
UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING, ELECTRICAL ENGINEERING AND APPLIED MATHEMATICS

Project Report

# Mobile Robots for Critical Missions

prof. Saggese Alessia

*Group 03*

| Students | Id number |
|---|---|
| Alberti Andrea | 0622702370 |
| Capaldo Vincenzo | 0622702347 |
| Squitieri Giuseppe | 0622702339 |

Academic Year 2024 – 2025

# CONTENTS

# ABSTRACT

In this project, an advanced software system was developed and validated for the autonomous navigation of the TurtleBot4 mobile robot in indoor environments, leveraging the capabilities offered by the ROS 2 framework. The system allows the robot to move fully autonomously within a known map, planning and following in real time the shortest path between a starting point and a destination, while fully respecting kinematic constraints, safety requirements, and changing environmental conditions.

During the mission, the robot must be able to autonomously reach an assigned goal, passing through doors delimited by colored cones (always passing to the left of yellow cones and to the right of red ones), while also avoiding both static and dynamic obstacles that are not known in advance. The developed software architecture integrates modules for perception, localization, path planning, and navigation control, each implemented as an independent ROS 2 node that communicates with the others via custom topics and services.

Environmental perception is handled by an OAK-D PRO camera, capable of simultaneously acquiring RGB images and depth data, on which automatic cone detection is performed using a YOLOv8 neural network. The extracted 3D points are properly transformed into the global reference frame, filtered, and used to generate reliable intermediate waypoints that guide the robot along the optimal path.

A central role is played by a finite state machine (FSM) that governs the overall system behavior: this FSM allows the system to reactively manage abnormal situations such as the "kidnapped robot" (i.e., the robot being manually lifted and repositioned), to trigger automatic recovery maneuvers, and to maintain mission robustness even under unexpected conditions.

The entire system has been thoroughly tested, both at the level of individual modules and as an integrated pipeline, demonstrating high operational reliability and the ability to successfully tackle the main challenges typical of vision-based mobile robotics in complex real-world environments

# CHAPTER 1

# INTRODUCTION

The aim of this project is to design and implement a software system that allows to the **Turtlebot4 mobile robot** to autonomously navigate within an indoor environment giving the static map of the environment. The robot must be able to plan and execute the shortest possible path from any given starting position to a specified goal position, while satisfying a set of navigation constraints and reacting to dynamic environmental conditions.
The navigation task must be performed in compliance with the following constraints:

- the robot must always pass **to the left** of **yellow traffic cones**;
- the robot must always pass **to the right** of **orange traffic cones**;
- when cones are located along corridors or in close proximity, the robot must pass exactly **in between them**, respecting the side constraints mentioned above;
- the number and position of cones is not known a priori and may vary during operation;
- the environment may contain an arbitrary number of **unknown obstacles**, both **static and dynamic**, which the robot must detect and avoid in real time.

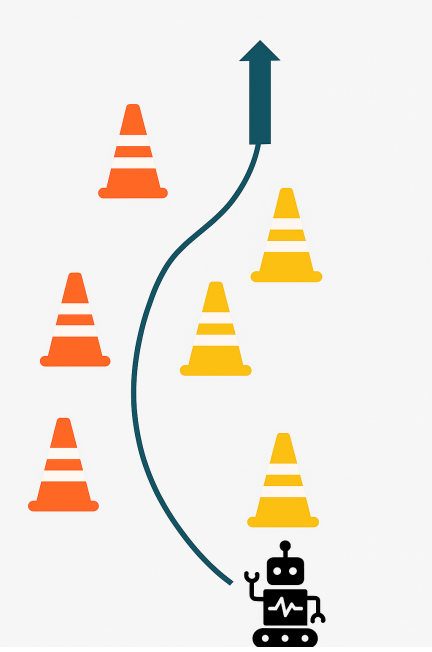

Figure 1.1: Example of robot's navigation in the environment

To accomplish this task, the robot must combine perception, localization, path planning, real-time obstacle avoidance, and constraint-aware motion planning. The software must ensure that the robot always complies with the cone-passing rules while dynamically adapting to changes in the environment.

Although the system is designed to work in any known environment, during the project activities the map used for development and testing was the one corresponding to the *DIEM indoor environment*.
The project was structured into the following core modules:

- **perception**: this module is responsible for acquiring and processing sensor data to detect and classify objects in the environment, such as traffic cones and obstacles, both static and dynamic;
- **localization**: uses sensory information to continuously and accurately estimate the robot's position and orientation within the environment map, supporting the correct operation of the other modules;
- **navigation**: integrates information from the other modules to plan and follow, in real time, an optimal path toward the goal, ensuring compliance with navigation constraints and safe obstacle avoidance.

## 1.1 Camera configuration

The first preliminary step for using the *OAK-D* camera was its **configuration and calibration**, in order to ensure accurate and synchronized acquisition of both RGB and depth data. The final parameters used were saved in the configuration file *conf/camera_conf.yaml*. This file includes a wide range of parameters that regulate the behavior of the acquisition pipeline, but among these, some key aspects stand out as fundamental for the correct functioning of the system:

- The parameter *camera.i_pipeline_type* was set to **RGBD**, enabling simultaneous output of color images and depth maps.
- The option *camera.i_enable_sync* was enabled to **synchronize the RGB and Depth streams**, a crucial condition to ensure correct spatial association between data from the two modalities.
- The **lazy publisher** mode was disabled on all relevant streams, forcing continuous publication of messages even when no active subscribers are present, which helps guarantee constant real-time data availability.
- Both cameras (RGB and stereo) were configured to operate at **10 fps**, a choice aimed at reducing computational load, considering that the Depth camera would not exceed this frame rate anyway.
- The image resolutions were set as follows:
  - **RGB**: (400, 400), sufficient to ensure proper detection.
  - **Stereo**: (1280, 720), to obtain **high-resolution depth**. A lower resolution would have increased the frame rate but at the cost of depth estimation accuracy.
- The parameter *rgb.i_keep_preview_aspect_ratio* was set to *false*, to guarantee a **correct pixel-to-pixel conversion** between the RGB image and the depth map during the 3D projection phase.
- For the stereo component, several crucial parameters were used to improve the quality of the depth map:
  - *stereo.i_depth_preset = HIGH_ACCURACY*, to prioritize accuracy over speed in depth map generation.
  - *stereo.i_disparity_width = DISPARITY_96*, which determines the **disparity field resolution** and consequently the maximum observable distance and depth resolution.

  - *stereo.i_subpixel: true*, to enable sub-pixel disparity estimation, improving depth resolution even at long distances.
  - *stereo.i_align_depth: true*, to obtain a depth map aligned with the RGB image, a fundamental requirement for accurate 3D projection of detected objects.

- Despite the availability of several optional filters offered by the DepthAI pipeline (spatial, temporal, and speckle filters), **none were enabled** because, although they sometimes improved depth map quality, they caused a **significant increase in processing time**, incompatible with the system's real-time requirements.

## 1.2 Path Configuration

In order to enable the robot to navigate correctly from a starting point (point A) to a destination point (point B), it must be provided with knowledge of both its **initial pose** and its **final pose** within the environment. A dedicated configuration file named *conf/path_config.txt* has been defined.
This file contains, on each line, the information describing the initial and final poses required for a specific navigation task. The format of each entry is as follows:

```
x_initial, y_initial, orientation
x_final, y_final, orientation
```

where:

- *x_initial*, *y_initial* denote the Cartesian coordinates of the robot's starting position;
- *x_final*, *y_final* denote the coordinates of the target destination;
- *orientation* specifies the robot's facing direction, expressed using the *TurtleBot4Directions* enumeration (e.g., *TurtleBot4Directions.SOUTH*).

This structure provides the navigation system with all the necessary information to properly initialize and execute the autonomous path planning and control procedures.

# CHAPTER 2

# ROS2 BASED ARCHITECTURE

This chapter presents the overall software architecture designed for the TurtleBot4 platform, built upon the ROS2 framework. The architecture is organized as a modular collection of nodes and services that collaborate to enable perception, waypoint computation, and navigation tasks. By leveraging ROS2's communication primitives and transformation capabilities, the system ensures efficient data flow and spatial consistency across all components. The design choices emphasize robustness, flexibility, and maintainability, facilitating real-time operation and future scalability.

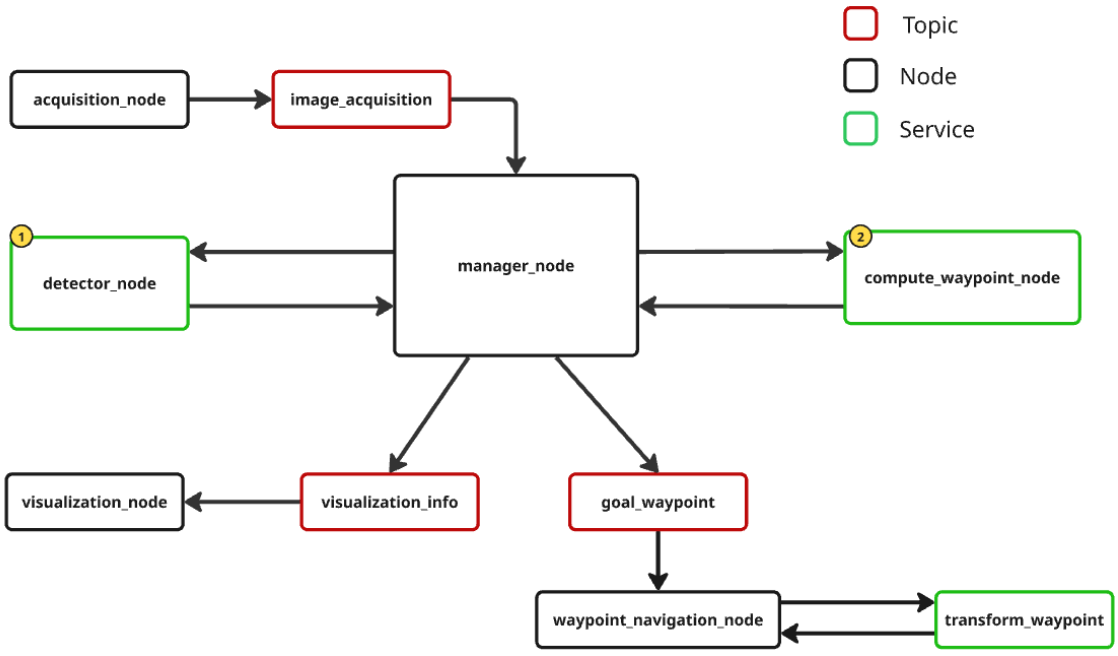


Figure 2.1: ROS2-based Architecture of the turtlebot4.

## 2.1 Architecture in details

- **Acquisition_node**: this node performs synchronized acquisition of RGB and Depth images from an OAK-D device. It leverages an approximate **timestamp-based synchronization** mechanism to pair RGB and Depth images, ensuring they are captured within a configurable maximum delay. Once synchronized, the images are published as a custom message on the *image_acquisition* topic. The use of timestamp-based synchronization ensures flexibility and robustness, particularly in scenarios where hardware introduces slight timing discrepancies between image streams (i.e. RGB and Depth frame).

- **Manager_node**: This node acts as a central manager that coordinates the processing of incoming images to detect cones and compute waypoints. It subscribes to *image_acquisition* topic, forwards the RGB frame to a cone detection service, and if detections are successful, it sends the Depth image and relative detections to the waypoint computation service. The results, including detections and computed waypoints, are then published on the *visualization_info* topic for the visualization modules and on the *goal_waypoint* topic for the navigation systems. The architecture separates responsibilities across services and topics to improve modularity and maintainability. By decoupling detection and waypoint computation into distinct services, the system remains flexible and allows for independent upgrades or replacements of these components. The use of asynchronous service calls with timeout mechanisms increases robustness, preventing the node from hanging due to unresponsive services.

- **Detector_node**: this node provides a service for detecting traffic cones in RGB images using a YOLOv8 deep learning model. The service receives an image through a custom service call, performs cone detection, and returns the list of detected cones along with their bounding boxes and estimated colors. YOLOv8 is selected for its real-time performance and accuracy, while dominant color estimation within each bounding box enables basic cone classification (e.g., red, yellow, blue cones).

- **Compute_waypoint_node**: this node exposes a service that receives a depth image and a list of detected colored objects, processes the data to identify left and right cones, computes their 3D positions relative to the camera frame, and calculates an optimal waypoint for navigation. The service publishes the waypoint coordinates along with supporting 2D visualization points directly in the service response. The system leverages a service-based architecture to compute waypoints on demand, rather than continuously publishing data. This reduces bandwidth consumption and ensures computations happen only when explicitly requested, which is advantageous for resource-constrained embedded systems. The system introduces virtual cones to maintain functionality when only one cone is visible, improving robustness. Depth estimation is extracted from a specific region of interest around each detection, optimizing precision while filtering noisy data.

- **Visualization_node**: This node is responsible for visualizing processed image data and detections in real time. It subscribes to the *visualization_info* topic where visual information is published, converts the incoming images to an OpenCV-compatible format, overlays bounding boxes around detected objects with color labels, draws specific reference points and lines, and displays the result in a window. The node is designed to run independently from detection and navigation components, focusing only on graphical visualization. This separation ensures that visualization tasks do not interfere with the performance or timing of critical perception or control processes.

- **Waypoint_navigation_node**: This node manages the navigation of the TurtleBot from a defined starting point to a final destination by controlling its movement. Its logic is implemented through a Finite State Machine (FSM), which governs transitions between different navigation phases to ensure robustness and adaptability. The node receives waypoint data from the *goal_waypoint* topic and processes this data by sending requests to the *transform_waypoint* service, which converts the received points—originally computed in the camera coordinate system—into a global reference frame suitable for navigation. Since the received waypoints may be noisy or imprecise, the node filters and averages them after transformation to compute a reliable target position. The robot is then commanded to move toward this refined waypoint. If no valid waypoints are currently available, the FSM can trigger a *look around* behavior, allowing the perception module to attempt detection of new cones. If no cones are detected during this maneuver, the robot proceeds to navigate directly toward the predefined final goal.

- **Transform_waypoint_node**: This node acts as a dedicated transformation service within the robot's software architecture. Its primary function is to convert 3D points — specifically waypoints detected by the robot's camera — into a global reference frame used for navigation and planning. Since sensor data is often obtained in the sensor's own local coordinate system (in this case, the camera frame), it is essential to translate these points into a common,

global coordinate frame (the map frame) to enable meaningful spatial reasoning. The node leverages ROS2's TF2 framework, which maintains a dynamic tree of coordinate frames and their relative transformations. This approach guarantees that all navigation-related components work with spatial data aligned to the same global frame, which is critical for path planning and movement control.

## 2.2 Custom messages

In this section, the custom ROS messages designed specifically for this project are presented. These messages define the data structures exchanged between the various nodes of the system, enabling the transmission of information such as images, detection results, waypoints, and visualization data. Each message is tailored to the specific needs of the perception and navigation pipeline.

### 2.2.1 ImageAcquisition.msg

The *ImageAcquisition* message contains two fields:

- **sensor_msgs/Image** rgb_image
- **sensor_msgs/Image** depth_image

This message is constructed by the acquisition node and is used by the manager node. Its primary purpose is to carry the images acquired from the camera — the RGB image for color information and the depth image for distance measurement — so they can be processed downstream for detection and waypoint computation.

### 2.2.2 ConeDetection.msg

The *ConeDetection* message contains two fields:

- **int32**[**4**] bbox
- **string** color

This message is used as part of the response from the detector service. It conveys the bounding box coordinates (x1, y1, x2, y2) of a detected cone along with its color classification. The detector node constructs and returns this message to provide detailed information about each detected cone to the client requesting the detection.

### 2.2.3 PixelPoint.msg

The *PixelPoint* message contains two fields:

- **int32** x
- **int32** y

This message is used as part of the response from the *ComputeWaypoint* service. It represents a specific point in pixel coordinates within the image, indicating the waypoint computed by the service for navigation or path planning purposes. These coordinate are useful for visualization purpose.

### 2.2.4 VisualInformation.msg

The *VisualInformation* message contains three fields:

- **sensor_msgs/Image** image
- **ConeDetection**[] boxes
- **PixelPoint**[] points

This message is used to provide visual data for visualization purposes. It carries the original image alongside detected cone bounding boxes and computed pixel points, allowing downstream nodes or interfaces to display detection results and navigation waypoints in a synchronized and coherent way.

### 2.2.5 Waypoint.msg

The *Waypoint* message contains two fields:

- **geometry_msgs/Point** point
- **float32** distance

This message is sent by the manager node to the navigator. It specifies the target waypoint that the vehicle should pass through, with the *point* field indicating the coordinates of the waypoint in the camera reference frame, and the *distance* field representing the Euclidean distance to that point.

## 2.3 Services

In this section, the custom ROS services implemented for this project are described. These services define the request-response mechanisms through which the system's nodes interact to perform cone detection, waypoint computation, and coordinate transformations. Each service is designed to handle a specific task within the perception and navigation pipeline, ensuring modularity and clear data flow between components.

### 2.3.1 DetectCones.srv

The *DetectCones* service consists of a request and a response message. The request contains a single field:

- **sensor_msgs/Image** rgb_image

which carries the RGB image to be processed for cone detection.

The response contains two fields:

- **bool** success
- **ConeDetection**[] detections

The *success* flag indicates whether the detection was successful, and *detections* is an array of detected cones with their bounding boxes and colors. This service is called by the manager node to obtain cone detections from an input image.

### 2.3.2 ComputeWaypoints.srv

The *ComputeWaypoint* service is responsible for calculating the optimal waypoint based on detected cones and depth information. The request contains two fields:

- **sensor_msgs/Image** depth_image
- **ConeDetection**[] detections

The *depth_image* provides depth information for the scene, while *detections* contains the list of cones previously detected by the detector node.

The response contains the following fields:

- **geometry_msgs/Point** waypoint
- **float32** distance
- **PixelPoint**[] points
- **bool** success

The field *waypoint* represents the computed point in the camera reference frame, where coordinates $(x, y, z)$ are expressed relative to the camera itself, which is considered the origin of the coordinate system. The field *distance* indicates the Euclidean distance to the computed waypoint and is intended to be used by the navigation system. The *points* field contains an array of auxiliary points used exclusively by the visualization tool to render visual elements on the graphical interface. These points are employed to display bounding boxes, the computed waypoint, and the direction of traversal. The *success* flag indicates whether the service was successfully performed.

### 2.3.3 PointTransformer.srv

The *PointTransformer* service is used to transform a point from one reference frame to another, in particular from the camera or frame to the map frame.

The request contains:

- **geometry_msgs/Point** point

This field represents the 3D point in the original reference frame that needs to be transformed.

The response contains:

- **geometry_msgs/Point** transformed_point
- **bool** success

The *transformed_point* represents the input point after being converted into the target reference frame. The *success* flag indicates whether the transformation was successfully performed.

## 2.4 Architecture evolution

The system design evolved through two main architectural approaches. The first attempt followed a linear pipeline model (Figure 2.2), where each node performed a specific task and passed data downstream via topics. In this setup, nodes were loosely coupled and interacted only with adjacent stages in the processing chain.

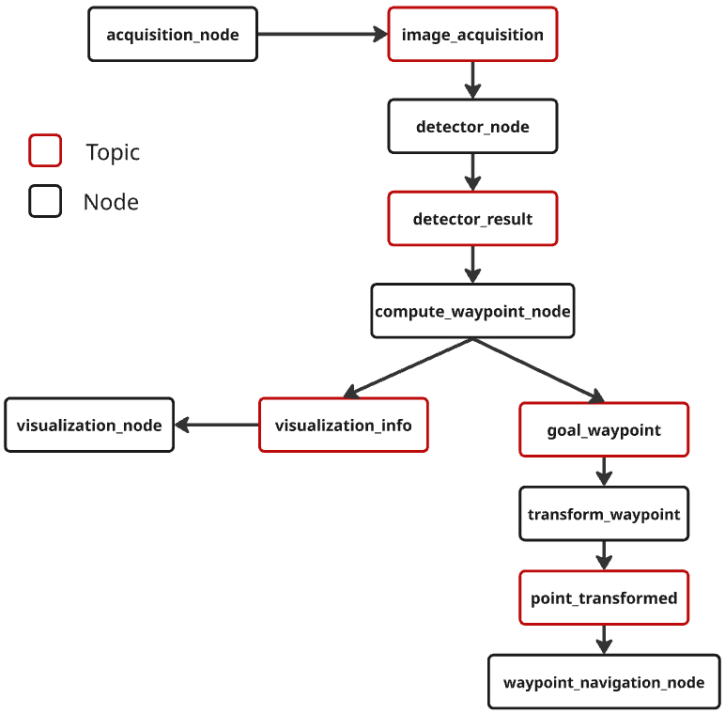


Figure 2.2: Architecture first stage.

This architecture offered simplicity and better scalability. Each node could be developed and tested independently, and adding new stages to the pipeline was straightforward. However, this design introduced a fundamental inefficiency in data handling: to keep information consistent across modules, it was necessary to forward the same data multiple times. For example, in order to maintain coherence between the detection result and the image it was computed from, the detector_node had to publish both the detection output and the original RGB and depth frames. This duplication of data had to be repeated at every step, increasing the bandwidth usage and leading to redundancy throughout the entire chain.

To address these limitations, we transitioned to a centralized architecture (Figure 2.1) based on a *manager_node*. This node acts as the core coordinator of the system, handling service calls, topic publications, and internal data management. While this design introduces more complexity—especially in the implementation of the manager logic—it significantly improves bandwidth efficiency and guarantees synchronization of data across modules without redundant forwarding.

This architectural shift reflects a trade-off between modular simplicity and centralized efficiency. In our case, the need to preserve data consistency and optimize communication overhead, to respect the real time constraint, justified the move toward the manager-based solution.

# CHAPTER 3

# PERCEPTION MODULE

The Perception Module is responsible for extracting meaningful information from raw sensor data to enable autonomous navigation and decision-making. It processes visual and depth information acquired by the OAK-D camera mounted on the TurtleBot to detect traffic cones, estimate their position, and compute navigation waypoints. The architecture follows a modular approach based on ROS 2 nodes and services, allowing for clear separation of responsibilities and scalability.
This chapter describes in detail each component of the Perception Module, outlining their functionality, interaction, and implementation.

## 3.1 Acquistion node

The *Acquisition_Synchronized* class is a ROS 2 node responsible for acquiring and synchronizing RGB and depth images from two distinct topics. Specifically, it subscribes to the RGB images on the topic */oakd/rgb/preview/image_raw* and depth images on */oakd/stereo/image_raw*. These subscriptions are handled using the *message_filters* library, which provides an *ApproximateTimeSynchronizer* to pair messages from the two topics based on their timestamps. The synchronizer allows a maximum time difference (slop) of 500 milliseconds between the RGB and depth images. This threshold was determined empirically; considering the TurtleBot's relatively slow movement speed, such temporal misalignment is tolerable and does not significantly impact the accuracy of sensor data fusion.

Once a synchronized pair of RGB and depth images is identified, the node combines them into a single custom message of type *ImageAcquisition*. This message encapsulates both image streams, allowing downstream nodes to receive perfectly matched RGB-depth pairs. The node publishes this custom message on the *image_acquisition* topic with a queue size of 10, facilitating further processing such as perception, mapping, or navigation.

## 3.2 Detection node

The *DetectorServiceNode* is a ROS 2 node that provides a service to detect traffic cones within RGB images. It receives requests containing RGB images and returns detected cones with bounding boxes and associated color labels. At initialization, the node loads a YOLOv8 model trained specifically for cone detection, which is used to perform inference on the input images.

When a detection request arrives, the node converts the ROS image message into an OpenCV format to prepare it for processing. The YOLOv8 model runs inference on this image, producing bounding boxes for detected cones along with confidence scores. For each detected bounding box, the node verifies that its coordinates are valid and fall within the image boundaries. The **confidence threshold** used to filter the detections returned by the YOLOv8 model is set to 0.5, meaning that only detections with a confidence score equal to or greater than 0.5 are considered valid and processed further. This confidence value can be adjusted depending on the desired balance between detection sensitivity and precision. The confidence threshold of 0.5 was chosen

because, during testing in the target environment, it provided the best balance between detecting as many cones as possible and minimizing false positives. Lowering the threshold would increase sensitivity and potentially detect more cones but at the risk of including more incorrect detections. Conversely, raising the threshold would reduce false positives but might cause some valid cones to be missed. Therefore, 0.5 represents a tradeoff that optimizes both detection reliability and completeness for the conditions experienced during development.

To determine the **cone's color**, the node analyzes the dominant color in the lower half of the bounding box rather than the entire box. This approach is chosen because the lower half of the bounding box typically contains more of the cone itself, while the upper half often includes background or irrelevant elements. By focusing on the bottom half, the color estimation is more accurate and less affected by background colors. The color extraction is performed by applying k-means clustering to the pixel colors within this region, then converting the dominant color from BGR to HSV space. Based on the hue value, the cone is classified as red, yellow, green, or blue. Each valid detection is packaged into a custom *ConeDetection* message containing the bounding box and the identified color. These detections are collected and returned in the service response, which also indicates success or failure to enable client-side error handling.

## 3.3 Manager node

The ManagerNode acts as a central coordinator that subscribes to an image stream from the topic *image_acquisition*. When it receives a new image message, it first checks if the detector service is currently busy processing a previous image. If so, it discards the new image to avoid overloading. Otherwise, it stores the image and initiates the detection process by calling asynchronously the *detect_cones* service, passing the RGB image extracted from the message.

This detection service returns information about detected cones, including their bounding boxes and colors. Once the detection response is received, if successful, the ManagerNode evaluates the result: if cones are detected and the waypoint computation service is free, it proceeds by forwarding the detections along with the corresponding depth image to the *compute_waypoint* service. This service calculates the next navigation waypoint based on the cone positions in the scene.

The calls to both services are asynchronous and protected by timeouts to avoid blocking the node indefinitely. If the detection service times out or fails, the node records the error and allows the next image to be processed. Similarly, the waypoint computation service response triggers publishing the calculated waypoint to the */goal_waypoint* topic and also updates visual information published on */visual_info*. If the waypoint calculation fails or times out, it logs the failure and clears the processing state to accept new requests.

## 3.4 Compute waypoint node

The ComputeWaypointNode acts as a dedicated ROS 2 service provider that combines visual detections and depth data to generate a reliable 3D waypoint for navigation. Once the compute_waypoint service is called, it first verifies that the request contains both a valid depth image and a non-empty list of cone detections. If either is missing, the node immediately responds with failure. The depth image is then converted to an OpenCV-compatible format.

Detections are separated into two categories based on color: red cones as left-side references, and yellow cones as right-side references. Other colors are ignored. For each bounding box, **the node calculates the average depth within a small, centered ROI** (Region Of Interest) that is scaled from the RGB image to match the stereo resolution (400x400 is the size of the RGB image and 1280x720 is the size of the depth image). It **filters out bounding boxes that are too large** (area larger than 15,000 pixels, when the cone is too close to the camera), **too small, or have invalid aspect ratios or depth values**. Depth values are also checked to be within a valid range, between 0.75 and 8 meters to have reliable values since this is the optimal work range of the depth camera.

If both left and right cones are detected, their closest instances are selected. If only one side has a valid cone, the node creates a virtual cone on the opposite side using a horizontal offset six times the cone's width. This heuristic allows the system to compute a midpoint far enough from the cone to avoid the collision with it when a single cone is visible.

For the target cones is necessary to compute the 3D coordinate from the camera reference system. To compute it we use the intrinsic parameters of the camera and the depth information at the center of the detected bounding boxes. The method follows a standard pinhole camera model and assumes the depth is known for the cone.
First, given the bounding box of the cone in the image, defined as:

$$\text{bbox} = (x_{min}, y_{min}, x_{max}, y_{max})$$

we compute the lower center of the bounding box in image coordinates:

$$u = \frac{x_{min} + x_{max}}{2} \qquad\qquad v = y_{max}$$

The intrinsic matrix **K** for the camera was obtained from the `/oakd/rgb/preview/camera_info` topic, and is as follows:

$$\mathbf{K} = \begin{bmatrix} 322.03 & 0 & 200 \\ 0 & 322.03 & 200 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix represents the intrinsic parameters of the RGB camera used by the system. In particular:

- The elements $K_{1,1}$ and $K_{2,2}$ (322.03) represent the focal length **fx** and **fy** expressed in pixels along the two main axes, i.e., the focal length multiplied by the pixel-to-distance ratio.
- $K_{1,3}$ (200) and $K_{2,3}$ (200) indicate the coordinates of the principal point, that is, the optical center of the camera on the image plane.
- The zeros and the one in the last row complete the representation in homogeneous coordinates.

Finally, given the depth value $d$ (expressed in meters) corresponding to the average depth in a dynamic area (based on the dimensions of the bounding box) around the center of the bounding box, the 3D coordinates in the camera frame are computed as:

$$X = d$$

$$Y = -\frac{(u - c_x) \cdot d}{f_x}$$

$$Z = -\frac{(v - c_y) \cdot d}{f_y}$$

The negative signs for $Y$ and $Z$ are introduced to respect the chosen coordinate system convention, to match the map's one, where:

- the $X$-axis points forward along the optical axis;
- the $Y$-axis points to the left from the camera's perspective;
- the $Z$-axis points upwards.

Successively, the 3D coordinate of the waypoint, it's computed the midpoint between left and right cone positions.

## 3.5 Visualization node

The *VisualizationNode* subscribes to the */visual_info* topic, which publishes messages of type *VisualInformation*.

When a message is received, the callback function converts the image from ROS format to an OpenCV-compatible format using the CvBridge utility. The image is then processed to draw the bounding boxes around detected objects. Each bounding box is drawn with a color that corresponds to the object's classification (for example, red, blue, yellow, green), and a label is

placed above the box to indicate the type of object—in this case, cones of different colors. If the color information doesn't match any of the predefined categories, the bounding box is drawn in white by default.

In addition to the bounding boxes, the code visualizes a set of 2D points received in the message. The first three valid points are drawn as small white circles, which represents the bottom-center of the bounding boxes and the middle point. A fourth point is computeed to visualize the direction considering the left and right cone (if cones are positioned in the correct position, the turtlebot should go in the forward direction, otherwhise in in the opposite direction). If both the first and second points are valid, a straight line is drawn connecting them. If the third and fourth points are valid, an arrow is drawn from the third to the fourth point.

All drawings are overlaid on the original RGB image, which is then displayed in a window using *OpenCV*. The window updates in real-time with each new image received.

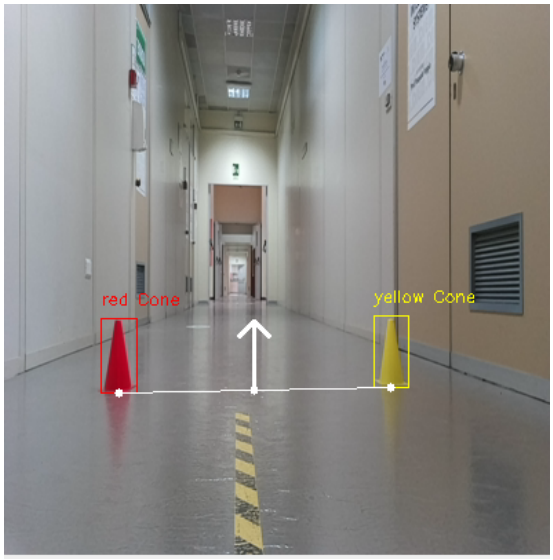


Figure 3.1: Example of visualization with two cones in the correct order.

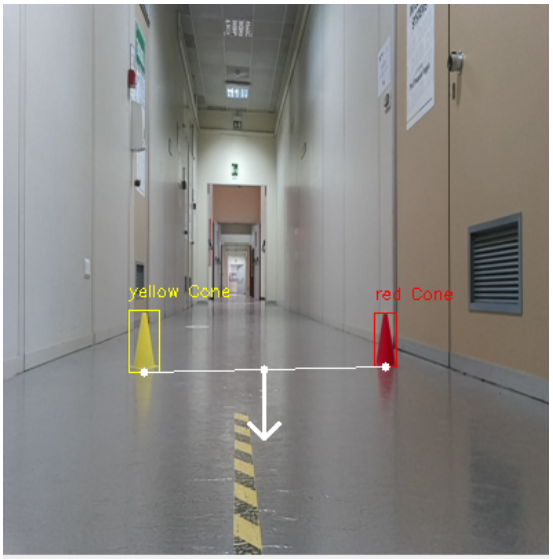


Figure 3.2: Example of visualization with two cones in the reversed order.

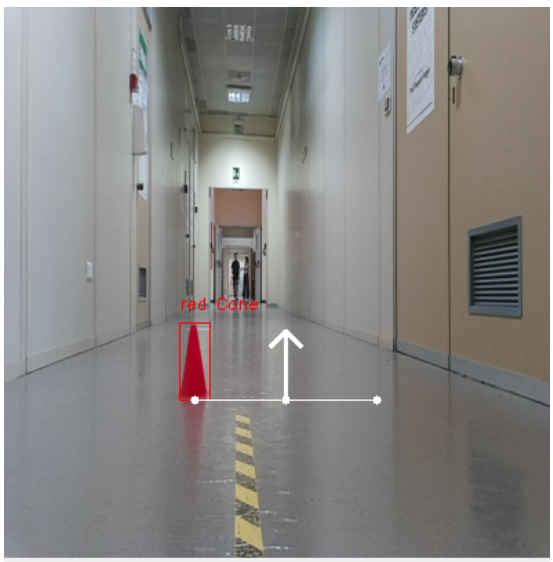


Figure 3.3: Example of visualization with only the left cone.

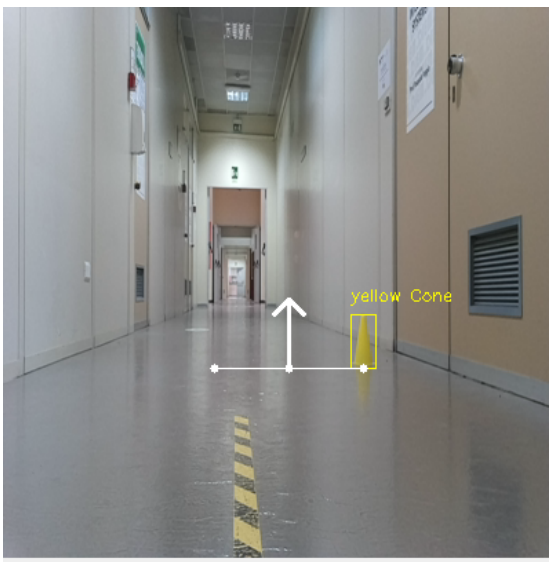


Figure 3.4: Example of visualization with only the right cone.

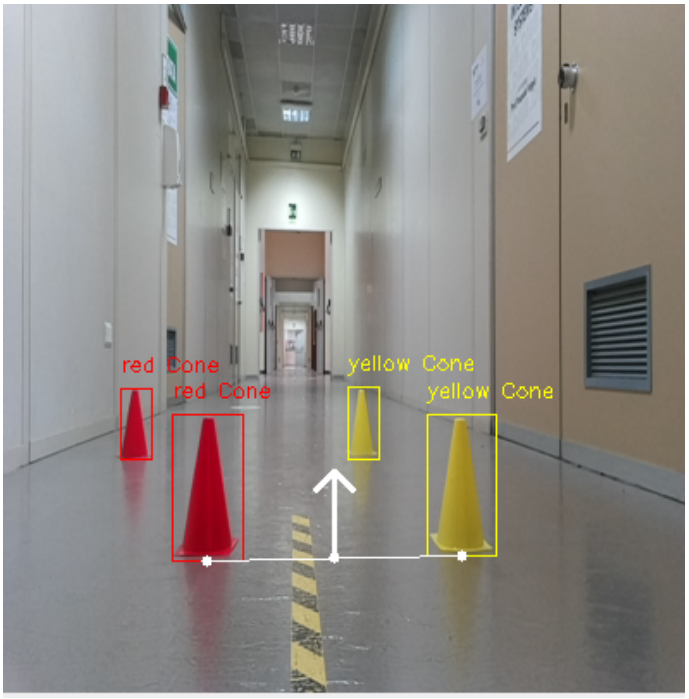


Figure 3.5: Example of visualization with two gates, where only the closest waypoint is shown.

# CHAPTER 4

# LOCALIZATION MODULE

The localization module is designed to enable the robot to determine its position within a known indoor environment. The module combines the laser scanner, odometry, and a 2D map of the environment to estimate the robot's current position within the global reference frame of the map.

Accurate localization is a fundamental prerequisite for the correct operation of the path planning and motion control modules. The module continuously provides an estimate of the robot's position in the map frame, which is used to:

- generate trajectories towards targets provided by other modules;
- avoid obstacles and navigate safely;
- update the position on the map for visualization purposes.

To this end, the **amcl** node *(Adaptive Monte Carlo Localization)* is launched, which implements a **particle filter-based probabilistic localization** algorithm. This algorithm maintains a distribution of hypotheses about the possible position of the robot, updating it as new sensor observations and motion information are received.

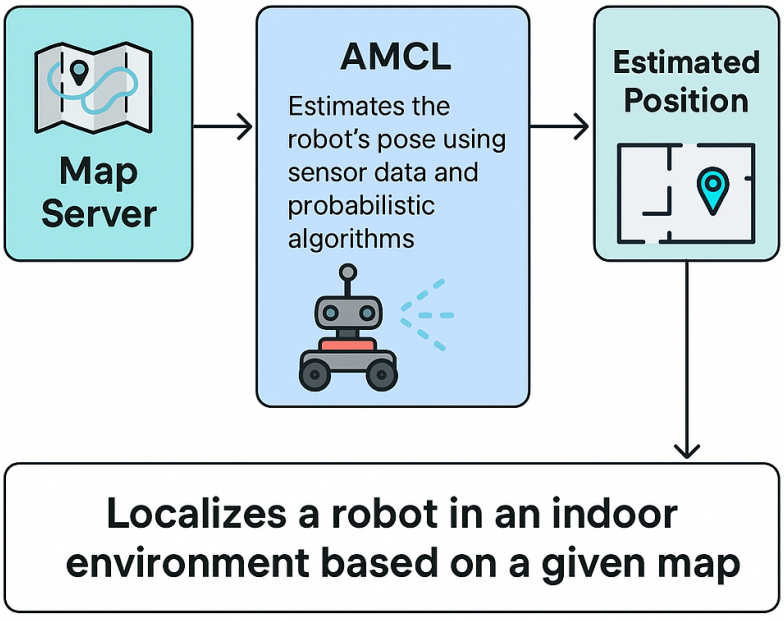


Figure 4.1: Localization module architecture.

## 4.1 Main Components

The entire process is launched through a ROS 2 launch file, which organizes the execution of the nodes. The module consists of three key components:

- **amcl**: manages the probabilistic localization of the robot using a particle filter;
- **map_server**: loads the static map of the environment, making it available to the navigation system. The file containing the map is *src/navigation_pkg/maps/diem_map.yaml*;
- **map_saver**: optionally allows saving a new map of the environment based on the robot's current perception. In this project, it was not used, as the map was provided beforehand.

## 4.2 Configuration Parameters

The localization parameters are defined in the file *src/navigation_pkg/config/localization.yaml*. In particular:

- the *amcl* section specifies the algorithm parameters, such as the number of particles, the laser sensor model, and the tolerances for frame broadcasting;
- the *map_server* section indicates the map file to be loaded;
- the *map_saver* section manages the parameters for saving the map.

### 4.2.1 Configuration of `amcl` Parameters

To achieve stable and accurate localization, careful tuning of the *amcl* node parameters was necessary. The final configuration is the result of several empirical tests conducted in a real environment, in different areas of the map. Below, the main parameters used and the rationale behind their selection are described.
The parameters `alpha1`–`alpha5` define the robot's motion noise model and affect how much the particles spread after a movement:

- `alpha1`, `alpha2`, `alpha3`, `alpha4`, `alpha5` = **0.2**, the default values previously present in the file were kept to represent a realistic uncertainty of the odometry. Values that are too low would reduce the filter's ability to adapt to errors, while values that are too high would increase instability.

The number of particles used directly affects both localization accuracy and computational load:

- `min_particles` = **500**, `max_particles` = **2000**, after numerous tests, these values proved to be a good compromise between performance and accuracy, especially in complex environments with similar geometries.

Regarding the use of the laser sensor:

- `max_beams` = **60**, uses 60 laser beams for each update. This value provides good spatial coverage without excessive computational load.

# CHAPTER 5

# NAVIGATION MODULE

The navigation module enables the robot to move autonomously and safely along a complete path, from the starting point (point A) to the destination (point B), by following a sequence of waypoints provided by the perception module. These waypoints represent the central points between detected cones and are used to build a dynamic trajectory that adapts in real time to the surrounding environment, avoiding obstacles and respecting the robot's kinematic constraints.

The module consists of three main components that cooperate to transform visually detected points into concrete navigation actions:

- the **path planning** system (Nav2), which forms the core of the navigation module. It handles both global and local path planning, motion control, and obstacle avoidance. It communicates with the localization module to maintain an up-to-date estimate of the robot's position within the map;
- the **point transformer node**, which receives 3D points expressed in the camera frame and transforms them into the global map frame using the TF2 transformation system;
- the **waypoint navigator node**, which manages the navigation logic. It receives the transformed waypoints and dynamically decides when and where to send new goals to the navigation system based on their arrival and distribution. This node acts as a **behavioral planner**, coordinating the robot's operational states to enable context-aware decision-making. In this way, the robot can robustly and adaptively react to dynamic scenarios, while delegating motion execution to Nav2.

These three components work in synergy as a complete pipeline: the perception system detects a point of interest, the *Point Transformer* converts it into the global reference frame, the *Waypoint Navigator* processes it and forwards it to *Nav2* as a goal, which is then tracked by the robot through the planning and control modules, all supported by the localization system.

## 5.1 Path Planning

The *path planning* module is responsible for computing an optimal trajectory that allows the robot to move safely and efficiently from the starting point to the desired destination, avoiding both static and dynamic obstacles in the environment. The planner must ensure that the trajectory is feasible according to the known map or a map updated in real time, while accounting for the robot's kinematic constraints, safety margins (e.g., avoiding proximity to walls), and the need to quickly react to environmental changes.

The planning and navigation process typically consists of several integrated layers, each responsible for a specific aspect of the robot's movement:

- **Global planning**: Computes the optimal path from the robot's current position to the goal on the global map, considering all known static obstacles. This level provides a high-level route that avoids impassable areas.

- **Local planning**: Continuously adapts the global path to local conditions in real time. The local planner generates velocity commands to the robot's actuators, adjusting the trajectory to avoid newly detected obstacles, deal with dynamic environments, and comply with kinematic constraints.
- **Behavioral layer**: Implements higher-level behaviors for navigation resilience and safety, such as automatic recovery from stuck situations (e.g., spinning, backing up, or waiting if a blockage is detected), following sequences of waypoints, and handling failure cases.
- **Velocity smoothing**: Ensures that the velocity commands sent to the robot are smooth and within the hardware's safe operating limits, filtering abrupt changes and respecting maximum acceleration and deceleration profiles.
- **Costmaps**: Maintain both global and local occupancy grids (costmaps) that represent the static map and the current perception of obstacles. These are constantly updated to reflect the robot's surroundings and are used by both planners for safe trajectory computation.

The navigation node leverages the Nav2 framework from ROS 2, which implements all these planning layers and provides a modular, configurable infrastructure for selecting and tuning the algorithms best suited to the robot and the operational scenario. This approach ensures robust, flexible navigation even in the presence of uncertainty, dynamic obstacles, or partial map information.

## 5.2 Global Planning Algorithm: Dijkstra vs A*

Within the Nav2 pipeline, the component responsible for global path planning can employ different algorithms. The two most commonly used are Dijkstra and A*, both widely validated in mobile robotics.

The configuration file *navigation.yaml* allows switching between them simply by modifying the *use_astar* parameter:

```
planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: true  # True = A*;  False=Dijkstra
      allow_unknown: true
```

**Technical Comparison**

- **Dijkstra** systematically explores the entire map to find the shortest path from the starting point to the goal, without any "intelligent" preference for direction.
  - *Pros*: always guarantees the shortest path; more robust in very noisy maps or in environments with highly variable costs.
  - *Cons*: can be very slow, especially on large or dense maps, as it evaluates all nodes with equal priority.
- **A*** is an extension of Dijkstra that adds a heuristic function (typically the Euclidean or Manhattan distance to the goal), which guides exploration toward the destination, avoiding unnecessary exploration of distant areas.
  - *Pros*: same level of optimality as Dijkstra, but much faster in most cases, especially on structured maps or when the goal is "easily reachable" without complex obstacles.
  - *Cons*: solution quality depends on the heuristic. In pathological cases, a poor heuristic can degrade performance, but in our case the standard heuristic is more than adequate.

Dijkstra is well-suited for small, highly complex maps or when pure robustness is the top priority. A* is generally preferable for medium to large-scale maps, as it drastically reduces computation time without compromising path quality.

In conclusion, the selected algorithm is A*, for the following reasons:

- the maps used are not particularly noisy or irregular;
- fast computation and responsiveness are critical for integration with perception and reactive navigation;

The system remains modular: switching back to Dijkstra is as simple as changing a single parameter, making it easy to adapt to specific environments or testing needs.

## 5.3 Navigation parameters

In configuring the navigation stack, **each parameter** was set following an extensive testing phase and taking into account the specific characteristics of both the robot and the operational scenario. The rationale behind the chosen values is detailed below:

#### Global Planner

- *use_astar: true*
- *tolerance: 0.5.* A tolerance value of 0.5 m was selected as a trade-off between goal-reaching accuracy and the risk of oscillatory behaviors or blockage near obstacles. Lower thresholds led to frequent oscillations and unnecessary replanning, whereas higher values reduced the precision in reaching waypoints.
- *inflation_radius: 0.45.* The inflation radius defines a virtual buffer around obstacles in the costmap, ensuring the robot maintains a safe distance during path planning. This parameter was calibrated based on the robot's actual footprint and a suitable safety margin. The value was increased with respect to the default configuration after observing that certain planned paths were too close to nearby obstacles.

#### Local Planner and Kinematics

- *min_vel_x*, *max_vel_x*, *acc_lim_x*, *acc_lim_theta*, ecc: these parameters were set in accordance with the mechanical and dynamic limitations of the real (or simulated) robot. Maximum velocity values were intentionally reduced from their theoretical limits to prevent loss of control and to ensure safe maneuvers in narrow or cluttered environments. Acceleration limits were tuned to minimize the risk of skidding or unstable motion.
- *xy_goal_tolerance:* an intermediate value was adopted to prevent the robot from oscillating indefinitely near a waypoint, while still avoiding overly permissive thresholds that might lead to premature goal acceptance far from the intended target.

#### Costmap and Recovery

- *update_frequency*, *publish_frequency:* relatively high frequencies were maintained to ensure system responsiveness to dynamic changes in the environment, particularly in the presence of moving obstacles. However, excessively high values were avoided in order not to overload the system and introduce processing delays.
- *Occupation threshold (occupied_thresh, free_thresh, ecc.):* These thresholds were conservatively tuned to prevent the robot from misclassifying partially occupied or noisy areas as free space, which could result in unsafe planning decisions.

**Recovery behaviors**

- recovery behaviors were kept enabled, with parameter values carefully selected to avoid aggressive or potentially hazardous maneuvers. Backup distances and rotation angles were calibrated to allow the robot to free itself from deadlocks without increasing the risk of collisions with nearby objects.

All parameters were empirically tuned through extensive practical experimentation, balancing accuracy, safety, smoothness, and robustness. The configuration process followed a case-by-case approach, iteratively adjusting values until the robot's behavior was deemed satisfactory under all expected operational conditions.

## 5.4 Point Transformer node

The **Point Transformer** node exposes the ROS 2 service *transform_waypoint*, which receives a 3D point (*geometry_msgs/Point*) expressed in the *oakd_left_camera_frame* and returns the same point transformed into the global map frame (*map*), along with a *success* flag.

Upon startup, the node creates a *tf2_ros::Buffer* with a related *TransformListener*, and then waits until the transformation *oakd_left_camera_frame* → *map* becomes available on the */tf* bus before registering the service.

In ROS, the **TF2** system is designed to maintain and distribute, in real-time, the spatial relationships (translations and rotations) between all coordinate frames on the robot. We use it to avoid manually computing rotation matrices: we simply ask TF2 "where the camera is with respect to the map", and apply that transformation to the received point.

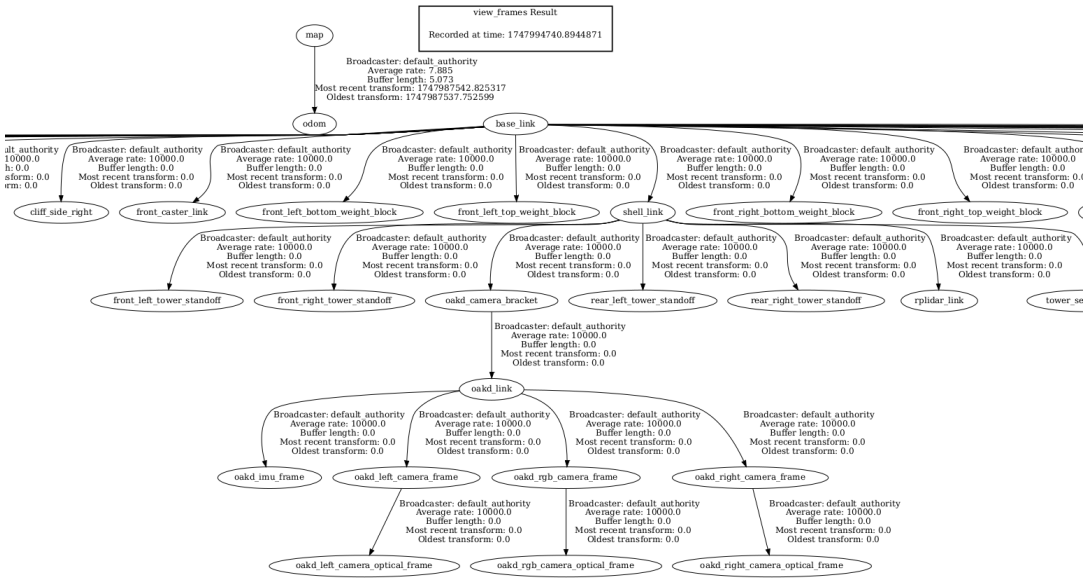


Figure 5.1: TF Tree

The TF tree provides an overview of all coordinate frames present in the robot system and how they are linked. This structure allows the Point Transformer node to retrieve the latest transformation between the camera frame and the global map frame at any time, ensuring that every point can be consistently transformed in real time.

For each service request, the node wraps the incoming point into a *PointStamped* message with the current timestamp, checks the availability of the transform using *lookup_transform*, and if present, applies it using *tf2_geometry_msgs::do_transform_point*. The transformed point is placed in the *transformed_point* field and returned with *success = true*.

If the transform is unavailable or TF2 raises an exception (e.g., *LookupException*, *ConnectivityException*, *ExtrapolationException*), the node responds with *success = false* and logs the error.

The node does **not** publish or subscribe to any topics, **it communicates exclusively via the service interface and the TF2 system**.

## 5.5 Waypoint Navigator node

The *Waypoint Navigator* node represents the core of the developed reactive navigation system, integrating mission control through a finite state machine (FSM) that ensures robustness and flexibility in executing movement tasks, managing waypoints, detecting anomalous conditions (such as the *kidnapped problem*), and performing automatic recoveries.

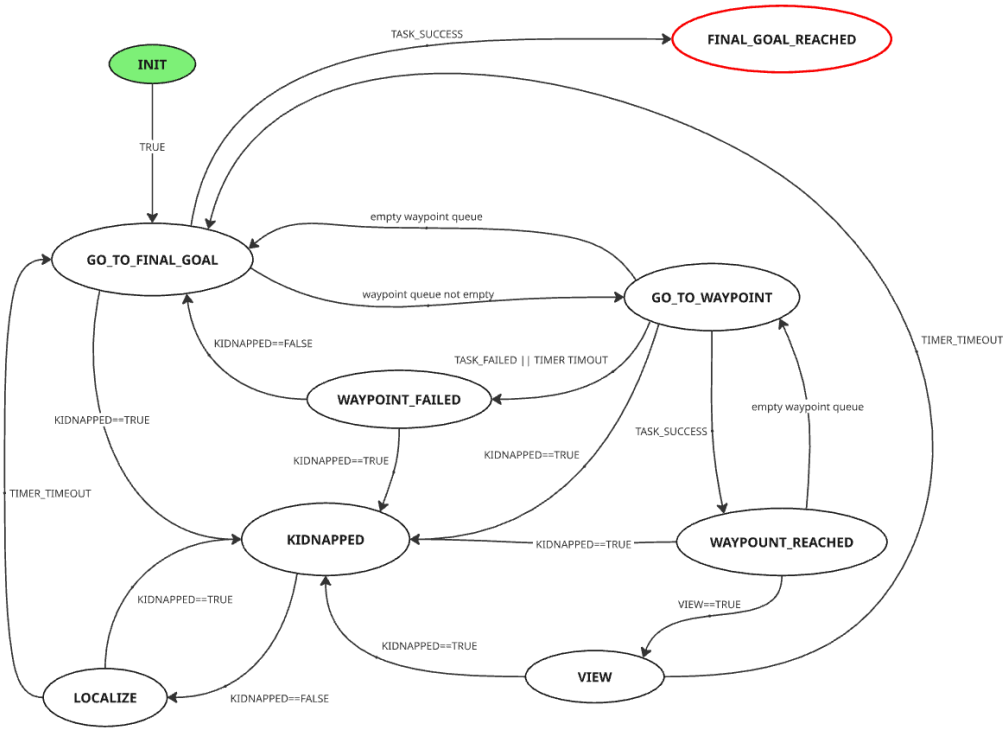


Figure 5.2: Waypoint Navigator FSM scheme.

At startup, the node initializes the robot's pose and establishes connections with ROS 2 services for point transformation (*transform_waypoint*) and local costmap clearing, subscribing to the waypoint topics generated by the perception stack and the state topic */kidnap_status*. The node's behavior is governed by the finite state machine illustrated in Fig.5.2, where each state corresponds to a different phase of navigation:

- **GO_TO_FINAL_GOAL**: navigation towards the final goal, if no intermediate waypoints are present.
- **GO_TO_WAYPOINT**: navigation towards the average position of the waypoints received from the perception stack; activated when the queue exceeds a minimum threshold of points.
- **WAYPOINT_REACHED** and **WAYPOINT_FAILED**: handle the success or failure of navigation towards the waypoint, deciding the transition to new states (such as *view-around* or return to the final goal).
- **VIEW_AROUND**: execution of an exploration maneuver (sequential rotation), useful to acquire new data when no waypoints are available after reaching a position.
- **KIDNAPPED**: special state triggered by the */kidnap_status* topic, which halts navigation and requires a recovery/localization phase if the robot is lifted or manually moved.
- **LOCALIZE**: automatic relocalization behavior, where the robot performs a circular movement to increase the chances of recovering its position through sensors.
- **FINAL_GOAL_REACHED**: terminal state that concludes the mission.

**Waypoint Management** The node receives waypoints from the perception stack through a dedicated topic, transforms them into the *map* frame using a custom service, filters possible outliers, and calculates the *robust average* of the valid points. A new navigation command is generated only if the distance between the average waypoint and the current target exceeds a certain threshold. Each navigation towards a waypoint is protected by a timeout timer: if the robot does not reach the position within the allotted time, a failure is reported and the recovery strategy is activated.

**Handling the Kidnapped Problem** Continuous listening to the */kidnap_status* topic enables real-time detection of situations where the robot is lifted or manually moved ("*kidnapped problem*"). In these cases, the FSM immediately suspends any navigation command, clears the waypoint queue, and transitions to the *KIDNAPPED* state. When the condition ends, a recovery procedure is executed: **local costmap clearing and relocalization** through automatic circular movement (*LOCALIZE*), before resuming the mission.

**Recovery and Robustness** All critical transitions are protected by timeout checks and automatic state variable resets. Multiple and filtered waypoint sending, combined with the view-around maneuver and recovery strategies, have mitigated the main issues encountered in cone detection and navigation in complex environments.

**View-Around Maneuver** In the absence of new visible waypoints upon reaching an intermediate waypoint, the node can activate a local exploration maneuver called *view around*. If enabled through the global variable *VIEW_AROUND*, navigation towards the final goal is temporarily suspended and the robot is commanded to perform a controlled rotation: first 20 degrees to the left, then 20 degrees to the right. The objective is to acquire new environmental data in hopes of detecting additional cones and generating new exploratory waypoints. If new waypoints are detected, the mission continues towards them; otherwise, navigation resumes towards the predefined goal.

However, during the final project phase, **this feature was chosen to be disabled**. Although potentially useful in low-visibility situations, the view-around maneuver caused significant delays in mission progress: the robot had to stop, perform rotations, wait for potential new waypoint reception, and in some cases attempted to return to previously visited cones, further slowing progress. It was considered more efficient to proceed directly to the final goal when no new waypoints are visible, assuming cones are positioned to be easily visible along the planned path. This choice reduced errors and saved time, improving overall navigation effectiveness.

**Conclusion** The implementation of the Waypoint Navigator node, based on an explicit finite state machine, allowed integration of advanced mission management and recovery logic without modifying the underlying Nav2 node structure, significantly improving the overall robustness of the system.

# CHAPTER 6

# TESTING

This chapter describes the validation process carried out on both individual components and the integrated system as a whole. The goal of the testing phase is to ensure that each module performs its intended functionality correctly and robustly, and that the system as a whole operates reliably when all components interact according to the designed architecture.

The testing process has been divided into two main categories:

- **Testing individual components**: focused on verifying the functionality and robustness of each software module in isolation, independently of the rest of the system.
- **Integration testing**: aimed at validating the overall behavior of the system, ensuring correct interaction among the various modules and verifying that the complete perception and navigation pipeline performs as expected in realistic scenarios.

## 6.1 Testing individual components

Before validating the entire system, each software component was tested individually to verify its correct behavior in isolation. These tests were designed to assess the functionality, robustness, and limitations of each node or module without interference from other parts of the system. The goal is to ensure that every component meets the design specifications and is able to perform its tasks reliably under different operating conditions.

### 6.1.1 Acquisition node

The first set of individual tests focused on the acquisition node, responsible for capturing synchronized RGB and depth data from the camera. These tests aimed to verify the node's ability to respect real-time constraints and ensure proper synchronization between RGB and depth frames, which is critical for subsequent processing stages. For practical results, see the Figure 6.1.

| ID | Test Type | Result |
|---|---|---|
| 1 | **Real-time constraint verification** | The real-time constraint is respected, with an average delay of less than one second from the actual scene. This delay is considered acceptable given the low speed of the robot's movement. |
| 2 | **RGB and depth frame synchronization constraint** | The delay between the RGB and depth frames is on the order of milliseconds, ensuring consistent and coherent data acquisition for subsequent computations. |

Table 6.1: Individual Tests - Acquisition Node

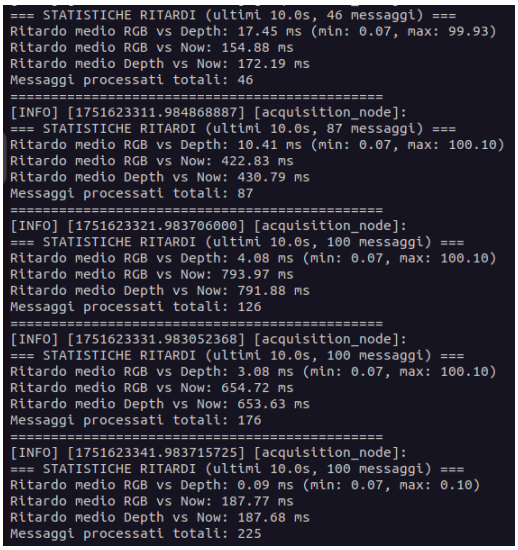


Figure 6.1: Statistics printed from the acquisition node every 10 seconds.

### 6.1.2 Detection node

This set of tests is focused on the detection node, which is responsible for identifying colored cones in the environment. The goal of these tests is to verify the system ability to correctly detect cones at various distances, under different environmental conditions, and in the presence of partial occlusions. The following table summarizes the results of the individual tests carried out on the detection node.

| ID | Test Type | Result |
|---|---|---|
| 1 | **Minimum detection distance** | The system is able to reliably detect cones starting from a distance of approximately 25 cm for red ones. For other colors, the minimum detectable distance is slightly higher. It is not feasible to get closer due to the physical space occupied by the TurtleBot. |
| 2 | **Maximum detection distance** | The maximum detection distance is approximately 7 meters for red and yellow cones, with the exception of blue ones, which are detectable up to 5.3 meters. The green one is detected at a maximum distance of 2 meters. |
| 3 | **Color accuracy** | The system correctly identifies the class of each color. However, we classify orange and red cones to the same "red" class due to their visual similarity. |
| 4 | **Detection performance under varying environmental conditions (e.g., lighting)** | Under bright lighting conditions, yellow and green cones are not reliably recognized. The detection system shows reduced robustness in highly illuminated environments for these colors. |
| 5 | **Detection performance under occlusion** | Red cones show good robustness to occlusion, maintaining reliable detection even when up to 30% of the cone is obscured, especially from the top. Green cones have the lowest robustness under occlusion, followed by yellow and blue. |

Table 6.2: Individual Tests - Detection Node

### 6.1.3 Compute Waypoint node

The following tests were conducted on the compute waypoint node, which calculates the next waypoint based on the estimated position of the cones using depth data. These tests aimed to evaluate the accuracy, robustness, and limitations of the distance estimation and waypoint computation under different conditions.

| ID | Test Type | Result |
|---|---|---|
| 1 | **Minimum distance for reliable estimation** | The system provides a reliable distance estimation starting from approximately 70 cm. Below this distance, the accuracy is not guaranteed. |
| 2 | **Maximum distance for reliable estimation** | The maximum effective estimation distance is 7 meters, which is limited by the detection range of the RGB detector. Beyond this range, no waypoint is computed. The OAK-D datasheet declare a good estimation up to 12 meters. |
| 3 | **Estimation with double cone present** | The system correctly estimates the position of the cone and correctly computes the waypoint in the middle of them. |
| 4 | **Estimation with a single cone present** | The system correctly estimates the position of the cone and computes the waypoint either to the left or right of the cone, maintaining a safe distance to avoid collisions with the TurtleBot. |
| 5 | **Robustness in noisy environments** | In noisy environments, the system shows reduced estimation accuracy beyond 2.5 meters. In particular, the estimation is unreliable when the background is uniform and bright, such as a white wall. |
| 6 | **Estimation while moving** | During motion, vibrations and the reduced frame rate negatively affect the estimation quality. However, as the TurtleBot approaches the cone, the estimation progressively improves, becoming reliable up to the distance of 70 cm. |

Table 6.3: Individual Tests - Compute Waypoint Node

### 6.1.4 Manager node

The following test was conducted on the manager node, responsible for coordinating the perception pipeline by managing data exchange between the different processing nodes. The test aimed to verify the correct synchronization of service calls and the respect of the designed execution flow.

| ID | Test Type | Result |
|---|---|---|
| 1 | **Synchronization with services** | The node correctly synchronizes the interaction with the required services, ensuring that the designed pipeline is executed in the proper sequence without introducing inconsistencies or timing issues. |
| 2 | **Discarding incoming frames if services are busy** | The node properly discards incoming frames when the services are still busy processing previous requests. This mechanism prevents overlapping operations and guarantees the real time constraint. |

Table 6.4: Individual Tests - Manager Node

### 6.1.5 Transformation system (TF)

The following tests were conducted to verify the correct functioning and robustness of the transformations between different coordinate frames, which are essential for projecting detection results from the camera frame into the global map.

| ID | Test Type | Result |
|---|---|---|
| 1 | **Correctness of the TF from camera frame to map frame** | The transformation from the camera frame to the global map frame works as expected, allowing computed waypoints to be correctly projected into the global reference system. |
| 2 | **Robustness of projected points in the map frame** | The system projects all detected points into the map frame, including those located outside the physically reachable area. This behavior represents a limitation, as the system does not inherently filter out unreachable points based on map boundaries. |
| 3 | **Filtering of points above ground level** | The system correctly filters out cones located significantly above the ground plane, ensuring that only valid ground-level cones are considered for further processing. |

Table 6.5: Individual Tests - TF and Coordinate Transformations

### 6.1.6 Navigation node

The following tests were performed to verify the correct functioning of the navigation FSM, responsible for driving the TurtleBot towards target waypoints while avoiding obstacles.

| ID | Test Type | Expected Result |
|---|---|---|
| 1 | **State coverage** | Every state of the finite state machine is visited at least once during execution. |
| 2 | **Transition coverage** | Every transition between states is triggered at least once during execution. |
| 3 | **Sequence coverage** | All meaningful sequences of transitions (including critical paths and exception handling paths) are executed and verified. |
| 4 | **Reachability test** | Every state is reachable from the initial state under some valid input sequence. |
| 5 | **Acceptance test** | The final (accepting) state is reached under the expected input conditions and scenario flow. |

Table 6.6: Navigation Test Plan

## 6.2 Integration testing

After individually validating each software component, the process moved on to the integration testing phase, with the goal of verifying the behavior of the entire robotic system as a whole. This phase is essential to ensure that the interfaces between the various modules (perception, localization, navigation) are correctly implemented and that the robot is able to complete the assigned mission reliably.

During the integration tests, the system was launched in its final configuration, using the complete pipeline:

- acquisition of RGB and depth images through the OAK-D;
- cone detection and waypoint calculation;
- transformation of waypoints into the global frame (map) via the TF service;
- path planning and navigation using Nav2.

The following main functionalities were verified:

| ID | Test Type | Result |
|---|---|---|
| 1 | **Correct data flow between modules** | The waypoints computed by the perception module are correctly received, transformed, and forwarded to the navigation module. |
| 2 | **Interaction between localization and navigation** | The position estimate provided by AMCL remains consistently updated, allowing the navigator to plan robust and coherent paths even in the presence of drift or minor errors. |
| 3 | **Reaction to obstacles** | The robot is able to adjust its trajectory in the presence of unexpected obstacles, exploiting the costmap updated in real time. |
| 4 | **Overall robustness** | Real navigation sessions were conducted in which the robot followed a sequence of waypoints generated in real time, while respecting cone crossing constraints. The system remained stable even under small perturbations (vibrations, lighting changes). |

Table 6.7: Integration Tests

Overall, the integration tests confirmed that the developed pipeline is capable of handling autonomous navigation in real and partially uncontrolled scenarios, ensuring information consistency across modules and robustness against the main types of errors typical of a real-world environment.

# CHAPTER 7

# LIMITATIONS

During the experimental phase, several technical limitations emerged, mainly due to sensor characteristics and non-ideal environmental conditions.

- **Cone detection**: the cone detector showed sensitivity to environmental lighting conditions. In particular, under strong sunlight or intense artificial lights, the system fails to correctly detect yellow, green, and blue cones. This leads to reduced reliability in waypoint generation, since some cones may be missed. However, this limitation does not critically affect navigation, thanks to the implemented redundancy in waypoint publication.
- **Distance estimation via depth camera**: the depth camera exhibited significant limitations, especially in presence of highly reflective white surfaces (e.g., walls) or intense lighting. These conditions compromise the accuracy of cone localization. A potential solution could be the use of alternative sensors such as LiDAR. However, in our specific setup, the cones were too low with respect to the LiDAR scan plane and were therefore never detected.
- **Residual noise and depth filtering**: although software filters were implemented to reduce the stereo camera noise (as described in Chapter 1.1), the improvements were limited. Residual noise, especially in complex environments, remains one of the main challenges for the perception system.
- **Camera vibrations during motion**: during navigation, strong camera vibrations were observed in the real-time image stream, further complicating cone recognition. The unstable point of view, combined with the aforementioned issues, decreases the reliability of detection. Nevertheless, the logic involving multiple waypoint transmissions as the robot approaches a cone partially mitigates this issue and improves navigation robustness.
- **Kidnapped robot problem**: the so-called *kidnapped robot problem* was encountered: when the robot is physically lifted and placed in a random location on the map, it loses localization. The use of a particle filter helped to reduce the impact, but some limitations remain. In very open environments (e.g., lobbies), relocalization takes longer and may result in the loss of nearby cones. In more structured indoor environments, relocalization is faster and more reliable.
- **Waypoints outside the map**: occasionally, waypoints were generated outside the map boundaries. These were not pre-filtered and were still sent to the global planner. However, in such rare cases, the planner successfully recognized the unreachable points and discarded them without triggering abnormal behavior.
- **Behavior near walls**: another issue emerged when waypoints were placed adjacent to or too close to walls or static obstacles. In such cases, the robot sometimes failed to proceed, behaving as if it were already inside the obstacle. This is mainly due to the configured **inflation radius** in the costmap, which inflates obstacles by a safety margin. If a waypoint lies within this inflated zone, the planner treats it as non-traversable or even as a collision

point. This problem is worsened by the partial filtering of target points: not all invalid goals are discarded in advance, which may lead the robot to attempt reaching a theoretically "reachable" location that is actually blocked by the costmap safety logic.

- **Robust view around**: Another limitation concerns the navigation's dependence on the visibility of cones along the path. The current approach assumes that cones, and thus the generated waypoints, are located near the planned trajectory. However, in scenarios where cones are placed in lateral or slightly offset positions, the robot might fail to detect them and, consequently, exclude them from the navigation process.

  In an early version of the system, a *view around* maneuver was implemented, allowing the robot to laterally explore the environment in search of cones not directly visible in front. This strategy was later disabled in the final version due to the excessive time required and the emergence of redundant behaviors (e.g., attempts to return to previously visited waypoints).

  To mitigate this limitation, a more efficient and robust version of the view around maneuver could be introduced in the future, including a filter for already reached or irrelevant waypoints and a reduced time threshold.

  These improvements would allow the robot to extend its exploration capabilities without significantly compromising the temporal performance of the mission.

All these limitations are typical of vision-based robotic systems operating in uncontrolled real-world environments. Despite the challenges, the developed system demonstrated solid robustness, thanks to software mitigation strategies and careful parameter tuning.

# CHAPTER 8

# CONCLUSIONS

The project described in this report successfully addressed the design, implementation, and validation of an autonomous navigation system based on TurtleBot4 and ROS 2, capable of operating in indoor environments with non-trivial constraints and the presence of both dynamic and static obstacles. The modular architecture, structured around the three main modules of perception, localization, and navigation, enabled a clear separation of responsibilities and flexible management of the various challenges encountered during development.

From an *experimental* standpoint, each software component underwent dedicated testing phases, both in isolation and as part of the fully integrated system. This approach ensured high overall robustness and allowed for precise identification of weaknesses within the pipeline, such as sensor behavior under poor lighting conditions or the limitations of the depth camera.

The choice of core algorithms, such as the use of **AMCL** for localization and **A*** for global planning, proved appropriate for the application context. Parameter tuning allowed for a good compromise between reliability, accuracy, and responsiveness, while maintaining maximum modularity for potential future developments.

The system exhibited the typical limitations of real mobile robots operating in uncontrolled environments, particularly in terms of sensitivity to challenging environmental conditions, as thoroughly analyzed in the previous chapter. Nonetheless, thanks to targeted software strategies and careful waypoint management, many of the encountered issues were effectively mitigated. Future implementations could effectively address some of the limitations identified in the previous chapter.

In conclusion, the developed system, despite its inherent limitations, demonstrated solid reliability and robustness in executing autonomous navigation on a real robot. The architectural choices and parameter configurations remain easily extensible, leaving room for future software optimizations.

The delivery folder includes three demonstration videos showcasing the system in action:

- **first video – simple path:** the TurtleBot navigates a straightforward environment containing multiple gates, each formed by one or two cones, and two slalom sections. During the execution, it can be observed that when the robot receives a waypoint located too close to a wall, it fails to proceed toward the final goal. This behavior is caused by the inflation of obstacles in the costmap, which leads the planner to believe the robot is trapped within an obstacle, even though there is still physical space available;
- **second video – circular path:** the robot is shown navigating a circular trajectory, where it is intentionally forced to follow a longer path than the one it would naturally choose. This scenario highlights the system's ability to adapt to constrained environments and navigate around obstacles even when the direct path is not viable. During the execution, cones are dynamically added to the environment as the robot progresses, simulating a scenario

where new information is acquired in real time. The perception and navigation pipeline successfully detects the newly introduced cones, computes updated waypoints, and adapts the robot's trajectory accordingly, demonstrating the robustness and flexibility of the system in dynamic setups;

- **third video – static obstacle avoidance:** this video illustrates the TurtleBot's capability to navigate around static obstacles placed along its path. The robot is tasked with reaching a goal while traversing an environment populated with immovable objects such as boxes or walls that are not part of the predefined map. Thanks to the onboard sensors and local costmap updates, the robot continuously updates its understanding of the environment and modifies its trajectory to avoid collisions. The local planner computes alternative paths in real time, ensuring smooth and safe movement even in cluttered or narrow areas.

# CHAPTER 9

# HOW TO RUN

To correctly run the project, carefully follow the steps below:

1. **Navigate to the project workspace**: open a terminal and move to the main directory of the project workspace.
2. **Power on the TurtleBot4**: make sure the TurtleBot4 robot is turned on and properly connected to the network. If the robot's lights do not turn on at startup, it's likely that the services have not started correctly. In this case, run the following command:

   ```
   sudo bash turtlebot4_restart_services.bash 192.168.0.100
   ```

3. **Build the project**: run the build script located in the main project folder with the following command:

   ```
   ./build.bash
   ```

4. **Load the camera configuration**: use *rqt* to load the YAML configuration file for the camera, located at:

   ```
   config/camera_conf.yaml
   ```

   This file enables the depth camera and correctly sets its parameters.

5. **Restart the camera**: to apply the parameters, restart the camera using the following commands:

   ```
   ros2 service call /oakd/stop_camera std_srvs/srv/Trigger {}
   ros2 service call /oakd/start_camera std_srvs/srv/Trigger {}
   ```

6. **Start the simulation:** run the following script:

   ```
   ./start.bash
   ```

This bash script automatically opens several terminal tabs and launches the main ROS2 components of the system. The script performs the following actions:

- **localization**: starts the localization node, which allows the robot to estimate its position in the environment.
- **navigation Server**: starts the nav2 navigation server, responsible for path planning.
- **perception**: starts the perception nodes, which process information from sensors and cameras.
- **navigation**: starts the control nodes responsible for robot navigation.
- **turtleBot4 Rviz**: launches the TurtleBot4 visualization tool, useful for real-time monitoring of the system and sensors via RViz.

By following these steps, the entire system will be correctly initialized and ready for execution in a simulated environment.

# LIST OF FIGURES