

UNIVERSITY OF PADUA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE IN CONTROL SYSTEMS ENGINEERING

**Design and development of an
Autonomous Mobile Robot (AMR)
based on a ROS controller**

Supervisor:

PROF. AUGUSTO FERRANTE

Candidate:

DANILO GERVASIO

2008570

Accademic Year: 2021/2022

Graduation Date: October 17 2022

*To my father,
who never gives up*

Abstract

An Autonomous Mobile Robot is a robot that runs and navigates by itself without human intervention. AMRs are fundamental for automating materials handling into warehouses. This thesis work explains the procedure followed for designing and developing an AMR.

The hardware components are chosen considering the way in which the robot will be controlled. A layered controlling approach is used.

The high-level controller is a mini PC running ROS nodes. The ROS controller manages the actions that drivers have to perform and the data received from sensors. The medium-level controller translates the actions into signals that will be sent to the low-level controllers. The medium-level controller is an Arduino Mega, which sends PWM signals for controlling the speed of two BLDC motor wheels. The low-level controllers are two BLDC motor drivers that power, sense and drive the motor wheels. The Arduino implements a PI controller on the speed of rotation of each motorized wheel. Real tests are performed on the robot in order to evaluate the efficiency of the controlling system.

A future implementation consists in an integration of a LiDAR sensor in order to execute a SLAM algorithm, which will allow the robot to move autonomously in a space with obstacles.

Summary

1	Introduction	1
1.1	Various types of autonomous robots	1
1.1.1	AGV	1
1.1.2	AMR	1
1.1.3	AMR System	2
1.1.4	Types of mechanical structures of AMRs	2
1.1.5	AMR's components	4
1.1.6	Navigation Technology	5
1.2	The company	6
1.3	Thesis's contents	7
2	Hardware Components	8
2.1	Kinematic analysis	10
2.1.1	Drivers configuration	10
2.1.2	Degrees of freedom	11
2.1.3	Differential drive kinematics	13
2.2	Components	18
2.2.1	Motion	18
2.2.2	Navigation	23
2.2.3	Controllers	24
2.2.4	Security	25
2.2.5	Power	27
2.3	Robot's structure	30
2.4	Schematics	38
3	Control	40
3.1	Communication between Controllers	41
3.2	Speed Control	42

3.2.1	PI Controller	43
3.2.2	High-Level Implementation	44
3.2.3	Medium-Level Implementation	46
3.2.4	Acceleration Ramp	51
4	Code	53
4.1	Workspace	53
4.2	Speed Control	55
4.2.1	High-Level Implementation	55
4.2.2	Medium-Level Implementation	60
5	Tests and Results	69
5.1	Speed Controller	69
5.1.1	Single Wheel Motion	71
5.1.2	Linear Motion	72
5.1.3	Angular Motion	73
5.1.4	Payload Test	77
6	Future Implementations	80
6.1	SLAM	80
6.1.1	Functioning	81
6.2	Multi-Sensor Fusion Method	83
6.2.1	Extended Kalman Filter	83
6.2.2	Other Architectures	85
	Bibliografia	88
A	Burhless DC motor	90
A.1	Similarities BLDC and DC motors	90
A.2	Differences BLDC and DC motors	91
B	PWM	92
B.1	How it works	93
C	Microcontroller	95
D	IMU	96

E	ROS	98
E.1	What is ROS?	98
E.1.1	What is a Middleware	98
E.2	Communication Tools	99
E.3	Other features	99
E.4	ROS Languages	100
E.5	ROS Nodes	101
F	Client-Server Architecture	102
G	Publisher-Subscriber Architecture	104
H	C++	105
H.1	Why use C++?	105
H.2	Differences between C and C++	105
I	Step Response	106

Chapter 1

Introduction

The objective of the following thesis work is to design and build an *Autonomous Mobile Robot* capable of moving in the working environment without the need of human interactions.

1.1 Various types of autonomous robots

1.1.1 AGV

An **Automated Guided Vehicle (AGV)** is an electric mobile robot that runs and navigates by itself without human intervention. Automatic guided vehicle consists of one or more PLCs or PCs controlled, wheel-based load carrier that navigates on the factory or warehouse floor without the need for an onboard driver. Its main purpose is to transport materials from a location *A* to a location *B* of the space. The AGV performs the navigation task with the use of physical components placed on the environment (eg tapes, sensor, magnets).

1.1.2 AMR

Autonomous Mobile Robot (AMR) is a smarter version of an AGV Robot. It is a vehicle that uses on-board sensors and processors to autonomously move materials without the need of physical guides or markers.

The greatest advantage of AMR over AGV is that AMR provides alternative navigation options. An AMR navigates using a predefined map and plans its own routes to the destination. It can detect obstacles similar to an AGV, but it is slightly smarter because it can avoid the obstacles by navigating around them.

Therefore, AMR robot is considered more flexible because it can change its path dynamically with less effort. AMR robot is perceived to be much more expansive compared to an AGV. In reality, the AMR could be more cost effective due to its flexibility and ease to set up.

1.1.3 AMR System

An **Autonomous Mobile Robot System** is a set of automated elements united by a superior management system. The main elements of the system are the *AMR Vehicles*, the *AMR Management System* and the *Peripherals*.

The **AMR Management System** controls the AMR robots, managing orders and traffic of the vehicles.

Some **Peripherals** are needed to make the system works (such as charging stations, communication devices, etc.).

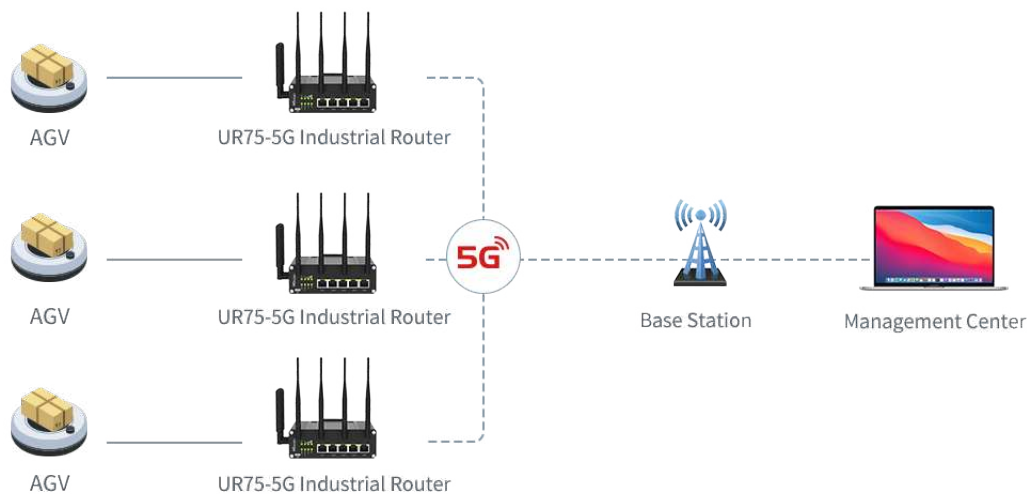


Figure 1.1: Representative model of a base configuration of an AGV System

1.1.4 Types of mechanical structures of AMRs

The choice of AMR's body structure depends on the task the vehicle will have to perform and the complexity of the infrastructure in which the robot will work.

There are 3 main types of robot: unit load, forklift and tugger.

- **Unit load vehicles:** these are motorized vehicles capable of transporting a single product (i.e. a coil, a motor, a pallet or a bin containing products).



Figure 1.2: Example of an unit load AMR.

- **AMR Forklifts:** they are used to move pallets. Many models have sensors on their forks (for example infrared sensors).



Figure 1.3: Example of a forklift AMR.

- **Towing (or tugger) AMR:** they are motorized vehicles capable of pulling one or more non-motorized vehicles with loads behind them, like a train.

They have a capacity up to 8 tonnes. They are also equipped with trays that can be lifted, lowered, with motorized rollers, belts, etc. to ensure the automatic transfer of loads.



Figure 1.4: Example of a Towing (or tugger) AMR.

1.1.5 AMR's components

An AMR vehicle is composed by 5 elements:

1. **Navigation System** in charge of acquiring the information needed to drive and follow a given track or direction.
2. **Safety System** that ensures that all the movements and manoeuvres are performed in safe conditions.
3. **Motion System:** it converts the energy drawn from the power supply into mechanical motion.
4. **Vehicle Controller:** it manages all the information required by all the sensors in order to succeed in the working purposes.
5. **Power supply:** that provides the energy needed for the vehicle movements and accessory functions. The main elements are the battery and the charging solution.

1.1.6 Navigation Technology

AMRs are able to move in the space through the use of a *Navigation System*. The mainly used systems are the following:

- **Wire guidance:** the AMR travels along a ground track that can be composed of wires, tracks, magnetic lines or cables. In this way the vehicle can follow the track mechanically, by using a magnetic sensor or by detecting a signal through the use of an antenna. This option does not offer flexibility and it requires the installation of rails on the working floor.
- **Laser guiding:** reflective tape is mounted on objects such as walls, fixed machines and poles. AMRs are equipped with a laser transmitter and receiver. The lasers reflect off the tape and the AMR is able to compute its position in the space precisely.
- **Inertial (gyroscopic) navigation:** some AMRs are controlled by a computer system with the aid of transponders embedded into the facility floor to verify that the AMR is on the proper course.
- **Visual guidance:** the AMR follows a path painted on the ground that its camera recognizes. The installation cost is lower and it is not needed any special installation on the working area.
- **Geoguidance:** the AMR contains a mapped representation of its environment in its system that allows it to move independently, without having to adapt infrastructures. It calculates its trips by itself automatically. This technology is very flexible because it is possible to modify the AMR's mapping at any moment by working directly on the mapping software. It is the most reliable solution.

In the recent years new AMR Navigation technologies have been developed thanks to the improvements of sensors software and technology.

- **LiDAR:** a LiDAR sensor transmits laser pulses that measure the distance between the hit object and the AMR equipped with it. This data is compiled to create a 360° degree map of the environment, allowing robots to navigate the facility and avoid obstacles without the need for any additional infrastructure. An AGV Vehicle equipped with this system can be called *Autonomous Mobile Robot (AMR)*.

- **Camera Vision System:** the camera allows information to be captured in real time, which helps the AMR to see obstacles. When this information is combined with the data provided by a LiDAR sensors, a dynamic and complete 3D image of the operational area is obtained.

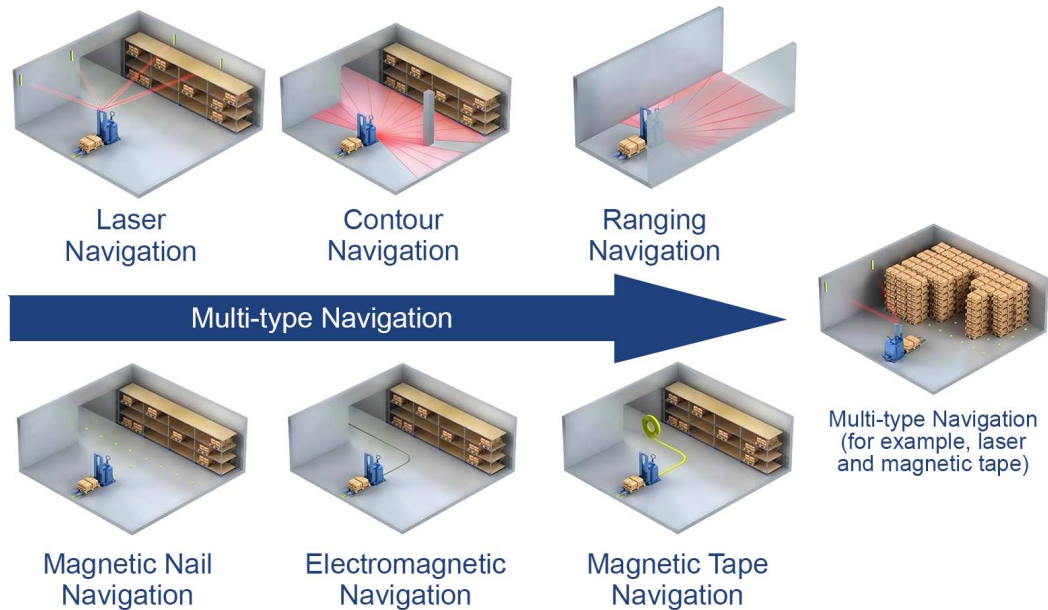


Figure 1.5: Various methods of AMR Navigation.

1.2 The company

IDEA soc. coop. is an engineering company located in Ancona which designs, projects and develops intelligent systems related to domotic, industry and services.

It offers practical smart solutions for solving specific problems and for automating processes. The company is composed by a young team of engineers, technicians and marketing agents.

The following thesis work has been performed in the company, during the internship period, started from February 2022 and ended in July 2022.



Figure 1.6: Logo IDEA soc. coop.

1.3 Thesis's contents

The thesis is divided in the following chapters:

- **Chapter 2:** describes the hardware components and the structure of the robot.
First, it is presented an analysis about the kinematic of the vehicle, which determines its design.
There is a section where all the hardware parts, with their functions, are listed.
It is described the physical structure of the robot's chassis.
The electric schemes are shown.
- **Chapter 3:** describes the logic used for controlling the robot. It shows the top-down control levels scheme, specifying the role of each controller.
- **Chapter 4:** contains part of the code executed by the programmed controllers (high-level and medium-level controllers).
- **Chapter 5:** contains plots relative to test results and an analysis of the data acquired during the experiments.
- **Chapter 6:** future implementations are described, showing the relative algorithms and methods needed for implementing new functionalities.






Chapter 2

Hardware Components

The hardware architecture of autonomous vehicles can be divided into two macro groups. The first one involves **accademic works** and concerns the study of architectures suitable for mobile vehicles devoted to research or service applications. The second line concerns **assembled industrial architectures** with standard components from industrial AGV suppliers.

These two fields have some points of contact and some differences: for example, the positioning accuracy of the vehicle is a strong requirement for the industrial AGV, but it seems to be less relevant in the research world, which is mainly oriented to other navigation issues. An example of convergence of industrial and academic goals is related to the use of cheap sensors to improve the navigation system.

An analysis about the actual AGVs and AMRs on the market has been done before the design process of the robot in order to reward awerness of the kind of solutions implemented by other companies.

Robot	Model	Cinematic System	Navigation and Security Systems
	MiR100	6 wheels with 2 traction wheels in the middle length driven by differential drivers.	Laser guidance front and back side. 3D camera front side. Proximity sensors in the 4 corners for identifying object less high than 20 cm.
	RB-THERON	6 wheels with 2 traction wheels in the middle length driven by differential drivers.	Laser guidance with RGBD cameras.
	Seit100	6 wheels with 2 traction wheels in the middle length driven by differential drivers.	Environment recognition via LiDAR, inertial localisation. Usage of laser sensors and cameras.
	RB-1 BASE	5 wheels with 2 traction wheels with servomotor driven by differential drivers.	Standard configuration integrates the UST-20LX sensor and the Laser Orbbec Astra sensor.
	RB-VOGUI	4 wheels with one traction motor for each wheel. 6 steering motors: 4x OMNI 4, 2x Ackerman.	LiDAR Camera.

2.1 Kinematic analysis

2.1.1 Drivers configuration

Considering a unit load AMR, there are mainly 3 different wheels setups:

1. **Steer drive:** the front wheel is controlled by a steering motor that allows the forward and steering motion.



Figure 2.1: Steer drive system.

2. **Quad drive:** the front and back wheels are controlled by steering motors that allow the forward and steering motion.

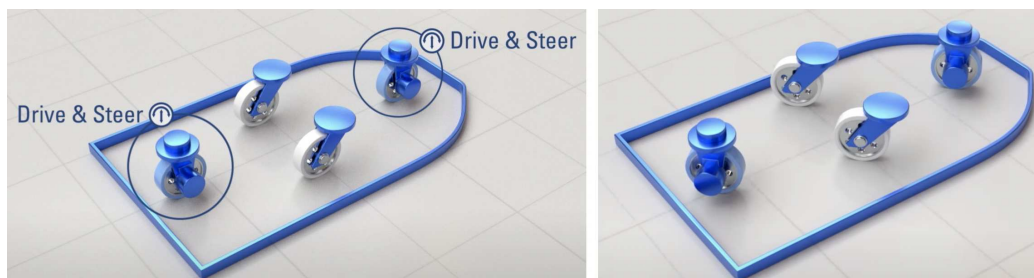


Figure 2.2: Quad drive system.

3. **Differential drive:** the mid-length wheels are controlled by two drivers that allow the rotation of each wheel. The steering movement of the robot is achieved when the speed of rotation of the driven wheels are different.

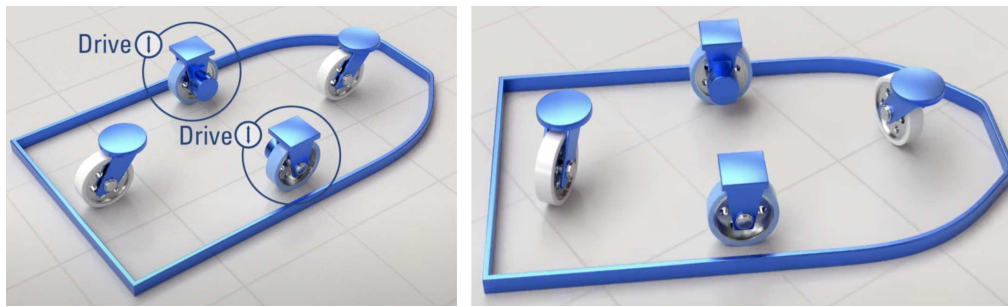


Figure 2.3: Differential drive system.

2.1.2 Degrees of freedom

A robot with *6 Degrees of Freedom* Kinematics can perform translations over the frame axes (x , y , z) and rotations about them.

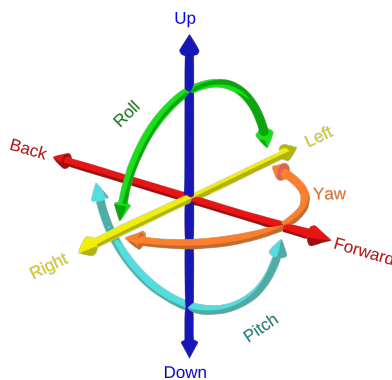


Figure 2.4: Possible movements allowed in a robot with a 6 DoF kinematics.

A robot with *3 Degrees of Freedom* Kinematics can either perform translations over the frame axes (x , y , z) or rotations about them. Differently, more usual cases either perform two translations over two axes and a rotation about one axis or another combination of three movements.

Taking in consideration the robot reference frame, the *Steer drive* and *Differential drive* explained in section 2.1.1 offer 2 DoF: the robot can perform translations over the x axis and rotations about the z axis.

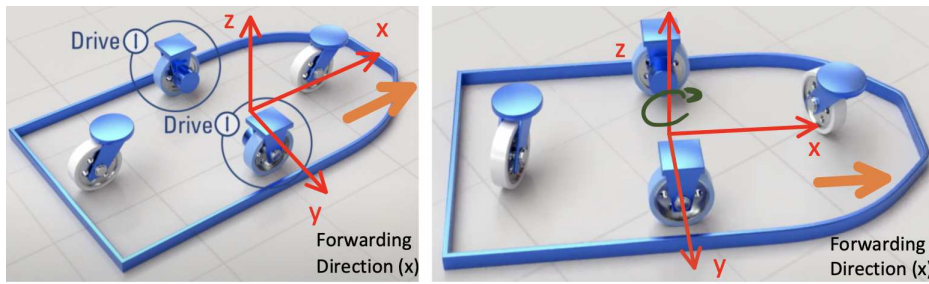


Figure 2.5: Kinematics of the Differential Drive implementation: 2 DoF allowed.

With a *Quad drive* the robot has 3 DoF: it can perform translations over the x and y axes and rotations about the z axis. It has the advantage of rotating around its center and moving sideways.

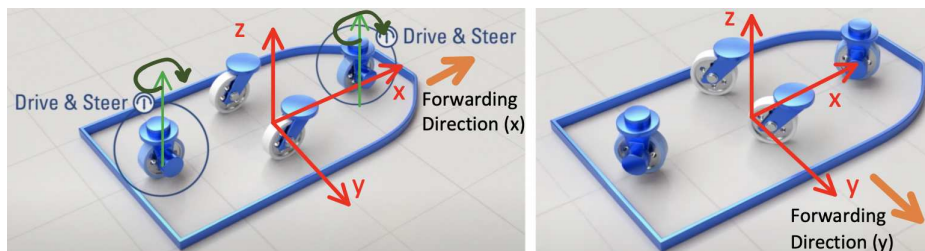


Figure 2.6: Kinematics of the Quad Drive implementation: 3 DoF allowed. The body reference frame alignment is not changed for a translation on the y axis.

Due to the high cost of electrical DC driving and steering wheel (starting from 1500\$ per unit). It has been chosen to use 2 differential drivers positioned as depicted in figure 2.5 with 4 ball wheels placed in the robot's corners for lowering the stability problem. The wheels setup depicted in figure 2.5 may result unstable, especially if the distance between the two driving wheels is not small. Placing an heavy object in a corner of the deck may cause the fall of the vehicle.

Considering the figure 2.5, it is chosen to insert 4 castors ball wheels at the deck's corners, in place of the 2 castor not spherical wheels. This solution allows to have a more stable robot. In addition, the driving and steering errors caused by the dynamics of traditional castor wheels are not present using ball wheels, resulting in a more responsive robot.

Despite implementing steering motors may lead to a more accurate angular positioning of the robot, many AGVs/AMRs manufactures have chosen this solution for its easy implementation and control.

2.1.3 Differential drive kinematics

It consists of 2 drive wheels mounted on a common axis, and each wheel can independently being driven either forward or backward. In order to perform rolling motion, while varying the velocity of rotation of each wheel, the robot must rotate about a point that lies along their common left and right wheel axis. The point that the robot rotates about is known as the *ICC - Instantaneous Center of Curvature* (see figure 2.7). The robot's position is expressed considering its center's coordinates (x, y) .

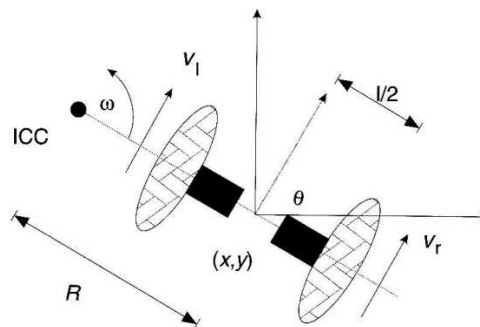


Figure 2.7: Differential Drivers kinematics.

By varying the velocities of the two wheels, it is possible to vary the trajectories that the robot takes. Because the rate of rotation about the ICC must be the same for both wheels, the following equations can be written:

$$w(R + l/2) = V_r \quad (2.1)$$

$$w(R - l/2) = V_l \quad (2.2)$$

Where l is the distance between the centers of the two wheels, V_r , V_l are the right and left wheel velocities along the ground (expressed in *meter over*

second), and R is the signed distance from the ICC to the midpoint placed in the middle of the wheels axis. At any instance in time it is possible to compute R and w (expressed in *radicant over second*):

$$R = \frac{l V_l + V_r}{2 V_r - V_l} \quad w = \frac{V_r - V_l}{l} \quad (2.3)$$

It is possible to consider three different canonical scenarios of the robot motion:

- (a) **Straight motion:** if $V_l = V_r$, a forward linear motion is achieved in a straight line. R becomes infinite, and there is effectively no rotation, in fact w is zero.
- (b) **Rotation about the robot center:** if $V_l = -V_r$, then $R = 0$, and a rotation about the midpoint of the wheel axis is achieved.
- (c) **Rotation about the left wheel:** if $V_l = 0$, then a rotation about the left wheel is obtained. In this case $R = \frac{l}{2}$. The rotation about the right wheel can be achieved if $V_r = 0$.

Forward kinematics

Forward kinematics determines where the end effector will be if the joints are set to a specific position. There is only one solution to the forward kinematic equation. When the joints are set to a specific position, the end effector will always end up in the same place.

For a differential drive robot, the solutions of the problem is the position and orientation of the robot in the space, after rotating each wheel of a specific velocity.

Considering the parameters defined in figure 2.7, knowing velocities V_l , V_r and using equation 2.3, the ICC location x and y coordinates are:

$$ICC = [x - R \sin(\theta), y + R \cos(\theta)] \quad (2.4)$$

Considering a time $t + \delta t$, with $t \geq 0$, $\delta \geq 0$, the new robot position x', y', θ' with respect to the actual position x, y, θ referred to the world reference frame will be:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(w\delta t) & -\sin(w\delta t) & 0 \\ \sin(w\delta t) & \cos(w\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ 0 \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ w\delta t \end{bmatrix} \quad (2.5)$$

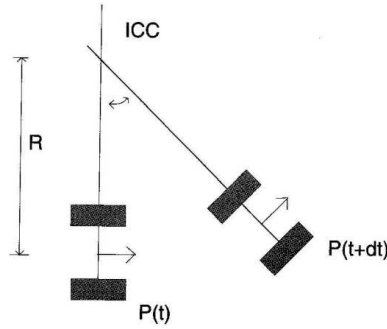


Figure 2.8: Forward kinematics for differential drive robot.

The 3x3 matrix is the **Revolute Matrix** which is multiplied for the position vector (3x1 vector) describing the center of the robot with respect to a reference frame that has the *ICC* as origin (called *ICC reference frame*). Then, it is summed the coordinates of the origin of this reference frame (the frame with the *ICC* as origin) expressed with respect to the world reference frame.

Generally equation 2.5 can be expressed in the following way:

$$[P']_{world} = [R_z(w\delta t)] * [P]_{ICC} + [ICC]_{world} \quad (2.6)$$

Where P is the actual robot position and orientation with respect to the *ICC reference frame*; P' is the new robot position and orientation, with respect to the *world reference frame*, after the transformation; $R_z(w\delta t)$ is the revolute matrix used for accomplishing a rotation about the z axis of an angle equals to $w\delta t$; ICC is the coordinates of the instantaneous center of curvature with respect to the *world reference frame*.

It is possible to write equation 2.6 in the following form:

$$\begin{bmatrix} P'_{world} \\ 1 \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} R_z(w\delta t) \\ 0 \ 0 \ 0 \end{bmatrix} & ICC_{world} \\ & 1 \end{bmatrix} \begin{bmatrix} P_{ICC} \\ 1 \end{bmatrix} \quad (2.7)$$

Which can be written with respect to the **Transformation Matrix**: 4x4 matrix used for performing a transformation in the coordinates, passing from the *ICC* reference to the *world* reference frame.

$$[P]_{world} = [T_{ICC,world}] [P]_{ICC} \quad (2.8)$$

Inverse kinematics

Inverse kinematics is a mathematical process used to calculate the joint positions that are needed to place a robot's end effector at a specific position and orientation, known as its *pose*. In a differential drive robot, the solution of the inverse kinematics problem consists in the velocity that each wheel has to rotate for making the robot reach a goal position and orientation in the space.

It is possible to describe the position of a mobile robot capable of moving in a particular direction $\Theta(t)$ with a given velocity $V(t)$.

$$x(t) = \int_0^t V(t) \cos(\theta(t)) dt \quad (2.9)$$

$$y(t) = \int_0^t V(t) \sin(\theta(t)) dt \quad (2.10)$$

$$\Theta(t) = \int_0^t w(t) dt \quad (2.11)$$

For a differential drive robot:

$$x(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)] \cos(\theta(t)) dt \quad (2.12)$$

$$y(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)] \sin(\theta(t)) dt \quad (2.13)$$

$$\Theta(t) = \frac{1}{l} \int_0^t [v_r(t) + v_l(t)] dt \quad (2.14)$$

A differential drive robot imposes what are called **non-holonomic** constraints on establishing its position. For example, the robot can not move laterally along its axis. A similar non-holonomic constraint is a car that can only turn its front wheels. It can not move directly sidewise, in fact parallel parking a car requires a more complicated set of steering maneuvers. So it is not possible to simply specify an arbitrary robot pose (x, y, θ) and find the velocities that will take it there.

For the special cases of $v_l = v_r = v$ (robot moving in a straight line, $w = 0$) the motion equations are:

$$\begin{bmatrix} x' \\ y' \\ \theta' \delta t \end{bmatrix} = \begin{bmatrix} x + v \cos(\theta) \delta t \\ y + v \sin(\theta) \delta t \\ \theta \delta t \end{bmatrix} \quad (2.15)$$

If $v_r = -v_l$, then the robot rotates in place, $R = 0$ and the equations are:

$$\begin{bmatrix} x' \\ y' \\ \theta' \delta t \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v \delta t \frac{1}{l} \end{bmatrix} \quad (2.16)$$

If $v_l = 0$ the robot rotates about the left wheel and $R = \frac{1}{l}$. The opposite happens when $v_r = 0$.

In each specific case depicted above, it is possible to find the values of v_l, v_r knowing x, y, δ, x', y' and δ' .

There are often multiple different solutions and multiple approaches to calculate the solution of the inverse kinematics problem. This motivates a non optimized navigation strategy that is sometimes used for making the robot curves over an arch, by moving the robot in a straight line, then rotating for a turn in place, and then moving straight again and so on.

2.2 Components

In this section all the hardware components used for building the robot are described.

2.2.1 Motion

Wheels

The wheels chosen for the implementation are two **Brushless DC (BLDC) motors wheels** (see appendix A). Each motor is controlled by a driver controller. The motor is incorporated in the wheel unit (figure 3.2), it has the following characteristics:

- Voltage: 36 V
- Power: 250 W
- Wheel diameter: 0.25 m
- Thickness: 0.05 m

The motor is provided by an **Hall Effect Sensor**, used for knowing the rotor position in order to feedback this information to the motor driver.

Buying a motorized wheel had a cost advantage since it would have been more expensive to buy the wheel unit and the motor gearbox separately.

The castor ball wheel chosen for the implementation is depicted in figure 2.2.1. It has a diameter of 30 cm.



Figure 2.9: M365 250W 36V Motor Wheel Tire with Hall Sensor.



Figure 2.10: Castor spherical wheel.

Motor Driver

A motor driver is an hardware component responsible for the motion of the motor connected to it. It acts between the motor and the motor controller by sending the right signal for moving the wheel in the wanted position.

For the motorized wheel in figure 3.2 it has been chosen the Brushless DC Motor Driver Board 6V-60V 400W with Hall Driver Module ZS-X11D1. One driver for each wheel has been implemented.

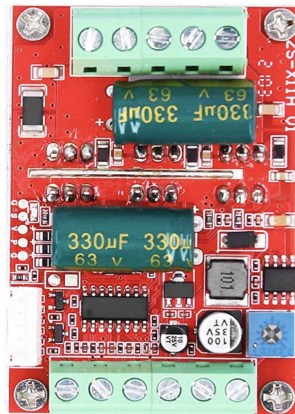


Figure 2.11: Brushless DC Motor Driver Board 6V-60V 400W model number ZS-X11D1.

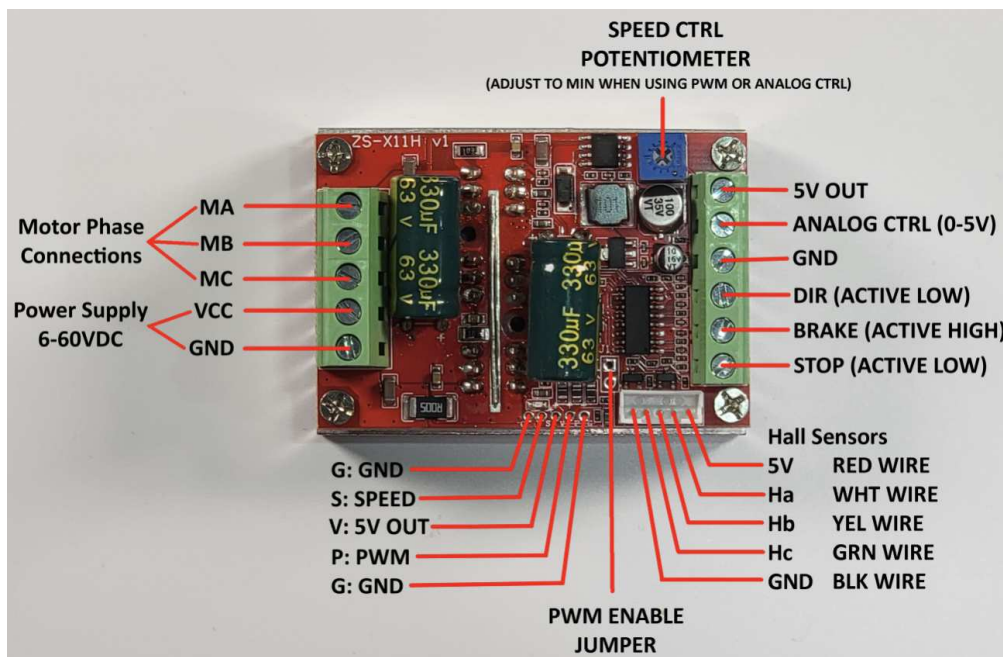


Figure 2.12: BLDC Motor Driver ports description.

Screw terminal left connector

- MA : Motor Phase A
- MB : Motor Phase B
- MC : Motor Phase C

- VCC : Power Supply Input 6 to 60 VDC
- GND : Power Supply Return

Screw terminal right connector

- 5V :Power output from the onboard voltage regulator.
- ANALOG CTRL (0-5V): Speed control input. 0 to 5V signal that can be driven from the wiper terminal of a potentiometer connected to 5V and ground, or an analog output of a Digital to Analog Converter (DAC).
- GND : Ground connection
- DIR : Direction control. This signal is active low, so shorting it to ground or applying a logic 0 changes the motor direction. Leaving the pin floating or applying a logic 1 will spin the motor in the default direction.
- BRAKE : Controls the motor brake. This signal is active high, so shorting it to 5V or applying a logic 1 will apply the motors brake. Leaving the pin floating or applying a logic 0 will disconnect the brake and allow the motor to spin.
- STOP : Functions as an enable input for the board. This signal is active low, so shorting it to ground or applying a logic 0 will disable the drive signals. This could be considered a coast or free spin mode. Leaving the pin floating or applying a logic high will enable the boards drive signals.

Aux control pads The auxiliary control pads allows a user to control the speed of the wheel using a Pulse Width Modulated (PWM, see appendix B) signal and to read the speed of the motor via the SC speed pulse output interface.

- G – GND: Ground connection for the auxiliary signals.
- S – SC: Speed pulse output interface. The output signal changes state whenever one of the hall sensors changes state. It ends up with 90 state changes for 1 complete rotation of the wheel (figure 2.13). To measure the speed of the wheel it is needed to measure the time between two transitions (w), in fact the R_{ps} (*Rotation Per Second*) can

be evaluated:

$$R_{ps} = \frac{1}{w * 90}$$

$$R_{pm} = R_{ps} * 60$$

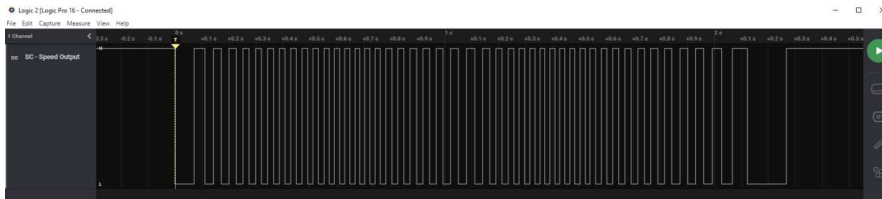


Figure 2.13: Speed pulse output signal for one wheel rotation using the ZS-X11D1 BLDC motor driver.

- V – 5V: 5V output PIN used to provide power to a small micro-controller.
- P – PWM: This pin can be used to control the motor via a Pulse Width Modulated (PWM) signal. The documentation states the frequency can be from 50Hz to 20kHz, with an amplitude of 2.5V to 5V. The typically Arduino PWM signal is either 490Hz or 980Hz so it can be used for controlling the motor speed via this pin. The PWM Jumper must be shorted and the Speed Control Potentiometer must be adjusted to the minimum when using the PWM control pin.
- G – GND: Ground connection for the auxiliary signals.
- J1 Jumper – The J1 jumper is used to connect the PWM control line. To use the PWM for motor control the J1 jumper must be shorted. Also the Speed Control Potentiometer must be adjusted to the minimum.

Hall sensor connector

- GND : Black wire. Used as a ground for the hall sensors.
- Hc : Green wire. Hall sensor for the motor C phase.
- Hb : Yellow wire. Hall sensor for the motor B phase.
- Ha : White wire. Hall sensor for the motor A phase.
- 5V : Red wire. Hall sensor power.

2.2.2 Navigation

Many sensors and components are needed for allowing the robot to move in the working environment. The navigation of the robot is based on the **Multi-Sensor Fusion** concept (better explained in section 6.2). In the following are described the components used.

LiDAR

The LiDAR Slamtech RPLIDAR A1 has been chosen for its affordable cost, the good performances that it offers and the easy implementation with the ROS environment.



Figure 2.14: Slamtech RPLIDAR A1.

IMU

The Adafruit 9-DOF IMU (see appendix D) has been implemented.

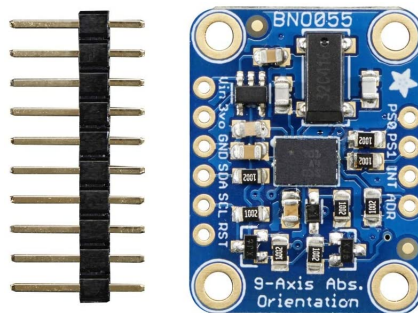


Figure 2.15: Adafruit 9-DOF IMU.

2.2.3 Controllers

As explained in section 3, the control part of the robot is divided into *High-Level Control* and *Medium-Level Control*.

Medium-Level Controller

The microcontroller (see appendix C) **Arduino Mega 2560 REV3** has been chosen for controlling the BLDC Motor Drivers (the low-level controllers) and for interacting with the IMU for the following advantages:

- User-friendly IDE
- Easy to program
- Cheap
- Well online documentations

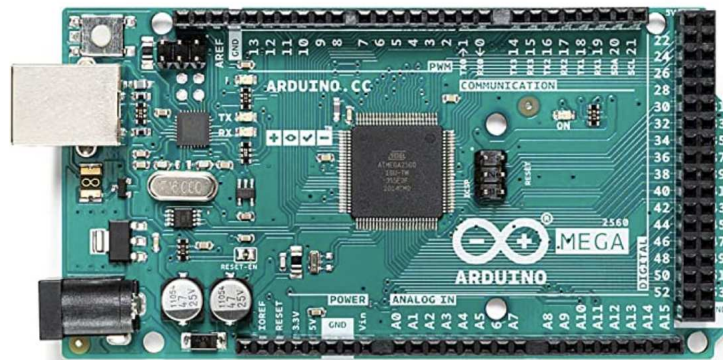


Figure 2.16: Arduino Mega 2560 REV3 board.

High-Level Controller

The mini PC Asus PB50 has been chosen for controlling the robot motions, by interacting with the medium-level controllers, and for managing the path planning task.

It is directly connected to the LiDAR and to the Arduino.

The choice of this components derives for the fact that the company had already bought this hardware part, which was not used anymore. Otherwise a *NVIDIA Jetson Xavier* would be used.



Figure 2.17: Mini PC Asus PB50.

2.2.4 Security

Emergency Stop Button

An Emergency Mushroom Push Button is installed in the front part of the robot. When pushed it cuts the three phasis of each bldc motor wheel, in order to achieve an istantaneous stop of the robot's motion.



Figure 2.18: Siemens emergency push button.

Relays

Two industrial relays have been installed for cutting the phasis of each motor. The Finder Relays 62.33 24V are used. When the mushroom button is pushed the relays are not powered anymore, resulting in cutting off the power of the motor's phasis.



Figure 2.19: Finder relay 62 series, 3PDT 24V DC.

Fuse box

Each electric component is connected to a fuse, located after the power supply. Four fuses are needed for the 2 BLDC motor drivers, depicted in

figure 2.2.1, and for the 2 DC-DC converters introduced in section 2.2.5.



Figure 2.20: Fuse box.

2.2.5 Power

Battery

Mobile robots use different kind of batteries: Sealed GEL, AGM Pure-Lead, Lithium, Flooded Lead Acid.

A **Lithium battery** has been chosen for the following reasons:

- **Longer cycle time:** For a given depth of discharge (DOD), Lithium batteries grant more recharging cycles so more life.
- **Deepest Depth of Discharge (DOD):** in order to have a good cycle life for an AGM or GEL battery, it is needed to keep DOD close to 40%-50%. Lithium batteries can handle DOD of 80% maintaining excellent battery life (still close to 2500 cycles).
- **Higher efficiency:** lithium batteries are more efficient. Lithium batteries efficiency is near 95% while in Lead batteries such AGM or GEL, it is close to 80-85%. Roughly, it means that when charging 1 kwatt, a lithium battery losses around 50 watts (you really get 950 watts). Lead battery instead, loses 15-20%, it means that more time is needed to fully charge the AGM/GEL battery.
- **Faster charging:** In AGM/GEL batteries it is needed around 4-5 hours to charge from 60% to 100%. With lithium it is only needed close to 1.5 hours.

- **Higher energy density:** More energy available in a given volume with respect to other types batteries.

Lithium batteries require complex electronics to keep life parameters under control and maintain safe operating conditions. It is important the presence of a **Battery Management System (BMS)** that controls the operation conditions of the battery, avoiding overcharging and overvoltages. Nowadays manufactures integrate BMS into the battery package itself.



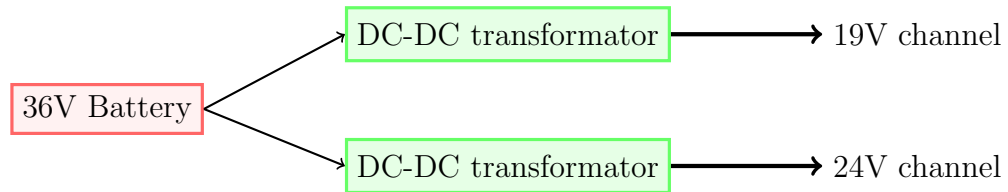
Figure 2.21: Chosen Lithium battery 36V 20Ah 500W with BMS.



Figure 2.22: Chosen battery charger 42V 2A.

DC-DC Converter

The robot contains different devices that work with different power voltages, some voltage transformers are used in order to convert the nominal voltage provided by the battery (36 V) to the desired ones.



The DC-DC transformers have been chosen considering the needed currents of the loads connected to the channels.

The 19V channel is used for powering the mini PC which manages all the ROS nodes.

The 24V channel is used for powering the relays used in the security system, explained in section 2.2.4.

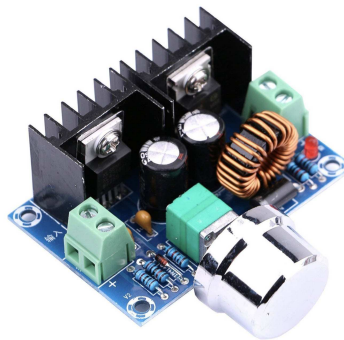


Figure 2.23: WINGONEER XL4016E1 DC-DC transformer for the 19V channel.

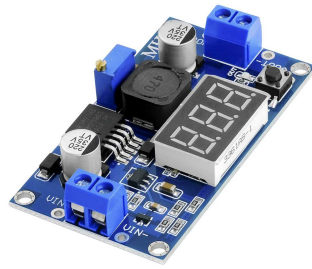


Figure 2.24: AZ-Delivery LM2596S LM2596 DC-DC transformer for the 24V channel.

Battery switch

A DC switch has been installed near the positive terminal of the power supply. In this way it is possible to shut off all the robot's components by turning off the switch. The battery switch allows to perform safely maintenance operations on the robot's components.



Figure 2.25: Battery DC switch used.

2.3 Robot's structure

The material chosen for the robot's chassis is stainless steel, AISI304, because it has good mechanical characteristics: high resistance to corrosion,

good workability.

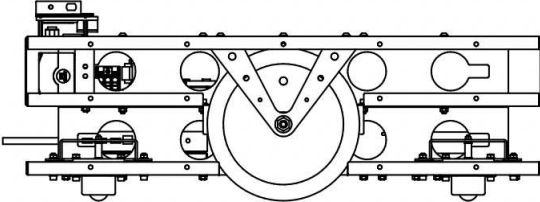


Figure 2.26: Lateral side view of the robot.

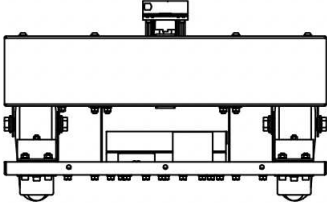


Figure 2.27: Front side view of the robot.

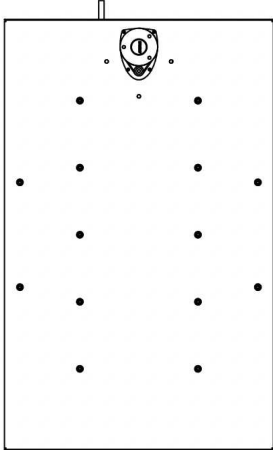


Figure 2.28: Upper view of the robot.

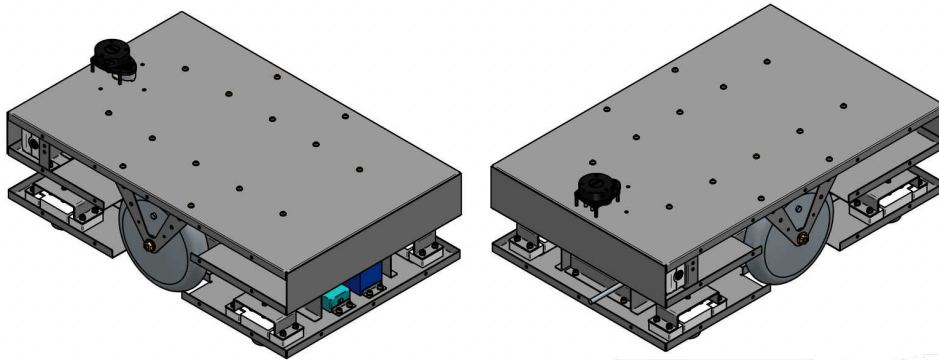


Figure 2.29: 3D model of the robot's chassis with the hardware components.

Some software tests have been performed in order to see how the chassis behaves when carrying a payload, the results are shown in figure 2.30 and 2.31

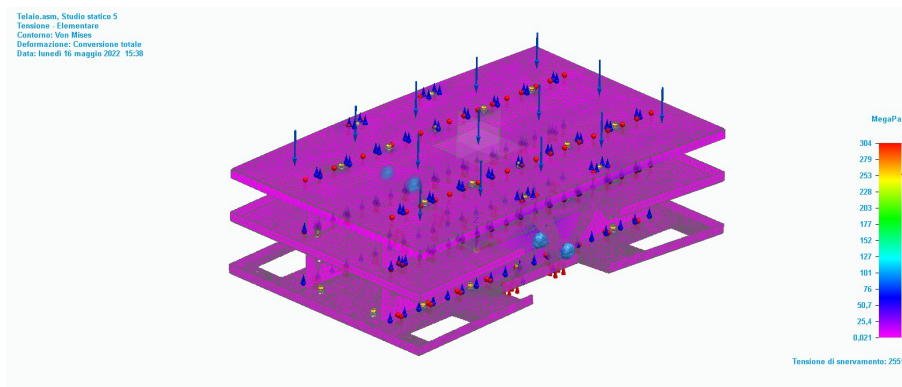


Figure 2.30: Chassis's stress-strain state obtained considering the interlocking constraint in the point of contact between the motor wheels and the chassis and applying in the upper frame a payload of 50Kg. In every point the chassis has a stress-strain value below the yield stress value of the material. This ensure its capability to hold up the payload.

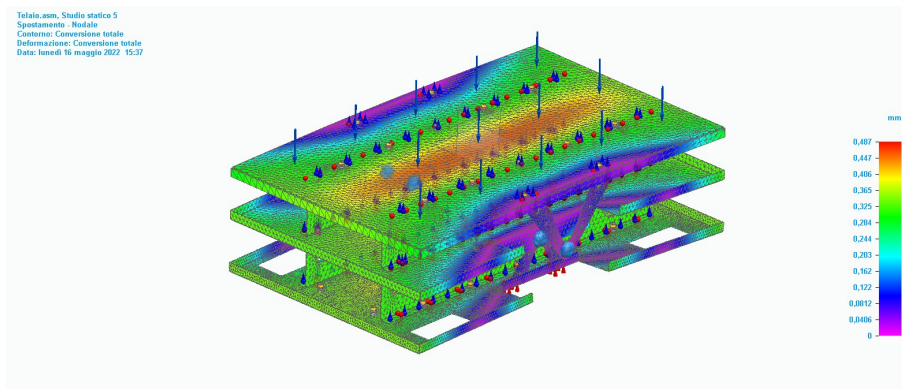


Figure 2.31: Graphical description of the chassis's deformation considering the same boundary conditions considered in figure 2.30.

The AMR's structure consists of three levels:

- (a) The first floor contains the power supply, the battery's charger (ready to be plugged into a 220V electrical outlet for charging), the mini PC and a switch for shutting off the robot. 2 screws bus bars are installed in order to have 4 positive channels at +36V and 4 negative channels at 0V. For each castor spherical wheel is present a spring mechanism that works as a car shock absorber.

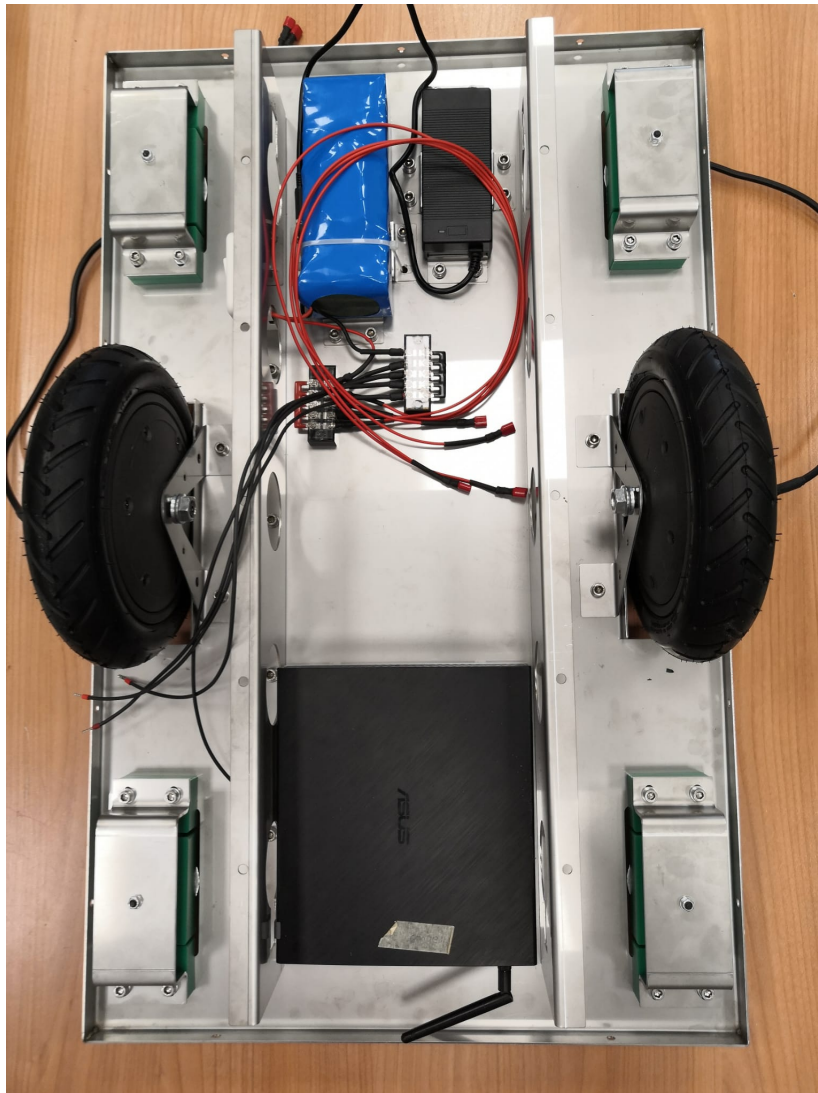


Figure 2.32: First floor of the AMR.

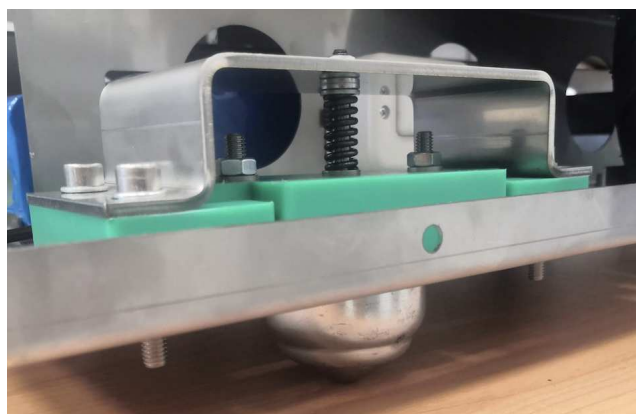


Figure 2.33: Shock absorber implemented for each castor spherical wheel.

- (b) The second floor contains a removable aluminium deck in which are installed all the other hardware components: the Arduino, 2 BLDC motor drivers, the IMU sensor, 2 relays, 2 DC-DC converters. These components have not been installed on the chassis of the robot in order to ease the maintenance operations. Contrary, 2 screws bus bars, the fuse box and the emergency stop button have been installed on the chassis of the robot. The 2 screws bus bars are used as positive and negative channels of the 24 output voltage, used for powering the relays.

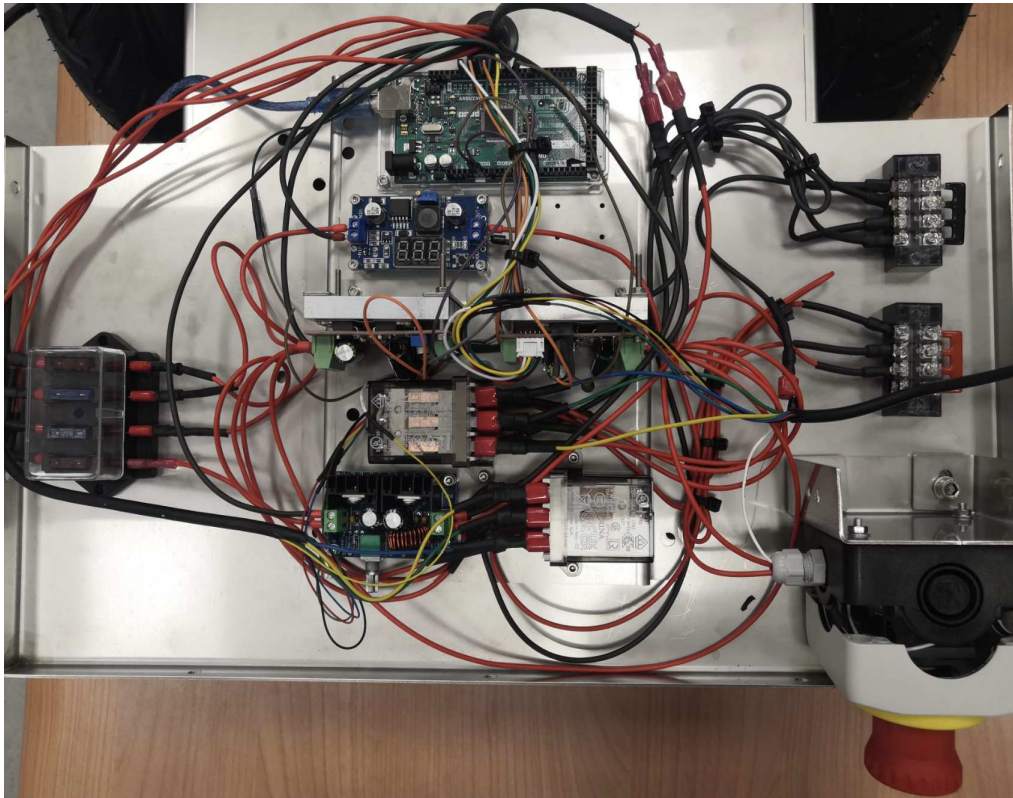


Figure 2.34: Hardware components located at the second floor of the AMR's chassis.

- (c) The third floor is used for loading the weights needed to be moved around the warehouse. The LiDAR sensor is placed in the front frame's border, in order to ease the scan operation of the environment (in the figure depicted the LiDAR sensor is not presented due to shipping delay).



Figure 2.35: Payload floor of the robot.



Figure 2.36: Left side view of the robot.



Figure 2.37: Front side view of the robot.



Figure 2.38: Right side view of the robot.

2.4 Schematics

In the following figures it is shown the electric schemes of the connection of all the components involved into the robot.

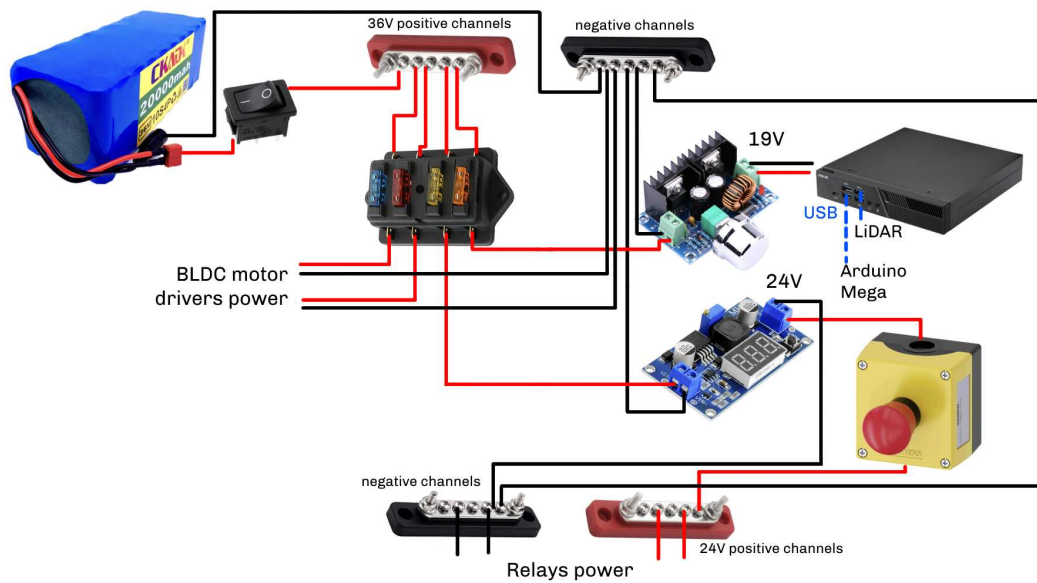


Figure 2.39: Schematic regarding the power channels of the system.



Figure 2.40: Schematic of the connection between each wheel with the motor driver.

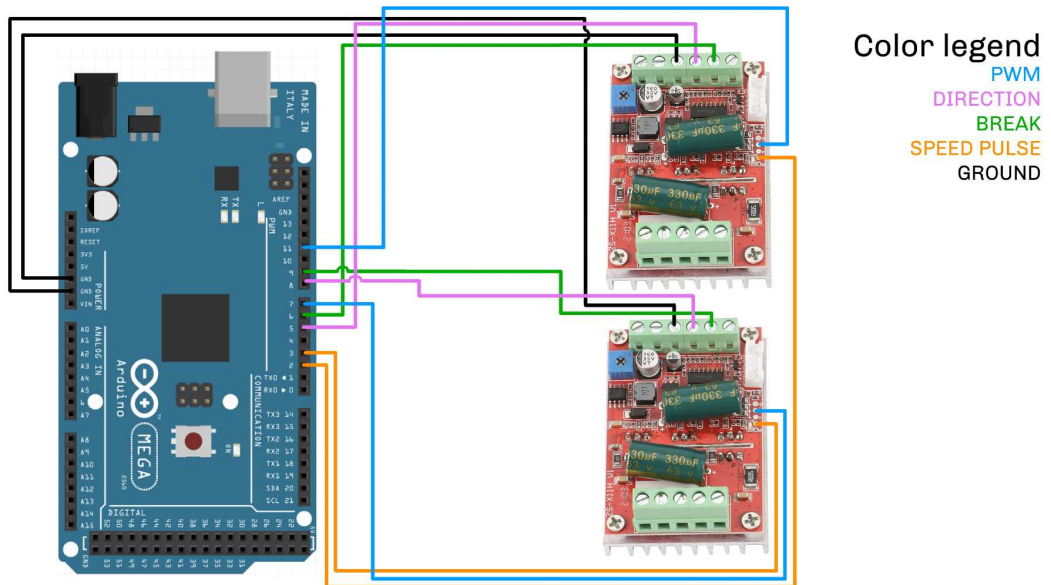
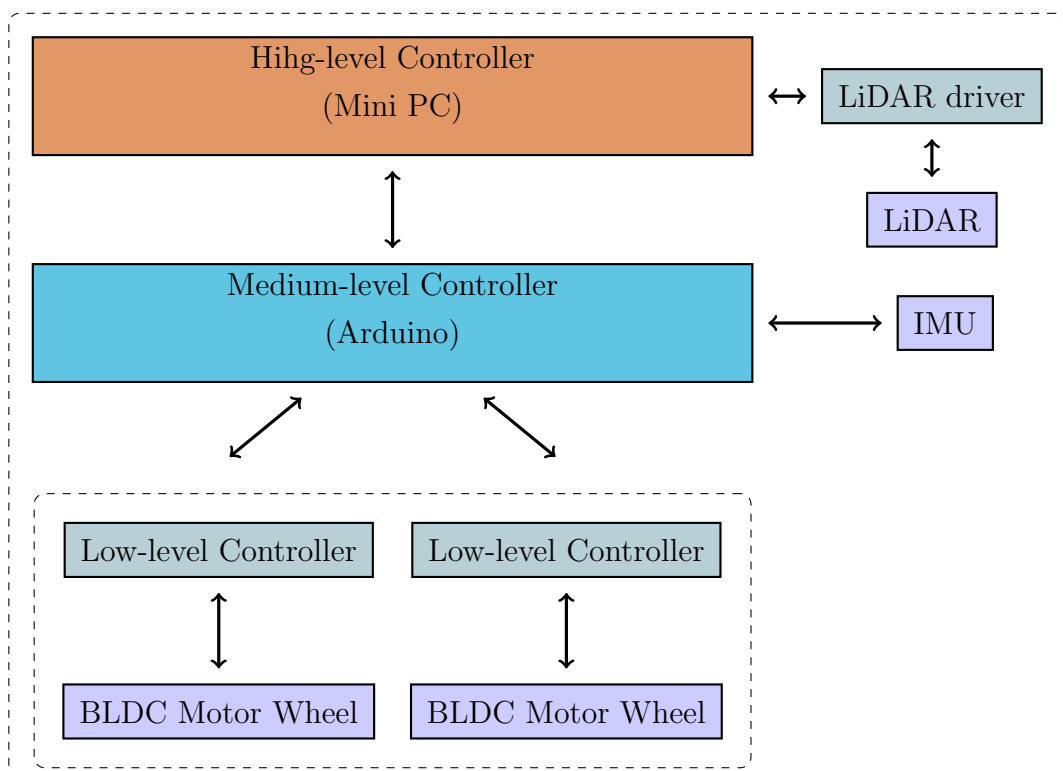


Figure 2.41: Schematic of the connection between the Arduino and the 2 motor drivers for controlling the wheels.

Chapter 3

Control

The control part is divided in 3 different levels



The **High-level Controller** manages all the robot operations through the use of ROS (see appendix E) nodes:

- Sends motion commands to the Medium-level Controller through the use of ROS topics.

- Receives odometry information from the Medium-level Controller through the use of ROS topics.
- Takes values from the LiDAR sensor.
- Process a 3D map of the space and locates the robot inside the map (using a SLAM algorithm).

The **Medium-level Controller** interacts with the BLDC motor wheels through the communication with the BLDC motor drivers (the Low-level Controllers).

- Translates the motion commands received from the upper level into PWM signals sent to the motor drivers.
- Senses the speed of rotation of each wheel received from the motor drivers.
- Implements a PI controller on the speed of rotation of each wheel to reach the set point value (**Speed Controller**).
- Implements a winding up action on the control voltage values sent to the DC motor wheels (acceleration and deceleration ramps).

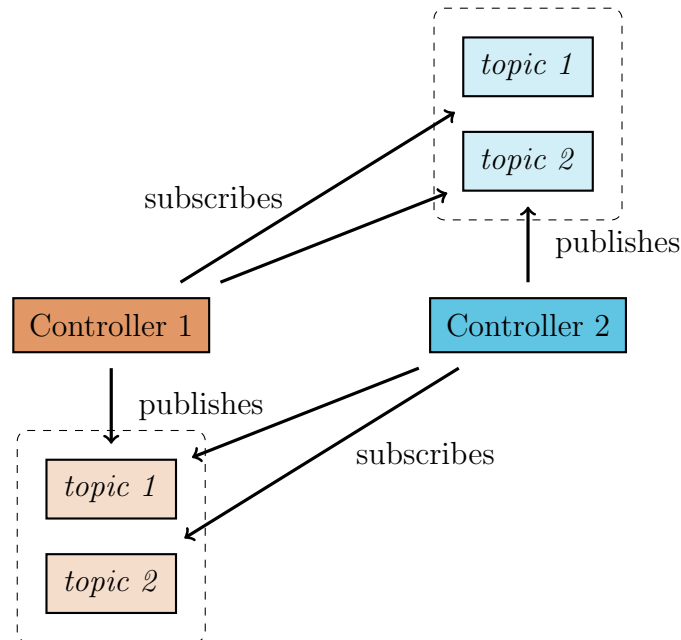
The **Low-level Controller** powers the phasis of each motor in order to reach the reference speed of rotation, managing the current sent to each phasis. It detects the wheel's speed of rotation through the use of the hall sensor, directly connected to the driver.

3.1 Communication between Controllers

The communication between the High-Level Controller and the Medium-Level Controller is managed by **ROS nodes**, running in both the controllers.

The mini PC sends and receives data from the Arduino through the use of **ROS topics**, using a *publisher-subscriber* architecture (see appendix G). Each controller publishes its topics and it subscribes to the topics published by the other controller. In this way the exchange of information is made possible.

This method of exchanging data has been chosen in order to let other ROS nodes reading the information published, at any time.



The `rosserial_arduino` package has been used for integrating the Arduino environment with ROS, indeed for using the ROS library in the Arduino project.

3.2 Speed Control

The robot's cartesian speed of motion is controlled by the **Medium-level Controller**. Since the kinematics of the robot consists of two differential drivers, it is needed to control the speed of rotation of each wheel.

Each motor is controlled through the motor driver using a *Pulse Width Modulated* (**PWM**) signal (see appendix B). By knowing the maximum velocity of each wheel, the speed value can be converted into duty cycle used for generating the driving PWM signal.

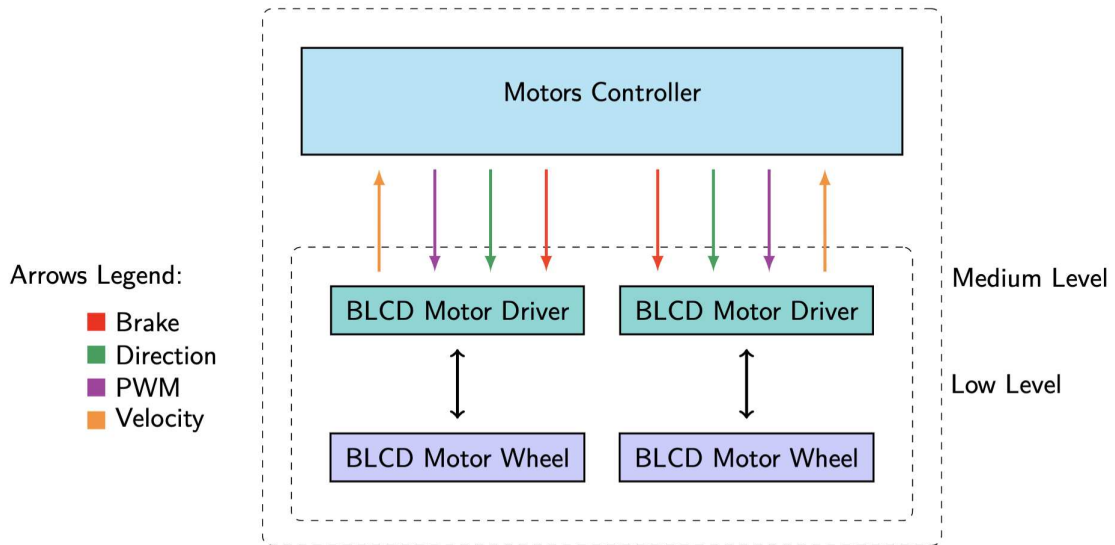


Figure 3.1: This block diagram explains the communication between the Arduino and the motors drivers.

The *PWM signal* drives the wheel in the forward (clockwise) or backward (anti-clockwise) direction, depending on the value of the *Direction signal*.

The *Brake signal* is used for stopping the spin of the wheel, indeed for stopping the wheel also in free spinning (when the value of this signal is a logic 1 the wheel is not able to rotate anymore).

The *Velocity signal* is used as feedback for knowing if the wheel is spinning at the correct velocity (measured in R_{pm}).

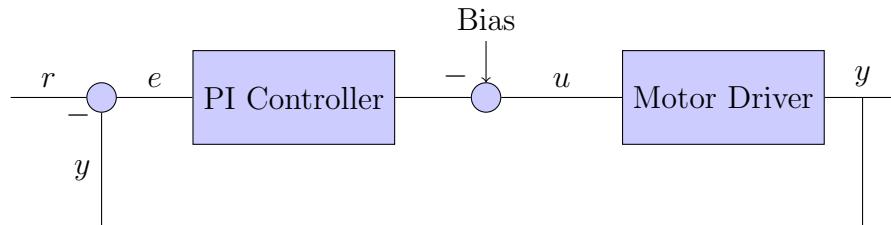
3.2.1 PI Controller

It is needed to implement a **PI Controller** for perfect tracking of the reference speed value, which is forwarded to the motor driver through a *PWM signal*. The PI controller will mainly work in the following contexts:

- The wheel is spinning on a surface with friction. This can be seen as a disturbance for the rotation of the wheel, not allowing the perfect tracking of the reference speed.

- The wheel is spinning on a inclined surface: the gravity force will cause an undesired acceleration of the wheel.

The final **closed loop control system** is depicted in the following scheme:



- r : reference speed of rotation of the wheel (in R_{pm}).
- u : final *control input signal* obtained by subtracting a $Bias$ value from the controller output.
- y : sensed speed of rotation by the motor driver (in R_{pm}), used as feedback value for computing the error, e , input of the controller.

The PI controller has been chosen because it helps in reducing both the rise time and the steady state errors of the system. A derivative component has not been used since a derivative controller is required to minimise the transient errors like overshoot and oscillations in the output of the plant. But this can create heavy instability in noisy environments, like the warehouse in which the robot will work (characterized by friction of the ground, slope etc.).

3.2.2 High-Level Implementation

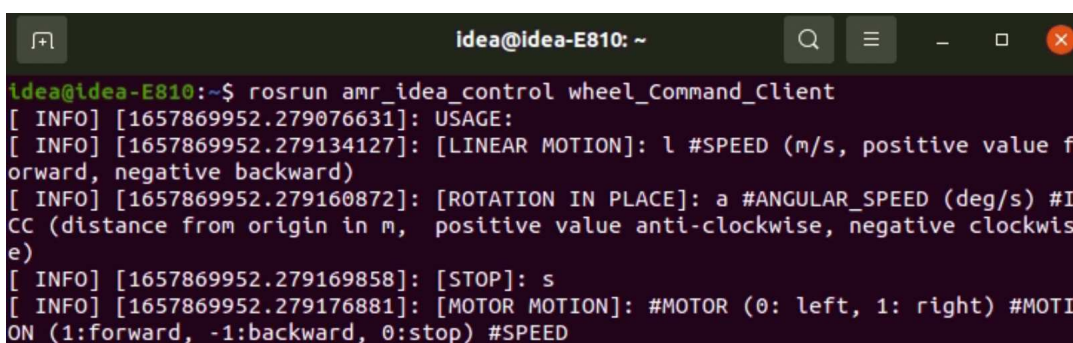
The High-level Controller runs two ROS nodes which communicate using a *client-server* approach (see appendix F, code in section 4.2.1):

- **wheel_Command_Service**: ROS service that sends data to the Medium-Level controller. It waits for data from a client. The data packet contains the following information (**service format** data):
 - **motionL**: determines the sense of rotation of the left wheel; 1 for forward rotation, -1 for backward rotation, 0 for braking the rotation.
 - **speedL**: determines the speed of rotation of the left wheel (expressed in RPM).

- **motionR**: determines the sense of rotation of the right wheel.
- **speedR**: determines the velocity of rotation of the right wheel.
- **wheel_Command_Client**: ROS client that sends data to the *wheel_Command_Service* by invoking the *requestCommand* method with the necessary data (previously explained).

The client is run by terminal and it implements a user friendly menu with the following options:

- *Linear motion*: for a linear movement of the robot backward or forward of a specified speed (expressed in *meter/second*).
- *Rotation in place*: for a rotation of the robot with respect to an ICC point with a specified angular speed (expressed in *degree/second*). If the distance of the ICC value is positive it is performed a clock-wise rotation (since the ICC is placed in the proximity of the right wheel) and viceversa. The computations performed are the ones explained in section 2.1.3.
- *Stop*: for stopping the robot's wheels. The brake signals explained in section 2.2.1 are set to a logical 1.
- *Motor motion*: for controlling a single motorized wheel by expressing the motor number, the sense of rotation and the desired speed of rotation (0 for the left motor, 1 for the right motor).



```

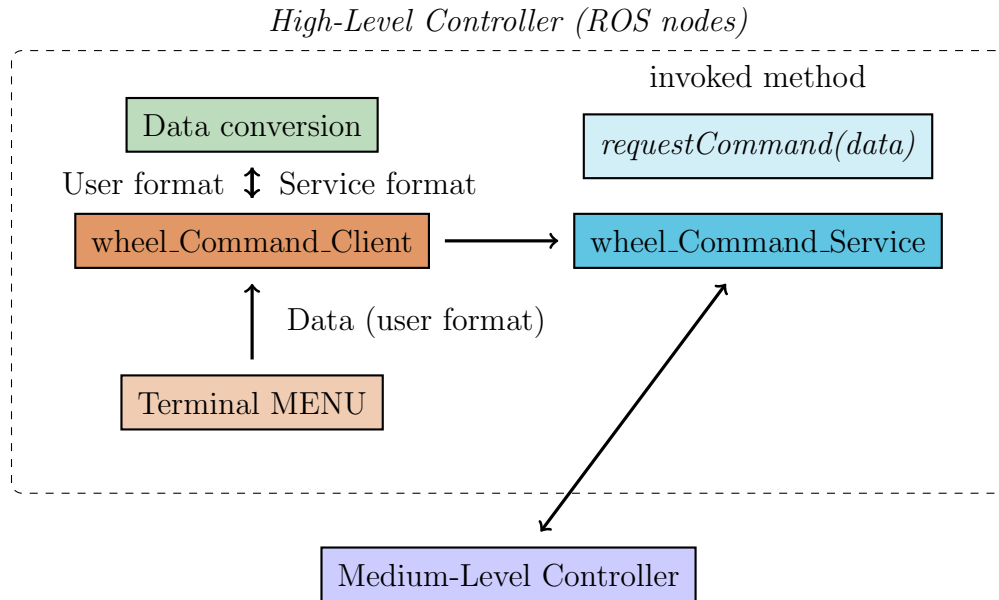
idea@idea-E810: ~
[ INFO] [1657869952.279076631]: USAGE:
[ INFO] [1657869952.279134127]: [LINEAR MOTION]: l #SPEED (m/s, positive value forward, negative backward)
[ INFO] [1657869952.279160872]: [ROTATION IN PLACE]: a #ANGULAR_SPEED (deg/s) #ICC (distance from origin in m, positive value anti-clockwise, negative clockwise)
[ INFO] [1657869952.279169858]: [STOP]: s
[ INFO] [1657869952.279176881]: [MOTOR MOTION]: #MOTOR (0: left, 1: right) #MOTION (1:forward, -1:backward, 0:stop) #SPEED

```

Figure 3.2: `wheel_Command_Client` menu.

The data received from the terminal menu (**user format** data) are converted in the correct unit and sent to the **wheel_Command_Service** by invoking the specific method. The client performs a control on the data before of invoking the method, checking if the desired speed of ro-

tation is inside a range between two values (MIN_RPM and MAX_RPM), otherwise an error is shown.



3.2.3 Medium-Level Implementation

The communication between the High-Level Controller and the Medium-Level Controller is managed through the use of **ROS topics** as explained in section 3.1.

The **wheel_Command_Service** (run as ROS node on the mini PC) publishes a ROS topic called **wheel_Command**. The data of this topic is a vector of *std_msgs/Float64.msg* (data type used in the ROS environment for transmitting float value). This vector contains 4 data in the following order:

- (a) *Sense of rotation of left wheel*
- (b) *Speed of rotation of left wheel (RPM)*
- (c) *Sense of rotation of right wheel*
- (d) *Speed of rotation of right wheel (RPM)*

The **wheel_Command_Service** publishes this information as soon as it receives the data from the **wheel_Command_Client**.

```

idea@idea-E810: ~
rostopic echo wheel_Command
layout:
  dim: []
  data_offset: 0
data: [1.0, 38.19718551635742, 1.0, 38.19718551635742]

```

Figure 3.3: Data of the `wheel_Command` topic for achieving a forward linear motion of 0.5 meter/second.

```

idea@idea-E810: ~
rostopic echo wheel_Command
layout:
  dim: []
  data_offset: 0
data: [-1.0, 45.83662414550781, -1.0, 45.83662414550781]

```

Figure 3.4: Data of the `wheel_Command` topic for achieving a backward linear motion of 0.6 meter/second.

```

idea@idea-E810: ~
rostopic echo wheel_Command
layout:
  dim: []
  data_offset: 0
data: [1.0, 40.5333282470703, 1.0, 66.1333129882812]

```

Figure 3.5: Data of the `wheel_Command` topic for achieving an anticlockwise rotation of 40 degree/second, with the ICC placed 1 meter left with respect to the mid length point of the wheels axis.

```

idea@idea-E810: ~
rostopic echo wheel_Command
layout:
  dim: []
  data_offset: 0
data: [1.0, 76.39437103271484, 1.0, 0.0]

```

Figure 3.6: Data of the `wheel_Command` topic for controlling a single motor (the left wheel) in order to reach a velocity of 1 meter/second.

```

idea@idea-E810: ~
rostopic echo wheel_Command
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.0, 0.0, 0.0]

```

Figure 3.7: Data of the `wheel_Command` topic for stopping the wheels.

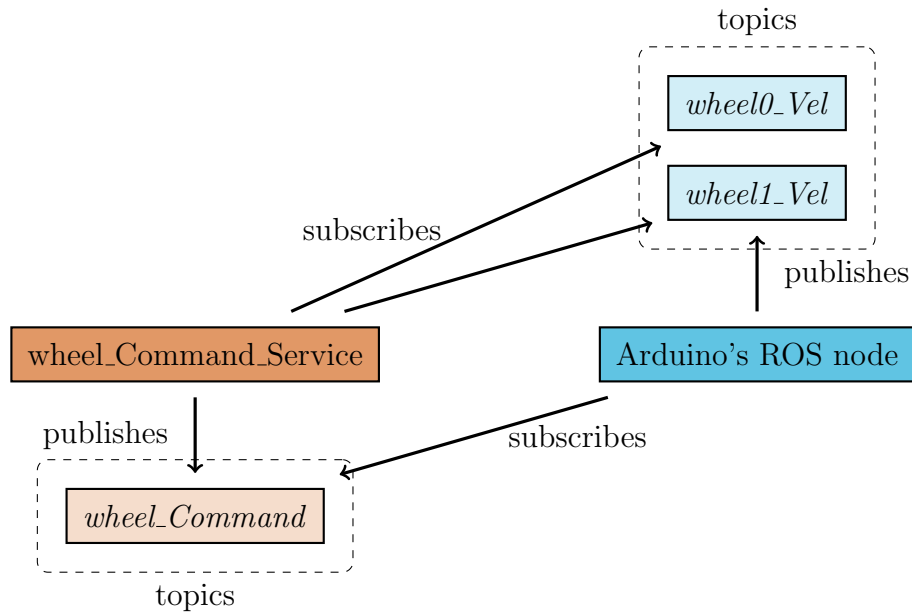
When it is necessary to perform a rotation in place, indeed one wheel is spinning and the other wheel is not spinning, also the sense of motion of the not spinning wheel is set to 1 (forward motion), or -1 (backward motion), as depicted in figure 3.6. This is due to the fact that the Arduino would stop the wheel if a 0 value is set in the sense of motion, not making it to free spin. The rotation in place would not be possible. The PI controller running on the Arduino does not operate when a wheel has a sense of motion 1 or -1 with a 0 value for the speed velocity. This are the conditions of *free spinning*. In this conditions the PI controller has not to track the reference speed for that specific wheel.

The Medium-Level Controller (the *Arduino MEGA* board) publishes two ROS topic:

- **wheel0_Vel**: used for publishing the speed of rotation of left wheel.
- **wheel1_Vel**: used for publishing the speed of rotation of right wheel.

The data type of both topics is a *std_msgs/Float64.msg* variable. The speed is expressed in RPM (**service format** data). For each wheel, the Arduino publishes the information through the use of an interrupt: the digital pins 2 and 3 of the board are used for catching the velocity information coming from the motor drivers. When the interrupt is triggered 90 times, it is possible to compute the speed of rotation of the wheel (this due to the functioning of the speed pulse output of the motor driver, explained in section 2.2.1). When the velocity is known, it is published on the respective topic.

The overall scenario is the following:



PI Controller implementation

There are two different ways for implementing a PI controller in a program:

- (a) **Polling method:** the program continuously checks if the actual speed of rotation is equal to the reference value (plus or minus a threshold), in case of negative answer the PI controller executes the control action.
- (b) **Timer interrupt method:** at each specified interval of time (defined by a timer) the program checks if the actual speed of rotation is equal to the reference value (plus or minus a threshold), in case of negative answer the PI controller executes the control action.

The *polling method* is surely more accurate because it guarantees to have an immediate control action when it is needed. Contrary, this method involves widely the computation of the microcontroller: its normal operations are delayed because of the repeated polling. For this reason the *timer interrupt method* has been implemented.

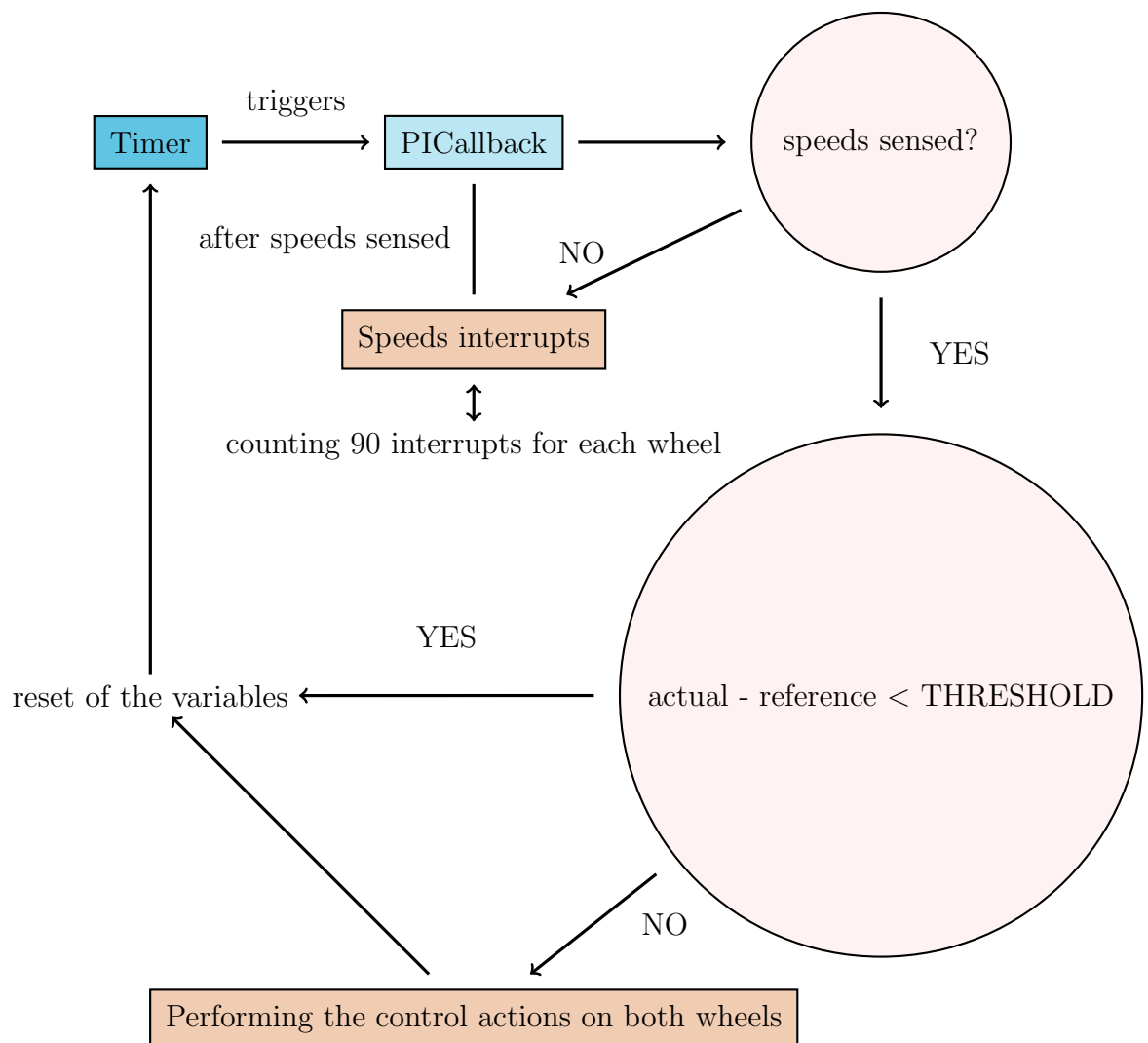
A timer is defined in the setup method of the Arduino program, every 0.2 seconds it triggers an interrupt which invokes the **PICallBack** function.

This function checks if both the velocities of the wheels have been sensed. In the negative case, it attaches two interrupts used for catching the speed

pulse signals coming from the motor drivers. When the velocities are collected from the motor drivers, indeed when for each wheel 90 interrupts have been triggered, the sensed flags are switched to true and the interrupts used for catching the speed pulses are detached.

The next time that the timer interrupt is triggered, the velocities have been sensed so the PI Controller computes the control actions, if it is needed, indeed two new velocities signals are sent to each motor driver in order to reach the reference speeds.

Then, the PI controller waits that the speeds of the two wheels are sensed again to compute the new control actions. The explained computation flow is repeated until the velocity of the wheel is equal to the reference value plus or minus a threshold. The functioning scheme is the following:



3.2.4 Acceleration Ramp

It is necessary to perform a winding process of the voltages that control the DC motors. If the motor driver suddenly requires a speed of rotation much major of the actual speed, the DC motor wheel may get damaged because it is suddenly powered with an higher voltage.

For this reason, the Medium-Level Controller perform a winding up action on the control signals. It creates a linear ramp of acceleration. It evaluates the difference of speed between the actual and the reference speed, then it acts considering the value obtained.

Considering the module of this difference, it is performed either and acceleration or a deceleration ramp.

$$|\text{actual speed} - \text{reference speed}| = \text{delta speed}$$

In the following it is shown the winding algorithm used.

Algorithm 1 Acceleration ramp Algorithm

Input: Actual Speed, Reference Speed

Output: Control Speed

```

1: if delta speed > THRESHOLD then
2:
3:   if delta speed > THRESHOLD_HIGH_ACC then
4:     delay time = DELAY_HIGH_ACC
5:   else if delta speed > THRESHOLD_MEDIUM_ACC then
6:     delay time = DELAY_MEDIUM_ACC
7:   else if delta speed > THRESHOLD_SMALL_ACC then
8:     delay time = DELAY_SMALL_ACC
9:   else
10:    Motor control with no acceleration ramp
11:  end if
12:  if delta speed > THRESHOLD_SMALL_ACC then
13:    Motor Control with acceleration ramp
14:  end if
15: else
16:  No action performed
17: end if

```

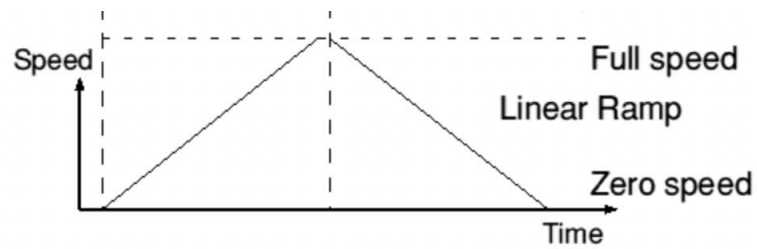


Figure 3.8: Linear acceleration/deceleration ramp generated for controlling the DC motor wheels.

If the *delta speed* is below the *THRESHOLD* value (set to 1.5 RPM), the speed of rotation of the wheel is considered correct, indeed the reference value is reached (with a small error).

If the *delta speed* is below the *THRESHOLD_SMALL_ACC* value, no acceleration ramp is executed and the DC motor wheel is controlled directly either with the voltage value relative to the reference speed value or with the voltage value relative to the control action. Otherwise an acceleration/deceleration is performed depending on the difference value.

Chapter 4

Code

The programming language used is **C++** (see appendix H).

4.1 Workspace

The code files of the projects are divided in:

- **High-Level Controller files:** compiled and executed by the mini PC for running the ROS framework. The PC uses **Ubuntu Linux** where it is installed **ROS 1 Noetic version**.

The source code files are located in the *src* folder of the *Catkin Workspace* (ROS's workspace).

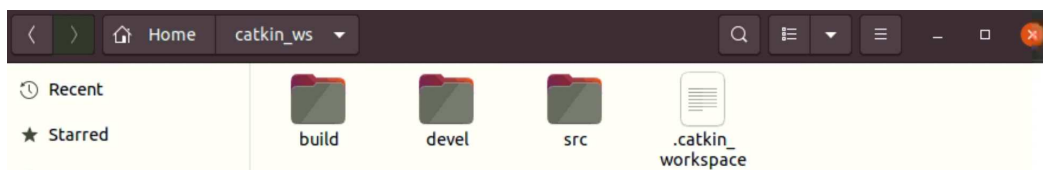


Figure 4.1: Contents of the Catkin Workspace folder

The *src* folder contains the ROS projects of the workspace. The *amr_idea* folder contains the code files of the robot.

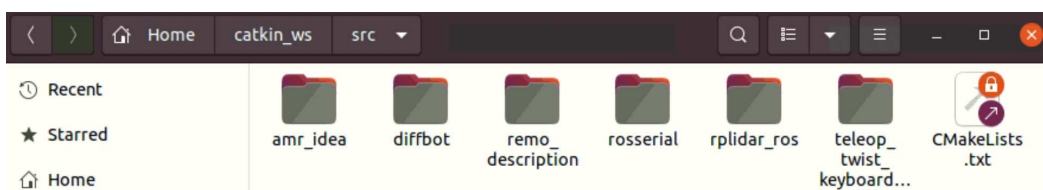


Figure 4.2: Content of the src folder in the Catkin Workspace

The project is divided in two ROS sub-projects:

- (a) *amr_idea_control*: contains the High-Level implementation control code.
- (b) *amr_idea_bringup*: contains the code for running all the *amr_idea* project at the booting of the mini pc.

The *amr_idea_bringup* project contains the Autostart ROS service, which executes the launch file of the project at the booting of the mini pc. The **launch** file of a ROS project sets up the working environment, indeed it runs the ROS nodes when executed.

```
1
2 <launch>
3
4   <node name="wheel_Command_Service"          pkg="
5     amr_idea_control"          type="wheel_Command_Service
6     " output="log" >
7
8   </node>
9
10  <rosparam file="$(find amr_idea_control)/config/
11    diff_driver.yaml" command="load" />
12  <node name="serial_node"          pkg="
13    rosserial_arduino"          type="serial_node.py">
14    <param name="port"          type="string"
15      value="/dev/ttyACM0"/>
16    <param name="baud"          type="int"
17      value="115200"/>
18
19  </node>
20
21 </launch>
```

Listing 4.1: Launch file for Autostart ROS service of the *amr_idea_bringup* project

The launch file runs the **wheel_Command_Service** ROS node and the **rosserial_arduino** ROS node for allowing the communication among the High-Level and the Medium-Level Controller. This last execution requires the information about the serial port of communication and the baud rate value.

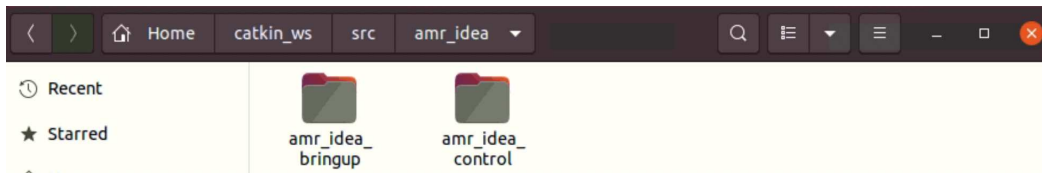


Figure 4.3: Sub-projects of the workspace.

- **Medium-Level Controller files:** compiled and executed by the Arduino board at the booting of the microcontroller.

4.2 Speed Control

4.2.1 High-Level Implementation

The *wheel.cpp* file contains utility methods. Its header contains the definition of functional parameters.

```

1
2 #ifndef wheel.h
3 #include "math.h"
4
5 #define MAX_VEL 1054 //(RPM) evaluated considering that 0,38
   Amp --> 71,5 RPM . Max amps are 250/36 = 6,944
6 #define WHEEL_DIAM 0.25
7 #define THRESHOLD 0.01
8 #define R 0.24 //distance between the center of the wheel
   and the middle point of the robot
9 #define L 0.48 //distance between the center of the two
   wheels
10 #define FORWARD 1
11 #define BACKWARD -1
12 #define STOP 0
13
14 double computeRPM(double speed);
15
16 double computeRadFromDeg(double vel);
17
18 #endif

```

Listing 4.2: *wheel.h* file

The mini PC executes two ROS nodes:

- `wheel_Command_Service`
- `wheel_Command_Client`

Both have their own `.cpp` file and header `.h` associated.

`wheel_Command_Service`

The `sendCommand` function publishes the `wheel_Command` topic with the motion information. It requires the data in **service format**.

The `requestCommand` function is the service function invoked by the client for requesting the motion of the wheels. It reads the data of the request and it invokes the `sendCommand` method for achieving the motors motions, by sending the data to the Medium-Level Controller through the ROS topic.

```

1 #include "ros/ros.h"
2 #include "std_msgs/Float64MultiArray.h"
3 #include <iostream>
4 #include <sstream>
5 #include "amr_idea_control/RequestPacket.h"
6
7
8
9 ros::Publisher publisher;
10
11 void sendCommand(int motionL, float speedL, int motionR, float
    speedR)
12 {
13     std_msgs::Float64MultiArray msg;
14     msg.data.resize(4);
15     msg.data[0]=motionL;
16     msg.data[1]=speedL;
17     msg.data[2]=motionR;
18     msg.data[3]=speedR;
19     publisher.publish(msg);
20     ros::spinOnce();
21
22     ROS_INFO("Data sent");
23 }
24 bool requestCommand(amr_idea_control::RequestPacket::Request
    &req, amr_idea_control::RequestPacket::Response &res)
25 {

```

```
26
27     sendCommand(req.motionL, req.speedL, req.motionR, req.
28               speedR);
29     return true;
30 }
31
32
33 int main(int argc, char **argv)
34 {
35     ros::init(argc, argv, "wheel_Command_Service");
36     ros::NodeHandle n;
37     publisher=n.advertise<std_msgs::Float64MultiArray>("
38               wheel_Command", 100);
39     ros::ServiceServer service = n.advertiseService("
40               wheel_Command_Service", requestCommand);
41     ROS_INFO("wheel_Command_Service is running");
42     ros::spin();
43
44     return 0;
45 }
```

Listing 4.3: *wheel_Command_Service.cpp* file

wheel_Command_Client

In the following are listed the three main functions used in the *wheel_Command_Client* for sending a request of a particular wheel motion (as explained in section 3.2.2).

```
1 int linear(float speed)
2 {
3     ros::NodeHandle n;
4     float rpm=speed*60.0/(WHEEL_DIAM*PI);
5     if (checkSpeed(rpm))
6     {
7         int motion;
8         if (rpm>=0)
9             motion=FORWARD;
10        else
11            motion=BACKWARD;
12        rpm=abs(rpm);
```

```

13     client = n.serviceClient<amr_idea_control::RequestPacket
14     >("wheel_Command_Service");
15
16     srv.request.motionL=motion;
17     srv.request.motionR=motion;
18     srv.request.speedL=rpm;
19     srv.request.speedR=rpm;
20
21     if (!client.call(srv))
22     {
23         ROS_ERROR("ERROR while linear motion");
24         return -1;
25     }
26
27 }else
28 {
29     ROS_ERROR("ERROR: linear velocity not valid");
30     return 1;
31 }
32
33 return 0;
34 }

```

Listing 4.4: Linear motion function of *wheel_Command_Client.cpp* file

```

1
2 int angular(float speed,float icc)
3 {
4     ros::NodeHandle n;
5
6     float radSpeed=speed*PI/180.0;
7     float vel [2];
8     vel [0]=radSpeed*(icc-R)*60.0/(WHEEL_DIAM*PI);
9     vel [1]=radSpeed*(icc+R)*60.0/(WHEEL_DIAM*PI);
10    client = n.serviceClient<amr_idea_control::RequestPacket
11    >("wheel_Command_Service");
12
13    if (abs(icc)<R)
14    {
15        if (vel [0]>0)
16            srv.request.motionL=FORWARD;
17        else
18            srv.request.motionL=BACKWARD;

```



```

19     if (vel [1]>0)
20         srv.request.motionR=FORWARD;
21     else
22         srv.request.motionR=BACKWARD;
23 }else
24 {
25     srv.request.motionL=FORWARD;
26     srv.request.motionR=FORWARD;
27 }
28
29     vel [0]=abs(vel [0]);
30     vel [1]=abs(vel [1]);
31
32     if (checkSpeed(vel [0])==0)
33     {
34         ROS_ERROR("ERROR: angular velocity not valid");
35         return 1;
36     }
37     srv.request.speedL=vel [0];
38     if (checkSpeed(vel [1])==0)
39     {
40         ROS_ERROR("ERROR: angular velocity not valid");
41         return 1;
42     }
43
44     srv.request.speedR=vel [1];
45     if (!client.call(srv))
46     {
47         ROS_ERROR("ERROR while angular motion");
48         return -1;
49     }
50
51     return 0;
52 }

```

Listing 4.5: Angular motion function of *wheel_Command_Client.cpp* file

```

1 int stop()
2 {
3     ros::NodeHandle n;
4
5     client = n.serviceClient<amr_idea_control::RequestPacket>
6     >("wheel_Command_Service");
7     srv.request.motionL=0;
8     srv.request.motionR=0;

```

```
8     srv.request.speedL=0;
9     srv.request.speedR=0;
10    if (!client.call(srv))
11    {
12        ROS_ERROR("ERROR while stop motion");
13        return -1;
14    }
15
16
17    return 0;
18 }
```

Listing 4.6: Stop motion function of *wheel_Command_Client.cpp* file

The three methods return an integer value:

- **0**: the request has been executed successfully.
- **1**: a value error has occurred. It has been require a wrong motion with a speed value out of the allowed range.
- **-1**: an error has occurred during the invocation of the *wheel_Command_Service* function.

4.2.2 Medium-Level Implementation

The code of this implementation is contained into the Arduino project.

As in the code of the High-Level Implementation (explained in section 4.2.1), it is defined a *wheel.cpp* and *wheel.h* files for utility methods and parameters relative to the wheels.

```
1
2 #ifndef wheel.h
3 #define Morse_h
4 #include "math.h"
5 #include "Arduino.h"
6
7 #define MAX_VEL 1054 //(RPM) evaluated considering that 0,38
8     Amp --> 71,5 RPM . Max amps are 250/36 = 6,944
9 #define WHEEL_DIAM 0.25
10 #define THRESHOLD 0.01
11 #define R 0.24 //distance between the center of the wheel
12     and the middle point of the robot
```

```
11 #define L 0.48 //distance between the center of the two
    wheels
12 #define FORWARD 1
13 #define BACKWARD -1
14 #define STOP 0
15
16
17 double computeDuty(double speed); //return the
    corresponding duty cycle knowing the speed (meter over
    second)
18
19 double computeDutyFromRPM(double rpm);
20
21 double computeRPM(double speed);
22
23 double computeVolt(double refDuty); //return the
    corresponding voltage knowing the duty cycle
24
25 double computeReal(double speed,String direction); //
    compute a input speed to be given to the motor in order
    to reach the speed inserted by the user
26
27 double computeSpeedFromSignal(double val);
28
29 double computeSpeedFromRPM(double rpm);
30
31 #endif
```

Listing 4.7: *wheel.h* file of the Arduino project

It has been created a **Motor** class for managing each motor wheel using the signals attached to the digital pins of the Arduino connected to the motor driver. Two instances of this class are created: one for the left DC motor and one for the right DC motor.

```
1
2 #ifndef motor.h
3 #include "wheel.h"
4
5 #define THRESHOLD_CONTROL 4
6 #define THRESHOLD_SMALL_ACC 0.005
7 #define THRESHOLD_MEDIUM_ACC 0.009
8 #define THRESHOLD_HIGH_ACC 0.011
9 #define DELAY_SMALL_ACC 50
10 #define DELAY_MEDIUM_ACC 150
```

```
11 #define DELAY_HIGH_ACC 250
12
13
14 class Motor{
15 public:
16     int PIN_STOP;
17     int PIN_BREAK;
18     int PIN_DIRECTION;
19     int PIN_CONTROL;
20     int PIN_SPEED;
21     int motion= STOP;
22
23     //in RPM
24     double referenceSpeed=0.0;
25     double actualSpeed=0.0;
26     double drivenSpeed=0.0;
27
28     bool sensed=false;
29     bool controlled=false;
30
31
32     Motor(int stops,int breaks,int dirs,int control,int
speed);
33     void stop();
34     void start();
35     void forward();
36     void backward();
37     void changeDirection();
38     void drive(double referenceSpeed);
39     void driveNoAcc(double drivSpeed);
40     void acceleration(double initialSpeed,double finalSpeed
);
41 };
42
43 #endif
```

Listing 4.8: *motor.h* file of the Arduino project

The constructor of the Motor class requires as input the values of the digital pins attached to the motor driver's signals.

The driving action is executed knowing the RPM desired for each wheel, which is converted into a duty cycle value and then into a voltage value.

```
1 double computeDutyFromRPM(double rpm)
2 {
3     return rpm/MAX_VEL;
4 }
5
6 double computeVolt(double refDuty)
7 {
8     double val=refDuty*255.0;
9     return refDuty*255; //Since duty = 1 corresponds to 5 Volt
10    which is analogWrite 255
11 }
```

Listing 4.9: Utility functions for values conversion.

Knowing the voltage value (from 0V to 5V) it is possible to generate a PWM signal using the *analogWrite* function of the Arduino library.

Driving functions

The motor class contains three methods used for driving the DC motor wheels.

```
1
2 void Motor::drive(double drivSpeed)
3 {
4     Motor::start();
5     if (motion==FORWARD)
6     {
7         forward();
8     }
9     else if (motion==BACKWARD)
10    {
11        backward();
12    }
13    else
14        return;
15    drivenSpeed=drivSpeed;
16    Motor::acceleration(actualSpeed,drivenSpeed);
17 }
18
19 void Motor::driveNoAcc(double drivSpeed)
20 {
```

```
21     drivenSpeed=drivSpeed;
22     double finalDuty=computeDutyFromRPM(drivSpeed);
23     analogWrite(PIN_CONTROL, computeVolt(finalDuty));
24 }
25
26 void Motor::acceleration(double initialSpeed, double
finalSpeed)
27 {
28
29     double initialDuty=computeDutyFromRPM(initialSpeed);
30     double finalDuty=computeDutyFromRPM(finalSpeed);
31     double deltaDuty=(finalDuty-initialDuty)/10;
32
33     int delayTime;
34
35     if (abs(deltaDuty)>=THRESHOLD_HIGH_ACC)
36         delayTime=DELAY_HIGH_ACC;
37     else if (abs(deltaDuty)>=THRESHOLD_MEDIUM_ACC)
38         delayTime=DELAY_MEDIUM_ACC;
39     else if (abs(deltaDuty)>=THRESHOLD_SMALL_ACC)
40         delayTime=DELAY_SMALL_ACC;
41
42     if (deltaDuty>=THRESHOLD_SMALL_ACC)
43     {
44
45         for (int i=1; i<=10; i++)
46         {
47             analogWrite(PIN_CONTROL, computeVolt(initialDuty+
deltaDuty*i));
48             delay(delayTime);
49
50         }
51     }else
52         Motor::driveNoAcc(finalSpeed);
53
54 }
```

Listing 4.10: Driving functions of the Motor class

- **drive**: turns on the motor, sets the sense of motion and invokes the acceleration method.
- **acceleration**: computes the *delta speed* value between the actual speed and the reference value (expressed in duty cycle and normal-

ized in 10 units) and checks which kind of acceleration/deceleration ramp has to be used (explained in section 3.2.4).

If the *delta speed* value is below every threshold, no acceleration is performed and the `driveNoAcc` method is invoked.

Otherwise it is performed a cycle with 10 instants of time (each last *delayTime* seconds); at each interval a control action signal is sent to the motor, with a speed equals to the initial value plus an extra value that depends on the *delta speed* and the number of instants of time passed.

- **driveNoAcc**: sends a control action signal to the motor driver, considering the reference speed value.

PI Control Action

The method that performs the control action, **PI_Control**, is defined in the Arduino's project main file: *motor_control.ino*. The method is invoked by the **PI_Callback** function triggered by the timer (as explained in section 3.2.3).

The function requires as input the reference speed value (*setPoint*), the actual speed of rotation (*sensedOutput*) and the index of the motor (0 for the left motor, 1 for the right motor).

```
1 void PI_Control(double setPoint, double sensedOutput, int
   index){
2   noInterrupts();
3   motors[index].sensed=false;
4   if (setPoint!=0)
5   {
6     unsigned long current_time = millis(); //returns the
       number of milliseconds passed since the Arduino started
       running the program
7     int delta_time = current_time - last_time[index]; //
       delta time interval
8
9     if (delta_time >= T){
10
11     if(setPoint!=previousSetPoint[index])
```

```
12     {
13         total_error[index]=0;
14         previousSetPoint[index]=setPoint;
15     }
16
17     double error = setPoint - sensedOutput;
18
19     if (abs(error)<=THRESHOLD_CONTROL)
20         return;
21
22     total_error[index] += error; //accumalates the error -
23     integral term
24     if (total_error[index] >= max_control) total_error[
25     index] = max_control;
26     else if (total_error[index] <= min_control)
27     total_error[index] = min_control;
28
29     double delta_error = error - last_error[index]; //
30     difference of error for derivative term
31     double bias=setPoint/10;
32     double controlSpeed=setPoint+(kp*error + (ki*T)*
33     total_error[index] + (kd/T)*delta_error)-bias;
34
35     if (controlSpeed >= max_control) controlSpeed =
36     max_control;
37     else if (controlSpeed <= min_control) controlSpeed =
38     min_control;
39
40     last_error[index] = error;
41     last_time[index] = current_time;
42     motors[index].driveNoAcc(controlSpeed);
43 }
44 motors[index].controlled=true;
45 interrupts();
46 }
```

Listing 4.11: Function that performs the control action of the PI Controller.

The interrupts are disabled during the computation of the control action for ensuring an immediate control of the motorized wheel.

The gain values of the PI Controller have been tuned performing real tests with the DC motor wheels. The best values obtained are the following:

- **Proportional gain:** $K_P = 0.001$:
- **Integral gain:** $K_I = 0.15$:

Speed Interrupts

As explained in section 3.2.3, the **PICallback** function attaches the speed interrupts for both the DC motor wheels.

```
1 void speedChangeInt0(){
2   speedChangeInt(0);
3 }
4
5 void speedChangeInt1(){
6   speedChangeInt(1);
7 }
8
9 void speedChangeInt(int index){ //Interrupt for speed
10   //changing of wheel 0
11
12   if (!motors[index].sensed)
13   {
14     if (counter[index]==0)
15     {
16       lastRead[index]=micros();
17       counter[index]++;
18       reader[index]=true;
19     }
20     else if (counter[index]==90)
21     {
22       detachInterrupt(digitalPinToInterrupt(motors[
23 index].PIN_SPEED));
24       noInterrupts();
25       actualRead[index]=micros()-lastRead[index];
26       lastRead[index]=micros(); //Evaluating the time
27 the pulse is on
28
29       double a=actualRead[index];
30       motors[index].actualSpeed=computeSpeedFromSignal(
31 a); //Computing the speed considering the time the pulse
32 is on
33 }
```

```
28         toPub_vel[index].data=motors[index].actualSpeed;
           //speed value to publish on the ROS topic
29
30         //publishing the velocity
31         publisher[index].publish(&toPub_vel[index]);
32         node.spinOnce();
33
34         counter[index]=0;
35         reader[index]=false;
36         motors[index].sensed=true;
37         motors[index].controlled=false;
38         interrupts();
39     }
40     else
41     {
42         counter[index]++;
43     }
44 }
45 }
```

Listing 4.12: Speed interrupts.

When an interrupt occurs the `speedChangeInt` method is invoked. It counts the number of state of changes occurred for each motor wheel. When 90 changes are counted, the speed is evaluated considering the duration in time of the last pulse (the speed computation is performed by the `computeSpeedFromSignal` method). The value is published on the ROS topic and the `sensed` flag relative to that specific motor wheel is set to true, so that the PI Control action can be computed, if it is needed, when the **PICallback** function is triggered again by the timer.

Chapter 5

Tests and Results

Some tests were performed in order to ensure the reliability of the robot.

- It has been ensured that all the hardware components turn on/off correctly as soon as the battery switch (explained in section 2.2.5) is set on/off.
- It has been tested that the power supply charges correctly
- The speed controller and the kinematics of the robot have been tested.

5.1 Speed Controller

The Speed Controller tests were executed in the following steps:

- (a) On each single DC motor wheel, before that the robot was built.
- (b) Simultaneously on the two DC motor wheels, in order to tests the effective kinematics of the AMR.



Figure 5.1: Work station used for testing the functioning of each BLDC motor wheel.

The wheels behaviours between step *a* and *b* were slightly different. In fact in case *a* the wheels did not have a load; in case *b* the wheels sustained the robot's components weights, so they acted with a load.

The following tests were performed using the final built of the robot. It was sent a motion command using the `wheel_Command_Client`. The velocities of each wheel were tracked by reading the `wheel0_Vel` and `wheel1_Vel` ROS topics. The motion command can be considered as an input step signal to the system. The response is a **Step Response** in the time domain (see appendix I).

```
idea@idea-E810:~$ rostopic echo wheel0_Vel
data: 0.7741804718971252
---
data: 38.40383529663086
---
data: 54.69222640991211
---
data: 64.38127899169922
---
data: 69.4566650390625

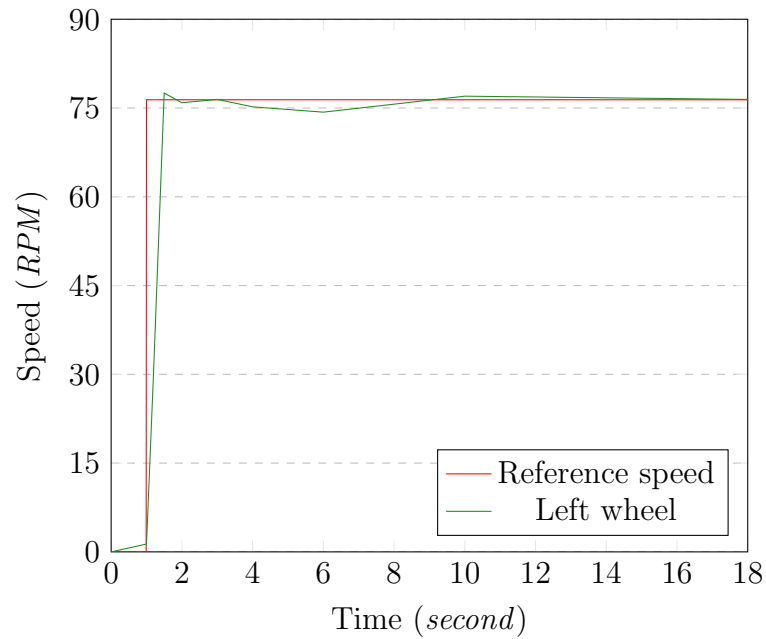
idea@idea-E810:~$ rostopic echo wheel1_Vel
data: 0.7386925220489502
---
data: 14.91486644744873
---
data: 11.774402618408203
---
data: 11.560506820678711
---
data: 13.538041114807129
```

Figure 5.2: ROS topics used for reading the speed of rotation (in RPM) of each wheel.

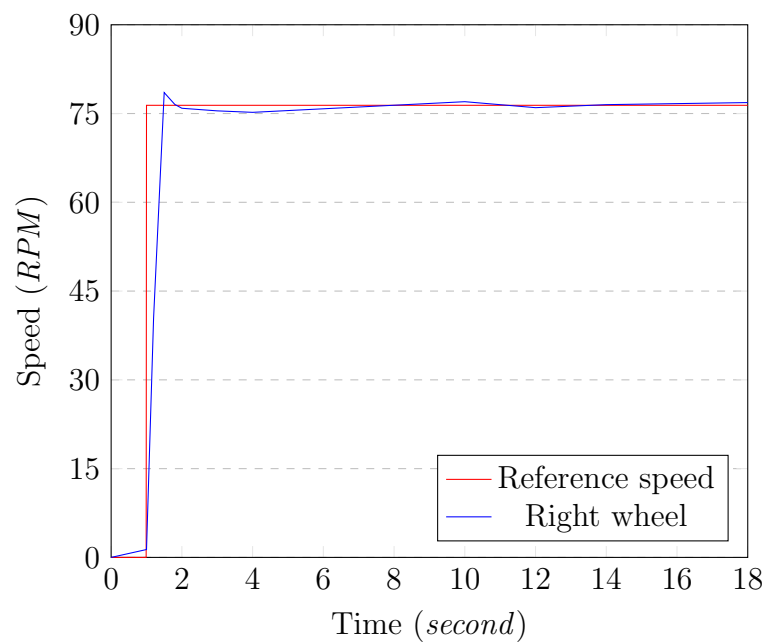
The tests were performed in the IDEA's warehouse.

5.1.1 Single Wheel Motion

COMMAD: Left wheel, forward, 1 (*meter/second*) (76.39 RPM), tracked for 18 seconds.

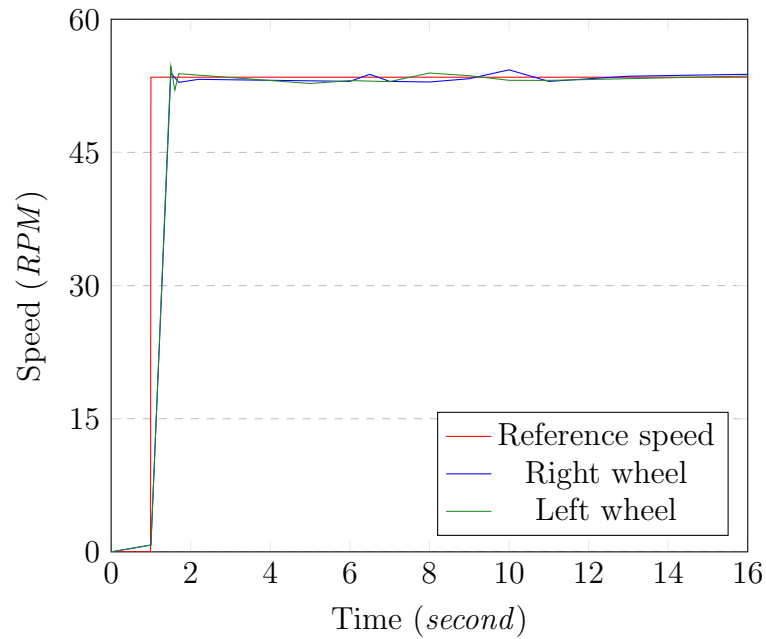


COMMAD: Right wheel, forward, 1 (*meter/second*) (76.39 RPM), tracked for 18 seconds.

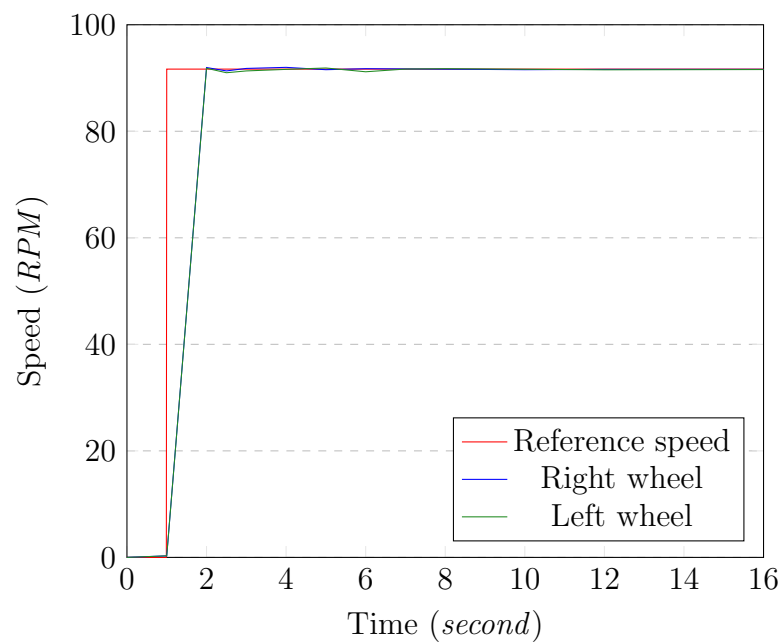


5.1.2 Linear Motion

COMMAND: Linear motion, forward, 0.7 (*meter/second*, 53.47 RPM), tracked for 16 seconds.



COMMAND: linear motion, backward, 1.2 (*meter/second*, 91.67 RPM), tracked for 16 seconds.

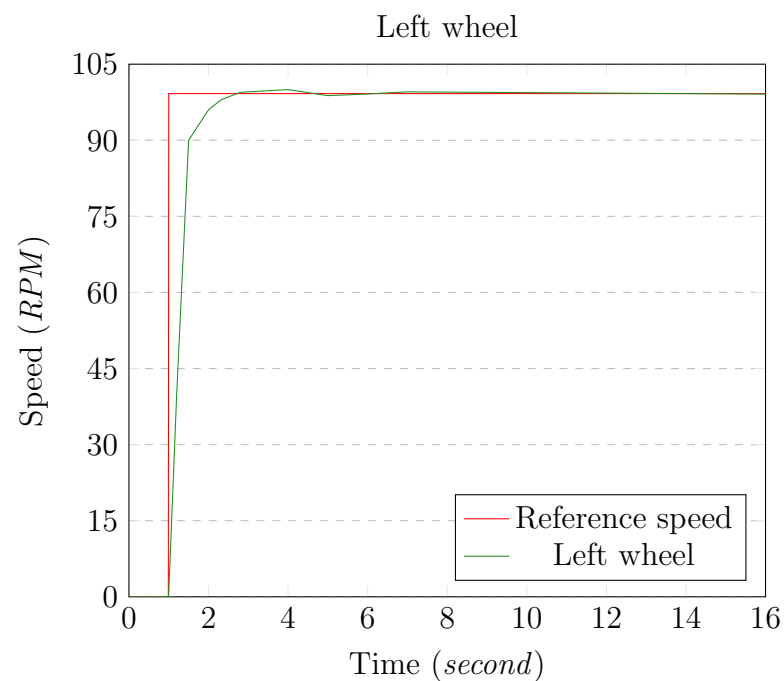


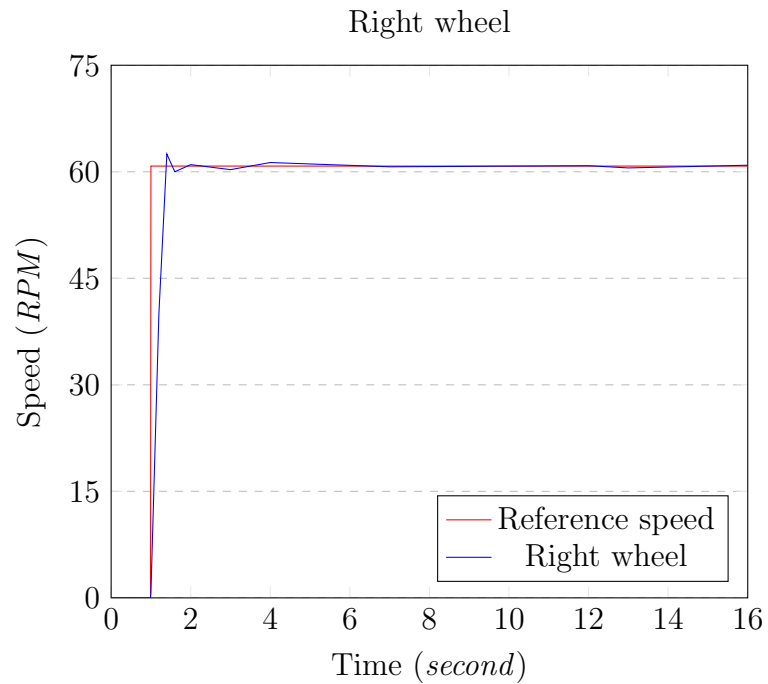
5.1.3 Angular Motion

COMMAND: Angular motion, forward, 60 (*deg/second*), ICC 1 meter away on the right from the mid length point of the wheel axis (clockwise rotation), tracked for 16 seconds.

The computed nominal speed are:

- **Left wheel:** 99.20 RPM
- **Right wheel:** 60.80 RPM

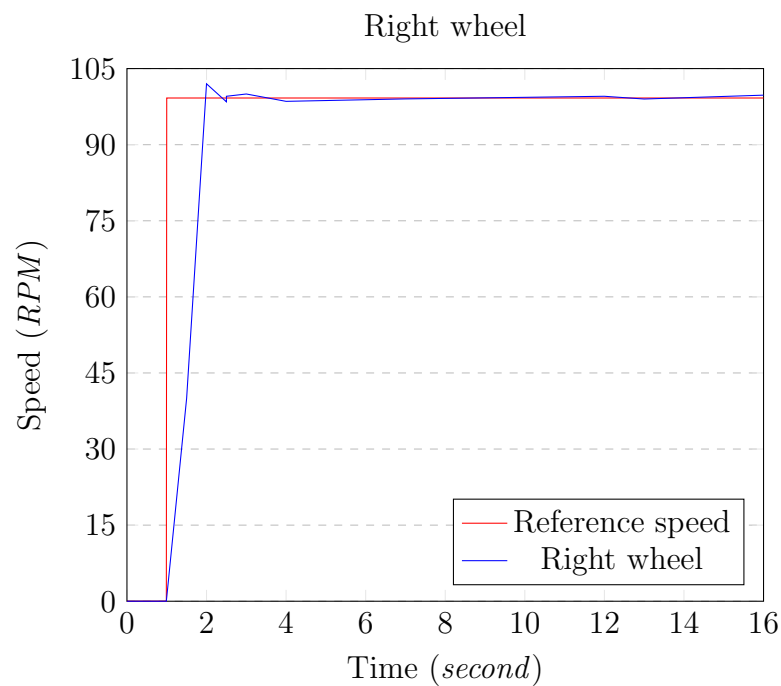
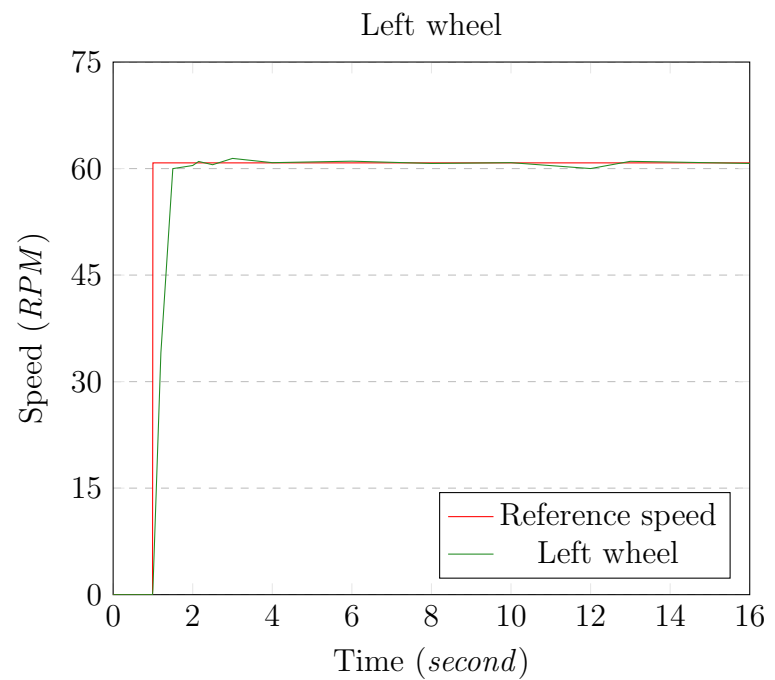




COMMAND: Angular motion, forward, 60 (*deg/second*), ICC 1 meter away on the left from the mid length point of the wheel axis (anticlockwise rotation), tracked for 16 seconds.

The computed nominal speed are:

- **Left wheel:** 60.80 RPM
- **Right wheel:** 99.20 RPM



Results Analysis

As shown in the plots, the Speed Controller acts correctly, allowing an almost perfect track of the reference value for the speed of rotation of each wheel.

Looking at the plots it is possible to determine the time domain specifications of the step responses (see appendix I).

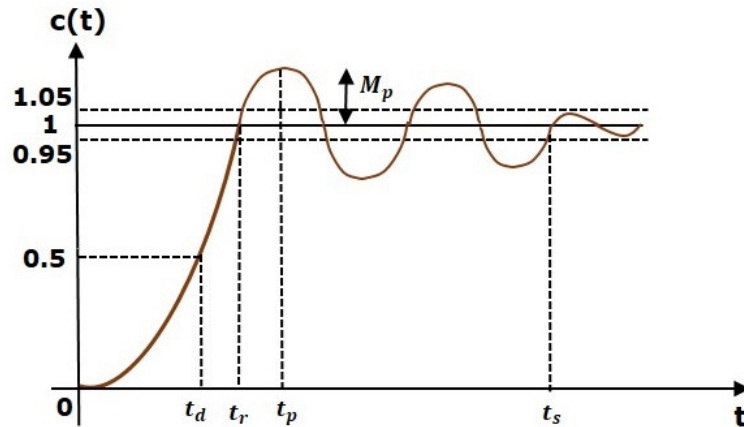


Figure 5.3: Time domain specifications of a step response.

- **Delay Time:** almost equals to 0.2 seconds for responses to step valued at maximum 75 RPM; almost equals to 0.5 seconds for responses to step with higher value.
- **Rise Time:** almost equals to 0.5 seconds for responses to step valued at maximum 75 RPM; almost equals to 1 seconds for responses to step with higher value.
- **Peak Time:** almost equals to 0.6 seconds for responses to step valued at maximum 75 RPM; almost equals to 1.1 seconds for responses to step with higher value.
- **Settling Time:** in the worst case it is equal to 1.3 seconds.
- **Maximum Overshoot:** the overshoot is mostly present in the right wheel, of 4 RPM; in the left wheel it is generally not present or it is lower (about 2 RPM).

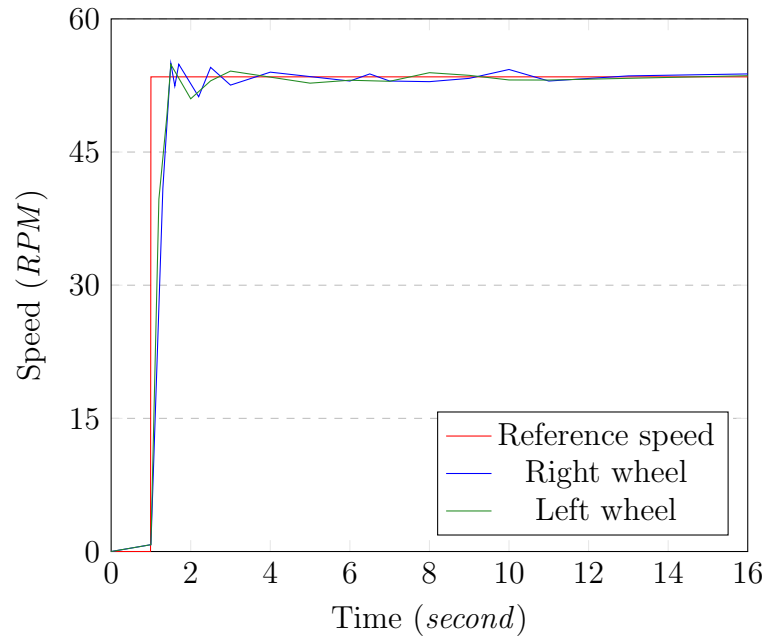
The performances of the right DC motor wheel are worse with respect to the left one. This can be caused by a factory issue. In fact, the engine of the left wheel sounds slightly different when the wheel is controlled.

The control speed value signal has a direct effect on the time domain specifications (delay time, rise time, peak time and settling time). This is caused by the different acceleration ramp used, depending on the *delta_speed* value (explained in section 3.2.4)

5.1.4 Payload Test

Tests with a payload of 20 kg were performed.

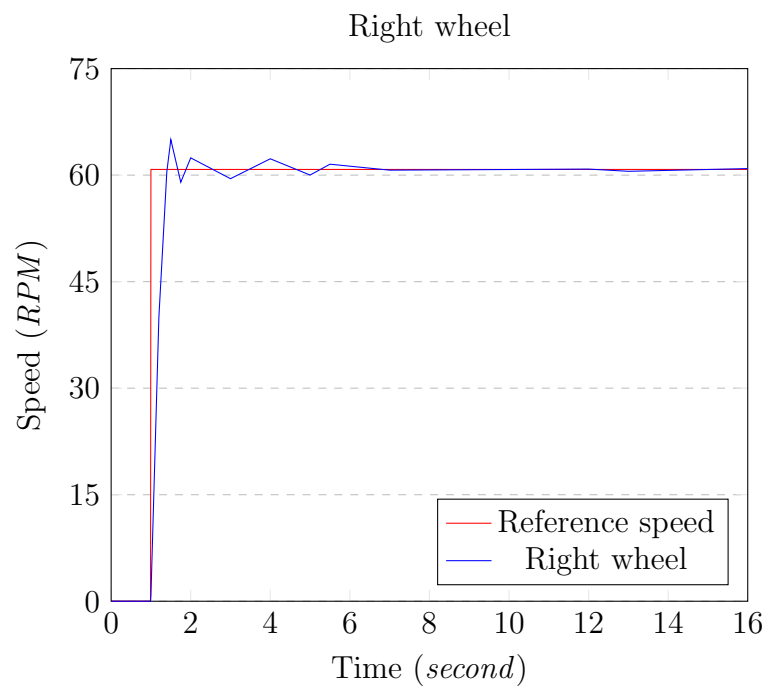
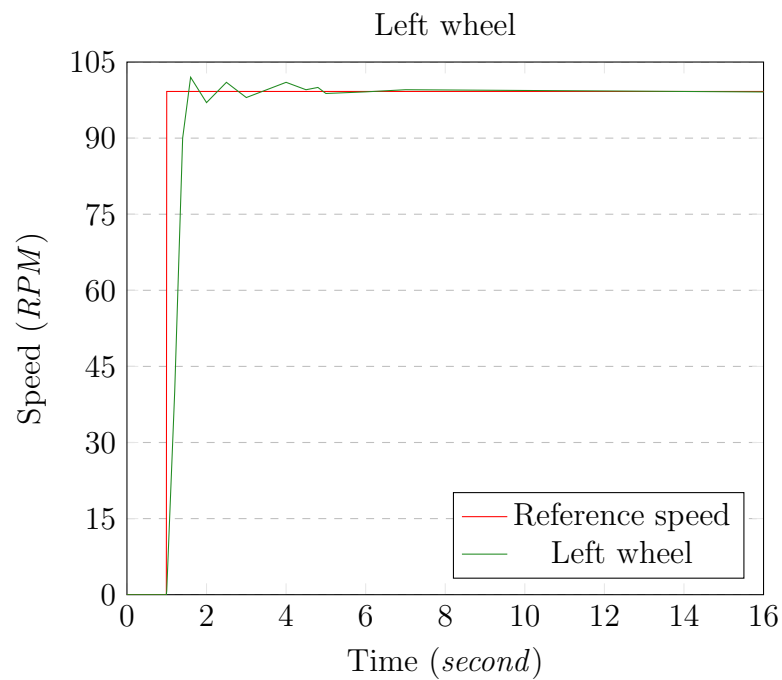
COMMAND: Linear motion, forward, 0.7 (*meter/second*, 53.47 RPM), tracked for 16 seconds.



COMMAND: Angular motion, forward, 60 (*deg/second*), ICC 1 meter away on the right from the mid length point of the wheel axis (clockwise rotation), tracked for 16 seconds.

The computed nominal speed are:

- **Left wheel:** 99.20 RPM
- **Right wheel:** 60.80 RPM



Result Analysis

The payload affects the performances, causing more oscillations during the transient time. The original control action is not sufficient to reach the reference speed. The PI Controller computes a control action to compensate the effect of the payload. It is needed to power the motor's phases with an

higher voltage for tracking the reference signal, indeed a PWM signal with a greater duty cycle has to be used for controlling the motor wheel. The research phase of the correct control signal is the one characterized by the oscillations of the response.

Chapter 6

Future Implementations

The robot requires others implementations and integrations in order to be used in the industrial context. During the traineeship, shipping delays of the robot's components and hardware issues have compromised the developpe and implementations of the following points:

- Integration of the **LiDAR** sensor with the ROS nodes of the robot.
- Development of the **SLAM algorithm**.
- Integration of the SLAM algorithm with the **Multi-Sensor Fusion** method.
- Development of the path planning and obstacle avoidance algorithms based on the map built by the SLAM algorithm.

6.1 SLAM

SLAM stands for *Simultaneous Localization and Mapping*. It is an algorithm that allows a robot to build its surrounding map and localize its location on the map at the same time. The map can be used for different tasks, such as path planning, obstacle avoidance etc.

Generally, SLAM uses devices and sensors for collecting:

- **Visible data**: from cameras.
- **Non-Visible data**: from radar, sonar, LiDAR.

It is possible to build a picture (2D or 3D) of the surrounding environment by using the collected data.

6.1.1 Functioning

The SLAM algorithm can be broke down into **Front-End data collection** and **Back-End data processing**.

Front-End Data Collection

The *Front-End data collection* of SLAM is of two types **Visual SLAM** and **LiDAR SLAM**.

The LiDAR SLAM has been chosen for the implementation since lasers (such as LiDAR) are more precise and accurate with respect to cameras. The rate of data collection is also much higher, making the solution suitable for use in high-speed applications.

Back-End Data Processing

The data of the sensor is then used for the *Back-End data processing* task. The output data of LiDAR sensors is called point cloud data; it is available with 2D (x, y) or 3D (x, y, z) positional information.

The laser sensor point cloud provides high-precision distance measurements. The movement of the robot is estimated sequentially by matching different point clouds given by the laser. The calculated movement (travelled distance) is used for localizing the vehicle. For LiDAR point cloud matching, iterative closest point (ICP) and normal distributions transform (NDT) algorithms are used. 2D or 3D point cloud maps can be represented as grid maps.

RPLiDAR A1

The **Slamtec RPLiDAR A1** is the sensor chosen for the application (as shown in section 2.2.2), it supports 2000/4000 samples per second. There

exists a ROS package (*rplidar_ros*) that provides basic device handling for 2D Laser Scanner RPLIDAR A1/A2 and A3 models.

The driver of the RPLiDAR reads RPLiDAR raw scan result using RPLiDAR's SDK and converts the data to a ROS LaserScan message. In fact the RPLiDAR's driver publishes `sensor_msgs/LaserScan` data on the scan ROS topic.

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header:          # timestamp in the header is the acquisition time of
                        # the first ray in the scan.
                        #
                        # in frame frame_id, angles are measured around
                        # the positive Z axis (counterclockwise, if Z is up)
                        # with zero angle being forward along the x axis

float32 angle_min       # start angle of the scan [rad]
float32 angle_max       # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment  # time between measurements [seconds] - if your scanner
                        # is moving, this will be used in interpolating position
                        # of 3d points
float32 scan_time       # time between scans [seconds]

float32 range_min       # minimum range value [m]
float32 range_max       # maximum range value [m]

float32[] ranges        # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities   # intensity data [device-specific units]. If your
                        # device does not provide intensities, please leave
                        # the array empty.
```

Figure 6.1: ROS LaserScan RAW data definition.

Two services, `start_motor` and `stop_motor`, are provided by a ROS node for starting and stopping the RPLiDAR A1 sensor.

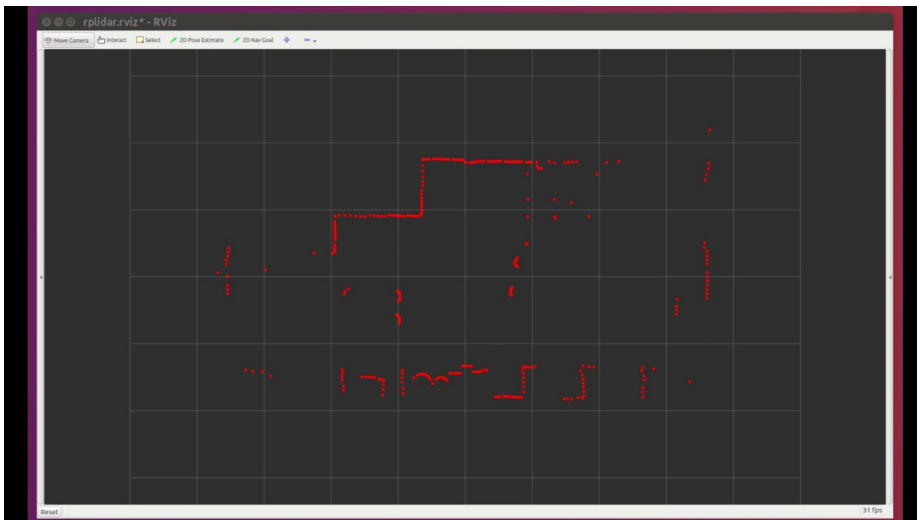


Figure 6.2: RPLiDAR data scan opened using RVIZ and the RPLIDAR's SDK

6.2 Multi-Sensor Fusion Method

In order to overcome environmental problems and reduce positioning errors, different types of sensors can be used for improving positioning accuracy and apply it to SLAM. There are three types of sensor information: odometer (given by the motors), inertial measurement (given by the IMU sensor) and ultra-wideband (UWB information). The data fusion is performed through an **Extended Kalman Filter**. Depending on the sensors used and the way in which the information is combined, there exist different architectures for performing the fusion method. The one depicted in figure 6.3 has been chosen.

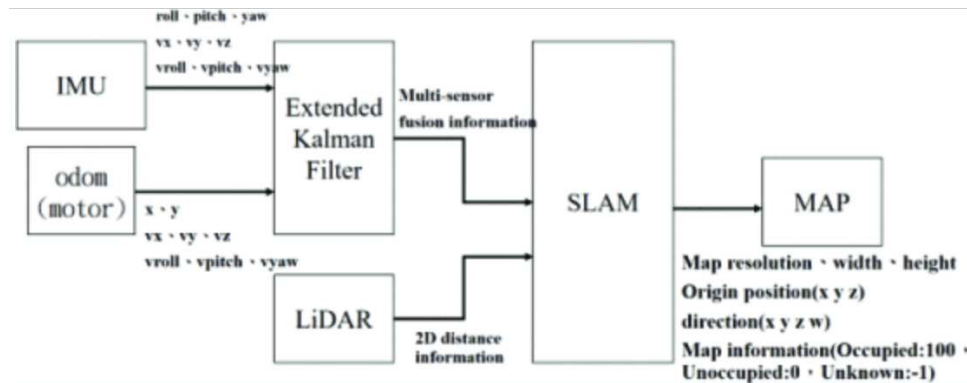


Figure 6.3: Multi-Sensor Fusion method approach chosen for the SLAM algorithm.

6.2.1 Extended Kalman Filter

Kalman Filtering, also known as *linear quadratic estimation* (LQE), is an algorithm that uses a series of measurements observed over time, including statistical noise and other inaccuracies, and produces estimates of unknown variables.

The Kalman filter model assumes the true state at time k is evolved from the state at $(k - 1)$ according to:

$$x_k = F_k x_{k-1} + B_k u_k + w_k \quad (6.1)$$

Where:

- x_k : is the state of the system at instant k .
- F_k : is the state-transition matrix.
- B_k : is the input-transition matrix.
- u_k : is the control input at instant k .
- w_k : is a gaussian noise at instant k with zero mean and Q_k covariance.
 $w_k \sim \mathcal{N}(0, Q_k)$.

Considering that at instant k an observation z_k of the true state x_k is taken. It follows the equation:

$$z_k = H_k * x_k + v_k \quad (6.2)$$

Where:

- H_k : is the observation state matrix that maps the true state space in the observer space.
- v_k : is the observation noise assumed to be $v_k \sim \mathcal{N}(0, R_k)$.

Considering:

- $x_{k|k}$: the a posteriori state estimate at time k given observations up to instant k .
- $P_{k|k}$: the a posteriori estimate covariance matrix

The predicted state estimate is :

$$\hat{x}_{(k|k-1)} = F_k x_{(k-1|k-1)} + B_k u_k$$

Predicted estimate covariance:

$$\hat{P}_{(k|k-1)} = F_k P_{(k-1|k-1)} F_k^T + Q_k$$

Update state estimate:

$$\hat{x}_{(k|k)} = \hat{x}_{(k|k-1)} + K_k \tilde{y}_k \quad (3)$$

With:

K_k : Kalman gain at instant k

\tilde{y}_k : is the innovation at instant k , $y_k = z_k - H_k \hat{x}_{k|k-1}$

Optimal Kalman gain:

$$K_k = \hat{P}_{(k|k-1)} H_k^T S_k^{-1} \quad (4)$$

Update estimate covariance:

$$P_{(k|k)} = (I - K_k H_k) \hat{P}_{(k|k-1)} \quad (5)$$

The **Extended Kalman Filter** works following the previous equations. It is generated an estimate of the position of the robot with an estimated noise. Then, the information from the sensors is used for updating the position information. Those steps are repeated recursively to get a more accurate result.

6.2.2 Other Architectures

Four different fusion methods are compared:

- (a) **Gmapping SLAM**: the SLAM construction requires information from the odometer (motor) and LiDAR. The odometer (motor) provides x , y , v_x , v_y , v_z , v_{roll} , v_{pitch} , v_{yaw} . LiDAR provides 2D distance information.

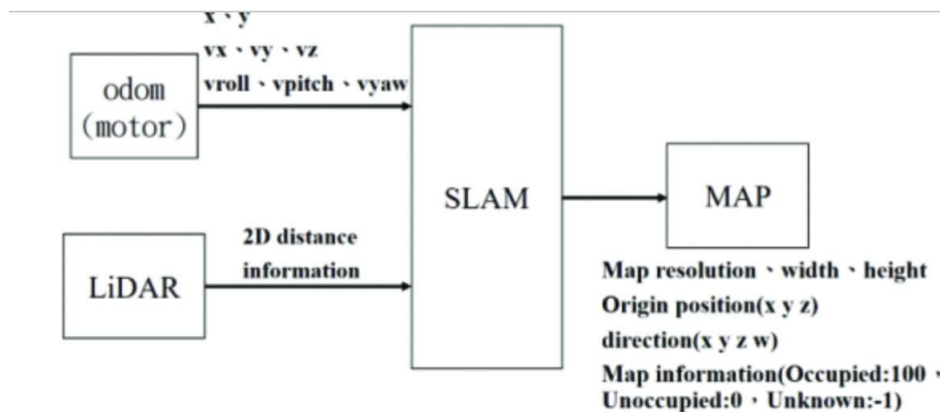


Figure 6.4: Gmapping SLAM architecture

- (b) **Method 1: odom + IMU fusion positioning architecture.**

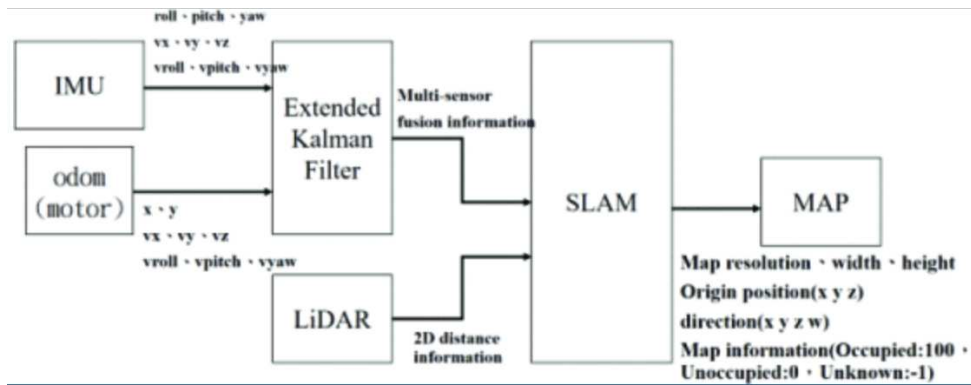


Figure 6.5: Odom + IMU fusion positioning architecture.

(c) Method 2: odom + UWB fusion positioning architecture.

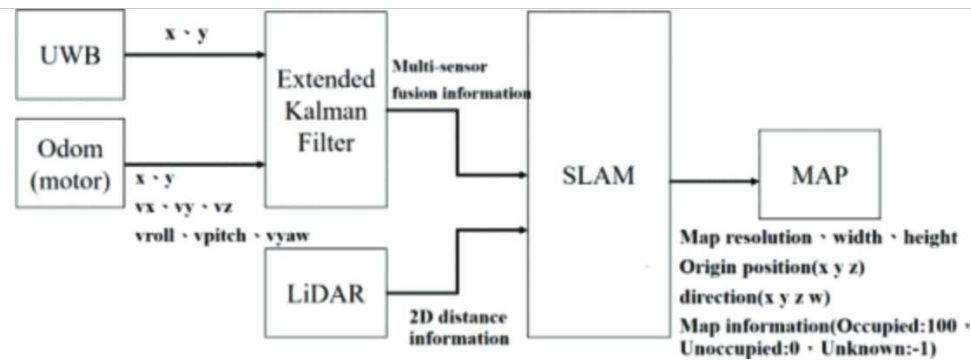


Figure 6.6: Odom + UWB fusion positioning architecture.

(d) Method 3: odom + UWB + IMU fusion positioning architecture.

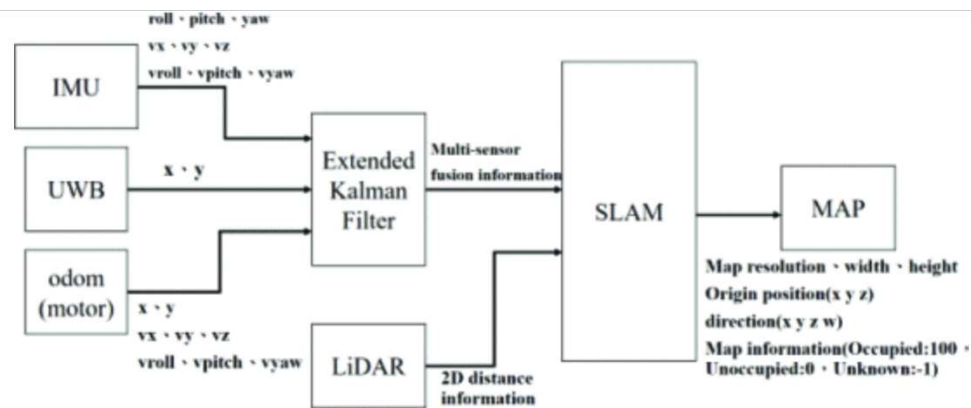


Figure 6.7: Odom + UWB + IMU fusion positioning architecture.

Researchers have compared the four methods described, obtaining the following results:

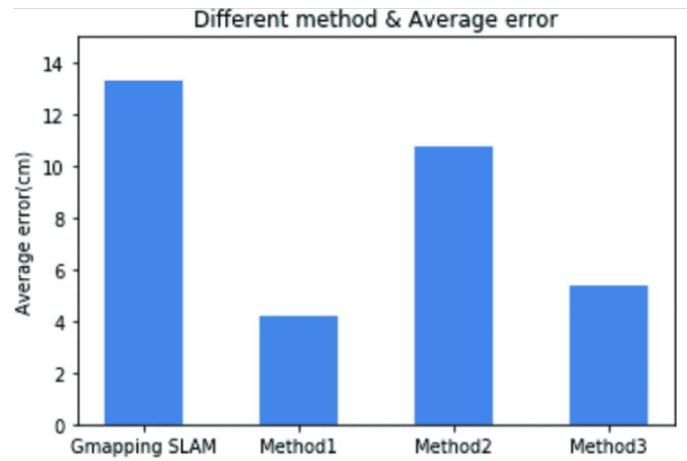


Figure 6.8: Comparisons of the 4 methods described.

Bibliography

- [1] BLDC Motor drivers <https://mad-ee.com/easy-inexpensive-hoverboard-motor-controller/>
- [2] Xiaomi M365 Wheel schematic <https://imgur.com/gallery/rGhSpSN>
- [3] ROS IMU and Arduino integration <https://atadiat.com/en/e-ros-imu-and-arduino-how-to-send-to-ros/>
- [4] Yongcheng Du;Changsheng Ai;Zhiquan Feng
Research on navigation system of AMR based on ROS <https://ieeexplore-ieee-org.ezproxy.cad.univpm.it/document/9303274/>
- [5] Murat Köseoğlu;Orkan Murat Çelik;Ömer Pektaş
Design of an autonomous mobile robot based on ROS <https://ieeexplore-ieee-org.ezproxy.cad.univpm.it/document/8090199/>
- [6] Li Zhi;Mei Xuesong
Navigation and Control System of Mobile Robot Based on ROS <https://ieeexplore-ieee-org.ezproxy.cad.univpm.it/document/8577901/>
- [7] Buitinck L., Louppe G., Blondel M., Pedregosa F., Mueller A., Grisel O., Niculae V., Prettenhofer P., Gramfort A., Grobler J., Layton R., Vanderplas J., Joly A., Holt B., Varoquaux G., *API design for machine learning software: Experiences from the scikit-learn project*, 2013.
- [8] *Convolutional Neural Networks*, <http://deeplearning.net/tutorial/lenet.html>, last consultation: 07/01/2020.
- [9] BLDC motors <https://www.magneticinnovations.com/faq/what-is-a-bldc-motor/>

-
- [10] PWM explained <https://www.circuitbread.com/ee-faq/what-is-a-pwm-signal>
 - [11] Microcontroller <https://en.wikipedia.org/wiki/Microcontroller>
 - [12] IMU <https://towardsdatascience.com/what-is-imu-9565e55b44c>
 - [13] What is ROS <https://roboticsbackend.com/what-is-ros/>
 - [14] What is C++ https://www.w3schools.com/cpp/cpp_intro.asp
 - [15] Client-Server architecture <https://www.geeksforgeeks.org/client-server-model>
 - [16] Publisher-Subscriber architecture https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
 - [17] What is a Step Response <https://lpsa.swarthmore.edu/Transient/TransInputs/TransStep.html>

Appendix A

Brushless DC motor

A motor converts supplied electrical energy into mechanical energy. Various types of motors are in common use. Among these, brushless DC motors (BLDC) feature high efficiency and excellent controllability, and are widely used in many applications.

A Brushless DC Electric Motor (BLDC) is an electric motor powered by a direct current voltage supply and commutated electronically instead of by brushes like in conventional DC motors. BLDC motors are more popular than the conventional DC motors nowadays, but the development of these type of motors has only been possible since the 1960s when semiconductor electronics were developed.

A.1 Similarities BLDC and DC motors

Both types of motors consist of a stator with permanent magnets or electromagnetic coils on the outside and a rotor with coil windings that can be powered by direct current on the inside. When the motor is powered by direct current, a magnetic field will be created within the stator, either attracting or repelling the magnets in the rotor. This causes the rotor to start spinning.

A commutator is needed to keep the rotor rotating, because the rotor would

stop when it is in line with the magnetic forces in the stator. The commutator continuously switches the DC current through the windings, and thus switches the magnetic field too. This way, the rotor can keep rotating as long as the motor is powered.

A.2 Differences BLDC and DC motors

The most prominent difference between a BLDC motor and a conventional DC motor is the type of commutator. A DC motor uses carbon brushes for this purpose. A disadvantage of these brushes is that they wear quickly. That is why BLDC motors use sensors – usually Hall sensors – to measure the position of the rotor and a circuit board that functions as a switch. The input measurements of the sensors are processed by the circuit board which accurately times the right moment to commutate as the rotor turns.

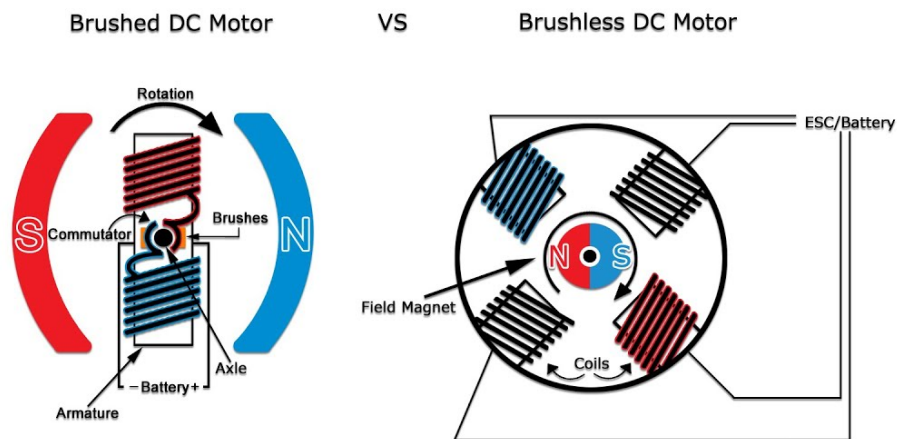


Figure A.1: Functioning principle of a BLDC and a DC motors.

Appendix B

PWM

Digital signals are signals that can be represented by a 0 or 1. Analog signals on the other hand have a greater range of possible values than just a 0 or 1. Both of these signals are used in the electronics but they are handled very differently.

If it is needed to take an analog input, it is possible to get the real-time analog data from a sensor, and then using an analog-to-digital converter (ADC) for converting it to digital data for a microcontroller.

If it is needed to control an analog device using a microcontroller a DAC (digital-to-analog) converter should be used. A DAC is expensive and it takes a lot of silicon area. To overcome these issues and to easily achieve the functionality of a DAC in a much more cost-efficient way, it is possible to use the PWM.

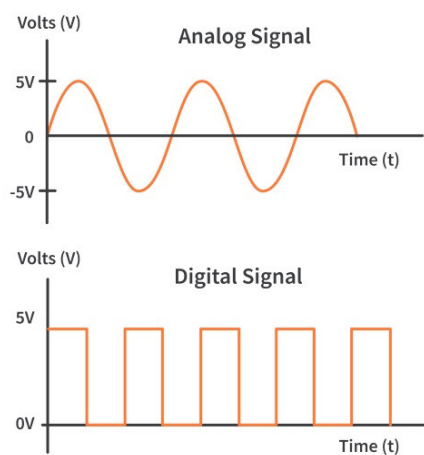


Figure B.1: Analog vs Digital Signal.

PWM or **Pulse Width Modulation** is a technique used to control analog devices, using a digital signal. This technique can be used to output an analog-like signal from a digital device, like a microcontroller. It is possible to control motors, lights, actuators, and more using the generated PWM signal.

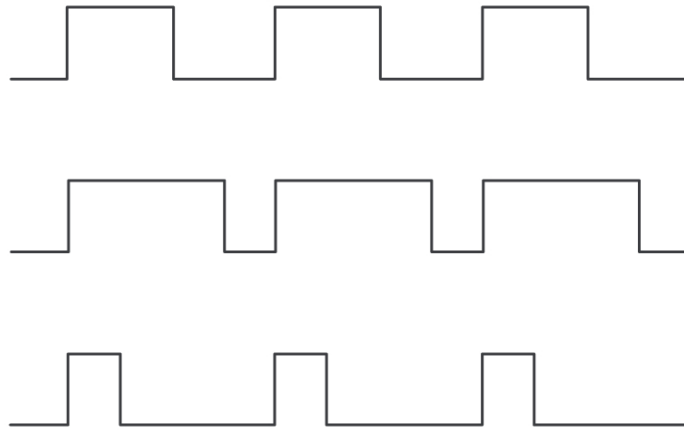


Figure B.2: PWM Signal Waveforms.

B.1 How it works

The PWM reduces the average power delivered by an electrical signal, by effectively chopping it up into discrete parts. The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power is supplied to the load.

The term duty cycle describes the proportion of 'on' time to the regular interval or 'period' of time; a low duty cycle corresponds to low power, because the signal is off for most of the time. Duty cycle is expressed in percent, 100% being fully on. When a digital signal is on half of the time and off the other half of the time, the digital signal has a duty cycle of 50% and resembles a "square" wave.

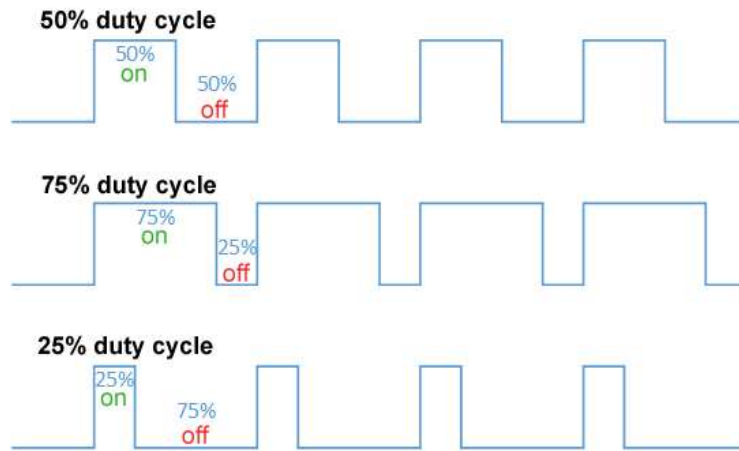


Figure B.3: PWMs with different duty cycle.

$$D = \frac{T_{on}}{Period} * 100 \quad V_{avg} = \frac{D}{100} * V_{max} \quad (B.1)$$

Where:

D = Duty Cycle in Percentage

T_{on} = Duration of the signal being in the “on” state

$Period$ = Total time taken to complete one cycle ($T_{on} + T_{off}$)

V_{avg} = Average voltage of the signal

V_{max} = Max voltage of the signal

It is possible to generate a PWM using the `analogWrite(pin,value)` function of Arduino (`value` determines the duty cycle of the signal, 0 for 0% duty, 255 for 100% duty). In the Arduino Mega board, it is possible to generate a PWM using the digital pins from 2 to 13.

Appendix C

Microcontroller

A microcontroller (MCU for microcontroller unit) is a small computer on a single VLSI integrated circuit (IC) chip. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. Program memory in the form of ferroelectric RAM, NOR flash or OTP ROM is also often included on chip, as well as a small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips.

In modern terminology, a microcontroller is similar to, but less sophisticated than, a system on a chip (SoC). A SoC may connect the external microcontroller chips as the motherboard components, but a SoC usually integrates the advanced peripherals like graphics processing unit (GPU) and Wi-Fi interface controller as its internal microcontroller unit circuits.

Appendix D

IMU

An Inertial Measurement Unit (IMU) is a device that can measure and report specific gravity and angular rate of an object to which it is attached. An IMU typically consists of:

- **Accelerometer:** is a sensor that measures the specific force (the body mass normalized the force). It provides the acceleration across the x, y, and z axes in its local frame.
- **Gyroscope:** is a sensor that measures angular velocity around the x, y, and z axes, in its local frame. Generally, integrating the measurements results in the angles themselves.
- **Magnetometer:** is a sensor that measures the Earth's magnetic field and provides the heading (the compass is one such device). If it is included in the IMU, it is called a "9-axis IMU."
- **Barometer:** is a sensor that measures air pressure and can provide altitude.

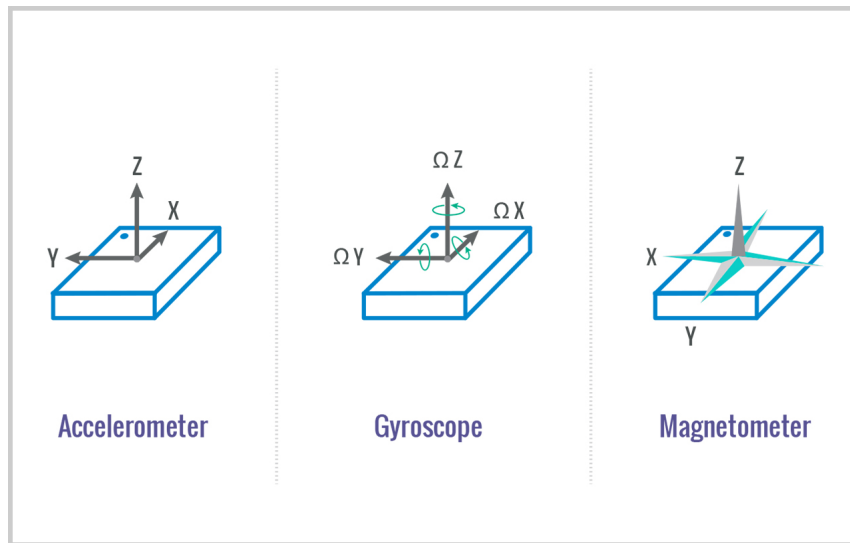


Figure D.1: Main sensors of an IMU.

Appendix E

ROS

The *Robot Operating System* (ROS) is an open-source framework that helps researchers and developers build and reuse code between robotics applications. ROS is also a global open-source community of engineers, developers and hobbyists who contribute to making robots better, more accessible and available to everyone.

E.1 What is ROS?

Robot Operating System, despite its name, is not an operating system. Nor it is really a framework.

ROS is more of a middleware, something like a low-level “framework” based on an existing operating system. The main supported operating system for ROS is Ubuntu.

ROS is mainly composed of 2 things:

- A core (middleware) with communication tools
- A set of plug & play libraries

E.1.1 What is a Middleware

A middleware is responsible for handling the communication between programs in a distributed system. In a robotic applications, the software is

divided into modules: small sub-programs which handle different tasks. ROS allows the communication between each other of all the modules.

E.2 Communication Tools

ROS comes with 3 main communication tools:

- **Topics:** used mainly for sending data streams between nodes.
- **Services:** allow to create a simple synchronous client/server communication between nodes. Very useful for changing a setting on the robot, or ask for a specific action.
- **Actions:** based on topics, provide an asynchronous client/server architecture, where the client can send a request that takes a long time (ex: asking to move the robot to a new location). The client can asynchronously monitor the state of the server, and cancel the request anytime.

For using these communication tools it is needed to define specific **messages** data.

ROS uses standard **TCP/IP sockets** to communicate between nodes.

E.3 Other features

It is possible to create some global settings named **parameters**, so multiple nodes can get access to the same set of settings from anywhere in the robotics application. It is possible to launch the complete robot's application with just one XML file, called **launch file**.

ROS also includes a complete **logging system**.

ROS allows to use a huge amount of ROS libraries for developing different functionalities of the robot.

Using all the ROS's features, it is possible to develop a *distributed system* for controlling the robot, by making many **ROS packages**.

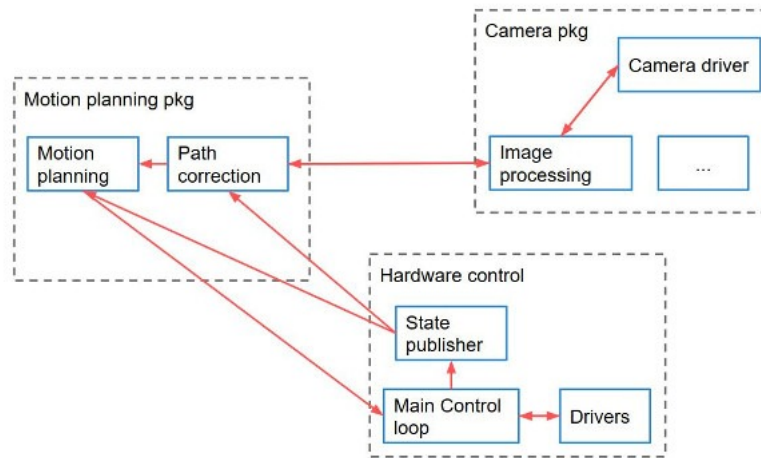


Figure E.1: Modules of a distributed system used for controlling a ROS robot.

E.4 ROS Languages

Robot Operating System is mainly developed using 2 languages: C++ and Python. Those are often the most preferred and used languages when developing robotics applications. It is needed to use the **roscpp** library to write C++ code, and the **rospy** library to write Python code. There are also some libraries to make a bridge with other languages, such as **rosjava** for Java, and **roslibjs** or **roscppjs** for JavaScript.

The ROS's modules can be written in any language. The communication is allowed since the **Communication Layer** is below the *"language-level"*.

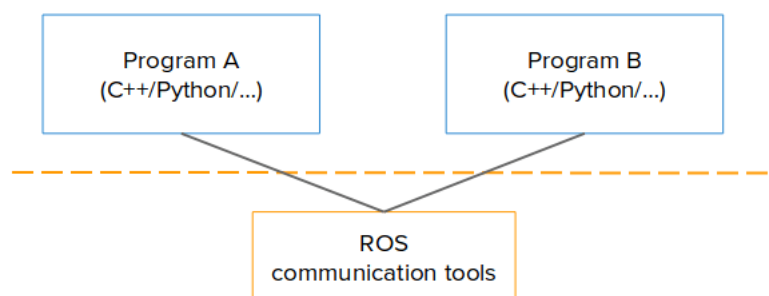


Figure E.2: ROS Communication Layer and Language Layer.

E.5 ROS Nodes

A ROS Node is a sub-part of the robotics application. It is basically a process that performs computation. It is an executable program running inside the robotics application. The application will contain many nodes, which will be put into packages. Nodes will communicate to each other.

Considering figure E.1, *Motion planning pkg*, *Camera pkg*, *Hardware control* are packages that contain many ROS nodes.

Appendix F

Client-Server Architecture

The Client-server model is a distributed application structure that partitions tasks or workload between the providers of a resource or service, called **servers**, and service requesters called **clients**.

In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the request, then process and delivers the data packets requested back to the client. Clients do not share any of their resources. Examples of Client-Server Model are Email, World Wide Web, etc.

- **Client:** is a computer (*Host*) i.e. capable of receiving information or using a particular service from the service providers (Servers).
- **Server:** a remote computer which provides information (data) or access to particular services.

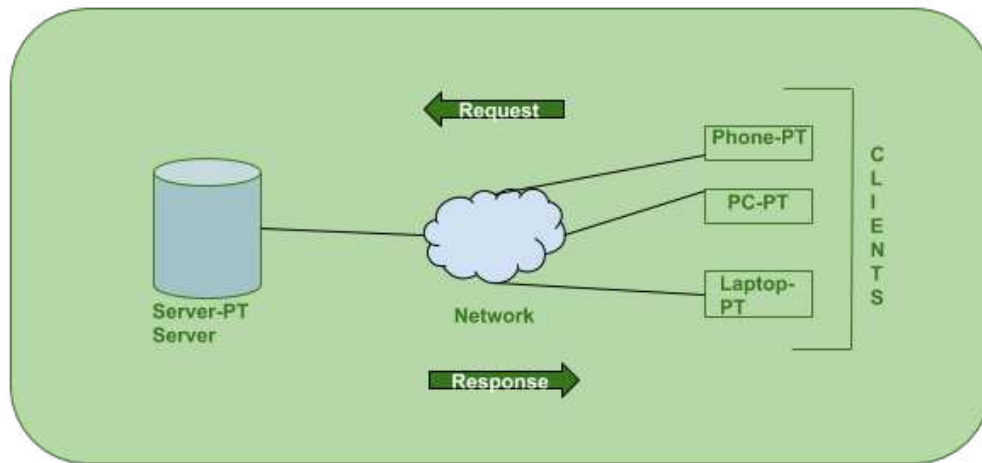


Figure F.1: Client-Server architecture.

This architecture can be adopted also for the communication of modules (sub-programs) between each other in a single application of a distributed system.

Appendix G

Publisher-Subscriber Architecture

In software architecture, publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

In the publish-subscribe model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called **filtering**. There are two common forms of filtering: topic-based and content-based.

- **topic-based system**: messages are published to **topics** or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe. The publisher is responsible for defining the topics to which subscribers can subscribe.
- **content-based system**: messages are only delivered to a subscriber if the attributes or content of those messages matches constraints defined by the subscriber. The subscriber is responsible for classifying the messages.

Some systems support a hybrid of the two; publishers post messages to a topic while subscribers register to one or more topics.

Appendix H

C++

C++ is a cross-platform language that can be used to create high-performance applications. It was developed by Bjarne Stroustrup, as an extension to the C language. It gives programmers a high level of control over system resources and memory.

H.1 Why use C++?

C++ is one of the world's most popular programming languages. It can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

C++ is portable and can be used to develop applications that can be adapted to multiple platforms. It is close to C, C# and Java, so it is easy for programmers to switch to C++ or vice versa.

H.2 Differences between C and C++

C++ was developed as an extension of C, and both languages have almost the same syntax. The main difference between C and C++ is that C++ support classes and objects, while C does not.

Appendix I

Step Response

When a dynamical system is studied, one of the most common test input used is the **unit step function**. The response of a system (with all initial conditions equal to zero at $t=0^-$, i.e., a zero state response) to the unit step input is called the **unit step response**.

Given a system with $x(t)$ as input, $y(t)$ as output and $H(s)$ as transfer function:

$$H(s) = \frac{Y(s)}{X(s)} \quad (\text{I.1})$$

The output with zero initial condition is given by:

$$Y(s) = H(s) * X(s) \quad (\text{I.2})$$

For the unit step function as input $X(s) = \frac{1}{s}$ so the unit step response is:

$$Y_\gamma(s) = \frac{1}{s} * H(s) \quad (\text{I.3})$$

Acknowledgements

I wish to express my deepest thanks to Prof. Augusto Ferrante for the opportunity I was given. I am extremely grateful for the guidance given by Dr. Gianluca Di Buò. I am extremely grateful to all the IDEA's team for the support given during the traineeship: Andrea Gaggiotti, Marco Brutti, Mirco Maggiori, Carlo Castagnari, Nicola Elisei, Zachary Kiernan, Daniele Belmonti, Francesco Esposti, Nazzareno Villa, Alessandro Ciancaglione, Daniele Marcozzi, Gloria Rossi, Corina Savoiu and Umberto Florio. An heartfelt thanks goes to my girlfriend, my parents, my sister and my girlfriend's family for their never-ending support and love.