ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

School of Engineering and Architecture

Academic year 2025/2026

Master Degree Thesis

performed by the Student:

Simone Comastri (nr. 0001138853)

enrolled on the degree programme:

Automation Engineering (LM-25)

at:

SIR Robotics S.p.A

on the following topic:

Development of an Automated Container Depalletization System Using AMRs, Industrial Robots and Vision Technologies

Student: **Simone Comastri**

Academic Tutor: **Gianluca Palli**

Host Structure Supervisor: **Stefano Cozza**

Abstract

This thesis presents the development and evaluation of an automated solution for depalletizing boxes from shipping containers, carried out at SIR Robotics S.p.A. The proposed system integrates an Autonomous Mobile Robot (AMR), an industrial robotic manipulator, and an AI-based vision system to enable efficient and safe container unloading.
The work was divided into three main phases. First, the AMR platform was tested and a strategy was developed to align the robot with the loading ramp and navigate safely inside the container. The second phase focused on evaluating different vision systems for box detection and localization. Finally, the picking procedure was validated by testing the grasping of boxes and their transfer onto a conveyor belt.
The results demonstrate the feasibility of the proposed approach and provide a basis for the future development of fully automated container unloading systems.

Contents

Abstract 2

Introduction 9

1 KUKA KMP 1500P-EU-D diffDrive 11

1.1 KUKA Mobile Robots 11

1.2 Hardware Description 11

1.2.1 Operation Modes 13

1.2.2 Navigation Mode 14

1.2.3 Obstacle Detection System 14

1.3 KUKA.AMR Studio 15

1.3.1 Creation and Management of Maps 16

1.3.2 Localization of the AMR on a Map 17

1.4 KUKA.AMR Fleet Manager 17

1.4.1 Layout Management 17

1.4.1.1 Building Layout Management 17

1.4.1.2 District Management 18

1.4.2 Graph Editor 19

1.4.3 Model Management 20

1.4.3.1 Fine Localization with Reflectors 21

1.5 Tests with KMP 1500P-EU-D diffDrive 22

1.5.1 Map Creation 23

1.5.2 Navigation Graph 24

1.5.2.1 Charging Node 25

1.5.2.2 Rack Node 26

1.5.2.3 Parking Node 26

1.5.2.4 Ramp Alignment using Reflective Stickers 26

1.5.3 Container Internal Navigation 28

1.5.4 Python Modules 29

1.5.5 Depallettization Workflow 31

1.5.5.1 Initialization of Map Data and Robot State Monitoring 31

1.5.5.2 Step 1: Rack Lifting 31
1.5.5.3 Step 2: Rack Placement 32
1.5.5.4 Step 3 and 4: Container Position Update 32
1.5.5.5 Step 5: Target Computation 33
1.5.5.6 Step 6: Best Path Computation 33
1.5.5.7 Step 7: Container Emptying 34
1.5.5.8 Step 8: Return to Starting Position 35
1.5.5.9 Step 9: Recharging 35
1.5.6 Final Considerations 36

2 Vision Systems **37**
2.1 Operating Scenario and Vision Requirements 37
2.2 IT+ Robotics 38
2.2.1 Graphical User Interface 39
2.2.1.1 FrontEnd 39
2.2.1.2 BackEnd 40
2.2.1.3 System Configuration 41
2.2.2 Tests 42
2.2.2.1 Setup 42
2.2.2.2 Ordered Layout on a Single Layer 43
2.2.2.3 Unordered Layout on a Single Layer 44
2.2.2.4 Ordered Layout on Multiple Layers 45
2.2.2.5 Unordered Layout on Multiple Layers 47
2.2.2.6 Detection in the Presence of Visual Disturbances 49
2.2.2.7 Pick Trajectory Simulation 50
2.2.3 Robot Integration 51
2.2.4 Final Considerations 51
2.3 Other Vision Systems Tested 51
2.3.1 Photoneo 51
2.3.2 SICK 53

3 Robot **57**
3.1 Introducion to ABB 57
3.1.1 ABB IRB 4600 57
3.1.2 Robot Studio 58
3.1.2.1 RAPID Programming Language 59
3.1.2.2 Home 59
3.1.2.3 Modeling 60
3.1.2.4 Simulation 60
3.1.2.5 Controller 60

3.1.2.6 RAPID 61
3.2 Simulation Setup 61
3.3 Pick and Place Trajectory 62
3.3.1 Home Position 62
3.3.2 Scan Position 63
3.3.3 Approach Position 63
3.3.4 Grip Position 64
3.3.5 Leave Position 65
3.3.6 Box Orientation 65
3.3.7 Positioning the Box in Front of the Conveyor 66
3.3.8 Lowering and Reorientaion 66
3.3.9 Place Position 67
3.4 Integration with Vision System 67
3.4.1 Communication between Robot and Vision System 67
3.4.1.1 Socket Communication Function 68
3.4.2 Main Vision System Commands 69
3.4.3 Acquisition Cycle for Robot-Camera Integration 70
3.4.4 Final Considerations 72

Conclusions and Future Developments **73**

Bibliography **75**

List of Figures

1.1 KMP 1500P . . . 11
1.2 KMP 1500P – Front and Rear Views . . . 12
1.3 Charging station . . . 13
1.4 Wired remote controller . . . 14
1.5 Laser scanner detection field . . . 15
1.6 3D camera detection field . . . 15
1.7 Map creation page . . . 16
1.8 Localization page . . . 17
1.9 Building and Layout creation . . . 18
1.10 District Creation . . . 19
1.11 Graph overview . . . 20
1.12 Rack . . . 20
1.13 Holder . . . 20
1.14 Tray . . . 20
1.15 Model editor . . . 21
1.16 Stickers on rack . . . 21
1.17 Reflective sticker model editor . . . 22
1.18 AMR for tests . . . 23
1.19 Map of the building . . . 23
1.20 Graph for tests . . . 24
1.21 AMR charging . . . 25
1.22 Function parameters . . . 25
1.23 Basic parameters . . . 25
1.24 AMR with rack loaded . . . 26
1.25 Boxes configuration for ramp simulation . . . 27
1.26 Sticker applied on box . . . 27
1.27 Edge settings . . . 27
1.28 Ramp and Container nodes . . . 28
1.29 Obstacle Detection Plan . . . 29
1.30 Edge settings . . . 29

2.1 Zivid 2+ L110 38
2.2 Front End 39
2.3 System Configuration 42
2.4 Test 1 43
2.5 Test 2 43
2.6 Test 3 44
2.7 Test 1 44
2.8 Test 2 45
2.9 Step 1 45
2.10 Step 2 46
2.11 Step 3 46
2.12 Step 4 46
2.13 Step 5 47
2.14 Step 1 47
2.15 Step 2 48
2.16 Step 3 48
2.17 Step 4 48
2.18 Step 5 49
2.19 Test 1 49
2.20 Test 2 50
2.21 Starting Point 50
2.22 Approach Pose 50
2.23 Picking Pose 50
2.24 MotionCam 3D M 51
2.25 PhoXi Control GUI Overview 52
2.26 Pick Trajectory 52
2.27 Picking Pose 52
2.28 Exit Trajectory 52
2.29 Detection Results 53
2.30 Failed Detection 53
2.31 PALLOC Visionary-S 54
2.32 GUI Overview 54
2.33 Mistakes in top layer computation 55
2.34 Localization on a single layer 55
2.35 Localization on multiple layer 56
2.36 Localization with low lightning 56

3.1 IRB 4600-45/2.05 58
3.2 Robot Studio main page 59

3.3 Home panel 60
3.4 Modeling panel 60
3.5 Simulation panel 60
3.6 Controller panel 61
3.7 RAPID panel 61
3.8 Simulation setup 61
3.9 Home position 63
3.10 Scan position 63
3.11 Approach position 64
3.12 Grip position 64
3.13 Leave position 65
3.14 Box orientation 66
3.15 Box in Front of the Conveyor 66
3.16 Lowering and Reorientation 67
3.17 Place Position 67

Introduction

The thesis project was carried out at SIR Robotics S.p.A. (Soluzioni Industriali Robotizzate). SIR Robotics S.p.A. is an Italian company specialized in the design and integration of advanced robotic and automation solutions for industrial environments. Founded in 1984, SIR develops customized systems for a wide range of sectors, including assembly, foundry processes, handling, pallettizing, finishing processes and welding. The company combines industrial robots, vision systems, and automation technologies to create highly flexible and efficient production processes. With strong experience in robotic integration and customized solutions, SIR supports clients during the entire project development, from design to installation and commissioning.

The main objective of this project was to evaluate the feasibility of developing an automatic depalletization system for boxes inside a shipping container.
Today, many companies must unload a large number of containers filled with boxes of different sizes and types. To meet the need for higher speed, efficiency, and safety during unloading operations, fully autonomous solutions are being developed, reducing or eliminating the need for human intervention.

Another important motivation for this type of automation is the working environment inside shipping containers. During the summer months, the temperature inside a closed container can become extremely high, making manual unloading physically demanding and potentially unsafe for operators. By using an automated robotic system, it is possible to avoid the need for workers to enter the container, improving both safety and working conditions.

The application involves the use of an industrial robotic arm which, equipped with a suitable end-effector, will pick the boxes from inside the container. After gripping the box, the robot will place it on a conveyor belt that transports it to the sorting stations. A system is required to carry both the robotic arm and the conveyor belt on which the boxes will be placed. Furthermore, the robot must be equipped with a device that allows it to detect and identify the boxes.
The application therefore requires the collaboration of three main systems:

- an AMR (Autonomous Mobile Robot), used for transporting the robotic arm on a metal support and carrying the conveyor belt;

- a robotic arm, used for picking the boxes from the container;
- a vision system, integrated with the robot, which detects the position of the boxes and determines an optimal unloading sequence.

The purpose of this project was to identify and test the components needed for the future implementation of the project.

The first phase of the project focused on testing and evaluating the KUKA AMR. In particular, I worked on implementing a reliable solution to correctly align the AMR with the loading ramp it must climb to transport the robotic arm inside the container. For the tests, a 20-foot shipping container was used (although the system is designed to operate also with 40-foot containers). The container opening has a width of approximately 2.3 m, and the loading ramp must not exceed a slope of 5%. As a consequence, the ramp needs to be roughly 3 m long to guarantee safe access for the AMR. An additional objective of this phase was to develop a navigation strategy that enables the AMR to move inside the container and reach the back of it, ensuring complete unloading of all boxes.

The second phase of the project involved configuring and testing several vision systems equipped with AI-based box localization capabilities. These systems were evaluated based on their ability to accurately detect and locate boxes inside the container, as well as on their overall ease of use and integration.

The third phase focused on validating the box picking process. In this stage, the grasping procedure generated by the selected vision system was tested, including the execution of the picking trajectory and the placement of the boxes onto a conveyor belt. The objective was to verify that the identified boxes could be reliably grasped and transferred, ensuring a smooth and continuous depalletizing workflow.

1. KUKA KMP 1500P-EU-D diffDrive

1.1 KUKA Mobile Robots

Mobility is becoming very important in modern Industry 4.0. For this reason, KUKA develops mobile robotic solutions that make production more flexible and efficient. Instead of fixed production lines, factories are now using intelligent mobile units that can move and work by themselves.
KUKA's Autonomous Mobile Robots (AMRs) are designed to be easy to integrate into different industrial environments, such as automotive, aerospace, and internal logistics.
KUKA mobile platforms offer:

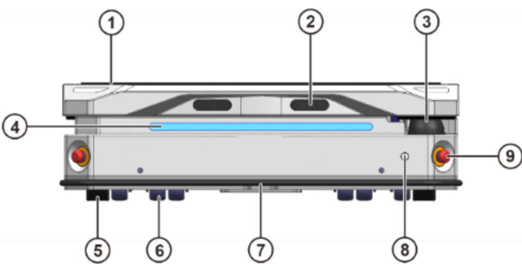
- High flexibility and free movement, even in tight spaces
- Precise omnidirectional motion, thanks to technologies like mecanum wheels
- Fully autonomous navigation, without floor markings or magnetic guides
- The ability to work together, allowing multiple robots to operate together

These features make KUKA AMRs a strong base for the future factory, where mobility, autonomy, and adaptable production systems are essential.

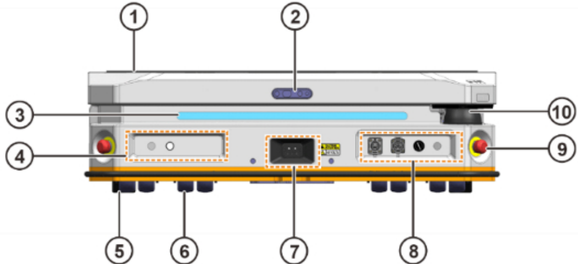
1.2 Hardware Description

Figure 1.1: KMP 1500P

The KUKA KMP 1500P EU-D diffDrive is an autonomous mobile robot designed to improve intralogistics, material transport, and process integration in industrial environments. It uses a differential-drive system, with two drive wheels and spherical caster wheels that provide support and balance. These wheels enable forward and backward motion (at a maximum speed of 1.8 m/s) as well as in-place rotation, which provides excellent mobility in narrow or constrained spaces such as container interiors. The AMR is equipped with a 60 mm lifting stroke and supports payloads up to 1500 kg, allowing it to transport a wide range of load carriers and heavy equipment.



Front View:
(1) Payload Plate
(2) 3D Camera
(3) Laser Scanner
(4) LED Strip
(5) Drive Wheel
(6) Roller
(7) Bumper
(8) Reset Button
(9) Emergency Stop



Rear View:
(1) Payload Plate
(2) 3D Camera
(3) LED Strip
(4) Control Panel
(5) Drive Wheel
(6) Roller
(7) Charging Connection
(8) Control Panel
(9) Emergency Stop
(10) Laser Sscanner

Figure 1.2: KMP 1500P – Front and Rear Views

The onboard electrical system is powered by a battery located at the rear of the AMR. The battery has a total capacity of 48 Ah. It can be charged using a portable charger, a conductive charger, or an inductive charger.

Figure 1.3: Charging station

The system integrates several advanced technologies, including SLAM navigation for accurate localization and mapping, 3D cameras for obstacle detection, and 360-degree safety coverage through two laser scanners and emergency-stop bumpers. Additional features include modern charging options such as inductive charging, long battery autonomy of up to ten hours, and optional 5G connectivity. Load identification is supported through built-in technology and QR-code readers, improving workflow accuracy and tracking.

These capabilities make the KMP 1500P particularly suitable for highly dynamic and demanding applications. For this project, it was selected as the transportation platform for the robotic arm and the conveyor system that must be moved inside the container. Its high payload capacity, precise navigation, modularity, and easy integration into existing workflows make it an excellent fit for developing an automated depalletization solution in a restricted environment.

1.2.1 Operation Modes

The AMR can operate in the following modes:

- **Automatic mode**: The AMR is controlled by the Fleet Manager. It moves autonomously with a programmed speed, and the laser scanner monitoring is always active. This mode will be used during the operational phase of the project, allowing the AMR to execute the entire container unloading process autonomously.
- **Maintenance mode**: The AMR is moved at low speed (0.2 or 0.6 m/s) using a wired remote control. The safety field monitoring is disabled as long as the speed stays below 0.2 or 0.6 m/s.

Figure 1.4: Wired remote controller

Maintenance mode will only be used during the initial mapping phase of the warehouse.

- **Brake release mode**: The AMR brake is released so the platform can be manually moved by an operator.

1.2.2 Navigation Mode

The AMR operates using SLAM navigation (Simultaneous Localization and Mapping), a technique that enables the robot to build a map of the environment while simultaneously determining its position. In this mode, the AMR continuously acquires data from its laser scanners, inertial sensors, and wheel odometry. These measurements are combined through SLAM algorithms to identify walls, obstacles, and stable reference features in the surroundings. By comparing real-time sensor data with the previously generated map, the AMR can estimate its pose, update the map when necessary, and plan safe and efficient trajectories.
SLAM navigation is particularly suitable for this project, as it provides the level of accuracy required for aligning the AMR with the loading ramp and for navigating inside the container, where the robot must operate in narrow, partially structured, and dynamically changing environments.

1.2.3 Obstacle Detection System

The AMR is equipped with the following sensors for obstacle detection:

- **Laser scanners**: Two laser scanners are installed, one on the front-left corner and one on the rear-right corner of the AMR. Each scanner can detect obstacles at a height of 170 mm from the ground, within a 270° field in front of the scanner. The scanners monitor the area around the AMR, known as the protection field. If something enters this protection field, the AMR stops. The protection field can be up to 3000 mm long. In automatic mode, the size of the protection field increases gradually as the AMR speed increases.

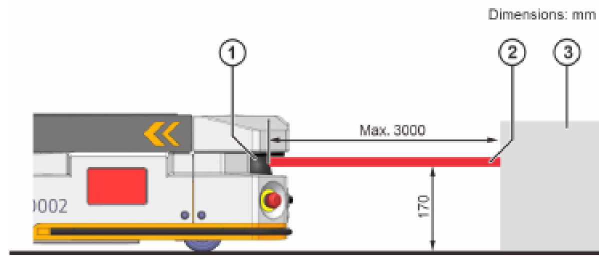


Figure 1.5: Laser scanner detection field

- **3D cameras**: Two 3D cameras are mounted at the front and rear of the AMR. Each camera can detect obstacles:
  - Up to 4000 mm in front of the camera
  - Within a horizontal field of 72° and a vertical field of 50°
  - Up to a height of 2000 mm from the ground
  - Minimum detectable object at 2000 mm: 80×80×80 mm
  - Minimum detectable object at 4000 mm: 200×200×300 mm

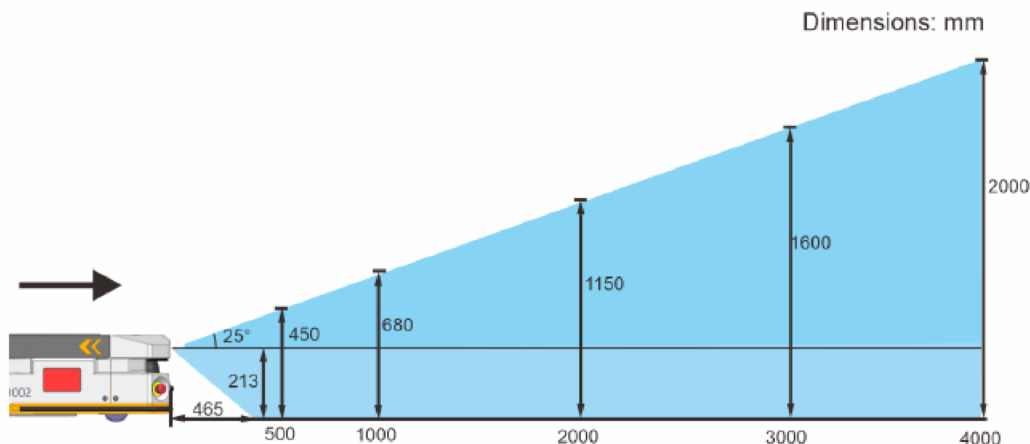


Figure 1.6: 3D camera detection field

1.3 KUKA.AMR Studio

KUKA.AMR Studio is a software application used to configure the AMR, create and edit the environment map, set navigation parameters, and manage the robot's main functionalities. It acts as the main commissioning tool for the KMP platform: the operator connects directly to the AMR to configure the vehicle, while AMR Studio also provides the interfaces required to register the AMR within a Fleet Manager system. In this way, AMR Studio represents the link between the physical robot and the fleet-level coordination software, ensuring that the AMR is correctly set up, networked, synchronized, and ready to receive missions from the Fleet Manager. It offers all

necessary functionalities to commission, configure, and maintain the vehicle, including system parameter adjustments, diagnostics, and software updates through a structured and intuitive interface.

In this project, KUKA.AMR Studio was employed for several key configuration tasks:

- **Wireless Network Configuration**: Configuring WiFi connectivity and IP parameters to allow the AMR to communicate with the Fleet Manager.
- **Fleet Manager Configuration**: Registering the AMR within the fleet manager and enabling mission control.
- **Time Synchronization**: Aligning system time between the AMR, the Fleet Manager, and the network infrastructure.
- **Navigation Commissioning**: Generating and validating the map, calibrating sensors, and testing autonomous navigation.

1.3.1 Creation and Management of Maps

The basis for SLAM navigation is the building map, which records the surrounding contours, using the laser scanners of the vehicle. Since the robot does not yet have any reference data, it is necessary to create the map manually. KUKA.AMR Studio is used to map the environment as part of the commissioning. The AMR is jogged, using its jogging function or through the remote controller, to cover all the areas where it will be operating.

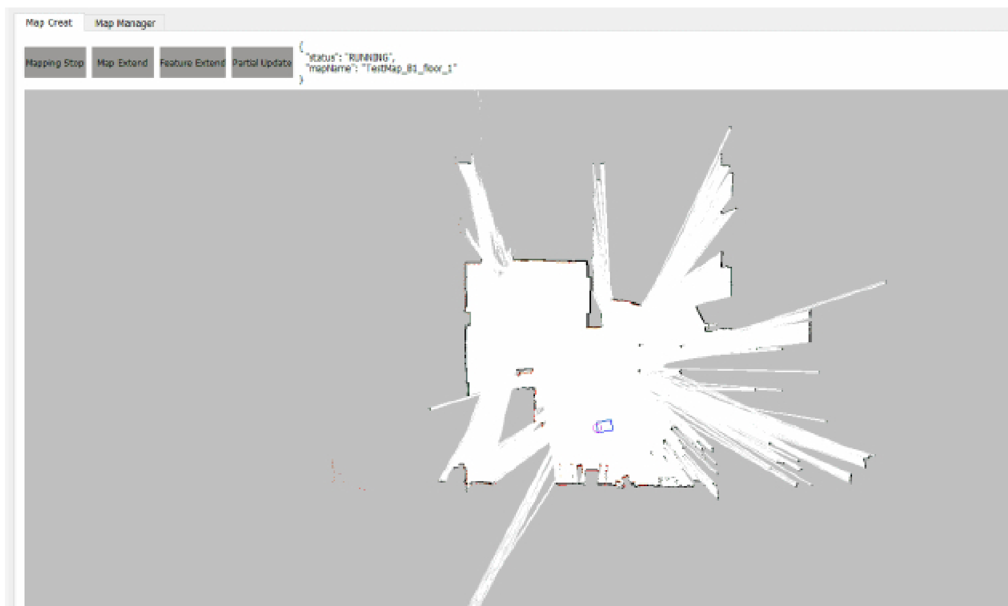


Figure 1.7: Map creation page

1.3.2 Localization of the AMR on a Map

After activation of the map, the robot's pose in the map must be verified and corrected if necessary. The goal of this localization step is to teach the AMR its initial position in the environment map. On the screen, red contours represent real-time obstacles detected by the AMR, while black contours represent the static obstacles stored in the SLAM map. The objective is to align these two sets of contours. If the AMR appears outside the map or shifted with respect to its real position, the robot must first be moved manually to approximately the correct location. Then, the *Initialize* command automatically adjusts the map alignment using the live laser data.
If the contours do not perfectly overlap, manual fine alignment can be performed using the adjustment keys until the real-time red obstacles match the black map obstacles.

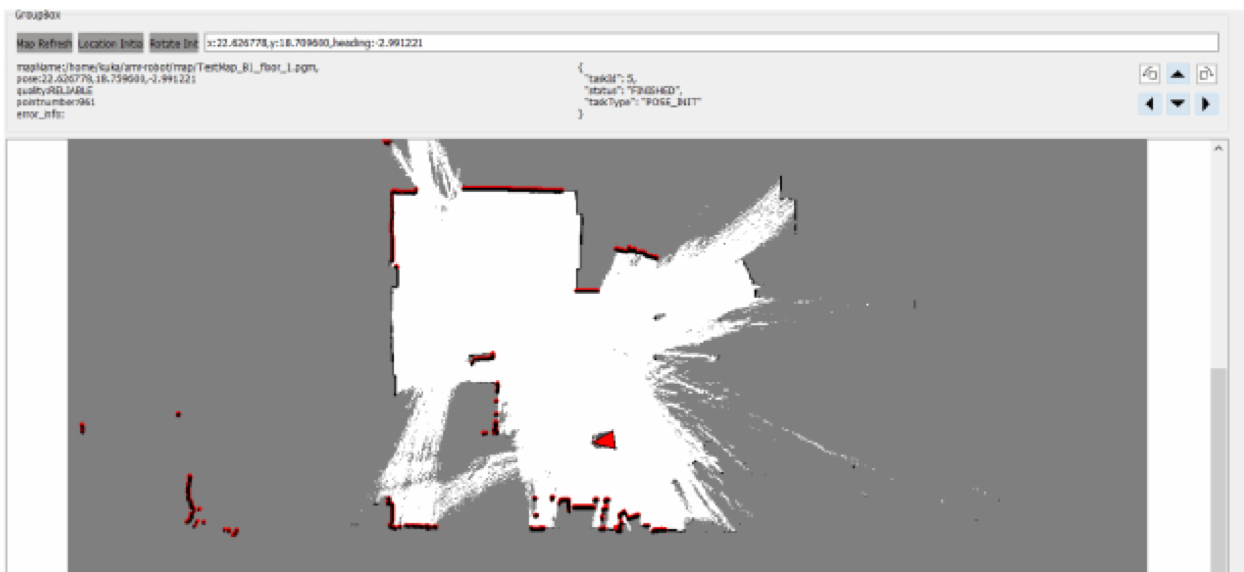


Figure 1.8: Localization page

1.4 KUKA.AMR Fleet Manager

KUKA.AMR Fleet is a web-based application used to manage multiple AMRs, configure maps and navigation graphs, and supervise all robot operations, it coordinates the AMRs in the fleet. The Robot Control System is responsible for managing resources, graphs and missions. It ensures efficient interaction and coordination of the basic elements. The robot control system enables robot to be controlled on a graph that is based on the requirements of the mission.

1.4.1 Layout Management

1.4.1.1 Building Layout Management

Before the AMRs can operate in an industrial environment, it is necessary to define the structure of the building in which they will move. In KUKA.AMR Fleet, this is done by creating a

building layout that represents the physical organization of the facility. Inside this layout, the user can define the different areas or districts of the plant and associate each of them with the corresponding SLAM map used by the AMR.
The purpose of building layout management is to provide a clear and consistent representation of the factory, ensuring that the Fleet Manager knows how the environment is structured and how the different robots are distributed across it. This information enables the system to coordinate navigation, mission assignment, and resource allocation correctly.
The user can create new buildings, import layout files, modify existing information, and remove elements that are no longer needed. Each layout contains specific details such as its name, code, associated building, status, and optional notes, allowing the facility to be organized in a structured and intuitive way.

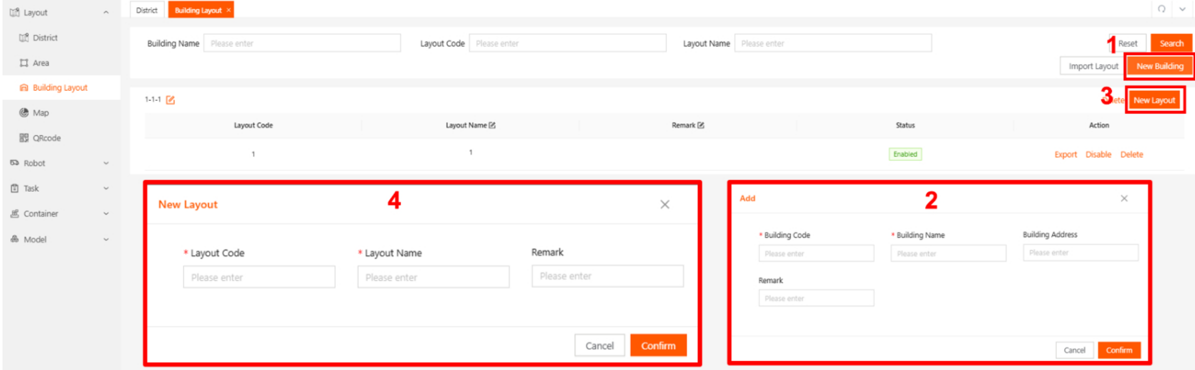


Figure 1.9: Building and Layout creation

1.4.1.2 District Management

In KUKA.AMR Fleet, the building layout is organized into districts, which represent different areas of the facility, such as halls, warehouse zones, or production sections.
Each district must be associated with a floor map, which represents the reference for AMR localization and navigation. These maps are created, as mentioned before, using KUKA.AMR Studio. Once the map is created and validated in the commissioning phase, it is uploaded to the Fleet Manager and assigned to the corresponding district.
Through the district configuration, the system combines information from the building layout, the associated map, and the district properties, allowing the AMR to correctly position itself, interpret the environment, and follow predefined routes or mission paths.

Figure 1.10: District Creation

1.4.2 Graph Editor

The Graph Editor is the tool used to define the navigation graph of a district based on the uploaded layout map. It allows users to draw nodes, edges, and related configurations that determine how AMRs move within the environment. Through this interface, motion paths can be created interactively by selecting a starting point and an endpoint on the map. Each motion path consists of two nodes connected by an edge, all of which can be parametrized to influence the behavior of the vehicle.

Nodes can be configured with basic and advanced parameters. Basic parameters define how the AMR behaves when reaching that point on the graph, while advanced functions allow the node to represent specific operational points such as rack nodes, parking nodes, charging stations, bracket nodes, or other functional positions. Nodes may also be associated with objects, such as container models, depending on the needs of the application.

Similarly, edges can be configured to control how the AMR moves between nodes. Standard edge parameters influence the robot's motion along the path, while advanced parameters allow the user to specify the vehicle size or adjust safety fields, ensuring that the AMR navigates safely in narrow or constrained areas. The resulting navigation graph coordinates the behavior of the AMRs and serves as the basis for mission execution within the Fleet Manager, providing a consistent and structured representation of all valid motion paths in the district.

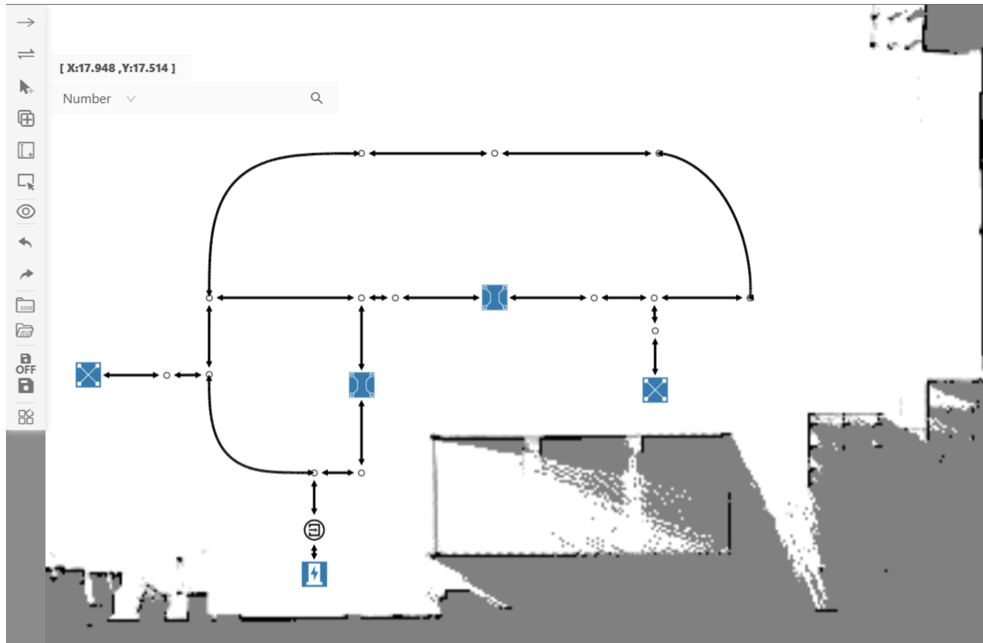


Figure 1.11: Graph overview

1.4.3 Model Management

Model management is used to configure:

- **Container model**: Used to set the specification and dimensions of the containers and the applicable types of AMRs.
- **Fine localization model**: It supports object model recognition and reflective sticker recognition.

Loads carried by the AMR can be of different types:

- **Rack**: Load in which the support frame is moved with it. The robot transports the load including the frame.
- **Holder**: Support frame fixed in place. The robot moves into the frame and only picks up the load.
- **Tray**: Load picked up in case of a stationary support frame.

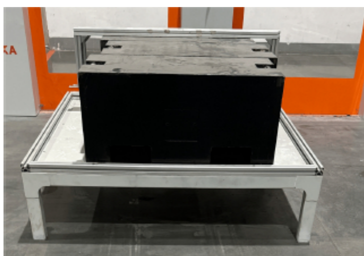


Figure 1.12: Rack

Figure 1.13: Holder

Figure 1.14: Tray

A model must be associated with every type of load, the definition of the model must be done in KUKA.AMR Studio and then imported in KUKA.AMR Fleet.
In the model definition the following parameters must be specified:

- Model type (rack, holder or tray)
- Approach direction of the AMR
- Length and width of the support
- Length and width of the legs

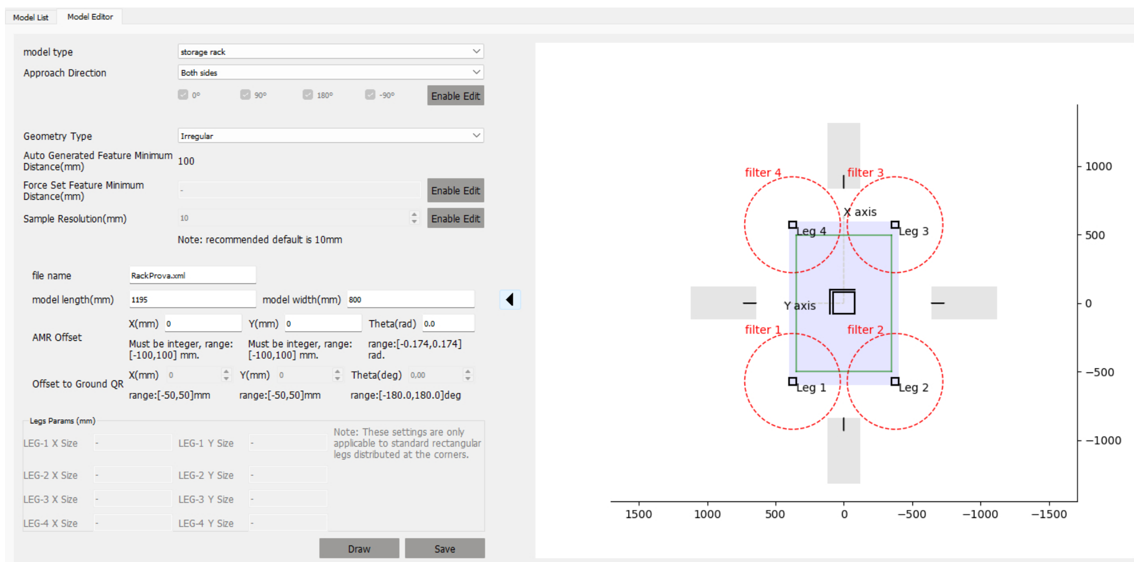


Figure 1.15: Model editor

1.4.3.1 Fine Localization with Reflectors

Reflective stickers can be used to provide faster and more accurate fine localization. This is due to their high-intensity reflectivity. For improved accuracy, additional reflective stickers can be used in the rear legs of the bracket or rack. The front stickers guide the KMP into the rack/bracket while the rear stickers determine its final position.
In this project, reflective stickers are used to align of the AMR with the loading ramp, ensuring a stable and repeatable entry trajectory before entering into the container.

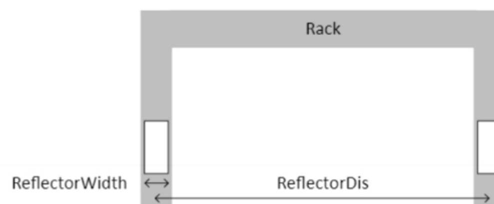


Figure 1.16: Stickers on rack

Similar to the container, fine localization with reflectors must first be defined in KUKA.AMR Studio and then imported in KUKA.AMR Fleet.
The following parameters must be specified:

- Length and width of the rack
- Minimum and maximum detection distance
- Reflector width and distance
- Target offset in the stickers' coordinate system

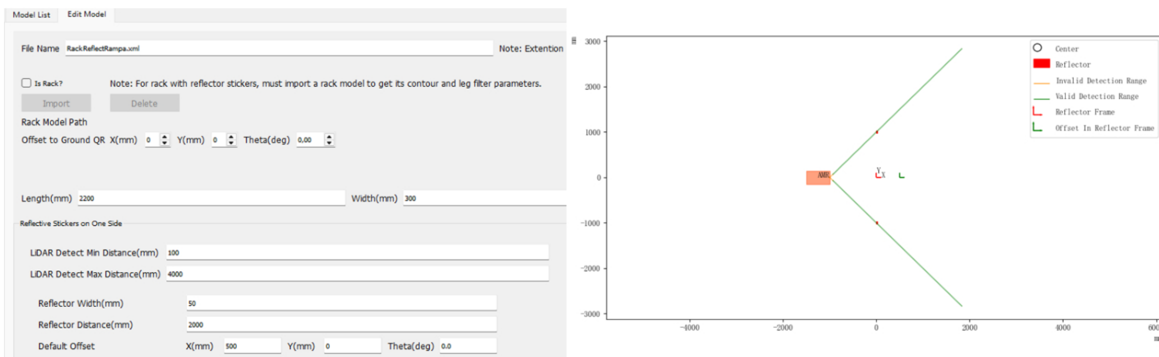


Figure 1.17: Reflective sticker model editor

1.5 Tests with KMP 1500P-EU-D diffDrive

A series of tests were carried out to evaluate whether the KMP 1500P could enter and operate inside a shipping container, which represents the main requirement for the depalletization application. The goal of these tests was to ensure that the AMR could correctly align itself with the loading ramp and move straight into the container, even when the container was not positioned perfectly in its expected location. In real conditions, a container may be slightly rotated or shifted forward, backward, or sideways, and the AMR must compensate for these variations to approach the ramp accurately.
A second objective was to verify the robot's ability to navigate inside the container once boarding was completed. The AMR must be capable of moving along the narrow internal corridor to transport the robotic arm toward the back of the container, allowing the system to gradually unload all the boxes.
These results provide essential insights into the feasibility of using the KMP 1500P for automated container depalletization and help identify the most reliable configurations for real industrial deployment.

Figure 1.18: AMR for tests

1.5.1 Map Creation

After configuring the wireless network and the Fleet Manager, the first step is to create the map of the building. Using the jogging function in KUKA.AMR Studio or the remote controller, the robot is manually driven through the area so it can scan and record the environment. Once the mapping is completed, the map can be refined using the Map Edit tools available in KUKA.AMR Studio. Next, the map is imported into KUKA.AMR Fleet, where it is assigned to a district. At this point, we can build the navigation graph by creating the required nodes and paths.

Figure 1.19: Map of the building

1.5.2 Navigation Graph

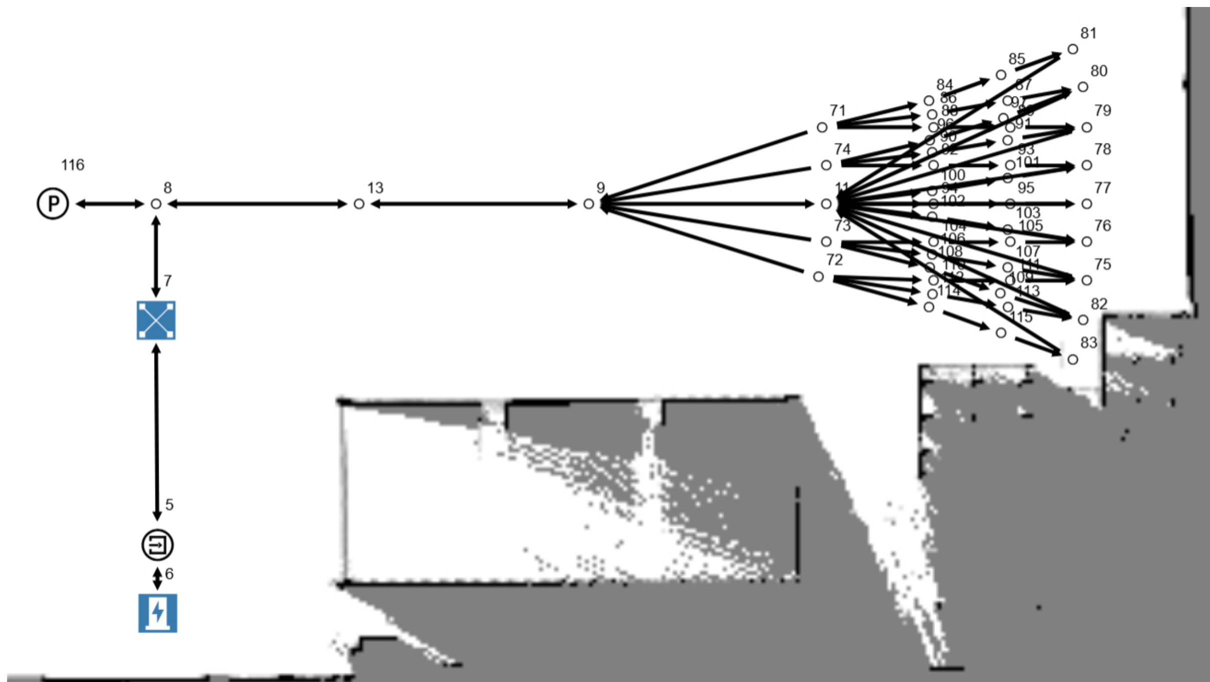


Figure 1.20: Graph for tests

The navigation graph was designed to simulate the complete sequence of operations required for the AMR to enter and exit the container during the depalletization process. The graph includes all the key navigation points and transitions that the vehicle must follow, allowing the Fleet Manager to execute missions that accurately reproduce the real operating scenario.

A charging node is included in the graph, enabling the AMR to return to its recharging station when needed, while a dedicated parking node serves as the default waiting position when no missions are assigned. A rack node is also present, this is the point where the robotic arm is loaded onto the AMR before entering the container and unloaded once the depalletization process is completed.

The approach path toward the container begins with node 9, which marks the start of the loading ramp. Depending on the rotation and position of the container, the end of the ramp and the entry point into the container correspond to one of the nodes 11, 71, 72, 73 or 74.
Inside the container, the graph expands into a set of nodes that represent the different orientations and intermediate checkpoints the AMR may assume while navigating the narrow internal corridor. These nodes define the possible trajectories the robot can follow to progressively advance toward the back of the container. Once the AMR reaches the final nodes corresponding to the deepest point of the container, the graph also provides the return paths that allow the vehicle to reverse its route, exit the container, and return toward the external working area.

1.5.2.1 Charging Node

Figure 1.21: AMR charging

The charging procedure is managed through a dedicated *Charge Enter Node*, which allows the AMR to align itself correctly before entering the charging station. This node is configured by selecting the charging function and specifying the associated charging point, the charging direction, and the robot type allowed to charge. During operation, the AMR stops at an appropriate distance from the station and approaches it with the correct orientation, defined by the *AMR Stop Angle* and the *Charge Direction*. Rotation at the entry node is disabled to ensure precise placement, and the robot's forward or backward approach is set by adjusting the *Robot Direction* along the graph path.
The correct charging direction depends on the AMR's orientation on the map and ensures that the rear of the vehicle is aligned with the charging interface.

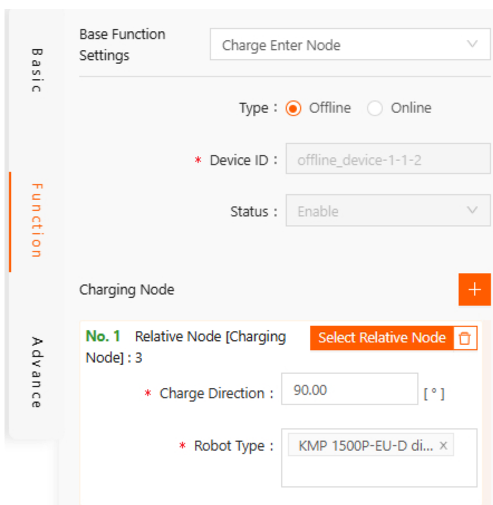


Figure 1.22: Function parameters

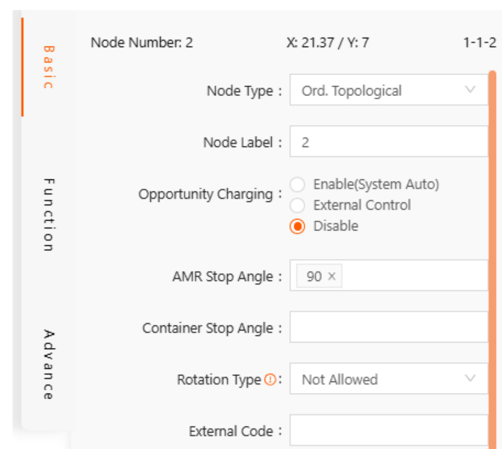


Figure 1.23: Basic parameters

1.5.2.2 Rack Node

Figure 1.24: AMR with rack loaded

In this phase, the rack node was created to provide a fixed location where the robotic arm could be loaded at the start of the process and unloaded once the depalletization cycle was completed. A rack model was created with a length of 1195 mm and a width of 800 mm, then imported into KUKA.AMR Fleet and placed on the map for testing purposes.
The rack model was assigned to the node, and its rotation was set to *Not Allowed* to ensure that the AMR approaches it by moving straight forward and exits by reversing.

1.5.2.3 Parking Node

The parking node is a designated standby position used by the AMR whenever it remains idle for a predefined amount of time. In this project, if the vehicle does not receive new missions for 10 seconds, the Fleet Manager automatically sends it to the parking node. This ensures that the robot does not block operational areas, avoids unnecessary occupation of task nodes, and stays in a safe and known location while waiting for its next assignment.

1.5.2.4 Ramp Alignment using Reflective Stickers

To verify whether the AMR could correctly center itself and align with the loading ramp, a series of tests were carried out using reflective stickers as artificial reference points. Since the real ramp was not available during the testing phase, the setup was simulated using four cardboard boxes positioned to match the actual dimensions of the ramp, approximately 2 m in width and 3 m in length. These boxes represented the four posts mounted on the sides of the real structure. Reflective stickers were applied to the front and back of each box, with a fixed width of 5 cm and a predefined length, creating the visual markers required for fine localization and alignment.

Figure 1.25: Boxes configuration for ramp simulation

These boxes were positioned near node 9, which marks the beginning of the ramp, and near the various possible entry nodes of the container (71, 74, 11, 72, or 73), depending on the container's orientation. The previously created sticker model was imported into KUKA.AMR Fleet and associated with these marker positions.
In the navigation graph, Fine Position Guidance was added as an advanced configuration on the approach paths, both for entering and exiting the ramp area. This allowed the AMR to detect the reflective stickers during motion and refine its lateral and angular alignment with the simulated ramp.

Figure 1.26: Sticker applied on box

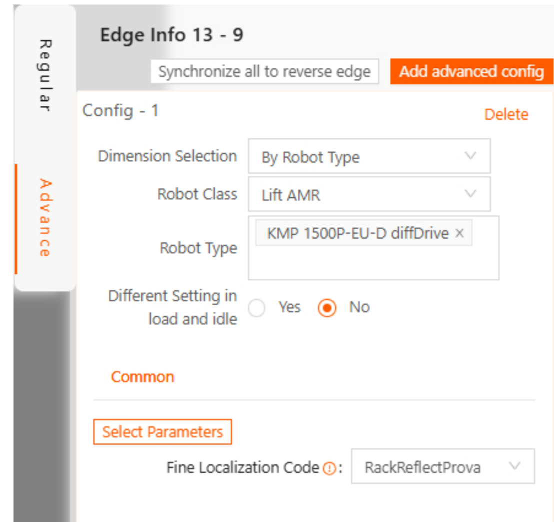


Figure 1.27: Edge settings

1.5.3 Container Internal Navigation

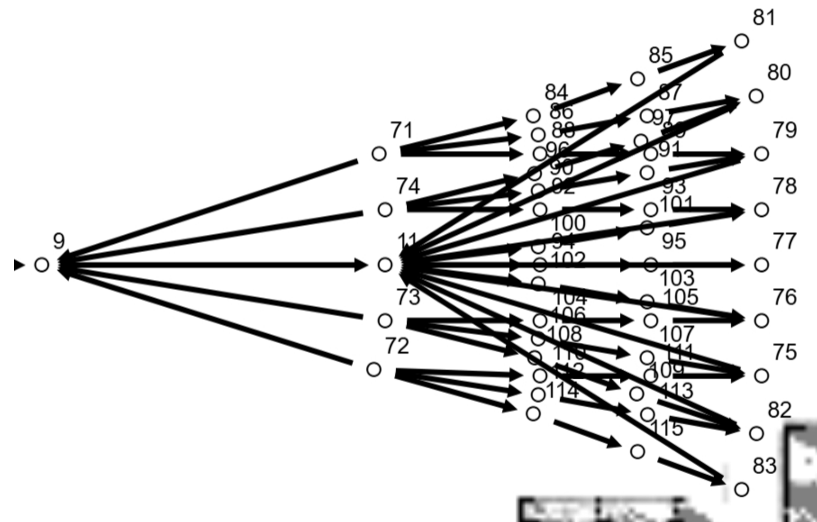


Figure 1.28: Ramp and Container nodes

To enable navigation inside the container, a set of nodes was created to define the AMR's possible trajectories once it has completed the ramp alignment. After determining which entry node corresponds to the current orientation of the container (one among nodes 11, 71, 72, 73, or 74), the next step is to identify the node representing the back of the container. This end-point can be any of the nodes from 75 to 83, depending on the container's position and rotation. Once these nodes are defined, the Fleet Manager automatically calculates the path between the entry node and the selected end-point.

From each possible container entry node, three different internal paths were created, each composed of three intermediate nodes spaced approximately one meter apart. These paths represent the different orientations the AMR may need to assume while progressing toward the back of the container, ensuring sufficient flexibility for navigation in a narrow, constrained environment.
All paths leading toward the back of the container were configured with an advanced Obstacle Detection Plan. This configuration reduces the AMR's safety fields to allow the robot to detect boxes at a distance of roughly one meter, even when operating in very tight spaces. The Obstacle Detection Plan is defined within the Fleet Manager by adjusting the frontal, rear, lateral, and vertical safety distances. This ensures that the AMR maintains safe movement inside the container while still being able to detect potential obstacles early, supporting reliable operation during the depalletization process.

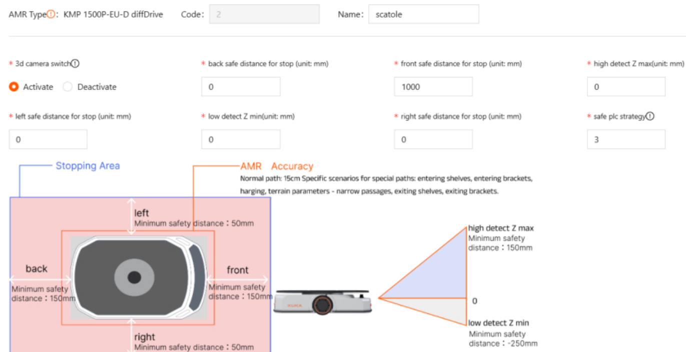


Figure 1.29: Obstacle Detection Plan

Figure 1.30: Edge settings

1.5.4 Python Modules

In this project, the AMR can be remotely controlled through the official KUKA Fleet Manager APIs, which accept commands and missions in the form of HTTP requests encoded as JSON messages. To automate the entire depalletization process, a set of Python modules was developed to structure the communication with the Fleet Manager and implement all the required logic. These modules send correctly formatted API requests, monitor the robot's state in real time, compute navigation targets, and execute the full depalletization routine in a fully automated way. Their roles are summarized below:

- `client.py`: This module implements the `FleetClient class`, which handles all HTTP communication with the KUKA Fleet Manager. It manages a persistent session, sets standard request headers, and provides generic GET and POST methods used by the entire system to send missions, retrieve robot information, and interact with the Fleet Manager API.

- `commands.py`: This module defines the `MissionAPI class`, which provides high-level functions for sending missions and commands to the KUKA Fleet Manager. Each method builds the required JSON payload and corresponds to a specific operation, such as robot movement, load/unload actions, charging, mission cancellation, or container handling. It serves as the command layer of the system, allowing the workflow to control the AMR through simple and structured function calls. Below is showed an example of a JSON payload for an API command that implements a simple movement to a defined node.

```python
def move_to()
    # Create the full JSON payload
    mission: Dict[str, Any] = {
        "orgId": org_id,
        "requestId": request_id,
        "missionCode": mission_code,
        "missionType": "MOVE",
        "viewBoardType": "",
        "robotType": robot_type,
        "robotModels": [robot_model],
        "robotIds": [robot_id],
        "priority": priority,
        "templateCode": "",
        "lockRobotAfterFinish": lock_after_finish,
        "unlockRobotId": "",
        "unlockMissionCode": "",
        "idleNode": "",
        "missionData": [
            {
                "sequence": 1,
                "position": target_node,
                "type": "NODE_POINT",
                "passStrategy": pass_strategy,
                "waitingMillis": waiting_millis,
            }
        ],
    }
```

- `robot_state.py`: This module continuously retrieves the real-time state of the AMR by periodically querying the Fleet Manager. Through the `update_robot_state()` thread, it extracts information such as position, orientation, status, current node, battery level.

- `utils.py`: This module defines the `Utils class`, which groups together several helper functions used for map handling, path planning, and container navigation.

- `workflow.py`: This module implements the main control workflow that coordinates all steps required for the automatic container emptying routine. It loads the map and node information, starts the real-time robot state monitoring thread, and then executes a state machine that sequentially manages the entire depalletization process.

1.5.5 Depallettization Workflow

1.5.5.1 Initialization of Map Data and Robot State Monitoring

```
# Extract node data from the JSON file
json_file = "map_1_3_v4n.json"
data, nodes = utils.jsonextraction(json_file)

# Loop that updates the robot parameters
utils.update_robot_state()
```

At the beginning of the workflow, the system loads the map information exported from the Fleet Manager. The JSON file contains the list of nodes, their coordinates, and the structure of the navigation graph. Through the `jsonextraction()` function, the workflow extracts both the full JSON data and a simplified list of node coordinates, which will later be used for target computation and path planning.

After loading the map, the system starts a background thread that continuously updates the robot state by querying the Fleet Manager at regular intervals. This thread retrieves real-time information such as position, orientation, status, current node, and battery level. These parameters are essential for all subsequent operations of the workflow.

1.5.5.2 Step 1: Rack Lifting

In the first state, the workflow starts the lifting of the rack mounted on the AMR. This is achieved by sending a `robot_lift` command to the Fleet Manager, instructing the vehicle to raise its lifting platform while keeping the rack attached. After the command is issued, the system waits briefly and then continuously monitors the robot status through the real-time state update thread. When the AMR reports that the operation has been completed (status code 3), the workflow transitions to the next state, where the rack is positioned at the container entrance.

```
while True:
    if state == 1:
        print("STATE 1: RACK LIFTING")

        response = mission_api.robot_lift(
            robot_id="1",
            move_lift=1,
            container_code="1"
        )
        time.sleep(2)

        while True:
            if robot_state.status == 3:
                state = 2
                breaks
```

1.5.5.3 Step 2: Rack Placement

In the second state, the workflow commands the AMR to place the rack at the designated node located at the entrance of the container. This is performed by sending a `robot_drop` mission to the Fleet Manager, specifying the target node where the rack must be deposited. After issuing the command, the system waits and continuously checks the robot status.

1.5.5.4 Step 3 and 4: Container Position Update

In the third state, the workflow updates the container information stored in the Fleet Manager. This is done by issuing a `container_out` command, which removes the existing container entry associated with the current process. This step is made because, after releasing the rack, the AMR's position changes and the container must be re-registered at the correct node in the following state.
In the fourth state, the workflow re-inserts the container inside the Fleet Manager at the node closest to the AMR's current position. After releasing the rack, the robot no longer corresponds to the previously stored container position; therefore, a new insertion is required to maintain an accurate representation of the container location within the system.

```
def closest(self, x: float, y: float, nodes):
    return min(nodes, key=lambda n:
        math.hypot(n["x"] - x, n["y"] - y))
```

To determine the correct node, the workflow computes the node closest to the AMR's real-time coordinates. This is done through the `closest()` function, which compares the robot's position

with all available nodes and returns the one with the minimum Euclidean distance. This ensures that the container is placed at the node that best corresponds to the actual AMR location inside the map.
Once the closest node has been identified, the workflow sends a `container_in` command to insert the container at the computed node.

1.5.5.5 Step 5: Target Computation

In the fifth state, the workflow determines the next target point that the AMR should reach inside the container. This target is computed as a point located three meters in front of the robot, based on its current position and orientation. The purpose of this step is to identify a meaningful forward direction that the AMR can follow during the path-planning process, ensuring that navigation proceeds naturally along the container's depth.

```
def compute_target(self, x, y, theta):
    theta_rad = math.radians(theta)
    xtarget = x + 3 * math.cos(theta_rad)
    ytarget = y + 3 * math.sin(theta_rad)
    return xtarget, ytarget
```

Once the target coordinates are calculated, the workflow identifies the graph node closest to this point. This is achieved using the same nearest-node search described in the previous state, ensuring that the target is mapped to an actual navigable node of the environment.

1.5.5.6 Step 6: Best Path Computation

In the sixth state, the workflow computes the optimal navigation path that the AMR must follow to reach the target node identified in the previous phase. The process begins by defining the starting node (the one closest to the robot's current position) and the goal node (the one closest to the computed target point). To build the navigation graph, the workflow extracts all nodes and edges from the JSON map using the `path_info()` function. Once the graph structure is available, the system applies the Dijkstra algorithm to calculate the shortest path between the start and goal nodes.

```
Dijkstra(start, goal, graph):
  Initialize dist[start] = 0, all others = inf    # initial distances
  prev[start] = NULL    # predecessor map
  pq = priority queue with (0, start)    # nodes sorted by distance

  while pq not empty:
```

```
    (d, u) = extract_min(pq)   # get closest unvisited node
    if u = goal: break   # stop early if goal reached

    for each neighbor (v, w) of u:   # explore outgoing edges
        if d + w < dist[v]:   # shorter path found
            dist[v] = d + w   # update distance
            prev[v] = u   # store predecessor
            push (dist[v], v) into pq   # reinsert node

Reconstruct path by backtracking from goal using prev[]
return path
```

The result is a sequence of connected navigation nodes that the AMR can follow safely and efficiently.

1.5.5.7 Step 7: Container Emptying

In the seventh state, the workflow executes the sequence of movements required for the AMR to progress inside the container and perform the emptying routine. This logic is implemented in the `containernav()` function, which receives the previously computed navigation path and commands the robot to move step-by-step through the nodes that lead toward the back of the container.

```
containerNav(path):
    cont = 0   # flag for first movement

    for each node in path[1:]:   # skip first node
        wait until robot_state.status = READY   # status 3

        if cont = 0:   # first movement in the sequence
            target = node
            send rack_move(pick = path[0], place = target)
            cont = 1
        else:   # next movements
            target = node
            send rack_move(pick = current_node, place = target)

        # monitor mission execution
        while robot not READY:
            if error == "noError-obstacle":   # obstacle detected
                cancel current mission
```

```
            # temporarily drop the rack to allow box removal
            send robot_drop(at = current_node)
            wait until boxes are removed   # simulated wait

            # resume movement after obstacle handling
            send rack_move(pick = current_node, place = target)

        if robot reached target:
            break   # proceed to next node
```

The function sends a series of `rack_move` missions: the first one moves the rack from its initial position to the first node of the path, while the following commands move the AMR from one node to the next. During navigation, the system constantly checks the robot's status and monitors for obstacles.
If an obstacle is detected, a layer of boxes, blocking the path the current mission is cancelled, the rack is lowered to allow box removal, and the system waits for the layer to be cleared. After the simulated picking delay, the AMR lifts the rack again and continues toward the next node in the sequence.
This state continues until all nodes in the path have been visited, meaning that the AMR has reached the back of the container and completed the emptying cycle.

1.5.5.8 Step 8: Return to Starting Position

In this state, the AMR completes the container-emptying sequence and must return to its initial working position. Once the robot has reached the last node in the computed navigation path (stored in the variable `goal`), the workflow commands a final `rack_move` mission to bring the rack back to the rack node.
This step ensures that the robot exits the container, aligns itself correctly with the outbound path, and restores the rack to a known, fixed location. This allows the next cycle either another unloading iteration or a charging to begin from a consistent and predictable state.

1.5.5.9 Step 9: Recharging

In the final state, the workflow checks the battery level of the AMR to determine whether a charging procedure is required. If the battery level drops below a predefined threshold (50% in this implementation), the system automatically issues a charging request by sending a charge mission to the Fleet Manager.
The robot is instructed to move to the charging node (in this case, node 1-3-6), align with the charging station, and begin the recharging process. This ensures that the AMR remains operational and ready for the next unloading cycle without manual intervention.

1.5.6 Final Considerations

The complete set of tests carried out on the depalletization workflow allowed us to evaluate both the reliability of the implemented Python routine and the behavior of the AMR inside a highly constrained environment such as a shipping container.

The alignment phase proved to be stable and repeatable: the AMR was able to correctly identify the approach nodes and align itself with the ramp orientation before entering the container. The customized obstacle-detection profile also proved effective. Cardboard boxes placed along the path were consistently detected as obstacles, triggering the emergency procedure, cancelling the mission, and correctly resuming operations after the rack was lowered and the "virtual unloading" phase was completed.

Navigation inside the container was successfully achieved through the node-based graph and Dijkstra path planning. However, the process highlighted a natural limitation of absolute node navigation: performing fine forward movements based solely on graph nodes is less intuitive than using relative motion commands (e.g., "move forward 30 cm"). A native relative-motion command in the Fleet Manager would significantly simplify short movements required during progressive unloading.

Overall, the tests confirmed that the workflow is functional and capable of managing the complete depalletization cycle, while also identifying potential improvements for future work.

2. Vision Systems

2.1 Operating Scenario and Vision Requirements

Automated depalletization inside a shipping container requires a reliable vision system capable of detecting boxes, estimating their pose, and providing a consistent picking sequence. For this reason, before analysing and comparing different vision solutions, it is important to outline the characteristics of the working environment and the constraints imposed by the task.

The tests were carried out inside a standard 20-foot container, with internal dimensions of approximately 2.3 m × 2.3 m × 6 m. The boxes to be detected and picked may vary significantly in size, ranging from a minimum of 300 × 300 × 200 mm to a maximum of 900 × 600 × 500 mm, with the most common format being 600 × 400 × 400 mm. Several visual disturbances may also be present: boxes may differ in color and material and may include tape, labels, printed text, or stains, all of which can affect detection accuracy.
The vision system is mounted on the wrist of the robotic arm and must identify the boxes using AI-based algorithms, returning both their pose and a coherent picking order. The unloading strategy requires a strictly top-down approach to prevent boxes from falling; the lateral picking direction (left to right or right to left) is not relevant. A major challenge is ensuring that the container walls are not mistakenly recognized as boxes and that the narrow spaces between the load and the walls are correctly ignored.

During the project, several vision systems from different developers were tested and compared, including solutions from IT+ Robotics, Photoneo and SICK. Each system was evaluated under the same operating conditions to assess its suitability for the depalletization task.
To compare the performance of these vision solutions, several criteria were considered: the correctness and stability of the proposed picking order, the ability to manage ambiguous or intermediate cases (such as a box partially located between two quadrants), robustness against the presence of container walls, and tolerance to variability in box dimensions, colors, textures, and surface inconsistencies.
Among the tested solutions, the IT+ Robotics system demonstrated the best overall performance and was therefore selected as the preferred vision system for the application.

2.2 IT+ Robotics

The vision solution selected for the project is the **Smart Pick 3D Flex** system developed by IT+ Robotics. The software uses 2D images and high-quality 3D point clouds to detect each box, determine its position according to sorting rules defined by the user, and compute a collision-free trajectory for the manipulator.

The system uses a **Zivid 2+ L110** structured-light 3D camera, which produces high-resolution RGB images and dense, accurate point clouds. Thanks to its compact and rugged design, the camera can be mounted directly on the robot wrist, enabling precise perception even during fast movements.

Figure 2.1: Zivid 2+ L110

2.2.1 Graphical User Interface

2.2.1.1 FrontEnd

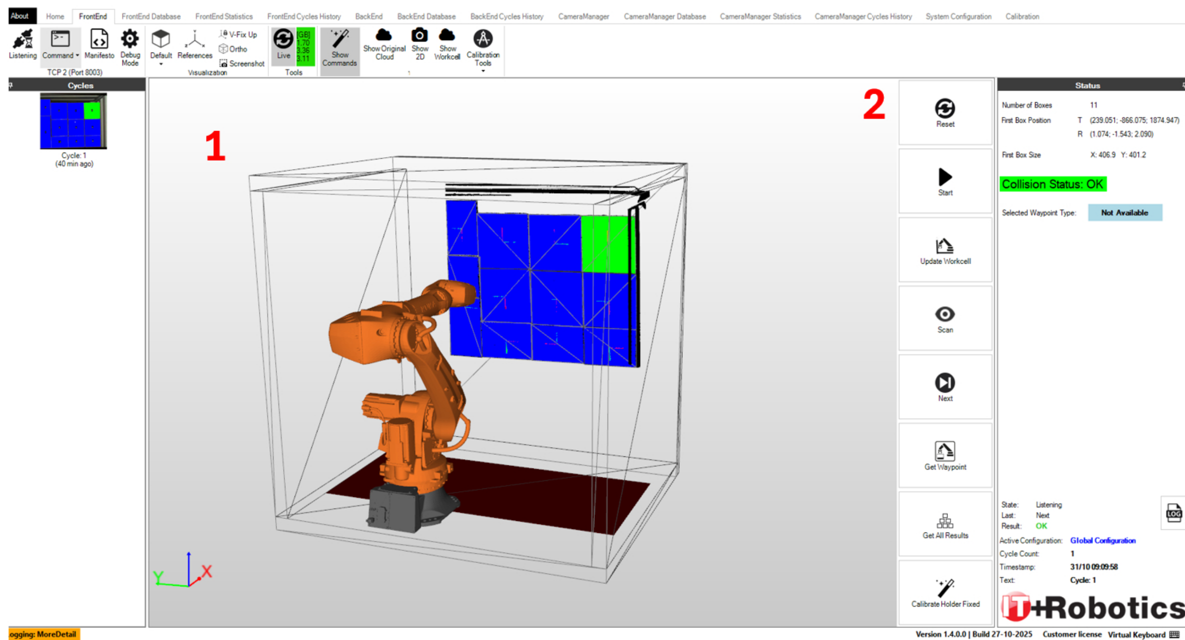


Figure 2.2: Front End

The **FrontEnd** section is where acquisitions can be performed.
The main screen (1) shows a simulation of the environment, while the right column (2) contains the available commands used to execute the different operations.

To perform a scan, the following sequence of commands must be executed:

1. **Reset**: Resets all processes and sensors, reloads the workcell, and performs an initial check to ensure that all required components are available.

2. **Start**: Starts the process on the selected bin. It is also possible to specify the box dimensions; in our case, since they vary, the default value of 0 is used.

3. **Scan**: Performs a scan on the selected bin using the configured camera parameters. There is also an option that can be used to acquire additional images, which is useful when multiple scans must be combined to cover a larger area.

4. **Next**: Returns the next available result for the selected bin. It is also possible to define an initial configuration for the six robot axes.

   To complete a scan, these four commands are sufficient; however, the following may also be useful:

- **Update Workcell**: Updates the robot axis values in the simulation, ensuring that the virtual representation always matches the robot's real position.
- **Get Waypoint**: Plans the trajectory to reach the pick point and returns the subsequent robot waypoints. The software computes three waypoints in total (approach, pick, and exit); this command must be called repeatedly until all trajectory points have been generated.

There is a **FrontEnd Database** section where it is possible to configure the box sorting criteria, generate gripping points, and modify the approach and exit points used in the robot trajectory.
In our case, the highest boxes were assigned the highest priority. As a second criterion, when two boxes have the same height, the one with the lowest Y value is selected, meaning the box located further to the right. If two boxes fall within the same height range and lateral position, a third criterion based on depth is applied: a lower X value corresponds to a position closer to the front of the container. Note that the axis orientation may vary depending on the system configuration.
To configure the gripping points, it is possible to define how many gripping options are available for a single object by assigning values for Z-axis rotation, inclination, and position. Each generated point can also be assigned a priority based on its effectiveness.
Finally, the three points that form the gripping trajectory can be customized by adding offsets along the X or Z axis with respect to the box center.

There is also a **FrontEnd Cycles History** section that lists, in chronological order, all scans performed and saved on the PC. From this section, it is possible to access images, point clouds, and the results of the processing algorithms. It is also possible to select a previously acquired scan and reprocess it using different sorting criteria, algorithm parameters, or, more generally, a different system configuration.

2.2.1.2 BackEnd

The **BackEnd Database** section contains the settings used to configure the algorithms for box detection and gripping-point generation, based on the point cloud obtained from the scan.
The algorithm performs the following steps:

- **Cloud Segmentation 2D**: Starting from the acquired point cloud, the upper portion of the scene is extracted, limiting processing to the areas relevant for box detection.
- **Image Segmentation**: A segmentation algorithm is applied to the corresponding image to identify the regions that represent boxes.
- **Objects Creation**: An object is created for each segmented region.
- **Objects Merging**: Objects belonging to the same box are merged to avoid duplicates.

- **Pick Point Estimation**: The optimal gripping point is estimated for each detected object.

For each of these steps, it is possible to adjust various processing parameters depending on the application needs.

2.2.1.3 System Configuration

The **System Configuration** section is used to set up all components of the system, both physical (such as the robot, camera, and container) and logical (such as the FrontEnd and BackEnd processes), as well as to manage communication between the different elements.
Within the list of all configurable components, the *Workcell* section is particularly important, as it allows the configuration of the entire simulation environment. The main components of the workcell are:

- **Camera**: Allows the selection of the camera used for acquisitions. It is also possible to import the camera's CAD model and correctly position it inside the simulation.
- **Manipulator**: Allows the selection of the robot model; the software automatically places it in the environment.
- **Gripper**: Similar to the camera, the CAD model of the gripper can be imported, and the appropriate reference frame can be defined for trajectory computation.
- **Bin**: Defines the robot's and camera's usable workspace, setting its size, position, and orientation with respect to the global reference frame.
- **Transforms**: A key section for defining the position of all reference frames, including those of the camera, manipulator, gripper, and any additional elements in the simulation.

In this project, the simulation environment was configured to reproduce the internal dimensions of a standard shipping container. A cubic structure was created with an area of approximately $2.3\,\text{m} \times 2.3\,\text{m}$, representing the container's interior walls. Inside this structure, the Bin was defined as the effective workspace of the robot and the camera. For the tests, the Bin was positioned 1.5 m in front of the robot and given a depth of 1 m. In the image below, the Bin corresponds to the green area, which indicates the region where box detection and pick-point computation take place.

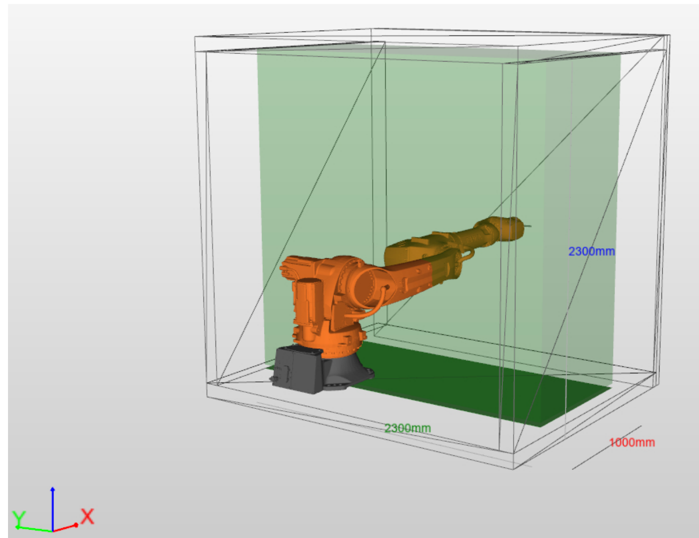


Figure 2.3: System Configuration

2.2.2 Tests

2.2.2.1 Setup

For the execution of the tests, the camera was mounted on a fixed support. The support was positioned so that the camera had a frontal distance of 1.5 meters from the boxes, a lateral distance of 0.7 meters from the container wall, and a height of 1.7 meters from the ground. This configuration made it possible to capture the upper-right area of the container.

2.2.2.2 Ordered Layout on a Single Layer

The first tests were carried out using an ordered arrangement of boxes, all placed on a single layer and with relatively similar dimensions.
The system was able to correctly detect and sort the boxes according to the configured criteria. Below are some examples of the results obtained during these tests.

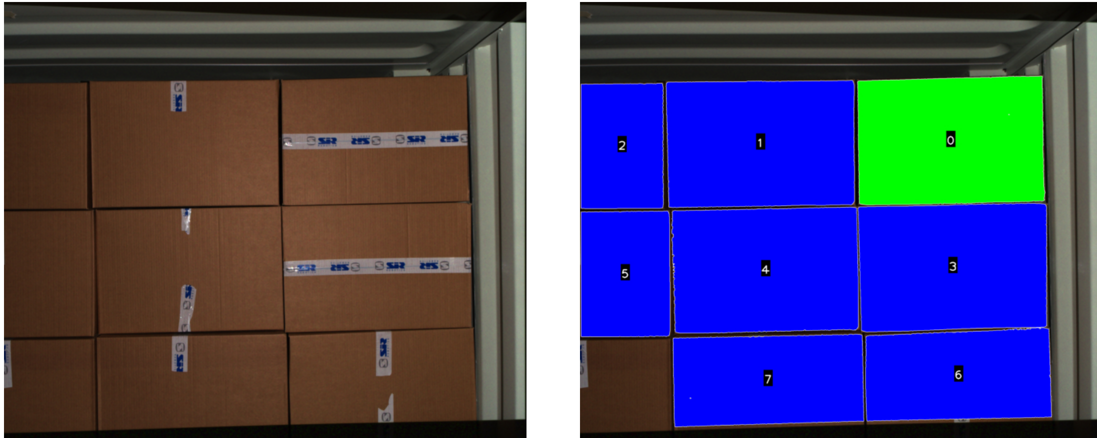


Figure 2.4: Test 1

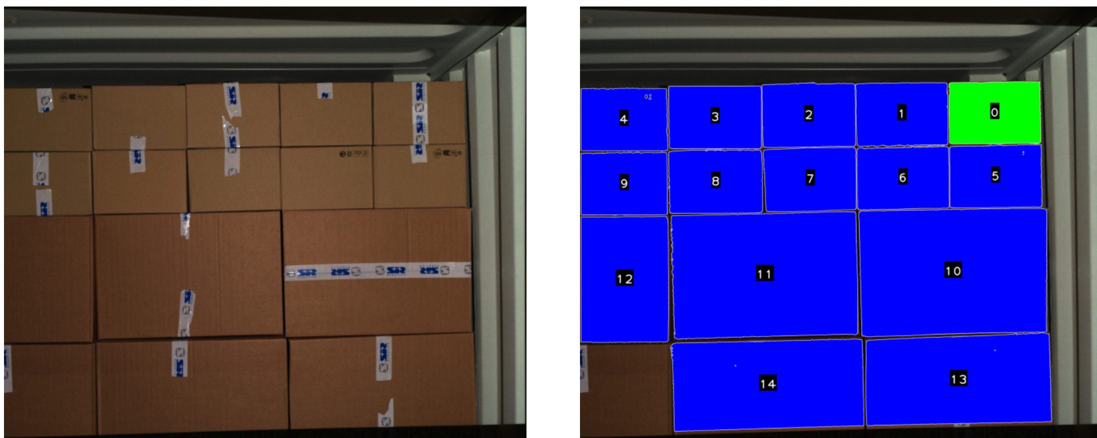


Figure 2.5: Test 2

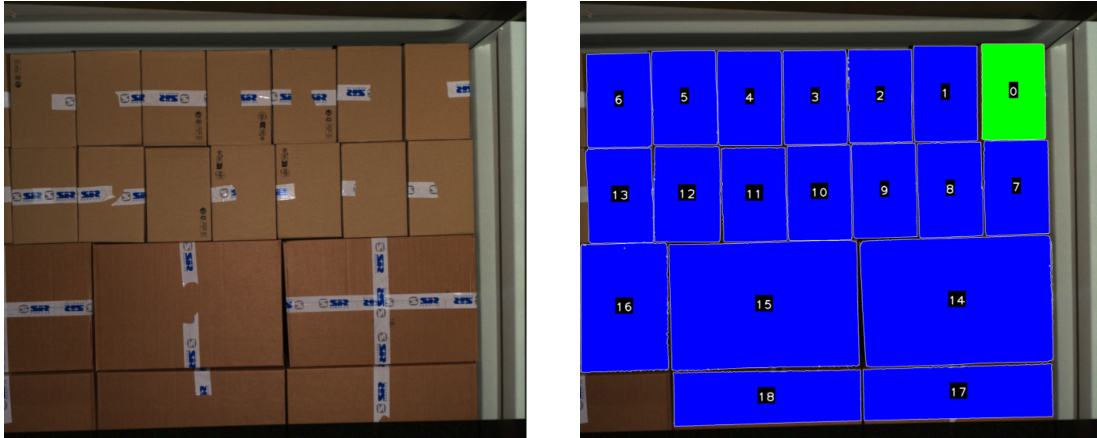


Figure 2.6: Test 3

2.2.2.3 Unordered Layout on a Single Layer

Next, configurations arranged on a single layer but with a disordered placement of boxes were tested. This type of experiment was conducted to verify the system's ability to correctly sort boxes even when pick points are located at different heights.

Also in this case, as shown in the images below, the system was able to provide an effective ordering for box picking.

Figure 2.7: Test 1

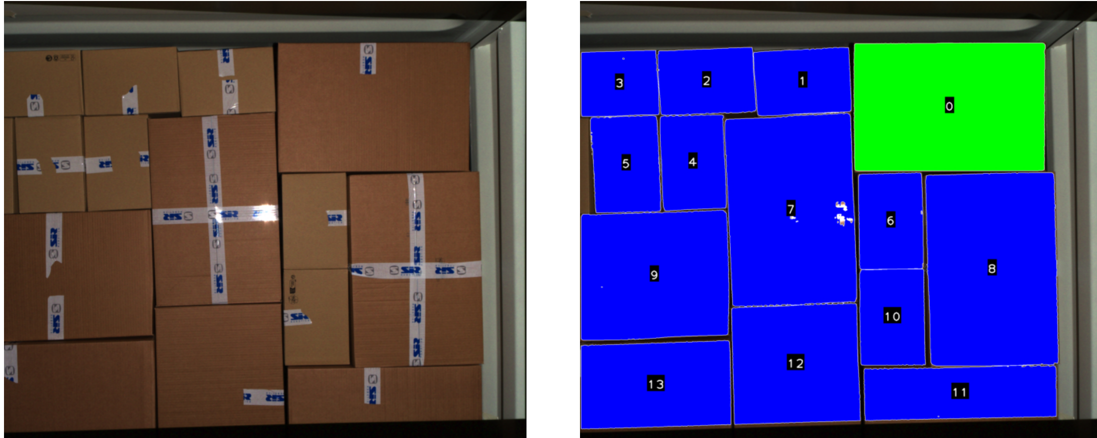


Figure 2.8: Test 2

2.2.2.4 Ordered Layout on Multiple Layers

After verifying the correct behavior of the system on a single layer, tests were extended to scenarios with multiple layers of boxes. This type of evaluation makes it possible to check that, when objects are arranged on different levels, the boxes in the rear layers are not incorrectly detected or given lower priority during the picking process.

In the example shown below, a multi-layer picking routine is simulated. As can be seen, the system detects and makes the boxes in the rear layers pickable only when they become fully visible (Step 3). When the boxes are partially occluded (Step 2), the system correctly ignores them, preventing incomplete or incorrect identifications.

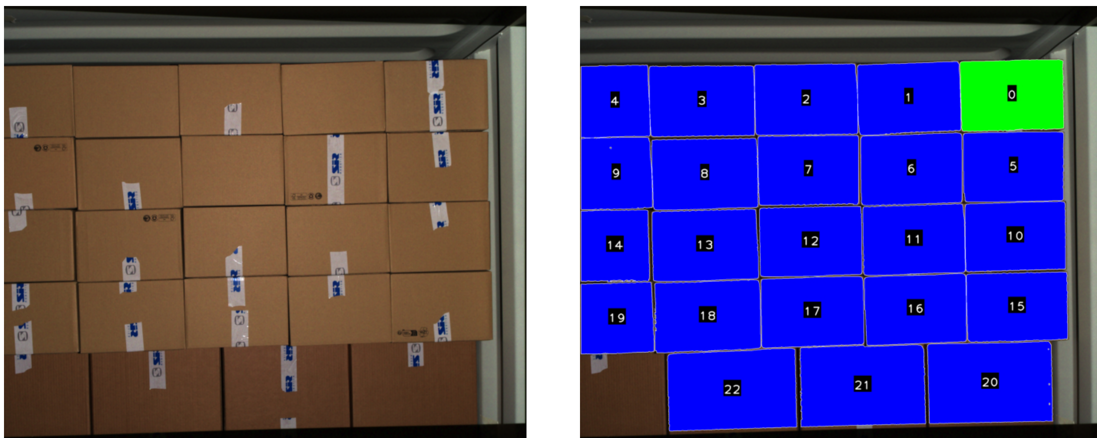


Figure 2.9: Step 1

Figure 2.10: Step 2

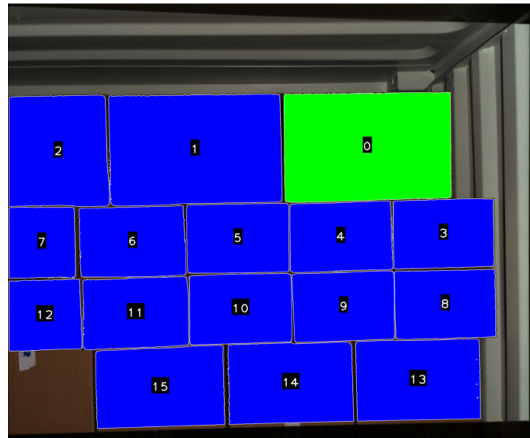


Figure 2.11: Step 3

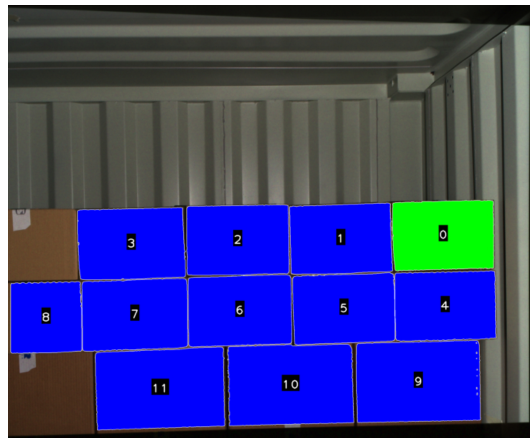


Figure 2.12: Step 4

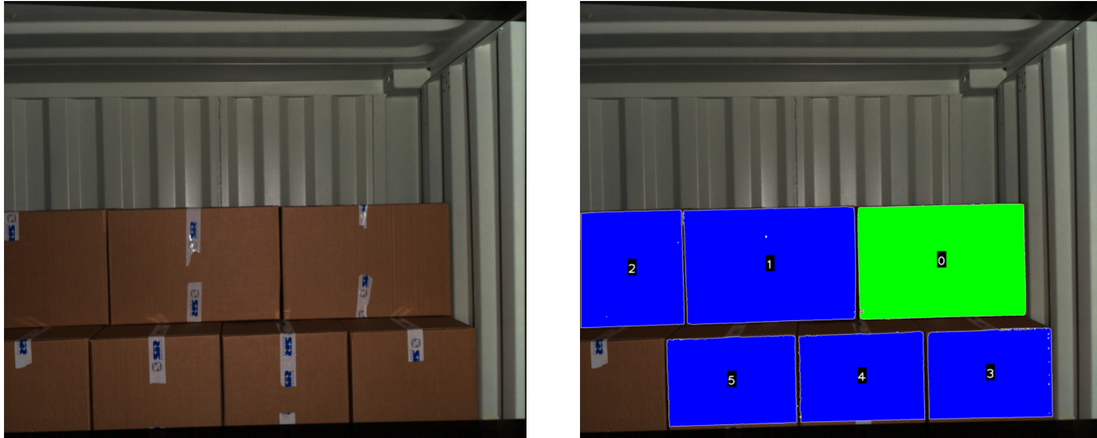


Figure 2.13: Step 5

2.2.2.5 Unordered Layout on Multiple Layers

In this phase, a higher level of complexity was introduced in the arrangement of the boxes. Multiple layers were created with a layout that was as random and disordered as possible, in order to evaluate whether the system would experience difficulties when processing more complex configurations.
In the example shown below, it can be seen that in some cases the system marks as graspable boxes that are only partially visible. For instance, box 5 in Step 2 and box 1 in Step 4 are detected as pickable even though they are only partially exposed. This behavior could be corrected by adding additional checks on the detected box dimensions; however, such verification cannot be performed directly by the vision software.

Figure 2.14: Step 1

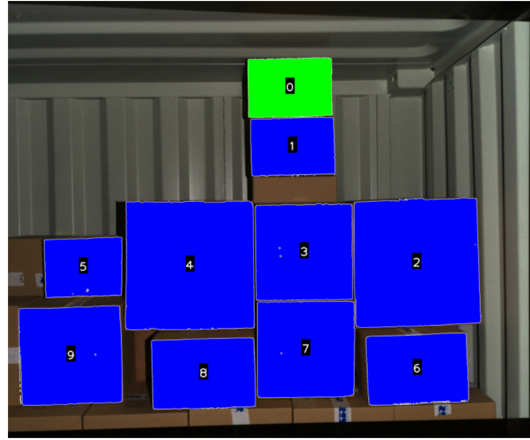


Figure 2.15: Step 2

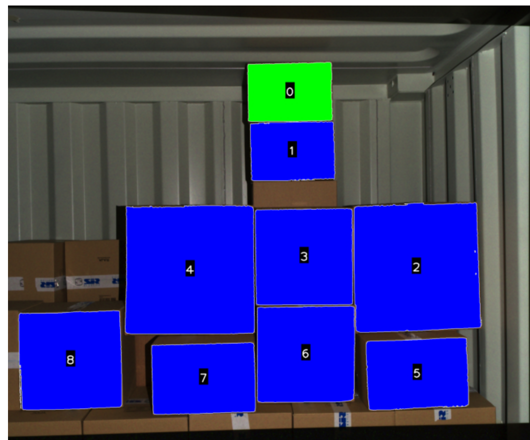


Figure 2.16: Step 3

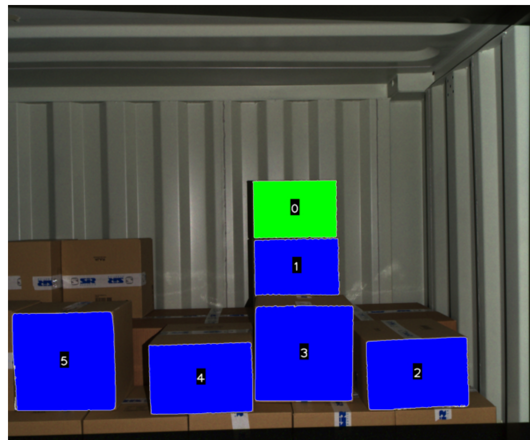


Figure 2.17: Step 4

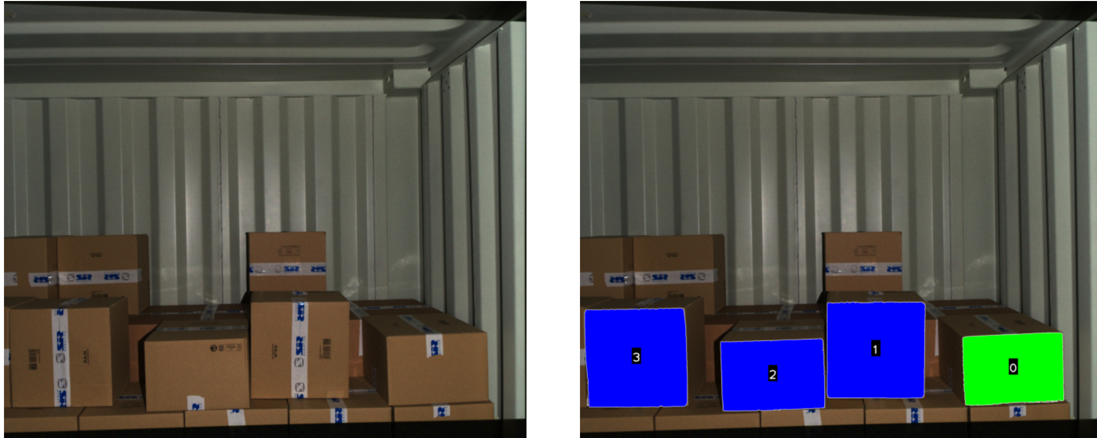


Figure 2.18: Step 5

2.2.2.6 Detection in the Presence of Visual Disturbances

In all tests performed up to this point, the boxes were presented in almost ideal conditions: the only potential obstacle for correct detection was the adhesive tape used for closing them.
The next step was to make the evaluation more challenging by introducing visual disturbances, simulating real situations in which boxes may have labels, printed text, or surface stains.
As shown in the tests, the system proved sufficiently robust when these disturbances were applied to large boxes (Test 1), while it showed some uncertainty in the case of smaller boxes (Test 2).

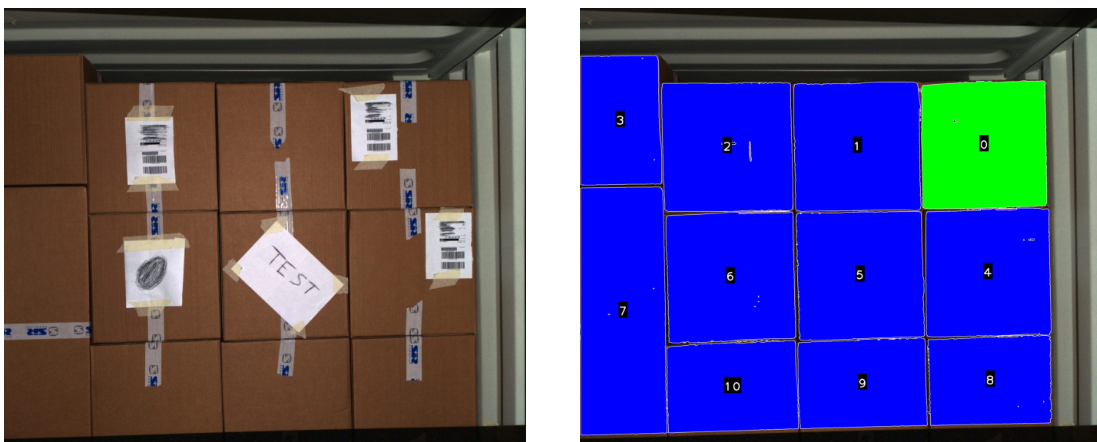


Figure 2.19: Test 1

Figure 2.20: Test 2

2.2.2.7 Pick Trajectory Simulation

As mentioned earlier, the software is able to compute a complete pick trajectory for each detected box. The trajectory is defined using three key points:

- **Approach Point**: used to safely reach the target area
- **Pick Point**: where the gripper makes contact with the box
- **Exit Point**: which guides the robot away after the grasp

By generating these waypoints, the software can simulate the robot's motion and provide a clear visualization of how the gripper moves around the box. This simulation is particularly useful for validating whether the planned motion avoids collisions, respects the workspace limits, and maintains a stable approach to the object. An example of the resulting trajectory visualization is shown below.

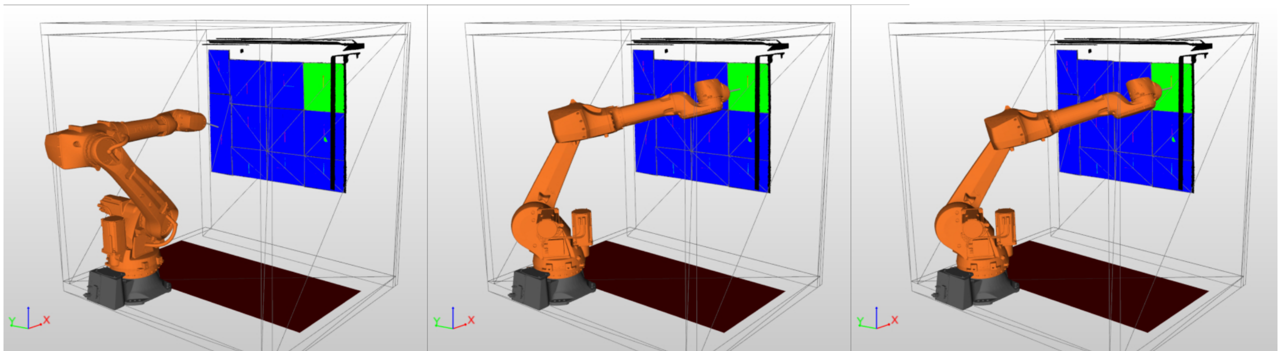


Figure 2.21: Starting Point   Figure 2.22: Approach Pose   Figure 2.23: Picking Pose

2.2.3 Robot Integration

2.2.4 Final Considerations

The vision system tested proved to be a complete and well-structured tool for detecting and ordering boxes inside the container. The separation between the FrontEnd and BackEnd components makes the software easy to understand and use, allowing parameter configuration and data processing to be handled independently.
The graphical interface is clear and intuitive. The different sections are well organized and provide quick access to the main functions.The graphical simulation of the workcell is really helpful for visualizing the robot's behavior and understanding the outcome of each operation.

From the tests performed, the system showed good reliability and accuracy. It successfully recognized and sorted boxes in various scenarios: from neatly arranged single-layer configurations to more complex, disordered multi-layer setups. It also demonstrated robustness against small visual disturbances, such as labels, printed text, or tape on the boxes. Some difficulties appeared only in cases involving very small boxes or partially visible ones, where detection may be less accurate.

2.3 Other Vision Systems Tested

2.3.1 Photoneo

The Photoneo vision system tested in this project was based on the **MotionCam-3D M**, a high-precision 3D camera designed for scanning medium-sized objects in industrial environments. The device uses Photoneo's *Parallel Structured Light* technology, which enables the acquisition of detailed point clouds even when objects are moving, without introducing motion blur. This makes it suitable for applications that require both high speed and high-quality 3D reconstruction.

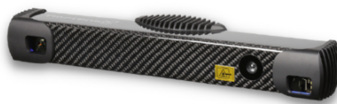


Figure 2.24: MotionCam 3D M

During the testing phase, several aspects of the Photoneo vision system were evaluated, including usability, workflow organization, and detection performance. The system is structured around two separate software tools: **PhoXi Control**, used exclusively for configuring the camera and managing acquisitions, and **Locator Studio**, a web-based application used to configure and

simulate the entire vision workflow.

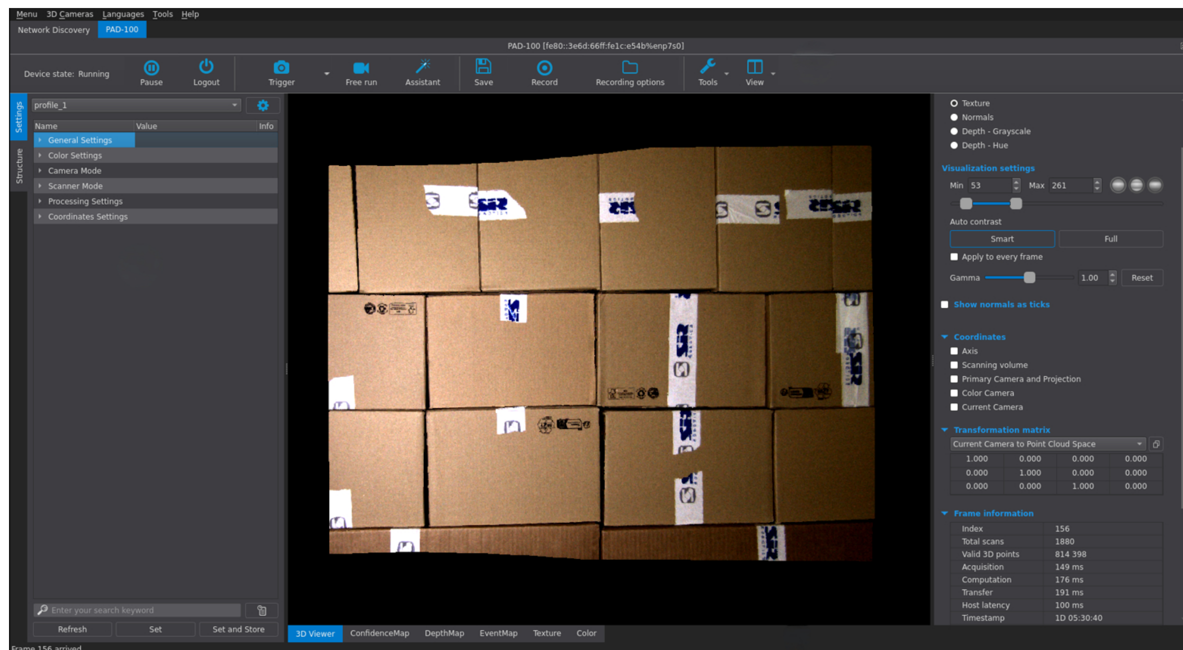


Figure 2.25: PhoXi Control GUI Overview

PhoXi Control proved simple and intuitive to use, thanks especially to the *Assistant* feature, which guides the user step by step through the configuration of the main acquisition parameters. Scans are saved directly through this software and can later be imported into Locator Studio, allowing quick and flexible simulation of the picking process.

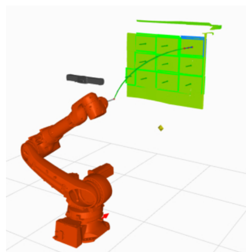


Figure 2.26: Pick Trajectory

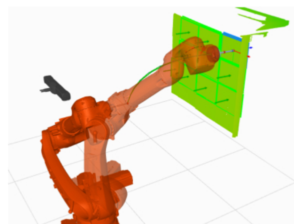


Figure 2.27: Picking Pose

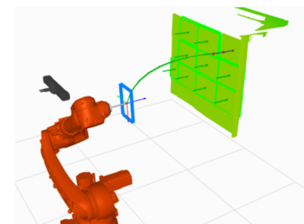


Figure 2.28: Exit Trajectory

Overall, the procedure for creating and running simulations is effective and easy to follow. However, not all scans are saved automatically, and the need for manual saving may slow down the workflow in some cases.

From a functional perspective, the system demonstrated good performance in many scenarios, particularly with small boxes. A positive aspect observed during testing is that the system generally ignores boxes that are only partially visible, reducing the risk of unreliable picks. Some limitations nevertheless emerged. The configuration of picking priorities is not very intuitive,

and it is currently not possible to define a tolerance or weighting for these priorities. Moreover, the simulated picking sequence becomes visible only after the trajectory has been generated; being able to preview it earlier, during the localization phase, would improve usability.

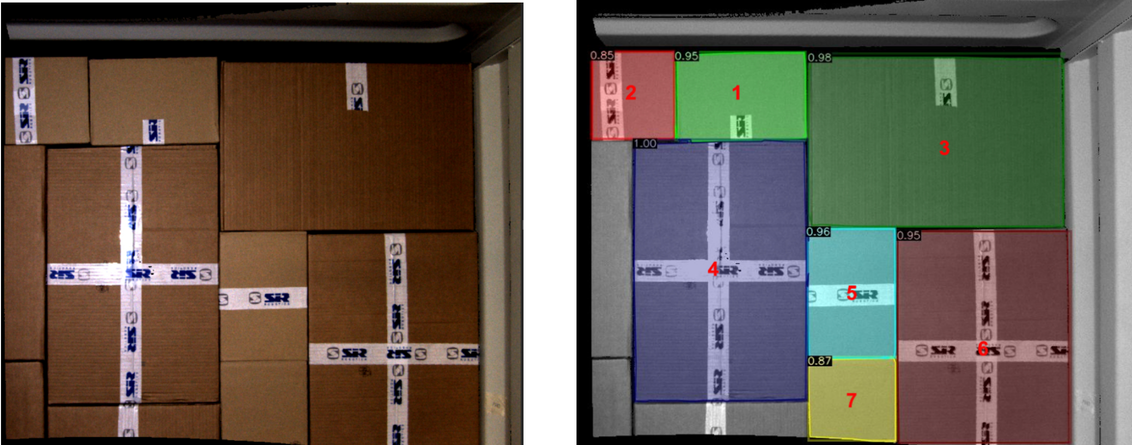


Figure 2.29: Detection Results

In terms of detection reliability, the system occasionally struggled to correctly localize all boxes, especially larger ones. This behavior suggests that a retraining of the neural network might be necessary to improve robustness. Additionally, under certain lighting conditions or when boxes had adhesive tape on their surfaces, segmentation errors were observed, sometimes causing two adjacent boxes to be merged into a single detected object.

Figure 2.30: Failed Detection

2.3.2 SICK

The evaluation also included tests with SICK's **PALLOC** vision system.
PALLOC is a robot guidance system used for automatic depalletizing, comprising a 3D color sensor with embedded PALLOC software based on deep learning. The PALLOC system is used

for three-dimensional localization and automatic depalletizing of objects of various sizes and appearances.

Figure 2.31: PALLOC Visionary-S

Overall, the PALLOC system proved to be well structured and supported by **PLx Manager**, a configuration software that is intuitive and easy to use, allowing parameters to be set quickly and efficiently. However, the system is highly sensitive to different environmental conditions and geometric configurations, which can significantly affect the stability and quality of the localization process.

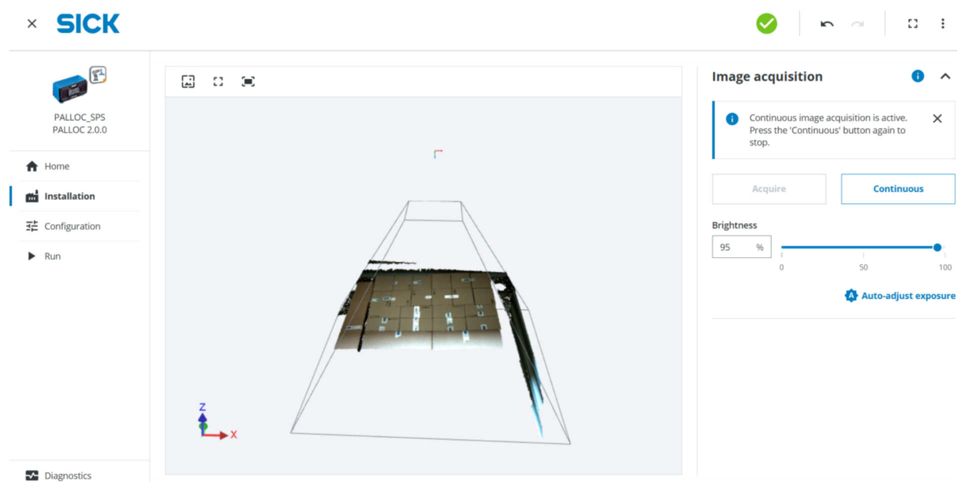


Figure 2.32: GUI Overview

The main issue identified concerns the calculation of the **top layer**, which is essential for detecting the boxes. The system automatically determines this plane based on the points detected inside the search volume, identifying which surface represents the layer closest to the camera. However, the presence of unwanted surfaces, such as side walls or the container ceiling, can interfere with this process, causing the system to incorrectly interpret these elements as the top layer.

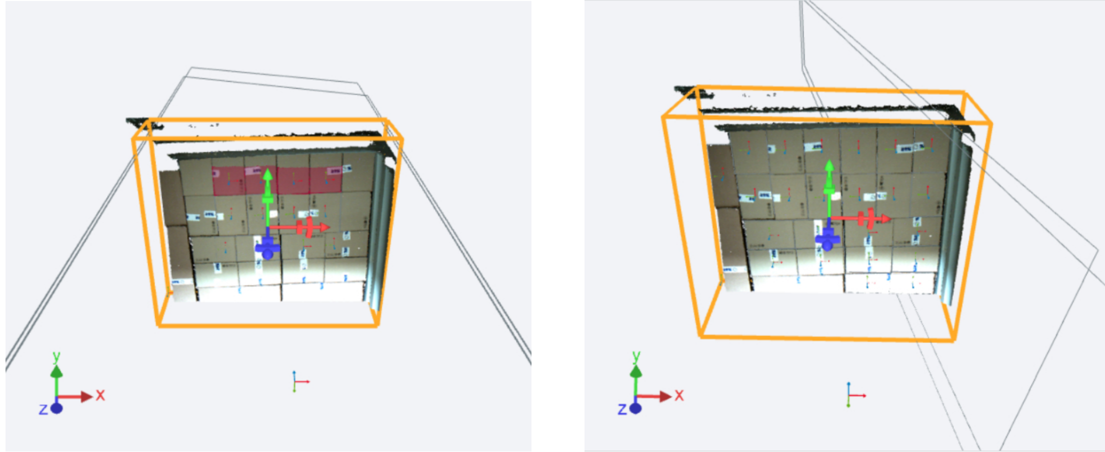


Figure 2.33: Mistakes in top layer computation

This leads to inconsistent localization results, incorrectly oriented planes, and failures in detecting the actual boxes. Even when the top layer is correctly estimated, the depth of this layer must be set with great care: non-optimal values easily cause the system to miss the boxes on the uppermost layer. As a result, the system sometimes behaves unpredictably and shows limited robustness against small geometric or environmental variations.

Tests performed on different box configurations further highlighted this sensitivity. In ordered, single-layer scenarios, the first attempts often produced correct results, whereas in other cases multiple acquisitions were required before achieving a reliable localization. In disordered layouts or when boxes were placed at slightly different heights, detection performance became inconsistent: some tests required several attempts to obtain a valid result, and others were unable to identify all boxes.

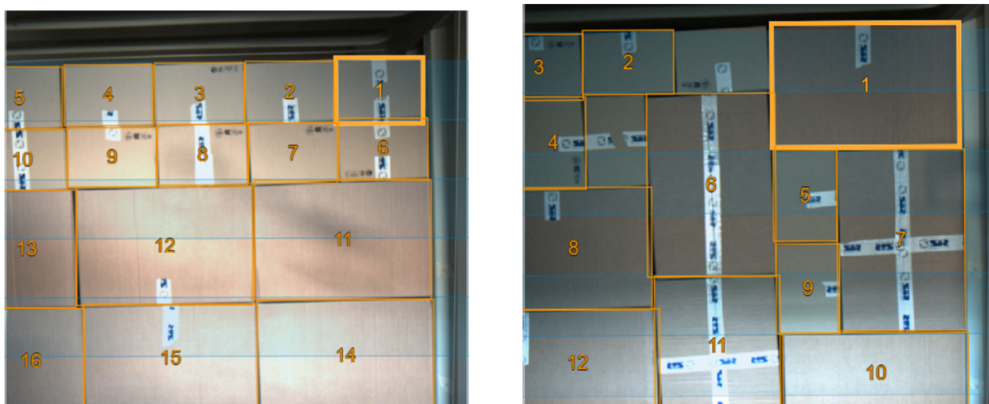


Figure 2.34: Localization on a single layer

The difficulties increased even more when testing multiple layers that were not perfectly aligned: in such conditions, the top layer was almost always calculated incorrectly, and the system detected only the layer closest to the camera while ignoring the lower ones, even when the

configured depth should have allowed their detection.

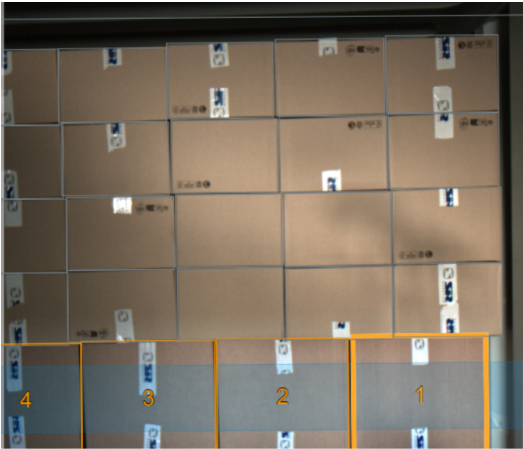


Figure 2.35: Localization on multiple layer

Another significant limitation is the strong dependence on lighting conditions. The camera uses invisible infrared light for the 3D component, but the 2D images, crucial for box recognition, are strongly affected by ambient illumination. As natural light decreases, the images become progressively darker until they are no longer usable, preventing the system from performing localization. Artificial lighting could mitigate the issue, but it introduces the risk of reflections on the surfaces, which may further degrade scan quality.

Figure 2.36: Localization with low lightning

3. Robot

3.1 Introducion to ABB

ABB is one of the leading global companies in industrial automation and robotics. The company develops a wide range of robotic solutions used in manufacturing, logistics, assembly, material handling, and many other industrial applications. ABB robots are known for their reliability, high precision, and advanced control systems, which allow them to operate in complex and dynamic environments.

ABB provides robots with different payload capacities and reaches, making them suitable for a variety of tasks, from small assembly operations to heavy material handling. These robots can be programmed and simulated using the RobotStudio software environment, which allows engineers to develop and test robotic applications both online and offline. In the context of this project, an ABB industrial robot is used to perform the pick-and-place operations required for the automated depalletizing process.

3.1.1 ABB IRB 4600

The robotic arm used for the tests is the **ABB IRB 4600-45/2.05**, a compact and high-performance industrial robot designed for applications requiring speed, precision, and reliability. It features a maximum payload of 45 kg and a reach of 2.05 m, making it suitable for handling medium-sized boxes while maintaining high dynamic performance. The robot's compact design allows it to operate efficiently in confined spaces, such as the interior of a shipping container.
Thanks to its high positioning accuracy and repeatability, the IRB 4600 is well suited for vision-guided picking tasks, where precise alignment between the robot, the detected box, and the gripper is essential. These characteristics make it an appropriate choice for testing and validating the box picking and placement procedures required by the automated depalletization system.

Figure 3.1: IRB 4600-45/2.05

3.1.2 Robot Studio

RobotStudio is a development tool used for the configuration and programming of ABB robots, both physical and virtual. This application supports the modeling, offline programming, and simulation of robotic cells. Its advanced modeling and simulation features help visualize multi-robot control systems, safety functions, 3D views, and remote robot supervision.
The integrated programming environment of RobotStudio allows both online and offline programming of robot controllers. In online mode, it is connected to the physical robot controller, while in offline mode it is connected to a virtual controller that emulates the real robot controller on a PC.

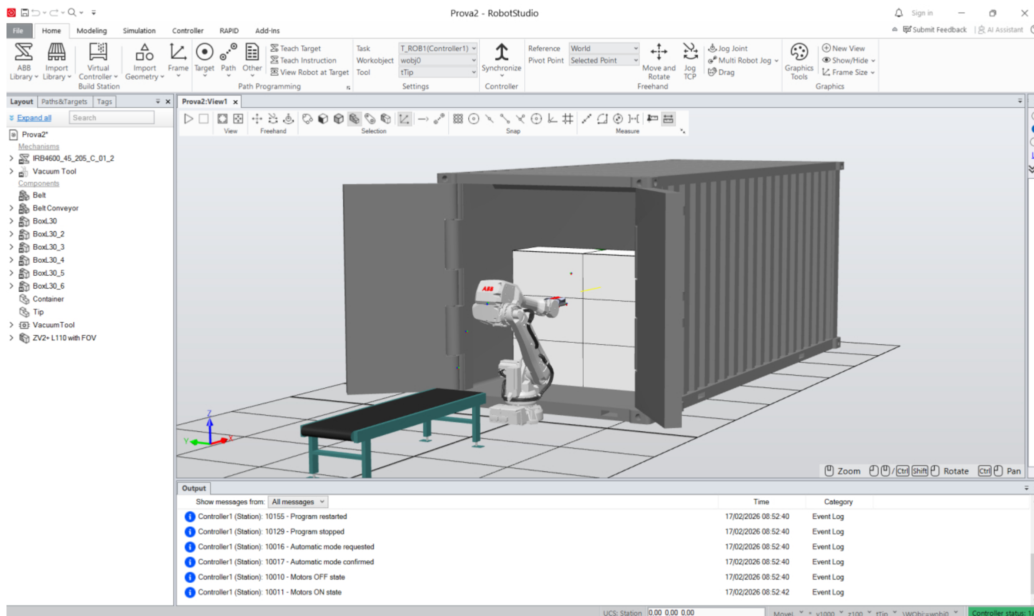


Figure 3.2: Robot Studio main page

3.1.2.1 RAPID Programming Language

The programming language used to control ABB robots is **RAPID**. RAPID is a high-level programming language specifically developed for industrial robotic applications. It provides all the necessary tools to define robot motions, control program flow, and manage communication with external devices.

The language supports common programming structures such as loops, conditional statements, procedures, and functions. It also allows the use of different types of variables, including numeric values, strings, Boolean variables, and structured data types used to represent robot positions and orientations.
In addition to general programming features, RAPID includes a large set of dedicated instructions for robot control, such as motion commands for joint and linear movements, configuration of tools and work objects, and interaction with sensors and external systems. These capabilities make RAPID suitable for developing complex robotic applications involving perception, motion planning, and interaction with other components of an automated system.

3.1.2.2 Home

The **Home** panel of RobotStudio provides quick access to the main tools required for setting up and programming a robotic application.
With this interface, it is possible to create and manage virtual controllers by selecting a profile, an existing controller, or a robot model from the library.
The user can also create multiple targets at the same time, either by manually entering their coordinates or by selecting positions directly in the graphical window. Target orientations can be

automatically aligned with nearby CAD surfaces, simplifying accurate positioning. In addition, the Home panel allows the creation of robot paths and work objects, enabling the definition of reference frames and motion sequences needed for programming and simulation.

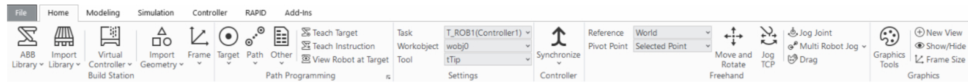


Figure 3.3: Home panel

3.1.2.3 Modeling

The **Modeling** panel in RobotStudio is used to build and customize the virtual workcell by adding and configuring 3D components. It allows the import of external CAD geometry to represent elements such as containers, conveyors, or fixtures within the station. The panel also supports the creation of mechanisms to simulate external devices, including rotary tables, grippers, and positioners.

In addition, tools mounted on the robot can be defined using the Tool Creation Wizard, which simplifies the process of creating a tool either from an existing CAD model or from a placeholder geometry.

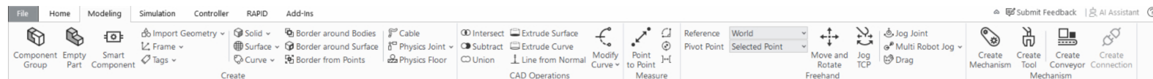


Figure 3.4: Modeling panel

3.1.2.4 Simulation

The **Simulation** panel in RobotStudio is used to execute and analyze robot programs in a virtual environment. It allows users to run simulations of robot motions, visualize the execution of paths and tasks, and verify the behavior of the robotic system before deploying it on the real robot. This panel is useful for checking timing, detecting possible collisions, and validating the overall sequence of operations in a safe and controlled setting.

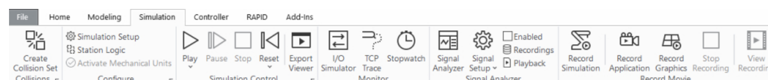


Figure 3.5: Simulation panel

3.1.2.5 Controller

The **Controller** panel in RobotStudio provides tools for connecting to and managing robot controllers. It allows users to add and connect to physical robot controllers or virtual controllers available on the network.

Through the configuration interface, system parameters can be viewed and modified, with

changes applied directly to the controller.
The panel also includes the FlexPendant Viewer, which displays the robot pendant interface within RobotStudio. Advanced motion configuration functions are available to manage controller positioning, base frame definition, calibration procedures, and the setup of external axes.

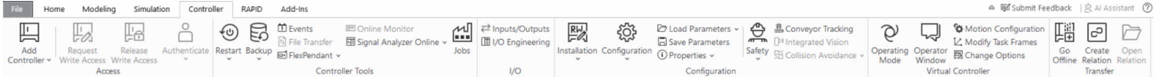


Figure 3.6: Controller panel

3.1.2.6 RAPID

The **RAPID** panel in RobotStudio provides tools for creating, editing, and managing RAPID programs. It allows users to work with programs that are running online on a physical robot controller, stored on a virtual controller, or handled as standalone offline files. Through the integrated RAPID Editor, it is possible to view and modify the robot's RAPID code, supporting program development and maintenance across different operating modes.

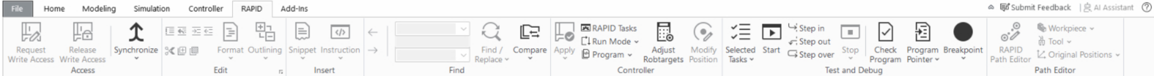


Figure 3.7: RAPID panel

3.2 Simulation Setup

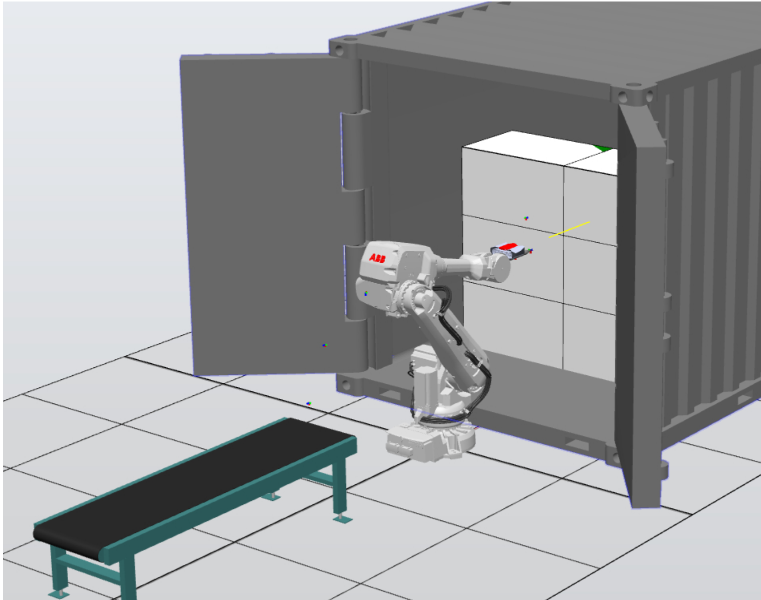


Figure 3.8: Simulation setup

The simulation setup consists of the robotic arm, the vision camera, the container, a layer of boxes inside the container, and a conveyor belt. The robot geometry is imported from the ABB library and positioned at the origin of the workspace.
The container is placed in front of the robot, at a short distance from its base, while the conveyor belt is positioned behind the robot and represents the area where the picked boxes are deposited. Inside the container, a single layer of boxes with dimensions of 900 × 600 × 500 mm, corresponding to the largest box size considered, is used for the tests.
Since no vacuum gripper was available during the testing phase, a calibration tip was used to simulate the gripping action. It is important to note that this setup represents only a testing configuration and not the final layout of the application. In the real system, the robotic arm would be mounted on a dedicated support and transported by the AMR, and the conveyor belt would be physically integrated with the robot structure.

3.3 Pick and Place Trajectory

After the setup of the simulation, the next step is to develop a trajectory for picking the boxes and place them on the conveyor.
The routine consist in a series of points that the robot has to reach precisely, in order to avoid collisions with the walls of the container and with other boxes.

The pick-and-place operation can be divided into two main phases: the approach and picking of the box inside the container, and the transfer of the box to the conveyor belt. For each box, the robot first moves to an approach position in front of the target, then reaches the picking point where the box is grasped. After the grasp, the robot moves upward to a safe exit point and finally follows a trajectory that leads to the conveyor, where the box is released. These intermediate points allow the robot to perform smooth and safe movements while avoiding collisions with the surrounding environment.

3.3.1 Home Position

The Home position is the intial position of the robot. The robot starts and ends the routine in this position. As we can see from the figure the robot has the axis 2 horizontally with respect to the floor, the axis 3 pointing a little bit upward and axis 5 placed such that the tip mounted on axis 6 is perpendicular to the layer of boxes.

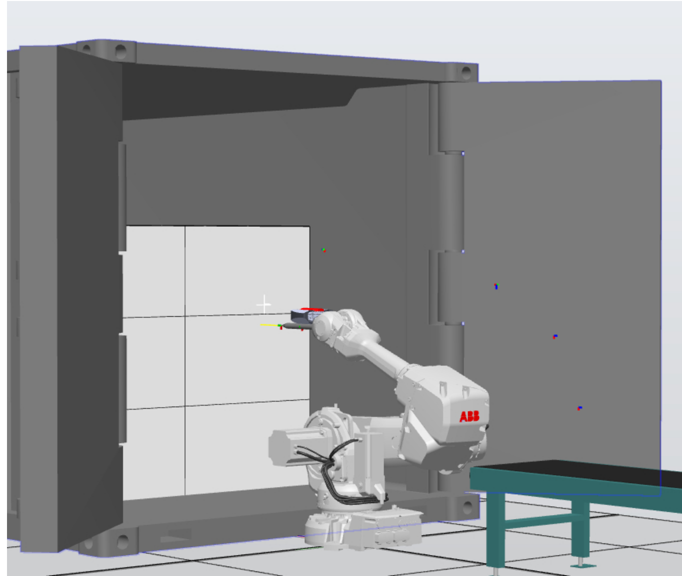


Figure 3.9: Home position

3.3.2 Scan Position

From the Home position we move to the Scan position. Four different Scan positons have been saved, allowing to scan the whole area of the container.

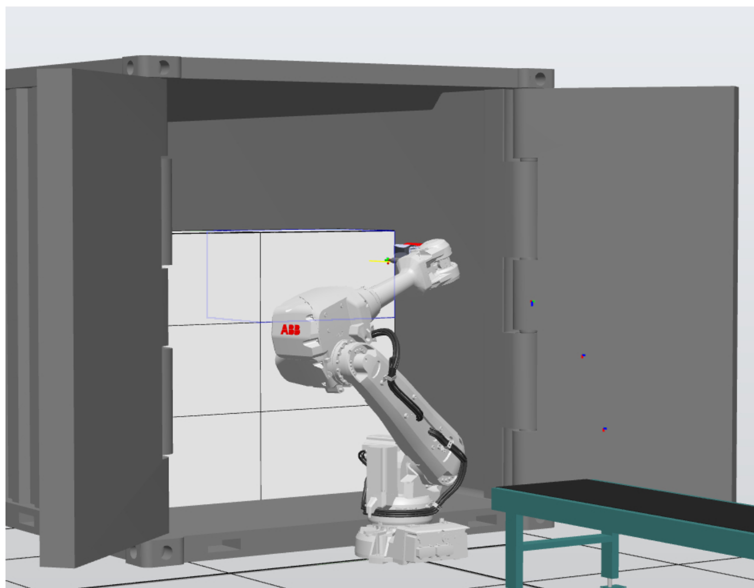


Figure 3.10: Scan position

3.3.3 Approach Position

The Approach position is a predefined point located near the center of the box layer inside the container, offset from the exact center. This position is chosen to allow the robot to move efficiently toward any target box detected by the vision system.

By starting from this central position, the robot can reach all areas of the container with relatively short movements, including the corners. This reduces the overall motion time and allows the robot to quickly approach the box that has been selected for picking.

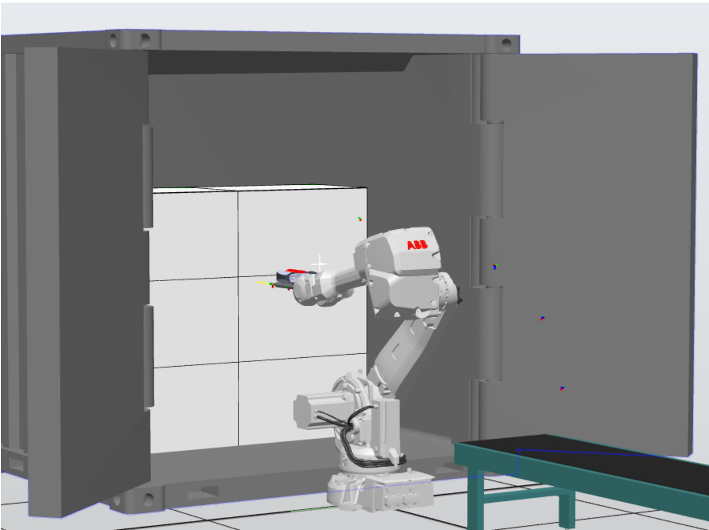


Figure 3.11: Approach position

3.3.4 Grip Position

Once the robot reaches the Approach position, the next step is to move toward the center of the box in order to perform the gripping operation. The Grip position corresponds to the point where the gripper makes contact with the top surface of the box. For a stable grasp, this point is defined at the geometric center of the box and with the tool oriented vertically with respect to the box surface.

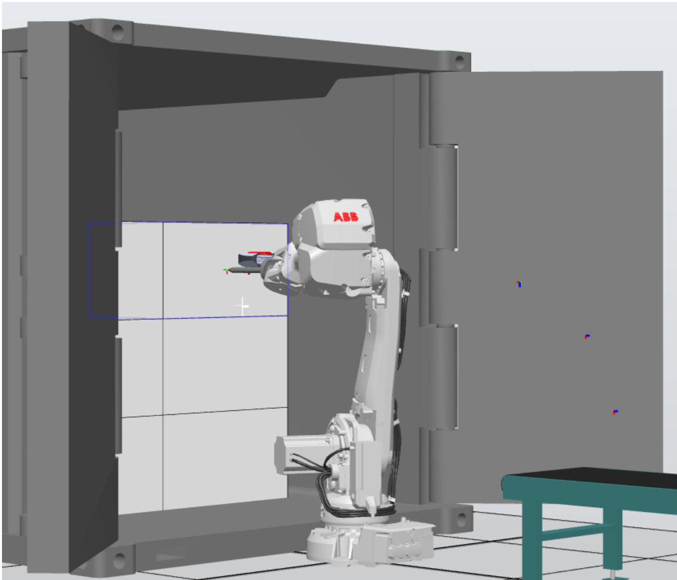


Figure 3.12: Grip position

3.3.5 Leave Position

After the box has been picked, the robot must move it out of the box layer without colliding with the surrounding boxes. To ensure safe clearance, the robot moves outward by at least 900 mm, which corresponds to the maximum length of one side of the boxes. This movement allows the robot to exit the picking area safely and reach the Leave position.

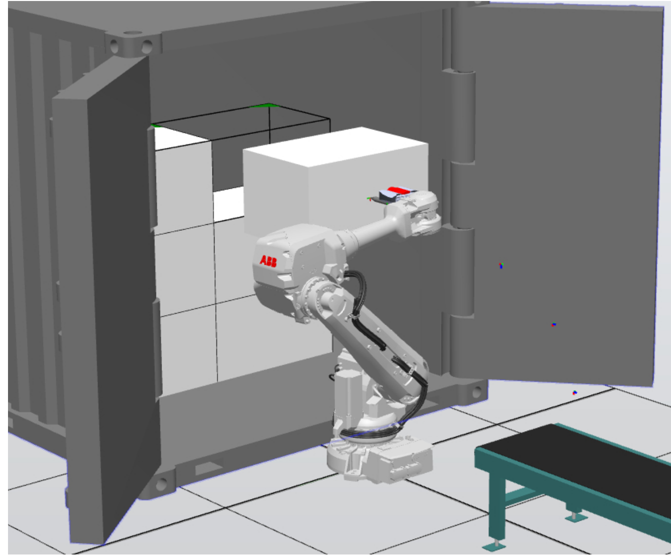


Figure 3.13: Leave position

3.3.6 Box Orientation

Once the robot reaches a safe position outside the box layer, the next step is to leave the container without colliding with its walls. To reduce the occupied space during this movement, the orientation of the box is adjusted.

This is achieved by rotating the wrist of the robot, specifically axis 5, so that the box is oriented vertically. By changing the orientation in this way, the robot can move more safely inside the narrow space of the container and avoid possible collisions with the side walls while exiting.

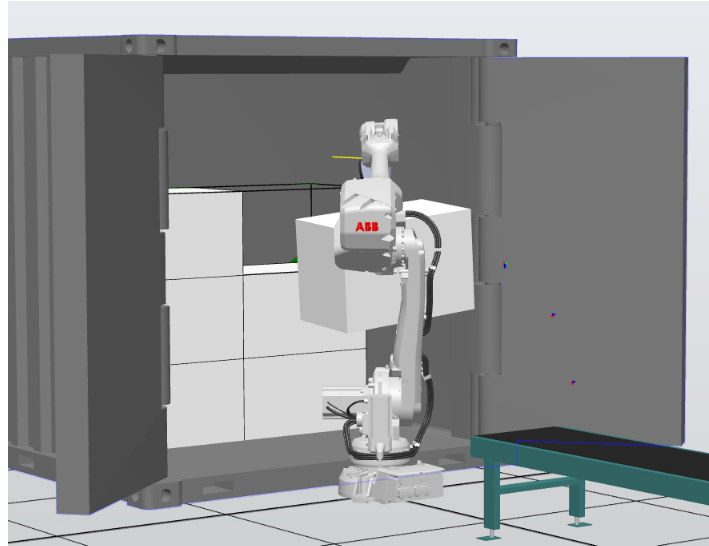


Figure 3.14: Box orientation

3.3.7 Positioning the Box in Front of the Conveyor

After leaving the container, the robot must move to the area where the box will be released. To achieve this, the robot rotates around its base by moving axis 1. This rotation allows the robot to reorient the arm and bring the box in front of the conveyor belt.

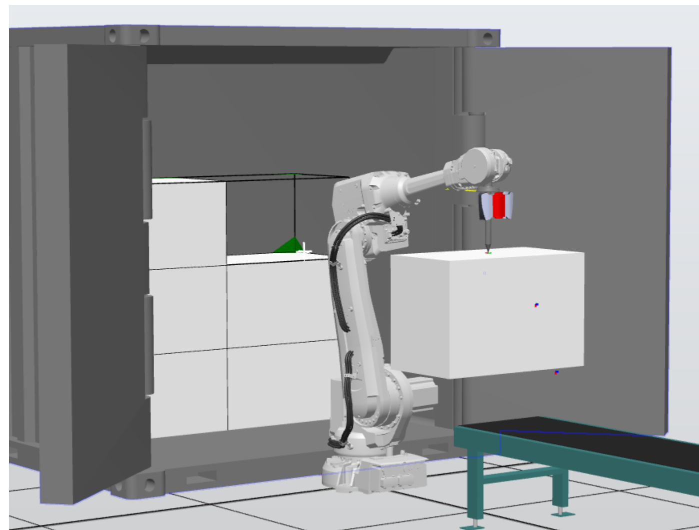


Figure 3.15: Box in Front of the Conveyor

3.3.8 Lowering and Reorientaion

Once the robot reaches the position in front of the conveyor, the box must be reoriented before being placed. The robot rotates the wrist so that the box returns to a horizontal orientation, which is the correct position for placing it on the conveyor belt.

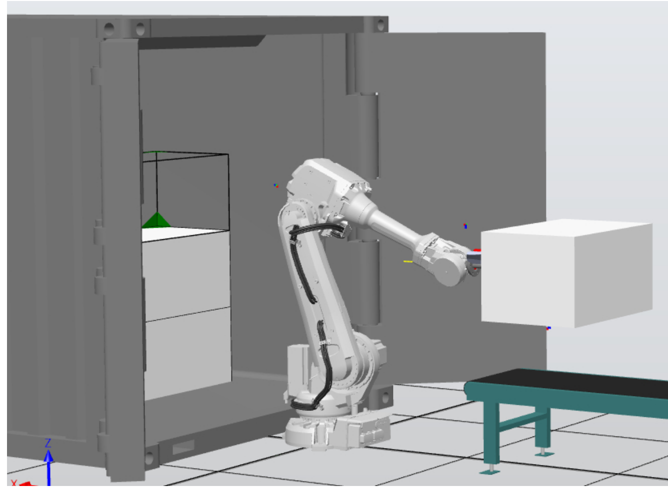


Figure 3.16: Lowering and Reorientation

3.3.9 Place Position

After the reorientation, the robot moves downward in a controlled motion to approach the conveyor surface. This movement reduces the distance between the box and the conveyor, ensuring that the box can be released safely and placed in a stable position.

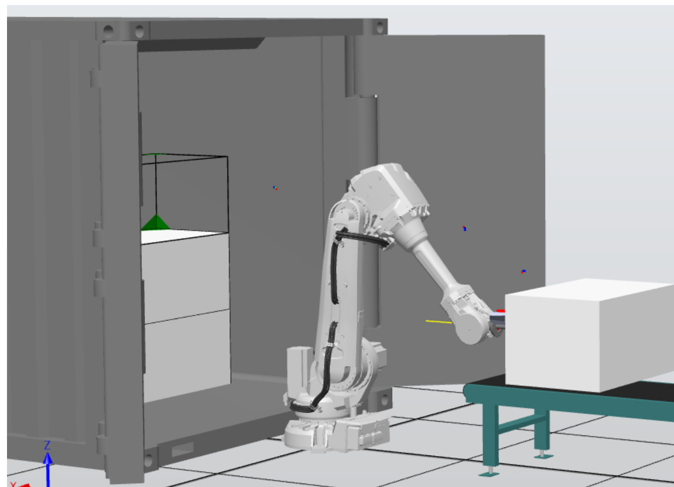


Figure 3.17: Place Position

3.4 Integration with Vision System

3.4.1 Communication between Robot and Vision System

The communication between the robot controller and the vision system is implemented through a **TCP/IP socket connection**. This TCP/IP communication mechanism represents the interface between the robot controller programmed in RobotStudio and the vision system. Through this interface, the robot can request scans, obtain the position of the detected boxes, and retrieve the

trajectory information required to perform the picking operation.
In this architecture, the robot acts as a client that sends command requests to the vision system, while the vision software running on the PC listens for incoming connections and returns the corresponding responses.
The robot establishes a socket connection to the IP address of the computer where the vision system is running, using the configured port. Once the connection is established, the robot can send command messages and receive responses from the vision system.

The communication follows a request–response protocol. Each command sent by the robot contains a command identifier and, when required, a list of input parameters. The command identifier specifies the operation to be executed by the vision system, while the additional parameters define the inputs required by that command.
After receiving the request, the vision system processes the command and returns a response message containing a result identifier and, when available, the output parameters produced by the command execution. The result identifier indicates the execution status of the command, while the output parameters contain the data generated by the vision system, such as the coordinates of the detected object or other information required by the robot.

3.4.1.1 Socket Communication Function

The communication between the robot controller and the vision system is implemented through the RAPID function `HandshakeSocketRaw`.

```
FUNCTION HandshakeSocketRaw(command, parameters)
    commandString = NumToStr(command)

    IF length(commandString) > 4 THEN
        RETURN invalid_command_error
    ENDIF
    WHILE length(commandString) < 4 DO
        commandString = "0" + commandString
    ENDWHILE

    fullMessage = commandString + ";" + parameters
    SocketSend(fullMessage)
    SocketGetStatus()
    response = SocketReceive()

    resultToken = UnpackRawBytes(response,4)
    resultID = StrToVal(resultToken)
    RETURN resultID
```

This function is responsible for sending a command to the vision system through the TCP/IP socket and receiving the corresponding response.

The function takes two input parameters: the command identifier and a string containing the command parameters. The command identifier is first converted into a string using the `NumToStr()` function. Since the communication protocol requires the command ID to have a fixed length of four characters, the code checks the length of the generated string and adds leading zeros if necessary. After formatting the command identifier, the function concatenates the command ID and the parameter string using a semicolon separator.

Once the message is created, it is transmitted to the vision system using the RAPID instruction `SocketSend`. The socket status is then checked with `SocketGetStatus` to ensure that the communication channel is active. After sending the request, the function waits for the response from the vision system using the `SocketReceive` instruction. The received raw data is stored in a buffer variable.

The function `UnpackRawBytes` is then used to extract the first four ASCII characters from the received message. These characters correspond to the result code returned by the vision system. Finally, the extracted string is converted back into a numerical value using the `StrToVal()` function. This value represents the *Result ID*, which indicates the outcome of the executed command. The result code is returned by the function and used by the main robot program to decide how to proceed with the execution of the pick-and-place routine.

3.4.2 Main Vision System Commands

The interaction between the robot controller and the vision system is based on a set of predefined commands provided by the camera software. These commands are executed by the vision system but can be invoked by the robot through a TCP/IP socket connection. From the robot side, the RAPID program sends formatted request messages through the socket interface, which are interpreted by the vision system as specific commands.
Through these commands, the robot can control the scanning process, request the detection of objects inside the bin, retrieve the pose of the selected box, and obtain the trajectory information required to perform the picking operation.

The **Reset** command is used to initialize the vision system before starting a new picking cycle. This command resets all internal processes and sensors, reloads the workcell configuration, and performs a basic check of the main system components. At this stage the position of the robot is not relevant, since the command only prepares the system for the next operations.

The **Start** command begins the processing pipeline for a specific bin. When this command is executed, the vision system loads the configuration associated with the selected bin and prepares the processing pipeline in the backend. The command also allows the user to specify the bin identifier, the frontend configuration job, and the estimated size of the boxes to be detected.

The **Stop** command terminates the current process associated with a given bin. This command performs a soft stop of the system, allowing the current operation to finish before resetting the internal state machine.

The **Scan** command triggers the acquisition of images from the camera. During this operation the vision system performs a scan of the selected bin using the parameters defined in the camera configuration. The command may also include a flag indicating whether additional images are required to cover a larger portion of the bin. Since the operation is synchronous, the camera must remain fixed until the acquisition process is completed.

The **Next** command requests the next available picking result from the vision system. The command returns the best reachable object detected in the scanned scene, according to the configured ordering criteria. The response typically includes the estimated position and orientation of the gripping point, the dimensions of the detected box, and a status code indicating whether a valid object has been found or if additional computation is required.

The **GetWaypoints** command is used to retrieve the trajectory required for the robot to reach the selected object. The vision system computes a collision-free path and returns the next waypoint of the trajectory. This command must be called repeatedly until all trajectory points have been received. The returned waypoint can be expressed either as joint coordinates or as a Cartesian pose of the robot tool.

3.4.3 Acquisition Cycle for Robot-Camera Integration

The core of the integration between the robot and the vision system is implemented in the `Acquisition` procedure. This routine coordinates the complete interaction between RobotStudio and the Smart Pick 3D Flex system, from scene acquisition to box picking and final placement on the conveyor. Its purpose is to manage the full pick-and-place cycle by combining robot motion commands with vision system API calls.

The first operational step of the cycle is to move the robot to a predefined Home position. This ensures that every cycle starts from a known and safe configuration. After that, the communication with the vision system is reset with the `Reset` command and the process is started. The `Start` command initializes the vision workflow for the selected bin and loads the necessary

configuration. If the returned result is not valid, the procedure is interrupted immediately.

Once the process has started, the robot enters the acquisition loop. In this phase, the robot moves to the Scan position and triggers the `Scan` command. This command asks the vision system to acquire the scene and process the images. The flag `bMoreImages` is used to indicate whether additional scans are required. If multiple acquisitions are needed to cover the whole area, the loop continues until the scan sequence is completed.

After the scan phase, the robot moves to the Approach position in front of the boxes. This is the position from which the system requests the next available object to be picked. The `Next` command is then executed. This command returns the gripping point of the selected box, its dimensions, and other information.

If the command execution is successful, the procedure proceeds with the pick-and-place operation. At this stage, two different operating modes are possible, depending on the value of the flag `bGetTrajectory`.

If `bGetTrajectory` is enabled, the robot does not move directly to the gripping point. Instead, it asks the vision system to compute a full trajectory through the `GetWaypoints` command. This command returns one waypoint at a time. This process is repeated until all waypoints have been received. In this mode, the trajectory is generated by the vision system and can therefore take into account collision avoidance and optimized approach paths.

During the waypoint loop, the vision system computes three main points that define the picking trajectory. The first point is another approach position, located perpendicularly above the center of the box with a horizontal offset, allowing the robot to approach the object safely. The second point corresponds to the actual gripping position, where the robot reaches the center of the box and performs the grasp. The third point is a safe exit position, used to move the box out of the layer without colliding with the surrounding boxes. Once all three points have been reached, the robot leaves the container and follows the predefined trajectory toward the conveyor. Finally, the box is placed on the conveyor belt and released before the robot returns to its initial position.

If `bGetTrajectory` is disabled, the procedure follows a simpler strategy. In this case, the robot moves directly to the gripping pose returned by the `Next` command. After grasping the box, it moves linearly away from the box layer by 900 mm along the X direction, in order to safely exit the picking area. It then executes the same predefined sequence of movements toward the conveyor, places the box, and returns to the initial Home position.

The routine also handles special result codes returned by the vision system. If the layer of boxes

is empty the procedure terminates because no more boxes are available for picking. Any other unexpected result also causes the cycle to stop. In this way, the procedure remains robust and avoids unsafe behavior in the presence of communication errors or unsuccessful detections.

Overall, the `Acquisition` procedure combines perception, motion planning, and manipulation into a single automated workflow: the robot acquires the scene, requests the next pickable box, optionally receives a complete trajectory, performs the grasp, moves the box to the conveyor, and repeats the cycle until the container is empty.

3.4.4 Final Considerations

The results obtained in this chapter show that the selected robot is well suited for the proposed depalletizing task. The ABB IRB 4600 provides high speed and good maneuverability, which allows it to operate efficiently inside the confined space of a shipping container. Its compact design and dynamic performance make it particularly suitable for applications where both precision and fast cycle times are required.
Another positive aspect is the use of the RAPID programming language. The language offers clear and intuitive programming structures that make it relatively easy to implement robot routines, motion sequences, and communication procedures.

Finally, the integration between the robot controller and the vision system proved to be simple and efficient. The communication based on TCP/IP sockets allows the robot to quickly exchange commands and data with the camera system, enabling a smooth interaction between perception and manipulation during the depalletizing process.

Conclusions and Future Developments

The main objective of this project was to investigate the feasibility of an automated solution for unloading boxes from shipping containers using robotic technologies. The study aimed to evaluate how the integration of an Autonomous Mobile Robot (AMR), an industrial robotic manipulator, and a vision system could enable a fully or partially autonomous depalletizing process. In particular, the work focused on assessing the capability of the AMR to access and navigate inside the container, the effectiveness of vision systems in detecting and localizing boxes under realistic conditions, and the ability of the robotic arm to execute reliable pick-and-place operations for transferring the boxes onto a conveyor system. The overall goal was to explore a robotic approach that could improve efficiency, reduce manual labor, and increase safety during container unloading operations.

The experimental tests performed on the AMR platform confirmed the feasibility of using a mobile robotic system to support the depalletization process inside a shipping container. In particular, the alignment procedure with the loading ramp proved to be reliable and repeatable, allowing the robot to consistently reach the correct entry configuration before accessing the container. The customized obstacle-detection configuration also demonstrated good effectiveness, enabling the robot to detect boxes placed along its path and to safely interrupt and resume operations when required. Finally, the navigation strategy based on a node graph and Dijkstra path planning allowed the robot to reach the rear area of the container and manage the unloading workflow. Although the current implementation relies on absolute node navigation, the tests highlighted that the availability of relative-motion commands could further simplify short movements during the progressive unloading of the container.

The evaluation of the different vision systems demonstrated the importance of reliable perception for the depalletization task. Among the tested solutions, the IT+ Robotics Smart Pick 3D Flex system showed the best balance between detection accuracy, robustness, and ease of configuration. The software architecture, based on the separation between FrontEnd and BackEnd components, proved to be clear and practical for configuring the acquisition process and processing the data independently. The tests carried out in different scenarios confirmed that the system is capable of correctly detecting and ordering boxes even in complex configurations, including disordered and multi-layer arrangements. Although some limitations were observed in cases involving very

small or partially visible boxes, the overall performance demonstrated that the vision system is suitable for supporting the automated depalletization process.

The results obtained during the experiments confirm that the selected robotic platform is suitable for the depalletization task. The ABB IRB 4600 robot combines high speed, precision, and good maneuverability, allowing it to operate effectively even within the limited space of a shipping container. The RAPID programming language also proved to be clear and intuitive, making the implementation of motion routines, control logic, and communication procedures relatively straightforward. In addition, the integration between the robot controller and the vision system through TCP/IP socket communication enabled a fast and reliable exchange of commands and data, allowing the robot to receive the box positions detected by the vision system and execute the corresponding pick-and-place operations in an efficient and coordinated way.

Nevertheless, several aspects require further development before the system can be deployed in a real industrial environment. In particular, the robustness of the vision system must be further improved to handle a wider variety of box configurations, lighting conditions, and potential occlusions. Additionally, the navigation strategies of the AMR could be refined to optimize the robot's positioning inside the container and reduce cycle times during the unloading process.

Another important aspect concerns the operating conditions inside the container. During summer, the temperature inside a closed container can become very high. For this reason, it will be necessary to evaluate the thermal conditions not only for human operators but also for the electronic devices used in the system. Cameras, sensors, and robot components could suffer from excessive heat, therefore future developments may include the design of dedicated cooling or ventilation solutions to protect the equipment and ensure stable operation.

Future developments of this project will also focus on improving the mechanical setup used during the tests. During the experiments with the robot, a real gripper was not available and a calibration tip was used to simulate the grasping operation. In a real application, a dedicated gripper should be designed. A possible solution could combine a vacuum gripper with a mechanical support similar to the forks of a pallet truck, which could slide under the box to provide a more stable and reliable grasp.

Finally, further work will involve the design and testing of a real loading ramp for the AMR. During the experimental phase the ramp was simulated, but the use of a real ramp will be necessary to validate the alignment procedure, verify the accessibility of the container, and evaluate the overall performance of the system in realistic operating conditions.

In conclusion, this work provides a solid foundation for the development of an automated

container depalletizing system. The conducted experiments and evaluations highlight both the potential and the current limitations of the proposed approach, offering valuable insights for the future implementation of fully autonomous container unloading solutions in industrial logistics environments.

Bibliography

[1] KUKA AMR *KMP 1500 Autonomous Mobile Platform – Manuals and Datasheets*. Available at: https://my.kuka.com/s/product/kmp-1500peud-diffdrive-baseversion/01t1i000003gtDPAAY?language=it&tab=Downloads

[2] Coding. *Implementation of Dijkstra's Algorithm in Python*. Available at: https://www.youtube.com/watch?v=QymXuCgw6Ho

[3] IT+Robotics. *Smart Pick Flex – Product Page and Documentation*. Available at: https://eyetplus.com/en/prodotti/pick-flex-eng/

[4] Photoneo. *MotionCam-3D M – Manuals and Technical Datasheets*. Available at: https://www.photoneo.com/products/motioncam-3d-m/

[5] SICK AG. *PALLOC Robotic Guidance System – Manuals and Datasheets*. Available at: https://www.sick.com/ch/it/catalog/prodotti/sistemi/sistemi-di-gestione-robot/palloc/c/g584735

[6] ABB Robotics. *IRB 4600 Industrial Robot – Product Documentation*. Available at: https://www.abb.com/global/en/areas/robotics/products/robots/articulated-robots/medium-robots/irb-4600