POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Design of a Behavior-Based Navigation Algorithm for Autonomous Tunnel Inspection

Supervisors
Prof. Marcello Chiaberge
Dott. Andrea Eirale
Dott. Marco Ambrosio

Candidate
Riccardo Tassi

December 2022

Summary

The modern world is characterized by the presence of a significant amount of subterranean infrastructures, including networks of tunnels, caverns, and the urban underground. Each of these environments offers intricate settings that could create difficulties for exploration, inspection, and several other activities. Conditions might deteriorate and vary over time, and there may be various risks. Most of the time, this confluence of difficulties and dangers creates circumstances that are too dangerous for workers. Situations like gas leaks, explosions, rock falls, confinement, and prolonged exposure to dust are all potentially lethal.

Robotic solutions are thus required to operate when and where human risk is too high. Autonomous robots can lower risks by taking over potentially hazardous jobs for workers. For instance, an autonomous robot can do tasks like assessing the air quality or checking the conditions in hazardous mines. To achieve this goal, a robot platform must be developed with a series of sensors and algorithms that allow it to navigate autonomously inside the tunnel, collecting the necessary data without any human intervention.

This thesis project intends to contribute to this field by designing a robotic platform with the least amount of sensors and algorithms to autonomously accomplish the tunnel inspection task. Specific requirements and environmental difficulties guided the design phase: the rover must be able to cover the entire unknown tunnel plan to perform a good inspection, in the presence of challenging terrains, low-light visibility, and GPS-denied environments. The selected robotic platform consists of a Clearpath Husky rover. A LiDAR sensor is employed to perceive walls and obstacles, while localization is achieved by fusing IMU and encoder data.

The main effort was focused on developing the navigation algorithm. It consists of a behaviour-based navigation algorithm that does not require a global map of

the environment to work. A state machine interprets LIDAR data and processes specific instantaneous paths tracked by the robot through a Pure Pursuit controller. The paths generation is designed to let the robot always keep the left wall, in such a way, the exploration of the entire tunnel plan is ensured.

The navigation algorithm has been tested in simulation with three tunnel models with different sizes and characteristics. In all models tested the robot successfully covered the entire tunnel plans and returned to the starting point. Moreover, the entire system was also successfully deployed and tested on the robotic platform in a real environment.

Table of Contents

List of Tables

List of Figures

Acronyms

IMU

Inertial Measurement Unit

LIDAR

Light Detection And Ranging

ODR

Orthogonal Distance Regression

PPA

Pure-Pursuit Algorithm

RF

Reference frame

ROS

Robotic Operating System

Chapter 1

Introduction

1.1 Autonomous Navigation Problem

In the recent years, autonomous navigation has gained popularity in the community of Robotics. It is one of the most challenging competences required of a mobile robot. Success in navigation requires success at the four building blocks of navigation: perception, localization, cognition and motion control [1]. Figure 1.1 shows a general control scheme for mobile robot comprising all the above functional blocks.

Perception concerns how the robot perceives its surrounding environment interpreting sensors measurement and extracting from them meaningful data. Motion control is the module that allows the robot to modulate its motor outputs to achieve the desired motion. Localization is the capability of the robot to determine its position with respect to a fixed reference frame. Cognition is the elaboration process of the perception information – both about the external environment and about the robot itself – and the consequent decision-making and execution tasks that the robot actuates to achieve its goals. Control architectures specify how these functional blocks should be combined to achieve the desired results; each one suggests new notions and solutions to the navigation problem. The control architectures could be classified into three categories [2]:

- Deliberative (Centralized) navigation
- Reactive (Behaviour-based) navigation

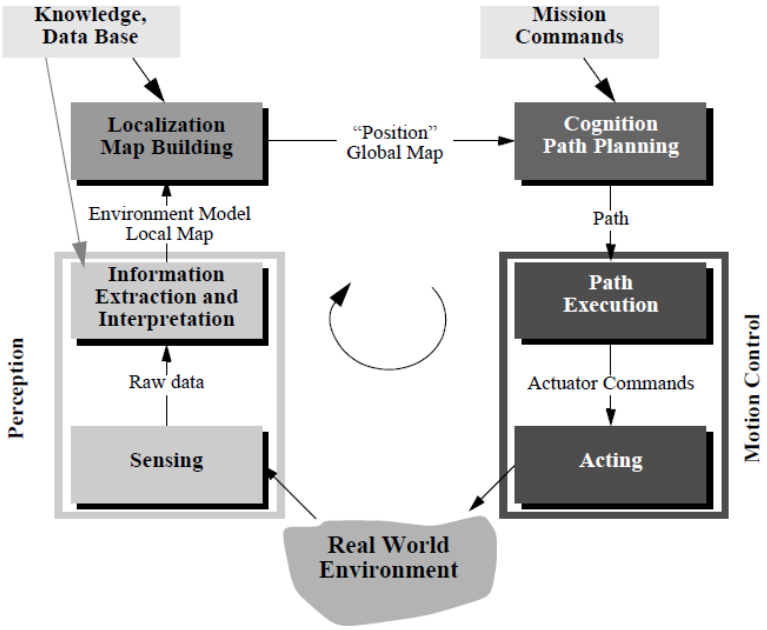


Figure 1.1: General control scheme for a mobile robot [1]

- Hybrid (Deliberative - Reactive) navigation

The oldest schema is the Deliberative (Centralized) navigation architecture. It generates appropriate actions for the mobile robot to approach the target using a global world model provided by human input or sensory inputs. Deliberative control architecture is made up of three modules: sensing, planning, and action. The first robot senses its surroundings and builds a map of the static world by merging sensory data. The path planning module is then used to find the best way to the target and develop a plan for the robot to follow. Finally, the robot performs the appropriate movements in order to reach the target. Following a successful action, the robot comes to a halt and updates information in order to perform the following motion. The technique is then repeated until the target is reached. In deliberative navigation, however, a precise model of the environment is required to determine a globally feasible path. Massive computing power and memory are required to complete the necessary calculations. Furthermore, the top-down method to planning causes delays in the navigation process, and if any modules fail to work properly, the system may fail completely. Figure 1.2 shows

the typical deliberative architecture.

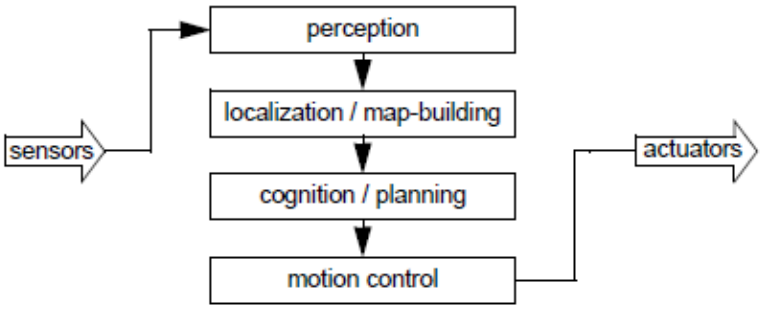


Figure 1.2: Deliberative architecture [1]

In literature there are a great variety of global path planning algorithm which can be classified on the basis of how the environment is transformed into a discrete map suitable for the chosen path-planning algorithm into two main categories [1]:

- Graph search: a connectivity graph in free space is first constructed offline through a graph generation method (e.g. visibility graph, Voronoi diagram, cell decomposition, lattice graph) and then searched. The searching algorithm are again classified int two classes:

  - Deterministic graph search: Whatever map representation is used, the purpose of path planning is to discover the best path between the start and the goal in the map's connectivity graph, where best refers to the selected optimization criterion. The main algorithm of this class are Breath first search, Depth first search, Dijkstra's algorithm, A* algorithm and D* algorithm.

  - Randomized graph search: When confronted with sophisticated high-dimensional path planning issues, exhaustively solving them within reasonable time constraints becomes impossible. In such cases, randomized search is advantageous since it sacrifices solution optimality for faster solution computing. An example is the Rapidly Exploring Random Trees (RRTs).

- Potential field path planning: Potential field path planning involves the creation of a field, or gradient, across the map of the robot that drives the robot to the destination position from many preceding positions. The potential field method considers the robot to be a point influenced by an artificial potential field. The goal (a minimum in this area) attracts the robot, whereas the obstacles act as repulsive forces.

To address deliberative navigation challenges in dynamic and unfamiliar surroundings, reactive (behavior-based) navigation architecture was developed. These methods create control directives based on current sensory data. To take actions, the robot relies on a local model of the environment rather than a planning process. As a result, it is not necessary to create a complete environment model. In a dynamic and uncertain environment, reactive navigation responds quickly. Figure 1.3 represents the overall architecture of behaviour-based approaches. The robot gathers sensory input at the first layer. Then, a transfer function known as behavior accepts certain sensory inputs perception and converts them into the specified reaction. Finally, based on the results of active behavior, the robot performs an action. In reality, difficult navigation tasks are broken down into multiple simpler and smaller sub-level jobs that increase the navigation system's overall performance.

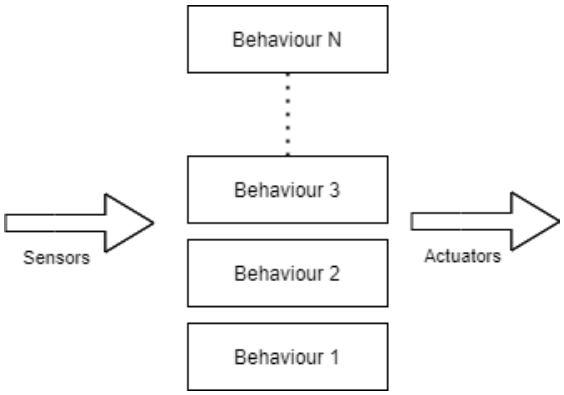


Figure 1.3: General reactive architecture [1]

There are two basic behavior-based control architectures: the subsumption and the motor schema. The subsumption behavior-based control system is made up of numerous layers of task-accomplishing behaviors, each of which can receive sensory information for a specific task (obstacle avoidance, wall following, target seeking,

etc.). Priority-based arbitration is referred to as behavior layer coordination. When numerous conflicting behaviors are activated, priority-based arbitration determines which behavior will be active. As a result, the architecture's overall output is generated by the most active behavioral module. Figure 1.4 shows the subsumption architecture.

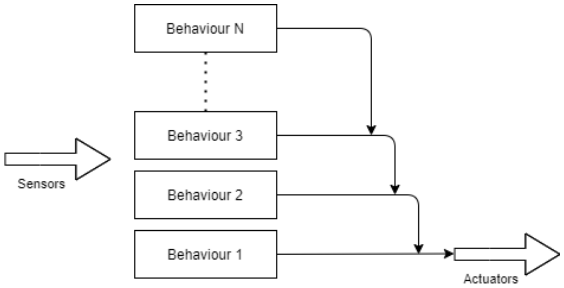


Figure 1.4: Subsumption architecture [2]

The motor schema theory describes motor behavior in terms of several activities being controlled at the same time. Each behavior can generate a vectorial output. These outputs are pooled, and the system's total reaction is obtained through vector summation. The subsumption architecture promotes competitive behavior selection, whereas the motor schema relies on cooperative coordination. Motor schema allows you to use the outputs of several behaviors at the same time while capturing their specific influence on total output. Figure 1.5 shows the subsumption architecture.

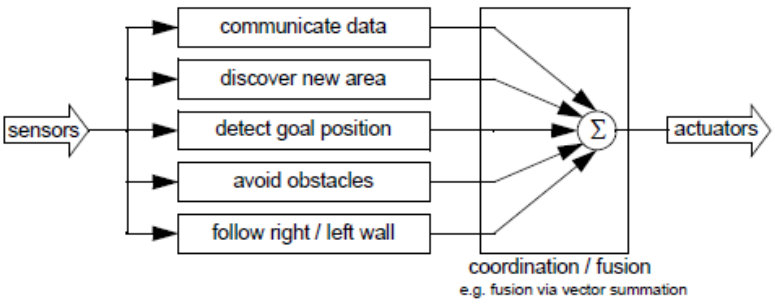


Figure 1.5: Motor schema architecture [1]

From the 1980s, reactive (behavior-based) architectures were developed to address the inadequacies of deliberative techniques in dynamic and unknown

contexts. Based on the present perception of the environment, these designs create control commands. As a result, it is not necessary to construct a complete model of the environment, and the sensed input is immediately coupled to the robot's actuators via a specific set of transfer functions known as task accomplishing modules or behaviors.

Hybrid control architecture is the earliest proposed control schema. It combines the benefits of deliberative designs with the quick response of reactive structures in a dynamic or unpredictable environment. The common hybrid control architectures consist of three layers: deliberative layer, control execution layer and reactive (behavior-based) layer (Figure 1.6). For high-level concerns, deliberative navigation is used to build an optimal approach. Sensor fusion, map construction, and planning are the high-level restrictions. The optimal commands from the higher level are then transferred to the reactive layer to generate the robot's action. The execution layer (behavior-coordinator) is in charge of supervising the interaction between the high level layer and the low level layer.

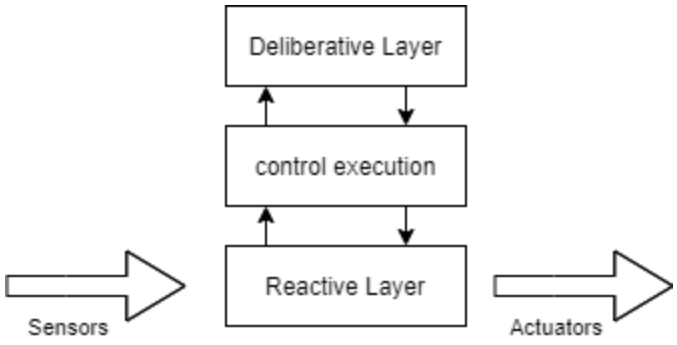


Figure 1.6: Hybrid control architecture [2]

And so, as regards this latest architecture, integration between global and local path planning algorithms is necessary. In this context, some used local path planning, used as obstacle avoidance, is the Bug algorithm, Vector field histogram, the bubble band technique, curvature velocity technique, dynamic window approach, the gradient method, etc.

Before going on, a point must be clarified. The localization building block may seem mandatory in order to navigate successfully but it is not always true. In fact, it is necessary for the map-based architectures, but it can be avoided by the

behavior-based one since this approach avoids explicit reasoning about localization and position, and thus generally avoids explicit path planning as well.

1.2 Motivation and Aim of the work

One of the most difficult issues engineers confront is inspecting, assessing, maintaining, and operating underground infrastructure safely. Networks of mining tunnels, caves, and the urban underground (e.g. subway tunnels and underground shopping malls) are examples [3]. Each of these environments offers intricate settings that could create difficulties for exploration, inspection, and several other activities. Conditions might deteriorate and vary over time, and there may be various risks. Most of the time, this confluence of difficulties and dangers creates circumstances that are too dangerous for workers. Situations like gas leaks, explosions, rock falls, confinement, and prolonged exposure to dust are all potentially lethal. Robotic solutions are thus required to operate when and where human risk is too high. Autonomous robots can lower risks by taking over potentially hazardous jobs for workers. For instance, an autonomous robot can do tasks like assessing the air quality or checking the conditions in hazardous mines. To achieve this goal, a robot platform must be developed with a series of sensors and algorithms that allow it to navigate autonomously inside the tunnel, collecting the necessary data without any human intervention [3]. Having said this, the goal of this work is the design of a robotic platform equipped with the least number of sensors and algorithms to carry out autonomous inspection of underground tunnels. Given the context in which this project takes place, a series of requirements and critical issues have been identified that have guided the design of the platform. First of all, we must consider all the criticisms inherent in the work environment, namely the possible presence of rough terrain, the absence of light, and the absence of GPS. Taking into account these environmental requirements, the selected components of the robotic platform are: a 4-wheeled differential drive mobile robot, LiDAR, IMU, wheel encoders, and a camera used to record a video of the tunnel during navigation that will serve for an offline inspection. Given the possible absence of light, the robot will also be equipped with spotlights to illuminate the environment.

The main effort in this work has been focused on the development of a navigation

algorithm that allows the autonomous inspection of the tunnel. In the design of the algorithm it has been taken into consideration as a fundamental requirement that, in order to carry out an exhaustive inspection, the robot has to go through the whole plan of the tunnel without knowing a priori the map and trying to stay as much as possible in the center of the road that it is following. So, the goal is not to reach a specific point, but to turn the whole tunnel and return to the starting point, possibly stopping once reached. The implemented algorithm follows a behavior-based architecture: sets of behavior that together result in the desired robot motion are designed. Specifically, the algorithm takes data input from the LiDAR, a state machine processes them, and generates each second a specific path that the robot has to follow. The logic according to which the state machine generates the path is to always follow the left wall so that the robot can always manage to exit the tunnel. As previously explained, usually this kind of approach doesn't need a localization block, in fact, the local paths can be defined with respect to the local frame of the robot through a geometric interpretation of the laser data. In this case, localization is also part of the problem because, in order to stop at the starting point, the robot must know its position with respect to a global reference frame. Since the tunnel is a GPS-denied environment, the localization is performed by fusing sensor data from IMU and wheel encoders.

During the work Robotic Operating System 2 (ROS2) has been used as a framework for the development of the algorithm and for its deployment on the real robot. Simulations have been performed in Gazebo to test the algorithm with tunnel models made in Blender. Finally, the algorithm has been also tested in real-world scenarios created in the corridors of the PIC4SeR (PoliTO Interdepartmental Center for Service Robotics).

This thesis is divided into 3 macro chapters following the structure of a scientific paper (Materials and Methods, Results, Conclusions and Future Developments). The first part is the core of the thesis, it introduces the development and simulation environments, it deepens the selected robotic platform and all its components, it describes in detail the working principles of the proposed navigation algorithm and finally it describes the tests made in simulation and in real scenarios. The last two show the results obtained during the tests and conclude the thesis with a series of considerations for possible future developments of the platform.

Chapter 2

Materials and Methods

2.1 Work Environment

This section presents the work environment of this thesis project. Sections 2.1.1 and 2.1.2 explain the main features of ROS and Gazebo which which had been the virtual environment used throughout this thesis study for algorithm implementation, simulations, and deployment on the physical robot.

2.1.1 Robotic Operating System 2

Robot Operating System or ROS, is a meta-operating system for developing robotic applications. It is an open source collection of tools, libraries and conventions that help to write robot specific software in an organized and modulated way. It provides the typical services of an operating system, including hardware abstraction, low-level device control, message-passing between processes, and package management [4]. Since it is not an actual operating system, it must be installed on an existing one, such as Ubuntu, a Linux distribution.

This peer-to-peer framework simplifies the user's task of interfacing with several sensor modules and robot platforms representing a key instrument in software integration of the systems. For example, interfacing or retrieving sensor data in a sophisticated robot system might be difficult due to the various types of sensors employed and the various formats of data representation for each sensor. ROS provides a consistent data representation format for each sensor, making it simple

for users to extract and analyze data from various sensor systems. ROS also includes software packages for most typical sensors that allow users to directly interact with the sensor and get data, reducing the user's workload when communicating with and reading data from a sensor. Another significant challenge in designing mobile robots is sending commands from software to hardware motors, but now, most mobile platforms have software packages to send control commands thanks to ROS. Following is a list of basic ROS terminology and architectural components:

- **Nodes**: nodes are the basic units of a ROS application. They are single-purpose, executable programs written in Python or C++. The whole robotic application is composed by a series of nodes communicating with each other. The nodes sending data are called publishers while the receiving one are called subscribers.
- **Messages**: They are the mean through which nodes send data to each other; they are data structure of a specific type, either standard (Boolean, integer, float, etc.), or application-specific (Twist, Pose, etc.).
- **Topics**: Topics are the means through which nodes communicate. Communication over topics occurs in a unidirectional manner using Publish/Subscribe logic: a publisher node publishes a message over a topic, and all nodes subscribing to that topic get that message.
- **Services**: Unlike topics, services enable synchronous Request/Response communication across nodes: a service server node answers only when a request is received from a service client node, which can both submit and receive requests. The connection between two nodes is lost once the service request and response are done.
- **Actions**: Action clients, like services, send a request to an action server in order to achieve a goal and receive a response. Unlike services, an action server delivers a feedback to the client as the action is being done. As a result, actions are employed when a response may take a long time.

It is thus possible to imagine a robot as a graph comprised of nodes exchanging messages with each other through topics, services or actions. Figure 2.1 shows an example of the communication between nodes in ROS trough a topic and a service.

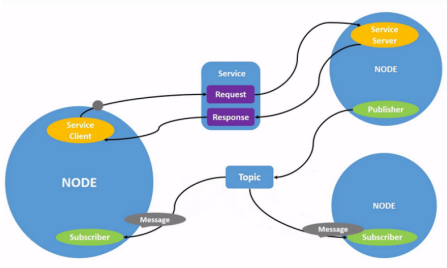


Figure 2.1: Nodes communication example [4]

ROS provides several tools useful for analysis and debugging, giving users the possibility of getting visual feedback on what is happening within the system. The ones used during this project are listed below:

- **rqt_graph**: This tool may graphically depict the correlation between the processes (nodes and topics) that are active at the moment of execution. In this manner, debugging analysis may be performed to ensure that the nodes are appropriately run, in addition to quickly visualizing which nodes are functioning as publishers and which are acting as subscribers. Figure 2.2 shows an example of rqt_graph, here the nodes are represented with circles while topics with rectangles.

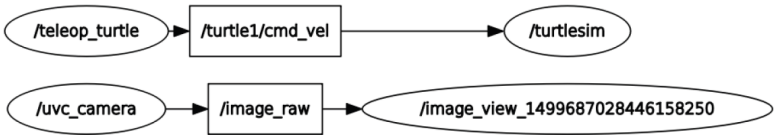


Figure 2.2: Example of rqt_graph [5]

- **RViz**: It allows to visualize within a three-dimensional space any data the software publishes through its topics. For example, it can visualize the distance from the sensor of a Laser Distance Sensor to an obstacle, the Point Cloud Data of the 3D distance sensor or the image value obtained from a camera, and many more without having to separately develop the software [5]. In a sense represents the robot's point of view, allowing to easily visualize what the robot is sensing. Figure 2.3 represents an example of the RViz window. It shows the model of the robot, the points identified by a Laser Distance Sensor

(red points) and the path the software is sending to the robot in order to track it (green line).

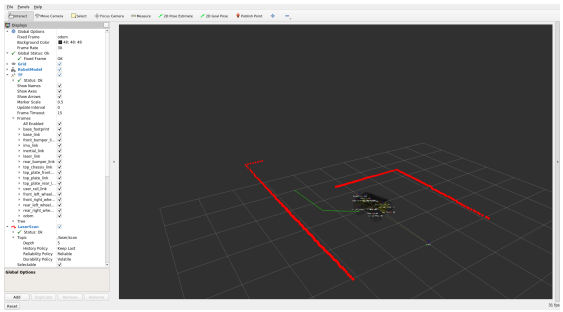


Figure 2.3: Example of RViz window

- **rqt_plot**: This tool is used for plotting 2D data, it is useful to visualize how signals varies over time. For example, it is particularly suitable for displaying the sensor value over a period of time, such as speed and acceleration. Figure 2.4 shows an example of signals visualized with rqt_plot.

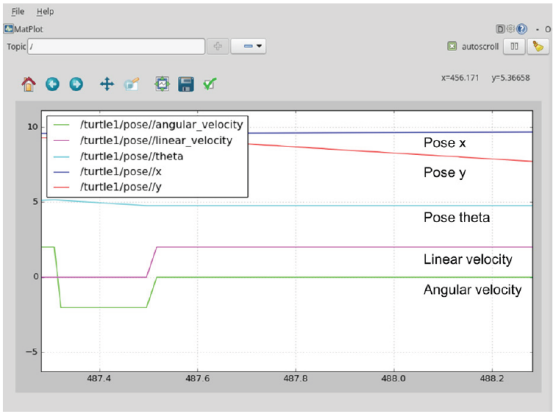


Figure 2.4: Example of rqt_plot [5]

2.1.2 Gazebo

When developing ROS-based robotic applications, a realistic and responsive simulation environment is essential for visualising complicated behaviours such as an autonomously navigating robot. For this thesis project, Gazebo simulator has been chosen. It is a 3D dynamic simulator with the ability to accurately and efficiently

simulate populations of robots in complex indoor and outdoor environments. It supports four physics engine, but only one is compatible with ROS: ODE (Open Dynamics Engine) which is an open-source, high-performance library for simulating rigid body dynamics with applications ranging from games to 3D animation tools and industrial simulators. Gazebo also uses advanced 3D graphics by relying on OGRE (Open-source Graphics Rendering Engines), which enables the generation of realistic environments using textures, lights and shadows. Thus, it provides high-fidelity physics simulation, as well as environment models and support for a large range of sensors and real robot models. In addiction, noise can be applied to the sensor data making the simulation as realistic as possible. Custom sensors, models, and scenarios can also be created through the use of plugins, files in .sdf format, and files in .world format, respectively [6]. ROS works closely with Gazebo via the Gazebo_ros package. This package includes a Gazebo plugin module that allows Gazebo and ROS to communicate bidirectionally. Gazebo can transmit simulated sensors and physical data to ROS, and ROS can stream actuator commands back to Gazebo. These characteristics allow extremely accurate testing of robotics algorithms and execution of tests with realistic scenarios. Figure 2.5 shows an example of the Gazebo view representing the model of the Husky robot in an empty scenario. The husky_gazebo package was used to simulate Husky in Gazebo [7].

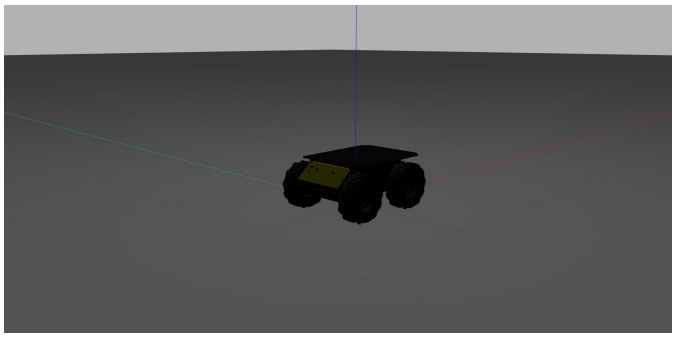


Figure 2.5: Example Gazebo graphic

In order to test the navigation algorithm in simulation, various models of tunnels and scenarios with obstacles have been created. Obstacle scenarios are simple models created directly in the Gazebo model editor using a series of cubes and then saved as .world files. Instead, the tunnel models have been realized in Blender

which is a free and open-source 3D computer graphics software tool set used for creating animated films, visual effects, art, 3D-printed models, motion graphics, interactive 3D applications, virtual reality and video games [8]. In this case it has been used only as a 3D modelling tool; the resulting models have been exported in Gazebo as .dae files and than saved as .world ones so as to be used in simulations. The pictures of the models will be presented in section 2.5.

2.2 Robotic Platform

This section will present in detail all the components making up the robotic platform which are needed to accomplish the autonomous navigation task. It will deepen the mobile robot chosen and its locomotion system, and it will also talk about the sensing elements equipped in the robotic platform used to acquire knowledge about its environment by extracting meaningful information from the measurements. All the composing elements are:

- **Robot model**: It is the Clearpath Husky which is a four wheeled differential drive mobile robot equipped with precise wheel encoders providing a raw odometry measure.
- **IMU**: The one used is the UM7, it is electronic device composed by a series of sensors that measures important robot quantities such as angular rate, linear velocity and linear acceleration.
- **LIDAR**: It is a sensor which permits to determine the distance of an object or surface using a laser pulse. The one selected within this project is the RPLIDAR A2M8.

In this list, and in the following subsections, it is not present the model of the camera, this is because it is used only for the inspection and not for navigation purpose.

2.2.1 Robot Model and Kinematic Equations

The Husky of Clearpath Robotics has been chosen in the wide panorama of ground mobile robots. It is a medium-sized robotic platform that, thanks to its high-performance, maintenance-free drive-train and wide lug-tread tyres, enables Husky to tackle difficult real-world terrain such as that found inside tunnels [9]. Husky also has very high resolution encoders that deliver improved state estimation and dead reckoning capabilities. Another reason this model was chosen is that it supports ROS from its factory settings, in fact it has an onboard computer which runs Ubuntu with ROS installed with it plus appropriate ROS packages and other configuration for use with Husky. Since it was designed to be tele-operated or

driven in an autonomous mode, it is possible for the user to send input control commands with a remote computer to the on board one which communicates with the motor controllers using an RS232 protocol. In this manner, the internal software executes the specified command while the user monitors the robot's performance on his computer in real-time.

Figure 2.6 shows the Husky platform with sensors in real environment and Figure 2.7 shows its dimensions.

Figure 2.6: Husky platform in real environment [10]

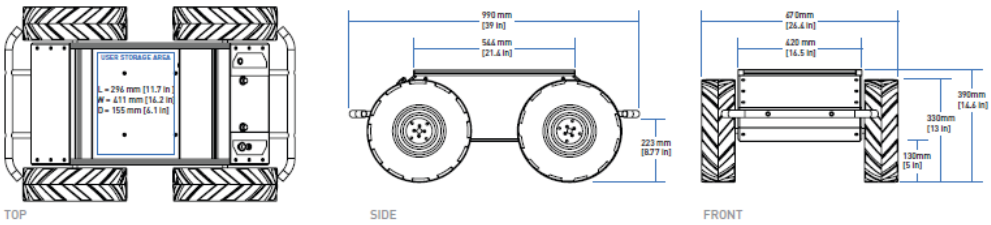


Figure 2.7: Husky dimensions [9]

The locomotion system of the Husky is called differential drive. Recently, the interest of the differential drive wheeled mobile robots are increasingly used in many applications due to its advantages such as flexible motion capability, simple and robust structure, lower costs [11]. In fact, unlike steering wheel mobile robots,

they don't have an explicit steering mechanism; instead, steering is accomplished by independently controlling the speed and direction of the wheels rotation on each side of the vehicle; if the rotations are not equal, the vehicle will turn, causing the wheels to skid on the ground [10]. This way of moving significantly improve their maneuverability and traction in loose terrain compared to conventional wheeled designs. The disadvantage of such configurations is coupled to the skid steering. Because of the large amount of skidding during a turn, the exact center of rotation of the robot is hard to predict and the exact change in position and orientation is also subject to variations depending on the ground friction. Therefore, dead reckoning on such robots is highly inaccurate [1]. Anyway, this problem can be overcome by fusing with particular filters data coming from different sensors in order to obtain a better odometry measure, as will be seen in Section 2.3. In the following the kinematic equation of a differential drive robot will be presented. Figure 2.8 represents a differential drive robot in light grey. The vehicle's body coordinate frame is shown in red, and the world coordinate frame in blue.

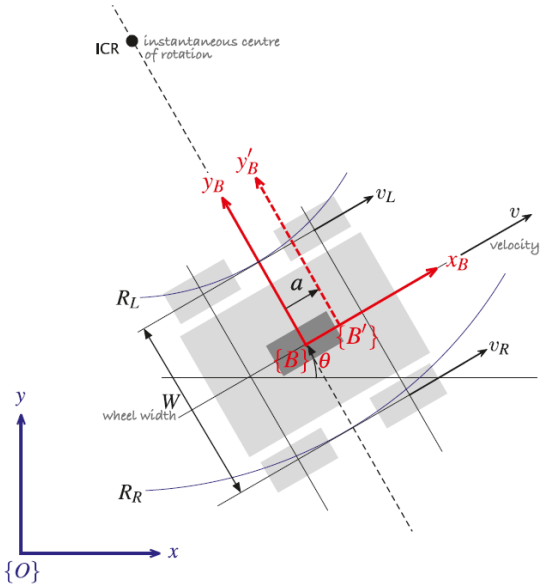


Figure 2.8: Differential drive mobile robot [10]

The vehicle's pose is represented by the body coordinate frame {B}, with its x-axis pointing forward and its origin at the centroid of the four wheels. The configuration of the vehicle is represented by the generalized coordinates q = (x, y,

θ) expressed in the world coordinate frame {O}. The vehicle follows a curved path centered on the Instantaneous Center of Rotation (ICR). The left-hand wheels travel at a speed of v_L along an arc with a radius of R_L while the right-hand wheels travel at a speed of v_R along an arc with a radius of R_R. The angular velocity of {B} is

$$\dot{\theta} = \frac{v_L}{R_L} = \frac{v_R}{R_R} \tag{2.1}$$

and since $R_R = R_L + W$ we can write the turn rate

$$\dot{\theta} = \frac{v_R - v_L}{W} \tag{2.2}$$

where W is the wheels separation. The equations of motion are therefore

$$\begin{aligned} \dot{x} &= v \cos\theta \\ \dot{y} &= v \sin\theta \\ \dot{\theta} &= \frac{v_\Delta}{W} \end{aligned} \tag{2.3}$$

where $v = \frac{v_R + v_L}{2}$ and $v_\Delta = v_R - v_L$ are the average and differential velocities respectively. And so, for a desired speed v and angular velocity $\dot{\theta}$ the necessary turn rate of both the wheels can be computed by rearranging the previous equations, obtaining:

$$\begin{aligned} w_L &= \frac{v - \dot{\theta} \times W/2}{r} \\ w_R &= \frac{v + \dot{\theta} \times W/2}{r} \end{aligned} \tag{2.4}$$

where r is the wheels radius. This is exactly the operations computed by the diff_drive_controller, which is the ROS plugin used to control the Husky in simulation. It takes as input the desired linear velocity and the angular rate, and computes with Eq. 2.4 the necessary wheels angular velocity [12]. Instead for the experimental part the robot is controlled thanks to open-source ROS packages.

2.2.2 LIDAR

As said above, the LIDAR used during the experimental tests is the RPLIDAR A2M8 from SLAMTEC. It is a 2D LiDAR sensor providing laserscan data, the core runs clockwise to perform a 360 degree omnidirectional laser range scanning of the surrounding environment and then it generates a 2D point cloud data that can be used in mapping, localization and object/environment modeling. Figure 2.9 shows a picture of the LiDAR sensor and of the scanning data.

(a) *LiDAR sensor [13]*

(b) *RPLIDAR Scanning Data Coordinate System Definition [13]*

Figure 2.9: LiDAR

The RPLIDAR A2M8 is based on laser triangulation ranging principle and adopts an high-speed vision acquisition and processing hardware. During every ranging process, the RPLIDAR emits modulated infrared laser signal and the laser signal is then reflected by the object to be detected. The returning signal enters a pinhole lens and is sampled by the vision acquisition system, it is then processed by the digital signal processor embedded in the sensors which, finally, outputs distance value and angle value between object and sensor [13]. Figure 2.10 show the LiDAR working mechanism.

The senor operates at 5V, it has a a horizontal FOV of 360 degrees and an angular resolution of 0.45 degrees. The operating range is 0.02m - 12m. During the experimental tests a ROS node implementing the low-level communication between the software and the sensor has been used. The node has been modified in order to have a resolution of 1 degree, in this way each scan gives as output 360

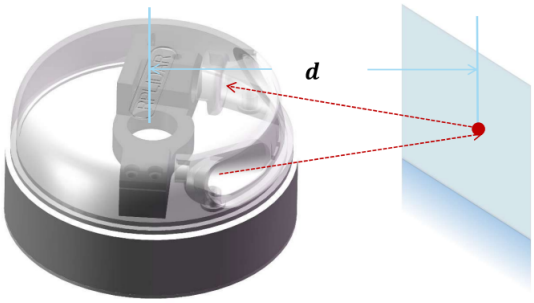


Figure 2.10: LiDAR working mechanism [13]

points simplifying data treatment. Instead, during simulations a Gazebo plugin substituting the sensor has been used. As it will be shown, LiDAR is the main sensor used for the navigation.

2.2.3 IMU

IMU stands for Inertial Measurement Unit, it is a component provided with a triaxial accelerometer, gyroscope and magnetometer. UM7 from Redshift Labs is the orientation sensor used in the robotic platform. It is an Attitude and Heading Reference System (AHRS), it means that it has an embedded IMU inside it providing raw sensors reading which are then combined by the onboard microcontroller to generate orientation estimates 500 times per second. Figure 2.11 shows a picture of the sensor.

Figure 2.11: UM7 sensor

Also in this case the sensors has been simulated in Gazebo through a plugin, instead in the experimental part a specific ROS node has been used implementing the driver for the communication.

2.3 Localization

This section will threat the localization problem in mobile robotics, and then it will present a famous algorithm used to solve this kind of problem performing sensor fusion, i.e the technique used to merge data from different sensors to obtain a more precise measure: the Extended Kalman Filter.

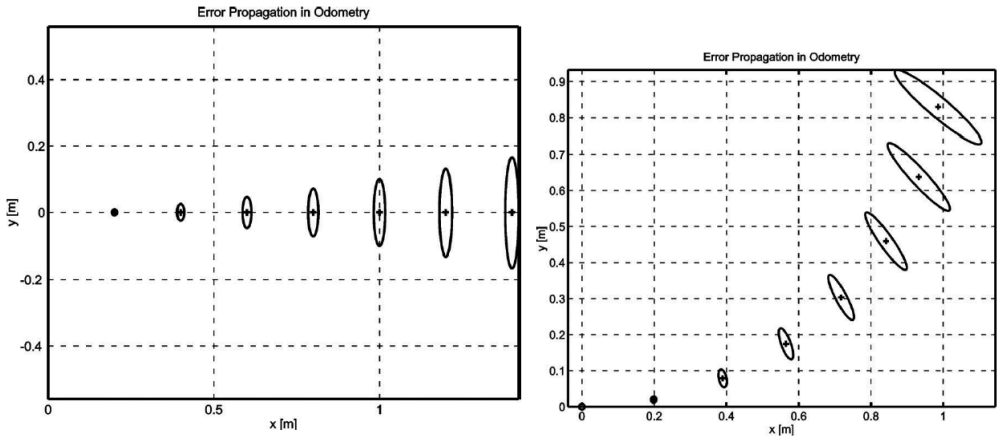
2.3.1 Localization Problem

If a mobile robot could be equipped with an accurate GPS (global positioning system) sensor, much of the localization problem would be solved because the robot would know its actual location. Unfortunately, the current GPS network only gives accuracy to a few metres, which is insufficient for localising human-scale mobile robots. Furthermore, GPS technologies cannot operate indoors or inside tunnels since a line of sight between satellites and the GPS receiver mounted on the robot is required. As a result, another method for localising the robot must be developed in this situation. The basic ways of localization are called odometry (wheels encoders only) and dead reckoning (also heading sensors), with these approaches the movement of the robot, sensed with wheel encoders or heading sensors or both, is integrated to compute position. The problem of these approaches is the inevitable presence of uncertainty which arises from five different factors [14]:

- Sensor noise lowers the amount of meaningful information in sensor measurements.
- Robot effectors are also noisy. In particular, a single action taken by a mobile robot may have several different possible results, even though from the robot's point of view the initial state before the action was taken is well known. From the robot's point of view, this error in motion is viewed as an error in odometry, or the robot's inability to estimate its own position over time using knowledge of its kinematics and dynamics.
- Incomplete model of the environment also introduce uncertainties. For instance, the robot does not model the fact that the floor may be sloped, the wheels may slip, and a human may push the robot.

- Models are inherently inaccurate since they are abstractions of the real world. As such, they only partially model the underlying physical processes of the robot and its environment.

All of these unmodeled sources of error result in discrepancies between the robot's physical motion, intended motion, and proprioceptive sensor estimates of motion. This uncertainty contribute to the fact that, as a mobile robot moves about its environment, the rotational error between its internal reference frame and its initial reference frame develops rapidly over time (drift error). An error model for odometric accuracy of a differential drive robot can be developed to show how errors propagate over time, as presented in [1]. Using this error model results like the ones in Figure 2.12 can be obtained. These figures show how the position error grow with time both for a linear movement and a circular one.



(a) *Linear error* **(b)** *Circular error*

Figure 2.12: Comparison between linear odometric error (left) and circular odometric error (right) [1]

As can be seen this error accumulates while robot moving producing an unacceptable increasing of the position uncertainty over time. This means another approach must be used. The following subsection will present the main algorithm used in mobile robotic in order to solve this issue.

2.3.2 Extended Kalman Filter

The problem is determining how to estimate our new pose in light of the previous pose and noisy odometry. We want the most accurate assessment of where we are and how certain we are about it. The mathematical tool that we will use is the Kalman filter. It is formulated for linear systems and, under this condition, it provides the optimal estimate of the system state, assuming that the noise is zero-mean and Gaussian. Because our model of the vehicle's motion is nonlinear, the nonlinear version of the filter, known as the extended Kalman filter, will be used. As will be demonstrated later, the optimality of the estimate is not guaranteed in this instance. Figure 2.13 shows a block scheme example of the Kalman filter application.

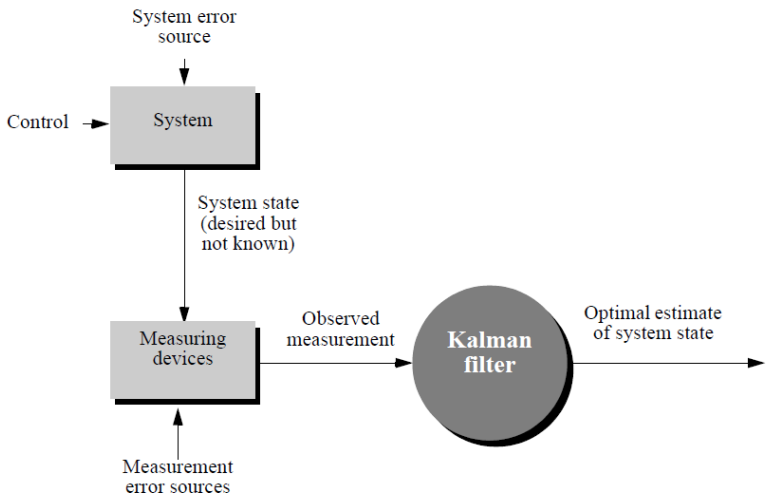


Figure 2.13: Kalman filter block scheme [1]

In this scenario, we have a system in which the true state x evolves over time in response to the applied inputs; we cannot directly measure the state, but sensors on the robot produce outputs that are a function of the initial state. The equations that follow describe the system's evolution in a discrete manner. The motion model is described by the non linear function f, which specifies how the state evolves in a step given the previous state and the input u. Instead, the nonlinear function h characterises the sensor measurement model, which is how the system state is transferred to the observable system outputs.

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, v_k) \\ z_k &= h(x_k, w_k) \end{aligned} \tag{2.5}$$

To account for errors in the motion model or unmodeled disturbances we introduce a Gaussian random variable v termed the process noise with zero mean and covariance V. Also the sensor measurement model is not perfect either and this is modeled by sensor measurement noise, a Gaussian random variable w with zero mean and covariance W. Covariance is a symmetric matrix quantity that represents the variance of a multidimensional distribution [10].

Since the system is non linear, it is linearized about the current state estimate $\hat{x_k}$ obtaining:

$$\begin{aligned} x'_{k+1} &\approx F_x x'_k + F_u u_k + F_v v_k \\ z'_k &\approx H_x x'_k + H_w w_k \end{aligned} \tag{2.6}$$

where $F_x = \frac{\delta f}{\delta x}$, $F_u = \frac{\delta f}{\delta u}$, $F_v = \frac{\delta f}{\delta v}$, $H_x = \frac{\delta h}{\delta x}$, $H_w = \frac{\delta h}{\delta w}$ are Jacobians of the functions f and h.

The filter is a recursive algorithm that updates, at each time step, the estimate of the unknown true configuration and the uncertainty associated with that estimate based on the previous estimate and noisy measurement data. It has two steps: prediction and update. The prediction is based on the previous state and the inputs that were applied, considering in the computation the absence of noise $v_k = 0$:

$$\begin{aligned} \hat{x}^+_{k+1} &= f(\hat{x_k}, u_k) \\ \hat{P}^+_{k+1} &= F_x \hat{P}_k F_x^T + F_v \hat{V} F_v^T \end{aligned} \tag{2.7}$$

Here $\hat{V}$ is best estimate of the covariance of the process noise, it is a chosen matrix which must be a reasonable estimate of the covariance of the actual process noise. Instead $\hat{x}^+_{k+1}$ and $\hat{P}^+_{k+1}$ are the estimate of the state and the corresponding estimated covariance. Since the prediction of $\hat{P}_{k+1}$ involves the addition of two positive definite matrices, the uncertainty will increase having used an uncertain model to predict the future value of an already uncertain estimate. To counteract this increase in uncertainty, new informations must be provided, such as sensor data, which are dependent on the state. The discrepancy between what the sensors

actually measure and what the sensors are supposed to measure is called innovation and is

$$v = z_{k+1} - h(\hat{x}^+_{k+1}) \tag{2.8}$$

This difference is due to the measurement noise, this remainder provides valuable information related to the error between the actual and the predicted value of the state. The second step of the Kalman filter is called update step. It maps the innovation into a correction for the predicted state, adjusting the estimate based on what the sensors observed

$$\begin{aligned} \hat{x}_{k+1} &= \hat{x}^+_{k+1} + Kv \\ \hat{P}_{k+1} &= \hat{P}^+_{k+1} - KH_x\hat{P}^+_{k+1} \end{aligned} \tag{2.9}$$

Uncertainty is now decreased, since new information, from the sensors, is being incorporated. The matrix

$$K = \hat{P}^+_{k+1}H_x^T(H_x\hat{P}^+_{k+1}H_x^T + H_w\hat{W}H_w^T)^{-1} \tag{2.10}$$

is known as the Kalman gain . The term indicated is the estimated covariance of the innovation and comprises the uncertainty in the state and the estimated measurement noise covariance $\hat{W}$. The filter must be initialized with some reasonable value of $\hat{x}$ and $\hat{P}$, as well as good choices of the covariance matrices $\hat{V}$ and $\hat{W}$. As the filter runs the estimated covariance $\hat{P}$ decreases but never reaches zero.

A fundamental problem with the extended Kalman filter is that the power spectral densities of the random variables are no longer Gaussian after being operated on by the nonlinear functions f and h. The Jacobians that appear in the EKF equations appropriately scale the covariance but the resulting non-Gaussian distribution breaks the assumptions which guarantee that the Kalman filter is an optimal estimator.

For the purpose of this thesis, the robot_localization package developed by Clearpath Robotics has been used [15]. It implements the EKF in ROS just letting the user chose the process noise and measurement noise covariance matrices and the sensor signals one want to fuse in order to estimate the robot state. The package has the flexibility to accept data from various sensors such as IMU, GPS, Wheel

encoders. In this case data from wheel encoders and IMU have been fused, taking as input /odom and /imu/data topics and giving as output /odometery/filtered topic at 10 hz. The version used is a bidimensional mode, meaning that only the x and y directions are considered. The following signals have been fused:

- x,y linear positions and velocities from the wheel encoders
- x,y positions from imu

2.4 Navigation Algorithm

This section will present the developed navigation algorithm, deepening in each subsection all the features it has. The basic idea is to perform the navigation inside an unknown tunnel without mapping the environment in order to reduce the number of needed algorithms and, as a consequence, the CPU payload.

The navigation algorithm is based on a Pure Pursuit controller which tracks a path by continually generating speed and steering commands that compensate for the tracking errors. These mainly consist of vehicle's deviations in distance and heading from the path. The PPA is a widespread and effective geometric method used in mobile robotics as tracking strategy. In this work, the path which have to be tracked is a set of points generated along a line representing the instantaneous center of the tunnel which is updated every 1.0 seconds. The central line is computed as the mean line between the lines representing the left and the right walls of the tunnel, which are generated by fitting some specific LIDAR points through an Orthogonal Distance Regression model [16]. Both the PPA and ODR will be explained in the next sections, but before going on, some other aspects must be pointed out.

As said above, the robot is able to localize itself in the bi-dimensional space by fusing the measured data from IMU and wheel encoders (odometry measure), in this manner the robot knows every time where it is with respect to a fixed RF which, in the ROS environment, is called "odom". Instead the navigation is performed elaborating LIDAR data in a specific manner after a pre-processing step:

- A laser scanner mounted on a vehicle generates a light beam that rotates in a horizontal plane parallel to the ground. As a result, a discrete time scan at time kT_0 may be described as an ordered set of points $\{q_k\}$. This set corresponds to the laser's successive intersections with the nearby objects in the surroundings.

  But each raw scan is obtained as a list of ranges $\{\rho k(j)\}$, where index j is an integer between 0 and N. Thus these data must be manipulated in order to obtain the set of points. Since the readings are evenly spaced within the sensor field of view θ. Then, the beam bearing for $\rho_k(j)$ is

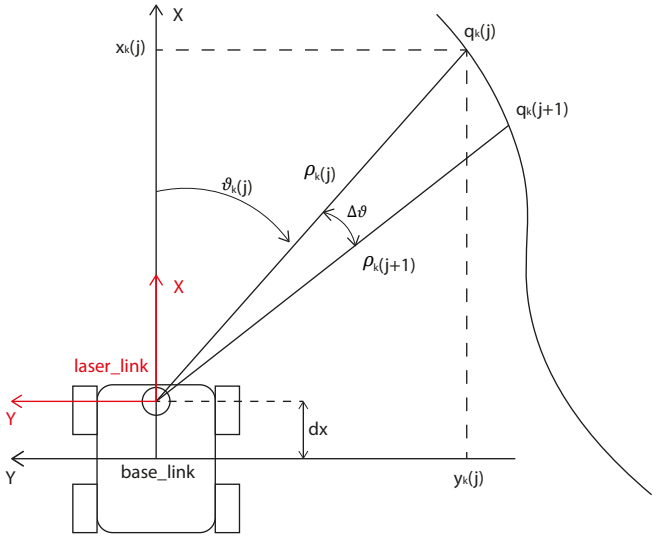


Figure 2.14: Local cartesian coordinates of the laser point $q_k(j)$

$$\theta_k(j) = j\Delta\theta \tag{2.11}$$

where $\Delta\ \theta = \theta\ /\ N$ represents the angular resolution. In the case of the used laser scan, since $\theta = 360$ and $N = 360$, a resolution $\Delta\ \theta = 1°$ is obtained. After these geometrical reasoning, it is possible to associate each laser beam with a range value and an angle value with respect to the x axis of the laser reference frame ("laser_link"). In this way, the position of the point $q_k(j)$ can be expressed in the vehicle's local frame ("base_link") as follows:

$$\begin{aligned} x_k(j) &= dx + \rho_k(j)\cos\theta_k(j) \\ y_k(j) &= -\rho_k(j)\sin\theta_k(j) \end{aligned} \tag{2.12}$$

As can be seen in Figure 2.14, since the position of the laser scan does not coincide with the origin of the vehicle's local frame ("base_link"), the x coordinate of each point $q_k(j)$ is shifted of the quantity dx, representing the distance between the RFs [17].

- Now that the coordinates of each point $q_k(j)$ in the robot RF are obtained,

they must be transformed into the fixed RF, bearing in mind that, at each instant, the robot local frame is translated and rotated about the Z axis with respect to the fixed frame as shown in Figure 2.15. The coordinates transformation is defined as

$$R(t) = R_0(t) + T_{2\times2}(\alpha(t))r(t) \tag{2.13}$$

where R(t) is the coordinates vector of the point q with respect to "odom" frame, $R_0(t)$ is the coordinates vector of the origin of "base_link" frame with respect to "odom" frame, r(t) is the coordinates vector of the point q with respect to "base_link" frame and $T(\alpha(t))$ is a 2x2 rotation matrix around the Z axis of "odom" frame which depends on the angle of rotation α and is defined as

$$T_{2\times2}(\alpha(t)) = \begin{bmatrix} \cos\alpha(t) & -\sin\alpha(t) \\ \sin\alpha(t) & \cos\alpha(t) \end{bmatrix}$$

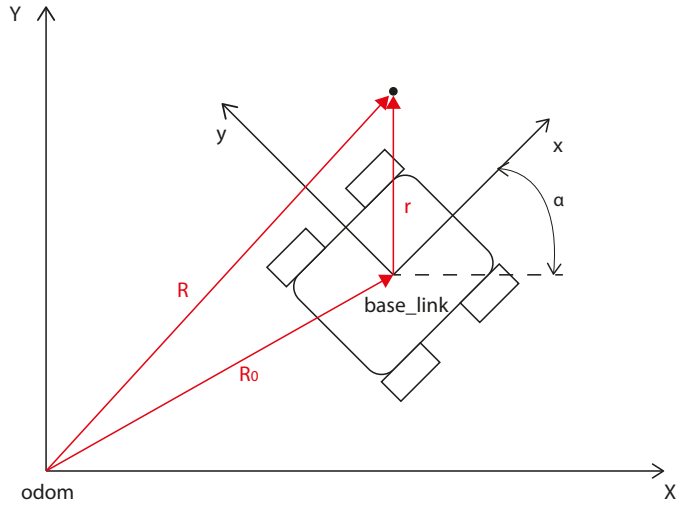


Figure 2.15: Coordinates transformation

Now that the pre-processing of the LIDAR data is known, in the following two subsection i will present the theory behind an algorithm and a method which are used as fundamental bricks of the navigation algorithm: Pure-Pursuit Algorithm

and Orthogonal Distance Regression, respectively.

2.4.1 Pure Pursuit Algorithm

The pure pursuit approach is a method of geometrically determining the curvature that will drive the vehicle to a chosen path point, called the goal point. This goal point is a point on the path that is one look-ahead distance from the current vehicle position. To derive the equations of the pure-pursuit curvature control law, let us consider a mobile robot on the XY plane as shown in Figure 2.16.

The curvature γ of the vehicle can be defined as the inverse of the distance r between the vehicle's frame origin and its instantaneous center of rotation (ICR). In the pure-pursuit strategy, the vehicle changes its curvature set-point γ_{sp} every control interval T_0 by fitting a circumference arc to a goal point in the path at a certain look-ahead distance L. This circumference is tangent to the Y axis of the vehicle. Thus, the goal point local coordinates (x_g , y_g) can be geometrically obtained as

$$\begin{aligned} x_g &= \frac{\cos\Delta\phi - 1}{\gamma_{sp}} \\ y_g &= \frac{\sin\Delta\phi}{\gamma_{sp}} \end{aligned} \tag{2.14}$$

where $\Delta\,\phi$ is the heading change of the vehicle along the arc. Then, by using 2.14,

$$L^2 = x_g^2 + y_g^2 = \frac{2 - 2\cos\Delta\phi}{\gamma_{sp}^2} = -\frac{2x_g}{\gamma_{sp}} \tag{2.15}$$

Therefore, the curvature control law is given by

$$\gamma_{sp} = -\frac{2x_g}{L^2} \tag{2.16}$$

where the terms x_g and $2/L^2$ can be interpreted as the error signal and the gain of the control law, respectively [17].

The main parameter to be set in the pure pursuit is the look-ahead distance.

A longer look-ahead distance implies a smaller gain, so steering control is smoother at the expense of a worse tracking accuracy. On the other hand, a shorter

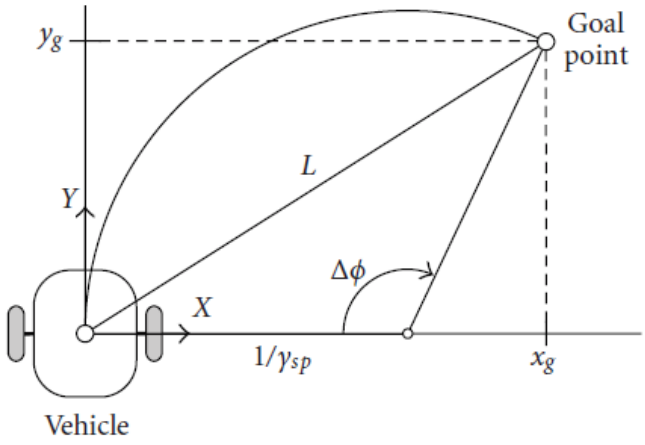


Figure 2.16: Pure Pursuit geometry [17]

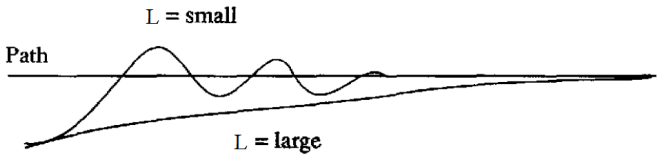


Figure 2.17: Effect of the look-ahead distance tuning [18]

look-ahead can reduce tracking errors, but steering commands increase and the vehicle's motion can become unstable introducing oscillations. The response of the pure pursuit tracker looks similar to the step response of a second order dynamic system and the value of L tends to act as a damping factor as shown in Figure 2.17 [18].

2.4.2 Orthogonal Distance Regression

As anticipated above, an important aspect of the navigation algorithm is the linear fitting of the laser data in order to obtain features representing the lateral tunnel walls. This problem can be addressed through various regression models. Regression basically involves determining the relationship between a dependent variable (usually y(x)) and one (or more) independent variable (usually x). In the classical linear regression problem, we assume that we have samples of a function y(x), where x is the independent variable, meaning that we take the values of the

x_n to be perfectly known. The y_n values, on the other hand, are presumed to have been measured experimentally and hence suffer from numerous noise effects such as measurement error, quantization error, modelling error, and so on. This problem is thus called 'one dimensional' as there is a single independent variable x. Defining the approximation error of a data point (x_n, y_n) as the difference between the approximated y and the measured value $\epsilon_n = \hat{y_n} - y_n = (mx_n + b) - y_n$, the line that 'best fits' the data is the one that minimizes the sum of the squares of these approximation errors [19].

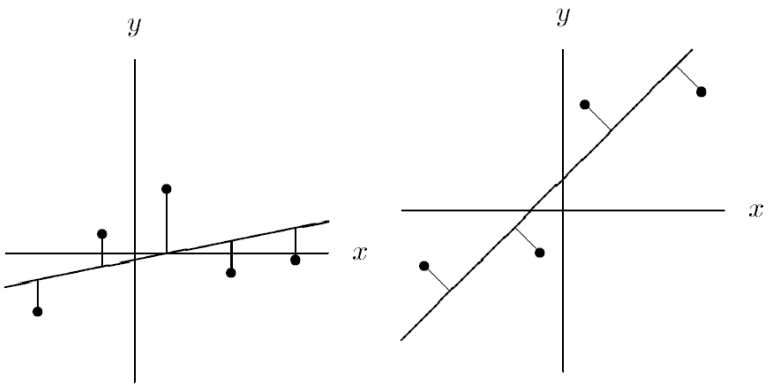
In many applications, such as this one, the goal is to discover the line that best approximates a two-dimensional set of data points where neither x nor y are independent variables on which the other depends. In fact, because every point obtained through the measurement of a sensor (laser) is influenced by inaccuracies and measurement mistakes, both the x and y values are affected. Because it only minimises the vertical distance of each point from the line, the typical least square method (one dimensional linear regression problem) for line fitting may produce unacceptable parameter estimate results in this circumstance.[20].

In contrast to one-dimensional linear regression problems, where errors are measured vertically with respect to the fitted line, orthogonal distance regression (also known as two-dimensional Euclidean regression) involves calculating the orthogonal distance of the points with respect to the fitted line, allowing errors in measurements for both independent and dependent variables along the x and y-axes to be taken into account. A graphical comparison between these approaches is shown in Figure 2.18. This way of calculating the perpendicular distance adds more robustness to the model.

Going deeper, since neither x nor y have special status, the natural error criterion is the distance of data point from the approximation line, that is the Euclidean distance from the data point to the nearest point on the line represented in the form y = mx + b. The distance from the data point (x_n, y_n) and the line is given by equation

$$\epsilon_n = \left| \frac{y_n - (mx_n + b)}{\sqrt{m^2 + 1}} \right| \tag{2.17}$$

The line which minimizes the sum of the squares of these errors ϵ_n is the



(a) *Vertical error* **(b)** *Orthogonal error*

Figure 2.18: Comparison between vertical error (left) and orthogonal error (right)

orthogonal regression line (two dimensional Euclidean regression line), which parameters are obtained via partial derivation of Eq. 2.17, obtaining

$$\begin{aligned} m &= \frac{-R \pm \sqrt{R^2+4}}{2} \\ b &= \bar{y} - m\bar{x} \end{aligned} \tag{2.18}$$

where

$$R = \frac{\sigma_x^2 - \sigma_y^2}{\sigma_{xy}} \quad where \quad \begin{cases} \sigma_x^2 & = \frac{1}{n}\sum_i (x_i - \bar{x_i})^2 \\ \sigma_y^2 & = \frac{1}{n}\sum_i (y_i - \bar{y_i})^2 \\ \sigma_{xy}^2 & = \frac{1}{n}\sum_i (x_i - \bar{y_i})(x_i - \bar{y_i}) \\ \bar{x} & = \frac{1}{n}\sum_i x_i \\ \bar{y} & = \frac{1}{n}\sum_i y_i \end{cases} \tag{2.19}$$

The slope that minimizes the error is the one for which the sign of $\hat{m}$ matches that of σ_{xy} [19].

2.4.3 Working Principle

Now that the main bricks of the algorithm are known, the working principle of the navigation algorithm will be explained. The objective of the algorithm is to allow the robot to autonomously explore an underground tunnel whose map is not known. The idea is to perform the localization fusing IMU and wheel encoders data, the navigation simply elaborating in a geometric manner LIDAR data and the inspection using a recording camera equipped with spotlights enlightening the environment. Figure 2.19 shows the ROS graph which represents the architecture of the system emphasizing the way in which each nodes communicate with the others. Here the nodes are the ellipses while the topics are the rectangles.

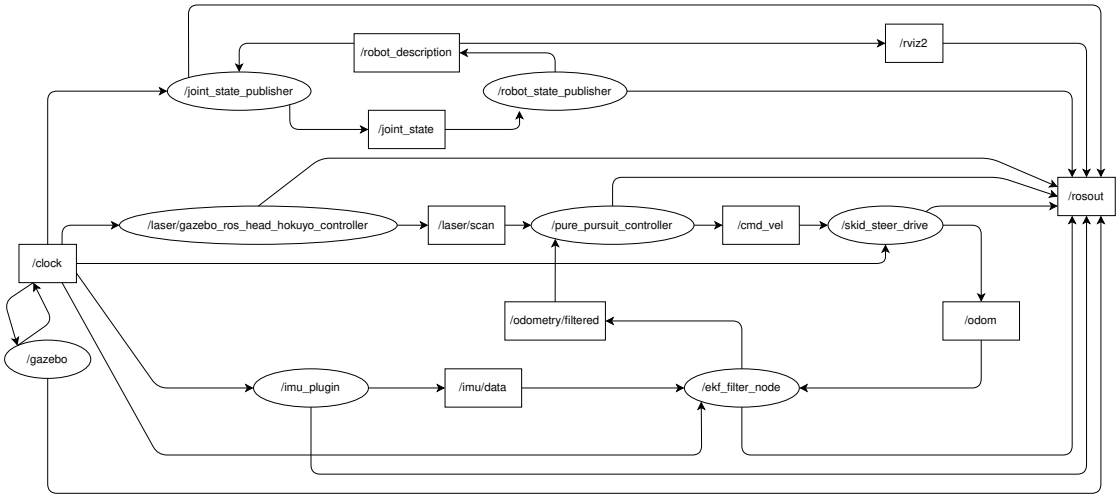


Figure 2.19: ROS graph of the Husky robot used during simulations. The image was taken while using the Gazebo simulator, hence the presence of a node named gazebo.

Table 2.1 shows the detailed information of the ROS graph main nodes, underlining the name, the input and output topics of each node and their functions.

Inside the pure pursuit controller node there is the proposed behaviour-based navigation algorithm. To carry out the inspection task in a more efficient way the robot must remain in the centre of the tunnel and should cover the entire plan of the tunnel, therefore a solid decision-making logic must be given to the robot. The strategy used is the one followed to be able to get out of a maze, that is to keep always the same wall in a specific side, in this way you are sure that you will get

Node Name	Input Topic (message type)	Output Topic (message type)	Function
imu_plugin	none	/imu/data (sensor_msgs/Imu)	ROS plugin simulating data from IMU
laser/gazebo_ros_ head_hokuyo_ controller	none	/laser/scan (sensor_msgs/LaserScan)	ROS plugin simulating LIDAR data
skid_steer_drive	/cmd_vel (geometry_msgs/Twist)	/odom (nav_msgs/Odometry)	ROS plugin simulating a 4 wheel differential drive mobile robot
ekf_filter_node	/imu/data (sensor_msgs/Imu) /odom (nav_msgs/Odometry)	/odometry/filtered (nav_msgs/Odometry)	Node implementing the Extended Kalman Filter
pure_pursuit_ controller	/laser/scan (sensor_msgs/LaserScan) /odometry/filtered (nav_msgs/Odometry)	/cmd/vel (geometry_msgs/Twist)	Node implementing the navigation

Table 2.1: Detailed informations about ROS graph nodes

out of the maze. Specifically, in this case the algorithm has been implemented so that the robot navigates inside the tunnel always following the left wall, both in the presence of crossroads and forks. Thus, unless there are special cases such as internal loops, the robot will travel through the entire tunnel plan and then exit the entry point. For the algorithm to work as intended, two hypothesis regarding the tunnel characteristics are made:

- The width of the tunnel corridors remains more or less always the same. If a narrowing or an enlargement happens, it should be for a short space.
- The tunnel should have only one entry point

Figure 2.20 show the block scheme of the navigation algorithm. The inner loop is a low-level motion controller. It translates speed and steering set-points into references for the actuators, represented in simulation by the skid_steer_drive

plugin, every $T_{control}$ seconds. The motion commands send are a constant linear velocity v_x= 0.5 $\frac{m}{s}$ or v_x= 0.0 $\frac{m}{s}$ depending on the manuvre the robot has to accomplish, and an angular velocity ω_z proportional to the angle difference between the goal point and the robot heading. The angular velocity is also saturated at a maximum angular speed of ω_z= 0.7 $\frac{rad}{s}$. The motion controller also updates the localization with a period T_{inner}, which is the period at which the EKF outputs the robot state (the internal sensors are IMU and wheel encoders). The outer loop has a longer period T_{outer}, representing the time at which new laser data are fed into the state machine. This block is in charge of establishing the state of the algorithm and therefore to define, every T_{update} seconds, on the basis of the new laser data which will be the next path to track. Table 2.2 defines all the working periods of the algorithm.

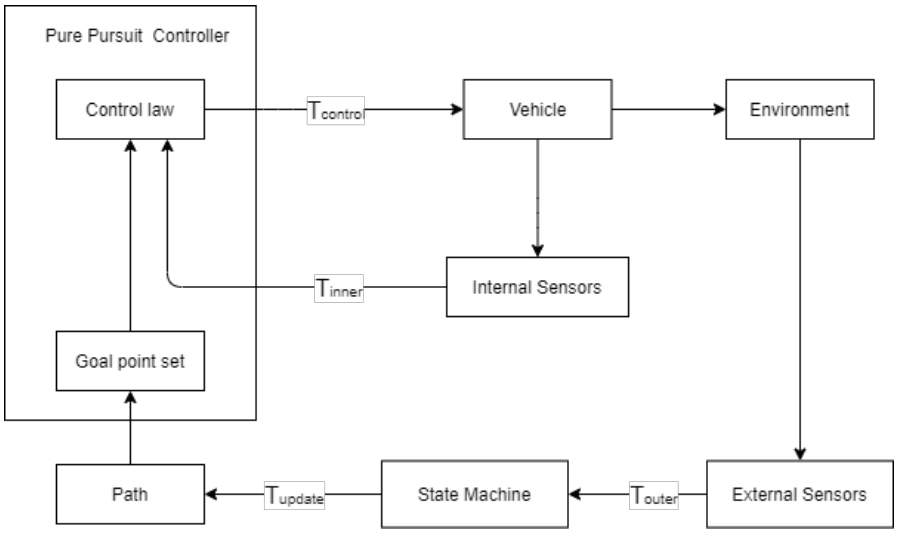


Figure 2.20: Block scheme of the navigation algorithm.

T_{inner}	0.1 s
T_{outer}	0.033 s
$T_{control}$	0.1 s
T_{update}	1.0 s

Table 2.2: Working periods of the algorithm

Now the State Machine block will be exploded and all the possible states in which the algorithm can occurs will be explained. Since the tunnel map is unknown and is not generated by mapping, the algorithm must be able to recognize the various situations that can occur within the tunnel in order to make the robot take the right path. Moreover, it has to take into account the basic idea of always following the left wall. I tried to cope with the most classic situations that can occur:

- Straight and curved paths
- Left and Right sharp turns
- Crossings
- Open spaces
- Blind ends
- Forks
- Obstacles

In order to address the listed situations five states have been created as shown in Figure 2.21. The working of each state will be discussed in detail in the next subsections, so far I will discuss at high-level the mechanism of the state machine, considering that, whenever i will mention "line", it means that it has been obtained by fitting some LIDAR data points trough orthogonal regression.

The default state of the algorithm is the NORMAL STATE, it means that the algorithm remains here if none of the conditions for a state transition is verified (the "check" conditions represented in Figure 2.21). In this state the path points are generated at the central line of the tunnel and updated every 1.0 s, this update rate of the path also makes it possible to let the robot run efficiently curves with reduced bending radii. When the tunnels enlarges in an open space, the algorithm switches in the OPEN SPACE STATE and the robot start to track a path parallel to the left wall's line. If the algorithm is in this state and the robot reaches a wall in front of it, it steers to the right and starts to follow a path parallel to the frontal wall line. This condition occurs also in case of sharp right turns, and so this state will cope with them. When the robot reaches a sharp left turn, the

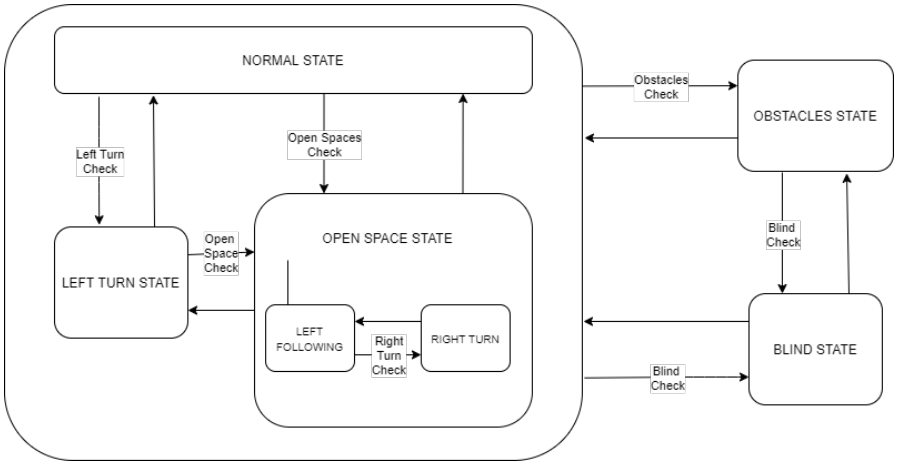


Figure 2.21: State machine

algorithm recognizes it in advance and switch in the LEFT TURN STATE which is in charge of generating a special path to make the robot take the left way. This is also the state in charge to let the robot choose the left way when approaching a crossing. For this reason, when the robot enters in this state, the path generated is not updated with the usual frequency but the updating is suspended until the robot approaches a specific point in the path ensuring to the robot the completion of the turn and, in the case of crossing, ensuring its positioning at the center of the crossing's left channel. These three states are the ones inside the big box on the left of Figure 2.21, the grouping was done to emphasize that for these states the path update is done every second. Instead the condition of outer states is checked every 0.033 seconds, corresponding to 30 Hz, which is the working frequency of the LIDAR. This frequency is the resolution of the robot view since it receives new laser scan data at that rate. The reason behind this choice is that these states are the ones in charge of cope with the most dangerous situations such as obstacles and blind ends, and so, facing that situations the robot must be able to generate a reactive action. In fact, when the robot recognizes a blind end it is able to readily reverse the path and turn back switching in the BLIND STATE. Finally, if the robot is following a path generated at the last update that would lead him against an obstacle, it recognize the obstacle in advance and enters in the OBSTACLES STATE. Here the robot understands if there are accessible ways at the obstacle's sides and, if it so, it starts to follow a special path that leads it to overcome the

obstacle, otherwise it exits from the OBSTACLE STATES and enters the BLIND STATE treating the obstacle as a blind end since there were no way to overcome it. Also when the obstacle path is generated, it is kept until the robot reaches a specific point of the path, in a similar manner of the sharp left turn path. In this way the overcoming of the obstacle is assured.

The following subsections will threat in deeper the algorithm states and the way in which the paths are generated, also underlining which state is in charge of cope with the above mentioned situations.

2.4.4 Normal State

As anticipated above, when the algorithm is in the NORMAL STATE, some lateral LIDAR data points are fitted trough orthogonal regression and the parameters (angular coefficient and intercept) of the left and right lines are extracted. Then, averaging these parameters, the ones of the central line are obtained. Figure 2.22 shows the robot from above in the simulation environment. There can be seen blue segments representing the LIDAR beams, the ones highlighted in red are the indices of the beams considered in the orthogonal regression. The RF depicted is the base_link frame.

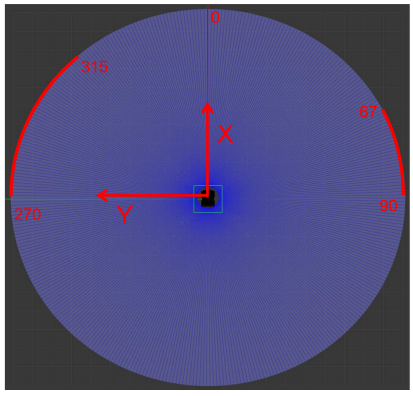
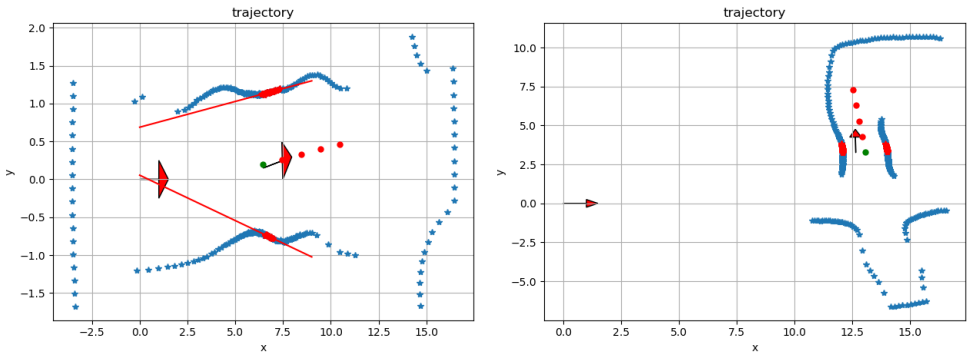


Figure 2.22: Laser indices considered in the orthogonal regression when the algorithm is in the NORMAL STATE

As can be observed, the number of points is asymmetric: the left line is generated fitting 45 points while the right one fitting 23 points. The reason behind this choice is that as the robot must always follow the left wall it is good that it has an

extended view so as to be more sensitive to variations in that side. Only points that stand in the first and fourth quadrant of the base_link RF are considered, in this way the robot's view is projected forward resulting in more responsive navigation in case of curves. Once the parameters of the central line are computed, the projection of the robot along this line is obtained. From this points, four path points spaced 0.5 meters in the forward direction are computed. This will be the path in input to the Pure Pursuit controller. Figure 2.23 shows two example of the path generation in the NORMAL STATE. These pictures are created offline through a python script, all the necessary data are obtained printing them in the terminal during a simulation. Here the blue dots represent the LIDAR data points, the lateral red ones are the points used for fitting. The arrow in the left of both the images represent the x-axes of the fixed RF (odom) while the other arrows are the x-axes of the robot base_link RF thus representing the robot position and its heading. The green dot is the robot projection along the central line and the central red dots are the instantaneous path points.



(a) *Path generation when the robot is going along X direction of odom RF*

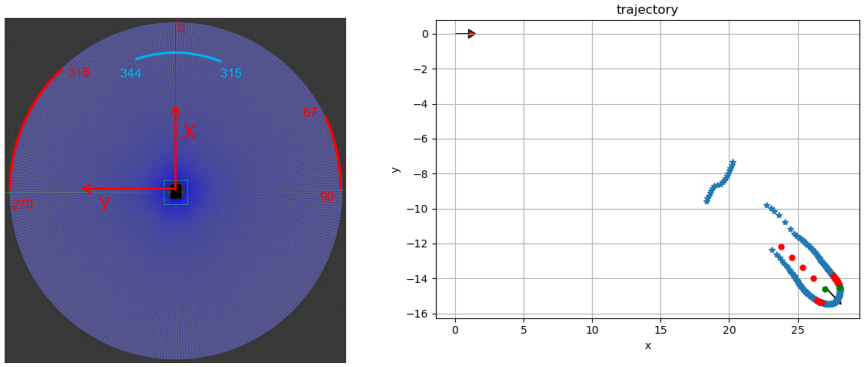
(b) *Path generation when the robot is going along Y direction of odom RF*

Figure 2.23: Examples of path generation in NORMAL STATE

2.4.5 Blind State

Since inside tunnels there could be blind ends, the robot must be able to recognize them and and react accordingly by coming back, for this reason also a BLIND STATE has been implemented. The transition condition for this state is evaluated

every 0.033 seconds in order to let the robot be enough reactive to these situations. Figure 2.24(left) represents in light blue the indices of the laser beams used for the transition check. When the length of all these beams is less then 1.2 meters the transition to the BLIND STATE occurs and the path points are generated as in the NORMAL STATE (using the same beam indices for the lines fitting) but in the opposite direction. Figure 2.24(right) shows an example of the generated path in BLIND STATE. When the robot approaches a blind end, the algorithm set a null linear velocity in order to let the robot remain in place and rotate only in order to recover the angle difference between the goal point and the robot heading without the risk of hitting the walls. In order to do so, during this maneuver, the path update is suspended for 6 seconds which is the time required by the robot to steer of about 180 (angle difference between goal point and robot heading when coming back).



(a) *Laser indices considered when the algorithm is in the BLIND STATE*

(b) *BLIND STATE path generation*

Figure 2.24: BLIND STATE considered indices (left) and path example (right)

2.4.6 Open Space State

Another important situation to consider in which the robot can be found is the enlargement of the tunnel in an open space as the ones highlighted in Figure 2.25:

In these cases the robot recognizes the presence of an open space when, at least, one of the light blue laser beams showed in Figure 2.26-a reaches a range

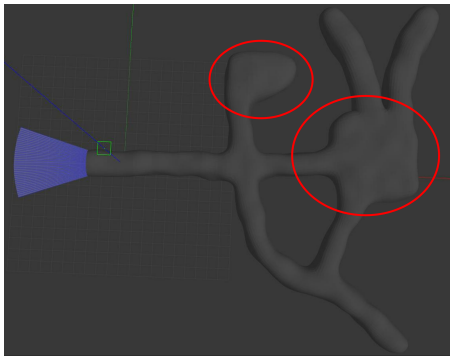


Figure 2.25: Examples of open spaces

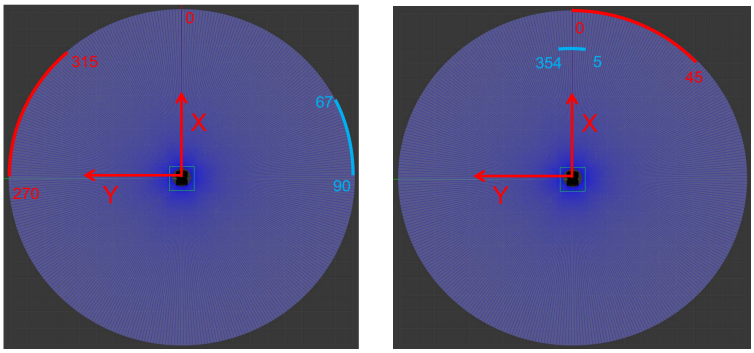
larger then the tunnel width. If this condition is verified, the algorithm switches in OPEN SPACE STATE. When the Pure Pursuit Controller node is initialized, a function which measures the tunnel width is called (it measures the distance between points 270 and 90), the parameter is saved as a global variable and used as a reference inside the OPEN SPACE STATE and the LEFT TURN STATE. As showed in Figure 2.21, the OPEN SPACE STATE is composed by two sub-states, "Left Following" and "Right Turn"; when the transition check for the OPEN SPACE STATE is verified, the algorithm enters in "Left Following" (default condition) and starts to fit the red points in Figure 2.26-a and generates a four points path along a line parallel to the left wall line and with a distance from it equal to the half of the tunnel width.

The "Left Following" condition means that the robot is skirting the left wall in an open space, in this condition three situations can occur that disrupt the current behavior of the robot:

1. the open space shrinks again in a tunnel channel making the algorithm switch to the NORMAL STATE
2. a left sharp turn occurs which will the algorithm switch to the LEFT TURN STATE, as will be seen later
3. a right sharp turn occurs

In order to cope with the third case, while the robot is in the "Left Following" condition, the algorithm checks if all the light blue beams in Figure 2.26-b reach a range less then half the tunnel width, if so, the robot enters the "Right Turn"

condition. In this situation it fits the red points of Figure 2.26-b and begins to follow a four points path generated along the line parallel to the frontal wall and with a distance which is half the tunnel width. In this way the robot automatically turn right and start again, in the next step, to follow the left wall.

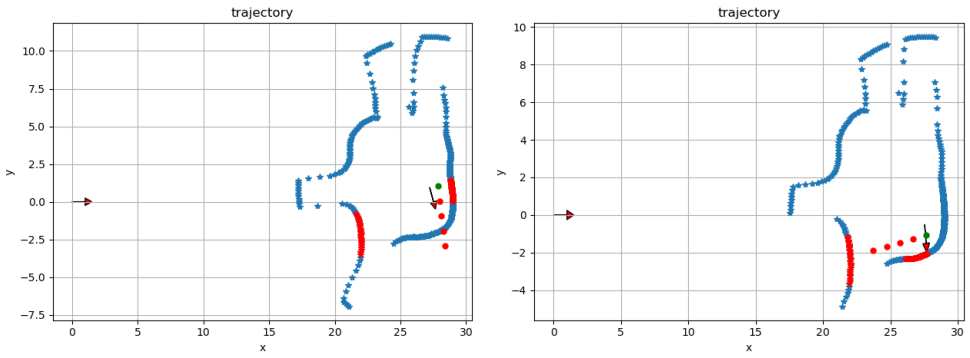


(a) *Left Following indices* **(b)** *Right Turn indices*

Figure 2.26: Indices used during Left Following condition (left) and Right Turn condition (right)

Figure 2.27 shows two examples, when the robot is in the "Left Following" condition, and when it is in the "Right Turn" condition. The path is generated as in the states previously considered: the points are spaced 0.5 meters and the green point is the projection of the robot position along the line and, from this one all the other points are computed.

As a consequence of this mechanism, if the robot encounter a sharp right turn it is able to cope with it by switching to this state. In fact, when approaching the right turn, it enters before in the "Left Following" condition and then in the "Right turn" condition overcoming the obstacle. Figure 2.28 shows another possible situation in which the OPEN SPACE STATE can be useful. In this case there is a fork, if the robot had faced this obstacle in the NORMAL STATE it would have crashed against the wall of the fork. However, some of the right laser beams responsible for the state transition encountered a range grater then the the tunnel width making the algorithm switch to the OPEN SPACE STATE in "Left Following" condition. In this way the robot begins to follow the left wall going trough the left tunnel's channel, as desired.



(a) *Left Following condition* **(b)** *Right Turn condition*

Figure 2.27: Examples of the Left Following condition (left) and Right Turn condition (right)

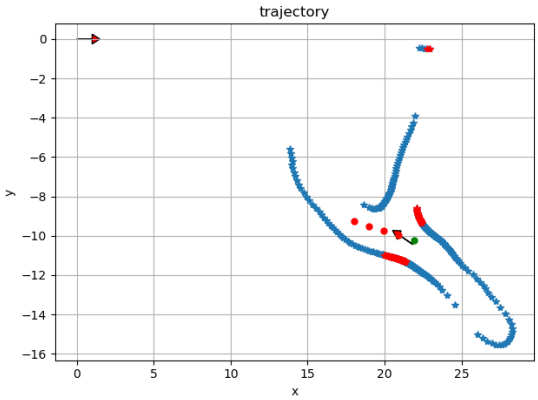


Figure 2.28: Fork example coped with the OPEN SPACE STATE

2.4.7 Left Turn State

The LEFT TURN STATE is the state in charge of recognizing the presence of a sharp left curve and to create a specific path. As anticipated above, by extension, this state is also adopted in the case of cross where it allows the robot to follow the left road, as desired. Figure 2.29 shows the laser indices used during the transition check for the LEFT TURN STATE. In order to enter this state and generate the curve path, two condition must be satisfied. The necessary but not sufficient condition is that the first laser beam in Figure 2.29 (index 320) and, at least, other five consecutive laser beams depicted in light blue increase their range of the tunnel

width (or more) with respect to the previous path update (T_{path} = 1.0 s).

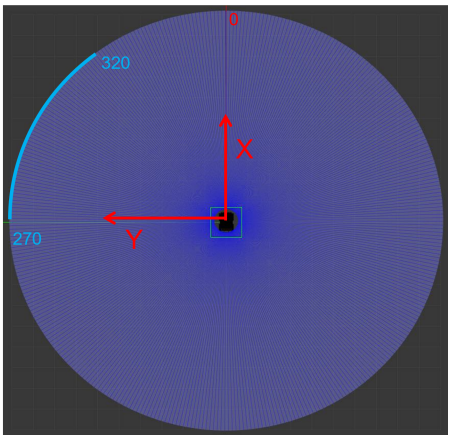


Figure 2.29: Laser indices considered when the algorithm is in the LEFT TURN STATE

In the following will be before explained the steps followed for the path generation when the algorithm is in this state, then, it will also be explained the sufficient condition for the LEFT TURN STATE transition because, later, it will be clearer. Figure 2.30-a shows a typical situation in which the robot can be found when both the necessary and sufficient condition for the state transition are verified; at this point the left curve path is generated following 5 steps:

1. Identification of the three green pivot points in Figure 2.30-b:

   - the point generated by the laser beam with index 320 (P3),
   - the point generated by the last laser beam which has increased its range more then the tunnel width (P2),
   - the point generated by the first laser beam which has not increased its range of that quantity (P1).

   Pivot point P3 always corresponds to the laser beam with index 320 because its presence is a constraint of the necessary condition. Pivot point P1 and P2 may vary depending on the situation.

2. Identification of the middle points of the connecting segments: as shown in Figure 2.30-c, the linking segments $\overline{P1-P2}$ and $\overline{P1-P3}$ are computed, and their middle points are generated.

3. Creation of the line passing through the middle points as shown in Figure 2.30-d.
4. Creation of 8 path points spaced 0.5 m along the line as shown in Figure 2.30-e.
5. Adding of two more path points between the robot position and the first point of the previously generated path as shown in Figure 2.30-f. These points are added in order to let the PPA smooth as little as possible the path during the curve.

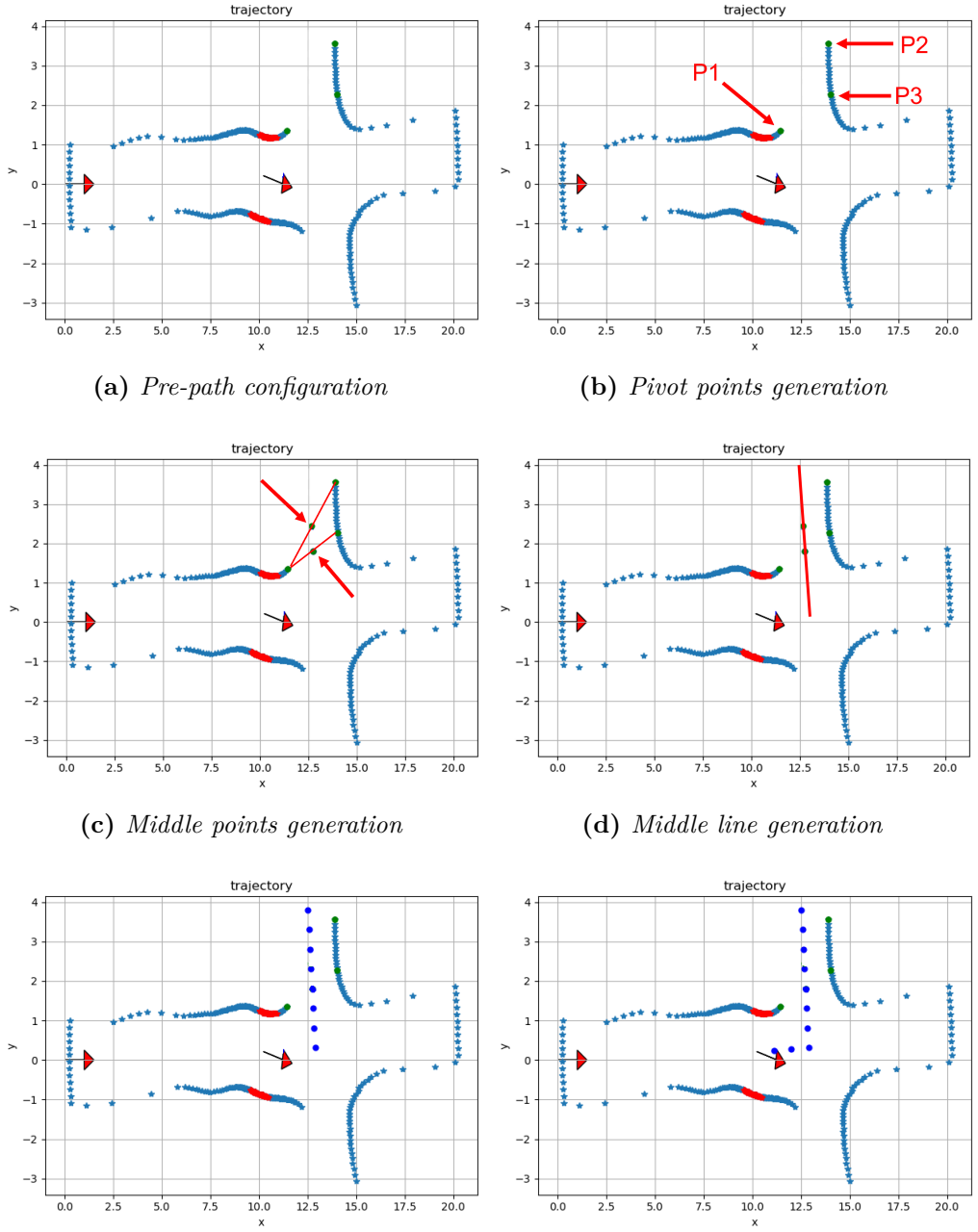
As anticipated above, once the left turn path is created, the Pure Pursuit controller start to track it until the robot reaches at 0.75 meter the seventh point of the path. Together with the blind end maneuver, also in this case the path is not updated every 1.0 seconds so as to give the robot time to complete the curve.

The necessary but not sufficient condition means that on the left side of the tunnel there will be a gap, but, this condition alone doesn't mean that that gap represents a possible left sharp curve, it could also be a sudden enlargement of the tunnel width. For this reason a second sufficient condition is added. Figure 2.31 represents an example of the sufficient condition, the figure on the left shows a real left turn while the right one shows a tunnel enlargement. Let's call the linking segment $\overline{P1-P3}$ as d1, and the linking segment $\overline{P1-P2}$ as d2, the sufficient condition tells that if d1 $<$ d2 there is a left turn and so the state switches to the LEFT TURN STATE, instead if d1 $>$ d2 there is no left turn and no state transition.

The last aspect to consider about this state is that, before the left turn path generation, a function is called that checks if there is a gap grater then 0.5 meters between the points that have undergone an increase in range from the previous step. If so, the points before the gap are not considered in the path generation process. Figure 2.32 shows an example, all the points inside the red square are not considered due to their excessive distance from P2. This feature helps to create more precise paths during left turns.

2.4.8 Obstacles State

The last state to be discussed is the OBSTACLES STATE. It is the state in charge of let the robot avoid obstacles along the tunnel checking the transition condition



(a) *Pre-path configuration*

(b) *Pivot points generation*

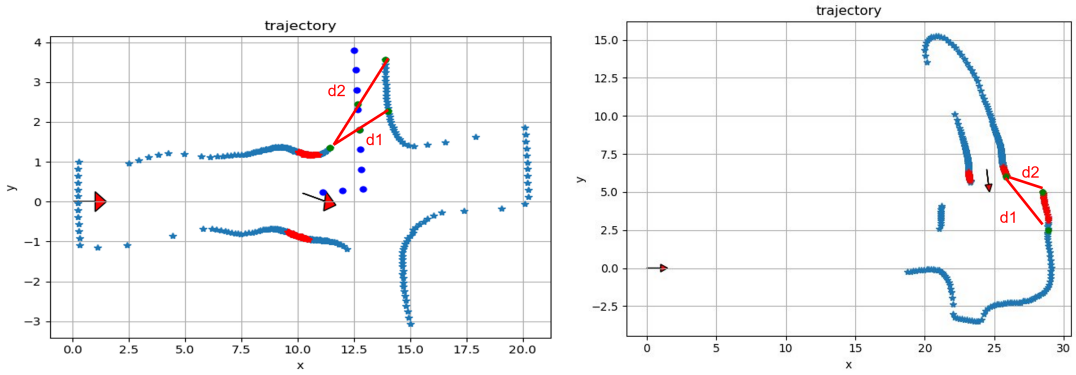
(c) *Middle points generation*

(d) *Middle line generation*

(e) *Curve path point generation*

(f) *Addition of two more points to the curve path*

Figure 2.30: Steps of the left turn path generation



(a) *Real left turn* **(b)** *False left turn*

Figure 2.31: Example of the sufficient condition

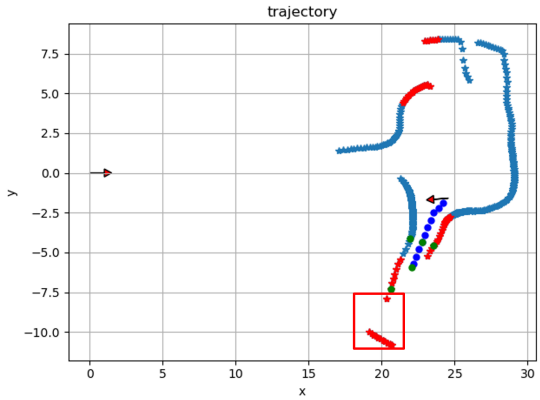


Figure 2.32: Example of outlier points elimination

with the same frequency of the LIDAR data (30 Hz). When a laser beam of the frontal cone with opening 18 degrees (within the laser indices ranging from 350 to 9 as shown in 2.33) reaches a range of 1.8 meters or less for a predefined number of consecutive checks, the OBSTACLES STATE is accessed. Once inside the state the specific path for obstacle overcoming is still not defined because, before, an accessible near path to overcome the obstacle must be found. The search is done checking the points defined by the LIDAR beams with indices ranging from 320 to 39 as shown in Figure 2.33. If this search is successful, the new path is generated, otherwise the robot continue to follow the previous path.

In the following, the detailed description of the operation implemented by the

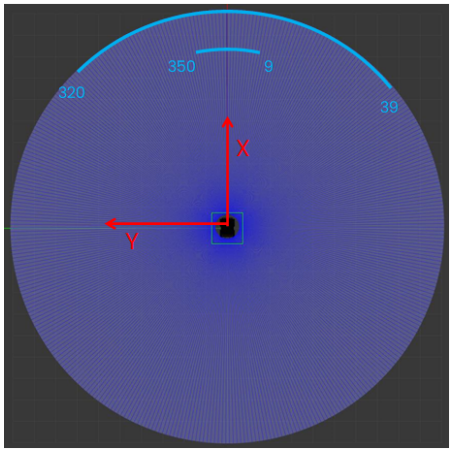
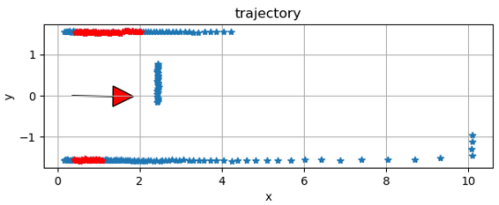


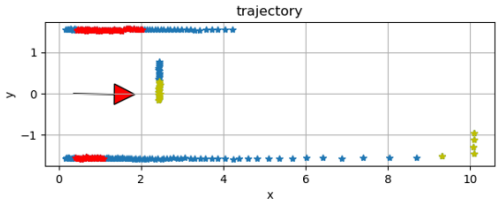
Figure 2.33: Laser indices considered when the algorithm is in the OBSTACLES STATE

algorithm for the gap searching, accessibility check and path definition will be presented. From the transition inside the OBSTACLES STATES to the path generation there are 5 steps;

- STEP 1: Figure 2.34 shows the robot approaching an obstacle in front of it. As usual, the red points are the ones used to extract the features of the lateral walls during the NORMAL STATE, instead the yellow dots are the ones obtained by the laser beams used for the transition check inside the OBSTACLES STATE. As said above, once one of these beams reaches 1.8 meter or less, the algorithm enters this state.



(a) *Robot approaching an obstacle in front of it*

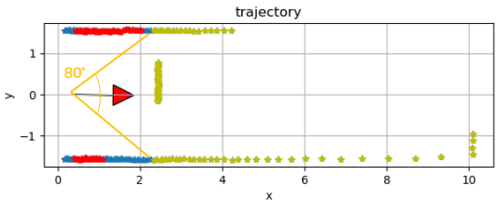


(b) *Laser beams responsible of the transition check*

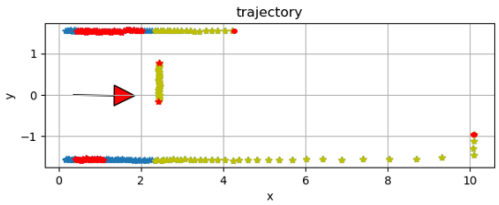
Figure 2.34: Step 1

- STEP 2: Once the OBSTACLES STATE is accessed, the algorithm searches

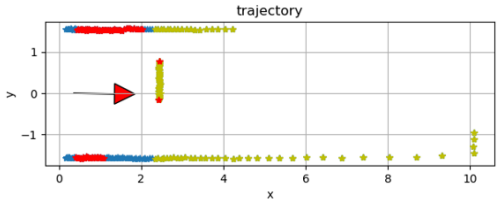
from left to right gaps grater then 1 meters between two consecutive points identified by the laser beams of the frontal cone with opening 80 degrees (points defined by the beams with indices ranging from 320 to 39). Figure 2.35-a shows this situation, here the yellow dots are the ones considered during the search. When gaps are found, of the two extreme points at each single gap, only the one closest to the robot is taken and passes to the next check. In the situation depicted in Figure 2.35-b, two gaps are identified at the sides of the obstacle, one on the left and one on the right. Of these two gaps, two red dots in Figure 2.35-c are taken.



(a) *Laser beams where the gaps are searched*



(b) *Upper view of the first tunnel*



(c) *Upper view of the first tunnel*

Figure 2.35: Step 2

- STEP 3: The next step is to check the proximity of the points obtained in the previous step, all those points that are farther than 2 meters from the robot are discarded. Figure 2.36 shows that both the points of the example are inside the circle with radius 2 meters and so both are accepted. Lets call the two dots as pivot points and laser beams indices which identifies the two dots as "k" and "n".

- STEP 4: At this point, starting from the left pivot point, the algorithm searches its nearest point starting from the one with index 270 up to the one with index k-1. Once it has found the nearest point it calculates the distance

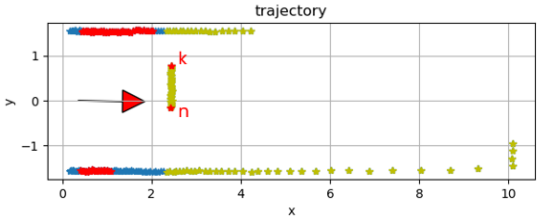
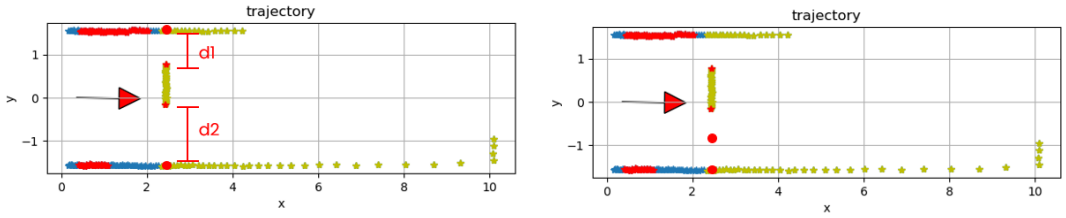


Figure 2.36: Step 3

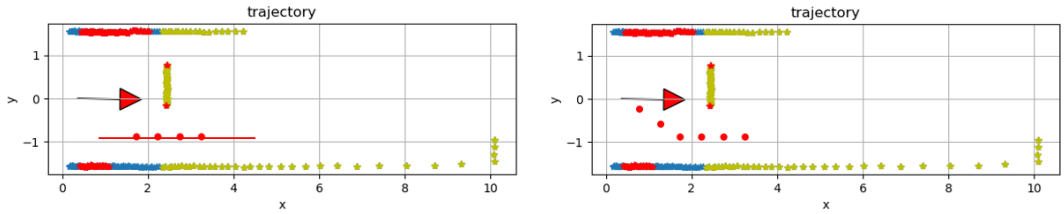
there is with the pivot point, if this distance is lesser than 1.5 times the track of the robot the pivot point is discarded because in that direction there is no sufficient space for the robot. This process is repeated until an accessible space is found. In the example, the left pivot point is discarded as shown in Figure 2.37. The check is repeated with the right pivot point by searching its nearest point starting from the one with index n+1 up to the one with index 90. This time the pivot point is accepted because the robot can access in this direction and overcome the obstacle.



(a) *Nearest points* **(b)** *Distance check*

Figure 2.37: Step 4

- STEP 5: Once the final pivot point and its nearest point have been identified, the line passing through them and the middle point between them are computed. Then, path points spaced 0.5 meter are generated along the line perpendicular to the one previously computed and passing through the middle point, as shown in Figure 2.38-a. Finally, other two equally spaced points are added between the robot and the first path point. As for the path generated in the LEFT TURN STATE, these two points are are added to let the PPA smooth as little as possible the path during the obstacle avoidance. Figure 2.38-b shows the complete generated path in the situation of the example.



(a) *Path points generation along the middle line*

(b) *Adding of two more path points*

Figure 2.38: Step 5

Once the obstacle avoidance path is created, the Pure Pursuit controller start to track it until the robot reaches at 0.5 meter the last point of the path. Together with the blind end maneuver and the sharp left turn path, also in this case the path is not updated every 1.0 seconds so as to give the robot time to overcome the obstacle. Together with the addition of the two points other precautions have also been taken to make the tracking of the obstacle avoidance as accurate as possible. Firstly, when the path is generated, the linear velocity is put to zero for 3 seconds in order to let the robot steer towards the first goal point and then follow the path. Secondly, only in the case of obstacle avoidance, the look-ahead distance is set to 0.6 meters to track the path faster.

The path generation for the obstacle avoidance can be aborted for several reasons:

- if step 2 found no gap, the robot has in front of it a blind end and so the algorithm will enter the BLIND STATE once the robot arrives sufficiently close to the wall.
- if step 3 found no pivot points means that all the points identifying the gaps are too distant. In this case the checks will continue while the robot is moving until one of them become close enough.
- if step 4 found no accessible ways the robot will treat the obstacle as a blind end entering the BLIND STATE.

To conclude this section, two summarizing tables are presented. Table 2.3 summarizes the laser beams indices used by each state during the transition checks

and the one used to generate the path. Table 2.4 underline which state of the algorithm is able to cope with each of the above mentioned situations (linear and curve paths, cross, forks, blind ends,obstacles, left and right turns).

STATE		TRANSITION CHECK INDICES	PATH GENERATION INDICES
NORMAL STATE		none	left: 270 - 315 right: 67 - 90
OPEN SPACE STATE	Left Following	67-90	270-315
	Right Turn	354-5	0-45
BLIND STATE		344-15	left: 270 - 315 right: 67 - 90
LEFT TURN STATE		270-320	variable - 320
OBSTACLES STATE		350-9	gap search range: 320-39

Table 2.3: State indices

STATE	COPE WITH
NORMAL STATE	Linear and Curve Path
BLIND STATE	Blind End
OPEN SPACE STATE	Open Spaces, Right Turn and Fork
LEFT TURN STATE	Left Turn and Cross
OBSTACLES STATE	Obstacles

Table 2.4: Situations with which each state can cope

2.5 Tests

This section presents the tests performed to analyze the behavior of the navigation algorithm, first in a simulated environment and then in a real one.

2.5.1 Simulation Tests

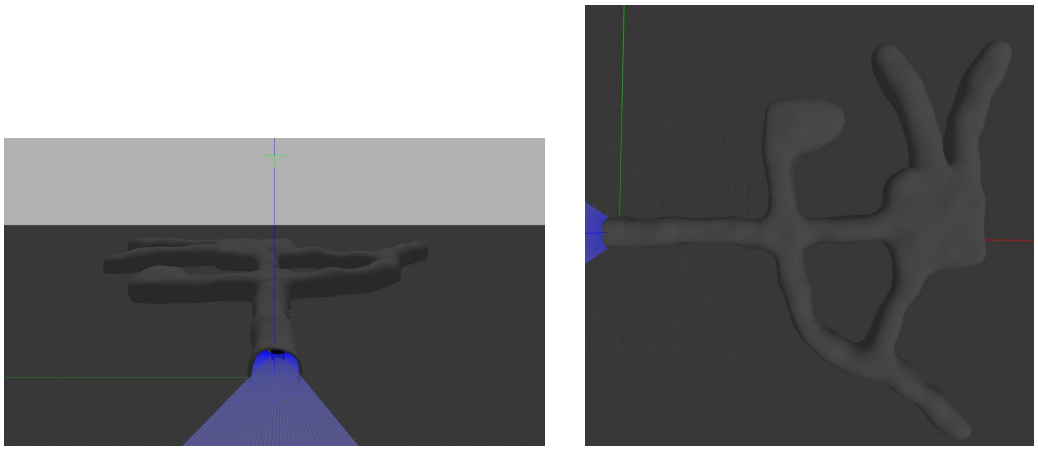
In order to test the navigation algorithm in simulated environment five models have been realized. Three of them are tunnel shaped models without obstacles. They have been implemented in Blender, converted in a .dae file and then exported into Gazebo. The other two models are smaller then the tunnels but have obstacles inside them, they have been simply realized with the Gazebo model editor using cubes. Figure 2.39, Figure 2.40 and Figure 2.41 shows the models of the tunnels. Table 2.5 summarizes their sizes.

	width (m)	length (m)	narrowness (m)
First Tunnel	32	32.2	2.3
Second Tunnel	45	42	3.5
Third Tunnel	22	30.4	4.5

Table 2.5: Tunnels sizes

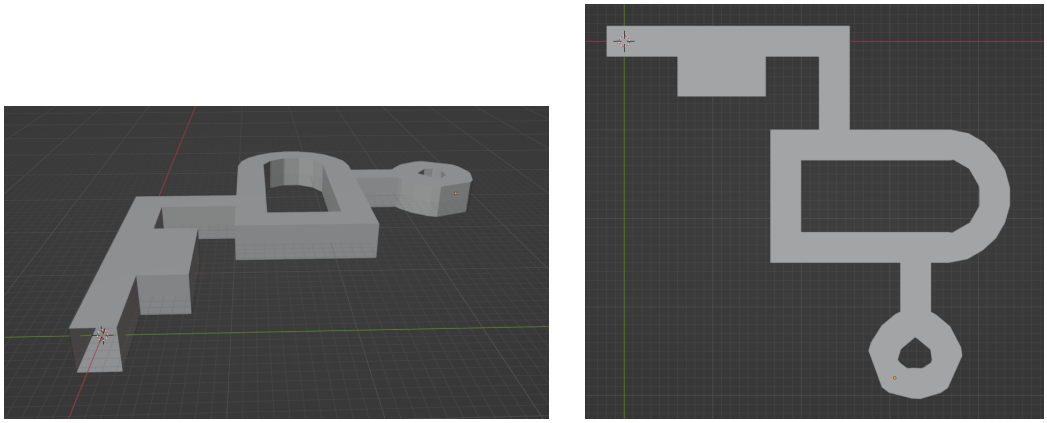
The first one (Figure 2.39) is a fairly narrow tunnel with wavy walls that simulate a real tunnel. Its open spaces, blind ends and 4-way crossing represent a good challenge for the algorithm.

The second one (Figure 2.40) is wider than the first and it has smooth walls. This aspect allows to understand how the algorithm works even in office corridors. In this case the tunnel was designed to analyze the behavior of the algorithm in situations that had not been tested with the previous model. Examples are the sudden widening of the tunnel width and its subsequent narrowing, and curves with different bending radii.



(a) *Frontal view of the first tunnel* **(b)** *Upper view of the first tunnel*

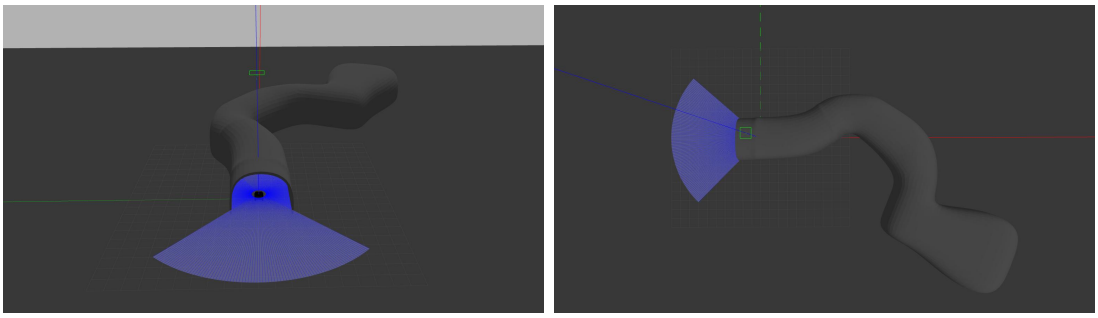
Figure 2.39: First tunnel model



(a) *Frontal view of the second tunnel* **(b)** *Upper view of the second tunnel*

Figure 2.40: Second tunnel model

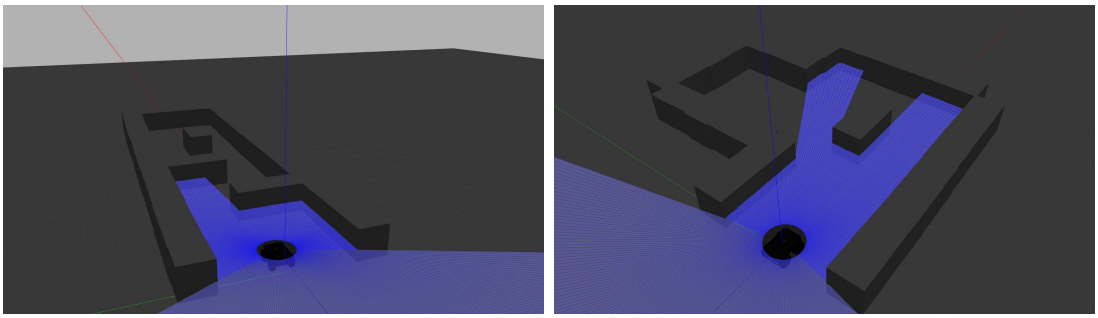
The third tunnel (Figure 2.41) has the simplest shape but it's the widest, this characteristic has been useful to understand the algorithm behaviour also in case of LIDAR beam saturation (10 meters).



(a) *Frontal view of the third tunnel* **(b)** *Upper view of the third tunnel*

Figure 2.41: Third tunnel model

Figure 2.42 shows the obstacle models. The first one is a straight route that has a S curve at the beginning and an obstacle at the center of the way. It has been mainly used to test the ability of the algorithm to let the robot overcome the obstacles. The second one has a sharp left turn with an obstacle in the middle. It has been useful to test in a deeper way the algorithm's switching capability between the OBSTACLE STATE and the LEFT TURN STATE. In fact in this situation, while the robot is following the path to tackle the sharp left turn, it encounters an obstacle on the way.

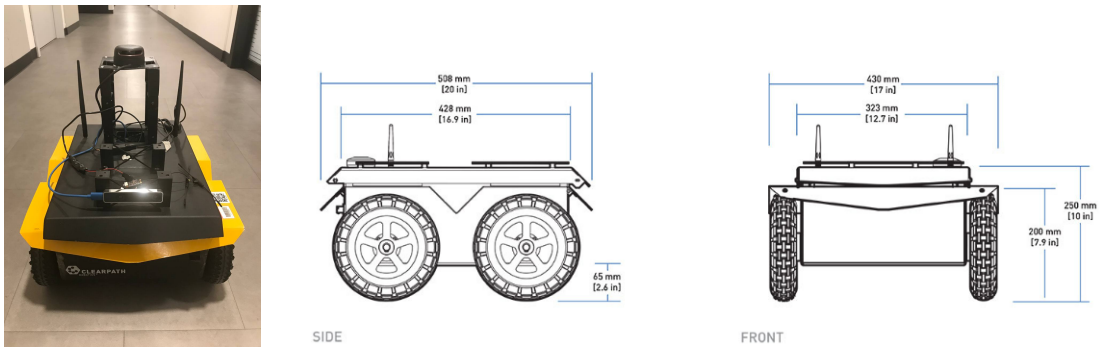


(a) *First model with obstacles* **(b)** *Second model with obstacles*

Figure 2.42: Models with obstacles

2.5.2 Experimental Tests

Since there was no possibility to test the algorithm in real tunnels, experimental tests were carried out in the corridors of the laboratories of PIC4SeR. Because of the reduced width of these corridors, the Husky was not used but the Jackal, another small differential drive mobile robot from Clearpath robotics. Also in this case the platform is fully integrated with ROS. Figure 2.43 shows a picture of the used Jackal and its dimensions.

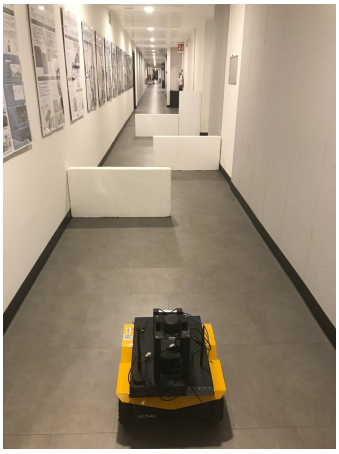


(a) *Picture of Jackal with accessories*

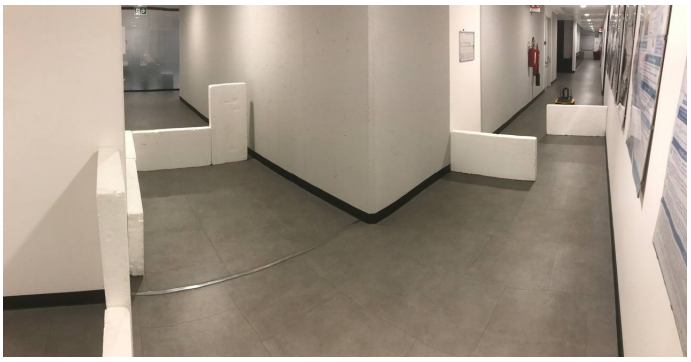
(b) *Jackal dimensions [9]*

Figure 2.43: Jackal

The robotic platform in question was the only one available in the laboratory to carry out experimental tests, but is equipped with a set of sensors slightly different from those designed in this thesis for Husky. The sensor mounted on the turret is the RPLIDAR A2M8 described in section 2.2.2. Instead, in this platform localization dosen't work with sensor fusion, it uses visual odometry through the camera mounted in front of it. Visual odometry is the process of determining the position and orientation of a robot by analyzing sequential camera images. Having said this, two scenarios have been realized to test the navigation algorithm in a real environment. The first one is shown in Figure 2.44, it is a small route which comprises two obstacles staggered at the beginning, a right turn on the outward which become a left turn at the return and a dead end. Even if small, this path makes the algorithm enter all its states and shows their interoperability. Table 2.6 shows some characteristics measure of this scenario. The main ones are the corridor width, the distance between the obstacles and the obstacle opening.



(a) *Jackal point of view*



(b) *Panoramic picture of the scenario*

(c) *Map of the first scenario*

Figure 2.44: Pictures of the first experimental scenario

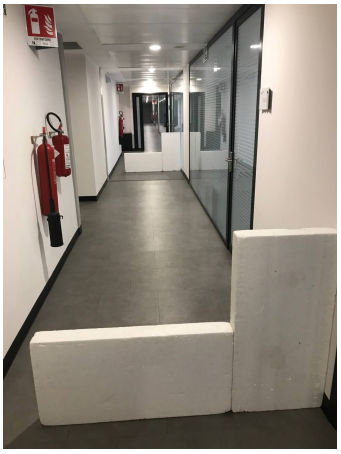
Corridor width (m)	1.8
Length (m)	7.7
Distance between Obstacles (m)	2
Obstacle Opening (m)	0.8

Table 2.6: First scenario dimensions

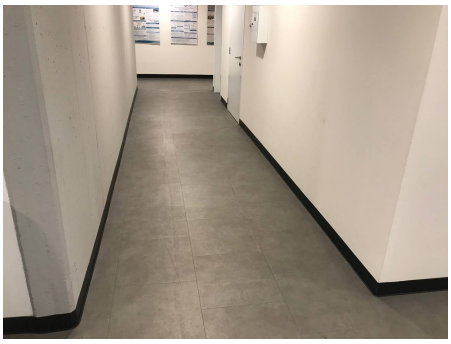
The second scenario is the one shown in Figure 2.45, it has no obstacles but it's longer then the first one. It is composed by a long corridor which ends with an open space, a secondary corridor with two blind ends and a linking corridor between them. Having intersections of several ways this scenario allows the verification of the decision logic behind the algorithm. Table 2.7 summarizes the main dimensions of this scenario.

(a) *First corridor*

(b) *Open space*



(c) *Second corridor*



(d) *Linking corridor*

(e) *Map of the second scenario*

Figure 2.45: Pictures of the second experimental scenario

Corridors width (m)	1.8
Length first corridor (m)	20.6
Length linking corridor (m)	10.7
Length second corridor (m)	8.3
Length open space (m)	4.2
Width open space (m)	1.8 + 1.3

Table 2.7: Second scenario dimensions

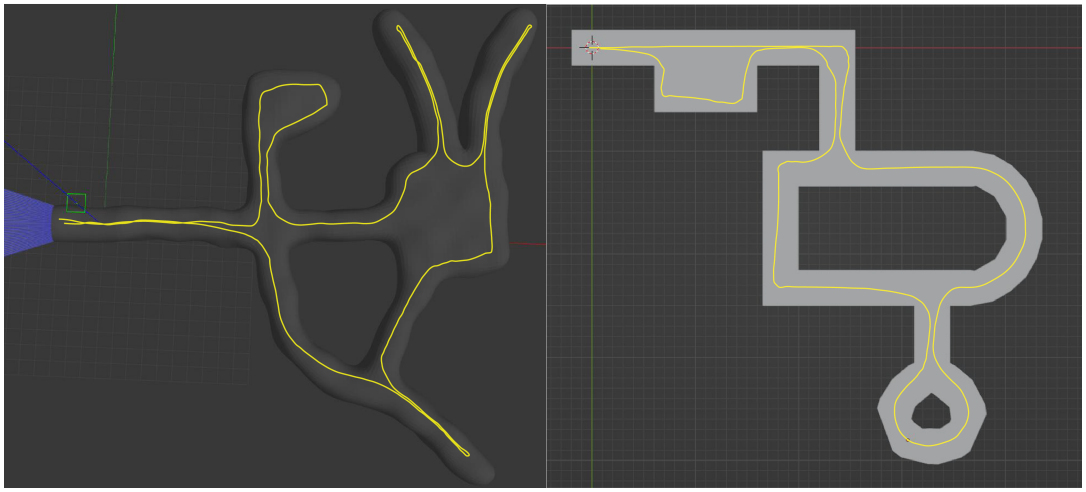
Chapter 3

Results

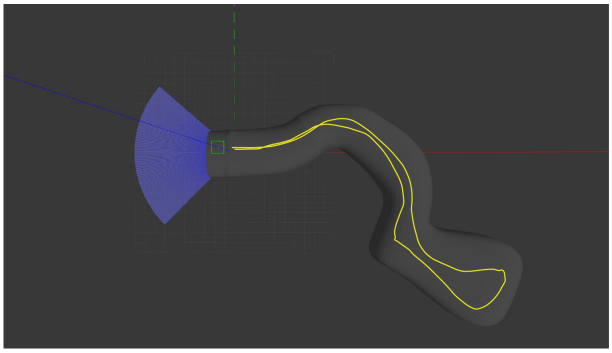
3.1 Simulation Results

This section presents the simulation results. Figure 3.1 shows the overlapping of the robot real path (yellow lines) and the tunnel models. Data representing the paths have been obtained recording, during simulations, into bag files messages coming from /odometry/filtered topic. Then they have been plotted offline with a python script. As can be seen, the algorithm worked successfully in all the cases, letting the robot cover all the tunnel plans. Indeed, the robot stays at the center of the channels also in presence of curves with small bending radii, in the presence of crosses it always take the left way ignoring the others and reactively come back when approaching blind ends. Figure 3.1-b shows that when the robot approaches the widening of the corridor from the left it is ignored since the algorithm enters in OPEN SPACE STATE. Instead, when the robot arrives from the right it runs along the perimeter of the enlargement as desired since the exploration should cover all the tunnel plan. But there is no specific state that has to deal with these situations, indeed, this behaviour is made possible thanks to the combination of path's high update rate and OPEN SPACE STATE. In particular, when the robot approaches the enlargement from the right, before it tries to enter in the LEFT TURN STATE, but the transition is aborted since the sufficient condition is not met. Then, it sees a sharp change of slope of the line to follow towards the enlargement, as a consequence it starts steering left until it enters the OPEN SPACE STATE (the

laser beams on the right become almost parallel to the right wall). Being entered in this state, it continues to follow the left wall running all the perimeter of the enlargement.



(a) *First tunnel result* **(b)** *Second tunnel result*

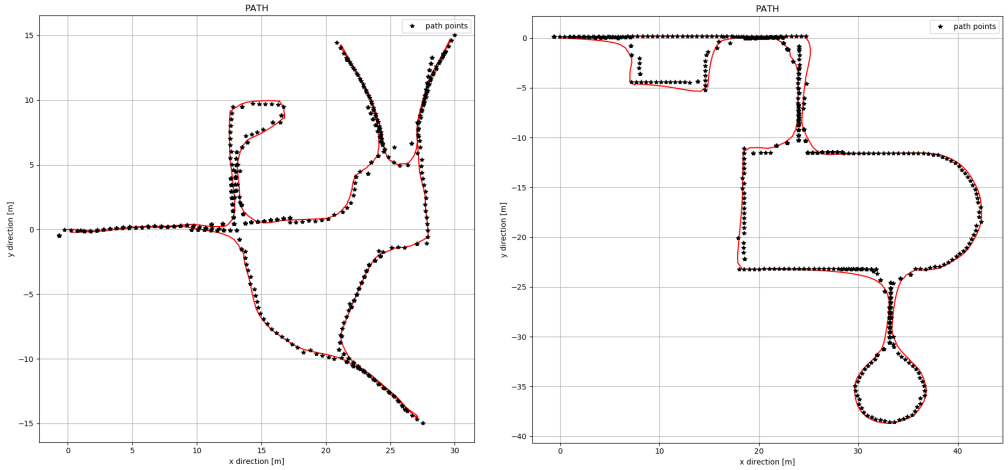


(c) *Third tunnel result*

Figure 3.1: Paths overlapped with tunnel models

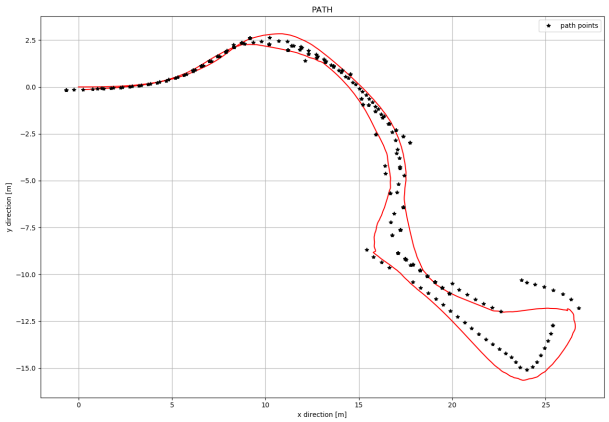
Figure 3.2 represents a comparison between the robot real path (red lines) and the path points generated by the state machine (black stars). Path points have been obtained recording into bag files messages coming from /path topic, representing the instantaneous path the robot sees. It is good to remember that the instantaneous path generated by the state machine is 4 points in all states except in the LEFT TURN STATE where they are 10 and in the OBSTACLES STATE where they are 6. Therefore, to keep the graph more readable, only the first point

of the 4 points paths and all the points of the other paths have been represented. These pictures show in a visual way that the wider the tunnel section the greater the deviation between the real path and the points. Indeed, Figure 3.2-c shows, at the end of the tunnel (about 6 meters wide), the maximum deviation. It also clearly underlies the effect of an high look-ahead distance (1.3 meters) since the path converges slowly at the tracking points.



(a) *First tunnel result*

(b) *Second tunnel result*

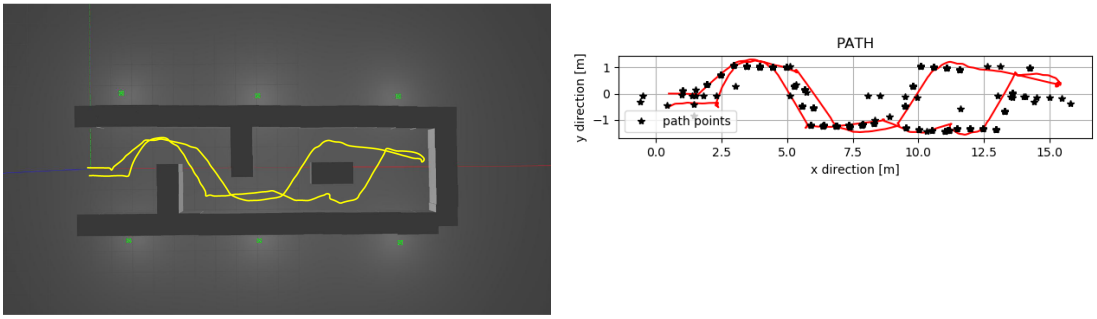


(c) *Third tunnel result*

Figure 3.2: Comparison between actual path and path points of tunnel models

Instead Figure 3.3 shows the results of the simulation with the first obstacle model. Figure 3.3-a shows the overlapping of the robot real path (yellow lines) and

the obstacle model. Here it is possible to observe that the robot runs nicely the route despite all the obstacles inside it. It first perform the manoeuvres to overcome the S curve by detecting two times obstacles in front of it, then it recognizes another obstacle at the warning distance and takes the first accessible path (which is on the left). Then it proceeds straight until it reaches the blind end, it turns back and, once arrived again at the central obstacle, it overcomes the obstacle by passing trough the other accessible way. It should be pointed out that the same path of the previous avoidance is not taken because the algorithm checks accessible routes starting from the left side. In Figure 3.3-b the reduction of the look-ahead distance when approaching the obstacles path can be appreciated, in fact in these situations the robot track faster the path.

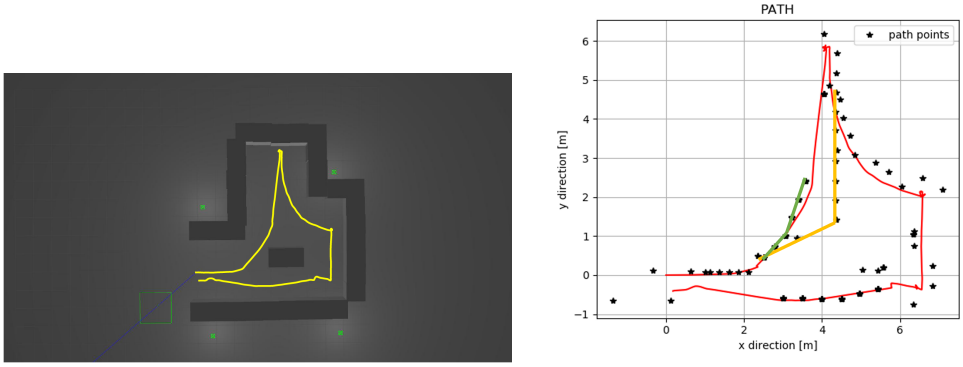


(a) *Paths overlapped with the first obstacle model* **(b)** *Comparison between actual path and path points*

Figure 3.3: First obstacle model result

Figure 3.4 presents the result of the simulation with the second model, here the robot enters the LEFT TURN STATE, generates the corresponding path but while following it the robot approaches an obstacle. At this moment, even if the robot has not yet completed the path of the curve, the algorithm readjusts the path for overcoming the obstacle preventing the robot from crashing. This result demonstrates the ability of the algorithm to switch between states when needed and can be better appreciated in 3.4-b. Here the path represented in yellow is the one undertaken for the LEFT TURN STATE that would lead the robot to crash

into the obstacle. As soon as the robot notices the obstacle, the algorithm goes into OBSTACLES STATE and generates the green path that allows the robot to overcome the obstacle.



(a) *Paths overlapped with the first obstacle model*

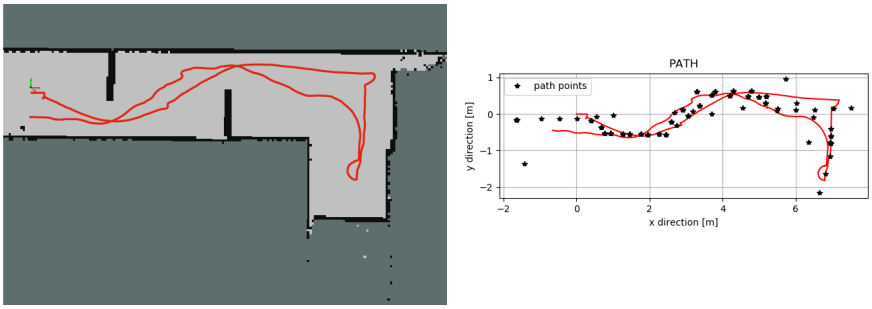
(b) *Comparison between actual path and path points*

Figure 3.4: Second obstacle model result

3.2 Experimental Results

As in simulation tests, also during experimental tests messages from the main topics have been recorded into bag files. This time the topics recorded are: /odom (pose estimate obtained from visual odometry), /laser/scan (data from LiDAR), /tf (topic which keeps track of all the transformations between reference frames) and /path. All these topics have been then analyzed and processed offline plotting them or playing them back in RViz. Figure 3.5 shows the path computed by the robot in the first scenario.

As usual, the path represents the plot of the robot position at each time step, while the map below the path has been obtained offline playing back /tf and /laser/scan data in RViz. Then through a ROS package called slam_toollbox the map of the scenario have been obtained and saved. As can be seen from the picture, the robot has completed successfully the first scenario overcoming all the obstacles. More information about this test can be found in the Table 3.1. Here are some metrics used in the literature in the analysis of navigation algorithms.



(a) *Paths overlapped with the map of the first scenario*

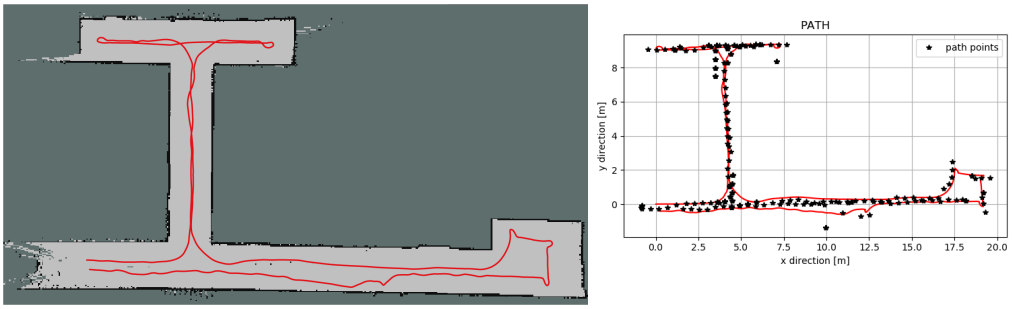
(b) *Comparison between actual path and path points*

Figure 3.5: Results of the first scenario

The coverage represents the percentage of the scenario's plan that the robot has explored; in the case of the first scenario there are no crossing and so the coverage is obviously 100%. Instead the other metrics have been computed offline. While the bag is played back in RViz, a node is launched in parallel which measure the minimum range and the average of all the laser ranges at each laser scan. Once the node is stopped, it return the minimum (min min), the average (mean mean), the minimum of the averages (min mean) and the average of the minimum (mean min) of all the scans. The minimum represents the minimum distance achieved by the robot from an obstacle during all the test. The average is the average distance that the robot has held from objects during all the test. The minimum of the averages is the moment when the robot has had the average of all the laser ranges smaller. Finally, the average of the minimum represents how much the robot has averagely approached more to the objects during the test.

Concerning this case, the most significant metric is the minimum reached during all the test and the average of the minimum of all the scans in order to understand how far the path of the robot has deviated from the halfway crossing of the obstacle opening. As can be seen in Table 3.1 the minimum of the first scenario is 0.3 meters which has been reached when the robot is overcoming an obstacle. This is a quite good result being the obstacles opening 0.8 meters wide.

With the same procedure also the results for the second scenario have been obtained as shown in Figure 3.6 and Table 3.1.



(a) *Paths overlapped with the map of the second scenario*

(b) *Comparison between actual path and path points*

Figure 3.6: Results of the second scenario

Also in this case the result is good, the robot has covered all the plan of the scenario reaching a 100% coverage also in presence of crossing. As can be seen in Figure 3.6, from the starting point the robot has always followed the left wall, as wanted. Also in the presence of the open space it has skirted the perimeter as provided by the OPEN SPACE STATE. Concerning this scenario, since there are no obstacles in the way, the main metrics to consider are the coverage and the average of the minimum of all the scans. This last parameter allows to quantify how much the robot has held the center of the path during the test. In fact, in the absence of obstacles, being the corridor 1.8 meters wide, the excellent would have been to obtain a value of 0.9 meters. Instead the value obtained is 0.77 meters, which is still a good result. This parameter deteriorated during the final part of the test, in fact, as it can be seen from Figure 3.6-a, in the final part of the return the path of the robot deviates from the center of the corridor. This deviation can be attributed to a visual odometry drift. Another metric to note is the minimum obtained in this scenario, which turns out to be less than the first scenario even if here there are no obstacles. This value was obtained at the return of the main corridor as can be observed in the Figure 3.6-a. The robot's sudden deviation towards the left wall is due to the fact that a glass door in the corridor was poorly covered, leaving two strips of glass uncovered at the sensor height. In fact, the LiDAR's laser beams are not reflected by the glass surfaces but pass through them. The result is that the resulting points of these rays are much more distant than the

wall. Since these anomalous points are still fitted by the algorithm, the resulting line, which should represent the left wall, is actually much more inclined than the latter. This chain of events leads the algorithm to follow the robot along an instantaneous path that would collide with the left wall. This can be seen in the Figure 3.6-b, where at the bottom there are path points that are very far from the center of the corridor. Fortunately the robot then recovered from the detour by entering the OPEN SPACE STATE and starting to follow the left wall.

	First Scenario	Second Scenario
min min (m)	0.30	0.26
mean min (m)	0.63	0.77
mean mean (m)	1.58	1.99
min mean (m)	1.30	1.40
coverage	100%	100%

Table 3.1: Experimental metrics

Chapter 4

Conclusions and Future Development

Simulations and experimental tests are promising, having demonstrated the effectiveness of the navigation algorithm for the task it is required to perform. In fact the design requirements were all met. Since the experimental tests unfortunately allowed only to test the algorithm and not the entire robotic platform designed, in future, tests of the algorithm should be performed in real tunnels together with the Husky and sensors described in this thesis. The obstacle avoidance functionality has also been successfully tested both in simulation and in real environment. It should be stressed, however, that the proposed obstacle avoidance approach can only be suitable for static and not dynamic obstacles. Anyway, in the context in which the robotic platform is called to operate should not present scenarios with mobile obstacles. Moreover, since a two-dimensional LiDAR is used, the platform can detect obstacles that are higher than the height at which the sensor is mounted (45 centimeters for the platform of experimental tests). Therefore in the future it should be added another distance sensor on the front of the robotic platform at wheel height with a field of view of about 180 degrees. One of the negative aspects of behaviour-based architectures is the difficulty of adding a new behavior (state in this case) and integrating it to those already present. However, this should not be a problem with the proposed algorithm, as the states (or behaviors) implemented are already sufficient to meet the design requirements. It must be said, however,

that the algorithm fails in the requirement of the total exploration of the tunnel in the presence of internal loops. Figure 4.1 shows an example of a scenario with internal loop. In the case where the tunnel to be explored is done in this way and the robot's inspection starts at the red arrow, the robot would walk the perimeter of the tunnel and exit without inspecting the inner ways.

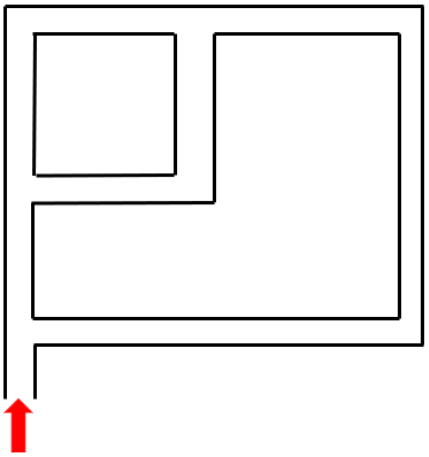


Figure 4.1: Example of a scenario with internal loop

Improvements also can be made in the robustness of the algorithm, as currently transition checks for some states depends on considerations over individual laser beams and it has been noted that in reality the LiDAR can sporadically give untrue values on single beam ranges. In conclusion, this algorithm can be a starting point to get a better one by adding such refinements.

Bibliography

[1] D. Scaramuzza R. Siegwart I. R. Nourbakhsh. *Introduction to Autonomous Mobile Robot*. Massachusetts Institute of Technology, 2011 (cit. on pp. 1–5, 17, 23, 24).

[2] S. B. Mohd Noor D. Nakhaeinia S. H. Tang and O. Motlagh. «A review of control architectures for autonomous navigation of mobile robots». In: *International Journal of the Physical Sciences* 6(2) (Jan. 2011) (cit. on pp. 1, 5, 6).

[3] W. Al-Sabban J. Martz and R. N. Smith. «Survey of unmanned subterranean exploration, navigation, and localisation». In: *IET Cyber-systems and Robotics* (Mar. 2020) (cit. on p. 7).

[4] URL: `http://wiki.ros.org/ROS/Introduction` (cit. on pp. 9, 11).

[5] R. Jung Y. Pyo H. Cho and T. Lim. *ROS Robot Programming*. ROBOTIS Co.,Ltd., 2017 (cit. on pp. 11, 12).

[6] URL: `https://www.linuxadictos.com/it/robot-simulatore-di-gazebo.%20html` (cit. on p. 13).

[7] URL: `http://wiki.ros.org/husky_gazebo` (cit. on p. 13).

[8] URL: `https://www.blender.org/` (cit. on p. 14).

[9] URL: `https://clearpathrobotics.com/` (cit. on pp. 15, 16, 58).

[10] P. Corke. *Robotics, Vision and Control, 2nd edition*. Springer International Publishing, 2017 (cit. on pp. 16, 17, 25).

[11] O. Aydogmus and G. Boztas. «Implementation of Pure Pursuit Algorithm for Nonholonomic Mobile Robot using Robot Operating System». In: *Backal Journal of Electrical Computer Engineering* 9(4) (Oct. 2021) (cit. on p. 16).

[12] URL: `http://wiki.ros.org/diff_drive_controller` (cit. on p. 18).

[13] URL: `https://www.slamtec.com/en/Support#rplidar-a-series` (cit. on pp. 19, 20).

[14] W. Burgard S. Thrun and D. Fox. *Probabilistic Robotics*. 2000 (cit. on p. 22).

[15] URL: `http://wiki.ros.org/robotlocalization` (cit. on p. 26).

[16] S. Sun J. Iqbal R. Xu and C. Li. «Simulation of an Autonomous Mobile Robot for LiDAR-Based In-Field Phenotyping and Navigation». In: *Robotics* (June 2020) (cit. on p. 28).

[17] J. Morales, M. A.Martìnez J. L.Martìnez, and A. Mandow. «Pure-Pursuit Reactive Path Tracking for Nonholonomic Mobile Robots with a 2D Laser Scanner». In: *EURASIP Journal on Advances in Signal Processing* 2009 (Jan. 2009) (cit. on pp. 29, 31, 32).

[18] R. Craig Coulter. «Implementation of the Pure Pursuit Path Tracking Algorithm». In: 1992 (cit. on p. 32).

[19] Y. Stein. «Two Dimensional Euclidea Regression». In: (June 1983) (cit. on pp. 33, 34).

[20] Y. Fang H. Gao X. Zhang and J. Yuan. «A line segment extraction algorithm using laser data based on seeded region growing». In: *International Journal of Advanced Robotics Systems* (Dec. 2017) (cit. on p. 33).