Politecnico di Torino

Master 's Degree in Mechatronic Engineering

Corso di Laurea

A.a. 2022/2023

Sessione di Laurea Luglio 2023

Development of a semi-autonomous terrestrial robot for forest management

Relatore:

Prof. Marcello Chiaberge

Tutor:

Prof. Carlos Xavier Pais Viegas

Candidato:

Francesco Di Giorgio

Abstract

Nowadays, the frequency of fires is increasing due to climate change. Effective forest management is vital in mitigating the intensity and spread of this natural disaster. Currently, the task of forest cleaning is carried out by tractors equipped with forestry mulchers, which are operated remotely using joysticks.

This thesis is part of a project to automate the cleaning process using a fully sensor-equipped electronic box. Specifically, it addresses the crucial aspect of precise line following, which is essential for efficient cleaning operations.

The unmanned ground vehicle (UGV) under study consists of two main components. The vehicle part comprises the tractor *LV 600 Pro*, which is responsible for the vehicle's movement. The second component is the sophisticated electronic box called *Sentry,* developed by Bold Robotics Lda, which allows the tractor to move autonomously.

This thesis aims to develop software for the component mentioned above. Various algorithms will be presented to handle and filter raw sensor data, also employing fusion methods to achieve higher measurement precision. Special attention will be given to navigation aspects, including the presentation of different motion planning strategies.

Extensive testing will be conducted in simulated environments using Gazebo to validate the proposed approaches. Subsequently, field tests will be carried out to provide real-world comparisons and assess the effectiveness of the developed solutions.

Keywords: ROS, UGV, line following, obstacle avoidance, sensor acquisition, control.

Acknowledgements

I want to thank Professor Marcello Chiaberge and Professor Carlos Xavier Pais Viegas for the opportunity they gave me. It was a fantastic project working strictly with my near collaborators Tiago Pereira and Tiago Gameiro; I would like to thank all the Bold Robotic Lda employers for their technical and moral support.

To conclude, I thank my parents for their sacrifice to support me economically and emotionally and all the fantastic people I met on my journey.

Table of Contents

List of Tables

List of Figures

Acronyms

UGV : Unmanned Ground Vehicle

ROS : Robot Operating System

COM : Common Sense Algorithm

LTG : Local Tangent Graph

SDK : Software Development Kit

SLAM : Simultaneous Localisation Mapping

AMCL : Adaptive Monte Carlo Localization

DWA : Dynamic Window Approach

SOM : System-on-module

DOF : Degrees Of Freedom

IMU : Inertial Measurement Unit

RTK : Real-Time Kinematic

LERP : Linear IntERPolation

SLERP : Spherical Linear intERPolation

EMAF : Moving Exponential Average Filter

EKF : Extended Kalman Filter

LIDAR : Light Detection and Ranging

YOLO : You Only Look Once

UTM : Universal Transverse Mercator

CTE : Cross-track Error

URDF : Unified Robotics Description Format

V-REP : Virtual Robot Experimentation Platform

MORSE : Modular Open Robotics Simulation Engine

ODE : Open Dynamics Engine

OGRE : Open-source Graphics Rendering Engines

LRF : Laser Range Finder

GNSS : Global Navigation Satellite System

GUI : Graphical User Interface

Chapter 1 Introduction

This chapter aims to present the main aspects of the thesis.

A first description of the general use of mobile robots in our society is done. Then the main reason for developing this particular project is given, going then to the objectives of this work, its structure, and its methodologies.

Section 1.1 Background

'Agriculture businesses, warehouses, military operations, healthcare institutions, and logical companies are all searching for novel and contemporary ways to improve operations efficiency, increase safety, ensure precision, and improve speed. Hence, all require autonomous vehicle support in the future' *[1].*

It is possible to see that this phrase represents how autonomous systems will change and are changing the world in which we live; its impact is due to these technologies' advantage: they can complete tasks that are hazardous or challenging for people to carry out, like in hazardous environments where humans would be at risk; the precision and accuracy that they can achieve, for example, to perform a surgery; furthermore they can operate without needing breaks or rest, making them highly efficient and effective in performing tasks that require continuous operation, such as monitoring a production line or continuous maintenance. All these aspects increase interest in study and research, making them fundamental and impactful in the development of society.

Section 1.2 Motivation

This thesis aims to develop the software part of a multi-component box equipped with several sensors and computers able to make a semi-autonomous available tractor. This machine is operated by a remote joystick, primarily designed to manage and care for the forests and green spaces. Its functionality includes the removal of branches, shrubs, bushes, and pruning trees, ensuring that these areas are kept clean and well-maintained at all times.

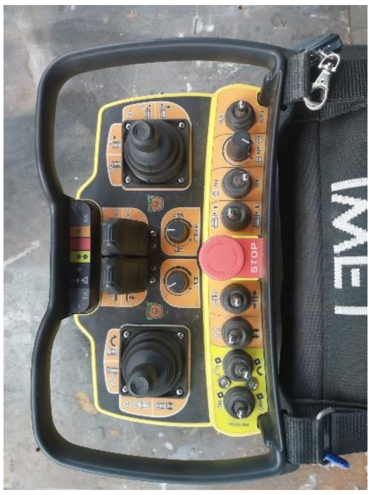


Figure 1 - Command joystick

This cleaning action is fundamental to decreasing the possibility of fire and reducing its devastating effects. This is a crucial aspect in the country where this thesis is developed (Portugal), but it is also taking more and more importance worldwide since the worsening of the condition of this problem.

For example, the European Environment Agency reports as forest fire risk has increased across Europe due to climate change. Despite this, the Mediterranean region has seen a slight decrease in burnt areas since 1980, indicating successful fire control efforts. However, in recent years, regions in central and northern Europe, which are not traditionally prone to fires, have experienced forest fires during record droughts and heatwaves.

Fire-prone areas are projected to expand, and fire seasons will lengthen in most European regions, particularly under high emissions scenarios. Therefore, additional adaptation measures are necessary, and Europe has experienced severe wildfire outbreaks and devastating fire seasons lately, mainly due to extreme weather conditions.

For example, there were record droughts and heatwaves during the spring and summer of 2017 and 2018 *[2]*. In *Figure 2* is a reported representation of the overall weather-driven forest fire danger in the present and under two climate change scenarios in the same article *[2]*.

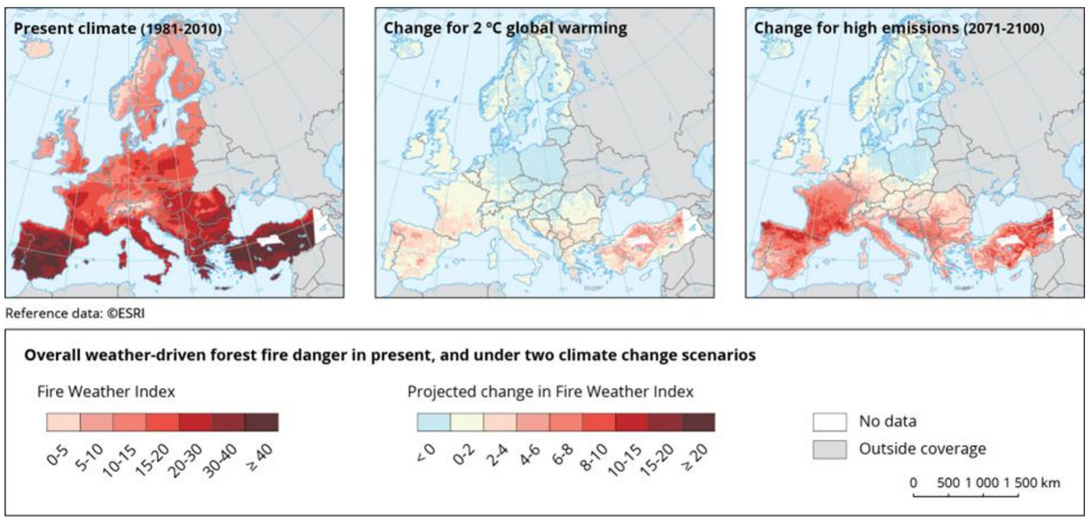

Present climate (1981-2010)
Change for 2 °C global warming
Change for high emissions (2071-2100)
Reference data: ©ESRI
Overall weather-driven forest fire danger in present, and under two climate change scenarios
Fire Weather Index
0-5
5-10
10-15
15-20
20-30
30-40
≥ 40
Projected change in Fire Weather Index
< 0
0-2
2-4
4-6
6-8
8-10
10-15
15-20
≥ 20
No data
Outside coverage
0 500 1 000 1 500 km

Figure 2 - Forest fire danger

It is possible to notice as climate change is likely a significant driver in increasing fire activity. According to *[3]*, there has been a five-fold rise in the frequency of intense heat waves compared to 150 years ago. Unfortunately, this trend will continue as a result of the planet's ongoing warming; the drier landscapes brought on by the rise in temperatures have increased the number of forest fires, which in turn have increased emissions and aggravated climate change, creating a feedback loop that leads to even more fires.

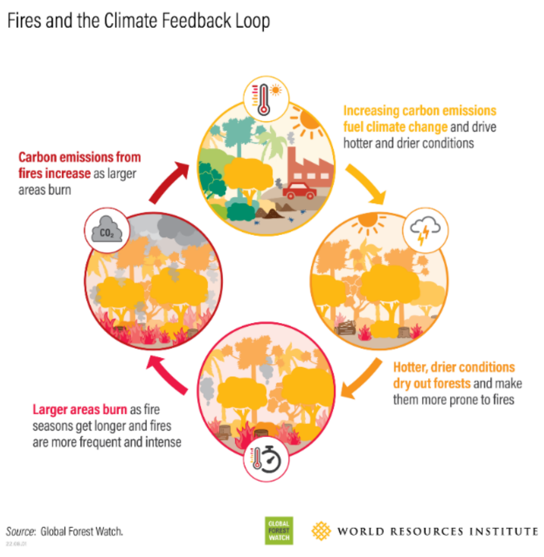

Fires and the Climate Feedback Loop
Increasing carbon emissions fuel climate change and drive hotter and drier conditions
Hotter, drier conditions dry out forests and make them more prone to fires
Larger areas burn as fire seasons get longer and fires are more frequent and intense
Carbon emissions from fires increase as larger areas burn
CO2
Source: Global Forest Watch.
GLOBAL FOREST WATCH
WORLD RESOURCES INSTITUTE

Figure 3 - Climate Feedback Loop

In this case, it is simple to see how the entire project's eventual objective will aid in putting out fires by lowering their impacts; in the meantime, various steps must be taken to lessen global warming's consequences.

Section 1.3 Objectives and Scope of the Study

As previously mentioned, this work focuses on the software development of the *Sentry*, that is, the brain able to make autonomous a general tractor.

Going deeper, the objective of this thesis is primarily the ability to follow a line with a mean distance of less than 1 meter. This value is evaluated based on the tractor's dimensions (1 meter is less than the width of the tractor's tool) and the project's maturity. Indeed the goal will be to show that this can be realised while many improvements will be made in the following years.

As regards the speed, it is not required to have a high one to be able to cut the vegetation efficiently, and this must be low, even if the cutting option will not be analysed in this thesis since it needs further analysis, such as how to understand if the vegetation is cut, how to control the speed as a function of the cutting process other than all the noise reduction need for the dust and vegetation raised by the tool.

Then during its navigation, the machine must avoid possible obstacles while going as much as possible on the path defined.

A key development point will ensure safety during the robot's functioning; it must stop if a person is detected inside a specific range or if the user requires an emergency stop action.

These aspects will be presented and supported by a further analysis of the sensors and algorithms used to achieve the requirements.

Resuming what will be made autonomous is the ability of the tractor to follow a predefined trajectory performing the cleaning action; in particular, the user will give some GPS coordinate about the area to be cleaned, and an API developed by the Bold Robotic company will generate a set of points to be achieved following a straight line between them; since the path generation is aspect outside of the robot development the problem it is defined as semi-autonomous.

Section 1.4 Methodology

The approach used in developing the UGV for line following can be resumed as a trial and error process. Indeed during the project, several presentations of the work were done, requiring to show something new and better than the last presentation each time. For this reason, different versions of the path-following algorithm will be presented with growing complexity and precision.

The work is based on studying the state of the art and solutions founded in the ROS community but also developed under the strict time requirement.

Section 1.5 Thesis Structure

In conclusion, the study will start from the analysis of state of the art on UGV and path-following; the other theoretical aspects like sensor technology, procedures and algorithms will be analysed directly before their discussion to avoid a too long and dispersive debate of state of the art since the widely of the topics studied and the will to focus on the navigation problem.

After this, an entire chapter will be dedicated to *ROS (Robot Operating System)* due to its importance in the process. Then the aspects of the sensors used are covered; how and why they are needed for the navigation, focusing on algorithm and methodology to reduce noise and increase performance.

Finally, the different algorithms used for navigation will be presented. To conclude, simulation aspects will be described, and the experimental test result will be shown.

Chapter 2 Literature Review

This chapter provides an overview of the two main aspects covered in the thesis.

In particular, the first part will define and present some examples of UGV in the market; then, some examples of this problem in other fields are shown.

Finally, some available algorithms for navigation, line path applications and obstacle are presented.

Section 2.1 UGV

A UGV is a vehicle that can operate on the ground without needing a human onboard [4].

Thanks to their versatility, it is easy to see that these machines can be used in several different situations, changing their properties in the function of the task and the environment.

To get the idea as a general UGV is done and in which field is used, some examples of machines available on the market are proposed.

Subsection 2.1.1 TurtleBot3

TurtleBot3 is a popular and widely used open-source UGV developed by the *Open Robotics* and *ROBOTIS* collaboration. It is designed as an affordable and accessible educational, research, and hobbyist platform. The TurtleBot3's core technologies are *SLAM*, *Navigation* and *Manipulation* in indoor environments.

It is presented in two versions: *Burger* and *Waffle Pi*. They will be described below based on *[5]*.

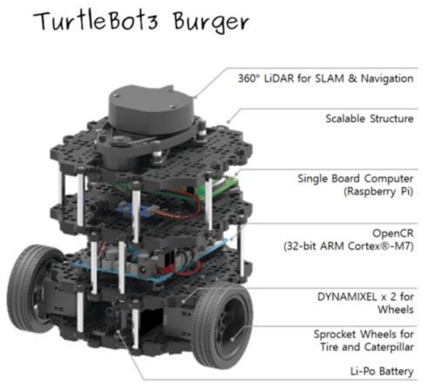

TurtleBot3 Burger
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 4 - TurtleBot3 Burger Specification

TurtleBot 3 Burger is the cheapest version. It features a round-shaped base with two wheels and a 2D 360-degree lidar sensor mounted on top. The *Burger* version is compact, lightweight, and highly manoeuvrable. The main computing unit is a Raspberry Pi.

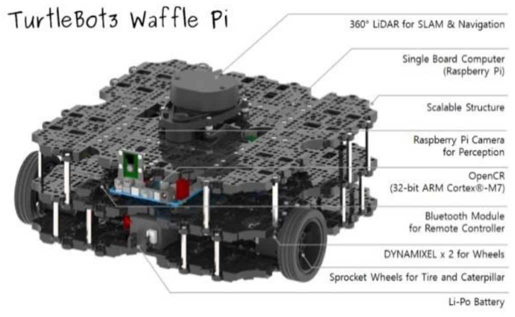

TurtleBot3 Waffle Pi
360° LiDAR for SLAM & Navigation
Single Board Computer (Raspberry Pi)
Scalable Structure
Raspberry Pi Camera for Perception
OpenCR (32-bit ARM Cortex®-M7)
Bluetooth Module for Remote Controller
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 5 - TurtleBot3 Waffle Pi Specification

The *TurtleBot 3 Waffle Pi* is the upgraded variant of the TurtleBot 3 platform. It has better actuators from XL430-W250 to XM430-W210 and a Raspberry Pi Camera Module v2.1. The *Waffle Pi* version is often used for research and development purposes requiring higher processing capabilities.

Subsection 2.1.2 Husky UGV

The *Husky UGV* is a popular and versatile platform developed by *Clearpath Robotics*.

It is designed for outdoor operations, but indoor configurations are available. It is widely used in research, exploration, and industrial applications.

As TurtleBot3, it is completely integrated into the ROS ecosystem; for some applications, it is used as a benchmark or to establish new robot research and development efforts. It is supplied with different preconfigured packages containing various sensors and hardware specific to the purpose of their application *[6]*.

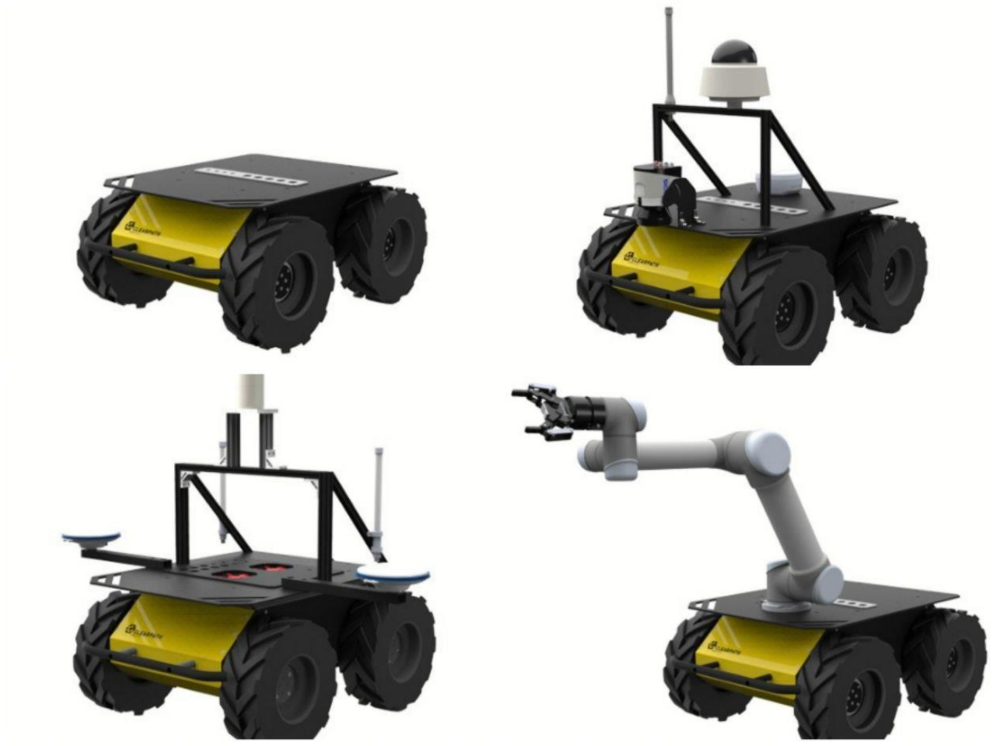


Figure 6 - Different Husky configuration.
From the left: Starter- Pro explorer- Mapping- Manipulator

Subsection 2.1.3 Roboteam's PROBOT

Roboteam's PROBOT is a UGV designed for military and security applications. Developed by Roboteam, it offers advanced capabilities and features to support logistics, intelligent gathering and casualty evac missions *[7]*.

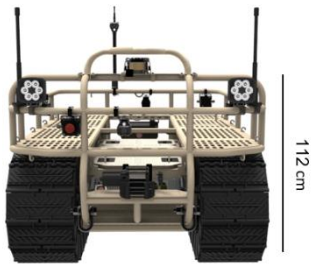


Figure 7 - Roboteam's PROBOT

As it is possible to see, all of these machines were built to be autonomous, while the vehicle used in the thesis is a general tractor that will be made autonomous by the *Sentry*. This will lead to a more complex scenario due to the necessity of handling aspect as the not optimised position of the sensors or the imperfect knowledge of the correlation between input command and tangible effect on the vehicle.

In particular, the machine used in this thesis is a remote-controlled tractor produced by *MDB SRL [8]*. The main characteristic of this machine is the ability to handle slopes until 60° in all directions (pitch and roll), a very robust chassis and thanks to the tool, it can cut vegetation; furthermore, the tool is able also to cut rocks during the working process, so that the presence of stone will not be a problem during the cleaning operation.

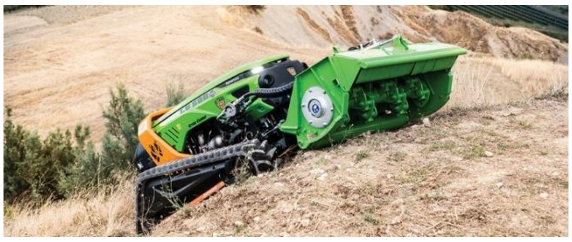


Figure 8 - lv-600-pro

The main disadvantages of using this machine instead of a specific UGV are:

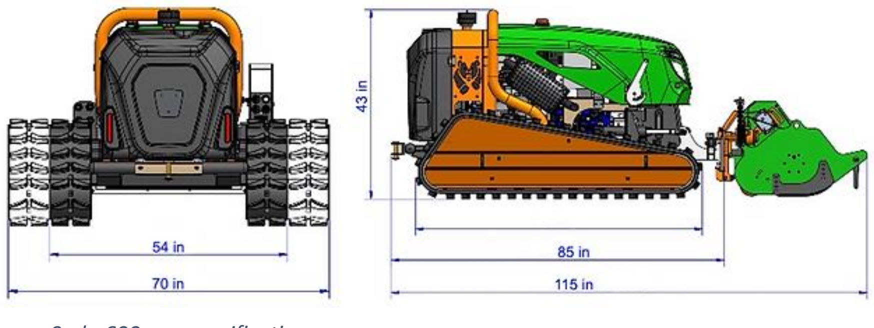


Figure 9 - lv-600-pro specification

- The *nonlinearity* between command input and actuators' effort; this is due to several reasons; firstly, the goal of the project is to build a 'box' able to make an autonomous general machine without the need for a precise model of its kinematic and dynamic; this means for example that the control of the speed will be possible only using as feedback the position of the robot and not the level imposed; secondly as proved by test the velocity of the machine at the same input change as a function of the type of field and the presence/absence of slope.

- *Straight motion* is not precise; even on a car road, the machine tends to go away from the path, as proved by the test.
- All sensors are placed in the *same spot*; this initially requires particular attention for noise generation, while during a routine mission does now allow to have measurements from different parts of the vehicle that could be useful for the big size of the machine.
- *Required calibration*; strictly connected with the first point, a first calibration part is needed before the first mission to get the minimum and maximum level of input for the forward and angular motion, depending on the terrain and the slopes present.

Section 2.2 Line-following problem

The problem of following a line or, more generally, a path is one of the most common in mobile robot applications. Indeed it is related to how a robot goes from one point to another, a generally valuable task for several applications such as industrial or agricultural.

Looking from an industrial perspective, vehicles were initially guided through optical or inductive guidelines. However, guidelines have their drawbacks, such as inflexibility regarding modifying or changing the routing and requiring installations on or in the ground *[9]*; it is easy to understand how this approach cannot be applied to the unconstrained environment other than being very restrictive.

Therefore a new type of navigation was proposed based on more complex sensors and algorithms, making the robot available to navigate without guidance.

Before entering in detail is important to define the two parts that constitute the problem: *trajectory planning* and *motion control*.

The first part is related to creating the reference points that will be used as input for the motion controller. In particular, the term trajectory indicates a path (locus of points in the operational space) combined with time.

It is possible to define it in two different spaces, *operational* and *joint*; the first is recommended for mobile robots because it is more efficient to deal with problems such as obstacle avoidance, while the second space is recommended for manipulators because it is easier to deal with singular configurations.

Since this study is about motion planning without time constraints, path generation in the operational space will be analysed.

The second part is related to the controller, which is how the reference error is transformed into the input. It is defined as path tracking: once a path or trajectory has been planned, path tracking algorithms are responsible for executing that path by controlling the robot's motion. Its goal is to ensure that the robot accurately follows the desired route and maintains its position and orientation along the path while accounting for factors like disturbances, uncertainties, and sensor noise. It can present different constraints, from the minimal requirement of reaching the final goal to more complex aspects such as smoothness, time needed, etc.

Subsection 2.2.1 Path planning

An overview of algorithms for path planning is presented based on *[10]*.

Firstly is essential to distinguish between *global path planning* and *local path planning*.

The first one is related to the construction of the path in a static way based on a previous complete/semi-complete knowledge of the environment. It can be run before the start of the mission saving computational power.

Local path planning instead answers to the need to compute the path dynamically according to the acquisition of the sensor; it is needed for unknown environments and generates new ways in response to new data.

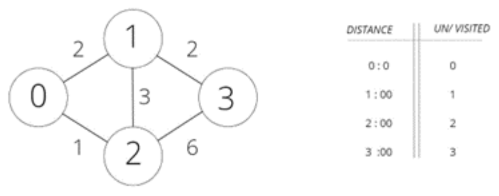
It is easy to understand how some algorithms are better for one or another problem in the function of the computational time and the optimal solution.

2.2.1.1 Dijkstra algorithm

Dijkstra's Algorithm is a widely used algorithm in computer science to find the shortest path between two nodes in a weighted graph. The algorithm is efficient and guarantees the optimal solution for finding the shortest route.

The functioning can be resumed in these steps:

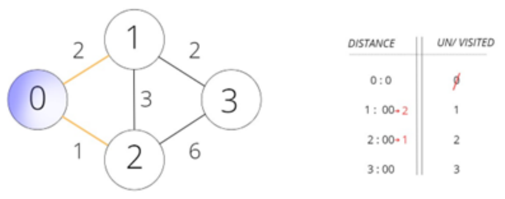
- Initialise the algorithm by setting the starting node as the current node and assigning a value of 0 to it while assigning a value of infinity to all other nodes in the graph. So mark all nodes as unvisited.



DISTANCE	UN/ VISITED
0 : 0	0
1 : 00	1
2 : 00	2
3 :00	3

Figure 10 - Inizialization

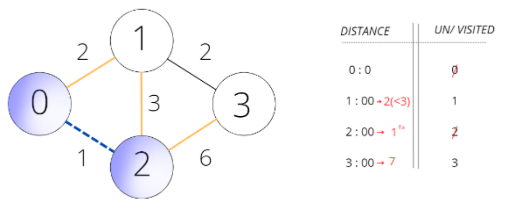
- For the current node, visit all of its neighbouring nodes (nodes connected to it by an edge) that have not been seen. Calculate the weighted distance from the starting node to that node for each adjacent node.



DISTANCE	UN/ VISITED
0 : 0	~~0~~
1 : 00→2	1
2 : 00→1	2
3 : 00	3

Figure 11 - Neighbour visiting

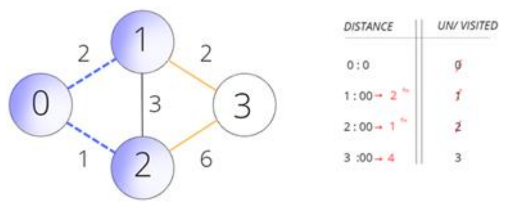
- If the calculated distance for a neighbouring node is less than that node's current assigned distance value, update the distance value to the newly computed value.



DISTANCE	UN/ VISITED
0 : 0	~~0~~
1 : 00→2(<3)	1
2 : 00→ 1^{fix}	~~2~~
3 : 00→ 7	3

Figure 12 - Next node selection

- After visiting the current node's neighbours, mark it as visited. A visited node will not be processed again.
- Choose the new node with the lowest distance value as the current node, and repeat until all nodes are visited, or the destination is reached.



DISTANCE	UN/ VISITED
0 : 0	~~0~~
1 : 00→ 2	~~1~~
2 : 00→ 1	~~2~~
3 :00→ 4	3

Figure 13 - Goal reached

Once the destination node has been visited, the algorithm terminates. To find the quickest route from the beginning point to the endpoint, trace back from the endpoint to the starting point by utilising the assigned distances and the graph's structure.

As reported in the paper, many versions of the *Improved Dijkstra algorithm* are present in the lecture, adapted for the precise purpose of the task. Some versions are reported in the table below, *Table 1*,taken by the article cited before *[10]*.

Algorithm	Static Constraints	Dynamic Constraints
Dijkstra	x	
Improved Dijkstra	x	
Multi-layer dictionary	x	x
Floyd and Dijkstra	x	

Table 1 - Dijkstra algorithm's version

As it is possible to see, due to the high computation required, this algorithm is efficient and used only for static constraints, then in a layer of global planning. At the same time, it is unsuitable for handling dynamic environments.

2.2.1.2 A* algorithm

This algorithm is commonly used in computer science and artificial intelligence to locate the shortest path between two nodes in a graph. It combines the benefits of *Dijkstra's algorithm* and a heuristic evaluation function to efficiently search for the optimal way. In particular, for each node, it computes a function *f(n)*:

$$f(n) = g(n) + h(n)$$

Where *g(n)* is the actual cost from node n to the initial node, *h(n)* is the cost of the optimal path from the target node to n computed by the heuristic function.

The most common heuristic functions are:

- *Euclidean distance* $\frac{x_1-x_2}{2} + \frac{y_1-y_2}{2}$
- *Manhattan distance* $|x_1 - x_2| + |y_1 - y_2|$
- *Octile distance* $max(|x_1 - x_2| + |y_1 - y_2|)$

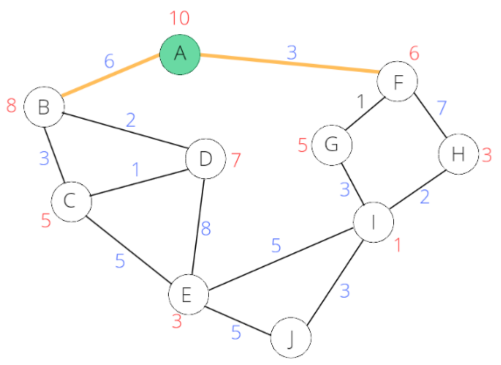


Figure 14 - Starting from A, F and B are analysed

An example taken from *[11]* is shown:

Imagining a path from A to J, weighted costs are shown in blue, while heuristic costs are in red. *F(n)* is then computed, summing the two costs.

As for the *Dijkstra algorithm*, all the neighbours are visited, and the one with the lowest cost is visited, iterating until the goal is reached. Is it possible to see from the picture that only its neighbours are analysed

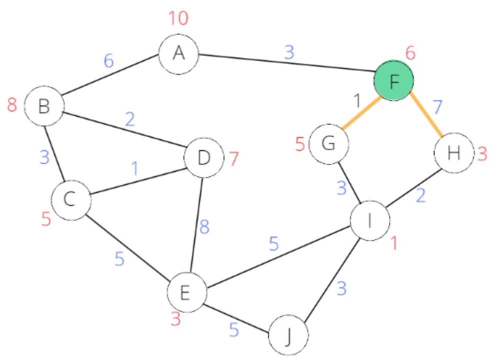


Figure 15 - F is selected since F has the lowest cost. Therefore, G and H are considered while B is not more considered

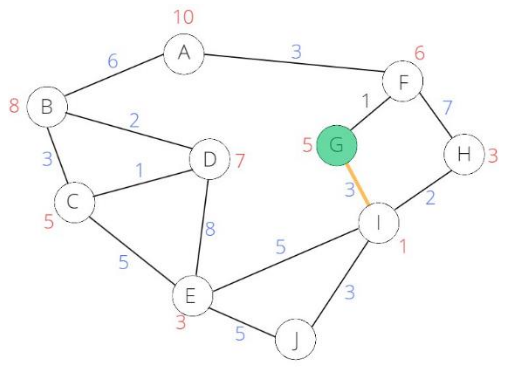


Figure 16 - G is selected.

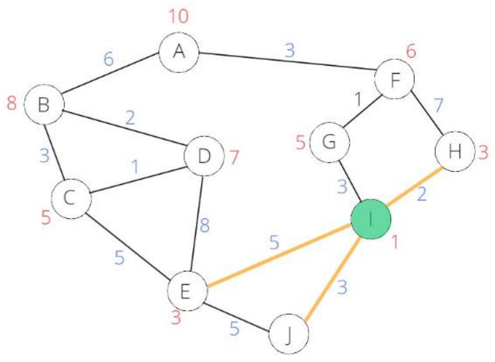


Figure 17 - I is selected, its neighbour is J therefore the goal is reached

once the node is selected, not considering the previously visited nodes, as in the Dijkstra algorithm. This makes the algorithm much faster.

Furthermore, it guides its search towards the most promising states using the heuristic function, potentially saving a significant amount of computation time. For this reason, it is widely used in static environments, but there are also instances where this algorithm is used in dynamic environments.

As for *Dijkstra's algorithm*, different variants are presented in the literature, designing and improving the performance for that specific purpose. The are reported in *Table 2*, taken from *[10].*

Algorithm	Static Constraints	Dynamic Constraints
A*	x	
Hierarchical A*	x	
Improved Hierarchical A*	x	
Hybrid A*		x
Guided Hybrid A*		x
A* with equal step sampling	x	
Diagonal A*	x	
A* with smart heuristics		x
Lifelong Planning A*	x	x

Table 2 - A* algorithm's version

2.2.1.3 D* algorithm

The *D* algorithm*, or *Dynamic A* algorithm*, is a path-planning algorithm used in robotics and artificial intelligence.

It is an extension of the *A* algorithm*. It is designed to handle dynamic environments where the cost of moving between nodes or the presence of obstacles can change over time. It updates the path as the environment changes, allowing the robot or agent to adapt and find an optimal path despite new obstacles or changed costs.

A simplified explanation is reported:

- The robot has a path from the starting position to the goal position based on the *A* algorithm*.
- *Obstacle detected*: The obstacle blocks a portion of the previously computed path.
- *Update cost estimates*: The *D* algorithm* updates the cost estimates for the affected nodes. The nodes directly affected by the obstacle receive a high cost.
- *Recalculate path*: Starting from the goal position, the *D* algorithm* performs a backward search, considering the updated cost estimates. It propagates the changes through the affected nodes until it reaches the starting position.
- *New path*: The *D* algorithm* computes a new way to avoid obstacles.

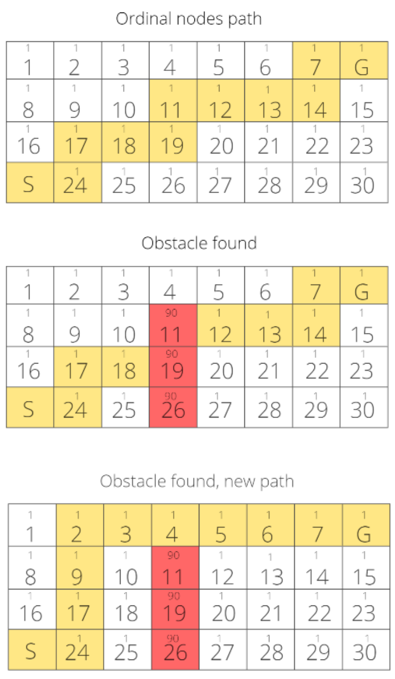


Figure 18 - D algorithm representation*

The *D* algorithm* allows the robot to dynamically adapt to environmental changes and find an optimal path in real-time by continuously updating the path based on new information.

As for the other algorithms, different versions are presented in*Table 3* taken from *[10].*

Algorithm	Static Constraints	Dynamic Constraints
D*		x
D* Lite		x
Enhanced D* Lite		x
Field D*		x

Table 3 - D algorithm's version*

Rapidly-Exploring Random Trees, *Genetic Algorithm*, Ant *Colony Algorithm*, and *Firefly Algorithm* are other possible ways to solve the problem presented in the article.

2.2.1.4 BUG algorithm

This subsection discusses another method specific for avoiding obstacles, based on *[12]*. The algorithms presented are in the family of *Bug Algorithms BAs*; as reported in the article, they are a path-planning technique that evolved from *maze-solving* algorithms. Different versions include the *Basic Bug Algorithm, Tangent Bug Algorithm, Bug2 Algorithm*, and *Bug Trap Algorithm*. General concepts will be resumed, starting from their first advantages requiring less potential memory and processing requirements.

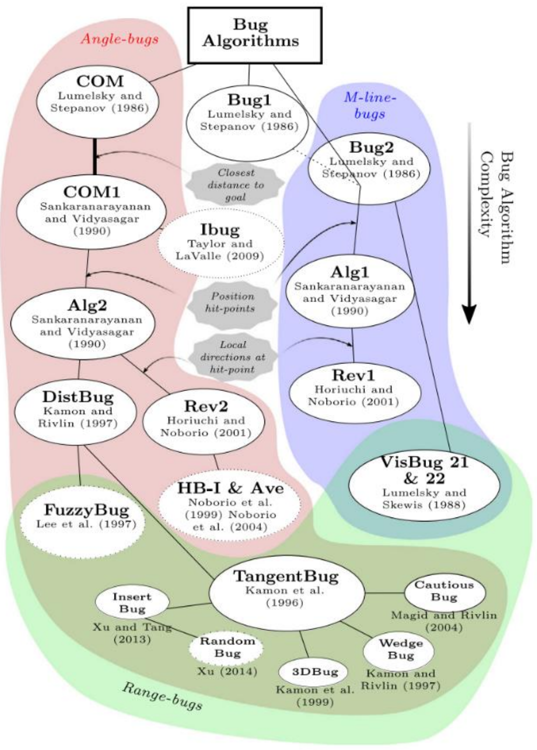

Bug Algorithms
Angle-bugs
M-line-bugs
Range-bugs
Bug Algorithm Complexity
COM Lumelsky and Stepanov (1986)
Bug1 Lumelsky and Stepanov (1986)
Bug2 Lumelsky and Stepanov (1986)
Closest distance to goal
COM1 Sankaranarayanan and Vidyasagar (1990)
Ibug Taylor and LaValle (2009)
Alg1 Sankaranarayanan and Vidyasagar (1990)
Position hit-points
Alg2 Sankaranarayanan and Vidyasagar (1990)
Local directions at hit-point
Rev1 Horiuchi and Noborio (2001)
DistBug Kamon and Rivlin (1997)
Rev2 Horiuchi and Noborio (2001)
VisBug 21 & 22 Lumelsky and Skewis (1988)
HB-I & Ave Noborio et al. (1999) Noborio et al. (2004)
FuzzyBug Lee et al. (1997)
TangentBug Kamon et al. (1996)
Cautious Bug Magid and Rivlin (2004)
Insert Bug Xu and Tang (2013)
Random Bug Xu (2014)
3DBug Kamon et al. (1999)
Wedge Bug Kamon and Rivlin (1997)

Figure 19 - Different Bugs algorithms [12]

Their general idea is based on the exact start and end position knowledge. Then as the obstacle is detected, its contour is used to avoid it, using it in different ways.

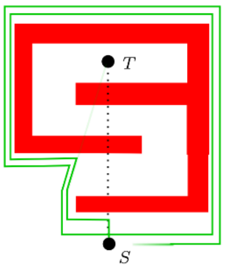


Figure 20 - Com [12]

The first version is called *Com* (common sense algorithm), and its main functioning principle is based on a wall following procedure. Indeed the robot always points to the goal, and as an obstacle is founded, it follows its contour until a free direction straight towards the goal is not reached. This algorithm has a problem with situation similar to *Figure 20*.

Different versions were evolved to solve this problem as *BUG 1* and *Bug 2*. The second one, for example, used an *M-line* approach, in which *M-line* connects the start point and the goal. The robot follows the wall from the *hit-point* (the first point of intersection between the trajectory and the boundary of the obstacle) until it reaches the intersection with the *M-line* on the other side.

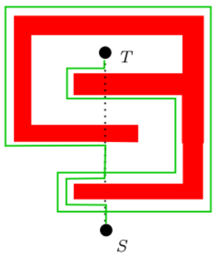


Figure 21 - Bug 2 [12]

From the ones described in the article, the *Tangent Bug* has a particle interest. This algorithm utilises a local tangent graph (LTG) within the range of the robot's sensors to navigate around obstacles and reach the target.

The construction of the LTG involves identifying the boundaries and discontinuities of the detectable obstacle field surrounding the robot. It represents the obstacles' edges in relation to the robot's current position.

Initially, the robot starts by moving towards the target while traversing the LTG edge that offers the quickest path to reach the target from its current position. However, if the length of path D on that edge increases, the algorithm saves the current range to the target as a local minimum. The robot then continues following the remaining boundary of the obstacle.

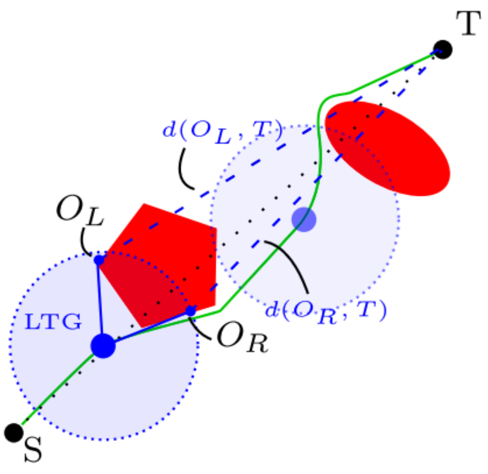


Figure 22 - Tangent Bug [12]

If, during the boundary traversal, the robot senses a node on the obstacle's boundary with a distance smaller than the previously saved local minimum, it triggers a leave condition. If feasible, the robot moves directly towards the target, *Figure 22*.

Also, the *InsertBig* can be of particular interest since it can be seen as a *Tangent Bug* that adds a safety margin to each obstacle detected.

Subsection 2.2.2 Path tracking

Once the plan is computed, the reference points are given to the controller to make the robot follows the path; several algorithms can be used some of them are presented.

2.2.2.1 Non-linear controller

A non-linear controller is presented in *[13]*.

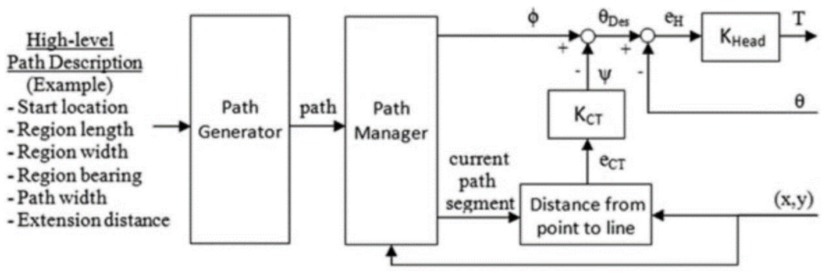
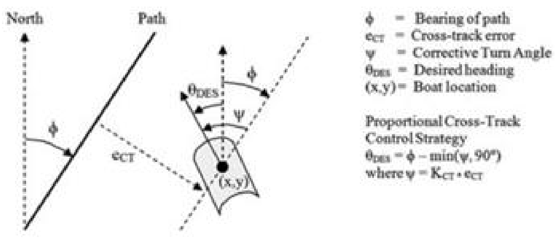


Figure 23 - Non-linear controller [13]

As it is possible to see, the non-linear controller is composed of two parts.

Firstly, a reference angle theta is given by the path manager. The outer loop takes the robot's position as feedback and compares it with the current path provided by the path manager to compute the distance from the reference line; this value is converted to an angle by using a proportional controller with gain K_{ct}, obtaining a correction term φ. This correction is subtracted to θ to obtain the θ *desired*; this is used as input for an inner loop that takes the actual heading as feedback to compute the actuation effect needed using another proportional controller K_{head}, as it is written in the article PI or PID controller can be used instead of the easy P.


North
Path
ϕ
θ_{DES}
ϕ
ψ
e_{CT}
(x,y)
ϕ = Bearing of path
e_{CT} = Cross-track error
ψ = Corrective Turn Angle
θ_{DES} = Desired heading
(x,y) = Boat location
Proportional Cross-Track Control Strategy
$\theta_{DES} = \phi - \min(\psi, 90°)$
where $\psi = K_{CT} \cdot e_{CT}$
Figure 24 - Angles representation [13]

2.2.2.2 Vector Field

This method is proposed in *[14]*. The control proposal is based on the construction of a vector field; as regarding the straight line case, a reference vector (going from the original position to the goal) is created, making the robot point to the destination. After that, the space is divided into two parts. An internal part near the reference path, the *transition region*, is dedicated to gradually moving the actual heading to the reference one; instead, the other region aims to push the robot back to the path. This is shown in *Figure 25* taken from the previously cited article.

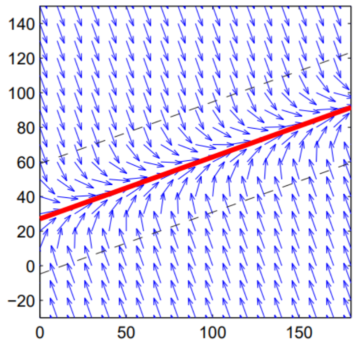


Figure 25 - Vector field representation [14]

As discussed in *[15]*, this method can be updated to avoid possible obstacles in the path. Indeed a vector field can be associated with the obstruction, making the robot deviate from the reference path to avoid it.

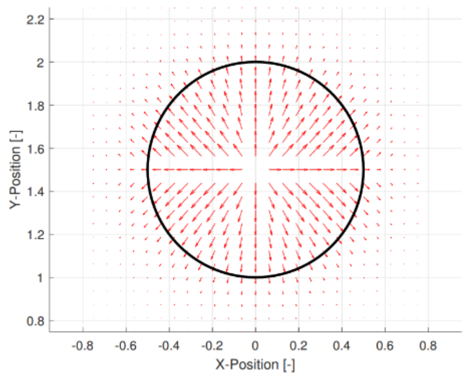


Figure 26 - Obstacle vector field [15]

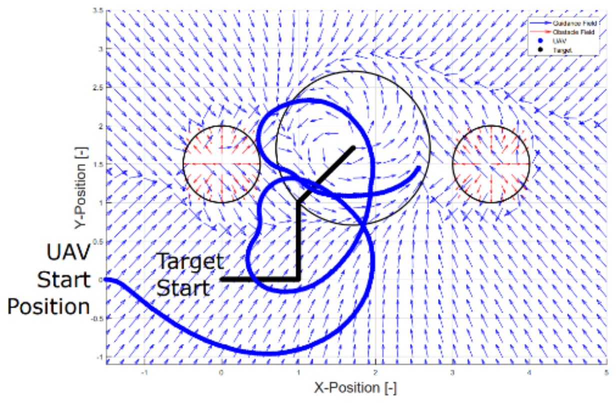


Figure 27 - UAV trajectory given by the vector field [15]

Chapter 3 ROS

The reason for an entire chapter about this aspect is its importance in the developing process as it allows the user to easily integrate hardware and software, other than making available a whole variety of packages that can be used to speed up the project's progress. All the following sections are taken from the very detailed ROS documentation *[16]*.

Section 3.1 Introduction to ROS

The Robot Operating System (ROS) is a flexible framework for developing and controlling robots. It provides a comprehensive collection of software libraries and tools specifically designed for building diverse robotic applications. By offering a distributed computing framework, ROS enables seamless communication between multiple processes or nodes by exchanging messages. These features foster modular development, simplifying the creation of complex robot systems by integrating independent packages.

One key aspect that differentiates ROS is its classification as a Meta-Operating System. This distinction arises from its capability to handle critical functions like scheduling tasks, managing resources, monitoring system activities, and handling errors.

ROS has a significant impact on the development process. It provides a comprehensive suite of functionalities that facilitate the integration of hardware and software components, greatly expediting project progress.

Additionally, ROS offers an extensive range of pre-existing packages that can be readily utilised, minimising development time and effort.

Section 3.2 ROS Ecosystem

ROS can be described as a comprehensive software development kit (SDK). It can be summarised into four main blocks, each crucial in the development process.

Subsection 3.2.1 Plumbing

Communication lies at the heart of robot development, and ROS excels. Its message-passing system forms the backbone of communication between distributed nodes, employing an anonymous publish/subscribe pattern. This standardised approach enables seamless interaction with components like LIDAR, cameras, localisation algorithms, and user interfaces. As mentioned before, ROS encourages the modular system by adopting this communication model.

Subsection 3.2.2 Tools

Asynchronous interaction with the physical world through sensors and actuators is crucial when developing software. ROS provides a suite of developer tools that greatly assist in this aspect. These tools include launch configuration, introspection capabilities, debugging utilities, visualisation aids, plotting tools, logging mechanisms, and playback functionality.

Subsection 3.2.3 Capabilities

The ROS ecosystem encompasses many device drivers, algorithms, and user interfaces that are application building blocks. Developers can use these previously available capabilities to better focus on the applications' specific requirements, accelerate development, and support innovation.

Subsection 3.2.4 Community

A strong and cooperative community forms the core of the ROS project. The point is straightforward: anyone interested in robotics applications should be able to make his idea real without understanding hardware and software complexities. The ROS community fosters knowledge sharing and encourages the realisation of robotic ideas.

Section 3.3 ROS Architecture / Main Concepts

ROS has three levels of concepts: the *Filesystem*, the *Computation Graph* and the *Community* levels. Each level is explained in detail in the following subsections.

Subsection 3.3.1 ROS Filesystem Level

The filesystem-level concepts in ROS encompass various resources encountered on disk. These resources include:

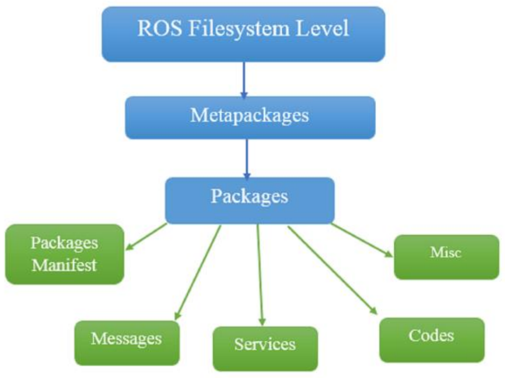


Figure 28 - ROS Filesystem Level

- *Packages*: They serve as the fundamental unit for organising software in ROS. They encapsulate ROS runtime processes (nodes), ROS-dependent libraries, datasets, configuration files, and other components. Packages are considered the elementary block in ROS.

- *Metapackages*: Metapackages are specialised packages designed to represent a group of related packages.
- *Package Manifests*: Package manifests, represented by the package.xml file, provide metadata about a package. The relevant metadata for a package consists of its name, version, description, license, dependencies, and other essential details. The package.xml format follows the specifications outlined in REP-0127.
- *Repositories*: Repositories comprise packages with a standard version control system. Packages within a repository share the same version and can be released using the *Catkin* release automation tool.
- *Message Types*: they describe the data structures for messages exchanged in ROS. These message descriptions outline the format and content of data sent and received between nodes.
- *Service Types*: Service types specify the request and response data structures for ROS services. Service descriptions define the format of data exchanged during service calls.

Subsection 3.3.2 ROS Computation Graph Level

The *Computation Graph* in ROS refers to the network of interconnected ROS processes collaborating to process data. It relies on several fundamental concepts that contribute to its functionality. The concepts mentioned can be found in the *ros_comm* repository.

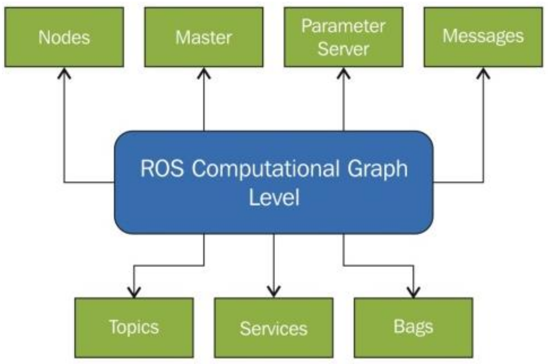


Figure 29 - ROS Computation Graph Level

- *Nodes*: Nodes are individual processes responsible for computation within the ROS system. They are designed to operate modularly, allowing a robot control system to comprise multiple nodes. ROS nodes are developed using ROS client libraries such as roscpp (C++) or rospy (Python).
- *Master*: The ROS Master is the central hub that provides the *Computation Graph* name registration and lookup services. It enables nodes to discover each other, exchange messages, and invoke services by facilitating communication.
- *Parameter Server*: The Parameter Server acts as a central storage location for data, allowing key-value pairs to be stored and accessed.

- *Messages*: In a network of nodes, communication happens through message passing. A message is a data type that has structured, typed fields. Typically, it includes standard primitive types like integers, floating-point numbers, Booleans, and arrays of primitive types.
- *Topics*: Messages are routed through a publish/subscribe-based transport system. A node publishes a message to a specific topic, which serves as an identifier for the content of the message. Other nodes interested in that data type can subscribe to the related topic.
- *Services*: Services provide a request/reply communication pattern compared to the publish/subscribe model. Two message structures are required to define services: one for the request and another for the response. A node offering a service represents a name and waits for requests, while a client sends a request message and awaits the corresponding response. Services enable synchronous communication between nodes.
- *Bags*: are a file format for storing and replaying ROS message data. They serve as a valuable tool for recording and analysing sensor data, facilitating the development and testing of algorithms.

Before concluding this subsection, it is important to remark on the different concepts between topic, service and action.

- *Topics*: Topics in ROS are designed for handling continuous data streams such as sensor data or robot state. They support a many-to-many connection model, where data can be published and subscribed at any time independently of specific senders or receivers. Callback functions are used to receive data as soon as it becomes available. The publisher controls when data is sent, making topics suitable for scenarios requiring continuous data flow.

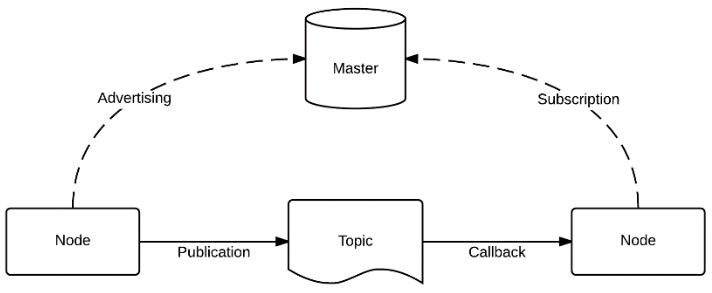


Figure 30 - Topics in ROS

- *Services*: Services in ROS are intended for remote procedure calls that involve quick termination, such as querying the state of a node or performing rapid calculations like inverse kinematics. They are unsuitable for long-running processes. Indeed, services use a blocking call mechanism.

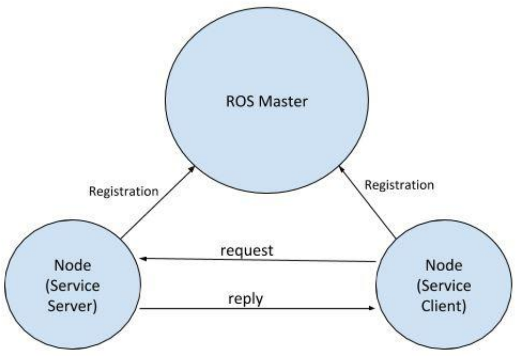


Figure 31 - Services in ROS

- *Actions*: Actions in ROS are suitable for discrete behaviours involving moving a robot or running tasks that take longer to complete. Actions provide feedback during execution that suits tasks requiring longer execution times, such as slow perception routines or initiating lower-level control modes. Actions are designed to be non-blocking and commonly used to execute complex real-world scenarios.

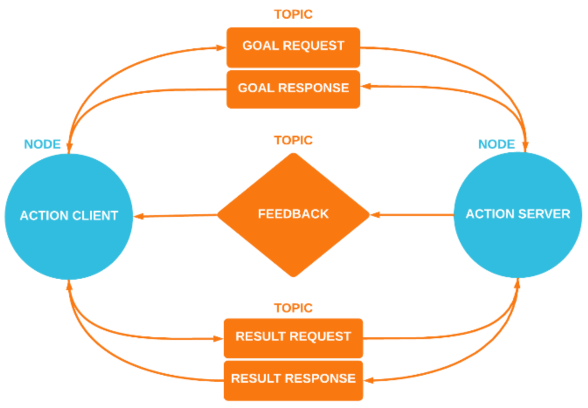


Figure 32 - Actions in ROS

Subsection 3.3.3 ROS Community Level

The *ROS Community Level* introduces resources that promote the exchange of software and knowledge among different communities.

- *Distributions*: ROS Distributions are like a collection of software stacks, similar to Linux distributions. They simplify the installation process by bundling together a set of software components.
- *Repositories*: ROS thrives on a federated network of code repositories, where diverse institutions and organisations contribute their robot software components.
- *The ROS Wiki*: At the heart of the ROS community lies the ROS Wiki, a dynamic platform for sharing and documenting ROS-related information. This Wiki is a central hub where individuals can register, share their expertise, and contribute to collective knowledge.

Section 3.4 Robot functions

In addition to its core middleware components, ROS offers a range of robot-specific tools and frameworks that accelerate the development process quickly. Some examples are reported below.

Subsection 3.4.1 Standardised Robot Messages

ROS provides a set of community-selected messages that cover a wide range of standard robot functionalities. These standardised messages include geometries, kinematics and dynamics, and sensors. These pre-defined messages allow us to focus on developing parts instead of building messages from scratch, enhancing interoperability with existing tools within the ROS ecosystem.

Subsection 3.4.2 Transforms Library

Managing a robot's static and dynamic geometry is a common challenge in robotics projects. The *tf* library in ROS helps address this challenge by providing tools for managing transformations between different frames of reference. Whether it is needed to transform sensor data to a global reference frame or determine the location of a robot's end effector in its local frame, *tf* can assist. It supports defining static and dynamic transforms, accommodating scenarios with multiple degrees of freedom.

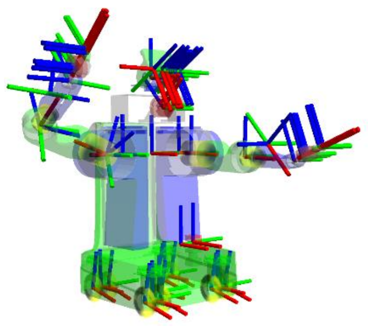


Figure 33 - Example of various Reference Frames connected by 'tf' library

Subsection 3.4.3 Robot Description

The description format used in ROS is *URDF*, which uses an XML document to define static and dynamic transforms and the robot's visual and collision geometries. *XACRO* format can be used to simplify the code reducing the lines of code.

Subsection 3.4.4 Diagnostics

ROS provides a standardised approach for producing, collecting, and aggregating diagnostics about the robot. This feature allows us to quickly assess the robot’s state and address any issues that may arise.

Subsection 3.4.5 Pose Estimation, Localization, and Navigation

ROS offers pre-existing packages that provide fundamental functions such as pose estimation, localisation, simultaneous localisation mapping (SLAM), and mobile navigation. More details are given in _Section 3.5_.

Section 3.5 Control and Motion Planning

This section wants to give a more profound description of the packages available in ROS regarding navigation problems.

Subsection 3.5.1 Navigation Stack

Navigation Stack in ROS is a comprehensive framework that enables robots to navigate their environments autonomously. It combines various components and algorithms to provide robust and reliable navigation capabilities. The more important aspects are described below.

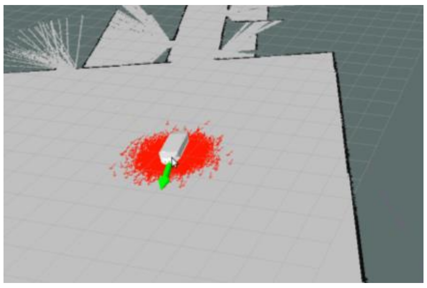


Figure 34 - Example of AMC

- *Localization*: Localization is crucial for a robot to determine its position and orientation within the environment. ROS offers several localisation methods, including *AMCL* (Adaptive Monte Carlo Localization), which utilises particle filters to estimate the robot's pose based on sensor measurements. *AMCL* uses sensor data, such as laser scans or camera images, to match against the existing map and accurately estimate the robot's location.

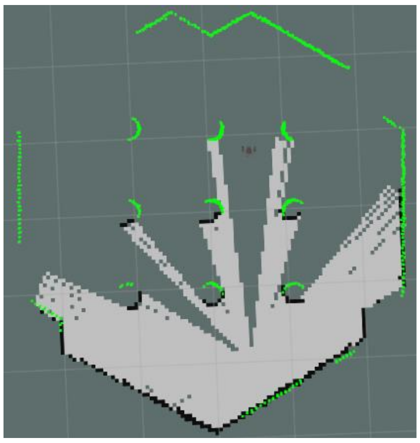


Figure 35 - Example of TurtleBot3 SLAM application

- *Mapping*: Mapping is creating a representation of the robot's environment. ROS provides mapping packages like *GMapping* and *Cartographer*, which implement SLAM algorithms. *GMapping* utilises laser scans to build 2D occupancy grid maps, while *Cartographer* can create 2D or 3D maps using laser scans or point clouds. These mapping algorithms integrate sensor data over time to construct a map of obstacles and free space in the environment.

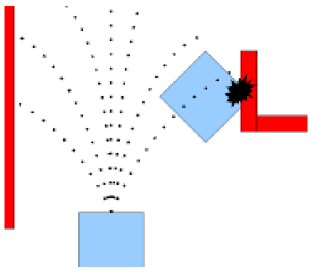


Figure 36 - DWA example

- *Path Planning*: Path planning involves computing a collision-free path from the robot's current position to a desired goal location. ROS offers path-planning algorithms such as *A** and *Dijkstra*, which can generate global ways based on the robot's map representation. The *Dynamic Window Approach (DWA)* algorithm also considers the robot's kinematics and dynamic constraints to generate real-time local paths, ensuring safe and efficient navigation.

- *Obstacle Avoidance*: Obstacle avoidance is essential for robots to navigate safely in their environment. ROS provides obstacle detection and avoidance techniques using sensors such as laser range finders or depth cameras. These sensors provide real-time data about the surrounding obstacles, which the planner uses to generate collision-free trajectories. Techniques like local cost maps and global cost maps are used to represent and update obstacle information in the robot's environment.

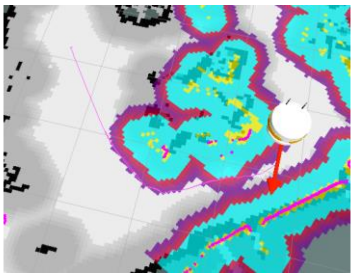


Figure 37 - Global cost map example

Section 3.6 Tools

ROS provides a range of tools to help to manage the complexity of robot systems and understand the state of the robot. Here are some essential tools provided by ROS:

Subsection 3.6.1 Command-Line

ROS offers over 45 command-line commands that allow access to all core functionality and introspection tools. These tools enable to launch nodes, introspect topics, services, and actions, record data, and perform various other operations.

Subsection 3.6.2 rviz

rviz is a widely known and powerful tool in ROS for the three-dimensional visualisation of standard sensor data types, and robots described using URDF. It supports visualising common message types like laser scans, point clouds, and camera images.

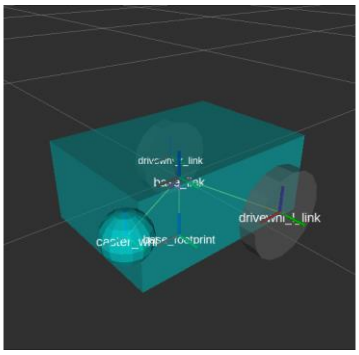


Figure 38 - rviz example

Subsection 3.6.3 rqt

ROS provides *rqt*, a Qt-based framework for developing custom dashboards. With *rqt*, personalised dashboards can be created by organising built-in *rqt* plugins and your own Qt/ROS plugins into tabs and split-screen layouts. This flexibility allows user interfaces tailored for the specific robot system requirements.

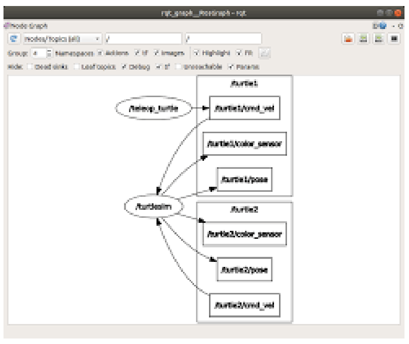


Figure 39 - rqt example

Subsection 3.6.4 rqt_graph

rqt_graph is a tool that provides introspection and visualisation of the live ROS computational graph. Thanks to a comprehensive overview of your robot's running ROS nodes and connections, it allows debugging.

Chapter 4 System Design and Architecture

This chapter will be focused on the sensor part; as previously mentioned, all of these components are mounted inside and on the metallic box, building the so-called *Sentry*, shown in *Figure 40*. The devolving of the sensor part is crucial for the autonomous machine since its precision determines the reliability of the localisation and detection of the environment.

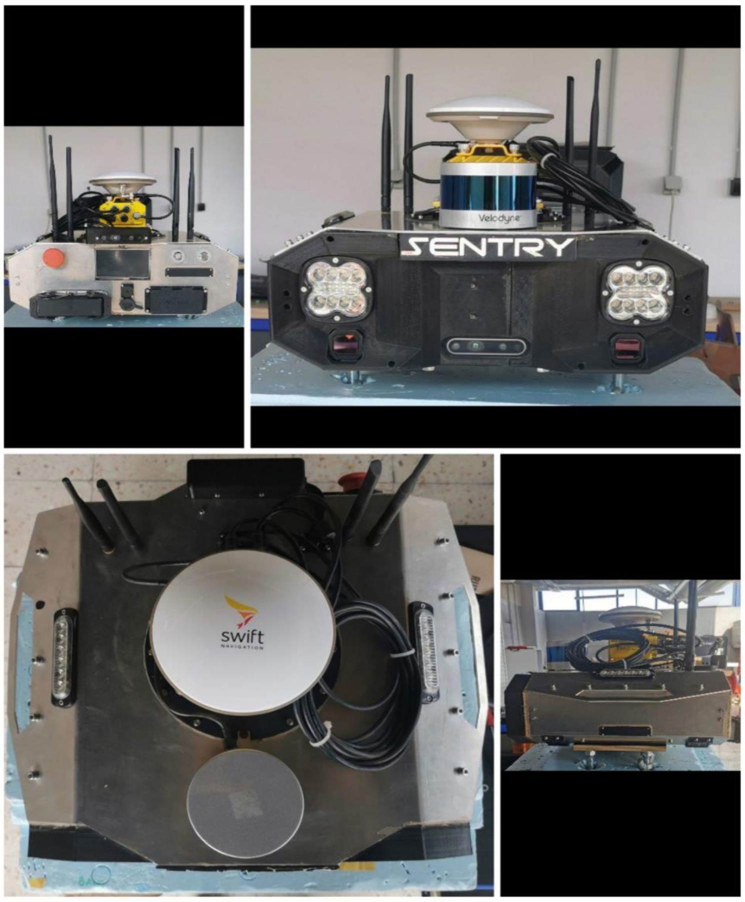


Figure 40 - Sentry

Choosing the components is separate from this work since it was already done, and only a short description of the materials is done. This chapter will focus instead on software development as calibration and noise reduction, other than a presentation of the packages utilised.

Section 4.1 Sentry's components

This section will present the electronic components' main features and the software used or developed.

Subsection 4.1.1 Arduino Mega 2560 Rev3

The Arduino Mega 2560 is a microcontroller board that uses the ATmega2560 chip. It has 54 pins that can be used for digital input/output (15 of which can be used for PWM outputs), 16 inputs for analogue signals, four hardware serial ports for communication, a 16 MHz crystal oscillator,

Figure 41 - Arduino Mega 2560 Rev3

a USB port, a power jack, an ICSP header, and a reset button *[17]*.

This component interfaces the ROS environment and the machine's actuator, transforming the ROS message in voltage.

The connection between the host and the device is established by utilising the *rosserial* meta-package, which is part of the ROS libraries. *rosserial* is designed as a protocol that encapsulates standard ROS serialised messages, allowing multiple topics and services to be multiplexed over a character device, such as a serial port or network socket.

The package *rosserial_python* establishes a connection between the host and a rosserial-enabled device using Python. It encompasses a Python implementation that simplifies the setup, publishing, and subscribing process, enabling seamless communication with the connected device *[18]*.

Once the connection is established, the code built in Arduino transforms the message from ROS into voltage for the machine's actuator. It turns on/off the pin connected to the machine; here, only the management of the pin levels associated with the motion is analysed.

Firstly pure rotation and pure straight motion maximum and minimum levels are defined using teleoperation since, as mentioned in Chapter 2, these values must be tuned as a function of the field and of the computation speed of the sensors (taking care that the maximum level of voltage allows by the robot is 255). The values in *Table 4* are the ones chosen in the test field :

	Linear	**Angular**
Minimum	70	85
Maximum	100	100

Table 4 - Arduino values

Defined these values, the correlation between the ROS message and level is done by linear interpolation, imposing as minimum and maximum values of the ROS message value 0.1 and 1.

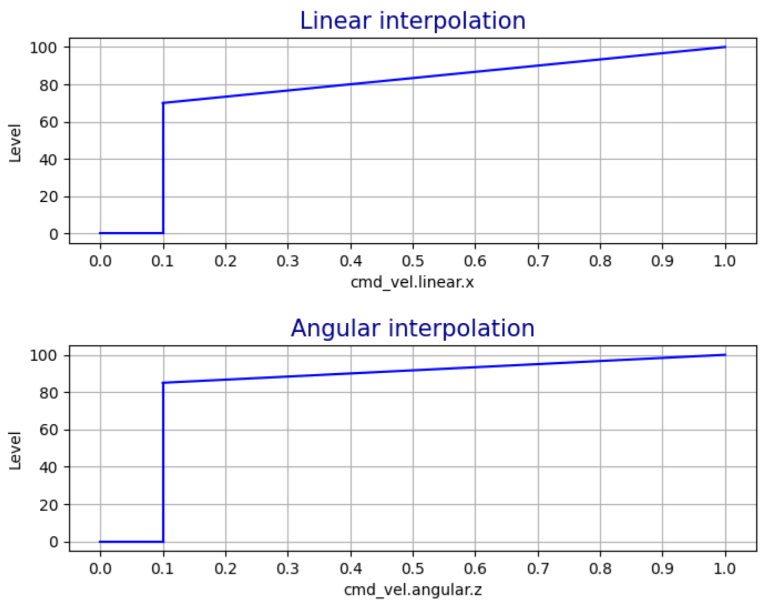


Figure 42 - Arduino's values

Three versions of code are presented; the first is related to a more straightforward solution in which the two types of motion are divided, while the second also allows a combination of the two. The last one used a non-linear function to improve the control of the wheels.

- *Version 1*: it is a solution easy to implement and allows discontinuous motion (linear or angular). It is used for the first motion algorithm (Bang-Bang controller) defined in the *Chapter 5*. Going into the details, the straight motion is defined as simply making both wheels go front, rotating proportionally with the command received; the rotation state is a reproduction of the actual behaviour of the machine (the one obtained using a remote control to move the robot); in particular the rotation state is implemented making the two wheels rotating with the same speed but in the opposite direction (using a different sign for the voltage produced).

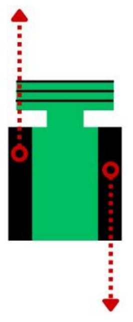


Figure 43 - Pure rotation (V1)

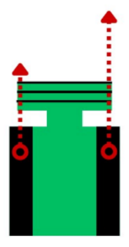


Figure 44 - Combined motion (V2 and V3)

- *Version 2*: It is based on the previous version, but it adds the possibility to combine the linear and angular motions; it is done by imposing on the two wheels different voltages but with the same sign.

The equation that regulates the transformation is shown below:

$$x = \frac{Max_{lin_{level}} - min_{lin_{level}}}{0.9} \cdot actual_{lin_{command}} + Max_{lin_{level}} - \frac{Max_{lin_{level}} - min_{lin_{level}}}{0.9}$$

- *Version 3*: It improves the previous version thanks to implementing a non-linear function.

  The functions shown below are for the forward-left motion:

$$Vel\_right\_wheel = int[x \cdot (1 + actual_{angle_{command}})]$$

$$Vel\_left\_wheel = int(x)$$

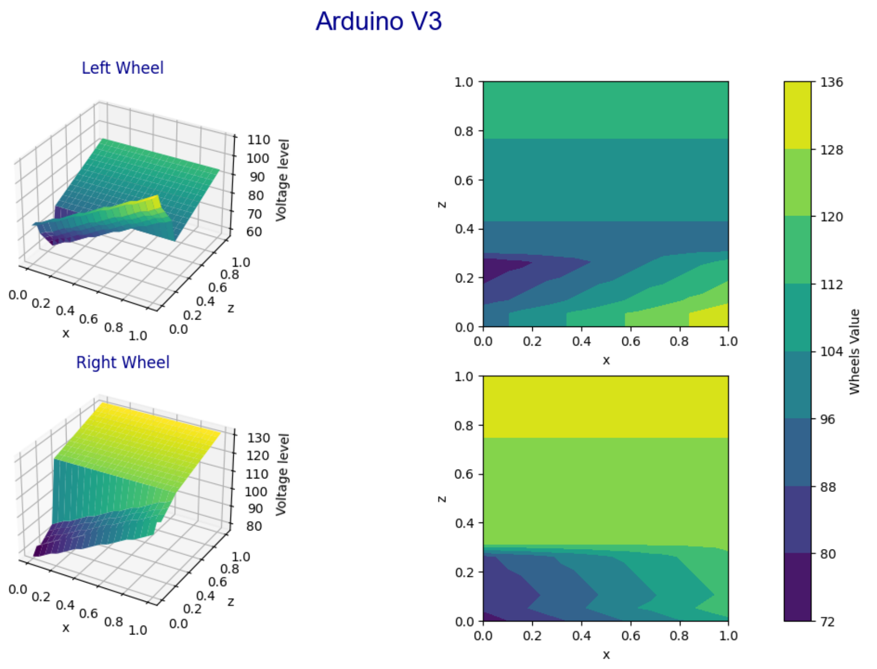


Figure 45 - Version 3 Arduino non-linear function

Subsection 4.1.2 Nvidia Jetson Xavier Nx

The NVIDIA® Jetson Xavier™ NX is a compact system-on-module (SOM) with high-performance computing capabilities at the edge. With the accelerated computing power of up to 21 TOPS, it enables the execution of contemporary neural networks in parallel and facilitates data processing from numerous high-resolution sensors. This capability is essential for the operation of comprehensive AI systems. The Jetson Xavier NX is particularly well-suited for

Figure 46 - Nvidia Jetson Xavier Nx

demanding AI applications, including commercial robots, medical instruments, smart cameras, high-resolution sensors, automated optical inspection, smart factories, and other embedded systems within the IoT domain *[19].*

Subsection 4.1.3 Duro

Duro is an enclosed version of the Piksi® Multi dual-frequency RTK GNSS receiver designed for outdoor use. It combines exact positioning accurate to the centimetre level with robust military-grade durability.

It features a comprehensive sensor suite with 9 degrees of freedom (DOF). Six DOF are given by the IMU (Inertial Measurement Unit), which consists of an accelerometer and gyroscope, enabling precise motion sensing and tracking. Then a magnetometer is included; it further enhances Duro's capabilities by allowing it to measure the Earth's magnetic field, facilitating navigation and accurate heading estimation.

Furthermore, Duro incorporates a GNSS antenna, enabling it to receive signals from multiple global navigation satellite systems. With the integration of an RTK (Real-Time Kinematic) system, Duro has the potential to achieve high-precision positioning by utilising carrier-phase measurements and correction data.

Overall, Duro offers a comprehensive outdoor positioning and navigation solution, combining centimetre-level accuracy, rugged construction, and advanced sensor capabilities *[20]*.

Since it comprises several sensors, each part will be analysed individually and how it relates to the other.

The first aspect is how it is connected to the ROS environment; this is done thanks to the ROS package *duro_ros [21]*, which publishes ROS topics related to the raw data the component sends.

This data represents a measurement of the angular velocity (gyroscope) and linear acceleration (accelerometer) other than the magnetic field measurement in the three axes.

Figure 47 - Duro starter kit

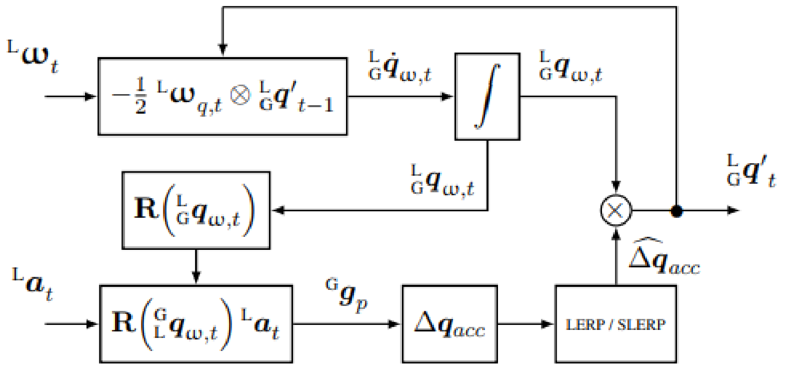
The first two data types (from the gyroscope and accelerometer) are sent to a complementary filter to get orientation as rotation in the x and y axes (pitch and roll).

4.1.3.1 Complementary filter

This algorithm is in a ROS meta-package *imu_tools [22]* as a ROS package called *imu_complementary_filter*. It is built on the work done by *[23].*

Resuming the paper, the choice of using the complementary filter instead of another filter, such as the Kalman Filter (*4.1.3.4*), is due to its simplicity and effectiveness. It uses analysis in the frequency domain to filter the signals and combine them to obtain an orientation estimation without any statistical description.

Its working principle can be resumed by the scheme below:



$^{L}\omega_t$
$-\frac{1}{2}\,^{L}\omega_{q,t} \otimes \,^{L}_{G}q'_{t-1}$
$^{L}_{G}\dot{q}_{\omega,t}$
$\int$
$^{L}_{G}q_{\omega,t}$
$^{L}_{G}q_{\omega,t}$
$^{L}_{G}q'_t$
$\mathbf{R}(^{L}_{G}q_{\omega,t})$
$\widehat{\Delta q}_{acc}$
$^{L}a_t$
$\mathbf{R}(^{G}_{L}q_{\omega,t})\,^{L}a_t$
$^{G}g_p$
Δq_{acc}
LERP / SLERP

Figure 48 - Complementary filter

As it is possible to see, a first orientation is taken by the gyroscope measurement; this is used to rotate the measure from the accelerometer obtaining a correction term. This term is filtered by a *Linear intERPolation* (LERP) if the correction term is below a decided threshold; if it is higher, a *Spherical Linear intERPolation* (SLERP) is used.

This filter allows the addition of another term of correction from the magnetometer orientation. Still, the choice done in this work is to handle it in a different node, to calibrate and filter it correctly, using the roll and pitch angle (from the complementary filter) to compensate for the tilt.

4.1.3.2 Magnetometer calibration

Continuing with the analysis of the magnetometer, special attention was given to this component due to its susceptibility to noise. As presented in *[24]*, the magnetometer data can be affected by various sources of disturbances, including wide-band measurement noise, stochastic biases, installation errors, and magnetic interferences in the vicinity of the sensors.

These magnetic interferences can be categorised into two groups.

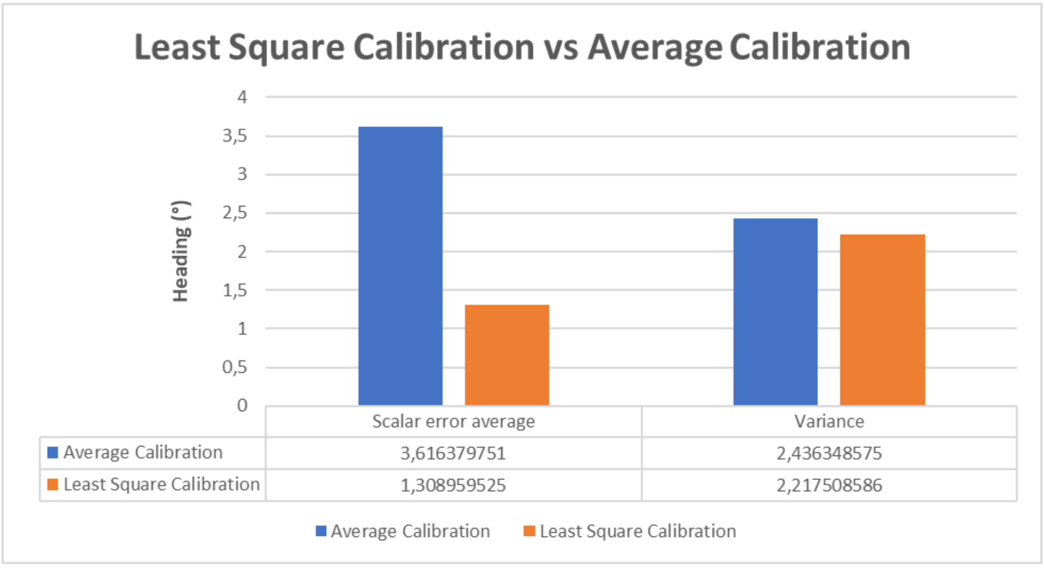
The first group, hard iron interference, refers to the presence of a fixed or slightly time-varying magnetic field generated by ferromagnetic materials.

The second group, soft iron interference, pertains to the magnetic field generated within the device.

Calibration is crucial for this component to get the 12 matrix parameters to compensate for the abovementioned effect. In this work, two different ways of calibration are presented:

The first is based on *[25]*, while the second is based on *[24]*.

A test is done to show the best approach between them; the result is shown in *Figure 49*.



	Scalar error average	Variance
🟦 Average Calibration	3,616379751	2,436348575
🟧 Least Square Calibration	1,308959525	2,217508586

Figure 49 - Calibration comparison

As it is possible to see, the *Least Square Calibration* performs better; furthermore, it is worth noting that this calibration required complete rotations around the three axes. Since this is impossible in the field, a symmetry of the points collected is done to complete the dataset on which the calibration procedure is applied.

4.1.3.3 Magnetometer filter

After the calibration, the magnetometer measurements are submitted to a filter action before getting the final heading value.

This work proposes a *Lowpass filter* and a *Moving Average Filter.*

Going deeper into why a filter action is needed and which type of noise is present, it is said that the magnetometer presents *pink or 1/f noise* at low frequencies and *white noise* at high frequencies *[26]*.

For this reason, the first and more standard approach of using a *Lowpass filter* is insufficient, while a *Moving Average Filter* can better suit the work expected.

Even if it has a straightforward implementation, this filer has an excellent response in the time domain, suiting for the application. At the same time, it cannot be applied to frequency domain applications *[27].*

Given the general characterisation of the noise found in state of the art and to use a more practical approach, both filters are implemented and compared to find the best method for our goal.

The implementation of the two filters is summarised below:

Firstly, the data collection object is created; for both filters, a *deque* is used to collect the data in chronological order fixing its dimension.

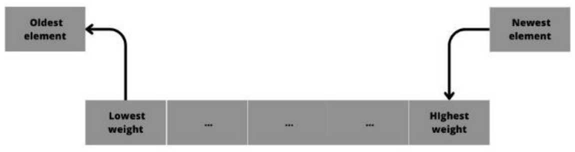


Figure 50 - EMAF deque

For the *EMAF* (Moving Exponential Average Filter) in each position, an exponential weight proportionally to its position is associated with the data collected, obtaining for each iteration the exponentially weighted average of the element inside; the choice to not use a simple average is made to give more priority to recent data then to dynamics instead of noise reduction as reported in *[27]*.

For the *Low pass*, the measurements stored are used in the equation of a digital Low Pass filter:

$$a_0\, y[n] = b_0\, x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2]$$

In which *y[n]* is the output, *x[n-k]* is the past inputs and *y[n-k]* is the past output *[28]*.

Where the *a* and *b* coefficients are computed by the function *scipy.signal.iirfilter* of the library *Scipy* given as input the order of the filter, the cut-off frequency, the sampling frequency, the type of filter (bandpass, lowpass, highpass or bandstop, and Butterworth, Chebyshev I or II, elliptic or Bessel).

A *Butterworth* type is chosen since it has a frequency response as flat as possible in the pass band region with a sampling frequency of 20 given by the sensor one.

A first analysis was done for both sensors to tune the parameter that satisfies stationary and dynamic requirements; an analysis example is shown below in *Figure 51 and Figure 52.*

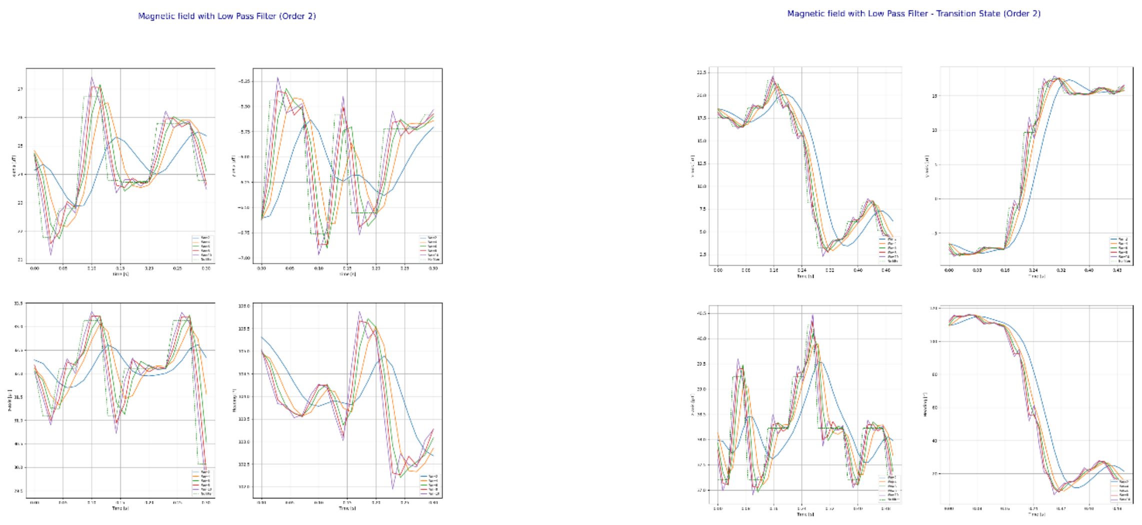


Figure 51 - Example of analysis for LP order 2 stationary(left) transition (right)

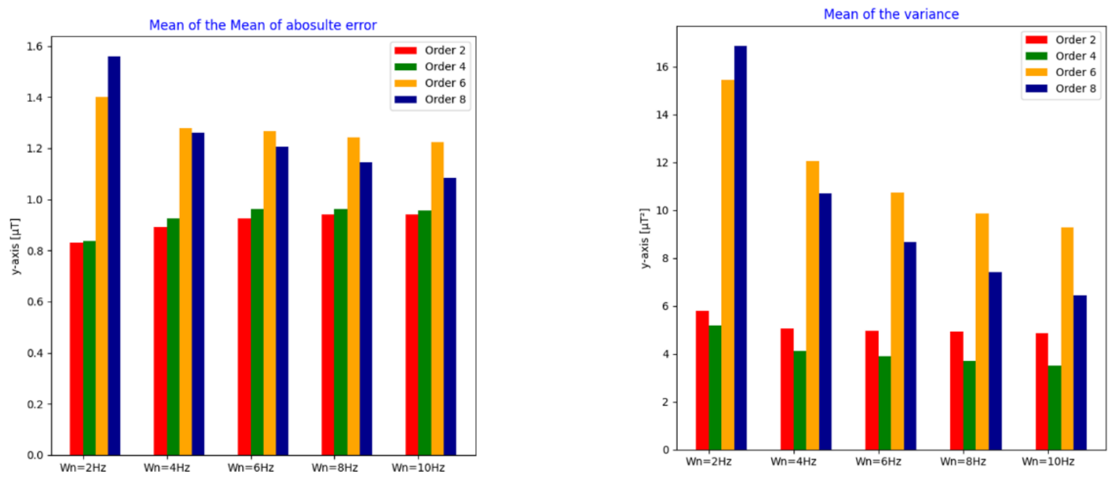


Figure 52 - Error analysis for LP

The final values chosen for the two filters are reported in *Table 5*:

Low Pass	**EMAF**
ω_C=4 Hz	Size =10
Order=2	-

Table 5 - Filters parameters chosen

A further comparison was made between the two, showing that *EMAF* presents better results for the application.

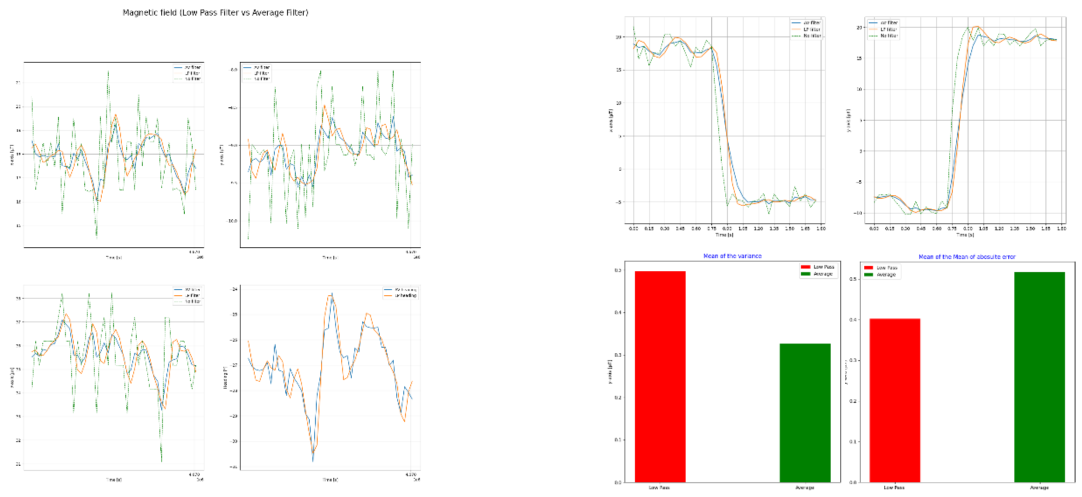


Figure 53 - Comparison LP / EMAF

4.1.3.4 Extended Kalman Filter

To gain an accurate state estimation, the fusion of data from multiple sensors is required. To address this need, the package *robot_localization* is used *[29]*. As described in *[30]*, this package contains a generalised *extended Kalman filter* (EKF) implementation. The package has several advantages, including supporting unlimited inputs from various sensor types, customising which sensor data fields are fused with the current state estimate, supporting 3D state estimation, and handling multiple ROS message types.

The central node of the package is the *ekf_localization_node*, which implements an extended Kalman filter algorithm. The implementation allows prediction and correction steps, projecting the state estimate and error covariance forward in time and updating them based on sensor measurements. The implementation supports partial updates of the state vector, enabling the fusion of sensor data that measure only a subset of the variables. Users can customise the process noise covariance for optimal performance.

To better understand how it works, the article continues to describe two experiments to evaluate the performance of the *ekf_localization_node*.

One was interested in the distance between the robot's start and end positions, and another was interested in the behaviour with infrequent GPS; for both, different configurations were utilised: dead reckoning via the platform's odometry, fused odometry with a single IMU, fused odometry with two IMUs; fused odometry with two IMUs and a single GPS, and fused odometry with two IMUs and two GPS units.

The result of the first experiment is shown in *Figure 54* taken by the article.

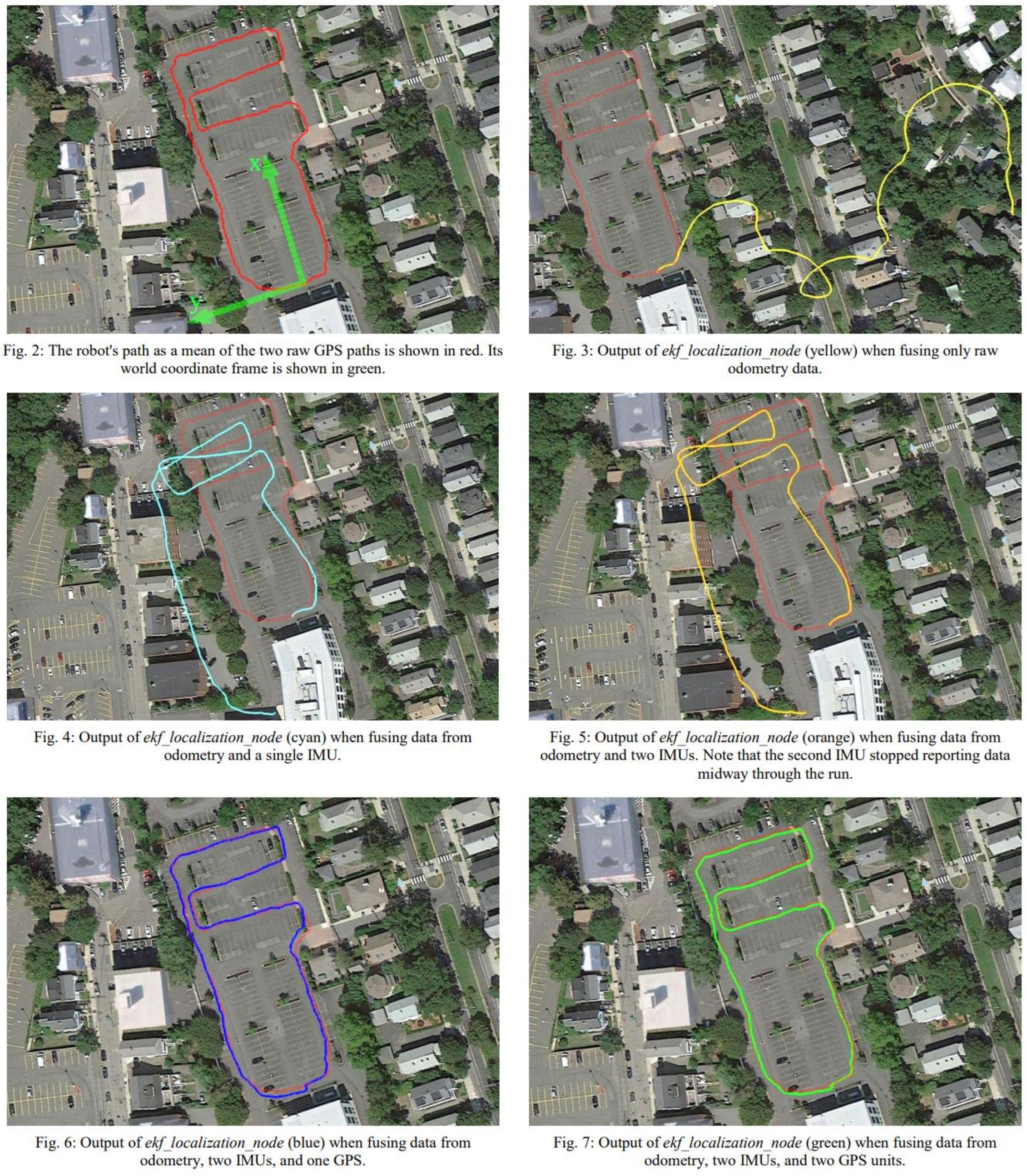


Fig. 2: The robot's path as a mean of the two raw GPS paths is shown in red. Its world coordinate frame is shown in green.

Fig. 3: Output of *ekf_localization_node* (yellow) when fusing only raw odometry data.

Fig. 4: Output of *ekf_localization_node* (cyan) when fusing data from odometry and a single IMU.

Fig. 5: Output of *ekf_localization_node* (orange) when fusing data from odometry and two IMUs. Note that the second IMU stopped reporting data midway through the run.

Fig. 6: Output of *ekf_localization_node* (blue) when fusing data from odometry, two IMUs, and one GPS.

Fig. 7: Output of *ekf_localization_node* (green) when fusing data from odometry, two IMUs, and two GPS units.

Figure 54 - Localisation comparison [30]

In contrast, the second experiment shows as being between the current state and measurement despite the significant difference between state estimate and measurement due to the infrequency of the GPS; the filter's covariance matrix retains its stability.

Other than describing the potentiality of this package, this article has much interest since it shows how the configuration of a redundant sensor helps to reduce the error.

In the thesis configuration, Imu and a GPS+RTK are used. Relying on an odometry measurement using encoders or visual odometry by the camera or the LiDAR can also increase the precision in the condition of low GPS reception.

Subsection 4.1.4 Lidar Velodyne VLP16

The Velodyne Puck is a sensor that uses 3D lidar technology. It is known for its reliability, power efficiency, and ability to provide a surround view. This makes it an excellent choice for affordable low-speed autonomy and driver assistance applications *[31].*

Figure 55 - Velodyne VLP16

Lidar stands for Light Detection and Ranging. It uses laser beams to measure distances and create a detailed 3D map of the environment. In this work, it will be crucial to detect obstacles thanks to an adaptative clustering algorithm.

In particular, this sensor is connected to the ROS environment thanks to the ROS package *velodyne [32]*. It publishes a point cloud that is the input for an algorithm of clustering. This algorithm is also available in the ROS community in *[33]*, based on the work in *[34]*.

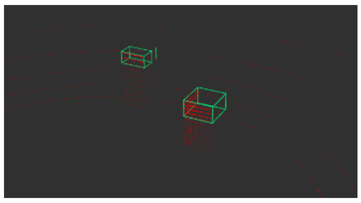


Figure 56 - Example of clustering

Thanks to this algorithm, boxes surrounding the element are published once the minimum and maximum height from the Velodyne, the minimum and maximum size, and the distance are tuned.

Subsection 4.1.5 Intel® RealSense™ Depth Camera D435

The stereo Intel RealSenseTM depth camera D435 provides high-quality depth for various applications. In fields such as robotics or augmented and virtual reality, having a broad view of the scene is essential. Therefore, this technology's wide field of view is perfect for such applications.

Figure 57 - Camera

This small form factor camera can be easily integrated into any solution and has a range of up to 10 metres. It also includes Intel RealSense SDK 2.0 and cross-platform support *[35].*

In the thesis application, it is used for people detection during navigation.

The visual cameras are connected to the ROS environment through a node (a code written in Python language). This node uses an algorithm called *YOLO* (You Only Look Once) to identify the feature in the space around it.

As reported in *[36]*, *YOLOv5* is one of the most recent and frequently used iterations of a well-known deep learning neural network; it is utilised for various machine learning applications, mainly in computer vision. Due to the *YOLO* algorithm's excellent performance in complicated and noisy data contexts, availability, and simplicity of usage in conjunction with popular programming languages like Python, it has rapidly gained favour in the data science community.

The network architecture of *Yolov5* consists of three parts: *CSPDarknet* (Backbone), *PANet* (Neck) and *Yolo Layer* (Head).

The data are first input to *CSPDarknet* for feature extraction and then fed to *PANet* for feature fusion. Finally, the *Yolo Layer* outputs detection results (class, score, location, size) *[37].*

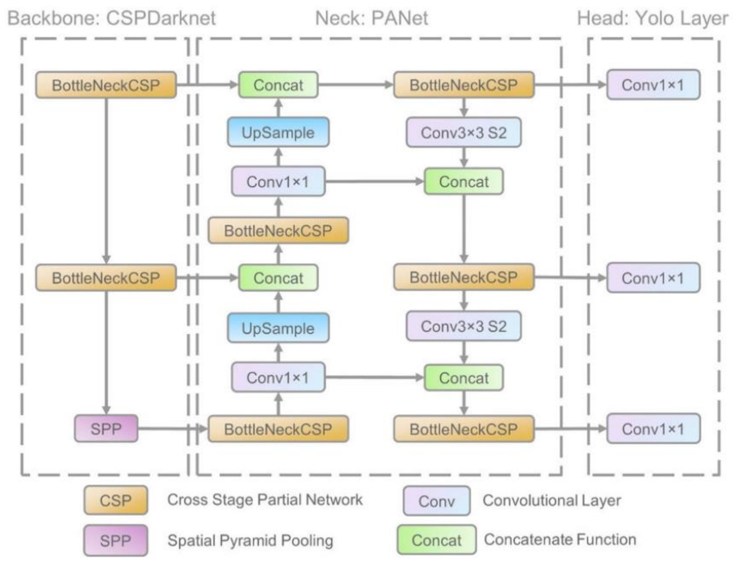


Figure 58 - YOLO structure

This system uses a convolutional neural network as its backbone to collect and organise image features at various levels. The neck network then merges these features and prepares them for prediction. Finally, the head network predicts targets on the feature maps *[37].*

Given a brief definition of what *YOLO* is, the libraries/modules used are presented:

- *Torch* is a well-liked machine learning library that is open-source and mainly used for deep learning tasks. To create and train neural networks, it offers a high-level interface. The *PyTorch* library's utility module *torch.hub*, which offers a practical method to load previously trained models, is used. To expedite the process, the model has already been downloaded locally.
- *OpenCV*, also known as *CV2*, is a widely-used open-source library that offers a variety of image processing and computer vision features.
- *Pyrealsense2* is a tool that allows developers to use Intel RealSense depth cameras in Python applications. It provides a straightforward and user-friendly interface for operating RealSense cameras, including taking colour and depth frames, getting sensor information, and carrying out various processing operations.

- *utils.dataloaders* is a module of a library download by *[38]* to manage the loading and handling of different data types.
- The other libraries *rospy*, *numpy*, and *std_msgs* are commonly used for this type of work.

Finally, a schematic presentation of the node is done:

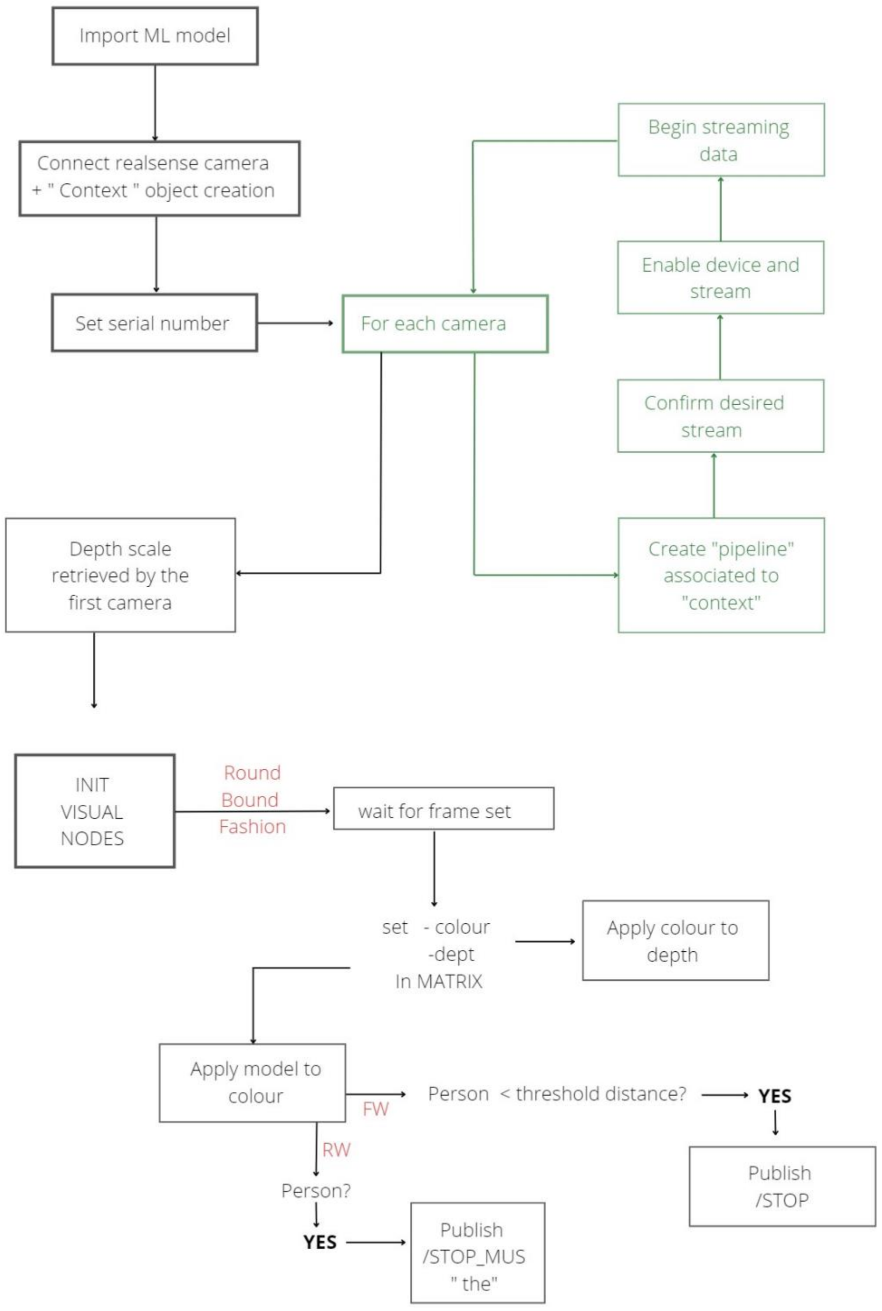


Figure 59 - Person detection node

Subsection 4.1.6 FLIR ADK

Thermal infrared cameras are the best sensor technology for detecting people and animals, day or night. It measures the amount of infrared radiation emitted or reflected by objects *[39].*

Figure 60 - FLIR ADK

This sensor is not yet used but will be necessary for subsequent developments thanks to the possibility of integration with an optical camera and lidar, achieving better results and reducing noise for low light or dust.

Chapter 5 Motion Planning and Control

This chapter is divided into two parts; the first will discuss the different solutions implemented to achieve outdoor navigation between GPS points. The second part will instead add the obstacle avoidance problem, increasing the problem's complexity. Before entering the details, a global scheme for the development is shown in *Figure 61*:

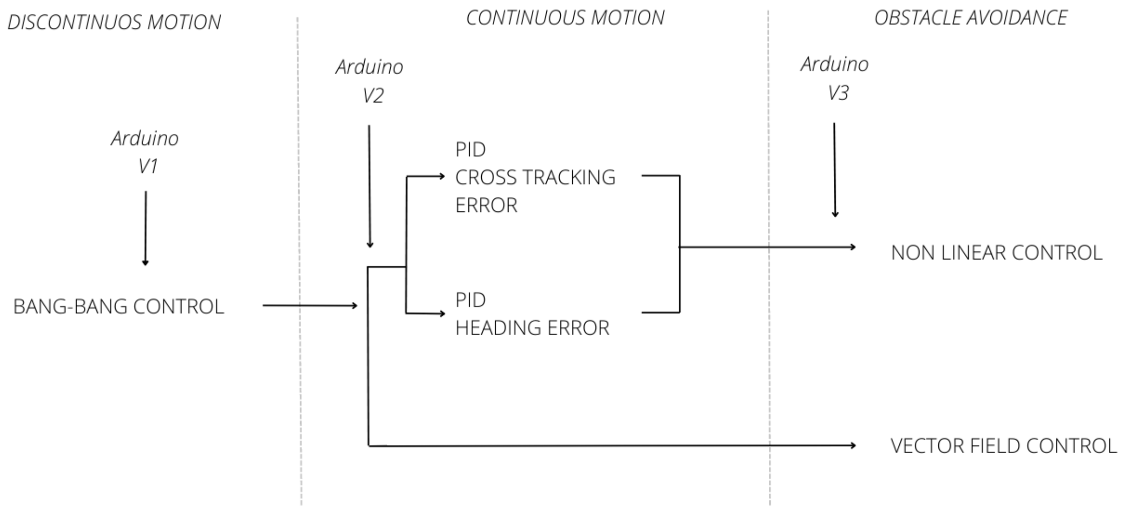


Figure 61 - Motion development

Section 5.1 Navigation Part

All the algorithms that are proposed have the goal of making the robot reach a specific point (sent by the user through the company's API), minimising the distance from the line that joins the start to the goal; they also share the ability to stop if a person is detected and to close the program other than control the robot if an emergency message is sent.

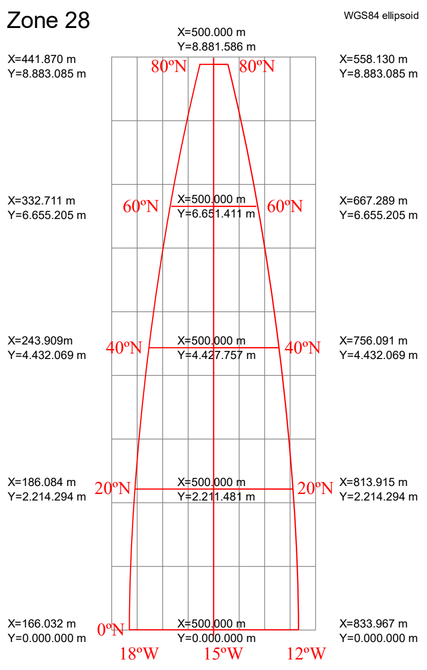

Zone 28
WGS84 ellipsoid
X=500.000 m Y=8.881.586 m
X=441.870 m Y=8.883.085 m
X=558.130 m Y=8.883.085 m
80ºN
X=332.711 m Y=6.655.205 m
X=500.000 m Y=6.651.411 m
X=667.289 m Y=6.655.205 m
60ºN
X=243.909m Y=4.432.069 m
X=500.000 m Y=4.427.757 m
X=756.091 m Y=4.432.069 m
40ºN
X=186.084 m Y=2.214.294 m
X=500.000 m Y=2.211.481 m
X=813.915 m Y=2.214.294 m
20ºN
X=166.032 m Y=0.000.000 m
X=500.000 m Y=0.000.000 m
X=833.967 m Y=0.000.000 m
0ºN
18ºW
15ºW
12ºW

Figure 62 - Example of UTM transformation

Before starting the navigation, several nodes must be run, in particular:

- *duro_node* is run to get raw data from the IMU, the GPS and the magnetometer.
- *imu_complementary_filter* is run to get orientation for the *imu_tools* package.

- *utm_odometry_node* converts latitude-longitude readings into UTM odometry (available in the *gps_common* package). The UTM (Universal Transverse Mercator) coordinate node system is a positioning system which divides the world into sixty north-south zones, each 6 degrees of longitude wide, making in the plane and then adding the possibility to configure a xy Reference frame.
- *robot_pose_ekf* from the package *robot_pose_ekf* receives the UTM coordinate and fuse them to obtain the robot's odometry. Finally, the coordinate in the XY frame is received by the algorithm that performs the navigation.
- Other than that, the node based on the *YOLO5* algorithm must be run to perform people detection, as the node related to the filtering of the magnetic field to obtain the heading of the robot and the node *adaptative_clustering* from the package *adaptative_clusterig* to get the position of the obstacle.

Subsection 5.1.1 Bang-bang controller

A bang-bang controller is a feedback controller in control theory that rapidly switches between two states *[40].*

The two states are the forward motion state and the rotation state. This controller is allowed by the first version of Arduino code, getting a first knowledge of ROS commands and Arduino's programming.

The base concept of the motion algorithm is that, given a starting point and a goal, firstly is performed a rotation to be aligned to the goal, and then the forward motion is made. To correct motion and sensor noise, two imaginary boundaries are drawn, and as soon as the robot overcomes one of them, the rotation is performed again, realigning to the goal. The functioning is schematised below:

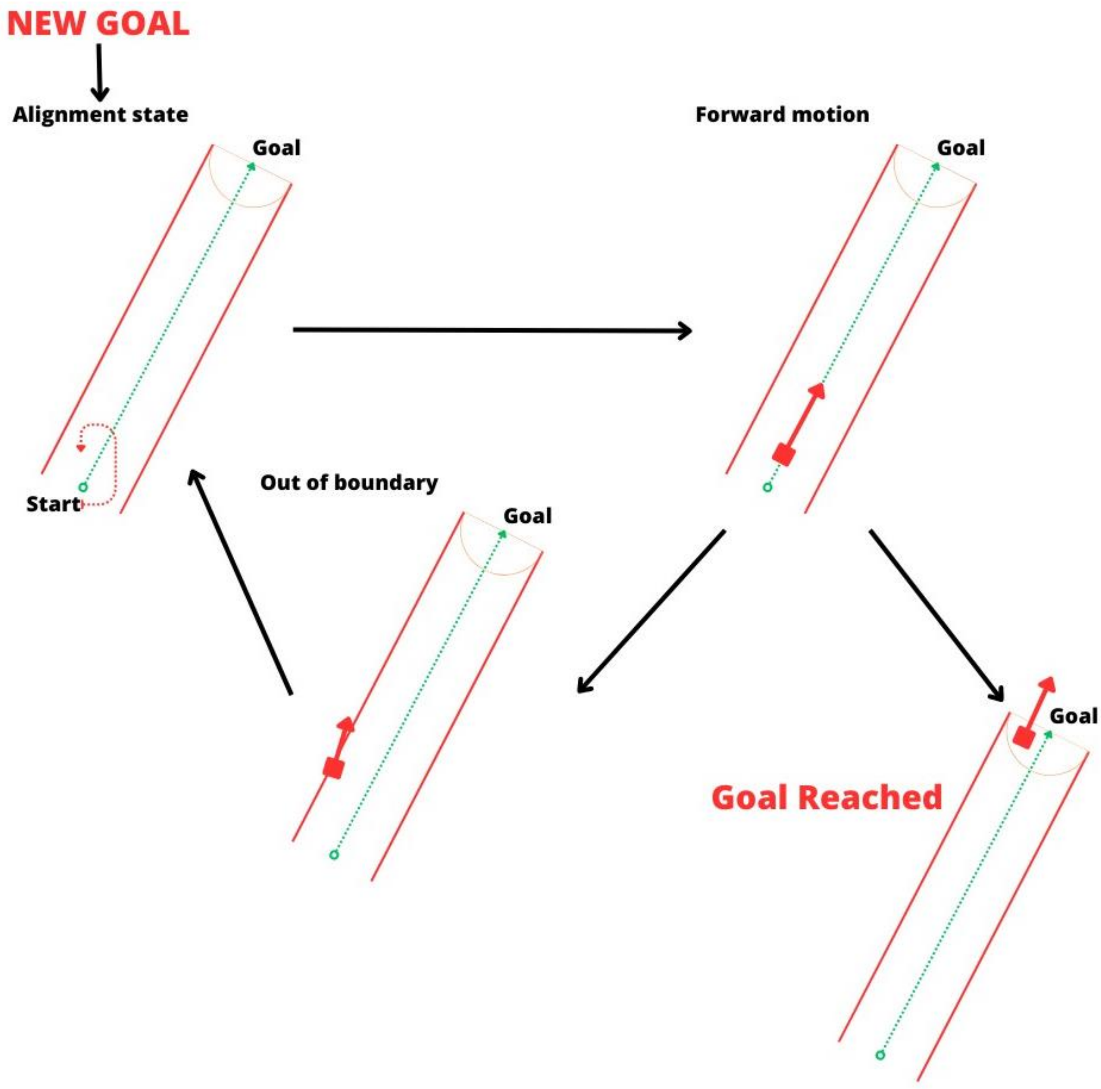


Figure 63 - Bang-Bang algorithm

This algorithm guarantees the reach of the goal as the tolerance used for the boundaries is less or equal to the tolerance used to check if the robot reaches the destination. By reducing the boundaries' tolerance, higher precision can be acquired at the cost of a more discontinuous path.

Once the path creation is explained, the motions' speed must be analysed since both angular and linear are not constant. Indeed, the angular velocity is proportional to the angular rotation that must be done. Still, the growth is limited by a function that imposes a max increment (0.2) and a maximum value (1 by default).

For the linear, a trapezoidal profile is proportional to the distance between the starting point and the goal.

In this way, velocities present a smoother profile avoiding step variation.

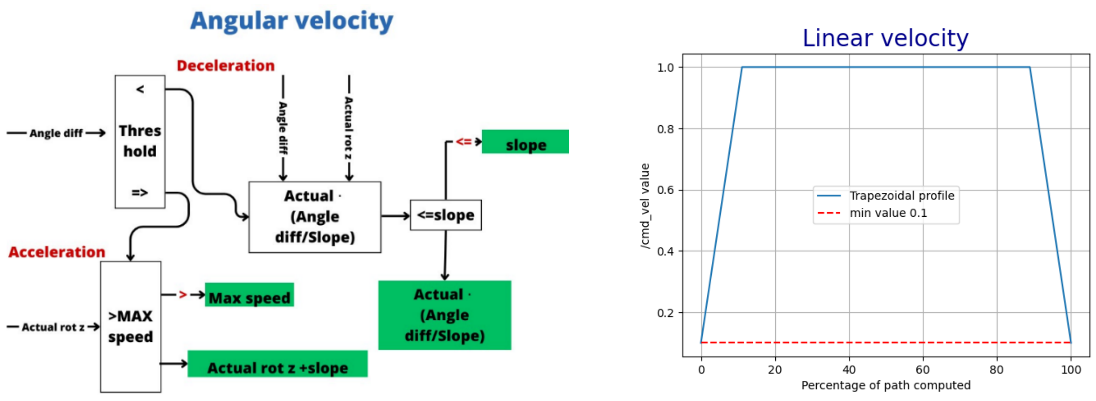


Figure 64 - Velocities profiles

Finally, a resume of the code is provided below:

Initialisation:

- Imports the necessary modules and packages.
- Defines the *Calc_Trajetory* class, which represents the task structure.
- Initializes the ROS node and sets up the necessary publishers and subscribers.
- Implements callback functions for receiving data from subscribed topics, such as heading information, actual coordinates, emergency stop signal, and stop command (people detection).
- Implements a *mover* method that inputs a goal position, a logging object, and a file object and controls the movement of the robot to reach the goal position; in the meantime, it provides the possibility to gain a *log* file for debugging and a *txt file* to get in a matrix shape way information to make analyses and graphs.

For each goal, the *mover* is called:

- It starts by initialising some variables and publishing zero velocities to stop the robot as safe.
- It enters a loop until the goal position is reached.
- Within the loop, it checks for emergency and people detection stops. If it can continue, two states are possible: rotation and straight-line movement.
- In the rotation state, the robot adjusts its heading to align with the desired angle.
- If the alignment is not achieved within a specified time, it transitions to the forward motion. In this way, the robot does not get stuck.

- In the straight-line movement state, the robot moves towards the goal position.
- The calculated velocities are published to the */cmd_vel* topic to control the robot's motion.

In the __*main*__ block:

- Initializes a logging object and opens a file to write the trajectory information.
- Retrieves the goal points from a file.
- Creates an instance of the *Calc_Trajetory* class.
- Executes the mover method for each goal point, controlling the robot's movement.
- Handles exceptions and gracefully exits the program.

Subsection 5.1.2 PID controller

This section will present two different versions of the PID controller. In particular, both versions share the same path planner and the linear PID controller, while they differ for the angular PID.

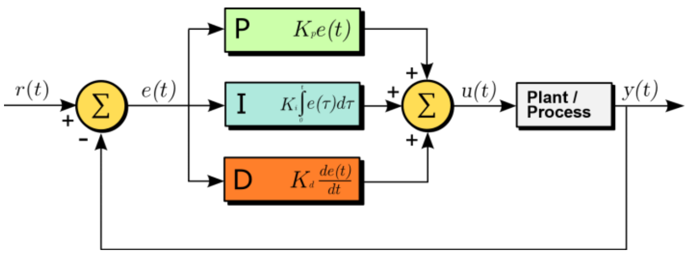


Figure 65 - PID scheme

Starting from the general concept, a proportional–integral–derivative controller (PID controller) is a control loop mechanism employing feedback, widely used in the scientific field. The PID controller regularly computes the difference between a desired setpoint and a measured value, called the error value *e(t)*. It then uses proportional, integral, and derivative terms to make necessary corrections *[41].*

It is then essential to the way this error is computed. Instead, different quantities can be utilised; here are the three primary measurements used in this scenario:

- *Distance to the goal* is the Euclidean distance between the desired goal and the actual position.
- *Heading error* is the difference between the actual heading of the robot and the heading needed to reach it.
- *Cross-track error (CTE)* is the orthogonal distance between the line of desired travel and the robot's location.

It is possible to see as the first one doesn't give information about the orientation, while the last two provide only the orientation term. For this reason, the first error will be used as a reference for the linear motion PID while the others will be used for the angular one.

As regards the path planning part, in both cases, the track is divided into the equidistant point that changes as the tractor gets near to them, working as a reference for the linear PID; that is done to obtain an almost constant value during the travelling reducing it just when it is arriving.

5.1.2.1 Heading PID

As written before, this algorithm uses two PIDs to control the linear and the angular speed separately; at the same time, it continues to use the central concept of the *Bang-Bang* controller; indeed, as it receives the actual goal, it aligns the robot to the destination, reducing the angular error of some degrees (around 4°); in this way the angular PID has to compensate for the small angular error, obtaining a smoother behaviour.

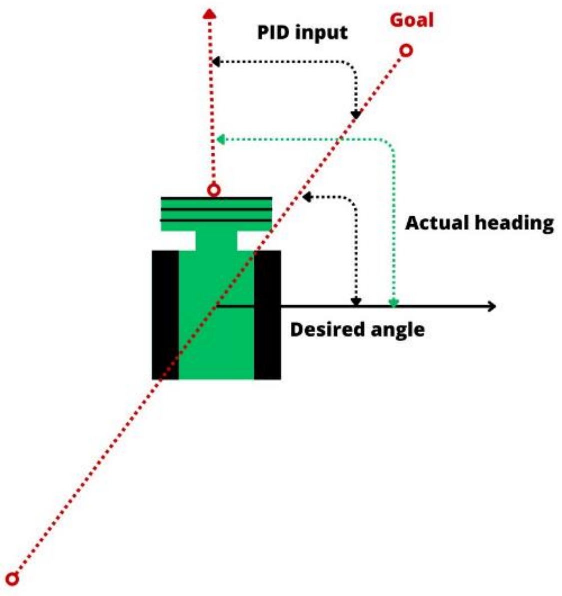


Figure 66 - Heading PID

So as the robot starts the forward motion, the angular error is continuously computed and given to the PID, which has a 0 has a set point, to transform it in the velocity command component (*/cmd_vel.angular.z*).

Furthermore, since for external disturbance like the loss of the RTK, the robot can go too further from the goal, starting a circular motion around it (the linear velocity is proportional to the distance), a linear limiter is added to regulate the final speed inversely proportional with the angular one. This limiter is also used in the other controllers to avoid going for several meters without the correct heading.

Also in this case, a resume of the code developer is provided:

Initialisation:

- The necessary libraries and modules are imported.
- PID controllers class are defined for linear and angular motion.
- The necessary publishers and subscribers are set up.
- Callback functions are defined to receive and process data from various topics, such as heading, coordinates, and stop signals.

Movement Execution:

- It receives the goal coordinates, a log object for debugging information, and a file object to store navigation data.
- The function starts by initialising variables and setting the initial velocities to zero (safety measurement).
- It enters a loop until the goal is reached.
- During each loop iteration, the function checks for stop signals and handles them accordingly.
- If the robot can move, it checks if it needs to perform an initial rotation to align with the desired heading.
- To align the robot with the desired heading, it may require an initial rotation that adjusts its angular velocity. If the alignment is not achieved within a set time, the robot will move on to the next step to prevent it from getting stuck.·
- Once aligned, the function computes the reference point for the robot to follow using the *path_creator* function. This function creates a trajectory path based on the current position and the desired final goal. The function returns the new reference point for the robot.
- So, it computes the error between the current and desired heading and uses the PID controllers to calculate the angular and linear velocities based on the error values.
- The linear velocity is limited using the *velocity_acceleration_limiter* function. In this way, if a strong rotation is needed, the line velocities are reduced to avoid a considerable radius trajectory.
- The calculated velocities are published to the */cmd_vel* topic to control the robot's motion.
- Once the goal is reached, the function stops the robot, sets the initial rotation flag, and prepares for the next goal.

In the *__main__* block:

- Initializes a logging object and opens a file to write the trajectory information.
- Retrieves the goal points from a file.
- Creates an instance of the *Calc_Trajetory* class.
- Executes the mover method for each goal point, controlling the robot's movement.
- Handles exceptions and gracefully exits the program.

5.1.2.2 CTE PID

This control scheme is similar to the previous one, but the angular speed is based on the distance from the path. The controller will apply angular rotation to keep the space from the line as small as possible.

In this way, the angular controller is able only to keep the robot on the line, but if the goal is passed outside the tolerance, it will not turn the robot to the back; to solve this problem, an imaginary rectangle is drawn surrounding all paths; in this way, if the robot touches this boundary a realignment is performed, creating a new route straight forward to the goal.

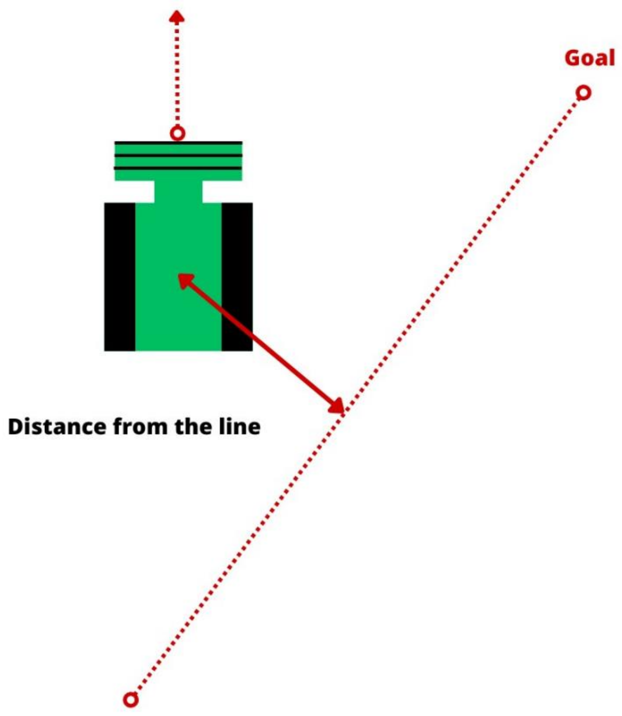


Figure 67 - CTE

As shown in the *Chapter 7* , this controller has better results concerning the *Heading PID* thanks to the higher precision of the localisation concerning the heading computation. Anyway, this controller is not suitable for obstacle avoidance application since in a not straight path (like in the case of the presence of an obstacle), the distance requires time to get bigger, differently from the heading that pointing to the following reference will show instantly the needed of a strong rotation.

Finally, a resume of the code developed is provided:

Initialisation:

- The necessary libraries and modules are imported.
- PID controllers class are defined for linear and angular motion.
- The necessary publishers and subscribers are set up.
- Callback functions are defined to receive and process data from various topics, such as heading, coordinates, and stop signals.

Movement Execution:

- It receives the goal coordinates, a *log object* for debugging information, and a *file object* for stored data.
- The function starts by initialising variables and setting the initial velocities to zero.

- It enters a loop until the goal is reached.
- During each loop iteration, the function checks for stop signals and handles them accordingly.
- If the robot can move, it checks if it needs to perform an initial rotation to align with the desired heading.
- If an initial rotation is required, it adjusts the angular velocity of the robot to align it with the desired heading. As for the others, if the alignment is not achieved within a specified time, it transitions to the next step to avoid the robot getting stuck.
- Once aligned, the function computes the reference point for the robot to follow using the *path_creator* function. This function creates a trajectory path based on the current position and the desired final goal. The function returns the new reference point for the robot.
- Therefore, the error is computed between the current position and its projection, and it uses the PID controllers to calculate the angular and linear velocities based on the error values.
- The linear velocity is limited using the *velocity_acceleration_limiter* function. In this way, if a strong rotation is needed, the line velocities are reduced to avoid a considerable radius trajectory.
- The calculated velocities are published to the */cmd_vel* topic to control the robot's motion.
- Once the goal is reached, the function stops the robot, sets the initial rotation flag, and prepares for the next goal.

In the *__main__* block:

- Initializes a logging object and opens a file to write the trajectory information.
- Retrieves the goal points from a file.
- Creates an instance of the *Calc_Trajetory* class.
- Executes the mover method for each goal point, controlling the robot's movement.
- Handles exceptions and gracefully exits the program.

Subsection 5.1.3 Non-linear controller

This control strategy is based on the research presented in the article *[13]*, which is already discussed in the State of the Art (*Chapter 2*). In addition to being appropriate for obstacle avoidance paths, it uses heading and cross-track errors to ensure higher precision.

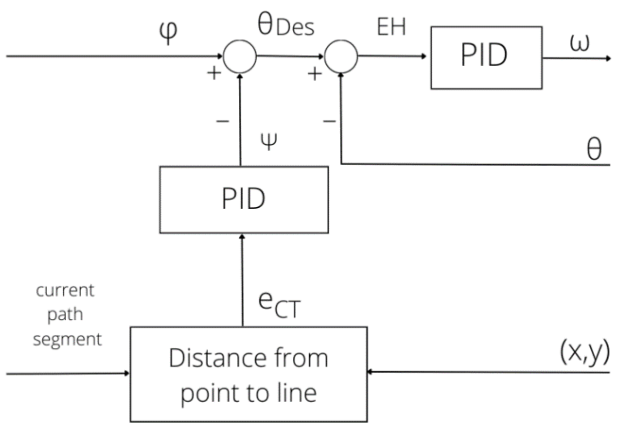


Figure 68 - Non-linear controller

As shown in the proposed scheme, the controller uses the error given by the cross-track to correct the heading error. Unlike the original plan, PIDs are used instead of just proportional controllers to obtain a fast response avoiding the oscillation given by the only use of the proportional term.

So, firstly, the correction term is computed based on the distance to the reference segment. After that, the correction angle is subtracted from the reference inclination of the path to obtain the input for the heading computation. The scheme of the angular velocities is shown in *Figure 68*.

A schematic representation of the code is proposed:

Initialisation:

- The necessary libraries and modules are imported.
- PID controllers class are defined for CTE and heading error.
- The necessary publishers and subscribers are set up.
- Callback functions are defined to receive and process data from various topics, such as heading, coordinates, and stop signals.

Movement Execution:

- The method mover initialises various variables and parameters for the movement control.
- It enters a while loop until the goal position is reached.
- During each loop iteration, the function checks for stop signals and handles them accordingly.
- If no stop signal is detected, the method checks if the robot is in the rotation state.

- If the robot is in the rotation state, it checks if the alignment with the goal position needs to be adjusted. If alignment is required and the time since the rotation start is less than or equal to a predefined interval, the method enters the rotation state (in this way, stuck conditions are checked).
- If alignment is not required or the time exceeds the specified threshold, the method ends the rotation state. So, the *path_creator* function creates the path for the robot to follow after rotation.
- If the goal hasn't been achieved, the method activates the *PID_motion* function, which utilises PID controllers to manage the robot's angular and linear velocities.
- The method first checks if the reference point needs to be updated. If the actual reference point is nearer than a tuned distance, it updates the *pid_reference* attribute and stores the previous reference as *last_reference* available, which will be used to get the reference segment.
- Indeed, the distance of the current position from the line formed by the last and current reference points is calculated.
- Depending on the orientation of the path and the robot's heading, the method determines the angular value to be used as a correction for the robot to go back on the path. The computation is done using a PID.
- The calculated angular value is then used to update the desired orientation by subtracting it from the absolute orientation of the path.
- To determine the input value for the PID controller, the method calculates the difference between the desired orientation and the robot's current heading. This results in the computation of the angular velocity component.
- For the linear one, the robot’s progress along the path is used as input of a trapezoidal profile.

In the __*main*__ block:

- Initializes a logging object and opens a file to write the trajectory information.
- Retrieves the goal points from a file.
- Creates an instance of the *Calc_Trajetory* class.
- Executes the mover method for each goal point, controlling the robot's movement.
- Handles exceptions and gracefully exits the program.

Subsection 5.1.4 Vector field

This method has a different implementation from the previous ones; indeed, a vector field is created instead of computing a series of points to be used as a reference, according to *[14]*. So firstly, a constant vector is created and fixed to the object; this will constantly push the vector through the direction of the path. Then, a vector field is built along the way to make the vector remain in the way. Accordingly, with the article, this field is divided into two parts, the nearer to the path is smooth behaviour, while the outsider has a step force as shown in the picture, *Figure 69*.

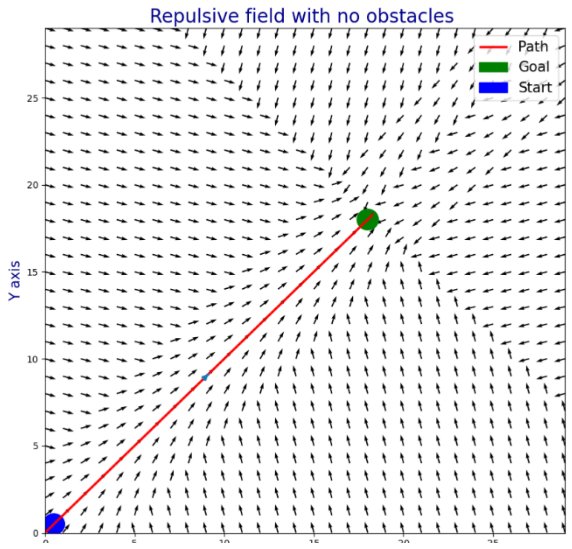


Figure 69 - Path vector field

A schematic representation of the code is proposed:

Initialisation:

- The necessary libraries and modules are imported.
- PID controllers class are defined for the heading error.
- The necessary publishers and subscribers are set up.
- Callback functions are defined to receive and process data from various topics, such as heading, coordinates, and stop signals.

Movement Execution:

- The method mover initialises various variables and parameters for the movement control.
- It enters a while loop that continues until the goal position is reached.
- During each loop iteration, the function checks for stop signals and handles them accordingly.
- If no stop signal is detected, the method checks if the robot is in the rotation state.
- If the robot is in the rotation state, it checks if the alignment with the goal position needs to be adjusted. If alignment is required and the time since the rotation start is less than or equal to a predefined value, the method enters the rotation state (in this way, stuck conditions are checked).

- If alignment is not required or the time exceeds the specified threshold, the method ends the rotation state.
- If the goal still needs to be achieved, the *PID_motion* function is executed. This function uses PID control to manage the robot's angular and linear velocities.
- The *PID_motion* method is a part of the *Calc_Trajetory* class that controls the robot’s motion. The main characteristic different from the previous implementation is the function *lineMission*.
- Firstly, it first calculates the differences between the goal and original positions in the x and y axes and the Euclidean distance between the current and goal positions.
- If the distance to the goal is less than a tunable value, the function calculates the attractive force towards the goal position. Therefore the desired heading is computed.
- If not, the function continues computing the perpendicular distance from the current position to the line connecting the original and goal positions.
- If it is more significant than a threshold value, a strong vector is applied to point to the line.
- If not, the distance is used to compute the vector the point to the goal while maintaining the robot on the line
- Finally, it computes the error between the desired and actual heading and returns the error (input for the angular PID) and the progression.
- For the linear one, the robot’s progress along the path is used as input of a trapezoidal profile.

In the *__main__* block:

- Initializes a logging object and opens a file to write the trajectory information.
- Retrieves the goal points from a file.
- Creates an instance of the *Calc_Trajetory* class.
- Executes the mover method for each goal point, controlling the robot's movement.
- Handles exceptions and gracefully exits the program.

Section 5.2 Obstacle Avoidance

This section is related to the ability of the robot to avoid possible obstacles on the road. Three different methods were exploited: the first one was the simplest and had less computational requirement since it did not require a dynamic map. Indeed it performs a *BUG* logic algorithm, discussed in *Chapter 2*. The other two instead require the construction of a map where navigate.

Subsection 5.2.1 Bug algorithm

As previously mentioned, this algorithm does not require a map; some memory is used to localise the obstacle but is deleted as the goal is reached. As the adaptive clustering algorithm detects an obstacle, it is located in the space, and the actual path is checked; if it intersects the safe contour of the obstacle (using a safe distance), a new one is computed, as shown in *Figure 70*.

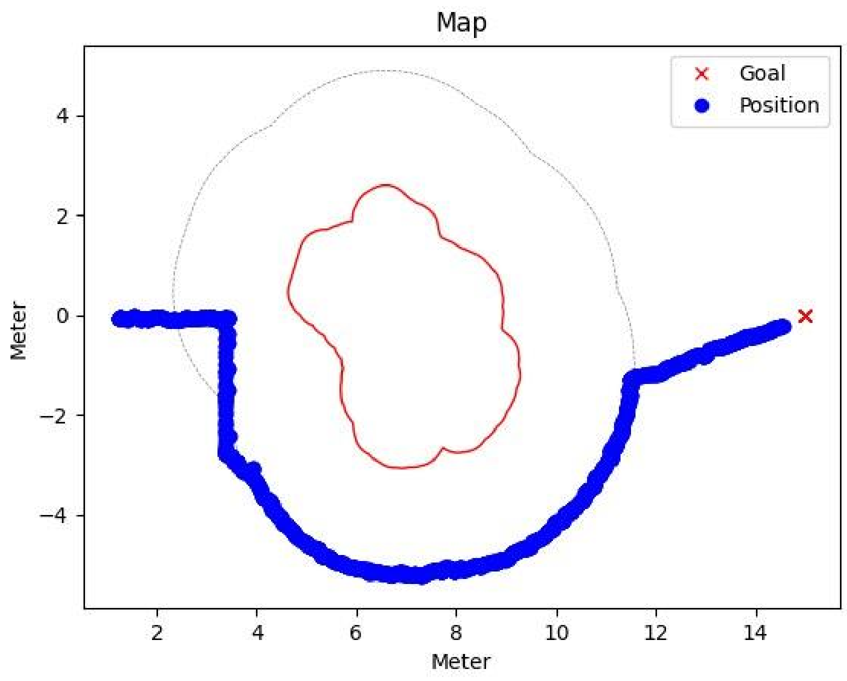


Figure 70 - BUG algorithm

The code is based on the non-linear controller with the difference of the path change to account for the obstacles. This aspect is resumed below, explained by the new *path_creator*:

- It initialises an empty list called *pid_path* to store the path's points.

- The method calls the *check_goal* function. It creates a Point object by the coordinates of the goal position. Therefore it checks if it lies within the obstacle area. If the goal is within it, it indicates it is not reachable. Therefore a new goal is computed using the nearest intersection (between the reference line and the obstacle) to the goal.
- After the check, the step size is defined, indicating the distance between the reference points on the path.
- It starts computing the point by the type of line (vertical, if the tractor's current position is aligned vertically with the goal position within tolerance or an oblique line). If an obstacle is encountered during the path creation, the method *obstacle_avoidance_path* is called to add a path that avoids the obstacle, using the obstacle’s contour to add a distance of a tunable way, as in the logic of *BUGs.*
- Finally, the goal position is added to the path.

Subsection 5.2.2 Occupancy Grid Map

An occupancy grid map is a map that for each cell shows the probability that the obstacle occupies it. Different versions can be implemented, from a binary map [0, 1] to complex solutions. In this work, each cell can go from 0 to 100 using integer numbers to reduce memory allocation.

Therefore each time the LiDAR clusters an object, its coordinate is used to add a pre-defined value to the cell inside the contour until a maximum of 100.

To solve the problem of the ‘ghost’ obstacle given by the machine’s vibration, each T (a period defined) all the matrix is diminished by a tuned value until a minimum value of 0.

The code is built with two functions.

One is called each time the *adaptative_clustering* algorithm sends an obstacle.

- The function iterates over each marker (obstacle box) in the message.
- It extracts the marker points' maximum and minimum x and y coordinates. They are used since the box is computed in the velodyne reference frame, always perpendicular to it.
- The function defines two transformation functions to convert the coordinates from the marker's frame of reference to the map's frame of reference, considering the robot's heading, position and the map translation vector.

- The function utilises transformation functions to calculate the box's transformed coordinates of the box's four vertices *(A, B, C, D)* in the absolute reference frame.
- Afterwards, it determines the polygon's bounding box in the absolute reference frame by identifying the minimum and maximum x and y values among the transformed vertices. To ensure a safe margin from the obstacle, the polygon's vertices are adjusted by adding or subtracting a value from the x and y coordinates based on their position relative to the bounding box's average x and y values.
- Finally, the function iterates over the cells within the polygon's bounding box. For each cell, the function checks if it is inside the polygon using a *ray-casting* algorithm. If the cell is inside the polygon, the function updates the corresponding cell value in the map.
- If the current cell value in the map is greater than or equal to the *100-increase value*, it is set to 100. Otherwise, it is incremented by the *increasing value.*

The second part is a function with a predefined interval to update the map, deleting false obstacle detection.

- The method takes an optional parameter *fixed_quantity* which defaults to 1.
- It uses the *NumPy* subtract function to subtract the *fixed_quantity* from each cell in the map. The out parameter stores the result in place, modifying the map directly.
- Then is ensured that any cell value in the map that becomes negative after the subtraction is set to zero.

5.2.2.1 A* application

The first algorithm to move the robot avoiding obstacles is the *A**. Each time an obstacle is founded, the map is updated; meanwhile, the *A** computes the accessible path. If one of the inputs is near one obstacle cell (probability higher than a threshold), the robot is stopped for safety reasons and the *A** is called computing a new free path.

The heuristic function must be trained to obtain the desired path and a short computation time.

The final one is shown below:

$$h(n) = W_e \cdot d_e(n) + W_o \cdot d_o(n) + W_l \cdot d_l(n)$$

with $W_e = 0.5$, $W_o = 8$, $W_l = 0.5$.

In the equation above, the W_e, W_o, and W_l are tuneable weights of the corresponding functions. In particular, d_e (n) is the Euclidean distance from the goal, d_o (n) is the inverse of the distance from the obstacle and d_l (n)is the distance from the line.

Regarding d_o (n), this value is computed by looking for the nearest obstacle cell in a pre-defined range; this range is imposed to reduce the computational time required.

Since this parameter is essential for the good dynamics of the robot, further research is done on that.

Regarding the code, the main difference with the non-linear controller is the occupancy grid map (the part before) and the new function for path planning. This last one is presented below:

- The method initialises an empty list called *pid_path* to store the computed path.
- The *ax*, *bx*, *cx*, and *dx* variables are computed by transforming the actual position and goal coordinates from the absolute reference frame to the matrix frame based on the translation vector and cell size.
- From *ax*, *bx*, *cx* and *dx* a function is used to find the row and column numbers in the grid correspondent.
- The *A** algorithm is called with the map, start position, and goal position to compute the possible path. The result is stored in the path variable.
- If no path is found, an error message is printed, and the *callback_emergency_stop* method is called.
- If a path is found, the method proceeds to process the path.
- The first two points are removed from the path (assuming they are very close to the current position), and the remaining path is stored in the *path_memory* list. The *round_edges* function is called to filter the path.
- It checks if the length of the path is less than 3. In that case, it returns the original path as there are low points for edge rounding.
- The start and end points of the path are extracted.
- The function initialises variables to track the maximum distance and the index of the point with the maximum distance.
- The function iterates over the points in the path (excluding the start and end points) and calculates the distance of each point from the line formed by the start and end points.

- If the calculated distance exceeds the current maximum distance, the maximum distance and its corresponding index are updated.
- After iterating through all the points, the function checks if the maximum distance is within the specified tolerance (input of the function). If so, it returns a new path consisting of only the start and end points, effectively removing the points in between.
- If the maximum distance exceeds the tolerance, the function repeatedly calls to round the edges of the two segments divided by the maximum distance point. It gives the recursive calls to the left segment (from the beginning to the maximum point) and the right segment (from the maximum point to the end).
- The left and right paths returned from the recursive calls are concatenated, excluding the last point of the left path, and returned as the rounded path. In this way, the discontinuous effect due to the discrete value of the map is smoother, allowing the robot to follow the trajectory without a continuous change of orientation.
- Achieved a smoother path; two versions of the path's storage are constructed, one with and one without the *round_edges* effect. If a path is found, the first point is removed from the path list (as it corresponds to the current position), and the remaining points are translated back to the map frame. The first path is used for navigation, while the other is to check for new obstacles.

Finally, some tests are done to compute the time needed to complete the algorithm in different situations like *Figure 71*, 200 random scenarios are tested. The results are shown in the following figures:

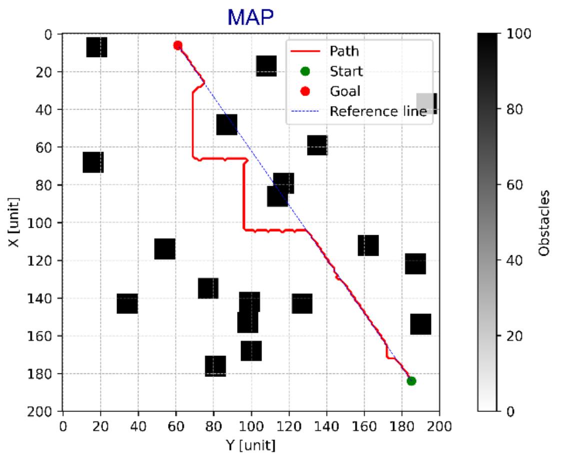


Figure 71 - A star path computed*

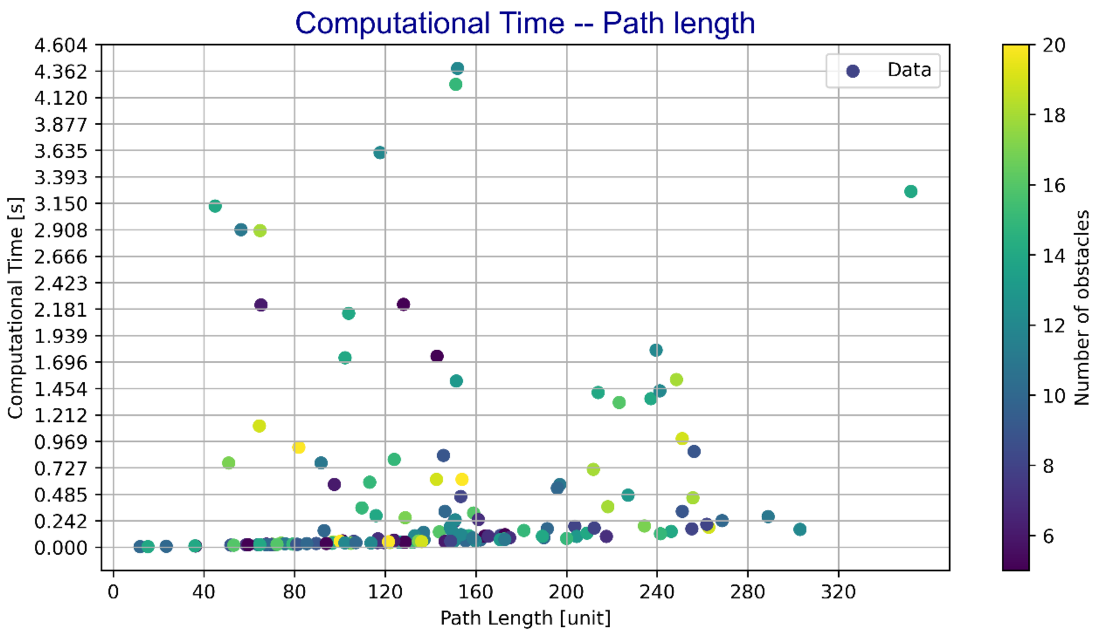


Figure 72 - A Time-Path length*

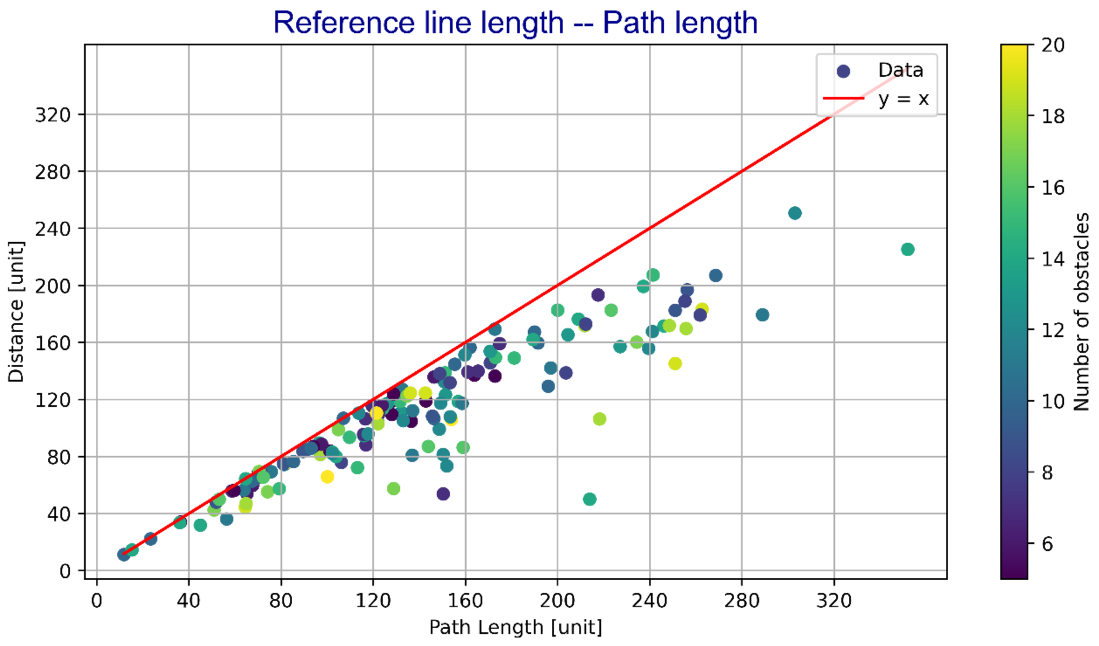


Figure 73 - A Reference – real length*

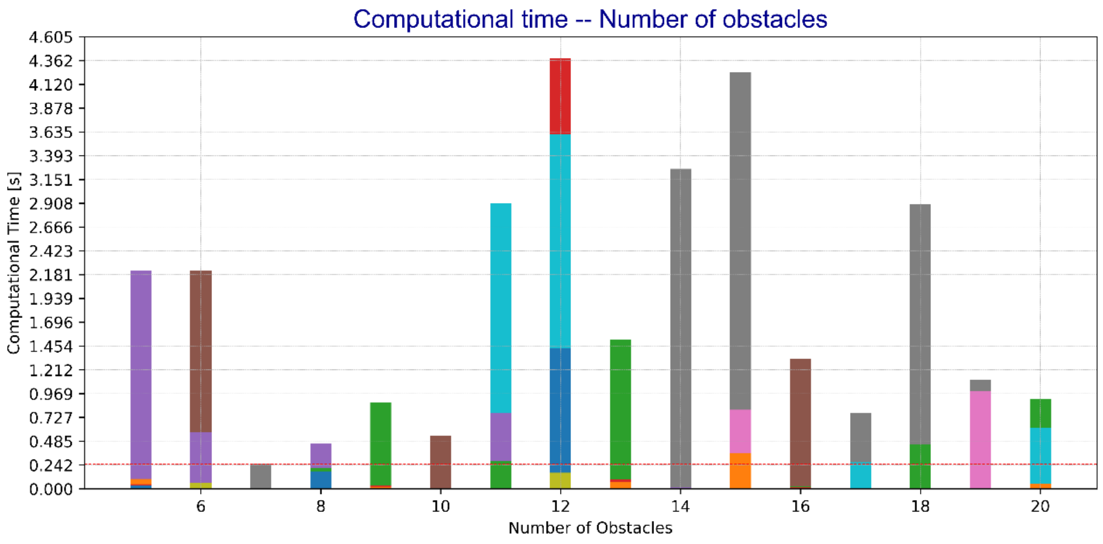


Figure 74 - A Time – number obstacles*

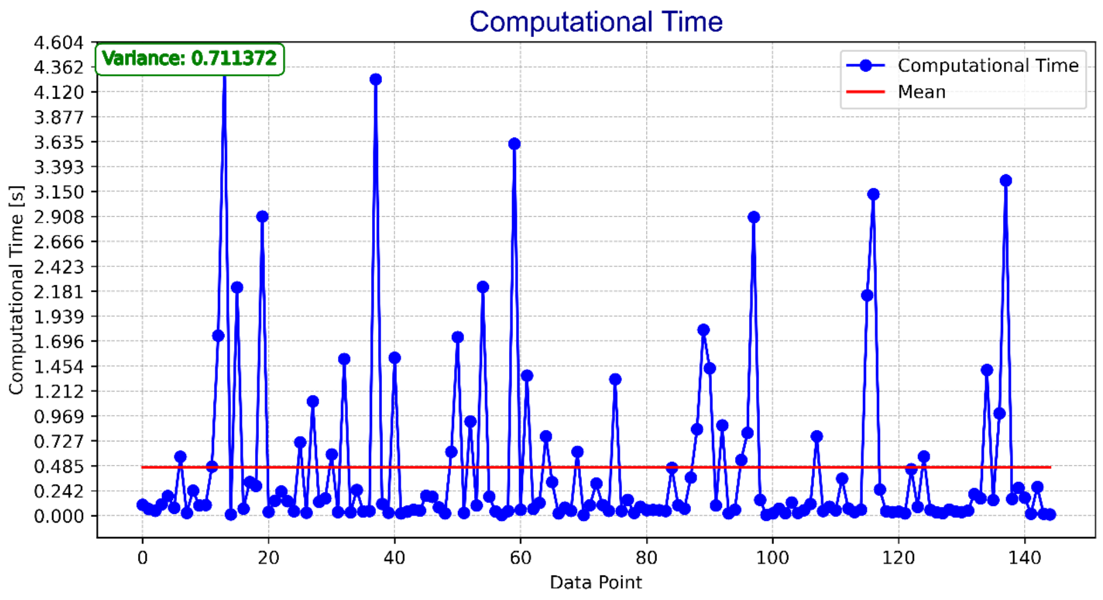


Figure 75 - A Time analysis*

5.2.2.2 Vector field application

This is the last algorithm proposed. It improves the code done, from the article *[15]*, imposing a vector field around the obstacle, making it able to avoid it.

Also in this case, further analysis is needed to choose the right angle of rotation of the repulsive field of the obstacle to make the robot not just pushed away and, in the meantime, not captured in his orbit.

The result is shown in the figures below, *Figure 76*.

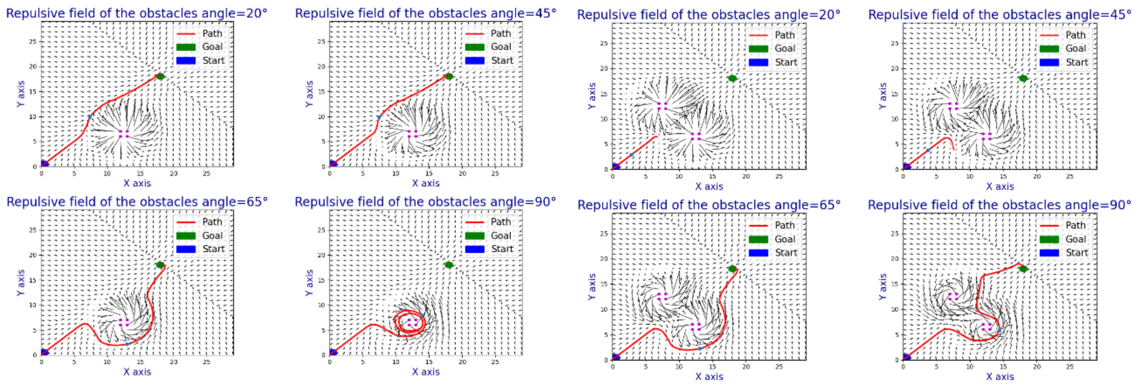


Figure 76 - Vector Field Obstacles

Regarding the code, the main difference with the first version of the vector field is using the *occupancy grid map* (already explained in *Subsection 5.2.2*) and the new *lineMission*.

This last one is presented below:

- The function takes several parameters as inputs: the current, starting and goal positions, the current heading, the grid map, the size of each cell in the map and the repulsion gain.
- The function computes the differences in x and y coordinates between the goal and original positions.
- It calculates the distance from the actual to the goal position using the Euclidean distance formula.
- If the distance to the goal is less than 2, it directly computes the steering commands based on the attractive force. In this way, the goal's field strongly pulls the robot.
- If the distance to the goal is greater than or equal to 2, the function proceeds to compute the *epsilon* value, which represents the perpendicular distance between the current position and the line connecting the original and goal positions.
- Based on the value of *epsilon*, the function determines the entity of the force that pushes the robot to the line, as explained in *Subsection 5.1.4*. Therefore, the steering command is computed.

- Next, the function iterates over a range of cells within a tuned range and computes the repulsion forces given by the probability of the cell being an obstacle. The repulsion force is added to the vector force components.
- The final steering command is computed by taking the arctangent of the vector force components.
- The steering error is computed using the desired and actual heading angles. It will be used as input for the angular PID.

Chapter 6 Simulation in Gazebo and URDF

In this chapter, the general aspect of simulation will be treated: firstly, a presentation of the most common simulation environment in ROS and robotic applications with particular attention on Gazebo; then the creation of the URDF (Unified Robotics Description Format) of the tractor is described; finally, some results of the algorithm in the simulation environment are shown.

Section 6.1 Introduction to simulation tools

Using simulation tools in ROS offers a valuable means to study and evaluate the capabilities of robotic systems. There are several notable simulation platforms available, including:

- *V-REP* (Virtual Robot Experimentation Platform): it provides a comprehensive 3D simulation environment compatible with ROS, enabling the realistic simulation of robots and their interactions with sensors.

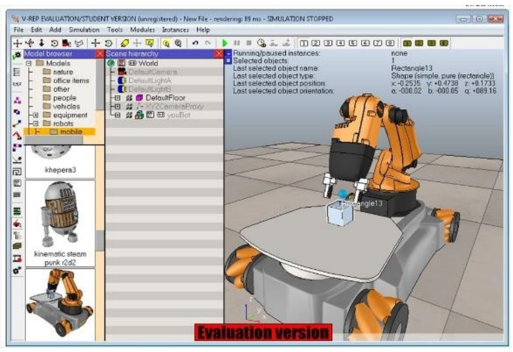


Figure 77 - V-REP

- *Gazebo* is renowned as one of the leading simulation environments for ROS; it facilitates the physics-based 3D simulation of robots and their dynamic engagement with the environment.

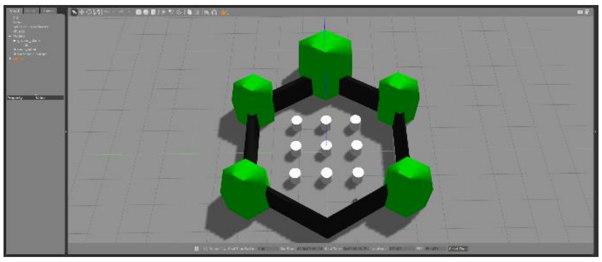


Figure 78 - Gazebo

- *MORSE* (Modular Open Robotics Simulation Engine): it is an open-source robotics simulator designed to integrate with ROS seamlessly. It focuses on delivering a high-fidelity simulation environment that accurately emulates robotic systems and complex environments.

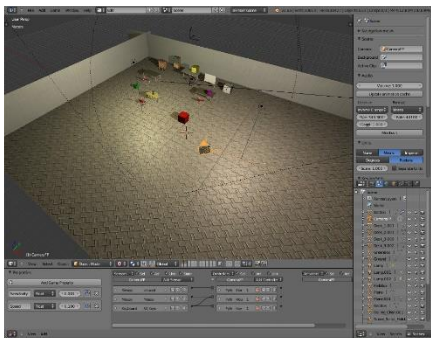


Figure 79 - MORSE

- *Webots*: it is a commercial robot simulator that supports ROS integration and provides a feature-rich 3D simulation environment.

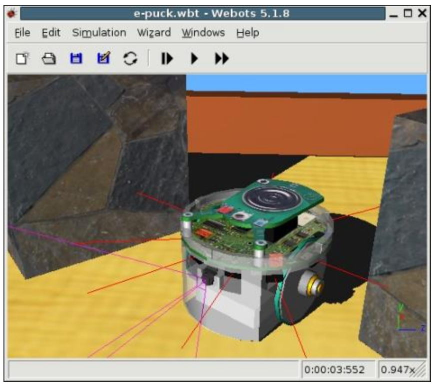


Figure 80 - Webots

- *USARSim*: it is highly regarded for its emphasis on high-fidelity simulation. USARSim offers accurate physics simulation and sensor models that closely resemble real-world scenarios. Its versatility in simulating diverse robot models and sensors makes it suitable for complex and varied robotic systems. Additionally, it supports multiple programming languages, enhancing its flexibility for simulation.

Figure 81 - USARSim

- *STDR/Stage*: it is a 2D robot simulator in the ROS package. It provides a simplified and effective simulation environment primarily focused on 2D navigation and sensor simulation. This lightweight solution is beneficial for the quick prototyping and testing of simple robotic systems within the ROS framework. Although it may lack the same level of realism and complexity as other simulation tools, it is an efficient option for basic robotic simulations.

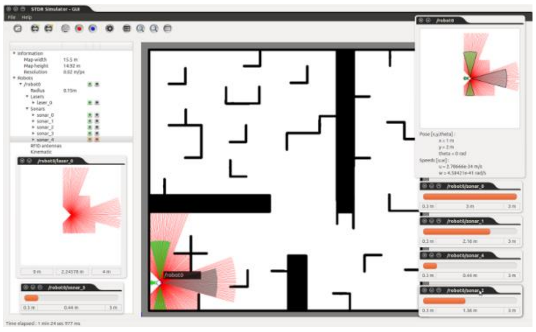


Figure 82 - STDR/Stage

- *Unity*: it, primarily a game development engine, also offers capabilities for robot simulation. Its robust 3D simulation environment leverages realistic physics and graphics. Unity's user-friendly interface and support for scripting in languages like C# enable users to customise and extend the platform to suit their simulation requirements. With a vast library of plugins and assets, Unity facilitates the creation of visually engaging simulations.

Figure 83 - Unity

After the first general presentation, the main characteristics of the different simulations environment are shown in *Table 6*, taken from *[42].*

	V-REP	Gazebo	MORSE	Webots	USARSim	STDR/Stage	Unity
Main Program. Language	C++	C++	Python	C++	C++	C++	C++
Operating System	Mac, Linux	Mac, Linux	BSD, Mac, Linux	Mac, Linux	Linux	Linux	Linux
Simulation Type	3D	3D	3D	3D	3D	2D	3D
Physics Engine	ODE, Bullet, Vortex, Newton	ODE, Bullet, Dart	Bullet	ODE	Unreal	OpenGL	Unity 3D
3D Rendering Engine	Internal, External	OGRE	Blender game	OGRE	Karma	-	OGRE
Portability	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Support	****	*****	****	****	***	****	**
ROS Compatibility	****	*****	****	***	**	****	*

Table 6 - Simulation tools comparison [42]

Finally, in the article, it is also shown as between the two popular simulation environments, *MORSE* and *Gazebo*, *MORSE* performed better than *Gazebo* in the tests conducted', despite that in this work, *Gazebo* is preferred for its better performance in terms of Support and ROS Compatibility, as shown in the table. It is analysed in detail in the following *Subsection 6.1.1*.

Subsection 6.1.1 Gazebo

As described before, *Gazebo* allows to build of a virtual environment and offers realistic simulation with its physics engine *[43].*

It has been selected as the official simulator of the DARPA Robotics Challenge7 in the US. It is a very popular simulator in robotics because of its high performance, even though it is open source.

Moreover, Gazebo is developed and distributed by *Open Robotics*, which controls ROS and its community, so it is very compatible with ROS. The following are the characteristics of the Gazebo.

- *Dynamics Simulation:* Only *ODE*(Open Dynamics Engine) was supported in the early development days. However, since version 3.0, various physical engines such as *Bullet, Simbody, and DART* have been used to meet the needs of multiple users.
- *3D Graphics:* Gazebo uses *OGRE*(Open-source Graphics Rendering Engines), which is often used in games; not only the robot model but also the light, shadow, and texture can be realistically drawn on the screen.
- *Sensors and Noise Simulation*: Laser range finder (LRF), 2D/3D camera, depth camera, contact sensor, force-torque sensor and much more are supported, and noise can be applied to the sensor data like the actual environment.
- *Plug-ins:* APIs enable users to create robots, sensors, and environment control as a plug-in.
- *Robot Model: PR2, Pioneer2 DX, iRobot Create, and TurtleBot* are already supported in the form of SDF, a Gazebo model file, and users can add their robots with an SDF file.
- *TCP/IP Data Transmission:* The simulation can be run on a remote server, and *Google's Protobufs*, a socket-based message passing, is used.
- *Cloud Simulation*: Gazebo provides cloud simulation *CloudSim* environment for use in cloud environments such as *Amazon*, *Softlayer*, and *OpenStack.*
- *Command Line Tool*: GUI and CUI tools are supported to verify and control the simulation status.

The word created with Gazebo in simulation in this work is based on the already defined space, *empty_word*, where then obstacles are placed randomly, as shown in *Figure 84*. It is not needed in a complex environment because the robot's goal is to navigate from goal to goal avoiding obstacles if present in outdoor environments. More complex scenarios can be used in future work, like the scenarios available in *[43]*. An example is reported in *Figure 85*.

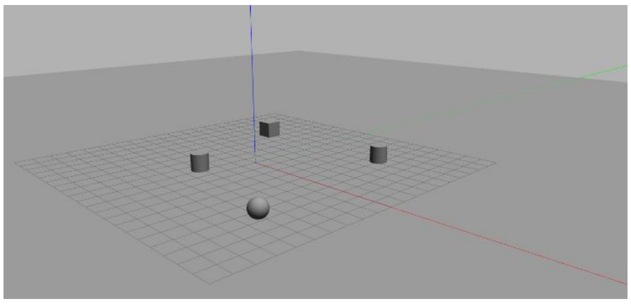


Figure 84 - Gazebo Empty World

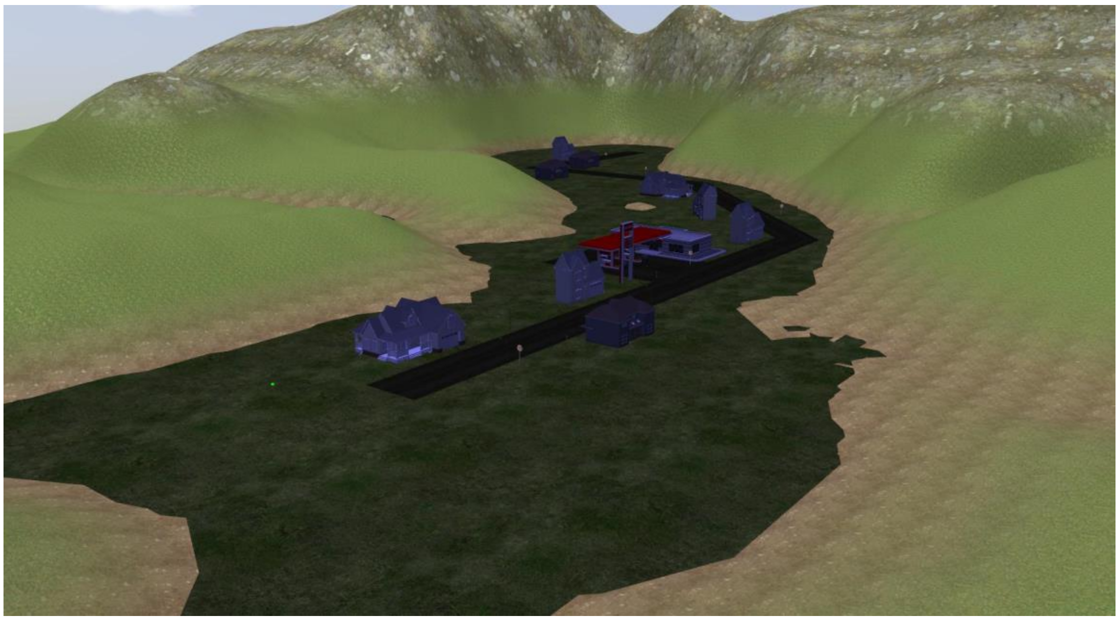


Figure 85 - Gazebo complex scenario [43]

Other than that, Gazebo has a relevant part in the development of the URDF file of the robot; indeed, thanks to the several plugins available simulated sensors and actuators can be added to simulate the actual behaviour (including the noise present), making available also the message exchanged.

Going deeper, the actuator and sensor used are presented:

- *Differential Drive*: it simulates the actuator of the wheels, making them rotate proportionally with the ROS message published.
- *IMU sensor (GazeboRosImuSensor)*: simulates an Inertial Motion Unit sensor.
- *GazeboRosMagnetic*: This plugin simulates a 3-axis magnetometer

- *GazeboRosGps*: *GazeboRosGps* simulates a GNSS (Global Navigation Satellite System) receiver which is attached to a robot. It publishes *sensor_msgs/NavSatFix* messages with the robot's position and altitude in WGS84 coordinates, together with the IMU sensor and the magnetometer, to simulate the presence of the DURO in the real robot.
- *VLP16*: is the digital twin of the LiDAR present on the robot. It is available in the package *velodyne* in the ROS Community *[32]* together with different types of LiDAR; a difference from the previous component is that this one is presented as a xacro file that can be added to the robot.

Section 6.2 Creation and Configuration of the URDF file

In this section, there will be a presentation on how a URDF file and the central concept are related, while the model of the tractor with be shown. The book *[44]* is used as a reference for the first part.

Subsection 6.2.1 URDF file

The Unified Robot Description Format (URDF) is an XML specification used in the Robot Operating System (ROS) to describe the properties of a robot, including its kinematics, dynamics, visual representation, and collision model. This standardised format provides a way to represent a robot's geometry, joints, sensors, and other attributes.

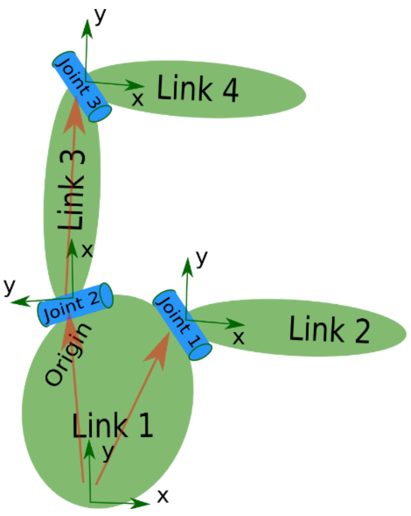


Figure 86 - General URDF tree

A URDF file consists of a set of link elements and joint elements that define the robot’s structure.

The link element describes a rigid body with mass, visual features, and collision properties; specified as *<inertial>*, *<visual>*, and *<collision>*.

The <*inertial*> element provides information about the mass, centre of mass and moments of inertia of the link. It includes sub-elements like <*origin*> to define the position and orientation of the link's centre of mass frame relative to its frame. The <*visual*> element describes the visual representation of the link using shapes such as boxes, cylinders, spheres, or meshes. It also allows for specifying materials and textures for visualisation. The <*collision*> element defines the collision properties of the link, which may differ from its visual representation for efficiency purposes.

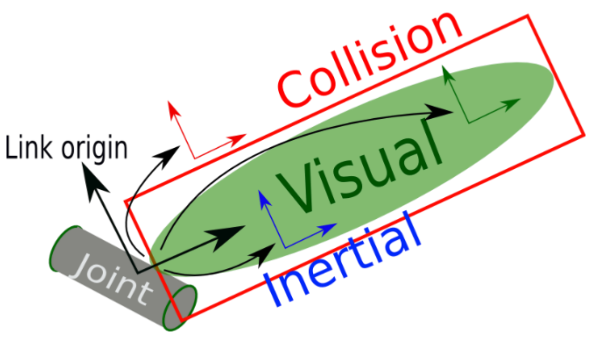


Figure 87 - Link representation

The joint element describes the kinematics and dynamics of the joint connecting two links. It includes features such as <*origin*> to specify the transform from the parent link to the child link, <*axis*> to define the axis of rotation or translation for the joint, and <*limit*> to set the joint limits for position, velocity, and effort. The type of joint can be revolute, prismatic, fixed, floating, or planar, each with its specific characteristics.

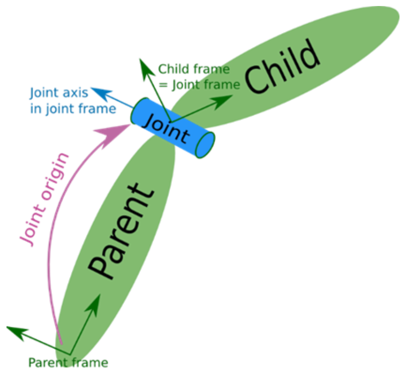


Figure 88 - Joint representation

Additional elements such as <*calibration*>, <*dynamics*>, <*mimic*>, and <*safety_controller*> can be used to provide further details about joint properties, such as calibration reference positions, physical dynamics parameters, mimic relationships with other joints, and safety limits.

Subsection 6.2.2 XACRO file

For the tractor URDF model, a XACRO file is used. XACRO is an extension of the URDF used in ROS for describing the properties of a robot. It provides a more flexible and modular way to define robot models using macros and parameters and includes them within the XML structure.

XACRO files are typically pre-processed to generate valid URDF files before being used. The XACRO format enhances the reusability and maintainability of robot descriptions by introducing several features:

- *Macros*: XACRO allows the definition of macros, which are reusable XML snippets used to generate repetitive structures or components within the robot model. *Macros* enable code modularity and reduce duplication by encapsulating commonly used elements.
- *Parameters*: XACRO supports using parameters, which can be assigned values and utilised within the XACRO file. *Parameters* provide flexibility to define robot properties dynamically, making it easier to customise and configure robot models for different scenarios.
- *Includes*: XACRO allows the inclusion of other XACRO or URDF files, enabling the composition of complex robot models from multiple modular components. *Includes* facilitates code organisation and promotes code reuse by splitting the robot description into smaller, manageable files.

Resuming XACRO simplifies creating and maintaining robot descriptions. It enhances the readability, flexibility, and modularity of the URDF files, making it easier to manage large and complex robot models.

Subsection 6.2.3 Tractor URDF

Since the need to use a model with a similar size to the real tractor is crucial above all for the simulation part, a URDF file is built.

During the construction, different choices were taken to obtain a collision element similar to the real one and simultaneously respect all the physical laws in *Gazebo*. In particular, the robot's size and weight required specific attention to the vehicle's inertia; in this case, homogeneous components were supposed.

Then since the robot is a two differential robot, two motion wheels are imposed, visualising them in the back but placing really in the middle to be congruent with the inertia chosen.

In the above part, the evolution of the robot URDF file is shown:

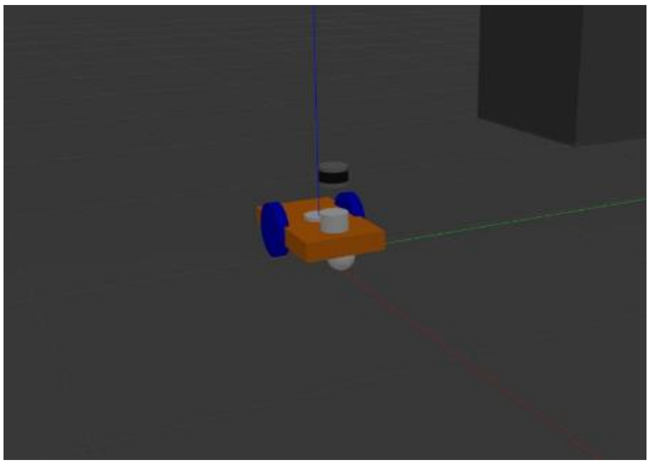


Figure 89 - Robot first version

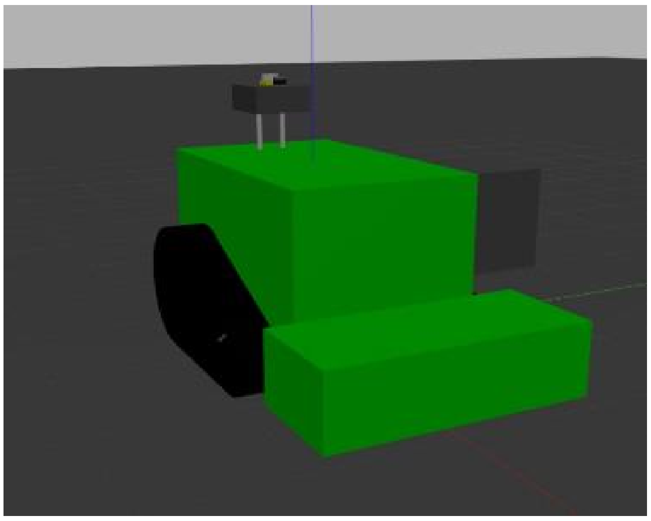


Figure 90 - Robot second version

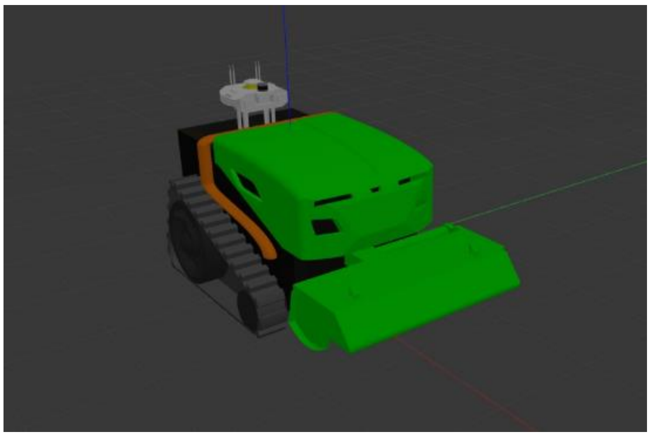


Figure 91 - Robot final version

As it is possible to see, firstly, the actual dimensions were applied, and then the mesh component was used to make the visual part more similar.

Section 6.3 Simulation result

In robotic applications, the simulation part is essential since it allows fast debugging and speeds up the developing process; in the thesis applications, it takes greater importance due to the danger given by the size and the strength of the machine tested in undesired behaviours. For this reason, all the field tests are preceded by a simulation part.

The simulation also allows us to test critical situations that are challenging to reproduce in the field.

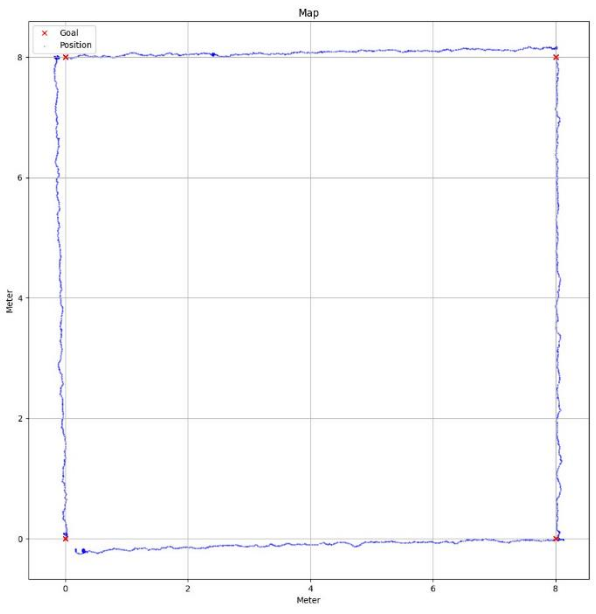


Figure 92 - Vector Field Simulation (Goal radius 0.01m)

For this reason, two experiments are done: the first with more than one obstacle in a single path; this is done two times with different configurations to see the algorithm's efficiency. The second one, instead, will analyse the performance of doing a triangular path with different obstacles in the path.

The navigation simulation without obstacles is not shown because since we choose to put minimal values for the sensor's noise, all the results have excellent results, making it difficult to compare. An example is *Figure 92*, in which the *Vector Field* algorithm achieves a precision of 0.01 m to reach the goal.

The reason not to have noise was because the aim was to check the correctness of the algorithm. As regards the noise problem is analysed directly on the field test.

Subsection 6.3.1 Three Obstacles in the Path (right way)

As presented in the chapter's introduction, the first experiment is to use the two algorithms in a single path with more than one obstacle. In both cases, the robot could reach the goal safely; further details are provided nextly.

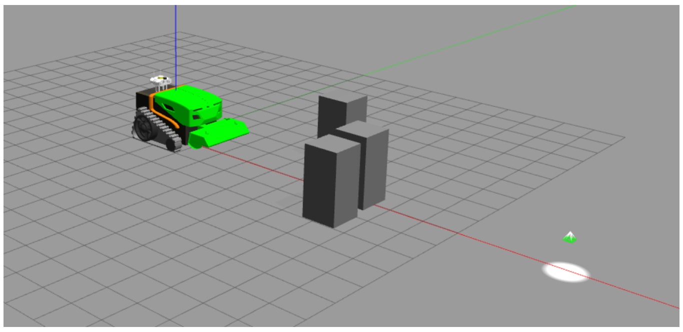


Figure 93 - Three obstacles in the path, right way optimal path

6.3.1.1 A* algorithm with non-linear controller (right way)

The results of the first simulation are reported in this part.

First, the path computed by the *A* algorithm* is shown in *Figure 94*. As it is possible to notice, this path is filtered by the *round_edges* described in *Chapter 5*, which allows removing the 'stairs' aspect given by the discretisation of the map.

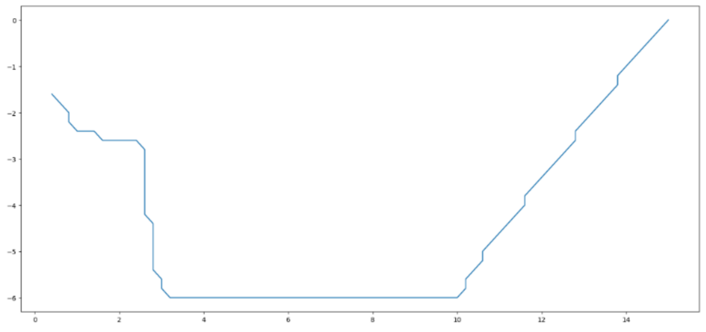


Figure 94 - Reference path, A* algorithm (right way)

Secondly, the map obtained at the end of the simulation is reported in *Figure 95*. As it is possible to see, the two obstacles on the right are fused in just one due to their nearness. Despite that, the final path makes a good trajectory for the robot.

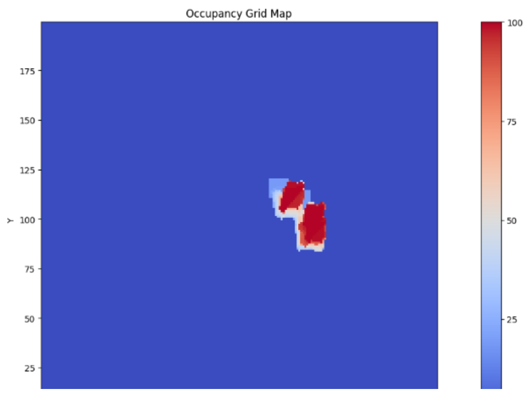


Figure 95 - Occupancy Grid Map, A* algorithm (right way)

Finally, the real trajectory done by the robot during the simulation is shown in *Figure 96*.

The simulation is good overall since no problems are present.

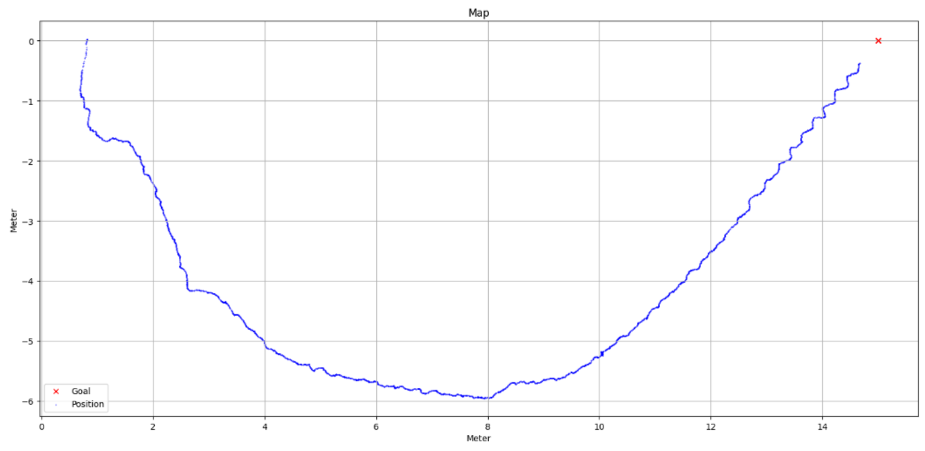
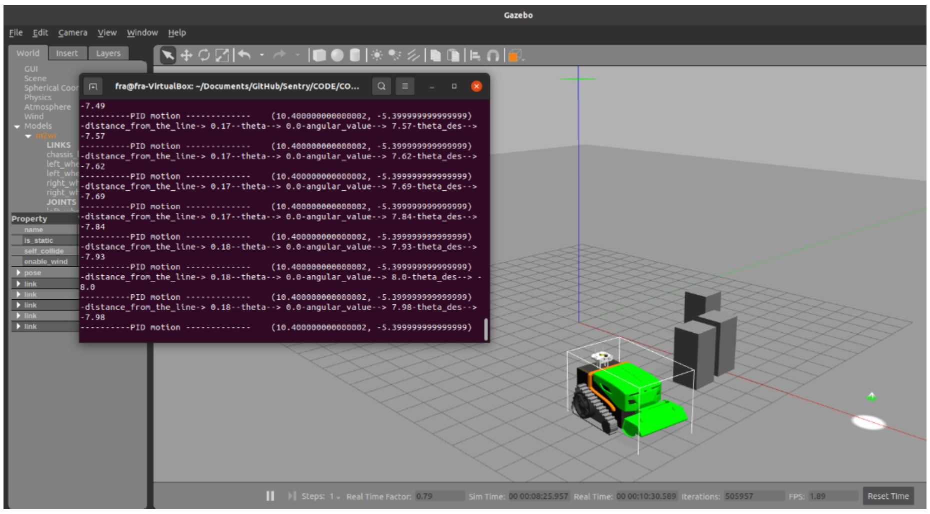


Figure 96 - Path done, A* algorithm (right way)


Gazebo
File Edit Camera View Window Help
World Insert Layers
fra@fra-VirtualBox: ~/Documents/GitHub/Sentry/CODE/CO...
Reset Time

```
-7.49
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.17--theta--> 0.0-angular_value--> 7.57-theta_des-->
-7.57
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.17--theta--> 0.0-angular_value--> 7.62-theta_des-->
-7.62
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.17--theta--> 0.0-angular_value--> 7.69-theta_des-->
-7.69
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.17--theta--> 0.0-angular_value--> 7.84-theta_des-->
-7.84
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.18--theta--> 0.0-angular_value--> 7.93-theta_des-->
-7.93
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.18--theta--> 0.0-angular_value--> 8.0-theta_des--> -
8.0
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
-distance_from_the_line-> 0.18--theta--> 0.0-angular_value--> 7.98-theta_des-->
-7.98
---------PID motion ------------    (10.400000000000002, -5.399999999999999)
```

Figure 97 - Simulation instant, A* algorithm (right way)

6.3.1.2 Vector Field Algorithm (right way)

The results of the second simulation are reported in this part.

A reference path is not shown in this case since it is not computed.

So the map obtained at the end of the simulation is reported in *Figure 98*.

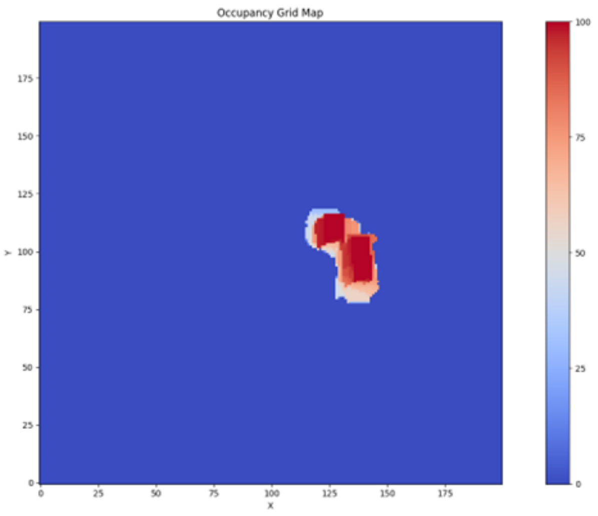


Figure 98 - Occupancy Grid Map, Vector Field algorithm (right way)

Finally, the path done by the robot in the simulation is shown in *Figure 99*; the only

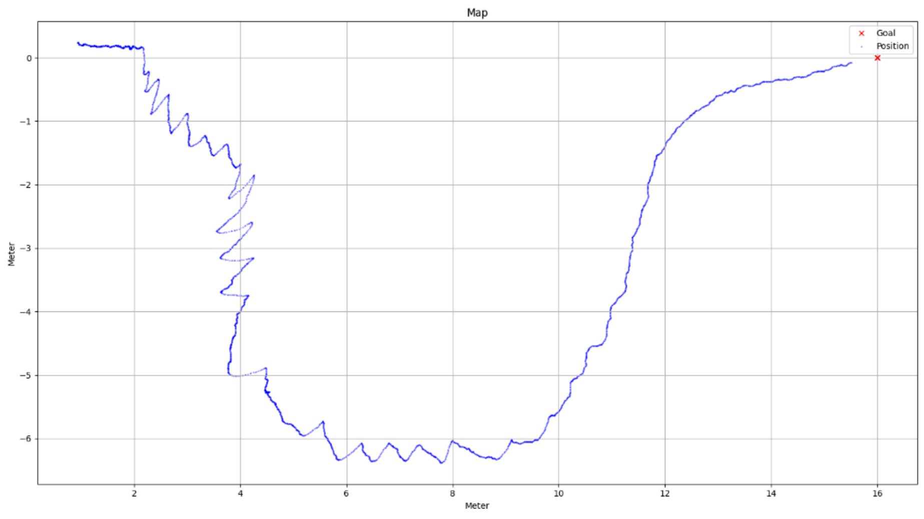
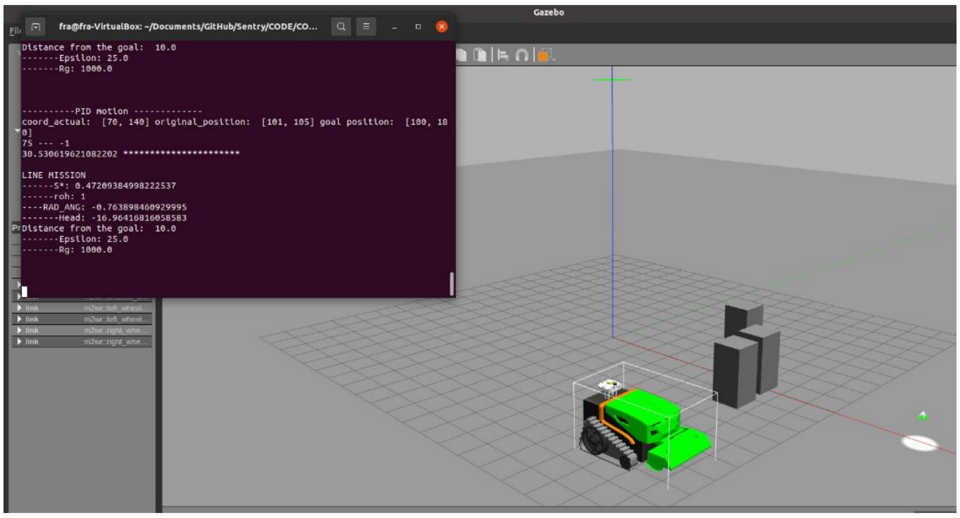


Figure 99 - Path done, Vector Field algorithm (right way)

observation that can be done in this case is the explanation of the oscillation behaviour; this is due to the angle of rotation used for the obstacle, as explained in *Chapter 5*. Despite that, the field test does not show this behaviour thanks to the absolute precision (worse).



```
Distance from the goal:  10.0
-------Epsilon: 25.0
-------Rg: 1000.0



----------PID motion ------------
coord_actual:  [70, 140] original_position:  [101, 105] goal position:  [100, 18
0]
75 --- -1
30.530619621082202 **********************

LINE MISSION
------S*: 0.47209384998222537
------roh: 1
----RAD_ANG: -0.763898460929995
-------Head: -16.96416816058583
Distance from the goal:  10.0
-------Epsilon: 25.0
-------Rg: 1000.0
```

Figure 100 - Simulation instant, Vector Field algorithm (right way)

Subsection 6.3.2 Three obstacles in the path (left way)

This is the second part of the first test, in which the optimal path is in the opposite direction of the first one.

Also, in this case, both solutions were successful. Indeed the robot reached the goal safely; further details are provided next.

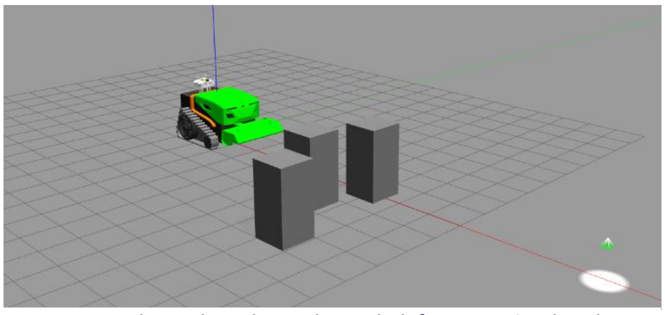


Figure 101 - Three obstacles in the path, left way optimal path

6.3.2.1 A* algorithm with non-linear controller (left way)

The results of the *A* algorithm* are reported in this part.

To begin, *Figure 102* shows the path computed from the *A* algorithm*.

Also in this case, it is filtered by the *round_edges* described in *Chapter 5*.

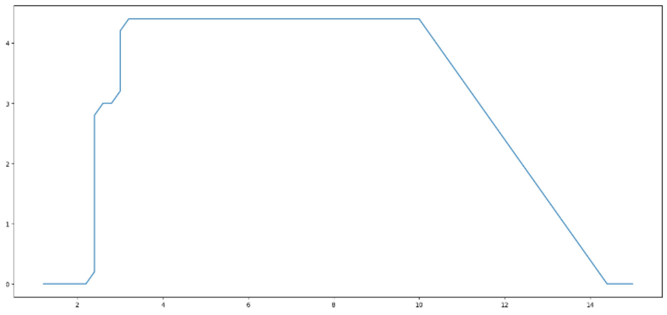


Figure 102 - Reference path, A* algorithm (left way)

The figure shows the map gained at the end of the simulation, while the real path is shown in *Figure 103*.

Also in this case, the results of the simulation are positive.

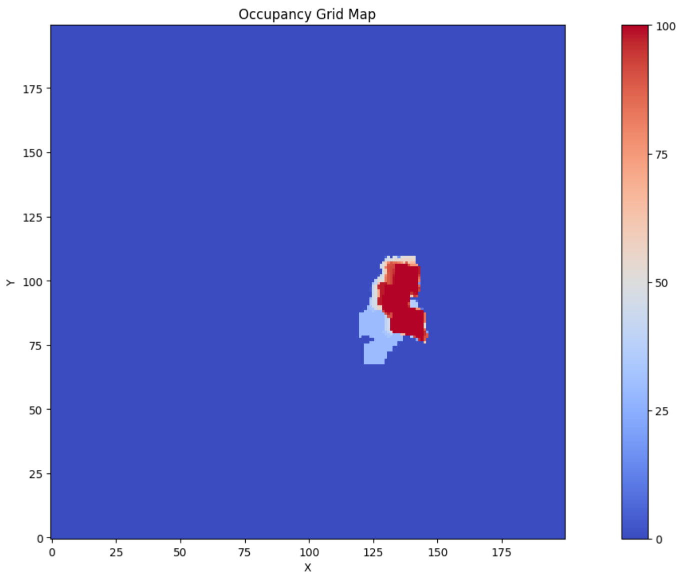


Figure 103 - Occupancy Grid Map, A* algorithm (left way)

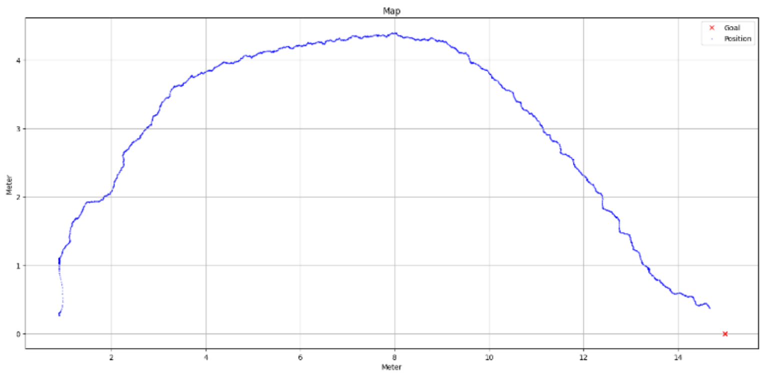
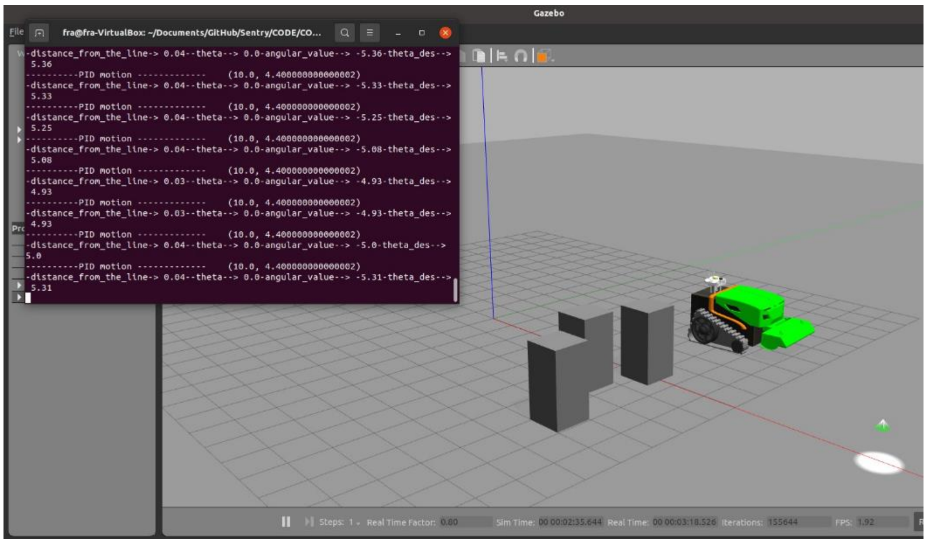


Figure 104 - Path done, A algorithm (left way)*


Gazebo
File
fra@fra-VirtualBox: ~/Documents/GitHub/Sentry/CODE/CO...
Steps: 1
Real Time Factor: 0.80
Sim Time: 00 00:02:35.644
Real Time: 00 00:03:18.526
Iterations: 155644
FPS: 1.92

```
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.36-theta_des-->
 5.36
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.33-theta_des-->
 5.33
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.25-theta_des-->
 5.25
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.08-theta_des-->
 5.08
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.03--theta--> 0.0-angular_value--> -4.93-theta_des-->
 4.93
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.03--theta--> 0.0-angular_value--> -4.93-theta_des-->
 4.93
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.0-theta_des-->
5.0
----------PID motion -------------    (10.0, 4.400000000000002)
-distance_from_the_line-> 0.04--theta--> 0.0-angular_value--> -5.31-theta_des-->
 5.31
```

Figure 105 - Simulation instant, A algorithm (left way)*

6.3.2.2 Vector Field Algorithm (left way)

As for the *Vector Field* test, no reference paths are computed, but it is essential to notice that since the vector field of the obstacle is constantly rotating counter-clockwise, the robot takes a longer path to avoid the obstacle (going to the right side), as shown in *Figure 107*.

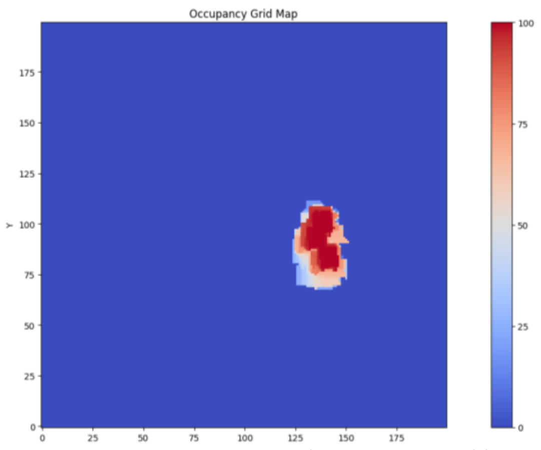


Figure 106 - Occupancy Grid Map, Vector Field algorithm (left way)

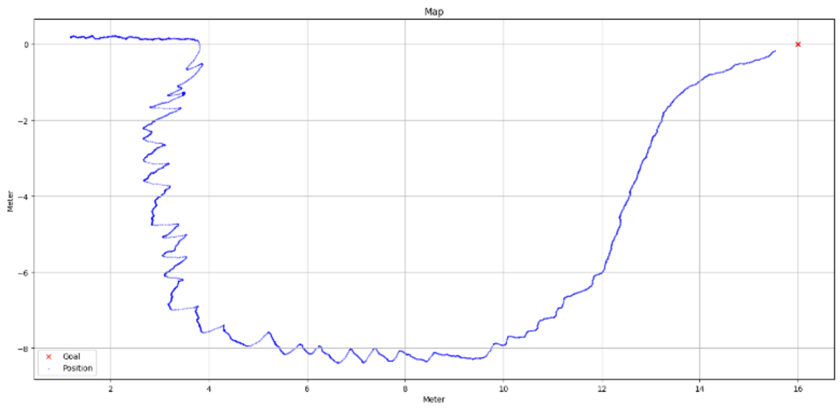
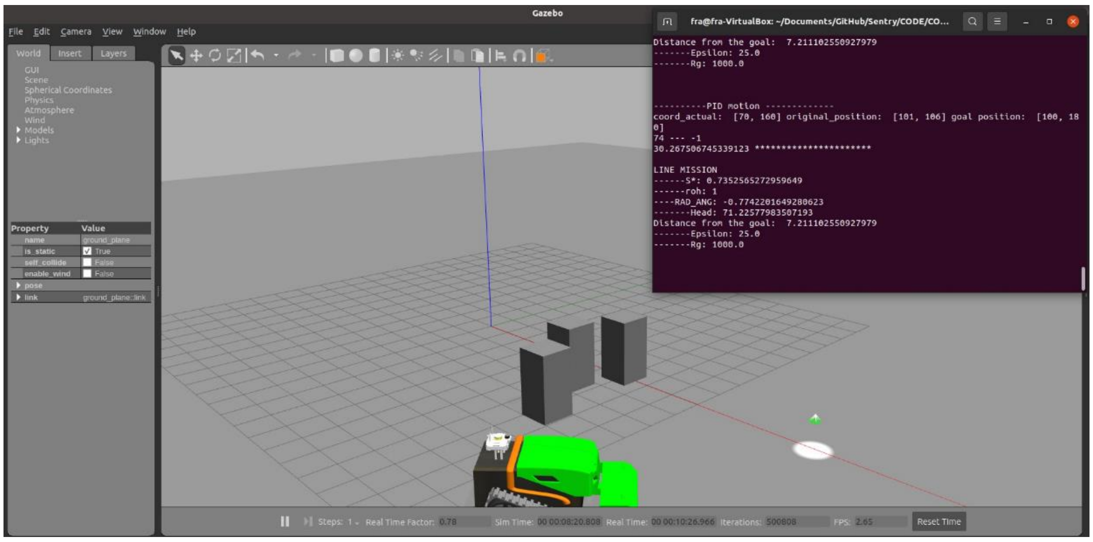


Figure 107 - Path done, Vector Field algorithm (left way)



```
Distance from the goal:  7.211102550927979
-------Epsilon: 25.0
-------Rg: 1000.0



----------PID motion -------------
coord_actual:  [70, 160] original_position:  [101, 106] goal position:  [100, 18
0]
74 --- -1
30.267506745339123 **********************

LINE MISSION
------S*: 0.7352565272959649
------roh: 1
----RAD_ANG: -0.7742201649280623
-------Head: 71.22577983507193
Distance from the goal:  7.211102550927979
-------Epsilon: 25.0
-------Rg: 1000.0
```

Figure 108 - Simulation instant, Vector Field algorithm (left way)

Subsection 6.3.3 Complex environment

The second experiment aims to simulate the robot’s behaviour in a complex scenario where more than one obstacle (four in the simulation) are placed on the map. The environment is shown in *Figure 109*.

The goals to be reached are shown in *Table 7*.

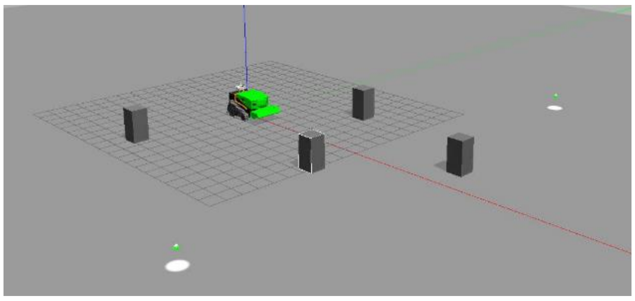


Figure 109 - Complex environment test

Point	**X**	**Y**
A	15.0	-15.0
B	15.0	15.0
C	0.0	0.0

Table 7 - Complex environment simulation path

6.3.3.1 A* algorithm with non-linear controller (complex test)

The first result shown is about the path done by the robot using the A* algorithm. As shown in *Figure 111*, the robot can reach all the goals by avoiding obstacles in the path. No particular observation must be done.

Finally, the map at the end of the simulation is shown in Figure. In this case, it is possible to see that the *adaptative_clustering* algorithm detects obstacles in the wrong position during navigation. The *A** does not suffer from this problem, but a better

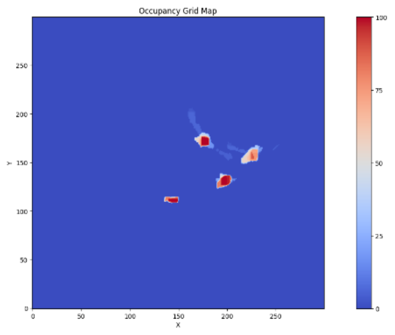


Figure 110 - Occupancy Grid Map, A algorithm (complex test)*

tuning of the map’s parameters can be done to ensure that false detections are deleted in a shorter time.

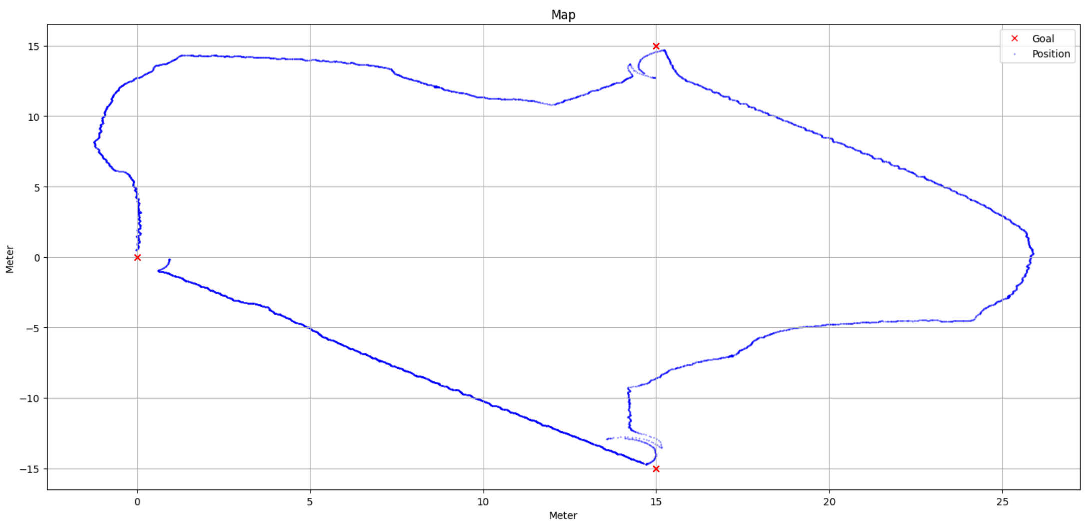
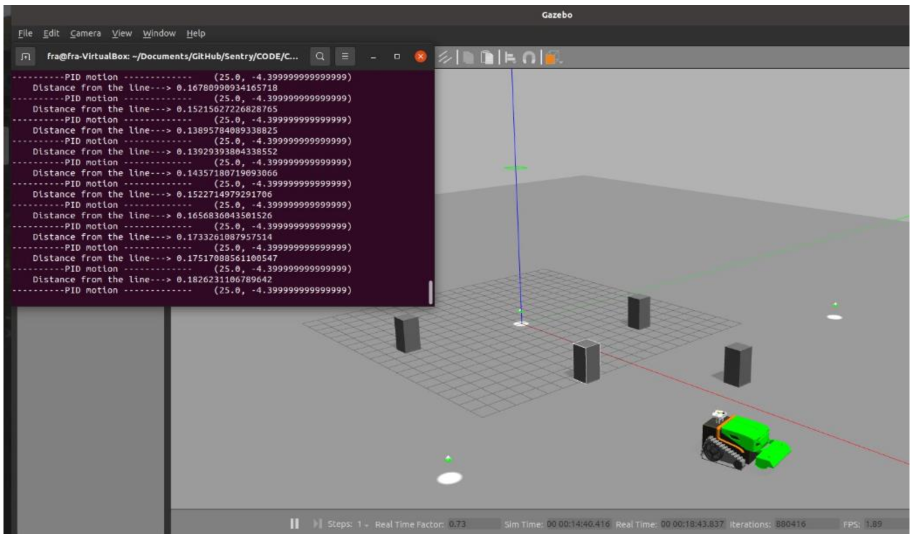


Figure 111 - Path done, A algorithm (complex environment)*


Gazebo
File
Edit
Camera
View
Window
Help
fra@fra-VirtualBox: ~/Documents/GitHub/Sentry/CODE/C...
Steps: 1
Real Time Factor: 0.73
Sim Time: 00 00:14:40.416
Real Time: 00 00:18:43.837
Iterations: 880416
FPS: 1.89

```
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.16780990934165718
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.15215627226828765
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.13895784089338825
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.13929393804338552
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.14357180719093066
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.1522714979291706
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.1656836043501526
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.1733261087957514
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.17517088561100547
----------PID motion -------------    (25.0, -4.399999999999999)
    Distance from the line---> 0.1826231106789642
----------PID motion -------------    (25.0, -4.399999999999999)
```

Figure 112 - Simulation instant, A algorithm (complex environment)*

6.3.2.2 Vector Field Algorithm (complex environment)

This simulation shows some problems with its algorithm. Indeed the wrong localisation of the obstacle in the map creates a particular vector field that makes the robot go back, starting a cycle trajectory as shown in *Figure 114*. Thanks to the map dynamic, as soon as these false detections are deleted, the robot avoids them and reaches the goal.

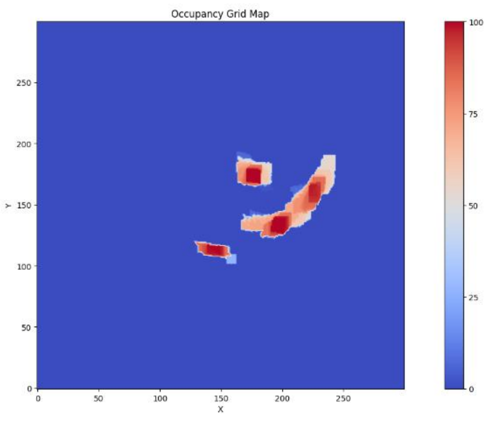


Figure 113 - Occupancy Grid Map, Vector Field algorithm (complex test)

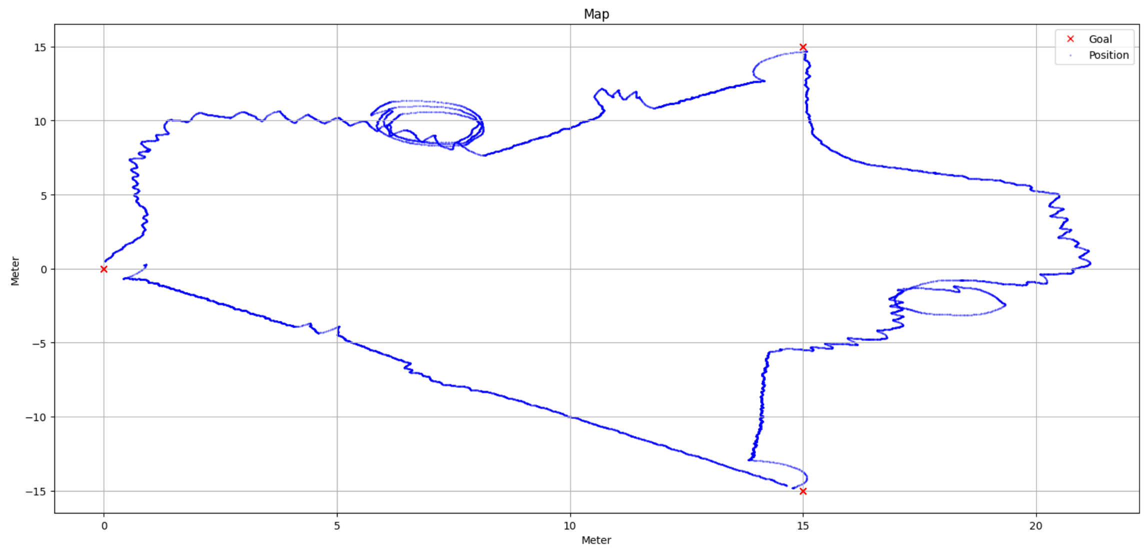
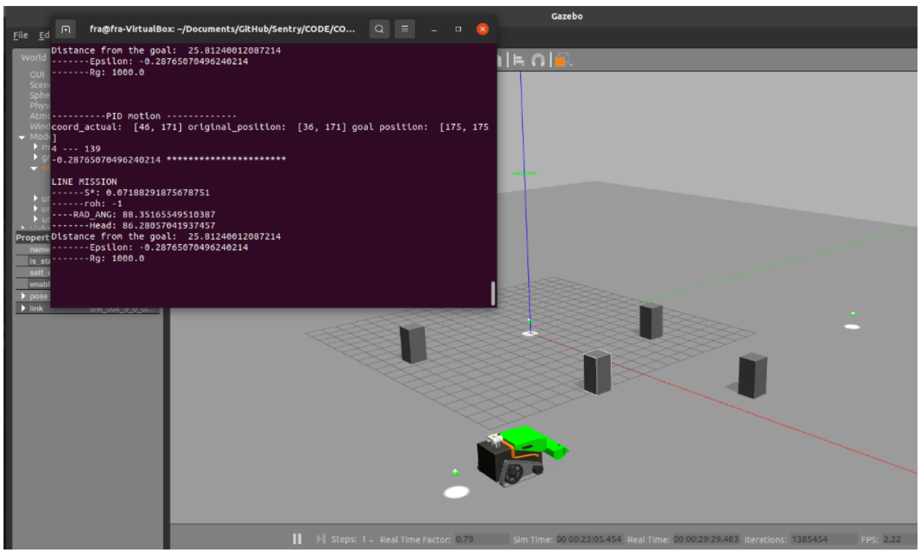


Figure 114 - Path done, Vector Field algorithm (complex environment)



```
Distance from the goal:  25.81240012087214
-------Epsilon: -0.28765070496240214
-------Rg: 1000.0



----------PID motion ------------
coord_actual:  [46, 171] original_position:  [36, 171] goal position:  [175, 175
]
4 --- 139
-0.28765070496240214 **********************

LINE MISSION
------S*: 0.07188291875678751
------roh: -1
----RAD_ANG: 88.35165549510387
-------Head: 86.28057041937457
Distance from the goal:  25.81240012087214
-------Epsilon: -0.28765070496240214
-------Rg: 1000.0
```

Figure 115 - Simulation instant, Vector Field algorithm (complex environment)

Chapter 7 Results and Analysis

In this chapter, the results of the navigation are shown. The first test is related to the ability to make a square. The second test wants to observe the navigation of the two more updated algorithms during a longer path; the third is instead related to avoiding an obstacle in the path.

Section 7.1 GUI

To begin, *Figure 116* shows the *GUI* (Graphical User Interface) created to have real-time information during the test. *Figure 116* shows the use of it in a simulation environment, but its application was crucial during the test to have a precise idea of where the robot was on the map and which were the actual values of velocity; it also shows if a person is detected using colouring by red the fourth bar.

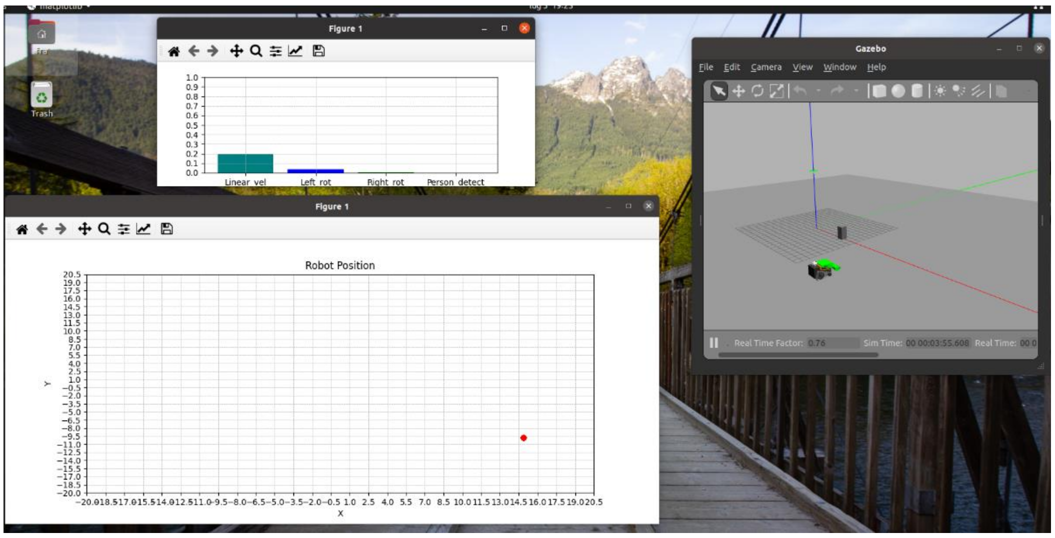


Figure 116 - GUI

Section 7.2 Square navigation

The first test proposed is about the precision of performing a squares path; for this purpose, the coordinates sent to the robot were:

Point	**X**	**Y**
A	8.0	0.0
B	8.0	8.0
C	0.0	8.0
D	0.0	0.0

Table 8 - First test path

To obtain a good comparison, all the tests are done by aligning initially to the east (x-axis) the robot and imposing its actual position as zero (in the map, the initial position will be around 1 meter above since there is a transformation between the GPS position and the centre of the robot); in this way, they all will start without the alignment step. Furthermore, the same filters are used for all (Average for the magnetometer, EKF for the position), and all are done with a full signal of GPS+RTK. The final result a shown in *Figure 117*.

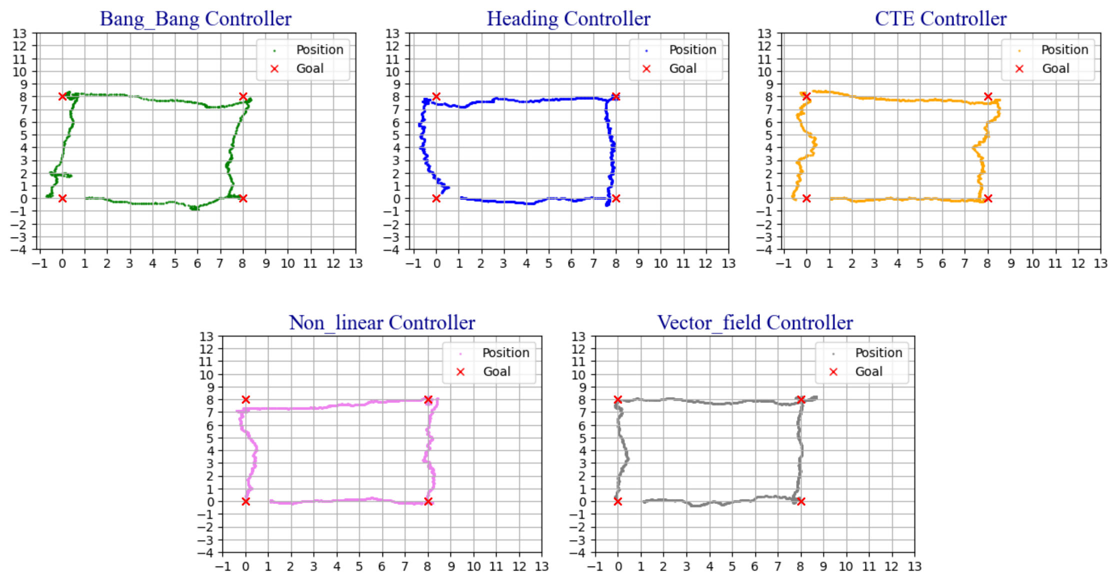


Figure 117 - Trajectory of each algorithm

So for each test, the distance from the reference line is computed to get performance indices like the mean error, the maximum error and the variance. The result is shown below, *Figure 118*.

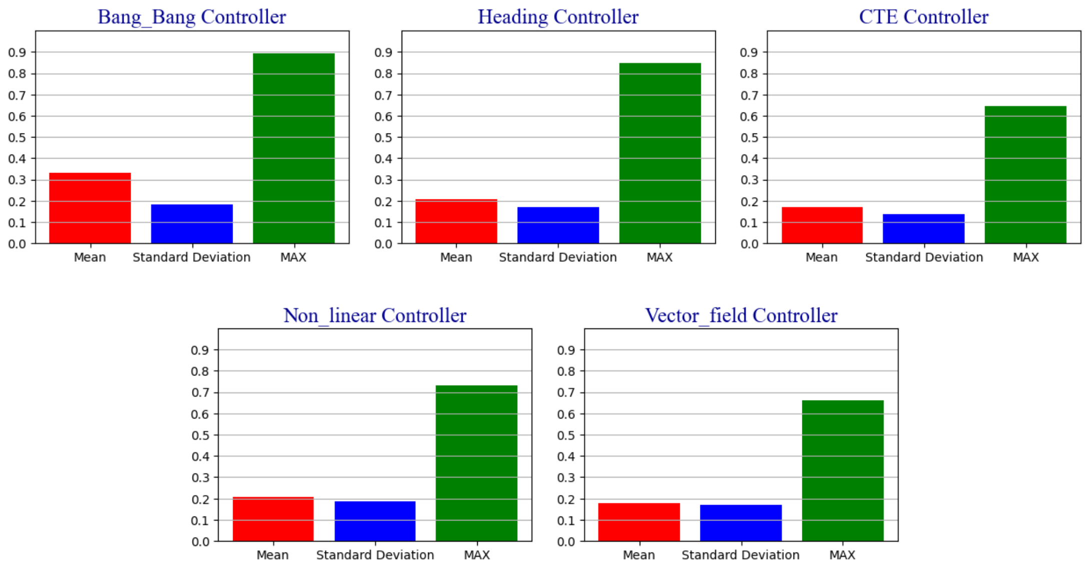


Figure 118 - Performance indices

As it is possible to see, the algorithm with the highest performance is the *Vector Field* controller, but all the controllers have a precision of around 20 cm.

It is worth noting that only the two most minor proposals can be utilised in a condition of the path with obstacles.

Section 7.3 Multipath

For the last two codes, the ones able to avoid possible obstacles, another test about navigation is performed. In this case, the robot must follow a more complex path similar to a working one. Also in this case indices of performance are computed.

The *Vector Field* trajectory is shown below, *Figure 119*:

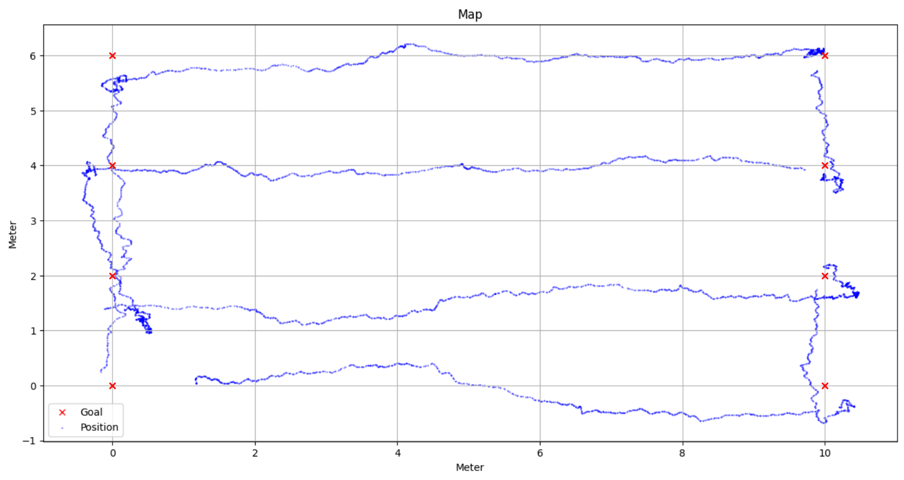


Figure 119 - Vector Field Multipath

The same is done for the *non-linear controller*, *Figure 120*:

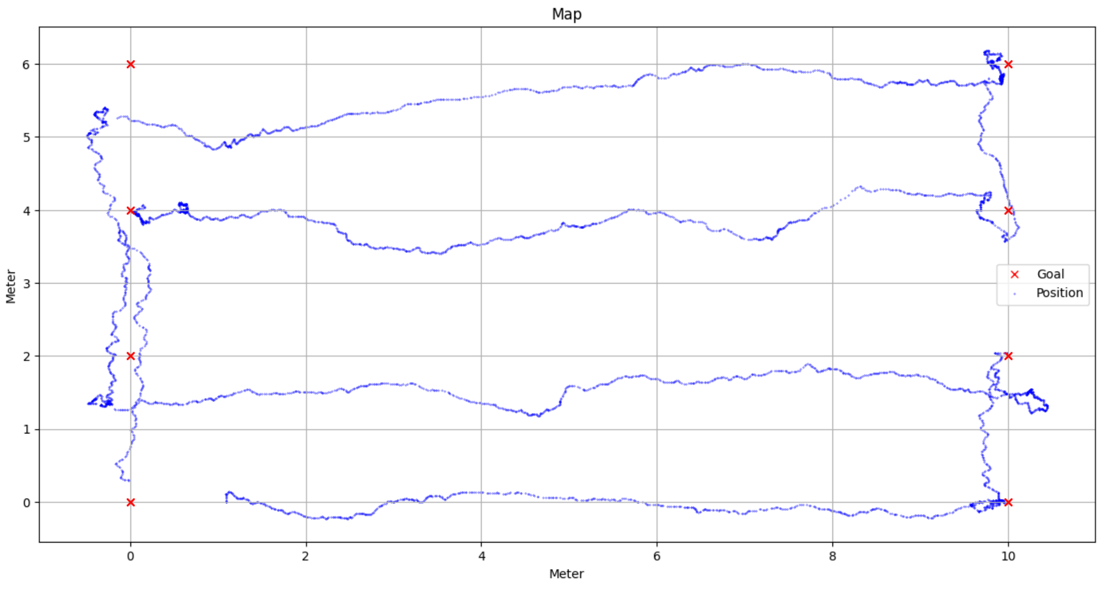


Figure 120 - Non-linear controller Multipath

Finally, the conclusive indices are shown in *Figure 121*:

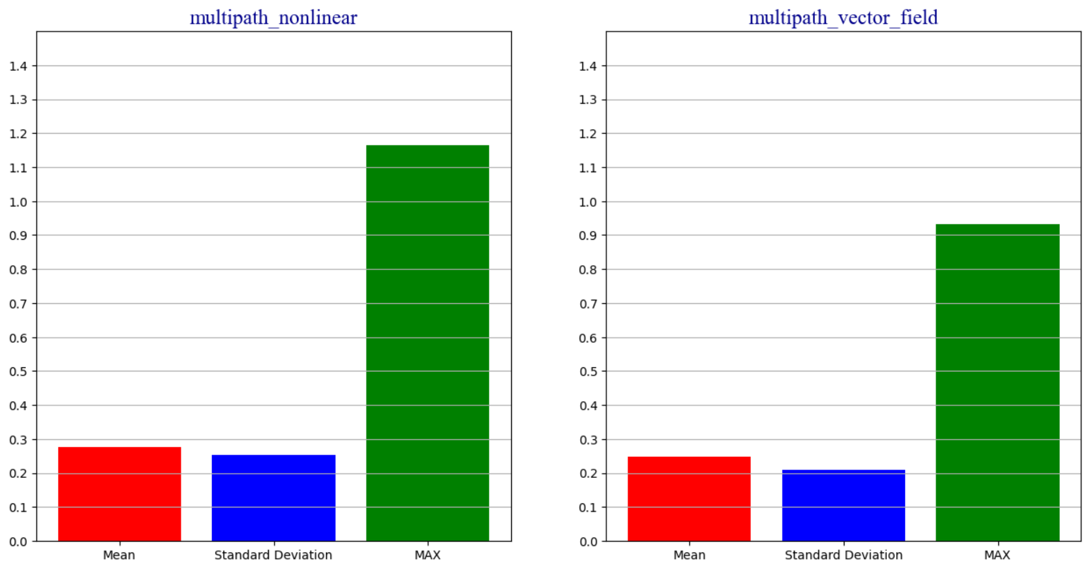


Figure 121 - Multipath result

It is worth noting that the performance collected must be used just as indicators since better results can be achieved through a more precise calibration of the value of the PIDs, only partially done for time reasons.

Section 7.4 Obstacle Avoidance

The last section relates to obstacle avoidance performance; as for the previous ones, all the algorithms are tested in the same condition. However, differently from before, no parameters are computed as indicators. Instead, the graphs are used for this scope.

In both cases, the algorithms could make the robot avoid the obstacle in reaching the goal. Both the trajectory computed will match the map of the field obtained at the end of the test.

Furthermore, unlike the simulation, only one obstacle was on the path for two reasons: the difficulty of building obstacles for the machine in the exam and the relatively small space to do the tests.

The result of the vector field is shown below in *Figure 122*.

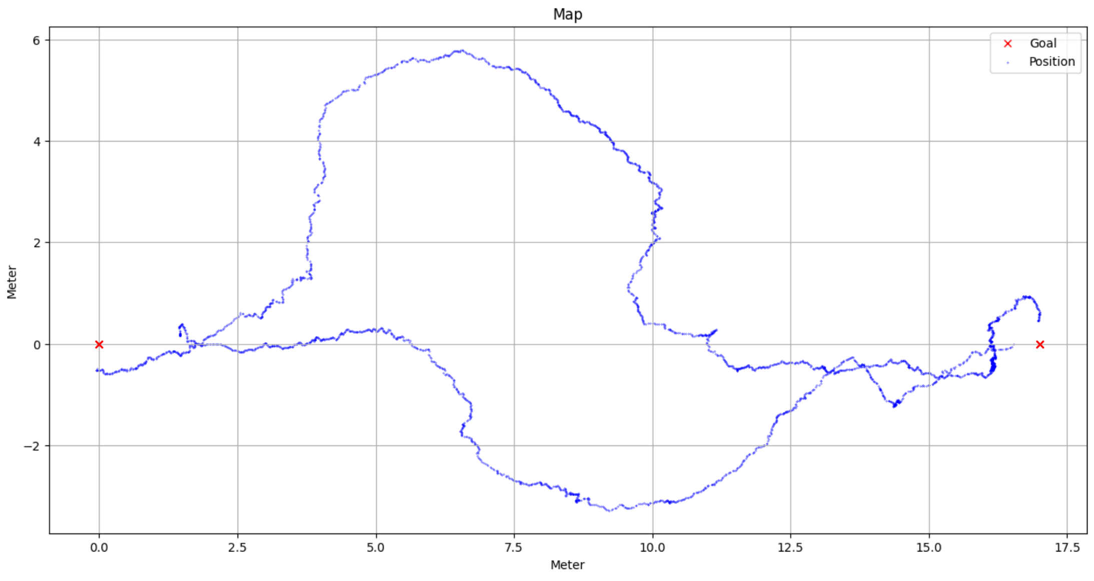


Figure 122 - Vector Field Real test

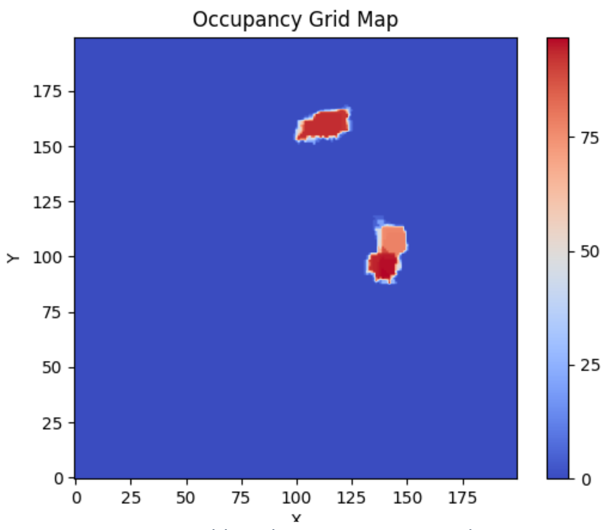


Figure 123 - Vector Field Real test Occupancy grid map

The same is done for the non-linear controller in which the path is computed thanks to the *A* algorithm*, *Figure 124*.

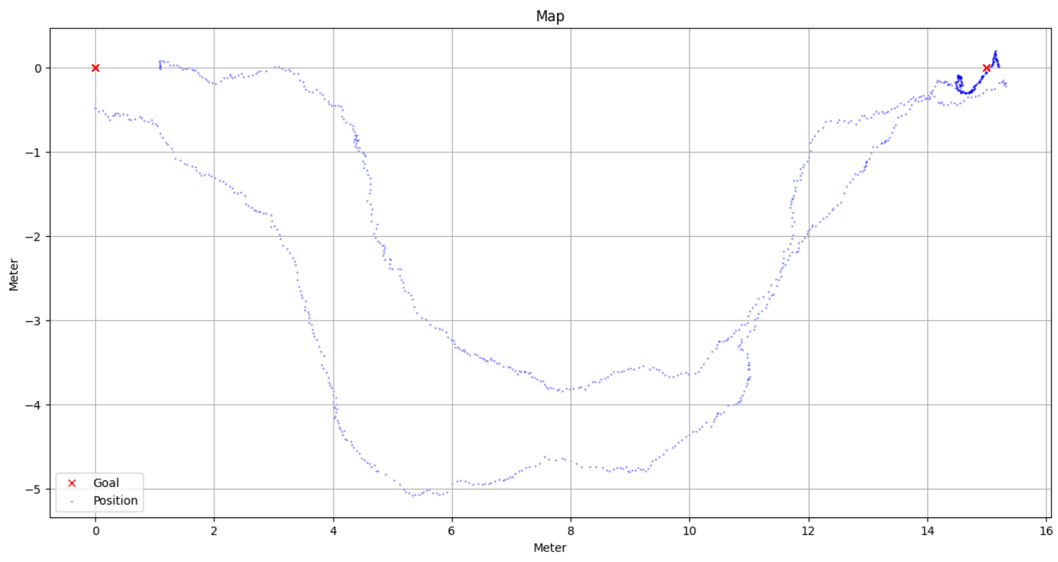


Figure 124 - A algorithm Real test*

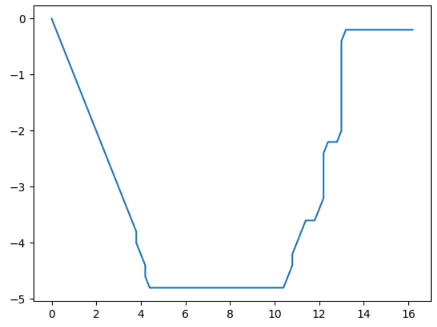


Figure 125 - A algorithm Real test reference path*

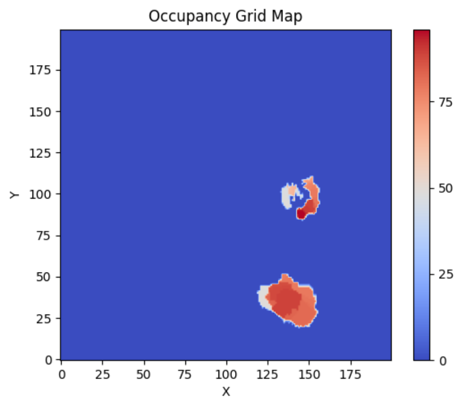


Figure 126 - A algorithm Real test Occupancy grid map*

The results show that both complete the task successfully, even though better results can be achieved by improving the parameters chosen. In particular, as regards the *A* algorithm*, the weights in the eucharistic function can be tuned better to achieve a faster return to the reference path; despite that, it maintains an advantage concerning the vector field since it computes the optimised path while the *Vector Field* always creates a counter-clockwise vector field, making the path longer in some situations.

To conclude, images from the obstacles avoidance test are reported in *Figure 127*:

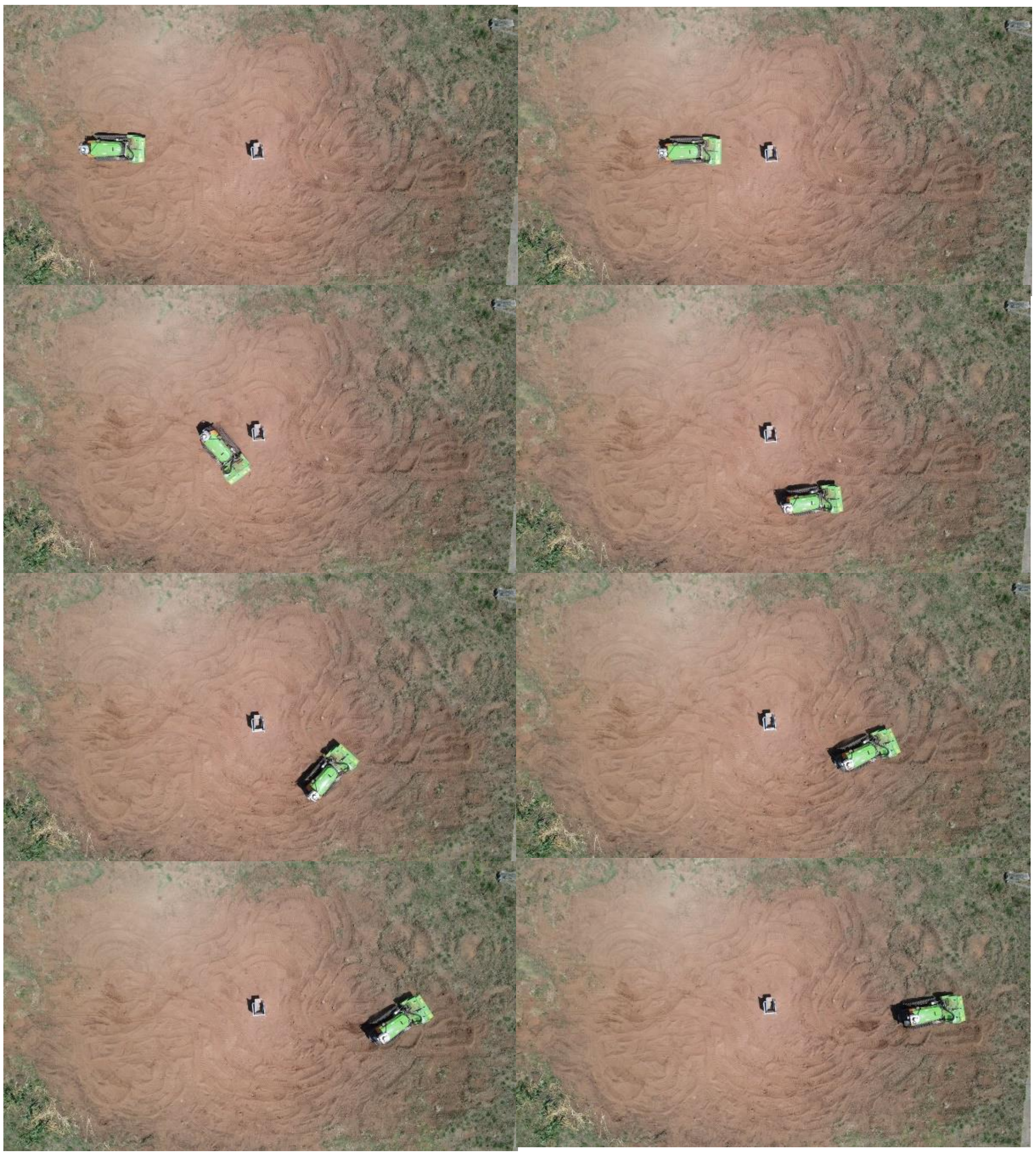


Figure 127 - Obstacle avoidance instants

Chapter 8 Conclusion and Future Directions

This work wants to show an overview of all the aspects related to the development of a UGV, going deeper into the motion aspects. Indeed after the first part, needed to have a base knowledge about how navigation is done and what ROS is, an analysis of the sensors and the methodology used is done.

Then the work moves to the motion part showing different algorithms for motion control and finally proposing two solutions to make the robot navigate autonomously, avoiding obstacles.

All the different solutions are followed by data collected during the test, making available comparisons.

As regards future development, different aspects can be highlighted. Firstly work can be done to increase the performance of the other sensors, fusing them to increase the precision and reliability of the system. In this process, a first start should be using the FLIR during navigation to use LIDAR and the cameras better to perceive the world and the robot's localisation.

Regarding the navigation, improving the A* algorithm to a Lite version will make the computational process faster, increasing the velocity of computation of the new path.

Finally, a study about the tool must be done to detect its position, remove the sensor noise produced by its activation and understand how and when the vegetation is cut.

Bibliography

[1] Raj, R., & Kos, A. (2022). A Comprehensive Study of Mobile Robot: History, Developments, Applications, and Future Research Perspectives. Applied Sciences, 12(14), 6951. https://doi.org/10.3390/app12146951

[2] https://www.eea.europa.eu/ims/forest-fires-in-europe

[3] https://www.wri.org/insights/global-trends-forest-fires

[4] https://en.wikipedia.org/wiki/Unmanned_ground_vehicle

[5] https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications

[6] https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

[7] https://robo-team.com/products/probot/#s-1

[8] https://www.mdbsrl.com/prodotto/lv-600-pro/8/#home

[9] L. Schulze and A. Wullner, "The Approach of Automated Guided Vehicle Systems," 2006 IEEE International Conference on Service Operations and Logistics, and Informatics, Shanghai, China, 2006, pp. 522-527, doi: 10.1109/SOLI.2006.328941.

[10] Karur, K., Sharma, N., Dharmatti, C., & Siegel, J. E. (2021). A Survey of Path Planning Algorithms for Mobile Robots. Vehicles, 3(3), 448-468. https://doi.org/10.3390/vehicles3030027

[11] https://www.geeksforgeeks.org/videos/a-search-algorithm/

[12] McGuire, K., de Croon, G., & Tuyls, K. (2019). A comparative study of bug algorithms for robot navigation. Robotics and Autonomous Systems, 121, 103261. https://doi.org/10.1016/j.robot.2019.103261

[13] Kitts, C., Mahacek, P., Adamek, T., Rasal, K., Howard, V., Li, S., Badaoui, A., Kirkwood, W., Wheat, G., & Hulme, S. (2012). Field operation of a robotic small waterplane area twin hull boat for shallow-water bathymetric characterization. Journal of Field Robotics, 29(6), 924-938. https://doi.org/10.1002/rob.21427

[14] D. R. Nelson, D. B. Barber, T. W. McLain and R. W. Beard, "Vector field path following for small unmanned air vehicles," 2006 American Control Conference, Minneapolis, MN, USA, 2006, pp. 7 pp.-, doi: 10.1109/ACC.2006.1657648.

[15] Vector Field UAV Guidance for Path Following and Obstacle Avoidance with Minimal Deviation, Jay P. Wilhelm and Garrett Clem, Journal of Guidance, Control, and Dynamics 2019 42:8, 1848-1856

[16] http://wiki.ros.org/ROS/Introduction

[17] https://store.arduino.cc/products/arduino-mega-2560-rev3

[18] http://wiki.ros.org/rosserial

[19] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

[20] https://www.swiftnav.com/sites/default/files/duro_product_summary.pdf

[21] https://github.com/szenergy/duro_gps_driver

[22] https://github.com/CCNYRoboticsLab/imu_tools

[23] Valenti, R. G., Dryanovski, I., & Xiao, J. (2015). Keeping a Good Attitude: A Quaternion-Based Orientation Filter for IMUs and MARGs. Sensors, 15(8), 19302-19330. https://doi.org/10.3390/s150819302

[24] Kuncar, A., Sysel, M., Urbanek, T. (2016). Calibration of Triaxial Accelerometer and Triaxial Magnetometer for Tilt Compensated Electronic Compass. In: Silhavy, R., Senkerik, R., Oplatkova, Z., Silhavy, P., Prokopova, Z. (eds) Automation Control Theory Perspectives in Intelligent Systems. CSOC 2016. Advances in Intelligent Systems and Computing, vol 466. Springer, Cham.

[25] Pang, H., Li, J., Chen, D., Pan, M., Luo, S., Zhang, Q., & Luo, F. (2013). Calibration of three-axis fluxgate magnetometers with nonlinear least square method. Measurement, 46(4), 1600-1606. https://doi.org/10.1016/j.measurement.2012.11.001

[26] Mateos, I., Ramos-Castro, J., & Lobo, A. (2015). Low-frequency noise characterization of a magnetic field monitoring system using an anisotropic magnetoresistance. Sensors and Actuators A: Physical, 235, 57-63. https://doi.org/10.1016/j.sna.2015.09.021

[27] https://codemonk.in/blog/moving-average-filter/

[28] https://www.samproell.io/posts/yarppg/yarppg-live-digital-filter/

[29] http://wiki.ros.org/robot_pose_ekf

[30] Moore, T., Stouch, D. (2016). A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In: Menegatti, E., Michael, N., Berns, K., Yamaguchi, H. (eds) Intelligent Autonomous Systems 13. Advances in Intelligent Systems and Computing, vol 302. Springer, Cham. https://doi.org/10.1007/978-3-319-08338-4_25

[31] https://velodynelidar.com/products/puck/#downloads

[32] http://wiki.ros.org/velodyne

[33] https://github.com/yzrobot/adaptive_clustering

[34] Yan, Z., Duckett, T. & Bellotto, N. Online learning for 3D LiDAR-based human detection: experimental analysis of point cloud clustering and classification methods. Auton Robot 44, 147–164 (2020). https://doi.org/10.1007/s10514-019-09883-y

[35] https://www.intelrealsense.com/depth-camera-d435/

[36] Horvat, Marko & Jelečević, Ljudevit & Gledec, Gordan. (2022). A comparative study of YOLOv5 models performance for image localization and classification.

[37] Xu, Renjie & Lin, Haifeng & Lu, Kangjie & Cao, Lin & Liu, Yunfei. (2021). A Forest Fire Detection System Based on Ensemble Learning. Forests. 12. 217. 10.3390/f12020217.

[38] https://github.com/ultralytics/yolov5

[39] https://www.flir.com/products/adk/

[40] https://en.wikipedia.org/wiki/Bang%E2%80%93bang_control#

[41] https://en.wikipedia.org/wiki/PID_controller

[42] F. M. Noori, D. Portugal, R. P. Rocha and M. S. Couceiro, "On 3D simulators for multi-robot systems in ROS: MORSE or Gazebo?," 2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR), Shanghai, China, 2017, pp. 19-24, doi: 10.1109/SSRR.2017.8088134.

[43] https://github.com/leonhartyao/gazebo_models_worlds_collection/blob/master

[44] http://wiki.ros.org/Books/ROS_Robot_Programming_English