Politecnico di Torino

POLITECNICO DI TORINO

Master of Science Degree in Mechatronic Engineering

Master Thesis

LiDAR-based Dynamic Path Planning of a mobile robot adopting a costmap layer approach in ROS2

Supervisor:
Prof. Marina INDRI
Advisor at Links:
Dott. Gianluca PRATO

Candidate:
Matteo DE ROSE

December 2021

Contents

Abstract

In the last two decades, the deep technological developments in the fields of artificial intelligence, computing platforms and sensors boosted the use of autonomous mobile robots in complex environments, such as factories, airports, offices, caves and also in hospitals. One of the first problems that agents have to overcome for a successful navigation is the avoidance of dynamic obstacles.

The LINKS Foundation in Turin carries out research in different domains, such as AI, IoT, Cyber-physical and autonomous systems. This thesis was born as one of the first building blocks of a LINKS project: build up an autonomous system of internal mail delivery across its offices, based on a fleet of mobile agents. At a first stage Turtlebot robots will be used for this purpose; each of which shall achieve the navigation goal relying on LIDAR data only. The objective of this thesis is to implement an efficient navigation within these constraints, exploiting ROS2, the recently released version of the widely established Robotic Operating System. At the same time, this thesis work aims at helping to solve the open points about dynamic obstacles avoidance in ROS2 Navigation Stack (Nav2), raised by the ROS2 community. Indeed, the latter does not embeds a strategy specific for dynamic obstacles handling during navigation.

A research about the autonomous navigation problem and the ROS2 functionalities has been carried out. Then, it has been proposed and implemented a method to integrate dynamic obstacles handling within the Nav2 stack available in ROS2, exploiting the concept of costmap layers. The proposed approach aims at laying the base for a highly modular pipeline for navigation, allowing future adaptation to specific use cases in LINKS projects. Finally, simulations in virtual environments have been carried out using Webots simulator and Rviz, outlining improvements marings and critical points for the future work at LINKS.

Introduction

In the last two decades, the deep technological developments in the fields of artificial intelligence, computing platforms and sensors boosted the use of autonomous mobile robots in complex environments. Indeed, this research area is more relevant than ever before, as shown by a recent analysis on "IEEE Xplore" digital library [1]. Mobile robots are actually used in industrial automation, surveillance, transportation, personal and medical applications. In industries, mobile robots are revolutionising flexible manufacturing systems and logistics, where automatic guided vehicles (AGV's) have been dominant for years. Localization and navigation of commercial AGVs is still commonly accomplished by wire guidance where induction is sensed from electrified wires embedded in the floor. Several other navigation methods are available today, including ceiling mounted bar codes, range or camera-based wall-following, using floor markers or magnets, and following magnetic tape. However, even though AGV's are a well established solution for material handling, they have two main disadvantages as highlighted by Arkin and Murphy [2]: a limited drivepath and a limited interaction with the workstations. On the other hand, mobile autonomous robots are developed as intelligent agents that can interact actively with the industrial environment and so, they better attain the level of flexibility envisioned by Industry 4.0 revolution. For example, Angerer, Pooley, and Aylett [3] propose a multi-agent system for dynamically reconfiguring mobile robots to accomplish a range of variations of tasks in an automobile factory thanks to the customizable features of each individual vehicle. Their fleet management system includes an ontology describing the objects and where they are located in the facility, and a set of tasks that the system is able to carry out. The system can dynamically generate new agents to execute tasks that may arise when the environment changes. Robotic automation is relieving human workers of many repetitive tasks also in healthcare. When combined with a fleet management system, autonomous mobile robots (AMRs) boost worker productivity and safety, and improve patient care. For example, the University of California San Francisco (USCF) Medical Center has been using TUG robot by Aethon since 2015 [4]. This robot securely delivers medications to nursing

units. It automates deliveries that are normally made through pneumatic tubes or by manual couriers, even in case of controlled substances and refilling carts. Best of all, by using biometric security and unique pin codes, TUG ensures only authorized medical personnel to add or remove specimens to the secured cabinet. Finally, TUG provides also automated, cost-effective delivery of meals to patient floors and returns dirty trays to Food Service. Finally, one of the most developed applications of mobile agents are surveillance and inspection. The increasing need for automated surveillance of indoor environments, such as airports, warehouses, production plants, etc. has stimulated the development of intelligent systems based on mobile sensors. Differently from traditional passive surveillance devices, which can only detect events and trigger alarms, to active surveillance, robots can be used to interact with the environment, with humans or with other robots for more complex cooperative actions. For example, S5 security robots by SMP Robotics Systems Corp. [5] are designed to replace patrolling security guards and to provide mobile CCTV[1] monitoring. This security robot moves around a restricted area automatically, without direct operator supervision. Images from its built-in cameras are transmitted to the security station. If a stationary security sensor is triggered, the robot changes its route and moves to the location of the possible alert. On the other hand, many inspection robots have been developed in order to carry out fast and precise inspections in hazardous environment in place of humans. Quadruped robots that can adapt to almost any kind of terrain are quite diffused nowadays. These robots can provide invaluable help after catastrophes by navigating and mapping the environment. Some of them are even equipped for rudimentary manipulation, enabling them to interact with the environment and use certain tools. Sensors mounted on the robots can provide critical information regarding the state of the surroundings, like radiation levels, temperature, the presence of toxic or flammable gas to ensure the safety of human resources. Robot *Spot* developed by Boston Dynamics is an exemplary case. Among the different applications of this robot, there is Kidd Creek case [6]. Kidd is the metal mine with deepest base on Earth and boasts the longest surface-to-bottom underground ramp. Here, Spot is used to inspect underground development faces after a blast and check that all the blast charges have detonated before crews arrive to dig deeper into the Kidd Mine. In this way, people are kept out of harm's way and the job can be done without the need of waiting until the toxic gasses have been vented out of the area.

This thesis work is realized in collaboration with the LINKS foundation of

[1] closed-circuit television

(a) *Agile 1500* by *Comau* - https://www.comau.com/en/competencies/robotics-automation/collaborative-robotics/automatic-guided-vehicles-agv/

(b) TUG robot by *Aethon* in the University of California San Francisco Medical Center.

(c) S5 security robot by *SMP Robotics Systems Corp* patrolling an airport.

(d) Spot robot during a mine inspection. - https://www.techeblog.com/boston-dynamics-spot-robot-dog-underground-kidd-creek-mine/

Figure 1: Some mobile robots applications.

Turin. The LINKS Foundation – Leading Innovation & Knowledge for Society – is a no-profit private Foundation founded in 2016 with the aim to boost the interaction between research and the business world towards the internationalization of the local socio-economic system. It carries out research in different domains, such as AI, IoT, Cyber-physical and autonomous systems and cybersecurity. One of the projects of LINKS is to build up an autonomous system of internal mail delivery across its offices. The system consists in an intelligent infrastructure capable of managing a fleet of robots in charge of transporting the mails. Each robot shall be able to navigate autonomously on the base of a pre-loaded static map of the building, avoiding every kind of dynamic obstacle encountered on its way. At a first stage Turtlebot robot will be used, which shall acheive this goal relying on LIDAR data only. The objective of this thesis is to implement an efficient navigation within these constraints, exploiting ROS2, the recently released version of the widely established Robotic Operating System. At the same time, this thesis work aims at helping to solve the open points about dynamic obstacles avoidance in Navigation Stack (Nav2), raised by the ROS2 community. Indeed, the latter

does not embeds a strategy for dynamic obstacles handling during navigation.
First of all, a research about the state of the art of autonomous navigation has been carried out. The aim was to define the problem of navigation in formal terms and then explore the multiple currently exploited solutions, in order to identify the best ones for the indoor autonomous navigation of the Turtlebot. Secondly, an accurate study of the ROS2 framework was needed. As mentioned above, ROS2 is a quite recent platform, thus it lacks of exhaustive documentation, requiring a direct look at the source code to understand some basic new functionalities. Moreover, the LTS release (Foxy) still had some problems when the thesis work started in March. So, the goal was to correctly configure the whole development environment (Linux Ubuntu OS, ROS2 Foxy, Nav2 stack and simulation tools) on the Intel® NUC 8 computer given by LINKS, solving the eventual conflicts. After that, Webots simulation environment had to be configured with the new ROS2 version, and validated through the available demo packages simulations. After studying Webots architecture, two simulated environments have been prepared and validated with the basic navigation stack configuration. Once that the whole framework was understood, configured and validated, the efforts focused on how to implement a strategy for dynamic path planning. The idea was to exploit the possibility to create costmap layer plugins, in order to lay the base for a highly modular pipeline for navigation, allowing future adaptation to specific use cases in LINKS projects. The developed approach has been finally validated by simulations. However, margins of improvements are multiple and an extensive batch of tests is necessary to optimize performances, but this is out of the scope of this thesis work.
Here it follows a brief description of the chapters content, in order to ease the reading of this thesis work.

Chapter 1 provides the formal definition of the indoor autonomous navigation problem, outlining the taxonomy of the current algorithm solutions.

Chapter 2 explains the major innovation of ROS2 with respect to ROS and outlines the main characteristic of the Navigation Stack used to implement dynamic path planning in this thesis work.

Chapter 3 presents some information about the current navigation strategies developed in Navigation Stack by the research community. Then it explains the theory behind the developed approach and its implementation.

Chapter 4 illustrates how the methods outlined in the previous chapter has been implemented in the ROS2 framework.

Chapter 5 shows how the simulation environments have been created and the results of the simulation tests.

Chapter 1

Indoor autonomous navigation

The aim of this chapter is to introduce the reader to the topic of autonomous navigation of mobile robots in industrial domains as well as in common service robotics applications. The problem will be tackled with a specific focus on indoor environments, providing a formal description and highlighting all the critical functions needed to accomplish the navigation task. Furthermore, some of the most common algorithms and methodologies to solve the planning problem in navigation are presented; alongside there is a classification, useful to locate the solution proposed in this thesis work in the current panorama.

1.1 Introduction

The foundations of mobile robotics are the fields of locomotion, perception, cognition and navigation [7]. Although there exists a huge amount of diverse mobile robots concerning the locomotion system, this work only deals with wheeled robots.

The term *perception* indicates the way by which the robot acquires knowledge about the surrounding environment. Therefore, it usually refers to the multiplicity of sensors that can be integrated on a robot for this purpose: encoders, ultrasonic sensors, infrared sensors, laser range finders, depth sensors, etc.

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 high-level objectives. Usually, the robot adopts cognitive models to represent itself and the environment and, based on that, the control system plans the robot's motion. Therefore, cognition is the central node of a robot's 'intelligence'.

Finally, *navigation* is the robot's comprehensive skill allowing it to move from one place to another in a known or unknown environment. Thus, the

navigation skill is the result of the integration of perception data, localization (the robot awareness of its position and orientation), cognition and motion control. Navigation can be decomposed into the following tasks [7]:

- Generate the model of the world in the form of a map.
- Compute a collision-free trajectory from a starting position to a target position.
- Move along the calculated trajectory, avoiding collision with obstacles.

The following section will provide a more precise formulation of the navigation problem.

1.2 Problem statement

The problem of autonomous navigation is broad and complex, since it requires the integration of many different functional blocks to be solved. In Fig. 1.1 there is a simplified scheme that highlights the basic relationships among the cited functional blocks and introduces some of the terminology that will be used in this work. This scheme is valid both for most robots [8] and for autonomous vehicles (self-driving cars) [9] and it displays the architecture of an autonomous agent on three levels: perception, decision making, vehicle.

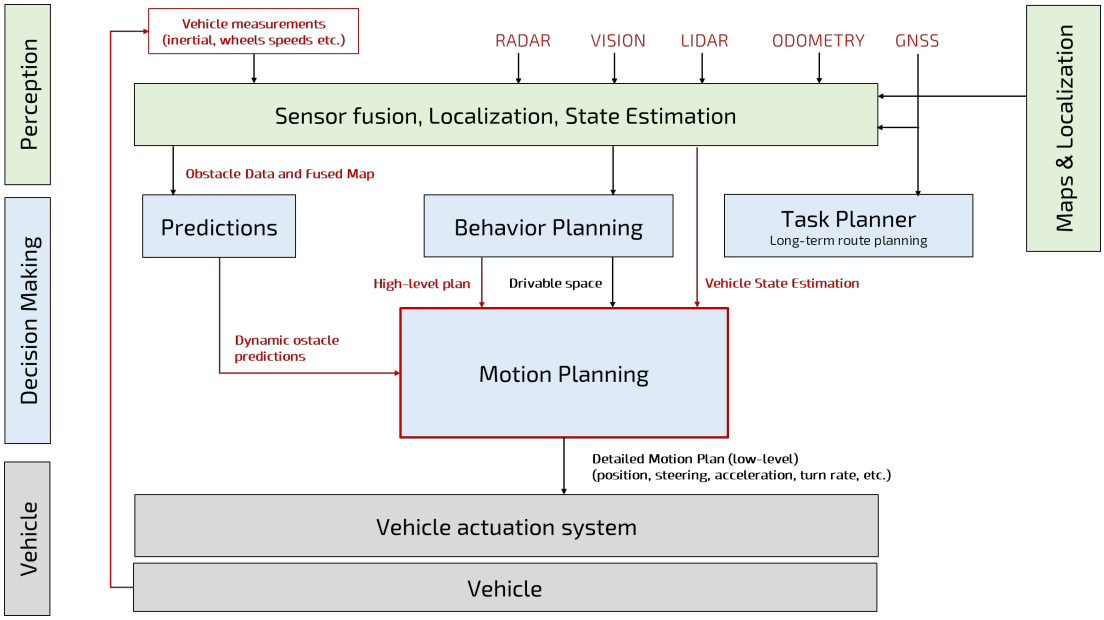


Figure 1.1: General architecture of the autonomous navigation problem.

Perception. As mentioned in Section 1.1, perception includes the software components that process the sensor data in order to determine the position of the agent in the workplace (*Localization*) and its state variables (*State Estimation*). These knowledge might be augmented by additional information such as pre-loaded maps of the environment.

Decision Making. When a pre-loaded map is available, this is usually feeded to the *Task Planner* (or *Route Planner*) – possibly together with GNSS data if available, which considers high-level objectives and plans a route to the goal state. Then, the *Behavior Planner* combines the goal/task information from the Task planner with the sensor data and uses them to check feasibility, adjust the plan based on environmental constraints and trigger re-planning if needed. Alongside Task and Behavior Planners, usually, a separated functional block (*Predictions*) extracts information about obstacles – such as pedestrians, other robots or moving objects – and estimates their behavior. The *Motion Planner* elaborates the dynamic obstacle predictions, the high-level plan and the drivable space to compute the actual motion sequence to reach the goal without breaching any constraint.

Vehicle. The Motion Planner can output either steering, acceleration or velocity commands directly, or a sequence of states. This depends on the vehicle actuation system and type of robot (differential drive, car-type, omnidirectional, synchro drive, etc.).

1.2.1 Motion Planning

After the presentation of a general architecture of an autonomous agent, this work will focus mainly on the motion planning topic. Hence, it follows a more formal definition of the motion planning problem, based on the configuration space theory, as proposed in La Valle book [10]. First, some basics definitions are given below:

- The *world* (or *workspace*) $\mathcal{W}$ is the space where the agent exists; so, usually $\mathcal{W} = \mathbb{R}^3$, but for simpler problems it is sufficient $\mathcal{W} = \mathbb{R}^2$. Therefore, the robot $\mathcal{A}$ and the obstacle regions $\mathcal{O}$ are considered closed subsets of $\mathcal{W}$.
- A configuration q of a robot is the set of all the parameters defining the pose of the agent in $\mathcal{W}$. For example, in the simplest robot model $q = (x_t, y_t, \theta)$ if $\mathcal{W} = \mathbb{R}^2$.

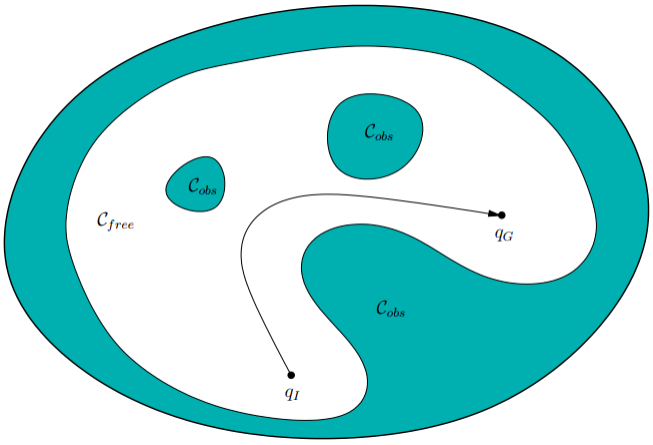


Figure 1.2: Graphical representation of the motion planning problem based on the $\mathcal{C}$-*space* concept [10]. The black arrow represents the path from the initial to the goal position across the $\mathcal{C}_{free}$ space.

- The *configuration space*, expressed as $\mathcal{C}$-*space*, is the set of all the possible configurations the robot can assume due to a *transformation*. Indeed, it is a *topological manifold*.
- Given $\mathcal{A}(q)$ the agent in configuration $q \in \mathcal{C}$, the *free-space* is defined as $\mathcal{C}_{free} = \{q \in \mathcal{C} \,|\, \mathcal{A}(q) \cap \mathcal{O} = \emptyset\}$.
- The *obstacle-space* is so defined as $\mathcal{C}_{obs} = \mathcal{C} \setminus \mathcal{C}_{free}$
- $q_I \in \mathcal{C}_{free}$ indicates the *initial configuration* and $q_G \in \mathcal{C}_{free}$ indicates the *goal configuration*.

Therefore, the **motion planning** problem can be defined as the problem of finding a continuous path τ that moves an agent from a starting pose q_I to a goal pose q_G avoiding obstacles (Fig. 1.6). Formally, It is about finding τ such that:

$$\tau : [0, 1] \rightarrow \mathcal{C}_{free}, \qquad \tau(0) = q_I, \qquad \tau(0) = q_G$$

1.3 Planning Algorithms

In the years, a great number of algorithms have been developed to solve the motion planning problem, based on different mathematical approaches and technologies. In this section, some of the most typically deployed algorithms are categorized in families based on their common basic principles. However, this is not meant to be an exhaustive taxonomy of all the available planning algorithms, it rather aims at helping the reader in locating the solution proposed in this thesis work inside the current scenario of motion planning approaches.

1.3.1 Metrics

In order to identify a family of algorithms, the following metrics are taken into consideration [11]:

- **Type of output**: a path, a trajectory, a symbolic representation or a maneuver, compatibly with the control interface of the actuation system. In this regard, *set-algorithms* are distinguished from *solve-algorithms*. The former return as output only a decomposition of the workspace – so a complementary algorithm is needed to find the feasible motion, the latter directly output the sequence linking the initial pose to the goal pose.
- **Space-time property**: essentially, according to this property, planning algorithms can be *predictive* or *reactive*. Predictive algorithms generate a set of possible motions and then select the best one according to the objective behavior, based on a longer time horizon. Reactive algorithms deform the high-level plan considering a shorter range of actions, in a shorter time horizon. Thus, the algorithms of the first class have better performance, but they are more computationally demanding than the latter.
- **Mathematical domain**: heuristic, biomimetic, geometric, logic, etc.

1.3.2 Taxonomy

Here it follows a taxonomy of algorithms based on the metrics mentioned above, six categories have been identified.

Space configuration

It is a class of set-algorithms whose aim is a particular decomposition of the *evolution space* – the workspace $\mathcal{W}$ augmented with the time dimension. These algorithms are based on geometric aspects and can be used either in a predictive fashion – choosing a coarser space decomposition to reduce computational costs, or in a reactive manner with a finer decomposition for better accuracy. Therefore, the criticality of these algorithms is the denseness of the space decomposition: with a too coarse decomposition it might not be possible to accomplish kinematics constraints, while a too fine decomposition may result in shoddy real-time performance. They are characterized by three subsequent steps [11]:

1) sample or discretize the evolution space;
2) discard the discrete elements of the space (points, cells or lattices) in collision with obstacles or unfeasible;

3) Either output the resulting $\mathcal{C}_{free}$ space, or operate a pathfinding step and directly send the waypoints to the control block.

In this class, three families can be identified on the base of the denseness of the decomposition:

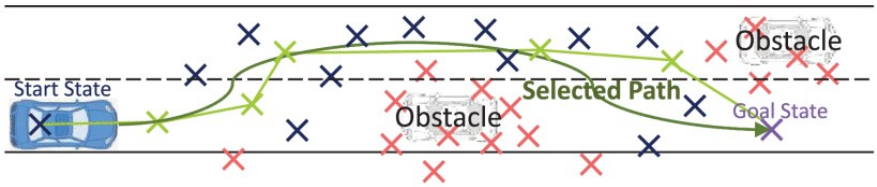
- **Sampling-based decomposition**: the points of the evolution space are picked at random, as in the Probabilistic Roadmap (PRM) algorithm [12] for instance, then connected together to build an obstacle free roadmap (a graph) and finally solved by a pathfinding algorithm that outputs the waypoints to the mobile agent. An example is shown in Fig. 1.3a
- **Connected cells decomposition**: the space is decomposed into cells using geometry and then an occupancy grid or a cells connectivity graph is derived as in Fig. 1.3b. Common methods are the Dynamic Window Approach (DWA) [13], the Voronoi decomposition [14] and Vector Field Histograms (VFH) [15]. However, these algorithms may require large memory and ineffective moving obstacles management.
- **Lattice representation**: the space is decomposed into motion primitives, each one connected to the subsequent one. The state evolutions represented by the lattice are then provided as a reachability graph of maneuvers (see Fig. 1.3c). These algorithms are very common in highway motion planning.

Pathfinding algorithms

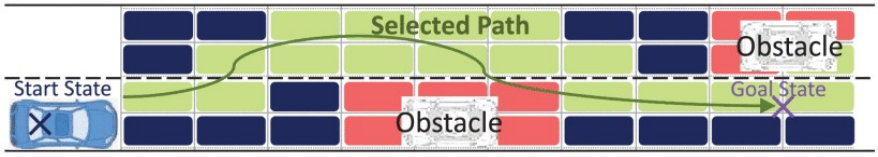
This family is directly derived from the graph theory used to solve combinatorial problems in operational research. Essentially, they find a path in a graph optimizing a certain cost function.

For known environments, the graph earlier generated by a space configuration algorithm is searched by common algorithms such as Dijkstra [16] – which finds the path of minimum total length between two given nodes – and A* [17] – which reduces the computational times of Dijkstra by exploring the fewest number of nodes. Actually, there exist several further versions of these algorithms: Anytime Weighted A* (AWA*) , D*, etc.

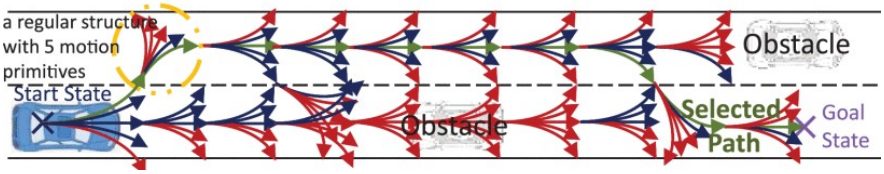
For known environments, the most common solutions are based on the Rapidly-exploring random trees (RRT) [18] algorithm: similarly to PRM, the points of the configuration space are picked up randomly, then they are connected to build a graph according to a "nearest" criterion. Even then, there are different versions: a popular improvement is RRT* [19], which, unlike simple RRT, is guaranteed to converge to a sub-optimal solution.



(a) Sampling-based decomposition.

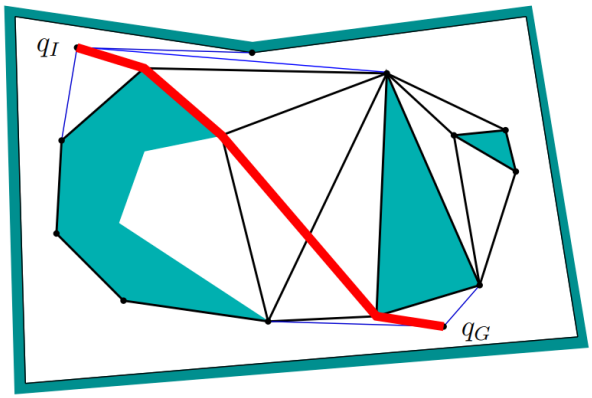


(b) Connected cells decomposition.

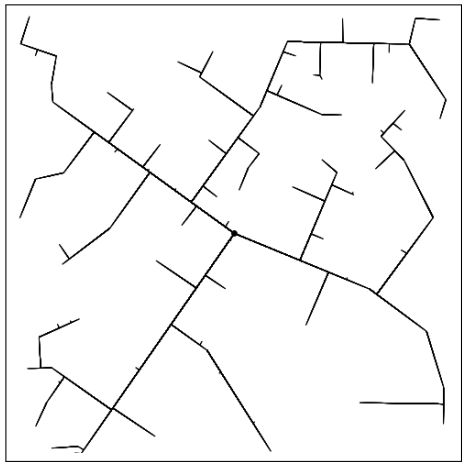


(c) Lattice representation.

Figure 1.3: Space configuration illustrations in a highway autonomous driving scenario [11]. All the possible points / cells / maneuvers are displayed in blue, the discarded ones in red, and the solution in green.



(a) Example of the shortest path computed with Dijkstra algorithm (in red) from q_I to q_G inside the graph represented with black lines connecting the nodes. The blue regions are obstacles and external contours.



(b) The black lines represent the paths discovered by the RRT algorithm after 45 iterations.

Figure 1.4: Pathfinding algorithms, illustrative examples from [10]

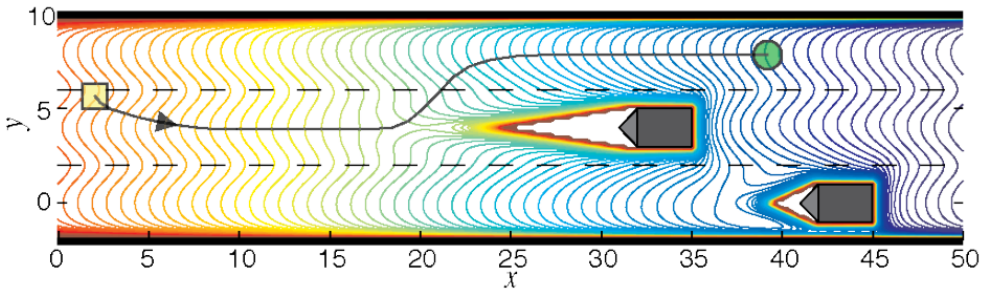
Attractive and Repulsive Forces

These types of algorithms are biomimetic-inspired: the evolution space is represented with attractive forces for desired motions (e.g. legal speed) and repulsive forces for obstacles (e.g. road borders, lane markings, obstacles). Therefore, they are typically used to the purpose of reactive motion deformation.

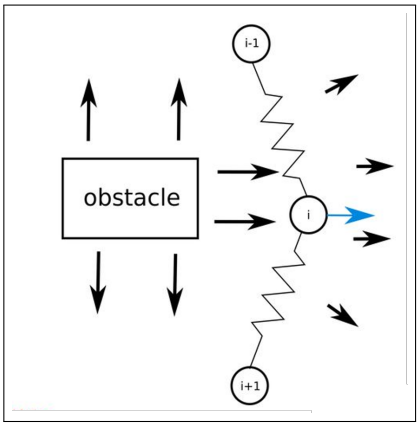
Many of these algorithms are based on the Artificial Potential Field (APF)

concept [20], where the robot goal position is associated to attractive forces, while all the obstacles to repulsive ones. Thus, the robot's configuration is treated as a point in a potential field that guides it to the goal, and the output of the algorithm is a path. This path can be further smoothed by adding a Velocity Vector Field (VVF) as in [21].

Another common approach is the elastic band [22]. It is a collision-free path generated by modelling the environment as a spring-mass system considering N number of nodes on which the potential forces act (Fig. 1.5b). The initial shape of the elastic is the free path generated by a (global) planner. Subjected to artificial forces (potential field), the elastic band deforms in real-time to a short and smooth path that maintains clearance from the obstacles.



(a) Artificial Potential Field generated by two cars on a highway. The field lines ranging from blue (lower potential) to red (higher potential) shape the path from the initial position (yellow square) to the goal (green circle). [23]



(b) Elastic band example: here three nodes of the path (circles) are modelled as a spring-mass system. The repulsive field generated by the obstacle deforms the path, which would be straight otherwise.

Figure 1.5: Attractive and Repulsive Forces algorithms illustrative examples.

Parametric and semi-parametric curves

They are curve-based geometric methods, built as a succession of simple and predefined curves, so they can easily take into account the kinematic constraints of the vehicle. There are two main categories: point-free and point-based curves.

However, these algorithms are more suitable for highway autonomous driving, rather than for common mobile robotics applications.

Artificial Intelligence

AI algorithms for motion planning have met a wide spread in recent years, since they are adaptive, flexible and reactive to the environment. They are mostly employed as solve-algorithms for predictive planning, but their versatility ensures an applicability as set-algorithms as well. Four families can be distinguished:

- **Logic approaches** are adopted also as set-algorithms. They generate or select a set of time/space states or actions on the base of an a-priori knowledge and an inference engine.The most used inference engine is *rule-based* reasoning, adopted by Nilson et al. [24] to perform lane change maneuvers on highways. The engine is populated of easily implementable logic conditions that clearly identify the cause and effects, however, if the model becomes too complex, cyclic reasoning and the exhaustive enumeration of rules decisively impact the computation time. Therefore, they are mostly suitable in constrained and predictable environments. Sample approaches: decision trees, FSM, Bayesian networks [25].

- **Heuristic algorithms** are experience-based and aim at an approximate solution, resulting in a fast and efficient solution to the navigation problem. They usually return a set of actions and require low computational time. Usually, these algorithms involve modelling of the agent as endowed with a rational and social behavior, accepting observations from the environment and implementing a multipolicy decisionmaking.Thus, they are particularly useful in uncertain environment, indeed, Mehta et al. [26] used this approach for making a small robot in a social environment - similarly to the aim of this thesis work. Additionally, decision making can also be supported by learning methods, such as Support Vector Machines (SVM) and Evolutionary methods [27].

- **Approximate reasoning** mimics human reasoning. With respect to the logic approach, here the knowledge base is non-Boolean and there is a learning approach to classify new knowledge to adapt to future situations. Fuzzy logic belongs to this cathegory. Fuzzy systems have rules based on boolean compromises, so the permissiveness of the designed rules guarantees adaptability to uncertain data. Fuzzy controllers turn out to be good solutions for obstacle avoidance [28]. On the other hand, their main drawbacks are their lack of traceability and the absence of a systematic design methodology.

Besides this, there are Artificial Neural Networks (ANN) [29], as well as Convolutional Neural Networks (CNN) and Reinforcement learning. Their main advantage is their ability to learn by training on multidimensional data. Nevertheless, their main drawbacks for autonomous driving are the absence of a causal explanation to a solution and the large amount of data required for the learning process - less if unsupervised learning based.

- **Human-like methods** propose high-level models that mirror human processes for solve-algorithms. They are potentially learning and cognitive procedures. They are able to handle complex situations on the base of some established rules, so they are useful for both predictive and reactive planning. However, they are difficult to model and to use accurately. Examples are Risk estimators [30], taxonomic models [31].

Numerical optimization

The optimization problem for motion planning is defined as a solve-algorithm based on logic and heuristic approaches. The optimization is usually expressed as the minimization of a cost function of some state variables under a set of constraints, and avoiding the combinatorial explosion of these problems is a competitive research topic. Numerical optimization is widely used in motion planning, either to decrease the solving time of a graph's exploration, or to exploit the mathematical properties of the problem, in order to find a predictive solution in a restrictive space for instance. Examples: Linear Programming (LP) [32], Quadratic Programming [33], Model Predictive Control (MPC) [34], Dynamic Programming (DP) [35].

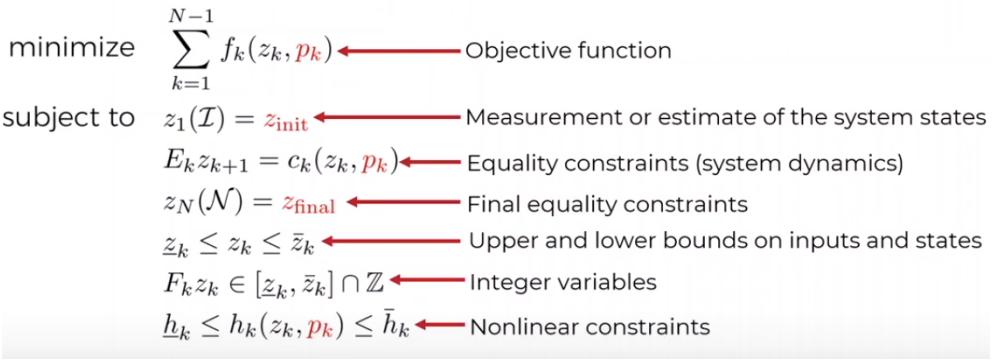


Figure 1.6: Example formulation of a constrained optimization problem.

As presented above, there are multiple approaches to the problem of autonomous navigation, but let's focus on the specific application case of this thesis work. A single robot in LINKS is expected to navigate in a partially known environment: the static map of the building, where the offices are located, shall

be provided to the mobile agent, then, dynamic obstacles - such as people - and unforeseen small static obstacles - such as additional tables, chairs or plant pots - shall be handled separately. Moreover, the mobile agent will be equipped with only LIDAR sensor, so high-level visual information about the environment are not available. Basically, this allows to discard too complex solutions like the AI-based types, avoiding spending much time in data collection, which would be necessary otherwise, since the project is at its first stage. On the other hand, the mathematical formulation of a tailored optimization problem would not be convenient to integrate it in the existing navigation solutions provided within the ROS2 framewrok. Therefore, this thesis will be dealing with space configuration algorithms, such as DWA, together with traditional pathfinding algorithms, and attractive and repulsive forces approach. More details about this will be provided in Chapter 3, after having introduced the Navigation Stack in ROS2.

Chapter 2

ROS2 framework

The aim of this chapter is to describe the main features of ROS2, the upgraded new version of ROS (Robotic Operating system), which has been the virtual environment adopted along the course of this thesis work both for algorithm implementation, for simulations and for deployment on the physical robot. First, some common ROS concepts are explained, so to make the reader feel more comfortable in reading. Then, there is a focus on the specific novelties of ROS2 with respect to ROS, which makes the former finally suitable for real-time embedded applications. Section 2.3 illustrates the ROS2 Navigation Stack (originated from ROS Navigation Stack), which is the ROS software stack for autonomous navigation. Finally, few words are spent to describe TurtleBot3, the chosen hardware platform to test the solution developed in this work.

2.1 ROS concepts

ROS is an open-source, meta-operating system for building robotic applications. It provides the typical services of an operating system (OS), including hardware abstraction, low-level device control, message-passing between processes, and package management. It also provides tools and libraries for writing, building and running code across multiple computers [36]. Since it is not an actual OS, it has to be installed on an existing one, such as Ubuntu, one of Linux distributions. Following is a list of basic ROS terminology and architectural components:

- **Nodes**. ROS nodes are the basic units of a ROS application. They are single-purpose, executable programs/processes.
- **Messages**. They are the mean through which nodes send data to each other; they are declared with a specific type, either standard (Boolean, integer,

float, etc.), or application-specific (Twist, Pose, etc.) for more complex nested messages.

- **Topics**. Topics are the channels though which nodes exchange messages. The communication through topics occurs with a *Publish/Subscribe* logic: a publisher node publishes a message over a topic and all the nodes subscribed to that topic receive that message.

- **Services**. Unlike topics, services realize a *Request/Response* synchronous communication among nodes: a service server node responds only when there is a request from a service client node, that can send requests as well as receive responses. Once request and response of the service are completed, the connection between two nodes is lost.

- **Actions**. Similar to services, action clients send a request to an action server in order to achieve some goal and will get a result. Unlike services, while the action is being performed, an action server sends progress feedback to the client. Therefore, actions are used when a response may take a significant length of time.

ROS software is organized on a **package basis** and each node-to-node connection is managed by a particular server node called *Master* (see Fig. 2.1). A package contains nodes, configuration files, libraries, and so on. On top, there are metapackages, which are sets of packages having a common purpose, such as the Navigation Stack mentioned above.

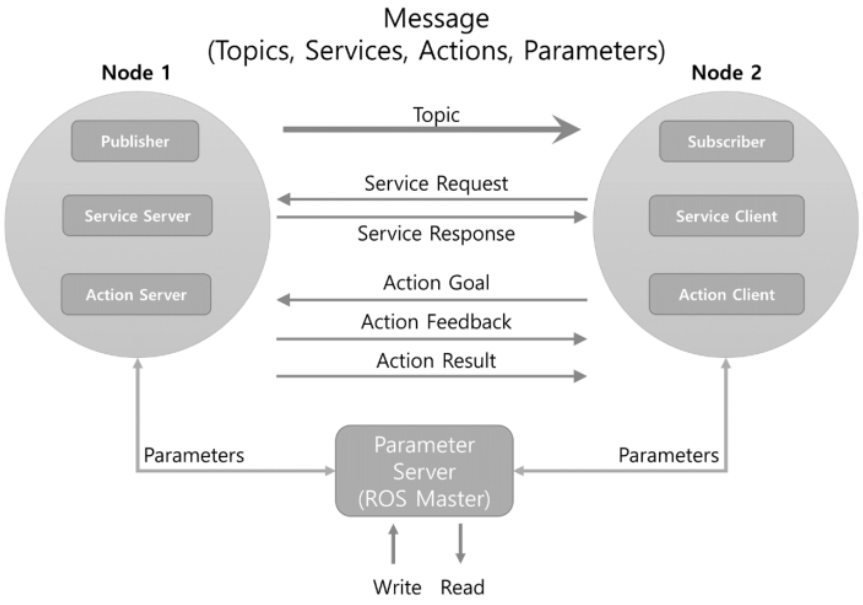


Figure 2.1: ROS communication scheme.

2.2 ROS2 main innovations

Even though ROS usage has increased a lot since its first release, it has some architectural limitations that prevent it from being the basis of products for the market. Actually, ROS requires significant resources (memory, network bandwidth, CPU) and cannot guarantee fault-tolerance, deadlines or process synchronization but, above all, it does not satisfy real-time run requirements [37]. The first ROS2 distribution was released in 2017 with the following design goals, unreachable by ROS: support teams of multiple robots, small embedded platforms, real-time control, non-ideal networks, multi-platform support (Linux, Windows, RTOS) [38].
To reach these goals, ROS2 involves a structural change and adopts new technologies such as the *Data Distribution Service* (DDS) [39]. Figure 2.2 shows the architecture of ROS2 compared to ROS. The main differences are the following:

1) ROS mainly supports Linux-based operating system. ROS2 provides more portability of deployment on underlying operating systems, such as Linux, Windows, Mac, and RTOS.

2) ROS data transport protocol uses TCPROS/UDPROS, and communication is controlled by the Master node. Communication in ROS2 is based on DDS standard, enhancing fault tolerance capabilities. QoS (Quality-of-Service) of DDS gives flexible parameters settings to control the reliability of communication. Thanks to the DDS standard, each topic in ROS2 acquires the additional capability of storing historical message data.

3) Intra-process in ROS2 provides a more optimized transmission mechanism.

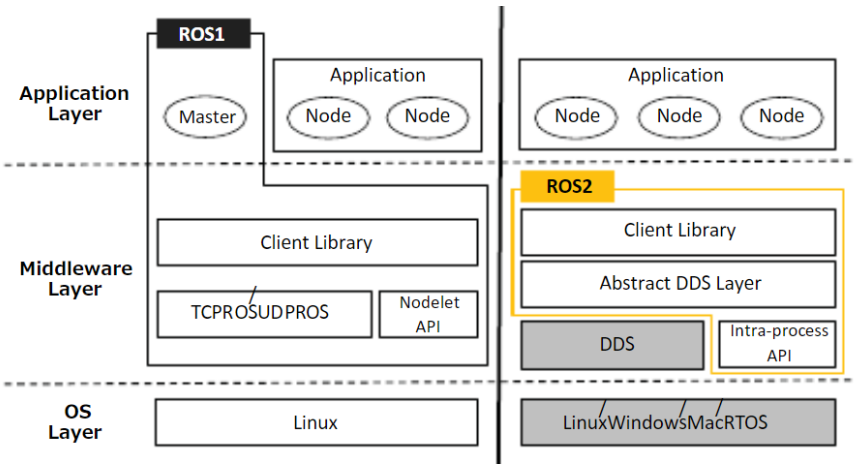


Figure 2.2: ROS/ROS2 architecture.

A completely new component of ROS2, not present in ROS, are ***Lifecycle*** (or ***Managed***) ***nodes*** [40]. They are nodes that contain state machine transitions for the bringup and the teardown of ROS2 servers (see Fig. 2.3). Each of them has four primary states: *Unconfigured*, *Inactive*, *Active* and *Finalized*; transitions out of a primary state are triggered by an external supervisory process, such as a common ROS2 node. This is made possible by a well known interface, which exposes a managed node to the ROS ecosystem. Using ROS2's managed/lifecycle nodes feature allows to ensure that all required nodes have been instantiated correctly at the system startup, before they begin their execution. Lifecycle nodes also allows on-line restarting or replacing of nodes. Therefore, they are widely used in the Navigation stack to realize safer navigation applications.

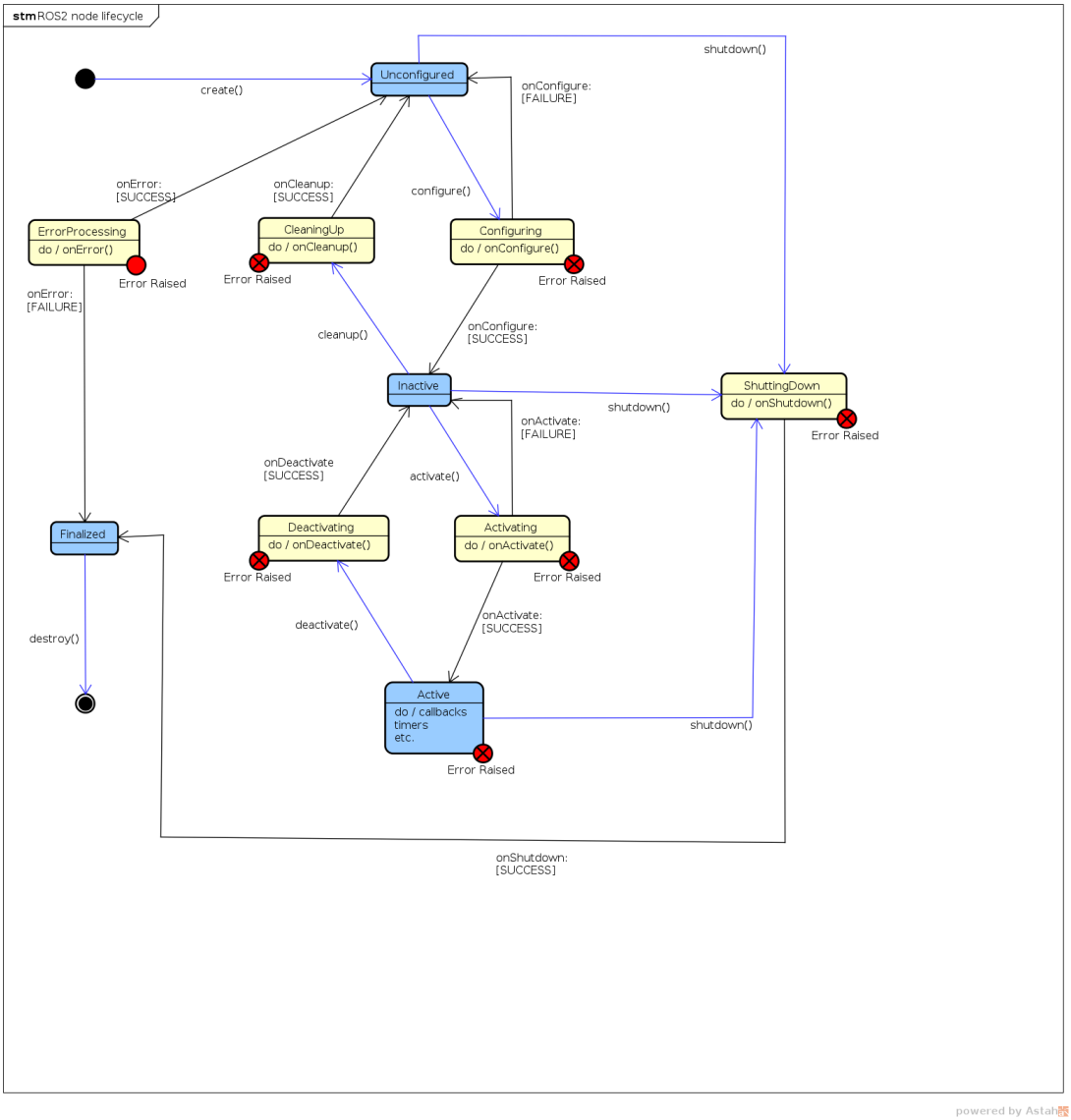


Figure 2.3: State machine at the base of a lifecycle node.

2.3 ROS2 Navigation Stack

Navigation in ROS1 (Nav1) is based on `move_base` [41], an unconfigurable single process state machine that plans and executes the path in order to reach the goal position on the map. It relies on a `global_planner` node that explicates the functionality of a *Route Planner* (see Section 1.2) on the base of the `global_costmap` environmental representation, and on a `local_planner` working on a `local_costmap`, which explicates the functionalities of a motion planner. However, `move_base` allows only one single global and local trajectory planner at a time. In principle, this allows full autonomous navigation, but actually, it supports effectively only differential and omnidirectional robots.

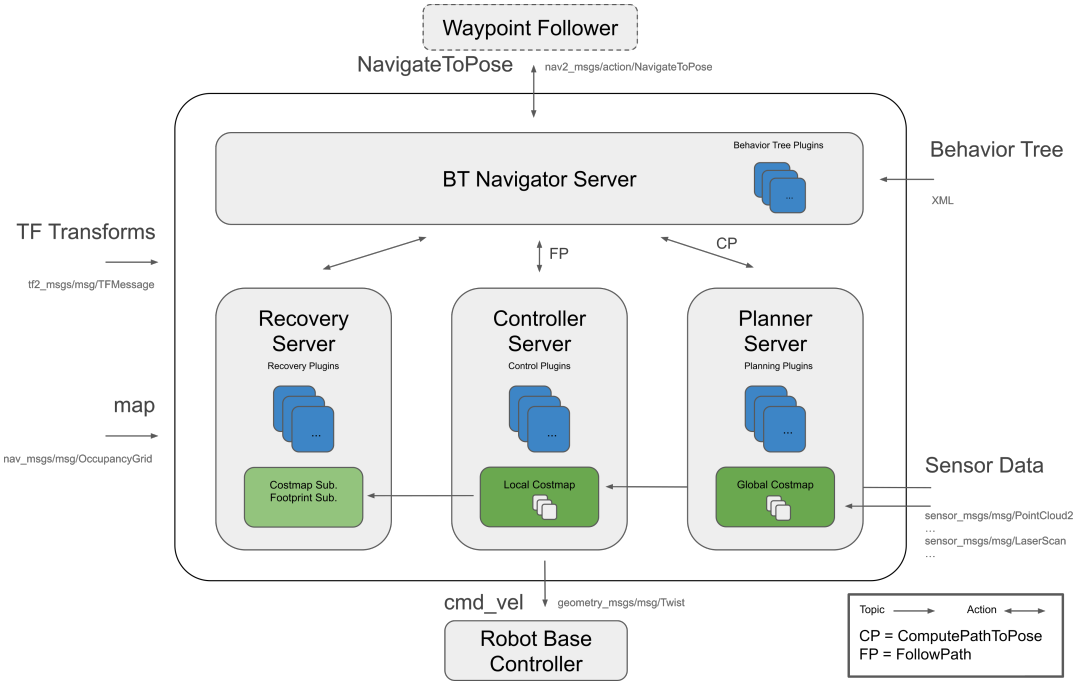


Figure 2.4: Navigation 2 architecture.

Navigation 2 on ROS2 (Nav2) is the new implementation of the navigation metapackage already available on ROS1, but it has a completely new architecture, shown in figure 2.4:

- **Unchanged from Nav1.** The TF transformations among coordinate frames [42] (`map` → `odom` → `base_link` → `[sensor frames]`) are provided as `tfMessages` obtained from sensors, odometry and the `amcl` node, which is an Adaptive Monte-Carlo Localization [43] technique based on a particle filter for the localization in a static map. The static initial map is provided by the `map_server` and finally the output is a `cmd_vel` message sent to the specific robot controller.

- **New in Nav2.** The true novelty of Nav2 is that it comes with a modular and reconfigurable (at run-time) core, consisting in a behavior tree (BT) navigator and task-specific asynchronous servers. They have the following features:
  - Each server has an action server interface to handle goals, provide feedback and return results to client.
  - Each server has a map of plugins, so it can support N plugins; thus different algorithms can be used in different unique contexts.
  - They can leverage multi-core processors using multi-processing frameworks, being able to compute more complex navigation tasks.

  Details about these concepts are provided in the following paragraphs, since they are crucial in the implementation phase of this thesis project.

BT Navigator. The *Behavior Tree Navigator* uses a behavior tree to «orchestrate» the navigation tasks, loading it from an `.xml` file. BTs were first developed in the computer game industry as a better alternative to Finite State Machines (FSMs), used in the control structures of Non-Player Characters (NPCs). Indeed, BTs ensure better reactivity and modularity with respect to FSMs (a detailed theory about BTs can be found in this book [44]). Therefore, it is possible to create custom robot behaviors expanding, pruning or adding sub-trees to the branches of the BT, as it is done in this work. Nav2 implements the BT Navigator upon the `BehaviorTree.CPP` library [45], which simplifies the building of complex navigation behaviors starting from basic primitive ones.

Navigation servers. *Planner*, *Controller* and *Recovery* action servers are used to host a map of algorithm plugins to complete various tasks. They also host the environmental representation used by the algorithm plugins to compute their outputs.

- In general, Planner plugins are for computing a valid, and potentially optimal, path from the current pose to a goal pose or computing a complete coverage path.
- Controller plugins replace the `local_planner` of Nav1: they compute a feasible control effort to follow the global plan, based on a local environmental representation.
- The Recovery behaviors are plugins triggered by the BT when a navigation failure occurs.

A last important notice has to be made on the Controller plugins. Nav2 comes essentially with two default plugins for local motion planning: the **DWB** controller [46] (a modified version of the previous implementation of DWA algorithm in Nav1) and the **TEB** (Time Elastic Band) controller [47], which uses Timed-elastic-bands for time-optimal point-to-point nonlinear model predictive control. These plugins arise from the need to make Nav2 capable of managing dynamic obstacles efficiently. However, this goal is still an open challenge for the ROS community. Therefore, this work tries to integrate a dynamic obstacles tracking into the costmap environmental representation together with these existing controller plugins. This will be the topic of the next chapters.

To sum up, some of the main differences between Nav1 and Nav2 are collected in Table 2.1

Table 2.1: Summing up on Nav2 advantages with respect to Nav1.

Navigation 1	Navigation 2
Based on `move_base`: **unconfigurable** single process state machine.	Relays on a super **configurable** behavior tree. Recovery, planner and controller are **independent servers**: they can be removed, swapped out, etc.
Allows **only one** single global and local trajectory planner at a time.	**Multiple** local trajectory and path planners.
Costmap 2D environmental model (occupancy grid).	**Positioning system agnostic**: either 2D/3D or visual SLAM can be utilized.
It supports effectively **only** differential and omnidirectional robots	Works with **all major robot types**: differential, omnidirectional, Ackermann, circular, arbitrary shape.
Dynamic Window Approach (**DWA**) trajectory planner.	Implements an improved version of DWA, which is the **DWB** planner plugin.

2.3.1 Environmental representation and Costmap2D

As mentioned above, the robot relies on a costmap representation of the environment, on top of which planner and controller servers compute a preferred route through obstacles minimizing a cost function. In Nav2, this is made possible by the dedicated package `nav2_costmap_2d`, which subscribes to the sensor data and builds a 2D or 3D space representation in the form of an occupancy grid: based on the input size and resolution parameters, each cell can assume an integer value value between 0 ans 255 in compliance with the sensor data. However,

the underlying used structure actually can represent only three values: *FREE*, *UNKNOWN* and *OCCUPIED*, which are set to 0, 255 and 254 respectively by default.

The huge potential of the costmap representation in ROS2 recides in the adoption of the costmap layers method [48]: unlike traditional monolithic costmaps, where all the data are stored in a singular grid of values, in costmap layers approach, each layer tracks one type of obstacle or constraint, and then modifies a master costmap which is used for the path planning (see Fig. 2.5). This approach has much better performances in dynamic, people-filled environments, with respect to the monolithic costmap approach. In particular, the benefits are:

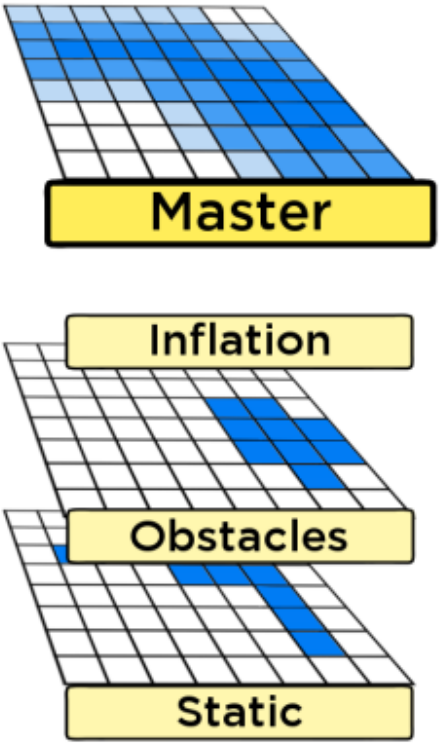


Figure 2.5: Example of application of costmap layers: a static and an obstacle layer are inflated and combined into the master costmap.

- Clearer Update Step: keeping information separated in different layers makes the update step more clearly delineated and helps avoiding conflicts between sensor data and costmap values already stored.
- Dynamic Update Areas: the layered costmap only updates the region of the map that the individual layers reckon necessary. Only values within a predefined bounding box are updated, giving the costmap extra stability and potentially a greater efficiency, since smaller amounts of map can be updated at separate steps.

- Ordered Update Process: costmap layers are updated with an explicit ordering, allowing the user to specify custom laws to combine values of different layers in order to suit best their application use case. This leads to and incredible flexibility and modularity, expanding the costmap's semantic possibilities.

ROS2 manages the costmaps through the `costmap_2d::Costmap2DROS` object which provides a user-friendly interface to the users. It contains a `costmap_2d::LayeredCostmap` which is used to keep track of each of the layers. Each layer is instantiated as an individual plugin in the parameter file used to launch Nav2. In this way `local_costmap` and `global_costmap` are differentiated not only on the base of size and update frequency, but also with respect to the layers they are made up of. Base layers in Nav2 are essentially three:

- Static Layer: stores the costs relative the static map provided at launch time.

- Obstacle Layer: continuously mark and clear cells according to sensor data.

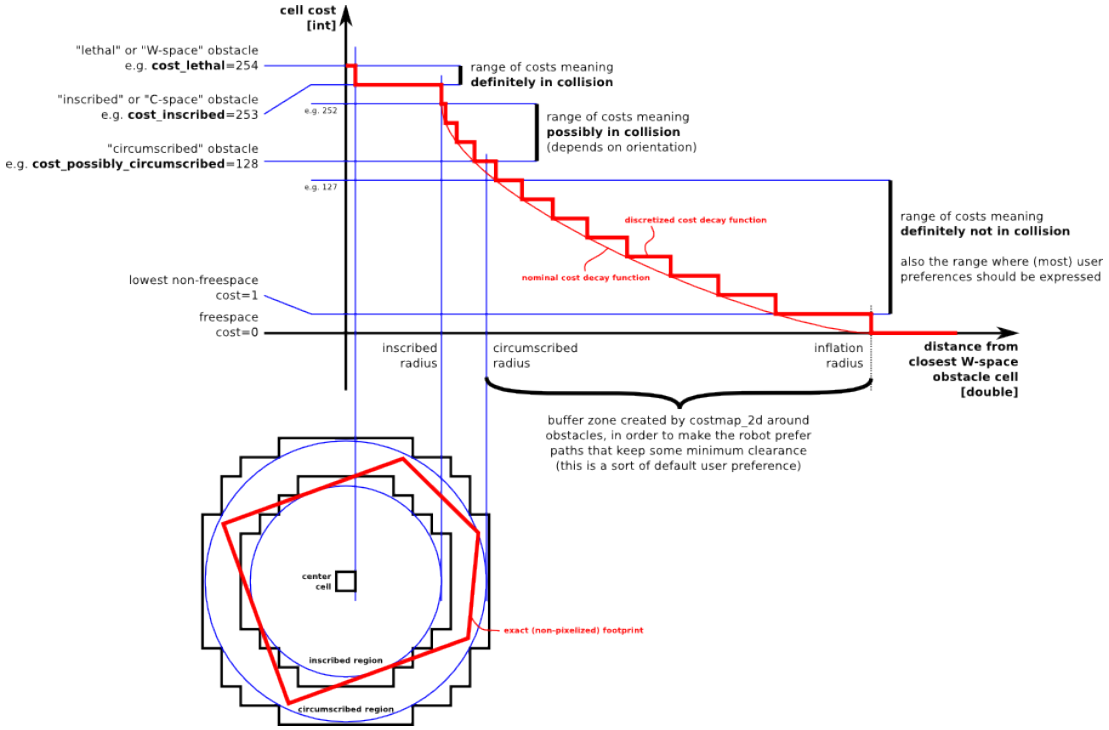


Figure 2.6: Inflation Layer costs assignment.

- Inflation Layer: propagates cost values out from occupied cells that decrease with distance, in order to provide a safety margin for the robot navigation. In particular, five cost levels are defined, as shown in figure 2.6 [49]:

- "Lethal" cost means that there is an actual (workspace) obstacle in a cell.
- "Inscribed" cost means that a cell is less than the robot's inscribed radius away from an actual obstacle. So the robot is in collision if its center is in a cell greater or equal cost than the inscribed cost.
- "Possibly circumscribed" cost is similar to "inscribed", but using the robot's circumscribed radius as cutoff distance. Thus, if the robot center lies in a cell at or above this value, only the robot orientation can tell whether it collides with an obstacle or not.
- "Freespace" cost is assumed to be zero, meaning that there is nothing that should prevent the robot from going there.
- "Unknown" cost means there is no information about a given cell.
- All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

2.4 TurtleBot3

TurtleBot is a low-cost, personal robot kit with open-source software, created in November 2010. The specific model used in this project is TurtleBot3, developed in 2017 with features to supplement the lacking functions of its predecessors: TurtleBot1 and TurtleBot2 (2012).

TurtleBot3 is small, affordable, programmable and ROS-based, thus it is widely used for education, research, hobby, and product prototyping. The TurtleBot3 can be customized into various ways, providing an easy way to assemble its mechanical parts and use optional parts such as a computer and sensor. In addition, TurtleBot3 is evolved with cost-effective and small-sized Single-Board Computer that is suitable for robust embedded system, 360 degree distance sensor and 3D printing technology.

The TurtleBot kit consists of a mobile base, 2D/3D distance sensor, laptop computer or SBC(Single Board Computer), and the TurtleBot mounting hardware kit. In addition to the TurtleBot kit, users can download the TurtleBot SDK from the ROS wiki. TurtleBot is designed to be easy to buy, build, and assemblable, using off the shelf consumer products and parts that easily can be created from standard materials. It comes in two models: *Burger* and *Waffle*; in this thesis work, Turtlebot3 Burger has been used – both in simulation and in LINKS laboratory. The TurtleBot's core technology is SLAM (Simultaneous Localization

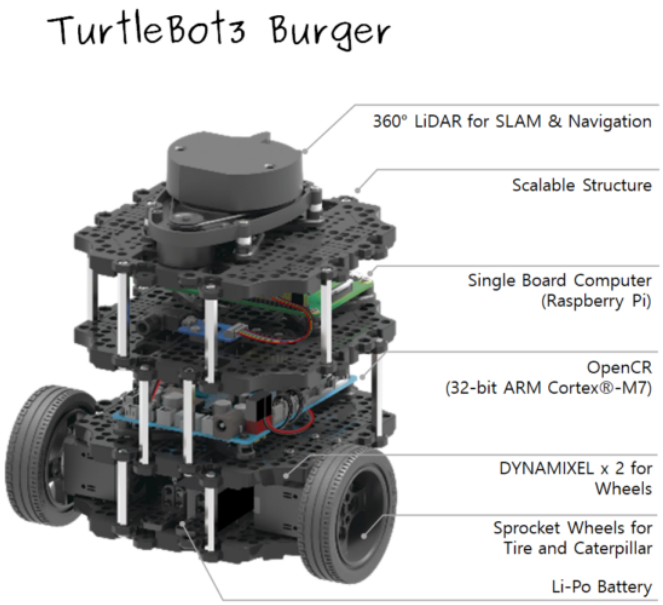


Figure 2.7: TurtleBot3 Burger platform.

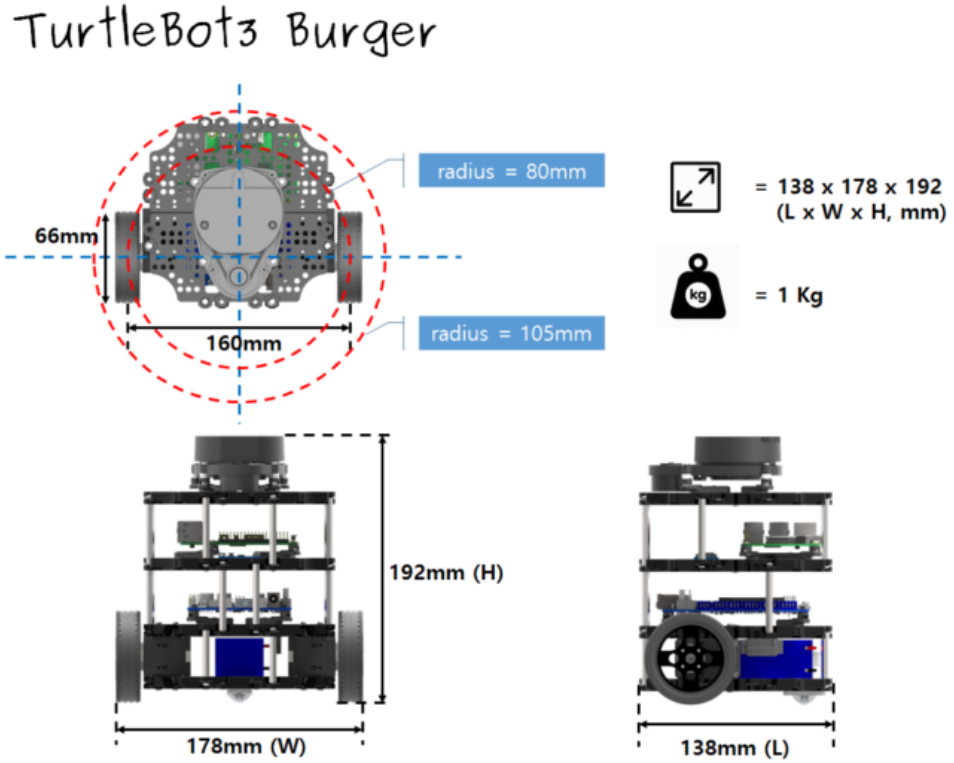


Figure 2.8: TurtleBot3 Burger dimensions.

and Mapping) and Navigation, making it suitable for home service robots. It can also be controlled remotely from a laptop, a joypad or an Android-based smart phone.

TurtleBot3 Burger is equipped with different sensors: IMU sensors (gyroscope, accelerometer, magnetometer), LiDAR and motors encoders. For driving, it adopts

ROBOTIS smart actuator DYNAMIXEL [50], which is an all-in-one actuator embedding a DC motor, a controller, a driver, sensors, reduction gear and a network (Fig. 2.9). Some other useful specification are listed in table 2.2:

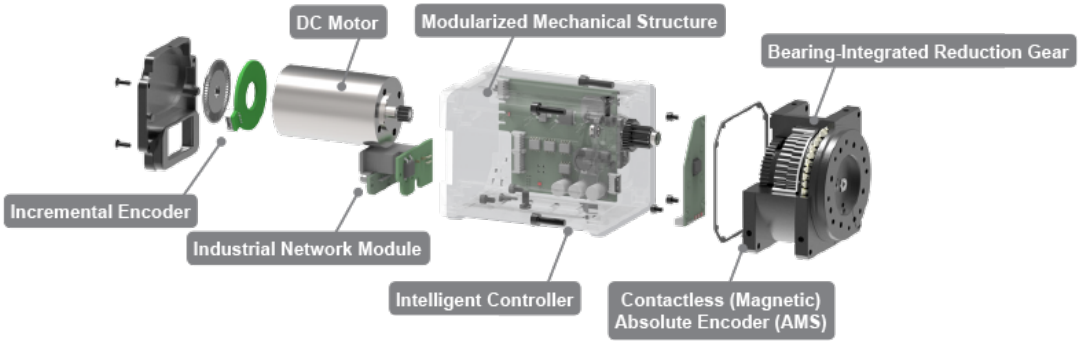


Figure 2.9: TurtleBot3 DYNAMIXEL actuator.

Items	**Specifications**
Maximum translational velocity	0.22 m/s
Maximum payload	15 kg
Size (L x W x H)	138 mm x 178 mm x 192 mm
Weight (+ SBC + Battery + Sensors)	1 kg
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis, Magnetometer 3 Axis
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
SBC (Single Board Computers)	Raspberry Pi
LiDAR Sensor	360° Laser Distance Sensor LDS-01

Table 2.2: Some of the TurtleBot3 Burger specifications.

Concerning the LiDAR sensor, the LDS-01 [1] is the 2D laser scanner mounted by default on the TurtleBot3 Burger. It is capable of sensing 360 degrees, collecting a set of data around the robot that can be used either for SLAM (Simultaneous Localization and Mapping) or navigation. It supports USB interface for an easy installation on a PC. However, LDS-01 has a maximum distance range of 3.5 m. Therefore, in order to obtain an effective comparison between the solution developed in this thesis and the actual available Nav2 capabilities, a LiDAR with

[1] `https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/`

greater maximum distance range and finer resolution has been adopted in place of LDS-01: RPLIDAR A3 [2].

(a) LDS-01 sensor.

(b) RPLIDAR A3 sensor.

Figure 2.10

Table 2.3 collects the most significant differences between the two sensors.

Items	**LDS-01**	**RPLIDAR A3**
Maximum distance range	3.5 m	25 m (light objects) 10 m (dark objects)
Scan rate	5 Hz	15 Hz (adjustable bertween 10 $Hz - 20$ Hz)
Sampling rate	1.8 kHz	16 kHz
Angular Resolution	1°	0.225°

Table 2.3: Comparison of some specifics of LDS-01 and RPLIDAR A3.

The setting of the ROS2 Foxy interface for TurtleBot3 will be illustrated in **Chapter 5**.

2.4.1 RPLIDAR working principle

The RPLIDAR A3 adopts laser triangulation ranging principle.

The core runs clockwise to perform a 360 degree omnidirectional laser range scanning of the surrounding environment and then generate an outline map for the environment. It contains a laser emitter and a camera to receive the reflected beams. In particular, the laser emits an infrared laser signal that is then reflected by the object to be detected (Fig. 2.11). The beam passes through a pinhole lens and hits a CCD[3] camera sensor. By construction, thus, the triangles defined by (b, d_k) and by (b'_k, d') are similar: this means that the distance to the object is

[2] `https://www.slamtec.com/en/Lidar/A3`

[3] charge-coupled device

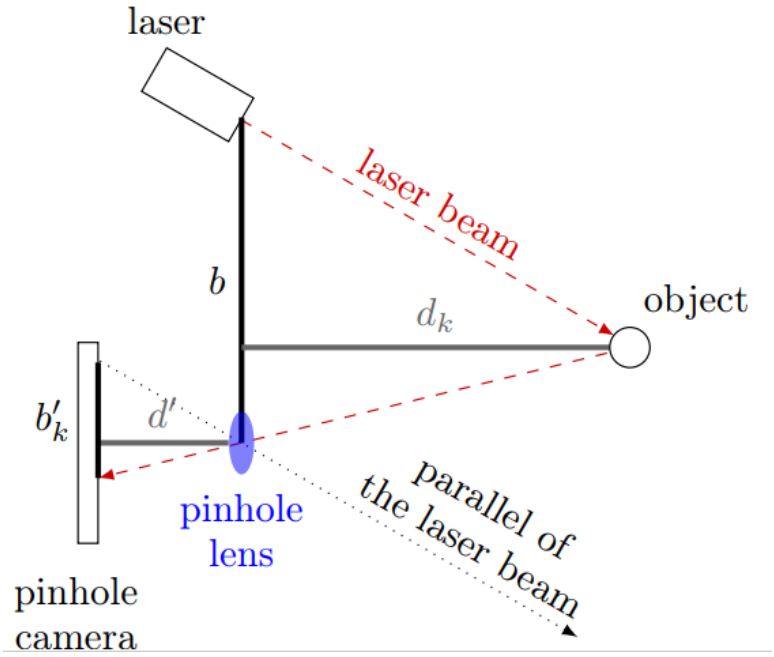
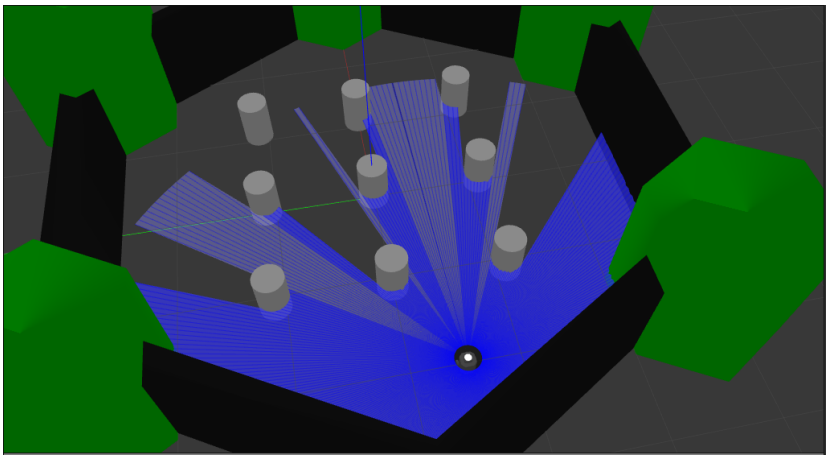


Figure 2.11: Laser triangulation ranging principle - illustrative scheme.

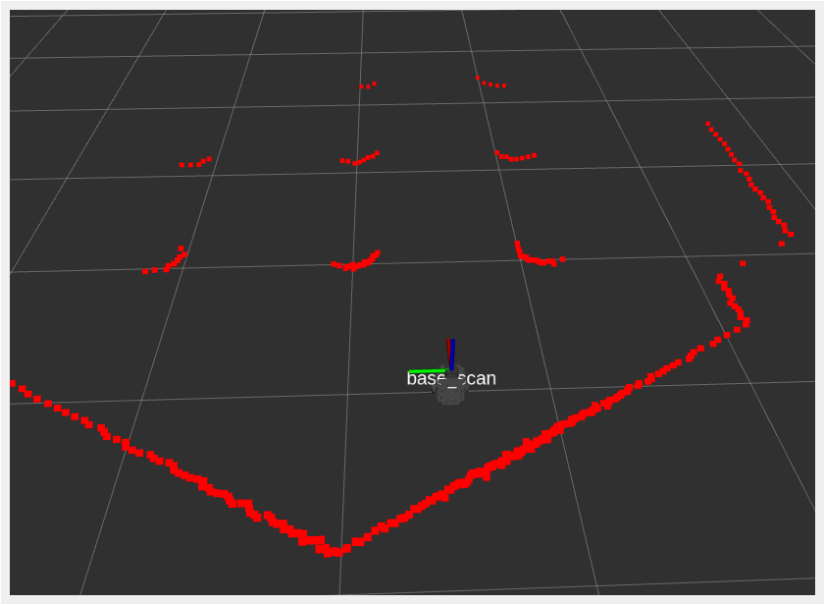
nonlinearly proportional to the angle of the reflected light, and as soon as the camera measures the distance b'_k one can estimate the actual distance d_k using triangles similarities concepts.

In ROS2 the measured distances are shared using the `LaserScan.msg` message[4]. The costmap used in Nav2 subscribes to a specific topic to read the LIDAR data and then marks the costmap cells accordingly to the received distances.

[4]Message definition: `https://github.com/ros2/common_interfaces/blob/master/sensor_msgs/msg/LaserScan.msg`



(a) Gazebo virtual environment.



(b) Rviz

Figure 2.12: Laser beams emitted by LiDAR sensor are displayed in the Gazebo virtual environment and corresponding points are visualized in Rviz.

Chapter 3

Dynamic path planning

Nowadays, given the huge developments of AI in computer vision and enhancement of camera performances for robotics, most of dynamic obstacles handling strategies are based on visual information. However, this reflects in higher hardware costs due to the greater computational demand. LIDAR-based solutions are not common, but they allow a cheaper setup of the robot; the key points are so the LIDAR points filtering, interpretation as obstacles and tracking. In the following sections, more detailed information about local planning strategies implemented in Nav2 are provided. Then, from their comparison arise the reasoning behind the Dynamic Obstacle Layer approach developed in this work.

3.1 LiDAR-based planning solutions

As explained in Chapter 1, the dynamic obstacle handling is faced in motion planning at local level, meaning that in Nav2 dynamic obstacle avoidance should be addressed by the *Controller server*. In fact, Nav2 provides two implementation approaches for local planning as *Controller server* plugins: DWB Controller (`nav2_dwb_controller`[1]) and TEB Controller (`teb_local_planner`[2]).

The DWB Controller is a plugin that implements the Dynamic Window Approach. It works in the domain of translational and rotational velocities. It considers all velocities that the robot, limited by its maximum acceleration, can reach within a time interval between two steering commands. All the velocities that would result in a collision are excluded. Among the remaining velocities, the most promising one is chosen by using a cost function, represented as a grid map, encoding the cost of traversing through each grid cell. It takes as inputs the direction of the goal, the distance to the nearest obstacle and the current velocity

[1] `https://github.com/ros-planning/navigation2/tree/main/nav2_dwb_controller`
[2] `https://github.com/rst-tu-dortmund/teb_local_planner`

of the robot into account [13].

The TEB (Timed Elastic Band) Controller is the evolution of the Elastic Band Algorithm (EBA) implemented in ROS1. Basically, the EBA is the implementation of the elastic band approach cited in paragraph 1.3.2. It uses a full, obstacle free path from a start point to a goal point. This path is split in sections, which are then repositioned under the impact of newly detected obstacles using an artificial force model. A force between the sections holds them together while a repulsive force pushes the path away from obstacles. In order to use this approach, the velocities for the robot have to be chosen by an additional control algorithm. TEB enhances the concept of the EBA: the sections of the path are additionally associated with a timing information. Using this information, a cost function can take both the driving time and the distance to obstacles into account.

At its first stable release, Nav2 was validated through an exemplary experi-

Figure 3.1: *Tiago* robot operated in the "Marathon 2" experiment [51] while navigating the corridors of Technical School of Telecommunications Engineers at the Rey Juan Carlos University.

ment, documented in "The Marathon 2: A Navigation System" [51]. Here, two professional robots are operated in a human-filled environment in a University setting running at near-industrial speeds and implementing the complete Nav2 stack, using the TEB Controller plugin. The authors – the main developers and maintainers of the Nav2 stack – show that the robot can successfully navigate in a crowded environment. This might make think that the TEB Controller can deal with dynamic obstacles as well. In truth, students walking in the university are seen as every other obstacle in the local environment representation. Indeed,

the navigation was successful, but the number of recovery behaviors was high (4.3 recoveries per mile). This is not generally a poor-performance indicator, since it is part of a fault-tolerant and robust system. However, the majority of recovery behaviors executed were due to two cases: low localization confidence – usually when too many people surrounded the robot, preventing enough LiDAR rays to reach the static map – and inability to compute a route to the goal at local level. This second condition often verified when students walked traversing the computed local path, getting too close to the robot and forcing it to a sudden replanning. This triggered the *wait* recovery behavior, which pauses the navigation until the path is clear.

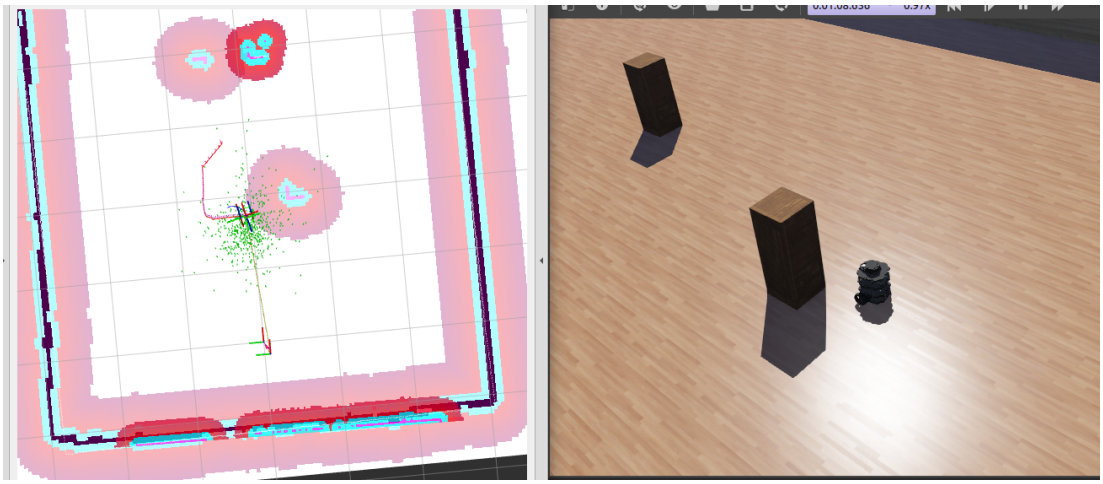


Figure 3.2: A moving obstacle getting to close to the robot during navigation and triggering the *wait* recovery: the TurtleBot restarts its navigation only when the obstacle get away its proximity area.

In conclusion, the authors claim the TEB controller can account for dynamic obstacles in computing velocity commands, however, no explicit obstacle detection is utilized. Therefore, they aim to integrate these explicit detections in the actual implementation of Nav2, further enabling robots to operate more safely around humans and other agents. Indeed, the main maintainer of Nav2 has launched a challenge to the community to develop such solution on the Nav2 documentation website[3], also asking for a 2D LiDAR based dynamic obstacle detection. The method proposed in the following sections tries to accomplish this challenge.

3.2 Methods

Even if TEB algorithm does not explicictely takes into account dynamic obstacles, [52] explains in detail a configuration of the ROS package to do this. However, this

[3] `https://navigation.ros.org/2021summerOfCode/projects/dynamic.html`

setup takes a higher computational time than DWA and simple Elastic-Band (not available in ROS2) as documented in [53]. The method developed in this thesis work is inspired to the `costmap_converter` package provided in [52] – albeit developed in ROS1 – and it is based on the analysis of the LiDAR data, already provided in their costmap representation, for the obstacle detection step.

In principle, a more heuristic approach was evaluated, such as the ones presented in [54] and [55]. Here the dynamic obstacles detection is performed applying some heuristic algorithms directly on the LiDAR pointclouds, clustering the obstacles based on parameters such as cluster radius, distance inbetween points and others. However, this would have resulted in higly tailored solution for the specific use case, requiring a re-tunig of the parameters in case of a change of the environment. On the other hand, the approach developed in this work is articulated in three steps:

1) **Object detection** – Starting from the costmap representation of the environment, dynamic obstacles are identified and separated from static ones, applying image processing algorithms and running average filters.

2) **Object tracking** – Detected dynamic obstacles are tracked and their velocity is estimated applying a Kalman Filter.

3) **Cost assignment** – a developed costmap layer assigns costs around each moving obstacle in the `local_costmap` according to a 2D Gaussian shape with variances proportional to the obstacle velocity and oriented in its moving direction.

The theory behind each step will be illustrated in detail in the following subsections, while the implementation strategy will be presented in Chapter 4.

3.2.1 Object detection

Once the robot is powered up and ready to navigate, its costmap representation of the environment is handled as an image during the object detection step.

From now on, the operation of separating the static obstacles from dynamic ones in the costmap is referenced as *background subtraction*. In principle, this term indicates any technique which allows an image's foreground to be extracted for further processing, such as object recognition. Background subtraction algorithms can be divided into running average background, Gaussian mixture background, kernel density background and eigen-background according to the background models [56]. Running average background subtraction has the lowest

computational complexity, but sometimes selecting the model updating rate may be troubling. If the updating rate is too high, it may cause artificial "tails" to be formed behind the moving objects. In order to overcome the existing problem in the traditional running average background model, the proposed approach consists in the combination of the running average background with the temporal difference method.

The basic recursive equation of a running average filter is the following:

$$B_{t+1}(x, y) = (1 - \alpha)B_t(x, y) + \alpha F_t(x, y) \tag{3.1}$$

Where, (x, y) denote the coordinates of a single pixel in the image, B_t is the background image at the time t , F_t is the current input image at time t. The updating rate α represents the speed of new changes in the scene updated to the background frame.

In the context of this work, the foreground detection is obtained by applying two running average filters: a "slow" and a "fast" filter, applied to each cell (pixel) of the costmap image:

$$P_f(t + 1) = (1 - \alpha_f)P_f(t) + \alpha_f C(t) \tag{3.2}$$

$$P_s(t + 1) = (1 - \alpha_s)P_s(t) + \alpha_s C(t) \tag{3.3}$$

$P_f(t)$ and $P_s(t)$ represent the output of the fast and the slow running average filters at time t, respectively. The gains α_f and α_s define the effect of the current costmap C(t) on P. The two filters together allow to compute the temporal difference between two consecutive image frames in order to isolate the dynamic obstacles (foreground mask) from the background, without affecting the latter with to much noise. Therefore, the two filter rates are chosen as follows:

$$0 \leq \alpha_s < \alpha_f \leq 1$$

However, large objects form blocks of cells in the local costmap, therefore, the equations 3.2 and 3.3 are extended by a term that captures the running average filter of the 8 cells nearest neighbors (NN). β denotes the ratio between the contribution of the central cell filter and the effect of the neighboring cells to $P_f(t)$ and $P_s(t)$. So, the complete equations defining the slow and fast filters become:

$$P_f(t+1) = \beta[(1-\alpha_f)P_f(t) + \alpha_f C(t)] + \frac{1-\beta}{8}\sum_{i \in NN} P_{f,i}(t) \tag{3.4}$$

$$P_s(t+1) = \beta[(1-\alpha_s)P_s(t) + \alpha_s C(t)] + \frac{1-\beta}{8}\sum_{i \in NN} P_{s,i}(t) \tag{3.5}$$

In order to identify if a cell is occupied by a moving obstacle, it undergoes two thresholding steps after computing the filter output values:

1) Fast filter activation: the fast filter has to exhibit an activation that exceeds a threshold c_1:

$$P_f(t) > c_1 \tag{3.6}$$

2) The difference between the fast and the slow filter has to exceed a threshold c_2 in order to eliminate quasi-static obstacles with low frequency noise:

$$P_f(t) - P_s(t) > c_2 \tag{3.7}$$

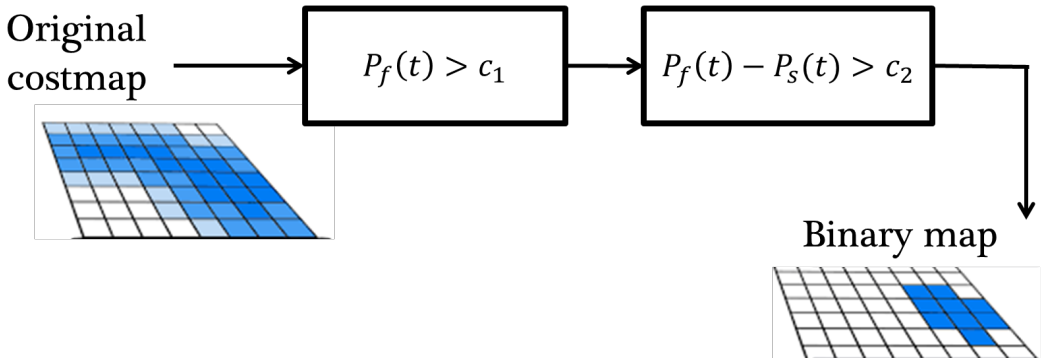


Figure 3.3: Dynamic objects filtering steps – Consider the navigation costmap at a certain instant, where a moving obstacle is traversing a region with some noise.

The output of this thresholding operations is a binary map (Fig. 3.3): if a cell is compliant to both thresholding operations, it is marked with one, while free space and static obstacles are labeled with zeros. Thus, a binary map marking all the dynamic obstacles is obtained.

The next step is to give a meaning to the zeros and ones of the binary map so to provide a suitable input to the object tracking step. Indeed, in order to estimate obstacles velocity, we need first to identify and represent them as obstacles (clustering) and compute the coordinates of a representative point, the obstacle centroid in this case. To this purpose, an heuristic blob detection and clustering

algorithm, provided by the OpenCV library, has been implemented through the `SimpleBlobDetector` class [57]. The algorithm receives as input an image and it is controlled by parameters. It works in the following steps:

1) **Thresholding** : the source image is thresholded with with different thresholds starting at `minThreshold` and converted to many binary images. These thresholds are incremented by `thresholdStep` until `maxThreshold`. So the first threshold is `minThreshold`, the second is `minThreshold` + `thresholdStep`, the third is `minThreshold` + (2 × `thresholdStep`), and so on.

2) **Grouping** : In each binary image, connected dark pixels are grouped together to form binary blobs.

3) **Merging** : Blobs centers are computed, and blobs located closer than `minDistBetweenBlobs` are merged.

4) **Center & Radius Calculation** : Centers and radii of the new merged blobs are computed and finally returned.

In the specific implementation used in this work, the Thresholding step is skipped, since the input image is already a binary image. The parameters for `SimpleBlobDetector` can be set to filter the type of blobs based on different criteria:

- **By color** : setting `filterByColor` = 1, you can specify `blobColor` = 0 to select darker blobs, and `blobColor` = 255 for lighter blobs. In between values allow to filter out blobs with a particular grey tone.

- **By Size** : filter the blobs based on size by setting the parameters `filterByArea` = 1, and appropriate values for `minArea` and `maxArea`. For instance, setting `minArea` = 100 will filter out all the blobs that have less then 100 pixels.

- **By Circularity** : This just measures how close to a circle the blob is. For instance, a regular hexagon has higher circularity than a square. To filter by circularity, you can need to set `filterByCircularity` = 1. Then set appropriate values for `minCircularity` and `maxCircularity`. Circularity is defined as

$$\frac{4 * \pi * Area}{perimeter^2}$$

That is a circle has a circularity of 1, circularity of a square is 0.785, and so on.

- **By Convexity** : Convexity is defined as the (Area of the Blob / Area of it's convex hull). Recalling that the Convex Hull of a shape is the tightest convex shape that completely encloses it. To filter by convexity, you need to set `filterByConvexity` = 1, followed by setting $0 \leq$ `minConvexity` $\leq$ 1and `maxConvexity` (≤ 1)

- **By Inertia Ratio** : This measures how elongated a shape is. For a circle this value is 1, between 0 and 1 for an ellipse, and for a line it is 0. To filter by inertia ratio, you need to set `filterByInertia` = 1, and set $0 \leq$ `minInertiaRatio` ≤ 1 and `maxInertiaRatio` (≤ 1) appropriately.

In the specific implementation used in this work, blobs representing the dynamic obstacles are filtered out based on size (area), circularity and inertia ratio. All the parameters used for the blob detection have been set based on intuition and validated in the simulation environment developed in this thesis, however, they could be optimized in future, performing some tuning in different scenarios, but without compromising too much the versatility of the whole dynamic obstacle detection pipeline.

The blob detection steps provides as output obstacles represented by their centroid – the coordinate of the cell (pixel) in the weighted center of the blob – and contours – a list of the cells which define the blob contours.

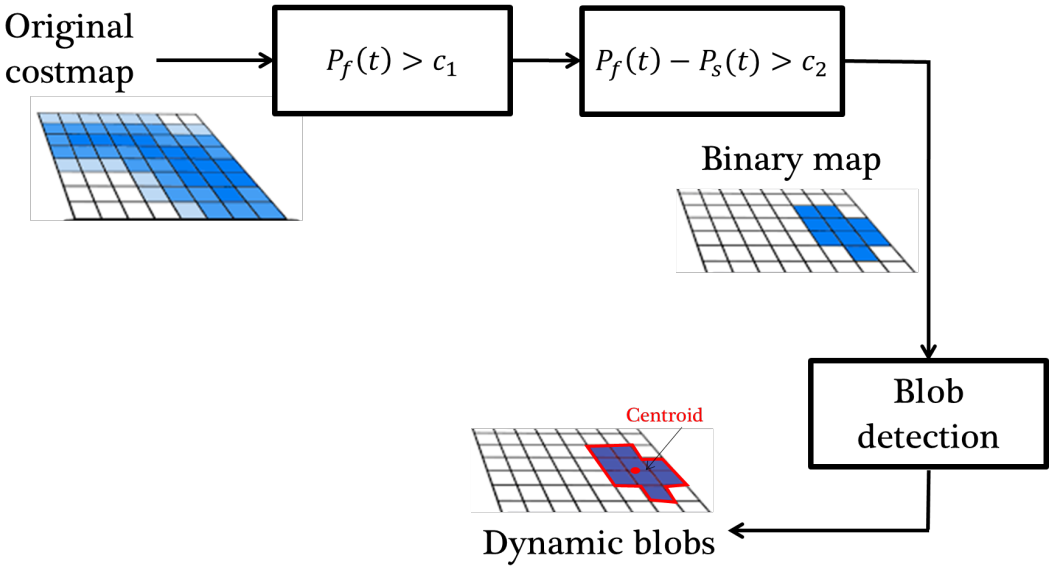


Figure 3.4: Dynamic objects detection output: centroids and contours.

3.2.2 Object tracking

The centroid of dynamic obstacles progresses with each costmap update and subsequent foreground detection. The assignment of blobs in the current map to obstacle tracks constitutes a data association problem. In order to disambiguate and track multiple objects over time, the current obstacles are matched with the corresponding tracks of previous obstacles. A new track is generated whenever a novel obstacle emerges that is not tracked yet. Tracks that are not assigned to current objects in the foreground frame are temporarily maintained. The track is removed if it is no longer confirmed by object detections over an extended period of time. The assignment problem is solved by the so-called Hungarian algorithm, which was originally introduced by [58]. The algorithm efficiently solves weighted assignment problems by minimizing the total Euclidean distance between the tracks and the current set of obstacle centroids. Then, a Kalman filter estimates the current velocity of tracked obstacles assuming a first order constant velocity model. The constant velocity model sufficiently captures the prevalent motion patterns of humans and robots in indoor environments.

Assignment Problem

The linear sum assignment problem to match a tracked obstacle with a detected obstalce can be formalized as follows. A problem instance is described by a matrix C, where each $C_{i,j}$ is the cost of matching a previously tracked obstacle i (set A) to a detected obstacle j (set B). Here the cost is calculated as the Euclidean distance between the obstacle objects of the two sets. The goal is to find a complete assignment of objects of set A to the ones of set B of minimal cost.

Formally, let X be a boolean matrix where $X_{i,j} = 1$ iff row i is assigned to column j. Then the optimal assignment has cost

$$\min \sum_{i \in A} \sum_{j \in B} C_{i,j} X_{i,j}$$

s.t. each row is assignment to at most one column, and each column to at most one row.

However, the original Hungarian algorithm solves the assignment problem with a square C matrix. This is not the case, since at each step new obstacles can appear in the scene, as well as others may disappear, so that matrix C might be often rectangular. Thus the exploited algorithm is a modified version of the original one [59], whose steps are directly reported in the following. It describes the manual manipulation of a two-dimensional matrix by starring and priming zeros and by covering and uncovering rows and columns.

- **Step 0**: Create an NxM matrix called the cost matrix in which each element represents the cost of assigning one of N tracks to one of M detections. Rotate the matrix so that there are at least as many columns as rows and let $k = min(n, m)$.
- **Step 1**: For each row of the matrix, find the smallest element and subtract it from every element in its row.
- **Step 2**: Find a zero (Z) in the resulting matrix. If there is no starred zero in its row or column, star Z. Repeat for each element in the matrix. Go to Step 3.
- **Step 3**: Cover each column containing a starred zero. If K columns are covered, the starred zeros describe a complete set of unique assignments. In this case, Go to DONE, otherwise, Go to Step 4.
- **Step 4**: Find a non covered zero and prime it. If there is no starred zero in the row containing this primed zero, Go to Step 5. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until there are no uncovered zeros left. Save the smallest uncovered value and Go to Step 6.
- **Step 5**: Construct a series of alternating primed and starred zeros as follows. Let Z0 represent the uncovered primed zero found in Step 4. Let Z1 denote the starred zero in the column of Z0 (if any). Let Z2 denote the primed zero in the row of Z1 (there will always be one). Continue until the series terminates at a primed zero that has no starred zero in its column. Unstar each starred zero of the series, star each primed zero of the series, erase all primes and uncover every line in the matrix. Return to Step 3.
- **Step 6**: Add the value found in Step 4 to every element of each covered row, and subtract it from every element of each uncovered column. Return to Step 4 without altering any stars, primes, or covered lines.
- **DONE**: Assignment pairs are indicated by the positions of the starred zeros in the cost matrix. If $C_{i,j}$ is a starred zero, then the element associated with row i is assigned to the element associated with column j.

Kalman Filter

Kalman filtering is an algorithm that provides estimates of some unknown variables given the measurements observed over time. It is generally used to estimate states based on linear dynamical systems in state space format. Here, it is first provided

the general formalization of the one-step Kalman Filter algorithm, secondly it is specified how it is adapted to the object tracking case, similarly to what is done in [60].
The process model defines the evolution of the state from time $k-1$ to time k as:

$$\mathbf{x}_k = \mathbf{F}x_{k-1} + \mathbf{B}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \tag{3.8}$$

where $\mathbf{F}$ is the state transition matrix applied to the previous state vector $\mathbf{x}_{k-1}$, $\mathbf{B}$ is the control-input matrix applied to the control vector $\mathbf{u}_{k-1}$, and $\mathbf{w}_{k-1}$ is the process noise vector that is assumed to be zero-mean Gaussian with the covariance $\mathbf{Q}$, i.e., $\mathbf{w}_{k-1} \sim \mathcal{N}(0, \mathbf{Q})$.

On the other hand, the measurement model describes the relationship between the state and the measurement at the current time step k:

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \nu_k \tag{3.9}$$

where $\mathbf{H}$ is the measurement matrix, $\mathbf{z}_k$ is the measurement vector, and ν_k is the measurement noise vector that is assumed to be zero-mean Gaussian with the covariance $\mathbf{R}$, i.e., $\nu_k \sim \mathcal{N}(0, \mathbf{R})$.

The role of the Kalman filter is to provide estimate of $\mathbf{x}_k$ at time k , given the initial estimate of $\mathbf{x}_0$, the series of measurement, $\mathbf{z}_1$,$\mathbf{z}_2$,. . . ,$\mathbf{z}_k$, and the information of the system described by $\mathbf{H}$, $\mathbf{B}$, $\mathbf{F}$, $\mathbf{R}$, and $\mathbf{Q}$. Even though the covariance matrices are supposed to reflect the statistics of the noises, the true statistics of the noises is not known or not Gaussian in many practical applications. Therefore, $\mathbf{Q}$ and $\mathbf{R}$ are usually tuned by the user to get the desired performances. The Kalman filter algorithm consists of two stages: prediction and update, which are repeated over and over one after the other. In the notation used in the following, superscript ˆ means 'estimate' of the underlying variable, so $\hat{\mathbf{x}}_{n|m}$ represents the estimate of $\mathbf{x}$ at time n given observations up to and including time $m \leq n$.

Predict	
Predicted (a priori) state estimate	$\hat{\mathbf{x}}_{k\|k-1} = \mathbf{F}_k\hat{\mathbf{x}}_{k-1\|k-1} + \mathbf{B}_k\mathbf{u}_k$
Predicted (a priori) estimate covariance	$\mathbf{P}_{k\|k-1} = \mathbf{F}_k\mathbf{P}_{k-1\|k-1}{\mathbf{F}_k}^\intercal + \mathbf{Q}_k$

Update	
Measurement residual	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k\|k-1}$
Kalman gain	$\mathbf{K}_k = \mathbf{P}_{k\|k-1}\mathbf{H}_k^\intercal(\mathbf{H}_k\mathbf{P}_{k\|k-1}\mathbf{H}_k^\intercal + \mathbf{R}_k)^{-1}$
Updated (*a posteriori*) state estimate	$\hat{\mathbf{x}}_{k\|k} = \hat{\mathbf{x}}_{k\|k-1} + \mathbf{K}_k\tilde{\mathbf{y}}_k$
Updated (*a posteriori*) estimate covariance	$\mathbf{P}_{k\|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\,\mathbf{P}_{k\|k-1}$

The predict stage produces an estimate of the state at the current timestep using the state estimate from the previous timestep. This predicted state estimate is also known as the *a priori* state estimate because it does not include any observation information from the current timestep, even though it is an estimate of the state at the current timestep. In the update phase, the innovation (the measurement residual), i.e. the difference between the current a priori prediction and the current observation information, is multiplied by the optimal Kalman gain and combined with the previous state estimate to refine the actual state estimate. This corrected estimate is termed the *a posteriori* state estimate.

In this work, the used Kalman filter estimates the current velocity of tracked obstacles assuming a first order constant velocity mode. Thus, the state space has 6 dimension, while the observation space has 3 – only position is known for a newly detected blob. The considered state vector for each dynamic obstacle is:

$$\mathbf{x} = [\mathbf{p}^\intercal, \mathbf{v}^\intercal]^\intercal$$

where $\mathbf{p} = [p_x, p_y, p_z]^\intercal$ is the position vector and $\mathbf{v} = [v_x, v_y, v_z]^\intercal$ is the velocity vector whose elements are defined in x, y, z axes. The state at time k can be predicted on the base of the previous state as:

$$\mathbf{x}_k = \begin{bmatrix}\mathbf{p}_k \\ \mathbf{v}_k\end{bmatrix} = \begin{bmatrix}\mathbf{p}_{k-1} + \mathbf{v}_{k-1}dt + \frac{1}{2}\tilde{\mathbf{a}}_{k-1}dt^2 \\ \mathbf{v}_{k-1} + \tilde{\mathbf{a}}_{k-1}dt\end{bmatrix}$$

where $\tilde{\mathbf{a}}_{k-1}$ is the eventual acceleration applied to the ego vehicle and dt is the sampling time. The above equation can be rearranged as:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{G}\tilde{\mathbf{a}}_{k-1} \tag{3.10}$$

There is no $\mathbf{B}u$ term since there are no known control inputs. Instead, $\tilde{\mathbf{a}}_{k-1}$ is the effect of an unknown input and $\mathbf{G}$ applies that effect to the state vector. In

particular:

$$\mathbf{F} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{I}_{3\times3}dt \\ \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix} \qquad \mathbf{G} = \begin{bmatrix} \frac{1}{2}\mathbf{I}_{3\times3}dt^2 \\ \mathbf{I}_{3\times3}dt \end{bmatrix}$$

where $\mathbf{F}$ is the state transition matrix, $\mathbf{0}_{3\times3}$ and $\mathbf{I}_{3\times3}$ denote 3×3 zero and identity matrices, respectively. Here a constant velocity model for the obstacles is assumed, meaning that between the $k-1$ and k speed is considered constant. Uncontrolled forces, instead, are supposed to cause a constant acceleration that is normally distributed, with mean 0 and standard deviation $\sigma_a = [\sigma_{ax}\ \sigma_{ay}\ \sigma_{az}]^\intercal$. So,Eq. (3.10) becomes:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{w}_k$$

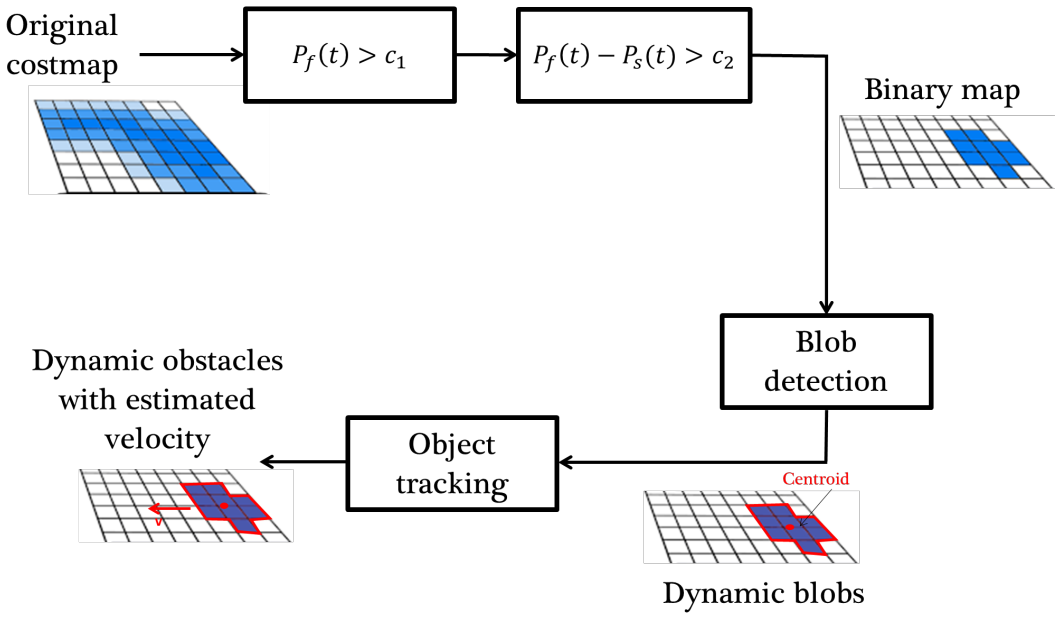


Figure 3.5: Dynamic objects tracking output.

where $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q})$. Therefore, within this assumption, the process noise covariance matrix is:

$$\mathbf{Q} = \mathbf{G}\mathbf{G}^\intercal {\sigma_a}^2 = \begin{bmatrix} \frac{1}{4}\mathbf{I}_{3\times3}dt^4\sigma_a^2 & \frac{1}{2}\mathbf{I}_{3\times3}dt^3\sigma_a^2 \\ \frac{1}{2}\mathbf{I}_{3\times3}dt^3\sigma_a^2 & \mathbf{I}_{3\times3}dt^2\sigma_a^2 \end{bmatrix}$$

where $\sigma_a^2 = [\sigma_{ax}^2\ \sigma_{ay}^2\ \sigma_{az}^2]^\intercal$. On the other hand, the measurement model is:

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \nu_k$$

where ν_k is the measurement noise with covariance matrix R and

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix}$$

since only position is measured. The variance of the position measurement errors has been reasonably assumed a priori, so that

$$\mathbf{R} = \mathbf{I}_{3\times 3}$$

Finally, matrix P can be initialized to:

$$\mathbf{P} = \begin{bmatrix} \mathbf{I}_{3\times 3} & \mathbf{0}_{3\times 3} \\ \mathbf{0}_{3\times 3} & 10 \times \mathbf{I}_{3\times 3} \end{bmatrix}$$

where error covariance on velocities has been set higher, since they are not measured.
All these matrices are provided to a solver which repeatedly computes the prediction and update steps, as better specified in the Implementation chapter (Chapter 4).

3.2.3 Cost assignment

The last step is embedding the information about detected dynamic obstacles into the costmap in a meaningful manner. In fact, the crucial problem is to translate the velocity vector information of each obstacle – magnitude and orientation – so that the robot navigation system, i.e. Planner and Controller, can easily account for it. The idea is to inflate the region around a detected dynamic obstacle with a two-dimensional Gaussian shape. In particular, the intuition suggested to associate the magnitude of the obstacle velocity with the peak of the Gaussian; in this way, faster obstacles are inflated more than slower ones. This has the aim to make the local planning aware of the obstacle with a sufficient heads-up for replanning. Furthermore, the orientation information is used to inflate more the cells along the moving direction of the obstacles. This is actually obtained blending two 2D Gaussian shapes, one inflating the cells in front of the obstacle and the other inflating the cells on its back region; this is an approach which has been inspired by a study about the social groups space modelling by Laga et al. [61].

In detail, given an obstacle O with centroid in position $c(x, y)$ in the map reference frame, we define a local coordinate system with origin in c, X-axis oriented in the velocity vector direction, Z-axis pointing outwards the costmap plane and Y-axis set according to the right-hand rule (Fig. 3.6). Therefore, the obstacle inflation region is represented by the following funtion:

$$\Phi_{c,\,\Sigma_{front},\,\Sigma_{back}}(q) = \delta(x_q)\Phi_{c,\,\Sigma_{front}}(q) + [1 - \delta(x_q)]\Phi_{c,\,\Sigma_{back}}(q) \tag{3.11}$$

where $q = (x_q, y_q)$ are the coordinates of a point in the map reference frame, $\Phi_{c,\Sigma_{front}}$ is the Gaussian function that inflates the frontal area of the obstacle, $\Phi_{c,\Sigma_{back}}$ is the Gaussian function that inflates the back space, and $\delta(x)$ is the parameter that blends the two functions to account for the obstacle orientation, being:

$$\delta(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

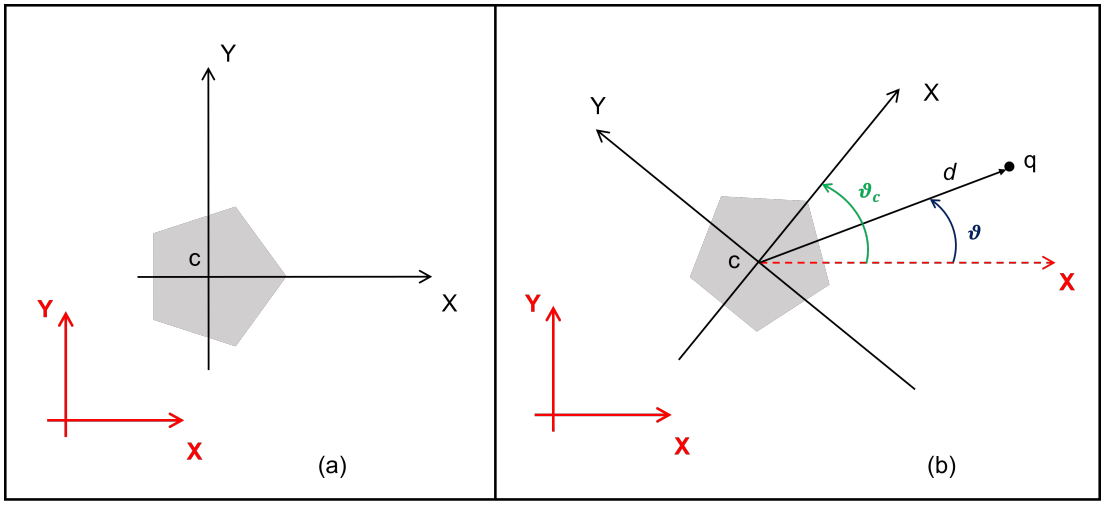


Figure 3.6: Local obstacle reference frame wrt the global (map) reference frame.

For both the Gaussian functions, it is adopted a formulation similar to the one proposed by Yang et al. [62]:

$$\Phi_{c,\Sigma}(q) = A\, exp\left\{ -\frac{[d\, cos(\vartheta - \vartheta_c)]^2}{2\sigma_x^2} - \frac{[d\, sin(\vartheta - \vartheta_c)]^2}{2\sigma_y^2} \right\} \tag{3.13}$$

where d is the Euclidean distance and between $q = (x_q, y_q)$ and $c(x, y)$ and ϑ is the angle formed bay the point position vector with respect to the global X-axis, ϑ_c is the angle between the obstacle moving direction and the X-axis of the map coordinate system (Fig. 3.6), A is an amplitude parameter set to the maximum cost possible on the costmap, i.e. 255, and σ_x^2, σ_y^2 are the diagonal entries of the Σ covariance matrix, which determines the shape of the inflation region. In particular, the two covariance matrices are defined as follows:

$$\Sigma_{front} = \begin{pmatrix} \sigma^2_{x\_front} & 0 \\ 0 & \sigma^2_{y\_front} \end{pmatrix}; \qquad \Sigma_{back} = \begin{pmatrix} \sigma^2_{x\_back} & 0 \\ 0 & \sigma^2_{y\_back} \end{pmatrix} \tag{3.14}$$

Therefore, σ_x and σ_y can be tuned to model a generic shape at will. Here, in order to take in account the obstacle velocity magnitude, a maximum obstacle speed

max_speed has been supposed. Then, in order to inflate more the front region, we define the *speed ratio* $r = \frac{vel}{max\_speed}$, where *vel* is the obstacle detected speed. Finally, the variances are modified according to the following heuristics:

$$\begin{cases} \sigma^2_{x\_front} = & (1+r)\sigma^2_{x\_front} \\ \sigma^2_{y\_front} = & (1-\frac{r}{2})\sigma^2_{y\_front} \\ \sigma^2_{x\_back} = & (1-r)\sigma^2_{x\_back} \\ \sigma^2_{y\_back} = & (1-\frac{r}{4})\sigma^2_{y\_back} \end{cases} \tag{3.15}$$

In this way, the Gaussian shape is lengthened in the direction of the obstacle motion and narrowed in the lateral area.

Gaussian functions blending Here few words are spent to make the meaning of the δ parameter clearer. To be more specific, let's consider Figure 3.7, where again a detected dynamic obstacle is located at position $c(x, y)$ and its estimated velocity vector forms an angle ϑ_c with the positive X-semiaxis of the map frame. To determine if a point (a costmap cell) is inside the obstacle front or back space, the simple concept of inner product between vector has been used. Being $\vec{v}$ the obstacle velocity vector and $\vec{q}$ the position vector of the considered point in the local reference frame, q will be in the front space if the following condition holds:

$$\vec{v} \cdot \vec{q} \geq 0 \Rightarrow cos|\vartheta_c - \vartheta| \geq 0 \tag{3.16}$$

Figure 3.7 shows the two types of possible cases. In the first scenario (Fig. 3.7-a),

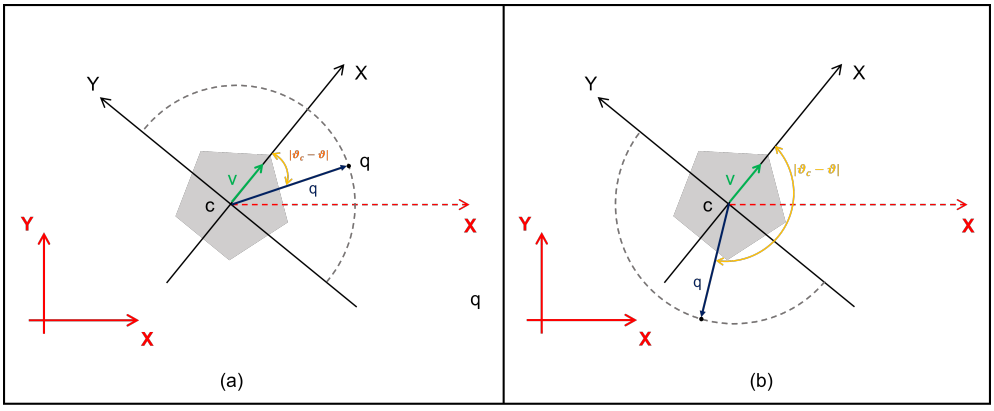


Figure 3.7: a) A point q in the frontal space. b) A point q within the back area.

where the q point is in the obstacle frontal space, the projection of the vector $\vec{q}$ along the obstacle's moving direction is parallel to the velocity vector $\vec{v}$, thus generating a positive inner product. In the second case (Fig. 3.7-b), with the q point in the human back space, instead, the projection of $\vec{q}$ has opposite direction

to $\vec{v}$ producing a negative inner product.

Chapter 4

Implementation

This chapter provides a general description of the implementation of the approach proposed above inside the ROS2 framework.

First of all, Linux Ubuntu 20.04 operating system was installed on the Intel NUC computer, after having reported several performance issues when trying to configure the whole environment in a virtual machine on my personal PC. Secondly, ROS2 Foxy has been installed from debian packages following the instructions on the official documentation website[1]. Then, four workspaces have been created to keep a clean management of the souce code:

- `nav2_ws` – Here the Nav2 stack source has been downloaded and built from Nav2 documentation website[2] in order to allow modifications of the source files.

- `turtlebot3_ws` – This is the workspace where I downloaded the Turtlebot3 packages from ROBOTIS[3]. It contains all the necessary packages to run Navigation 2 on TurtleBot3 robot, both in simulation and in real world; it offers a useful interface package for Gazebo simulator as well – not used in this thesis work.

- `ros2_webots_ws` – It contains all the Webots example packages[4] for the different types of robots, including Turtlebot3. They provide all you need to build the robot model in a virtual environment. Installing these packages from source allowed to modify the parameters defining the TurtleBot3 model and, in particular, the LiDAR sensor parameters, changed according to the RPLIDAR A3 specifications.

[1] `https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html`

[2] `https://navigation.ros.org/build_instructions/index.html#for-main-branch-development`

[3] `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/`

[4] `https://automaticaddison.com/how-to-install-and-demo-the-webots-robot-simulator-for-ros-`

- `ros2_ws` – It is the workspace where the packages implementing the proposed approach are built, namely `ros2_costmap_to_dynamic_obstacles`, `kf_hungarian_tracker`, `nav2_dynamic_costmap_layer_plugin` and `nav2_dynamic_msgs`. The details are outlined in the following paragraphs.

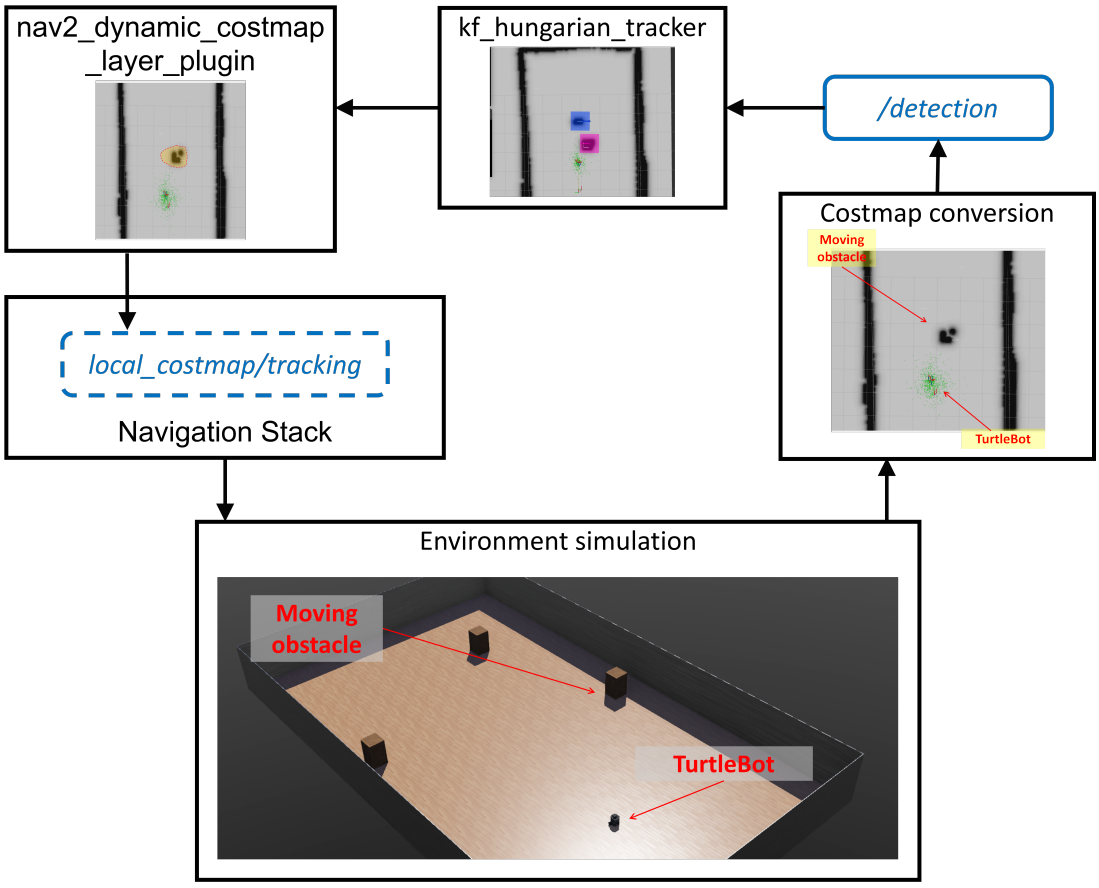


Figure 4.1: General scheme of how the different software functional blocks interact with each other and with the simulates external environment.

The very high level view of all these packages interactions is shown in Figure 4.1. As it is shown, the `ros2_costmap_to_dynamic_obstacles` package implements the obstacle detection (see Sec. 3.2.1) functions and outputs the blobs corresponding to detected obstacles through a specific custom ROS2 message type on the `/detection` topic. The `kf_hungarian_tracker` package provides a subscription to this topic, so that it can perform object tracking (see Sec. 3.2.2). Then, it publishes the dynamic obstacles and their estimated velocities on the `local_costmap/tracking` topic, directly communicating with the Nav2 stack. Finally, the costmap layer, implemented through `nav2_dynamic_costmap_layer_plugin`, processes this information to compute the Gaussian costs (see Sec. 3.2.3) and updates the master costmap used for the robot navigation. Each functional block is now illustrated in more details.

4.1 Messages and topics

The communication of the dynamic obstacles information occurs through two message types: `Obstacle.msg` and `ObstacleArray.msg` defined in the `nav2_dynamic_msgs` package:

- `Obstacle.msg`

```
std_msgs/Header header
unique_identifier_msgs/UUID id
geometry_msgs/Point position # center position
geometry_msgs/Vector3 velocity
geometry_msgs/Vector3 size
```

  where UUID is a 128-bit ID [63] uniquely associated to each obstacle, `position` contains the coordinates of the obstacle centroid, `velocity` is the estimated velocity vector ad `size` contains the dimensions of the obstacle bounding box along the three axies X-, Y- and Z-axis. Actually, the third dimension is ignored, since the LiDAR provides two-dimensional data.

- `ObstacleArray.msg`

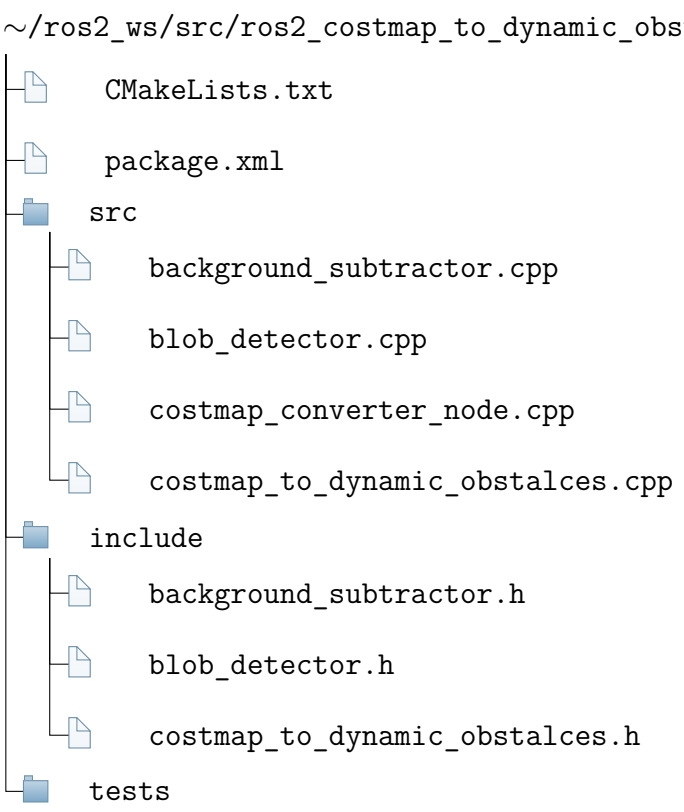
```
std_msgs/Header header
Obstacle[] obstacles
```

  which is basically an array of obstacle messages generated at the same time.

As shown in Figure 4.1, these messages are communicated through topics `/detection` and `local_costmap/tracking`, however, when `ros2_costmap_to_dynamic_obstacles` communicates the detected blobs to the `kf_hungarian_tracker`, the velocity information is not present yet, so this field is initialized to a zero vector, while `position` and `size` are filled as detailed in the following section.

4.2 Costmap conversion

The `ros2_costmap_to_dynamic_obstacles` package has the following composition:



costmap_converter_node The `costmap_converter_node` is the ROS2 node that executes the conversion process. When the node is spinned, its constructor `CostmapConversionNode::CostmapConversionNode()` creates a costmap (`costmap_ros_`) parallel to the one generated by the navigation process on a separate thread, with custom parameters and a publish frequency of $5\,Hz$, as the local costmap, using the `costmap_2d::Costmap2DROS` object wrapper. Then it instantiates the `/detection` topic creating a publisher with a timer of $150\,ms$, calling all the conversion processes (see next paragraph) from the publisher callback function `publishCallback()`.

costmap_to_dynamic_obstacles The `costmap_to_dynamic_obstalces .cpp` and `.h` files provide the object detection functionalities through the `CostmapToDynamicObstacle` class. An object of this class is instantiated by the `costmap_converter_node` and `CostmapToDynamicObstacle::initialize()` initializes all the parameters of the `CostmapToDynamicObstacle` object used for the successive background subtraction and blob detection. Then, the `publishCallback()` callback function executes the following function calls:

- `CostmapToDynamicObstacle::setCostmap2D()` Converts the `costmap_ros_` into an OpenCv matrix object (`cv::Mat` [5]), so that it can be manipulated

by `background_subtractor` and `blob_detector`.

- `CostmapToDynamicObstacle::compute()` performs the actual work (conversion of the costmap to obstacles):

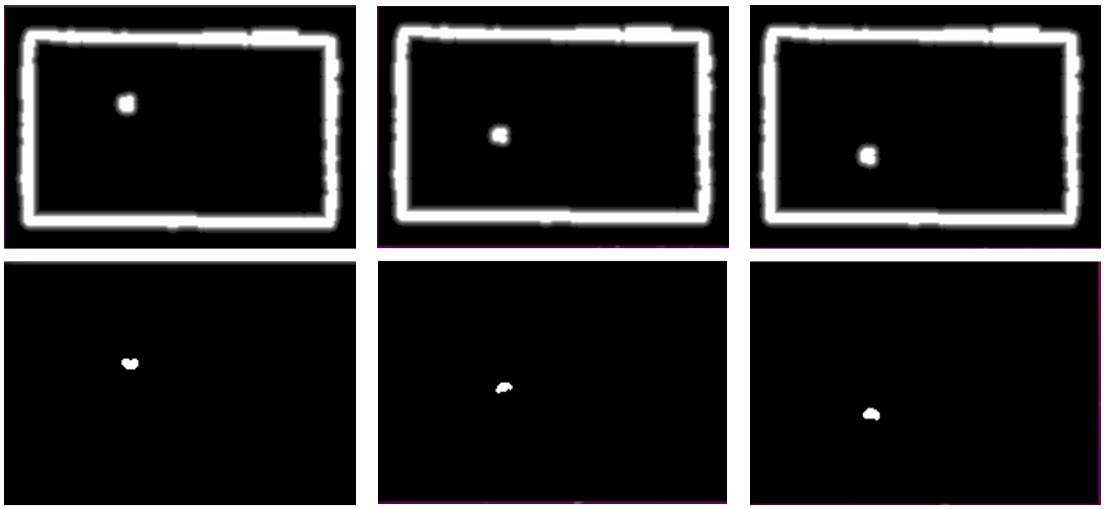
  1) Separation of dynamic obstacles from static ones. Here it calls `BackgroundSubtractor::apply()` method. Through this method, dynamic obstacles are isolated in the foreground mask `fg_mat`, an 8-bit binary image containing the filtered dynamic pixels.

  2) Blob detection is done calling `BlobDetector::detect()` method, which returns each blob in terms of two arrays: `keypoints_` containing the blobs centroids, and `contours` containing the blobs contours as an array of `cv::Point`(s) for each blob.

  3) Fill the `ObstacleArrayMsg` (the array of obstacles object messages): for each tracked objects it sets the obstacle message fields: a UUID is generated, `keypoints_[i]` is assigned to `position`, `header` is filled with the current timestamp and `'/map'` global reference frame, since when the `Costmap2D` object is converted into a `cv::Mat` object, the reference frame information is preserved. Finally, in order to simplify the `kf_hungarian_tracker` job, given an obstacle contours, a bounding rectangular area is generated calling `cv::boundingRect()`, rectangle width and length are then properly assigned to the `size` field.

Figure 4.2 shows the output of background subtraction and blob detection for a single moving obstacle in a rectangular empty environment. It can be noticed that the static obstacles (room walls appearing as the white rectangle) are correctly filtered out and only the dynamic pixels appear in the `fg_mask`.

[5]"The class Mat represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or color images, voxel volumes, vector fields, point clouds, tensors, histograms" [64].



(a) $t = 1s$. **(b)** $t = 2s$ **(c)** $t = 3s$

Figure 4.2: Time evolution of the filtered moving obstacles. Above there are the original `costmap_ros_` converted to `cv::Mat`, while below there are the corresponding `fg_mask`(s).

4.3 Hungarian tracker

The `kf_hungarian_tracker` package has the following composition:

```
~/ros2_ws/src/kf_hungarian_tracker
└── kf_hungarian_tracker
    ├── __init__.py
    ├── kf_hungarian_node.py
    └── obstacle_class.py
├── package.xml
├── setup.py
└── tests
```

`obstacle_class` The `obstacle_class.py` defines the `ObstacleClass`, which wraps a Kalman filter and extra information for one single obstacle:

```
class ObstacleClass:
"""
State space is 3D (x, y, z) by default,simply make z a constant value
and independent of x, y to work on 2D.
```

```
Arttibutes:
    position: 3d position of center point, numpy array (3, 1)
    velocity: 3d velocity of center point, numpy array (3, 1)
    kalman: cv2.KalmanFilter
    dying: count missing frames for this obstacle, if reach
    threshold, delete this obstacle
"""
[...]
```

In particular, the `cv2.KalmanFilter` is the OpenCV calss [65] which implements the Kalman fiter predict and update steps, precisely, through the methods `cv2.KalmanFilter.predict()` and `cv2.KalmanFilter.update()`. When an object of this class is instantiated, matrices $\mathbf{H}$, $\mathbf{F}$, $\mathbf{R}$, $\mathbf{P}$ and $\mathbf{Q}$ are initialized as specified at the end of Subsection 3.2.2.

`kf_hungarian_node` The `kf_hungarian_node` is the ROS2 node that executes the obstacle tracking and solves the assignment problem:

```
class KFHungarianTracker(Node):
'''
Use Hungarian algorithm to match presenting obstacles with new
detection and maintain a kalman filter for each obstacle.
Spawn ObstacleClass when new obstacles come and delete when
they disappear for certain number of frames

Attributes:
    obstacle_list: a list of ObstacleClass that currently
    present in the scene
    sec, nanosec: timing from sensor msg
    detection_sub: subscrib detection result from detection node
    tracker_obstacle_pub: publish tracking obstacles with
    ObstacleArray
    tracker_pose_pub: publish tracking obstacles with
    PoseArray, for rviz visualization
'''
[...]
```

First of all it creates a publisher to the `local_costmap/tracking` topic and a subscriber to the `/detection` topic. All tracked obstacles are stored in `obstacle_list`, while the incoming detected obstacles from message are stored

in `detections` list. The core processes are then executed by the subscription callback function in the following order:

1) Compute the delta time used as the Kalman filter timestep as the difference between the arriving message (on `/detection`) timestamp and the timestamp of the previous message, locally stored.

2) Read message from topic and execute the Kalman prediction step, calling `ObstacleClass.predict()`.

3) Having `obstacle_list` and `detections` compute the cost, i.e. the Euclidean distance, for all the combination of `obstacle_list[i]` and `detections[j]`. Then, solve the assignment problem with the Hungarin algorithm implemented by `scipy.optimize.linear_sum_assignment(cost_matrix)` (available in the `scipy` library.

4) Execute the Kalman update step by calling `ObstacleClass.correct()`.

5) Construct the `ObstacleArray` message and publish it on the `local_costmap/-tracking` topic.

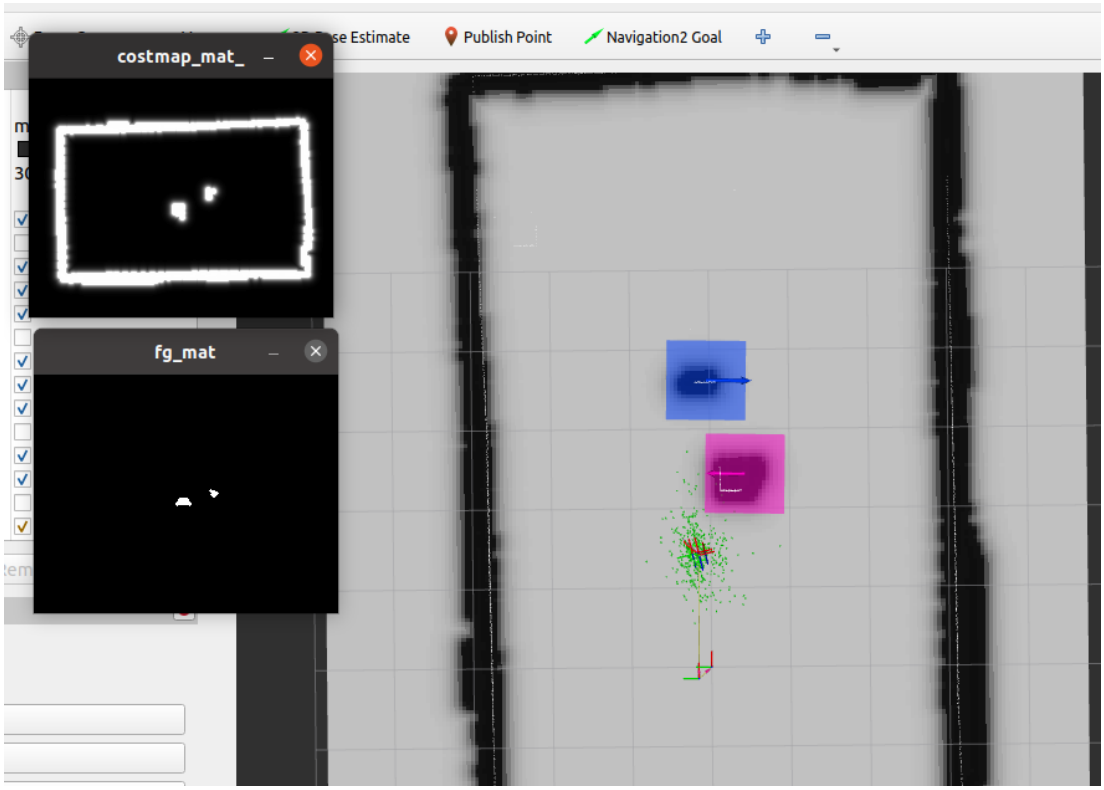


Figure 4.3: Rviz visualization of `kf_hungarian_tracker` node in case of two moving obstacles.

Figure 4.3 shows s typical output of the `kf_hungarian_tracker` node: two moving obstacles are detected and their estimated velocity is displayed, together with obstacles bounding boxes.

4.4 Dynamic costmap layer plugin

Before illustrating the `nav2_dynamic_costmap_layer_plugin` some modifications to the Nav2 source are presented, specifically regarding the `nav2_costmap_2d` package.

4.4.1 Modifications to `nav2_costmap_2d`

Each costmap layer plugin (`ObstacleLayer`, `InflationLayer`, etc.) is defined as a class derived from the `Layer` base class. The latter provides all the attributes and methods needed to define a custom costmap layer. Then, the `LayeredCostmap` class is used to handle the master costmap resulting from the layers composition – according to a specified policy – and to interface it with all the other Nav2 functionalities thorugh the `Costmap2DROS` Node class. When initialized, a `Layer` object provides the following ROS instances:

```
void Layer::initialize(
    LayeredCostmap * parent,
    std::string name,
    tf2_ros::Buffer * tf,
    nav2_util::LifecycleNode::SharedPtr node,
    rclcpp::Node::SharedPtr client_node,
    rclcpp::Node::SharedPtr rclcpp_node);
    [...]
```

So, a `Layer` object is basically a lifecycle node, directly connected to a parent `LayeredCostmap`. Actually, in the current ROS2 Foxy version, the Lifecycle nodes management of subscriptions has some problems. As a matter of fact, initially, the implemented `nav2_dynamic_costmap_layer_plugin` tried to subscribe to the `local_costmap/tracking` topic directly from within the plugin class (`DynamicLayer`), but without succes. Therefore, it was necessary to add some lines of code in the files defining the `Costmap2DROS` and `LayeredCostmap` classes.

First of all, in `layered_costmap.cpp` and `layered_costmap.h` a container of received dynamic obstacles has been added as a new attribute of `LayeredCostmap`

class:
`nav2_dynamic_msgs::msg::ObstacleArray::ConstSharedPtr dynamic_obstacles;`
Two methods are added to update the obstacle container and read from it safely:

- Update the stored container of dynamic obstacles:

```
void LayeredCostmap::updateDynamicObstaclesContainer(const
  ObstacleArrayPtr obstacle_msg)
{
    std::lock_guard<std::mutex> lock(mutex_);
    dynamic_obstacles = obstacle_msg;
 }
```

- Return the dynamic obstacles detected by the `kf_hungarian_tracker` in the form of `ObstacleArray` message:

```
ObstacleArrayConstPtr LayeredCostmap::getDynamicObstacles()
{
    std::lock_guard<std::mutex> lock(mutex_);
    return dynamic_obstacles;
 }
```

In this way, independently of the costmap plugin loaded – different plugin could be eventually created in the future – each generic costmap plugin could access the dynamic obstacles information.

After that, similarly to what happens for transmitting the information about the robot footprint across all the layers, here, it is the `Costmap2DROS` ROS node that subscribes to the `local_costmap/tracking` topic:

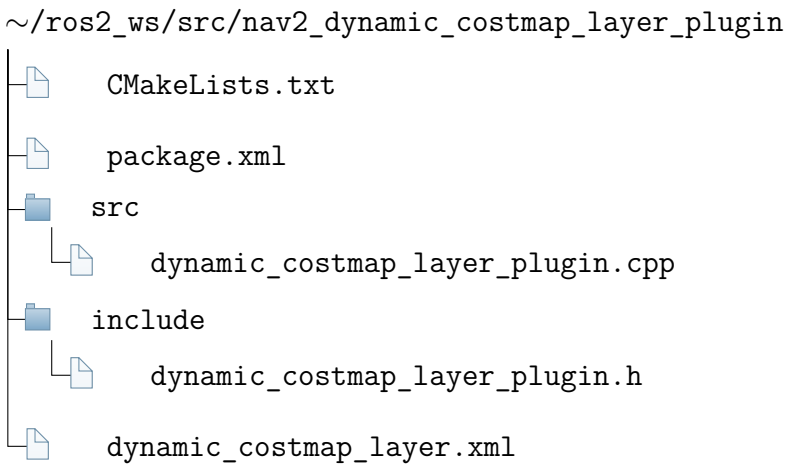
```
if(dynamic_obstacles_plugin){
    tracks_sub_ =
      create_subscription<nav2_dynamic_msgs::msg::ObstacleArray>(
      "tracking",
      10,
      std::bind(&Costmap2DROS::tracksCallback, this,
        std::placeholders::_1));
      RCLCPP_INFO(get_logger(), "Created subscription to dynamic
        obstacles topic");
  }
```

Subscription to the topic occurs only when `dynamic_obstacles_plugin = True`, that is the dynamic obstacle layer plugin is enabled at Nav2 launch from the `.yaml` file specifying all the launch information. The callback function simply reads the messages from the topic and updates the obstacles container:

```
Costmap2DROS::tracksCallback(const ObstacleArrayPtr obstacle_msg)
{
    layered_costmap_->updateDynamicObstaclesContainer(obstacle_msg);
}
```

4.4.2 nav2_dynamic_costmap_layer_plugin

The dynamic costmap layer plugin is developd in a package with the following structure:



The layer class `DynamicLayer` provides the Gaussian cost assignment around dynamic obstacles mainly through the `DynamicLayer::updateCosts()` method. In particular, it executes the following steps:

1) Get dynamic obstacles and store them locally:

```
[...]
ObstacleArrayConstPtr obstacles(new
  nav2_dynamic_msgs::msg::ObstacleArray);
[...]
obstacles = layered_costmap_->getDynamicObstacles();
[..]
```

2) Transform between frames: dynamic obstacles are computed and processed by `ros2_costmap_to_dynamic_obstacles` and `kf_hungarian_tracker` packages with respect to '`/map`' reference frame. However, the dynamic costmap layer is inteded to be plugged into the `local_costmap`, which uses the '`/odom`' as global reference frame. Therefore, it is necessary to transform the obstacles position and velocity information accordingly, in order to assign the costs correctly. This is done using the `tf2` package:

```
geometry_msgs::msg::TransformStamped map_to_odom_transform;
obstacles = layered_costmap_->getDynamicObstacles();
obs_frame = obstacles->header.frame_id;
global_frame = layered_costmap_->getGlobalFrameID();
obs_frame = obs_frame.substr(1);  // remove the "/" from topic
  name

try
{
  map_to_odom_transform = tf_->lookupTransform(global_frame,
    obs_frame, tf2::TimePointZero);
} catch (tf2::TransformException & ex) {
      RCLCPP_INFO(rclcpp::get_logger("nav2_costmap_2d"),
        "DynamicLayer: Could not transform %s to %s: %s",
         obs_frame.c_str(), global_frame.c_str(), ex.what());
      return;
}
```

3) Loop through all the obstacles in the container and assign costs for each of them by calling `DynamicLayer::markDynamicObstacle()`, which the function that implements the Gaussian composition as illustrated in subsection 3.2.3. Here is the function prototype:

```
/**
* @brief update the relevant grid cells in the cost map according
  to the given pose of a person
*
* @param obstacle_in_costmap_x map cell x coordinate of obstacle
  center
* @param obstacle_in_costmap_y map cell y coordinate of obstacle
  center
* @param angle angle of obstacle in map (orientation wrt the
  global frame)
```

```
* @param costmap costmap to update
* @param vel velocity module of the obstacle.
*/
void markDynamicObstacle(int obstacle_in_costmap_x, int
  obstacle_in_costmap_y, double angle, nav2_costmap_2d::Costmap2D
  *costmap, unsigned char * costmap_array, double vel);
```

The `nav2_dynamic_costmap_layer_plugin` typical output is shown in Figure 4.4: in the simulated world (on the right) an obstacle in robot's local space is moving leftwards. The robot successfully estimate its motion direction and the costmap layer assigns costs so that the Gaussian has a peak oriented towards the left.

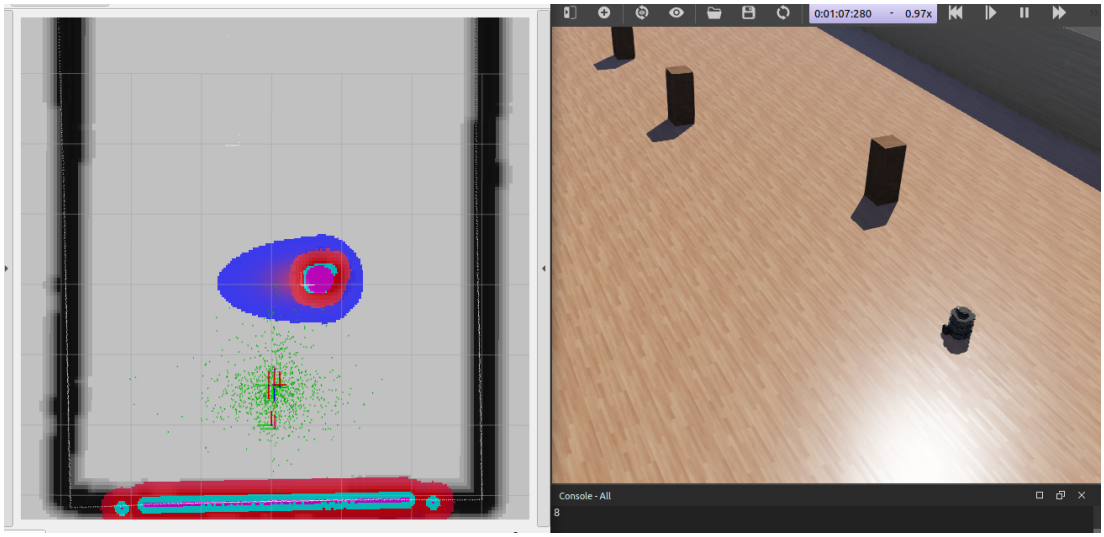


Figure 4.4: An obstacle moving leftwards with respect to the TurtleBot.

Chapter 5

Simulation environment and tests

This chapter illustrates the general setup of the simulation environment realized to test, debug and validate the navigation application developed in this thesis work. The essential tools used for this purpose are Webots 2021a [66], for the creation and simulation of the environments and the robot model, and Rviz [67] for the visualization of internal outputs and debugging.

Lastly, the path planning approach available in ROS2 (DWB) and the DWB integrated with the proposed Dynamic Obstacle Layer method (DWB + DOL) are compared through simulations.

5.1 Simulation Environment

When building robotic applications within the ROS framework, a realistic and fast responsive simulation environment is fundamental to visualize a complex behavior such as an autonomously navigating robot. Usually, the developers choice falls on the Gazebo simulator [68], but here, upon suggestion of LINKS, Webots has been adopted.

5.1.1 Webots

Webots allows to create 3D virtual worlds with physics properties such as mass, joints, friction coefficients, etc. The user can add simple passive objects or active objects called mobile robots. These robots can have different locomotion schemes (wheeled robots, legged robots, or flying robots). Moreover, they may be equipped with a number of sensors and actuator devices, such as distance sensors, drive wheels, cameras, motors, touch sensors, emitters, receivers, etc. Finally, the user can program each robot individually to exhibit the desired behavior.

A world, in Webots, is a 3D description of the properties of robots and of their environment. It contains a description of every object: position, orientation, geometry, appearance (like color or brightness), physical properties, type of object, etc., in the form of Webots *Nodes*. Worlds are organized in hierarchical structures defined as *Scene Tree*. For example, it specifies if a robot contains two or three wheels, a distance sensor and a joint, which itself contains a camera, etc. A robot can be also imported into a world as an external PROTO file, to keep the world file cleaner. A Webots *Controller* is a computer program that controls a robot specified in a world file. Controllers can be written in any of the programming languages supported by Webots: C, C++, Java, Python or MATLAB. When a simulation starts, Webots launches the specified controllers, each as a separate process, and it associates the controller processes with the simulated robots.

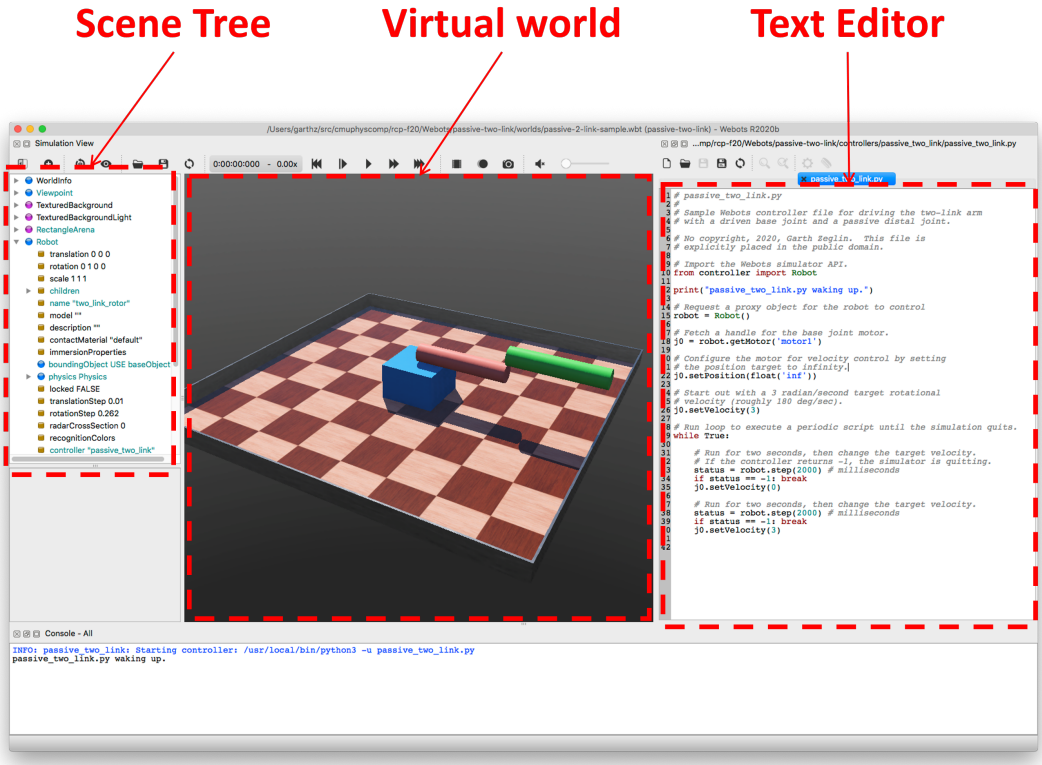


Figure 5.1: Sample Webots interface. The text editor on the right can be used to modify the world or the controller files without the need of and external IDE.

The Webots interface allows to modify the source files live through an embedded text editor and make the modifications effective just reloading the world.

5.1.2 Environments

First of all, the robot model in Webots had to be modified in order to reflect the custom hardware setup, i.e., mounting of the RPLIDAR A3 instead of the default LDS-01 LiDAR. This because a PROTO file for the RPLIDAR A3 is not available in Webots, yet. To do so, it has been necessary to create a new sensor PROTO file in Webots installation folder (`~/.ros/webotsR2021a/webots/projects/devices/-robotis/protos`. So, starting from a copy of the latter, the following lines have been changed in the new `RPLidar.proto` file:

```
[...]
 horizontalResolution 1600  // = fieldOfView/angularResolution, which
   is 0.225° for rplidar A3
 fieldOfView 6.28318   // 360°
 numberOfLayers 1
 near 0.07
 minRange 0.2  // [m]
 maxRange 25  // [m]
 noise IS noise
[...]
```

The result is a sensor with the same aspect and position on the Turtlebot of the LDS-01, but with performances reflecting the specifications of the RPLIDAR A3 (see Table 2.3).

Once the robot model has been defined, two virtual environments have been created. The main world is an empty rectangular arena $10\,m \times 6\,m$ where dynamic obstacles are simulated as wooden boxes of $20\,cm \times 20\,cm$ base and $50\,cm$ tall, configured as robot nodes. Once placed in position, the boxes move with a constant speed back and forth traversing the entire arena along the short side direction. A controller has been coded for each obstacle and the speed can be commanded when the world is launched, so that simulations scenarios can be easily modified. The presented environment is shown in Figure 5.2.

This has been the world used to debug the developed source code step by step with the use of Rviz.

The second environment is a more realistic world, intended to simulate the final robot behavior in a context similar to the LINKS offices (Fig. 5.3). Indeed, an office room setup has been realized starting from an existent template, adding a model of a walking person (`Pedestrian.proto`).

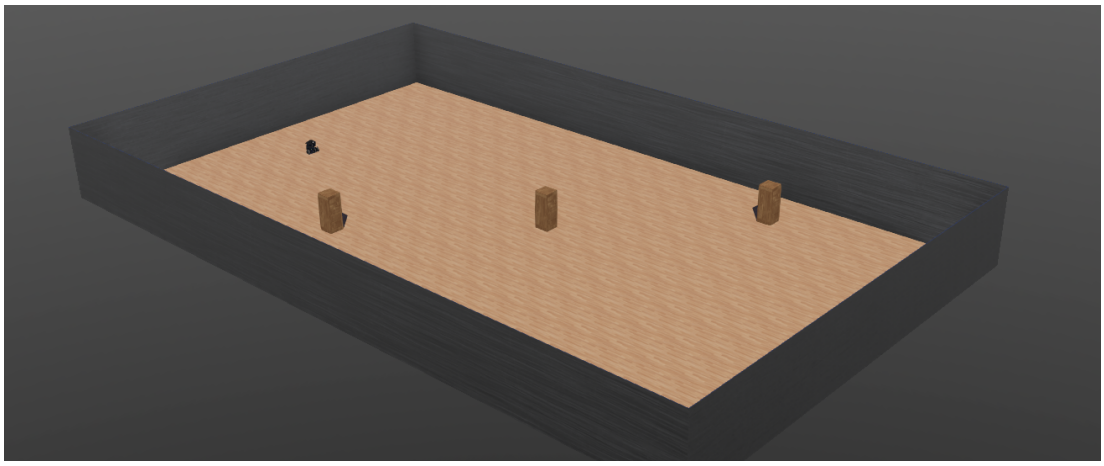


Figure 5.2: Main dynamic environment used for simulations and debugging (`dyn_env_1.wbt`).

Figure 5.3: LINKS offices simulation environment. (`break_room.wbt`).

5.1.3 Rviz

Rviz is a 3d visualization tool for ROS applications. It provides a view of robot model and surroundings, captures sensor information from robot sensors, and replays captured data. It can display data from cameras, lasers, from 3D and 2D devices, including pictures and point clouds. Rviz has been used throughout the whole development to visualize the output of each package illustrated in Chapter 4. In fact, `costmap_converter_node` and `kf_hungarian_tracker` node both contain a publish function for Rviz:

- `costmap_converter_node` publishes the bounding boxes of detected dynamic obstacles through `publishAsMarker()` function. This, together with the `fg_mat_` visualization, helped in verifying the output of the obstacles detection:

```
void publishAsMarker(
    const std::string &frame_id,
    const nav2_dynamic_msgs::msg::ObstacleArray &obstacles) {
    visualization_msgs::msg::MarkerArray box;
  for (auto obstacle : obstacles.obstacles) {
    box.markers.emplace_back();
    box.markers.back().header.frame_id = frame_id;
    box.markers.back().header.stamp = now();
    box.markers.back().ns = "Obstacles Markers";
    box.markers.back().type =
      visualization_msgs::msg::Marker::CUBE;
    box.markers.back().action =
      visualization_msgs::msg::Marker::ADD;
    box.markers.back().pose.orientation.w = 1.0;
    box.markers.back().pose.position = obstacle.position;
    box.markers.back().scale = obstacle.size;
    box.markers.back().scale.z = 0.01;

    box.markers.back().color.r = 0.0;
    box.markers.back().color.g = 1.0;
    box.markers.back().color.b = 0.0;
    box.markers.back().color.a = 0.5;
  }
  marker_pub_->publish(box);
}
```

- `kf_hungarian_tracker` publishes bounding boxes and velocity vectors, clearing the display when obstacles disappear from the range and properly adding newly detected ones:

```
# ruiz visualization
if self.tracker_marker_pub.get_subscription_count() > 0:
    marker_array = MarkerArray()
    marker_list = []
    # add current active obstacles
    for obs in filtered_obstacle_list:
        obstacle_uuid = uuid.UUID(bytes=bytes(obs.msg.id.uuid))
        (r, g, b) = colorsys.hsv_to_rgb(obstacle_uuid.int % 360 /
          360., 1., 1.) # encode id with rgb color
        # make a cube
```

```
        marker = Marker()
        marker.header = msg.header
        marker.ns = str(obstacle_uuid)
        marker.id = 0
        marker.type = 1 # CUBE
        marker.action = 0
        marker.color.a = 0.5
        marker.color.r = r
        marker.color.g = g
        marker.color.b = b
        marker.pose.position = obs.msg.position
        angle = np.arctan2(obs.msg.velocity.y,
          obs.msg.velocity.x)
        marker.pose.orientation.z = np.float(np.sin(angle / 2))
        marker.pose.orientation.w = np.float(np.cos(angle / 2))
        marker.scale = obs.msg.size
        marker_list.append(marker)
        # make an arrow
        arrow = Marker()
        arrow.header = msg.header
        arrow.ns = str(obstacle_uuid)
        arrow.id = 1
        arrow.type = 0
        arrow.action = 0
        arrow.color.a = 1.0
        arrow.color.r = r
        arrow.color.g = g
        arrow.color.b = b
        arrow.pose.position = obs.msg.position
        arrow.pose.orientation.z = np.float(np.sin(angle / 2))
        arrow.pose.orientation.w = np.float(np.cos(angle / 2))
        arrow.scale.x = np.linalg.norm([obs.msg.velocity.x,
          obs.msg.velocity.y, obs.msg.velocity.z])
        arrow.scale.y = 0.05
        arrow.scale.z = 0.05
        marker_list.append(arrow)
    # add dead obstacles to delete in rviz
    for uuid_ in dead_object_list:
        marker = Marker()
        marker.header = msg.header
```

```
            marker.ns = str(uuid_)
            marker.id = 0
            marker.action = 2 # delete
            arrow = Marker()
            arrow.header = msg.header
            arrow.ns = str(uuid_)
            arrow.id = 1
            arrow.action = 2
            marker_list.append(marker)
            marker_list.append(arrow)
        marker_array.markers = marker_list
        self.tracker_marker_pub.publish(marker_array)
```

As it has been anticipated above, Riz was the only tool usable to understand if the code behaved as expected. So, before starting any development and testing activity, it was necessary to configure the virtual world environment with Rviz through SLAM (Simultaneous Localization and Mapping).

5.2 SLAM

"SLAM is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it" [69]. That is, the robot is moved in an unknown environment (the one in Fig. 5.2 for example) and it builds the static map used for all the subsequent tests. In particular, the `cartographer` – a Google open source project [70] – has been used as a default SLAM method, implemented in the `turtlebot3_cartographer` package. To perform SLAM, for both the environments mentioned above, two static versions have been created. In particular, `dyn_env_1_static` world is the copy of `dyn_env_1` without moving boxes, while `break_room_static` world is the copy of `break_room` but without the pedestrian. The Cartographer method uses a grid map-based world representation; it actually performs two steps. The first one is called "local SLAM" and it uses a Ceres scan matcher [71] on a small map, known as a submap, to estimate the pose and orientation of the vehicle. The second step is "global SLAM", which optimizes the pose by utilizing a larger scan matcher on the aggregation of the submaps. The steps followed to acquire the static map for a given environment are the following:

1) Launch the static virtual environment to be mapped in Webots (`Terminal 1`):

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch webots_ros2_turtlebot dynamic_robot_launch.py
  world:=$HOME/ros2_webots_ws/src/webots_ros2/
webots_ros2_turtlebot/worlds/<WORLD_FILENAME.wbt>
```

2) Launch the Cartographer package that pops up an Rviz window for visualization (`Terminal 2`):

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_cartographer cartographer.launch.py
  use_sim_time:=True
```

3) Lauch the `teleop_keyboard` node, which allows to send velocity commands to the Turtlebot through the keyboard. Then, start navigating the whole environment until it is fully covered and the acquired map is satisfactory shown in Rviz (`Terminal 3`):

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 run turtlebot3_teleop teleop_keyboard
```

4) Finally save the map as a `.pgm` file in the target folder using the `nav2_map_server` package (`Terminal 4`):

```
$ ros2 run nav2_map_server map_saver_cli -f
  ~/ros2_webots_ws/src/webots_ros2/webots_ros2_turtlebot/resource/
<MAP_FILENAME.yaml>
```

5.3 Simulation tests

In principle, the goal of this thesis is to perform a preliminary evaluation of the proposed method, comparing it to the current solutions. A substantial performance evaluation of the dynamic obstacle layer approach would require several configurations and different simulation environments, since the parameters involved are more than a hundred; however, this is out of the scope of this work, which aims at laying the base for LINKS future improvements. According to these considerations, the tests proposed here are for comparing the behavior of the TurtleBot3 robot in presence of dynamic obstacles implementing the default DWB

Controller, with the DWB Controller integrated with the developed Dynamic Obstacle Layer (DOL).

In particular, the virtual environment `dyn_env_1` has been used. Here, the Turtlebot3 is commanded to navigate from one side of the rectangular arena to the opposite side, covering a total distance of $8\,m$ (Fig. 5.4). The parameters recorded for the evaluation purpose are: the travel time, the number of *wait* recovery behaviors triggered during travel and if any collision occurred. Data have been collected in two test cases: obstacles (boxes) constant speed set to $0.6\,m/s^2$ (Test set 1) and $0.8\,m/s^2$ (Test set 2). The results are commented in the following two subsections.

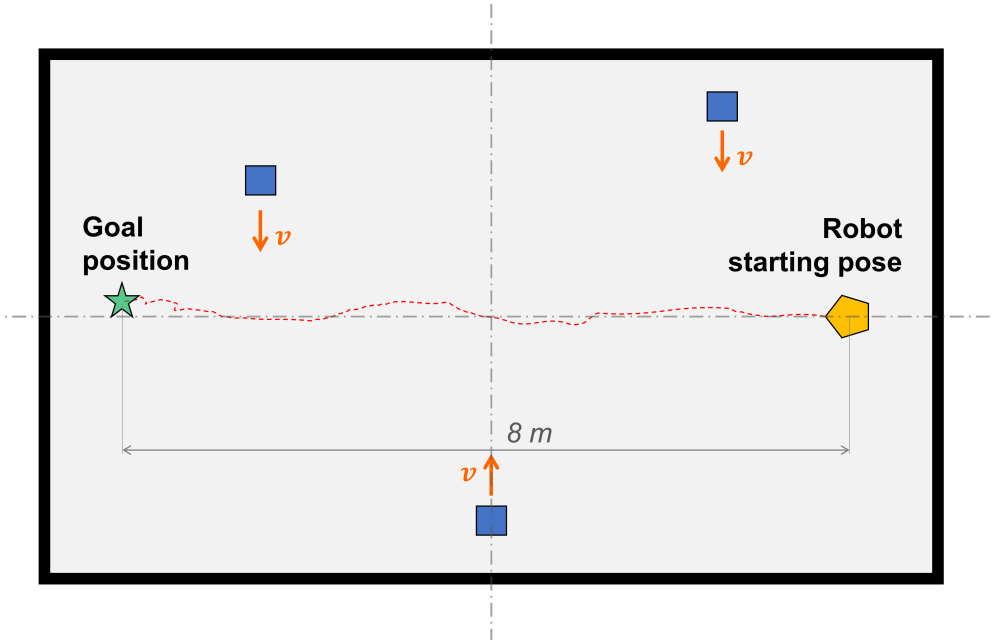


Figure 5.4: `dyn_env_1.wbt` test scheme.

5.3.1 Test set 1 – $0.6\,m/s^2$

For the first test set, $N_1 = 50$ simulations have been launched for both DWB and DWB + DOL configurations. Table 5.1 sums up the global navigation results, where:

- Smooth navigation – is the percentage of total navigations launched during which the TurtleBot3 smoothly navigated until reaching the goal position. At most, the robot has reduced its speed to $0\,m/s^2$ for few milliseconds in order to avoid collision with a moving box in its proximity.
- Wait recoveries – is the percentage of total navigations during which the *wait* recovery behavior has been triggered, still successfully reaching the

goal. *Wait* recovery is usually triggered when an obstacles comes suddenly too close and the Controller Server cannot find an affordable path within a certain timeout parameter, then the Recovery Server is triggered.

- Collisions – is the percentage of total navigations launched that were considered unsuccessful because the TurtleBot3 collided with a moving box. In some cases, one of the boxes collided with the robot dragging it with it for a long distance, thus messing up the TurtleBot localization system. In some other cases, collisions were less severe and, although its localization was partially compromised, the robot managed to reach the goal, thanks to effectiveness of the AMCL[1].

	DWB + DOL	**DWB**
Smooth navigation	86,0%	82,0%
Wait recoveries	10,0%	0,0%
Collisions	4,0%	18,0%
Total successful navigations	**96,0%**	**82,0%**

Table 5.1: Navigation results at $0.6\,m/s^2$ obstacles speed.

As it is also shown in Fig. 5.5, the proposed method combined with DWB reported a greater successful navigation rate than simple DWB: 96% against 82% respectively. In particular, the 'smooth navigation' percentage is quite similar between the two approaches, but the DWB+DOL solution reported fewer collisions because the *wait* recovery was triggered more times. This means that the dynamic obstacle layer provides a safer navigation if combined with the actual DWB planner. Indeed, the Gaussian costs forewarn the Turtlebot about an approaching obstacle and the Recovery Server is triggered on time if moving on would result in a collision. On the other hand, with the chosen obstacle speed ($0.6\,m/s^2$) and the same Nav2 parameters settings, DWB does not react on time if an obstacle suddenly approaches.

Concerning the 'smooth navigation' cases, though percentage over the total number of tests is similar – 86% DWB+DOL, 82% DWB – travel time performances are different. Figure 5.6 shows the box plots representing the travel time data of all the 'smooth navigation' cases for the two approaches. First of all, it can be noticed that the DWB distribution is more asymmetric and for sure non-Gaussian. Secondly, the mean travel time reported in DWB+DOL is lower than that of DWB, even if only of $1\,s$. However, the most important result is that the interquartile

[1]Adaptive Monte Carlo Localization embedded in Nav2 through `nav2_amcl` package

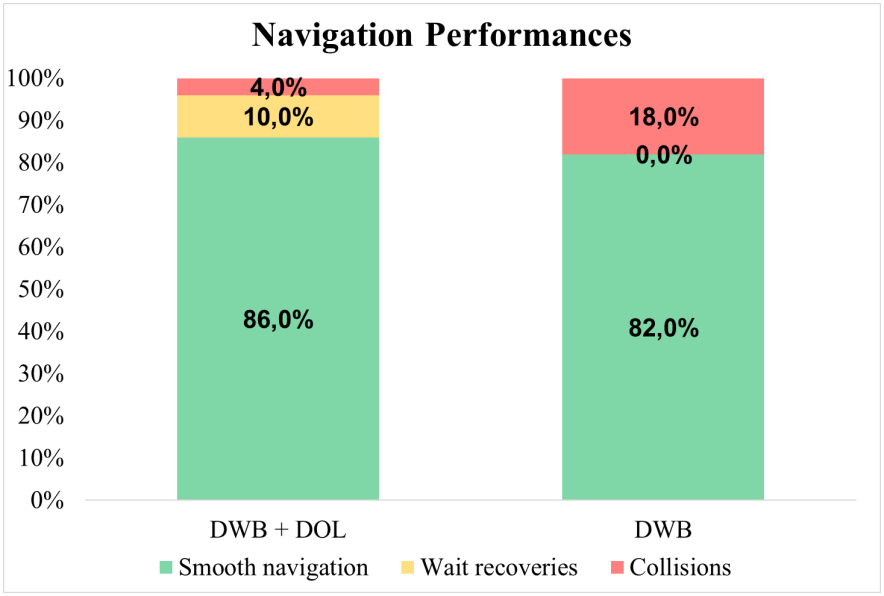


Figure 5.5: Test set 1 – global navigation performances.

(IQR) range for DWB+DOL is smaller than DWB box. Thus, it seems that the proposed method ensures to estimate a more confident travel time for a given environment and settings. Finally, it has to be noticed that outliers lays all below the box plot for the DWB+DOL (shorter travel times), while they are all longer travel times for the DWB plot. This suggests that carrying out more tests might produce less overlapped box plots and so more accurate considerations. Table 5.2 collects some statistical data about the box plots.

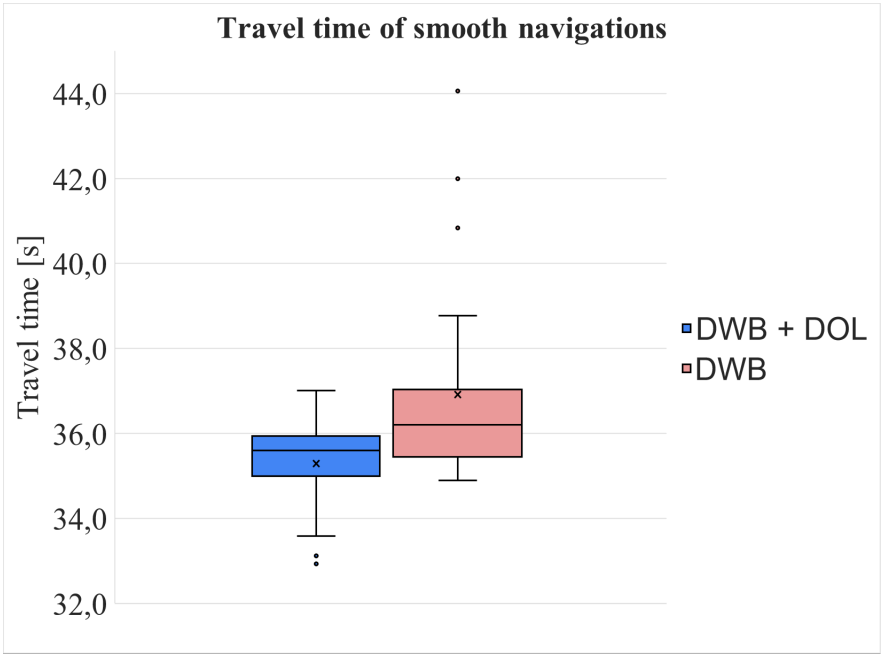


Figure 5.6: Test set 1 – Box plot of travel times for 'smooth navigation' test cases.

	DWB + DOL	**DWB**
Standard deviation	1,064	2,558
Mean value [s]	35,29127	36,91798
Median value [s]	35,59976	36,20260
Max travel time [s]	37,00594	38,76978
Min travel time [s]	33,58855	34,89583

Table 5.2: Test set 1 – box plot statistical data for 'smooth navigation' test cases.

5.3.2 Test set 2 – $0.8\,m/s^2$

For the second set of data, the obstacle speeds have been set to $0.8\,m/s^2$. This value has been chosen in order to push the available DWB Controller to its limits. Indeed, in this case, only $N_2 = 30$ simulations are sufficient to clearly prove the poor performances of the DWB with respect to the DWB+DOL approach. Actually, the only difference in the settings and parameters of the whole environment – and the Nav2 parameters – with respect to the previous test set is the obstacle speed. The same metrics have been taken into consideration and the results are shown in Figure 5.7 and Table 5.3.

	DWB + DOL	**DWB**
Smooth navigation	50,0%	43,3%
Wait recoveries	36,7%	0,0%
Collisions	13,3%	56,7%
Total successful navigations	**86,7%**	**43,3%**

Table 5.3: Navigation results at $0.8\,m/s^2$ obstacles speed.

It can be noticed that the number of collisions during the simulations launched with only DWB have remarkably increased (from 18,0% at $0.6\,m/s^2$ to 56,7%). This has happened also for the DWB+DOL (from 4,0% at $0.6\,m/s^2$ to 13,3%), but the percentage of triggered recoveries has increased as well, ensuring 86,7% of successful navigations, while the percentage of success in case of DWB has halved wrt Test set 1. This confirms the effectiveness of the proposed approach, at least in terms of navigation safety.

On the other hand, travel time data have been collected as well. However, starting from $N_2 = 30$ data points and considering that the 'smooth navigation' cases are 50% (DWB+DOL) and 43,3% (DWB), it is not possible to make robust consideration. Anyway, Figure 5.8 shows the box plot for the Test set 2. Here, data relative to the $0.8\,m/s^2$ are displayed in matt colours (blue tone for DWB+DOL and red tone for DWB), then data from the previous test set are displayed as well,

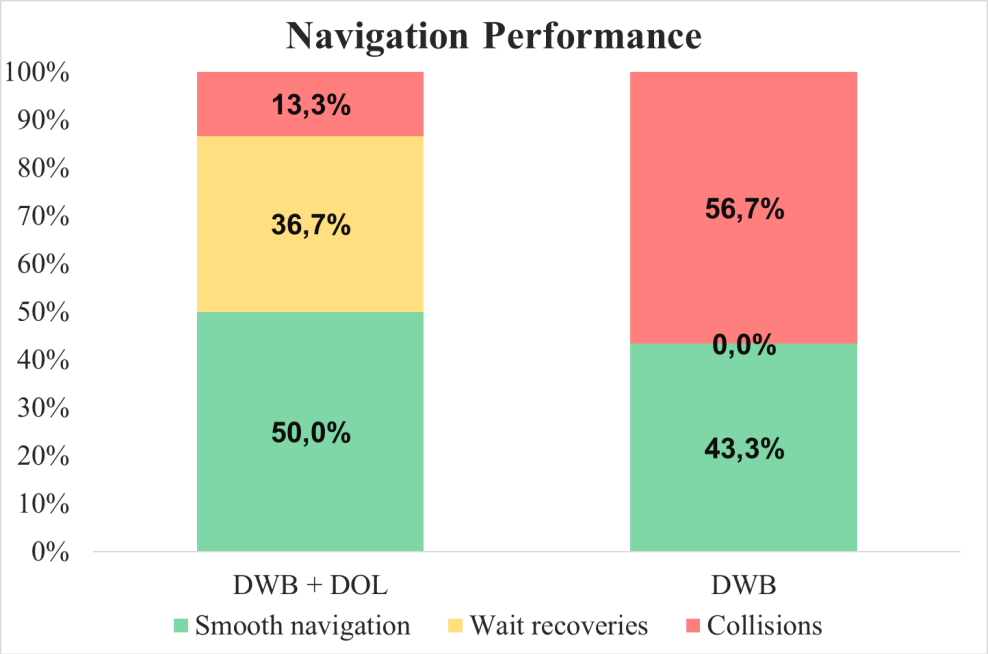


Figure 5.7: Test set 2 – global navigation performances.

using the same but transparent colours.

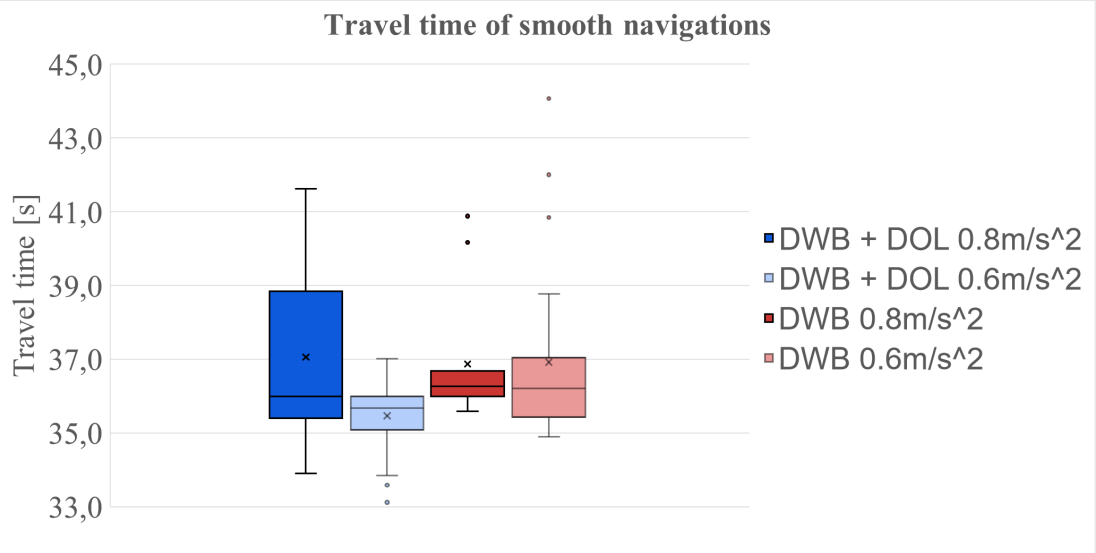


Figure 5.8: Test set 2 – Box plot of travel times for 'smooth navigation' test cases.

It can be noticed that median values are very similar to the ones of Test set 1. The first remarkable difference is that in Test set 2, due to the small amount of data, the DWB+DOL box plot has a greater variance. This is also because the faster are the obstacles, the more corrective actions are performed by the robot, making the travel time more unpredictable. Concerning the data for DWB, the variance has considerably reduced wrt the previous test set. However, the data sample is too small (13 'smooth navigation' cases) and 'smooth navi-

gation'(s) are reported only when the robot starts the navigation at a random 'lucky' instant, for which the planned path is not traversed by any of the moving boxes when the robot passes through the crossing point with an obstacle trajectory.

As it is shown by simulation results, the DOL approach definitely reports some performances improvements in terms of collisions rate, but still collisions occur, even at the lower obstacle speed of $0.6\,m/s^2$. One reason behind this could be the not optimal communication between the Webots virtual environment and the Navigation Stack, that may cause delays in the detection and costs computation. However, this is strictly related to the computational performances of the whole ROS2 + Linux environment installed on the Intel NUC platform. Despite this, some of the following Nav2 parameters could be tuned to reduce the collisions rate with the adopted HW/SW setup. Referring to the *Configuration Guide* section of DWB Controller in the Nav2 documentation page[2]:

- `controller_frequency` – default value `20 Hz` – Increasing the frequency at which the Controller Server runs, the local trajectory is re-planned faster by the DWB Controller plugin. So, the robot should react faster to an approaching obstacle, preventing collision.

- `update_frequency` – default value `5 Hz` – Frequency of update of the `local_costmap`. Increasing the update frequency of the costmap the robot would read more recent Gaussian cost values, thus, basing local path planning calculation on more reliable data would certainly help in a faster reaction to obstacles.

- `<dwb plugin>.sim_time:` – default value `1.7 s` – It is the time the DWB plugin simulates ahead by to generate the affordable local trajectories before scoring them and choosing the best one. Obviously, slightly increasing this parameter, the DWB Controller should better discriminate colliding trajectories from non-colliding ones. However, DWB is still a local planner, so it does not make sense to increase it too much, as it would end up doing the job of the Planner Server.

- `<dwb plugin>.BaseObstacle.scale:` – default value `0.02` – It is the scale according to which the DWB plugin scores a trajectory based on where the path passes over the costmap. Increasing this scale, the robot should take into more consideration of not passing through cells with a moderated cost (typically the ones resulting from inflation). This should also produce longer

[2] `https://navigation.ros.org/configuration/index.html`

but smother navigation, since the robot is encouraged to keep away from inflation regions.

Increasing the above listed parameters, especially the first three, may seem an obvious option to improve the performances. However, this would reflect in a greater computational demand. Therefore, their values should be chosen and fine tuned according to the desired hardware performances of final target hardware that will be installed on the physical Turtlebot.

A last comment is reserved to the Gaussian cost assignment. Here, the Gaussian shape parameters, i.e., the variances, have been set on the base of intuition and on an hypothetical obstacle maximum speed (recall eq. 3.15). In contrast, an additional function could be introduced to compute the costs combining the obstacle speeds with the robot speed. For instance, if the robot maximum speed is too small with respect to the obstacle one, it should not try to pass over it, so Gaussian costs should be modulated accordingly. On the other hand, an idea could be including the obstacle speed information directly into the local planner, however this approach results much more complex and it would not be as modular as the proposed DOL.

In conclusion, the proposed Dynamic Obstacle Layer approach seems to provide a safer navigation in presence of dynamic obstacles. At the same time, in the majority of the test cases, the DOL allows to plan a smoother trajectory than the DWB Controller alone, resulting in reduced travel times for equal starting conditions. Indeed, despite the Turtlebot has a maximum speed lower than the set obstacle speeds, it manages to adjust the trajectory when an obstacle is reported by the DOL, being able to dodge it or passing behind it. However, travel time performances have a lot of margins of improvement.

Conclusions and future developments

Autonomous vehicles navigation is a vast topic. The initial state of the art research has been useful to define the problem in the context of indoor navigation, providing a theoretical base for the LINKS project development. However, between theory and practice there is a huge gap. This work has tried to bridge this divide through the development of an implementable solution within the ROS2 framework, which has put not few constraints to the design. The proposed Dynamic Obstacle Layer approach implements a strategy for dynamic obstacle handling that can be easily integrated with the current ROS2 Navigation Stack, thus being a flexible and modular solution for the problem. The simulation tests carried out in a dynamic virtual environment have shown a relevant performance improvement in terms of collisions rate and travel times, integrating the DOL with the available DWB Controller. Therefore, it can be concluded that the thesis goals have been reached. Nevertheless, the DOL code involves many parameters (e.g. filters parameters, blob detection parameters, Gaussian costs scaling factors, etc.) some of which have been set based on intuition and manual tuning.

Being the first building block of a wider LINKS project, this thesis represents the starting point of many possible future works in different directions.

First of all, the DOL has been tested only in a virtual environment. Even though Webots has been set to provide as realistic simulations as possible, only tests in real world can validate the proposed approach, using a physical Turtlebot with real sensors and, above all, having a direct eye feedback about the global robot behavior.

Beyond that, object detection is based on the thresholds set for the running average filter. The chosen values have been manually tuned on the base of the examples reported in [52], however, the filtering is not 100% accurate for low obstacle velocities. Thus, a set of tests should be carried out to finely tune these parameters according to the hypothetical future application scenarios.

Furthermore, the original aim of this work was to integrate the DOL not only with DWB, but also with the TEB Controller. However, even if developers announced the porting of `teb_local_planner` in ROS2, in truth the package still does not install correctly upon the current Nav2 stack. Indeed, several issues have been raised in the related Github pages, but the community has not found a solution yet. So, after unsuccessfully trying to solve the problem on my own with the support of LINKS, the integration of DOL with TEB was discarded.

Finally, the incorporation of camera information can be studied to provide a categorical information about dynamic obstacles, e.g., discriminating a walking person from another robot, so that the robot could react accordingly with different planning strategies. Categorical information can be easily integrated in a multy-layer costmap on the base of the current DOL, and Planner and Controller plugins can be easily foreseen to implement different behaviors.

All these points might inspire future works at LINKS to carry on the development of the automated mail delivery system project.

Bibliography

[1] U. Jahn, D. Heß, M. Stampa, A. Sutorma, C. Röhrig, P. Schulz, and C. Wolff, "A taxonomy for mobile robots: Types, applications, capabilities, implementations, requirements, and challenges," *Robotics (Basel)*, vol. 9, no. 109, p. 109, 2020.

[2] R. Arkin and R. Murphy, "Autonomous navigation in a manufacturing environment," *IEEE transactions on robotics and automation*, vol. 6, no. 4, pp. 445–454, 1990.

[3] S. Angerer, R. Pooley, and R. Aylett, "Mobcomm: Using bdi-agents for the reconfiguration of mobile commissioning robots," in *2010 IEEE International Conference on Automation Science and Engineering*, pp. 822–827, 2010.

[4] M. Simon, "This incredible hospital robot is saving lives. also, i hate it," October 2015. `https://www.wired.com/2015/02/incredible-hospital-robot-saving-lives-also-hate/`.

[5] "Smp robotics systems corp. :." `https://smprobotics.com/products_autonomous_ugv/security-patrol-robot/`.

[6] "Kidd creek mine use case, boston dynamics:." `https://www.bostondynamics.com/spot/resources/kidd-creek-mine`.

[7] F. Rubio, F. Valero, and C. Llopis-Albert, "A review of mobile robots: Concepts, methods, theoretical framework, and applications," *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, p. 172988141983959, 2019.

[8] S.-H. Joo, S. Manzoor, Y. G. Rocha, S.-H. Bae, K.-H. Lee, T.-Y. Kuc, and M. Kim, "Autonomous navigation framework for intelligent robots based on a semantic environment modeling," *Applied Sciences*, vol. 10, no. 9, 2020.

[9] "Self-driving cars with ros and autoware." Online course available at `https://www.autoware.org/awf-course`.

[10] S. M. LaValle, *Planning Algorithms*, ch. Introductory Material. USA: Cambridge University Press, 2006.

[11] L. Claussmann, M. Revilloud, D. Gruyer, and S. Glaser, "A review of motion planning for highway autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, pp. 1826–1848, May 2020.

[12] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[13] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE robotics & automation magazine*, vol. 4, no. 1, pp. 23–33, 1997.

[14] D. T. Kuan, R. A. Brooks, J. C. Zamiska, and M. Das, "Automatic path planning for a mobile robot using a mixed representation of free space," *IEEE Computer Society Conference on Artificial Intelligence Applications*, 1984.

[15] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE transactions on robotics and automation*, vol. 7, no. 3, pp. 278–288, 1991.

[16] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on systems science and cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[18] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep. TR 98-11, Computer Science Department, Iowa State University, IA, USA, 1998. Available at: `http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf`.

[19] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[20] E. Rimon and D. Koditschek, "Exact robot navigation using artificial potential functions," *IEEE transactions on robotics and automation*, vol. 8, no. 5, pp. 501–518, 1992.

[21] D. A. de Lima and G. A. S. Pereira, "Navigation of an autonomous car using vector fields and the dynamic window approach," *Journal of control, automation & electrical systems*, vol. 24, no. 1, pp. 106–116, 2013.

[22] S. Quinlan and O. Khatib, "Elastic bands: connecting path planning and control," in *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pp. 802–807 vol.2, IEEE Comput. Soc. Press, 1993.

[23] M. Wolf and J. Burdick, "Artificial potential functions for highway driving with collision avoidance," in *2008 IEEE International Conference on Robotics and Automation*, pp. 3731–3736, IEEE, 2008.

[24] J. Nilsson, J. Fredriksson, and E. Coelingh, "Rule-based highway maneuver intention recognition," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pp. 950–955, IEEE, 2015.

[25] D. Iberraken, L. Adouane, and D. Denis, "Multi-level bayesian decision-making for safe and flexible autonomous navigation in highway environment," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3984–3990, IEEE, 2018.

[26] D. Mehta, G. Ferrer, and E. Olson, "Autonomous navigation in dynamic social environments using multi-policy decision making," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 2016-, pp. 1190–1197, IEEE, 2016.

[27] L. da Silva Assis, A. da Silva Soares, C. J. Coelho, and J. Van Baalen, "An evolutionary algorithm for autonomous robot navigation," *Procedia Computer Science*, vol. 80, pp. 2261–2265, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[28] H. Omrane, M. S. Masmoudi, and M. Masmoudi, "Fuzzy logic based control for autonomous mobile robot navigation," *Computational intelligence and neuroscience*, vol. 2016, pp. 9548482–10, 2016.

[29] C. Li, J. Zhang, and Y. Li, "Application of artificial neural network based on q-learning for mobile robot path planning," in *2006 IEEE International Conference on Information Acquisition*, pp. 978–982, IEEE, 2006.

[30] S. Lefèvre, D. Vasquez, and C. Laugier, "A survey on motion prediction and risk assessment for intelligent vehicles," *ROBOMECH journal*, vol. 1, no. 1, pp. 1–14, 2014.

[31] C. Chen, A. Gaschler, M. Rickert, and A. Knoll, "Task planning for highly automated driving," in *2015 IEEE Intelligent Vehicles Symposium (IV)*, pp. 940–945, IEEE, 2015.

[32] M. G. Plessen, P. F. Lima, J. Martensson, A. Bemporad, and B. Wahlberg, "Trajectory planning under vehicle dimension constraints using sequential linear programming," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, IEEE International Conference on Intelligent Transportation Systems-ITSC, pp. 1–6, IEEE, 2017.

[33] X. Qian, F. Altche, P. Bender, C. Stiller, and A. de La Fortelle, "Optimal trajectory planning for autonomous driving integrating logical constraints: An miqp perspective," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 205–210, IEEE, 2016.

[34] P. F. Lima, M. Trincavelli, J. Mårtensson, and B. Wahlberg, "Clothoid-based model predictive control for autonomous driving," in *2015 European Control Conference (ECC)*, pp. 2983–2990, 2015.

[35] J. Choi, "Kinodynamic motion planning for autonomous vehicles," *International journal of advanced robotic systems*, vol. 11, no. 6, p. 90, 2014.

[36] P. Yoonseok, C. Hancheol, J. Leon, and L. Darby, *ROS Robot Programming (English)*. ROBOTIS, 12 2017.

[37] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *2016 International Conference on Embedded Software (EMSOFT)*, pp. 1–10, 2016.

[38] "Why ros 2?," June 2014. Available at `https://design.ros2.org/articles/why_ros2.html`.

[39] G. Pardo-Castellote, "Omg data-distribution service: architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings*, pp. 200–206, IEEE, 2003.

[40] G. Biggs and T. Foote, "Managed nodes." Available at `https://design.ros2.org/articles/node_lifecycle.html` (accessed on 27/05/2021).

[41] E. Marder-Eppstein, "`move_base` package docs." Available at: `http://wiki.ros.org/move_base` (accessed on 28/05/2021).

[42] T. Foote, "tf: The transform library," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pp. 1–6, April 2013.

[43] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA99)*, May 1999.

[44] M. Colledanchise and P. Ögren, "Behavior trees in robotics and ai," Jul 2018.

[45] "Behaviortree.cpp library." `https://www.behaviortree.dev` (accessed on 29/05/2021).

[46] `https://github.com/ros-planning/navigation2/tree/main/nav2_dwb_controller` (accessed on 29/05/2021).

[47] D. H. Eitan Marder-Eppstein, David V. Lu!!, "`teb_local_planner`." `https://github.com/rst-tu-dortmund/teb_local_planner` (accessed on 29/05/2021).

[48] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 709–715, IEEE, 2014.

[49] "`costmap_2d` – ros wiki." `http://wiki.ros.org/costmap_2d`.

[50] "Robotis co.:." `http://en.robotis.com/`.

[51] S. Macenski, F. Martin, R. White, and J. G. Clavero, "The marathon 2: A navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2718–2725, IEEE, 2020.

[52] F. Albers, C. Rösmann, F. Hoffmann, and T. Bertram, *Online Trajectory Optimization and Navigation in Dynamic Environments in ROS*, pp. 241–274. 01 2019.

[53] B. Cybulski, A. Wegierska, and G. Granosik, "Accuracy comparison of navigation local planners on ros-based mobile robot," in *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*, pp. 104–111, 2019.

[54] H. Dong, C.-Y. Weng, C. Guo, H. Yu, and I.-M. Chen, "Real-time avoidance strategy of dynamic obstacles via half model-free detection and tracking with 2d lidar for mobile robots," *IEEE/ASME transactions on mechatronics*, vol. 26, no. 4, pp. 2215–2225, 2021.

[55] M. Przybyla, "Detection and tracking of 2d geometric obstacles from lrf data," pp. 135–141, 07 2017.

[56] Z. Yi and F. Liangzhong, "Moving object detection based on running average background and temporal difference," in *2010 IEEE International Conference on Intelligent Systems and Knowledge Engineering*, pp. 270–272, IEEE, 2010.

[57] "Blob detection using opencv." Available at `https://learnopencv.com/blob-detection-using-opencv-python-c/`.

[58] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[59] "Munkres' assignment algorithm modified for rectangular matrices." `https://brc2.com/the-algorithm-workshop/`.

[60] K. Saho, "Kalman filter for moving object tracking: Performance analysis and filter design," in *Kalman Filters* (G. L. de Oliveira Serra, ed.), ch. 12, Rijeka: IntechOpen, 2018.

[61] H. Laga and T. Amaoka, "Modeling the spatial behavior of virtual agents in groups for non-verbal communication in virtual worlds," in *Proceedings of the 3rd International Universal Communication Symposium*, IUCS '09, pp. 154–159, ACM, 2009.

[62] F. Yang and C. Peters, "Social-aware navigation in crowds with static and dynamic groups," in *2019 11th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games)*, pp. 1–4, 2019.

[63] "Universally unique identifier (uuid)." `https://en.wikipedia.org/wiki/Universally_unique_identifier`.

[64] "`cv::Mat` class reference." `https://docs.opencv.org/4.5.3/d3/d63/classcv_1_1Mat.html#details`.

[65] "`cv2.KalmanFilter` python class reference." `https://filterpy.readthedocs.io/en/latest/kalman/KalmanFilter.html`.

[66] "Webots simulator." `https://cyberbotics.com/`.

[67] "Rviz visualization tool." `https://index.ros.org/p/rviz2/`.

[68] "Gazebo simulator." `https://brc2.com/the-algorithm-workshop/`.

[69] Wikipedia contributors, "Simultaneous localization and mapping — Wikipedia, the free encyclopedia," 2021. [Online; accessed 16-November-2021].

[70] "Google open source: Cartographer." [Online; accessed 18-November-2021].

[71] "Cartographer – algorithm walkthrough for tuning." [Online; accessed 19-November-2021].