UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Corso di Laurea Magistrale in Ingegneria Informatica

Mobile Robots for Critical Missions
Project Work Report

Group 9

Authors:

Antonio LANGELLA
Mat. 0622702011
a.langella31@studenti.unisa.it

Salvatore PAOLINO
Mat. 0622702016
s.paolino6@studenti.unisa.it

Michele MARSICO
Mat. 0622702012
m.marsico10@studenti.unisa.it

Prisco TROTTA
Mat. 0622702014
p.trotta12@studenti.unisa.it

ANNO ACCADEMICO 2023/2024

CONTENTS

INTRODUCTION

The goal of this project is to create software that lets a mobile robot, the Turtlebot4, move around on its own in a predefined environment. Since we already have a map of the area, the robot should be able to start from any position and navigate by following commands provided through signposts placed at intersections. The Turtlebot4 will need to autonomously move through corridors, reach intersections, and read commands from the signposts. These commands can be of various types: STOP, RIGHT, LEFT, STRAIGHTON, and GOBACK. Based on the command it reads, the robot will make the corresponding decision, like turning right, turning left, going straight, stopping, or going back.

The navigation environment might have an unknown number of obstacles. So, the robot must be able to detect and navigate around them on its own, without any human intervention. In case there are any errors in reading the QRCodes or other emergencies, it's allowed to manually reposition the robot.

In this report we go through the design and development of the ROS2 packages that make up the proposed solution's navigation and perception stacks. We will also discuss the results obtained and the possible future developments of the project.

CHAPTER 1

SYSTEM ARCHITECTURE

The aim of this chapter is to give an high-level description of the architecture we designed for in our proposed solution.

1.1 System architecture

As the requirements ask to autonomously navigate the Turtlebot robot in a dynamic environment, which is officially supported by Nav2, the most sensible choice is to leverage the Nav2 navigation stack as much as possible, as it already provides a robust and tested solution for autonomous navigation. The stack essentially implements everything we need for navigating the robot to a goal position while avoiding (dynamic) obstacles, including the global, behavioral and local planners. For this reason, we only need to implement two essential features in as many ROS2 nodes:

- **Perception node**: this node is responsible for the perception of the environment. It receives the images from the camera mounted on the robot from the topic `/oakd/rgb/preview/image_raw/compressed` and decodes the signposts in the images. The node publishes the list of the detected signposts on the topic `/perception`. The format of the message is discussed in section 1.2;
- **Navigation node**: this node is responsible for the high-level decisions for the navigation of the robot. It subscribes to the topic `/perception` and receives the list of the detected signposts. It also subscribes to topics `/amcl_pose` to get the current pose of the robot and `/kidnap_status` to get the current kidnap state of the robot. The node uses the information contained in the received messages as well as some topological maps of the environment to plan the next area to reach. Then, it uses Nav2's action servers to set the goal and to monitor the status of the robot, as well as cancel the goal if necessary.

The two nodes are implemented in the `autonomous_navigation_challenge` package. The architecture of the system is shown in Figure A.1.

1.2 Perceived signpost interface

The messages exchanged between the perception and the navigation nodes are defined in the `autonomous_navigation_challenge_msgs` package. The message published by the perception node is a `PerceptedSignpostList` message, which is a list of `PerceptedSignpost` messages. The `PerceptedSignpost` message contains the following fields:

- **signpost_type**: the type of the interpreted signpost. The values it can take are `SIGNPOST_NONE`, `SIGNPOST_FORWARD`, `SIGNPOST_LEFT`, `SIGNPOST_RIGHT`, `SIGNPOST_BACKWARD`, `SIGNPOST_STOP`.
- **distance**: the estimated distance of the signpost from the robot in meters;
- **bounding_box**: the bounding box object of the perceived signpost in the image.

An important takeaway is that in the following with the word `signpost` we will refer both to the traffic sign (the icon) and the qr code. That is because the perception node is designed to detect and decode both of them, and the navigation node should be unaware of the difference. This fact is reflected in the message structure, which is designed to be generic enough to contain both types of signposts. In fact, as we will discuss in chapter 3, we both tried traffic signs pipelines as well as qr code pipelines, and the perception node is designed to be able to switch between the two pipelines by simply changing a ROS2 parameter.

Another important aspect is that the message is a list of `PerceptedSignpost` messages, which implies that the perception node can detect multiple signposts at the same time. This is important because the robot can be in a situation where it can see multiple signposts at the same time, and the navigation node should be able to handle this situation. In this case we prioritize the decoded signposts over the undecoded ones, and the closer signposts over the farther ones.

CHAPTER 2

NAVIGATION NODE

In this chapter we will discuss the design decisions we made for the navigation node. We will start by discussing the topological maps we created to simplify the navigation of the robot. Then, we will discuss the navigation node, whose behavior can be described with a Finite State Machine: we will describe the states and the transitions between them.

2.1 Topological maps

2.1.1 Motivations

In order to localize the robot in the environment and to navigate it autonomously, a map of the environment was provided to us. This map is in the form of an occupancy grid map, which is appropriate to precisely localize the Turtlebot4 with its laser scan. However, in the requirements some semantic information was provided, such as the presence of signposts with QR codes at the intersections. In order to encode these information we decided to create a topological map on top of the occupancy grid map, in which each intersection is a node of the map which is tied to a zone of the occupancy grid map via a polygon. This map allows us to interpret a signpost based on the node it is contained in and the observer direction with respect to the signpost. An example of the topological map used for the navigation of the robot is shown in Figure A.3.
A second topological map that we called `kidnapping_map` was used to handle the kidnapping of the robot as described in Section 2.2.4. The kidnapping map we used is shown in Figure A.5.

2.1.2 Local cardinal directions

The topological map's design tries to find a balance between generality of the proposed solution and simplicity. Thus, we decided to limit ourselves to a rectilinear grid map in which each node has a maximum of 4 neighbors, one for each cardinal direction. In order to compensate the defects of the underlying SLAM-based occupancy grid map we decided to use a local cardinal direction system, which means that every node of the map has a local reference system of cardinal directions (north, east, south, west) that is dynamically built based on the connections with the

node's neighbors. This system allows us to interpret the signals of the signposts in a more robust way, since the robot can be oriented in any direction when it reaches a node. An example of the local cardinal directions of a node is shown in Figure A.6.

2.1.3 Cardinal points

The navigation of the robot should be informed by the limitations of the perception stack. This is the reason why we sequence the robot's movements in a way that it tries to cover a node with its cameras as much as possible while looking for a signpost. In order to do this, we decided to define a set of cardinal points for each node of the topological map. These points are the first points that the robot should reach in order to maximize the probability of detecting a signpost. A robot coming from a neighbor node should stop at the cardinal point that is closest to the neighbor node, always looking at the centroid of the node. An example of the cardinal points of a node is shown in Figure A.6.

2.1.4 Localization

The problem of localizing the robot in the topological map is trivial as we already have the localization stack that provides us with the robot's pose in the occupancy grid map. The only thing we need to do is to find the node of the topological map that contains the robot's pose. This is done by checking if the robot's pose is inside the polygon of a node. If the robot's pose is inside the polygon of a node, then the robot is in that node.

2.1.5 Poly creator

In order to create the topological maps, we developed a graphical interface tool that we named `poly_creator`. It is based on RViz and allowed us to easily create and modify topological maps. The user interacts with the tool via "Publish Point" tool of Rviz and "RvizVisualToolGui" buttons. The tool consists of multiple tools:

- **Create node**: allows the user to create a new node by drawing a polygon on the map. Once an area is drawn, the tool automatically creates a corresponding node;
- **Delete node**: allows the user to delete a node by clicking on it;
- **Create edge**: allows the user to create an edge between two nodes by clicking on them. After that the user must specify the direction of the edge (north, east, south, west) on the terminal;
- **Move center**: allows the user to move the center of a node by clicking on it. This is useful as the cardinal directions of the node are calculated based on the center, and the cardinal points are calculated based on the cardinal directions;
- **Move cardinal point**: allows the user to set the offset of the cardinal points of a node with respect to the center. This is useful to set the cardinal points in a way that maximizes the visibility of the node.

By clicking on the "Continue" button of the RvizVisualToolGui, the user can switch the current tool. By clicking on the "Break" button the user can stop the current operation, while by clicking on the "Stop" button the user can save the current topological map with the name defined as the `export_path` ROS2 parameter. The tool can also import a previously created map by correctly setting the ROS2 parameter `import_path`. An example of the tool in action is shown in Figure A.4.

2.1.6 Conclusions

Using our tool to create the topological map simplified the process of defining areas and their relationships. This visual method not only improves the accuracy in the representation of the environment, but also makes the management of connections between nodes more intuitive, thus facilitating the robot's autonomous navigation. Of course, the tool we realized is independent of the underlying map and allows a previously saved topological map to be reloaded and modified. This feature is particularly useful when the environment changes, as it allows the topological map to be updated without having to start from scratch.

2.2 Navigation node

The navigation node is responsible for the robot's autonomous navigation. It receives real-time information from the perception node (as described in Section 1.2) and uses the topological maps to guide the robot's movements in order to follow the path defined by the signposts. The navigation node also handles the behavior of the robot at intersections and the kidnapping and repositioning of the robot, as well as failures from the Nav2 stack. The navigation logic can be interpreted as a Finite State Machine. In the following we describe the said FSM and then we will focus on some specific aspects of the navigation node.

2.2.1 Navigation logic

We implemented the robot's navigation logic within a finite state machine. This FSM doesn't strictly follow Mealy's nor Moore's formalism; it is rather more similar to Matlab's Stateflow formalism in which it is possible to have actions associated both to the states and to the transitions. The FSM consists of 7 states:

- **INITIAL_STATE**: the initial state in which the robot starts the navigation. The robot always starts at a node of the topological map and goes straight to the next node based on its current direction;
- **GO_TO_NEXT_NODE**: the state in which the robot moves to the next node, after it detects and decodes a QR code;
- **WAIT_AND_OBSERVE**: in this state the robot is stationary for a Δt, and waits for the perception node to detect the QR code;
- **APPROACH_TO_CENTROID**: in this state the robot approaches the centroid of the node with a certain number of steps;

- **OBSERVE_LEFT_RIGHT**: while approaching the centroid the robot enters this state and rotates on itself by 30 degrees to the left and right, to search for the QR code;
- **SPIN_AND_OBSERVE**: when the robot is on the centroid it enters this state it turns left and right by 40 degrees until it finds a new signpost;
- **KIDNAPPED**: the state in which the robot is kidnapped, when it is repositioned it exits this state and goes back to the previous one;
- **WAIT_NEAR_SIGNPOST**: the state in which the robot stops for 1 second if it detects a QR code that is within 3 meters. This allows us to detect QR codes that are not inside the expected areas, and to try to decode them.

This finite state machine is can be seen in Figure A.2. In the following, we define the transitions between the states of the automa:

- **T1**: the robot is kidnapped;
- **T2**: the robot is repositioned on the starting node;
- **T3**: the robot reached the next node, and it stops in the nearest cardinal point of the node;
- **T4**: the robot reached the centroid of the node;
- **T5**: the robot is still reaching the centroid of the node;
- **T6**: the waiting time at the cardinal point is over, and the robot did not decode any QR code;
- **T7**: the waiting time at the centroid is over, and the robot did not decode any QR code;
- **T8**: the robot has not yet finished observing left and right, and did not decode any QR code;
- **T9**: the robot has finished observing left and right, and did not decode any QR code;
- **T10**: the robot is repositioned on the ground on the last node;
- **T11**: the robot is ready to go;
- **T12**: the robot has not reached successfully the next node;
- **T13**: the robot has detected the QR code and decoded it;
- **T14**: The robot has detected the QR code, but it cannot decode it.

As for the general operation of the robot, it must be at a node in the topological map in order to start navigation. At the beginning it starts from the initial state, and it moves to the neighbor node according to its initial orientation. Once it reaches a node and has not yet decoded the QR code, it stops for 1 second at the cardinal point closest to the node to try to decode the QR code. In the case that it has not yet decoded the QR code, it progressively takes 3 steps

forward, and at each step it stops and rotates left and right by x degrees to try to decode the QR code. If it still cannot find and decode the QR code, it moves toward the center of the node. Once it reaches the center of the node, the robot rotates on itself to search for the QR code, indefinitely. An example of such behavior is shown in the following clip. As soon as it decodes the QR code, it moves to the next node, and repeats the process. In the case that the robot gets lost or fails to recognize the signal, one can physically lift the robot, and by doing so, it goes into the kidnapped state, in which it must be repositioned in the centroid of the previous node of the kidnapping map.

2.2.2 Interpreting the QR code

Using the topological map, the navigation node interprets the received signals. Based on the received signal and the robot's current position and orientation in the topological map, the navigation node identifies the next node to which the robot should go. For example, if the received signal indicates "Right" and the robot is facing south, the navigation node determines that the next node is the west neighbor of the current node. After determining the next node, the navigation node sets that node on the topological map as the robot's next destination. The navigation node, in synergy with the perception node, allows the system to respond in real time to signals detected in the surrounding environment. Taking advantage of the topological map, the navigation node interprets the signals and guides the robot along an optimal path, thereby improving the effectiveness and accuracy of autonomous navigation.

2.2.3 Avoiding multiple perceptions of the same signpost

In order to avoid decoding the same signpost multiple times we defined a set of rules that the robot must follow to determine when a new signal can be perceived. These rules are as follows:

1. **Non-perception condition for a node**: If the robot has already read a signal from a specific node in the topological map, it will not perceive any more signals from that node. This prevents the robot from processing multiple signals from the same node, which would lead to a wrong planned path;

2. **Neighboring condition**: The robot cannot consider signals that would lead it to nodes that are not neighbors of the last node it visited. This prevents problems when the robot is between two nodes and perceives a signal but it already perceived the signal of the node it is going to;

3. **Special condition for the "GO_BACK" signal**: For the "GO_BACK" signal, the previous conditions do not always prevent problems. In this case, the robot can perceive a new signal only if it is oriented in the opposite direction to the previous detection. This allows the robot to correctly handle go back signals.

2.2.4 Kidnapping feature

The kidnapping state is a crucial aspect of the navigation node. This state is triggered when the robot is physically lifted and moved to a different location. In this case, the robot loses its

position in the topological map and must return to the last node it visited, placed in the centroid of that node. The robot then resumes its navigation from that node. This feature ensures that the robot can recover from unexpected events, such as being moved by an external agent, and continue its autonomous navigation effectively. The effective position and direction where the robot have to be placed in the last node can be seen on Rviz interface, with a colored arrow A.7. When the robot is placed in the correct position, it relocates itself, it clears both the local and global costmaps, and it cancels the goal to the next node. To implement this feature, we created a new topological map with more nodes than the one used for the navigation. This map is used to store the last node visited by the robot, and the direction in which the robot was moving. This information is used to guide the robot back to the last node in the event of a kidnapping. A practical example of how kidnapping works can be seen in this clip.
Another solution could have been to use `amcl`'s services to evenly distribute the particles on the map, but it would have required more time for the robot to re-localize. Given the specific application of the robot, we decided to implement a more efficient solution.

2.2.5 Navigation improvements

We have implemented several improvements to the navigation node to enhance the robot's autonomous navigation capabilities. First, we have improved the speed of the robot, allowing it to move faster. In order to do this, we changed the following parameters of the navigation stack:

- **controller_server**: `yaw_goal_tolerance`, `max_vel_x`, `max_vel_theta`, `max_vel_xy`;
- **velocity_smoother**: `max_velocity`.

To apply these changes, we need to set the `safety_override` parameter of the motion controller to "full". This is done by the navigation node.

Another improvement we made was to improve the robot's ability to detect obstacles in its path. To do this, we changed the following parameters of other nodes:

- **bt_navigator**: `default_nav_to_pose_bt_xml`;
- **local_costmap**: `update_frequency`, `width`, `height`;

These changes allow the robot to detect obstacles in its path earlier and avoid more collisions, thereby improving its safety and reliability during autonomous navigation.

Finally, we used a tool named Groot [1] to visualize the behavior tree used by Nav2 and modify it. In particular, we increased the frequency of the rate controller, increasing the frequency of update of the recovery branch, and the width and height of the local costmap, in order to increase the robot's ability to detect obstacles in its path. We can see the behavior tree of the robot in Figure A.10.

We implemented a built-in logic for changing these parameters directly from the navigation node, and this behavior can be easily disabled with a ROS2 parameter during simulation.

2.3 Simulations in gazebo

To help us with the development of the navigation node when the real robot is not available, we have reconstructed the 3D map in Gazebo (Figs. A.8 and A.9), based on the one provided in 2D. This map is used to simulate the robot's movements and test the navigation node's functionality. In addition, we implemented a mock node that simulates the perception of the robot, publishing messages of user-selected signals via the keyboard on the correct topic. This allows us to test the navigation node's response to different signals and scenarios. We also implemented a mock node that simulates the kidnapping of the robot, publishing messages on the correct topic to trigger the kidnapping state. An example of its operation is shown here. This allows us to test the robot's response to being moved to a different location and verify that it can return to the last node visited.

CHAPTER 3

PERCEPTION NODE

The perception module's job is to understand the surrounding environment by capturing and processing images. Getting this task right is super important for the TurtleBot's self-navigation and overall project success. The accuracy and reliability of the perception module directly affect the safety and efficiency of TurtleBot operations. When it can accurately sense critical elements along the path, it not only improves navigation but also lowers the risk of collisions and makes the robot more adaptable in changing and tricky environments.

So, having a strong perception module is crucial because all of TurtleBot's moves depend on correctly reading signals along the way.

3.1 Solution design

When designing the solution, we took a systematic approach by thoroughly analyzing the requirements and specific challenges. We built a robust data processing pipeline that incorporates machine learning models to detect and decode QR codes. The architecture was optimized for real-time performance, considering hardware limitations and the need for precision and reliability. The design process also included extensive field testing to continuously validate and improve system performance. All steps are discussed in this chapter and are based on the success of two main tasks:

- **QR code detection**: the robot must be able to detect the QR code in the environment and it's crucial for subsequent decoding;
- **QR code decoding**: the robot must be able to decode the QR code to understand the correct movement command.

The reason why we focused on QR codes rather than traffic signals is that there is virtually any risk of false positives with the QR codes, while with the traffic signals, the robot could easily misinterpret a right turn signal as a left turn signal, for example. This could lead to the robot taking the wrong path and getting lost. However, we still believe that from further away it is

easier to decode a traffic signal than a QR code, so we could consider using traffic signals in the future.

3.1.1 Camera Pre-Processing

Analyzing the perception module must begin with an initial analysis of the camera, its characteristics, and the pre-processing it performs on the acquired images. This analysis is essential to better understand the characteristics of the images published on the camera's topic and to modify them consciously if needed.

The camera on the Turtlebot4 we are using is a **OAK-D Pro** [2][3] by Luxonis [4]. Its most important specifications are shown in Fig. A.11.

The pre-processing of the image in the DepthAI pipeline mainly happens through the Image Signal Processor (ISP) and post-processing of the images. The ISP performs a series of operations such as noise reduction, color correction, and auto-exposure, which are essential for improving the quality of the raw image captured by the sensor. After the ISP, the image undergoes further post-processing steps like **cropping** and **resizing**, producing various output formats. With the cropping operation, the output is resized up to a maximum of 4K (3840x2160). Afterward, an additional resizing is applied, reducing the output size to a fixed dimension (with equal height and width).

The operations performed on the image can be summarized in the pipeline seen in the Fig. A.12, while a practical example is provided in Fig. A.13.

Based on this, to better familiarize ourselves with the camera and its parameters, tests were conducted by varying the parameters. The goal of this initial phase was to find a camera configuration that would capture as much of the scene as possible with sufficient quality to effectively and efficiently detect and decode QR codes. From these experiments, the chosen parameters are as follows:

```
"keep_preview_aspect_ratio": false,
"resolution": "4K",
"width": 2560,
"height": 1440,
"framerate": 30.0,
"preview_size": 1000,
```

Parameters not listed were not changed from the default version.

This camera configuration allows for capturing a wide portion of the scene, utilizing the full Field of View (FoV) available. Additionally, the high resolution ensures a high Pixels Per Meter (PPM) value, providing greater detail and making image analysis easier.

A comparison before and after the parameter change is shown in Figures A.16 and A.17, which represent the camera view with the default parameters and the view with the changed camera parameters, respectively.

3.1.2 Vibration and Latency Testing (Field Trials)

Once the camera parameters were selected and validated, the next step was to understand how much the camera's performance was affected by network latency and the robot's movement. A test was set up to address two fundamental aspects:

- **Vibration**: the robot's movement can cause vibrations that affect the camera's ability to capture clear images. This test aimed to understand how the camera's performance was affected by the robot's movement.
- **Latency**: the network latency can affect the camera's ability to capture real-time images. This test aimed to understand how the camera's performance was affected by network latency.

It was observed that camera vibration made many of the acquired images unusable. To overcome this issue, instead of adjusting the camera parameters, we chose to modify the robot's navigation logic. For instance, the robot was programmed to stop at strategic points, allowing the camera to focus on the acquired image. This aspect is explained in more detail in the navigation chapter.

Regarding latency, we opted to work with the topic `/oakd/rgb/preview/image_raw/compressed`, which is the compressed version of the topic `/oakd/rgb/preview/image_raw`, to minimize the size of the data transmitted over the network.

3.1.3 Methodological approach

Once the issues related to the camera were addressed, we moved on to the problem of detecting and decoding QR codes, taking into account:

- Detection/decoding of QR codes;
- The distance at which the QR code is detected/decoded.

The workflow used to tackle this task involved recording some bagfiles of the robot in motion, placing QR codes at different positions and distances, and in various perspective/lighting conditions. This approach allowed us to work offline, without the need to use the robot directly, while still simulating its behavior. Both camera vibrations and network latency were preserved in this process.

3.1.4 Processing pipelines

For the task of detecting and decoding QR codes, various image processing pipelines were designed. The design of the pipelines started with validating the provided baseline, aiming to improve it where possible, and continued by exploring other viable alternatives.

The baseline involves using a Python library, OpenCV, and the class `QRCodeDetector()`. The `detect and decode` methods in OpenCV rely on a combination of image processing techniques, shape recognition, and algorithmic decoding to effectively detect and interpret QR codes and barcodes. These methods are particularly useful in scenarios where it is necessary to process images containing multiple QR codes quickly and efficiently. However, experiments

using this approach were not very satisfactory, as they only ensured detection and decoding of QR codes under limited conditions of distance and perspective/lighting.

After validating the baseline, we proceeded along two parallel paths:

1. **Improving the input provided to the QRCodeDetector class**: Although not always effective, this class returns a response extremely quickly. Therefore, we tried to exploit this speed by changing the input provided. The idea was to input multiple patches of the same image through a sliding window mechanism, which moves across the image and selects only a sub-area of the image. This idea is based on the principle behind the most established detection methods over the years, such as YOLO.

2. **Changing the QR code detection and decoding model**: This approach involves completely changing the pipeline, seeking valid alternatives to the OpenCV class.

Among the OpenCV alternatives analyzed, the **Qreader** module for detection and decoding was identified.

3.1.5 Qreader Module

QReader [5] is a robust and straightforward solution for reading challenging QR codes within images using Python, powered by a YOLOv8 model. The backbone of this model consists of two main blocks: a YOLOv8 QR detection model trained to detect and segment QR codes (also available as a standalone component) and the Pyzbar QR decoder, that is a Python version of ZBar, an open source software suite for reading bar codes from various sources, such as video streams, image files and raw intensity sensors. Leveraging the information extracted from this QR code detector, QReader seamlessly applies various image pre-processing techniques on Pyzbar to maximize decoding accuracy on difficult images.

Among the pre-processing techniques applied to the images are:

1. **Resizing**: Images are resized with different scale factors (1, 0.5, 2, 0.25, 3, 4). This allows testing QR code recognition at various sizes.

2. **Color inversion**: For QR codes with a black background and white foreground, the image is inverted (255 - image).

3. **Grayscale conversion**: If the image is in RGB or BGR, it is converted to grayscale using `cv2.cvtColor`.

4. **Binarization**: The image is binarized using Otsu's thresholding (`cv2.threshold` with `cv2.THRESH_BINARY + cv2.THRESH_OTSU`).

5. **Blur and threshold**: Gaussian blur filter is applied with different kernel sizes (e.g., (5, 5), (7, 7)) to reduce noise before attempting decoding.

6. **Sharpening**: A sharpening kernel is applied to the image (`cv2.filter2D` with `_SHARPEN_KERNEL`) to enhance QR code details and edges.

7. **Perspective correction**: Perspective correction is performed on QR codes using OpenCV's perspective transformation (`cv2.getPerspectiveTransform` and `cv2.warpPerspective`).

These techniques are combined seamlessly to optimize the probability of recognizing and decoding QR codes even in challenging images.

An important consideration for designing future processing pipelines is how this model can be instantiated. Specifically, the QReader model can be instantiated with variations such as:

- **Model size [str]**: The size of the model to use ('n' for nano, 's' for small, 'm' for medium, or 'l' for large). Larger models offer higher accuracy but slower performance. Default: 's'.
- **Min confidence [float]**: Sets the minimum confidence threshold for QR detection to be considered valid. Lower values can result in more false positives, while higher values may miss difficult QR codes. Default and recommended value: 0.5.

3.1.6 Comparison between different pipelines

The first comparison in terms of decoding between Qreader, pyzbar, and OpenCV is conducted by the providers of the Qreader model itself. This comparison is based on two test images and demonstrates how the Qreader module performs well even where pyzbar and OpenCV fail.

Based on these findings, various workflows have been devised that combine different aspects summarized as follows:

- Basic model for QR code detection and decoding:
  - OpenCV
  - Qreader: this model has been tested with different sizes and varying minimum confidence thresholds.
- Pre-processing on the raw image:
  - Cropping the top part of the image
  - Dividing the image into patches using the Sliding Window approach, resulting in 25 extracted patches, or through static division of the image into 3 areas: left, center, and right.
  - Applying an image stretch to restore the original 16/9 aspect ratio.

The test dataset was constructed from recorded bagfiles and consists of 456 images containing QR codes or none. The QR codes vary in distance, and the dataset also includes blurry images to make the test as realistic as possible. From the results obtained, it was observed that even when applying OpenCV's detect and decode module on smaller image patches, this module still fails to achieve its primary goal. Therefore, the choice for the module for QR code detection and decoding fell on QReader. Among the various alternatives based on this model, the selection eventually settled on the pipeline:

- Image stretching: We decided to apply image stretching to restore the aspect ratio of the image to its original 16:9 value.
- Cropping the top part of the image: During tests with the Turtlebot, it was observed that the upper part of the image is unnecessary for QR code detection. This is because whether the robot is far away from or close to the QR code, the QR code never falls within the cropped area. Therefore, we decided to crop 1/4 of the top part of the image to lighten the computational load on it. We can see this experiments in Figs. A.14 and A.15. The reasoning behind cropping a horizontal portion of the image cannot be applied to the bottom part because when the robot is very close to the QR code, it occupies the target area.
- Image division into 3 windows: We opted for an approach based on dividing the image into patches because the Qreader module performs better with square images. However, we chose not to use the more general sliding window approach as it would only slow down computation without significantly improving results.
- Model size reduction: We decided to keep the model size nao to meet the constraints of real-time system execution.
- Minimum confidence for QR code detection set to 0.3: We chose to maintain a minimum confidence value that is not too low to limit the number of False Positives. At the same time, we didn't want the confidence value to be too high to avoid missing True Positives altogether. Based on field tests, a confidence value of 0.3 seemed like a good compromise to handle this trade-off effectively.

3.2 Solution implementation

From an implementation standpoint, various modules were developed to meet different needs. The main focus was on getting the perception module working properly. Several tweaks were made to speed up testing and simulate the expected behavior of this module.

The main module, `perception.py`, is the actual perception node, aimed at publishing commands for the robot. A key part of the project was creating a common interface for all pipelines, making it easy and fast to integrate new solutions. This modular approach not only simplifies system updates and maintenance but also makes expanding the robot's perception capabilities a breeze with minimal effort. The code structure supports different processing pipelines easily, as shown by the PIPELINES dictionary. Each pipeline can be dynamically loaded based on ROS2 parameters, offering great flexibility and allowing quick testing and implementation of new solutions.

Another strong point is the automatic management of camera parameters. There's a built-in logic for changing these parameters directly from the node, which can be easily disabled with a flag during simulation. This ensures the camera can be optimized for various operating conditions without manual intervention.

The code also includes a debug mode that allows real-time visualization of the processing pipeline's output. This is particularly useful for monitoring and verifying the system's

performance during development and testing, helping to troubleshoot and optimize the system. Using CvBridge for image conversion and OpenCV for processing enables efficient image handling and analysis, making the system robust and responsive.

Error handling has been carefully addressed: if there's an issue loading a pipeline, the node safely shuts down with an appropriate error message. This helps maintain system reliability even when something goes wrong.

Additionally, measuring processing time and calculating frames per second (FPS) allows for performance monitoring and optimization. The processing results are published via ROS2, enabling the perception system to easily integrate with other components of the robotic system. These strengths make the perception module highly flexible, efficient, and easy to extend with new features, effectively meeting the needs of an advanced robotic system.

For calculating the distance at which the QR code is detected, two different formulas based on methods for designing video analysis systems were tested:

1. The first formula is based on information such as the camera's Field Of View (FoV) and dimensions in pixels and meters. The distance is calculated as the average of the distance estimates along the two dimensions of the image. Each distance is calculated as follows:

$$d = \frac{d_{QR,\text{real}} d_{I,\text{pixels}}}{2 d_{QR,\text{pixels}} \tan(\frac{FoV}{2})} \tag{3.1}$$

   where:

   - $d_{QR,\text{real}}$ is the real dimension of the QR code in meters;
   - $d_{I,\text{pixels}}$ is the dimension of the QR code in pixels;
   - FoV is the camera's Field Of View.
   - $d_{QR,\text{pixels}}$ is the dimension of the QR code in pixels.

2. The second formula, on the other hand, is based on triangle similarity. The distance calculation strictly depends on the camera's focal length and the object's dimensions in pixels and meters. Starting from the focal length, the distance is calculated along the two dimensions and then averaged. Each distance is calculated as follows:

$$d = \frac{f d_{QR,\text{real}}}{d_{QR,\text{pixels}}} \tag{3.2}$$

   where:

   - f is the camera's focal length;
   - $d_{QR,\text{real}}$ is the real dimension of the QR code in meters;
   - $d_{QR,\text{pixels}}$ is the dimension of the QR code in pixels.

Between the two formulas, the second one turned out to provide more accurate measurements and that is why the choice fell on it.

CHAPTER 4

RESULTS AND DISCUSSION

The objectives set for the project have been achieved. Despite this, there are some areas that can be improved and some that definitely need modification. Below, the results obtained are discussed along with their weaknesses and possible future developments.

4.1 Final results

4.1.1 Navigation Node

- The navigation stack works well in most cases, and it completed very long paths without any problem;
- The robot is approximately double as fast as before and this allows us to spare time especially in the corridors;
- Most of the time the robot is able to avoid obstacles and to replan the path when necessary, but given the high speed of the robot if an obstacle suddenly gets in front of the robot sometimes it crashes into an obstacle before circumventing it;
- Sometimes when the costmaps are particularly dirty the robot could plan a path to the correct goal via the wrong corridor.

4.1.2 Perception Node

- With the implemented solution, we are able to perform QR code detection even beyond a distance of 3 meters and when the robot is moving;
- The decoding is always successful when the robot is at most 3.5 meters away from the signpost, but the robot must stay still. If the robot is moving, the camera shaking makes the decoding process fail;
- Minimum confidence level set to 0.3. Based on our field experiments, we found this to be a good balance, however we are aware of the presence of false positives. In Fig. A.18 we can

find a specific scenario where the solution identified would fail to detect only QR codes. However, the reasoning applies to both operations to be performed on the image, namely detection and decoding: although in this case a traffic sign is detected as a QR code, the decoding fails, resulting in a message whose content regarding the type of signal detected is still invalid;

- The approach used for selecting the pipeline is based on both quantitative and qualitative analysis of the results obtained. Although it remains a sufficient analysis overall, the number of detections evaluated represents a non-automated estimate and is therefore subject to error. Such a "quick" approach was favored over other statistical approaches for design choices: due to limited time, greater priority was given to the overall functioning of the system to be developed rather than organizing tests on pipelines. Given more time, what would have been done is to calculate some classic evaluation metrics for detection such as Precision and Recall. The steps are summarized below:
  - Selection of a larger test set;
  - Labeling of the test set: each label consists of information about the bounding boxes of QR codes within the image (center, width, and height relative);
  - Calculate evaluation metrics such as Precision and Recall.
  - To improve performance estimation, vary the threshold and calculate mAP (mean Average Precision).

4.2 Future developments

4.2.1 Navigation

- A possible improvement could be, once a QR code is detected but not decoded due to its distance from the robot, to convert this distance into global world coordinates, also using information on the robot's orientation, and then move closer to the QR code to decode it;
- We could configure Nav2 to slow down the robot when it is close to a perception zone, so that the perception stack will more easily detect and decode the QR code;
- To solve the problem that sometimes the robot chooses the wrong corridor, we could modify the behavior tree to clean the costmap when the robot is in a specific area or at a specific frequency.

4.2.2 Perception

- Given the current behavior of the navigation node, which is quite cautious, the robot can detect a multitude of signposts, especially in the first part of the intersection. However, if the signal is located in the final part of that node, the robot takes a long time to perform all the planned steps. One solution would be to design a pipeline that decodes the signposts from further away;

- Distance estimation from signposts could be improved by using the robot's stereo camera. This would allow the robot to better estimate the distance of the signpost and slow down when it is close to it;
- Enhancement of QR code quality: Trying to improve the quality of the QR code can be useful for attempting to decode it even when the image is blurry or when the QR code is not clearly visible. There are various QR code enhancement modules based on generative models designed to generate a high-resolution image from a low-resolution QR code image, making the QR code easier to decode;
- An attempt was made to address camera vibrations by adding a video stabilization module to the processing pipeline. The idea was to stabilize the video flow to minimize blurry images, thereby simplifying QR code detection and decoding. However, tests using this approach did not improve performance, so the decision was made not to implement it.

APPENDIX A

FIGURES

In this appendix we put all the referenced figures. Each chapter has its own section, and each section contains all the images referenced in the corresponding chapter

A.1 Chapter 1

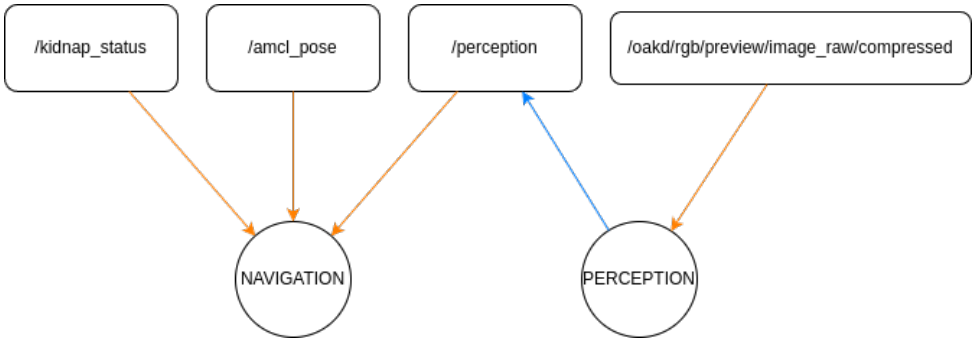

Figure A.1: Nodes interface

A.2 Chapter 2

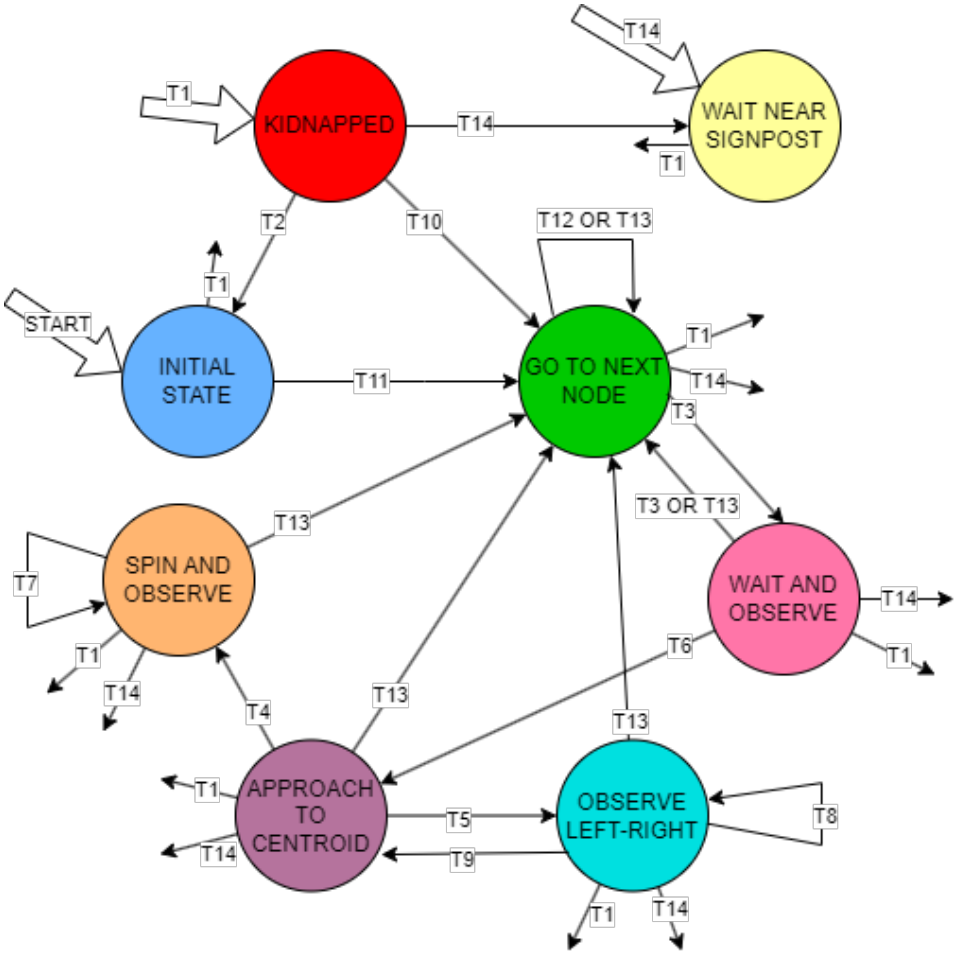

Figure A.2: Navigation finite state machine

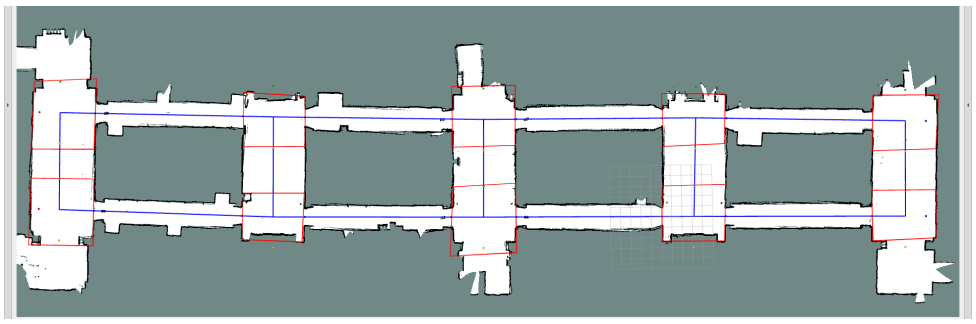

Figure A.3: Topological map of the environment

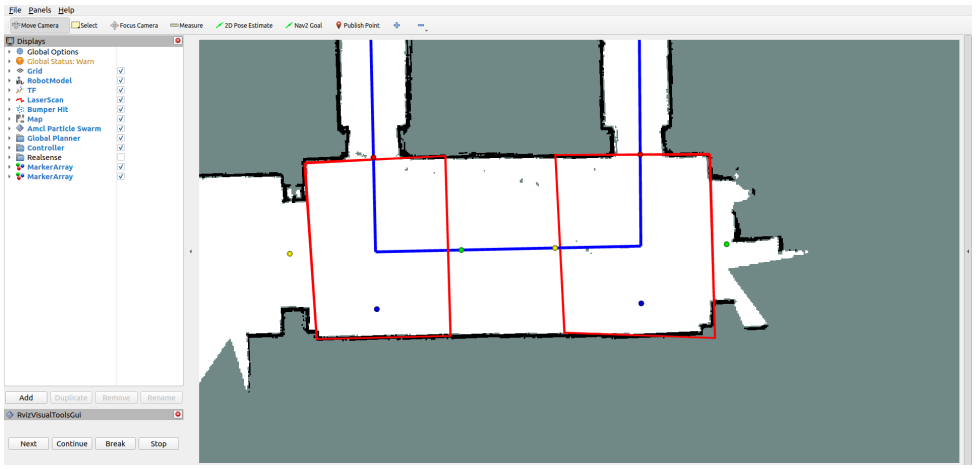

Figure A.4: A node of the topological map

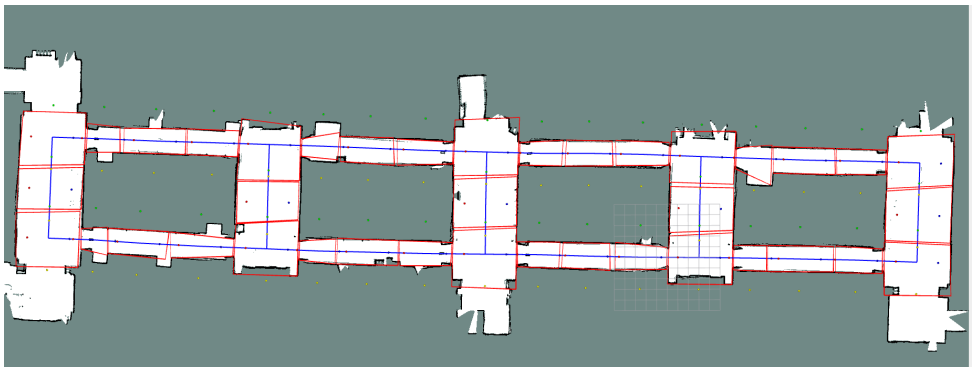

Figure A.5: Topological map used for the kidnapping feature

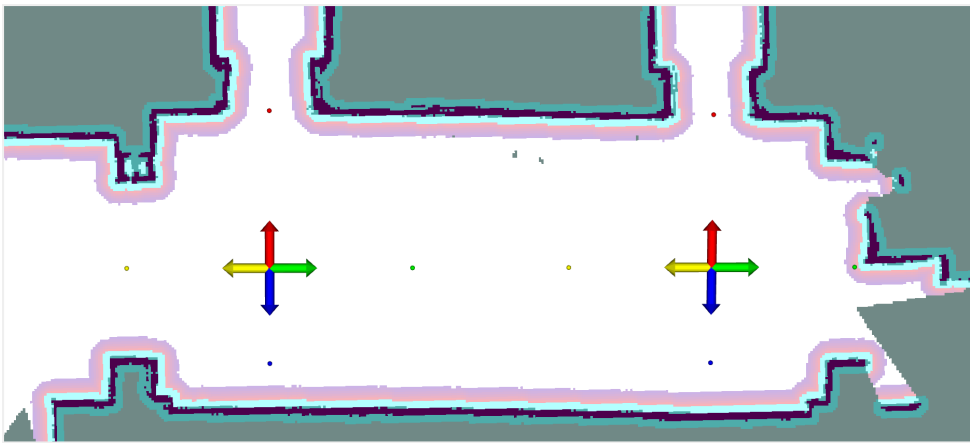

Figure A.6: The cardinal directions (arrows) and cardinal points (dots) of a node

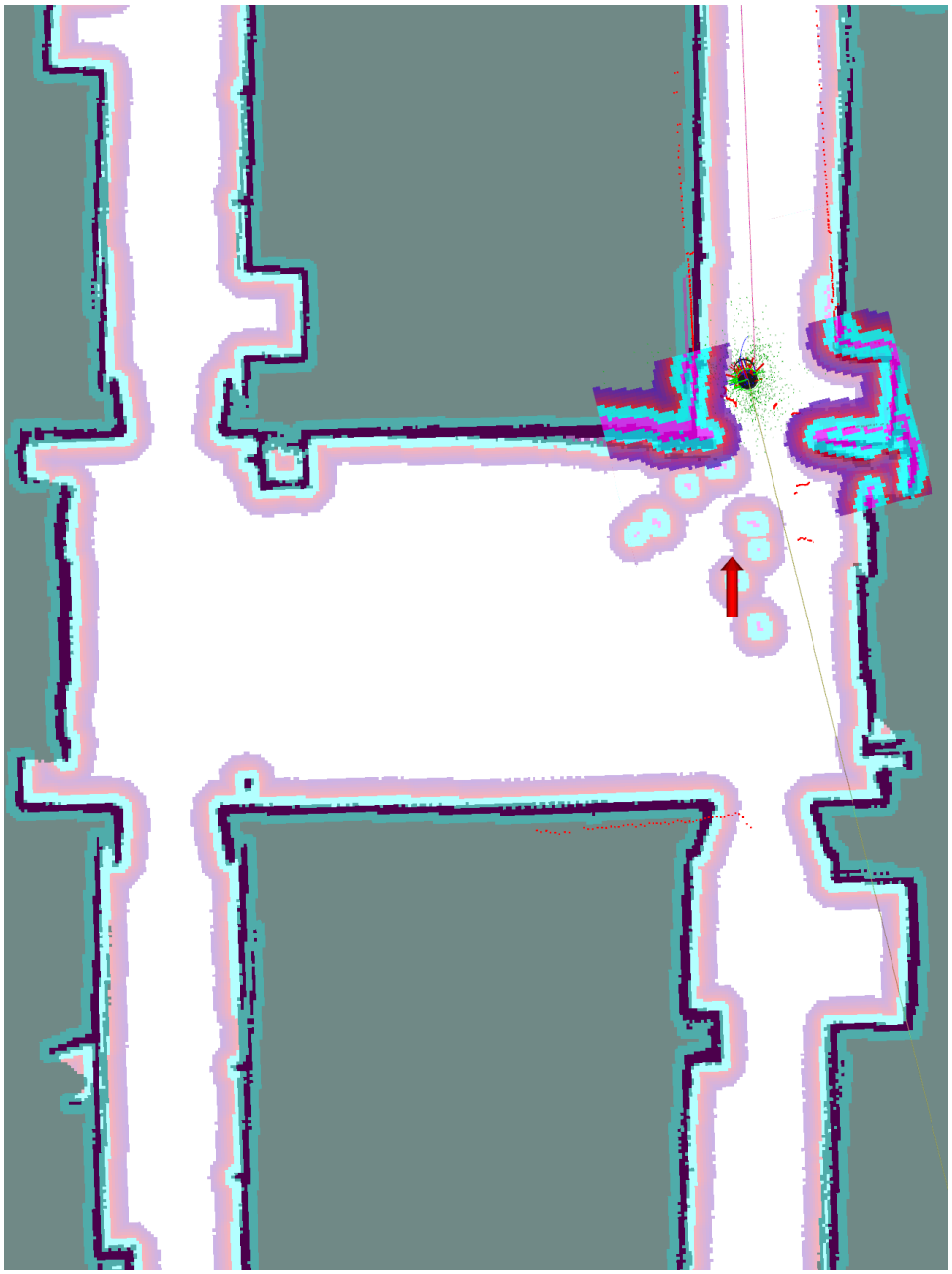

Figure A.7: Example of RViz view when kidnapping is perfomed

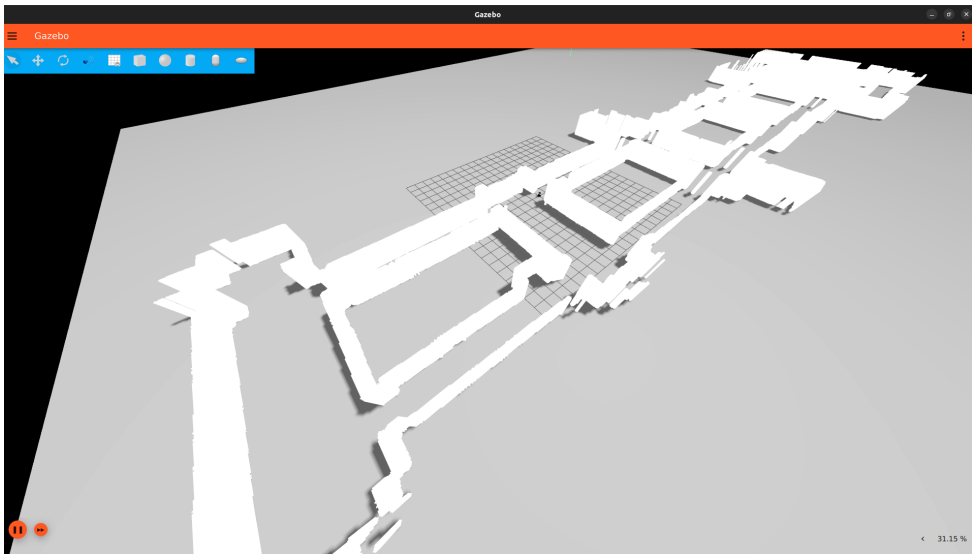

Figure A.8: DIEM World in Gazebo

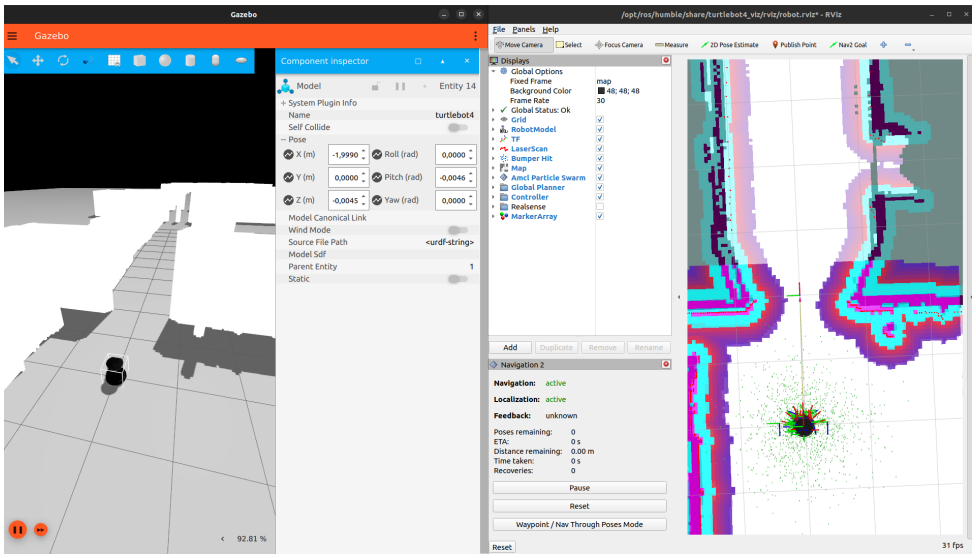

Figure A.9: Simulation in Gazebo

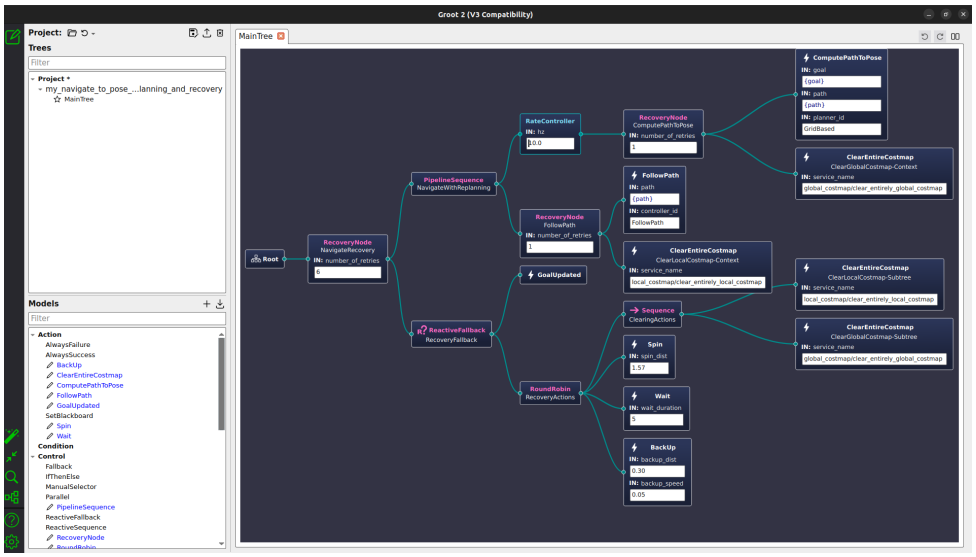

Figure A.10: Behavior tree of the robot

A.3 Chapter 3

Camera Specs	Color camera	Stereo pair
Sensor	IMX378 (PY004 AF, PY052 FF)	OV9282 (PY044)
DFOV / HFOV / VFOV	81° / 69° / 55°	89° / 80° / 55°
Resolution	12MP (4056x3040)	1MP (1280x800)
Focus	AF: 8cm - ∞, FF: 50cm - ∞	FF: 19.6cm - ∞
Max Framerate	60 FPS	120 FPS
F-number	1.8 ±5%	2.0 ±5%
Lens size	1/2.3 inch	1/4 inch
Effective Focal Length	4.81mm	2.35mm
Pixel size	1.55µm x 1.55µm	3µm x 3µm

Figure A.11: OAK-D Pro specifications

Figure A.12: Camera pipeline

Figure A.13: Example of cropping and resizing

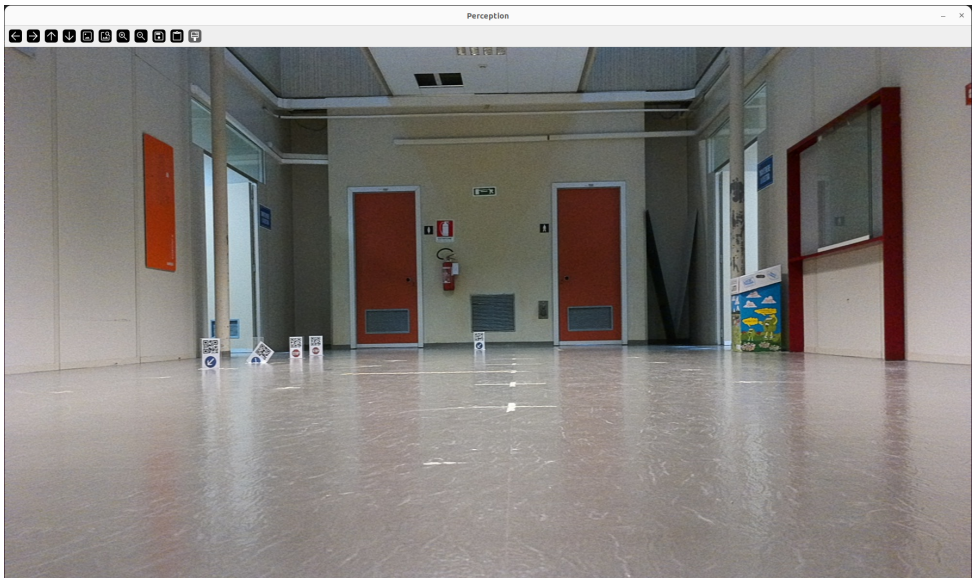

Figure A.14: Example of max distance view

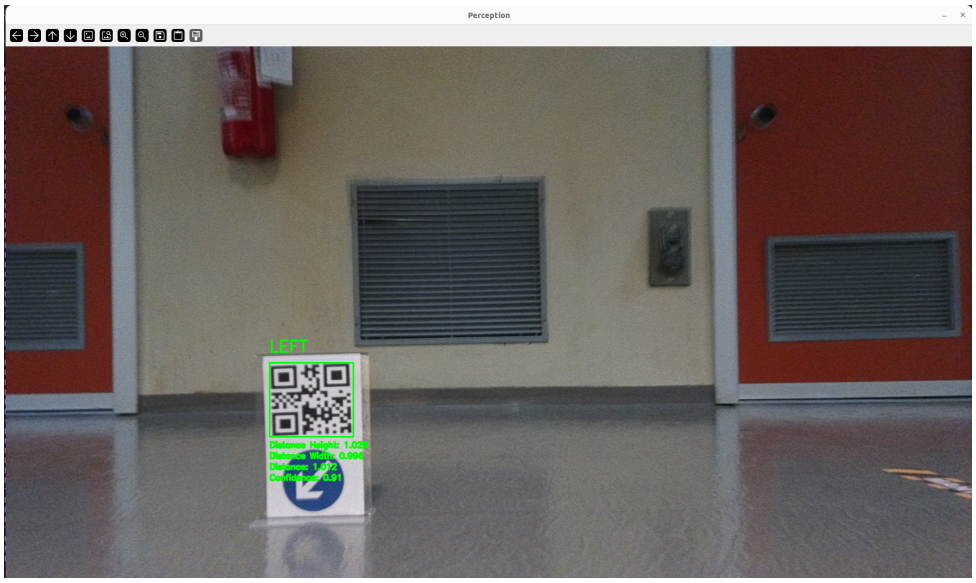

Figure A.15: Example of close up view

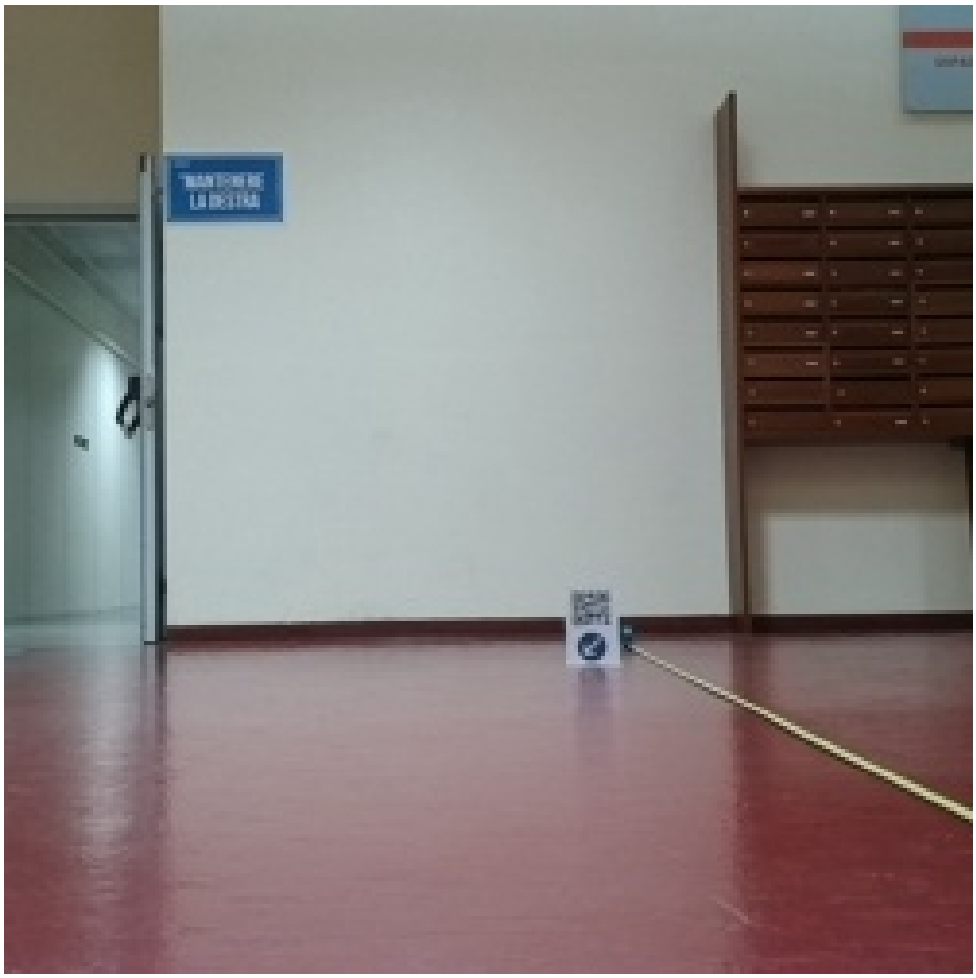

Figure A.16: Example of view with default parameters of camera

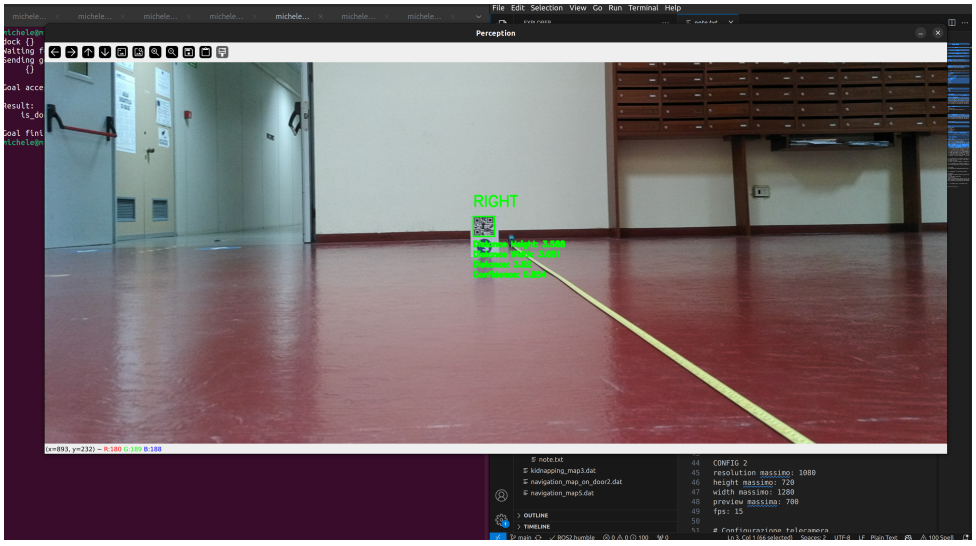

Figure A.17: Example of view with modified parameters of camera

A.4 Chapter 4

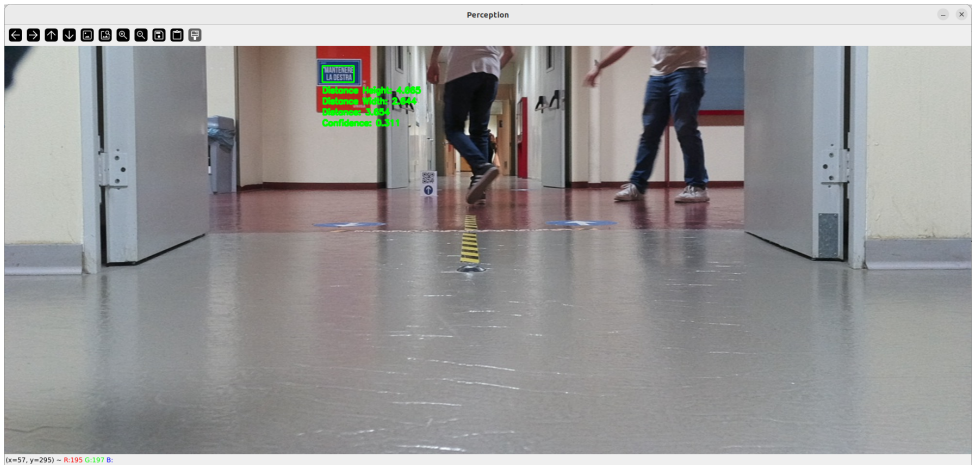

Figure A.18: Example of a scenario where the solution would fail to detect only QR codes.

LIST OF FIGURES

BIBLIOGRAPHY

[1] BehaviorTree. *Groot*. https://github.com/BehaviorTree/Groot [Accessed: (14/06/2024)].

[2] luxonis. *OAK-D: Stereo camera with Edge AI*. Stereo Camera with Edge AI capabilities from Luxonis and OpenCV. 2020. URL: https://luxonis.com/.

[3] luxonis. *OAK-D Pro Documentation*. https://docs-old.luxonis.com/projects/hardware/en/latest/pages/DM9098pro/ [Accessed: (14/06/2024)]. 2020.

[4] luxonis. *DepthAI: Embedded Machine learning and Computer vision api*. Software available from luxonis.com. 2020. URL: https://luxonis.com/.

[5] Eric-Canas. *QReader*. https://github.com/Eric-Canas/QReader?tab=readme-ov-file [Accessed: (14/06/2024)].