Design of an autonomous mobile robot based on ROS," *2017 International Artificial Intelligence and Data Processing Symposium (IDAP),* pp. 1-5, 2017.

[25] L. Zhi and M. Xuesong, "Navigation and Control System of Mobile Robot Based on ROS," *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC),* pp. 368-372, 2018.

[26] S. Gatesichapakorn, J. Takamatsu and M. Ruchanurucks, "ROS based Autonomous Mobile Robot Navigation using 2D LiDAR and RGB-D Camera," *2019 First International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP),* pp. 151-154, 2019.

[27] Q. Xu, J. Zhao, C. Zhang and F. He, "Design and implementation of an ROS based autonomous navigation system," *2015 IEEE International Conference on Mechatronics and Automation (ICMA),* pp. 2220-2225, 2015.

[28] Y. Li and C. Shi, "Localization and Navigation for Indoor Mobile Robot Based on ROS," *2018 Chinese Automation Congress (CAC),* pp. 1135-1139, 2018.

[29] K. Zheng, "Ros navigation tuning guide," in *Robot Operating System (ROS)*, Springer, 2021, pp. 197-226.

[30] X. Xiao, B. Liu, G. Warnell, J. Fink and P. Stone, "APPLD: Adaptive Planner Parameter Learning From Demonstration," *IEEE Robotics and Automation Letters,* vol. 5, no. 3, pp. 4541-4547, 2020.

[31] N. Bharathiraman, A. Kaundanya, J. Singhal, Y. Wadalkar and K. Talele, " An Empirical Approach for Tuning an Autonomous Mobile Robot in Gazebo," *Computational Vision and Bio-Inspired Computing. Advances in Intelligent Systems and Computing,* vol. 1420, 2022.

[32] [Online]. Available: https://img.dott.pt/EHSPAfOAvW03Tf5UMpex_F4r492xHkrL_xOm2R-zVOQ/fit/1500/1500/no/0/aHR0cHM6Ly9kMWt2Zm95cmlmNnd6Zy5jbG91ZGZyb250Lm5ldC9hc3NldHMvaW1hZ2VzLzgwL21haW4vbm9uZV82MDhmZjQ3YmExYWU5M2FlNTFjYjQzMmM5YTRmZjViYV82MDhmZjQ3LkpQRUc.jpeg. [Accessed 14 June 2022].

[33] OKdo, "Lidar Module with Bracket," [Online]. Available: https://www.okdo.com/p/lidar-module-with-bracket/. [Accessed 14 June 2022].

[34] Mauser, "Webcam AutoFocus 4Mpx 1440p c/ microfone USB2.0," [Online]. Available: https://mauser.pt/catalog/product_info.php?cPath=641_1350&products_id=013-0382. [Accessed 14 June 2022].

[35] plammers, "Raspberry Pi 4 case," Thingiverse, [Online]. Available: https://www.thingiverse.com/thing:3714695. [Accessed 8 June 2022].

[36] Beckhoff, "MC_MoveVelocity," [Online]. Available: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_mc2/70102411.html&id=. [Accessed 15 June 2022].

[37] Beckhoff, "MC_Halt," [Online]. Available: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_mc2/70107019.html&id=. [Accessed 15 June 2022].

[38] S. Kumar Malu and J. Majumdar, "Kinematics, Localization and Control of Differential Drive Mobile Robot," *Global Journal of Researches in Engineering: H Robotics & Nano-Tech,* vol. 14, 2014.

[39] Beckhoff, "MC_MoveRelative," [Online]. Available: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_mc2/70102411.html&id=. [Accessed 7 July 2022].

[40] T. YoungWonks, "A look at the GPIO pins on a Raspberry Pi," 2020. [Online]. Available: https://www.youngwonks.com/blog/Raspberry-Pi-4-Pinout. [Accessed 21 June 2022].

[41] ROS, "Setup and Configuration of the Navigation Stack on a Robot," 2018. [Online]. Available: http://wiki.ros.org/navigation/Tutorials/RobotSetup. [Accessed 22 June 2022].

[42] ROS, "costmap_2d," 2018. [Online]. Available: http://wiki.ros.org/costmap_2d?distro=noetic. [Accessed 27 June 2022].

[43] F. Dellaert, D. Fox, W. Burgard and S. Thrun, "Monte Carlo Localization for Mobile Robots," *Proceedings - IEEE International Conference on Robotics and Automation,* vol. 2, 1999.

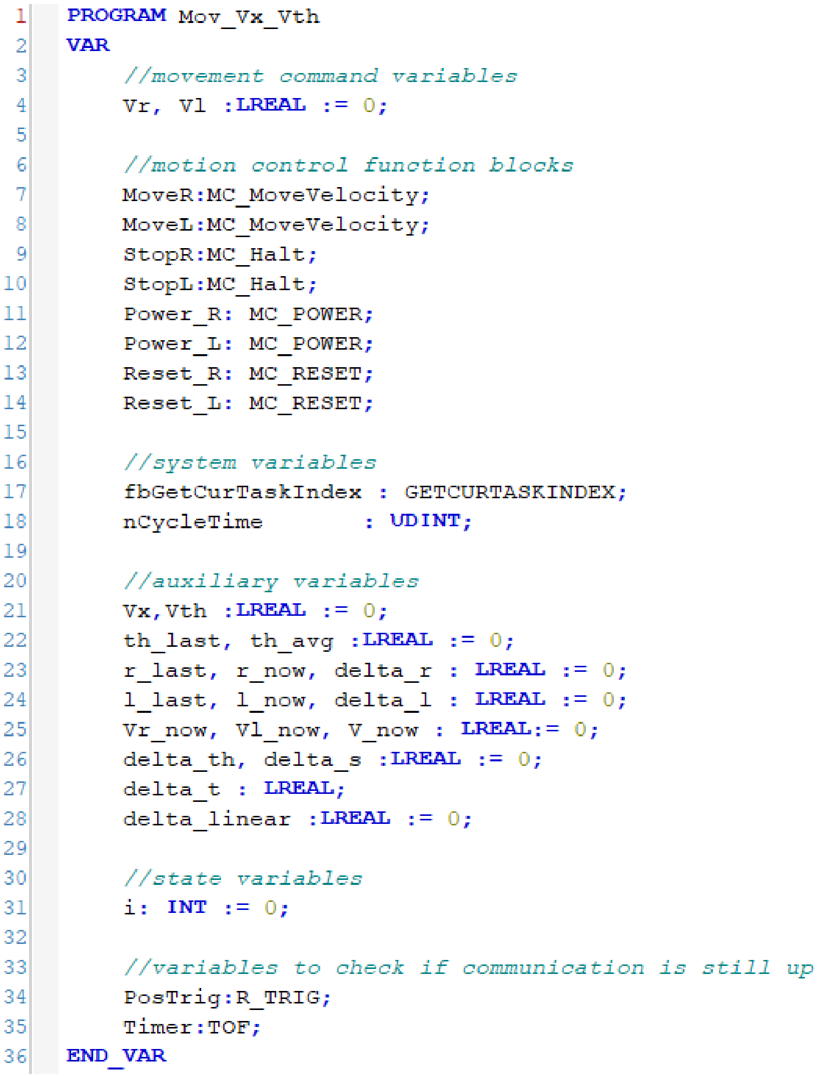
[44] S. Thrun, W. Burgard and D. Fox, Probabilistic Robotics, MIT Press, 2005.

[45] A. S. S. Oliveira, M. C. dos Reis, F. A. X. da Mota, M. E. M. Martinez and A. R. Alexandria, "New trends on computer vision applied to mobile robot localization," *Internet of Things and Cyber-Physical Systems,* 2022.

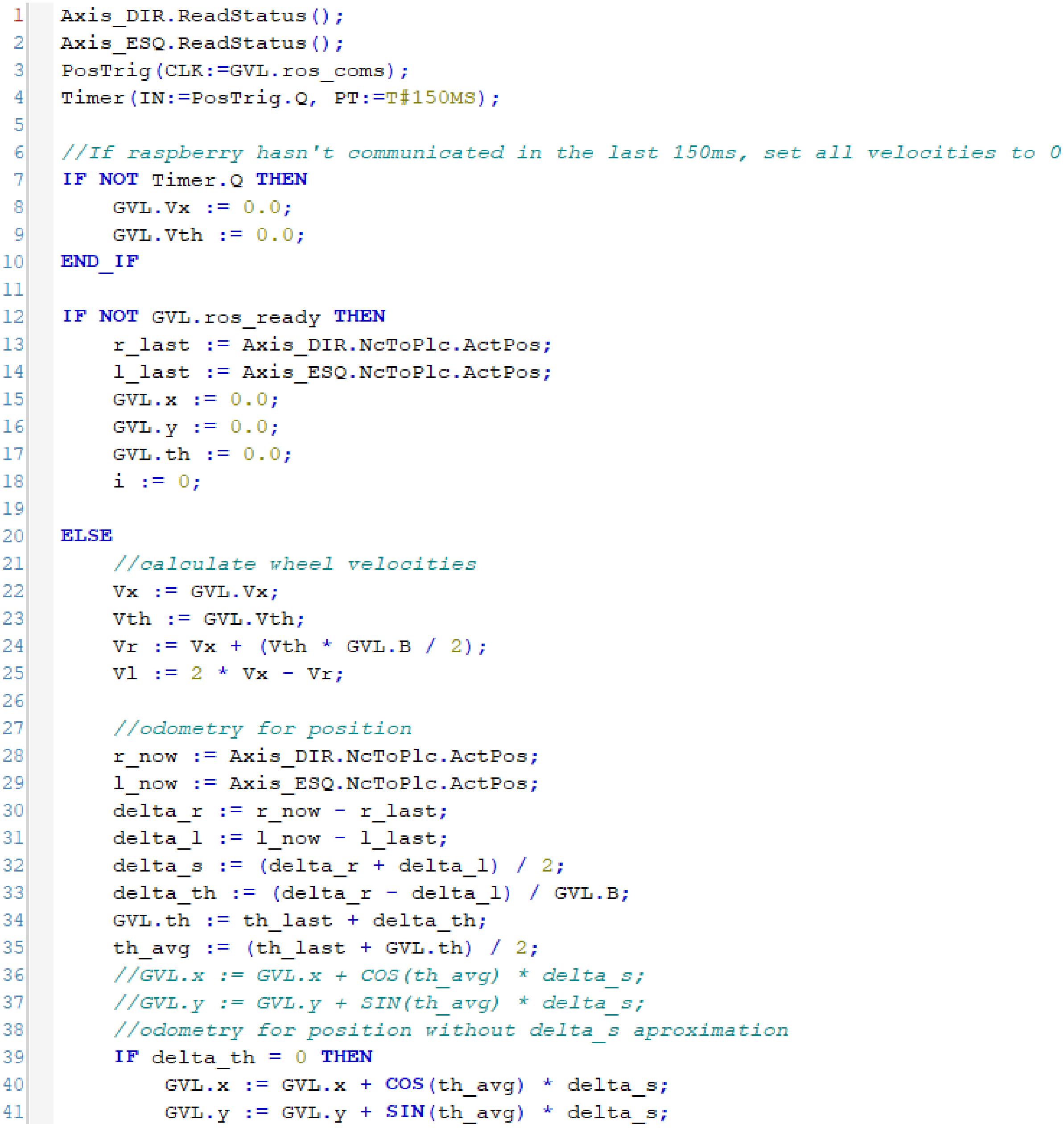
[46] Tesla, "Future of Driving," [Online]. Available: https://www.tesla.com/autopilot. [Accessed 28 June 2022].

Appendix A: TwinCAT 3 Programs

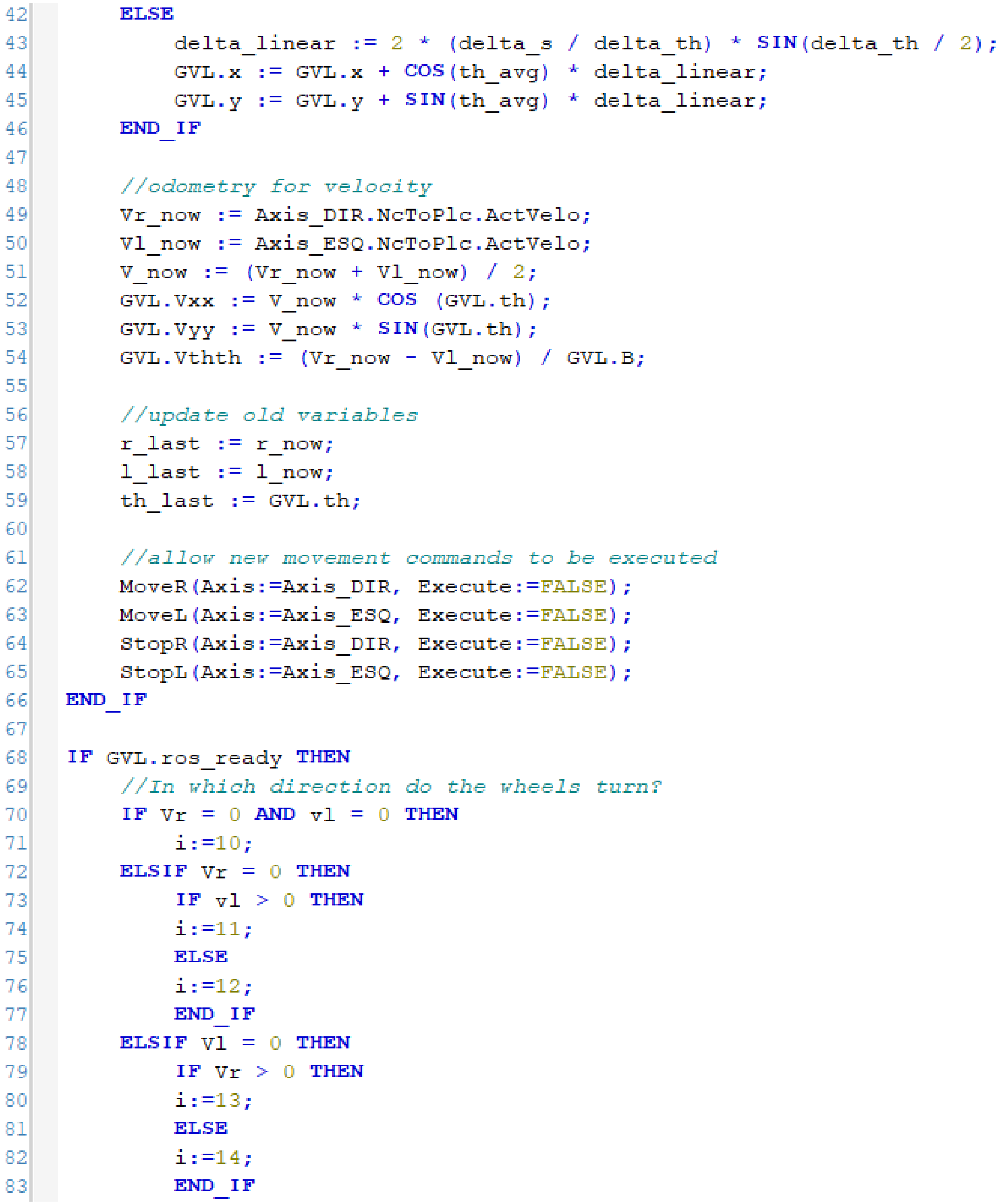
A.1 Movement command and odometry program in TwinCAT 3



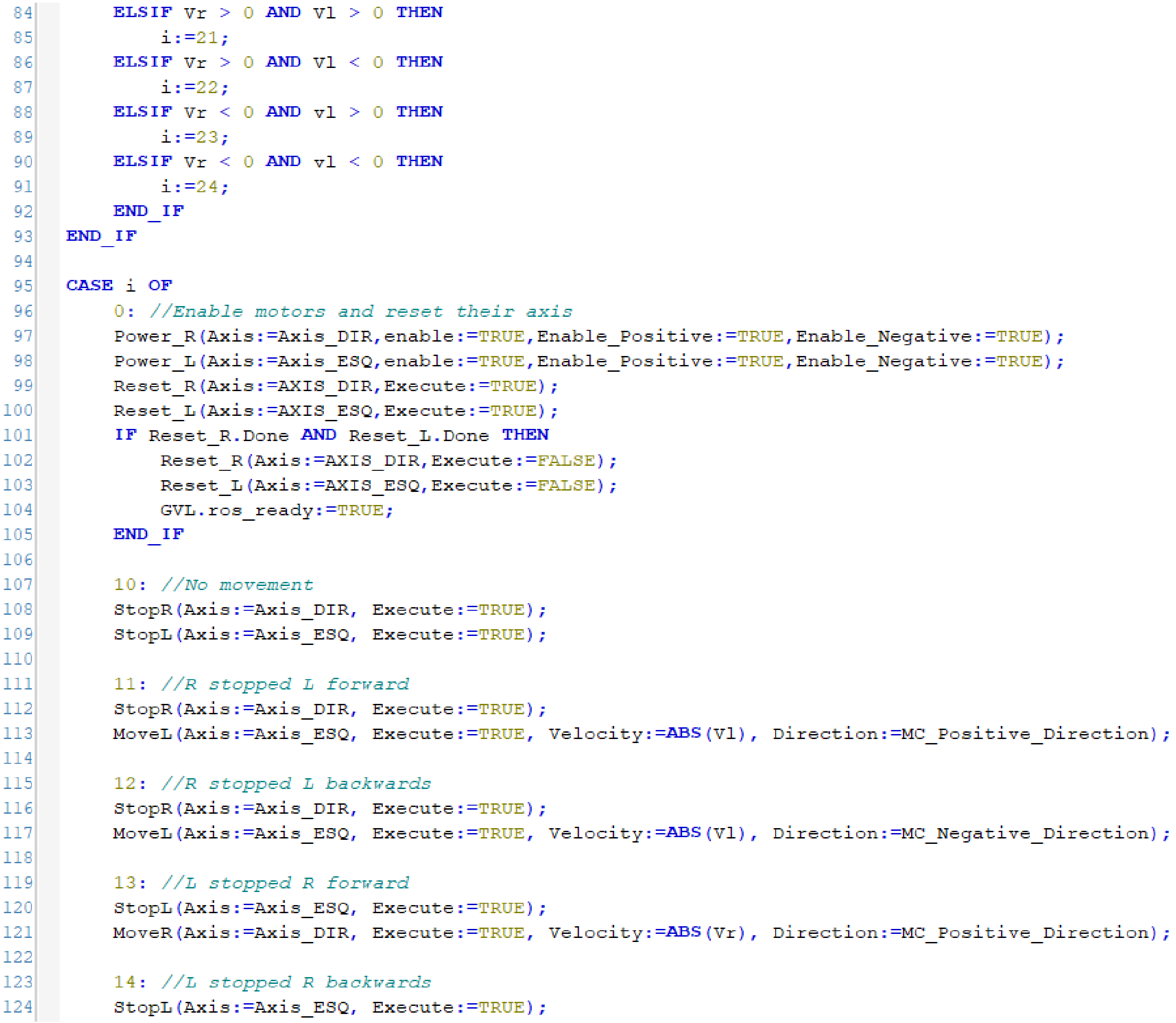
```
PROGRAM Mov_Vx_Vth
VAR
    //movement command variables
    Vr, Vl :LREAL := 0;

    //motion control function blocks
    MoveR:MC_MoveVelocity;
    MoveL:MC_MoveVelocity;
    StopR:MC_Halt;
    StopL:MC_Halt;
    Power_R: MC_POWER;
    Power_L: MC_POWER;
    Reset_R: MC_RESET;
    Reset_L: MC_RESET;

    //system variables
    fbGetCurTaskIndex : GETCURTASKINDEX;
    nCycleTime       : UDINT;

    //auxiliary variables
    Vx,Vth :LREAL := 0;
    th_last, th_avg :LREAL := 0;
    r_last, r_now, delta_r : LREAL := 0;
    l_last, l_now, delta_l : LREAL := 0;
    Vr_now, Vl_now, V_now : LREAL:= 0;
    delta_th, delta_s :LREAL := 0;
    delta_t : LREAL;
    delta_linear :LREAL := 0;

    //state variables
    i: INT := 0;

    //variables to check if communication is still up
    PosTrig:R_TRIG;
    Timer:TOF;
END_VAR
```



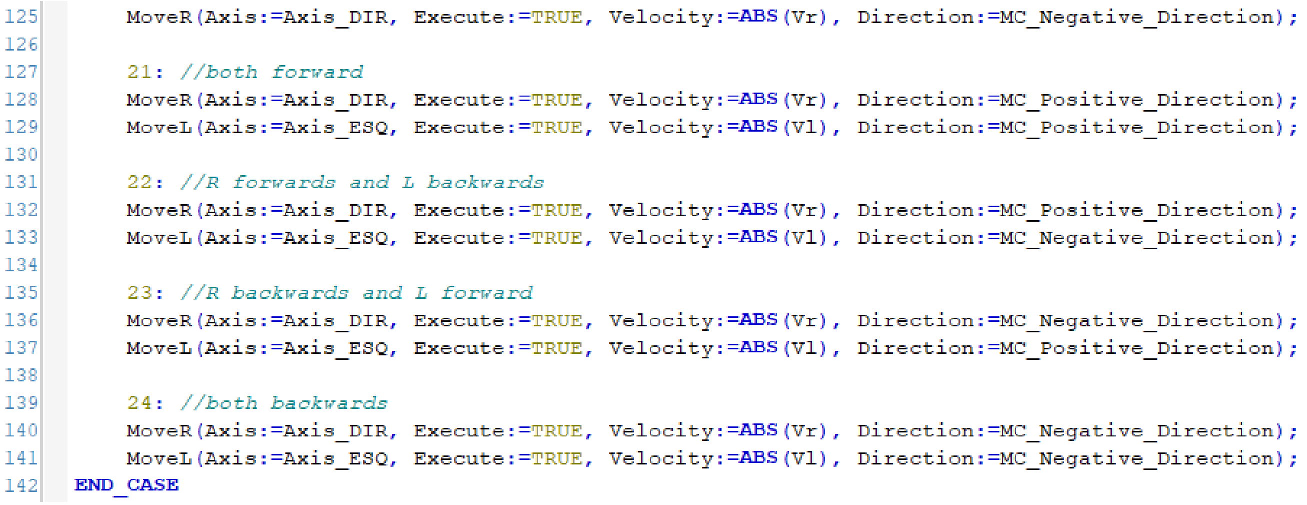
```
Axis_DIR.ReadStatus();
Axis_ESQ.ReadStatus();
PosTrig(CLK:=GVL.ros_coms);
Timer(IN:=PosTrig.Q, PT:=T#150MS);

//If raspberry hasn't communicated in the last 150ms, set all velocities to 0
IF NOT Timer.Q THEN
    GVL.Vx := 0.0;
    GVL.Vth := 0.0;
END_IF

IF NOT GVL.ros_ready THEN
    r_last := Axis_DIR.NcToPlc.ActPos;
    l_last := Axis_ESQ.NcToPlc.ActPos;
    GVL.x := 0.0;
    GVL.y := 0.0;
    GVL.th := 0.0;
    i := 0;

ELSE
    //calculate wheel velocities
    Vx := GVL.Vx;
    Vth := GVL.Vth;
    Vr := Vx + (Vth * GVL.B / 2);
    Vl := 2 * Vx - Vr;

    //odometry for position
    r_now := Axis_DIR.NcToPlc.ActPos;
    l_now := Axis_ESQ.NcToPlc.ActPos;
    delta_r := r_now - r_last;
    delta_l := l_now - l_last;
    delta_s := (delta_r + delta_l) / 2;
    delta_th := (delta_r - delta_l) / GVL.B;
    GVL.th := th_last + delta_th;
    th_avg := (th_last + GVL.th) / 2;
    //GVL.x := GVL.x + COS(th_avg) * delta_s;
    //GVL.y := GVL.y + SIN(th_avg) * delta_s;
    //odometry for position without delta_s aproximation
    IF delta_th = 0 THEN
        GVL.x := GVL.x + COS(th_avg) * delta_s;
        GVL.y := GVL.y + SIN(th_avg) * delta_s;
```



```
    ELSE
        delta_linear := 2 * (delta_s / delta_th) * SIN(delta_th / 2);
        GVL.x := GVL.x + COS(th_avg) * delta_linear;
        GVL.y := GVL.y + SIN(th_avg) * delta_linear;
    END_IF

    //odometry for velocity
    Vr_now := Axis_DIR.NcToPlc.ActVelo;
    Vl_now := Axis_ESQ.NcToPlc.ActVelo;
    V_now := (Vr_now + Vl_now) / 2;
    GVL.Vxx := V_now * COS (GVL.th);
    GVL.Vyy := V_now * SIN(GVL.th);
    GVL.Vthth := (Vr_now - Vl_now) / GVL.B;

    //update old variables
    r_last := r_now;
    l_last := l_now;
    th_last := GVL.th;

    //allow new movement commands to be executed
    MoveR(Axis:=Axis_DIR, Execute:=FALSE);
    MoveL(Axis:=Axis_ESQ, Execute:=FALSE);
    StopR(Axis:=Axis_DIR, Execute:=FALSE);
    StopL(Axis:=Axis_ESQ, Execute:=FALSE);
END_IF

IF GVL.ros_ready THEN
    //In which direction do the wheels turn?
    IF Vr = 0 AND vl = 0 THEN
        i:=10;
    ELSIF Vr = 0 THEN
        IF vl > 0 THEN
        i:=11;
        ELSE
        i:=12;
        END_IF
    ELSIF Vl = 0 THEN
        IF Vr > 0 THEN
        i:=13;
        ELSE
        i:=14;
        END_IF
```

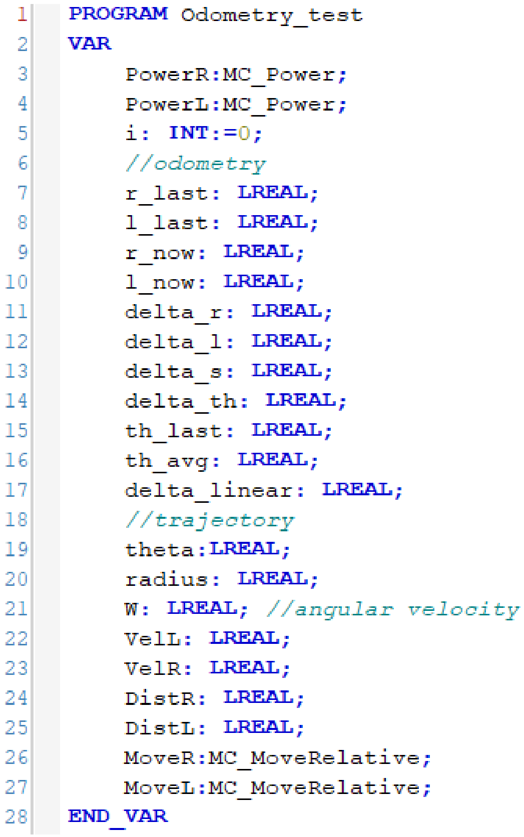


```
    ELSIF Vr > 0 AND Vl > 0 THEN
        i:=21;
    ELSIF Vr > 0 AND Vl < 0 THEN
        i:=22;
    ELSIF Vr < 0 AND vl > 0 THEN
        i:=23;
    ELSIF Vr < 0 AND vl < 0 THEN
        i:=24;
    END_IF
END_IF

CASE i OF
    0: //Enable motors and reset their axis
    Power_R(Axis:=Axis_DIR,enable:=TRUE,Enable_Positive:=TRUE,Enable_Negative:=TRUE);
    Power_L(Axis:=Axis_ESQ,enable:=TRUE,Enable_Positive:=TRUE,Enable_Negative:=TRUE);
    Reset_R(Axis:=AXIS_DIR,Execute:=TRUE);
    Reset_L(Axis:=AXIS_ESQ,Execute:=TRUE);
    IF Reset_R.Done AND Reset_L.Done THEN
        Reset_R(Axis:=AXIS_DIR,Execute:=FALSE);
        Reset_L(Axis:=AXIS_ESQ,Execute:=FALSE);
        GVL.ros_ready:=TRUE;
    END_IF

    10: //No movement
    StopR(Axis:=Axis_DIR, Execute:=TRUE);
    StopL(Axis:=Axis_ESQ, Execute:=TRUE);

    11: //R stopped L forward
    StopR(Axis:=Axis_DIR, Execute:=TRUE);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Positive_Direction);

    12: //R stopped L backwards
    StopR(Axis:=Axis_DIR, Execute:=TRUE);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Negative_Direction);

    13: //L stopped R forward
    StopL(Axis:=Axis_ESQ, Execute:=TRUE);
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Positive_Direction);

    14: //L stopped R backwards
    StopL(Axis:=Axis_ESQ, Execute:=TRUE);
```

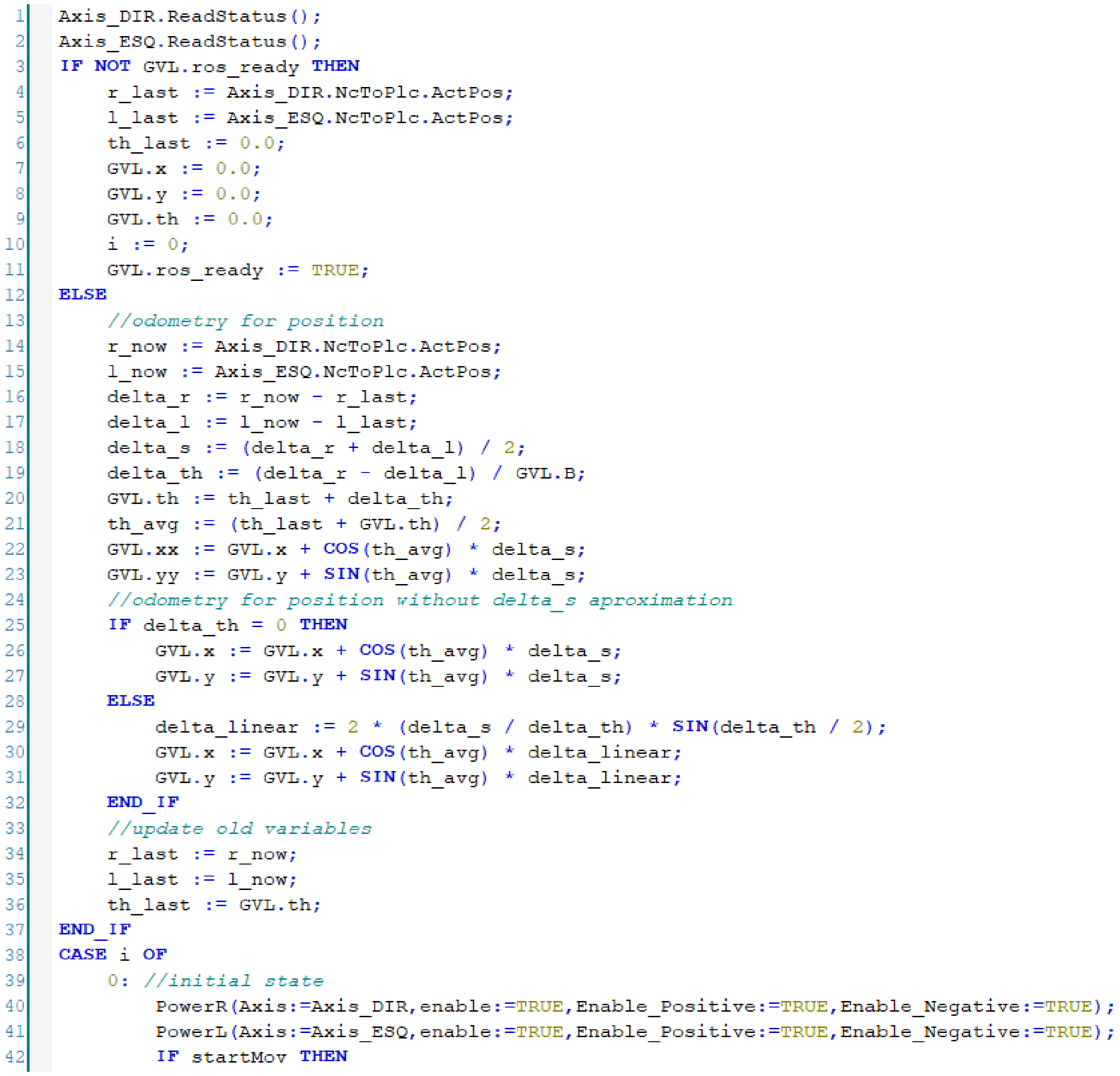


```
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Negative_Direction);

    21: //both forward
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Positive_Direction);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Positive_Direction);

    22: //R forwards and L backwards
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Positive_Direction);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Negative_Direction);

    23: //R backwards and L forward
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Negative_Direction);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Positive_Direction);

    24: //both backwards
    MoveR(Axis:=Axis_DIR, Execute:=TRUE, Velocity:=ABS(Vr), Direction:=MC_Negative_Direction);
    MoveL(Axis:=Axis_ESQ, Execute:=TRUE, Velocity:=ABS(Vl), Direction:=MC_Negative_Direction);
END_CASE
```

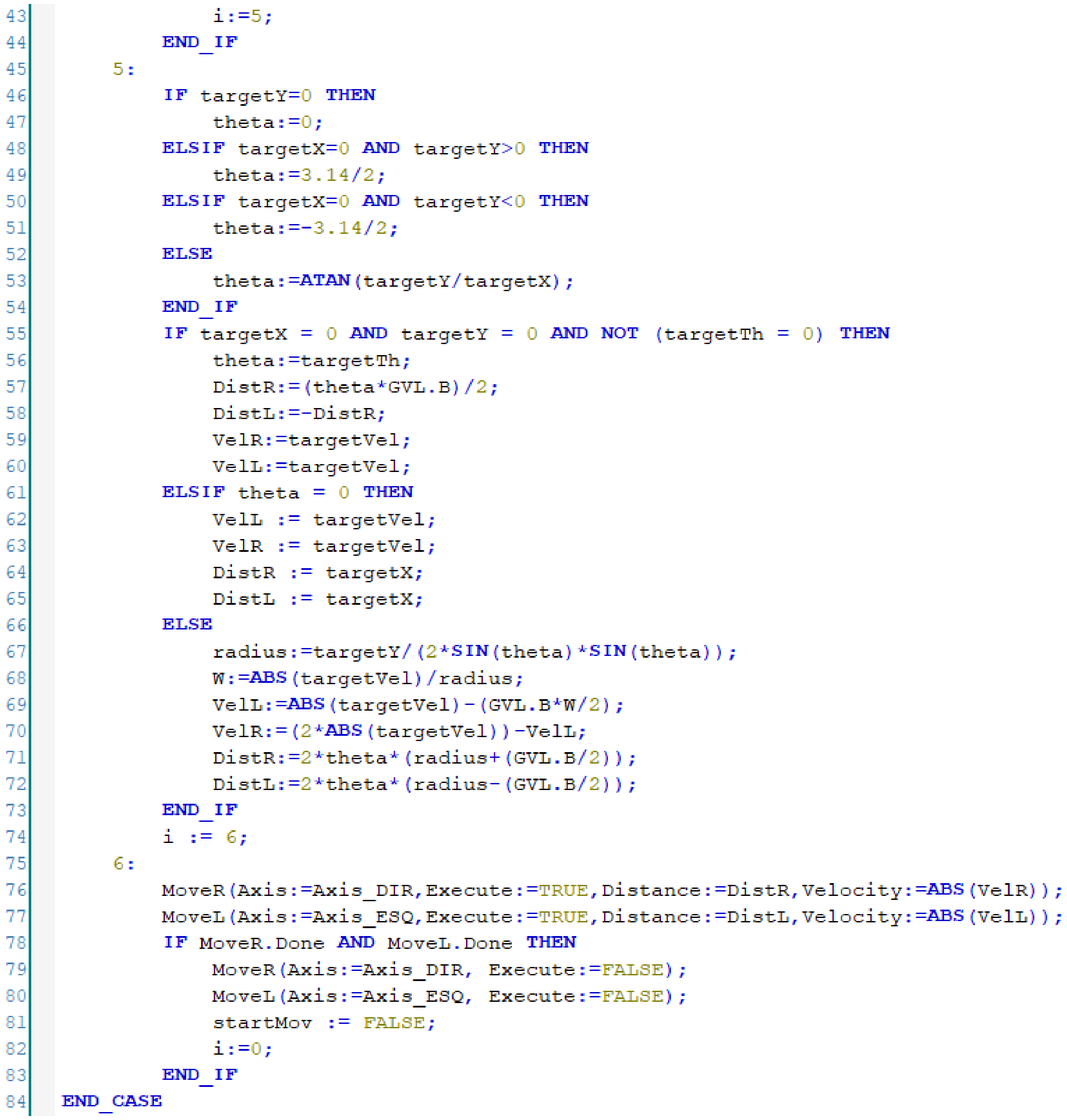
A.2 TwinCAT program to perform pre-defined trajectories to test the odometry



```
PROGRAM Odometry_test
VAR
    PowerR:MC_Power;
    PowerL:MC_Power;
    i: INT:=0;
    //odometry
    r_last: LREAL;
    l_last: LREAL;
    r_now: LREAL;
    l_now: LREAL;
    delta_r: LREAL;
    delta_l: LREAL;
    delta_s: LREAL;
    delta_th: LREAL;
    th_last: LREAL;
    th_avg: LREAL;
    delta_linear: LREAL;
    //trajectory
    theta:LREAL;
    radius: LREAL;
    W: LREAL; //angular velocity
    VelL: LREAL;
    VelR: LREAL;
    DistR: LREAL;
    DistL: LREAL;
    MoveR:MC_MoveRelative;
    MoveL:MC_MoveRelative;
END_VAR
```



```
Axis_DIR.ReadStatus();
Axis_ESQ.ReadStatus();
IF NOT GVL.ros_ready THEN
    r_last := Axis_DIR.NcToPlc.ActPos;
    l_last := Axis_ESQ.NcToPlc.ActPos;
    th_last := 0.0;
    GVL.x := 0.0;
    GVL.y := 0.0;
    GVL.th := 0.0;
    i := 0;
    GVL.ros_ready := TRUE;
ELSE
    //odometry for position
    r_now := Axis_DIR.NcToPlc.ActPos;
    l_now := Axis_ESQ.NcToPlc.ActPos;
    delta_r := r_now - r_last;
    delta_l := l_now - l_last;
    delta_s := (delta_r + delta_l) / 2;
    delta_th := (delta_r - delta_l) / GVL.B;
    GVL.th := th_last + delta_th;
    th_avg := (th_last + GVL.th) / 2;
    GVL.xx := GVL.x + COS(th_avg) * delta_s;
    GVL.yy := GVL.y + SIN(th_avg) * delta_s;
    //odometry for position without delta_s aproximation
    IF delta_th = 0 THEN
        GVL.x := GVL.x + COS(th_avg) * delta_s;
        GVL.y := GVL.y + SIN(th_avg) * delta_s;
    ELSE
        delta_linear := 2 * (delta_s / delta_th) * SIN(delta_th / 2);
        GVL.x := GVL.x + COS(th_avg) * delta_linear;
        GVL.y := GVL.y + SIN(th_avg) * delta_linear;
    END_IF
    //update old variables
    r_last := r_now;
    l_last := l_now;
    th_last := GVL.th;
END_IF
CASE i OF
    0: //initial state
        PowerR(Axis:=Axis_DIR,enable:=TRUE,Enable_Positive:=TRUE,Enable_Negative:=TRUE);
        PowerL(Axis:=Axis_ESQ,enable:=TRUE,Enable_Positive:=TRUE,Enable_Negative:=TRUE);
        IF startMov THEN
```



```
            i:=5;
        END_IF
    5:
        IF targetY=0 THEN
            theta:=0;
        ELSIF targetX=0 AND targetY>0 THEN
            theta:=3.14/2;
        ELSIF targetX=0 AND targetY<0 THEN
            theta:=-3.14/2;
        ELSE
            theta:=ATAN(targetY/targetX);
        END_IF
        IF targetX = 0 AND targetY = 0 AND NOT (targetTh = 0) THEN
            theta:=targetTh;
            DistR:=(theta*GVL.B)/2;
            DistL:=-DistR;
            VelR:=targetVel;
            VelL:=targetVel;
        ELSIF theta = 0 THEN
            VelL := targetVel;
            VelR := targetVel;
            DistR := targetX;
            DistL := targetX;
        ELSE
            radius:=targetY/(2*SIN(theta)*SIN(theta));
            W:=ABS(targetVel)/radius;
            VelL:=ABS(targetVel)-(GVL.B*W/2);
            VelR:=(2*ABS(targetVel))-VelL;
            DistR:=2*theta*(radius+(GVL.B/2));
            DistL:=2*theta*(radius-(GVL.B/2));
        END_IF
        i := 6;
    6:
        MoveR(Axis:=Axis_DIR,Execute:=TRUE,Distance:=DistR,Velocity:=ABS(VelR));
        MoveL(Axis:=Axis_ESQ,Execute:=TRUE,Distance:=DistL,Velocity:=ABS(VelL));
        IF MoveR.Done AND MoveL.Done THEN
            MoveR(Axis:=Axis_DIR, Execute:=FALSE);
            MoveL(Axis:=Axis_ESQ, Execute:=FALSE);
            startMov := FALSE;
            i:=0;
        END_IF
END_CASE
```

Appendix B: ROS Nodes and Configuration Files

B.1 Robot’s frame and Lidar’s frame transform broadcaster in Python

```python
#!/usr/bin/env python
import rospy
import tf2_ros
import tf2_msgs.msg
import geometry_msgs.msg

class FixedTFBroadcaster:

    def __init__(self):
        #create a ROS publisher to the topic /tf
        self.pub_tf = rospy.Publisher("/tf", tf2_msgs.msg.TFMessage, queue_size=1)

        while not rospy.is_shutdown():
            # Run this loop every X seconds
            rospy.sleep(1)

            t = geometry_msgs.msg.TransformStamped()
            t.header.frame_id = "base_link"
            t.header.stamp = rospy.Time.now()
            t.child_frame_id = "lidar_frame"
            #the Lidar is translated from the robot's frame by 0.0958m in x, 0.0742m in z, and rotated by 180 degrees in z
            t.transform.translation.x = 0.0958
            t.transform.translation.y = 0.0
            t.transform.translation.z = 0.0742

            t.transform.rotation.x = 0.0
            t.transform.rotation.y = 0.0
            t.transform.rotation.z = 1.0
            t.transform.rotation.w = 0.0
            tfm = tf2_msgs.msg.TFMessage([t])
            self.pub_tf.publish(tfm)

if __name__ == '__main__':
    #initiate the node
    rospy.init_node('fixed_tf2_broadcaster')
    tfb = FixedTFBroadcaster()

    rospy.spin()
```

B.2 ROS node to publish odometry information and send velocity commands to TwinCAT, written in Python

```python
#!/usr/bin/env python
import geometry_msgs.msg
import pyads
import rospy
import tf2_ros
# Because of transformations
import tf_conversions
from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3
from nav_msgs.msg import Odometry

def callback(data):
    #get the velocity commands from the /cmd_vel topic
    global cmd_Vth
    global cmd_Vx
    cmd_Vx = data.linear.x * 1000.0
    cmd_Vth = data.angular.z

def handle_robot_pose(x, y, th):
    #publishing the transform
    br = tf2_ros.TransformBroadcaster()
    t = geometry_msgs.msg.TransformStamped()

    t.header.stamp = rospy.Time.now()
    t.header.frame_id = "odom"
    t.child_frame_id = 'base_link'
    t.transform.translation.x = x
    t.transform.translation.y = y
    t.transform.translation.z = 0.0
    q = tf_conversions.transformations.quaternion_from_euler(0.0, 0.0, th)
    t.transform.rotation.x = q[0]
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]
    br.sendTransform(t)

    #publishing the odometry message
    odom = Odometry()
    odom.header.stamp = rospy.Time.now()
    odom.header.frame_id = "odom"
    odom.child_frame_id = "base_link"
    odom.pose.pose = Pose(Point(x, y, 0.0), Quaternion(*q))
    odom.twist.twist = Twist(Vector3(vx, vy, 0.0), Vector3(0.0, 0.0, vth))
    odom_pub.publish(odom)

def listener():
    #subscribe to the /cmd_vel topic and run callback
    rospy.Subscriber("cmd_vel", Twist, callback, queue_size=1)

if __name__ == '__main__':
    #initiate node with a frequency of X Hertz
    rospy.init_node('robot_broadcaster_and_mover')
    r = rospy.Rate(20.0)
    odom_pub = rospy.Publisher('/odom', Odometry, queue_size=10)
    #connect to TwinCat
    try:
        connection
    except NameError:
        connection = False
    while not rospy.is_shutdown() and not connection:
        try:
            plc = pyads.Connection('172.18.235.247.1.1', 851, '169.254.25.21')
            plc.open()
            plc.write_by_name('GVL.ros_start', True, pyads.PLCTYPE_BOOL)
        except:
            r.sleep()
        else:
            connection = True
            break
    while not rospy.is_shutdown() and connection:
        plc.write_by_name('GVL.ros_start', True, pyads.PLCTYPE_BOOL)
        try:
            cmd_Vx
            cmd_Vth
        except NameError:
            cmd_Vx=0.0
            cmd_Vth=0.0
        #subscribe to the /cmd_vel topic and run callback
        listener()
        #send safety variable to trigger a rising edge on the twincat, if twincat doesn't receive for X time, all velocities will be set to 0
        plc.write_by_name('GVL.ros_coms', True, pyads.PLCTYPE_BOOL)
        #write velocity commands to TwinCat
        plc.write_by_name('GVL.Vx', cmd_Vx, pyads.PLCTYPE_LREAL)
        plc.write_by_name('GVL.Vth', cmd_Vth, pyads.PLCTYPE_LREAL)
        #read odometry info from TwinCat
        x = plc.read_by_name('GVL.x', pyads.PLCTYPE_LREAL) / 1000.0
        y = plc.read_by_name('GVL.y', pyads.PLCTYPE_LREAL) / 1000.0
        th = plc.read_by_name('GVL.th', pyads.PLCTYPE_LREAL)
        vx = plc.read_by_name('GVL.Vxx', pyads.PLCTYPE_LREAL) / 1000.0
        vy = plc.read_by_name('GVL.Vyy', pyads.PLCTYPE_LREAL) / 1000.0
        vth = plc.read_by_name('GVL.Vth', pyads.PLCTYPE_LREAL)
        #publish TF and odom messages
        handle_robot_pose(x, y, th)
        plc.write_by_name('GVL.ros_coms', False, pyads.PLCTYPE_BOOL)
        r.sleep()
```

B.3 Program to teleoperate the robot written in Python

```python
import pyads
import keyboard
import time

#velocities to be used in m/s and rad/s
Vel_x = 0.15
Vel_th = 0.15
#initialize communication with twincat via pyads
plc = pyads.Connection('172.18.235.247.1.1', 851, '169.254.25.21')
while True:
    try:
        plc.open()
        #it uses the same POU as ros to allow for a map to be recorded in ROS
using the odometry
        plc.write_by_name('GVL.ros_start', True, pyads.PLCTYPE_BOOL)
    except:
        print("connection failed")
    else:
        break
#check for keyboard inputs and send velocity commands to twincat
while True:
    plc.write_by_name('GVL.ros_coms', True, pyads.PLCTYPE_BOOL)
    if keyboard.is_pressed('esc'):
        print("program ended.")
        break
    if (keyboard.is_pressed('w') or keyboard.is_pressed('up arrow')) and
(keyboard.is_pressed('s') or keyboard.is_pressed('down arrow')):
        Vx = 0.0
    elif keyboard.is_pressed('w') or keyboard.is_pressed('up arrow'):
        Vx = Vel_x * 1000
    elif keyboard.is_pressed('s') or keyboard.is_pressed('down arrow'):
        Vx = - Vel_x * 1000
    else:
        Vx = 0.0
    if (keyboard.is_pressed('a') or keyboard.is_pressed('left arrow')) and
(keyboard.is_pressed('d') or keyboard.is_pressed('right arrow')):
        Vth = 0.0
    elif keyboard.is_pressed('a') or keyboard.is_pressed('left arrow'):
        Vth = Vel_th
    elif keyboard.is_pressed('d') or keyboard.is_pressed('right arrow'):
        Vth = - Vel_th
    else:
        Vth = 0.0
    plc.write_by_name('GVL.Vx', Vx, pyads.PLCTYPE_LREAL)
    plc.write_by_name('GVL.Vth', Vth, pyads.PLCTYPE_LREAL)
    plc.write_by_name('GVL.ros_coms', False, pyads.PLCTYPE_BOOL)
    time.sleep(0.020) #wait 20 miliseconds
```

B.4 Launch file for the navigation stack pre-requisites

```
<launch>
  <node name="LD06" pkg="ldlidar" type="ldlidar" output="screen" >
  <param name="topic_name" value="LiDAR/LD06"/>
  <param name="port_name" value ="/dev/ttyAMA0"/>
  <param name="frame_id" value="lidar_frame"/>
  </node>
  <node pkg="fernandinho_tf2" type="robot_broadcaster_mover.py"
          name="fernandinho_broadcaster_and_mover" output="screen"/>
  <node pkg="fernandinho_tf2" type="laser_frame_tf2_broadcaster.py"
          name="laser_frame_broadcaster" output="screen"/>
</launch>
```

B.5 Costmap configuration files

B.5.1: Common costmap configuration

```
footprint: [[0.123, 0.32], [0.123, 0.1325], [0.1835, 0.1325], [0.1835, -0.1325], [0.123, -0.1325], [0.123, -0.32], [0.0, -0.32], [0.0, -0.3675], [-0.662, -0.3675], [-0.662, -0.115], [-0.752, -0.115], [-0.752, 0.115], [-0.662, 0.115], [-0.662, 0.32]]

inflation_radius: 0.55
cost_scaling_factor: 10.0
track_unknown_space: true
footprint_clearing_enabled: true

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: lidar_frame, data_type: LaserScan, topic: /LiDAR/LD06, marking: true, clearing: true, obstacle_range: 4.0, raytrace_range: 4.0}
```

B.5.2: Global costmap configuration

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 1.0
  static_map: true
  rolling_window: true
  width: 8.0
  height: 8.0
  resolution: 0.05
```

B.5.3: Local costmap configuration

```
local_costmap:
  global_frame: odom
```

```
robot_base_frame: base_link
update_frequency: 10.0
publish_frequency: 2.0
static_map: false
rolling_window: true
width: 4.0
height: 4.0
resolution: 0.05
```

B.6 Planner configuration files

B.6.1 Global planner configuration file

```
neutral_cost: 55.0
cost_factor: 0.8
orientation_mode: 0
```

B.6.2 Local planner configuration file

```
DWAPlannerROS:
  acc_lim_x: 0.3
  acc_lim_y: 0.0
  acc_lim_th: 0.7
  max_vel_trans: 0.3
  min_vel_trans: 0.01
  max_vel_x: 0.3
  min_vel_x: -0.3
  max_vel_y: 0.0
  min_vel_y: 0.0
  max_vel_theta: 0.5
  min_vel_theta: 0.01
  holonomic_robot: false

  yaw_goal_tolerance: 0.05
  xy_goal_tolerance: 0.05
  latch_xy_goal_tolerance: false

  sim_time: 4.0
  vx_samples: 20.0
  vth_samples: 40.0
  controller_frequency: 5.0
  sim_granularity: 0.075
```

B.7 AMCL launch file

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <remap from="scan" to="LiDAR/LD06"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="base_frame_id" value="base_link"/>
  <param name="global_frame_id" value="map"/>
  <param name="tf_broadcast" value="true"/>
  <param name="odom_model_type" value="diff-corrected"/>
  <param name="odom_alpha1" value="0.005"/>
  <param name="odom_alpha2" value="0.005"/>
  <param name="odom_alpha3" value="0.010"/>
  <param name="odom_alpha4" value="0.005"/>
  <param name="transform_tolerance" value="1.0"/>
  <param name="gui_publish_rate" value="2.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="100"/>
  <param name="max_particles" value="5000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <param name="laser_z_hit" value="0.95"/>
  <param name="laser_z_rand" value="0.05"/>
  <param name="laser_sigma_hit" value="0.045"/>
  <param name="laser_likelihood_max_dist" value="2.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="resample_interval" value="2"/>
  <param name="recovery_alpha_slow" value="0.0"/>
  <param name="recovery_alpha_fast" value="0.0"/>
</node>
</launch>
```

B.8 ROS node to read goals and initial positions from QR codes and publish them

```
#!/usr/bin/env python3

import rospy
import cv2
import tf2_ros
import tf2_geometry_msgs.tf2_geometry_msgs
from geometry_msgs.msg import PoseStamped, PoseWithCovarianceStamped
from tf.transformations import quaternion_from_euler
import time
import numpy
from math import pi

def calculate_position(points):
    global orientation_robot
    global real_offset
    robot_center = (320, 258)
    meters_per_pixel = 0.0008125
    sumx = 0
    sumy = 0
    points = points[0]
    for i in range(len(points)):
        # calculate center
        sumx = sumx + int(points[i][0])
        sumy = sumy + int(points[i][1])
        if i == 1:
            frontpoint_x = sumx/2
            frontpoint_y = sumy/2
    center = (int(sumx/4), int(sumy/4))
    # offset from center
    offset_center = (robot_center[0] - center[0], robot_center[1] - center[1])
    real_offset = (-offset_center[1] * meters_per_pixel, -offset_center[0] * meters_per_pixel)
    # calculate orientation
    if ((sumy/4) - frontpoint_y) > 0:
        orientation = numpy.arctan( ( (sumx/4) - frontpoint_x ) / ( (sumy/4) -
frontpoint_y ) )
    elif ((sumy/4) - frontpoint_y) < 0:
        orientation = numpy.arctan( ( (sumx/4) - frontpoint_x ) / ( (sumy/4) -
frontpoint_y ) ) + pi
    elif ((sumx/4) - frontpoint_x) < 0:
        orientation = pi/2
    else:
        orientation = -pi/2
    if orientation > pi:
        orientation = orientation - 2.0 * pi
    orientation_robot = - orientation
    print('The robot is distanced from the QR code in meters by:', real_offset)
    print('And at an angle of: ', orientation_robot, " in radians")
def movebase_client(goal_pos, goal_frame, goal_pub, tfBuffer):

    #Transform from original frame to odom frame
    base_goal=PoseStamped()
    base_goal.header.frame_id = goal_frame
    base_goal.header.stamp =rospy.Time(0)
    base_goal.pose.position.x=goal_pos[0]
    base_goal.pose.position.y=goal_pos[1]
    base_goal.pose.position.z=0.0
    q = quaternion_from_euler(0.0, 0.0, goal_pos[2])
    base_goal.pose.orientation.x = q[0]
    base_goal.pose.orientation.y = q[1]
    base_goal.pose.orientation.z = q[2]
    base_goal.pose.orientation.w = q[3]
    if goal_frame == "base_link":
        trans = tfBuffer.lookup_transform(goal_frame, 'odom', rospy.Time(0))
        base_goal=tf2_geometry_msgs.do_transform_pose(base_goal, trans)
        base_goal.header.frame_id = 'odom'
        base_goal.header.stamp = rospy.Time.now()
    #publish the goal
    goal_pub.publish(base_goal)
    print('goal given: ' , base_goal)

def initial_pos_client(initial_pos, pos_pub, tfBuffer):
    initial_pose = PoseWithCovarianceStamped()
    initial_pose.header.frame_id = 'map'
    initial_pose.header.stamp =rospy.Time(0)
    initial_pose.pose.pose.position.x = initial_pos[0]
    initial_pose.pose.pose.position.y = initial_pos[1]
    initial_pose.pose.pose.position.z = 0.0
    q = quaternion_from_euler(0.0, 0.0, initial_pos[2])
    initial_pose.pose.pose.orientation.x = q[0]
    initial_pose.pose.pose.orientation.y = q[1]
    initial_pose.pose.pose.orientation.z = q[2]
    initial_pose.pose.pose.orientation.w = q[3]
    pos_pub.publish(initial_pose)
    print('initial pose given: ' , initial_pose)

# If the python node is executed as main process
if __name__ == '__main__':
    rospy.init_node('qr_reader_py')
    goal_pub = rospy.Publisher('/move_base_simple/goal', PoseStamped,
queue_size=3)
    pos_pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped,
queue_size=1)
    tfBuffer = tf2_ros.Buffer()
    goal_pos = [0.0, 0.0, 0.0]
    initial_pos = [0.0, 0.0, 0.0]
    cap = cv2.VideoCapture(0)
    detector = cv2.QRCodeDetector()
    try:
        last_data
    except NameError:
        last_data = ''
        print("Program started.")

    while True:
        success, frame = cap.read()
        if not success:
            print("Couldn't read camera.")
            time.sleep(0.1)
            break

        try:
            data, points, _ = detector.detectAndDecode(frame)

        except:
            print("ERROR DETECTED IN detectAndDecode()")

        else:
            # check if there is a QR code
            if data != '':
                # check if it is a new QR code
                if data != last_data:
                    last_data = data
                    data_list = data.split()
                    type_qr = data_list[0]
                    print('The QR code says:', data)
                    if type_qr == "goal":
                        for i in range(1,4):
                            goal_pos[i-1] = float(data_list[i])
                        goal_frame = data_list[4]
                      movebase_client(goal_pos, goal_frame, goal_pub, tfBuffer)
                    elif type_qr == "initial_pos":
                        for i in range(1,4):
                            initial_pos[i-1] = float(data_list[i])
                        calculate_position(points)
                              initial_pos  =  [initial_pos[0]  +  real_offset[0],
initial_pos[1] + real_offset[1], initial_pos[2] + orientation_robot]
                        initial_pos_client(initial_pos, pos_pub, tfBuffer)
        time.sleep(0.1)
    rospy.spin()
    cap.release()
```