POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master Thesis

Design and Implementation of an Autonomous Navigation System for Paquitop, a novel mobile robot for indoor assistance

Supervisor

Prof. Giuseppe Quaglia

Co-Supervisor

Ing. Luigi Tagliavini

Candidate

Giulia Pasimeni

A.A. 2021-2022

To my Parents
and my Sister,
my strength.

Abstract

The goal of this thesis is the development of an autonomous navigation system for a mobile robot named *Paquitop*. Its main peculiarity is to achieve omnidirectional motion exploiting conventional wheels. The Paquitop platform has been designed and prototyped by the researches at Politecnico di Torino during the last two years. This work represents the first step to enable the robot with autonomous navigation capability to perform *indoor assistive tasks* in full autonomy. The software architecture is designed to fulfill the following tasks: *Mapping*, *Localization*, *Obstacle Avoidance* and *Path Planning*.

The first step for the design is the definition of a *Hybrid Control Architecture* based on the primitive functions *Sense-Plan-Act*. The basic idea behind this design is to decouple the navigation algorithms inside these primitive according to their task. Firstly, the platform extracts in real-time information perceived from the environment through the sensors, returning its local position estimate, the map of the surrounding environment and the obstacle detection. The ability for a mobile robot to localize itself and at the same time to build a map of the surrounding environment recognizing the obstacles, is defined *Visual Simultaneous Localization and Mapping* (VSLAM). A navigable path between the starting point and the target one is computed by the *global planner*. The global path is subdivided into suitable waypoints by the *local planner*, which takes into account the dynamic obstacles and the vehicle constraints. Finally, the planned actions are controlled until the goal is reached.

The adopted framework for the implementation of all tasks is the *Robot Operating System* (ROS). Inside this framework, the *Unified Robot Description Format* (URDF) is used to represent the robot kinematic structure in the form of links and joints. The ROS packages used to fulfill the global task are included inside the *Navigation Stack package*. In conjunction with ROS, *Gazebo simulator* and *RViz visualizer* are adopted to simulate robot in its operational environment.

To effectively achieve autonomous navigation, the Paquitop platform has been provided with *exteroceptive sensors*: a *LiDAR*, used to acquire data for the SLAM algorithm and a *commercial tracking camera*, adopted to estimate the pose changes in time through sensor fusion techniques. The brain of the whole system is the computer *Nvidia Jetson Nano*, able to provide the need computational capability, connected to *Arduino*, which receives and sends information to electronic devices.

The obtained result involves a both reactive and deliberative framework, which guarantees an accurate mapping and localization in the environment. This data can be used by the navigation algorithms to accomplish the mobile robot to the desired pose and to perform an obstacle avoidance logic.

The thesis is divided as follows: *Chapter 1*, presents the state of the art of mobile autonomous robots, the logic of the motion planning and the tools used to develop the thesis work; *Chapter 2*, describes the Paquitop platform analyzing its kinematic and its main components; *Chapter 3*, defines the software robotic architecture with its packages and their implementations; *Chapter 4*, focuses on validation tests; *Chapter 5*, presents the conclusions pointing out the system limits and its possible improvements.

Acknowledgements

First of all I would like to thank *Professor Quaglia* and my supervisor *Luigi* for guiding, supporting and passing on me all their knowledge about the robotic area. I am grateful to have been part of their team for a few months, as it has allowed me to grow my wealth of knowledge thanks to their immense preparation.

I wish to extend my thanks to *Lorenzo* for all his advice and exchanges of views throughout the research project.

My heartfelt thanks to *my parents* and *my sister Marta* for their constant and unwavering support in reaching this milestone, despite the current year full of obstacles. I will never stop thanking you for always believing in me.

I thank all my Turin friends, in particular *Davide*, who has supported me and above all endured me during the university course. Thank you for advice, transmitted lightness, our thousand laughs and all the times you rushed to me to cheer me up. You are my missed little brother.

I would like to thank *Monica*, my laboratory friend: I will remember our short lunches, our tired faces and the great support given to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *robot* is an electromechanical artificial system capable of performing a series of actions programmed and ascribed by humans. The term derives from the Czech word "robota", meaning "forced labour", coined in 1921 by the writer Karel Capek in his play R.U.R. (Rossum's Universal Robots), for featuring a humanoid machine.

Nowadays, a robot incorporates a *mechanical structure*, constituted of rigid components connected by joints, which allow either rotational or translational motions of the whole robotic structure. The data acquisition about the internal state of the mechanical structure, such as position and velocity, is provided by a *sensorial unit*, in particular by *proprioceptive sensors*. Instead, to have information about the surrounding external environment, robot adopts *exteroceptive sensors*. The ability to comprehend the workspace in which the robot operates, is named perception, based on the sensory data acquisition, process and representation, useful to outline the external word with static and dynamic obstacles. In addition, a *control structure* is fundamental for guaranteeing the fulfillment of the requested tasks, composed by electromagnetic motors or actuators. To accomplish a task, the robot must move autonomously from one place to another. In such sense, *hardware and software systems* are needed, to program and to check the activities of the robot.

Each robot has a different level of autonomy, ranging from robots fully controlled by human beings, to robots that perform tasks in fully autonomy. The remote-controlled robots are employed in hazardous environments for the operators, while the autonomous mobile robots are adopted for improving the speed and the accuracy of repetitive operations.

Unlike the *stationary robotics*, used for carrying out tasks on site, an *Autonomous Mobile Robot* (AMR) navigates in the world and makes decisions in real-time independently, without the help of human beings.

The operating environments of the robots can be classified intro three categories: *pre-defined and structured environment*, in which the robot has information about

the surrounding environment and objects in it; *semi structured environment*, where the robot knows only some things, as for example the map; *unstructured environment*, in which the robot relies on information taken from sensors and camera to work autonomously, as has no prior knowledge about the environment.

The fields of application of robotics are wide, listed below: medical, military, domestic, service, educational, industrial and aerospace application fields.

Later, the state of the art of a mobile robot, its fundamental subsystems and the adopted environment to develop the mobile platform, will be described in detail.

1.1 State of the art of Mobile Autonomous Robot

A subset of robots is the *mobile robot*, an automatic machine that is not attached to the environment, able to move in a given space [1]. As opposed to stationary robots, a mobile robot owns an unlimited motion within both known and unknown environments. Like a human, it can carry out a variety functions such as domestic helper, surveillance, cargo transporter, entertainment and also, to perform repetitive activities and to operate in areas that the humans cannot explore, as radioactive environments and spaces too far in distance and time.

Mobility requires a specific design, which can be organized into three parts: *Software*, *Hardware* and *Mechanical*.

The *Software part* consists of a *high level*, which includes all navigation algorithms to fulfill the designated mission and a *low level*, which possesses the commands to send to both the microcontroller and processors, in order to provide the mobility to the robot. The development of a software architecture includes all processes required by the robot to perform a specific task and it is based on the integration of its major components and on how they interact.

The *Hardware part* comprises electronics components that convert the software algorithms in form of control signals for the actuators.

The *Mechanical part* involves the design of a robotic structure, namely the type and the configuration of a robot and its main components, in detail the *links*, its rigid bodies, the *joints*, the connections between the links and the *end-effector*, a tool attached to a final part of a robotic arm, for example a gripper, used to interact with the environment for grabbing, holding and handling tasks.

Mobile robots can be categorized into three categories according to their operating environment: the *Ground Mobile Robot* (Figure 1.1a [2]), the *Aerial Mobile Robot* (Figure 1.1b [3]) and the *Water-based Mobile Robot* (Figure 1.1c [4]).

Each one of these categories can be further divided into several sub-categories. In particular, the Land-based Mobile Robots can be classified on the basis of their locomotion system into three groups: *Wheeled, Legged* and *Tracked*.

Wheeled robots (Figure 1.2a [5]) exploit the ground contact and the friction

(a) *Ground Mobile Robot*

(b) *Aerial Mobile Robot*

(c) *Water-based Robot*

Figure 1.1: The Mobile Robots Classification according to the operating environment

to perform their motion. The wheels design allows the robot to travel with high speed and energy efficiency on flat and even grounds. Wheeled robots have a simpler control system compared to the other two categories, but a limited ability to overcome obstacles and low adaptability to the different types of terrains.

Tracked drive (Figure 1.2b [6]) exploits bands of tracks driven by two or more pulleys. Tracked robots can move on uneven and yielding terrains, thanks to their major contact surface with the ground [7]. On the other hand, they are characterized by a slow motion and a high energy consumption due to the friction of the transmission system and especially to the sliding of the tracks during curvilinear trajectories.

Legged locomotion (Figure 1.2c [8]) is the solution to rough terrains and obstacles, characterized by a slow motion and less energetically efficient on flat grounds. Legged robots have a high architectural complexity of mechanical and control architecture, due to handle a high number of actuators. This implies a high cost of the robotic structure.

(a) *Wheeled Mobile Robot*

(b) *Tracked Mobile Robot*

(c) *Legged Mobile Robot*

Figure 1.2: The Mobile Robots Classification on the basis of the locomotion system

By combining all locomotion systems a *Hybrid Robot* is achieved, which integrates some characteristics of the previous locomotion systems. Example of Hybrid robots are: *Legs-Wheels, Legs-Tracked, Wheels-Tracked* and *Legs-Wheels-Tracked*.

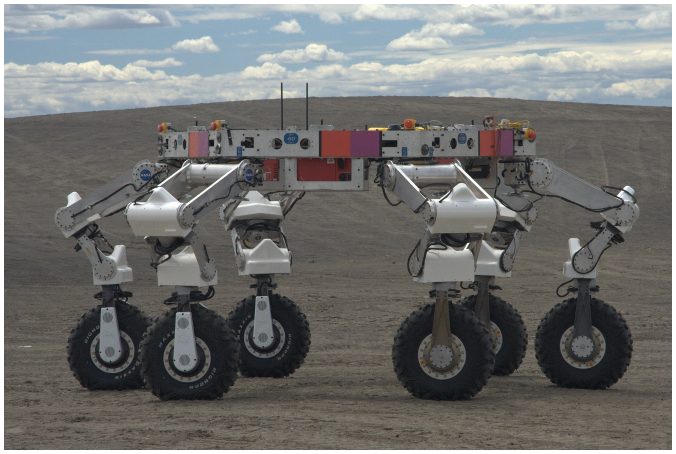
Legs-Wheels robots (Figure 1.3a [9]) guarantee a high speed and an energy efficiency of the wheeled locomotion and also both high mobility and flexibility of legs.

Legs-Tracked robots (Figure 1.3b [10]) are sturdy and reliable, with a greater capability of overcoming obstacles also in rough environments. On the other hand, they travel with both low speeds and energy efficiency.

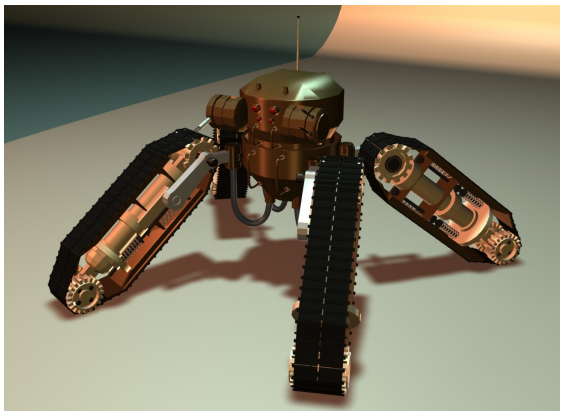
Wheels-Tracked robots (Figure 1.3c [11]) have a good performance on uneven and soft grounds and an energy efficiency on flat terrains.

Legs-Wheels-Tracked robots (Figure 1.3d [12]) combines all three types of locomotion. Their peculiarity is the high adaptability to various terrains and high speeds. The combination of the categories causes a high complexity of the architecture and

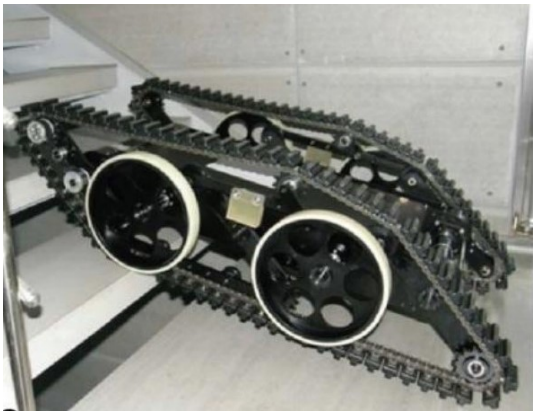
an elevated energy consumption.



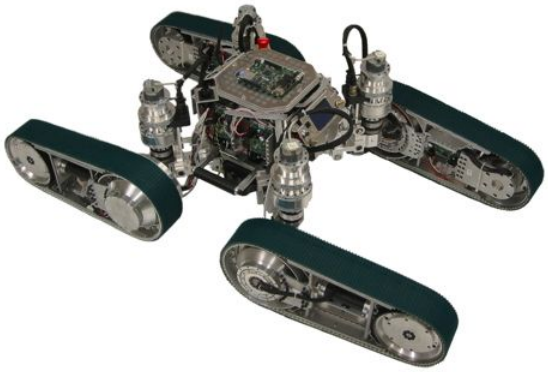
(a) *Legs-Wheels Robot*



(b) *Legs-Tracked Robot*



(c) *Wheels-Tracked Robot*



(d) *Legs-Wheels-Tracked Robot*

Figure 1.3: The Hybrid Mobile Robots Classification

In this chapter, the attention is focused on the *Wheeled Mobile Systems*, classified in five categories according to the platform's locomotion [13], in particular on the basis of the kinematics and dynamics properties:

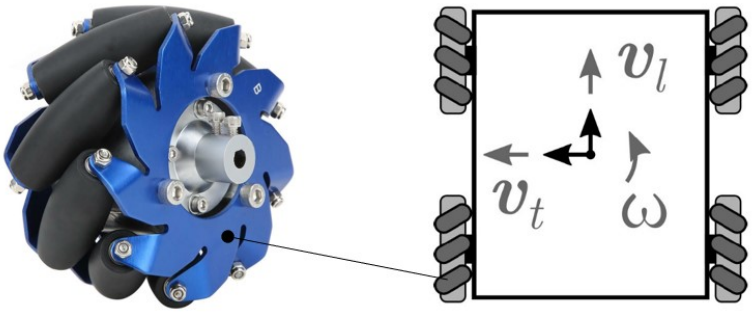
- Omnidirectional Drive Systems;
- Differential-Drive Robots;
- Wheeled Robots with at least one steering mechanism;
- Bicycle, Tricycle and Car-like Robots;
- Pseudo-Omnidirectional Robots based on swerve-drive systems.

The *Omnidirectional Drive Systems* have a specific wheels construction, which allows a rolling motion in all directions, enabling a full mobility in the plane. The wheels used for this type of motion are the *Mecanum* (Figure 1.4a [13] [14]), *Poly* (Figure 1.4b [13] [15]) and the *spherical* (Figure 1.4c [13] [16]) wheels. The first have the rollers mounted around the wheel rim and their rotation axis is rotated of 45 degree, which guarantees the motion in more directions. By setting the same velocity to all wheels, both forward and backward motions are possible; for the side motion, all wheels on one diagonal have opposite velocity to the wheels on the other diagonal. The motion in all directions can be achieved combining forward/backward with side motions.

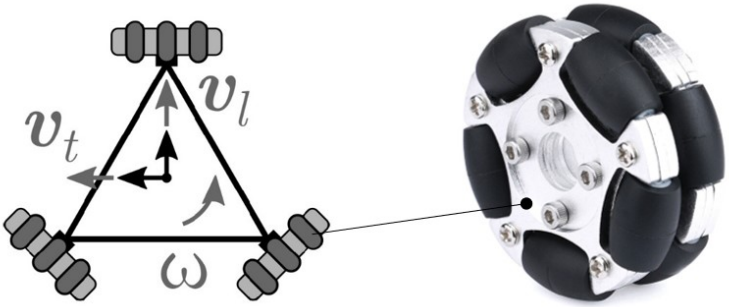
About the *Poly wheel*, a number of six free passive rollers are arranged around the wheel rim with their axes to 90 degree with respect the main wheel axis. These wheels can rotate and slide laterally.

The passive rollers of the Mecanum and Poly wheels are characterized by small dimensions, by a discontinuous contact with the ground, that causes vibrations above all on uneven terrain and difficulty to carry heavy loads, and by a high sensitivity to the floor condition. The wheels' construction accuracy affects strongly the navigation performance of the platform: a high control uncertainty could cause their slippage. Unlike the previous wheels, the omnidirectional robot based on *spherical wheels* has a continuous contact with the ground, a high controllability, both little slippage and sensitivity to the floor condition. Instead, the characteristics in common are the following: their actuation redundancy, the strong influence on the performance by the construction of the wheels and finally a high uncertainty and slippage in the case of control errors.

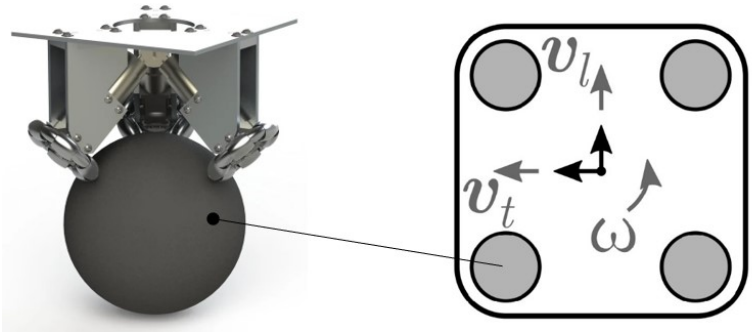
The *Differential-Drive Robots* (Figure 1.5 [13]) have two non-steering wheels, controlled by independent motors and sometimes, to give more stability the system, one or more non-driven wheels, called caster wheels, are implemented. The latter support the vehicle avoiding its tilt. The driven wheels are placed on the same axis at a certain distance from their centre. To maneuver any differential robot in a



(a) *Mecanum Wheel*



(b) *Poly Wheel*



(c) *Spherical Wheel*

Figure 1.4: Types of Omnidirectional Wheels

plane, the robot requires a linear velocity V_l with direction perpendicular to their common axis, computed as the average between the tangential velocities of right and left wheels, V_r and V_l respectively, and an angular velocity. The motion of the

robot changes according to tangential velocity value of the right and left wheels, named respectively V_r and V_f:

1. If $V_r = V_f$, the robot has a forward linear motion along a straight line and the rotational velocity w is null;
2. If $V_r = -V_f$, the robot rotates on itself around the intermediate point of the wheels axis.
3. If $V_l = 0$, the robot rotates around the left wheel. Otherwise, if $V_f = 0$, the robot rotates around the right wheel.

The motion cannot occur along the common axis of the driven-wheels.

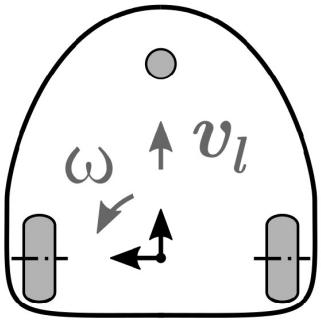


Figure 1.5: Differential-Drive Robot

The *Wheeled Robots* with at least one steering mechanism (Figure 1.6 [13]), point at the same directions because of the same steering angle of the wheels. Despite the presence of the castor wheel causing vibrations, the robot guarantees stability, thanks the continuous contact of the wheels with the ground. This implies both high payload and efficiency.

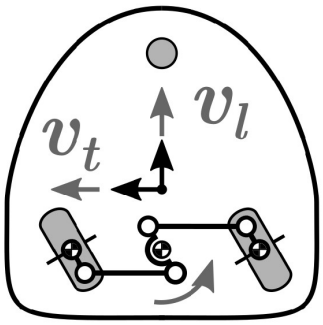


Figure 1.6: Wheeled Robot with at least One Steering Wheel

The *Bicycle*, the *Tricycle* (Figure 1.7 [13]) and the *Car-like Robots* are grouped into the same category. According to the type of robot chosen among those listed above, they are constituted of one or several steering front-wheels and one or several two fixed rear-wheels. If more than one steering wheels is adopted, the drive is based on the *Ackermann steering principle* [1]. Before describing it, the definition of the *instantaneous center of rotation* (i.c.r.) point is need, around which all the wheels follow a circular motion with the same angular velocity [1]. Taking into account the *Car-like Robot*, the idea behind the Ackermann steering principle is as follows: the inner wheel closer to i.c.r. will steer with a bigger angle with respect to the outer one, in order that the vehicle rotates around its middle point. As a consequence, the angular velocity of the inner front-wheel is slower than the outer front-wheel. The i.c.r. point is placed on the straight line given by the rear-wheels' axis.

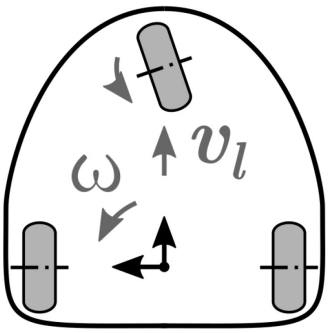


Figure 1.7: Tricycle Robot

The *Pseudo-Omnidirectional Robots* (Figure 1.8 [13]) are characterized by no fixed wheels and at least two independent steering wheels or a castor wheel or a spherical wheel or self-balancing algorithms. This type of locomotion architecture has several actuation strategies: on the one hand, if the steering wheels axis lies on the same line, the platform has the same configuration of the differential drive robots, and on the other hand, the platform can assume a pseudo-omnidirectional motion, with non-equal steering angles of conventional wheels. To have a pseudo-omnidirectional motion, it is necessary to have independently actuated wheels, which do not allow sideways sliding. The term *pseudo-omnidirectional* means that the robot can perform omnidirectional motions, but to do so there must be a reconfiguration of internal degrees of freedom of the platform, such as steering. Within the latter category Paquitop belongs, whose kinematics will be described in *Chapter 2*.

After an overview of all mobile robot types, with particular attention on the wheeled one and on their classification according to the kind of locomotion, the concepts that involving the autonomous navigation will be deepened in the following

sections.

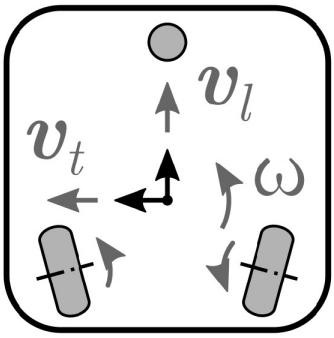


Figure 1.8: Pseudo-Omnidirectional Robot

1.2 Motion Planning

Motion planning problem is defined as the construction of a robot motion to execute a path from a start configuration to a target configuration. Motion must satisfy a set of requirements such as avoiding collisions, joint or effort limits, performing a specific task along a path, executing a set of waypoints [17]. To solve the problem of motion, the concept of the *configuration space* or *C-space* is useful, namely the set of all possible states that the system might have to reach a specific goal pose. The solution to this problem consists of computing a path inside of the configuration space, where each point of the route represents a unique configuration of the robot, described by a generalized coordinate q. All configurations made while executing a route, are enclosed within a vector, called C-space.

Besides the concept of motion planning, there is that of *path planning*, which is defined as a purely geometric problem consisting of finding a collision-free path between initial and goal points. Unlike motion planning, this problem does not take into accounts the dynamics, the duration of motion and constraints on the motion and the control inputs [17].

One of objective functions of the path planning is the *shortest path planning*, defined as the search of the path that minimizes the route cost. The cost includes all parameters that affect the choice of the path, as the time and the distance. Algorithms that employ this objective function are *Dijkstra* and *A**. The *first* transforms all working space into a network of nodes or points passable by the robot, each linked by edges or arcs connecting the motion from one node to another. The entire path will be constituted by the set of all edges, each associated to a cost and it will have as total cost the sum of all edges costs along the route. The *second*

is an extension of the Dijkstra algorithm, characterized by a best computation time thanks to the heuristic functions used. Both algorithms will be described in detail in the *Chapter 3*.

Another objective function in motion planning is the *optimal coverage path*, defined as the coverage of an area with minimal redundant traversals. One of the algorithms that employs this logic is *grid-based method*, which divides the working space into grid cells. Among the applications that employ this logic are cleaning and farming.

The path planning problem precedes the *trajectory planning* one, which consists of assigning a time law to a geometric path given by the path algorithm. In detail, the trajectory planning considers the solution given from the path planning algorithm and finds a way to move along it, satisfying all mechanical limitations of the robot [18]. The outputs of the trajectory planning are the trajectory of the joints, the links and the end-effector, in terms of position, velocity and accelerations values.

One of the competences of the motion planning is *perception*, namely the ability of a robot to collect information from the surrounding environment. This is a fundamental function for an autonomous system as provides crucial information on free drivable areas, as a consequence the obstacles' locations and other date as position, velocities and acceleration of a robot. In particular, the ability of the robot to calculate its position with respect to the environment is named *localization*. The latter is included in the competences of a mobile robot and it is closely linked to the *mapping* concept, considered another ability of an autonomous system. Localization and mapping are tasks that can be performed by means of the sensors placed on a robot: LiDAR, camera or the fusion between these two types of devices. The robot, equipped with on-board sensors, is able to identify the obstacles along its route and to modify its motion to avoid collisions through the implementation of some algorithms. This represents another competence of the motion planning called *obstacle avoidance*.

1.2.1 Perception

In robotics, perception is the ability of robot to perceive, comprehend and reason about the surrounding environment [19]. Robotics perception concept is based both on the acquisition and the process of sensor data, which is fundamental in motion planning problem to make decisions, plan and operate in real-world environments.

Components of the perception are sensory data processing, data representation (environment modelling) and algorithms for data analysis and interpretation. Mobile robots carry different types of sensors to perform different functions, which can be classified in *exteroceptive* and *proprioceptive sensors*. The first sensors acquire information from the surrounding environment (e.g. temperature, light intensity,

distance measurements), while the second ones monitor the internal state of a robot (e.g. motor speed, battery charge, wheel load).

Addressing the specific application of the project, Paquitop is equipped with the exteroceptive sensors: the *RPLIDAR A2 sensor* and the *tracking camera T265*, which will be present detailly in *Chapter 2*.

1.2.2 Localization and Mapping

Navigation of the mobile platform requires the ability of self-localization and mapping within outdoor and indoor environments. In this project, the key enabling these tasks is the implementation of the *SLAM technology*, which addresses the problem of constructing the environment model (map) that the sensors are perceiving, while simultaneously performing its localization in it. The most common SLAM systems can be classified as *LiDAR SLAM* or *Visual SLAM*, which exploit information from LiDAR and camera sensors respectively. These two technologies will be described detailly in the next two paragraphs.

LiDAR SLAM

LiDAR, acronym for *Light Detection And Ranging*, identifies a sensing technology that uses light in the form of pulsed laser beams to measure the ranges (distances) to a selected target. The devices that employ this technology, are able to transform the surrounding environment with a points cloud model, through their field of view of 360 degree of the optical window. In general, these sensors are constituted by three components: the transmitter, the receiver and the detector. An electromagnetic wave generator (transmitter) emits a laser signal and, after detecting any surface, it reflects on it and returns by the receiver. The reflected light is analyzed by the detector, which is based on the optical *Time-of-Flight* (ToF) measurement, namely the time between the emitted and reflected light beam. This parameter is needed for calculating the distance to each object, obtained knowing that the electromagnetic waves travel at the speed of light. The distance to the detected object can be calculated as follows (1.1):

$$d = \frac{c \cdot ToF}{2} \tag{1.1}$$

where d indicates the distance to the target, c is the speed of light and ToF is the Time of Flight described previously.

In addition to detect the presence of objects, Lidar SLAM-based sensors are also used for building 3D map of the environment through the *Hector-SLAM approach* thanks to the acquisition of the distances every second. This approach is one of the many algorithms that can be used and it will be discussed in *Chapter 3*.

As mentioned before, other task of this technology is to provide the pose of the robot, which is then fused with other computed poses given by more sensors through an *Extended Kalman Filter* (EKF), deepened in *Chapter 3*.

Visual SLAM

Visual SLAM or vSLAM, refers to the process to locate the pose of the robot, while simultaneously it builds the map of the environment through the images captured by the camera sensors. It is a technique for obtaining a 3D structure of an unknown environment by using the visual information. Visual SLAM can be divided into three main categories [20]: *Visual-Only SLAM*, *Visual-Inertial SLAM* and *RGB-D SLAM*.

VO SLAM acquires only 2D images captured from multiple points of view by a monocular or stereo camera. Then, it defines a global coordinates system through an initialization process, and finally it builds a map. This approach can adopt two different methods, namely the *Feature-Based* and the *Direct Methods*. The first extracts some feature points from the images, named key points, both to calculate the pose and orientation of the camera, and to construct a feature map. Unlike the Feature-Based Method, Direct Method has the aim to estimate the robot motion using pixels.

VI SLAM incorporates an *Inertial Measurement Unit* (IMU) to estimate the sensor pose. It employs a *gyroscope* for providing information relative to the angular rate; an *accelerometer* to compute the acceleration and a *magnetometer* device to determine the magnetic field around the device, in particular that of the earth in order to have information regarding orientation.

RGB-D SLAM employs a *depth sensor* and a *monocular RGB camera* to acquire the depth information and the color image data of the environment. These devices use the *Iterative Closest Point* (ICP) algorithm to provide the sensor pose and the map with a high accuracy.

In general, Visual SLAM logic [21] consists of five parts, as shown in Figure 1.9. Sensors send the image information to *Visual Odometer*, which is able to estimate the motion of the camera via the keyframe pose. *Back-End Optimization* considers both of the motion equation and the observation one, to optimize the pose of each point crossed building a trajectory. As Back-End Optimization, usually an *Extended Kalman Filter* (EKF) is used. The latter estimates the pose of the robot, and it is the nonlinear filter of the *Kalman Filter* (KF). *Loopback Detection* deletes the errors accumulated over time, re-estimates the pose, building globally the consistent trajectory and the map. Finally, from the information collected, the map measurement is established.

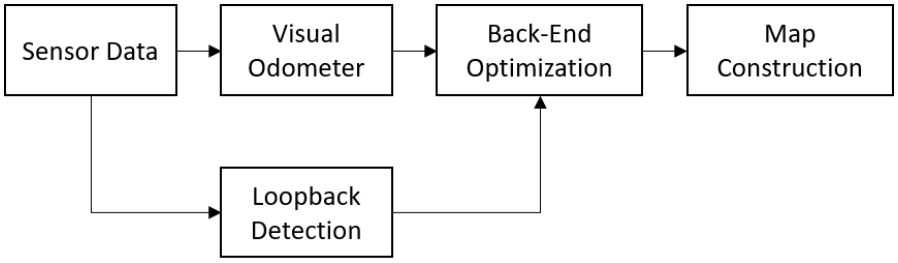


Figure 1.9: Visual SLAM Logic

1.2.3 Obstacle Avoidance

Obstacle avoidance is a crucial task for robotics systems, which must avoid to collide with any object present within its workspace. The detection of both static and dynamic obstacles occurs by means of the sensors. Each time that an obstacle is detected, it is inserted within the grid map, which represents the workspace of the robot in form of filled or empty cells, depending on the presence of the obstacles. The basic idea of the occupancy grid map is to construct a map, where each cell is marked with a binary number: the number 1 if the cell contains an obstacle; the number 0 if it a free space. Hence the occupancy grid map is the transformation of a continuous world into a discrete representation.

In order to build a feasible path from an initial configuration to a goal configuration, the occupancy grid is analyzed and constantly updated, so to provide a drivable path. Algorithms that fulfill this task are present in *Global* and *Local plannings*, which employ the obtained costmap to build a path.

1.3 Development Environments

This paragraph contains the tools used to develop the thesis work. In particular, the ROS framework is described, listing its key concepts and simulation tools. The simulation of robot movements is an essential tool for anyone working in the world of robotics. A well-designed simulation allows to quickly perform tests, design robots, observe artificial intelligence systems using realistic scenarios. ROS deals with distributing tools for analyzing and debugging the system, namely *RViz* and *Gazebo*.

1.3.1 ROS: Development framework description

ROS is the acronym of *Robot Operating System*, which indicates a set of software libraries and tools for the development and programming of robot. ROS is an open-source and meta-operating system, as it provides the its same performance,

including hardware abstraction, low-level device control, communication between processes and packages management.

ROS is designed according to a modular structure with several independent modules, or packages, responsible of small tasks in the overall software. The modularity has as advantages easy debug of small part of code and functionality and their update.

The distribution used for this project is *Melodic Morenia LTS version*, which is supported by *Ubuntu 18.04*. LTS stands for Long Term Support and means that the released software will be maintained for long period time (5 years in case of ROS and Ubuntu). Currently the supported languages are C++, Python, Matlab and Java. For this project the Python language has been used.

ROS architecture is based on the following fundamental concepts:

- **Master**: the core of the software, which has the task of orchestrating all nodes by managing their name registration, parallel execution and communication between them. Without the presence of the ROS Master, the nodes could not identify each other, exchange messages or invoke services. Master node can be launched with the roscore command.

- **Node**: a process that performs the main functions of the system and deals with the processing data. All the nodes are inserted inside an interconnected graph where each of them is able to communicate with all the others in a direct way. To have this graphical view of the system nodes, $rqt - graph$ command is needed. Each node will be related to only one specific functionality, in order to keep a both ordered and intuitive graph.

- **Message**: the communication between nodes occurs by means of messages, which contain the information necessary to fulfill the functionality of the nodes. ROS uses different types of messages, each for a particular purpose. The structure of the message is composed using basic data types as integer, float, char etc, which also can be grouped in arrays.

- **Topic**: each message needs to be located within the ROS network on a given topic. The messages are exchanged through the *publish/subscribe* communication system: a node is named publisher if it publishes a message on a topic, while subscriber if it subscribes to the appropriate topic as interested in a certain type of data.

- **Service**: the *request/reply* communication system is created through the use of services, which unlike messages, support a synchronous communication. Services are provided by the *server node*, while a *client node* calls the service by sending a request message and awaiting the response by the server node.

- **Bag**: a file format for saving ROS messages data. Bags subscribe to one or more ROS topics and, by means of a variety of tools, they are able to process, analyze, visualize messages or also remap them to new topics.

1.3.2 RViz

RViz, or *Ros Visualization*, is 3D visualization tool for ROS applications. It provides the visualization of robot model with kinematic chain parent-children relationships, reference frames of all the rigid bodies, environment map and all captured data from sensors.

For more details, take a look to [22].

1.3.3 Gazebo

Gazebo is a 3D robot simulation tool which able to test algorithms, design robot and simulate both real movements of robots and sensors, in indoor and outdoor environments. Plugins can be implemented to manage and control any aspect of the robot.

For more details, take a look to [23].

Chapter 2

Paquitop

This chapter presents an innovative mobile robot for assistive indoor applications, named Paquitop (Figure 2.1), designed and prototyped by the researches at Politecnico di Torino during the last two years.

The robot's objective is to autonomously perform basic actions that do not require the presence of a human operator. Its main tasks are the constant monitoring and tracking of the patients, as measuring of specific parameters among them the temperature or the blood pressure, and the delivering of medical products.

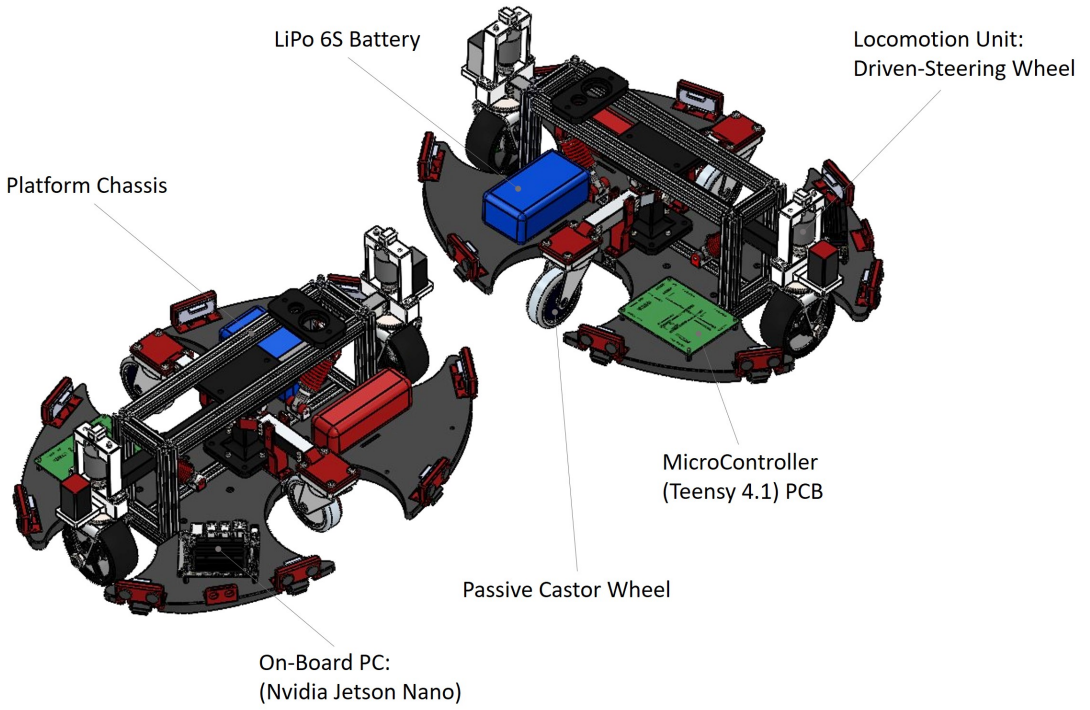
As a consequence the platform is designed for working in a domestic and unstructured environment, occupied by people [24]. The workspace strongly affects the mechanical design of Paquitop, which owns an elliptical chassis both to suggest to human footprint and to easily navigate narrow spaces lived by people, given its small dimensions.

The chassis is suspended on four wheels: *two passive castor wheels* and *two driven-steering wheels*, the latter controlled by two BrushLess Direct Current (BLDC) motors, to enable the forward/backward motions and by two stepper motors, for the steering motions.

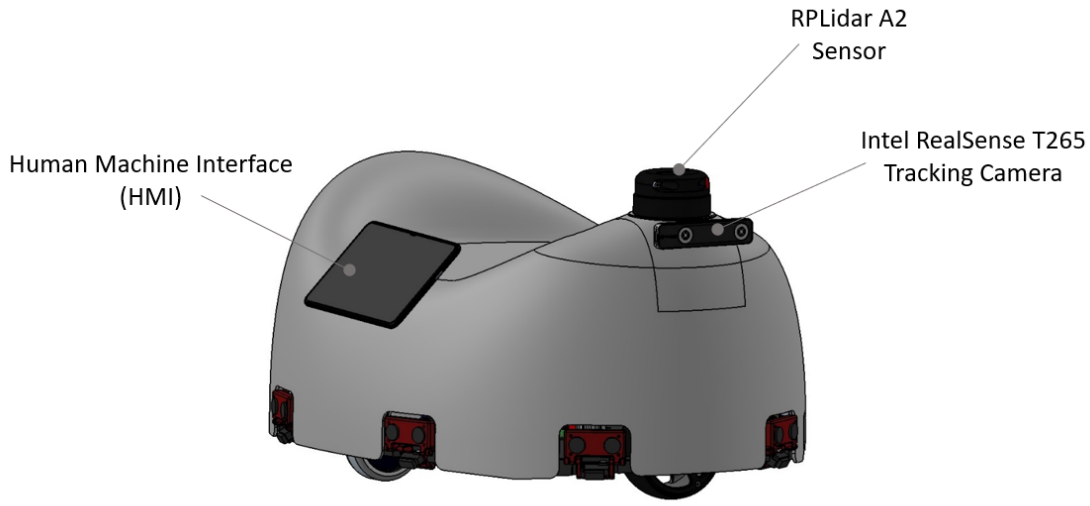
The motor control occurs by means of a microcontroller, the *Teensy 4.1 board*, which transforms the velocity twist commands imparted by a remote controller or by an autonomous navigation architecture, into commands for motors.

The microcontroller is connected to the *on-board PC*, in the case of the autonomous drive, or to a radio system, if a remote control is implemented. To accomplish the assistive tasks, the platform is provided with a *2D RPLidar A2*, both to measure the distance between itself and the surrounding obstacles and to map its workspace, and with a commercial tracking camera, named *Intel RealSense T265*, to localize itself within the built map.

In addition to these sensors mounted on a chassis' holder, a power supply battery, namely a *LiPo 6S*, and a commercial tablet, a *Human-Machine Interface* (HMI) have been implemented, the latter used by the user to control the robot state, the



(a) *Inner Part*



(b) *Outer Part*

Figure 2.1: Paquitop Platform

tasks' fulfillment and other information.

Navigation tasks are implemented within the ROS-based software architecture, which is based on Visual Simultaneous Localization and Mapping (VSLAM) algorithms, useful to constantly update the data coming from sensors, then used by the global and local planners to properly plan a safe path, and at the same time to guarantee the non-collision with obstacles.

In the following sections, the analysis of the robot features and the overall architecture of the platform are described, starting from the requested specifications until to the details relating to the mechanical, electrical and software parts of the system.

2.1 Robot Concept

The key concept behind the robot's design is the assistance of people owning a reduced mobility. The platform is designed to have two mainly functionality: the health conditions monitoring and tracking of a person during his domestic life, and a basic human assistance for the achievement of low-dexterity manipulation tasks (as objects transport or their simple manipulation). These duties are accomplished in an unstructured environment (Figure 2.2 [24]), occupied by people and obstacles, in which sensors have a crucial role for the autonomous navigation and the localization of robot. Due to its peculiar workspace, the robot is shaped on human scale in order to easily navigate in narrow environments lived by persons.

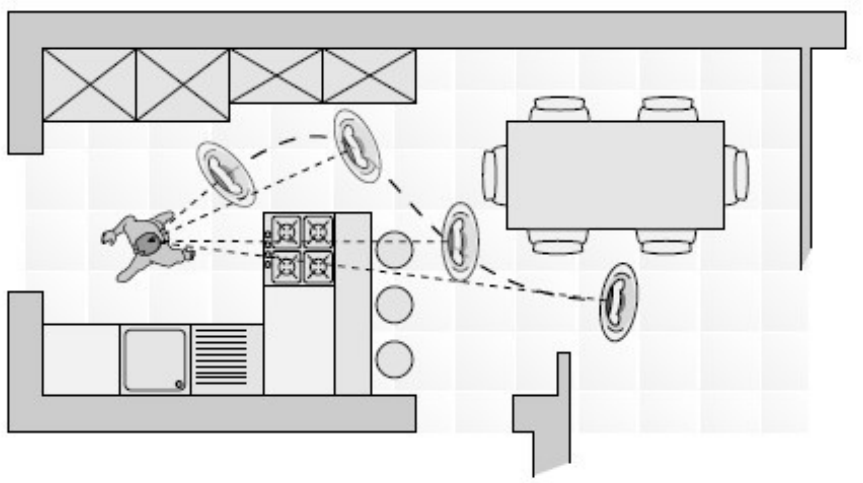


Figure 2.2: Paquitop Navigation in an Unstructured Environment

A high maneuverability is reached thanks to the non-axisymmetric and elliptical chassis shape (Figure 2.3) and the omnidirectional mobility, which gives a redundant actuation to the overall system. The robot is able to overpass small objects (carpets) or to avoid both static and dynamic obstacles.

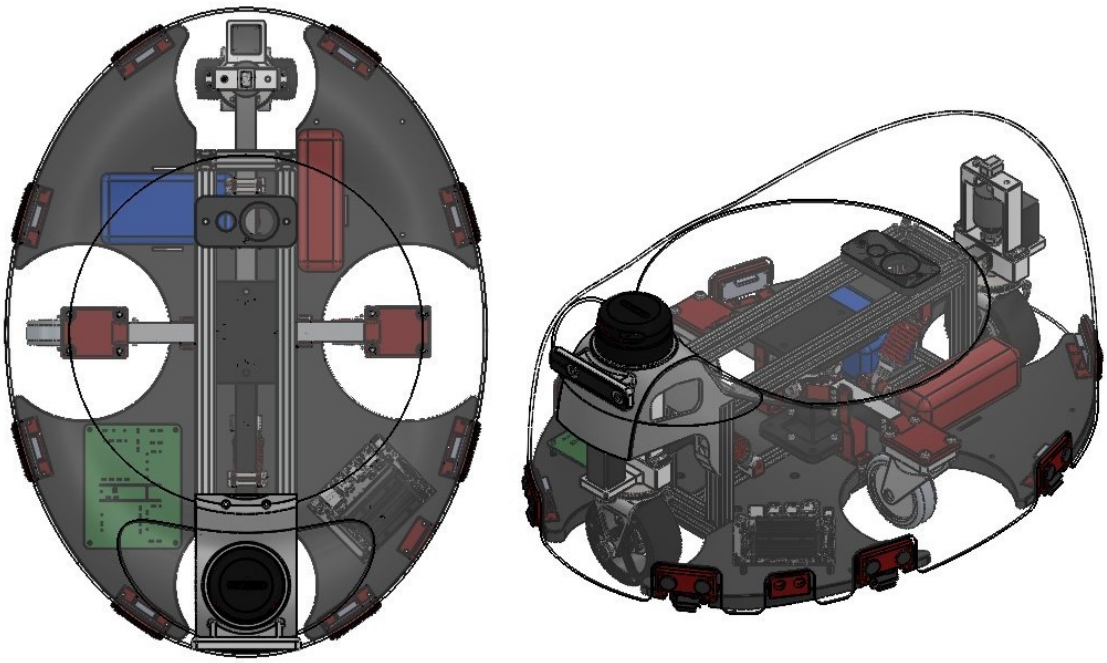


Figure 2.3: Non-Axisymmetric and Elliptical Shape of Paquitop

After analyzing the tasks to address and the working conditions, the next step is to define the adopted design process for the achievement of the requested specifications. A modular design has been used, in which the overall system is split into sub-systems, in order to develop a structure more robust. The whole architecture can be subdivided in two main layers (Figure 2.4), namely *navigation* and *motion layers*, within which all physical and non-physical parts (mechanics, electronics and software) of the robot are included. The *navigation layer* comprises the software part of the system within the on-board PC, containing all autonomous navigation algorithms and the *Human Machine Interface* (HMI), collecting information and tasks provided by the user. The exteroceptive sensors, the *Electronic Control Unit* (ECU) and the whole mechanical system, can be considered part of the *motion layer*. The interaction and the control of these layers, ensures the fulfilment of all tasks.

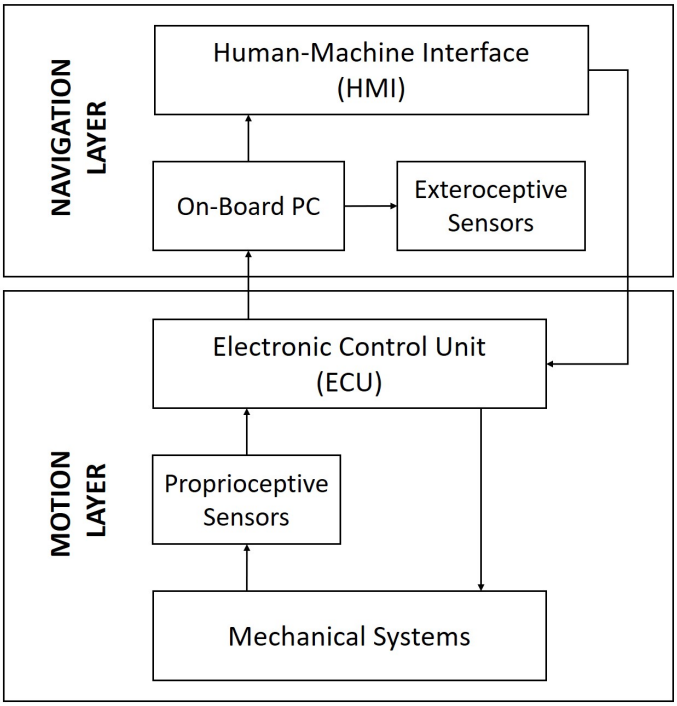


Figure 2.4: Layered Design of the Platform

2.2 Mechanical Design

The mechanical design must reflect the features of the requested tasks. As the mobile platform is mainly developed for personal assistance, its dimensions must be both compatible with the workspace in which the robot is asked to work, in order to guarantee a full mobility. As previously anticipated, the overall dimension of the robot (Figure 2.5) has been designed to suggest to human footprint: the chassis has an elliptical and non-axisymmetric shape.

Moreover, the robotic platform has a mass of 4.5 Kg, suspended by four wheels: two driven wheels with a steering angle, and two standard off-centered passive castor wheels. Both forward and backward motions of the first two wheels occur by using two different *BrushLess Direct Current* (BLDC) motors, while their steering angle is controlled by two independent *stepper motors* through a pinion gear transmission. The two driven wheels represent the locomotion units, whose steering angle is measured by an absolute encoder mounted on the chassis.

On the one hand, this specific structure has the advantage of both high maneuverability and stability in reduced environments during high-dynamics motions. Another advantage is the proper mechanical design of the wheels suspensions

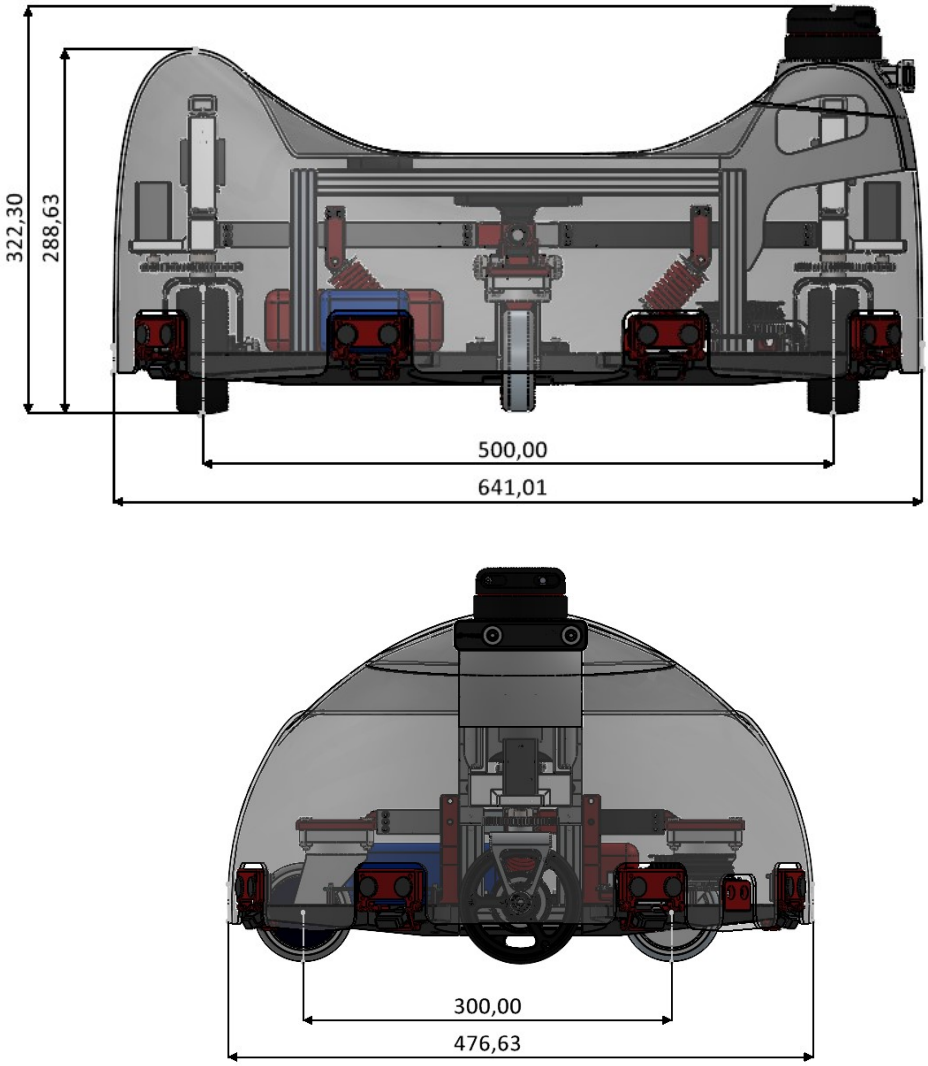
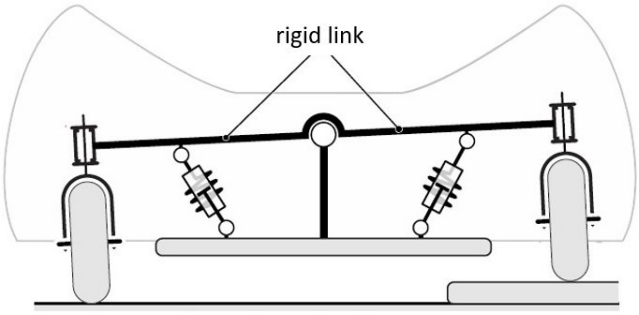


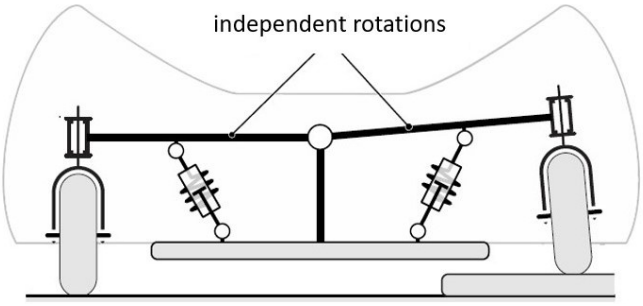
Figure 2.5: Paquitop's Dimensions expressed in millimeters

system, as guarantees a close contact of the wheels with the ground giving stability to the system. In this regard, the platform can assume two configurations (Figure 2.6 [24]): a one-arm or two-arms. The one-arm configuration guarantees a more stable behaviour when carries out a curve, unlike the other configuration that presents many oscillations. However, the two-arms configuration provides as a better response passing over an obstacle.

On the other hand, to obtain high manoeuvrability, the robot offers a redundant actuation, which implies the omnidirectional motions in the plane. As a consequence,



(a) *One-Arm Configuration*



(b) *Two-Arms Configuration*

Figure 2.6: Suspension System of Paquitop

this makes it difficult the motion planning and controlling.

The expected performances strongly influence the *kinematic architecture* of Paquitop, shown in Figure 2.7 [24]. To such aim, four frames are defined, each characterized by the same orientation. The main frame $\{c\}$ is the chassis one, whose origin coincides with the midpoint of the major axis connecting the two driven-steering wheels center. On the latter lies their frame, named $\{wr\}$ for the right wheel and $\{wl\}$ for the left one. The pose of the robot chassis, computed with respect to the space frame $\{0\}$, is determined by a coordinates vector $p_c^0 = [\gamma_c, x_c, y_c]^T$, where γ_c is the rotation of the chassis around the axis z of the frame $\{c\}$, while x_c and y_c are the coordinates of the mobile platform. The center position of each wheel i, is described with respect the chassis frame $\{c\}$ by a vector $p_i^c = [x_i, y_i]^T$. The left and right wheels are placed respectively at a distance $p_{wl}^c = [0, a]^T$ and $p_{wr}^c = [0, -a]^T$ from the frame $\{c\}$, where the parameter a represents half major axis. The wheels motion is defined by two specific parameters: an angle δ_i around the axis z, along which the wheel i ($\{wr\}$ or $\{wl\}$) can steer, and an angular velocity ($\dot{\theta}_i$) for traction motions. These two parameters constitute the actuation variables of the robot, two for each driven-steering wheel, enclosed within the actuation

vector $q = [\delta_{wl}, \dot{\theta}_{wl}, \delta_{wr}, \dot{\theta}_{wr}]^T$. In addition to this last parameter, it is necessary to introduce another, in order to define its kinematic equation: the 3-dimensional velocity twist $V_b = [\dot{\gamma}_c, \dot{x}_c, \dot{y}_c]^T$, obtained deriving its pose vector p_c^0. The actuation vector q and the platform velocity twist V_b are linked by a relationship (2.1), defined as follows:

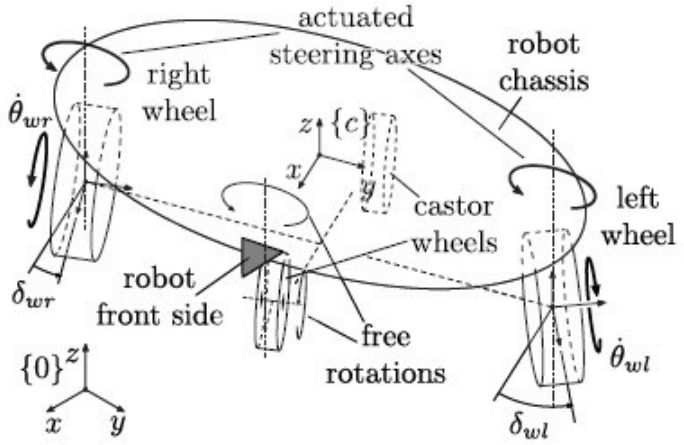


Figure 2.7: Kinematic Architecture of Paquitop

$$V_b = [\dot{\gamma}_c, \dot{x}_c, \dot{y}_c]^T = \frac{r}{y_{wl} - y_{wr}} \begin{bmatrix} -c_{\delta_{wl}} & c_{\delta_{wr}} \\ -y_{wr}c_{\delta_{wl}} & y_{wl}c_{\delta_{wr}} \\ x_{wl}c_{\delta_{wl}} + (y_{wl} - y_{wr})s_{\delta_{wl}} & -x_{wl}c_{\delta_{wr}} \end{bmatrix} \begin{bmatrix} \dot{\theta}_{wl} \\ \dot{\theta}_{wr} \end{bmatrix} \quad (2.1)$$

where r is the wheels radius. As the mobile platform cannot slide, a pure rolling condition must be applied, imposing the following kinematic constraint (2.2):

$$((x_{wl} - x_{wr})c_{\delta_{wl}} + (y_{wl} - y_{wr})s_{\delta_{wl}})\dot{\theta}_{wl} = ((x_{wl} - x_{wr})c_{\delta_{wr}} + (y_{wl} - y_{wr})s_{\delta_{wr}})\dot{\theta}_{wr} \quad (2.2)$$

By substituting the previous coordinates vectors $p_{wl}^c = [0, a]^T$ and $p_{wr}^c = [0, -a]^T$, the equation (2.1) and the kinematic constraint (2.2) are expressed as (2.3):

$$V_b = [\dot{\gamma}_c, \dot{x}_c, \dot{y}_c]^T = \frac{r}{2a} \begin{bmatrix} -c_{\delta_{wl}} & c_{\delta wr} \\ ac_{\delta_{wl}} & ac_{\delta wr} \\ as_{\delta_{wl}} & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_{wl} \\ \dot{\theta}_{wr} \end{bmatrix} \qquad with \quad s_{\delta_{wl}}\dot{\theta}_{wl} = s_{\delta_{wr}}\dot{\theta}_{wr} \quad (2.3)$$

The equation of the velocity inverse kinematics (2.4) between the twist velocity and the actuation vector is possible to derive:

$$\begin{bmatrix}\dot{\theta}_{wl}\\ \dot{\theta}_{wr}\end{bmatrix} = \frac{1}{r}\begin{bmatrix}\pm\sqrt{(\dot{x}_c + a\dot{\gamma}_c)^2 + (\dot{y}_c)^2}\\ \pm\sqrt{(\dot{x}_c - a\dot{\gamma}_c)^2 + (\dot{y}_c)^2}\end{bmatrix} \qquad \begin{bmatrix}\delta_{wl}\\ \delta_{wr}\end{bmatrix} = \begin{bmatrix}atan2(\frac{\dot{y}_c}{r\dot{\theta}_{wl}}, \frac{\dot{x}_c + a\dot{\gamma}_c}{r\dot{\theta}_{wl}})\\ atan2(\frac{\dot{y}_c}{r\dot{\theta}_{wr}}, \frac{\dot{x}_c - a\dot{\gamma}_c}{r\dot{\theta}_{wr}})\end{bmatrix} \tag{2.4}$$

This equation has four independent solutions. If $\dot{\theta}_{wl}$ and $\dot{\theta}_{wr}$ are null, there are not solutions, while if only a null velocity is required, namely $\dot{\theta}_{wl} = 0$ or $\dot{\theta}_{wr} = 0$, there is no point in evaluating the corresponding steering angle.

In addition to omnidirectional movement, Paquitop can have several locomotion strategies (Figure 2.8 [25]), obtained via the different positioning of the two driven wheels.

Configuration I, represents the general case of motion, enclosing all other configuration types. It defines the omnidirectional motion, obtained with the independent control of the two steering angles: $\delta_{wr} \neq \delta_{wl}$. The *instantaneous center of rotation* (i.c.r) is defined by the intersection of the two wheels axes.

Configuration II, outlines the ability of the robot to translate along a fixed direction without any rotation. The equal steering angles $\delta_{wr} = \delta_{wl} \neq 0, \frac{\pi}{2}$ cause the parallel and non-coincident wheels axes. As a consequence, the angular velocity of the two wheels is equal: $\dot{\theta}_{wr} = \dot{\theta}_{wl}$.

Configuration III, represents the differential drive as the steering angles are both null: $\delta_{wr} = \delta_{wl} = 0$; the i.c.r lies on the axis y of the frame $\{c\}$. The robot is able to have an angular velocity, without the possibility to have that along the axis y.

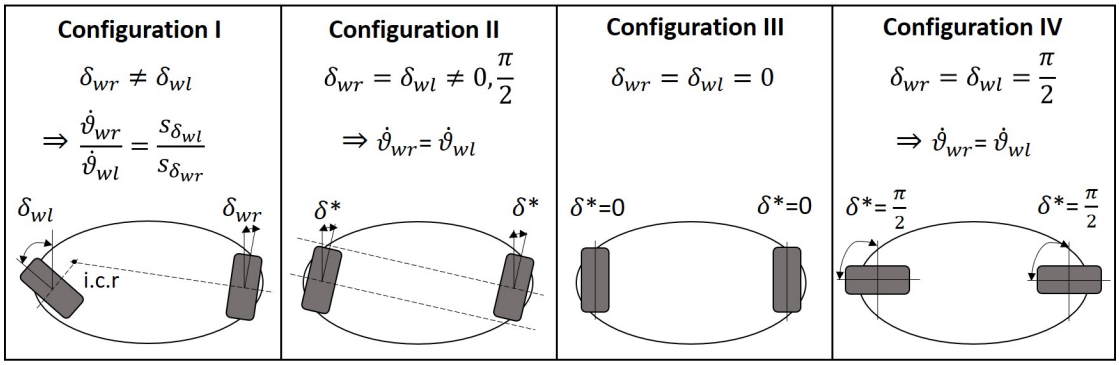


Figure 2.8: Locomotion Strategies

Configuration IV, enables the pure translational motion as the Configuration II, as the steering angles have the same value: $\delta_{wr} = \delta_{wl} = \frac{\pi}{2}$. The wheels axes are parallel and non-coincident and their angular velocity is the same. Moreover, this configuration can assume the car-like locomotion (fFigure 2.9 [25]), by modifying the steering angle to such one of the two wheels (Configuration IV.a-IV.b) or steering the wheels by opposed small angles around $\frac{\pi}{2}$ (Configuration IV.c).

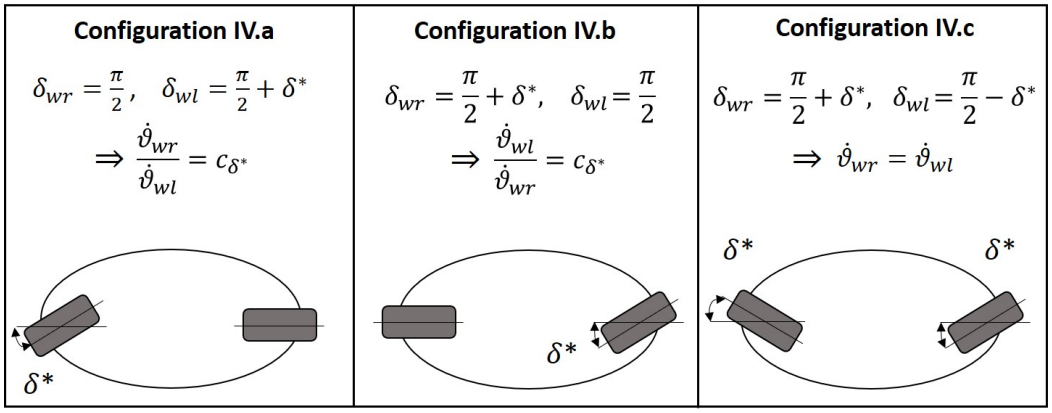


Figure 2.9: Locomotion Strategies of Car-like

All the configurations previously described are actuated with a velocity as like that of a human walk ($\approx$ 1-1.5 $\frac{m}{s}$) and an appropriate acceleration ($\approx$ 0.5-1.5 $\frac{m}{s^2}$), as Paquitop is built to follow a human being.

2.3 Electronic Design

The electronic design of the robot follows the main idea behind the design of the entire architecture, namely to develop a modular structure that allows the subdivision of all the robot's physical and non-physical parts into the two main layers previously mentioned. Focusing on the implemented electronic components, those enabling the navigation are inside the *navigation layer*, while those adopted for the motion control during the platform navigation are inside the *motion layer*. Figure 2.10 [26] shows the whole electronic architecture with such subdivision, where the motion layer and the Radio Controller device represent the *Low-Level Control Architecture*, while the Autonomous Navigation block, containing the navigation algorithms and the electronic devices, enabling the autonomous navigation of Paquitop represent the *High-Level Architecture*.

The entire electronic architecture is described in detail below. It revolves around to a component, namely the *MicroController*, responsible to read the inputs coming from a *Radio Controller*, a digital device used to remotely control the robot from a distance, or an *on-board computer*, equipped with all navigation algorithms employed in case of an autonomous drive. About the on-board computer has been chosen a Jetson Nano, for its computational complexity of the autonomous navigation, connected to two the exteroceptive sensors, namely LiDAR and camera, described in the following sections.

The adopted MicroController is Teensy 4.1 board, usually used in *Unmanned*

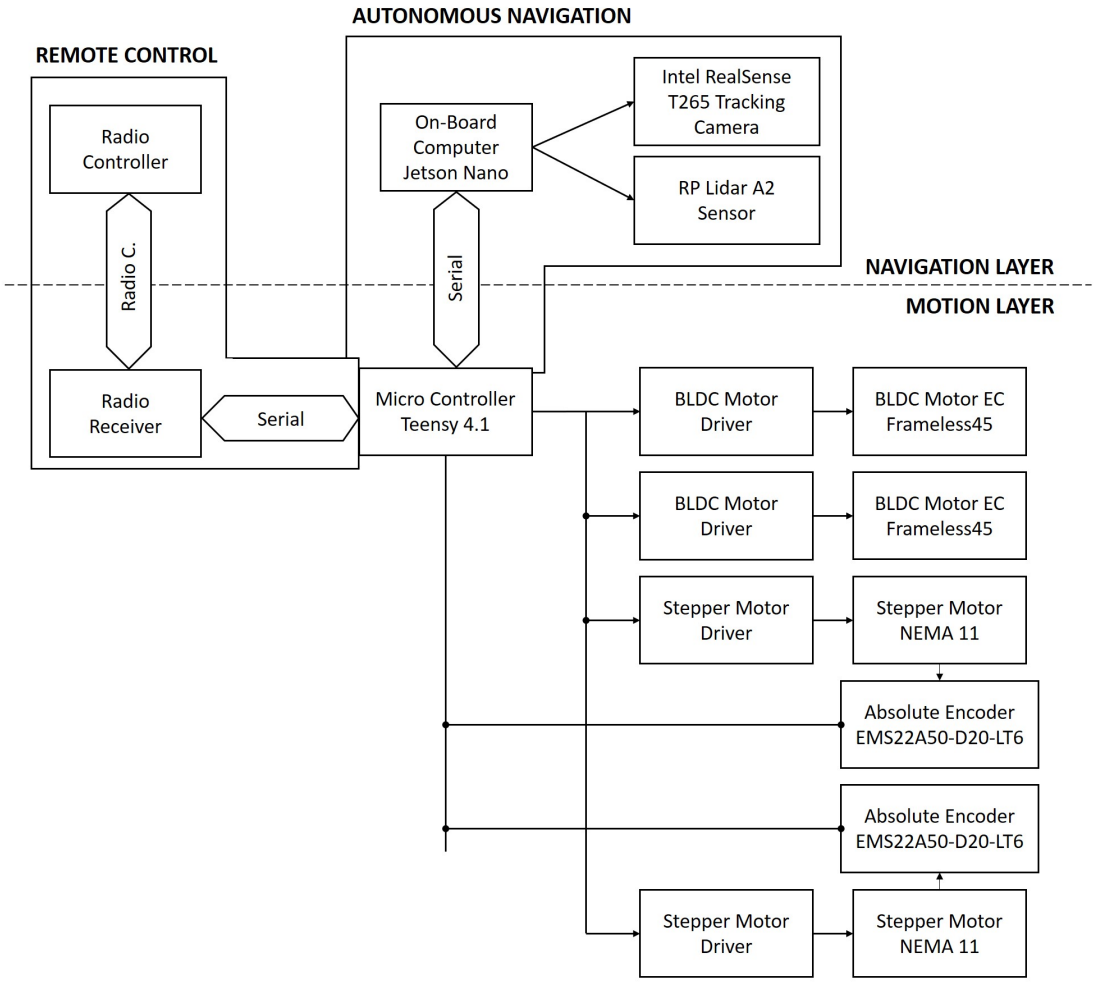


Figure 2.10: Electronic Architecture organized in the Navigation and Motion Layers

Aircraft System (UAS), and Arduino compatible, programmed in C++ language. This device has been chosen according to different specifications, among which the communication channels, the number and type of pins available, the processor performance, the floating-point unit and finally the number of timers and dimensions. Moreover, the MicroController has the purpose of computing the inverse kinematics of the platform, namely to determine the joint parameters providing a desired configuration of the end-effector pose (position and orientation), reached remotely or autonomously.

The *remote control* of the platform is performed by using a Digital Telemetry Radio System, in particular the FrSky 2.4G ACCST Taranis X9D Plus, consisting of

two parts: a Transmitter and a Receiver. The logic behind this device is to transmit the selected command to the Receiver at a frequency of 2.4 GHz, acquired in forms of radio waves and then converted into electrical signals to send to the motors. The input commands of the Radio Controller are converted in velocity commands by means of a mapping function, adopted to process the joystick commands and to guarantee the platform controllability. This Radio System is equipped with a switches' layout (Figure 2.11 [27]), characterized by different channels to control the directional movement of the robot, its Working Configurations, the motor state and both linear and angular velocities of the platform. The Digital Telemetry Radio System Layout has the following channels structure:

- **Channels [1,2]**, controls on the hand the linear velocity in Working Configuration I, on the other hand the transversal velocity and angular velocity in both Configuration II and III;
- **Channel [3]**, controls the angular velocity in Working Configuration I.
- **Channel [4]**, enables the engine ignition and shutdown;
- **Channel [5]**, activates the Configuration Mode III.a, III.b, III.c;
- **Channel [6]**, selects the Working Configuration I, II, III;

To manually control the mobile robot through the selection of the channels described below, the Radio-Receiver needs of a serial device to communicate with the MicroController at a frequency of 50 Hz. In particular, the Radio Controller receives the instructions in the form of velocity twists w commands through a map, and in turn the MicroController computes the inverse kinematics through the equation 2.4 to evaluate the reference actuation variables. Then the degrees of actuation are controlled employing a closed-loop speed control, by adopting a *Proportional-Integral-Derivative* (PID) control (Figure 2.12 [26]). The MicroController interfaces to a driver, to control the motor in terms of velocity (or current depending on the control mode) and direction. The forward/reverse direction is given by the polarity of the voltage source, while the velocity is chosen by a *Pulse Width Modulation* (PWM) signal at a frequency of 536 Hz. The voltage source value (V_{ctrl}) can be between 0 V and 3.3 V, and is influenced by both speed, expressed in rpm and current, present in the winding of the motor. In fact, to a 0 V value corresponds a 0.0 A value and a rotation velocity $-w_{max}$, while to a 3.3 V value corresponds a 15.00 A value and a rotation velocity w_{max}. Moreover, the Motor Driver supports an inner closed-loop current control (I_{fb} and I_{ref}) with the BLDC Traction Motor, linked to by the Hall-Effect Speed Sensor, whose aim is to control the motor speed and to close the loop speeds of the MicroController. The velocity signals provided by the speed sensor, are transformed by the driver

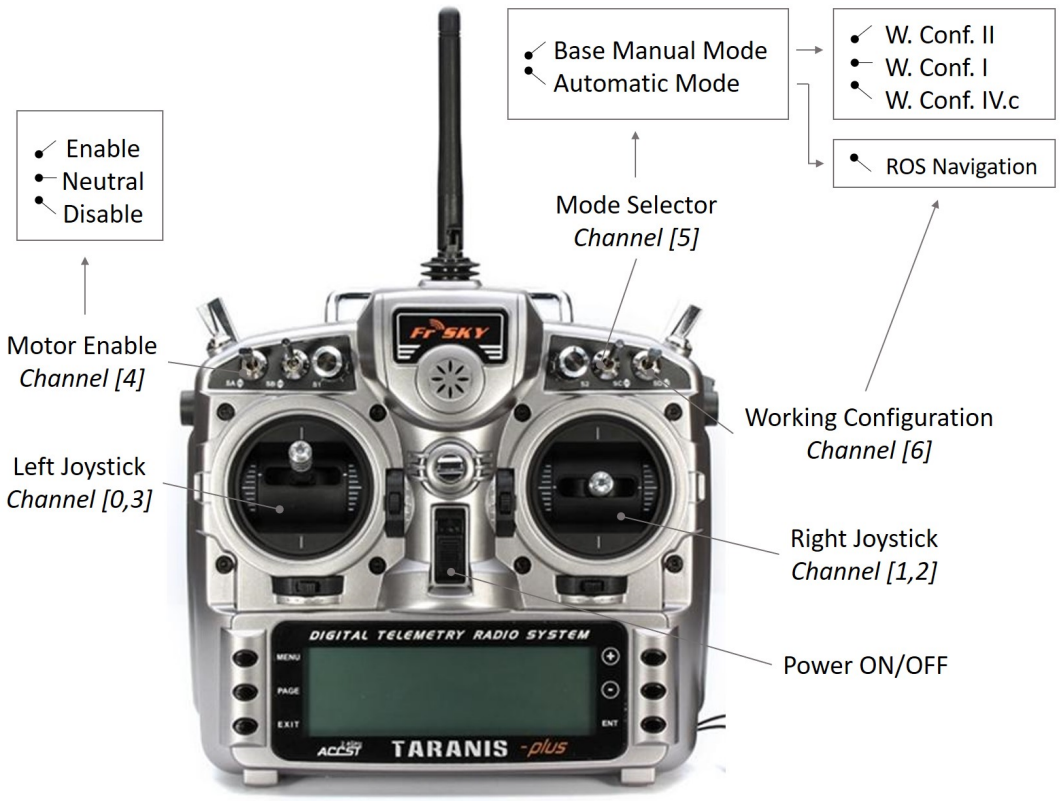


Figure 2.11: Digital Telemetry Radio System Layout

into an analog signals from 0 V to 3.3 V. These values can be modified by using a programming interface of the motor driver, assuming the form of a speed regulator (open-loop speed control).

To actuate the steering axes, the MicroController communicates with the two Stepper Motors, which receive as inputs a voltage and current values from the Stepper Motor Drivers. An Absolute Encoders read and update the steering angle values at the frequency of 1 Hz, and communicate with the MicroController through two *Serial-Peripheral-Interface* (SPI) buses: one is used to activate and program the Stepper Motor Drivers and the other is used to read the Absolute Encoders. At frequency of 200 kHz, the steering angles are converted from radiant to number of steps of the stepper motor, useful to drive the steering system to the correct angles.

As previously mentioned, the following sections will describe RPLidar A2 and camera T265.

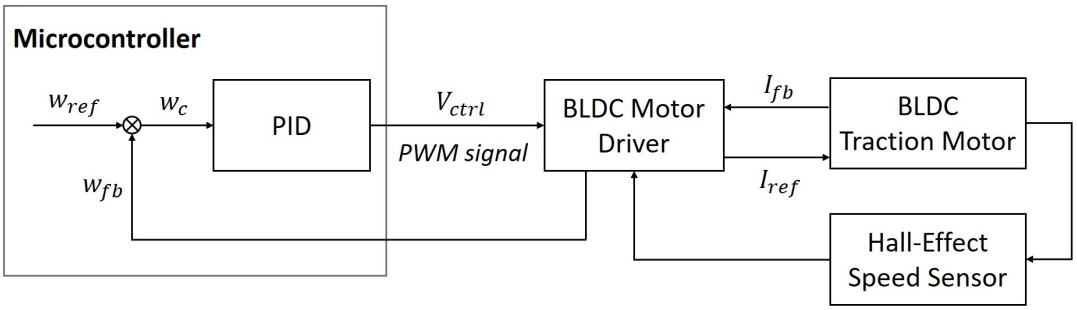


Figure 2.12: PID MicroController and Traction Unit Control

RPLidar A2

RPLidar A2 (Figure 2.13 [28]) is a two-dimensional 360-degree laser scanner developed by SLAMTEC. This device is equipped with a scanning system, a communication and power interface and an electric motor. The scanning system consists of two optical lens, deployed for the transmission and reception of the emitted signals. The clockwise rotation of the core occurs by using an electrical motor, which permits the scanning of the environment. The user can obtain the scan data through the communications interface, namely Serial/USB port. This device adopts the *triangulation ranging principle*, wherein a light beam is projected by sensor on the target and then reflected into a receiver, forming a triangle (Figure 2.14 [29]). The information released by the detector component is the value in polar coordinates of the distance d and angle θ created between the two beams.

Figure 2.13: RPLidar A2

Moreover, LiDAR adopts coordinate system of the left hand in its barycentre, but the default one adopted in ROS environment is as shown in Figure 2.15 [30].

The technical specifications of RPLIDAR A2 are reported below:

- 8 m ranging distance;
- 10 Hz typical scanning frequency, which can be adjusted in the range between 5-15 Hz;
- Good performance in both indoors or outdoors environments, even without direct exposure to sunlight;
- Low power consumption.

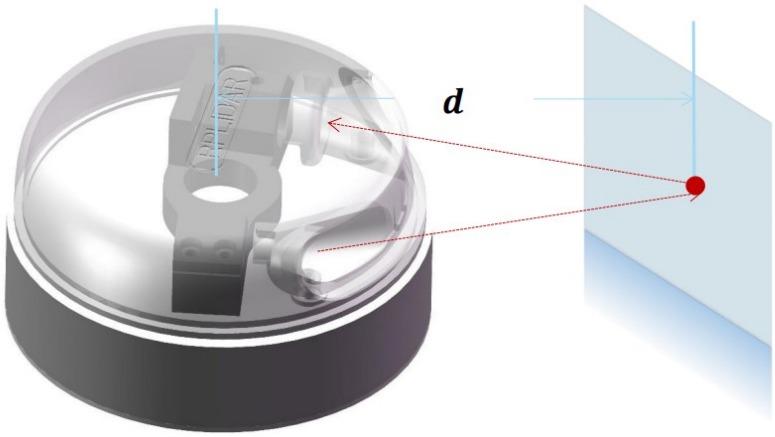


Figure 2.14: The RPLidar Working Schematic

Intel RealSense Tracking Camera T265

The *camera T265* (Figure 2.16 [31]) is a six-degrees-of-freedom (6DoF) tracking camera providing the odometry information develop by Intel Real Sense. This device consists of two fisheye lens cameras, an Inertial Measurement Unit (IMU) and an Intel Movidius Myraid Visual Processing Unit (VPU). The fisheye lenses collect inputs with a field of vision of 170 degrees; an IMU is a device that includes *gyroscopes* to measure the angular rate and an *accelerometers* to compute a force and an VPU is a microprocessor that has as objective image processing and computer vision. The device fuses inputs of the fisheye lens and the IMU data for providing an accurate real time position. Then the data fusion from these sensors is sent into VSLAM, running on VPU for visual-inertial odometry.

The coordinates system of the device is represented in Figure 2.17 [32].

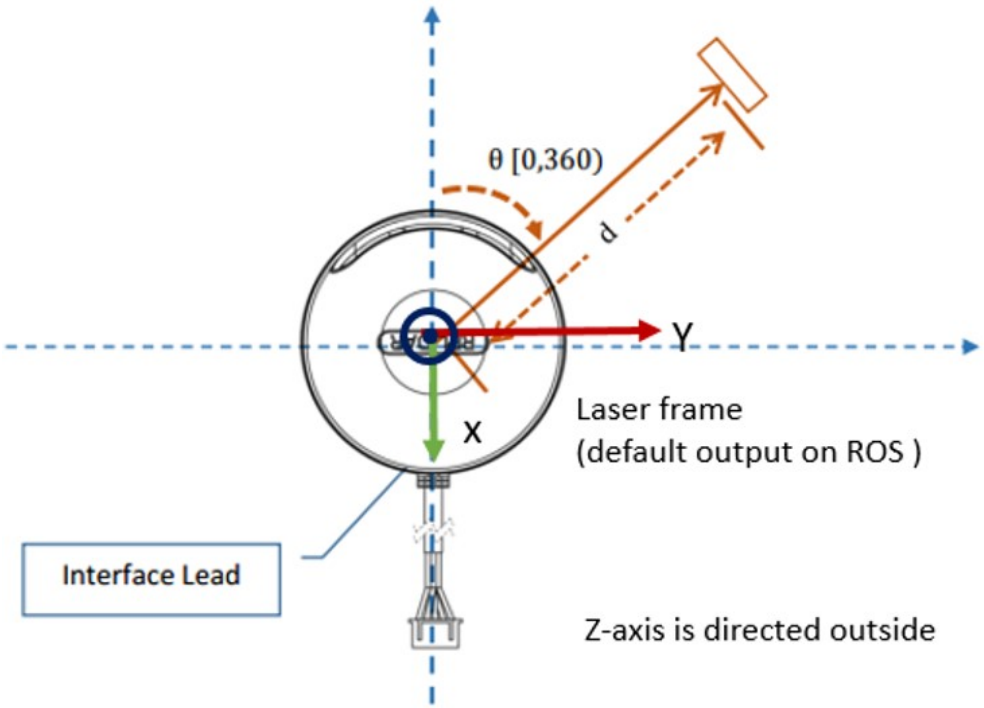


Figure 2.15: Coordinates System of RPLidar in ROS Environment

Figure 2.16: Intel RealSense Tracking Camera T265

2.4 Software Design

The software design constitutes the *High-Level Architecture* of the robotic system, namely all algorithms that enable the autonomous navigation of the platform. Persons tracking and monitoring in unknown environment can occur addressing some problems regarding the autonomous navigation of the platform: *obstacle avoidance*, *path planning*, *localization* and *mapping*. To solve these navigation tasks,

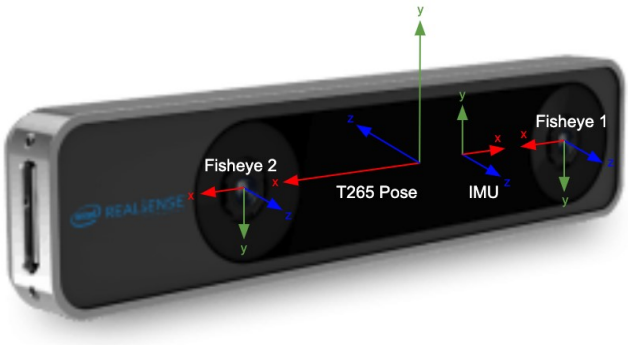


Figure 2.17: Coordinates System of the Tracking Camera

exteroceptive sensors are employed, LiDAR and camera, which are based on *Visual Simultaneous Localization and Mapping* (VSLAM) algorithm. The latter gives the robot the ability of mapping the surrounding environment in which operates, locating itself in it. Information coming from sensors is the inputs for the *global path planner*, which builds a free-path for reaching the patient, guaranteeing a safe path free of obstacles thanks to the *local planner*. By implement a modular approach, it is possible to draw the software architecture as the interaction of the two layers previously introduced (Figure 2.18). The software design will be described in detail in *Chapter 3*.

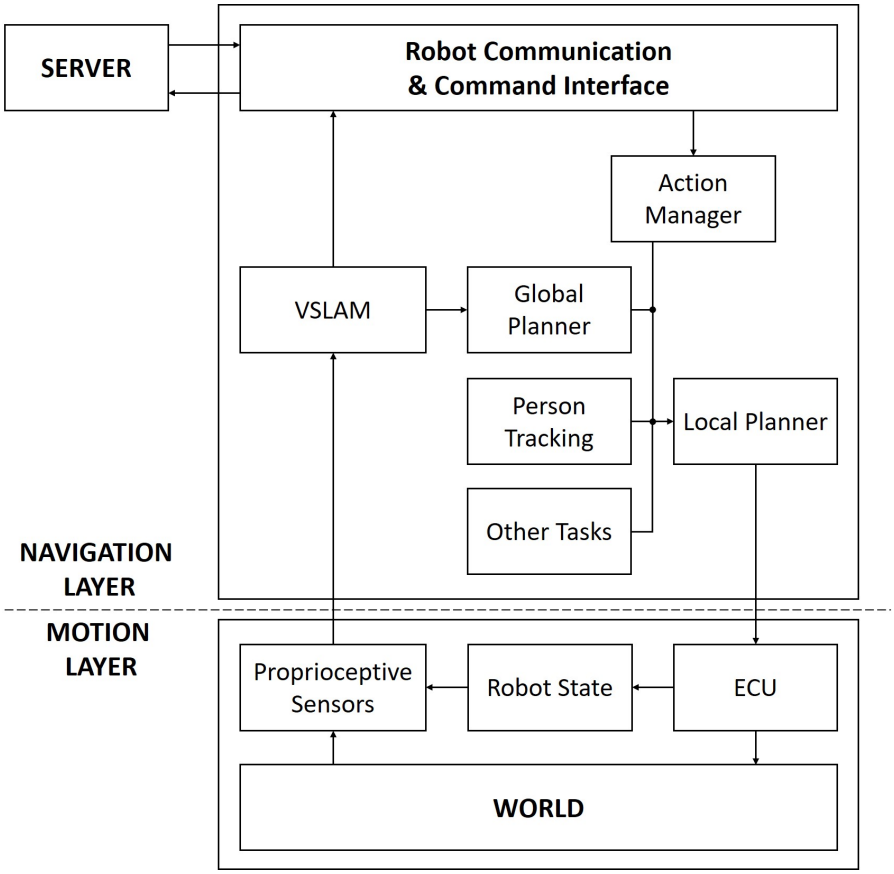


Figure 2.18: Software Architecture

Chapter 3

High-Level Architecture and Packages Implementation

In this chapter the High-Level Architecture of the robot will be described in detail, specifying all ROS packages needed to achieve the full robot autonomy. In this sense, the concept of *Robotic Paradigm* is defined with the aim of developing and describing the internal software organization of the Paquitop's robotic system.

All the kinematics and dynamic properties of the robot will be described within the *URDF format*, needed to visualize and control the motion planned by the developed software architecture, within the ROS simulation environments, namely RViz and Gazebo.

3.1 Robotic Paradigm

The term paradigm encloses a set of assumptions and/or techniques which characterize an approach to a class of problems [33]. In robotics, a paradigm is a mental model of how a robot works, which provides the principles to organize its control system for reaching a required application.

Each robotic paradigm is organized into subsystems or *modules*, each one orientated to a specific task. The modular logic allows of designing, developing and testing each block separately in a simpler and faster way. This property enhances the *flexibility* and the *extensibility* of the control system, allowing an incremental development over time through the integration, the verification and the improvement of its component.

A robotic system should be as *reactive* as possible to sudden changes in the environment, adapting and altering the control strategies accordingly *(adaptability)*. Also, it should have the ability to maintain and achieve different objectives even in case of unexpected events and sudden malfunctions *(robust and reliable*

system). To best fulfill these characteristics, the robot must own *multiple sensors*, as individually they suffer of limited accuracy, reliability and applicability, which must be compensated using several complementary sensors.

Each paradigm is based on the organization of the main primitive functions of a robot, which can be divided into following three general categories or modules:

- **SENSE**: encloses the functions consisting in perceiving the environment through sensory data, and in returning the perceived information, useful for subsequent modules;
- **PLAN**: incorporates all functions exploiting a priori knowledge of the surrounding environment, to produce directives or tasks to be performed by a robot;
- **ACT**: selects functions producing the commands to send to the actuators, from sensory information and directives.

There are currently three major paradigms for organizing intelligence in robots named *Hierarchical*, *Reactive* and *Hybrid (Deliberative/Reactive).* Applying the right paradigm to a robotic system, is the main key to being able to successfully program a robot for a particular application.

3.1.1 Hierarchical Paradigm

In the Hierarchical Paradigm, the primitive functions are organized in the following manner:

$$\text{SENSE} \longrightarrow \text{PLAN} \longrightarrow \text{ACT}$$

The robot perceives (SENSE) the surrounding environment on which operates, and it manages the processing data, which are elaborated and gathered into one internal world model, that contains:

- an a priori representation of the environment in which the robot operates (map);
- a perceived sensory information.

The perception module is used to establish and maintain a correspondence between the internal model of the world and the external world.

The PLAN module uses the built world model to generate a plan, based on the tasks that the robot has to execute.

Finally, the ACT module generates the commands to send to the motor control, which takes care of the real-time control of the actuators. After the ACT phase,

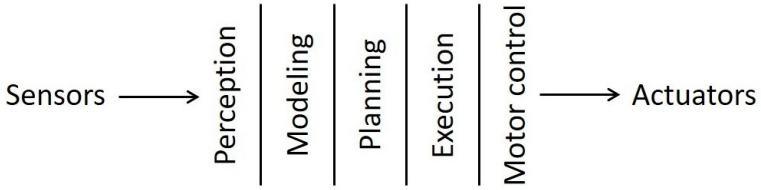


Figure 3.1: Horizontal and Sequential Subdivision of the Hierarchical Paradigm

the operations are repeated until the set objective is reached, carried out by the robot in a strictly horizontal and sequential manner, as shown in Figure 3.1:

A robot employing this approach, needs a complete and correct knowledge of the world for obtaining a predictable functioning and planning. This guarantees an efficient and stable system. If this information is inaccurate, or if the world has undergone changes during the planning phase, the execution will be wrong. In conclusion, this approach is characterized by a low adaptability to real-time changes of the environment, and consequentially characterized by a low reactivity. Furthermore, the strict sequencing of the modules causes low parallelism and delays in updating related to the changes of a dynamic environment.

3.1.2 Reactive Paradigm

In response to the problems encountered by the previous paradigm in terms of poor responsiveness and performance, an opposite approach called Reactive has been developed. In the Reactive Paradigm, the planning phase is completely eliminated, so there is a direct relationship between sensors and actuators, as shown below:

$$\text{SENSE} \longrightarrow \text{ACT}$$

Perception and action cooperate to produce timely responses in dynamic and unstructured environments, allowing the robots to operate in real-time. This characteristic enables to remove an internal state that had to be updated continuously, improving the response times. The representation of the world is not memorized, but extracted in real time from the world through sensors, and consequently, there is no prior planning of actions.

A Reactive robotic system decomposes its functionality into *behaviours*, which are reactions or stimuli in real time to information perceived by the environment. Behaviours are a direct mapping between the sensory inputs and the pattern of motor actions, used to achieve a specific task [33]. This is performed by different behaviours, each one with a specific competence. The overall behaviour is determined by the set of behaviours present.

Unlike the Hierarchical Paradigm characterized by a horizontal and sequential subdivision, the Reactive Paradigm has a vertical and parallel subdivision of the information chain, as shown in Figure 3.2:

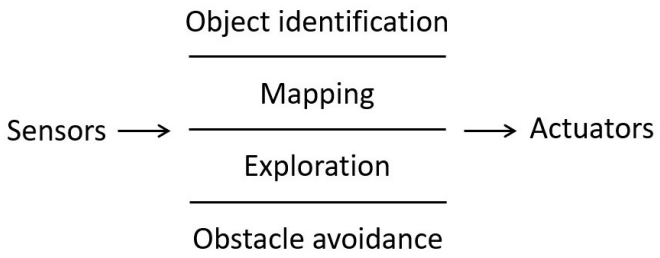


Figure 3.2: Vertical and Parallel Subdivision of the Reactive Paradigm

Each behaviour exploits a local representation of the world, and it elaborates the best action to fulfill independently of other pre-existing and active behaviours. The vertical decomposition produces many information flows, each relating to a particular function assigned to the robot. In this way, each sequence deals with a specific aspect in the overall functioning of the system, and can be developed in parallel with other processes.

In conclusion, constructing a robotic system with a Reactive Paradigm has several advantages:

- modular structure with independent behaviours between them, which enables to easily test them in isolation from the system;
- incremental extension of the capabilities of a robot by having more behaviours;
- high adaptability to changes in the environment because of the real-time response;
- parallelism in the control;
- inexistence of the world model, but only information persisting for a short period of time;
- low complexity of each level and low overall computational cost of the system.

Regarding the disadvantages can be listed as follows:

- difficulty in predicting the global behaviour of the robot in advance;
- complexity of modules management if their number is high, and difficulty in resolving conflicts.

3.1.3 Hybrid Paradigm

The architecture using the Reactive behaviours but also incorporates a general planning, are named Hybrid. The Reactive Paradigm removes planning or any functions which involved remembering or reasoning (deliberative functions) about the global state of the robot relative to its environment [33]. Consequently this implies, for example, a non-optimal path planning and no selection of the best behaviours to accomplish a task. In order to include all deliberative functions, the Reactive Paradigm has undergone an extension, giving rise to the Hybrid Paradigm, consisting of a reactive and a deliberative portions, just a fusion of Hierarchical and Reactive Paradigms, which combines a planning efficiency with the flexibility of Reactive control systems.

The primitive functions are organized in the following way:

$$\text{PLAN} \longrightarrow \text{SENSE, ACT}$$

The planning phase is performed in a single step, while SENSE and ACT primitives in another one: this was due to the fact that planning spent a lot of time and it required a global knowledge of the surrounding environment, while the SENSE and ACT blocks performed in real time. The primitive function PLAN includes all deliberations and global world modelling: the robot plans how to perform a mission using a task-oriented global world model, then it decides what skills are from hire to face a mission and, subsequently, it activates a set of skills (SENSE-ACT) to execute the plan. The operations would execute until the plan is completed, then the planner would generate a new set of skills, and so on. To notice the use of the term skill rather than behaviour, used into the Reactive Paradigm to identify purely reflexive behaviours.

Generally Hybrid architectures consist of the following modules:

- **Sequencer**, generates all behaviours to use to accomplish a specific subtask, and also it determines the sequence and activation conditions.
- **Resource Manager**, allocates resources to behaviours performing checks. For example, it verifies if all sensors can detect at a sufficient range, or if they can update fast enough to match the robot's desired velocity.
- **Cartographer**, creates and stores a map, and also it contains a global world model and information for accessing the data.
- **Mission Planner**, constructs a mission plan, and it traduces the commands into robot terms after the interaction with the human.
- **Performance monitoring and problem-solving**, checks the execution of each plan verifying their state.

3.2 Autonomous Navigation Architecture

This section introduces a general autonomous navigation architecture (Figure 3.3) based on a *Hybrid approach*, which makes use of both a Reactive control and a Deliberative planning. The Reactive part deals with the local navigation, that depends on both the obstacles and the commands of the deliberative planning, containing the waypoints for reaching the goal.

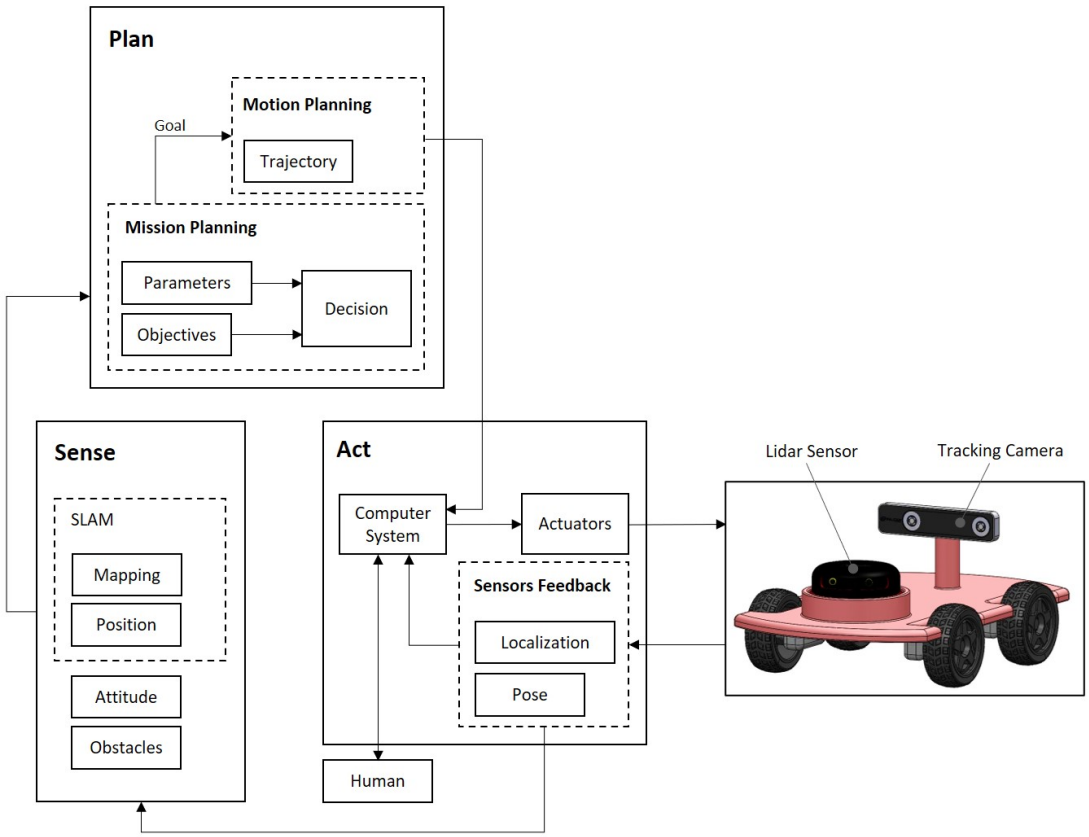


Figure 3.3: General Autonomous Navigation Architecture of a robotic platform equipped with Lidar and Tracking Camera

In order to perform the autonomous navigation in any environment, the sensing components are necessary for mapping, localization and detection of obstacles. In general, by considering a robot equipped with a Lidar sensor and a tracking camera, mapping and detection of obstacles are performed by the first device, which scans the environment and detects the obstacles; while localization is provided by the second device by means of its odometry. The sensor fusion is applied for obtaining the simultaneous localization and mapping, or SLAM, useful for the

collision avoidance in a dynamic environment. The sensorial data and all the tasks are combined and managed by the planner, which plans a sequence of movements and waypoints to achieve the goal. The planner builds a trajectory that is sent to the action planner, which generates references for the implementation system or actuators. Their motion is controlled by a human through a computer system, containing the software architecture that allows the autonomous navigation of a robot, where it is also possible to supervise the planned and performed actions, and to receive the constant updating of the sensors data on board the robot.

3.3 SW Architecture: Navigation Stack

The previous architecture has been adopted to develop the Software Architecture (Figure 3.4) of the mobile robot Paquitop. In order to enable autonomous navigation, a set of control algorithms, contained within the Navigation Stack package, have been implemented. All the navigation algorithms belong to specific packages, each of which is associated with one of three primitive functions according to its navigation task. However, many packages will belong to more primitive functions, as shown in Figure 3.4, causing the intersection of three blocks.

First, sensor data of Lidar and camera represent the inputs of the Sense block within the architecture. Lidar uses $rpilidar - ros$ package to scan the environment, while the $hector - slam$ package is used to build the map, and at the same time to provide a pose estimation of the robot within the map itself. The tasks of the latter package are enclosed in the acronym SLAM. A tracking camera is necessary for odometry information. The estimated poses by Lidar and camera, are merged into one by the $robot - pose - ekf$ package, taken into account as a localization parameter.

In order to correctly use the data from different sensors and produce navigation references consistent with the pose of the robot, it is necessary to define the robot as the connection between links and joints, each of which is described by its own reference system. The package that keeps track of these coordinate frames over time in a tree structure, is named tf.

Localization and mapping tasks are addressed in the Sense module, where the $map - server$ package is included, which is responsible for both saving the map built by the $hector - slam$ package, and providing a previously given map.

In order to engender a drivable path, a weighted grid map is built through a Local and Global Costmaps, contained inside the $costmap - 2d$ package. The latter exploits the map built in the Sense module to store and update information about obstacles both locally and globally. The information related to obstacles are used by the $move - base$ package to plan a path through both Global and Local Planners. The performed trajectory is saved through one of nodes contained within

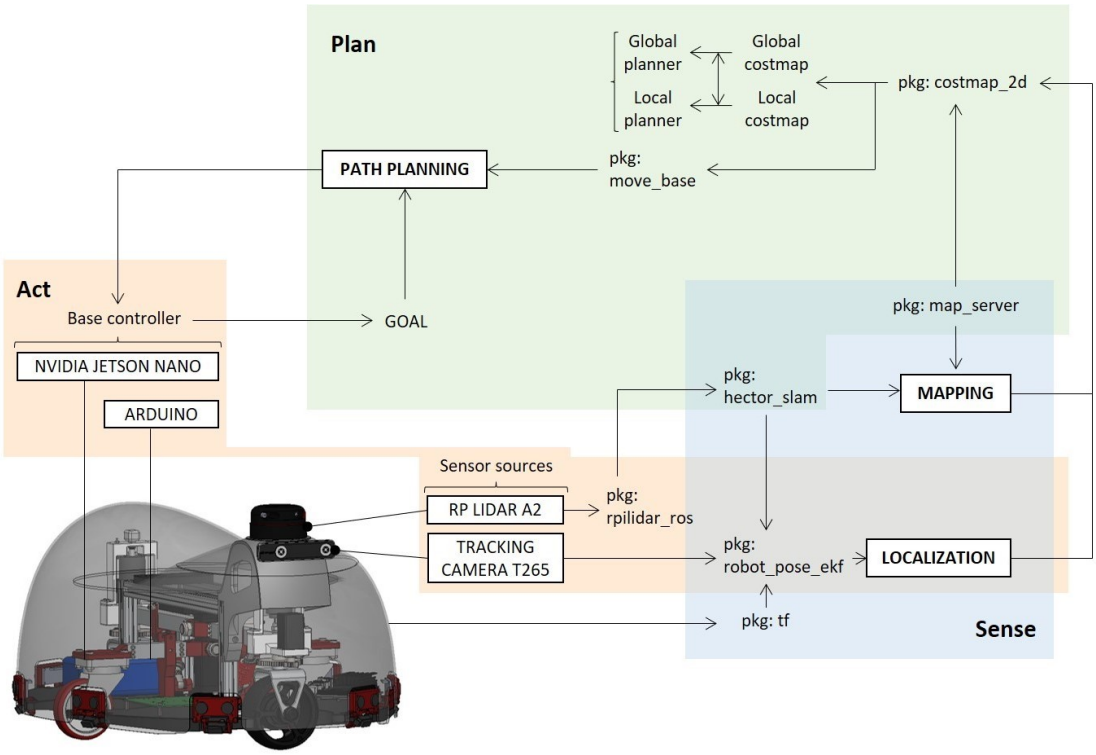


Figure 3.4: SW Architecture of Paquitop

the $hector - slam$ package and, for this reason this package is enclosed in the Plan module. This module also includes $map - server$, $costmap - 2d$ and $move - base$ packages, as useful to the path planning.

Finally, commands, processed sensor data and trigger signals are sent, by means of Arduino, to Jetson Nano, a computer that powers the platform. Both Arduino and Jetson Nano are inside the Act module and, besides them, both sensors in order to maintain a continuous updated pose over time.

The relationship between all several packages in detail, will be described in the next paragraph related to the Navigation Stack package description.

3.3.1 Description

The ROS Navigation Stack is a set of software packages useful for localization, mapping of the environment and path planning for the purpose of autonomous navigation of a mobile robot. The objective of Navigation Stack is to produce a navigable path connecting the starting point with the goal pose, processing data from odometry, sensors and environment map [34]. This package uses a series of different algorithms to navigate through known and unknown environment,

detecting and avoiding obstacles. All algorithms are included in the following packages: $move-base$, $costmap-2d$, tf, $robot-pose-ekf$ and $map-server$.

The Navigation Stack takes information about the surrounding environment from Lidar sensor, which is supported by the LaserScan package through the rplidarNode. Lidar is able to detect the obstacles in all directions sharing and publishing this infomartion in form of **sensor-msgs/LaserScan** messages on **/scan** topic.

To the latter, the $hector-slam$ package subscribes and allows Lidar to extract the map of the unknown environment through one of its nodes named $hecto-mapping$. The grid map information is published on the **/map** topic as **nav-msgs/OccupancyGrid** messages.

At the same time, this node subscribes to the **/tf** topic inside the *TF* package for knowing the information about the frame transformations as **tf/tfMessage** messages. This package allows the robot to keep track of its movements over time and, therefore, of positions of its different links and joints, and also, to associate a frame with each joint and link through a tree data structure.

Another task of the $hector-mapping$ is keeping track of the 2D robot pose, publishing it on **/poseupdate** topic.

In addition to Lidar, also the tracking camera manages the pose of the robot, publishing the start pose as **geometry-msgs/PoseWithCovarianceStamped** messages on the **/initialpose** topic. All information about camera is published on **/camera/odom/sample** as **sensor-msgs/PointCloud** message.

All information regarding the robot pose, extracted by Lidar and camera, are merged by the $robot-pose-ekf$ package (node: $ekf-se$) to estimate the 3D pose of a robot. This package subscribes to both $hector-mapping$ node and **/imu-data** topic, and it publishes the estimated pose on the **/tf** topic, so that the latter contains all inserted frames. In turn, the **/tf** topic publishes its information on the $n--rviz$ node.

The $hector-slam$ package contains a node named $hector-trajectory-server$ that saves trajectory data, extracted from **/tf** topic, and it publishes it on **/trajectory** topic [35].

For the path planning and its execution, the *Move Base* package is needed. This package subscribes to odometry (**/camera/odom/sample** topic), sensor (**/scan** topic) and tf (**/tf** topic) data and goal position (**/move-base/action-topics** topic) messages. The last topic contains all actions that the robot has to perform to reach the target point, while the message containing its coordinates is published on **/move-base-simple/goal** as **geometry-msgs/PoseStamped**. The coordinates can be given from the terminal and RViz. For executing all permitted actions, the *Move Base* package produces velocity commands, published on **/cmd-vel** topic as **geometry-msgs/Twist**, to be send to the mobile base and finally to $serial-node$ that supports Arduino. In order to reach the pre-established pose, the *Follow*

Waypoint algorithm is used, since it is able to set waypoints up to the goal.

Also, the *Move Base* package can also subscribe to a map on the following **/map** and **/map-updates** topic, so it can generate a better path with known information of the environment.

In order to guarantee the navigation process, the *Move Base* incorporates a *Global* and *Local Planner*. The Global Planner takes the current robot position and the goal and generates a trajectory, while the Local Planner manages a small portion of the environment around the mobile base, taking into consideration the dynamic obstacles and the vehicle constraints. The topics related to the planners are respectively: **/move-base/PlannerROS/local-plan** and **/move-base/PlannerROS/global-plan**. Both contain the points of the planned trajectory locally and globally, which are updated every time it is replanned a new trajectory as **nav-msgs/Path** messages. The Local Planner needs data from odometry, published on **/odom** topic as **nav-msgs/Odometry**, provided by the sensors.

Each planner references to a *Global* and *Local Costmap*: the Global Costmap represents the whole environment, while the Local Costmap is a scrolling window moving inside the Global one [36]. All information of both costmaps, regarding the obstacles in the surrounding environment in the form of an occupancy grid, is contained on the following topics: **/move-base/global-costmap/costmap**, **/move-base/global-costmap/costmap-updates**, **/move-base/global-costmap /footprint**, **/move-base/local-costmap/footprint**. In the last two topics the footprint of the robot is published as **geometry-msgs/Polygon** message and it is viewed on RViz. In the first topic the costmap is published containing the occupancy and inflation value of the cells on the map, while in the second topic the costmap value during the navigation are published. Both topics use messages of type **map-msgs/OccupancyGrid**.

ROS allows us to save the data relating to a map of the desired area on **/map** topic through the $map - saver$ node as **nav-msgs/GetMap** message, present within the $map - server$ package. This package contains also $map - server$ node, useful to provide static map data as a ROS Service.

For summarizing the relationship between all running nodes and the topics that they communicated with, a GUI plugin named *rqtgraph* has been used Figure 3.5:

In conclusion, the Navigation Stack setup can be seen in Figure 3.6, schematized in conceptual blocks and divided by areas of operation:

3.3.2 Move Base

The main component of the Navigation Stack is the $move - base$ package, which provides an implementation of an action for reaching a goal with its mobile base [37]. As mentioned before, the Move Base supports two planners to accomplish

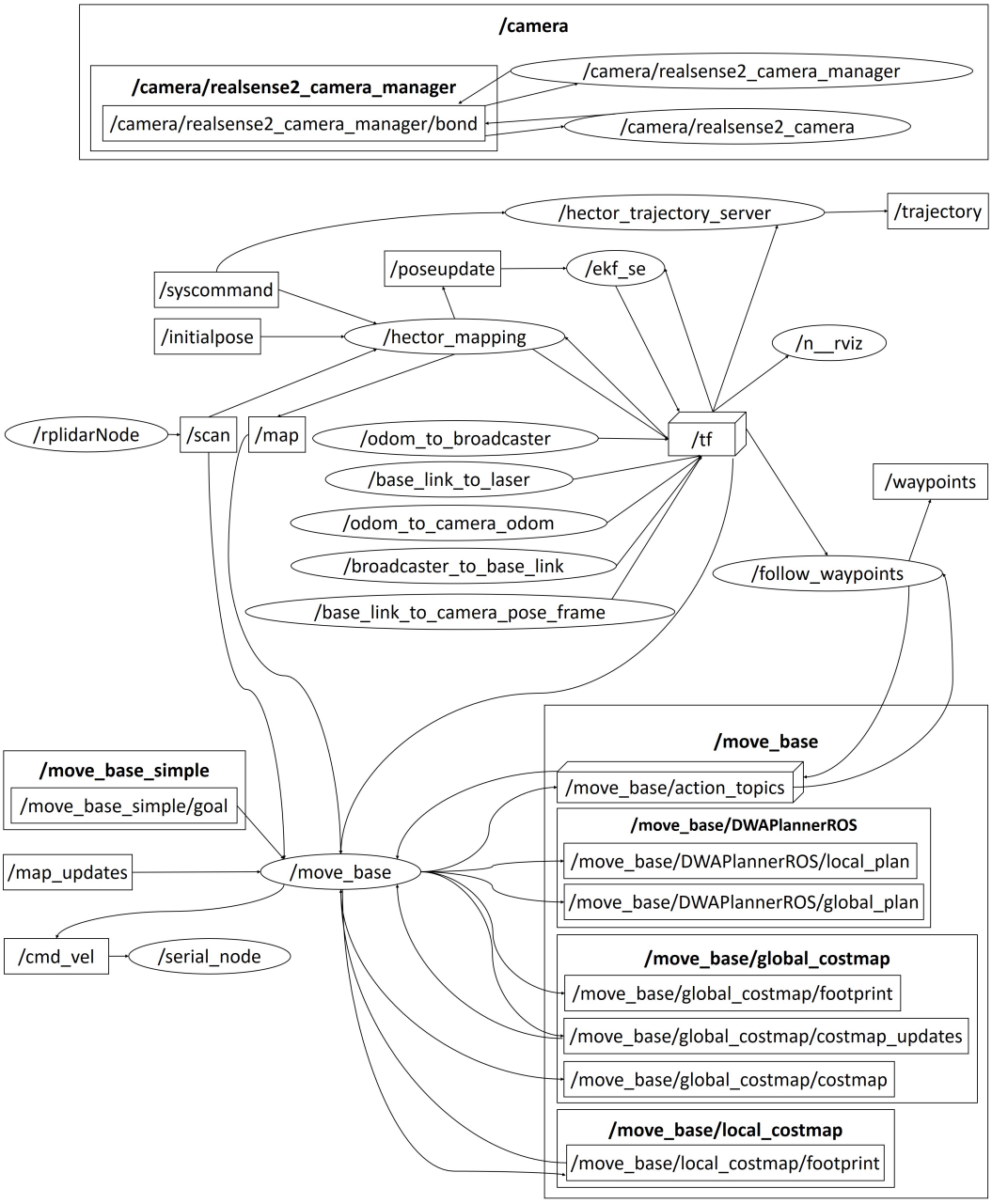


Figure 3.5: Rqt Graph of Navigation Stack

the global task: Global and Local Planner and also, it contains a set of algorithms used for *recovery behaviours*, whose purpose is to decrease the probability that the robot will end up in a stalemate during the navigation. The stall stands for a

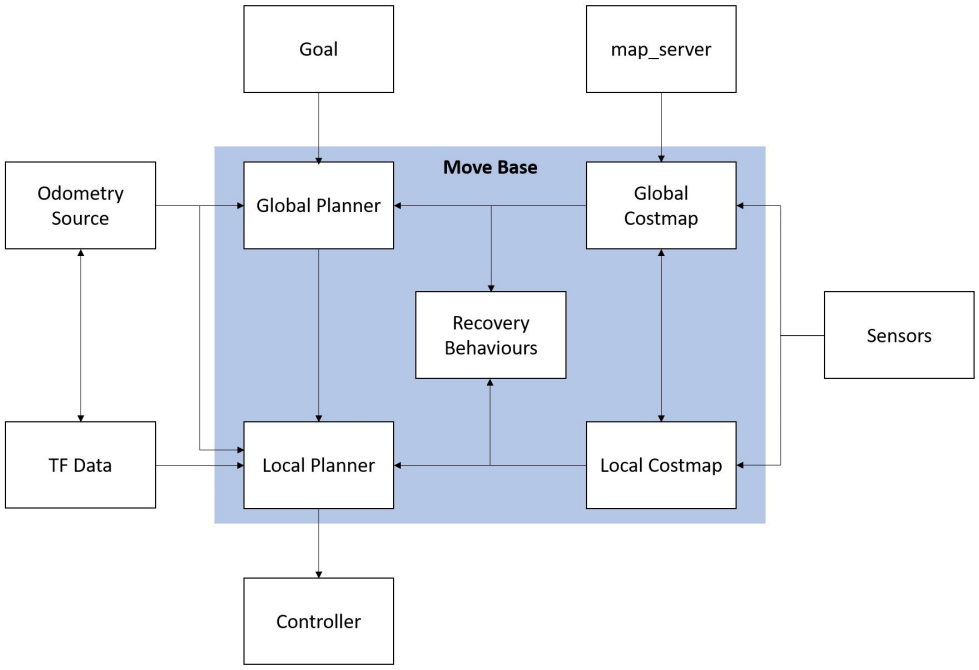


Figure 3.6: Conceptual blocks of Navigation Stack Setup

condition in which the robot is unable to reach the desired destination and ends up interrupting your navigation.

3.3.3 Local Planner: DWA

The Local Planner has the purpose to generate a trajectory in the nearby distance avoiding the obstacles. To achieve this result, a Local Costmap is created for including information about the navigation cost expressed as a number 0, if the cell is free, 1, if there is an obstacle inside the cell and -1, if the cell is unknown. So, given a path to follow and a cost map, the planner uses a controller to produce the velocity commands to send to robot. This package supports both holonomic and non-holonomic robots.

Each local planning algorithm must be adapted and configured according to the kinematic and dynamic specifications of the robot used and the navigation environment. Among the predefined algorithms, the *Dynamic Window Approach* local has been chosen, a collision avoidance strategy based solely on the robot dynamics. In detail, the approach consists of searching a two-dimensional space of translational and rotational velocities, reached by a robot along a trajectory until to a given goal point. The trajectory is approximate with a sequence of circular arcs, uniquely determined by the velocity vector (v, w), considered *admissible* only

if the robot is able to stop safely without collision with the obstacles. The search space is reduced to a *dynamic window*, namely to the only admissible velocities that can be reached in *short time interval*, taking into account the limited accelerations of the robot [38]. The DWA considers *time intervals sufficiently small* to calculate the trajectory, each one corresponding to a velocity pair, so that in each control interval $[t_i,t_{(i+1)}]$, the robot velocity is constant, and the velocity pair is not updated until the next operation period [38]. The combination of admissible translational (v_x, v_y) and rotational (w) velocities is called *velocity twist*, chosen by maximizing an *objective function* (3.1), expressed as follows:

$$G(v, w) = \sigma(\alpha \cdot angle(v, w) + \beta \cdot dist(v, w) + \gamma \cdot vel(v, w)) \tag{3.1}$$

where:

- $dist(v, w)$: indicated the distance to the closest obstacle on the trajectory.
- $angle(v, w)$: represents the current heading direction of the robot with respect the goal direction (Figure 3.7);
- $vel(v, w)$: defines the robot velocity used to calculate the progress of the robot along the corresponding trajectory;
- α,β,γ: are the weighting constants whose values are included between [0,1], chosen to optimize the navigation.
- σ: function used to smooth the weighted sum of the three components in the interval [0,1].

In detail, the velocity twist search space includes all linear and angular velocity values V_s allowed by the robot specifications. The objective function finds the admissible velocities vector V_a considering the dynamic window V_d. The admissible velocities V_a (3.2) and the dynamic window V_d (3.3) are computed as follows:

$$V_a = (v, w)|v \leq \sqrt{2 \cdot dist(v, w) \cdot \dot{v}_b} \wedge w \leq \sqrt{2 \cdot dist(v, w) \cdot \dot{w}_b} \tag{3.2}$$

$$V_d = (v, w)|v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge w \in [w_a - \dot{w} \cdot t, w_a + \dot{w} \cdot t] \tag{3.3}$$

where:

- $\dot{v}_b$ and $\dot{w}_b$: are the linear and angular accelerations respectively for breakage of the robot;
- v_a and w_a: represent the linear and angular current velocities;

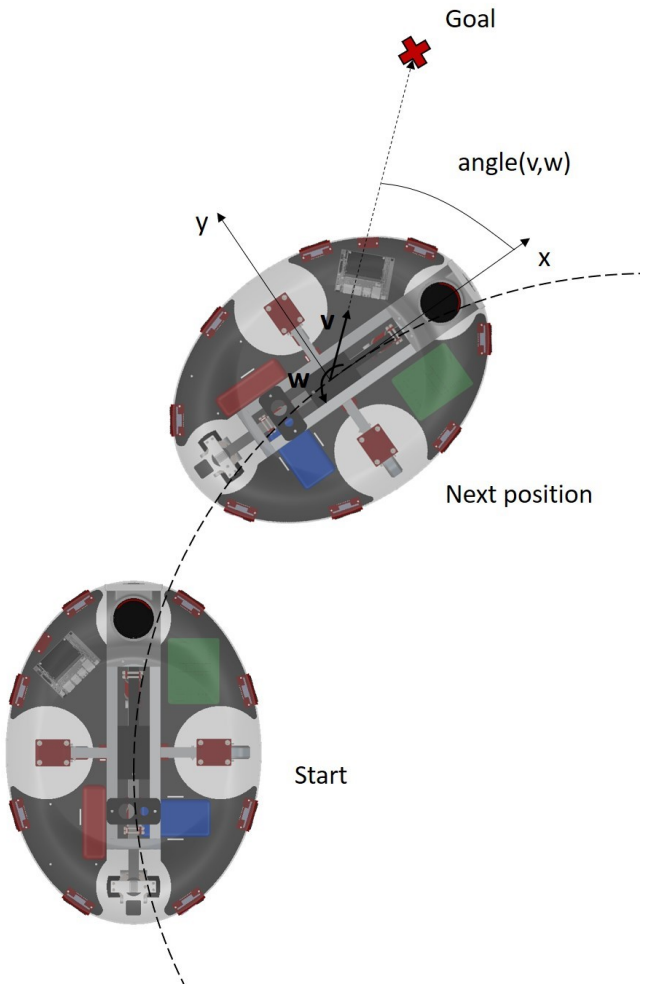


Figure 3.7: Angle(v,w) of the objective function

- t: is the time interval sufficiently small, to assume the robot velocity constant. This makes the optimization problem more feasible.

Finally, by considering the three velocity vectors described previously, namely the V_s, V_a and V_d, the set of the eligible velocities V_r (3.4) is obtained according to:

$$V_r = V_s \bigcap V_a \bigcap V_d \tag{3.4}$$

Figure 3.8 [39] illustrates the whole velocity search space, defined by the velocities V_s allowed by the robot specifications, by the admissible velocities V_a, by the velocity dynamic window V_d, centred in the current velocity position of the robot, and by the eligible velocities V_r, the area with the red outline in which the robot can travel without colliding with the obstacles.

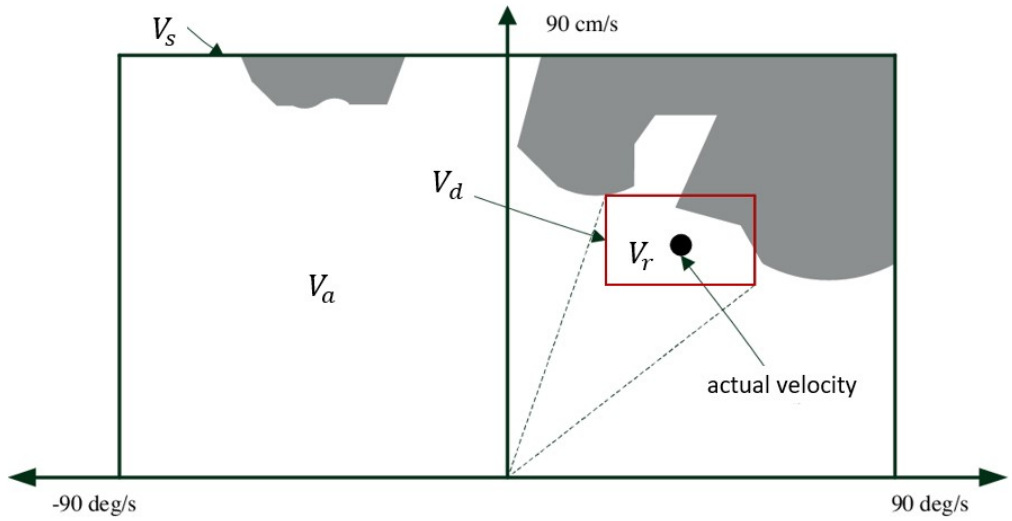


Figure 3.8: The whole Velocity Search Space for DWA

In conclusion, the basic idea of implementation of this algorithm is described as follows [40]:

1. Discretely sample in the robot's control space $(dx, dy, dtheta)$.

2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.

3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path and the speed. Discard illegal trajectories (those that collide with obstacles).

4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.

5. Repeat as necessary.

This algorithm can be summarized as follows [34]: the first step is to sample velocity pairs (v_x, v_y, w) in the velocity space within the dynamic window. The second step is basically obliterating velocities (i.e. kill off bad trajectories) that are not admissible. The third step is to evaluate the velocity pairs using the objective function, which outputs trajectory score. The fourth and fifth steps consist of taking the current best velocity option and recompute.

Parameters

In this section the ROS parameters of the Local Planner DWA are described according to several categories: *Forward Simulations*, *Robot Configuration*, *Goal Tolerance* and *Trajectory Scoring*. These parameters characterize and configure the behaviour of the DWA for a specific robot.

Forward Simulation Parameters

These parameters influence the simulations of the trajectories produced by the DWA during planning, and follow the description above:

- **controller-frequency** *(double, default: 20.0)*: parameter that specifies the planning frequency in hertz of the trajectory. A value of *4.0* is chosen, according to the performance computer.

- **sim-time** *(double, default: 1.7)*: parameter that sets the maximum time value in seconds for which the DWA elaborates and evaluates the trajectories during the navigation. By varying this value, the conditions made in the article [34] have been verified, namely longer this parameter value, heavier the computation load becomes. About the tuning of this parameter [34], if it is set to a lesser value equal to 2, the performance will be limited, since the planner has not sufficient time for processing of the trajectories. This causes excessively abrupt obstacle avoidance, and effects the quality of navigation. Instead, if the sim-time is set to a greater value equal to 5, the computational cost increases, but helps to better anticipate any avoidance of obstacles. For the thesis project, a value of *1.7* was set, although it did not follow the directives previously stated, as experiments demonstrated that setting a value greater than 1.7, caused problems in slowing down the planner, due to the performance of the on-board computer. Also, as all the trajectories are simple arcs, setting this parameter greater than 1.7, caused long arcs and the consequent rotational movement of the robot on itself.

- **sim-granularity** *(double, default: 0.025)*: parameter that defines the distance in meters of the points processed for the trajectories (step size). This means how frequently to control the points on the examined trajectory. To set this value too low compared to the default one, implied a high frequency which required more computation power [34], while a too high value risked having an inaccurate control of the points on the trajectory. After a few experiments, a value of *0.1* was selected, on the basis of the chosen controller frequency and on the feedback related to the built trajectory.

- **vx-samples** *(integer, default: 3)*: parameter that indicates the number of samples used to evaluate the trajectories with respect to the x direction. Since the number of samples depends on how much computation power the computer owns, *20* samples in x direction were picked.

- **vy-samples** *(integer, default: 10)*: parameter similar to the previous one, but evaluated with respect to y direction. By setting the *max-vel-y* value to zero, as described later, the velocity samples in y direction will not be needed, and hence this parameter was set to *0* value.

- **vth-samples** *(integer, default: 20)*: parameter that controls the number of rotational velocities samples. It is preferable to set this value higher than the *vx-samples* value, because turning is generally more complicated motion compared to the linear one. For this reason, *40* samples were collected.

Robot Configuration Parameters

The parameters influencing the robot kinematics are the velocity and acceleration limit with which the controller will have to operate, and are defined as follows:

- **max-vel-x** *(double, default: 0.55)*: represents the maximum x velocity for the robot in meters per second. As already stated, the robot has been developed with the idea to follow a human being to a velocity of *1-1.5* $\frac{m}{s}$. For safety, a maximum translational velocity along x lower than the human walk was set: *0.7*.

- **min-vel-x** *(double, default: 0.0)*: represents the minimum x velocity for the robot in meters per second. Experiments demonstrated that a value greater than or equal to 0.0, will not allow the robot to perform reverse movements, which instead will be granted with values smaller than 0. It has been observed that enabling robot reverse movements, has as consequence the fact that this will tend not to rotate in place, and to back off in case of stucking. In case of the robot used, a value lower than or equal to 0.0 caused a deadlock condition and difficulty in motions, and as a consequence, a value of *0.1* was chosen.

- **max-vel-y** *(double, default: 0.1)*: represents the maximum y velocity in meters per second. Only for a non-holonomic platform, such as the differential wheeled robot, this value should be set to 0, since as mentioned in *Chapter 1*, it cannot perform motions along the wheels' axis. However, the y velocity of Paquitop has been set to *0* as it had not good motion response along that axis. To move in the y direction, first the robot performed a rotational motion, and then a translational one along the x axis.

- **min-vel-y** *(double, default: -0.1)*: represents the minimum y velocity in meters per second. For setting this parameter, the same considerations of the previous one have been taken into account, and so the value of *0* was imposed.
- **max-rot-vel** *(double, default: 1.0)*: represents the maximum rotational velocity in radians per second with respect to the z axis. For setting this parameter value, the ROS Navigation Tuning Guide advised to control with a joystick the rotational motion of 360 degrees of the platform with a constant velocity, and to measure the respective time. A value of *3.0* was obtained.
- **min-rot-vel** *(double, default: 0.4)*: represents the minimum rotational velocity in radians for second with respect to the z axis. By applying the same considerations of the previous parameter, the value of *-3.0* has been imposed to have the same behavior in the opposite direction.
- **max-vel-trans** *(double, default: 0.55)*: represents the absolute value of the maximum translational velocity for the robot in meters per second, that in case of Paquitop is *0.7*.
- **min-vel-trans** *(double, default: 0.1)*: represents the absolute value of the minimum translational velocity for the robot in meters for second, that for Paquitop is *0.1*.
- **acc-lim-x** *(double, default: 2.5)*: represents the x acceleration limit of the robot in meters per second square. To set this parameter, the time with which the robot reached the maximum translational velocity imposed, was taken by odometry message in the ROS terminal. This procedure was carried out several times, and finally was made the average of the acceleration values obtained along x. However, for safety reasons, values lower than the default value are used, and the value of *2.0* was selected.
- **acc-lim-y** *(double, default: 2.5)*: represents the y acceleration limit of the robot in meters per second. As the y velocity was null, then this parameter must be set to *0*.
- **acc-lim-theta** *(double, default: 3.2)*: represents the rotational acceleration limit in radians for second square. As for the x acceleration parameter, the rotational maximum velocity of Paquitop from static was taken by the odometry message, and for safety reasons, a value lower than the obtained one was imposed: *3.0*.

Goal Tolerance Parameters

The following parameters define the dimensional tolerances inherent to the target points, namely the distance from the goal point considered successful when is reached by the robot:

- **xy-goal-tolerance** *(double, default: 0.10)*: represents the tolerance in meters for the controller in the x and y distance when achieving a goal, imposed to a value of *0.075*.
- **yaw-goal-tolerance** *(double, default: 0.05)*: represents the tolerance in radians for the controller in yaw/rotation when achieving its goal, imposed to a value of *0.5*.
- **latch-xy-goal-tolerance** *(bool, default: false)*: setting to true, if the robot reaches the goal xy location, it will start to rotate on itself, even if this movement takes it out of tolerance. For avoiding this behaviour, this parameter was set to *false*.

Trajectory Scoring Parameters

As mentioned earlier, the DWA maximizes the objective function to obtain the velocity pairs from a specific trajectory, which is chosen by cost function (3.5) expressed in the following form:

$$cost = path-distance-bias \cdot D + goal-distance-bias \cdot d + occdist-scal \cdot c \quad (3.5)$$

where:

- D = distance to path from the endpoint of the trajectory in meters;
- d = distance to local goal from the endpoint of the trajectory in meters;
- c = maximum obstacle cost along the trajectory in obstacle cost (0-254).

The robot will use the trajectory with the lowest total cost, so as to take the shortest path but with the lowest possible risk of collision. The parameters that affect the choice of trajectory are described above:

- **path-distance-bias** *(double, default: 32.0)*: indicates the weight for how much the local planner should stay close to the global path. If this value will be higher than the default one, the local planner will prefer trajectories of the global path to the detriment of that planned locally. Experiments demonstrated that the *default value* worked well in the Paquitop's simulation.

- **goal-distance-bias** *(double, default: 24.0)*: indicates the weight for which the robot will tend to follow the local path. This parameter works inversely to the previous one, means that by increasing this value the planner will tend to follow the local trajectory more to the detriment of the global one. As for the previous parameter, there is no rule to set its value, and experiments demonstrated that the value of *10.0* resulted to be compliant with the value set for the other parameters concerning the trajectory scoring.

- **occdist-scale** *(double, default: 0.01)*: indicates the weight for which the robot will avoid a specific obstacle. A much higher value than the default one, will cause an indecisive robot that stucks in place, in the sense that the obstacle will be considered too large in dimensions compared to its real ones, preventing the robot from planning new trajectories to overcome the nearby obstacle. For this reason a slightly higher value to the default one has been chosen, namely *0.02*.

All the parameters described below, will be tuned as their respective *default value*, as worked well in the simulations:

- **forward-point-distance** *(double, default: 0.325)*: represents the distance in meters from the center point of the robot to place in front of it an additional scoring point. This parameter is crucial for robots whose center of rotation is not in their middle, as is used to align the robot on the planned path. If it is 0, this parameter will not affect the robot trajectory.

- **stop-time-buffer** *(double, default: 0.2)*: indicates the amount of time that the robot must stop before a collision, in order for a trajectory to be considered valid in seconds.

- **scaling-speed** *(double, default: 0.25)*: represents the absolute value of the velocity at which to start scaling the robot's footprint, in meters per second. This parameter ensures greater safety during the robot navigation at high speeds, increasing its footprint size, and making it place as far away as possible from obstacles and walls.

- **max-scaling-factor** *(double, default: 0.2)*: is the maximum factor to scale the robot's footprint by, when the velocity value set by the previous parameter is reached.

3.3.4 Global Planner: A* - Dijkstra Algorithms

The Global Planner is an algorithm whose purpose is to compute the optimal trajectory chosen based on the obstacles and cost information contained in the

global costmap, the information about the localization system and the goal pose. From these, it creates a high-level plan for the robot to follow, to reach the target location. The global planner is able to create a series of *way-points* that the robot has to execute. The two most algorithms used in the robotic field for path planning, are Dijkstra and A*, which will be described below.

The *A* search algorithm* is the classic method for computing optimal paths for holonomic robots. It is a heuristic search algorithm which consists of finding the shortest path from the start node to goal node with the lowest cost among all possible paths. Starting from the start node, the algorithm considers all its neighbouring nodes, and choices the one with the least cost. For this reason, the algorithm needs to traverse around all nodes for selecting the right one until the goal node is reached, causing a long calculation time.

This algorithm represents the C-space, described in *Chapter 1*, as a *graph*, which consists of a collection of *nodes N* and *edges* ϵ, where each edge connects two nodes. A node typically represents a configuration or state, while an edge between nodes n_1 and n_2 indicates the ability to move from n_1 and n_2 without hitting the obstacles. This algorithm is formulated in terms of *weighted graph*, in which each edge has a positive cost associated. A tree of paths is created from the starting node (root), each of which consists of different nodes (leaf) with the respective cost. According to the latter, the algorithm choices how to extend its path at each iteration, considering only the lowest costs to reach the target node.

The heart of this algorithm is the function *f(n)* (3.6), which is the sum of the function *g(n)* and *h(n)*, expressed as follows:

$$f(n) = g(n) + h(n) \tag{3.6}$$

The term *g(n)* is the cost of the path from the start node to the current node n; *h(n)* is the heuristic function that estimates the cost of the path from the node n to the goal node through the selected sequence of nodes and *f(n)* is the heuristic function of the A* algorithm, which represents the shortest possible distance between two points. The algorithm selects the smallest *f(n)* value, and to make this, it involves maintaining two lists named open and closed. The open set contains the nodes that have to be explored, while the closed list stores the nodes that have already been visited (expanded). At the beginning, the open list only contains the starting node, while the closed one is empty. In each iteration, the algorithm removes the node from the open list that has the smallest *f-value*, expands this node inserting it in the closed list, and then puts its neighbouring nodes in the open list. The algorithm stops when find the goal node from the open list, or if there are no more nodes available in the open list.

In the case of intersection of the trajectory with an obstacle, the algorithm litters the path taken, and selects the path with the lowest cost between all possible paths listed in a priority queue.

If $h(n) \leq h(n)^*$ for all the nodes n in the graph, the algorithm is admissible and it is named optimal. The function $h(n)^*$ is the true cost to reach the goal state from n, while $h(n)$ is an estimate of $h(n)^*$.

Figure 3.9 [41] represents an illustration of A* search algorithm. The empty circles represent the nodes in the open set, i.e. those that remain to be explored, and the filled ones are in the closed set. The colour on each closed node indicates the distance from the goal: the red points indicate the points close to the start node, while the green points one close to the target node. The A* algorithm traces a straight line in the direction of the goal, but when encountering the obstacle, it explores alternative routes inside the open set.

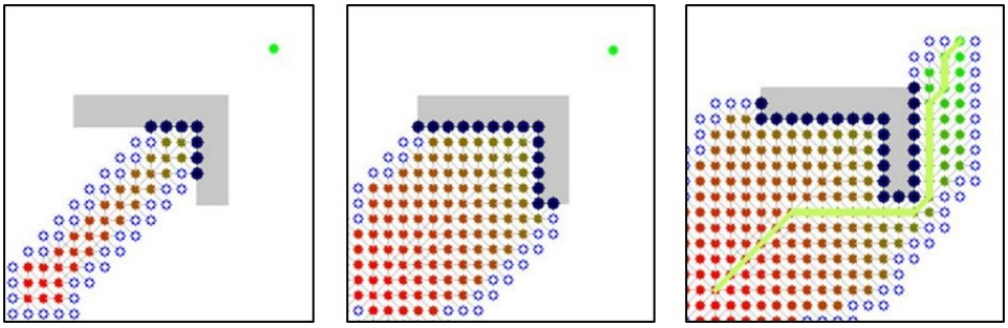


Figure 3.9: A* Search Algorithm

With the *Dijkstra algorithm*, the concept of Breadth-first search is defined. First, the algorithm stores all the *unvisited nodes* creating a list named unvisited set, to which assigns the zero value to the initial node distance, while infinity to all the other nodes. So all nodes one edge away from the start node, are considered first. Then the initial node is set as current and all the distance of its neighbors are considered, choosing only the smallest one. Once all the neighbors of the current node are visited, this will be deleted by the unvisited list. If the target node has been reached or if there is no connection between the start node and the unvisited nodes, the algorithm stops. Otherwise the unvisited node with the smallest distance is selected and considered as current one and so the algorithm keeps running.

Figure 3.10 [42] illustrates this algorithm. Open nodes represent the unvisited nodes, while filled nodes are the visited ones, with colour that represents the distance. The nodes are explored uniformly in all different directions.

Unlike the Dijkstra algorithm, A* is a best-first search algorithm, that means that the nodes of a graph are explored in the direction of the most promising node according to a specific rule. The A* search algorithm is just like the Dijkstra's algorithm, with the only difference that A* uses a heuristic function to find a better trajectory, according to the nodes priority, while Dijkstra explores all possible ways. Therefore, it runs more slowly than A* owing to the lack of the heuristic

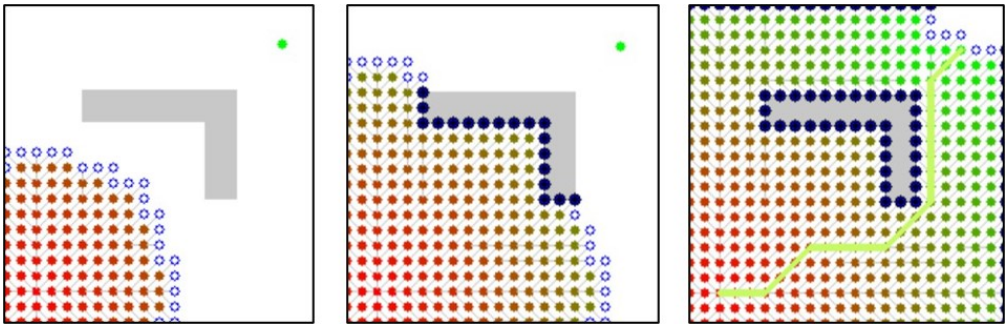


Figure 3.10: Dijkstra Algorithm

function. For the considerations listed above, the algorithm A* is adopted for the development of the Paquitop robot.

Parameters

In this section the ROS parameters of Global Planner A* adopted are described:

- **allow-unknown** *(bool, default: true)*: parameter that specifies wheter to allow the planner to create plans even in unknown space. It enables the search algorithm to plan on unknown cells, namely not yet visited, marked with the value of -1 as previously mentioned. For this thesis project, a *true* value was selected.
- **use-dijkstra** *(bool, default: true)*: indicates which algorithm is used between A* and Dijkstra. As the algorithm A* was adopted, a *false* value was set.
- **use-grid-path** *(bool, default: false)*: creates a path following the grid boundaries, otherwise a gradient descent method is used [43]. For the computation time reason, a *true* value was adopted.
- **publish-potential** *(bool, default: true)*: publishes potential costmap. For memory reasons, a *false* value was set.
- **outline-map** *(bool, default: true)*: outlines the global costmap with lethal obstacles. For the usage of a no static (rolling window) global costmap, this needs to be set to false. By adopting a dynamic map, a *false* value has been chosen.

The quality of the planned global path is determined by the parameters *cost-factor*, *neutral-cost* and *lethal-cost*, described below:

- **lethal-cost** *(int, default: 253)*: indicates the presence of the obstacle in the cell, and so safe collision with the obstacle by the mobile platform. The

planner will be unable to plan a trajectory passing through that cell. Setting it to a low value with respect to the default one, may result in failure to produce a path, even when a feasible path is obvious, and so after experiments, the *default value* was preferred to choose.

- **neutral-cost** *(int, default: 50)*: unlike the previous parameter, this indicates the absence of an obstacle. As a much higher value than the default one could cause the same effect of *lethal-cost* parameter when a much lower value the default one is set, a value of *66* has been selected.
- **cost-factor** *(double, default: 3.0)*: factor to multiply each cost from costmap by. Experiments demonstrate that setting this parameter to too low or too high with respect to the real one, lowers the quality of the paths [34], so a value of *1.0* was set in such a way as to have the actual weight value relating to the obstacle.

Follow Waypoints

The *Follow Waypoints* algorithm allows the robot to have a prior knowledge of the path information, to reach a target point with the minimum number of errors. To fulfill a requested task, it provides a path tracking using a set of ordered waypoints. To avoid of stopping the robot at each waypoint, and to make its smoothing motion, a *distance threshold parameter* enables an imaginary circle defined by a radius r, so that when the robot is within it, the control system will update the next waypoint, and it will drive the robot toward it. For this thesis project, a value of *0.75* was set. Figure 3.11 shows an example of planned path, where the robot starts from a point P_1 and arrives to goal position P_n, passing through n waypoints.

The Follow Waypoints algorithm is programmed in Python language, while the list of poses to reach are contained into an Excel file, where each goal point is written in the form of quaternions.

Figure 3.12 shows an example of the planned points that the Paquitop has to reach, marked with red arrows, which define a blue performed trajectory.

3.3.5 Costmap

A *costmap* is a weight matrix representing the environment with information about obstacles by means of the *costmap-2d* package. The costmaps used are the following:

- **Global Costmap**: stores information globally for long-term route planning, and its map is used by the global planner. This map will include the chromatic scale of the values assigned to each obstacle present in the environment (Figure 3.13).

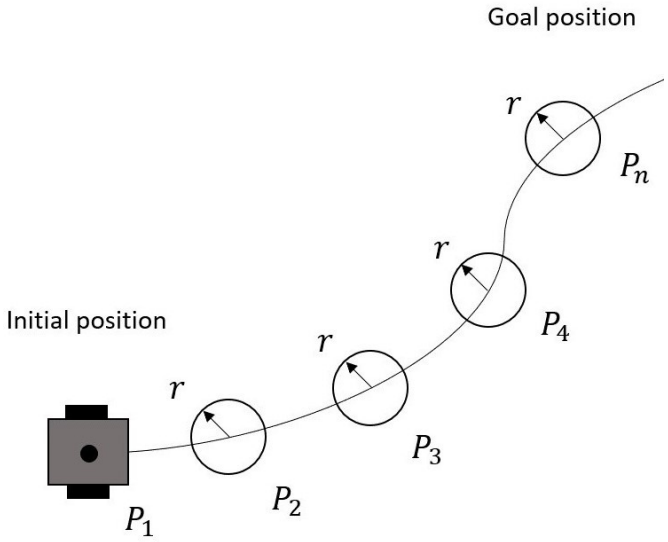


Figure 3.11: Example of a planned path through waypoints

- **Local Costmap**: stores information locally for short-term obstacle avoidance, and its map is used by the local planner. It has smaller dimensions than the global map, and it is integral with the robot. This is because the obstacles encountered by the robot during navigation will be stored inside it, so it is necessary that this is displayed taking the robot as a reference (Figure 3.14).

Both costmaps in ROS use the concept of overlapping layers, which are implemented as a plugin on RViz. Among the predefined ones:

- **Static map layer**, represents the static map of the environment provided by the *map-server* (Figure 3.15). This layer is used exclusively by the global costmap and is indicated with the name of *"costmap-2d::StaticLayer"*.

- **Obstacle layer**, represents the obstacles identified by the sensors, and is used by both the local and global costmaps. The obstacles are represented on a two-dimensional map indicated with the name of *"costmap-2d::ObstacleLayer"*. Optionally, obstacles can also be configured to be displayed in 3D. There are numerous ROS packages that allow the three-dimensional display of the navigation space, and the default on the ROS Navigation Stack is the *voxel-grid* package.

- **Inflation layer**, responsible for inflating obstacles, creating a collision-free region. This layer is used by both the local and global costmaps, and is indicated with the name of *"costmap-2d::InflationLayer"*.

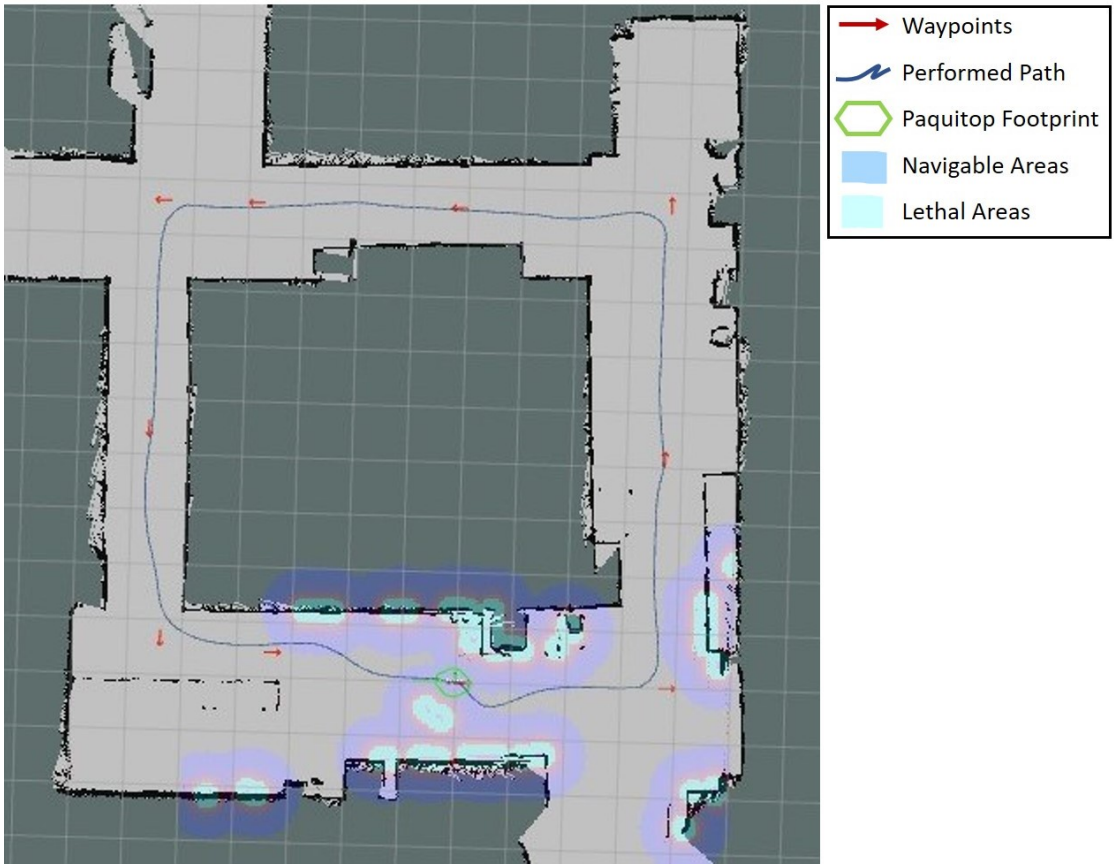


Figure 3.12: Follow Waypoints algorithm. The cost-map contains the points planned by this algorithm, the path performed by the robot footprint and the inflation radius of the local and global map, represented with light and dark blue strokes respectively, the values of which are 0.1 m and 0.8 m.

The costmap uses sensor data and information from the static map to build a 2D or 3D grid map, where each cell has a cost: higher costs indicate a smaller distance between the robot and an obstacle. The details on the allocation of costs are explained in the Inflation layer.

Inflation layer

The Inflation layer consists of cells with a cost going from 0 to 255, which defines the severity of an obstacles and allows to configure a margin of security with respect to obstacles. The cost scale is built on the basis of the robot footprint, obtaining 5 specific areas:

- **Freespace cost** *(value: 0)*: represents a passable cell by the robot with

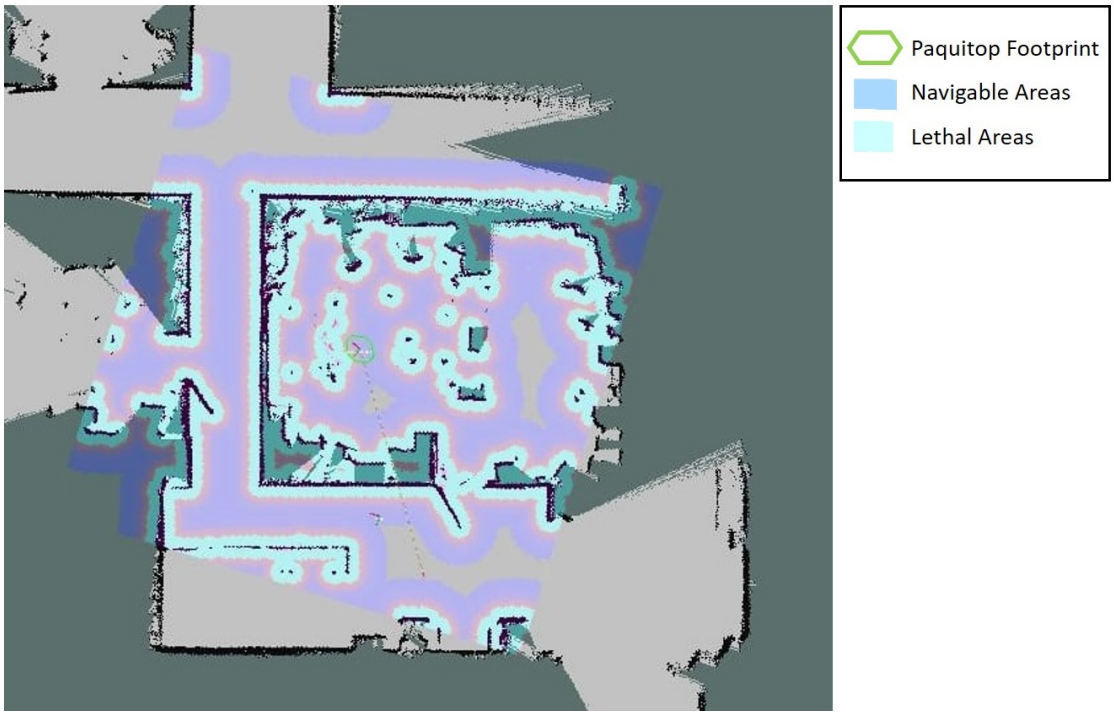


Figure 3.13: Global Costmap in RViz environment. The cost-map points out the maximum range of the Lidar, defined by a square. The light and dark blue strokes define the inflation radius of the local and global map respectively, the values of which are 0.1 m and 0.8 m.

assigned zero of collision;

- **Possibly circumscribed cost** *(value: from 1 to 127)*: indicates the cost range for which the robot can avoid the obstacle. The possible collision will be determined from the orientation of this during the circumscription of the cell. It should be noted that a cell with a value included in this range does not mean that is an obstacle, because it is possible to have such values in free map areas, where it is preferred that the robot does not plan or that it remains at a certain safety distance.

- **Inscribed cost** *(value: from 128 to 253)*: means that a cell is less than the robot's inscribed radius away from an actual obstacle. This represents the cost range in which it is likely that if the center of the robot is inside the cell, it will collide with the obstacle.

- **Lethal cost** *(value: 254)*: indicates the presence of the obstacle in the cell and so safe collision with the obstacle by the mobile platform. The planner

Figure 3.14: Local Costmap in RViz environment. The cost-map outlines the radius inflation of 0.1 m with light blue strokes and the robot footprint.

will be unable to plan a trajectory passing through that cell.

- **Unknown cost** *(value: 255)*: assigned to a map cell when information is not enough to evaluate the occupation of the cell.

In conclusion, the inflation indicates the process of propagating cost values out from occupied cells, that decrease with distance as shown in Figure 3.16 [44]:

The parameter used to modify the inflation of the maps is the following:

- **inflation-radius** *(double, default: 0.55)*: indicates the distance expressed in meters from the center of the robot to the obstacle. For the global costmap, the value of *0.8* was set, to ensure that the robot is in the middle of the path, while for the local costmap a value of *0.1*, to keep the robot neither too close nor too far from the obstacle.

- **footprint** *(list, default: [43])*: represents the outline of the robot base coinciding with the origin of the *base-footprint* frame. In Figure 3.16, the inscribed and circumscribed circles of the mobile base are represented, useful to inflate obstacles in relation to the size of the robot. Specifically, they refer to the footprint of the platform. In order to guarantee a certain safety margin, the

Figure 3.15: Static Map Layer in RViz environment

real perimeter of the robot can be increased, scaling its coordinates in the CAD software as Solidworks. The footprint can be described by two shapes: *circular* and *polygonal*. The first is expressed as the radius value of the circumference, which will have as its center the reference of the robot frame. The second is defined as a two-dimensional array containing the points of the vertices expressed with respect to the frame reference of the robot, such as:

$$[[x_0, y_0], [x_1, y_1], [x_3, y_3], [x_4, y_4], ..]$$

Both clockwise and counter-clockwise orderings of points are supported. Paquitop has a polygonal shape, and both local and global costmaps describe its footprint from the point P1 to one P8 (Figure 3.17), as follows:

$$[[0.3, 0.0725], [0.0725, 0.24], [-0.0725, 0.24], [-0.3, 0.0725],$$
$$[-0.3, -0.0725], [-0.0725, -0.24], [0.0725, -0.24], [0.3, -0.0725]]$$

Obstacles layer

The Obstacles layer is responsible for two main operations that can be performed on the grid map: *marking* and *clearing*. The first consists of inserting obstacle

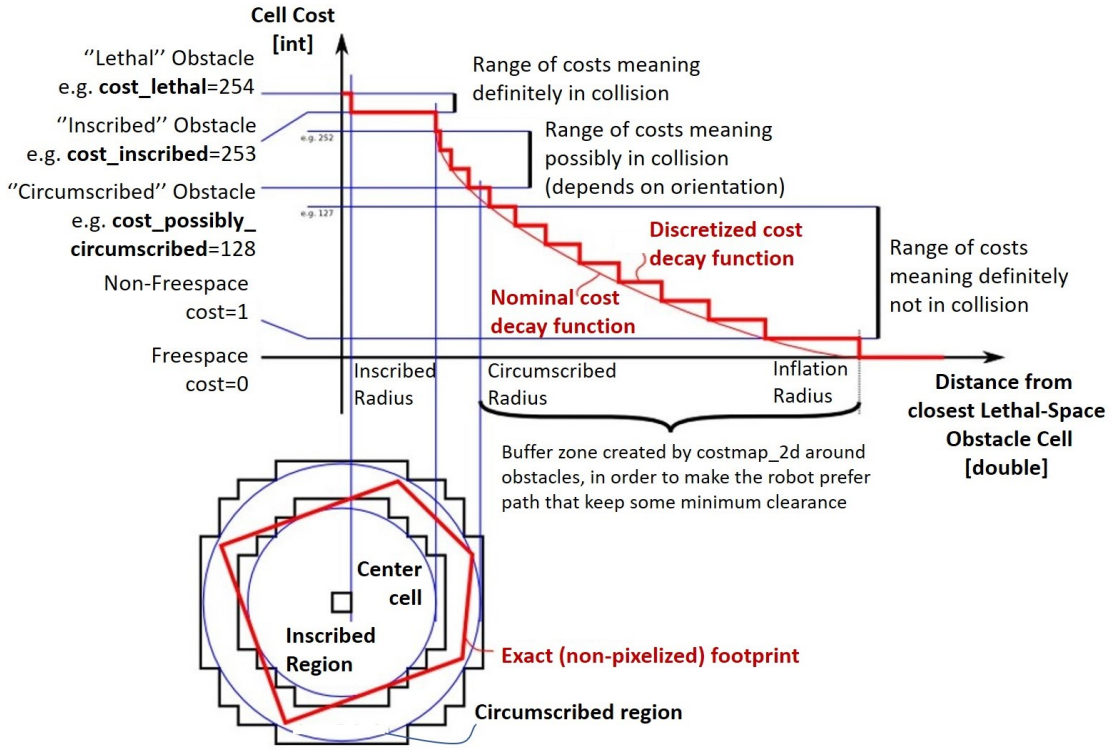


Figure 3.16: Inflation Process Graph

information into the costmap, while the second removes previously stored obstacles no longer present in that position of the map. These two operations are described by the following two parameters, and for this thesis project, assume the same value in both costmaps:

- **obstacle-range** *(double, default: 2.5)*: the default maximum distance from the robot at which an obstacle will be inserted into the costmap in meters. This parameter depends on the type of sensor and configuration implemented, and as the maximum distance range of Lidar adopted is 6 m, a lower value than the actual one has been chosen to reduce the calculation times, namely $4.0\ m$.

- **raytrace-range** *(double, default: 3.0)*: represents the default range in meters at which to raytrace out obstacles from the costmap using sensor data. In order to guarantee a clean of the costmap, it is convenient to configure this parameter to a value slightly higher than the previous one, and in this case $4.5\ m$.

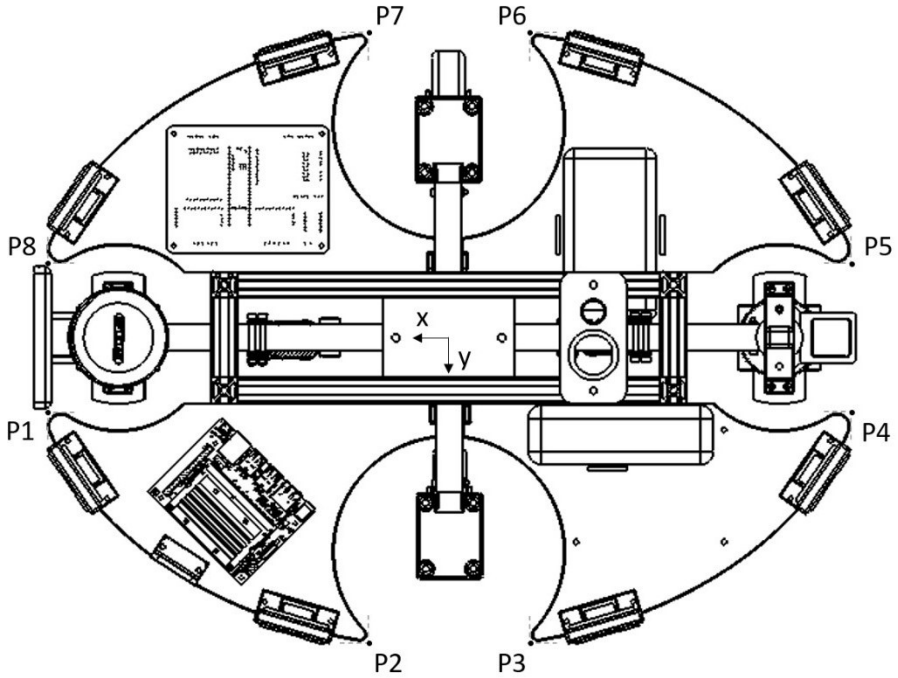


Figure 3.17: Paquitop Footprint Description in Costmaps

Static map layer

The Static map layer incorporates the following parameters:

- **map-topic** *(string, default: "map")*: indicates the topic where the costmap subscribes to for the static map: */map*.
- **subscribe-to-updates** *(bool, default: false)*: in addition to map-topic, also subscribe to map-topic plus "-updates". In this case *true* was set.
- **rolling-window** *(bool, default: false)*: fixes the origin of the map attached to the robot. If the *static-map* parameter is set to true, this parameter must be set to false. Since the *static-map* parameters has been set to false for both maps, this parameter was set to *true* since a dynamic map was used.
- **resolution** *(double, default: 0.05)*: defines the resolution of the map in meters per cell. For both costmaps a value of *0.1* was adopted.
- **static-map** *(string, default: true)*: determines whether or not the costmap is given by *map-server*, so if an initial map does not exist, this parameter is set to false. In this case, the map is built and updated by sensor constantly

through the SLAM algorithm, so for both the local and the global map it was set to *false*.

It has been observed that enabling the *rolling-window* and *resolution* parameters, avoided the risk of having costmap areas not updated during the navigation. Otherwise, it was updated only the portion of the map where changes were detected.

Through the simulations, it was observed that if an excessively small map was used, with the width and the height dimensions much lower than 50 meters, the local planner did not have sufficient planning space, and considered the obstacles overdue, causing abrupt maneuvers to avoid them, and subsequent stall condition. Instead, values higher than 50 of the width and the height, caused an excessive freedom of planning, and subsequent increase of the computation time. In ROS environment, the width and the height parameters are described as follows:

- **width** *(int, default: 10)*: represents the width of the map in meters.
- **height** *(int, default: 10)*: represents the height of the map in meters.

For the considerations listed above, a value of $50.0\ m$ has been set for both width and height, developing a grid of data with dimensions $50{\cdot}50\ m^2$.

Other Parameters

Rate Parameters

Sensor data, marking and clearing operations are performed and updated in the costmap at the rate specified by *update-frequency* parameter, and published by *publish-frequency* parameter, described below:

- **update-frequency** *(double, default: 5.0)*: indicates the frequency in hertz for which the map is updated. Increasing this value too much with respect to that of the controller frequency, risks increasing the computational calculation required, while decreasing it risks having a map not updated during the navigation of the mobile platform. To avoid increasing the computation time and occupying a lot of memory, a lower frequency value than the controller one was set: $2.0\ Hz$.
- **publish-frequency** *(double, default: 0.0)*: indicates the publication frequency value in hertz. It is recommended to use a value lower than the previous parameter, otherwise data not yet processed will be published. For local costmap a value of $1.0\ Hz$ was set, while for global one a value of $0.0\ Hz$ was selected in order not to influence the calculation times.

Coordinate Frame and TF Parameters

- **transform-tolerance** *(double, default: 0.2)*: sets the tolerance value on the transformations of reference systems over time. One of the most frequent errors during navigation is the propagation delays of messages about transformations. For both costmaps a value of *0.3* has been chosen.

- **global-frame** *(string, default: "map")*: with such parameter is configured the global reference used for navigation. This must coincide with the reference of the *map*.

- **robot-base-frame** *(double, default: "base-link")*: with this parameter the reference of the robot frame is configured. Although conventionally the name of *base-link* is assigned for this reference, the *base-footprint* frame was selected, as *hector-slam* package adopts a base frame placed on the same plane of the map one. This guaranteed no delay propagation about transformation from one frame to another.

3.3.6 TF

ROS provides a system called *tf* to introduce the coordinate transformations between frames assigned to each component (link) of the system. Each transform defines the translations and the rotations required to pass from a frame to other, without performing the calculations with trigonometry. This allows the description of the geometrical relation between all the part of the robot, where every frame is defined by its relationship to other frame, building a *tree structure*. Each transformation is described in the following way:

- **x/y/z**: cartesian triad indicating the position in meters of the reference system;

- **roll/pitch/yaw (rpy)**: terminology used to describe the orientation in radians of each reference system. The term *rpy* derives from the initials of the English terms corresponding to the different rotations (Figure 3.18): *roll*, rotation of an angle ψ around the x axis; *pitch*, rotation of an angle θ around the y axis and *yaw*, rotation of an angle ϕ around the z axis.

Before listing all the transformations, all the reference systems used for Paquitop (Figure 3.19) in the ROS environment are described:

- **map**: identifies the name of the reference system of the map. Its origin is arbitrarily chosen in the navigation environment and is fixed.

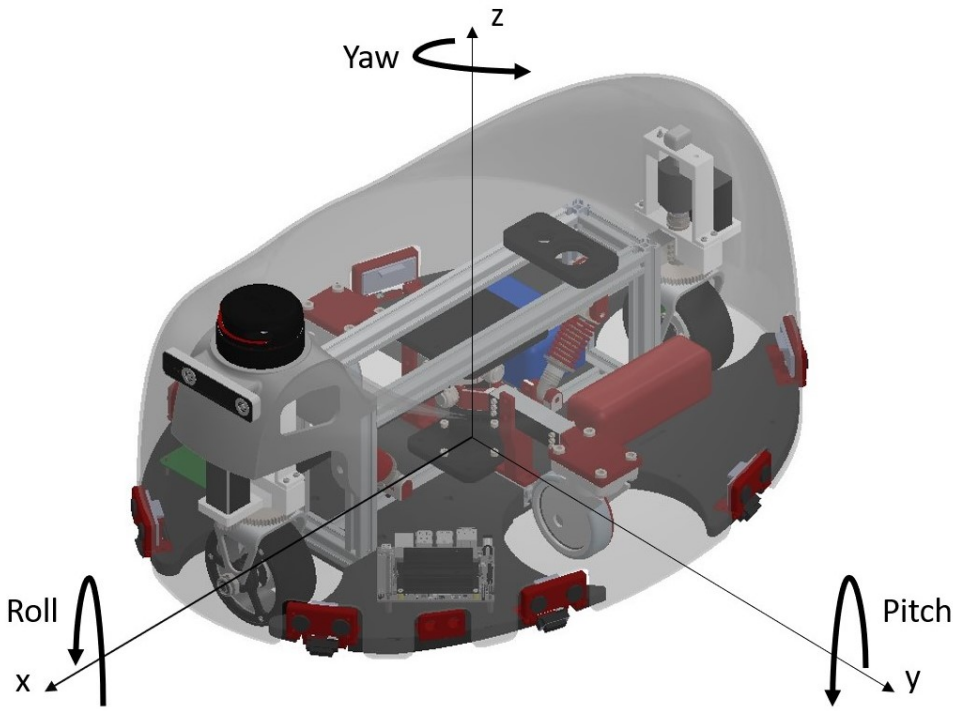


Figure 3.18: Roll/Pitch/Yaw of Paquitop

- **odom**: identifies the name of the reference system in which the robot is initialized.
- **base-footprint**: identifies the name of the reference system associated to the chassis' footprint, which has the origin in projection of the origin of the robot's mass center, but placed on the footprint. This reference system is not fixed, but it is integral with the robot.
- **base-link**: identifies the name of the reference system positioned in the center of gravity of the robot, representing the Paquitop's chassis.
- **laser-link**: identifies the name of the reference system of the laser's center of gravity. This reference system remains fixed with respect to the *base-link*.
- **camera-link**: identifies the name of the reference system of the camera's mass center.

Coordinate transformation can be static and dynamic: unlike the latter, the former does not change over time. All the transformations are listed below:

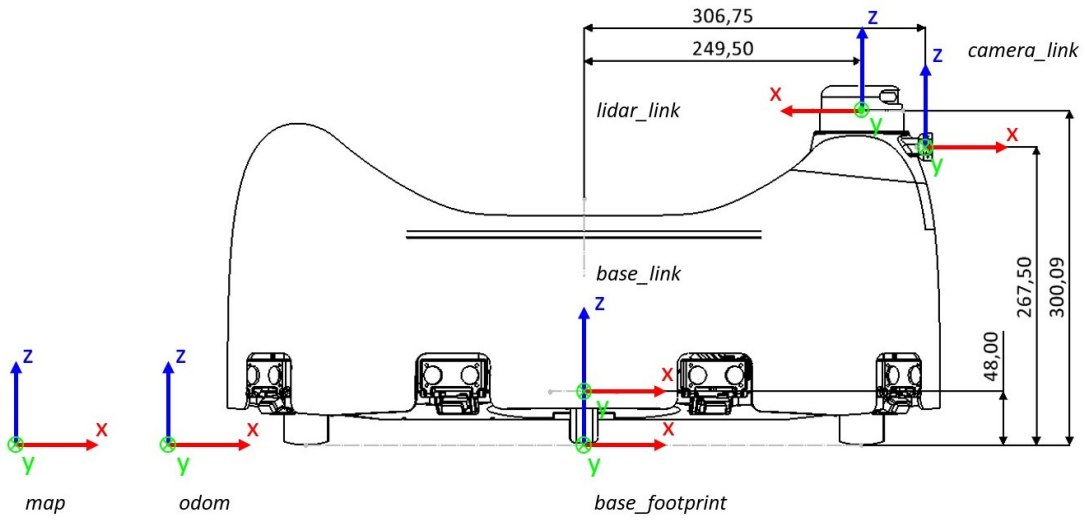


Figure 3.19: Paquitop's Reference Systems in ROS environment and their coordinates expressed in millimeters with respect to the footprint frame.

- **map to odom**: dynamic transformation, defined by the *hector-slam* package that takes care of localization and mapping.
- **odom to base-footprint**: dynamic transformation, defined by the ROS *pose-ekf* (Extended Kalman Filter) package, which estimates the position of the robot *(base-footprint)* in the environment with respect to the initialization frame *(odom)*.
- **base-footprint to base-link**: static transformation, both coordinate frames are integral to each other.
- **base-link to laser**: static transformation that provides the position and orientation of the laser with respect to the *base-link* frame.
- **base-footprint to camera**: static transformation which sets the pose of the camera to that of the robot.

As previously mentioned, the description of the frames within the *tf* package, occurs through a tree structure, that for Paquitop is the following (Figure 3.20):

3.3.7 Hector SLAM

Hector SLAM is a mapping algorithm which only uses laser scan information to extract the map of the environment. In fact the term SLAM, which stands for

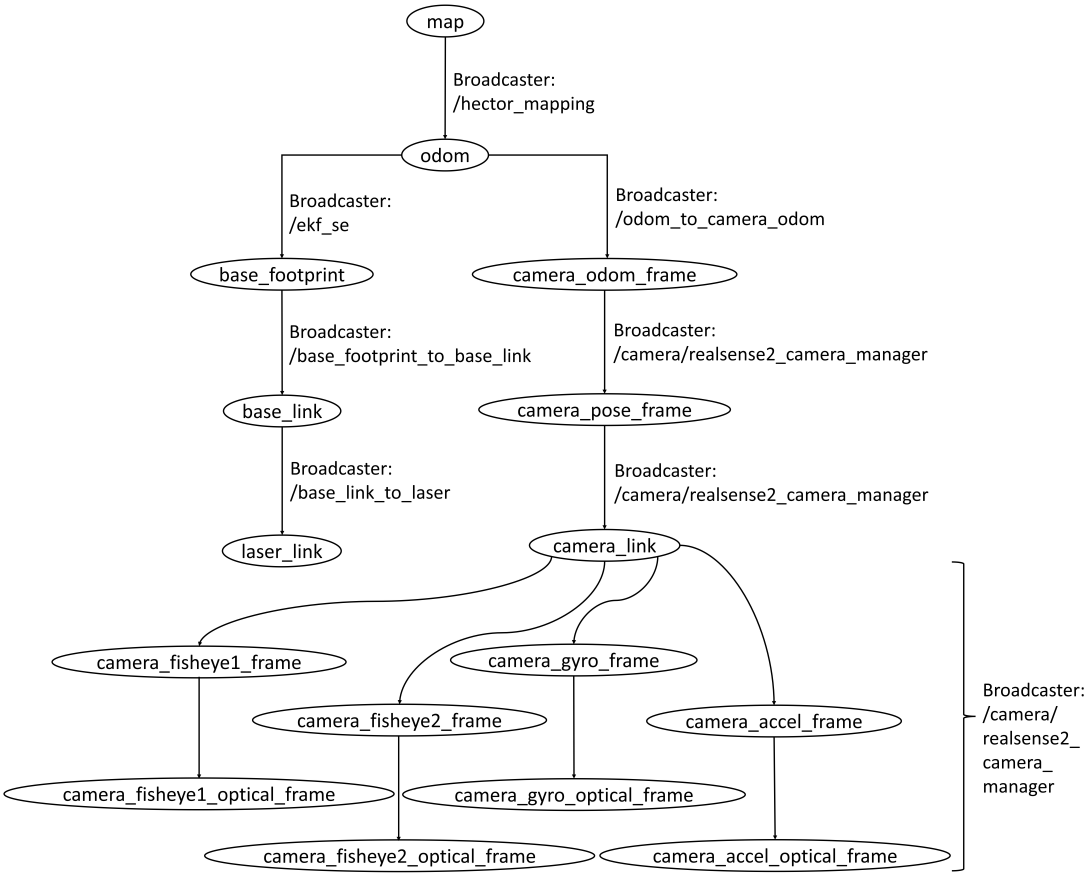


Figure 3.20: The Tree Structure of Paquitop

Simultaneous Localization and Mapping, indicates just an algorithm solving the problem of constructing a map of an unknown environment, while simultaneously keeping track of its position within it. This is an approach that can be used without odometry but only using the Lidar sensor, which is put on the platform to operate in real-world environments. The package name is *hector-slam* and its main node are:

- **hector-mapping**: SLAM node, which builds the environment map, estimating the 2D robot pose;
- **hector-geotiff**: saves the map and the robot trajectory of the performed path;
- **hector-trajectory-server**: keeps track of performed trajectory.

After providing a general idea of how the algorithm works, all the parameters

used during the experimental tests are listed.

Parameters

- **map-resolution** *(double, default: 0.025)*: as the name suggests, it indicates the map resolution in meters. In details, it is the length of a grid cell edge. For this project, a value of *0.02* was selected.
- **map-size** *(int, default: 1024)*: represents the number of cells for axis of the map, calculated as *map-size·map-size*. As the length of each cell edge is 0.02, and the map is a square with side of size 50 m, *2500* represents the number of cell for axis.
- **map-start-x** *(double, default: 0.5)*: indicates the origin coordinates of the map about x axis from 0.0 to 1.0. To have a map centered with respect to the global origin, the value *0.5* was imposed.
- **map-start-y** *(double, default: 0.5)*: indicates the origin coordinate of the map about y axis. As for the previous value, also this the value was set to *0.5*.
- **map-update-distance-thresh** *(double, default: 0.4)*: defines the value in addition to which is performed the map updates in meters. In this case the platform had to travel 0.0 m for performing map updated, causing its constant updating.
- **map-update-angle-thresh** *(double, default: 0.9)*: defines the value in addition to which is performed the map updates in radians. As the previous parameter, this was set to *0.0 rad*.
- **map-pub-period** *(double, default: 2.0)*: represents the period in seconds of the map publish, performed to *0.2 s*.
- **map-multi-res-levels** *(int, default: 3)*: indicates the number of map multi-resolution grid levels, which help the algorithm to compute the best position estimation. The default value *3* was adopted.
- **update-factor-free** *(double, default: 0.4)*: represents a range between [0.0, 1.0] for updating free cells. The value of 0.5 indicates no change, and a value of *0.4* was selected to update newly free cells more slowly (less reactive), and to have more time for path calculation.
- **update-factor-occupied** (double, default: 0.9): similar to the previous parameter, but refers to updating occupied cells. This parameter is in the range [0.0, 1.0], and a value of 0.5 indicates no change. A value of *0.9* was

imposed, in order to instantly update (more reactive) the obstacles on the map, and to avoid computing a possible path on that cell.

- **laser-min-dist** *(double, default: 0.4)*: indicates the minimum distance in meters from which the laser scans the environment, thus all points less than this distance are ignored. From datasheet, the minimum distance range of the adopted Lidar is 0.15, but to avoid seeing itself as an obstacle, a slightly greater value than the distance between the Lidar and the rear part of the platform was imposed, namely *0.6 m*.

- **laser-max-dist** *(double, default: 30.0)*: analogous to the previous parameter, but indicating the maximum distance in meters within which the laser scans, thus all points beyond this value are ignored. From datasheet, the maximum distance range of the adopted Lidar is *6.0 m*.

- **laser-z-min-value** *(double, default: -1.0)*: represents the minimum height in meters from which the laser scans the environment. All points lower than this value are ignored. This parameter and the next are needed for lasers 3D, and hence not important for the adopted Lidar.

- **laser-z-max-value** *(double, default: 1.0)*: similar to the previous parameter, but representing the maximum height in meters from which the laser scans the environment. All points upper than this value are ignored.

- **pub-map-odom-transform** *(bool, default: true)*: indicates whether or not the transformation from map to odom is published by the system. Hector SLAM computes this transformations, and to publish it, a *true* value was imposed.

- **output-timing** *(bool, default: false)*: output timing information for processing of every laser scan. For the computation time and occupying memory reasons, a *false* value was set.

- **scan-subscriber-queue-size** *(int, default: 5)*: represents the queue size of scan subscriber. If in hector-mapping, logfiles (computer-generated data file) have a speed higher than real time one, this parameter should be set to very high values than the default ones. In this case, a value of *10* was chosen.

- **pub-map-scanmatch-transform** *(bool, default: true)*: indicates whether or not the transformation from *scanmatcher* to *map* is published by the system. The *scanmatcher* frame indicates the lidar pose frame with respect to *map* frame, and as the laser frame was already defined as *laser-link*, a *false* value was selected.

- **tf-map-scanmatch-transform-frame-name** *(string, default: scanmatcher-frame)*: defines the frame name of the *scanmatcher* when the transformation is published. For this project, it was not needed as the transformation from *scanmatcher* to *map* was not published.

3.3.8 EKF

The EKF algorithm estimates the 3D pose of a robot through the implementation of an Extended Kalman filter with a 6D model, of which 3D position and 3D orientation. Kalman filtering, also known as linear quadratic estimation, is an algorithm that uses a series of measurements observed over time from sensors in the presence of noise, and produces an estimate of them by then making a weighted average [45]. This package is named *robot-pose-ekf*.

Each sensor source refers to own world reference frame, so different sensors cannot be compared between them. Therefore also if each sensor send an *absolute pose*, this package uses the *relative pose* differences of each sensor.

Parameters

In this section the parameters related to *EKF* package are described as follows [46]:

- **frequency** *(default: 30)*: the filter will output the position estimate at frequency specified in hertz. Until the filter does not receive at least one message from one of all the inputs, it will not work. The imposed frequency value was *20 Hz*.

- **sensor-timeout** *(default: $\frac{1}{frequency}$)*: represents the minimum period in seconds with which the filter will generate a new output. The term timeout derives from the fact that we consider timed out a sensor after reaching of this time specified. In this case, a predict cycle on the filter is performed without correcting it. A value greater than the default one obtained (0.05), was set: *0.1*.

- **two-d-mode** *(default: false)*: if this parameter is set to true, the effect of small variations in the ground plane is ignored. So, only 2D information will be used for pose estimate, while the 3D ones will not be considered. For this reason, a *true* value was set.

- **transform-time-offset** *(default: 0.0)*: this parameter gives an offset to the transform generated by the *ekf-localization-node*, useful for interaction with other packages. A value of *0.3* was selected, to make the transformation visible after a while in terms of seconds.

- **transform-timeout** *(default: 0.0)*: after this value, each transformation became available for *tf* listener. This parameter was set to *0.0*.
- **print-diagnostics** *(default: false)*: this parameter shows the problems encountered in the node through the command *echo /diagnostics-agg* topic, and so useful, it was set to *true*.
- **debug** *(default: false)*: if set to true, a large amount of information is outputted about debug settings inside the file *debug-out-file*. As not necessary, a *false* value was set.
- **debug-out-file** *(default: robot-localization-debug.txt)*: this parameter specifies the full path of debug file, and that relating to this project is the following */debug/robot-localization-debug.txt*.
- **publish-tf** *(default: true)*: indicates if to broadcast the transformation on the **/tf** topic, and it was set to *true*.
- **publish-acceleration** *(default: false)*: whether or not to publish the acceleration state. As not needed, a *false* value was selected.

All the coordinate frames have already been described in the *tf* package. The main ones contained inside the *ekf* package are *map*, *odom*, *base-link* and *world*. The position described by *odom-frame* will drift over time, but remains accurate in the short time. So the *odom-frame* is the best frame for executing local motion plans. The *map-frame* contains a globally accurate position estimate, but it is subject to discrete jumps (e.g. due to the fusion of GPS data or position updates). The *world-frame* gives a common reference frame to relate multiple map frames. Below, the frames settings are described:

1. To set the *map-frame*, *odom-frame* and *base-link* frames to the appropriate frame name of system. If the *map-frame* does not exist, to set the *world-frame* to the value of *odom-frame*.
2. To set *world-frame* to the *odom-frame* value, if position data as wheel encoder odometry, visual odometry or IMU data are fused. This is the default behaviour for the *robot-localization*'s state estimation nodes.
3. If a global absolute position data is fused, to set the *world-frame* to *map-frame* value.

The default values and those set for this project, are reported below:

- **map-frame** *(default: map)*: *map*

- **odom-frame** *(default: odom)*: *odom*
- **base-link-frame** *(default: base-link)*: *base-footprint*
- **world-frame** *(default: odom)*: *odom*

The parameters described below, indicate all inputs take into account by the filter, and this means not having the default values. In order to add an input, to append the next number in the sequence to its "base" name, e.g. odom0, odom1, twist0, twist1, imu0, imu1, etc.

- **odom0**: this parameter indicates the topic name of the received input by the filter. By taking into account the camera input, the topic was written as follows */camera/odom/sample*.
- **odom0-config** *(default:*

  [false, false, false,
  false, false, false,
  false, false, false,
  false, false, false,
  false, false, false]):

  this parameter allows us to read some or all updates of the filter's state, setting false or true to the requested update value. This guarantees a greater control on measurement of values that feeds the filter. The order of the values is x, y, z, *roll*, *pitch*, *yaw*, v_x, v_y, v_z, v_{roll}, v_{pitch}, v_{yaw}, a_x, a_y, a_z. For example, if the z position value of an odometry message is used, then to set the entire vector to false, except for the third entry. Note that some message types do not provide some of the state variables estimated by the filter. For example, a TwistWithCovarianceStamped message has no pose information, so the first six values would be meaningless in that case. Within the topic considered in the previous parameter, there were the data relating to the x and y camera coordinates and their respective velocities, its orientation and the angular velocity around the axis z. Accordingly, this parameter was set as follows:

  [true, true, false,
  false, false, true,
  true, true, false,
  false, false, true,
  false, false, false]

- **odom0-queue-size**: if high-frequency data or low frequency parameter value are used, the size of the subscription queue has increased so as to merge more measurements. A value of *20* was chosen.

- **odom0-nodelay**: setting to true, disables the Nagle's algorithm, which causes strange behaviour when a large messages in ROS arrive at a high frequency. This option tells the ROS subscriber to use the tcpNoDelay option to disable this algorithm. For avoiding this behaviour, the parameter was set to *true*.

- **odom0-relative**: if set to true, the first measurement for a given sensor is considered a "zero point" for all future measurements. The same effect can be reached with the next differential parameter, but the only difference is that the relative parameter doesn't cause the measurement to be converted to a velocity before integrating it. This parameter was set to *true*, for having an absolute measure of the data received from this topic. In reverse, the next parameter will be set to false, as indicates a relative measure.

- **odom0-differential**: this parameter is useful when the sensors under-report their covariances while measuring one pose variable. In this case it makes sense to correct the covariances measurement or, if the velocity is measured by one of the sensors, one sensor measures the pose value, while the second one the velocity. But this solution is not always feasible, and so the differential parameter is introduced, able of converting the absolute pose data to velocity one, by differentiating the absolute pose measurements. This parameter is used only for sensors that provide pose measurements, and not the twist ones. As anticipated, this parameter was set to *false*.

The following two parameters are used for setting a threshold values when the data is subject to outliers. These threshold settings are expressed as Mahalanobis distances to control how far away from the current vehicle state, a sensor measurement is permitted to be. Data is specified at the level of pose and twist variables, rather than for each variable in isolation. For messages that have both pose and twist data, the parameter specifies to which part of the message we are applying the thresholds. The selected values for these two parameters are reported below, appropriately chosen taking inspiration from other projects:

- **odom0-pose-rejection-threshold** *(default: numeric-limits<double>::max())*¦ *5*;

- **odom0-twist-rejection-threshold** *(default numeric-limits<double>::max())*¦ *1.*

The following parameters are further input, as odom described before, but they concern the **/poseupdate** topic.

- **pose0**: */poseupdate*
- **pose0-config**: the previous topic contained only the updated pose about the x and y camera coordinates, and its updated orientation around the axis. Consequently, the matrix obtained was:

  [true, true, false,
  false, false, true,
  false, false, false,
  false, false, false,
  false, false, false]

- **pose0-differential**: *true*
- **pose0-relative**: *false*
- **pose0-queue-size**: *20*
- **pose0-rejection-threshold**: a value of *4* was set, chosen as for the previous topic.
- **pose0-nodelay**: as for the previous topic, this parameter was set to true for disabling disables the Nagle's algorithm.
- **process-noise-covariance** *(default: [46])*: this parameter is expressed in form of matrix, and it represents the noise that we add to the total error after each prediction step, if set to true. The process noise covariance matrix can be difficult to tune, and can vary for each application, so it is exposed as a configuration parameter. The better the omnidirectional motion model matches your system, the smaller these values can be. However, if users find that a given variable is slow to converge, one approach is to increase the process-noise-covariance diagonal value for the variable in question, which will cause the filter's predicted error to be larger, and to trust the incoming measurement more during correction. The values are ordered as x, y, z, $roll$, $pitch$, yaw, v_x, v_y, v_z, v_{roll}, v_{pitch}, v_{yaw}, a_x, a_y, a_z.
- **initial-estimate-covariance** *(default: [46])*: this parameter represents the initial value for the state estimate error covariance matrix. The diagonal values represent the variance, and if they are set to large values, the initial measurements of the variables will be in rapid convergence. The values are ordered as x, y, z, $roll$, $pitch$, yaw, v_x, v_y, v_z,v_{roll}, v_{pitch}, v_{yaw}, a_x, a_y, a_z.

3.4 URDF

To describe and to visualize the overall 3D robotic structure of Paquitop in the ROS simulation environments, it is necessary to transfer all its mechanical characteristics in an XML file named *Universal Robot Description Format* (URDF). Unlike the *tf* package that maintains the relationship overtime between coordinate frames of the rigid parts of a robot, named *links*, this file also considers the movable components that enable the relative motion between adjacent links, named *joints*. In addition, this file describes, through a tree structure, the kinematic and dynamic properties of a robot, its appearance and its collision model. In the URDF file, the description of the robot's constituent elements occurs as follows.

The *links* are uniquely identified by a name, and their characteristics related to the *aspect*, the *inertia* and the *collision properties* are specified. The visual representation of a link consists of assigning a geometric shape among the graphic primitives made available by the URDF format (box, cylinder or sphere), or using the STL *(STereoLithography)* mesh file, which contains the 3D model of the examined link. Also the link's color is possible to add, by using the *Red-Green-Blue-alpha* (RGBa) *syntax*, namely a numeric array containing the colors with an alpha parameter, specifying the opacity. Besides, each link is provided of a mass, a mass center and an inertia matrix, all specified within the inertial properties. Unlike the visual representation, the collision properties outline the collision model, making a link size estimation through simpler models, in order to reduce computation time.

The *joints* connect the links together through a *parent-child relationship*, each identified by a unique name. As previously claimed, the purpose of the joints is to provide relative movements between two adjacent links, depending on their number of degrees of freedom. This component describes the *kinematics* and *dynamics* of the robot joints, which can be classified in six categories according to both their degrees of freedom and the range over which they can be moved. The joints classification is listed below:

1. **Revolute Joint**, represents a hinge joint, whose rotation occurs along one of its axes with a limited range of motion superior and inferior;

2. **Continuous Joint**, is similar to the previous joint but without any rotation limit;

3. **Prismatic Joint**, is a sliding joint, which allows a linear movement along its axis in a limited range;

4. **Fixed Joint**, admits no movement as its degrees of freedom are locked;

5. **Floating Joint**, enables the movement in all six degrees of freedom;

6. **Planar Joint**, allows the motion in a plane perpendicular to the specified axis.

As previously claimed, the Paquitop model consists of a link representing the chassis, named *base-link*, joined to two links representing the LiDAR sensor and the camera, namely *laser-link* and *camera-link* respectively. The latter are connected to the chassis through two fixed joints (Figure 3.21): *lidar-joint* and *camera-joint*.

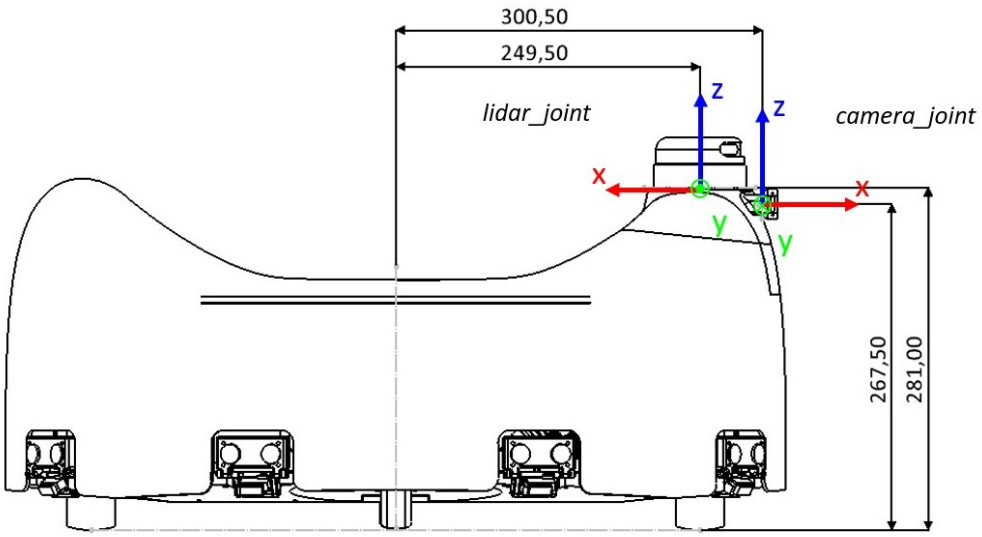


Figure 3.21: The Paquitop's Joints defined by frames and their coordinates expressed in millimeters with respect to the base.

The parent-child relationships present between the two joints are indicated in Table 1:

Joint	**Father**	**Child**
lidar-joint	base-link	laser-link
camera-joint	base-link	camera-link

Table 3.1: The Parent-Child Relationships of Links and Joints

Chapter 4

Validation Tests

In this chapter, the experimental tests for validating the control system implemented on Paquitop are described. The challenges that a mobile robot is being faced with to fulfill the autonomous navigation are path planning, localization and obstacle avoidance in either static and dynamic environments. These three sub-problems of the autonomous navigation are examined on the developed platform, testing its software part in the real-world conditions, which are described in detail, defining their purpose and their respective outcome, compared with that to be achieve.

4.1 Experimental Tests Description

A robot mobile must undergo tests to meet the customers demand and, at the same time, to ensure safety for both the human beings and the robotic platform, minimizing as possible as risks. Tests should be conducted repeatedly to guarantee that every part complies its desired specifications, and otherwise to diagnose the detected defect, outlining the source of the dysfunction coming from the mechanical, electronic or software parts of the system. Considering that the aim of the thesis is the development of the high-level of Paquitop, in this chapter the software testings are reported, fundamental to validate all the set parameters contained within the adopted packages, to fulfill the autonomous navigation of the platform.

The adopted procedure to check the whole behaviour of Paquitop, consisted of organizing tests into levels to maximize as possible as safety. Firstly, the sensors data were evaluated to check if they perfectly reflect either the surrounding environment represented in which operated, and the ability to localize the robot pose within it. To such aim, the mapping of many environments was performed to evaluate the capacity of the adopted Lidar to detect all the objects present in the considered workspace. Besides, the robot odometry given by the tracking camera was controlled to ensure that it was working properly. If sensors data were

reliable, tests about the motion can be conducted. Before operating the robot in a complex workspace, its software part has been tested firstly in a static environment, where nothing changes during the robot mission, and subsequently in dynamic environment, which evolves over time because of the moving entities. Compared to testing in a static environment, dynamic one considers the time factor crucial, as the frequency with which data was collected had to be fast enough to shun potential obstacles. In this case, the robot was placed in rooms with moving objects, whose purpose was to plan a safe trajectory, able to modify it in case of dynamic obstacles.

Specifically, the challenges addressed by Paquitop were initially carried out in the static environment, and subsequently in densely populated dynamic ones, as the main objective of the thesis was to develop a platform capable of assisting the human beings in domestic or also hospital environments. The robot motion planning in different conditions was studied, and the different challenges faced are listed afterward. The first consisted of locating the robot in a static environment where the navigation from an initial position to final one was required. The second was based on the static obstacle avoidance in a static environment, while the third checked the navigation in narrow static area. The last challenge incorporates the previous challenges, but in a dynamic environment. For each test, the evaluation was performed comparing the actual outcome with the expected one.

4.1.1 Navigation to a Goal Point

One of the skills of a mobile robot is to perform a task that requires the achievement of a desired pose. Specifically, this section will focus on the scenario where the robot had to reach a point along a straight line, as shown in the cost-map by Figure 4.1, where the red arrow represents the desired pose at a distance *3.0 m* from the starting point, the green trajectory indicates the path to perform, and the green polygon represents the Paquitop's footprint. To achieve the goal position as accurate as possible, it was necessary to set the right tolerance value through the *xy-goal-tolerance* parameter in both planners. As this parameter indicates the x and y distances in meters where the system considers the robot to have reached the goal, imposing a high value implied a non-precision to the desired point. To determine the parameter value to set, tests with values ranging from *0.025* to *0.20* were performed, and what has been noticed is that regardless of the imposed value, the robot reached always the final point at a distance of about *24 cm*. This means that the precision error was not related to the algorithm, but to an accuracy of the adopted sensors. However, the ideal desired value for this parameter to perform this task type in the best possible way, was of *0.075 meters*, even if the system neglected the value of the parameter set.

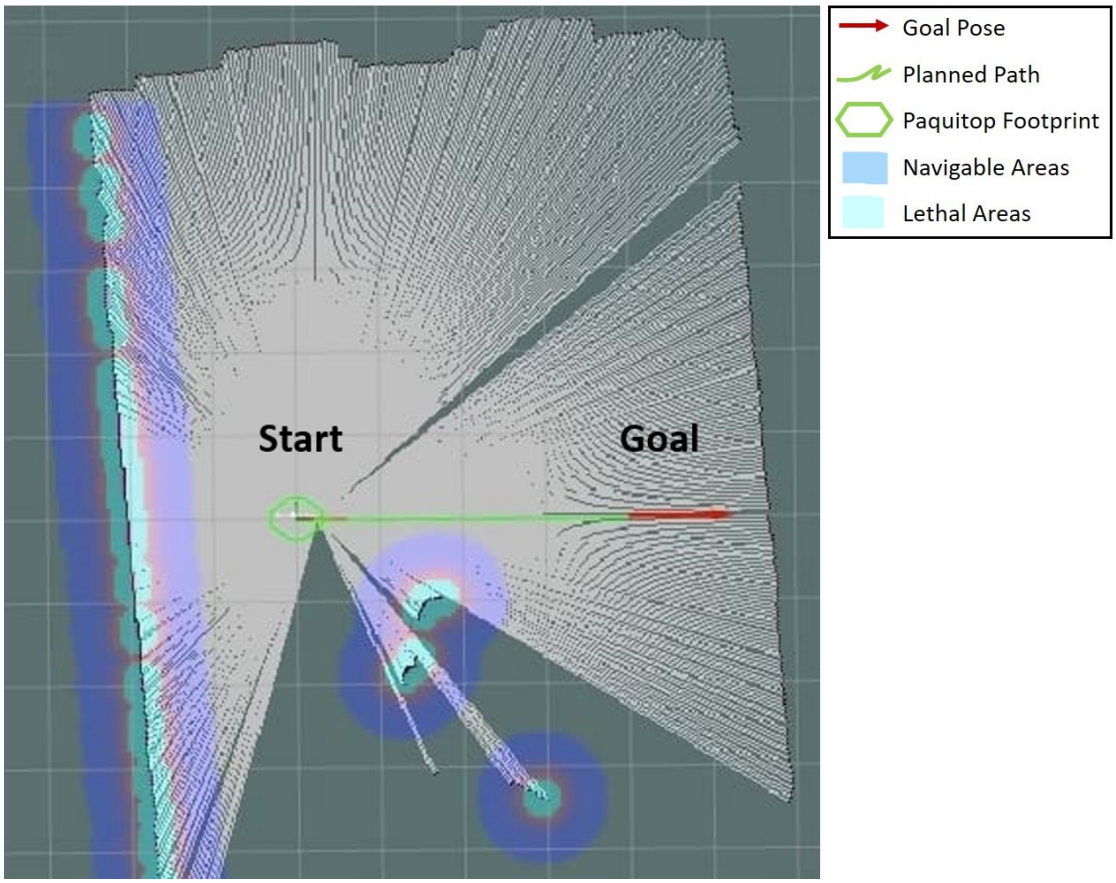


Figure 4.1: Navigation from an Initial Point to a Final Point. The global cost-map contains the Paquitop's footprint, the planned path, the desired pose, 3.0 m away from the starting point and the light and dark blue strokes, that define the inflation radius of the local and global map respectively, the values of which are 0.1 m and 0.8 m.

4.1.2 Obstacle Avoidance at Different Planners Frequencies

A basic requirement in the autonomous navigation is the obstacle avoidance, namely the ability of the robot to avoid both static and dynamic obstacles, modifying its planned path.

In this section, the challenge on the static-obstacle-avoidance at different controller frequencies is addressed. Specifically, the robot had to reach a pose overcoming an obstacle safely, in this case a panel with dimensions *84 cm·119 cm*. Figure 4.2 illustrates this scenario from the initial position (1) to the final one (4) distant *3.0 m*, where the red arrow indicates the goal pose, the green trajectory represents the planned path, while the blue trajectory, the performed one by the platform.

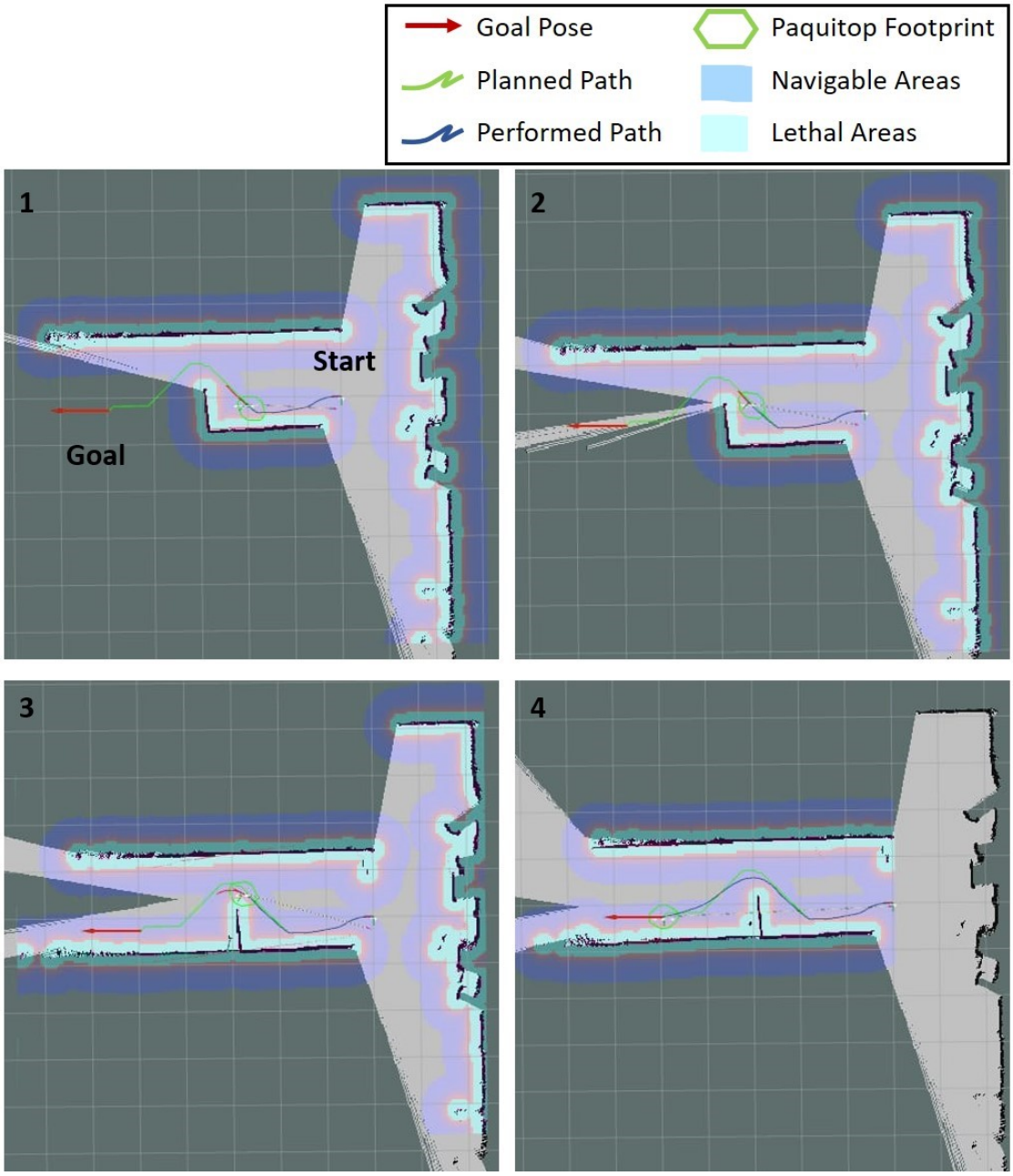


Figure 4.2: Static-Obstacle Avoidance between Two Points: the Start (1) and the Goal Points (4). The global cost-maps represent the steps to reach the goal, 3.0 m away from the starting point. The challenge consists of overcoming the static obstacle of dimensions 48 cm·119 cm, placed halfway through. Each map contains the Paquitop's footprint, the planned and performed paths, the desired pose and the light and dark blue strokes, that define the inflation radius of the local and global map respectively, the values of which are 0.1 m and 0.8 m.

To avoid the obstacle efficiently and move toward the free areas in the surrounding environment, it was necessary to impose the right value of the *controller-frequency* parameter to both local and global planners at the same time. Experiments demonstrated that varying this parameter from *2.0 Hz* to *5.0 Hz*, the planned trajectory was the same. The only difference is that setting a frequency lower than *3.0 Hz*, the robot spent a lot of time to perform the planned trajectory, as the system sent the velocity commands to the robot slowly. Instead, at a frequency higher than *3.0 Hz*, the robot received all the velocity commands quickly, so as not to be able to handle them, causing uncertainty in its motion, especially in the curvilinear section. A good compromise was to set the frequency value to *3.0 Hz*, as Paquitop was more reactive to shun an obstacle.

4.1.3 Navigation through Narrow Areas

The navigation through narrow spaces is a challenge of any autonomous vehicle, essential for a safe navigation of a robot. Sometimes, a robot perceives an insufficient space to navigate by means of the on-board sensors, due to the different reasons: the inaccuracies in the platform localization and in the motion execution, or noise and miscalibration error of the sensors adopted. An approach to address this issue is to plan waypoints in the considered area by a human operator, in order to successfully traverse it and to avoid the abortion of a possible path planning.

In this section, a procedure for satisfactorily performing this task is present, in order to achieve fully autonomous of Paquitop even in tight spaces, like corridors or doorways. Figure 4.3 illustrates the scenario where the robot has to cross a narrow space *119 cm* long, to reach the desired pose distant *3.5 m* from the starting point. To plan and execute the navigation, the robot had to have a margin of operation, which depended on the parameter of the inflation radius set in both local and global cost-maps, and the Paquitop's footprint width. The cost-map built by the on-board sensors is shown in Figure 4.4, which defined the navigable areas by Paquitop according to the navigation costs, and hence by the detected obstacles. Specifically, the light blue areas represented the lethal areas, while the blue one indicated the navigable areas. Considering that the value of the *inflation-radius* imposed in the local cost-map was of *10 cm* (light blue areas), and that the Paquitop's footprint (green polygon) width is about 48 cm, the minimum space required by Paquitop to plan a path was approximately of *65 cm*.

4.1.4 Path Planning in Narrow Areas with Static and Dynamic Obstacles

Path planning for mobile robots is a challenging problem, as the robot is required to reach a given goal in optimal way through the planning of a conflict-free path.

Figure 4.3: Paquitop's Path Navigation in a Narrow Space

Inaccurate localization during the navigation maneuvers and a high velocity, could cause significant drift and, as a consequence, performing paths that could be infeasible. The functioning of the localization system is fundamental, as allows the robot to complete the challenge without interruption, or to have mismatches either of the map and the trajectory, which could cause collision with any obstacle.

In this section, the software part of Paquitop has been tested planning a path in a narrow environment of size *6.5 m·6.8 m*, highly constrained by obstacles to avoid. Specifically, as the cost-map of the Figure 4.5 shows, the robot had to navigate from a predefined start location to a goal one, with the addition of two static and dynamic obstacles, which make the challenge even more difficult. In detail, the dynamic obstacle has a translational movement perpendicular to the goal direction, the first indicated in the Figure 4.5 with a dashed black arrow, the second represented with a red arrow.

To study the behaviour of the robotic platform, the experiments consisted of performing several times the planned trajectory by the robot from the initial point to the final one, and to validate the choice of the parameters set within the

implemented packages. The results demonstrated a good repetitiveness with small variations in the path between the considered points, a stable motion, sometimes little fluid, executed with a low velocity without any oscillations. This experiment proved that the robot was able to navigate to the goal efficiently, although the tight space full of objects with the addition of the two static and dynamic obstacles.

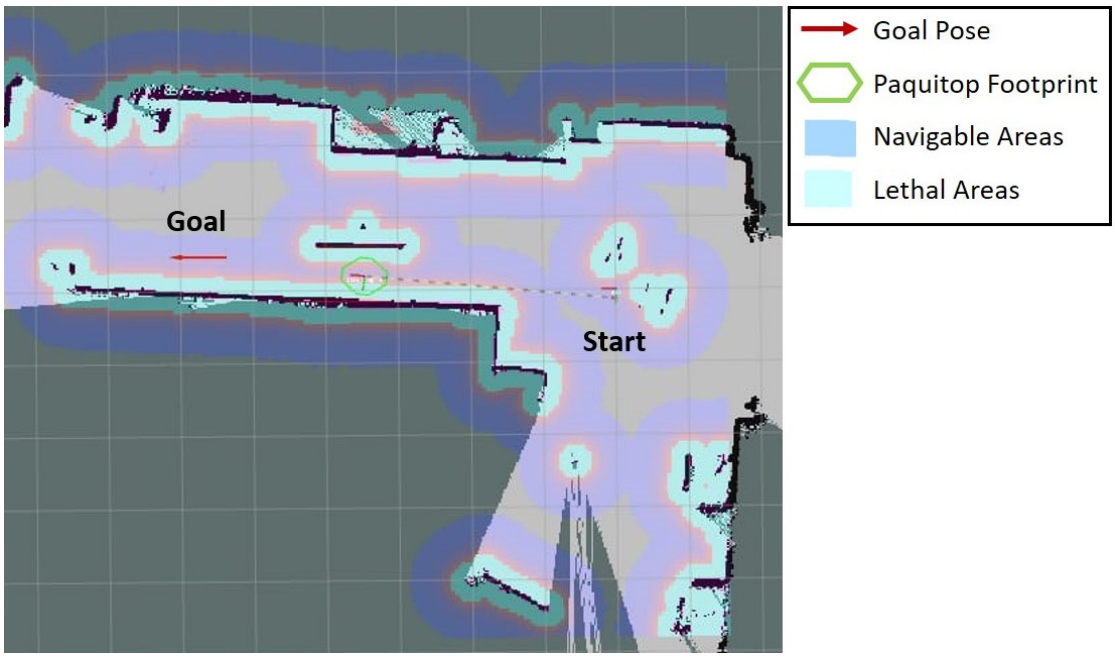


Figure 4.4: Cost-map of a Static Environment with a Narrow Space to cross. The scenario represents the route planning through a narrow space 119 cm long, where the starting point is 3.5 m from the end point. The light and dark blue strokes define the inflation radius of the local and global map respectively, the values of which are 0.1 m and 0.8 m.

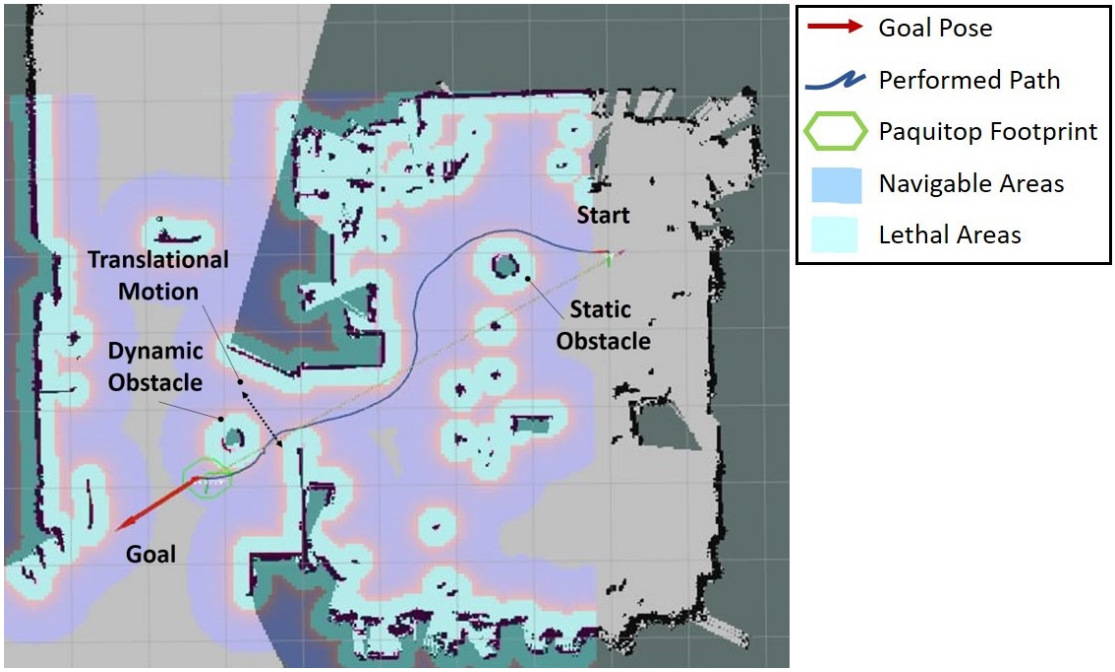


Figure 4.5: Navigation to a Desired Point with Static and Dynamic Obstacles in a narrow environment of size 6.5 m·6.8 m. The black dotted arrow indicates the translational motion of the dynamic obstacle. The light and dark blue strokes define the inflation radius of the local and global map respectively, the values of which are 0.1 m and 0.8 m.

Chapter 5

Conclusions

This thesis project has the objective to develop a *High-Level architecture* of a novel-actuated mobile platform, named *Paquitop*, for assistive robotics tasks such as monitoring or tracking of the human beings in domestic and non-structured environments. Its main peculiarity is to achieve omnidirectional planar motion on conventional wheels, thanks to the over-abundant set of actuators with respect to its degrees of freedom, crucial to enhance the dynamics of traditional robots.

The high manoeuvrability affects the motion planning and controlling, which constitute the ROS-based *software architecture* of the robotic system. Such SW architecture comprises all autonomous navigation algorithms, which address specific challenges as *obstacles avoidance*, *path planning*, *localization* and *mapping*.

Fundamental elements to solve these navigation tasks are the *exteroceptive sensors*, in particular a *Lidar*, which maps the surrounding environment taking into consideration the obstacles present, and a *tracking camera* for the robot localization. The ability to map and localize at the same time is called VSLAM *(Simultaneous Localization and Mapping)*. These information are crucial for the global and local planners, which build a free-path to reach a given goal without collision with any obstacle.

The obtained results through the experimental tests, presented a navigation architecture with a good capability to efficiently navigate to the goal, both within large and tight spaces lived by people and full of static and dynamic objects. On the one hand, this depended on an efficient accuracy of the sensors adopted, which developed a precise mapping of the surrounding environment and an accurate location of the robotic system inside it. On the other hand, a valid software design allowed a good planning of a path thanks to proper use of the implemented algorithms.

Future developments foresee the development of a waypoints planning system based on Bezier curves, to achieve a more stable motion of the robot without interruptions. In detail, the path planner would first generate a series of points

that satisfy the cost function and the avoidance of obstacles, and secondly it would use the Bezier curves to smooth the path.

Besides, the replacement of the adopted on-board computer with another more performing in terms of processor's response time, will allow the platform to improve its computational capabilities, such as to own a higher control frequency and consequently to be a more reactive system with dynamic obstacles.

To improve the performance of the autonomous navigation system, it is recommended to carry out more experimental tests in the future.

Another future development could be the development of a control algorithm in the ROS2 environment, which works not only on Linux systems but also Windows, MacOS e RTOS. This will improve the timeliness of the control and the performance of the entire robot, thanks to a system called DDS *(Data Distribution Service)*, which manages the distribution of data in real time while maintaining the publish/subscribe paradigm. Such system will enhance the communications network performance between multiple robotic systems.

The implementation of robotic system on the Paquitop's chassis such as a commercial collaborative arm (Figure 5.1), it could be a future work to implement manipulation tasks.

However, it will be necessary to adopt a more robust mechanical structure of Paquitop to ensure good performance despite the weight of the arm, for example

Figure 5.1: Paquitop and a Collaborative Arm for Manipulation Tasks

implementing a new motor that allows a more torque above all during executing uphill paths or curvilinear trajectories. The addition of a depth camera on its end-effector will check the achievement of the desired pose of the arm by means of the data collected by the camera.

Bibliography

[1] Gregor Klancar. *Wheeled Mobile Robotics: From Fundamentals Towards Autonomous Systems*. Oxford: Elsevier Science Technology, 2017 (cit. on pp. 2, 9).

[2] *Robotnik Summit XL*. URL: https://www.msarobotics.com.au/ (cit. on p. 2).

[3] *Phantom4 Robot*. URL: https://dronpro.cz/dji-phantom-4-advanced (cit. on p. 2).

[4] *Qysea FIFISH PRO V6 Plus Underwater Robot*. URL: https://www.ipapro shop.fi/tuote/qysea-fifish-v6-plus (cit. on p. 2).

[5] *Wheeled Mobile Robot*. URL: https://www.salfdsdfsl.top/ProductDetail.aspx?iid=81613242&pr=41.88 (cit. on p. 2).

[6] *Bunker Robot*. URL: https://www.agilex-italia.it/prodotti/#bunker (cit. on p. 3).

[7] Luca Bruzzone. *Functional Design of a Hybrid Leg-Wheel-Track Ground Mobile Robot." Machines (Basel)*. Vol. 9. Oxford: Elsevier Science Technology, 2021, pp. 1–11. URL: https://doi.org/10.3390/machines9010010 (cit. on p. 3).

[8] *Spot Robot*. URL: https://www.hstoday.us/industry/asylon-and-boston-dynamics-partner-to-combine-air-and-ground-robotic-security/ (cit. on p. 3).

[9] *Athlete Robot*. URL: https://ciencia.nasa.gov/science-at-nasa/2009/08apr_apolloupgrade (cit. on p. 4).

[10] *ANM1F2*. URL: https://beau_melelo26.artstation.com/projects/mLl3Y (cit. on p. 4).

[11] *Wheel and Track Hybrid Robot*. URL: https://www.semanticscholar.org/paper/Wheel-26-Track-hybrid-robot-platform-for-optimal-in-Kim-Kim/b50333f26ae79f8f31ed4e5a5eeb57e434aedd77 (cit. on p. 4).

[12] *Azimut Robot*. URL: https://www.pinterest.it/pin/833658580992856523/ (cit. on p. 4).

[13] Colucci G. Tagliavini L. *Wheeled Mobile Robots: State of the Art Overview and Kinematic Comparison Between Three Omnidirectional Locomotion Strategies*. Springer, 2021 (cit. on pp. 6, 8, 9).

[14] *Mecanum Wheel*. URL: https://www.amazon.com/Mecanum-Aluminum-Industrial-Accessories-Coupling/dp/B08H2F29CJ (cit. on p. 6).

[15] *Poly Wheel*. URL: https://toykidmama.com/15kg-load-58-82mm-aluminum-alloy-omni-wheel-metal-fulai-wheel-omni-robot-ros-platform-omnidirectional-motion-46959 (cit. on p. 6).

[16] *Spherical Wheel*. URL: http://hackproj.blogspot.com/2012/10/ (cit. on p. 6).

[17] Kevin M. Lynch. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017 (cit. on p. 10).

[18] Fernando Gomez-Bravo Giuseppe Carbone. *Motion and Operation Planning of Robotic Systems*. Springer International Publishing Switzerland, 2015 (cit. on p. 11).

[19] Marton Premebida C. Ambrus R. *Intelligent Robotic Perception Systems*. IntechOpen, 2018. URL: https://doi.org/10.5772/intechopen.79742 (cit. on p. 11).

[20] Andrea Macario Barros. *Visual-SLAM Algorithms*. Scholarly Community Encyclopedia, 2022. URL: https://encyclopedia.pub/entry/20846 (cit. on p. 13).

[21] Junning Zhang Binbin Xu Pengyuan Liu. *Progress in the Research of Visual SLAM*. Atlantis Press, 2018 (cit. on p. 13).

[22] *RViz*. URL: http://wiki.ros.org/rviz (cit. on p. 16).

[23] *Gazebo*. URL: https://gazebosim.org/home (cit. on p. 16).

[24] et al. Tagliavi L. Botta A. *Mechatronic Design of a Mobile Robot for Personal Assistance*. 2021 (cit. on pp. 17, 19, 22, 23).

[25] et al. Carbonari Luca. *Functional Design of a Novel Over-Actuated Mobile Robotic Platform for Assistive Tasks*. Vol. 84. Springer International Publishing, Cham, 2020, pp. 380–389 (cit. on p. 25).

[26] Tagliavini Luigi. *Low Level Architecture* (cit. on pp. 26, 28).

[27] *Digital Telemetry Radio System*. URL: https://it.banggood.com/Original-FrSky-2_4G-ACCST-Taranis-X9D-Plus-Transmitter-With-X8R-Receiver-for-RC-Drone-FPV-Racing-p-940819.html?cur_warehouse=ES&ID=42482 (cit. on p. 28).

[28] *SLAMTEC RPLIDAR A2 2D 360*. URL: https://alexnld.com/product/slamtec-rplidar-a2-2d-360-12-meters-scanning-radius-lidar-sensor-scanner-for-obstacle-avoidance-and-navigation-of-a-g-v-uav/ (cit. on p. 30).

[29] *RPLIDAR A2: Introduction and Datasheet*. Slamtec, 2016 (cit. on p. 30).

[30] *ROS:tf with RPlidar*. URL: https://stackoverflow.com/questions/69786218/ros-tf-with-rplidar (cit. on p. 31).

[31] *Intel RealSense Tracking Camera T265*. URL: https://www.intelrealsense.com/tracking-camera-t265/ (cit. on p. 31).

[32] et al. Anders Grunnet-Jepsen Michael Harville. *Introduction to Intel® RealSense™ Visual SLAM and the T265 Tracking Camera* (cit. on p. 31).

[33] D T. Pham. *Introduction to Ai Robotics*. Cambridge University Press, 2002, pp. 569–569 (cit. on pp. 35, 37, 39).

[34] Kaiyu Zheng. *ROS Navigation Tuning Guide*. 2017 (cit. on pp. 42, 49, 50, 58).

[35] *Hector Trajectory Server*. URL: http://wiki.ros.org/hector_trajectory_server (cit. on p. 43).

[36] et al. Quang Hiep Do. *An Approach to Design Navigation System for Omnidirectional Mobile Robot Based on ROS*. Vol. 9. International Journal of Mechanical Engineering and Robotics Research, 2020, pp. 1502–1508. URL: https://doi.org/10.18178/ijmerr.9.11.1502-1508 (cit. on p. 44).

[37] *Move Base* (cit. on p. 44).

[38] et al. Fox D. *The Dynamic Window Approach to Collision Avoidance*. Vol. 4. IEEE Robotics Automation Magazine, 1997, pp. 23–33. URL: https://doi.org/10.1109/100.580977 (cit. on p. 47).

[39] et al. Li Xiuyun. *Obstacle Avoidance for Mobile Robot Based on Improved Dynamic Window Approach*. Vol. 25. 2017, pp. 666–676. URL: https://doi.org/10.3906/elk-1504-194 (cit. on p. 48).

[40] *DWA: Local Planner*. URL: http://wiki.ros.org/dwa_local_planner (cit. on p. 49).

[41] *A Star*. URL: https://en.wikipedia.org/wiki/A*_search_algorithm#/media/File:Astar_progress_animation.gif (cit. on p. 56).

[42] *Dijkstra*. URL: https://en.wikipedia.org/wiki/Dijkstra-27s_algorithm#/media/File:Dijkstras_progress_animation.gif (cit. on p. 56).

[43] *Global Planner*. URL: http://wiki.ros.org/global_planner (cit. on pp. 57, 62).

[44] *Costmap Inflation*. URL: https://answers.ros.org/question/254368/question-about-avoiding-obstacle/ (cit. on p. 62).

[45] *Kalman Filter*. URL: https://en.wikipedia.org/wiki/Kalman_filter (cit. on p. 73).

[46] *EKF*. URL: https://github.com/h2r/p3_pkg/blob/master/params/ekf.yaml (cit. on pp. 73, 77).