Politecnico di Torino

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

In collaboration with California State University Los Angeles

System Integration, Localization and Control Implementation for an Autonomous Mobile Service Robot

Supervisors

Prof. Marcello CHIABERGE

Prof. Marina MONDIN

Candidate

Lorenzo SEGHESIO

December 2021

Abstract

Autonomous mobile robots are a technology extensively used in many different industrial and home applications. The ability to move towards a desired location without human intervention is one of their fundamental features as well as the final goal of this thesis work. The project is part of a larger one carried out by InnoTech System L.L.C. together with the California State University, Los Angeles. The development of this service robot has different purposes such as escorting people in airports and hospitals as well as food delivery in university campuses. After a theoretical introduction on the autonomous mobile robots and the background material, the document focuses on the main topics of the work: the system integration between different robot components (using I2C and UART communications), the LIDAR localization using Scan Matching and AMCL algorithms and the path planning using the A* search algorithm, implemented with a path tracking and a control logic in order to reach a goal position on a mapped environment. The thesis concludes with the description of the robot hardware structure, of the test phase and of the obtained results.

Summary

The thesis work regards the implementation of the autonomous motion on a service mobile robot. The project carried out at the California State University Los Angeles in collaboration with InnoTech Systems has the goal of building a fully autonomous mobile machine for different applications with the general aim of helping people in professional and everyday tasks. The robot is designed for assistive purposes like escorting people in environments such as hospitals and airports, medicines and food delivery for healthcare and on university campuses, assisting visitors' or helping schoolchildren cross the street once the school bus has driven them to their destination.
The thesis focuses on three main aspects of the robot development: system integration, LiDAR mapping and localization, and the final path planning and tracking algorithms implemented through a MicroController Unit (MCU) control logic. The work described in this thesis is the continuation of a previous one done by different students and professionals that built the robotic platform and started implementing the robot localization and the path planning algorithms. More precisely the robot's autonomous reaching of a desired location in a mapped environment is the final contribution of this thesis to the project.

Autonomous Mobile Robots, AMRs, are machines that move unbounded and without human intervention throughout known or unknown environments. These systems are able to detect their surroundings through sensors, to extract information from the obtained data such as the map in which they are navigating, their localization, and the presence of obstacles. Then, once they receive a target location, they are able to plan a path and execute a motion till the final destination has been reached. Furthermore, these robots can have many extra features that make them suitable for a wide range of different applications. In the thesis, the essential navigation functionality of the machine is studied, implemented, and tested, since the first step to make an autonomous robot useful and usable is obviously to make it move by its own.

The first part of the work regards the system integration between different robot

components, in particular the communication between the onboard computer, the Nvidia Jetson Nano, and the MCU, the Texas Instruments C2000 Piccolo MCU LAUNCHXL-F28069M. The computer operates all the main logic of the robot, metaphorically it is its brain. The sensor data collection and elaboration, the map building and localization, the path planning and path tracking are all implemented thanks to the Jetson Nano codes. The ROS framework has been used, coding all the source files in C++ and a minority in Python, on the Ubuntu software of the device. The MCU, instead, is used for the motion implementation of the robot, the one-by-one path points reaching. Its role can be compared to the one of a cerebellum for a human. Thanks to a negative feedback, it is able to control the motion and make the robot behave as precisely as possible and desired. Its logic is developed as a Matlab Simulink model that thanks to the Code Generation of the used program was transformed into a C/C++ code, deployed inside the MCU.
The communication between these two system components is clearly essential. The MCU actuates the motors for making the robot move according to a proportional logic to the feedback error between the actual robot position and the desired one (the path point to be reached). The MCU needs to receive from the Jetson Nano the computed robot position in the map and the coordinates of the desired goal point. The other important signals sent from the computer to the MCU are the enable signals that activate the MCU control logic only once the path points and the localization are correctly and reliably computed and sent and the keyboard commands for the keyboard motion of the robot. The transmission of this information is done using the Inter Integrated Circuit (I2C). This serial, synchronous communication method has been chosen since, thanks to its clock signal, it allows synchronization between master and target that leads to a more precise message comprehension. The I2C message structure helps increase the complexity of the transmitted information (a byte is used in order to identify the kind of message sent and the other ones for the main information, which could also be an unstructured float number). The CAN communication is not used since it is not supported by the Jetson Nano device. The message creation is coded in the master device (the computer) while the unpacking and data reconstruction are implemented on the target device (the MCU).
A second communication channel is used in order to check the goodness of the first one and in general of the MCU control logic. This is done because the live simulation on Simulink is too slow for a good debug comprehension of the inner behavior of the device. This additional communication is the Serial Communication Interface, SCI, protocol (also known as Universal Asynchronous Receiver-Transmitter, UART, communication). This serial and asynchronous communication method allows us to retrieve the inner signals of the MCU checking and debugging its functionality. Thanks to the second communication channel a series of wrong data reception was noticed in the transmission of the different information, probably due to the high

traffic of transmitted messages. As a consequence, for the localization coordinates, an outlier rejection logic has been implemented in the MCU in order to reject the wrong receptions that could prevent the robot from a good motion. While for the other messages sent through I2C a feedback control of the reception has been developed in order to make the computer publish such messages till they are not correctly received.

The second part of the work focuses on the environment mapping and robot localization using a LiDAR sensor (Laser Imaging Detection And Ranging), the Slamtech Rplidar A3. The mapping phase is done by exploiting the Hector Slam package. Thanks to a Simultaneous Localization and Mapping, SLAM, logic the robot is able to obtain a good map of the environment, saving it in a PGM image file and a YAML file containing the map information.
After this mapping phase, an Adaptive Monte Carlo Localization (AMCL), is used for localizing the robot in the mapped environment in which the robot is navigating. This algorithm needs to be used together with odometry localization of the robot, since the AMCL is a discrete localization method and so it is used as a powerful correction to the continuous localization methods (such as the odometry), that usually became less and less accurate as the total movement increases, due to integrative errors. In our case, instead of the odometry, we decided to use a scan matching logic as a continuous source of robot localization, that can be shown to lead to better localization performances (reducing the integrative errors).

The last part of the project is about the path planning implementation over the pre-obtained map and the subsequent path tracking point by point. After the decision of the objective position in the known environment, the path is computed using the A* search algorithm. In order to avoid unwanted obstacles and wall crushes, before the path logic is launched, the map undergoes a process of obstacle boundary increase. Once the path is obtained a specific tracking node has the role of deciding the closest path point to be reached by the robot according to its actual position in the map, publishing this point over the I2C channel for the MCU actuation of the motion. The various goal point coordinates published on the communication channel are formerly expressed in a useful reference frame, aligning the path with the map and the robot localization.
As previously reported the final motion actuation is a MCU prerogative. This allows the robot to reach the single path point given by the tracking node. Once this localization is reached the goal position is updated with the following path node. This keeps it going till the final localization is reached by the robot.
The motion control done by the MCU is based on a negative feedback in which the robot linear and rotational speeds, and so the motor speeds control (done by tuning the Duty Cycle of the motor control square wave), are proportional to the

error between the actual robot localization and the desired position, weighted on suitably tuned gains.

In conclusion, the different designed features' functioning has been checked. The communication between the two devices is reliable enough thanks to the developed outlier rejection and feedback transmission check technologies leading to a low number of incorrect data receptions. The system is also able to correctly map and localize itself in the environment with an average accuracy of 10 cm during the motion. Even the path planning logic correctly computes a good obstacle-free path. Furthermore, the MCU control logic is proven to be able to lead the machine to the desired location in simulation.
In the end, a final test has been run, checking the overall robot behavior during the autonomous motion till the desired location. Unfortunately, the control logic had to be moved from the MCU to the on-board computer, due to the microcontroller's low computational speed. Anyway, the used control logic is the same, so its nice functioning attests to the goodness of its logic. From the test, it has been evinced that the robot is able to autonomously move following the path and reaching the final destination with a good proximity to the path points. This path closeness can be tuned by changing the path point reaching parameter Threshold (T), which defines when a single path point is considered reached. Varying this T parameter also changes the qualitative goodness of the robot motion (expressed as smoothness and rapidity of the navigation). A lower T leads to a better following of the ideal path, but a slower and less smooth and harmonious motion, while a higher T gives a more approximate path tracking but also a faster reach of the final destination with smoother navigation at a constant speed. So this T parameter can be tuned in order to obtain a trade-off between the closeness of the obstacle-free path tracking and the qualitative goodness of the motion.

To summarize, the thesis' aim was to make a robot able to autonomously move to a required location. This goal was achieved. The work remains open to future improvements, such as the design of a more robust control logic, a local path planning avoidance of unexpected obstacles, and the addition of more sensors (integrated through a sensor fusion algorithm) for improving the robot localization performance.

Table of Contents

List of Tables

List of Figures

Acronyms

AMCL Adaptive Monte Carlo Localization.

AMR Autonomous Mobile Robot.

CAN Controller Area Network.

DC Duty Cycle.

FIFO First-In-First-Out.

GNSS Global Navigation Satellite System.

I2C Inter Integrated Circuit.

IC Integrated Circuit.

IMU Inertial Measurement Unit.

KLD Kullback-Leibler Distance.

LiDAR Light Detection and Ranging.

LRB Localization Ready Bit.

MCL Monte Carlo Localization.

MCU MicroController Unit.

PGM Portable Gray Map.

PRB Path Ready Bit.

PWM Pulse Width Modulation.

RCU Robot Control Unit.

RMSE Root Mean Square Error.

ROS Robot Operating System.

RX Receiver.

SCI Serial Communication Interface.

SCL Serial Clock line.

SDA Serial Data line.

SLAM Simultaneous Localization and Mapping.

SPI Serial Peripheral Interface.

T Threshold.

TX Transmitter.

UART Universal Asynchronous Receiver - Transmitter.

Chapter 1

Introduction

The following thesis deals with the design, physical development and testing of different parts of an Autonomous Mobile Robot (AMR). The robot was already assembled and its inner logic partially coded by previous years' students and professionals that worked for Innotech Systems LLC, the company for which the project has been developed. My part of the work was mainly focused on the System Integration, LiDAR mapping with robot localization and Path Planning together with control logic for reaching a desired position in a known environment. These different parts lead to final tests in order to check the accuracy of the autonomous motion of the robot.

1.1 Thesis motivations and context

AMRs represent a class of machines that are able to achieve complex tasks moving autonomously from human intervention. They perceive their surroundings and are able to make decisions and interact with the environment according to the received stimulus and the goal for which they move [1].

Nowadays, the utility of AMRs has become crucial for many different applications. From home cleaning and management to industrial material movement and organization [2]. They are used in different contexts assisting human workers helping them with accessory tasks, making their jobs easier. Another important application is substituting humans in dangerous environments or for dangerous tasks, thus reducing the risks for some specific jobs [3].

The basic features of the AMRs can be reduced to four main fields:

- Perception - the ability to correctly interpret the sensors' data. It makes the robot able to extract information from the surroundings.
- Localization and cognition - they consist in the interpreted data computation in order to obtain a correct robot localization and the consequent robot logic deciding the actions to take to complete a required task.
- Locomotion - the ability of the robot to move itself. It relies both on the mechanical features of the machine, but also on other technical criteria such as maneuverability, controllability, terrain condition, efficiency and stability.
- Navigation - the ability of the robot to move unbounded throughout a real-world environment reaching a specific location and task. This field implies all the previous ones for its development.

InnoTech Systems LLC is a company founded in 2018 to provide solutions and services for a vast array of problems using AI-based systems. Since their foundation, they have cooperated with the California State University Los Angeles on a wide number of different projects. This gave the possibility to students like me, to take part in an actual company reality and work on concrete projects, applying, exploiting, and enlarging the knowledge acquired in the previous years of studies. The different fields in which InnoTech Systems operates are all linked by the utilization of robotics and AI solutions for improving the quality of the results. In particular, their work finds applications in contexts like airports and hospitals for which utility devices for customers are developed. Also, traffic management systems and retail stores' visitors flow analysis are covered by a specific company branch dealing with Artificial Intelligence.

One of the main works on which they are focusing their research is the realization of an autonomous motion robotic platform. Its development has different and heterogeneous applications in more than one field:

- The robot is projected in order to be a good service robot for Hospitals. In such a context its key role would be to escort and assist people inside of the building helping them find the right department. It could also deliver medicines, take care of the patient greetings and assist nurses in their different tasks. It could also be equipped with some diagnostic equipment taking them where they are needed.
- The same robot could have applications in airports, for luggage carriage for passengers and staff, for food delivery during the take-off waiting, for

escorting travelers till the required terminal and for providing security services monitoring some specific areas.

- Another useful application is in the university campuses where the robot can deliver food and goods to Professors and Students in order to ease their studies.

- Finally, a recent task for which the robot could be useful is helping schoolboys crossing the street once the school bus has drove them to their destination, making this action much safer for the children.

The built AMR can be seen in Figure 1.1. As can be seen the robot is constructed in order to have a touch display on its front part, used for interacting with humans in order to help them or receive tasks. Then there's also a basket closed by an automated door for depositing or taking objects from the robot, so that it can deliver them where they are needed. In the lower part the entire robot logic is implemented. The on-board computer, the MicroController Unit (MCU), the battery, and all the sensors are placed there.

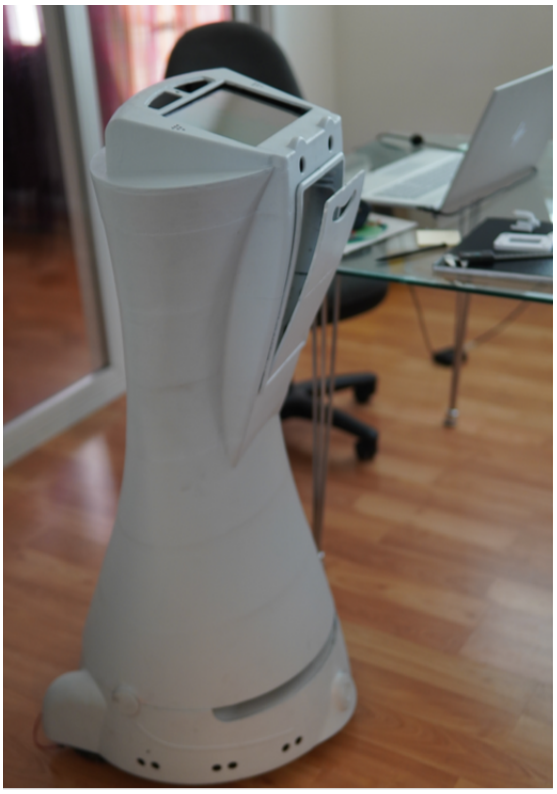


Figure 1.1: InnoTech robot picture

The work, exposed in this thesis, consisted in developing a portion of this Autonomous Service Robot and testing its behavior as discussed in the following sections.

1.2 Objective and methodologies

My thesis work is the continuation of a bigger project started by other Politecnico di Torino students, researchers and professionals for InnoTech Systems collaborating with California State University, Los Angeles. The main project goal is to create an Autonomous Service Motion Robot from scratch. The robot will have different functionalities according to its different applications, as exposed in the previous section. Its essential feature is to be able to autonomously navigate in a known or unknown environment, reaching a desired location. This is possible only if the robot has good sensors for identifying the surroundings and localizing itself on a pre-obtained map, or, if it is moving in an unknown environment, it is able to map it and avoid obstacles, localizing itself.
Once this main feature is implemented other additional ones can be added to the machine in order to make it able to accomplish a wider number of tasks, even more complex ones.
My objective was to add to the robot this fundamental feature, making it able to reach a desired location inside a mapped environment. This was implemented throughout different work parts: a system integration between the MicroController Unit (MCU), the Texas Instruments C2000 Piccolo MCU in the LAUNCHXL-F28069M development kit, also addressed as Robot Control Unit (RCU) and the on-board computer, the Jetson Nano from Nvidia. The LiDAR mapping of the environment and the consequent Adaptive Monte Carlo Localization (AMCL) of the machine. The path planning computation using the A* search algorithm, the path tracking and the MCU control logic development.

The entire work was directly implemented on the InnoTech robotic platform. Every part of the work coded on the Jetson Nano was written in the C++ code and some files in Python, the entire logic is developed under the Robot Operating System (ROS) environment (which is a set of software libraries and tools that help the designer build robot applications [4]). As regards the MCU logic design, it was first build as a model on Matlab Simulink from Mathworks, then transformed in code and deployed to the device thanks to the Code Composer Studio from Texas Instruments. During the test phase also other programs were used for elaborate the obtained data (Excel from Microsoft and Matlab from Mathworks) and for remotely connect and control the on-board computer during the motion (NoMachine software from NoMachine)

The method used for the project development consisted in first subdividing the work in smaller and easier sub-parts, like the first division in the three main exposed parts. Then the different problems were analyzed one-by-one, studying the necessary theory and thinking of a suitable solution/implementation. After this, a design part was implemented, together with a simulation of the proposed development if needed. Then, if the solution seemed correct, a physical implementation of the studied technology was done on the robot hardware and tested directly on the final machine.
These general methodology lines helped in organizing in an efficient way the job and obtaining good results on many different tasks. Once all the different sub-requirements were faced, a final test of the robot working was done and a series of measurements were collected in order to check the goodness of the overall obtained system.

1.3 Original contribution

This thesis is not a theoretical research work, even if obviously the used theory has been studied and analyzed. It is more a design development of different technologies on a functioning robotic machine for a company. As a consequence, the aim was more on applying consolidated technologies and theories to a physical device in order to make it behave as well as possible for real-world applications. For these reasons the personal original contribution that I've produced for this thesis is more based on improvements and bug-fixing strategies of the used technologies together with some integrative developments of the robot logic that contributed to the overall reaching of the main project goal. These elements are:

- The feedback transmission control for the I2C messages sent to the MCU. It is implemented thanks to the UART communication. The computer is able to recognize if the MCU has correctly received the I2C messages, if not it keeps publishing them till the correct value is present on the receiving device.
- The outlier rejection logic for the MCU localization coordinate messages. For the localization coordinates that are sent with a high frequency, the previous control logic is not suitable. A different one was implemented, based on the rejection of the coordinates too different from the previous received ones for being physically admissible. The logic starts with a tuning phase of the localization transmission.

- The use of the scan matching technology as virtual odometry for the robot localization, improved by the AMCL one. The Hector Slam package scan matching algorithm was adapted and used for the purpose.
- The reference system alignment, implemented to express all the different used coordinates in the same frame. Starting from the Map information contained in its YAML file the path is transformed from pixels to meters and expressed in the correct map frame, the same of the localization coordinates.
- The map processing through boundary increase. In order to avoid unwanted obstacles impact.
- Finally the overall control logic implementation for make the robot moves. Including the integration between path planning, path tracking and microcontroller control logic.

1.4 Thesis outline

After this first introduction to the AMRs, their main features, and applications, together with the InnoTech Systems' robot presentation. I will start introducing the fundamental concepts for my work in a background material chapter. The main topics will be the two communications used for the system integration between the robot onboard computer, the Nvidia Jetson Nano and the robot microprocessor. These two communication methods are the I2C and the UART ones.
After this, the main work of the project is exposed in three different chapters. The first one deal with the system integration between on-board computer in which the robot intelligence is set (the localization is computed as well as the path planning and its tracking) and the robot microprocessor whose goal is to implement the local motion till the next path goal point with the corrects motor control.
The second part of the project is focused on the LiDAR mapping of a closed environment and the localization of the robot in such a location. In particular the AMCL and the scan matching logics were exploited for this goal.
The third topic shown in the document regards the path planning using the A* algorithm search to reach a desired location in the obtained map and the robot motion execution thanks to the MCU control logic.
After these parts, a chapter is dedicated to the robot hardware presentation and to the tests that have been run to check the functioning of the machine. Then the results are exposed.
Finally, a conclusion chapter resumes the main key points and results of the work stating also which will be the future developments of the project.

Chapter 2

Background Material

In this chapter a theoretical presentation of the main thesis topics is exposed. Only starting from a consolidated knowledge of the different arguments the design and development of the various robotic systems was possible. The first section shows the different communications protocols used in the project for the system integration part. Then focusing on the robot mapping and localization the SLAM problem is presented and the Hector SLAM is presented as a possible solution adopted in the project. Then for the robot localization on a pre-obtained map, the AMCL has been used and so a chapter is devoted to its theoretical working principles. At the chapter end regarding the path planning a section is devoted to its development using the A* algorithm search.

2.1 I2C and UART communications

Communication protocols play an essential role in the organization of the data exchange between electronic devices. I2C and UART communication protocols are two serial computer buses used for data transmission. Both technologies have a quite long history from their invention. During this time of development and different applications their functionality and reliability have been consolidated, making them as good choices for Integrated Circuit (IC)s communication, as it is in our study.

2.1.1 I2C communication protocol

I2C, read as I-two-C or I-squared-C, acronym for Inter-IC, is a multi-master, multi-target, single-ended serial computer bus [5]. This protocol was invented by Philips

Semiconductors (today NPX Semiconductors) in the 1982.
In the 2021 this communication protocol was implemented in over 1000 different ICs manufactured by more than 50 companies. This makes the I2C-bus being considered a world standard. Additionally, the versatile I2C-bus is used in various control architectures such as System Management Bus (SMBus), Power Management Bus (PMBus), Intelligent Platform Management Interface (IPMI), Display Data Channel (DDC) and Advanced Telecom Computing Architecture (ATCA) [6].

Since it's creation the protocol has greatly developed during the years. The original specification that it was made for was only 100 kHz communications in which only 7-bit addresses could be used. Thus, limiting the number of devices connected on the bus to 112. Then, in 1992, the first public specification was published, communication speed was increased to 400 kHz fast-mode offering 10-bit addresses. Later three additional modes were specified; fast-mode plus at 1MHz, high-speed mode at 3.4MHz and ultra-fast mode at 5MHz [7].
Most modern microcontroller (MCU) support I2C communication at 400 kHz. The I2C bus was designed to ease both systems designers and equipment manufacturers but also to maximize hardware efficiency and circuit simplicity. All I2C-compatible devices incorporate an on-chip interface that allows them to communicate directly with each other via a simple two-wire bus [5]. Figure 2.1, illustrates a typical I2C Bus Connection.

I2C communication uses two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors, typical voltages used are +5 V or +3.3 V. [6]
The clock signal is used to define the bits on the data line. The SDA data bits are defined during the high state of the clock, while they can change only during the low SCL state.
Every device connected to the bus is identified by a unique address and can operate either as transmitter or receiver depending if they send or receive the transmitted data. A second distinction of the bus connected devices is between controllers or targets (in the past known as master and slave). The former initiate the data transfer and generates the clock signal, while the second ones send or receive the data when addressed according to the controllers requirements. It has to be noted that these two distinctions are not permanent but related to the single data transfer.

I2C is a multi-controller bus, this means that more than one controller can drive the transmission. In order to avoid conflicts in the lines control the arbitration and clock synchronization are implemented. The procedure relies on a AND-wired connection of the devices on the bus. As regards the SDA line thanks to the arbitration the first controller to produce a logic one when the other produces a

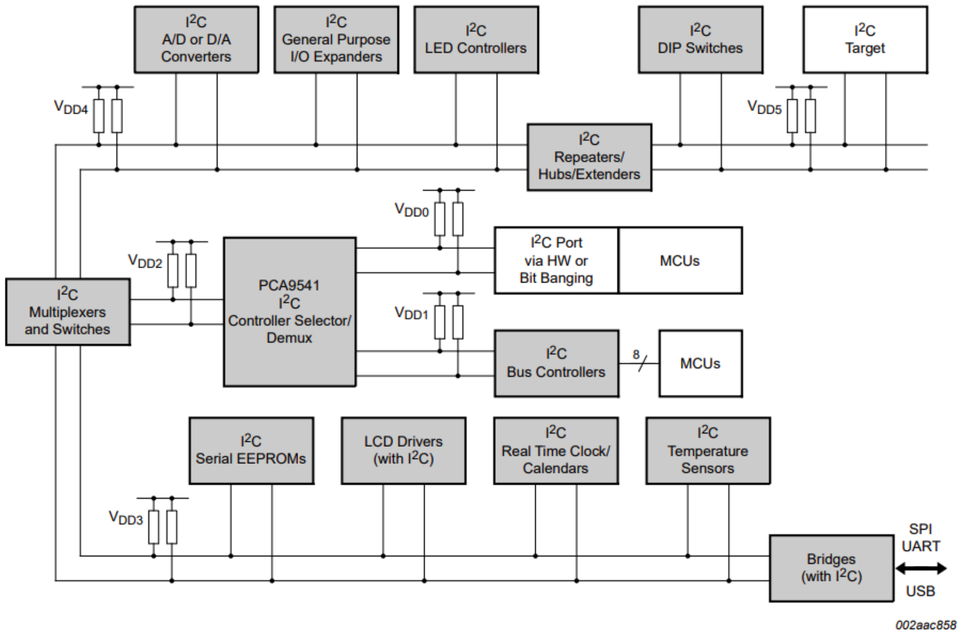

I2C A/D or D/A Converters
I2C General Purpose I/O Expanders
I2C LED Controllers
I2C DIP Switches
I2C Target
VDD4
VDD5
I2C Repeaters/ Hubs/Extenders
VDD0
I2C Port via HW or Bit Banging
MCUs
VDD2
PCA9541 I2C Controller Selector/ Demux
I2C Multiplexers and Switches
VDD1
I2C Bus Controllers
8
MCUs
I2C Serial EEPROMs
LCD Drivers (with I2C)
I2C Real Time Clock/ Calendars
I2C Temperature Sensors
VDD3
Bridges (with I2C)
SPI
UART
USB
002aac858

Figure 2.1: Example of I2C application

zero losses the arbitration. While for the SCL line the clock is given by the logic 'and' between all the controllers clocks that are trying to send data, this is the clock synchronization.

Another important optional feature of the I2C bus is the clock stretching for which the target can hold down the clock line making the controller adapts to its slower speed. The target can slow down the entire communication speed holding down the line every single clock beat or just after a whole byte reception.

The data transfer proceed accordingly to the following steps:

1. If microcontroller A wants to send information to microcontroller B:
   - A (controller) starts the communication and addresses B (target)
   - A (controller-transmitter) sends data to B (target-receiver)
   - A terminates the transfer.
2. If microcontroller A wants to receive information from microcontroller B:
   - A (controller) starts the communication and addresses B (target)
   - A (controller-receiver) receives data from B (target-transmitter)

- A terminates the transfer.

In both cases, the controller (A) generates the timing and terminates the transfer.

The data message is built as follows (see Figure 2.2):

- A starting condition. That is a pull down of the SDA line while the SCL is high.
- The 7 (or 10) bits address that identifies the target to which the controller want to communicate.
- A Read or Write (R/W) bit that sets if the controller want to send or receive information form the transmitter. It is followed by an Acknowledge/Not-Acknowledge (ACK/NACK) bit sent from the target for tell the controller if it has correctly read the first 8 bits.
- The different data bytes followed each one by an ACK/NACK bit sent by the receiver as check of the successful reception. The number of bytes per message is unrestricted. The byte is sent starting from its most significant bit.
- An end condition. a 'low' to 'high' transition of the SDA line while the SCL is 'high'.

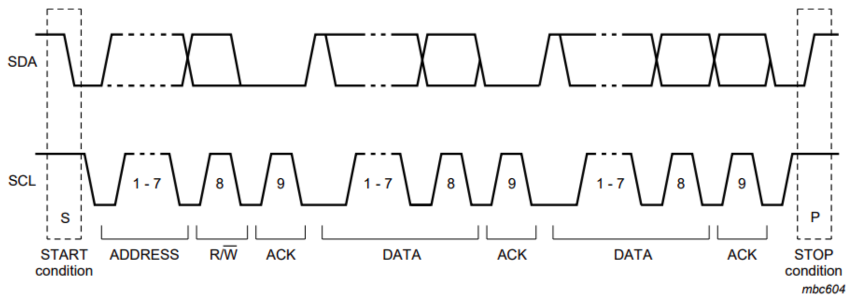


Figure 2.2: A complete data transfer

Are summarized now the most noticeable features of the I2C bus protocol that make this way of communicate a good design choice:

- Only a serial data line (SDA) and a serial clock line (SCL). So only two bus lines are required; This allows a big material saving.
- Assigning to each connected device an address, each one is uniquely software addressable and simple controller/target relationships exist at all times; controllers can operate as controller-transmitters or as controller-receivers.

- It is a real multi-controller bus including collision detection and arbitration avoiding data corruption if two or more controllers simultaneously initiate data transfer.
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.
- In the fastest Ultra Fast-mode the communication is still serial, 8-bit oriented, but the data transfer is unidirectional and it works up to 5 Mbit/s.
- An integrated filtering on the chip rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance. More capacitance may be allowed under specific conditions. This can be done increasing the number of bits dedicated to the addresses. Anyway, to date, with a 10-bit address a maximum of 1008 devices can be connected to the network (some addresses are reserved by the protocol).
- A large number of devices have a I2C-bus interface already integrated on chip, like the devices used in our study.
- Except for the physical wiring between the system elements. The communication is completely software defined and implemented.
- Devices can be added or removed from a system without affecting the other circuits on the bus.
- The CMOS ICs in the I2C-bus compatible range offer some useful features:
  - Extremely low current consumption
  - High noise immunity
  - Wide supply voltage range
  - Wide operating temperature range. [6]

2.1.2 UART communication protocol

The Universal Asynchronous Receiver - Transmitter (UART) is a serial data transmission protocol used to communicate devices serially and asynchronously. UART is mostly used for short distance and low speed applications [8]. UART identifies also the IC (embedded in more complex devices or created as a stand-alone element) used for implementing the homonym transmission protocol. It contains a parallel-to-serial converter for data transmitted from the computer and a serial-to parallel converter for data coming in via the serial line. The UART also has a buffer for temporarily storing data from high-speed transmissions [9] implementing a First-In-First-Out (FIFO) logic.

The UART protocol is serial, which means that the bits are sent one after the other on a single line, and asynchronous, indicating that the bus has not a clock synchronizing the communicating devices, which are a-priori tuned to work at the same baud rate. The allowable difference of baud rate is up to 10% before the timing of bits gets too far off [10]. The possible baud rates are: 9600 bps, 19200 bps, 38400 bps, 57600 bps, 115200 bps, 230400 bps, 460800 bps, 921600 bps, 1000000 bps, 1500000 bps. Some experiments were done reaching 20 Mbps baud rate [11] but the most commonly used speed is 9600 bps

The bus for the UART is composed of only two wires: the Transmitter (TX) and the Receiver (RX). The two lines allow a double way communication between the two connected devices, both can receive and transmit mutually. The wiring has to be crossed as shown in Figure 2.3. This transmission protocol allows a communication only between two devices.

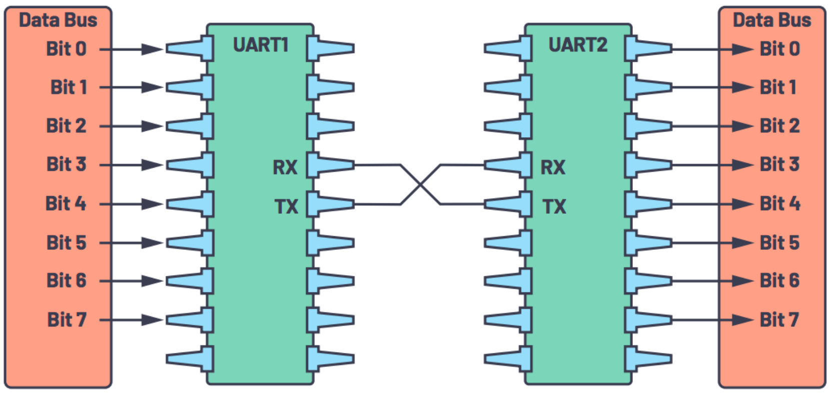


Figure 2.3: Two UART devices connected to each other with data bus

The data transmission is operated in the form of packets, Figure 2.4. Their creation and reading is directly implemented by the UART IC. The packet is composed of a start bit, the data frame (long form 5 to 9 bits), a parity bit and stop bits [10].

- **The start condition**: the transmission line is usually kept at a high voltage, to start the transmission the transmitter pulls down the line to a low voltage level for one sample time.

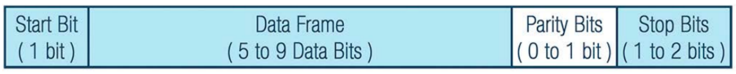


Figure 2.4: The UART packet

- **The data frame**: it is composed of 5 to 8 bits (9 if the parity bit is not used), this number is a-priori decided. It contains the data to be transmitted that are usually sent with the least significant bit first.
- **The parity bit**: it is sometimes used and it is a way to check the correct data transmission. If the data to be sent has an even number of ones the parity bit is set to 1, if the ones are even the bit is set to 0. The receiver counts the number of ones received in the data frame and if the parity coincides with the parity bit then the transmission should be free of errors. If, instead, the parity bit doesn't coincide, then the receiver knows that something went wrong in the communication.
- **The stop bits**: to notice the receiver of the end of the transmission the transmitter holds to a high voltage level the line for one or two bits duration.

Are presented now the different steps of the UART data transmission between two connected devices. In this case the component that sends the data is called transmitter (the transmission line is connected to its TX pin) while the one who receives it is the receiver (the line is connected to its RX pin), anyway in this communication protocol allows also the opposite data flow on the other connection line.

1. The transmitter IC receives data in parallel from a data bus.
2. The transmitter IC creates the packet, adding the start bit, the parity bit and the stop bit(s) to the data frame.
3. The transmitter sends via the TX pin the packet serially starting from the start bit. The receiver samples the communication line at the preconfigured baud rate. The transmission is shown in Figure 2.5.
4. The receiver IC extracts the data frame from the packet and checks the parity of the message.
5. If the parity is respected, the receiver IC convert the serial message into a parallel one and send it to other parts of the device.

The UART communication is a simpler protocol than the I2C that doesn't allows communication between more than two devices and works usually slower than the

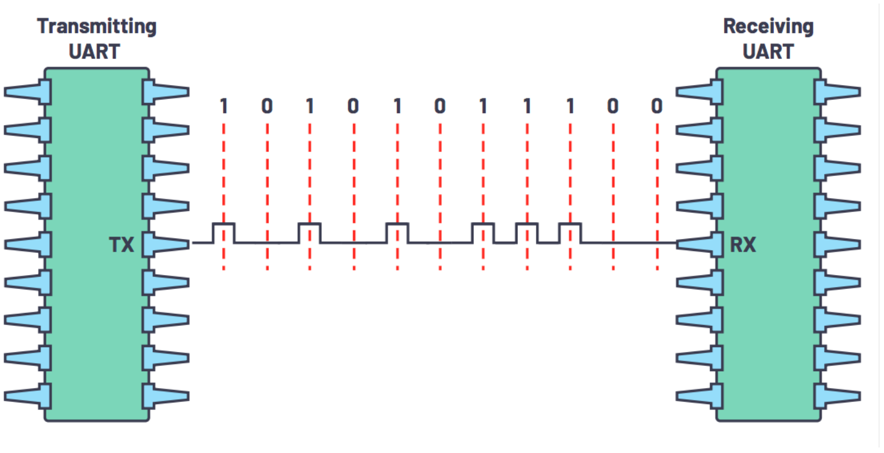


Figure 2.5: The UART transmission between transmitter and receiver devices

I2C; for these reasons the main communication channel in the system integration was the I2C one, while the UART was only used for debug and feedback purposes. Nevertheless this protocol requires to use only two wires, there's no need for a clock signal generation (that can sometimes be seen as a drawback), has a parity bit for the correct transmission check and it is widely used and documented. These reasons lead us to choose this method for our study as the communication bus for extracting information from the MCU, together with the fact that both our devices were predisposed for this protocol and testing the connection worked in a very good way.

2.2 Simultaneous Localization And Mapping

Simultaneous Localization and Mapping (SLAM) is the computational problem of mapping an unknown environment while, at the same time, localizing the vehicle itself in the obtained map [12]. It is one of the main and more essential issues of autonomous motion robots. For years this problem was considered to be a "chicken or the egg" issue. Nowadays, thanks to advanced and reliable visual and laser sensors together with a great development in mathematical and computational research different approximate solutions have been proposed as possible solution of this complex problem.

The SLAM is a wide technological system, implemented in various and different ways. Anyway every SLAM system has two main parts [13] (as shown in Figure 2.6):

- **The frontend** - The robot always needs some sort of sensor that allows it to observe and measure the surrounding environment. Different sensors can be used as laser ones, stereo camera or sonar sensors, together with odometry sensors. The frontend takes those measurements and transforms them into intermediate data such as constraints for optimization problems.

- **The backend** - The backend takes the intermediate data and uses them to solve the optimization or state estimation problem. It can localizes landmarks and understands the map in which the robot is navigating and at the same time it localizes the robot itself. Even in this case different solutions are possible such as interlacing algorithms, complex scan-matching algorithms, extended Kalman filters or particle filters.

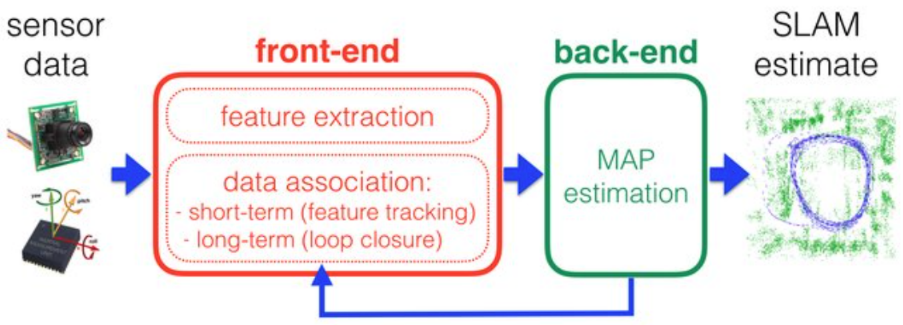


Figure 2.6: Front-end and back-end SLAM system division. The back-end can provide feedback to the front end for loop closure detection and verification.

In a simple description the basic resolutions of the SLAM problem relies on landmarks or occupancy grid map identifications (these are two possible approaches) from the sensor data together with the computation of the robot's own position in relation to those landmarks or in the grid map (also the landmarks are used for creating a landmark map of the environment). Then the robot explores the surroundings. During the motion the robot keeps track of its motion continuously localizing itself with respect to the landmarks or aligning the senors data with the gird map. Usually it exploits also odometry data and the control inputs for the robot motion (but this is not always required). When it has stored enough data of the surroundings it is able to save a complete map of the location.
The SLAM problem can be reduced to suitably find an expression for the equation 2.1 .

$$P\left(x_t, m_t | Z_{0:t}, U_{0:t-1}\right) \tag{2.1}$$

This is a probabilistic equation, since so are the variables forming it, and describes the probability at the discrete time t of estimating the robot pose (x_t) together with the landmarks coordinates or grid map values (m_t), given a series of robot control inputs (u_t is the single input at time t) and sensors data observations (z_t).

A main distinction between different SLAM technologies is due to the main sensor used for scanning the environment. The two main techniques are [14]:

- **Laser SLAM** - mainly a LiDAR sensor is used. This sensors have a wide range of measurement allowing the system to map bigger environments with smaller motions and the robot position is estimated as accurately as the laser measurements are (to date the sensors have reached a quite good accuracy). The problem of this technique is that if the environment has few obstacles and is pretty simple, the robot easily loses itself. This technology is usually used in warehouse robots, drones, mining operation robots and self-driving cars.

- **Visual SLAM** - it is done usually using Stereocameras or Time of Flight cameras. These sensors better localize the robot in obstacle-free environments, but have a shorter scan range and are sensitive to illumination changes.

In our study the laser SLAM for the environment mapping, since the robot has to move in differently illuminated environments and need wider range sensors for bigger environments such as universities or hospitals. In particular the SLAM implementation was the Hector SLAM.

2.2.1 Hector SLAM and Scan Matching algorithm

The Hector SLAM software was developed by Kohlbrecher, von Stryk, Meyer and Klingauf at the Technical University of Darmstadt. It is based on a scan matching algorithm for a 2D-SLAM resolution using a LiDAR sensor [15]. Hector SLAM returns a 2D estimate pose of the robot in the mapped environment, setting as the origin of the frame the motion starting point. The pose is returned at the laser scan frequency.
The software does not provide a loop closure detection but is anyway accurate enough for many real-world applications. The scan matching SLAM algorithm works computing the robot translation and rotation between two following laser scans according to the closest possible scan matching. In this software the Gaussian-Newton method is used to solve the scan matching problem by rigidly transforming the laser points on the existing map that is updated each time. This transformation is the same used for updating the robot position.
The environment is represented as an occupancy grid map. The laser scans are

processed in order to use only the ones inside two threshold planes eliminating the robot's 6 degrees of freedom and suitably operate on a bidimensional environment. Since the discrete nature of occupancy grids limit the occupancy probability and derivatives estimation, an interpolation scheme allowing sub-grid cell accuracy is employed using a bilinear filtering technique. The last technique used for improve the software is the one of using multi-resolution maps, in order to avoid local minima in the scan matching resolution.

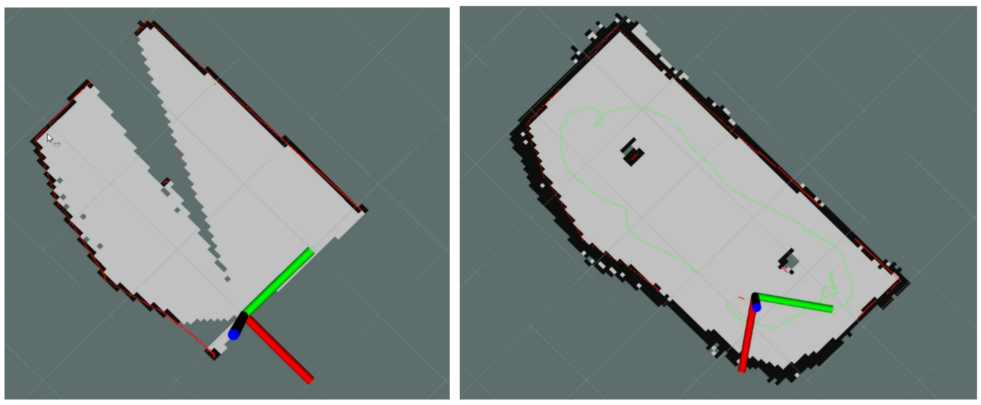


Figure 2.7: Hector SLAM on RViz desktop view. On the left the beginning of the motion, on the right the motion end with the final map and the robot path

This software works in good way with modern twodimensional LiDAR having a high scan frequency, updating the robot pose and map very frequently (in our case the Rplidar A3 from SLamtec is used, that has a maximum scan frequency of 20 Hz, that's not the highest possible value present on the market, but is enough for our purposes) and operating in a not too big environments (in our cases the inner rooms of universities and hospital are good sized environments).
Another important feature of this software is that it doesn't need odometry sensors, since the motion computed by the scan matching algorithm results very dense in time and accurate, even more than the one returned by odometry in different scenarios [15]. This allows the robot to save computational effort. The computed localization can be used as virtual laser odometry, as discussed in future chapters. This is a very good property of this technology, since in our robot the wheel encoders were not available. As a result of these considerations the Hector SLAM turned out to be a very good choice for our project.
The only drawback of the software is that the robot shouldn't move too rapidly, otherwise it is not able to compute the scan matching leading to a 'ghosting' effect of the map and of the localization [16], as can be seen in Figure 2.8 Anyway this is not a big problem for us since the robot won't move too fast and this software will

be improved by the use of the AMCL algorithm.

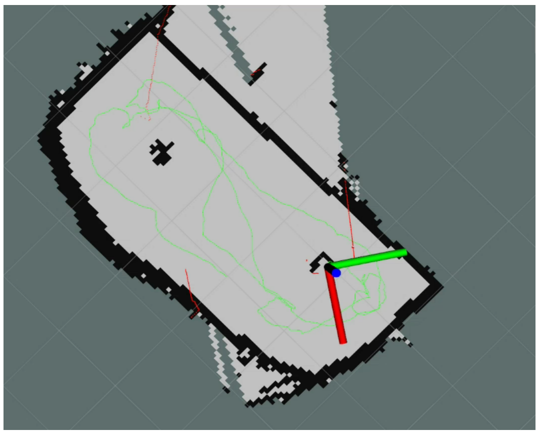


Figure 2.8: Hector SLAM 'ghosting' error.

It is now reported a more accurate description of the scan matching algorithm implementation [15]. As previously said the algorithm works aligning the laser scans with the previously obtained map, so the goal is to find the rigid transformation (that are the robot pose in world coordinates)

$$\xi = (p_x, p_y, \theta)^T$$

In order to minimize

$$\xi^* = \underset{\xi}{\text{argmin}} \sum_{i=1}^{n} [1 - M(S_i(\xi))]^2 \tag{2.2}$$

Where we have

- $\boldsymbol{S_i(\xi)}$ are the world coordinates of the scan endpoints $s_i = (s_{i,x}, s_{i,y})^T$ expressed as a function of the robot pose:

$$S_i(\xi) = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \begin{pmatrix} s_{i,x} \\ s_{i,y} \end{pmatrix} + \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{2.3}$$

- $\boldsymbol{M(S_i(\xi))}$ returns the map occupancy value at the $S_(\xi)$ coordinates.
- $\boldsymbol{n}$ is the number of laser scan endpoints.

So 2.2 search for a transformation that gives the best laser scans alignment with the map.
Given some starting ξ, we have to estimate the best $\Delta\xi$ that optimizes the error measure:

$$\sum_{i=1}^{n} [1 - M(S_i(\xi + \Delta\xi))]^2 \to 0 \tag{2.4}$$

So the first order Taylor expansion of $M(S_i(\xi + \Delta\xi))$ is used and the minimum is found by setting to zero the partial derivative with respect to $\Delta\xi$. The solution for $\Delta\xi$ leads to the following Gauss-Newton equation for minimization problem:

$$\Delta\xi = H^{-1} \sum_{i=1}^{n} \left[\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}\right]^T [1 - M(S_i(\xi))] \tag{2.5}$$

with

$$H = \left[\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}\right]^T \left[\nabla M(S_i(\xi)) \frac{\partial S_i(\xi)}{\partial \xi}\right] \tag{2.6}$$

So an approximation for the gradient of the map $(\nabla M(S_i(\xi)))$ is available, since, as previously said, a bilinear filtering technique is used to allowing a sub-grid cell accuracy for the occupancy grid map. While from 2.3 we obtain:

$$\frac{\partial S_i(\xi)}{\partial \xi} = \begin{pmatrix} 1 & 0 & -sin(\theta)s_{i,x} & -cos(\theta)s_{i,y} \\ 0 & 1 & cos(\theta)s_{i,x} & -sin(\theta)s_{i,y} \end{pmatrix} \tag{2.7}$$

Thus, thanks to the $(\nabla M(S_i(\xi)))$ and $\frac{\partial S_i(\xi)}{\partial \xi}$ expressions the Gauss-Newton equation 2.5 can be solved giving the $\Delta\xi$ the toward the minimum. Repeating the procedure the algorithm can converge to a minimum, even if this is not guaranteed, due to the non-smooth linear approximation of the map gradient. Anyway the procedure works with a sufficiently high accuracy in practice.

2.3 Adaptive Monte Carlo Localization

Adaptive Monte Carlo Localization (AMCL) is an non-Gaussian algorithm that implements a robot localization on a known map using a particle filter. This algorithm is the evolution of the previous Monte Carlo Localization (MCL) one using an additional Kullback-Leibler Distance (KLD) Sampling technique [17]. This software usually use laser sensors for obtain environment information such as obstacles distance.

AMCL algorithm gives a sampling-based representation of the localization, this gives different advantages with respect to other techniques [18] [19]:

1. It doesn't require Gaussian noise distribution assumptions in the model. It can accommodate almost every noise distribution and motion dynamics.

2. It can process raw sensor measurements, also negative ones. Without the need of extra processing of the sensor values. It can also elaborate these data at higher frequencies with respect to the Markov localization.

3. It is able to globally localize a robot (which means that it is able to localize the robot without knowing its starting position). Kalman filtering techniques are not able to do so.

4. In some instances it is also able to solve the kidnapped robot problem (which means that during the motion the localization fails and the robot is teleported to a wrong location, thinking to know where it is).

5. It greatly reduces the memory required for its implementation as the one required by the grid-based Markov localization. It focuses the computation in the most relevant areas and adapts to the computational hardware resources.

6. Since there's not a discretization of the state represented by the samples it is more accurate than the Markov localization using fixed cell size. Finally it is easy to implement.

The MCL algorithm is a statistical and probabilistic localization algorithm [20] that estimates the brief belief $bel(x_t)$ of the localization by particles. The algorithm starts from the initial pose $bel(x_0)$ iteratively combining it with sensor measurements and control sequence estimating the posterior robot's location distribution $bel(x_t)$. The single iteration of the MCL can be divided in the following steps:

1. The initial step is to draw sample M particles ($X_t = \{x_t^{[1]}, x_t^{[2]}, ..., x_t^{[M]}\}$) on the map starting from the initial pose belief.

2. The robot moves and odometry and control input data are collected.

3. A prediction of the robot movement is done making the different particles move, according to the previously stored data. The i-th particle pose belief is expressed by [17]:

$$bel_p(x_t^{[i]}) = \int p(x_k^{[i]}|u_t, x_{t-1}^{[i]})bel(x_{t-1}^{[i]})dx_{t-1}^{[i]} \tag{2.8}$$

   where u_t is the control input data.

4. Environment data sensors are collected in the robot position. Usually laser scans.

5. The weight of the single particles is updated according to the measurements data. The algorithm estimates which would be the data returned from each particle and gives higher weights to the particles whose data are more similar to the measured ones. The i-th particle pose is now described by the equation:

$$bel(x_t^{[i]}) = \eta\, p(x_t^{[i]}|y_t)bel_p(x_t^{[i]}) \tag{2.9}$$

   where $p(x_t^{[i]}|y_t)$ represents the belief of the i-th particle location according to the sensor's data η is a normalization constant [21].

6. The weight of the particles are normalized to 1.
7. Then the final phase consists in the resampling of the M particles that are re-organized on the map according to the weighted probability of the singles previous particles. This procedure is necessary in order to avoid degeneration.

At each time t the robot pose belief can be computed according to the different particles as in Equation 2.10, where $\mu_t^{[i]}$ is the i-th particle weight at time t and $\delta(\cdot)$ is the Dirac delta.

$$bel(x_t) \approx \sum_{i=1}^{M} \mu_t^{[i]} \delta\left(x_t - x_t^{[i]}\right) \tag{2.10}$$

The shown MCL phases can be seen in the Figure 2.9.

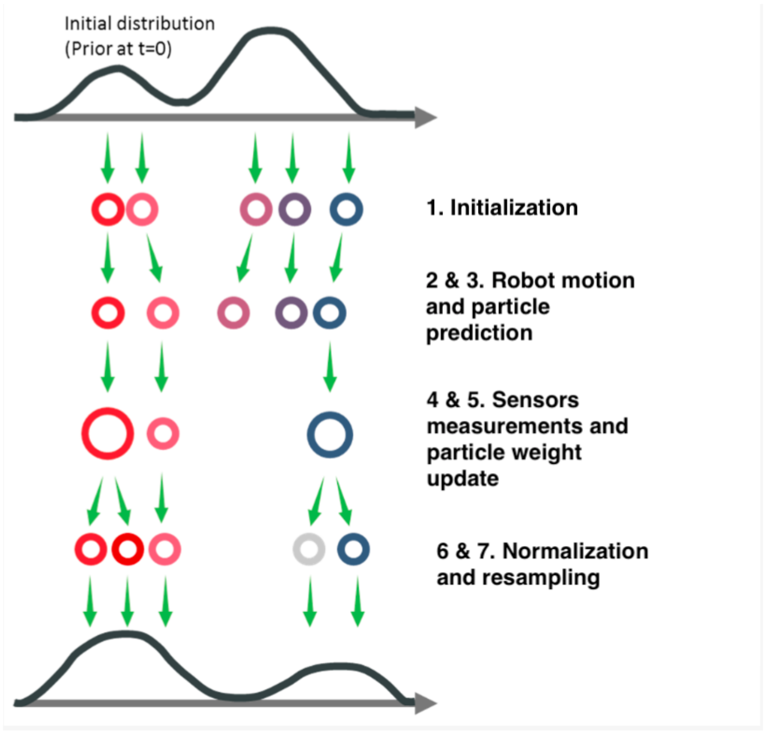
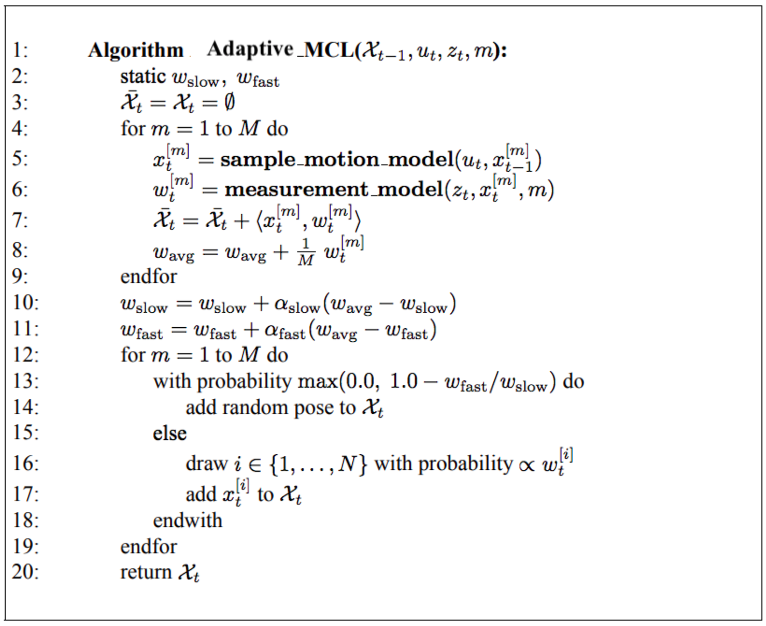

Initial distribution
(Prior at t=0)
1. Initialization
2 & 3. Robot motion and particle prediction
4 & 5. Sensors measurements and particle weight update
6 & 7. Normalization and resampling

Figure 2.9: Monte Carlo Localization phases

The AMCL algorithm starts from MCL and optimizes the resampling phase exploiting the Kullback-Leibler Distance (KLD) sampling technique. In a few words it decides in a probabilistic way which particles are to maintain and which one to eliminate suitably tuning the total number M of particles. This makes the computation faster and the localization more accurate. Another improvement of the AMCL is the possibility to solve the kidnapped robot problem. Since, if some conditions are met, in the resampling phase some extra poses are added into the map in different places with respect to the robot estimated pose, always according to KLD theory. This allows the robot to recover from bad localization convergence and to localize the robot even if the pose changes in a not continuous way.

The AMCL logic is presented as an algorithm in Figure 2.10 . It has to be noticed that ω_{avg} is the particles weight, ω_{slow} is the Long-term Likelihood Estimates while ω_{fast} is Short-term Likelihood Estimates. The parameters α_{slow} and α_{fast} ($0 \leq \alpha_{slow} << \alpha_{fast}$) tune the attenuation rates of the exponential filters with long-term and short-term weights [20].



```
Algorithm Adaptive_MCL($\mathcal{X}_{t-1}, u_t, z_t, m$):
    static $w_{\text{slow}}$, $w_{\text{fast}}$
    $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
    for $m = 1$ to $M$ do
        $x_t^{[m]} =$ sample_motion_model($u_t, x_{t-1}^{[m]}$)
        $w_t^{[m]} =$ measurement_model($z_t, x_t^{[m]}, m$)
        $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
        $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{M} w_t^{[m]}$
    endfor
    $w_{\text{slow}} = w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{avg}} - w_{\text{slow}})$
    $w_{\text{fast}} = w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{avg}} - w_{\text{fast}})$
    for $m = 1$ to $M$ do
        with probability $\max(0.0,\ 1.0 - w_{\text{fast}}/w_{\text{slow}})$ do
            add random pose to $\mathcal{X}_t$
        else
            draw $i \in \{1, \ldots, N\}$ with probability $\propto w_t^{[i]}$
            add $x_t^{[i]}$ to $\mathcal{X}_t$
        endwith
    endfor
    return $\mathcal{X}_t$
```

Figure 2.10: Adaptive Monte Carlo Localization algorithm

Finally is shown a series of images of a AMCL localization implementation in Figure 2.11 where we can see that the algorithm needs more updates to converge to a good and robot localization, close to the true one.

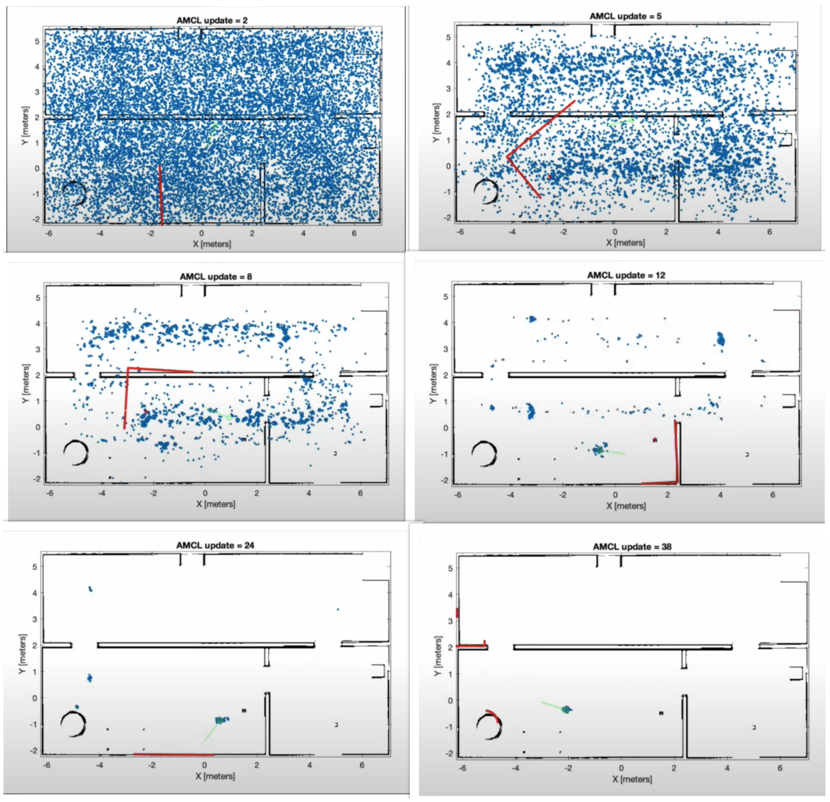


Figure 2.11: Series of images of AMCL for a mobile robot. From top-left to bottom-right the number of updates increases and so the localization accuracy

2.4 Path Planning using A* algorithm search

Path planning can be considered one of the key features of the AMR. It consists in finding the optimal or near-optimal from a starting to an end position avoiding obstacles. The optimality for the path is evaluated according to some indicators such as the lowest working cost, the shortest walking route, the shortest walking time, etc. [22]. In general simply the shortest path is chosen.

The path planning can be divided into two different types:

- **Global path planning** - in this technique the environment is considered completely known. The obstacles are known in shape and position. So the algorithm can compute the optimal path offline, before the robot motion. This technique obviously requires a preliminary phase, the environment map building. The global path can be computed using two different algorithm kinds:

  - *Heuristic search methods* such as Dijkstra algorithm and the A* algorithm (derived from the Dijkstra one)
  - *Intelligent algorithm* which examples are: bee colony, particle swarm, genetic, simulated annealing, and so forth.

- **Local path planning** - differently from the previous type, the local path planning assumes that the obstacles are unknown. They are recognized during the motion so the path is computed real-time in order to reach the goal position. Commonly used techniques are: rolling window, artificial potential field, and other intelligent algorithms.

In our study we decided to map the environment before leaving the robot to move autonomously. This is because in the real-world application of the developed robot the map will be known. So the path planning choice was among the different global planning techniques. We selected the A* algorithm since it's one of the most used algorithms, it allows to obtain the optimal path in many different cases and it is also very efficient [23].

The A* algorithm was first implemented by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968 [24]. A* algorithm is an heuristic graph search for state space algorithm for graph traversal and path search. It is an extension of the Dijkstra algorithm.
It works according to the following steps:

1. A map is uploaded and so are the start and end points
2. The map is treated as a series of nodes which are divided into pathable and unpathable nodes according to the obstacles presence in such map points. The pathable nodes are the ones upon which the logic acts.
3. The logic starts from the start node that is set as the current node.
4. The neighbor nodes to the current one are discovered according to a suitable neighbor search logic.
5. Each of the neighbor nodes is analyzed computing its distance from the starting point
6. If the neighbor node has never been considered or if in this analysis the distance is shorter than the previous time in which the node has been considered, the neighbor node is added to a priority queue (*open_set* in A) and its cost function (f) is updated as the sum of the distance traveled from the start point (g) and a suitably chosen heuristic function (h) that evaluates the distance between the actual point and the goal one (as shown in 2.11). At

the same time a path map is updated in order to be able to reconstruct the optimal path once the goal has been reached ($came\_from$ in A).

$$f(x) = g(x) + h(x) \tag{2.11}$$

7. The procedure is repeated from point 4 selecting as the current node the one in the priority queue that has the lower cost function.

8. The iterations stop when the final node has been reached, so the path that has led to it is obtained from the $came_from$ map. Or, otherwise, it ends if all the possible nodes have been evaluated and the search hasn't led to the goal position.

The shown logic can be found in the presented pseudo-algorithm in the appendix A .
In the A* algorithm two functions can be changed in different applications. The one that selects the neighbor nodes from the current one and the heuristic function (that is usually the straight-line distance or the Manhattan one).
In our study the neighbor set construction has been done expanding lines radially from the current point up to a chosen distance in every direction. The endpoint of the ray is stored as neighbor node but if the ray collides with an obstacle, the farthest pathable node from the current point in the ray direction is selected as neighbor.
As regards the heuristic function we chose the simple euclidean (or straight line) distance between the current and the end nodes, shown in 2.12 . This function gives a good computation cost together with good path results.

$$Distance(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \tag{2.12}$$

In Figure 2.12 is shown an example of a path computed thanks to the A* algorithm. In the image can be noted the red thin path and also the radial neighbor set selection (shown by the green lines).

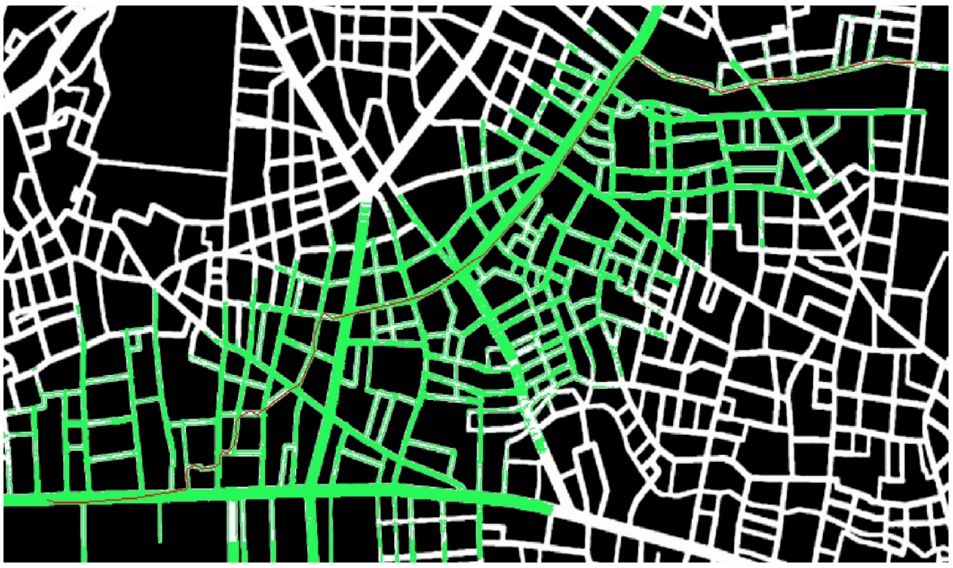


Figure 2.12: A* algorithm path computed example.

Chapter 3

System Integration

The first problem to be solved in this project was on how to make communicate in a fast and reliable way the on-board computer and the microcontroller. In this chapter the solution to this problem is exposed explaining why and how the I2C communication has been adopted. Then a section is devoted to how the communication and in general the microcontroller logic has been checked using a second communication channel, the UART transmission. Finally, the last two parts expose two different logics used to improve the communication reliability.

3.1 On-board computer and microcontroller connection using I2C communication

The robot logic is implemented in two main devices: the on-board computer, a Jetson Nano by Nvidia, and the microcontroller unit (MCU), a C2000 Piccolo MCU in the LAUNCHXL-F28069M development kit from Texas Instruments.
Many different tasks are assigned to the computer, in particular it is required to retrieve and elaborate the sensor's data. It has to implement the indoor localization logic in a known environment, starting from the sensor's data and computing the most probable location of the robot in a defined frame. The path planning is another task that it is required to do, knowing the starting and end position it elaborates a path on a previously obtained map avoiding the obstacles. It has also to implement the tracking of the obtained path, sending to the microcontroller the robot localization and the actual path point to be reached. Meanwhile, it has to check the feedback data coming from the MCU. Finally it receives the keyboard movement input, interprets them, and sends them to the MCU. In the future it will also be required to integrate the data from different sensors and fuse them in order to obtain a more precise robot localization in different environments. Summarizing

the computer is where most of the robot logic takes part, like a human brain.
The MCU, instead, has less tasks, but it is not not less relevant. It's main work is to compute the required speeds, transformed as Duty Cycles (DCs) of a square wave to drive the motor wheels (by mean of two suitable motor drivers) to make the robot reach point by point the entire path till the final destination. This is done using a suitably designed control logic. It also performs the motion required from direct keyboard robot control. So the MCU is the device that actuate the robot motion, like a human cerebellum.

So, both the devices have different and important tasks for the robot autonomous motion and they require a good mutual communication channel. As mentioned it is the computer that sends data to the microcontroller that actuates the motion. In particular the data that the Jetson Nano needs to send to the MCU are the robot localization, the path point to be reached, the enables signals that starts the motion when everything is ready and finally the keyboard input when the robot moves from direct keyboard signals.
It has to be pointed out that at the end of the work, during the overall testing phase, the MCU turned out to be not performing enough, slowing down the required computation. So the same control logic that had to be carried out by the microcontroller was then implemented directly on the on-board computer and the final DCs were directly sent to the MCU.
Are now presented more in detail the different data to be sent.

3.1.1 Data sent from the on-board computer to the MCU

The data that the Jetson Nano computer needs to send to the MCU are:

1. **The robot Localization**: It is composed of three floating point numbers, the X coordinate, the Y coordinate and the Z angle orientation. These coordinates are expressed in the *map* frame of the robot logic (these aspects will be explained in deeper later on).

2. **The next path point**: It is composed of just two floating point numbers, the X coordinate and the Y one. It is expressed in the same reference frame of the localization (the *map* one), this is important for making operations with the two data together. It is the actual path point that the robot has to arrive at, when it is reached the next one is published till the final destination is achieved.

3. **The enable signals**: A couple of bit values, the Localization Ready Bit (LRB) and the Path Ready Bit (PRB). They are used to communicate to the

MCU that the localization and the next path point are correctly sent and so the control logic can start and the robot can move.

4. **The speed keyboard commands**: a couple of numbers that can be 1, 0 or -1 that inform the MCU, with a suitable embedded code, the direction in which the robot has to move. The first number drives the linear speed: the value 0 means no motion, 1 moves straight for 1 second at a predefined speed and -1 makes the same linear motion but backwards. The second number drives the steering angular speed: 0 means no steering, 1 steer for 1 sec at a predefined speed counter-clockwise and -1 execute the same motion but clockwise.

5. **The Duty Cycles values**: It is a message composed by two floating point numbers the left DC (DCl) and the right DC (DCr). These values are directly sent to the MCU to make it execute them as rotational speed of the wheel motors.

As can be noticed, the data that need to be sent are of different sizes and different types. This led to the creation of data messages, composed by different bytes to send more complex and different information, identifying them by an address number in a way that they can be recognized and treated differently as they require.

3.1.2 I2C communication protocol for data sending

The Inter Integrated Circuit (I2C) protocol is a multi-slave, multi-master, single-ended serial computer bus. This standard uses only two bidirectional open-drain lines, the Serial Data line (SDA), for the data transmission, and the Serial Clock line (SCL), for the clock sending. The I2C is a widely used and reliable bus as exposed in the dedicated section in the Chapter 2, where also more detailed technical information can be found.
This communication protocol was chosen as the best way to make the on-board computer and the MCU communicate, since:

- It is a well known and widespread, fast enough and reliable communication protocol.
- With respect to other protocols, thanks to the clock synchronization, the master-target communication data are more easily comprehensible.
- It is organized in messages, as we need in order to send the different data kinds.

- Thanks to the devices addressing logic of the protocol, only two lines are required to make a communication between many controllers and targets. A new device just needs to be attached to the two bus lines (SDA and SCL) saving cable for complex wirings.
- Last but not least, this protocol is compatible with the two devices that we need to make communicate. They already have an integrated circuit for the I2C signals handling and built-in software resources for dealing with the communication. (the Jetson Nano is not CAN protocol compatible unless the use of some additional extension, while for the SPI protocol it requires some kernel modification)

So, in our case, the Jetson Nano computer is the controller that leads the communications and sends data to the target MCU. The data to be sent are organized according to the protocol of the chosen communication bus. The messages in the I2C are composed of a starting condition, 7 or 10 addresses to recognize the target to which the controller wants to communicate (in our case 7 is more than enough having till now just one target and one controller). Then there are the data bytes, followed each one by an acknowledge bit (we choose to use 4 data bytes that are enough for our usage) and a final end condition.
The previously exposed data that needs to be sent are then organized in such a message structure as follows in Table 3.1.

I2C Message organization					
Data	**Target address**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**
Localization - X	0x08	Data address 0x01	Sign	Integer part	Fractional part
Localization - Y	0x08	Data address 0x02	Sign	Integer part	Fractional part
Localization - Z	0x08	Data address 0x03	Sign	Integer part	Fractional part
Next path point - X	0x08	Data address 0x05	Sign	Integer part	Fractional part
Next path point - Y	0x08	Data address 0x06	Sign	Integer part	Fractional part
Keyboard motion commands	0x08	Data address 0x0A	Linear command	Angular command	*(Not used)*
Enable signal	0x08	Data address 0x0B	LRB	PRB	*(Not used)*
DCr	0x08	Data address 0x10	Sign	Integer part	Fractional part
DCl	0x08	Data address 0x11	Sign	Integer part	Fractional part

Table 3.1: Data organization according to the I2C message structure

As can be seen from the Table 3.1 the first byte of each message is the data address that uniquely identifies the kind of message. The correspondence between address and message type is known both by the controller and by the target.

The data sent as sign, integer and fractional parts are originally floating point numbers that are unstructured in order to be sent via single bytes.
The logic is to divide the floating point number in its sign, the integer part and the fractional part multiplied by one hundred and send these three values as three different bytes coding the numbers as data type *int8* (*int8* represents a signed integer number stored in 8 bits). The type *int8* is able to store numbers from -127 to 128. In this way the sign is sent as -1 if the number is negative and 1 if it is positive. The integer part is extracted from the floating point number and the fractional one too and then multiplied by 100, rounded to the closer integer number and saved as *int8* byte. In this way the maximum and minimum numbers that can be sent are + 128.99 and - 128.99, with an accuracy of 0.01. These limitations are not restrictive for anyone of the floating point data numbers. The localization and next path points numbers are expressed in meters, so for our applications we won't have values higher than $\pm$ 128.99 m with respect to the map frame in which the coordinates are expressed and also the accuracy of 1 cm is good enough for our actual applications. As regards the DCs values they are sent in percentage, so they

can go from -100 to + 100 and an accuracy of 0.01 percent is more than enough. If in future applications higher coordinates values will be required simply using the *uint8* (*uint8* represents an unsigned integer number stored in 8 bits) type instead of the *int8* one could enable the values to reach ± 255.99 m. If even higher values coordinates will need to be sent, a similar logic to the fractional part but applied to the thousands and hundreds can be implemented by adding an extra byte to the message and sending after the sign the thousands and the hundreds divided by 100 and then the tens and units. Doing this we can send numbers from - 9999.99 to + 9999.99. Another possible thing to do is to increase the accuracy of the sending, adding an extra final byte to the message using the exact same logic used for increase the integer part value but applied to the fractional one reaching an accuracy of 0.0001.

This messages are then received by the MCU which first first check the data address in order to understand which data kind it has received and then processed to build up again the original data values.
For the keyboard motion and the enable signal messages, since the bytes are already the final data values, no operations are done and they are directly used as they arrive. Just for the enable signals a check of the signal is done: if the value is not 0 or 1 it is reject and the old value maintained, in order to exclude wrong transmissions that could affect the robot working.
Instead the floating point numbers data that are unstructured in the three bytes are then reconstructed in the original values, by multiplying the integer part by the sign and adding to the resulting number the fractional part divided by one hundred.

Finally, the I2C communication and the related message creation/reading were directly implemented on the two software. This was done by setting the suitable Simulink block for the I2C reception on the model used for create the C code to be deployed inside the MCU and calling the kernel functions for the Jetson Nano computer on the C++ codes of the ROS logic nodes to send data through the I2C bus pins of the device (In the ROS logic a node is a process that performs computation [25], it is the basic operating part of the system). Then the physical wiring was done between the corresponding SDA and Serial Clock line (SCL) pins of the MCU and on the on-board computer.
The obtained communication system was tested and debugged.

3.2 UART communication for microcontroller check, debug and signals feedback

After different tests in the final part of the study, trying to check the autonomous motion logic, a need for a real time view of the MicroController Unit inner signals was noticed. The real time simulation of the microcontroller using Simulink wasn't useful since it requires too much elaborations to the MCU and so all the logic is slowed down and not real.
A good way to know in real time what is happening inside the Texas Instruments Launchpad was to use a second communication channel in order to extract from it some relevant signals in order to check the functioning of the device.
A second communication channel was used since on the I2C a good amount of data was already sent from the on-board computer to the MCU, so adding data exchange in the reverse way could have overloaded the channel, but more importantly a second channel was used since also the I2C communication logic needed to be verified.

The second communication channel chosen was the Universal Asynchronous Receiver - Transmitter (UART). This protocol was chosen since it is easily implementable. It requires only two wires and both the devices were already predisposed for implement such a communication (the Jetson Nano is not CAN protocol compatible unless the use of some additional extension, while for the SPI protocol it requires some kernel modification). This bus protocol is less articulated than the I2C since it enables communication only between two devices and the bytes are not organized in messages but only in simpler packets that go from 5 to 9 bits (In our case 8 bits are used). But these elements are not a problem since this channel is used to exclusively extract information from the MCU so no other devices should be connected to the network. While for the fact that the communication sends just single bytes, this thing can be overcome, allowing the sending of more complex data, by virtually creating messages as in the I2C and send them bytes by bytes one after the other and use a logic that recognize the received message on the receiving device.
A more detailed description of the UARTcommunication protocol can be found in the dedicated section in the Chapter 2.

The receiver device is obviously the on-board computer, being the only device that can receive such kind of data and is always on the robot close to the MCU making the wiring possible also during the autonomous motion tests.

The signals extracted from the MicroController Unit are many and of different types, in order to check the logic in different parts and have a more complete debug. Then, after the implementation of this second communication channel, it was noticed a need of increasing the quality of the I2C transmissions, so this extracted MCU signals were also used as feedback of the I2C sendings for implement a computer logic that improves the communication (as discussed in last section of this chapter).

3.2.1 UART extracted signals

Are now listed the different signals extracted from the MCU and their data type:

- Identification number 01: X coordinate localization (floating point number).
- Identification number 02: Y coordinate localization (floating point number).
- Identification number 03: Z coordinate localization (floating point number).
- Identification number 05: X coordinate next path point (floating point number).
- Identification number 06: Y coordinate next path point (floating point number).
- Identification number 10: Drive commands (from the keyboard command logic), this data are five different boolean signals: the enable keyboard movement, the forward command, the reverse command, the left turn command and the right turn command.
- Identification number 11: Enable signals, that are the two boolean bits: LRB and PRB
- Identification number 12: Omega wheels, two floating point values (the right omega wheel and the left one)
- Identification number 13: Duty Cycle wheels and Motor Enable, these signals are two floating point ones, the DCs, but rounded and treated as integer, while the motor enable signal is a boolean one.
- Identification number 14: Motor Enable and Motor Direction Right and Left. These signals are all of them boolean.
- Identification number 15: Bytes directly arrived from i2c. Three integer numbers.
- Identification number 16: DCr (floating point one) - for direct DC sending from Jetson Nano.

- Identification number 17: DCl (float one) - for direct DC sending from Jetson Nano.

These are all the signals that for different debug and feedback purposes were retrieved from the MCU. Are listed here already grouped as they compose the different messages and also the message identification number is exposed. In the following section the message creation logic will be discussed.

3.2.2 UART Message creation logic

As previously anticipated in order to increase the complexity of the information sent via UART the single packets sent (decided to be 8 bits long) are organized in messages, similarly to the I2C messages. This allows to send different kind of data, even more complex than a simple 8 bits information, and make them be recognizable by the receiver so that it can treat them differently, accordingly to their need.

The basic idea for the message creation is to exploit the seriality of the transmission, in a sense that the packets are sent on the channel one by one, one after the other, so thanks to the order of the sendings a message can be created. The message starts with a starting number or sequence (one or more bytes) that makes the receiver understand that a message is starting. The following byte is then the message identification number, it uniquely identifies the kind of message sent (both the transmitter and the receiver should know the numbers associated to which messages), these are the numbers presented in the previous list. Then the data bytes are sent, their number depends on the type of message, the receiver already knows how many bytes he has to receive according to the message identification number. Once terminated the information bytes sending no ending condition are sent and another message can be sent. In this way the communication is faster and testing it directly on the devices it works in a good way also without ending number and without any kind of byte sending check.

The data bytes are organized in the message as exposed in the Table 3.2 .

UART Message organization							
Kind of message	**Message ident. number**	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**	**Byte 6**
Coordinates	01, 02, 03, 05, 06	Sign	Integer part	Fractional part	-	-	-
Drive commands	10	Enable keyb. movement	Forward cmd.	reverse cmd.	Left cmd.	Right cmd.	-
Enable signals	11	LRBt	PRB	-	-	-	-
Omega wheels	12	Sign (R. wheel)	Integer part (R.)	Fract. part (R.)	Sign (L.)	Integer part (L.)	Fract. part (L.)
DCs and Motor enable	13	DCr (as integer)	DCr (as integer)	Motor Enable	-	-	-
Motor enable and Motor directions	14	Motor Enable	Motor Direction R.	Motor Direction L.	-	-	-
Arrived bytes	15	First byte	Second byte	Third byte	-	-	-
DCr and DCl	16 and 17	Sign	Integer part	Fractional part	-	-	-

Table 3.2: UART messages structures

The message bytes are all coded as *uint8* in order to have numbers that goes from 0 to 255 for each byte.
For the floating point values the same logic as for the I2C is implemented, dividing the number in its sign, integer and fractional parts and sending them as different bytes and then reconstructing the number in the receiver device. The only difference is that here the bytes are *uint8* so the sign is coded as 0 for positive values and 1 for negative ones.
The last thing to be noticed is that the empty cells means that those bytes doesn't exist, so each message has a different size and the receiver is aware of that, it knows how each message is long and how to process the different messages, if they already are the required signal or if they require a floating number reconstruction.

The last thing to be pointed out is that the UART communication works at a lower speed than the I2C, but since the sample time of the microcontroller is set to 0.001s, the communication is enough fast to send between each clock every data required. Obviously not all the messages were sent during each test, but just the ones that were required for the specific debug or feedback, sending a maximum of 5 messages.

The presented communication logic was then implemented in the two devices using the suitable UART transmitter block in the Simulink model deployed in the RCU (the microcontroller) and the suitable Python kernel functions for receive the UART signals on the Jetson Nano computer.
The physical crossed wiring between the RX and TX pins of the two devices was done.
Finally the communication channel was tested sending the inner MCU signals to the on-board computer in a good and precise way.

3.3 Outlier rejection logic for the localization coordinates

During the test phase of the overall autonomous motion some problems were encountered. In particular it was noticed that the I2C sendings weren't so precise every time and sometimes even very far from the real values sent from the computer. The data were wrongly received by the MCU and so the control logic was wrongly implemented. This was probably due to some hardware problems in the devices that weren't easy to fix. Alternatively two different logic were implemented to improve the quality of the I2C transmission. The first one is the outlier rejection for the localization coordinates that are sent at high frequency and are related to continuous values. The second one is the UART feedback of the sendings for the next path point and enable signals that are sent with lower frequency and each sending is not so easily relatable to the previous ones. The first logic is presented in this section, while the second one in the following one.
The I2C bus wasn't changed in favor to the UART one, since this second channel is slower and doesn't allow adding other devices to be connected (both controllers and targets).

The idea behind the outlier rejection logic for the localization coordinates is that these messages are sent at high frequency and they refer to a value that is continuous in time, the position. Since the frequency is high there's no time to check every single sending with a feedback and correct it, also because the following transmission will be different to the previous one. Instead, the idea was to exploit the continuity of the robot position.
The logic is to reject the new localization values, x, y and theta arriving from the I2C channel that differ more than a suitably tuned threshold to the previous localization values.

To suitably select the rejection thresholds I run a test to check the robot localization lowest frequency availability, since the SLAM doesn't return a localization with a constant frequency, but after different observations, the higher delta time between two localizations was 0.1 s. So during this time it was computed the maximum distance and turn that the robot can do. The maximum linear speed of the robot is around $v^{max} \simeq 1.1\, m/s$ (this value came from a test with the robot exposed further on in this thesis, the test consisted in running the robot straight in order to create a direct relation between DC and the robot speed). The related maximum wheel angular speed can be easily computed knowing that both the wheels turned at their maximum speed, so using the kinematic equations that describes a differential robot (as our robot) we obtained:

$$v = \frac{r(\omega_r + \omega_l)}{2} \tag{3.1}$$

With $r = 0.08\,m$ that is the robot wheel's radius. In our case $\omega_l = \omega_r = \omega_{wheels}$ so the equation became:

$$v = r \cdot \omega_{wheels}$$

$$\omega_{wheels} = v/r$$

So the maximum omega wheel is

$$\omega_{wheel}^{max} = 1.1/0.08 = 13.75\,rad/s \tag{3.2}$$

For the robot rotational speed (ω) the differential robot equation is:

$$\omega = \frac{r(\omega_r - \omega_l)}{d} \tag{3.3}$$

Where $d = 0.39\,m$ is the axis length between the two wheels.
So the maximum rotational speed is when the two wheels spin at their maximum speed in opposite directions, so $\omega_r = 13.75\,rad/s$ and $\omega_l = -13.75\,rad/s$, according to a reference system with z axis orthogonal to the robot and pointing upwards.

$$\omega^{max} = 0.08 \cdot 2 \cdot 13.75/0.39 = 5.64\,rad/s \tag{3.4}$$

So the maximum linear and rotational distances that the robot can run in the maximum delta time between two localization are:

$$\begin{aligned} distance^{max} &= v^{max} \cdot \Delta t^{max} = 0.11\,m \\ angle^{max} &= \omega^{max} \cdot \Delta t^{max} = 0.564\,rad \end{aligned} \tag{3.5}$$

As thresholds for the rejection logic we could use for the linear coordinates x and y the $distance^{max}$ and for the angular one the $angle_{max}$ values, since they are the maximum displacement that physically the robot can do between two localization reception, so higher values makes no physical sense and can be eliminated.
The problem was that during the test sometimes happens that the robot localization in the ROS environment in the computer is not suitably transformed in the correct reference frame (the *map* one, since the used localization is the SLAM one, corrected by the AMCL and is originally expressed in the *odom* frame, this part will be discussed in more details later on). In these cases the position value is not updated and so the MCU receives the same previous robot localization even if the robot has actually moved. This error lead to an higher delta time in which the RCU doesn't receive an updated value of the robot position and so this can lead, using low

thresholds for the outlier rejection, to a right localization rejection only because the robot has moved too much in the time between two correct localization sendings. As a consequence the MCU localization remains stuck in a fixed value and the logic works no more.
So the threshold has to be tuned higher than the computed maximum distance and angles. After several tests the right values to use were:

$$\begin{aligned} Linear\,Threshold\,(x\,and\,y) &= 0.5\,m \\ Angular\,Threshold\,(\theta) &= 1.5\,rad \end{aligned} \tag{3.6}$$

These values seems a little high but they are tuned in a way that no correct localization are rejected while almost all of the wrong ones are refused.

The implementation of this logic was done on the Simulink model used to deploy the code inside the MCU. The schemes are shown in Figure 3.1 and 3.2.

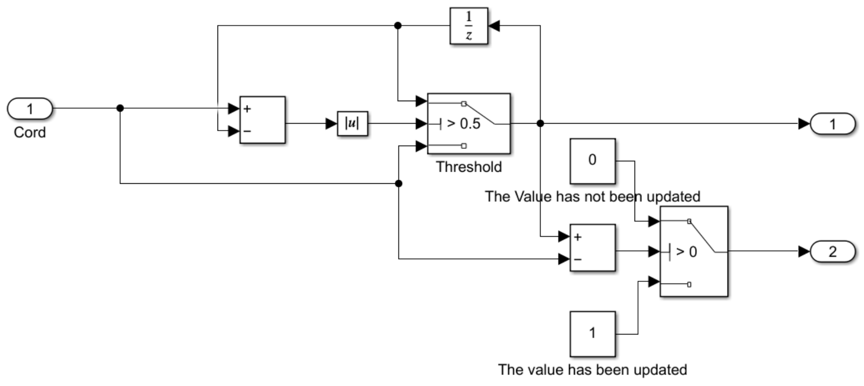


Figure 3.1: Simulink scheme representing the outlier rejection logic for x and y signals

For the linear coordinates the implementation was easier. As shown in Figure 3.1 the previous signal value is obtained using a unit sample delay block (the discrete-time operator z^{-1}). then the new and old values are subtracted, if the absolute value of the difference is greater than the threshold the previous value is hold (accordingly to the equation 3.7, where the value k stands for the discrete sample time $T = k\Delta t$ and T_x and T_y are respectively the x and y thresholds), if not the new received value updates the localization signal.

$$\begin{aligned} |x(k) - x(k-1)| < T_x \\ |y(k) - y(k-1)| < T_y \end{aligned} \tag{3.7}$$

If the value has crossed the zero and changed sign the logic works anyway but becomes more strict. It is no more the difference but the sum of the two values that should be less than the threshold. Since the values around zero have by definition low absolute values and since the threshold has been chosen a little bigger on purpose, this doesn't affect the functioning of the logic.

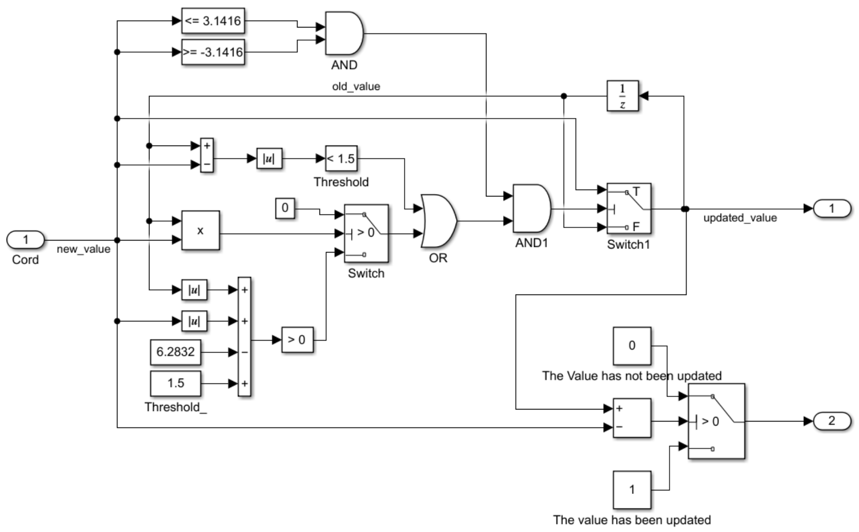


Figure 3.2: Simulink scheme representing the outlier rejection logic for angle signal

For the angular coordinate there was the need of an extra reasoning for the logic implementation, since the angles (in our system) goes from $-\pi rad$ to πrad and when the value correctly passes from the two extremes it has a difference gap of

$2\pi rad$. In order to not reject the values when passing from a limit of the angle range to the other the first thing to do is to effectively check if the previous and the actual coordinate have changed sign (as shown in Figure 3.2). If not, simply the absolute value of the difference between the old and new values is compared with the threshold, as for the linear coordinates (equation 3.8, where T_θ is the angle threshold value) and if the difference is less than the threshold than the value is updated, otherwise the old value is kept.

$$|\theta(k) - \theta(k-1)| < T_\theta \tag{3.8}$$

If, instead, the coordinate has changed sign, two cases are possible, that we have jumped from one angle range limit to the other or that we are crossing the zero. In both cases a second logic is activated and checks if, also taking into account the limit jump, the old and new values are anyway not more far then the wanted threshold, this is done by checking the following condition:

$$|\theta(k)| + |\theta(k-1)| - 2\pi + T > 0 \tag{3.9}$$

In the first case if the 3.9 is satisfied then the blocks logic returns a one that regardless the 3.8 blocks check it returns a one in the OR block and the signal is updated, if the equation is not satisfied then the value is not updated since it's difficult that the 3.8 became true.
If, otherwise, the signals have opposite sign because they are crossing the zero, then the more strict check became the one of the 3.8 equation, that became a check on the sum of the old and new signals that have to be less then the threshold. In this way the control is harder to be satisfied but since the threshold is enough high it allows the correct updating of the localization even when crossing the zero value.
Finally, all the shown cases in which the signal can be updated lead to a final signal updating only if the new value is contained inside the acceptability range $[-\pi; \pi]$. The result of this check enters as input in an AND block (the AND1 block in the Figure ...) together with the result of the previous logic, so only if both the signals are 1, and so both the parts of the model consent to update the value, it is effectively updated.
The final part, in order to make the outlier rejection working in a suitable way, was the design of a tuning phase at the start of the localization transmission on the computer.
The tuning part is necessary since the unit discrete-time delay in the Simulink model is initialized to zero, so with the first localization received the comparison for the rejection is between the actual received coordinates and a fictitious zero value, this can leads to a localization stuck to a zero value thanks to the outlier rejection itself. So once the localization is available on the computer, before starting

the effective I2C sending, the assigned ROS node sends for each coordinate an increasing (or decreasing if the localization is negative) value starting from zero till the real robot localization value. The messages are spaced with a distance slightly less than the outlier rejection threshold, in order to let the RCU receive one by one the values and tune the localization to the real value.
Thanks to this the coordinates reach their real value and the outlier logic can follow the robot motion refusing the wrong transmissions.

The shown logic was then tested by making the robot localize itself, making it move and check via UART the received MCU localization coordinates values, sent via I2C. A great increase in the I2C transmission quality was noticed. Almost all the wrong values were neglected, except for the one more close to the real values, that anyway doesn't create so much problem to the control logic, and the RCU localization follows the real one without remaining stuck at some values rejecting correct coordinates.

3.4 Feedback sending control for the I2C messages

The second implemented logic for improve the I2C transmission is the feedback sending control.
This transmission control works thanks to the second communication channel between the microcontroller and the computer, the UART bus. This second communication channel was more precise than the first one and so a good source for reliable feedback in the logic. As previously said the entire transmission wasn't moved from the I2C channel to the UART one since the second one doesn't allow to add extra devices to the bus and it's slower.
The simple idea was to use the retrieved signals coming from the MCU in order to check if they equal to the I2C sent data, if not, they are sent again, till the two values became the same.
In order to do so the UART received values were published on suitably created ROS topics (topics are named buses over which nodes exchange messages [26]). these values are then read by the nodes that have the task to publish over the I2C the relatives data and so these nodes publish over the bus only if the value between the feedback and the desired value to be sent, are different.

This simple logic was applied to the path points coordinates and to the enable signals, that are messages that require to be sent occasionally and it is important that they are received correctly.
The logic was then implemented on the computer and tested on the robot. It

greatly increased the quality of the I2C transmission. Quickly correcting the wrong communications and allowing a great improvement in the autonomous motion functioning, having on the RCU almost always the right path points coordinates and enable signal values.

Chapter 4

Laser Mapping and Localization

One of the main problems for autonomous mobile robots is to be able to accurately localize itself in the environment. Only after they know where they are, they can identify where their goal location is, and reach it. The second part of the work treat such a topic. In a first section we deal with the enclosed environment laser mapping in order to have a precise map in which the robot can localize itself. Then the robot LiDAR localization methods are shown, the scan matching used as virtual odometry improved in accuracy by the AMCL algorithm.

4.1 Laser mapping of the environment using Hector SLAM

Before operate a localization logic the knowledge of the robot environment is needed. In our project we decided to exploit the laser sensor, the Slamtech Rplidar A3, to which our robot has been provided, in order to suitably map the environment and save an occupancy grid map of the surroundings.
We decided to operate in such a way since in the future real-world applications in which our robot will be employed, it will always work in known environments, as university campuses, hospitals or airports. Thus, having a map of the place allows the robot to accurately localize itself and being able to implement a global path planning from a start to an end point, instead of exploiting just a local path planning with the risk of remaining stuck in local minima. Moreover knowing the navigation map is very important in close environments where the GPS doesn't work, so thanks to the knowledge of the in-building location and its coordinates in

a world reference frame the robot is anyway able to know where globally it is and autonomously navigate.

The software used for the environment mapping is the Hector SLAM one by Stefan Kohlbrecher and Johannes Meyer. The Hector SLAM is based on a scan matching algorithm which continuously compute the robot localization in the map computing the transformation (x, y, theta) which gives the best matching between the previously computed map and the actual laser scans. In parallel the laser scans matched with the map are used to update it. A deeper and more complete analysis of this technology can be found in the dedicated section in the Chapter 2. The robot localization is obtained with respect to a frame fixed in the starting position of the robot motion. This will be also the origin of the frame associated to the final computed map.
This software doesn't have an explicit loop closure but anyway works in a good fashion if the laser sensors has a sufficiently high scan rate. In our case the Rplidar A3 has a maximum scan rate of 20 Hz, enough for a good SLAM problem solution in our applications.
The Hector SLAM returns the computed map as an occupancy grid map, together with a YAML file with the related map informations.
The main feature of the Hector SLAM software is that it could also work without odometry data, since the scan matching algorithm return a robot localization at the laser scan frequency and with a enough high accuracy. Using scan matching can even lead to better results than the ones obtained with the classic odometry [15], which tends to have an accuracy degeneration during the motion for the integrative errors. In our study the odometry data from the wheels weren't available so this software was a perfect solution to this problem.

A short introduction to the standard navigation robot frames for ROS (according to the REP 105 conventions [27]) is required. As shown in Figure 4.1 the frames for the robot motion are: the *map* frame a world fixed frame, the *odom* a fixed frame in which the robot localization is continous in time, the *base_footprint* frame is simply the projection of the *base_stabilized* on the ground if we want to report a 3D robot movement over the plane, the *base_stabilized* frame which derives from the horizontal stabilization of the next one but keeping the height (z) information, the *base_link* (sometimes called *base_frame*) is the frame rigidly attached to the robot including also the roll and pitch angle information and finally the *laser_link* (in our study only called *laser*) which is the frame attached to the laser sensor, where the senors data are computed.
In our study, since the robot is expected to move only on plane surfaces we can move the logic to a simpler bi-dimensional system, collapsing the *base_footprint* and the *base_stabilized* frames into the *base_link* one.

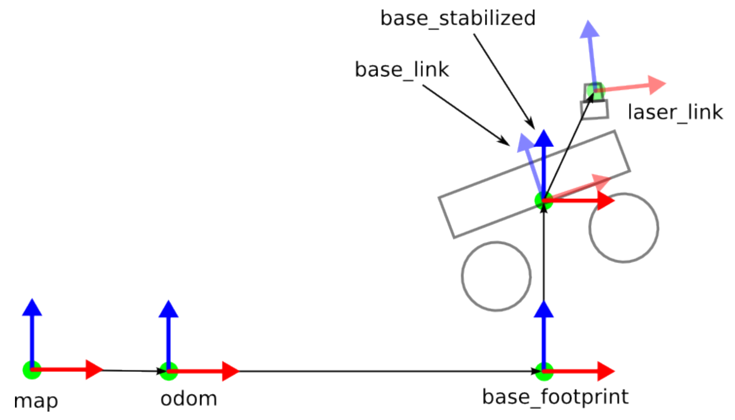


Figure 4.1: standard frames for mobile robots in the ROS environment

The procedure for the environment mapping is the following one:

1. The first thing is set the on-board computer for work with the laser sensor Rplidar A3, configuring the suitable kernel codes and allowing the communication between the two devices. The sensor is connected to the Jetosn Nano via USB cable.

2. So the ROS Rplidar A3 launch file is run (which is a ROS tool for easily launching multiple nodes [28]) in order to make the sensor work and the computer elaborate the received messages.

3. Then the Hector SLAM launch file is modified for the mapping phase. In particular hector_mapping package is updated for a suitable mapping phase:

    - During the mapping we don't need *odom* frame so the package parameters are set in order to use only the *base_link* one.
    - We use a ROS static transform publisher for statically set the transformation between the *base_link* frame and the *base_link* one.

    The other parameters were already correctly tuned for a good and precise mapping. The software will return the transformation between the *map* and the *base_link* during the mapping phase as the robot localization inside the map

4. At this point the software is ready for being run. Once the launch file is run the hector_mapping algorithm is activated and we can visualize on the RViz program its results. In particular we can observe the laser point cloud, the computed map and the robot position, represented by the *base_link* frame.

5. Now the robot has to move and discover every part of the environment. This part is done by manually moving the robot, since it is the most accurate and precise way in which we can obtain a very precise and detailed map. The robot has to be moved slowly for better results.

6. Once the entire location has been explored and the map is complete, we can run the map_saver ROS node in order to save the obtained map. The map is saved into two files PGM image file with the occupancy grid map and a YAML file containing the map information as the resolution, the origin position and other useful information.

Are shown now some results of the previously explained mapping procedure. On the left is shown the real environment while on the right the obtained occupancy grid map is shown.

4.2 Hector SLAM pose used as virtual odometry

After the mapping phase, the localization of the robot in the obtained map was the task to achieve. Before implementing the AMCL robot localization, which returns the localization with discrete jumps. We have to find a suitable odometry localization that gives us the robot estimated pose in a continuous fashion (also between the AMCL time jumps).

The wheel encoders of the robot motors returned data difficult to interpret, and so their odometry wasn't reliable. This leads us to think about a different source of odometry. So the idea was to exploit the Hector SLAM estimated robot pose, obtained thanks to a scan matching algorithm, as virtual laser odometry. This localization is returned at the same laser scan frequency, so an enough high rate for consider it almost continuous. Also literature confirms that using this virtual odometry can lead to better results than the normal odometry for the mapping phase [15], this makes us think that this method could be used for the localization as well.

So, normally the Hector SLAM software publish the robot pose expressed as the *base_link* frame pose with respect to the *map* one. But in the localization logic we need that the odometry gives us the transformation of the *base_link* with respect to the *odom* one. Then the AMCL correct such localization computing the robot pose (*base_link* pose) in the *map* one, but publishing only the transformation between the *odom* and the *map* knowing the odometry published transformation and taking it into account for have the final pose of the *base_link* with respect to the *map* exactly as he has computed (correcting the odometry drift). The presented functioning is shown in Figure 4.2.

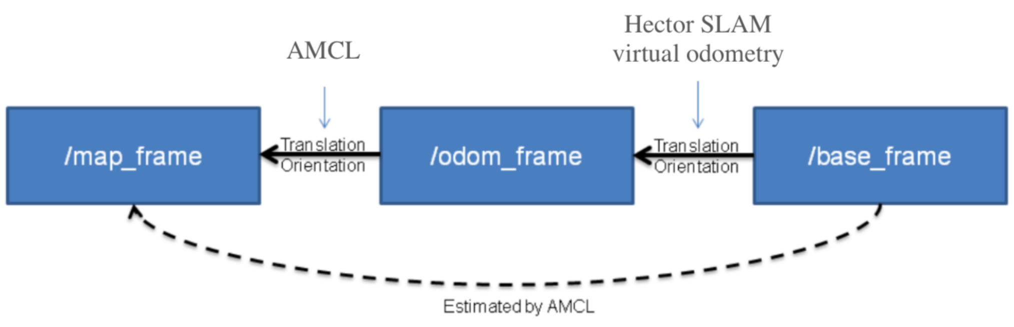


Figure 4.2: Frames and transformations published during the AMCL localization using Hector SLAM as virtual odometry.

In order to make the Hector SLAM publish the wanted transformation its mapping launch file was suitably modified just changing the name of the parameter map frame to *odom* while both the odom and base frames were called *base_linl* so the software publish the computed pose from the *base_link* to the *odom* one as wanted. Also the RViz configuration file has been modified in order to have a RViz representation of the ROS topics suitable for the navigation, showing the right frames and the right map (not the one computed by the Hector SLAM but the uploaded one from the express node for the localization, as explained in the next section). Also the static transformation from the *base_link* to the *laser* one is published in that modified launch file, publishing the real measured distance between the center of the laser sensor (in which its measurements are expressed) and the center of wheels axis (where the *base_link* frame is attached).

4.3 AMCL implementation in the obtained map

The final step for the robot localization consisted in the AMCL implementation. Before that the correct previously obtained map has to be published on a suitable topic form which the localization algorithm can read it and use it. In order to do this the standard map_server package by Brian Gerkey and Tony Pratkanis was modified.
the map server launch file was modified in order to read the map from the PGM file and its information form the corresponding YAML file, creating a map message fixing the origin of the *map* frame in the middle of the PGM file, reading those

coordinates from the YAML file. That middle point corresponds to the starting position of the robot during the mapping phase, so from now on, that would be our fixed *map* frame. The map message is then published by the map_server node on a dedicated topic, the *map_loaded* topic.

Thus, being the environment map available, the AMCL algorithm can be run. In particular we decided to use the AMCL package from the navigation stack by Brian P. Gerkey. The software was adapted for our project making it read the map on which run the localization from the expressly created *map_loaded* topic. The algorithm read from the *laser* frame the laser data and use them for compute the robot pose using a particle filter on the map, more details on its functioning are exposed in the dedicated section in the Chapter 2
Then running different tests the AMCL launch file parameters were suitably tuned in order to have the best possible robot localization. In particular the laser sensors specifications setting the 'laser_max_range' to 25 m. The 'use_map_topic' was set to true, in order to read the map from the wanted topic and also the 'first_map_only' was enabled, making the software read the map only once (being a static map). The 'selective_resampling' was set to true in order to reduce the resampling rate when not needed and help avoid particle deprivation according to [29]. Then the number of particles for the filter were set from a minimum of 500 to a maximum of 5000. The KLD sampling technique was enabled, in order to make the filter adaptive, and the parameters were set to 'recovery_alpha_slow' = 0.001 and 'recovery_alpha_fast' = 0.1 (as suggested by the author). Anyway during the motion tests, sometimes this parameters were set to zero, since we noticed that they leads to unwanted recover of the position, even if when it was correct (in the small maps in which we worked the recover wasn't so necessary). In the end the most important parameters that were tuned making the AMCL as precise as possible and also fast enough to follow the robot during the motion were the 'update_min_d' and 'update_min_a' which were the parameters that set the required minimum translational (d) and rotational (a) movements before performing a filter update. setting their value high lead to a too big gap in time between two consecutive pose updates decreasing the localization performance. On the other hand too low values make the computer computation too heavy. After different tests the best values for have a good initial localization and then a suitable quickness in the filter update in order to correctly follow the robot during the motion, were: 'update_min_d' = 0.01 (m) and 'update_min_a' = 0.015 (rad).

Finally a launch file was created which ran at the same time the Rplidar launch file for activate the LiDAR sensor, the Hector SLAM modified launch file for the virtual odometry and the RViz localization, the modified map_server file for upload the map and the AMCL launch file computing the robot localizaiton.

In a first moment the robot is placed in the origin of the *map* frame, then mving a little the robot the AMCL undertsand where the robot is in the map and moving the *odom* frame with respect to the *map* one it locate the robot in its real position. The results can be seen in Figure 4.3. In the other Figure 4.4 is instead reported the resulting frames tree (from the tf package [30]) obtained from the developed localization logic (the *scan_matcher* frame published by the Hector SLAM software since it requires it for the computation).

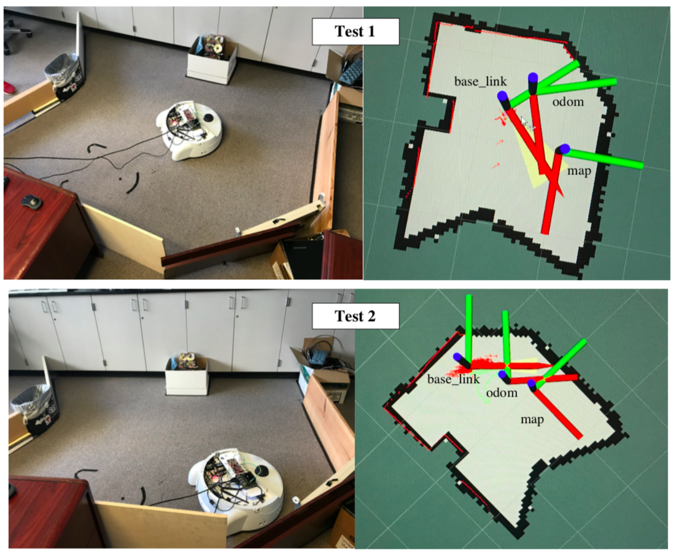


Figure 4.3: Two different robot localization tests.

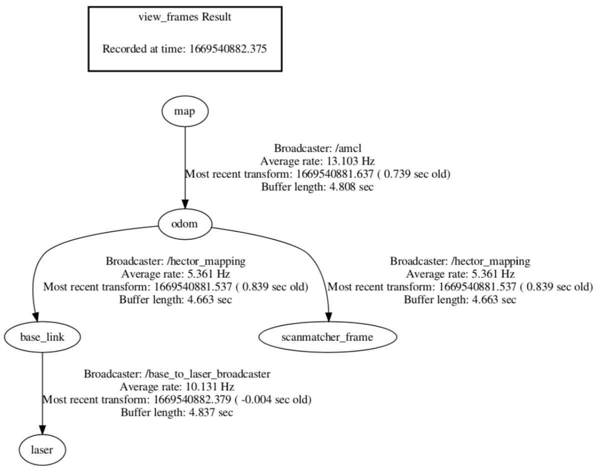

view_frames Result
Recorded at time: 1669540882.375
map
Broadcaster: /amcl
Average rate: 13.103 Hz
Most recent transform: 1669540881.637 (0.739 sec old)
Buffer length: 4.808 sec
odom
Broadcaster: /hector_mapping
Average rate: 5.361 Hz
Most recent transform: 1669540881.537 (0.839 sec old)
Buffer length: 4.663 sec
Broadcaster: /hector_mapping
Average rate: 5.361 Hz
Most recent transform: 1669540881.537 (0.839 sec old)
Buffer length: 4.663 sec
base_link
scanmatcher_frame
Broadcaster: /base_to_laser_broadcaster
Average rate: 10.131 Hz
Most recent transform: 1669540882.379 (-0.004 sec old)
Buffer length: 4.837 sec
laser

Figure 4.4: Frames tree in ROS during the robot localization.

Chapter 5

Path Planning and Control Logic

Once the robot knows the environment map and is able to place itself in such a map, the final steps for making it able to reach a desired location are the ability to compute an obstacle free path ending in the goal position and execute it point by point. This final robot development chapter deals with these aspects, in particular with a first map processing for a safe robot navigation through an obstacle boundary increase. Then the A* algorithm use for path planning is exposed and so the subsequent reference frames alignment to express the path points in the same frame of the map and robot localization coordinates. A section is then focused on the path tracking development on the on-board computer. Finally, the negative feedback navigation control logic for reaching the single path points is discussed.

5.1 Map processing through boundary increase

The first thing done before implementing the path planning was find a way for take into account the robot size during the path creation around the obstacles. The solution was found in a map processing increasing the obstacles boundaries. The robot size was then taken into account enhancing the object sizes in order to avoid a planned path too close to obstacles, leading to unwanted crushes.

The map image was an occupancy grid one, which is an image divided in cells and each one of them has an assigned number that indicates the likelihood to contain an object. The number (*occupancy*) is usually from 0 (0% likely to be occupied) to 100 (100% likely to be occupied), unknown areas are marked with a -1 number (usually only 0, 100 and -1 are used). Usually the occupied cell is represented by a black color, the free cell with white and the unknown areas in grey. Our map is saved as a Portable Gray Map (PGM) image, so every cell has assigned a byte for store it's occupancy value, from 0 (black representation) to 255 (white representation), this values x can be transformed into the occupancy probability p (slightly different) using the formula

$$p = \frac{255 - x}{255} \tag{5.1}$$

Thus x = 0 (black) means p = 1 while x = 255 (white) means p = 0. Then the occupancy probability is usually interpreted in the trinary way, assigning to the *occupancy* number only three values: 0, 100 and -1 in the following fashion:

- $occupancy = 100$ if $p > occupied_threshold$, the cell is occupied.
- $occupancy = 0$ if $p < free_threshold$, the cell is free.
- $occupancy = -1$ otherwise, the cell occupancy is unknown.

The two thresholds are suitably selected or given by the map generator.

The PGM processing was implemented through a function inside the path planning file. In this code logic we doesn't care about the unknown map parts, we want our robot to navigate only in the known environment, so only one threshold is used to distinguish between occupied or free cells. This threshold value was chosen very close to the occupancy null probability $p = 0$ that in byte stored value corresponds to the 255 x value, for safety reasons. So the threshold was set to $p = 0.98$ which in the byte value corresponds to $x = 250$.
The boundary increase function is designed in order to scan the entire map, point by point, and, if an obstacle is encountered (cell value $x < 250$, the threshold) and the cell is not yet marked as boundary, then for B rows up and down and for B columns on the left and on the right sides of that cell, to the encountered cells, if free, a $x = 100$ value is assigned (this value is considered obstacle by the logic and identify the boundary with its own grey color). The B value is the Boundary size value, passed to the function as a parameter. This parameter is very important for taking into account the robot size.
The B value take into account two main robot dimension, the robot lateral width from the point in which the *base_link* frame is attached (23 cm), important for not hit obstacles on the side, and the distance from the same machine point and the front part of the robot (38 cm), to avoid frontal crushes. These values transformed in pixels (that are the map cells), according to the map resolution, which can be found in the map YAML file, 0.05 m/px for each obtained map, became $lateral\,size \approx 4.6px$ and $front\,size \approx 7.6px$. The B value has thus been set to 5 px, since using 8px in the small environments in which we made our robot navigate

was too conservative and the path planning wasn't able to find a path. Anyway this value was enough high to never lead to unwanted crushes due to a too close movement with respect to the obstacles.

A map processed by the boundary increase function can be seen in Figure 5.1.

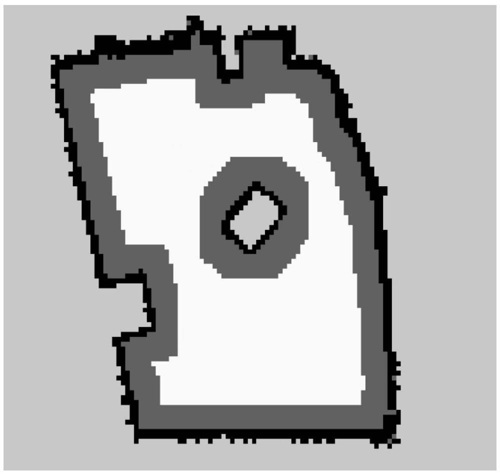


Figure 5.1: Map with added boundaries in dark grey.

5.2 A* search algorithm for path planning

So with a suitably processed map, the path planning could be run. The chosen software for this scope was the A* path search algorithm, implemented by previous years company workers. The A* algorithm is an heuristic graph search for state space algorithm. It search the best path according to a cost function evaluated for each path points composed by the distance traveled from the start to the actual point summed with an heuristic function taking into account the remaining path till the destination. In our case a simple euclidean distance was chosen as heuristic function, which beyond being simple leads also to good path results. More details can be found in the dedicated section in Chapter 2.

So the path planning node read the map from the PGM and YAML files and the start and end points from a dedicated topic. The start and end point message is generated another node used for select the wanted final destination of the motion, a future improvement of this node could be to receive the destination coordinates directly from a user interface, and chose as starting point the actual robot position, instead of a pre-defined one. So once the path planning node has both map and start and end points it run the A* algorithm searching for the shortest feasible path. If the path is found then the node publish two different things, a BMP image of the map containing the path and its trial points and it publishes over a topic the entire computed path. The resulting map containing the path can be seen in Figure 5.2. The red line is the path, the green pixels the different tried paths, the

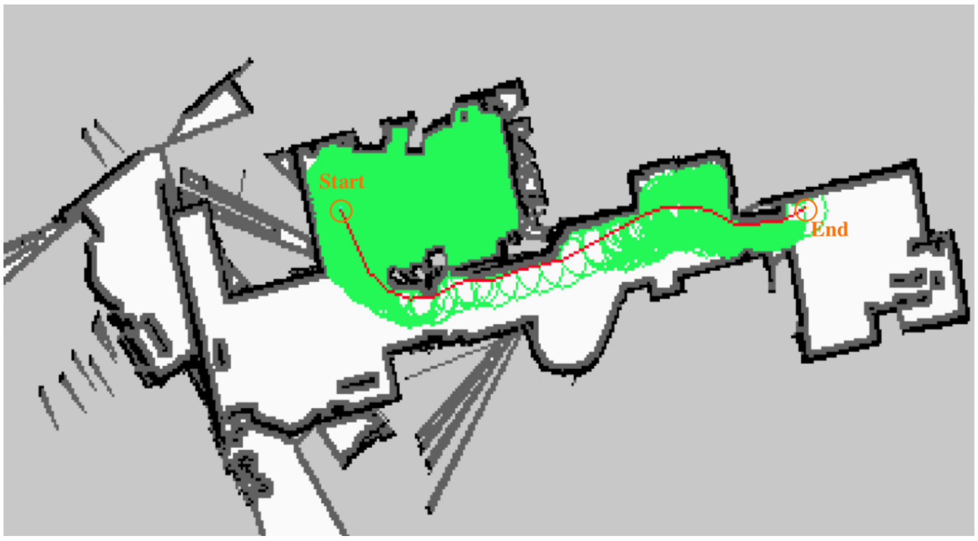


Figure 5.2: Map with computed path using A* algorithm.

The problem now is that the algorithm compute and return the path points in pixels and with respect to the image frame, thus a frame alignement is required.

5.3 Reference frames alignment

Since the computed path was expressed in pixels with respect to the high left corner of the image, a suitable coordinate and size transformation was required in order to have a useful path, expressed in meters and with respect to the *map* frame.
This was done developing a suitable function that transformed the points from the virtual image frame to the *map* one. The frame adapter function takes as inputs different values:

- X_0, Y_0, ψ - The 2-D pose of the lower-left pixel in the map expressed in meters and radiants (Ψ is the z-axis rotation, usually 0), this values are automatically extracted from the YAML file of the map. These parameters are used for obtain the translation between the image and map frames.
- θ - The z-axis angle rotation from the *image* frame to the *map* one in radiants.
- R - The map resolution $[m/px]$.

- X_S and Y_S - The x and y sizes of the map, in px (for our images are both 2048 px).
- x_p, y_p, z_p - The point 3D coordinates to be transformed from the *image* to the *map* one, expressed in pixels (obviously in our study $z - p$ is always equal to zero).

Thus the function implement an algebraic point coordinate transformation from the *image* to the *map* frames, in parallel to a dimension transformation from pixels to meters. The two frames are located in the image as shown in Figure 5.3.

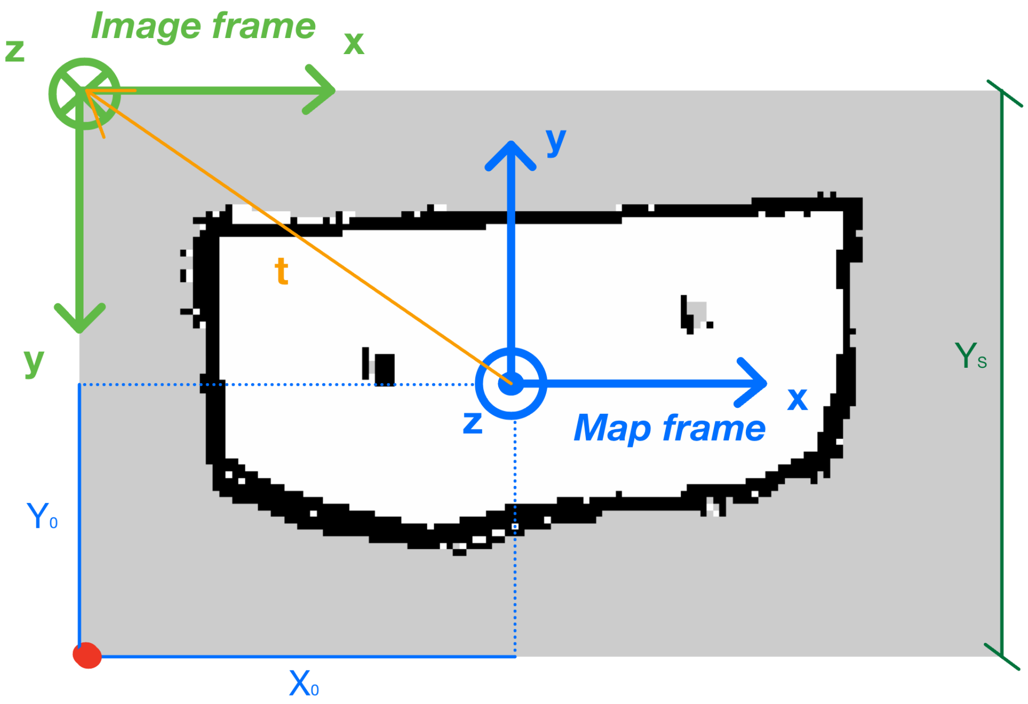


Figure 5.3: Image and Map frames and coordinates.

The equation to be used on order to transform the coordinates form the two frames is the following one:

$$\overline{x}_p^M = R_I^M \overline{x}_p^I + \overline{t}_{MI}^M \tag{5.2}$$

Where R_I^M and $\overline{t_{MI}^M}$ represent the rotational matrix and the translation vector that transforms the *map* to the *image* frame. While $\overline{x}_p^I = \{x_p^I, y_p^I, z_p^I\}$ and $\overline{x}_p^M = \{x_p^M, y_p^M, z_p^M\}$ are the point position coordinates expressed in the two frames M for the *map* one and I for the *image* one. The transformation between the two frames is composed by a rotation of π radiants around the x axis and a rigid translation.

Thus the resulting 5.2 equation became:

$$\overline{x}_p^M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} R\overline{x}_p^I + \begin{Bmatrix} X_0 \\ Y_0 + RY_S \\ 0 \end{Bmatrix} \tag{5.3}$$

In the equation is already implemented the meters from pixels unit transformation. This the path planning node after transforming each path point with the designed function, it publishes them on the *path_planning/Path_std* topic (as Path navigation message type).

5.4 Path Tracking logic

At this point we have to design a software that knowing the path and the robot localization decides which is the actual, closer path point that the robot has to reach in order to move from the start point till the destination. This was implemented on tracking node which read the robot position from the *slam_out_pose* topic and the path from the *path_planning/Path_std* topic. The Hector SLAM pose is used, since it is updated with a enough high frequency and it is anyway corrected by the AMCL. The only thing is that the SLAM pose is expressed in the *odom* frame, while the path it is correctly expressed in the *map* one. Thus, thanks to the tf package the localization is transformed and expressed in the *map* frame.

The tracking node, knowing the robot localization measure its distance from the first path point, if the value is greater than a specified Threshold (T) distance, the node publish the first path point coordinates on a specific topic as actual goal point. From that topic the path point will be then sent via I2C to the MCU. Instead, if the computed distance is minor than the T value, so the path point is considered reached and the algorithm moves to the next path point as actual goal point. This procedure is repeated till the reaching of the final path point.
This node is essential for make the robot follow the computed path. Tuning the T value the designer can also decide how much the path has to be closely followed, reducing the motion quickness and smoothness. The effects of the variation of this parameter on the autonomous navigation will be tested and discussed in the Chapter 6.

5.5 MCU data sending

During the robot motion, it is the MCU that implement the navigation control logic. In order to do that it requires the robot localization and the actual goal path point (selected from the tracking node). Both the data are sent to it via the

I2C bus, by two dedicated nodes. In particular the robot localization is the same used by the tracking node, the Hector SLAM one, suitably transformed in the *map* frame, always because an almost continuous robot pose is necessary for have a good motion control.
Once the first robot localization has been sent to the MCU a signal is sent to a specific topic, the same happens for the first path point sending. The i2c_enable node read those signals and send, always via I2C the enable message containing two bits, the Localization Ready Bit (LRB) and the Path Ready Bit (PRB), which are true only when both localization and path point have been received. These two signals are then used by the MCU in order to activate the control logic, and enable the motor spinning. Once the last path point is reached, the PRB is set to 0, thus stopping the robot motion.

5.6 Robot autonomous navigation control logic

the final step for make the robot able to move autonomously, was the motion control logic design, in order to make the robot able to reach the single path points one by one.
We decided to use a static gain controller with a negative feedback loop. This control technique is simple but allowed us to obtain a in a good way our goals, so we decided to not make a more complex controller for now. The control logic can be conceptually represented as in Figure 5.4. Where the $G(s)$ would be the robotic platform model, $H(s)$ is considered to be unitary in our case, K is the static gain. While the signals would be vectors, $r(t)$ would be the desired path point coordinates, $y(t)$ the system output, the robot localization coordinates, $e(t)$ the error value, to be set to zero, that would be the distance between the robot and the goal point together with the angle difference between the robot orientation and the angle connecting robot and goal point, finally $u(t)$ would be the control input to the machine that in our case would be the PWM signal used for drive the motors.

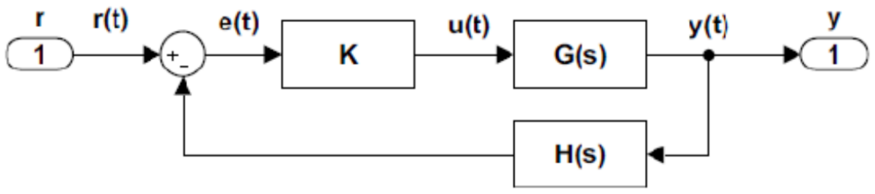


Figure 5.4: Static gain negative feedback control system conceptual scheme.

Thus, the MCU task for implement such a controller consisted in receiving the desired path point and the robot localization form the Jetson Nano. Then it had to compute the euclidean distance between the robot pose and the path point (5.4) together with the angle difference between the robot orientation in the *map* frame θ_r and the angle drawn by the vector connecting the robot to the goal point θ_n, expressed by the formula 5.5.

$$d_e = \sqrt{(x_{pp} - x_r)^2 + (y_{pp} - y_r)^2} \tag{5.4}$$

$$\begin{gathered} \theta_n = atan^{-1}\left(\frac{y_{pp} - y_r}{x_{pp} - x_r}\right) \\ \theta_e = \theta_n - \theta_r \end{gathered} \tag{5.5}$$

Where the subscript 'pp' means path point, while 'r' means robot. This distance and angular errors were then multiplied by the control gains leading to the linear and rotational robot speed control values, as shown in 5.6.

$$\begin{gathered} v = K_v \cdot d_e \\ \omega = K_\omega \cdot \theta_e \end{gathered} \tag{5.6}$$

After that the two computed speeds had to be converted in wheel rotational speeds. That was done exploiting the differential drive robot kinematic model, shown in Figure 5.5, for which:

$$v = \frac{r(\omega_r + \omega_l)}{2} \omega = \frac{r(\omega_r - \omega_l)}{d} \tag{5.7}$$

Where r is the wheel radius (0.08 m in our case) and d is the wheels axis length (0.39 in our case).

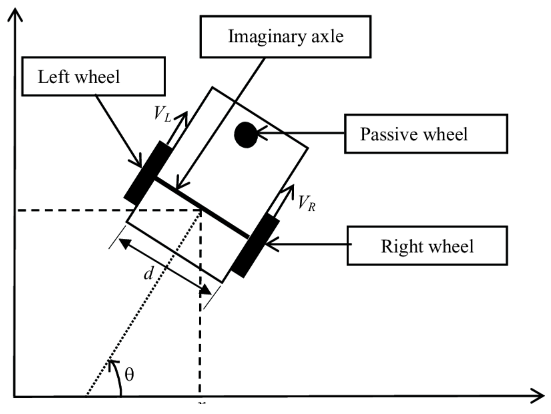


Figure 5.5: Differential drive robot scheme.

From 5.7 which we can obtain

$$\begin{aligned} \omega_r &= \frac{1}{r}\left(v + \frac{\omega\, d}{2}\right) \\ \omega_l &= \frac{1}{r}\left(v - \frac{\omega\, d}{2}\right) \end{aligned} \tag{5.8}$$

The final step consisted in transforming the wheel rotational speeds into suitable Duty Cycle (DC) values for the PWM signal generation.
In order to have a suitable relation between these two parameters different test were run, making the robot move straight, with the same DC value for both the wheels, exploiting the keyborad control of the machine. Thus different DCs value were tested. For each one of these motions the robot was required to drive for one second and for each navigation the travelled distance was measured. The robot mean speed was computed among the tests with the same DC values and a direct correlation was found between the linear robot speed and the DC values expressed by the formula 5.9 and shown in Figure 5.6, it has to be noticed that this relation is valid only in the tested range, from 15% to 50% DC, for lower values the robot almost doesn't move.

$$\begin{aligned} DC_{r\,and\,l} &= m \cdot v + q \\ m &= 37.299 \\ q &= 9.855 \end{aligned} \tag{5.9}$$

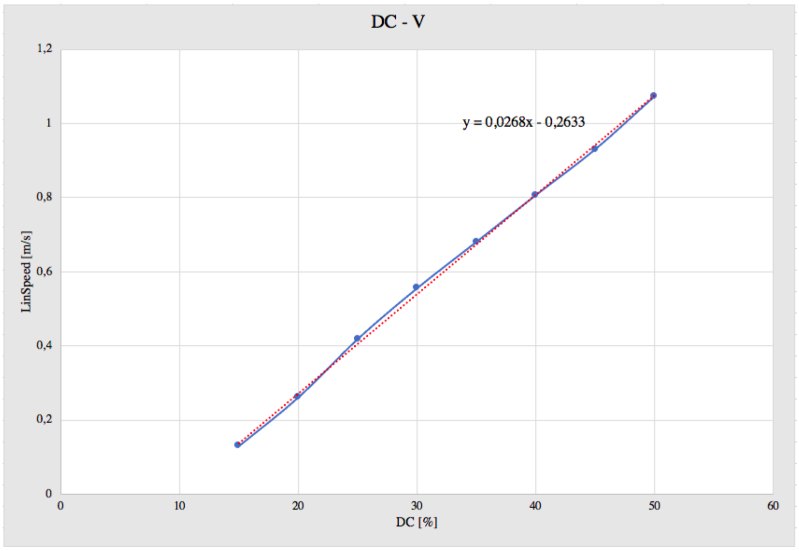


Figure 5.6: DC versus linear speed plot.

From this relation we can, with approximation, deduct that: the two wheel run at the same speed, so $\omega_r = \omega_l$, thus from 5.7 we can deduct that $v = v_l = v_r = \omega_r r = \omega_l r$ thus leads to the following equation expressing the direct correlation between single DC value and related wheel speed:

$$DC_{r\,or\,l} = m \cdot r \cdot \omega_{r\,or\,l} + q \tag{5.10}$$

This relation is not a strict real model, since the rotational speed is not taken into account, better test would require also its measurement, and its use for the correlation deduction. Anyway the obtained 5.10 has proven in different tests to be good enough for our purposes.
So the MCU is able to compute the suitable DCs values to send to the motor drivers in order to act the motion.
We have to make two final considerations on this logic, the first thing is that only suitably tuning the two speed gains the system is stable and converge to the desired points (this is done in a system simulation, as discussed after) and the fact that the logic has a small bad behavior. The fact that for some angular orientations of the robot with respect to the desired path point, the robot chose the longest steering direction in order to reach the goal position. This was due to the fact that for some θ_n values the control logic returned a control rotational speed ω opposite to the fastest turning direction for reach the path point. So we noticed that by default we used the θ_r and θ_n values in the $[-\pi; \pi]$ range, while in those specific cases using only positive angles in the $[0; 2\pi]$ range, would return a robot ω speed according to the shortest possible path. Thus the resulting code that identified and corrected such wrong angle cases was the following one:

```
    if (((theta_n < 0) && (theta_r > PI + theta_n)) || ((theta_n > 0)
    && (theta_r < -PI + theta_n)))
    {
        if (theta_n < 0)
            theta_n = theta_n + 2*PI;
        if (theta_r < 0)
            theta_r = theta_r + 2*PI;
    }
```

After the design part, the control logic was simulated on Matlab. We made the assumption that the robot motion between each sample time was constant, so no acceleration was present in the model. This was done because we had no information of the robotic platform dynamic model and its computation was beyond the purpose of this work. Anyway such a simplified model was good enough for have a general idea of the control logic functioning, and was used for tune the K_v and K_ω gains, in order to have a stable and smooth motion of the robot till the desired path point. These gain values were then also proved to be correctly tuned in the

Gain	Value
K_v	0.3
K_ω	0.55

Table 5.1: Control gain tuned values

real autonomous motion tests, discussed in the following Chapter. Their value is reported in the following Table 5.1. The simulation plot of the robot motion following the same planned path used in the next chpter tests is reported in Figure 5.7 (the T value was set to 0.25 m).

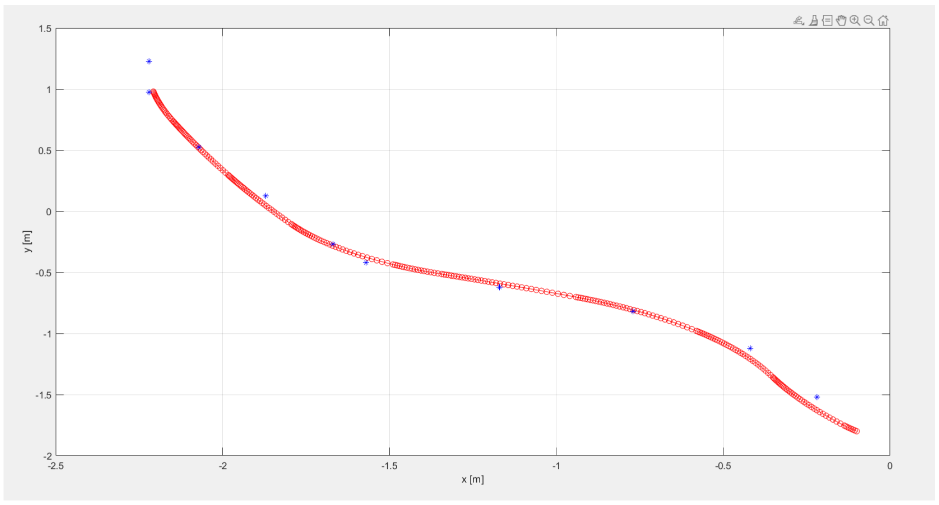


Figure 5.7: Simulated robot motion using Simulink.

The control logic was developed as Matlab Simulink model and then deployed in the MCU device thanks to the Code Composer Studio program.

Finally the tests for check the autonomous motion control are reported in the next Chapter 6. Unfortunately, due to a MCU too slow computational speed the logic wasn't directly tested on the microcontroller, which wasn't able to work in real-time. Instead, it was moved to the on-board computer. anyway the tests returned good autonomous motion results which can be used to validate also the MCU navigation control implementation which was the exact same logic that returned the exact same DCs values. More details are presented in the next chapter.

Chapter 6

Tests and Results

In this chapter the InnoTech robotic machine used during the thesis development is shown in its different hardware parts and features. Then a section focuses on the tests made to examine and validate the different thesis developments on the robotic system. In the last part the test data are elaborated and analyzed and the results are drawn.

6.1 Robot hardware

The robotic machine used in the entire thesis work, was a differential drive robot built by Innotech Systems, as a service robot for different purposes such as escorting people in the hospitals or airports and delivering food in university campuses. It is shown in Figure 6.1 with respect to Figure 1.1 here the robot is free from the upper part, used only to support a screen for human interaction and a basket for delivering objects. Only the platform with all the required devices and sensors for the motion was used during the work.

The different devices composing the robot were:

- **The robot chassis and shell** - The robot inner chassis and the shell were 3D-printed, the former part with a more stiff material, to bear the entire robot weight.

- **The on-board computer** - The Nvidia Jetson Nano. With the following specifications. GPU: NVIDIA Maxwell™ architecture with 128 core NVIDIA CUDA®, CPU: Quad-core processor ARM® Cortex®-A57 MPCore, memory: LPDDR4 4 GB 64-bit.

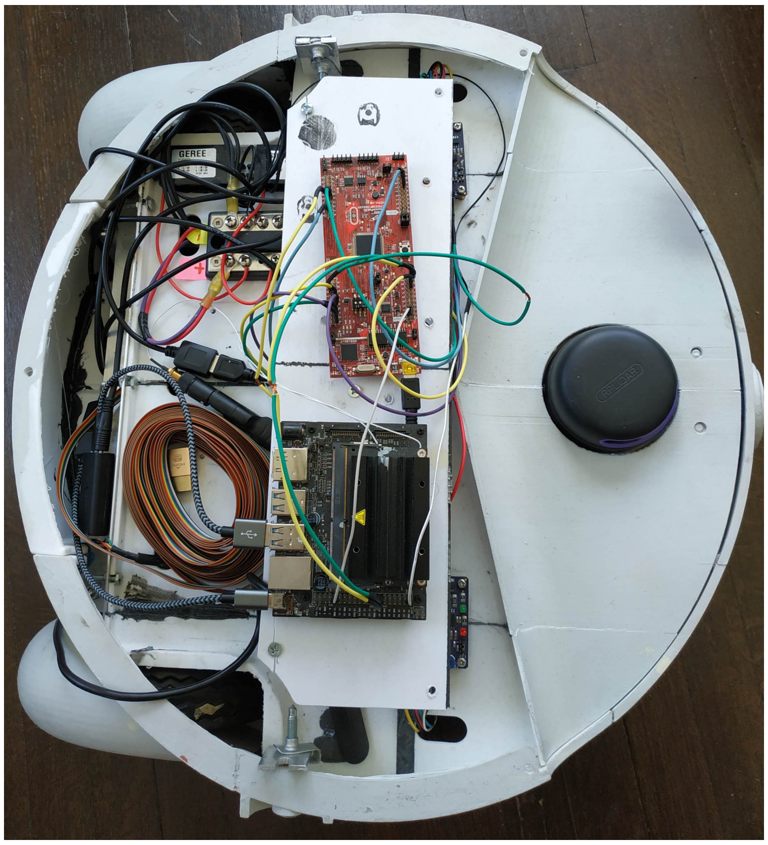


Figure 6.1: Top view of the InnoTech robotic platform

- **The MCU** - The Texas Instruments C2000 Piccolo MCU in the LAUNCHXL-F28069M development kit. Its specifications are now exposed. CPU: C28x, CLA with frequency 90 MHz, Flash memory 256 KB.
- **The two motor drivers** - the devices are MDC151-024031 Series 24V, 3A Brushless DC Controller.
- **The LiDAR sensor** - The Rplidar A3 from Slamtec. This sensor has the following specifications working in an indoor environment: a distance range from 10 m (dark object) to 25 m (white object), minimum range of 0.2 m, a sample rate of 16000 times per second, a typical scan rate value of 15 Hz (adjustable between 10 Hz - 20 Hz), a range resolution $\leq$ 1% of the range ($\leq$ 12 m) and $\leq$ 2% of the range (12 m $\sim$ 25 m) and an accuracy of 1% of the range ($\leq$ 3 m), 2% of the range (3 - 5 m) and 2.5% of the range (5 - 25 m).
- **The battery** - A LiFePO4 Lithium-IOn Chemistry battery from Haidi, the Lithium Deep Cycle Adventure model (12.8V,15Ah).

- **A power distribution device** - Two metal plates form which different cables take energy to the different devices. Also two DC/DC converters from 12V to 5V-3A maximum were used in order to have a suitable voltage for the devices.
- **Wheels and motors** - In particular the motors were two BLWSG23 Series - Brushless DC Spur Gearmotors.

The different robot devices are connected as shown in Figure 6.2. In the image are also distinguished with colors the different connections channels.

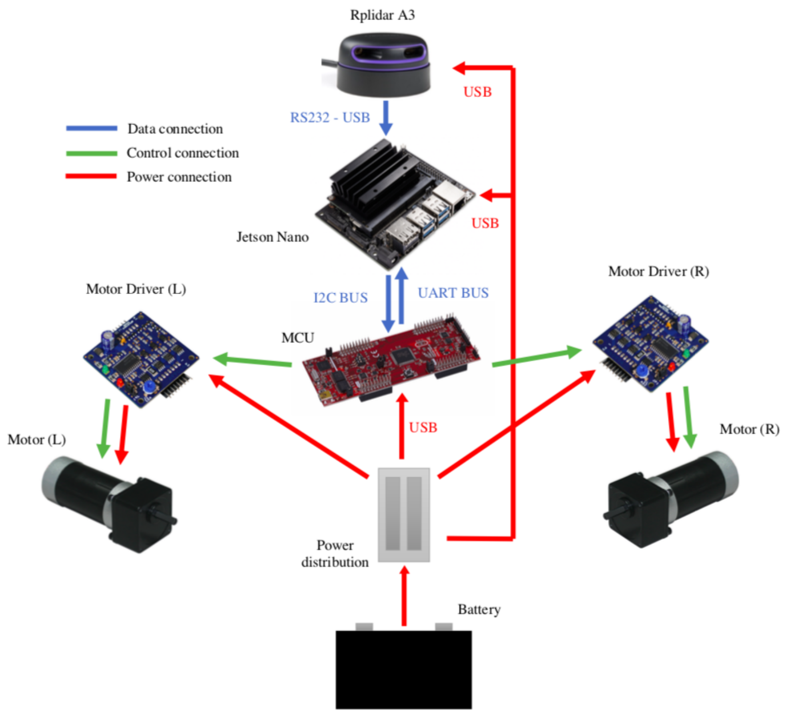


Figure 6.2: Robot devices scheme with the connections.

6.2 Project final test

At the end of the project a suitable final test was run in order to verify the different parts developed during the work.
The most obvious and at the same time the most validating final test is actually the ultimate thesis goal itself. Since the aim of this work was to make the vehicle move autonomously, and every different developed part was functional to this goal, make the robot reach a desired location in a known environment could evaluate

the goodness of the entire project, as well as of the single subsystems, pointing out which parts of work in a good way and the ones that need to be improved.
Anyway, the single work parts exposed in the different chapters have already been individually tested once implemented on the robot, as discussed in each one of the previous work parts. This is necessary for trace the origin of some possible malfunctioning of the overall logic, being aware that the single parts individually works.

So in the final test the robot was required to move till a desired position in a known map, computing and following the shortest possible obstacle free path.
The tests also aimed at evaluating how precisely the robot followed the path and, at the same time, how fast and harmoniously it moved. This aspect was evaluated at the variation of a suitably chosen parameter, the Threshold (T) distance for the next path point reaching in the control logic software.

As previously said, it has to be noticed that due to a MCU problem, the control logic wasn't implemented on the microcontroller itself, but on the on-board computer, which directly sent the DCs for the motor wheel speeds control. So the MCU had only to drive the motors according to the received data.
The found problem could be attributable to a not enough high computational speed of the MCU since testing the different microcontroller model parts singularly as the I2C communication, or the motor driving (through direct keyboard inputs), or the control logic part for compute the required DCs; they all worked in a good fashion. But then, during the live robot motion control where all the parts have to run together at high frequency, the MCU slows down its computational speed. This can be confirmed by the fact that during the autonomous motion tests implementing the control logic on the MCU, the computer worked at a suitable speed and sent via I2C the real time robot localization, which was processed by the microcontroller (returning corrects but late motor controls) only after different seconds from the time of the sending. This was noticed thanks to the debug UART communication, which extracted the inner MCU signals live during the test. This was a clear sign of a MCU computation overload (we didn't analyze if this was due to a simple not enough computational power of the device or to an hardware problem). Different resolution techniques were tried for this problem, such as making the microcontroller model as light as possible, also replacing some complex functions with look up tables, but this didn't solve the issue. At the end we decided to move the control to the computer in order to test it in a real application and test the overall robot autonomous motion. So we renounced to test the logic directly implemented on the MCU and the developed techniques for improve the I2C communication, the outlier rejection for the localization coordinates and the UART feedback for the other message transmissions. Anyway these developments were singularly tested and all worked in a good way. In particular, with the same inputs, the MCU control logic

returned the same computer control logic outputs that correctly drove the robot till destination.
The fact the I2C didn't worked in a perfect way and required improvement techniques, could be a hypothetically a consequence of the MCU bad behavior as well.

Returning to the final test, it was developed as follows. A suitable close environment was built and mapped, exploiting the Hector SLAM mapping technique, exposed in the dedicated section 4.1. The real-world environment, together with the obtained map are shown in Figure 6.3.

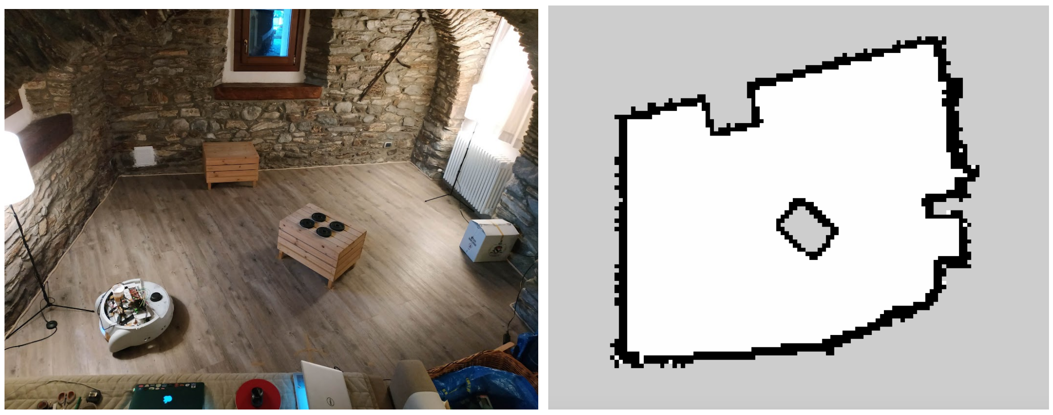


Figure 6.3: Final test map. On the left the real environment, on the right the corresponding occupancy grid map

Then a suitable goal position was decided. choose to place the robot on the bottom left corner (as shown on the right image in Figure 6.3) and the destination on the top right corner, behind the central obstacle (the start and end points for the motion can be seen in Figure ...). Thus trying to create a not too easy environment for the robot navigation, in order to test its ability in overcoming obstacles in its motion.

The rest of the work to do in order to reach the desired position was the robot task evaluated in the test. The only remaining things that we did before the motion was to tune the robot localization on the map and run all the required ROS softwares. In particular the procedure for run the test was the following one:

- Launch the localization launch file, running the following nodes: the Rplidar A3 node for activate the sensor, the Hector SLAM file modified for obtain the laser virtual odometry, the map uploader node and the AMCL software for the differential robot localization. This, among the other things, returns the robot localization in a continuous way on the *slam_out_pose* expressed in the *odom* frame and the same localization expressed in the *map* frame in the *amcl_pose* topic, updated each time the AMCL compute the robot pose. At the same time the RViz program is run and so the different frames and topic outputs are visually shown on screen, as in Figure 6.4 .

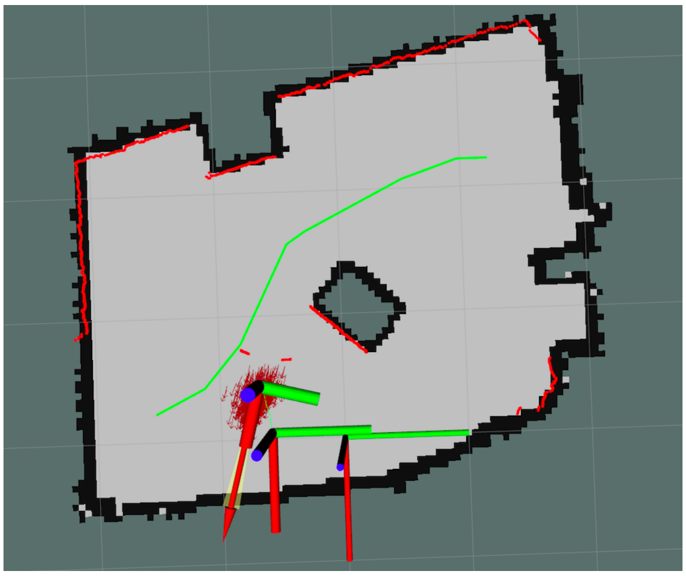


Figure 6.4: RViz display of the different topics and frames

- Launch the path planning software obtaining the path. This launch file runs the node that publishes the decided goal destination and the path planning node which computes the path from the start to the end positions in map, after the map boundary increases. The result is shown in Figure 6.5, the path is also published on a specific node for the control logic.
- Run the control logic node, which was a modification of the path tracking node. This node reads the robot localization in a continuous way from the *slam_out_pose* topic (transforming such a pose in the *map* frame) and the path points from the dedicated topic. It then computes the next path point, but also implements the control logic computing the required DCs in order to drive the robot and publish them on the *control_logic/DCs* topic.
- Then the UART communication node is run, in order to extract the received DCs values from the MCU, checking the correct reception of the sent information.
- Finally the I2C communication software is run in order sent via the I2C bus the DCs messages. From that moment the robot starts the autonomous motion till the final location.

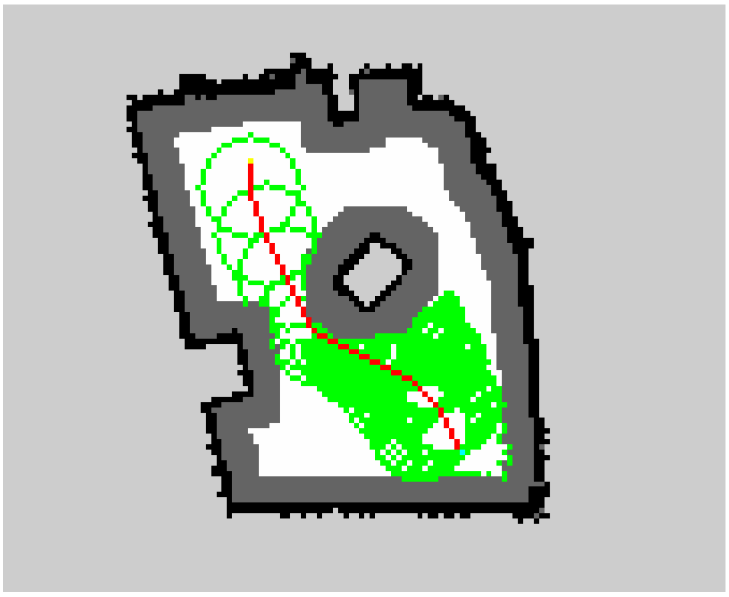


Figure 6.5: Occupancy grid map with the A* computed path

- During the motion a rosbag (which is a set of ROS tools for recording from and playing back to ROS topics [31]) is run in order to store the messages published on the *amcl_pose* topic and the *control_logic/DCs* topic from which we could extract the robot AMCL localization and the DCs values obtained from the control logic. During the test also the on-board computer screen is recorded, in order to have the available information obtained during the test also afterwards, as the data returned from the UART feedback node and the RViz representation of the robot motion.

During the tests we exploited a software for remote connection and control of the on-board computer, the NoMachine software from NoMachine.

The complete nodes and topic graph of the ROS environment is presented in Figure 6.6 . This summarizes the functioning of the computer logic.

This procedure was implemented at different times changing a suitable chosen parameter, the Threshold (T) distance for the next point reaching in the control logic. We chose to change this parameter in order to see how the precision of path tracking changed varying this value, compared with the change of the travel time and smoothness of the robot movement.
This is because we were interested in obtaining the best possible motion of the robot in terms of close following of the obstacle free path in order to avoid unwanted crushes. At the same time, we wanted to have a fast enough destination reaching

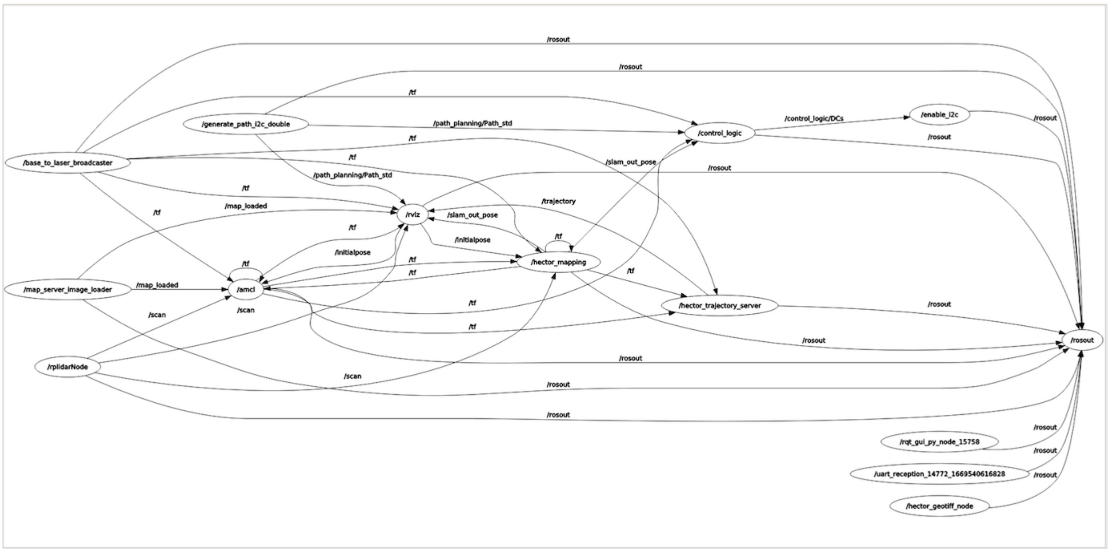


Figure 6.6: Graph showing the nodes (as ellipses) and the topics (as arrows) of the entire ROS control logic

Test name	**T value** $[m]$
T15	0.15
T25	0.25
T35	0.35
T45	0.45

Table 6.1: T values in the different tests

with a smooth robot motion to have a good service robot that is not too slow and moves harmoniously. We noticed that the main parameter that changed these behaviors was the path points distance reaching T, so by tuning such a parameter we could modify the robot motion as desired. The tests pointed out how in relation to the parameter numerical values.
The T parameter is the distance to which a path point is considered reached by robot from the control logic (or the path tracking one) its different values in the tests are reported in Table 6.1 .

After each test different data were collected, in order to draw suitable considerations from the robot different behaviors. The data collection can be summarized with the following steps:

- During the test the robot time of travel is measured, from the start of the motion till the final stop (precisely from the first DCs transmission to the MCU till the last non zero value sent, that indicates the destination reaching). Its value tells us how quickly the robot reaches its destination.

- The second thing done during the test is the visual analysis of the goodness of the motion. The goodness is evaluated in terms of smoothness of the navigation focusing on the absence of abrupt motions and stops of the driving.

- Then the collected rosbag file is run to extract the topic recorded values, in particular the AMCL data are stored in order to know how the robot localization has worked during the test.

- The robot's real path is measured by hand. Since the robot was equipped with a small mechanism, built by me, that makes a narrow powder flow on the ground under the robot during the motion, precisely the powder is released at the exact point in which the *base_link* is attached to the vehicle. The cartesian *map* frame is drawn on the environment ground, in order to easily measure some selected real path points. The measured points are the ones closer to the ideal path points (that are drawn on the ground as well), this in order to have a valid measurement of how much the robot has followed the ideal path.

After the data collection for each one of the tests, the data were elaborated in order to draw suitable observations of the robot navigation.

6.3 Results

The collected data from the different final tests were elaborated using the programs Microsoft Excel and MATLAB from MathWorks.
The elaboration mainly consisted in plotting for each test the measured path together with the AMCL one (which represents the robot localization one) and the ideal one (obtained from the path planning). The obtained plots are shown in the Figures 6.7, 6.8, 6.9, 6.10.

The plots also show the AMCL points covariance error ellipses as index of the robot uncertainty in the estimated localization (a confidence interval of 95% is used). The AMCL key path points were chosen in order to be as close as possible to the ideal path (these points also coincides with the closest points to the measured path), this allowed us to draw qualitatives considerations on the difference between each two of the three paths.

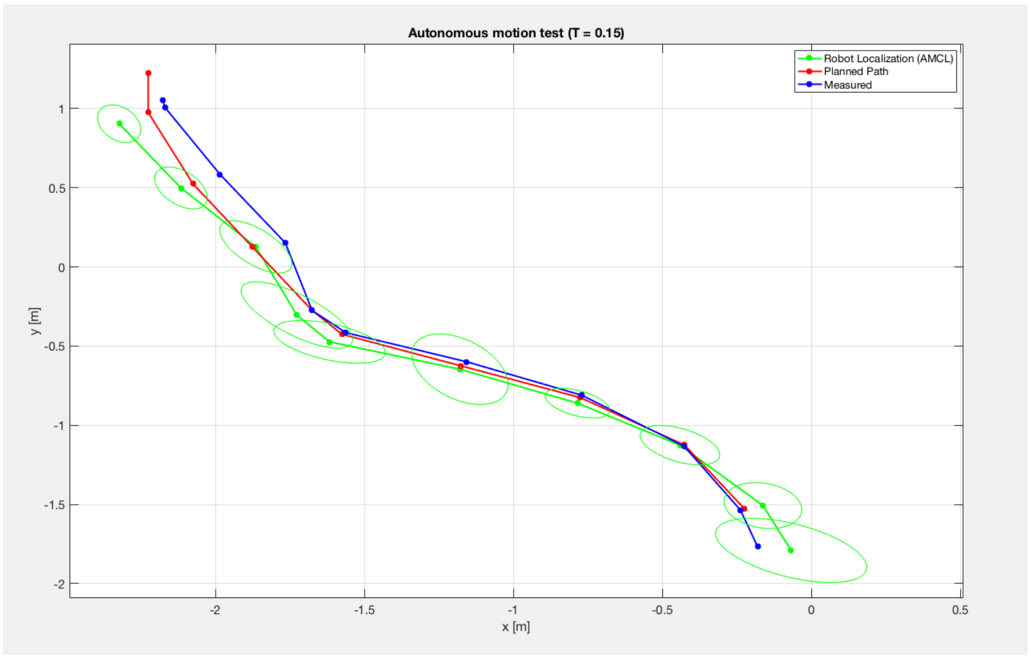


Figure 6.7: Ideal, measured and AMCL paths plot (T = 0.15 m)

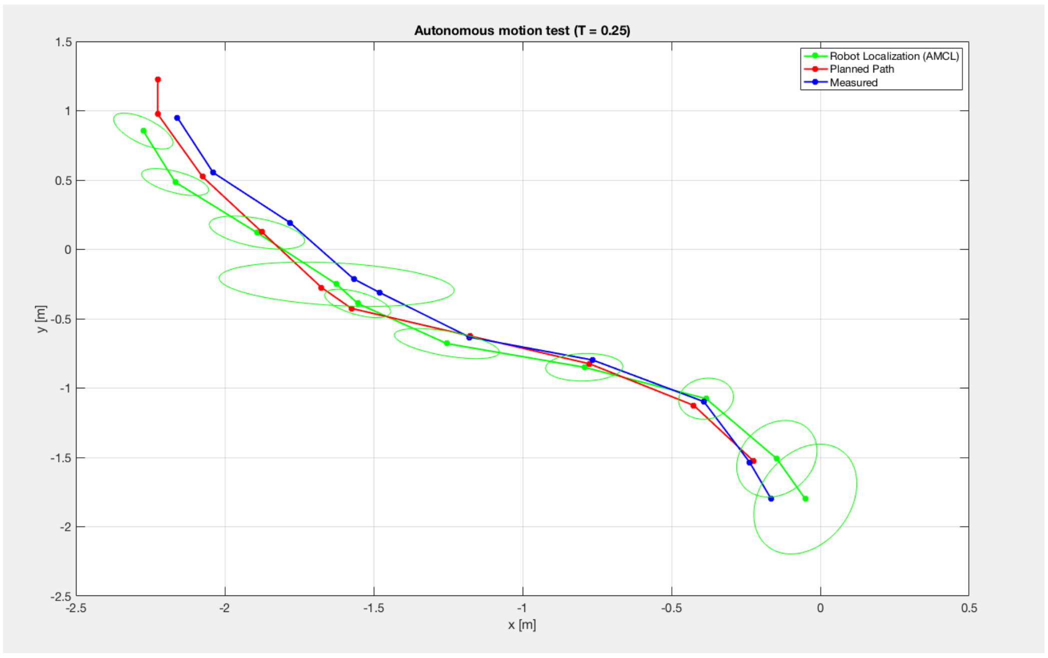


Figure 6.8: Ideal, measured and AMCL paths plot (T = 0.25 m)

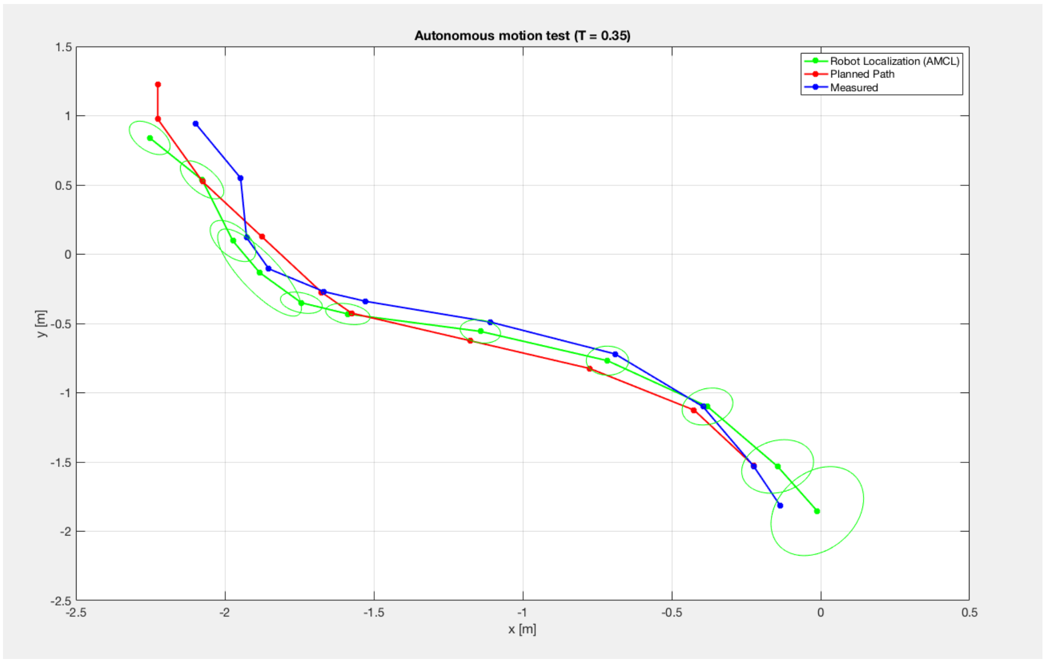


Figure 6.9: Ideal, measured and AMCL paths plot (T = 0.35 m)

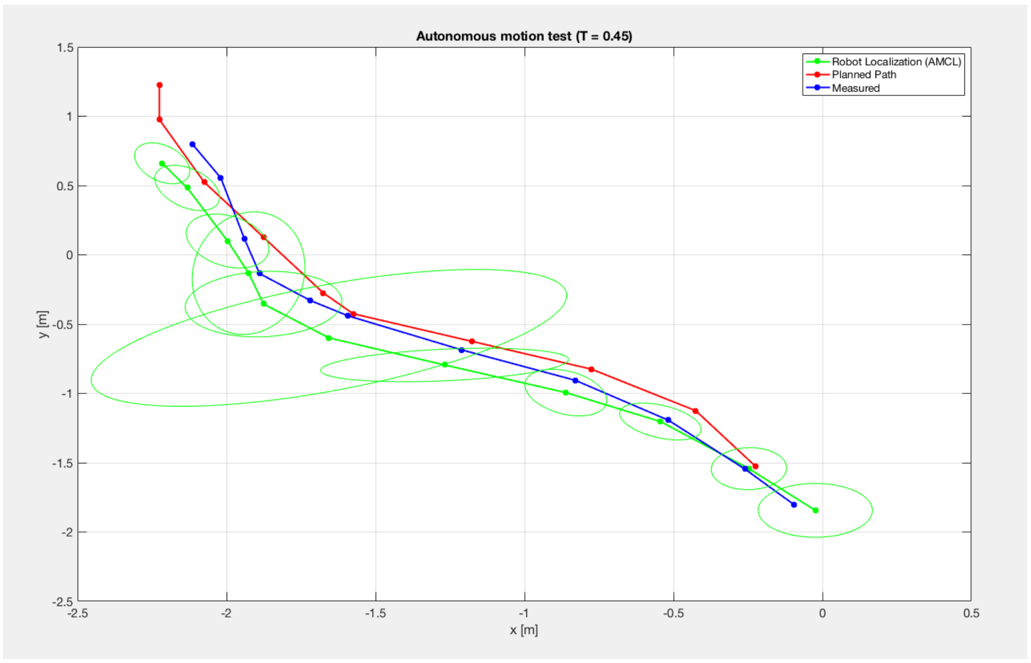


Figure 6.10: Ideal, measured and AMCL paths plot (T = 0.45 m)

The second data elaboration consisted in computing the Root Mean Square Error (RMSE) value between each couple of the three paths, considering as error the euclidean distance between the closest collected points in the two paths. The euclidean errors point by point are presented in the bar graph in Figure 6.11 while the resulting RMSE values can be seen in Table 6.2 . This computation expresses quantitatively how much the paths are close to each other.

RMSE values $[m]$				
Compared paths	**T = 0.15 m**	**T = 0.25 m**	**T = 0.35 m**	**T = 0.45 m**
Measured - Ideal	0.079	0.119	0.135	0.167
Measured - AMCL	0,116	0,110	0,115	0,123
AMCL - Ideal	0,120	0,139	0,145	0,248

Table 6.2: RMSE values comparison.

Thus is presented a list summarizing the qualitative evaluation of the motion goodness, together with the travel time for the different tests:

- **T = 0.15 m**

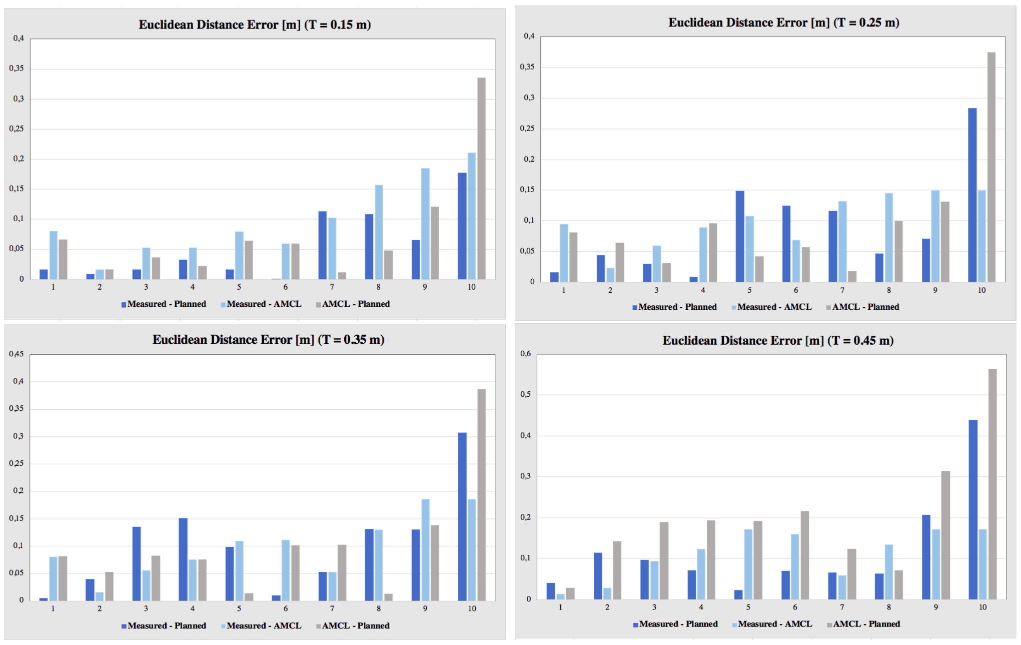


Figure 6.11: The euclidean error values between each couple of paths, point by point, in the four different tests. (Planned = Ideal path)

   - *Travel time* = 37.6 s.
   - *Motion goodness evaluation*: The robot moves very slowly, scattering at some points, adjusting its orientation many times in order to reach the single path points and sometimes also stopping the motion and restarting. It's the worst robot motion.

- **T = 0.25 m**
   - *Travel time* = 26.6 s.
   - *Motion goodness evaluation*: The robot moves quite slow, less scattering with respect then the previous experiment, but it anyway turns many times in order to reach the single path points making the motion less smooth. The speed is not very constant during the entire path.

- **T = 0.35 m**
   - *Travel time* = 19.4 s.
   - *Motion goodness evaluation*: The robot moves at a good enough speed, that is more or less constant, without big slow downs till the final stop. During the navigation the robot doesn't spend too much time correcting the orientation toward the single path points but more steering according

to the overall desired path shape. The motion can be considered smooth enough.

- **T = 0.45 m**
  - *Travel time* = 16.9 s.
  - *Motion goodness evaluation*: The robot moves fast and smoothly till the destination, the speed seems also to be quite constant during the test. It doesn't reach the single path points but just follows the general path shape. It's the best robot motion.

A final Table (6.3) summarizing the previous analysis has been built. For each one of the different tests are presented the three RMSE values (between the different paths), the travel time is presented as well and a final qualitative consideration of the goodness of the robot motion.

Results summary					
T value $[m]$	**RMSE** M.-I. $[m]$	**RMSE** M.-A. $[m]$	**RMSE** A.-I. $[m]$	**Travel time** $[s]$	**Motion evaluation**
0.15	0.079	0.116	0.120	37.6	Slow and quite scattering
0.25	0.119	0.110	0.139	26.6	Quite slow, less scattering
0.35	0.135	0.115	0.145	19.4	Enough fast, constant speed and quite smooth
0.45	0.167	0.123	0.284	16.9	Fast, constant speed and smooth

Table 6.3: Results summary table

6.3.1 Considerations

From the plots (Figures 6.7, 6.8, 6.9, 6.10) the error graph (Figure 6.11), the resulting RMSE Table 6.2 and the travel time and motion quality evaluation we can observe different important considerations:

- **Measured - ideal paths analysis** - the error between the measured and ideal path has for every T value a fluctuating behavior with a final increase toward the end of the path, this final peak is probably due to a parallel reduction of the robot localization accuracy. The RMSE value for the measured - ideal paths increase with the increase of the T value, as expected, since lower Threshold (T) means a more strict requirement for the path points reaching.

- **AMCL - measured paths analysis** - the error between the AMCL and the measured path has an overall floating behavior for each T value, with more or less constant values. The important thing to notice are the error ellipses in the plots. They always start bigger and once the motion starts they become more narrow around the real robot localization. This happens till the right turn of the robot around the sixth ideal path point, where the localization starts to diverge from the real robot position and so the error ellipse enlarges. This is a clear sign of the AMCL recovery feature for which during a rotation, in which the robot localization became harder and start diverging from the real one due to big laser scans shifting (this can also be a consequence of the fact that in order to ensure a stable control the rotational gain has to be set higher with respect to the linear one, leading to faster rotational speeds), the robot correctly enlarge its cloud of estimated poses adding extra particles. Thus it is able to realign the behaved robot pose to the real one.
  Anyway at the end of the motion the robot localization moves from the real one and the error ellipse doesn't enlarge including the real robot pose, so the robot wrongly convinces itself of a wrong pose. This is probably due to a simple fact, at the end of the motion the robot arrives very close to the end map wall, the LiDAR sensor takes scans only in the front and side parts of the robot, so once the robot approaches the wall it has a very limited environment view, this can easily lead to a wrong AMCL estimation of the robot pose.
  Comparing the RMSE values between the different tests, we can notice that the value doesn't change that much, except for the higher T value, in which the robot moves faster and so also the robot localization became harder and les accurate. As a consequence also the error ellipses are bigger with respect to the previous tests. The average localization error is around 11 cm. This leads to an enough satisfying motion in the tests but points out a not very precise localization during the navigation.

- **AMCL - ideal paths analysis** - the error between the AMCL and the ideal planned path has a floating behavior and a peak in the final part of the navigation, similarly to the measured versus ideal path errors. This final peak can be explained as the fact that the previous path points are taken as close as possible to the ideal one. While the last one couldn't be selected differently and the control logic was built in order to stop the robot motion once it reached a minimum T distance from the final point. So it is normal that the last error distance is higher. It can also be remarked that the final distance is not always minor then the required T value, this is probably due to two facts. The first one is the fact that the AMCL localization updates in

a discrete way, so it is possible that the robot has moved a little more after the last available AMCL robot pose. The second reason is that the control logic doesn't directly use the AMCL localization, but the Hector SLAM one, published in a continuous way (at high frequency) and occasionally corrected by the AMCL. The SLAM pose is also transformed from the *odom* frame to the *map* one, this value should be equal to the AMCL one, but due to some computational errors and approximations the two values can became a bit different.
Finally we can notice that the RMSE values for the AMCL versus ideal paths analysis increases in parallel to the T value. This is as expected, since, as previously said, a larger threshold value leads to a less close single ideal path point reaching and so an overall less accurate path tracking.

- **Travel time and quality motion analysis** - As reported in the previous list, the increase of the T parameter value lead to a reduction o the travel time, so a quicker destination reaching, and a more smooth and harmonious motion. This is because a higher path point reaching threshold leads to more frequent update of the next path point to reach during the motion, leaving the actual goal point always at a suitable distance from the robot. This gives a more constant robot speed and a smoother navigation, since the machine doesn't try to precisely reach the single points correcting the orientation and slowing down (having a smaller distance to the goal point).

From the previous analysis we can draw some important final considerations. We can observe a clear direct correlation between the T parameter and the accuracy of the path tracking, both considering the robot localization and the real robot poses. The higher the T value is, the higher are the RMSE values of the distance between the ideal path and the AMCL and measured ones. In parallel we can observe an increase in the quickness in the destination reaching and a quality enhancement in the motion smoothness with a T value growing.
So the T parameter has to be tuned, doing a trade-off between the two desired motion behaviors depending on the applications. A too high T value lead to a smoother and fast motion but a less accurate robot localization and consequently to a worse path tracking that can lead to obstacle crushes and a more dangerous robot navigation. On the other hand a too low T value leads to a closer path tracking, being sure to not hit obstacles with a slower and safer motion, but, at the same time, this makes the robot move too slow and in a scattering and not harmonious way. Obviously safety has to be considered as the most important index in the T value tuning and secondly also the other wanted navigation behaviors.

Chapter 7

Conclusions and Future Developments

7.1 Conclusions

In this conclusion chapter we are going to summarize the work that has been completed and the desired and obtained results.
The main thesis goal was to make a robot able to autonomously move to a desired position in a mapped environment. In order to reach this main objective, different work parts were developed. The first development regarded the system integration between the different robot's devices. In particular, the on-board computer (the Nvidia Jetson Nano) was connected to the MCU via an I2C bus, enabling a data flow from the computer to the controller device. This communication channel was then checked and improved thanks to a second one, the UART bus, used for retrieving the MCU inner signals. An I2C transmission feedback control logic was implemented, sending the messages to the microcontroller till a correct reception. For the localization messages, a different technique was developed to improve the reception, an outlier rejection logic based on the maximum feasible distance that the robot can move and rotate. If the received robot position is farther from the previous one than those thresholds, the message is rejected.
The second part of the work focused on environment mapping and robot localization in the obtained map. The mapping phase was done using a LiDAR sensor for exploring the environment, while the SLAM was implemented using the Hector SLAM software saving an occupancy grid map. Thus the robot localization in such a map was then assigned to an AMCL algorithm, working together with the Hector SLAM robot pose computation through a scan matching logic. The Hector SLAM robot pose was used as a virtual laser odometry since it was published at a high frequency (the laser scan frequency), and corrected by the AMCL each time the

algorithm update the robot localization.
The final part of the project deals with path planning, path tracking, and autonomous robot navigation control logic. The path planning was implemented using an A* path search algorithm on the previously obtained map, after a boundary increase process for the map, in order to take into account the robot size and obtain a suitable obstacle-free path. Then the path tracking logic was developed selecting the correct path point to reach according to the actual robot localization. In this algorithm, the path points are considered reached once the robot has a distance from them less than a T value. This parameter was then tuned during the test phase. At the end of the work, the autonomous navigation control logic was designed. It is a negative feedback static controller, which returns the wheels' desired speed according to the robot distance and misalignment from the actual goal path point multiplied by suitably tuned gains.
Finally, the autonomous navigation tests were run checking the robot's behavior according to the variation of the T value for the path points reaching. The robot proved to be able to suitably move and reach the desired final destination, thus confirming that the previous work was correctly carried out. It was then noticed that using a low T parameter value the robot had a close path tracking but at the same time a slow and scattering motion. On the other hand, a higher T value leads to a faster destination reaching with a smoother motion, but a less accurate robot path, with respect to the planned one. A too high T value also gave problems to the robot localization, thus is not recommended. We can conclude that the robot was able to correctly autonomously navigate to the desired position, and by tuning the T parameter we can modify its motion behavior, making a trade-off between quickness of the task accomplishment and smoothness of the motion with the correct path tracking, being sure not to hit obstacles.

7.2 Future developments

The thesis goal was reached but this doesn't mean that the work is completely done. As said at the beginning of the thesis, this project was part of a larger one that also at this moment is carried out by other students and researchers for the Innotech Systems company. Some of the possible future robot developments and improvements are presented in the following list.

- The robot localization in the map proved to be around 10 cm accurate as average. This accuracy was enough for the project task accomplishment, but is not a very low value. Thus the robot localization could be improved by adding other extra sensors to the machine and combining their measurements, possibly with a Kalman filter, obtaining a more accurate robot pose. Some possible sensors could be the IMU, precise wheel encoders and a GNSS sensor that could make the robot localize itself in outdoor environments, not suitable for mapping.

- A more performing MCU could be used in order to test the overall work making it implement, as designed, the control logic.
- A more robust control logic could be developed, designing a dynamic controller.
- A local path planning, maybe based on ultrasound sensors, could be implemented, ensuring a safer obstacle avoidance and for dealing with unexpected and dynamic obstacles.

Appendix A

A* pseudo-algorithm

```
// A* finds a path from start to goal.
// h is the heuristic function.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    open_set := {start}

    // For node n, came_from[n] is the node immediately preceding it
   on the cheapest path from start
    came_from := an empty map

    // For node n, g_score[n] is the cost of the cheapest path from
   start to n currently known.
    g_score := map with default value of Infinity
    g_score[start] := 0

    // For node n, f_score[n] := g_score[n] + h(n). f_score[n]
   represents our current best guess as to
    f_score := map with default value of Infinity
    f_score[start] := h(start)

    while open_set is not empty

        current := the node in open_set with lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, current)

        open_set.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from
   current to neighbor
```

```
            // tentative_g_score is the distance from start to the neighbor through current
            tentative_g_score := g_score[current] + d(current, neighbor)
            if tentative_g_score < g_score[neighbor]
                // This path to neighbor is better till now. Record it.
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := tentative_g_score + h(neighbor)
                if neighbor not in open_set
                    open_set.add(neighbor)

    // Open set is empty but goal was never reached
    return failure

function reconstruct_path(came_from, current)
    total_path := {current}
    while current in came_from.Keys:
        current := came_from[current]
        total_path.prepend(current)
    return total_path
```

Bibliography

[1] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. *A review of mobile robots: Concepts, methods, theoretical framework, and applications.* 2019. DOI: `10.1177/1729881419839596` (cit. on p. 1).

[2] Uwe Jahn, Daniel Heß, Merlin Stampa, Andreas Sutorma, Christof Röhrig, Peter Schulz, and Carsten Wolff. «A taxonomy for mobile robots: Types, applications, capabilities, implementations, requirements, and challenges». In: *Robotics* 9 (4 2020). ISSN: 22186581. DOI: `10.3390/robotics9040109` (cit. on p. 1).

[3] Roland Siegwart, Illah R Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots, Second Edition.* Vol. 23. 2011 (cit. on p. 1).

[4] Open Robotics. *ROS - Robot Operating System.* URL: `https://www.ros.org/`. (accessed: 01.11.2022) (cit. on p. 4).

[5] Alvin Jacob, Wan Nurshazwani Wan Zakaria, and Mohd Razali Bin Md Tomari. «Evaluation of I2C communication protocol in development of modular controller boards». In: *ARPN Journal of Engineering and Applied Sciences* 11 (8 2016). ISSN: 18196608 (cit. on pp. 7, 8).

[6] NXP Semiconductors. «UM10204 I^{2}C-bus specification and user manual». In: *Semiconductors* 3 (Oct. 2021) (cit. on pp. 8, 11).

[7] Peter Corcoran. «Two Wires and 30 Years : A Tribute and Introductory Tutorial to the I2C Two-Wire Bus». In: *IEEE Consumer Electronics Magazine* 2 (3 2013). ISSN: 2162-2248. DOI: `10.1109/mce.2013.2257289` (cit. on p. 8).

[8] Ashok Kumar Gupta, Ashish Raman, Naveen Kumar, and Ravi Ranjan. «Design and implementation of high-speed universal asynchronous receiver and transmitter (UART)». In: 2020. DOI: `10.1109/SPIN48934.2020.9070856` (cit. on p. 11).

[9] J. Norhuzaimin and H.H. Maimun. «The design of high speed UART». In: *2005 Asia-Pacific Conference on Applied Electromagnetics.* Dec. 2005. DOI: `10.1109/APACE.2005.1607831` (cit. on p. 12).

[10] Eric Peña and Mary Grace Legaspi. *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter*. 2020 (cit. on p. 12).

[11] Ritesh Kumar Agrawal and Vivek Ranjan Mishra. «The design of high speed UART». In: *2013 IEEE Conference on Information & Communication Technologies*. Apr. 2013, pp. 388–390. DOI: `10.1109/CICT.2013.6558126` (cit. on p. 12).

[12] Shubham Nagla. «2D Hector SLAM of Indoor Mobile Robot using 2D Lidar». In: *2020 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS)*. Dec. 2020, pp. 1–4. DOI: `10.1109/ICPECTS49113.2020.9336995` (cit. on p. 14).

[13] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard. «Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age». In: *IEEE Transactions on Robotics* 32.6 (Dec. 2016), pp. 1309–1332. ISSN: 1941-0468. DOI: `10.1109/TRO.2016.2624754` (cit. on p. 14).

[14] Shao-Hung Chan, Ping-Tsang Wu, and Li-Chen Fu. «Robust 2D Indoor Localization Through Laser SLAM and Visual SLAM Fusion». In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2018, pp. 1263–1268. DOI: `10.1109/SMC.2018.00221` (cit. on p. 16).

[15] Stefan Kohlbrecher, Oskar von Stryk, Johannes Meyer, and Uwe Klingauf. «A flexible and scalable SLAM system with full 3D motion estimation». In: *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*. Nov. 2011, pp. 155–160. DOI: `10.1109/SSRR.2011.6106777` (cit. on pp. 16–18, 45, 47).

[16] Zhang Xuexi, Lu Guokun, Fu Genping, Xu Dongliang, and Liang Shiliu. «SLAM Algorithm Analysis of Mobile Robot Based on Lidar». In: *2019 Chinese Control Conference (CCC)*. July 2019, pp. 4739–4745. DOI: `10.23919/ChiCC.2019.8866200` (cit. on p. 17).

[17] Isro Wasisto, Novera Istiqomah, I Kadek Nuary Trisnawan, and Agung Nugroho Jati. «Implementation of Mobile Sensor Navigation System Based on Adaptive Monte Carlo Localization». In: *2019 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*. Oct. 2019, pp. 187–192. DOI: `10.1109/IC3INA48034.2019.8949581` (cit. on pp. 19, 20).

[18] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. «Monte Carlo localization for mobile robots». In: *Proceedings - IEEE International Conference on Robotics and Automation* 2 (1999). ISSN: 10504729. DOI: `10.1109/robot.1999.772544` (cit. on p. 19).

[19] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. «Robust Monte Carlo localization for mobile robots». In: *Artificial Intelligence* 128 (1-2 2001). ISSN: 00043702. DOI: `10.1016/S0004-3702(01)00069-8` (cit. on p. 19).

[20] Ming-An Chung and Chia-Wei Lin. «An Improved Localization of Mobile Robotic System Based on AMCL Algorithm». In: *IEEE Sensors Journal* 22.1 (Jan. 2022), pp. 900–908. ISSN: 1558-1748. DOI: `10.1109/JSEN.2021.3126605` (cit. on pp. 20, 22).

[21] Sun Dihua, Qin Hao, Zhao Min, Cheng Senlin, and Yang Liangyi. «Adaptive KLD sampling based Monte Carlo localization». In: *2018 Chinese Control And Decision Conference (CCDC)*. June 2018, pp. 4154–4159. DOI: `10.1109/CCDC.2018.8407846` (cit. on p. 20).

[22] Hui Liu. *Robot Systems for Rail Transit Applications*. 2020. DOI: `10.1016/B978-0-12-822968-2.01001-9` (cit. on p. 23).

[23] Ade Candra, Mohammad Andri Budiman, and Kevin Hartanto. «Dijkstra's and A-Star in Finding the Shortest Path: a Tutorial». In: *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*. July 2020, pp. 28–32. DOI: `10.1109/DATABIA50434.2020.9190342` (cit. on p. 24).

[24] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 2168-2887. DOI: `10.1109/TSSC.1968.300136` (cit. on p. 24).

[25] Open Robotics. *ROS.org - ROS Graph Concepts: Nodes*. URL: `http://wiki.ros.org/Nodes`. (accessed: 25.11.2022) (cit. on p. 32).

[26] Open Robotics. *ROS.org - ROS Graph Concepts: Topics*. URL: `http://wiki.ros.org/Topics`. (accessed: 25.11.2022) (cit. on p. 42).

[27] Wim Meeussen. *REP 105 - Coordinate Frames for Mobile Platforms (ROS.org)*. 2010. URL: `https://www.ros.org/reps/rep-0105.html`. (accessed: 19.11.2022) (cit. on p. 45).

[28] Open Robotics. *ROS.org - roslaunch, Package Summary*. URL: `http://wiki.ros.org/roslaunch`. (accessed: 25.11.2022) (cit. on p. 46).

[29] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. «Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters». In: *IEEE Transactions on Robotics* 23.1 (Feb. 2007), pp. 34–46. ISSN: 1941-0468. DOI: `10.1109/TRO.2006.889486` (cit. on p. 49).

[30] Open Robotics. *ROS.org - tf, Package Summary*. URL: `http://wiki.ros.org/tf`. (accessed: 17.11.2022) (cit. on p. 50).

[31] Open Robotics. *ROS.org - rosbag, Package Summary*. URL: `http://wiki.ros.org/rosbag`. (accessed: 25.11.2022) (cit. on p. 68).