POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Mobile Manipulator as Smart Human Assistant for Safe Objects Handling

Supervisor

Prof. Marina INDRI

Co-Supervisor

Ph.D. Pangcheng David CEN CHENG

Candidate

Rosario Francesco CAVELLI

December, 2023

to Alessia, Anna, Carlo e Peppa,
milestones of all my life.

Summary

Nowadays robots, and in particular mobile manipulators, can assist humans to accomplish common tasks, behaving as smart assistants for the operators sharing the same workspace. Keeping this in mind, the objective of this thesis is to expand the capabilities of a mobile manipulator (Locobot WX250s) in order to make it able to: (i) localize itself in a partially known environment, (ii) avoid obstacles in it, (iii) recognize and estimate the pose of known objects, (iv) localize the most dangerous part of the requested item and (v) pick and place it in order to minimize the risk for humans. The indoor environment where the robot operates can be, for example, a warehouse or a laboratory, where objects are stored on shelves, cabinets and drawers whose fixed positions can be considered as prior knowledge by the robot. The entire software infrastructure has been implemented on Ubuntu 20.04 using ROS Noetic as base to let the different modules communicate with each other. ROS is the most used middleware when it is necessary to develop any software for robots of any kind. It is full of packages implementing the most recent algorithms and it is also readily expandable to enrich the capabilities of the system. The SLAM task is achieved using the RTAB-Map package that, starting from the information coming from both the RGB-D camera and the Lidar mounted on the robot, is able to construct a map together with two costmaps (Global and Local), used to move the mobile base avoiding undesired contacts. Obstacle avoidance is further expanded by building a 3D occupancy grid exploiting the Octomap capabilities to reinterpret the point cloud coming from the camera. For what concerns the motion control, the mobile base is controlled using the Move Base package along with the Dijkstra's algorithm as the global planner and TEB as the local planner, while the arm's controller has been designed using the MoveIt! framework and planners belonging to Open Motion Planning Library (OMPL). In addition, a custom trajectory planner that considers the arm and the base together as a holistic system has been realized. This allow to better position and correct the robot's gripper placement when picking an object. The robot is also able to recognize and estimate the pose of known items thanks to the use of a YOLO v5 neural network. It has been trained to classify images belonging to a specific types list and to compute the bounding boxes to understand their space occupation. By fusing this information together with the depth data of the camera, it is possible to estimate the pose of the requested item. The network has been trained using a fully custom dataset built from scratches specifically for this purpose; it contains images of common objects that can be found in any laboratory. Pick-an-place operations are executed following a human-in-the-loop approach similar to what is done in Interactive Machine Learning algorithms. The robot can recognize dangerous objects basing on their class and ask the human a feedback about the most dangerous part. Once this latter section has been found, it grasps the object from there. To test the designed infrastructure, some experiments have been conducted in both a simulated environment, thanks to Gazebo and Rviz, and on the real Locobot WX250s inside the laboratory.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

artificial intelligence

AMR

Autonomous Mobile Robots

API

Application Programming Interface

BB

Bounding Box

CHOMP

Covariant Hamiltonian optimization for motion planning

CML

Classical Machine Learning

CNN

Convolutional Neural Network

CSP

Cross Stage Partial

DOF

Degree of Freedom

EE

End Effector

EST

Expansive Search Tree

GA

Genetic Algorithm

GG-CNN

Generative Grasping CNN

GPU

Graphic Processing Unit

HRI

Human Robot Interface

IK

Inverse Kinematics

IML

Interactive Machine Learning

IoU

Intersection Over Union

IRRT*

Informed RRT*

mAP

Mean Average Precision

ML

Machine Learning

MPC

Model Predictive Control

NN

Neural Network

NUC
: Next Unit of Computing

OMPL
: Open Motion Planning Library

PRM
: Probabilistic Roadmaps

PSO
: Particle Swarm Optimization

QP
: Quadratic Problem

R-CNN
: Region-based Convolutional Neural Network

RL
: Reinforcement Learnig

ROS
: Robot Operating System

RRT
: Rapidly-exploring Random Tree

RTAB-Map
: Real-Time Appearance-Based Mapping

SGD
: Stochastic Gradient Descent

SLAM
: Simultaneous Localization And Mapping

SRDF
: Semantic Robot Description

STOMP	Stochastic Trajectory Optimization for Motion Planning
TEB	Timed Elastic Band
URDF	Universal Robot Description File
VSLAM	Visual-SLAM
YOLO	You Only Look Once

Chapter 1

Introduction and Problem Statement

During the last two decades, *Robotics* has attracted the attention and aroused the interest of lots of people, not only researchers or experts, but also non-expert people and curious. The spread and increasing fame of this young discipline is entirely due to its naive purpose: robots have been designed and created with the main aim of replacing humans in heavy work. Nowadays, robots can be found in lots of places with different purposes, starting from facilities, where they are employed in the production chains or to move products around, and ending with houses and public places, where they are employed as *service robots*.

The increasing success and employment of different kinds of robots, especially in the *production field*, can be found in their main advantages:

- **versatility**, meaning that they can be easily reprogrammed to be used in a different part of the plant or to complete a different work;
- **accuracy**: they are able to complete a specified job with a precision way higher than the one of a human worker;
- **speed**, since a process can be completed by a robot in much less time than usual;
- robots can be used for **repetitive jobs**, increasing the quality of the work done by humans.

Another important advantage of using robots is that they can be **teleoperated**: they can be controlled remotely, allowing them to achieve an objective without requiring a person to be in the same place. This aspect is particularly useful if we think of all those dangerous situations where robots replace humans, increasing the **safety** of the operator. From most recent years, it is possible to find robots employed also in the *service field*: they can be found while welcoming guests at conferences, or as waiters in restaurants/hotels. Robots today are increasingly being seen as **humans' assistants**, helping them in different ways and for achieving different objectives together. This is the reason why modern robots are way smaller and built with materials and shapes suitable for the *human-robot* cooperation, constituting the well-known **Collaborative Robots**.

As it is possible to understand from this brief description, robots nowadays are used in a wide variety of fields and with lots of purpose. In order to respond to this increase in request and usage, *Robotic research field* has grown and changed, leading researchers to develop different solutions based on the specific usage of a robot.

1.1 Type of robots

Since robots can be found in different scenarios, in order to better understand what is their purpose and their working environment it is possible to group them into three macro-groups:

1. **industrial robots**
2. **mobile robots**
3. **service robots**

1.1.1 Industrial Robots

Robots used in the industrial fields are mainly **Robotic Manipulators**. The most usual aspect of a manipulator is the one of a **Robotic Arm**: essentially it is a mechanical structure made of a sequence of *rigid bodies* called **links**, connected together by means of **joints**; joints usually are the actuated parts of a manipulator, so they are in charge of the relative motion between links and, indirectly, of the entire robotic arm [1]. An example of a robotic manipulator can be seen in Figure 1.1. Two main types of joints can be used to build the desired *kinematic configuration* of the robot. *Prismatic joints* allow the *traslation* of one rigid link with respect to a predefined direction of another, while *Revolute joints* allow the relative rotation of the 2 connected links. Based on which joints are used and with which orders, it is possible to build different *kinematic chains*. Each of them is characterized by a number of *Degree of Freedom* (DOF), that can be seen as the number of independent parameters or variables that determine the state or configuration of the robot. Directly attached to the end of the manipulator by means of a *Mechanical Wrist* it is possible to find the *End Effector* (EE): it is a tool used by the robot for the specific purpose it is employed for. Several types of end effectors exist, and, usually, it is possible to change them in order to reuse the robot.

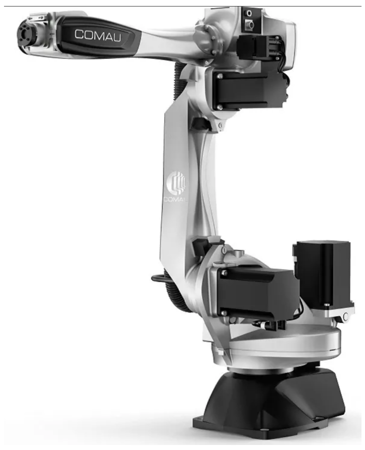


Figure 1.1: Example of Robotic Manipulator [2].

These types of robots are characterized by *high accuracy and speed* and a *high repeatability rate*, combined with the ability to deal with *a wide variety of payloads*. Thanks to these benefits, robotic manipulators are used to complete lots of operations in the production pipeline, like *handling*, *welding*, *assembly*, *painting*, *processing* [3].

Together with the robotic arms and manipulators, another type of robots belonging to the category of Industrial Robots are the *Autonomous Mobile Robots* (AMR) (Figure 1.2).

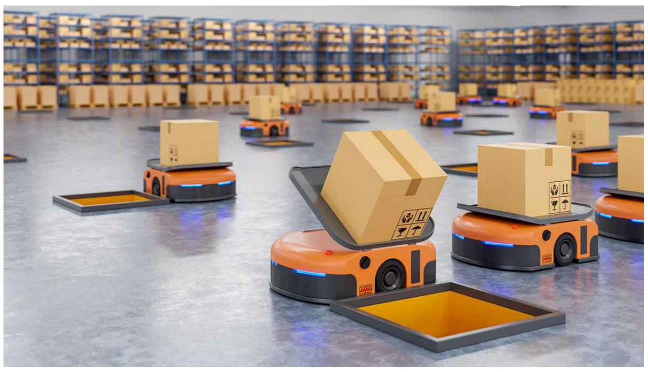


Figure 1.2: Representation of AMRs and their applications.

They are used in lots of facilities to *load*, *unload* and *move* goods and products from one location to another easily [4]. They are characterized by a large number of sensors and a high degree of independence thanks to their ability of *autonomous localization* and *navigation.* Moreover they are interconnected with each others, allowing rerouting and path planning according to the position and path being followed by others AMRs, or even collaborating to transport large objects [5].

1.1.2 Mobile Robots

Mobile robots are characterized by a mobile base that allow them to move inside the environment where they are placed (terrestrial, aerial, underwater). They are mainly exploited for all those jobs that require the ability of autonomous movement. This skill is strongly connected to the capability of the robot to sense the external environment, as well as all the internal component that constitute it. Indeed these robots are characterized by a strong *Sensing Subsystem*: different sensors are necessary to let the robot *localize itself*, *build a map*, *perceive the obstacles* and *avoid them.*

It is possible to distinguish two types of mobile robots based on the locomotion system: *wheeled robots* (Figure 1.3a), where the mobile base can move thanks to some actuated wheels, and *legged robots* (Figure. 1.3b), able to move thanks to the alternate movement of 2 or more kinematics chains.

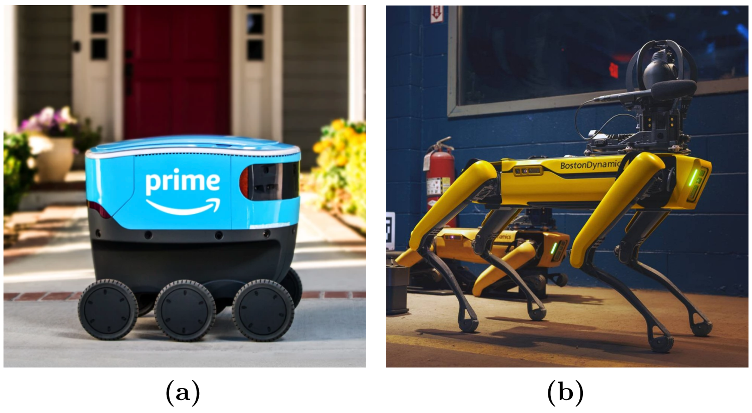


Figure 1.3: Examples of both types of mobile robots. (a)Wheeled Robot [6]. (b) Legged Robot [7]

1.1.3 Service Robots

Service Robots can be considered as part of the *Advanced Robotics* area. Giving a precise definition of a *Service Robot* may result difficult, since there are several application fields of these systems and the components characterizing them vary

[8]. Following the most general definition, Service Robots are all those robots characterized by a high level of automation that perform useful tasks for the humans and collaborate with them.

A deeper classification has been proposed by the ISO 8373:2012. As it can be seen from Figure 1.4, service robots can be further divided into 2 sub-classes: those designed for *personal use* and others designed for *professional use.* To the first class belong all the robots that are used to address everyday tasks, mainly in the domestic and assistance field. Professional robots, instead, are all those robots designed for a specific professional field and objective, and usually works as tool used by professionals.

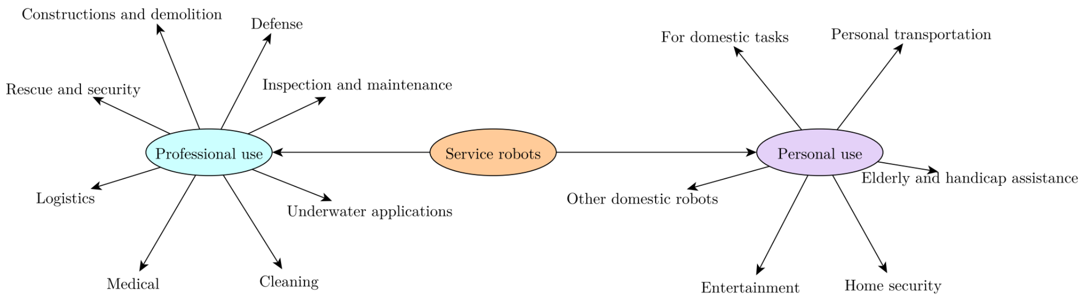


Figure 1.4: Classification proposed by ISO 8373:2012 for Service robots [8].

To the category of service robots it is possible to assign also the ***Mobile Manipulators***. They are characterized by a mobile base (driven by wheels, tracks or legs) on top of which are mounted one or more robotics arms. The contemporary presence of both a moving base and a manipulator gives to this type of robot the characteristic mobility of mobile robots and the dexterity and agility of manipulators. They are characterized also by high versatility and efficiency, making them the perfect tool to complete pick-and-place and material handling tasks in a large workspace, including a high level of security for other robots or humans. Thanks to the fusion of all the advantages of the two main categories of robots, mobile manipulators can be used as service robots for personal and professional uses: they can be used in *logistic and warehousing* fields, since they can pick, transport, and stack goods on shelves, in *offices and houses*, for *education reason* or even *space exploration.* Mobile Manipulators also come with challenges, such as control complexity, power management, and navigation in dynamic environments. However, ongoing advancements in robotics and *Artificial Intelligence* (AI) continue to improve their capabilities, making them more and more user-friendly.

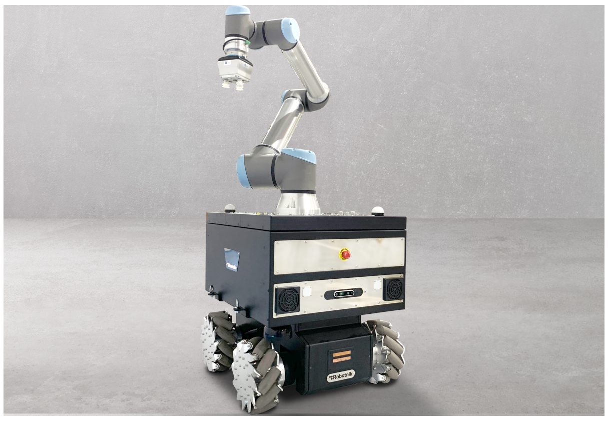


Figure 1.5: Example of a wheeled mobile manipulator [9].

1.2 Robots and AI

Robotics and AI are two closely connected research fields. The major scope in which AI is involved in the Robotics fields includes vision, grasping, motion control, fusing data coming from different sensors, making decisions and interacting with people. *Machine Learning* (ML) techniques are used together with cameras to help robots understand what they see, to recognize and treat unknown items. AI is also used to improve the grasping performances of manipulators; lots of algorithms can be found in literature that help the robots to understand the pose of a given object in the 3D space, while others are able to control EEs with several DOFs, and some of them have been trained exploiting *Reinforcement Learning* in order to emulate the human approach to grasping [10].

A relatively new and interesting approach to ML in the Robotics research fields is the *Interactive Machine Learning* (IML) [11]. IML represents a new way to train ML algorithms by inserting humans in the training loop. How these algorithms work can be easily understood by analyzing the main differences with respect to classical machine learning approaches. In *Classical Machine Learning* (CML) a human expert is responsible only of designing a suitable and efficient *Neural Network* (NN), selecting a *Data Set* representative of the data that will be present in the real application, choosing a suitable set of *hyperparameters* and *learning functions*. The training loop is entirely responsible for extracting knowledge from the data and examples provided: starting from a *labeled* or *unlabeled* samples (*Supervised ML / Unsupervised ML*), by processing input data and applying a training algorithm (usually based on the *Gradient Descent*), the NN learns how to assign the correct meaning to the received input. It is possible to notice how in CML humans are not part of the learning phase, in opposition to what happens during the learning phase of IML algorithms. Indeed in the latter type of ML humans are part of the learning phase. They provide feedback that is useful to select the best data and results and facilitate the learning phase by reducing the amount of time needed to train the network and to reach the desired performances. A simple diagram explaining how the training loop changes in IML can be seen in Figure 1.6.

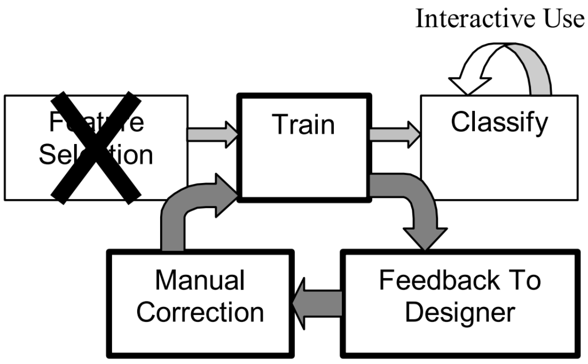


Figure 1.6: Interactive Machine Learning (IML) model [11].

1.3 Thesis Goal

The goal of this thesis is to expand the functionalities of an already-built Mobile Manipulators so that it can serve as a smart assistant for humans and accomplish the tasks of locating, recognizing, estimating the pose of an object, grasping it and delivering it into a depot near the human operator that requested it. Together with all these functionalities, the robot must be able to work in environments shared with humans. This aspect adds another challenging layer, since the robot must be able to identify the most dangerous parts of the requested objects and grasp them in a way such to not hurt people around it while navigating. The Mobile Manipulator must be able to perform recovery actions in case a major failure happens so as to be able to complete the assigned task even in the case of a non-catastrophic event. In order to exploit the last results coming from the State of the Art, the control system for the mobile base and the robotic arm has been designed considering them as a holistic system. The entire software structure has been realized using the *Robot Operating System* (ROS) environment made of packages that provide different functionalities, and using the typical structure built on top of nodes, topics and services.

1.3.1 Environment Description

The target environment is a Laboratory or a Warehouse, where mobile manipulators can be used by humans to retrieve known objects needed to complete the task they are working on without leaving their position. This environment is characterized by static obstacles like walls, shelves and cabinets, and dynamic obstacles like people and other mobile robots. In order to better reproduce their real use in such an environment, some fixed positions have been defined; the robot is aware of these predefined positions. Objects to pick-and-place are those that can be found in a common laboratory and that do not violate the constraint on the maximum payload: pens, scissors, screwdrivers, rulers and highlighters. The robot will construct and store a map of the environment where it will move before being able to reach the desired positions.

1.3.2 Requested Functionalities

The robot must be able to receive a request from the user through appropriate ROS topics, localize the item inside the environment, pick it and place it in a proper depot. It must evaluate the dangerousness of the requested item and identify its most hazardous part. This latter task is achieved by exploiting a mechanism similar to the one used in IML, with a human operator in the pipeline.

1.4 Thesis Structure

This thesis has a structure organized as follows. In Chapter 2 the state of the art about the requested functionality will be analyzed. An overview of what is ROS, its structure and a description of the used packages can be found in Chapter 3. Chapter 4 contains a description of the Mobile Manipulator used, focusing on both the hardware and software aspects. In Chapter 5, the structure of the software developed is detailed, while Chapter 6 depicts a description of the Data Set used and how it has been obtained. In Chapter 7, we are going to address the results obtained in a simulated environment, while the results obtained with the real robot will be discussed in Chapter 8. Finally, conclusions and future works can be found in Chapter 9.

Chapter 2

State of the art

2.1 Recent Mobile Manipulator Applications

During the last years, several steps have taken to extend the capabilities of mobile manipulators to let them collaborate with humans in increasingly complex tasks. An interesting example of the application of such versatile robots can be found in [12], where the authors developed a software architecture for a legged mobile manipulator to work in a human-scaled kitchen shared with humans. This environment is more challenging if compared to factories, because the latter can be precisely designed to fit the robot's needs, while kitchens, laboratories and offices are designed taking into account the human's needs. In order to successfully work in such an environment, the robot must be able to handle articulated objects and at the same time evaluate its constraints and those coming from the dynamic environment. The proposed solution is articulated in two different stages: first, an *object-centric* planner is exploited in order to understand which is the correct and best way to interact with the item, then an *agent-centric* planner is used to account all dynamics constraints while the interaction is taking place. The first planner provides some proposal about the interaction with the object without considering any information from the scene or from the robot itself: its aim is to find a suitable interaction point and an EE path to follow. Starting from the information coming from the RGB-D camera the robot constructs a 3D map of the environment in order to have all the important information to plan. Regarding the agent-centric planner, the authors propose a benchmark of methods to control the agent, and most of them allow to control the entire robot simultaneously, so considering the robot as a unique holistic system. The main technique used to control the robot is based on *Model Predictive Control* (MPC): starting from the robot's state and the map, the algorithm solves an optimization problem that includes the robot's dynamics, joints and collision avoidance constraints in order to minimize the cost associated to the EE path through a cost function.

Another interesting application can be found in [13]. Here a mobile manipulator was designed to work as a *socially assistive robots* in the *health care* field, working in hospitals and domestic settings. It is able to interact with human patients through different types of *Human Robot Interfaces* (HRIs) such as brain-signal-based, eye-gazed and tangible interfaces. The main objective of the robot is to ensure the blood bag matching during transfusions and general logistics assistance. Regarding the first kind of task, the robot reads the NFC tag of the patient or interacts with he/she through a tablet to ensure the correct match between blood bags; it can also store vital parameters in the meantime. As a logistic assistant, the robot carries tools and materials from one part of the hospital to another: this ability can be particularly useful when the hospital faces an emergency. In order to accomplish such challenging tasks, the mobile manipulator must be able to navigate in cluttered scenes, avoid dynamic obstacles and interact with both people and objects. To address these requirements, the authors employed an omnidirectional mobile base named *PAQUITOP*. The omnidirectional wheels allow the motion along any direction instantly, without any need of complex maneuver. On top of the base it is possible to find a Kinova Gen3lite collaborative robotic arm [14]. It is a 6 DOF manipulator specifically designed with materials and shapes suitable for the human contact with a 2-fingered gripper used to grasp objects. It has been located in an off-centered position to increase the success rate of operations involving interaction with objects, and it also enhances the stability of the entire system. This robotic arm is controlled using the manufacturer's software framework that is based on ROS. An RGB-D camera is mounted on top of the sixth arm link and allows the robot to create a precise and detailed map of its surroundings. In addition to the camera, the system comprises a 2D RPLidar. Data coming from these two sensors are combined to solve the SLAM and the Visual-SLAM problem (VSLAM). To move in cluttered scenarios it is crucial to solve the obstacle detection problem. Researchers inserted some proximity sensors to enrich the data already available from the map. ROS is then used to combine all data coming from all the sensors, manage the motion planning and handle specific requests from the users.

From these two papers, it is possible to notice how the robotic research field is increasingly considering mobile manipulators as effective human assistants. Crucial is the use of a suitable set of sensors that allow the robot to correctly sense the environment to avoid collision and, more importantly, to not hurt people. Also evident is the use of ROS as middleware used to construct the software architecture, thanks to its distinctive expandability by means of all available packages.

2.2 Arm and Base Motion Planning

In this section, a review about the strategies and algorithms currently composing the state of the art for path and motion planning of the arm and base is provided. First of all, it is important to understand which is the difference between path planning and motion planning. *Path planning* purely aims at finding a geometric path to be followed by a representative point of the robot (i.e. the center of the mobile base or the EE of the robotic arm) that satisfies some constraints, including obstacle avoidance and *kinematic constraints* coming from the robot structure. As a result, the computed path usually is a function of a scalar variable s ranging from 0 to 1 providing the percentage of the path that has been completed; no time-related variables are considered. Instead, *motion planning* extends the planned path including the time evolution of all kinematic variables that describe the motion of the representative point, such as velocity and acceleration.

Before entering into the details of the state-of-the-art planners, it is important to define what this planner controls and which variables are affected. Essentially, path and motion planners design the evolution of the variables describing the state of both the arm and the base; path planners decide which value each variable must assume at a given point of the path, while motion planner better expresses the time evolution of these variables that are related to the kinematic laws. The state of the robotic arm is usually described by a set of real values representing the joint's angle. So, if an arm is made of $n + 1$ links and n joints, its state can be expressed as:

$$\mathbf{q_a} = (\theta_1, \theta_2, ..., \theta_n), \tag{2.1}$$

where each $\theta_i, i \in [1, n]$ is the joint angle value assumed by the i-th joint. In a similar way, the state of the mobile base that moves in a 2D space is described by a set of 3 variables:

$$\mathbf{q_b} = (x, y, \delta), \tag{2.2}$$

where (x, y) are the coordinates of a representative point of the base with respect to a fixed reference frame, while δ is the orientation of the base with respect to the z-axis of this frame. Since a mobile manipulator is composed by a manipulator located on top of a mobile base, its complete state q can be expressed as:

$$\mathbf{q} = (x, y, \delta, \theta_1, \theta_2, ..., \theta_n) \tag{2.3}$$

Depending on how the mobile manipulation is treated, it is possible to group the planning algorithms in two classes:

- **two subsystems-separate planners**: in this type of algorithm, the manipulator and the mobile base are treated as two separate and independent systems, so a specific planner is used for the arm and another one is used for the mobile base.

- **holistic planners**: these algorithms consider the mobile manipulator as a unique system, planning for both the arm and the base simultaneously; the base's DOFs are considered an extension to those of the arm, introducing a layer of redundancy.

2.2.1 Independent Planners

The advantage of using two different planners, one for the mobile base and the other for the manipulator, is that the complete movement can be decoupled and divided into two sub-tasks [15]. Using this approach, it is easier to debug and understand which problem occurs and on which sub-module, but the position to be reached by the mobile base such that the goal pose of the EE belongs to the manipulator's workspace must be prerecorded [16, 17], or computed separately, exploiting the knowledge about how the robot is made.

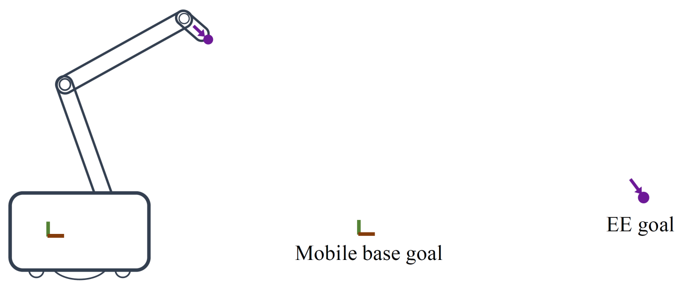
Figure 2.1 represents how the complete motion planning pipeline is divided when using two separate planners. It is possible to distinguish three separate phases:

1. definition of the EE and the mobile base goals (Figure 2.1a);
2. motion planning and movement of the mobile base (arm is kept stationary) (Figure 2.1b);
3. motion planning and movement of the manipulator (base keeps its pose) (Figure 2.1c).

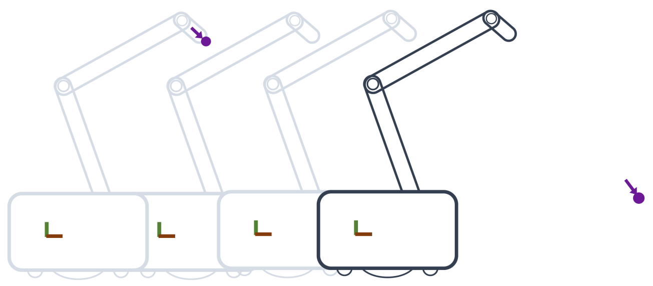
In the following sections, a summary about the main algorithms used as planners for the mobile base and the manipulator is provided.

Mobile Base Planners

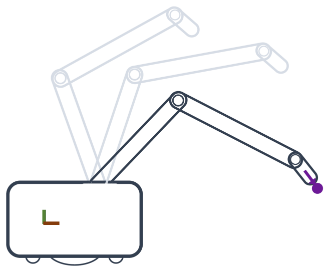
One of the oldest yet the most used path planning algorithm for mobile bases is the well-known *Dijkstra's algorithm* [18]. It allows us to find the shortest path connecting two nodes, source and destination, or the source node and all the other nodes in a (weighted) graph. This algorithm is often used as *global planner* [17],[19],[20]. Here the algorithm is used as a high-level algorithm to find a path connecting the base's starting and ending poses, only considering the free space: free space is divided into cells of predefined size, and each cell's corner constitutes a node in the graph. The global planner usually does not consider the space occupation of the base, instead represents it as a point; to avoid collisions with other obstacles, it is used a *local planner* that, considering the mobile base volume,



(a)



(b)



(c)

Figure 2.1: Illustration of the two subsystems-separate planner motion planning

tries to follow the path designed by the *global planner* adapting it to the shape of the encountered obstacles.

Another famous algorithm used as a global planner is *A** [21]. It is an informed search algorithm similar to Dijkstra (it works on a weighted graph), but its evolution is driven by a heuristic function: at each iteration, the function is computed for all nodes, and the one with minimum values is further explored. This *heuristic cost function* usually has the following expression:

$$f(n) = g(n) + h(n), \tag{2.4}$$

where $f(n)$ is the value computed for node n, $g(n)$ is the cost currently known to reach node n from the start node, and $h(n)$ is the true heuristic function that estimates the cost to reach the destination starting from node n. A* algorithm is again commonly used as global planner [20], [22], [23]. Some extended version of the A* algorithm exists: in [24], authors used the *Anytime Repairing A** (ARA*) algorithm [25]. It is an evolution of the A* algorithm, based on heuristic functions, designed to quickly find the shortest path between two nodes; at each iteration, it refines the computed path using the last information acquired.

In [16] and [26], the motion planning for the mobile base is achieved using a *Rapidly-exploring Random Tree* (RRT) algorithm [27]. The key idea behind this algorithm is to build a tree covering the free space by collecting some random samples. At each iteration a random sample $q_{b,rand}$ representing a possible pose for the mobile base is computed and, if it belongs to the free space (that is if it does not lead the robot into a collision state) it is added to the current tree. The algorithm searches for the nearest node already belonging to the tree $q_{b,nearest}$ inside a region (usually a circle or a sphere) defined by a parameter ρ: if this node exists, then the $q_{b,rand}$ is connected to $q_{b,nearest}$, otherwise the farthest point $q_{b,new}$ from $q_{b,nearest}$ along the line connecting $q_{b,nearest}$ and $q_{b,rand}$ is added to the tree. The algorithm continues until $q_{b,goal}$ is added to the tree or when the time limit is reached; in this latter case one last attempt to connect $q_{b,goal}$ is done. The advantage of this type of algorithm is that the tree can be further expanded at each use. Several extensions and evolutions of this algorithm have been proposed during the years. *RRT** used in [28] optimizes the tree structure in order to obtain a balanced trade-off between the exploration and exploitation of the tree, while *Informed RRT** (IRRT*) adds to RRT* some heuristics to better guide the algorithm evolution. *RRT-Connect* in [26], [29] and [30] has been specifically designed to build and simultaneously update two separate trees until they connect.

In recent years another type of algorithm started spreading. They all belong to the class of the *Genetic Algorithms* (GAs). They have been designed taking inspiration from the natural selection governing nature. At each iteration, a new population of possible solutions is created, and the "chromosome" (a possible solution) with the best characteristics, defined by some evaluation functions, is chosen to generate a new population. In this way, at each step the algorithm refines the solution, converging to the (sub)optimal one. Applications of this type of algorithm as mobile robot planners can be found in [31] and [32].

With the latest advances made in the AI fields, some researchers started using these models to drive some mobile robots, like in [33]. In particular *Reinforcement Learning* (RL) techniques have been used to train the model. They are based on rewards and penalties: the robot interacts with the environment and these interactions affect the environment; based on how it changes and on some other rules defined by the programmer, a reward or a penalty is generated. This method can be very useful in particular when the task to accomplish is very complex, but usually the result is not robust, because of the unstable nature of the algorithm.

Manipulator Planners

Regarding the manipulator trajectory and motion planning, there are several methods developed during the last two decades for industrial manipulators. Common solutions for robot manipulators in the industrial fields make use of algorithms based on the solution of the *Inverse Kinematics* (IK) problem: it requires to determine the joint angles or positions needed to reach a desired end-effector pose. The solution found, together with the initial joint states, can be used to generate a path and subsequently a trajectory in the joint state space. The IK problem can be really difficult to solve, especially if the number of joints increases. Moreover, the presence of revolute joints further complicates the problem solution: in robots built only using prismatic joints there's a one-to-one correspondence between joints values and cartesian coordinates, but this correspondence begins to break down as revolute joints are introduced until it is completely lost in manipulator robots made entirely using this joints. The procedure used to solve the inverse kinematics also depends on the joint's position and links characteristics, and an analytical solution is not always found. Although solving this problem is not so easy and may require a lot of resources, several solutions are based on it. In recent years, some steps have been taken in order to improve its resolution. In [34] the authors proposed several algorithms to speed up the process using both analytical and numerical methods. Thread level parallelism and the *Particle Swarm Optimization* (PSO) are used in [35] to reduce the time cost of the algorithm needed to find a solution. PSO is a technique inspired by the natural movement of some animals like birds or fish: each particle (potential solution) moves in the solution search space by adjusting its position and velocity, exploiting its personal best and the one found globally by all particles together.

More recent developments in the RL field led to apply this learning method to control the motion of a robotic manipulator as well as mobile robots. Some methods to train such a model are presented in [36], but their performances and stability suggest that this approach is still far from being adopted for real-world applications.

Although it is possible to adapt these solutions to the robotic arms on mobile manipulators, not many of them have actually been tested on this type of robot. On the contrary, methods like the RRT one and its variant are largely used and have been tested on mobile manipulators. In [19] and [20], the RRT-Connect algorithm has been used to plan the motion of the robotic arm, because of its efficiency and the possibility of including some collision avoidance constraints and to use it in real-time control loops. Other algorithms used to plan the motion of the arm, as well as the mobile base, are all those belonging to the class of GAs. In [37] authors used a *Self-Adjust GA* to control the continuous motion of the EE by calculating the intermediate configurations for the given trajectory; this algorithm is able to tune its own parameters during the optimization phase to find a near-optimal solution and to converge faster.

Several solutions make use of some pre-built framework available in ROS like the *Open Motion Planning Library* (OMPL) [38], *Stochastic Trajectory Optimization for Motion Planning* (STOMP) [39] and the *Covariant Hamiltonian optimization for motion planning* (CHOMP) [40] libraries. They are often used through the *MoveIt!* package because of their effectiveness and their ease of use and integration with the most commonly used sensors.

Holistic Planners

Separately planning for a mobile base a manipulator is convenient, but using two independent planners leads to sub-optimal plans: combining the optimal plan for the mobile base and the optimal one for the manipulator does not necessarily lead to a global optimal solution for the entire robot. To overcome this problem, it is possible to consider a mobile manipulator as a high DOF system. This allows us to obtain an optimal plan for both the base and the arm that can move simultaneously, thus reaching the mobile base goal and the ones of the manipulator at the same time instant, as shown in Figure 2.2. This way the movement is more fluid, harmonious and natural. The downside of planning considering the whole system at once is the computational complexity and time required to investigate the state space searching for a feasible solution, which means checking for collisions in an augmented state space.

Despite the larger complexity of this planning method, several searchers used it in their studies. In [41] and [42], the authors explicitly computed the free space in which the robot moves and then, by sampling it, computed the desired goal pose of the mobile manipulators. Explicitly computing the free space representation is computationally expensive and it also requires to be updated for any changes in the environment. For this reason, this method is not suitable for highly dynamic environments.

In [29], path planning is considered as a trajectory optimization problem to be solved using the CHOMP library. The developed holistic planner is based on a gradient descent algorithm that tries to minimize a cost function, accounting for both path smoothness and collision avoidance to transform a simple trajectory in a collision-free executable path.

The STOMP planner is presented in [39] and it has been tested on the Willow

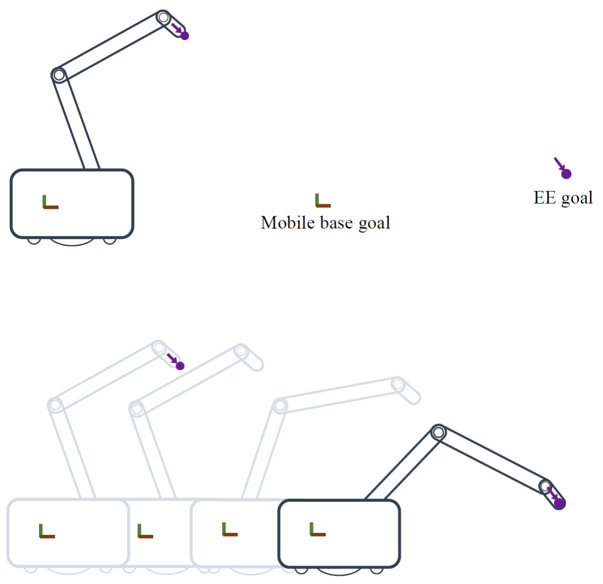


Figure 2.2: Visualization of the motion obtained using a holistic planner.

Garage PR2 mobile manipulators. This planner is based on a derivative-free stochastic optimization technique that refines some initial trajectories, while minimizing a cost function that includes both collision avoidance and EE pose constraints.

Sampling-base planners like RRT and its variants, *Probabilistic Roadmaps* (PRM) [43] and *Expansive Space Tree* (EST) [44] algorithms are also commonly used [45], [46].

Peter Corke et al. in [47] proposed a novel optimization-centered planner based on the solution of a *Quadratic Problem* (QP) to maximize arm dexterity and to obtain fast, smooth and graceful movements. A more natural motion is achieved by considering a 9 DOF system (6 DOFs from the arm, 3 DOFs from the mobile base), where the mobile base is not used only to position the robot manipulator, but it is taken into account also while performing pre-grasp and grasp operations. The algorithm takes as input the desired end-effector pose with respect to the mobile base reference system, converts it into a spatial displacement and exploits differential inverse kinematic to obtain the desired *joint-speed*, both real and virtual (virtual joints are then converted in speed to be applied by the base's wheels using the kinematic model of the base). The important aspect of this controller is that *final joint speed references are computed as a result of a Quadratic Program.* The objective function of QP takes into account the joint velocity and the manipulability, trying to minimize the first while maximizing the latter. Moreover, constraints formulation takes into account joints' position limits and their minimum-maximum joint velocities. A major issue of the discussed approach is that the motion planner is used in an open-loop fashion: planning and moving phases strongly rely on the initial estimation of the real-world environment and on the ability of the robot to follow the designed path. This aspect makes the approaches of this type not usable in dynamically changing environments.

2.3 Object detection and 3D pose estimation

Object detection is a crucial task for a mobile manipulator that has to work as a smart human assistant: the ability to recognize and classify surrounding items is fundamental to correctly address the request of a human. Some tasks also require to understand what parts an object is made of and which actions can be performed with a single part. These requirements can be addressed using some Semantic Segmentation algorithm on top of which a further logic is built to link together parts and actions. Moreover, the robot must be able to understand the position and orientation of the object in the 3D world to correctly grasp and handle the requested item: this is why 3D pose estimation is also needed. Nowadays all these tasks are mostly accomplished using AI and ML models that take images or even videos as the input source. An overview of the main algorithms is proposed hereafter.

2.3.1 Object detection and semantic segmentation

Several works and papers have been proposed in the research field of object recognition and semantic segmentation; their analysis shows how many steps have been done in order to make the used models more and more lightweight and usable on computers with low-performance resources. A first classification can be done by considering how the input images are treated before extracting some useful knowledge from them. To the first category, called *Two-stages detectors*, belong all the models that pre-process an image to generate region proposals, that are regions on which the detector has to focus to find known objects; an example of a two-stage detector's architecture can be seen in Figure 2.3. On the contrary, *Single-stage detectors* are able to extract knowledge from the inputs in a single shot, without pre-processing the image or focusing on specific parts of it (Figure 2.4). Nearly all the models belonging to both the categories are based on *Convolutional Neural Networks* (CNNs): they are characterized by convolutional layers made of weights arranged in matrices where information carried into pixels is combined and elaborated.

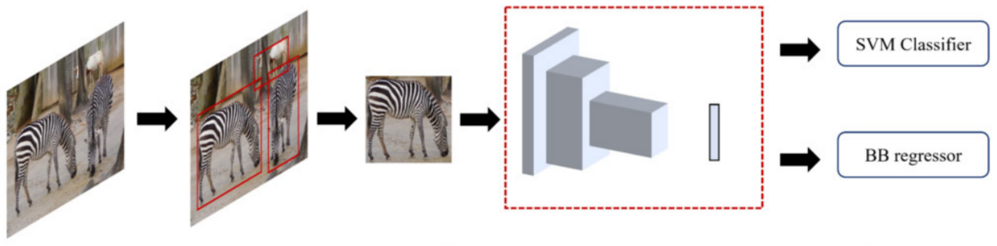


Figure 2.3: R-CNN architecture, one of the first two-stage detectors [48].

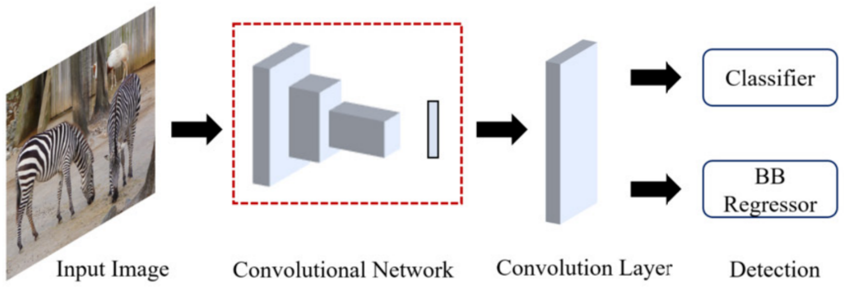


Figure 2.4: Internal architecture of YOLO, a single-stage detector [48].

Two-stages detectors

One of the first models belonging to this category is the *Region-based CNN* (R-CNN) [49]; its success is related to having demonstrated that convolutional layers are very effective in object detection tasks. The input image is first passed through the region-proposal module, then all the parts it found are elaborated by a CNN that computes a 4096-dimension vector for each proposal using the one-hot encoding. Training this network could be very hard: first, the CNN is trained using a classification dataset, then it is fine-tuned to be applied to the specific task and to work on the region extracted by the region proposal algorithm.

To solve this latter difficulty, a new model called *Fast R-CNN* [50] has been proposed. This NN can be trained following an end-to-end approach. Although the architecture is quite similar to the one of R-CNN, Fast R-CNN presents some improvements regarding the computational speed. A further improvement has been done by presenting the *Faster R-CNN*. The main difference with respect to its predecessor is that the region proposal module is way faster, so the overall time required to compute all the predictions has been reduced.

Faster R-CNN has been further improved, giving birth to the *Mask R-CNN* model [51]. It extends Faster R-CNN by adding a mask prediction branch to not only detect objects but also precisely outline their shapes in images, making it a powerful tool for tasks like image segmentation besides object recognition. Some recent applications of this model can be found in [52–54].

Among the state-of-the-art models, there are also the ResNet and its variants [55]: they are fully convolutional NNs that make use of residual block and skip connections to propagate low-level information to a deeper layer. An example of its application can be found in [56].

One-stage detectors

VGG model [57] and its variants have been used for several years to address the tasks of image detection and segmentation. They are CNNs characterized by small filters (convolutional layers) that make them easy to understand and use. Despite their ease of use, VGG networks can be very deep and with a high number of parameters, and this results in slowness of prediction. Moreover, these types of networks are difficult to train form scratch, and easily overfit on small datasets. Their overall performances in terms of accuracy and time needed have been outlasted by more recent works.

A surprisingly high-performance model was the *You Only Look Once* (YOLO) network [58]. It gained widespread popularity in the computer vision field thanks to its high accuracy and fast predictions. In particular, the short time required to compute a prediction leads to the use this network in real-time algorithms [59] or inside feedback control loop like robotic visual servoing [60]. The reasons behind such a high-speed performance are related to the fact that the entire image is processed only one time, and as a direct result, the network can provide labels for the entire image, bounding boxes and the label associated with the object inside of them, semantic regions of interest and their meaning. Such a versatile NN led researchers to further investigate its potential, and as a result of this several variants have been developed. YOLOv3, YOLOv4 and YOLOv5 nowadays represent the state-of-the-art networks used in the computer vision research field and are being used also in industrial products by several companies.

2.3.2 3D pose estimation techniques

The problem of 3D pose estimation refers to the task of determining the position and orientation of an object in a given scene. It is an important task to face in several search fields, like autonomous driving, augmented reality and above all in robotics. To better grasp and manipulate objects robots must perceive them completely, that is determining where an item is positioned, how it is oriented and the amount of space it occupies. Several methods are based on AI and ML models. As illustrated in [61], several 3D detection methods are actually 2D-driven: they use 2D detectors like R-CNN, Fast R-CNN and YOLO to compute the 2D bounding box of an object, and then this information is lifted to the 3D space using data coming from sensors about the depth. Also, those detectors able to directly generate 3D poses are built on top of them, so their performance is strongly related to those of the 2D predictor.

2.4 Grasp pose generation

Grasp pose generation is a fundamental task in robotics, especially in all those applications where a robot has to manipulate some objects by means of one or more arms. The final goal of the grasp generation task is to find a complete pose for the robot's end-effector in order to properly grasp an item. Different approaches to this problem can be found in literature, but some of the most recent ones make use of NNs to possibly generate the best grasping pose.

In the literature, it is possible to find several papers where the task of grasping an object is solved by adapting CNNs designed for object recognition, and in most cases, they sample and rank a set of possible candidates. This usually results in very complex networks with a very high number of parameters and requiring a computational time ranging from couples to tens of seconds. The slow response time leads to treat the simple grasping problem as an open-loop problem.

In [62], authors exploited the advantages of a custom CNN to face this task and to overcome the slowness of other adapted methods. They built a **Generative Grasping CNN** (GG-CNN), whose main purpose is to directly generate possibly the best grasping pose, starting from images collected by an RGB-D camera. The most important result is the size of the final GG-CNN: it has *about 60.000 parameters* and the entire grasping pipeline needs *16 ms* to perform a grasping action. The goal pose has the z-axis oriented along the z-axis of the object to grasp. The designed GG-CNN takes as input the RGB-D image of the object to grasp and outputs a *grasping map*. Each pixel of this grasping map contains 3 key values that characterize the corresponding grasp pose:

1. ϕ is the gripper's rotation angle around the z-axis of the camera's reference frame;
2. w is the gripper width: it indicates how much the gripper must be opened to complete the grasping of a given part of the object;
3. q indicates the probability of completing the grasping task successfully by using that pose.

The best pose in the camera reference frame is then retrieved considering the position (u, v) of the pixel with the highest value of q. From this brief description, it is possible to understand how a GG-CNN is used as a mapping function M that maps the RGB-D image I to the grasping-map G. Figure 2.5 shows a visualization of the input and output obtained using this GG-CNN, while Figure 2.6 depicts the complete pipeline developed to convert an RGB-D image into a grasp pose using a CNN.

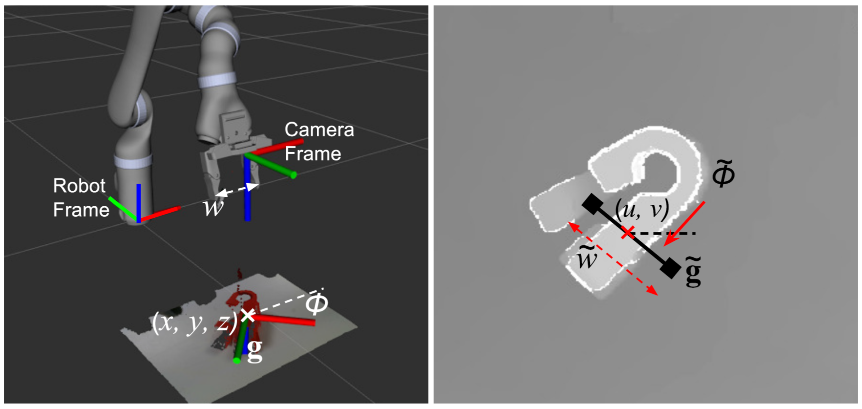


Figure 2.5: Left: visualization of the goal grasping pose. Right: best grasping pose found on the depth image and its parameters [62].

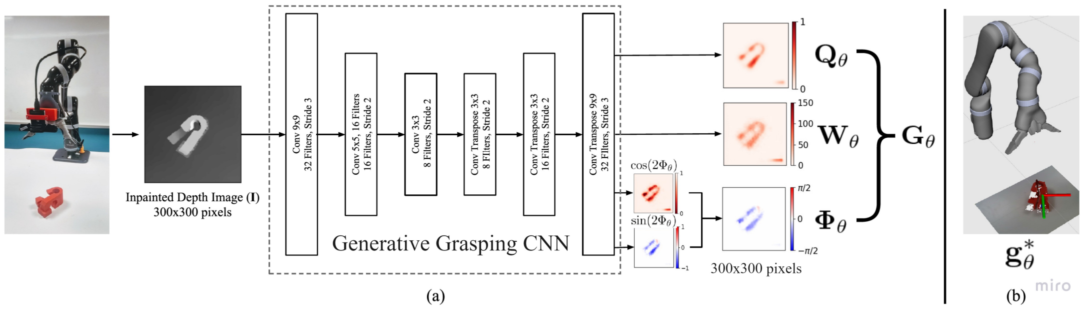

Inpainted Depth Image (I) 300x300 pixels
Conv 9x9 32 Filters, Stride 3
Conv 5x5, 16 Filters 16 Filters, Stride 2
Conv 3x3 8 Filters, Stride 2
Conv Transpose 3x3 8 Filters, Stride 2
Conv Transpose 3x3 16 Filters, Stride 2
Conv Transpose 9x9 32 Filters, Stride 3
Generative Grasping CNN
$\cos(2\Phi_\theta)$
$\sin(2\Phi_\theta)$
$\mathbf{Q}_\theta$
$\mathbf{W}_\theta$
$\mathbf{\Phi}_\theta$
$\mathbf{G}_\theta$
300x300 pixels
$\mathbf{g}^*_\theta$
(a)
(b)
miro

Figure 2.6: Full pipeline of grasping pose estimation [62].

Other works face the task of finding a suitable end-effector grasp pose by conditioning the grasping task to the action that must be carried out. This particular task is known as *Robotic Object Affordance*, and it is based on understanding the functional parts of an object. Several solutions that can be found in the literature are based on CNNs, which takes images or point clouds representing the scene inside which the object to grasp is present.

Paper [63] describes a method used not only to predict the functional part of an object, but also which is the best interaction point to complete a given task. Authors proposed a NN that uses both 2D images and 3D models of objects to interact with: images contain a representation of a human interacting with a given object and each of them has a label indicating which kind of task is being fulfilled. The training phase of this network aims to reproduce a sort of *learning by demonstration* where images constitute the representation from which the robot has to learn.

In [64], a new grasp pose generation algorithm is proposed. It studies the synergy between picking and placing of an object in a cluttered scene to develop an algorithm for task-aware grasp estimation. With task-aware grasp estimation, the authors mean that how the object is grasped depends on placement conditions, and such that the object can be placed in the most convenient way for future grasps. First, the RGB-D camera mounted on the wrist of the robotic arm collects a series of images to reconstruct a 3D render of the placement scene, by means of a series of 3D convolutional layers, and then a single depth image of the object to grasp is collected in order to reconstruct a 3D model of it. This information is then cross-correlated to produce an affordance placement map. The manipulator pick pose is obtained using a function determined through a reinforcement learning process, but the authors specifically designed it starting from the object position in the scene, in order to avoid problems that could occur during the training phase. Despite the high accuracy and success rate obtained, this method is really time-consuming, because of the number of depth images that have to be collected for scene reconstruction before each grasping task.

2.5 Interactive machine learning

Several ML techniques require the supervision of humans to build datasets and to organize the prior knowledge used by models to learn how to complete a given task. A novel way of training AI algorithms involves humans to directly supervise and filter the information that are being learned: this human-in-the-loop approach is known as *Interactive Machine Learning* (IML) [11]. IML can be used in the robotic field in different ways. One of the most used is the so-called *learning by demonstration*: a robot observes the actions of a human operator and learns how to replicate them, or a human controls the robot to instruct it to complete a specific task and the robot progressively learns how to work autonomously. A better understanding of how this complex task can be achieved together with some applications can be found in [65].

Chapter 3

ROS state-of-the-art

Robot Operating System (ROS) is the most used middleware for robotic applications in both the academic researche and industrial fields. It is an open-source framework rich of drivers, libraries and packages implementing state-of-the-art algorithms [66]. New developers can count on detailed documentation and a large community with which to discuss problems or new solutions. ROS can be used for several robotic applications, like indoor, outdoor, automotive and space exploration. Also, it is available for different platforms ranging from Linux to Windows and macOS and even some embedded platforms.

Another important characteristic of ROS is that the software can be easily developed to run on a single computer or multiple computers at the same time without having to implement many changes. This is essentially due to its architecture: it is sufficient for two computers to be connected to the same network to share information and data.

The goal of this chapter is to analyze software developed using ROS works and to provide the main information about the packages available for solving specific tasks.

3.1 ROS project structure

One of the basic features of ROS is that the same installation and packages can be used for multiple projects. Indeed, it is possible to create several projects and select the desired one just by sourcing its configuration file. The projects' independence and isolation are granted by building a **workspace** for each project: a ROS workspace is a directory where custom packages, configurations files and high-level code are contained. It is organized following a specific structure: the *src* folder contains all packages and code needed, the *devel* folder is where are located all files generated after building the project, and the *build* stores some intermediate-build files. Other folders, like the *log* one, can appear in a ROS workspace, depending on how it is built.

3.1.1 Main components

The most interesting thing about using ROS on a real robot is how the entire code structure is organized to deal with each module that constitutes the system. The main components of any ROS software architecture are *nodes*, *topics* and *services*.

Nodes

A ROS node is a running process specifically designed to address a task. Usually, a robot control system is made of several nodes that are part of a communication graph: each node provides a specific *Application Programming Interface* (API) thanks to which communications take place. All these nodes are meant to operate at a fine-grained scale, meaning that each of them has been thought to face a given task or control a specific part of the robot.

Each node is uniquely identified thanks to its *graph resource name*. It is also possible to create a new node in a hidden state, by adding a string randomly generated every time the node is created; in this way, it is not possible to know the specific name. ROS nodes are also characterized by a node type that essentially corresponds to the name of the executable file associated with the running process; node types are specified within each ROS package and identified by their names.

The use of ROS nodes provides several advantages if compared with monolithic codes:

- any problem or crash can be detected and located more quickly:
- code complexity is reduced and it is also possible to change implementations easily;
- node's details can be hidden thanks to the interface each node provides.

Messages

The simplest data that two nodes can exchange by means of the ROS communication graph is a ROS message. *ROS messages* simply are data structures comprising from one to several fields of different types. They can be identified by their message type: it defines the fields composing the message, their types, names and meaning. Usually, each package specifies its own message type that can also be used as field types by the developer to create its own custom type; both standard and custom message types are defined in *msg* files, which are simple texts files with the *.msg* extension.

To ensure data integrity and compatibility, every time a node sends a message to another one, it adds a hash code called MD5 that represents the type of the message. A message is correctly sent only if both the sender and the receiver have the same MD5.

Topics

ROS topics can be seen as the roads on which messages run from one node to another. They have been designed to work following an anonymous publish/subscribe semantics: usually, nodes do not need to know who they are communicating with, they only care about the type of operation to request or they have to execute. For this reason, a node that wants to send a message to another one has to *publish* onto a given topic, while the receiver has to *subscribe* to the same one. Each topic can have multiple publishers and subscribers, and a node can act as both of them.

A topic is uniquely identified by means of its name. They are created as soon as a node expresses its will to subscribe to or to publish onto it, but differently from nodes, topics cannot be created in a hidden state (anonymously). Topics are characterized by a specific *topic type* that corresponds to the type of message exchanged through it.

Regarding the distributed structure of ROS, topics constitute a way of exchanging messages between nodes running on different computers, thanks to TCP/IP-based and UDP-based connections.

Services

The publish/subscribe mechanism provided by means of topics is a flexible and easy-to-use paradigm, but it is not appropriate for all the interactions that are generated by sending a request and for which a reply is expected. This common type of interaction is supported in ROS by means of a service. *Services* in ROS are defined by a pair of messages, one for the request and one for the reply. The request/reply mechanism through services involves a provider and a client:

1. the provider is the node that offers the service and, as a callback to the receipt of a request, sends the reply;
2. the client is the node that sends the request to the service whose name it knows, and waits for the reply to be sent back.

ROS services are used for all those tasks where the client node needs feedback on the requested action to understand how to proceed, but not only. Another typical application is to retrieve data that are continuously published on a specific topic: subscribing to such a topic would mean executing the callback repeatedly, without the possibility to make any other computation. A solution to this problem is to design a node that subscribes to the continuously updated topic and manages the data stream, and in the meanwhile offers a request/reply mechanism by means of a service. In this way, all nodes that need to retrieve data in a specific time instant can send a request to this node and wait for the reply.

Master node

The *Master Node* is maybe the most important node of a ROS project, and for this reason, it is the first one to be created when the code starts. It provides naming and registration services to the rest of the nodes in the ROS system, tracks publishers and subscribers to topics and services, and it builds the ROS Parameter Server. Essentially, it is in charge of enabling each node to find one another and of establishing communication among all the nodes.

Parameter Server

The *ROS Parameter Server* is a dictionary shared with all the nodes and resources. It is used to store and retrieve parameters at runtime. It is mainly used to store static data, such as configuration parameters and robot's details.

3.2 Working in simulation

Every time a new robotic project has to start, one of the phases a developer has to face is the simulation of the expected environment and of the robot's behaviour. Working in a simulated environment has several advantages: first of all, the robot may be not available to conduct experiments directly on it, so simulating its behaviour is the only chance, but even in cases where it is possible to use the real robot there are lots of reasons to conduct preliminary experiments in simulation. Robotics often require to use and test robots one is not familiar with, resulting in a not well-controlled system that can be dangerous; simulating the robot's behaviour allows one to understand how it operates, which are its characteristics and limits improving safety. It is also possible to build a controlled environment to investigate how a given robotic system reacts to it and how it could be possible to improve it to face new challenges. Simulation is also used for its rapid response to any kind of change: it is possible to quickly test different configurations, collect the results and make decision on what to do next, and this fast iteration helps to find the best solution in a shorter time.

Because of its importance, ROS provides some useful tools to build a simulated and controlled environment to see how the robot behaves in it. In addition, it is also possible to understand how the robot perceives the environment and which is the knowledge extracted from it.

3.2.1 Gazebo

Gazebo [67] is the most used robotic simulator. It allows us to create a complete and complex 3D world and to simulate the dynamics of different robots and their interaction with the environment. Figure 3.1 shows a simulated warehouse created with Gazebo. It appears clear how it is possible to insert any kind of object in it together with one or more robots.

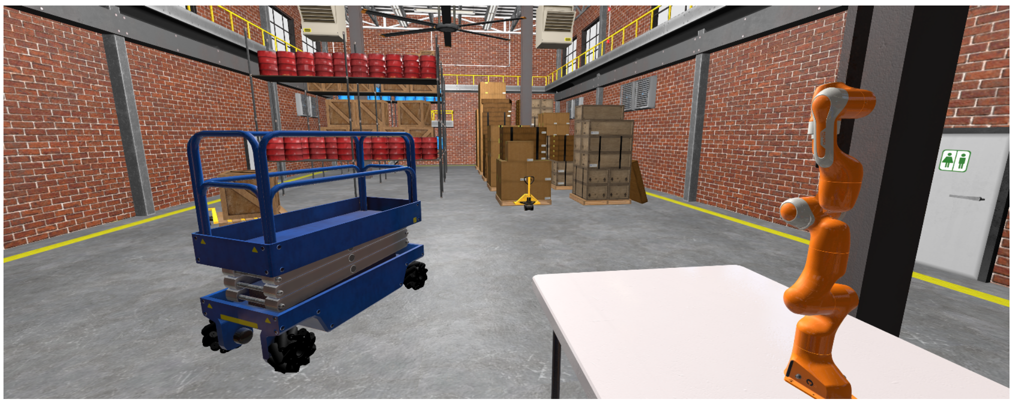


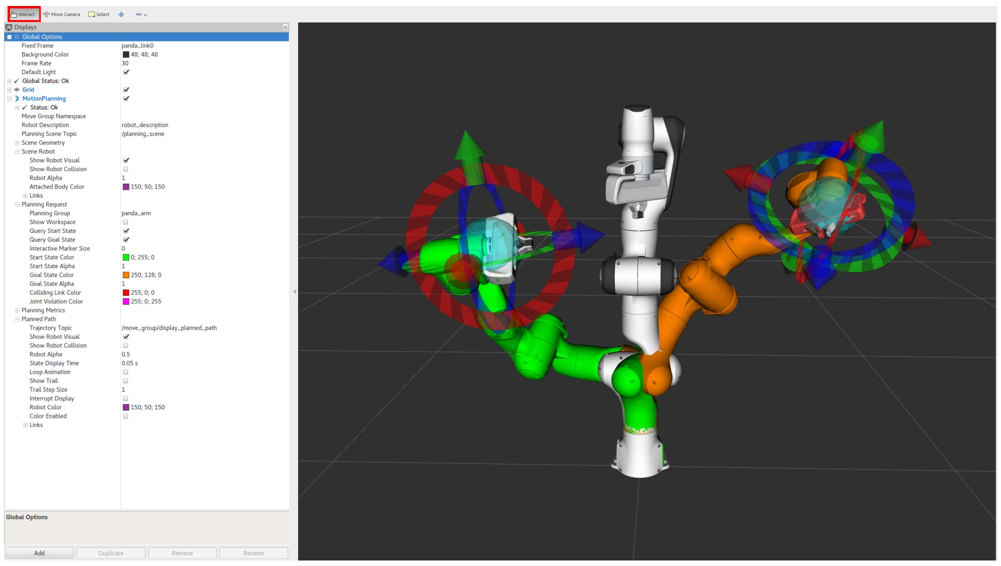
Figure 3.1: Simulated warehouse in created in Gazebo [67].

Gazebo uses physics engines, such as Open Dynamics Engine and Bullet, to simulate the physical behavior of objects and robots. This allows for accurate modeling of robot dynamics and interactions with the environment. It also supports various sensors, including cameras, LiDAR, IMUs, and more, and they are simulated within the environment, providing sensor data as if they were mounted on a physical robot. Several plugins are available and can be used to extend the simulation environment and to obtain a more realistic behaviour.

Gazebo's fame is mainly due to its integration with ROS: nodes can access simulated resources and data as they would do on the real robot. This aspect leads to easily use algorithms and complex solutions tested in simulation on the physical system.

3.2.2 Rviz

Rviz [68] is a 3D visualization environment that displays what a robot is seeing, thinking and doing; exactly understanding how the robot is perceiving the world allows to faster up the the debuggin process and solve problems. This powerful tool supports a lot of sensors like cameras, LiDARs, laser scans and others and allows to analyze all the data by means of specialized monitors and to customize how data are visualized thanks to their settings.



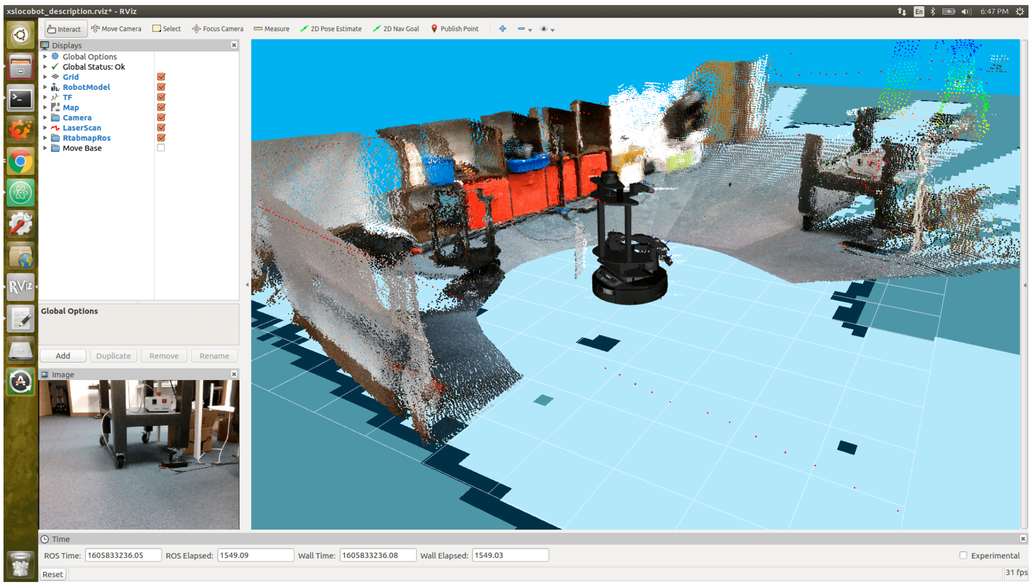
```
Global Options
    Fixed Frame                 panda_link0
    Background Color            ■ 48; 48; 48
    Frame Rate                  30
    Default Light               ✓
✓ Global Status: Ok
Grid                            ✓
MotionPlanning                  ✓
  ✓ Status: Ok
    Move Group Namespace
    Robot Description           robot_description
    Planning Scene Topic        /planning_scene
    Scene Geometry
    Scene Robot
      Show Robot Visual         ✓
      Show Robot Collision      ☐
      Robot Alpha               1
      Attached Body Color       🟪 150; 50; 150
      Links
    Planning Request
      Planning Group            panda_arm
      Show Workspace            ☐
      Query Start State         ✓
      Query Goal State          ✓
      Interactive Marker Size   0
      Start State Color         🟩 0; 255; 0
      Start State Alpha         1
      Goal State Color          🟧 250; 128; 0
      Goal State Alpha          1
      Colliding Link Color      🟥 255; 0; 0
      Joint Violation Color     🟪 255; 0; 255
    Planning Metrics
    Planned Path
      Trajectory Topic          /move_group/display_planned_path
      Show Robot Visual         ✓
      Show Robot Collision      ☐
      Robot Alpha               0.5
      State Display Time        0.05 s
      Loop Animation            ☐
      Show Trail                ☐
      Trail Step Size           1
      Interrupt Display         ☐
      Robot Color               🟪 150; 50; 150
      Color Enabled             ☐
      Links

Global Options
```

Figure 3.2: Example of the Rviz interface used to interact with a robotic arm [69].

Thanks to all the plugins and ROS packages supported, Rviz allows to understand how the robot intends to solve a given task. For instance, the presence of visualization markers in the form of arrows and lines led different motion planning researchers to use them to show goals, poses and even trajectories followed to accomplish the motion task. Even the navigation stack can be easily understood, thanks to the possibility of displaying costmaps and dynamically-updated paths followed by mobile robots, as it is possible to see in Figure 3.3.

3.3 Perception and localization

The capability of understanding how the surrounding world is made is crucial for a robot, especially for mobile robots that have to move within this world. Moreover, to properly navigate a robot has to understand its positions with respect to a



ROS Time: 1605833236.05 ROS Elapsed: 1549.09 Wall Time: 1605833236.08 Wall Elapsed: 1549.03

Figure 3.3: Rviz interface for the costmap used by a mobile manipulator to navigate.

reference frame. For these reasons, some ROS packages to map and create different representations of the environment are available.

3.3.1 Octomap

Regarding the environmental perception, one of the best tools available in ROS package is Octomap [70]. It is used to create a 3D occupancy grid such that the robot can use it to address the collision avoidance problem. Each *voxel* (3D generalization of the well-known pixel in the 2D domain) of the 3D occupancy grid can be free, occupied or unknown: if any point of the robot touches any of the occupied voxels then a collision takes place, while unknown voxels can be used for autonomous space exploration.

Occupancy grids created using Octomap (Figure 3.4) are based on the *octree* data structure: it is an efficient data structure that partitions the 3D space into hierarchical subdivisions, similar to a binary tree but in three dimensions. They are updatable, meaning that every time the occupancy grid is used it is possible to add new information or correct previous ones. Updating is done in a probabilistic fashion such as to deal with measurements noise and dynamic changes or obstacles. Grids created with Octomap are characterized by different resolutions, meaning that the number of voxels used to represent a portion of the space may change; a high-level planner could use a coarse map to speed up the planning, while a local planner uses a finer one. The most interesting feature of Octomap is that all

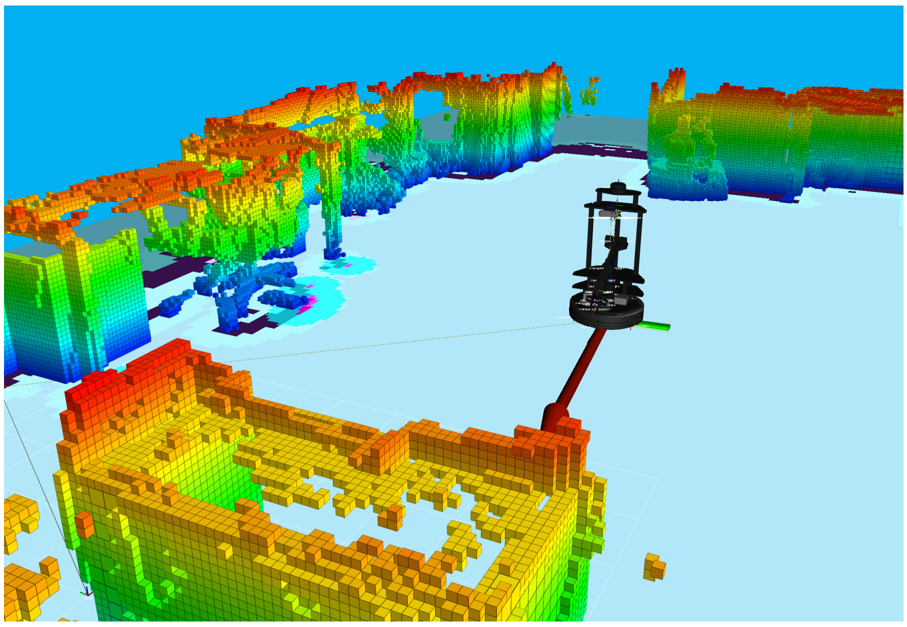


Figure 3.4: Occupancy grid of the experimental environment created using Octomap.

generated grids can be expanded by several robots at the same time and lots of motion planners can use them in order to directly generate collision-free paths.

3.3.2 RTAB-Map

RTAB-Map [71], an acronym that stands for *Real-Time Appeareance-Based Mapping*, is a well-known package widely used to solve the *Simultaneous Localization and Mapping* (SLAM) problem. It creates a precise map of the environment while the robot is moving. The generated map itself is then used to understand the exact position of the mobile robot, to determine the position of obstacles and hence possible and forbidden points that can or cannot be reached. On top of the generated map, planners work to find the most suitable path that satisfies a given set of criteria.

This package exploits an approach based on the similarity that can be noticed between different images coming from the scene: they are used to feed a global Bayesian loop-closure detection algorithm that incrementally expands and refines the map created. The main focus of this loop-closure detection is to find out how likely a new image comes from a previously seen scene or from a totally new one. Similarity detection is based on globally saving a set of core features, called *bag of words*, coming from every already processed image in order to compare them with those coming from new frames. According to the update frequency of the loop-closure detection algorithm, a new hypothesis is formulated to understand whether the image describes a completely new scene or not: core elements are analyzed and a decision is taken. If a new area is discovered, a new set of core elements is stored and will be used later together with the others.

Loop-closure is fundamental to solve the SLAM problem: algorithms like the one used by RTAB-Map essentially refine information coming from the odometry of the robot using information coming directly from the world to correct the intrinsic error of the robot actuators. The key ideas of loop-closure algorithms is to explore a new area, compute the position of the robot based on the odometry and then correct the error accumulated by looking at a previously visited location. Thanks to this redundant analysis it is possible to retrieve the scene's known bag of words (set of elements characterizing that scene) and by analyzing their detected positions it is possible to correctly localize the robot.

Regarding RTAB-Map, loop-closure is also used to refine and clean the map while the robot moves. This is necessary because of the probabilistic nature of the algorithm behind this package. Data coming from RGB-D or RGB camera and Lidars are used to classify each point in the 3D world, where the robot moves, as free or occupied, following a Bayesian procedure. As a result of this procedure, map delimiters appear to be noisy if a given scene is not analyzed enough times and if not involved in a sufficient number of loop-closure iterations. In Figure 3.5 it is possible to see how a noisy map of a certain area is firstly generated and how it is refined as the mobile robot keeps including it in loop-closure detection.

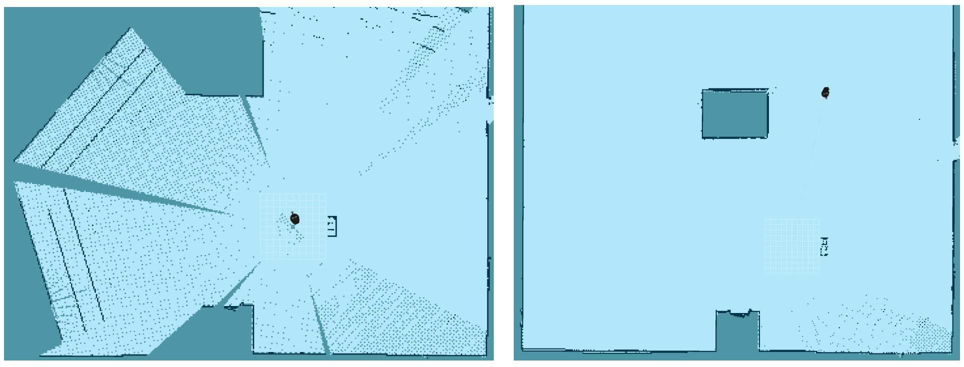

(a) **(b)**

Figure 3.5: Comparison between a coarse map (a) and the refined one (b).

Slam Toolbox

An alternative to the RTAB-map package is the Slam Toolbox. It is a comprehensive set of tools for 2D SLAM that includes various features, including standard 2D SLAM algorithms for mobile robots with additional utilities like map saving. It also allows ongoing refinement, remapping, or continuation of mapping using a saved pose-graph. The toolbox supports lifelong mapping, enabling the loading of a saved pose-graph to continue mapping while removing unnecessary information from new scans. Additionally, there is an optimization-based localization mode based on the pose-graph, with the option to run localization mode without a prior map for "lidar odometry" mode, incorporating local loop closures. The toolbox operates in both synchronous and asynchronous mapping modes and offers kinematic map merging, including a forthcoming elastic graph manipulation merging technique. It includes plugin-based optimization solvers, featuring a newly optimized Google Ceres-based plugin, an RVIZ plugin for user interaction, and graph manipulation tools in RVIZ for adjusting nodes and connections during mapping. Furthermore, it supports map serialization and lossless data storage.

3.3.3 Navigation stack

An important component that allows every mobile robot to effectively move and navigate is the so-called *Navigation Stack*, usually abbreviated as Navstack [72]. This element can be considered as a meta-component, meaning that it is based on several other components to work. First of all, it needs a map of the environment: without a precise map it would be impossible to correctly navigate. On top of that, some other maps with different meanings are built: they are called costmaps and represent the diverse positions the robot can reach without colliding with different kinds of obstacles. All these maps are then used by path planners, both global and local, to compute feasible collision-free paths that the mobile robots must follow.

Costmaps

These kinds of maps are usually built in the form of *inflation layers* or occupancy grids. Their aim is to extend the basic map of the environment to mark whether a point can be effectively reached by the robot or not, rather than simply saying if a point is free or an object (or part of it) is located there. In order to accomplish this task, costmaps must take into consideration not only data from the sensors, but also the robot's physical characteristics.

One of the most used ROS packages to create this type of map is the *costmap_2D* package [73]. With this package costmaps are built in the form of inflation layers and stored as 2D occupancy grids. *Inflation layers* are simply created considering the static map and the obstacle position and the geometry of the robot: a geometric shape that best suites the real robot's shape and space occupation slides along the walls and the obstacles, and the surfaces that are created as result are marked as forbidden, because they would result in a collision. In such a way, all free points in the static map that do not belong to any inflation layers are considered as feasible points that the robot can possibly reach. Costmap_2D exploits both the information coming from the static map and from any supported sensor: this allows to not only include static obstacles but also dynamic ones.

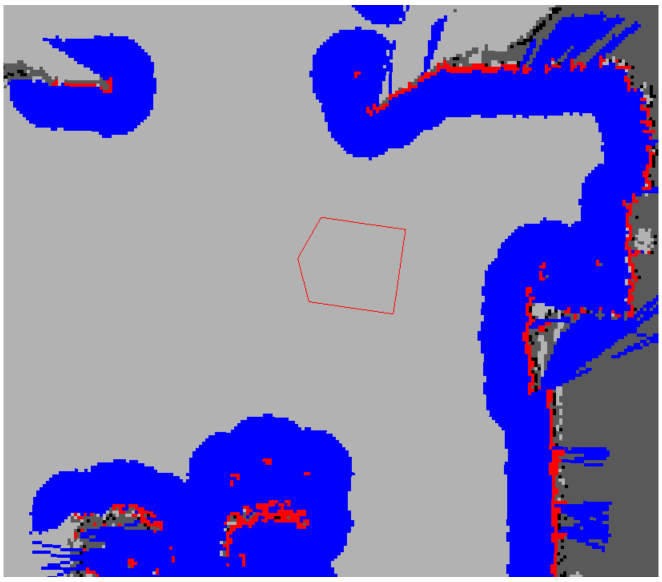


Figure 3.6: Visualization of a costamp created using costmap_2d. Red pixels indicate the exact position of the obstacles, while blue ones constitute the inflated obstacle regions.

Global and Local Planners

The navigation stack needs two planners to correctly work: a global planner that is in charge of finding a suitable path for the mobile robot to go from the starting position to the destination, and a local planner that dynamically adapts the planned path according to what is happening near the robot. Regarding the global planner, it usually exploits only knowledge coming from the static map created as a result of the SLAM problem, i.e. the coarse information about free and occupied points, differently from the local planner that uses mainly information coming from the costmaps, meaning that it also considers proper characteristic of the robot.

As already described in subsection 2.2.1, several algorithms do exist and work as global path planners, including *A**, *Dijkstra's algorithm*, *D**, *Genetic algorithms*. Regarding ROS, there exists a package that allows the selection of the desired planner among some of those cited before: it is the *global_planner* package [74]. Using this package, it is possible not only to choose the planner, but also how the path should be generated. In this thesis work *A** has been used as a global planner. Regarding the local planner, several choices are available, like the *Dynamic Window Approach*, *Trajectory Rollout* and *Elastic Band*. A particularly useful and fast one is a *timed* version of the Elastic Band algorithm: the *teb_local_planner* package [75] provides an implementation of this algorithm for the 2D navigation stack. *Timed Elastic Band* (TEB) is an online optimal local trajectory planner for navigation and control of mobile robots able to deal with non-holomonic robots while avoiding both static and dynamic obstacles. The idea behind this planner is to modify the original path planned by a global planner by connecting two points of the path in order to reduce the time required to complete the path; other metaheuristics and cost functions can also be taken into account. Using the TEB, when a path simplification is possible, the trajectory effectively followed by the mobile robot results to be smoother and faster with respect to the original one; an example can be found in Figure 3.7.

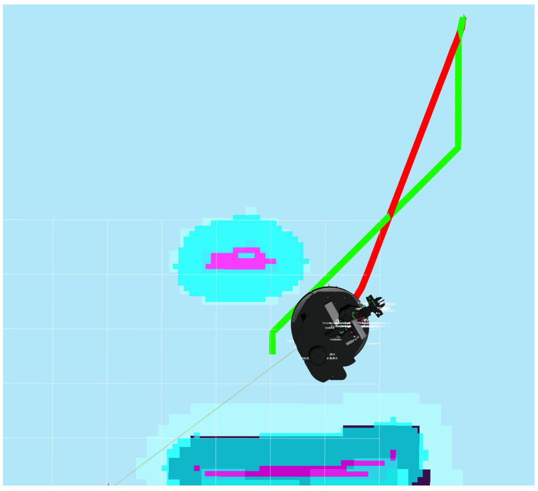


Figure 3.7: Path planned from the global planner (in green) and from the local one (in red).

3.4 Motion Planning

As illustrated in Subsection 2.2.1, to completely control the movement of a mobile manipulator it is needed to use a motion planer for the manipulator and another one for the mobile base, or alternatively a single planner for both of them if the robot is considered as a holistic system. Here, two of the most used packages to control the two separated parts are described.

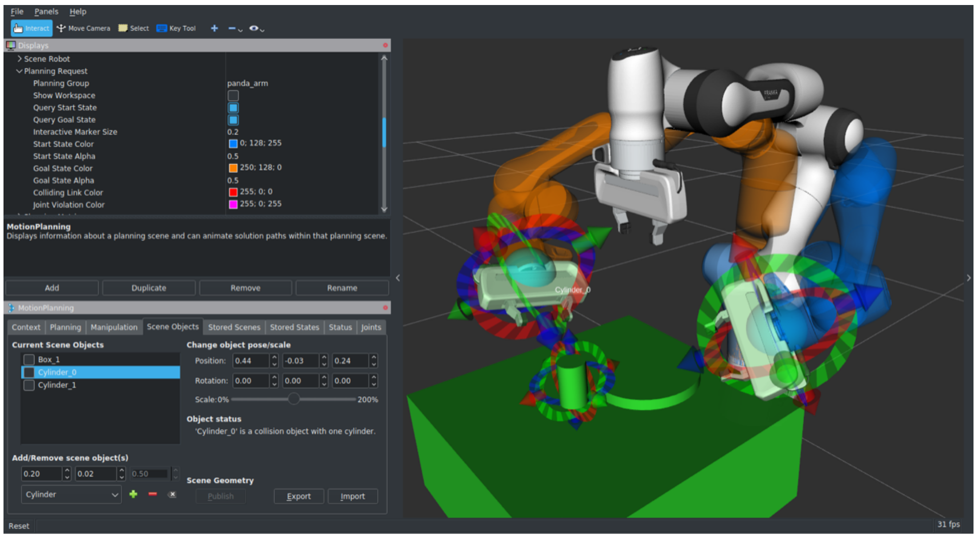
3.4.1 MoveIt! framework

MoveIt [76] is an open-source planning framework developed and widely used for robotic manipulators of any kind; it has been used to develop and test movement packages for over 100 different robots, from the deep sea to outer space, from hobbyist to industrial applications. It is a flexible way to generate a complete motion plan considering different constraints. Several plugins are available to further inspect a specific planning objective.

Starting from the *Universal Robot Description File* (URDF), that contains all information about the physical properties and constraints of the robot, it is possible to create a custom version of the MoveIt! package precisely designed to work with the desired robot. The *moveit setup assistant* is in charge of loading the URDF file, extracting all important information and creating the entire package following the requests provided by the user through a convenient graphical interface. Through this interface, exploiting the information coming from the URDF, it is possible to generate all information needed for the self-collision checking, i.e. which parts of the robot are adjacent, hence they collide by default, and those who may collide while moving. It is also possible to define some planning groups, sets of joints and links to control simultaneously; for each group, it is possible to define a set of known poses, called *named poses*, that can be assigned as desired poses for the planning pipeline. An end-effector can be identified and used in the planning process: it is enough to express which are the constitutive joints and to which link they are attached. MoveIt! makes available different cutting-edge planning algorithms from different libraries; robotics developers and researchers can switch easily from one to another to compare them and select the best one or for benchmark purposes. Several plugins are available to further include data coming from sensors and extend the planning abilities, updating the environment with which the robot interacts and making it a fundamental part of the planning pipeline.

Enhancing robot description

MoveIt! gives the possibility to slightly modify the robot description to make it able to move in the environment, with low effort. The key for this change is to exploit *virtual joints*: a virtual joint can be introduced between the root joint of the robot and the world frame to state the motion ability of the entire robot. The virtual joint movement can be exploited to update the robot's position with respect



Planning Group	panda_arm
Show Workspace	☐
Query Start State	☑
Query Goal State	☑
Interactive Marker Size	0.2
Start State Color	0; 128; 255
Start State Alpha	0.5
Goal State Color	250; 128; 0
Goal State Alpha	0.5
Colliding Link Color	255; 0; 0
Joint Violation Color	255; 0; 255

MotionPlanning
Displays information about a planning scene and can animate solution paths within that planning scene.

Current Scene Objects
Box_1
Cylinder_0
Cylinder_1

Change object pose/scale
Position: 0.44 -0.03 0.24
Rotation: 0.00 0.00 0.00
Scale: 0% 200%

Object status
'Cylinder_0' is a collision object with one cylinder.

Add/Remove scene object(s)
0.20 0.02 0.50
Cylinder

Scene Geometry

Figure 3.8: MoveIt! planner in Rviz [77].

to the world reference frame as a result of an indirect movement coming from a mobile base. This kind of joints can also be involved in the planning pipeline actively: planning algorithms are able to consider the virtual joints in order to extend the working space of the robot. They are able to generate a plan that includes the movement of the entire robot, thanks to the presence of virtual joints, even if the movement of the robot is then demanded to another planner or controller.

Regarding the movements that can be given to a robot thanks to virtual joints, they strongly depend on the type is being used. There are 3 types of virtual joints available in the MoveIt! framework:

1. **fixed** joints are used only to express that the robot is attached to the world through a specific joint;
2. **planar** joints give the robot the ability to move in a 2D surface; so they are able to move along the x and y directions, and rotate around the z axis;
3. **floating** joints make the robot able to move in the 3D, so they provide 6 new DOFs (translation along the x,y and z axes, and rotation about the x, y and z axes).

Motion Plan request

The motion plan request specifies what you would like the motion planner to do. Since the MoveIt! framework is usually employed to control manipulators, it gives the possibility to request to move the arm from one point to another one in the joint space or to reach a given end-effector pose.

The motion plan request can also include **constraints**:

- **position constraints** force a link to be in a precise region of the space;
- **orientation constraints** restrict the orientation of a link to lie within specified roll, pitch or yaw limits
- **visibility constraints** restrict a given link to rely on the visibility cone of a specific sensor;
- **joints constraint** force a joint to lie between two values.

Motion Plan result

The result that will be generated is a trajectory that satisfies all the submitted constraints; the trajectory can be executed by a proper controller. It is important to notice that the result is not simply a path, but it is a trajectory, so it includes acceleration and velocity constraints specified.

Planning pipeline

The entire motion planning pipeline is contained in a unique ROS node called **move_group**: it pulls together all the plugins (including the motion libraries) to provide a set of ROS actions and services for users to use.

Regarding the user interface, the user can access all services and actions provided by the move_group node by means of three different interfaces:

- C++ interface, thanks to the *move_group_interface*;
- in Python through the *moveit_commander* interface that simply is a wrapper for the C++ one;
- graphical interface in Rviz.

The first node with which move_group communicates is the ROS parameter server. They are connected because some important configuration parameters used in the planning pipeline are contained there, including the robot's URDF and SRDF (*Semantic Robot Description File*), together with proper MoveIt! configuration files. The other important player that directly communicates with the planning node is the robot itself: information like joint states and environment information are needed to obtain a suitable plan. This communication is possible thanks to a set of topics and actions:

- joint states information are obtained by listening to the */joint_states* topic that essentially indicates the state (in terms of position, velocity and effort) of each joint;
- information about joint states is not enough; for this reason, the move_group also communicates with the *robot_state_publisher*: it provides information about the location of each link with respect to the world reference frame, but also gives the possibility to the motion planner to update the *tf* tree while moving;
- the *planning scene monitor* is part of the move_group node itself and contains a representation of the world surrounding the robot, so it is used for collision checking.

An important note about the planning scene monitor is that it also allows to consider the object with which the robot may interact. Considering a pick-and-place task, after the robot picked a given object, it is possible to consider the new state of the robot with the object thanks to the planning scene monitor by attaching the object to the robot, and hence by including it in the planning scene. As a result, until the object is not removed from the planning scene the planner will also account for collisions of the object with other parts of the scene.

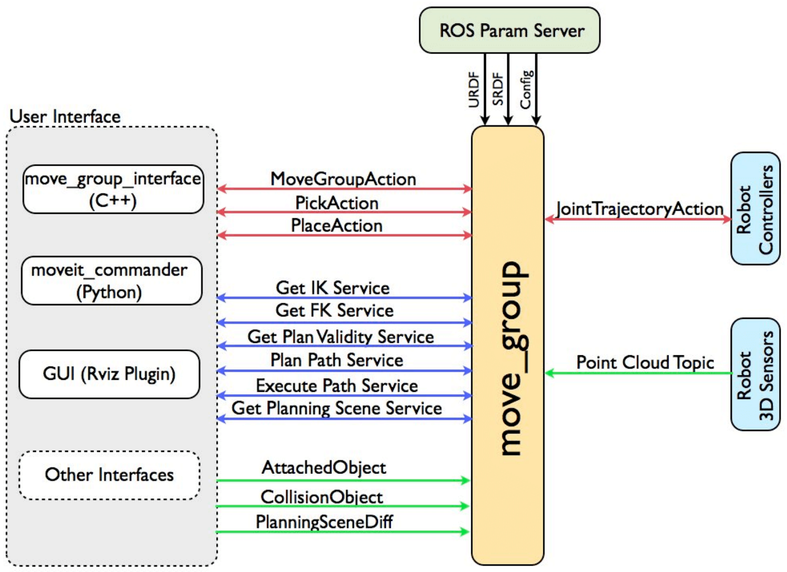


Figure 3.9: Representation of the move_group's communication graph [78].

Plugins

One last thing that is worth to point out is that MoveIt! gives the possibility to use several plugins to enhance the planning capabilities. One of the most useful plugin available is the one that allows the interaction with the occupancy map created using Octomap. This information is then passed to the library used for collision checking to extend the perception of the world, and possibly deal with dynamic obstacles.

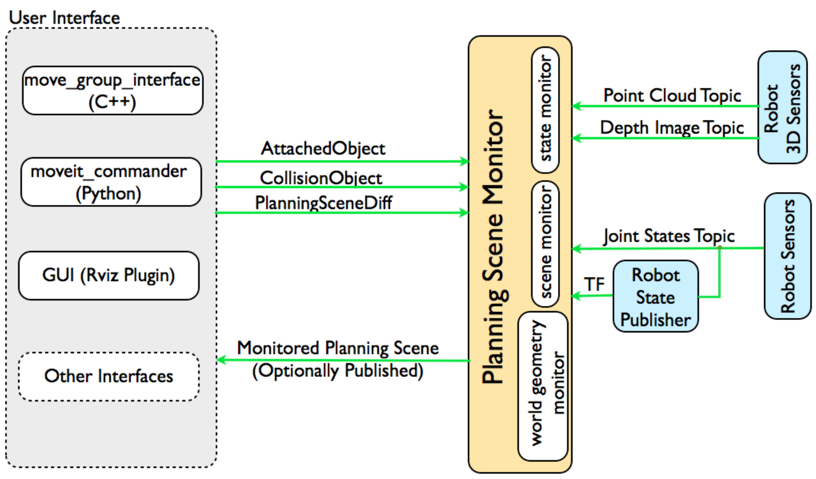


Figure 3.10: 3D perception pipeline available in MoveIt! [78].

3.4.2 Move Base package

A widely used package to control the movement of mobile robots is the *move_base* package [79]. It provides a flexible framework that implements an action server that, given a goal, will attempt to find a path to reach it while avoiding obstacles; different sensors can be included in the path planning. The move_base node links together a global and local planner to accomplish its global navigation task and they work on the global and the local costmaps thanks to the costmap_2d package.

Everything that is needed to control the movement of a mobile robot using the move_base package is contained in the *move_base node*: it provides a convenient interface with the navigation stack of the robot. Figure 3.11 shows the communication structure of this node. As it is possible to see, this node takes as input:

1. information coming from the tf tree and the odometry sensor of the robot to precisely determine where the robot is;

2. the static map of the environment;
3. other information coming from different sensors used to augment the sensing of the robot itself.

This information is then combined together with the *goal pose* to plan a path; the sensor's information and the static map are used to build the global costmap, while the local one is updated only using data coming from sensors. These maps are then fed to the planners, where the global is in charge of computing the path from the starting point to the destination, while the local one updates it according to what happens around the robot.

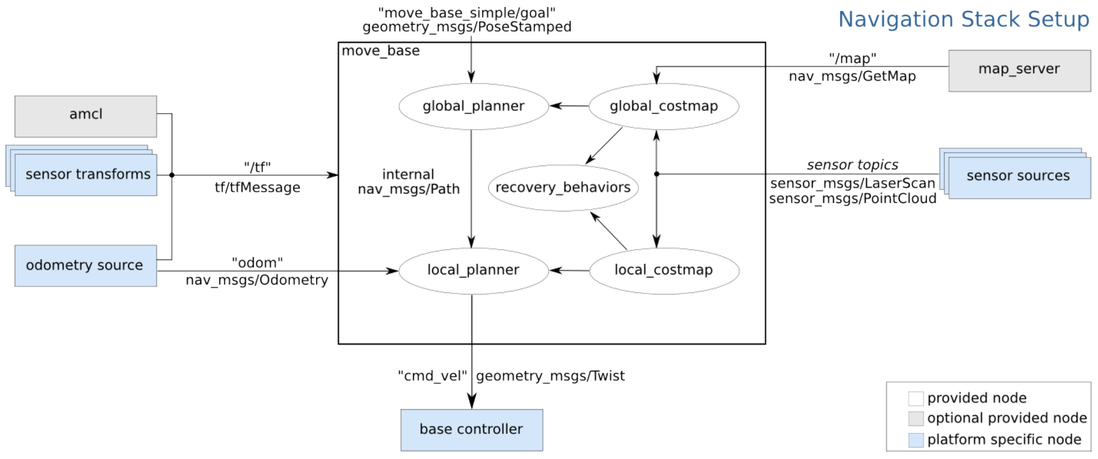


Figure 3.11: Description of the move_base node's communication structure [79].

What is interesting to notice is the presence of a *recovery_behaviors component*: it is need because it may happen that the robot get stuck while moving for multiple reasons (sensor data are too noisy, the goal becomes temporary unreachable). To face this challenge, the move_base package provides a plan to recover the planned movement; Figure 3.12 shows the intended pipeline executed every time the robot gets stuck or a fail occurs. Initially, the robot removes all obstacles that are beyond a region specified by the user on its map. Next, if possible, the robot performs a rotation maneuver to create additional free space. In case this rotation maneuver is unsuccessful, the robot will take a more assertive approach, removing all obstacles located outside a rectangular area where it can rotate in place. Next, the robot performs another rotation in place. If all of these attempts are unsuccessful, the robot will deem the intended goal unattainable and inform the user of the mission abort. Configuration of these recovery behaviors can be done via the "recovery_-behaviors" parameter and disabled via the "recovery_behavior_enabled" parameter.

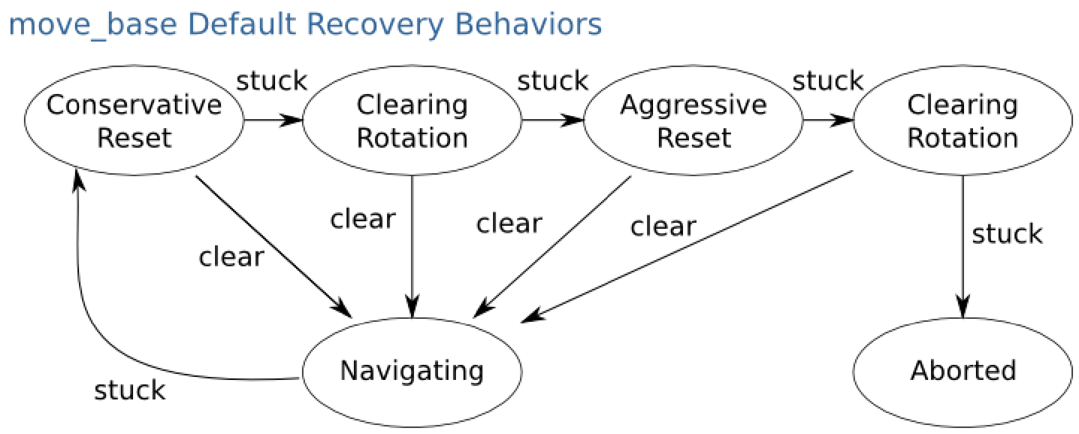


Figure 3.12: Visualization of the recovery behaviours pipeline provided by the move_base package [79].

One last thing that is important to notice is that in Figure 3.11 some components are marked using a light blue color. They are platform-dependent and must be provided by the user, meaning that the developer is in charge of providing and configuring them such that they can work together.

Chapter 4

Locobot WX250s description

The mobile manipulator used in this thesis work is the LoCoBot WX250s from Trossen Robotics [80]. It is equipped with a 6 DOFs arm directly on top of a holonomic base, an RGB-D camera and a 2D Lidar mounted on top of it; Figure 4.1 shows a front view of this mobile manipulator.

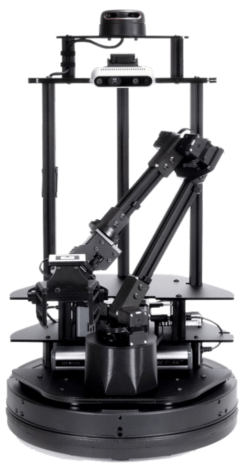


Figure 4.1: Front view of the LoCoBot WX250s [80].

The objective of this chapter is to provide a description of the robot's main components, starting from the characteristics of the two main constitutive components (the arm and the base) and ending with the sensor with which it is equipped.

4.1 Manipulation Hardware

The robotic arm is one of the two principal components that build up the LoCoBot. Several configurations are available on the Trossen Robotics website: it is possible to mount an arm with 4, 5 or 6 DOFs, or even choose to not include the manipulation functionalities. The manipulator mounted on the used robot is a WidowX 250-6DOF Robot Arm (Figure 4.2); all details can be found in [81].

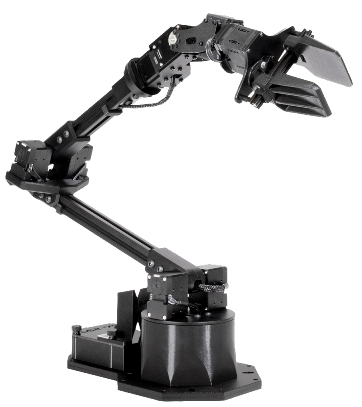


Figure 4.2: WidowX 250-6DOF Robot Arm used on the LoCobot WX250s.

This robotic arm is characterized by 6 degrees of freedom, actuated by a total of 9 Dynamixel servos. These servos are particularly known for their industrial-level precision, accuracy and versatility; thanks to them, the repeatability of the arm is about 1 millimeter. The manipulator is able to reach points at a maximum height of 680 mm, while the overall span is of about 1360 mm, leading to a workspace corresponding to a portion of a sphere (when the mobile base is fixed); the workspace is limited by the presence of the tower on top of which the Lidar and the camera are mounted, the mobile base and the floor. Thanks to the use of lightweight but rigid materials mainly composed of aluminum, this robotic arm is able to interact with and handle objects with a maximum weight of 250 grams. An open-source model of this arm is available and can be used by developers and researchers to work with several software and ROS packages like Gazebo, Moveit and MATLAB.

Regarding the grasping abilities, the default gripper is made of two parallel fingers actuated by an electric servo to open and close them. Besides the default setting is more than enough for most applications, the constitutive details and the schematic of the gripper can be easily found, leading to the possibility of 3D-printing sophisticated and purpose-driven end-effectors. The main characteristics of this manipulator considering the default gripper can be found in Table 4.1.

Degrees of Freedom	6
Reach	680 mm
Span	1306 mm
Repeatability	1 mm
Accuracy	5 mm - 8 mm
Working Payload	250 g

Table 4.1: Summary table of the WidowX 250 6-DOF robotic arm.

4.2 Mobile Base

The LoCoBot WX250s is available with two possible mobile bases that allow it to move: the Kobuki mobile base from Yujin Robotics, and the Create3 from iRobot. The mobile manipulator used for this thesis work uses the Kobuki mobile base (Figure 4.3). Its compact rounded design makes it perfect for indoor navigation and for fast prototyping. It is driven by two powered wheels located on diametrically opposite sides of the base, while stability is granted by 2 caster wheels, one on the front side and the other on the back one. Kinematic and geometric characteristics give the Kobuki base the differential drive properties; because of this aspect, it is necessary to add further constraints to the base controller. Regarding the autonomy, this mobile base is equipped with its own rechargeable battery that allows it to autonomously navigate for up to 90 minutes. Main features of the Kobuki base are summarized in Table 4.2.

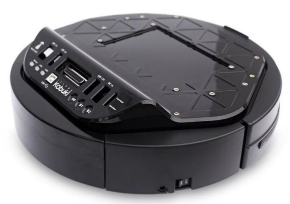


Figure 4.3: Rear view of the Kobuki mobile base.

4.3 On-board computer

All computations and operations required by the robot are executed on an on-board Intel NUC NUC8i3BEH Mini PC. Next Unit of Computing (NUC) is a line of

Diameter	335 mm
Actuated wheels	2
Omni wheels	None
Caster wheels	2
Differential drive	Yes
Wireless	Yes
Battery autonomy	90 minutes

Table 4.2: Summary table of the Kobuki mobile base.

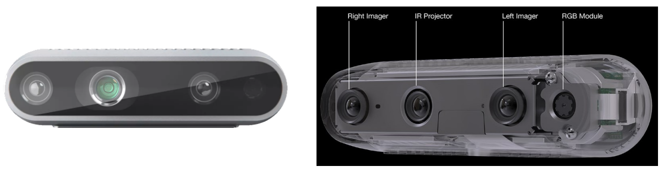
small-form-factor computers and barebone computer kits designed by Intel. The idea behind the concept of this mini PC is similar to the one governing the laptop, that is having all computation capabilities in a reduced amount of space. The on-board computer of the LoCoBot WX250s mounts an 8th Gen Intel Dual-Core i3 CPU with a total of two cores and 4 threads. The memory compartment is composed of 32 GB of DDR4 RAM and a an SSD of 256 GB. Regarding the graphical processing capabilities, the NUC present on the robot is equipped with an Intel Iris Plus Graphics 655 GPU (Graphic Processing Unit).

From this brief description, it is possible to understand that the NUC computer used is able to provide all the computational resources needed to control the robot. The main problem of this system is the reduced available power to address graphical tasks. Due to this limitation, it is not possible to make predictions implement any kind of neural networks on it, hence it is necessary to rely on a distributed architecture where these algorithms are needed. Although it is possible to process data coming from all sensors directly on the on-board computer, overheating and congestion are other major issues concerning the NUC; to avoid these problems, it is preferable to adopt a distributed setting and use the on-board mini PC only to collect data and run the necessary low-level controllers.

Figure 4.4: Mini Pc mounted on the LoCoBot WX250s.

4.4 Camera

The LoCoBot WX250s primarily senses its environment using the Intel RealSense D435 depth camera mounted on top. This RGB-D camera is widely used in robotic research, providing high-quality images for environmental scanning and object recognition. Depth information generates a 3D representation and occupancy grid of the world. This camera system consists of two cameras spaced approximately 5 cm apart, creating 3D images through stereo vision. The depth measurements are improved by combining data from these cameras with information from an infrared projector situated between them. Additionally, an RGB module captures colored scenes, which, when combined with depth data, create a more realistic representation similar to human visual perception.



(a) **(b)**

Figure 4.5: Views of the Intel RealSense D435 depth camera (extern 4.5a, intern 4.5b) [82].

It is important to notice that the orientation of the camera is not fixed: a Dynamixel 2XL servo allows the rotation around two different axes, while controlling both movements simultaneously. Considering a fixed reference frame located in the center of the servo, with the z-axis pointing upwards, the x-axis coming out of the camera and the y-axis such to complete the right-handed coordinate system, the camera can rotate around the z-axis and the y-axis; rotation angles are respectively called pan and tilt.

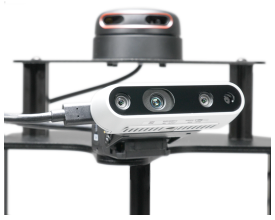


Figure 4.6: View of a possible camera orientation [80].

4.5 Lidar

The sensing capabilities of the LoCoBot WX250s can be optionally completed by a 2D Lidar. In particular, the sensor mounted on the very top of the robot is the RPLIDAR A2M8 - 360 Degree Laser Scanner (Figure 4.7). It exploits a laser triangulation system which, combined with the 8000 samples captured each second, ensures high performance for diverse indoor applications. This system performs high-resolution 2D 360-degree scans within a 12-meter range, generating crucial 2D point cloud data essential for mapping, localization, and detailed object and environmental modeling. This Lidar delivers a typical scanning frequency of 10 Hz (600rpm) and a resolution of 0.9°; the frequency can be adjusted within the 5-15 hz range as per user requirements. It automatically adjusts angular resolution based on the actual rotation speed, ensuring precision in various applications, like obstacle avoidance, autonomous mapping, route planning, and navigation in robotics and autonomous systems.

Figure 4.7: Lidar mounted on the LoCoBot WX250s [80].

The LoCoBot used for this thesis is equipped with the described Lidar. Despite the several advantages, the mounting point of this sensor presents a possible issue: since it is located at a height of about 620 mm, all lower obstacles are not detected. Usually, Lidars are used for localization and mapping, but if prevalent obstacles in the environment are below this height, the generated map will not be precise enough to ensure collision-free navigation.

Chapter 5

Software architecture

The objective of this chapter is to provide an overview of the software developed and tested. The entire structure follows the modularity principle in order to make it easier to develop, modify, debug and expand when necessary. Following this fashion, different modules controlling different parts and tasks of the robot have been realized, according to the high-level structure reported in Figure 5.1.

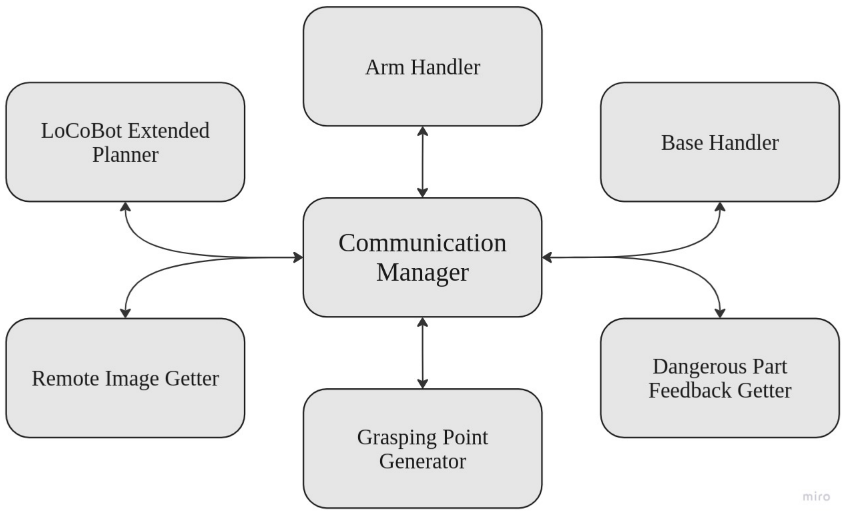


Figure 5.1: Diagram representing the overall software architecture developed.

The topology used for the entire structure is inspired by the one known as **star topology** in the computer network field. It is characterized by a center node, called star center or hub, connected to all other nodes which cannot communicate with another one directly. The center node acts as a message handler and transmitter: it receives all messages coming from the peripheral nodes and, depending on the requested operation, it processes and redirects them to the destination. Usually, two major problems of this type of topology are the low fault tolerance, due to the fact that if the communication manager breaks down the entire communication graph collapses, and the high probability of traffic congestion and latency, because of all messages pass through the center node. In the developed software architecture, all nodes building up the communication graph are ROS nodes, and thanks to their nature and how messages are directed to the communication manager, the two major issues pointed out before can be neglected. Each node spends a great part of the time waiting for messages, and only once a message is received, it processes it and eventually sends back a response; in this way, it is very unlikely to have a node sending a continuous stream of messages to the communication manager. Moreover, messages exchanged among different nodes represent asynchronous requests and replies sent to start a specific action, hence the probability of high congestion on the center node further decreases.

The communication graph is made of a total of 7 nodes:

1. **Communication Manager** constitutes the center node of the communication structure and it is in charge of coordinating messages coming from all nodes. It also has to handle the action pipelines needed to complete the different tasks, coordinating all modules and checking for action success. This node represents the interface between the robot and the human it is assisting;
2. **LoCoBot Extended Planner** is in charge of planning the motion for the entire mobile manipulator considering it as a holistic system. The reason why it is called "extended" is because the robot description has been slightly modified to ease trajectory planning using the MoveIt! framework;
3. The **Arm Handler** node is in charge of requesting the real movement of the robotic arm according to what has been planned by the extended planner, or in order to reach a desired pose without moving the mobile base. It also provides feedback on the actual status and result of the movement. Actions to be done by the gripper's finger are controlled using this node;
4. **Base Handler** is a node that essentially handles all requests regarding base movements: it receives a desired pose, sends it as a goal to the base controller and provides feedback on the status of the operation;
5. The **Remote Image Getter** is a crucial node used to complete the object detection task. Essentially, it is connected to the camera's topic and every time a new frame is available, it is stored so as to provide the last captured pictures only when requested;
6. The **Grasping Point Generator** estimates the point where to position the gripper in order to pick the requested object. It makes use of a trained YOLO network to find the object, and it combines its abilities with data coming from

a depth sensor to complete the assigned task;

7. **Dangerous Part Feedback Getter** comes into action every time the object to grasp is considered dangerous for the human operator. It is responsible of warning the person that the requested item is dangerous and which object's part the robot should consider to safely handle it.

Each node has its own interface through which it can exchange messages with the communication manager or the human operator. Updates and changes are possible also maintaining the designed interfaces untouched; this is the key to modularity, changes affect only the interested part without having to change other modules or how information among them is passed. In the following sections each node is deepened, also discussing how the interface is built.

5.1 Communication Manager node

The role of the communication manager, as briefly explained before, is to act as the coordinator among all nodes used to control the mobile manipulator. It is directly connected to all other nodes in the communication graph in order to have a single node in charge of executing complex actions by dividing them in simpler ones, collecting their results and adapting the robot's behaviour according to them.

5.1.1 Object request and search

The first task solved by the communication manager is to provide an interface through which a person can request a specific item. Using the ROS structure, the easiest way to build such an interface is by using a set of topics. Indeed, as shown in Figure 5.2, the interface used by a person to request a specific item and to get a feedback message about the operation is composed of two different topics.

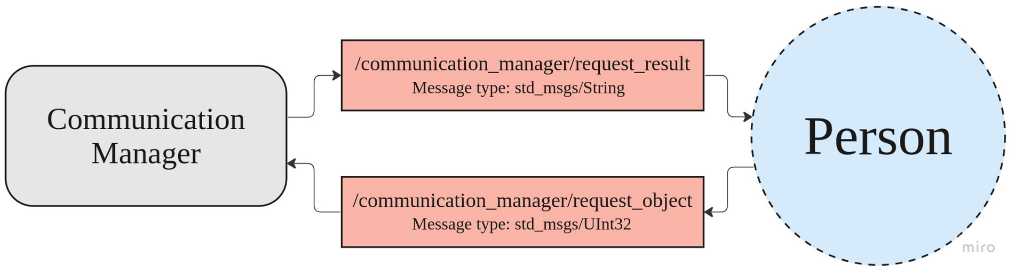

Communication Manager
/communication_manager/request_result
Message type: std_msgs/String
/communication_manager/request_object
Message type: std_msgs/UInt32
Person
miro

Figure 5.2: Interface between the communication manager and the person.

The human can request a specific object by publishing a message on the "*/communication_manager/request_object*" topic. It accepts a std_msgs/UInt32 message, that simply contains an integer number as a unique field. The value of this field is then used by the communication manager to understand which item the person requested and where to find it. Indeed, a one-to-one correspondence exists between each integer number ranging from 1 to 15 and a specific object category. In order to successfully request the desired item, the person must know which is the integer ID corresponding to it and that has to be published on a message. The ID is used by the communication manager in two ways: to understand where the item is possibly stored, and to check if it is present in the designed location. Regarding the positions where it is possible to find a specific object, the communication manager stores a dictionary representing the mapping between locations and objects. Each entry in the dictionary has two fields:

- the **key** of each entry is an integer number corresponding to the ID of the requested item;
- the **value** field is a set of strings containing the name of the locations where an object is possibly stored.

A second dictionary is used to map the locations' names to the corresponding positions and orientations with respect to the world reference frame. In this case, the key value is the string corresponding to the name of the location, and the value field is an array of three float numbers representing the position in terms of (x, y) coordinates, and orientation θ in terms of rotation angle around the z-axis. The use of dictionaries speeds up the process of retrieving data, thanks to the possibility of accessing directly the desired entry searching by value.

The sequence of actions taken to search an object is described in the flow diagram in Figure 5.3. It is important to notice that some actions are not directly executed by the communication manager; they are inserted in a light red box in the flow chart. To better understand how the pipeline is executed, a brief comment of each stage is given:

1. the Communication Manager receives a message on the designed topic and the corresponding callback is executed;
2. first of all the dictionary containing the mapping between the item id and the set of locations is accessed to retrieve the latter;
3. then, the first location's name is extracted and used to access the second dictionary to convert the string into a pose to be reached by the mobile base. The desired goal is sent to the *Base Handler*;
4. once the destination has been reached, the search phase starts. The communication manager moves the camera towards a new position, sends a request to the

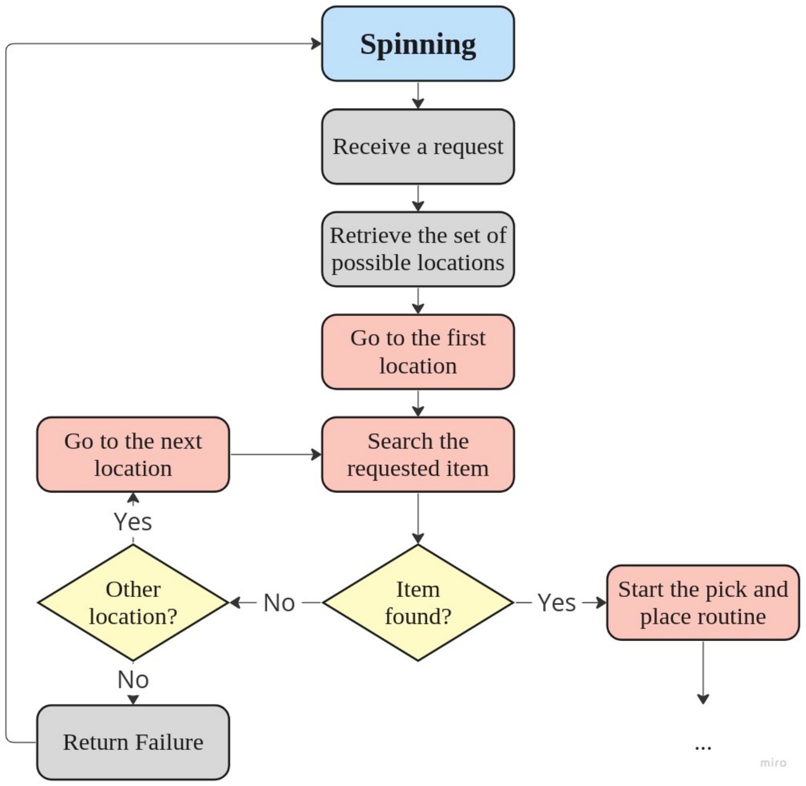


Figure 5.3: Flow chart of the pipeline executed to search an item.

Remote image Getter node to receive the last picture acquired and then sends it to the *Grasping Point Generator* node. It first checks if the object is present and sends the result back to the Communication Manager;

5. if the item is present the pick and place pipeline starts. If another possible location can be reached, the algorithm iterates from point 4, otherwise, a fail message is published on the "*/communication_manager/request_result*" topic.

5.1.2 Camera movements handling

Another job done by the communication manager is to modify the camera orientation. This can be simply done by requesting a specific angle to be reached by the **pan** and **tilt controllers**. Requests are sent by the Communication Manager simply publishing a message on the specific topics; they slightly change if working in simulation or on the real robot. Regardless of their names, these topics accept a std_msgs/Float message, a simple structure containing as a unique field a float number representing the desired angle.

Camera movements are necessary in two cases:

1. while looking for an object in the scene: the robot's field of view is limited, thus by changing it the result of the searching phase may completely change;
2. during the environment navigation. Indeed, visual SLAM algorithms like those used in RTAB-Map require keeping the camera orientation fixed in order to correctly localize the robot. For this reason, it is crucial to restore the camera's pose used while mapping before navigating.

Regarding the movements done during the object search phase, since a continuous motion is not possible due to the time taken by the object detection algorithm, a set of angles ranging from 0.3 to 0.9 rad with a step equal to 0.1 has been defined.

5.1.3 Pre-grasp and grasp poses

After an object has been requested, the search pipeline described before takes place, and the desired item can be found. The node in charge of scanning the image frame and determining whether an object is present or not is the *Generate Grasping Point* one. To complete the task demanded to it, this node provides also an estimation of the 3D pose of the grasping point. As part of the pick and place pipeline, the Communication Manager is also in charge of computing the pre-grasp and grasp poses to be reached by the gripper. Starting from the result given by the grasping generator node, and considering the origin of the gripper's reference frame perfectly centered between the two fingers, the computed poses have the x-axis (red ones) towards the object, the z-axis (blue ones) exit from the upper part of the gripper, and the y-axis (in green) oriented in a way to complete the right-handed system. What really changes is the origin of this reference frame: the pre-grasp pose has the origin 0.3 meters far from the origin of the grasping point along the negative part of its x-axis, while the grasp pose has the origin perfectly coincident with the one of the grasp point. Figure 5.4 shows an example of a pre-grasp pose.

5.1.4 Pick and place pipeline

The main operation carried out by the mobile manipulator requires to pick objects requested by the user and to place them in designed locations. In order to successfully complete this task, a pick-and-place pipeline has been designed, and the Communication Manager node is the one in charge of managing it, since it requires to talk with all other nodes that solve specific parts of the pipeline. The complete sequence of operations is shown in Figure 5.5 and Figure 5.6, where red blocks indicate that the operation involves at least one node different from the Communication Manager. Further details about the pick pipeline are reported below:

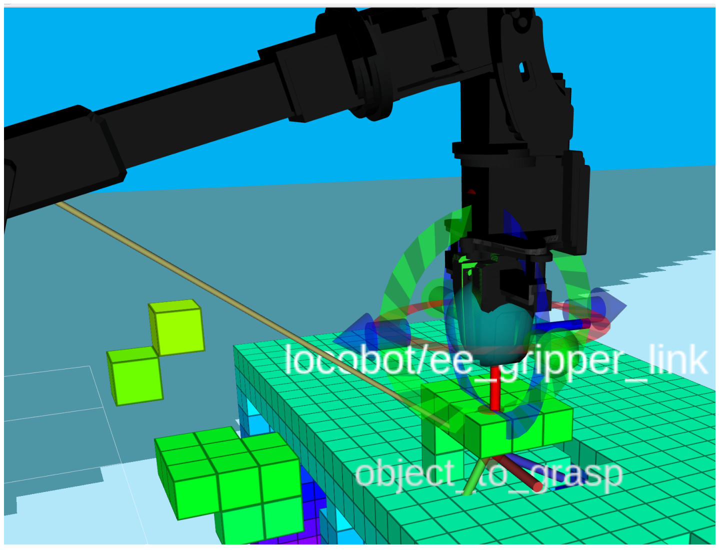


Figure 5.4: Example of pre-grasp pose computed in the simulated environment.

1. the user sends a request containing the ID of the desired object by publishing a message on the /communication_manager/request_object topic;
2. the item's ID is extracted from the received message;
3. the searching phase starts to check if the object is stored in at least one of the known locations. During this step, the Base Handler, the Remote Image Getter and the Grasping Point Generator nodes are involved;
4. if the object has been found, the end-effector's grasp and pre-grasp poses are computed, otherwise an error message is returned;
5. a motion plan for the entire mobile manipulator is requested to the LoCoBot Extended Planner node by publishing a message on the /locobot_extended_-planner/desired_pose topic;
6. after a waiting phase for the results, if a suitable trajectory has been found, the base is moved to its final pose, otherwise an error message is returned;
7. the planned arm trajectory is executed and a collision box around the object position is set;
8. the gripper is opened, then the grasp pose is reached and the gripper is closed. The gripper is controlled directly by publishing a message on the robot's topic designed to control single joints;
9. a box is attached to the gripper and the arm is retracted to the secure position. This latter operation is demanded to the Arm Handler node;

10. the environment is scanned to check if the requested item is still present (pick failure) or not (pick success);
11. on the basis of the previous result, the place pipeline is executed or an error message is returned;
12. the final phase includes return to the home location and wait for the next request.

It is important to notice that before and after grasping the object, some boxes are added to the scene. In the first case, it is necessary to specify to the motion planner that collisions with the object are allowed, and this is done by adding a collision box around the object's location. After having picked the item, a collision box is added to expand the collision area of the robot's gripper: in this way, the motion plan generated to reach the secure pose will take into consideration a larger collision volume, and this is necessary to avoid collision between the object itself and the environment.

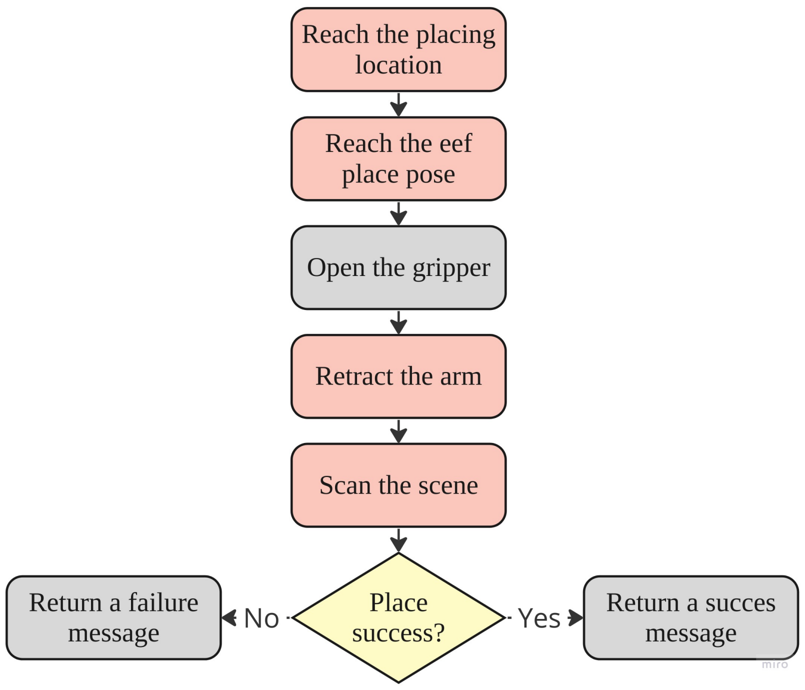


Figure 5.5: Flow chart of the place pipeline

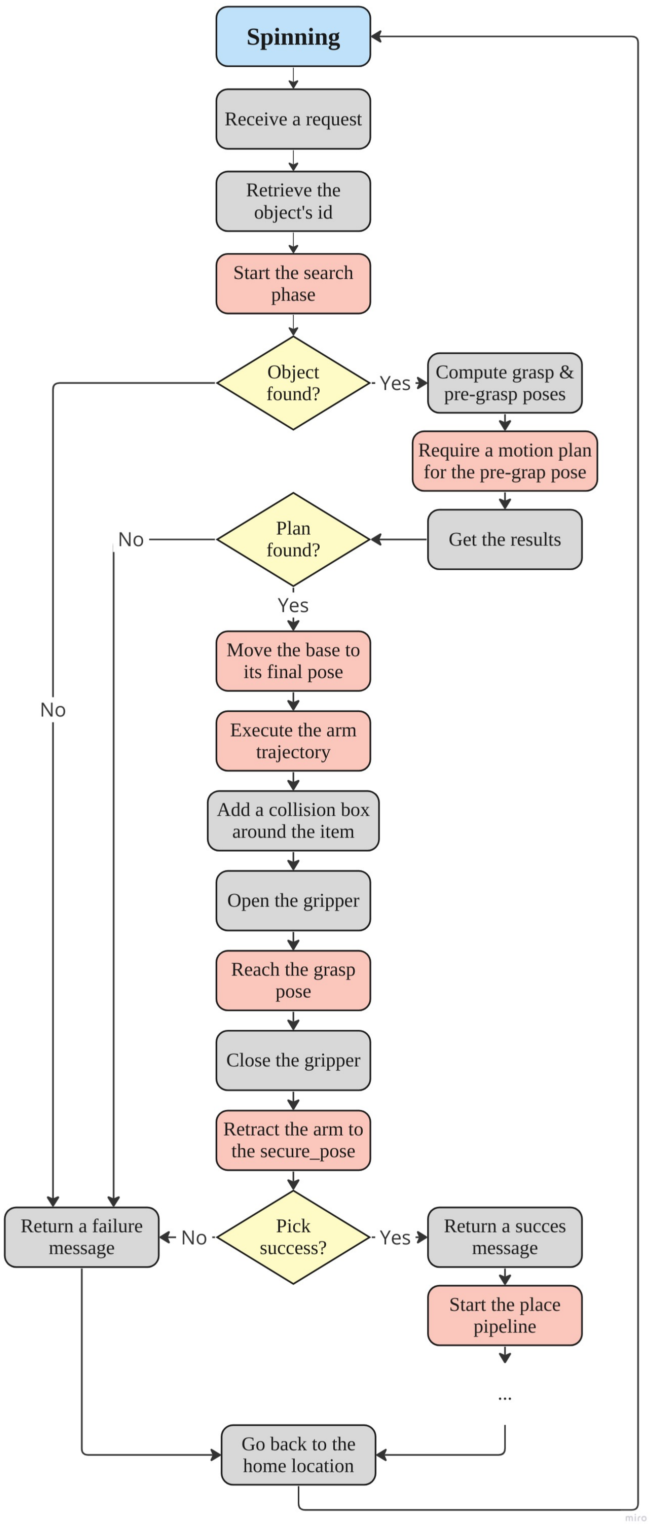


Figure 5.6: Flow chart of the pick pipeline.

5.1.5 Components status handling

In order to properly adapt the behaviour of the robot, the Communication Manager needs to know the current status of all nodes involved in a specific action. This need is satisfied by subscribing to specific nodes' services; in this way, not only the center of the communication graph can retrieve the state of the desired node asynchronously, but it can request it and wait for a response in a control loop or during the execution of a task. Received states are also used to determine if a recovery action is needed before continuing, and also to understand if the entire request succeeded or not. Although the description of the services and of the type of messages exchanged using them are demanded to the following sections, it is important to point out that in the developed software architecture only the Communication Manager can request the access to the current state of a node or component.

5.2 Base Handler node

The node that is in charge of handling all actions that must be done by the mobile base are handled by the *Base Handler* node. Its objectives are to collect all requests coming from the Communication Manager node regarding the movement of the base and to redirect them to the mobile base controller.

The interface through which the Communication Manager node and the Base Handler communicate is sketched in Figure 5.7.

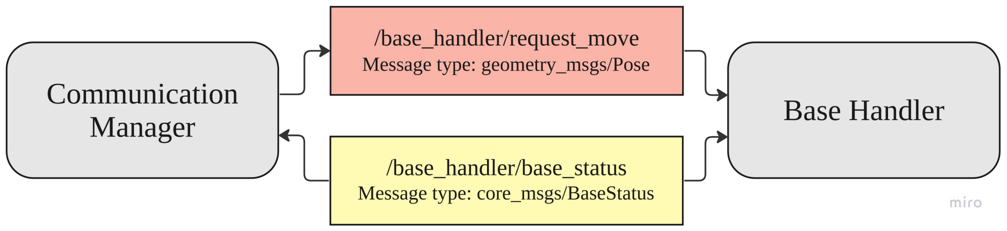

Communication Manager
/base_handler/request_move
Message type: geometry_msgs/Pose
/base_handler/base_status
Message type: core_msgs/BaseStatus
Base Handler
miro

Figure 5.7: Base Handler interface through the Communication Manager

As it is possible to see, this interface is composed of a topic (in red) and a service (in yellow):

- the Communication Manager publishes a message on the **/base_handler/request_move** topic to communicate the desired mobile base pose to be reached. On this topic, *geometry_msgs/Pose* messages can be published. The structure this message wraps contains two fields: *position* field is characterized by three float values corresponding to the x, y and z coordinates, while the *orientation* fields are made of the four floats components of the quaternion representing the desired rotation. It is important to notice that this type of message does not contain a header or any other information where to store the id of the frame with respect to which the desired pose has been defined. In order to solve this problem, as soon as a message is published on the "/base_handler/request_move" topic, the Base Handler executes the corresponding call back where it extracts the position and orientation fields contained in the received message and wraps them onto a *geometry_msgs/PoseStamped* message. The only difference between the type of the received message and the new one is that the latter allows the specification of the ID of the frame with respect to which the pose has been defined. As a matter of standardization, all desired poses published on the */base_handler/request_move* topic are defined with respect to the fixed frame *world* or *map*.

- the Base Handler node informs the Communication Manager about the status of the mobile base by means of a ROS service named **/base_handler/base_status**. The reason behind the use of a service rather than a topic is that in this way it can be used in an asynchronous way: every time the Communication Manager needs to know the state of the message, it can call this service and wait for the response. Service calls can be inserted inside loops if it is necessary to wait for a specific status of the base before continuing.

Regarding the type of responses received using the service described above, it is a custom type named *BaseStatus.srv.* It contains one field named base_status, that is simply a std_msgs/String message with the current status of the mobile base that can be:

1. **MOVE_BASE_NOT_CONNECTED**, which represents the fact the Base Handler node is still waiting for the move_base action server to come up, hence it is not able to handle any request yet;
2. **IDLE**, meaning that the base completed any previous job or it has not received any task yet, thus it is waiting for a request;
3. **PLANNING**, indicating that a request has been received and the corresponding message is being handled before being sent to the base controller;
4. **MOVING**, saying that the desired pose has been received and sent to the controller for the execution;
5. **EXECUTED_SUCCESS** which indicates that the mobile base reached the desired pose;
6. **EXECUTED_FAILURE**, meaning that the request has been executed but the controller failed.

State's name	Meaning
MOVE_BASE_NOT_CONNECTED	Base controller not ready
IDLE	Waiting to receive a request
PLANNING	Request is being handled
MOVING	Request sent to the base controller
EXECUTED_SUCCESS	Goal pose reached
EXECUTE_FAILURE	The controller failed

Table 5.1: Summary table with the mobile base possible states.

The flow of the Base Handler states is depicted in Figure 5.8: blue boxes indicate the states can be assumed by the Base Handler node, grey ones are the events that make the state pass from one to another, and the one in red is the pipeline executed by the move_action group, that includes both the planning and the real execution.

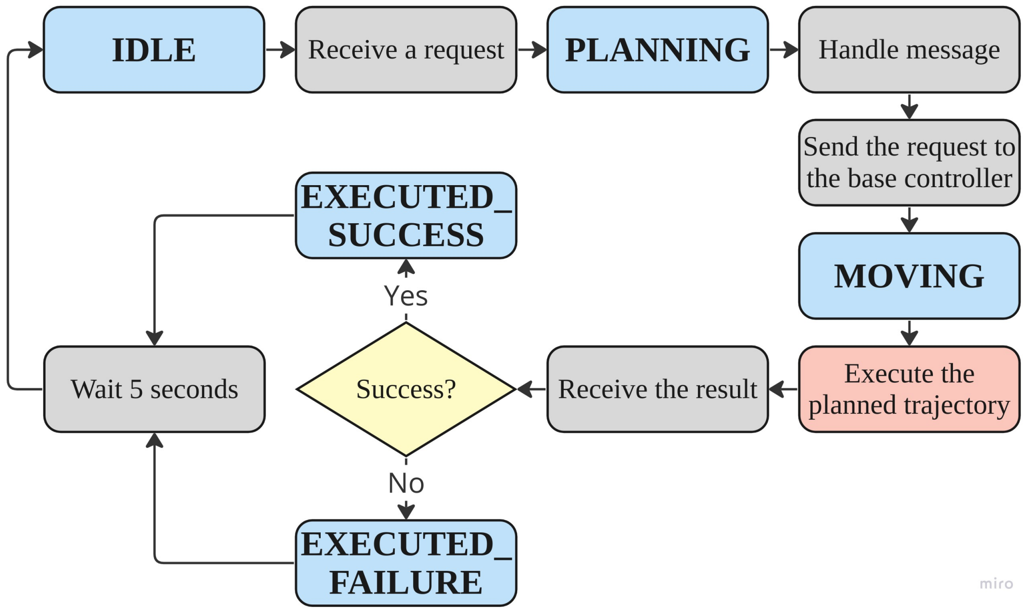


Figure 5.8: Flow of the Base Handler states.

As said before, the actual control of the mobile is demanded by the **move_base** action group. This action group is built using the move_base ROS package, and it allows to use a controller able to deal with the non-holomonic constraints of the differential drive; in this way, it is sufficient to send the goal pose to this controller to obtain a complete path planning and the execution of the corresponding trajectory.

To communicate the desired base pose, the Base Handler publishes the geometry_-msgs/PoseStamped computed as a result of the callback on the */move_base_-simple/goal* topic. The Base Handler interacts with the move_base action group to receive the results of the movement. It subscribes to the "/move_base/result" topic where, once the movement has been completed, a MoveBaseActionResult message is published: it contains all information regarding the last executed plan, including the result of the operation. By subscribing to this topic and handling the messages received through it, the Base Handler node is able to update its status and communicate it to the Communication Manager.

5.3 Arm Handler node

In a similar way to what has been done to deal with requests concerning the base of the mobile manipulator, all those tasks that must be completed by the robotic arm are handled by the Arm Handler node. Its role is to collect all requests coming from the Communication Manager and to request a motion planning built for the robotic arm.

The interface thanks to which this node is able to communicate with the center node of the communication graph is shown in Figure 5.9.

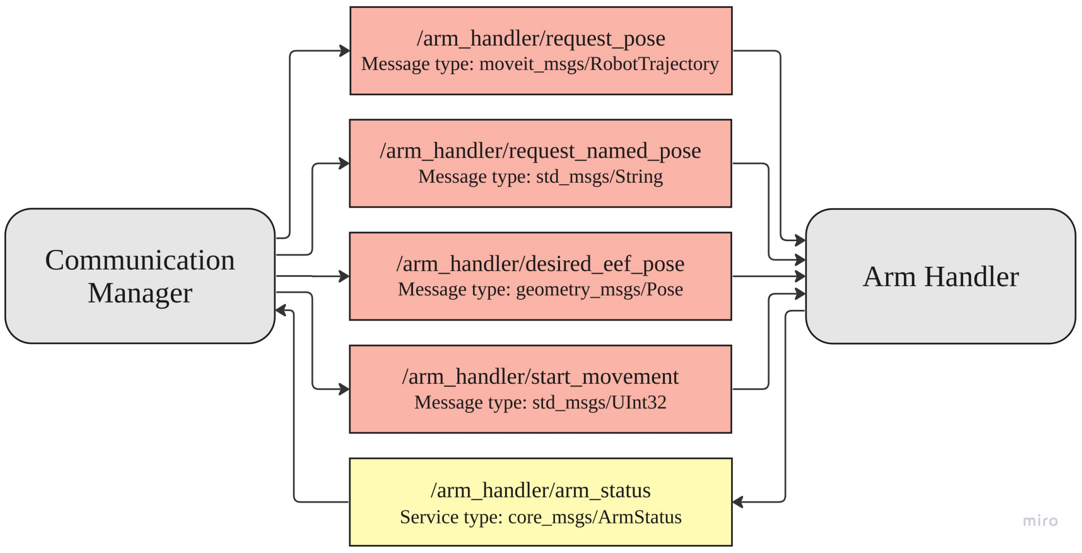

/arm_handler/request_pose
Message type: moveit_msgs/RobotTrajectory
/arm_handler/request_named_pose
Message type: std_msgs/String
/arm_handler/desired_eef_pose
Message type: geometry_msgs/Pose
/arm_handler/start_movement
Message type: std_msgs/UInt32
/arm_handler/arm_status
Service type: core_msgs/ArmStatus
Communication Manager
Arm Handler
miro

Figure 5.9: Representation of the Arm Handler interface

The complete interface is made of a total of four topics and one service; three of the topics can be used to reach a desired end effector pose but in different ways. Further details are given below:

1. the *"/arm_handler/request_pose"* topic allows to reach a desired pose passing to the Arm Handler node a joint trajectory. The messages exchanged on this topic are of type moveit_it/RobotTrajectory: it is a quite complex message containing several fields needed to encode a joint trajectory. Messages of this type are generated by MoveIt! framework as a result of a planning request;
2. the *"/arm_handler/request_named_pose"* topic is used to request the motion plan needed to lead the arm in a configuration that has been defined by the developer and that is denoted by its name. The MoveIt! ROS package gives the possibility to associate a name to a given joint configuration, and this name can be directly used to request a motion plan. Keeping this in mind, it is sufficient to communicate the name of a predefined arm configuration to get a plan as a response, and this is why messages exchanged through this topic are simple strings wrapped into the std_msgs/String type.
3. the *"/arm_handler/desired_eef_pose"* permits to obtain a trajectory describing the time evolution of the arm's joints needed to reach a desired pose for the LoCoBot's gripper. MoveIt! gives the possibility to define motion groups, that are sets of joints for which a planning can be requested, and to express if an end-effector is attached to one of them and through which link the connection is made. Having defined the correct connection, it is possible to specify a desired pose for the end-effector and obtain the corresponding plan. To request the motion plan it is sufficient to publish a geometry_msgs/Pose message on this topic in order to express the desired position and orientation of the end-effector link.
4. *"/arm_handler/start_movement"* is the topic used to start the real movement: the three topics discussed above are used only to request the path planning, but they do not start the real arm motion control. The Communication Manager publishes this topic every time a successful path-planning tasks is completed.
5. service *"/arm_handler/arm_status"* allows to retrieve the current status of the robotic arm. The asynchronous mechanism given by this ROS service can be exploited by the Communication Manager within the grasping pipeline: every time a task regarding the manipulator is needed, a request is sent to the Arm Handler node, and before continuing the current state of the arm is requested and checked in order to understand which type of action is needed.

5.3.1 MoveIt! settings and parameters

Planning tasks are completed by MoveIt! package. A custom version of this package has been created in order to have a complete control and the ability to customize any settings. Regarding the Arm Handler node, Table 5.2 summarizes which are the main configuration parameters that have been set.

Parameter name	Value
joint_states topic	/locobot/joint_states
Robot description paramater	/locobot/robot_description
Planning library	OMPL
Planner used	RRT*
Maximum planning time	30s
Maximum planning attempts	5
Goal tolerance	0.01 rad

Table 5.2: Summary table of the main MoveIt! parameters set by the Arm Handler Node.

A brief description of the parameters follows:

- **/locobot/joint_states** is the name of the topic that publishes all the updates about the current state of the joints;
- **/locobot/robot_description** is the name of the parameter on the parameter server that contains the description of the robot model. Important to notice the presence of the namespace *"locobot"* in these first two parameters;
- the Arm Handler node uses the **RRT*** planner from the **OMPL** to solve all the planning tasks it is in charge of;
- to limit the time required by the planner to find a solution, the **maximum planning time** available has been set to 30 seconds. This amount of time results to be reasonable to find a path for the manipulator all the times the desired arm configuration or the end-effector pose is reachable;
- the **maximum number of planning attempts** was fixed to 5: in this way, the planner will try to find a suitable path five times before returning a failure result;
- in order to deal with the tolerance of the servo actuating the arm's joints, the **goal tolerance** has been set to 0.01 rad.

All these parameters and the planning requests are sent to the "*move_group*" action server using the MoveGroup Python Interface available. In this way, it is possible to programmatically control the requests and actions demanded to the Arm Handler node, and it is also possible to change the parameters on the fly without the need to recompile and rebuild the entire workspace (as would have happened if C++ had been used).

5.3.2 Arm group and named poses

The Arm Handler node controls the robotic arm thanks to an action group named "locobot_arm". This action group comprehends the entire kinematic chain characterizing the arm, that contains:

- the *waist*, *shoulder* and *elbow* joints and the links connecting them, making the "arm" of the manipulator;
- the *forearm_roll*, *wrist_angle* and *wrist_rotate* joints that build up the wrist of the manipulator.

The LoCoBot's gripper has been attached to this kinematic chain in order to request the motion plan for a desired pose of the end-effector.

Regarding the named arm poses, three configurations have been defined:

1. the **arm_default** pose is used while the robot is not moving, hence while it is waiting for a request by the user. This pose can be seen in Figures 5.10 and 5.11, while in Table 5.3 joints value characterizing this configuration are reported;
2. while the mobile manipulator is moving, the arm assumes the pose shown in Figures 5.12 and 5.13b named **arm_secure_pose**; joints values defining it are listed in Table 5.4. This configuration is not used only while navigating, but also every time an object is grasped and the robot has to go from one location to another. Two main reasons lead to define this pose: the first reason is that while navigating the arm must not be outside the shape of the mobile base, otherwise accidental collisions may happen while navigating, and the second is the need of reducing the possibility of hurting people while moving with an object in the gripper;
3. **arm_sleep_pose** (Figures 5.14 and 5.15, Table 5.5) is a configuration used to reduce accidental damage when turning off the arm controller: as soon as the controllers are stopped, the arm falls down uncontrolled. To overcome this problem, before stopping the Communication Manager node, the arm assumes this configuration.

Joint name	Value	Joint name	Value
waist	0.0 rad	forearm_roll	0.0 rad
shoulder	-0.8299 rad	wrist_angle	0.0 rad
elbow	1.456 rad	wrist_rotate	0.0 rad

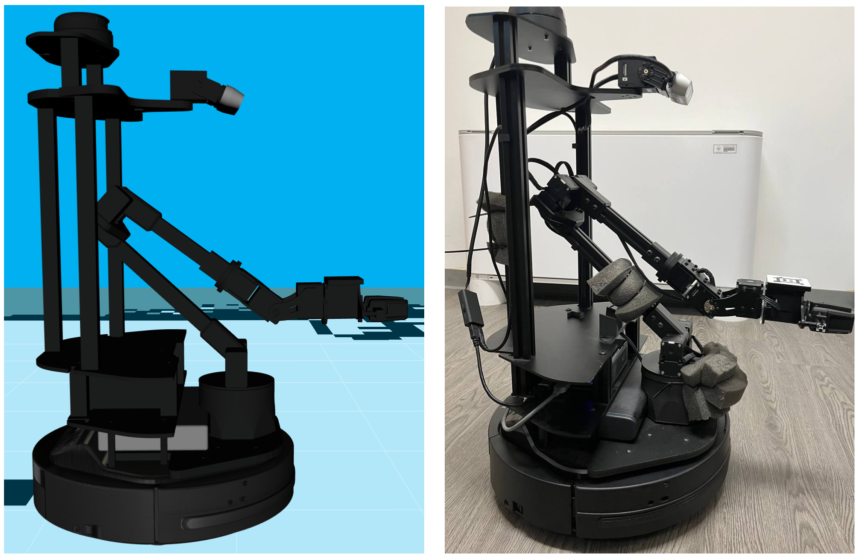
Table 5.3: Joints values defining the arm_default pose.

Joint name	Value	Joint name	Value
waist	-0.955 rad	forearm_roll	-1.7189 rad
shoulder	-0.8452 rad	wrist_angle	1.4695 rad
elbow	1.354 rad	wrist_rotate	0.4676 rad

Table 5.4: Joints values defining the arm_secure pose.

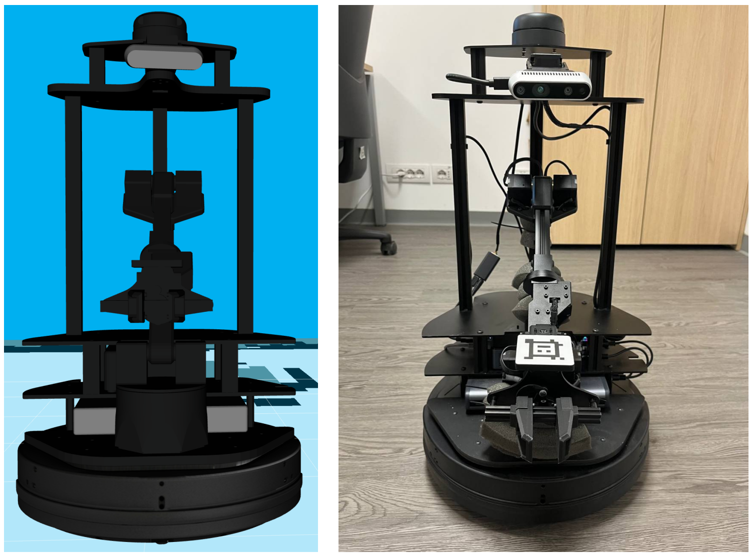
Joint name	Value	Joint name	Value
waist	0.0 rad	forearm_roll	0.0 rad
shoulder	-1.1294 rad	wrist_angle	0.4504 rad
elbow	1.5787 rad	wrist_rotate	0.0 rad

Table 5.5: Joints values defining the arm_sleep pose.



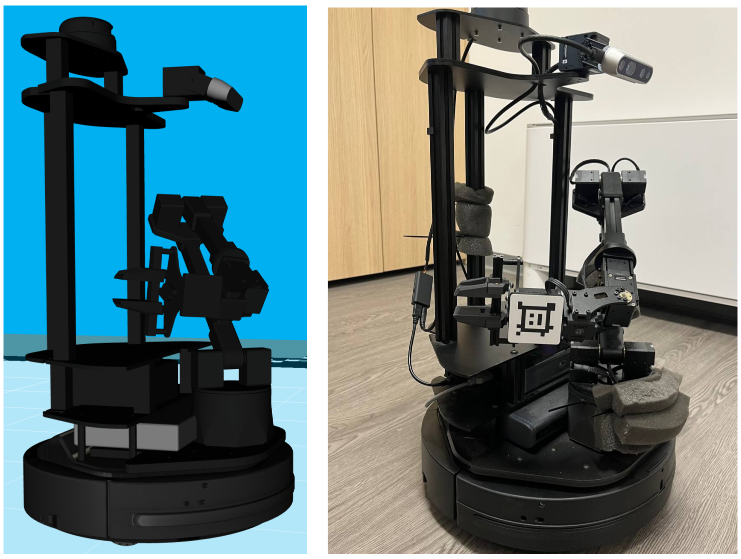
(a) **(b)**

Figure 5.10: Side view of the arm_default pose (in Rviz 5.10a and real 5.10b).



(a) (b)

Figure 5.11: Front view of the arm_default pose (in Rviz 5.11a and real 5.11b).



(a) (b)

Figure 5.12: Side view of the arm_default pose (in Rviz 5.12a and real 5.12b).

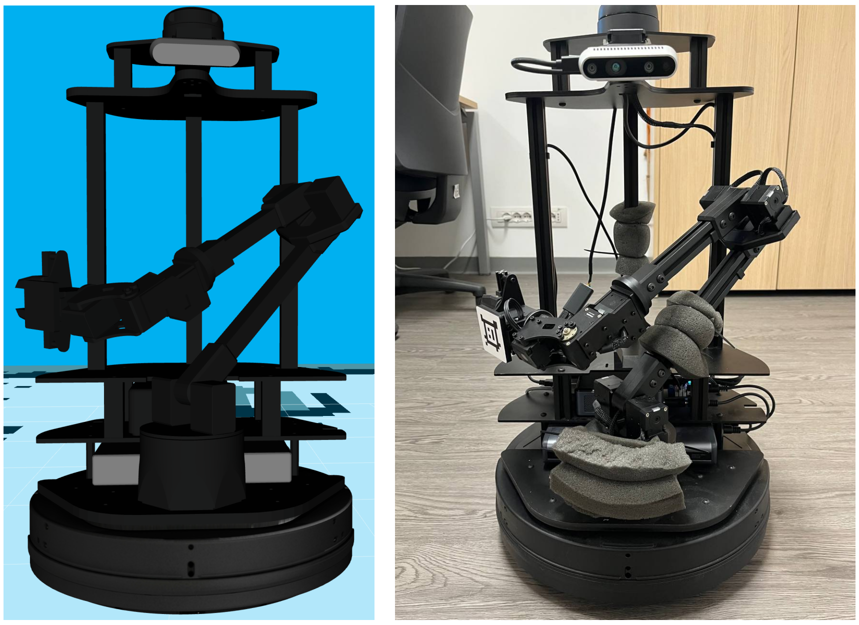


Figure 5.13: Front view of the arm_secure pose (in Rviz 5.13a and real 5.13b).

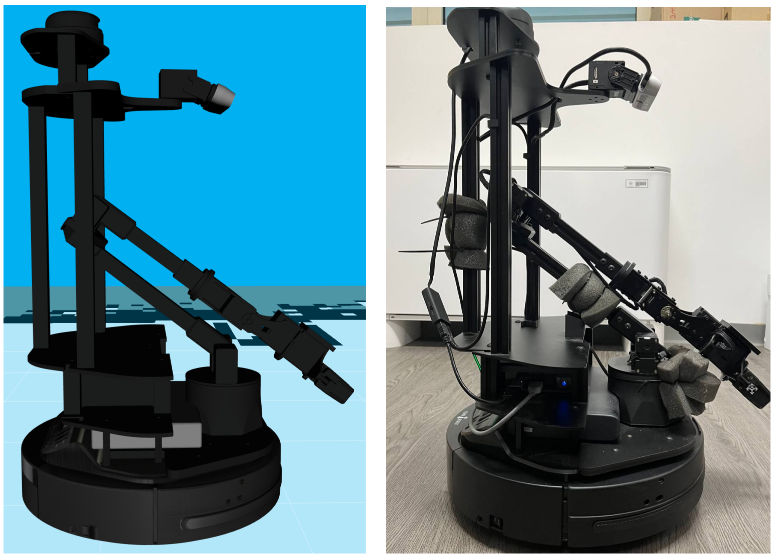


Figure 5.14: Side view of the arm_sleep pose (in Rviz 5.14a and real 5.14b).

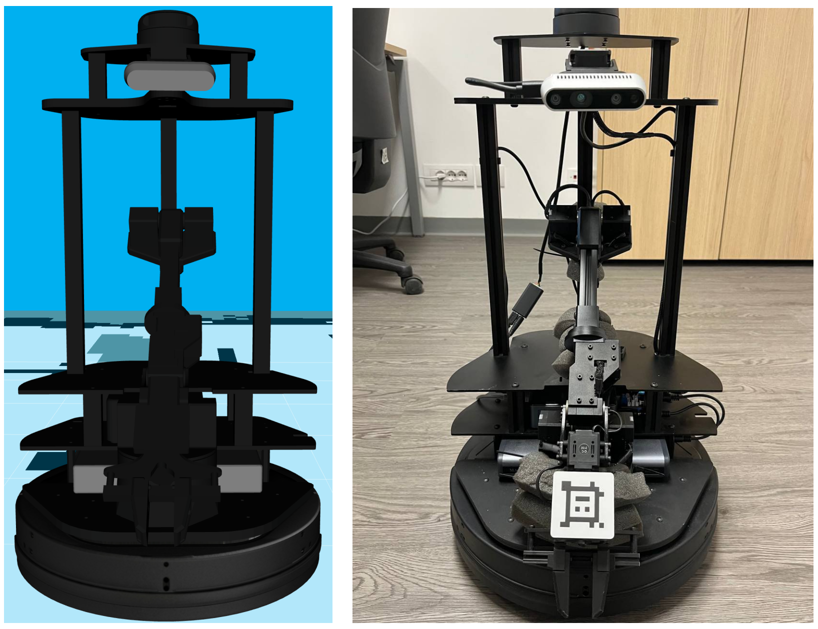

(a) (b)

Figure 5.15: Front view of the arm_sleep pose (in Rviz 5.15a and real 5.15b).

5.3.3 Arm states

In order to properly communicate the current status of the arm and of the requested motion, several states have been defined and can be assumed by the Arm Handler node:

- the **IDLE** status means that all requests have been completed or no requests have been submitted yet;
- **SETTING_TARGET_POSE** indicates that a planning task has been published on one of the three topics available in the interface, and that it is being handled;
- the **WAITING_TO_MOVE** status highlights that a collision-free trajectory has been found and the arm controller is waiting to execute it;
- **MOVING** states is set right after a message is received on the /arm_-handler/start_movement topic, meaning that the arm is currently moving;
- **EXECUTED_SUCCESS** says that the trajectory execution finished successfully, while **EXECUTED_FAILURE** means that the trajectory execution failed for some reason.

To better understand to which request the last two states refer to, a string is added to the last two states. The flow chart representing how the Arm Handler states evolve is reported in Figure 5.16

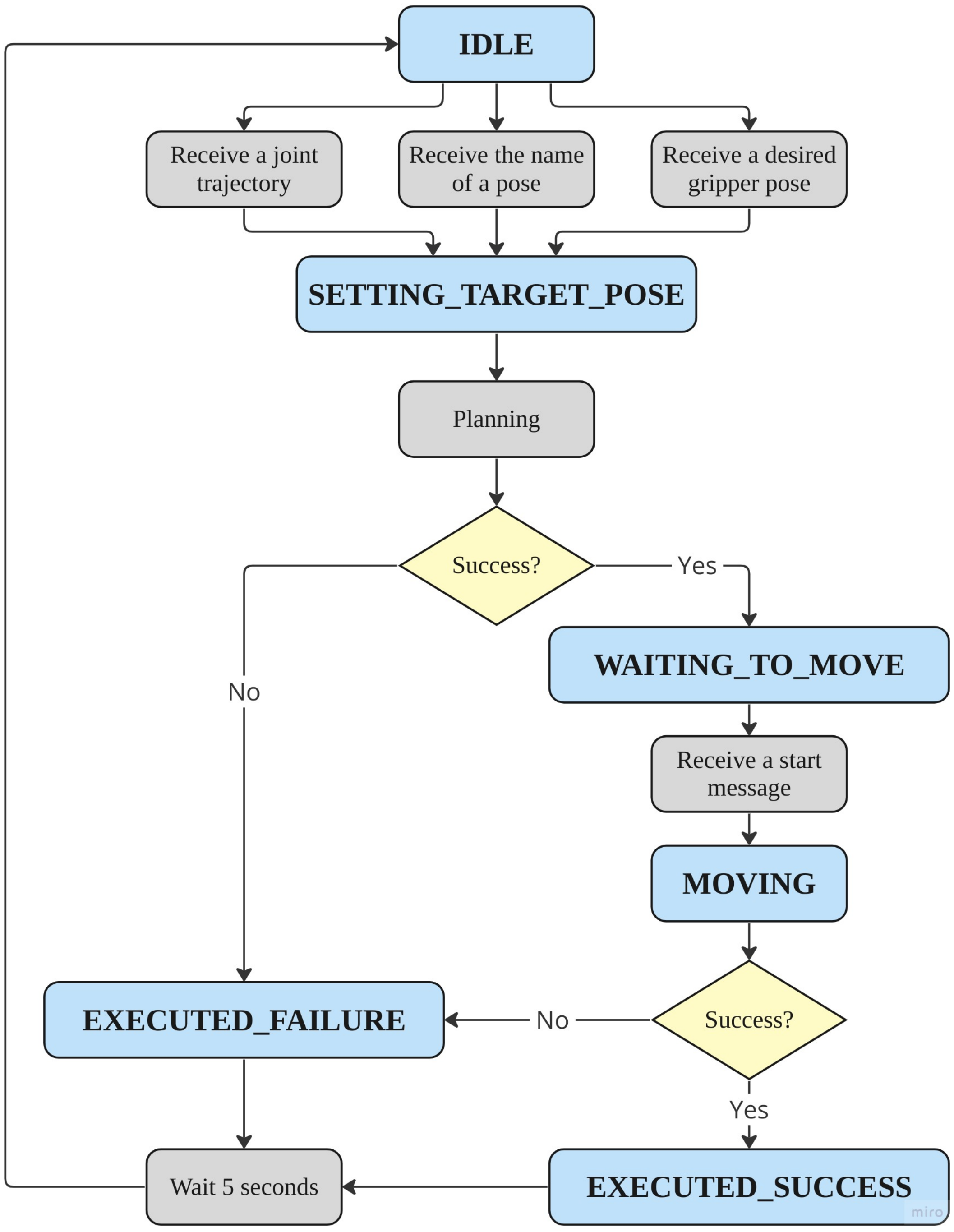


Figure 5.16: Evolution of the Arm Handler states.

5.3.4 Arm status service

The only communication channel going from the Arm Handler towards the Communication Manager is based on the /arm_handler/arm_status service; all other messages flow in the opposite direction. These service responses are simple String messages contained inside a custom service type, named **ArmStatus.srv**, from the **core package**; their only field is a string reporting the value of the current Arm Handler status.

5.4 Remote Image Getter node

The pick-and-place pipeline requires to understand if a certain object is present in the scene and where it is located. Both these tasks need to use the images coming from the LoCoBot's camera. In order to satisfy this need, the Remote Image Getter node has been designed.

The role of this node is to collect all image bundles coming from the RGB-D camera mounted on top of the mobile manipulator and to forward them to the requesting node. The interface thanks to which this node is able to communicate with the Communication Manager is depicted in Figure 5.17. It is possible to notice that the interface is made of only two services:

- */remote_image_getter/remote_image_srv* is used to retrieve the last RGB-D image bundles received;
- */remote_image_getter/get_camera_info_srv* is called every time the information about the intrinsic parameters of the LoCoBot's camera is needed.

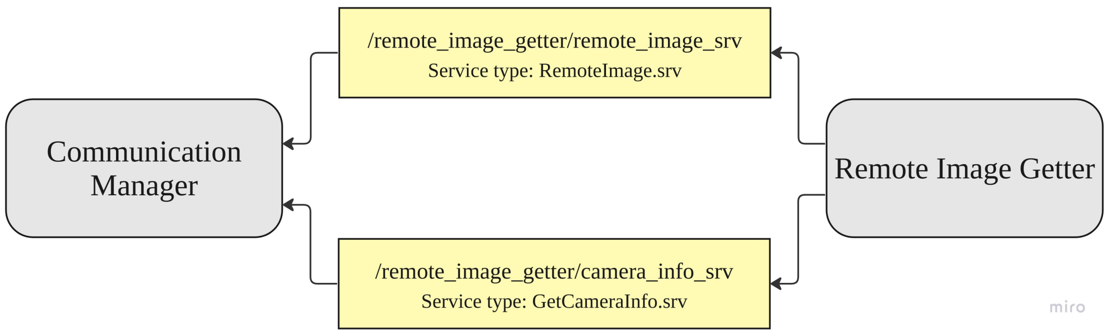

/remote_image_getter/remote_image_srv
Service type: RemoteImage.srv
Communication Manager
Remote Image Getter
/remote_image_getter/camera_info_srv
Service type: GetCameraInfo.srv
miro

Figure 5.17: Remote Image Getter node's interface.

This node simply acts as a forwarder for those nodes requiring any data regarding the camera. There is another way to obtain the same data directly from the camera, but it requires to subscribe to the camera topics. The reason why this strategy should be avoided is that the RGB-D sensor continuously publishes on those topics: any node subscribed to them would execute the callback repeatedly, without the possibility of doing any other operation. This is also the reason why the Remote Image Getter node has been designed: its only purpose is to subscribe to the topics on which the camera publishes the data and to store them by repeatedly executing the callbacks.

5.4.1 Implementation details

The topics from which the node takes all data needed are:

- "*/locobot/camera/color/image_raw*" topic for the RGB images;
- "*/locobot/camera/aligned_depth_color/image_raw*" topic where depth images are store.

Both the images have a resolution of 640x480 pixels, and streams are updated with a frequency of 30 frames per second (fps). In addition, depth image errors are corrected using a set of spatial filters and a hole filler that fills gaps in depth data (caused by sensor limitaions), enhancing the overall accuracy and completeness of the depth map. Table 5.6 summarizes the main parameters used by the Remote Image Getter node and those regarding the images acquired.

For what regards how the Remote Image Getter node handles image bundles, where an image bundle is a pair of RGB images and the corresponding depth image, two queues are used: one stores all color images, while the second store all the depth images. These data structures have a fixed length equal to 10 items, hence, also because of the nature and implementation of a queue, only the last 10 image couples are stored. Whenever a node sends a request to receive the last acquired frame, the last image bundle inserted into the queue is returned.

Parameter	Value
Width	640 px
Height	480 px
Refresh rate	30 fps
Hole filling algorithm used	Weighted Least Squares filtering
Data structure used	Fixed length queues
Queue length	10

Table 5.6: Summary table of the main camera parameters and those used by the Remote Image Getter Node.

5.4.2 Remote Image and Camera Info service

As previously discussed, the only communication channel thanks to which nodes can retrieve RGB-D images from the camera is through a service built by the Remote Image Getter and named "*/remote_image_getter/remote_image_srv*". The responses sent using this topic are of a custom type called **RemoteImage.srv**: it wraps together two fields of type sensor_msgs/Image, one for the last RGB image captured and the other for the depth one. In this way, every time a service request is sent, the Remote Image Getter node takes the last bundle acquired by the camera, inserts the two images in the correct field and then sends the response back.

In a similar way, all the camera's intrinsic information needed can be retrieved through the "*/remote_image_getter/camera_info_srv*". Every time a request is received, a response of type **GetCameraInfo.srv** is sent back. This service response type is a custom one, and it contains a sensor_msgs/CameraInfo field that is set equal to the message received from the /camera/camera_info topic of the Realsense Camera mounted on the LoCoBot.

5.4.3 Exception to the communication graph structure

Section 5.1 describes how the Communication Manager node is the only one in charge of talking with different nodes, and that two nodes cannot communicate with each other without passing through the Communication Manager. Actually, an exception to this rule holds: in order to reduce the communication overhead, and hence the time required to get the needed information, the Grasping Generator node and the Remote Image Getter one are directly connected. The grasp generator can request image bundles without asking the Communication Manager. Although the time required to publish messages on a topic is very short, the number of messages required to retrieve an image pair exploiting the manager node would increase the overall time unnecessarily, and for this reason, a direct link between the two nodes is used.

5.5 Grasping Point Generator node

The Grasping Point Generator node is used to scan the environment and understand where to grasp a given object form. This node is in charge of:

- scanning the environment using the RGB-D camera to check whether a specific object is present or not;
- estimating the pose of an item;
- generating a grasping point with respect to the world reference, frame from which to position the gripper and execute the grasp task.

To complete all these tasks, the Grasping Point Generator node uses a CNN trained on a custom dataset in order to recognize common objects that can be found in a laboratory or in a warehouse.

5.5.1 Node implementation

This node receives requests from the Communication Manager and provides responses to it thanks to the interface shown in Figure 5.18.

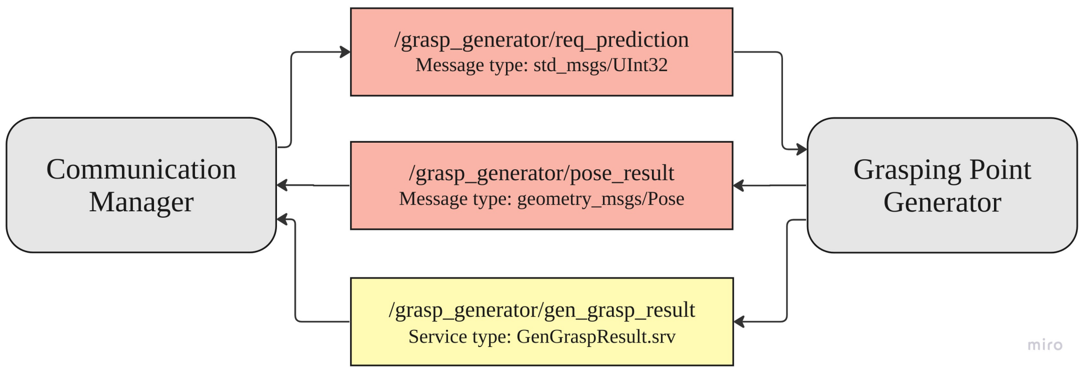

/grasp_generator/req_prediction
Message type: std_msgs/UInt32
Communication Manager
/grasp_generator/pose_result
Message type: geometry_msgs/Pose
Grasping Point Generator
/grasp_generator/gen_grasp_result
Service type: GenGraspResult.srv
miro

Figure 5.18: Grasping Point Generator interface.

It is possible to notice that there are three ways to establish communication to and from this node:

- */grasp_generator/req_prediction* topic allows to search for a specific object. Messages published on this topic are of type std_msgs/UInt32, whose only field is an integer number representing the ID of the desired object;
- */grasp_generator/pose_result* is the second topic of the interface. If the desired object is found in the current scene, the pose of the grasping point is published here; messages characterizing this topic are of type geometry_msgs/Pose;

- service */grasp_generator/gen_grasp_result* is used to get the results of the last task.

Pose messages published on the /grasp_generator/pose_result topic do not contain any information regarding the reference system against which the pose is defined, hence a convention is used: all poses published on this topic by the Grasping Point Generator are defined with respect to the /locobot/camera_depth_link frame. As shown in Figure 5.19, a reference frame with the origin fixed in the grasping point is computed just as a translation of the /locobot/depth_camera_link frame along its z-axis; x-axes are shown in red, y-axes are green and z-axes are blue. As a result, the orientation of the axes of the *object_to_grasp* reference frame with respect to the world reference frame is the same as the one of the depth camera.

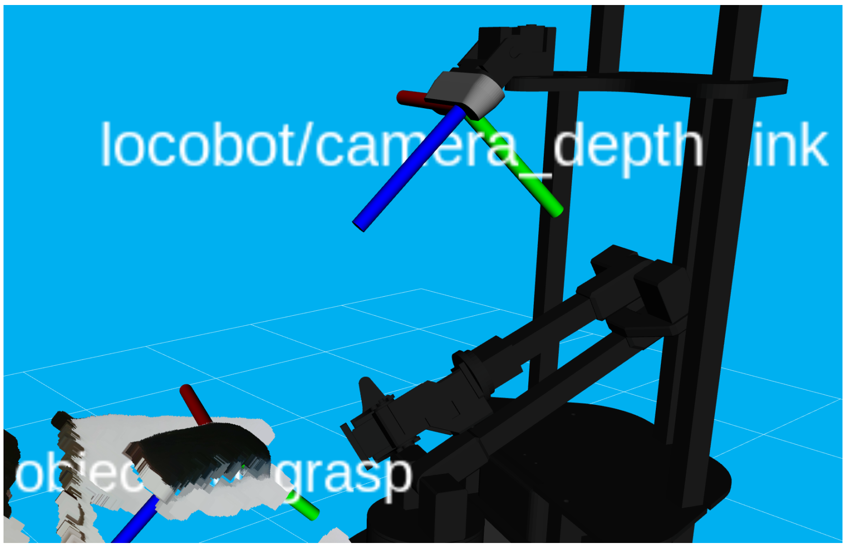


Figure 5.19: Visualization of the grasp pose orientation with respect to the /locobot/camera_depth_link frame (Rviz).

5.5.2 Object detection

In order to succeed in the generation of a suitable grasping point, this node is also in charge of scanning the environment and completing the object detection task. To face this challenge, the Grasping Point Generator node makes use of a YOLOv5 CNN trained on a custom dataset, containing objects that can be easily found in a laboratory. The trained network is then used to check if an item is present by considering the class labels obtained, and to compute the correct bounding boxes (BBs) of the object; they indicate the location of an item with respect to the image frame.

Class predictions are computed together with a confidence that estimates how accurate the prediction is. All predictions coming from the YOLO network are sorted by their confidences following a descending order, and to deal with erroneous predictions, a confidence threshold equal to 80% has been set: in this way all predictions with a confidence lower than this value are neglected, and only the best ones are effectively used.

5.5.3 Grasping point generation

Information about the BB of an object is used to compute an estimation of its center of mass, that is assumed to be equal to the center of the bounding boxes; information about the real mass distribution is neglected because it would have required to code them for any specific object or to train another network to solve this challenge. The center of mass is considered to be the origin of the grasp pose.

Using only the bounding box prediction it is possible to compute the grasping point in a 2D reference frame that corresponds to the image frame, so it is necessary to integrate this data with others coming from the camera to obtain its coordinates in the 3D world. To do that, depth image coming from the camera is exploited. This kind of data can be easily transformed into a 2D matrix using the intrinsic parameters of the camera. To complete the grasping point generation, the node in charge executes the following steps (Algorithm 1):

1. retrieve the last image bundle coming from the camera by requesting it to the Remote Image Getter;
2. use the RGB image as input for the CNN to perform the object detection task and obtain the BB associated to an object (if present);
3. compute the center of the BB.
4. request the intrinsic parameters of the camera to the Remote Image Getter Node and transform the depth image in a 2D matrix;
5. retrieve the depth value corresponding to the computed center of mass.

Knowing the position of the center of mass with respect to the camera 2D reference system and the corresponding depth value, it is possible to compute the origin position of object's reference system, that is then published to the topic in charge of computing the transformations between the different frames (*/tf*) indicating the /locobot/camera_depth_link as parent frame.

Algorithm 1 Algorithm executed to generate the grasping point.

```
 1: Retrieve the last image bundle from the Remote Image Getter
 2: Retrieve intrinsic camera's parameters
 3: Convert the depth image in a 2D matrix
 4: Compute the predictions
 5: Create an empty list for the predictions
 6: for $pred$ in $all\_preds$ do
 7:     Print info about the prediction
 8:     Append $pred$ to the list
 9: end for
10: $found = FALSE$
11: while $not\ FOUND$ do
12:     Extract one item from the list
13:     if $confidence \geq 0.8$ then
14:         Save the element
15:         $found = TRUE$
16:     end if
17: end while
18: Retrieve depth value
19: Compute the 3D coordinates
20: Publish the 3D coordinates on $/tf$ topic
```

5.6 LoCoBot Extended Planner node

This node is in charge of generating a suitable path to allow the end-effector to reach the desired grasping point considering the mobile manipulator as a holistic system. The main advantage of considering this type of system instead of planning for the mobile base and the manipulator separately is that it is possible to reposition the base, and consequently the robotic arm, in order to reach the best pose for the entire system.

In order to obtain such a motion planning, it is necessary to extend somehow the DOFs of the robotic manipulator to inform the planner that it is able to move in a 2D reference frame. Indeed, the three DOFs added allow the manipulator to translate along the x and y axes and to rotate about the z-axis. As a result, the planner is able to generate a trajectory starting from a desired end-effector pose that can be also outside the workspace of the manipulator, and that involves the movement of the base.

The planning abilities of the LoCoBot Extended Planner node are provided by configuring MoveIt! appropriately. Although the planning framework gives the possibility to add a **planar virtual joint** to the configuration, this type of joint is not considered while planning; virtual joints are converted to fixed joints before planning. Adding a virtual joint only would result in planning abilities that would consider the arm as fixed in the current position. To overcome this problem, the URDF has been modified by adding three joints at the base of the mobile manipulator:

1. a prismatic joint that extends along the x-axis of the mobile base reference frame;
2. a second prismatic joint attached at the end of the first that extends along the y-axis;
3. a revolute joint able to rotate around the z-axis.

These three joints together give the possibility to request a planning for the holistic system using the planning abilities of MoveIt! framework. Table 5.7 reports the joint limi of these three virtual joints added.

Joint name	**Type**	**Limit type**	**Value**
move_all_x	Prismatic	Translation	[-10.0;10.0] [m]
		Velocity	1 $[m/s]$
		Effort	5 [N]
move_all_y	Prismatic	Translation	[-10.0;10.0] [m]
		Velocity	1 $[m/s]$
		Effort	5 [N]
move_all_theta	Revolute	Rotation	$[-\pi;\pi]$ [rad]
		Velocity	π $[rad/s]$
		Effort	5 [Nm]

Table 5.7: Summary table with the limits of the virtual joints.

5.6.1 Node's interface

The interface through which the Communication Manager communicates with the LoCoBot Extended Planner node is shown in Figure 5.20. It is possible to see that it is composed of one topic and one service:

1. */locobot_extended_planner/desired_pose* is the topic through which it is possible to communicate the desired pose to be reached by the end-effector and to obtain a trajectory as a result. Indeed, messages exchanged through this topic are of type geometry_msg/Pose;

2. the */locobot_extended_planner/planner_status* service is used to retrieve both the node status and the planning results.

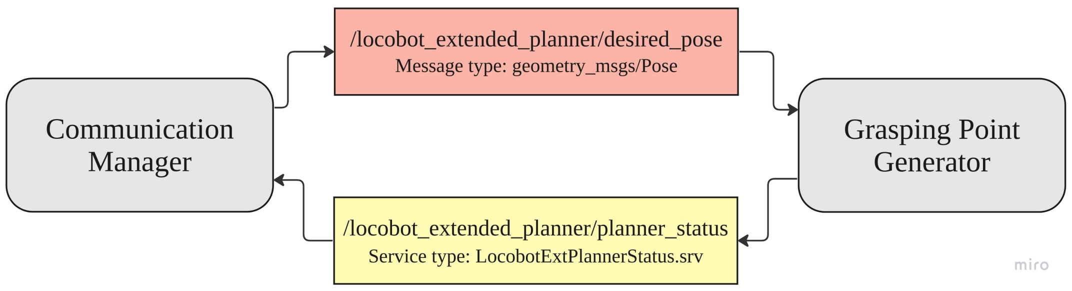

/locobot_extended_planner/desired_pose
Message type: geometry_msgs/Pose
Communication Manager
Grasping Point Generator
/locobot_extended_planner/planner_status
Service type: LocobotExtPlannerStatus.srv
miro

Figure 5.20: Communication interface of the LoCoBot Extended Planner node.

The responses obtained through the service are of type LocobotExtPlannerStatus.srv, that is a custom service type defined inside the core package. It contains three fields:

1. the **planner_status** field is a message of type std_msgs/String containing the current state of the planner;
2. **base_final_pose** field is a geometry_msgs/PoseStamped message indicating the final pose to be reached by the mobile base;
3. the **arm_trajectory** field is a message of type moveit_msgs/RobotTrajectory. It is a quite complex message containing the complete joint trajectory to be executed by the manipulator in order to reach the grasping pose, and it specifies all the values at each time instant for the joint positions, velocities and accelerations, besides the time instants themselves.

It is important to notice how the result of the motion planning is split in two separate parts, one for the mobile base and the other for the robotic arm. This is due to the need of assigning these targets to two different controllers, one for the mobile base and the other for the arm, since a unique controller able to govern both parts has not been implemented.

5.6.2 Node's states

Similarly to the other nodes, the LoCoBot Extended Planner node has a set of states that provide information about the current operation it is completing, and they are:

1. **IDLE**, that indicates the node is waiting to receive a request or that all the others requests have been completed;

2. **PLANNING** indicating that a desired pose has been received and the planning process is actually running;
3. **PLANNING_SUCCESS**, indicating that a plan for both the arm and the mobile base has been found;
4. **PLANNING_FAILURE** denoting that the planner failed while searching for a feasible trajectory.

5.21 describes how the states of the node change according to the different events.

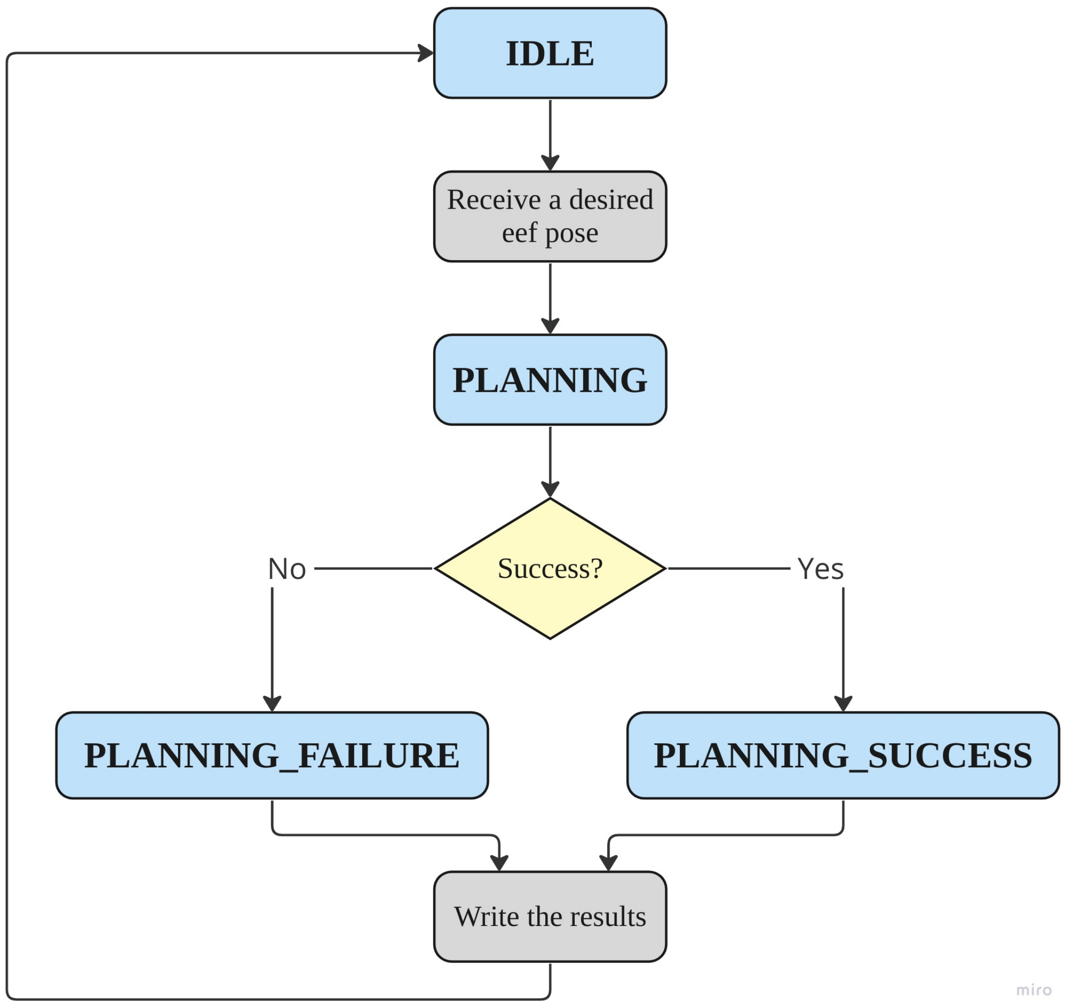


Figure 5.21: Flow chart describing the evolution of the LoCoBot Extended Planner states.

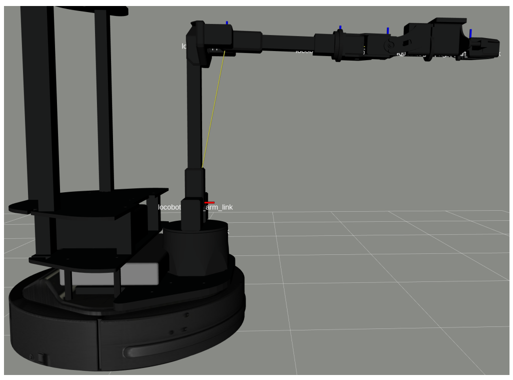
5.6.3 Planning pipeline

The only role demanded by the LoCoBot Extended Planner node is to find a path for the mobile manipulator considering it as a holistic system. The parameters of the MoveIt! planner are reported in Table 5.8.

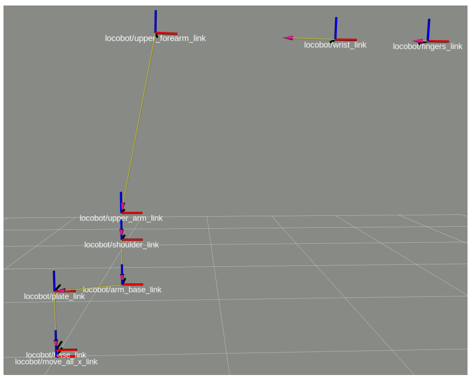
Parameter	Value
Joints state publisher	/locobot/joints_state
Robot description	/locobot/robot_description
Planning group name	locobot_ext_all
Planning library	OMPL
Planner	RRTConnect
Planning time	30 s
Planning attempts	5
Pose reference frame	map

Table 5.8: Table with the parameters used by the extended planner.

It is important to point out that the robot description considered by this planner is the one with the three custom virtual joints added by hand. Moreover, the **locobot_ext_all** planning group, that is part of the custom MoveIt! package created, is constituted by the kinematic chain going from the fixed joint of the mobile base (it is a virtual joint used by the planning framework to attach the robot to the world fixed frame) to the wrist joints, including obviously the three custom virtual joints created; Figure 5.22 shows the kinematic chain described before, while a complete list of the joints can be found in Table 5.9.



(a)



(b)

Figure 5.22: View of the locobot_ext_all kinematic chain (5.22a) corresponding to the pose 5.22b.

Joint name	Type
move_all_x	virtual prismatic
move_all_y	virtual prismatic
move_all_theta	virtual revolute
waist	revolute
shoulder	revolute
elbow	revolute
forearm_roll	revolute
wrist_angle	revolute
wrist_rotate	revolute

Table 5.9: List of joints building the locobot_ext_all group.

Algorithm 2 describes all the steps executed by the node to find a suitable trajectory.

Algorithm 2 Algorithm executed by the LoCoBot Extended Planner

```
1: Extract the pose of the end-effector
2: Added trajectory constraints
3: Set the target pose
4: Start the motion planning
5: if $plan = FOUND$ then
6:     Retrieve final mobile_base pose
7:     Extract the arm's joints trajectory
8: else
9:     Return error
10: end if
```

It is possible to notice that some trajectory constraints are added in Algorithm 2. Besides those coming from the occupancy grid created by Octomap, a further constraint regarding the orientation of the mobile base, and hence the value of the move_all_theta joints is added: the current value of this joint is retrieved and set as a goal for the planning. In this way, the mobile base is forced to have the same orientation it had before planning, that is the one that allows the manipulator to be directly in front of the object to grasp. This latter orientation is the one used to scan the environment while searching for the desired object.

5.7 Dangerous Part Feedback Getter node

The role of this node is to create a structure for the human to provide the robot with feedback on the most dangerous part of the object to be handled. Every time the human sends a request to the Communication Manager, it first checks whether the ID is present in a hard-coded list containing all identifiers of the dangerous objects, and if the requested item is considered somehow dangerous, then the Dangerous Part Feedback Getter node is involved.

The communication interface of this node can be seen in Figure 5.23. It is composed of:

1. a topic named "*/dangerous_part_getter/request*". It is used by the Communication Manager to give the Feedback Getter node the last RGB frame with the object to be handled, hence, messages published on this topic are of type sensor_msgs/Image;
2. the "*/dangerous_part_getter/result*" service is used to retrieve the status of the node together with the feedback provided by the human.

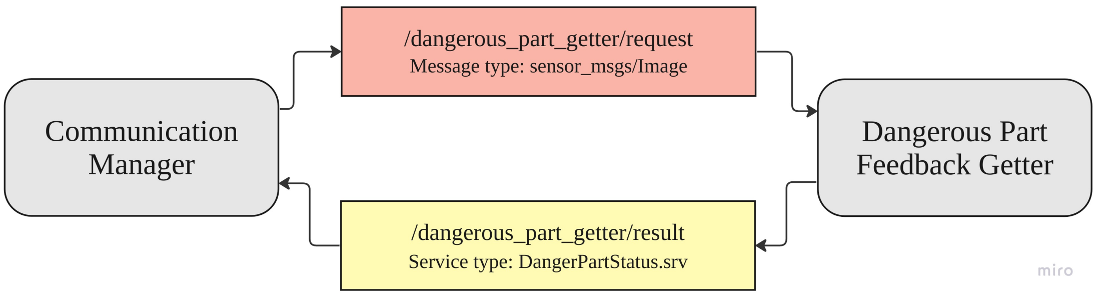

/dangerous_part_getter/request
Message type: sensor_msgs/Image
Communication Manager
Dangerous Part Feedback Getter
/dangerous_part_getter/result
Service type: DangerPartStatus.srv
miro

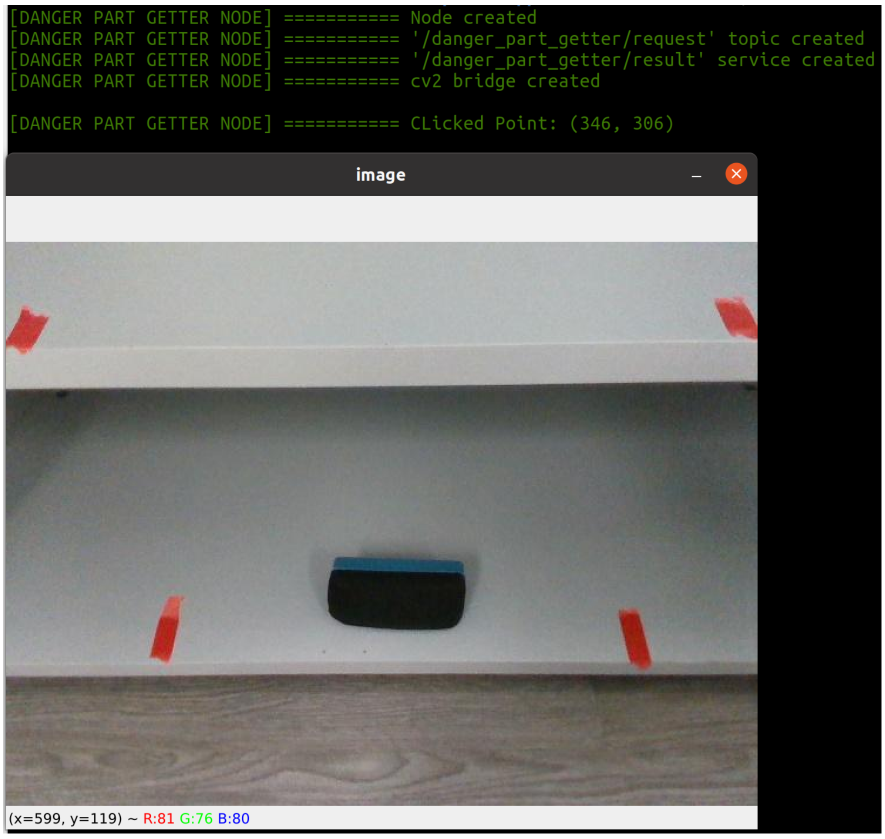
Figure 5.23: Communication interface of the Dangerous Part Feedback Getter node.

The service type is totally custom and contains three message fields: a std_msgs/String message called **node_status** representing the current state of the node, and two std_msgs/UInt32 messages (**danger_point_x**,**danger_point_y**) for the coordinates (u, v) of the point indicated as dangerous by the human.

5.7.1 Feedback infrastructure

The most important functionality of this node is the implementation of a structure through which the human can select the most dangerous part of the object. In order to complete this task, the OpenCV library [83] has been used: it is an open-source computer vision and image processing library that allows to manage and interact with images rapidly.

As soon as a sensor_msgs/Image message is received, the Dangerous Part Feedback Getter node executes the callback that simply consists of converting the Image message in a format that can be handled by OpenCV using a cv_bridge, and displays it in order to collect the feedback from the person. This feedback is provided by the human through a mouse click: a callback to read and store the coordinates of the clicked point has been defined. In this way, every time the result service is called, the response that will be sent will contain the coordinates of the last point indicated on the image. Figure 5.24 shows an example of the feedback structure and the result provided to the robot.



```
[DANGER PART GETTER NODE] =========== Node created
[DANGER PART GETTER NODE] =========== '/danger_part_getter/request' topic created
[DANGER PART GETTER NODE] =========== '/danger_part_getter/result' service created
[DANGER PART GETTER NODE] =========== cv2 bridge created

[DANGER PART GETTER NODE] =========== CLicked Point: (346, 306)
```

Figure 5.24: Example of the communication interface through which the human operator can provide a feedback message about dangerous point.

Chapter 6

Custom Dataset and YOLO training

In order to successfully generate the grasping point for an object, it is necessary to check if the requested object is present in the current scene or not. This task is called object detection: understanding which objects surround the robot is fundamental to find the correct way to interact with them.

Several techniques can be exploited to complete this task, using different methods like the analysis of the image gradients (Histogram of Oriented Gradients or HOG), detection of edges (where they are and how they are connected), color-based methods and others. With the last advancements made in the ML and AI fields, several new approaches have been proposed, and nowadays the most common way to face the object detection task is based on CNN: they are very versatile and lots of different models can be found to be adapted to a specific case of study. The type of architecture and the number of parameters characterizing the network strongly depends on the hardware available and impacts on the time required to make each single prediction. Taking into consideration that the hardware capabilities of the LoCoBot are limited and a GPU is not available, and keeping in mind also that the inference time must short since object recognition is just a step of a more complex pipeli, the Grasping Point Generator makes use of a YOLOv5 NN.

YOLOv5 has different architectures characterized by different sizes and parameter numbers; in this way, it is possible to choose and use the one that best suits one's own needs. From the Ultralytics website [84], it is possible to download and train five different types of architectures, and each one has its own characteristics; Table 6.1 summarizes their main features.
Thanks to its reduced number of parameters and the small amount of time required

Model's name	Size	Parameters [M]	CPU [ms]	GPU [ms]
YOLOv5n	Nano	2.6	73.6	1.06
YOLOv5s	Small	9.1	120.7	1.27
YOLOv5m	Medium	25.1	233.9	1.86
YOLOv5l	Large	53.2	408.4	2.50
YOLOv5x	Extra-large	97.2	763.2	3.81

Table 6.1: Summary table of all YOLOv5 architectures.

to compute the results, YOLOv5s has been chosen: it takes about 120.7 ms to compute the required predictions, making the object detection task fast and lightweight.

All versions of this CNN have the same structure, what changes is the number of convolutional layers characterizing each part. As can be seen from Figure 6.1, the entire architecture can be divided into three main parts:

1. the **CSP-Darknet backbone**, that is a deep CNN to which the authors applied the Cross Stage Partial (CSP) network strategy. YOLO makes use of residual and dense blocks in order to enable the flow of information to the deepest layers and to overcome the vanishing gradient problem. On the contrary, the use of these types of blocks introduces the redundant gradients problem, and this is where the CSP strategy comes into action: it helps tackling this problem by truncating the gradient flow;
2. the **neck** of the architecture is further divided into two parts: PANet (Path Aggregation Network) that enhances instance segmentation by utilizing multi-path aggregation, adaptive feature pooling, and instance-aware convolution for accurate object delineation, and Spatial Pyramid Pooling (SPP), used to capture features at multiple scales, improving object recognition in varying spatial contexts.
3. the **head** of the network, composed of three sets of convolution layers that predict the location of the bounding boxes in terms of their positions (x, y) and extensions (height and width), the scores and the objects classes.

6.1 Objects dataset

To properly train the YOLOv5 neural network, a custom dataset containing objects that can be commonly found in laboratories has been realised employing the useful tools made available by Roboflow [86]. This dataset contains a total of 804 images of objects belonging to 15 different classes that are:

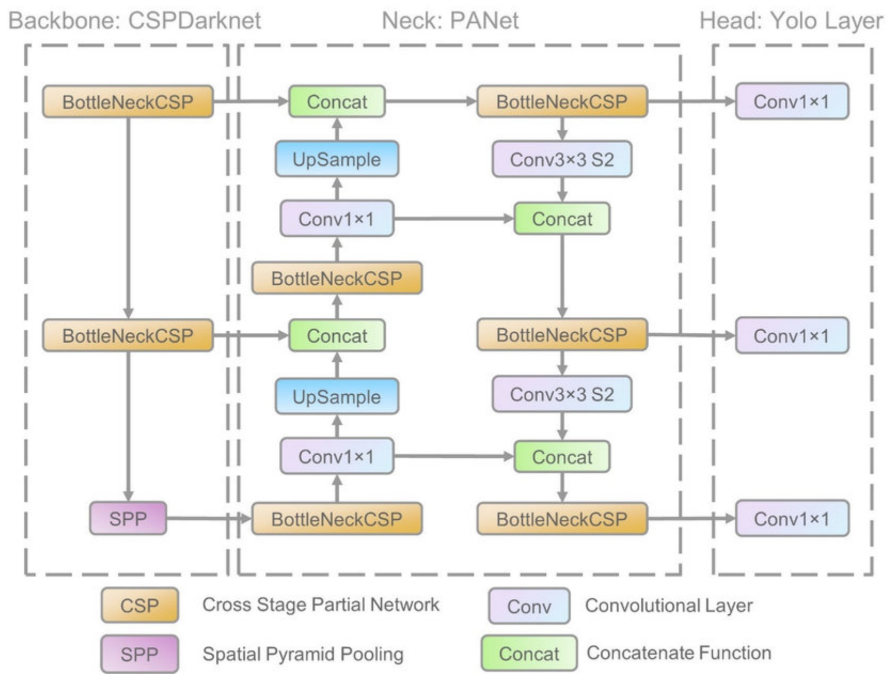


Figure 6.1: YOLO networks architecture [85].

1. blades
2. box
3. charger
4. cutter
5. eraser
6. glue_stick
7. large_box
8. level
9. marker
10. object
11. pen
12. pliers
13. screwdriver
14. tape_roll
15. whiteboard_pen

Some images taken from the dataset are shown in Figure 6.2. It is important to notice that, in order to make the images as similar as possible to those collected while running the entire grasp pipeline, elements present in the dataset have been captured directly from the camera mounted on the LoCoBot; in this way, the field of view and the size of the images are almost identical to those really used by the robot itself.

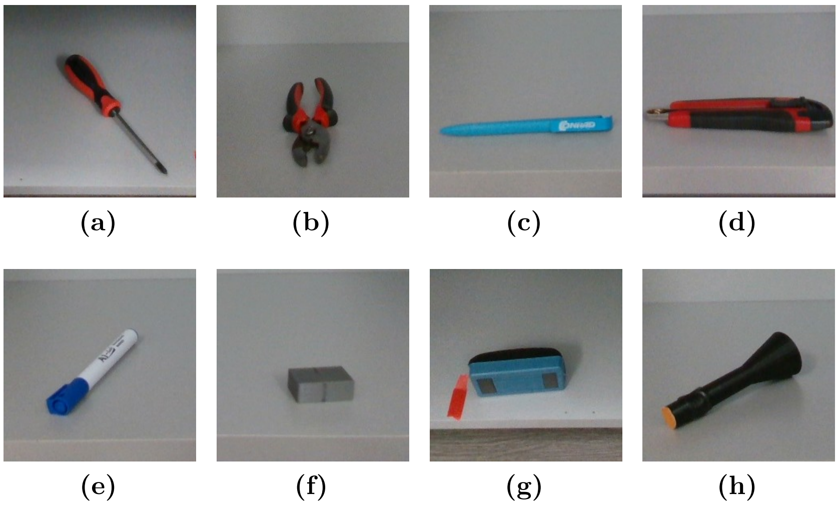
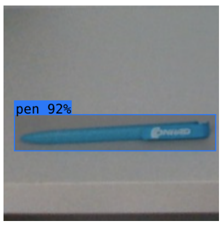


Figure 6.2: Some examples of the images composing the custom dataset.

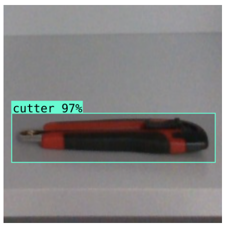
Understanding whether an object is present or not in the scene is not sufficient: it is necessary to compute its exact position with respect to the image frame. YOLOv5 architecture is able to compute the bounding box of an object if it is present in the image, but to properly work it has to be trained accordingly. In order to compute the correct positions as a result of the predictions, bounding boxes corresponding to the images in the dataset have been manually created and the corresponding classes associated with them. Bounding boxes of the images in Figure 6.2 are shown in Figure 6.3.

(a)

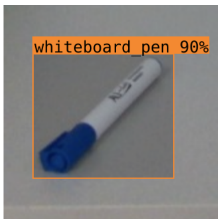
(b)



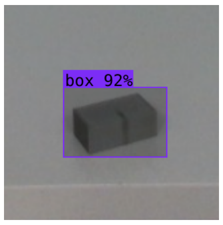
(c)



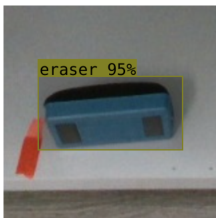
(d)



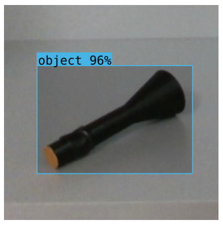
(e)



(f)



(g)



(h)

Figure 6.3: Some examples of the detected bounding boxes and the class associated to them.

6.2 Training results

The YOLOv5 loss function is a comprehensive combination of different components tailored for optimizing accurate object detection and bounding box prediction during training. It includes:

1. **Objectness Loss**: This term evaluates the model's ability to accurately predict whether an object exists within a given bounding box.
2. **Classification Loss**: This component measures the error in predicting the class of the object contained within the bounding box.
3. **Localization Loss**: The accuracy of predicting the coordinates (x, y, width, height) of the bounding box is assessed through the localization loss.
4. **Confidence Loss**: Combining objectness and classification losses, the confidence loss emphasizes correct localization and precise class prediction.

The formulation of the YOLOv5 loss function can vary slightly depending on the specific variant being used (e.g., YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) and the objectives of the training process. The overarching goal is to minimize errors in predicting object presence, class labels, and bounding box coordinates, contributing to the overall effectiveness of the model in object detection tasks.

YOLOv5 has been trained to recognize and localize objects belonging to the custom dataset using the Stochastic Gradient Descent (SGD) optimizer. SGD performs the optimization task on a subset of the training data (a mini-batch) rather than the entire dataset. This introduces randomness, which can help to escape local minima and speed up convergence. Hyperparameters used to train the network are reported below and summarized in Table 6.2:

- the network has been trained for a total of 150 **epochs**;
- at each epoch, a total of 320 images organized in **mini-batch** of 32 elements have been used;
- **lr0** (Learning Rate at Start) is the initial learning rate at the beginning of training and was set equal to 0.01;
- **lrf** (Final Learning Rate), which is the learning rate at the end of training, was also set to 0.01. Different values of lr0 and lrf are used to change the learning rate during the training process according to the behaviour of the loss function; here they have been set both equal to 0.01, thus keeping the learning constant for the entire training process;
- **momentum** value used for the SGD optimizer was 0.937. Momentum helps accelerate gradient descent in the relevant direction and dampens oscillations of the loss function;
- **weight decay** is L2 regularization term (set to 0.0005), which penalizes large weights to prevent overfitting;
- **bounding box loss weight** equal to 0.05;
- **class loss weight** was set to 0.5;
- the **objectness loss weight** was 1.0.

The entire dataset has been split into 3 sets, training, validation and test sets, containing respectively 644 (80%), 80 (10%) and 80 (10%) images. The number of elements constituting the training was increased using an image augmentation process where elements can be flipped horizontally and used to create a mosaic in order to have more than one object in each image; this latter technique allows to train the network to recognize and localize objects also in cluttered scenes.

Figure 6.4 shows the results obtained after the training phase. It is possible to see the metrics used to evaluate the performances obtained are:

1. **precision**, computed as

$$Precision = \frac{True\ positives}{True\ positives + False\ positives}, \tag{6.1}$$

Hyperparameter	Value
training epochs	150
mini-batch size	32
learning rate	0.01
momentum	0.937
weight decay	0.0005
bounding box loss weight	0.5
class loss weight	0.5
objectness loss weight	1.0

Table 6.2: Summary table of the hyperparameters used to train the YOLOv5 network.

where a prediction is considered a True positive when it is correct, while it is a False positive when the actual class of the object is different with respect to the one predicted. The precision metric measures how many predictions on images belonging to the test set have been computed correctly, and a value of 93% has been reached;

2. **recall**, computed as

$$Recall = \frac{True\ positivies}{True\ positives + False\ negatives}, \tag{6.2}$$

where false negatives are all those elements belonging to a class that have not been detected. Recall measures how many of the actual images containing an object have been correctly predicted. The maximum value reached for the recall metric is 95%;

3. **mean average precision** (mAP) is a comprehensive metric that considers the precision and recall across different confidence thresholds, providing a more nuanced evaluation of the model's performance. In order to obtain a metric that is more related to the bounding box detection task, the **mAP_0.5** has been used, where the subscript 0.5 denotes the intersection over union (IoU) threshold used to consider a detection as correct; a final value of 95% has been obtained for the mAP_0.5 metric.

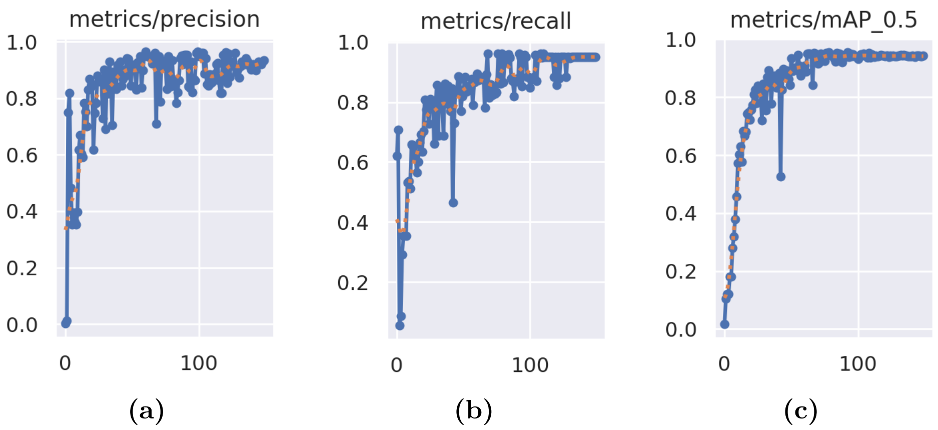


Figure 6.4: Graphs showing the evolution of the metrics used to evaluate the performances of the trained network.

Figure 6.5 shows the evolution of the loss functions with respect to the number of epochs. It is possible to notice how they all decrease smoothly towards zero.

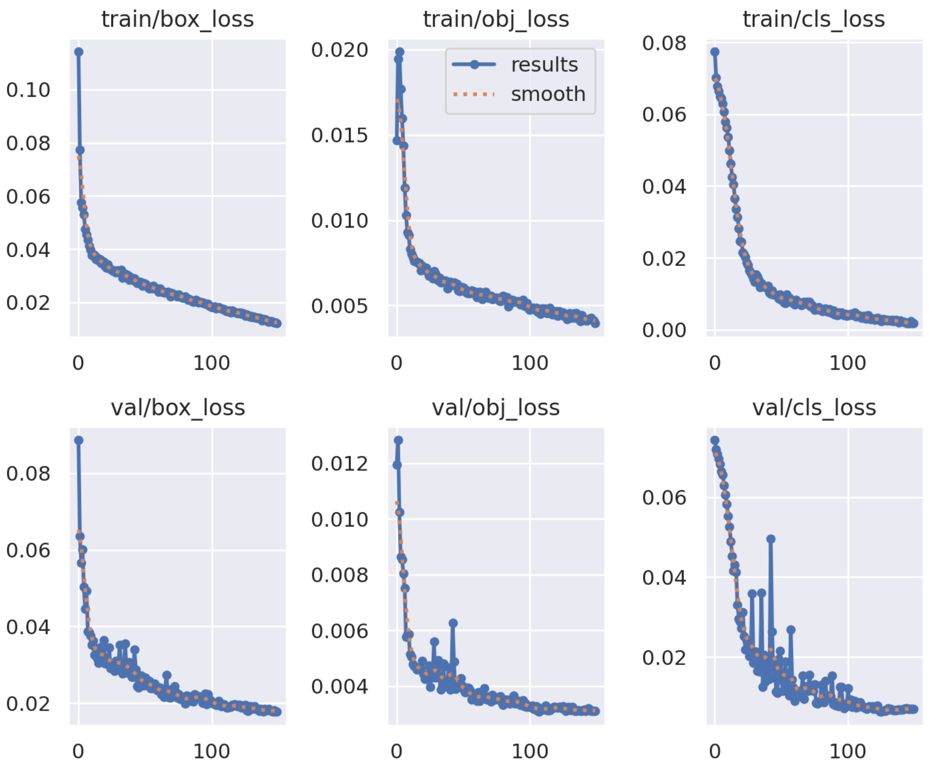


Figure 6.5: Evolution of the loss functions.

Chapter 7

Simulation environment and results

The first step to understand whether the software developed allows to reach the thesis goal is to create a simulation environment and perform some tests using it. The objective of this chapter is to provide an overview of the simulation world created to conduct the first experiments, and the analysis of the assumptions made and of the results obtained.

7.1 Gazebo world

Since the objective of the simulation is to test the code developed and to inspect how to robot behaves, it is important that the simulation environment is close to the real one, in which the mobile manipulator is expected to work. In order to satisfy this constraint, a fake world in Gazebo has been realized, and a representation of it can be found in Figure 7.1. It is possible to see that it reflects all the characteristics of the real environment where a LoCoBot should be used, which are:

1. **indoor** environment. This requirement is needed because of the mobile base used, that is not reliable on surfaces that are too rough and steep;
2. **structured** or **semi-structured** environment, that allows the creation of a map used for both navigation and localization;
3. presence of **shelves** and **drawers** that act as obstacles to be avoided while navigating, but at the same time are used to store objects to grasp;
4. presence of **humans** with whom the robot has to share its workspace.

Referring to the search pipeline (Figure 5.3), it is important to recall that the Communication Manager uses two dictionaries to map the locations, where a given

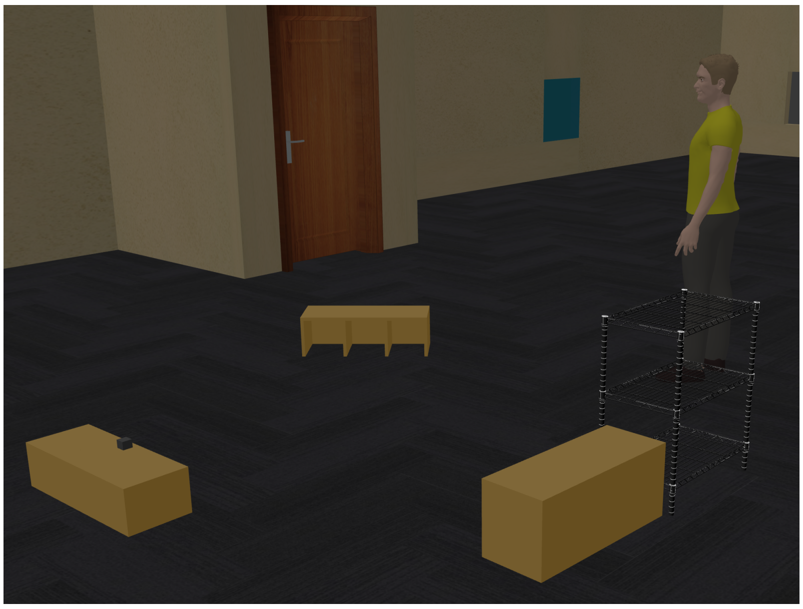


Figure 7.1: Representation of the Gazebo world used in simulation.

object can be found, and the coordinates of the locations themselves. In order to test the complete search phase, three known locations have been defined; Figure 7.2 shows their positions and names, while Table 7.1 contains their coordinates with respect to the world reference frame. It is possible to see that there are four known locations:

1. **Home** point is the position where the mobile manipulator waits to receive any request;
2. **Location 1** and **Location 2** are the positions where objects can be stored, hence they are used as targets for the mobile base during the search phase;
3. **Depot** is the base's target position during the place pipeline.

Name	Position [m]	Orientation [rad]
Home	(0, 0, 0)	(0, 0, 0)
Location 1	(1.2, 0, 0)	(0, 0, 0)
Location 2	(-0.8, -0.5, 0)	(0, 0, -2.44)
Depot	(0, 1.2, 0)	(0, 0, 1.57)

Table 7.1: Coordinates of the known locations in the Gazebo world.

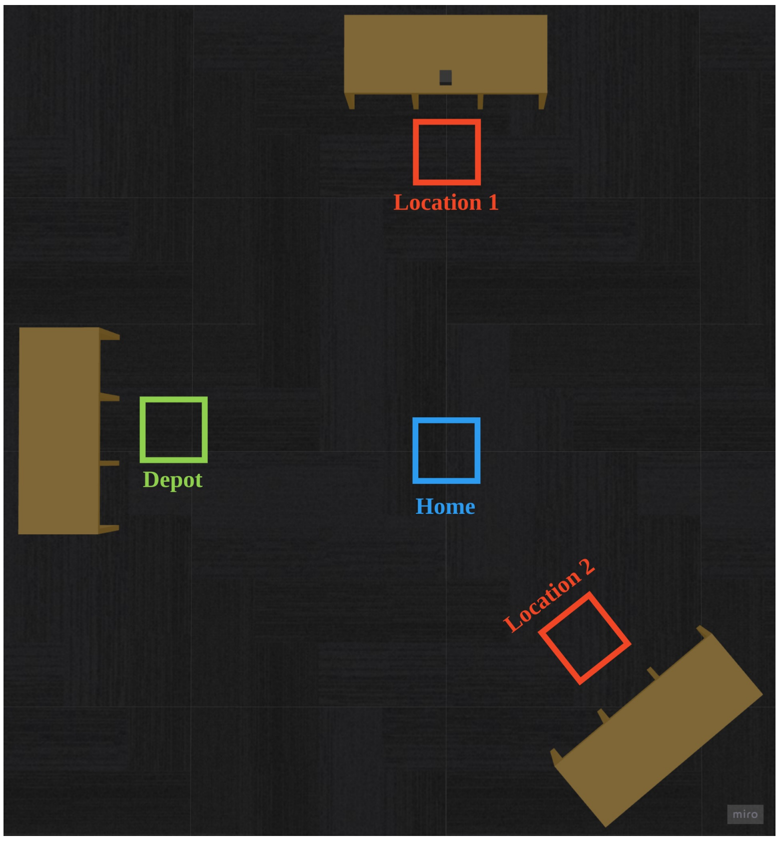


Figure 7.2: Visualization of the known location in the Gazebo world.

7.1.1 Occupancy Grid

To let the robot sense the environment and have the knowledge needed to navigate and obtain collision-free trajectories, an occupancy grid has been created using Octomap. Figure 7.3 shows the resulting occupancy grid of the main locations visited by the robot. It is important to point out that building a fine-grained occupancy grid is an expensive process from a computational point of view, especially if a GPU is not available. By running different simulations with diverse configurations of Octomap, the most impactful parameters seemed to be the **vision range**, that is the maximum distance at which it is possible to detect obstacles, and the **voxel size**, that is the size of the cells of the occupancy grid. Using a distributed architecture to enhance the performance available, it has been possible to set $max\_vision\_range = 5.0\ m$ and $voxel\_size = 0.02\ m$. The occupancy grid resulted to be fine-grained enough to give a suitable estimation of the space occupied by the objects to grasp, and also to have a perception of the empty spaces of the drawers containing the items.

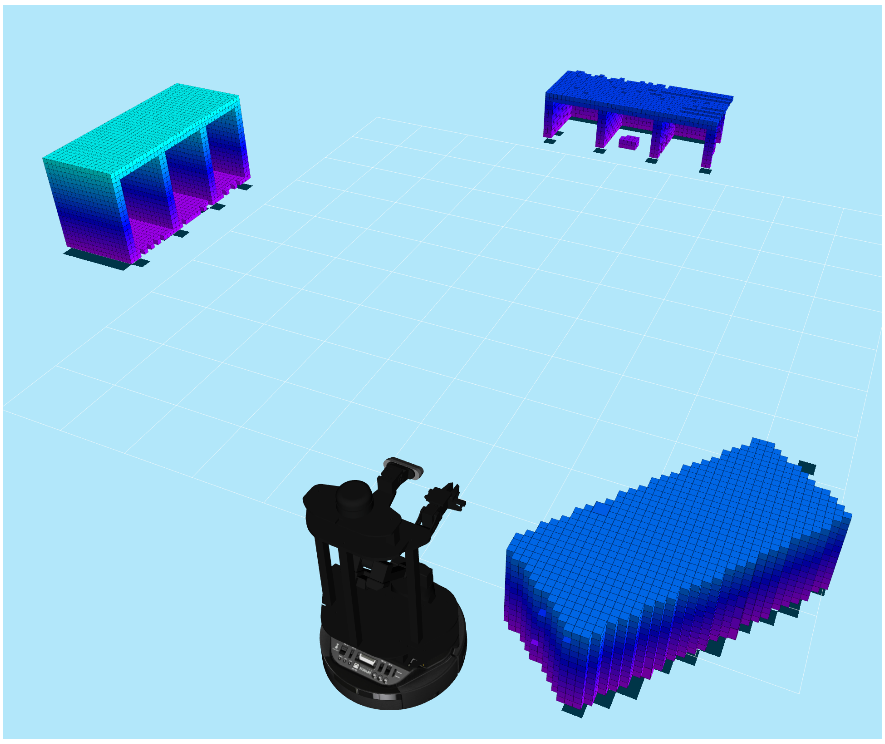


Figure 7.3: Occupancy grid of the simulation environment.

Another important consideration regards the filtering of the ground voxels. Due to the camera orientation and inclination with respect to the ground, the occupancy grid would have contained also a representation of the ground, leading the motion planners to consider the robot in a constant collision state. To solve this problem the voxels belonging to the ground have been deleted by setting a parameter of the Octomap server called **occupancy_min_z** equal to 0.03 m. Here the assumption is that all obstacles having a maximum height that is less than 0.03 m are considered as part of the floor and, hence, neglected.

Parameter name	Value
Point cloud topic	/locobot/camera/depth_registered/points
Frame_id	map
max_vision_range	5.0 [m]
voxel_size	0.02 [m]
occupancy_min_z	0.03 [m]

Table 7.2: Summary table of the configuration parameters used by Octomap to run the simulations.

7.1.2 Environment mapping

The robot needs to have a clear map of the environment in order to successfully navigate in it and to localize itself. To solve this task, the RTAB-map ROS package has been used to create the map shown in Figure 7.4. Is it possible to notice how smoothly the wall surfaces have been detected, and the clear presence of the drawers' projections on the map's flat surface, although the presence of some (simulated) noise that affects the vision sensor.

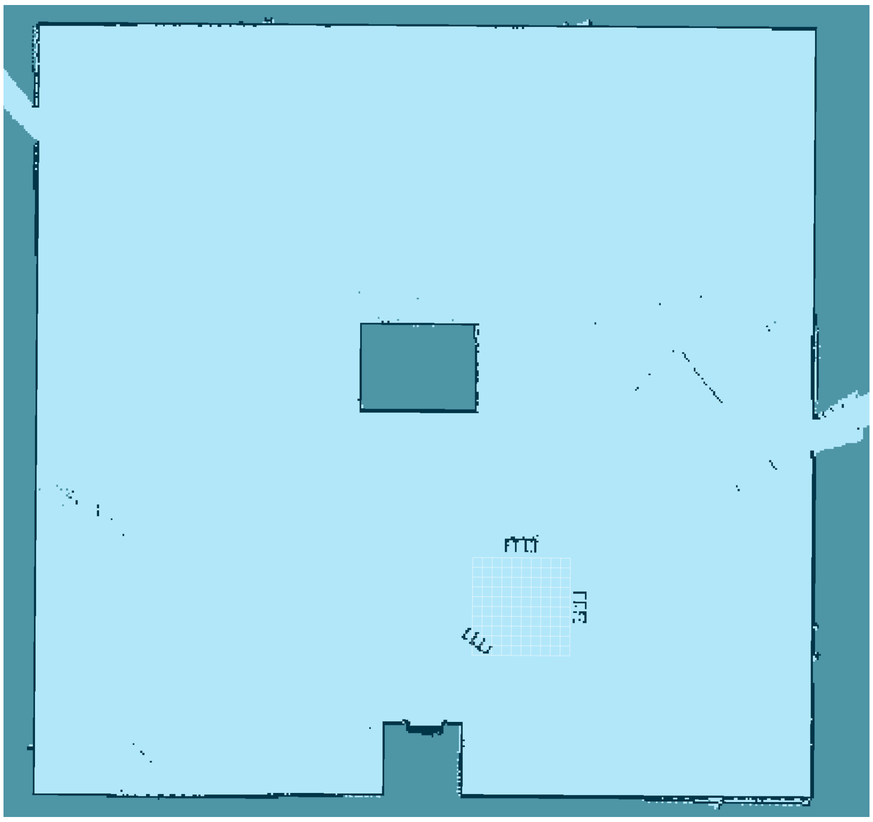


Figure 7.4: Map of the simulation environment created using the RTAB-map package.

7.2 Experiments in the simulated environment

The main goal of the experiments conducted in the simulated environment was to test the search phase and the grasp pipeline, and to measure how much time they require.

7.2.1 Search phase

To test the search phase, the mobile manipulator is requested to find an eraser, that has ID equal to 4; a proper whiteboard eraser model is not present to be loaded in the Gazebo world, hence a box with the same size as the real object that is detected as an eraser is used. The main steps characterizing the search phase are depicted in Figures from 7.5 to 7.10: the description of the various phases is directly provided in the figures' descriptions.

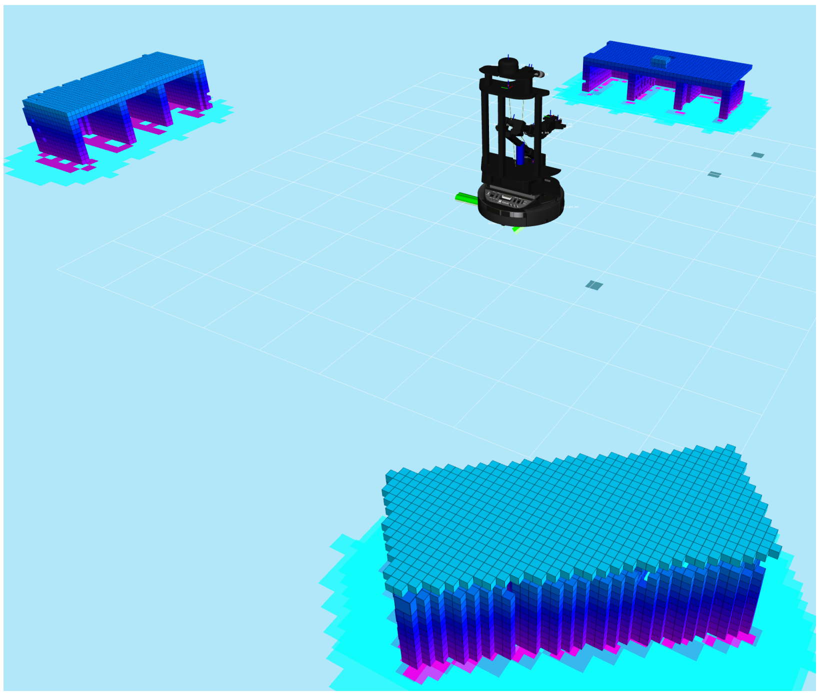


Figure 7.5: The LoCoBot starts from the home position and waits a request from the human.

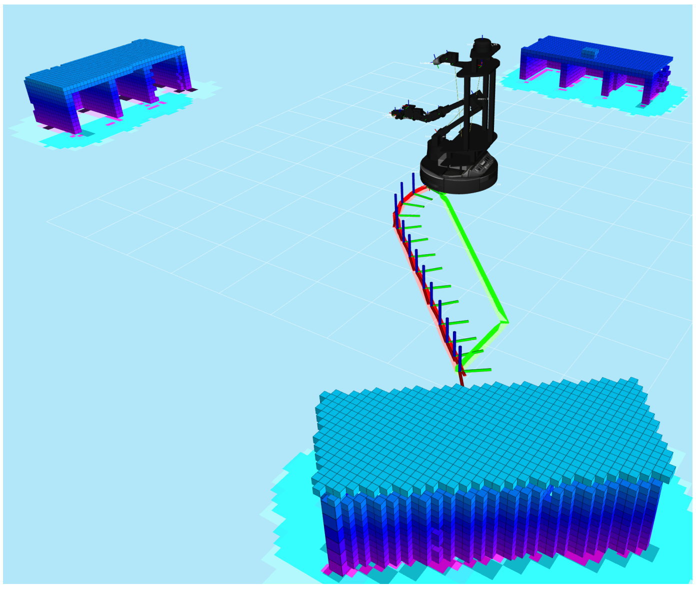


Figure 7.6: As soon as a request from the user is received, the first location is extracted from the dictionary. In this case, the first location where to search is Location 2, hence a global (green) and a local (red) trajectories are computed to reach this position.

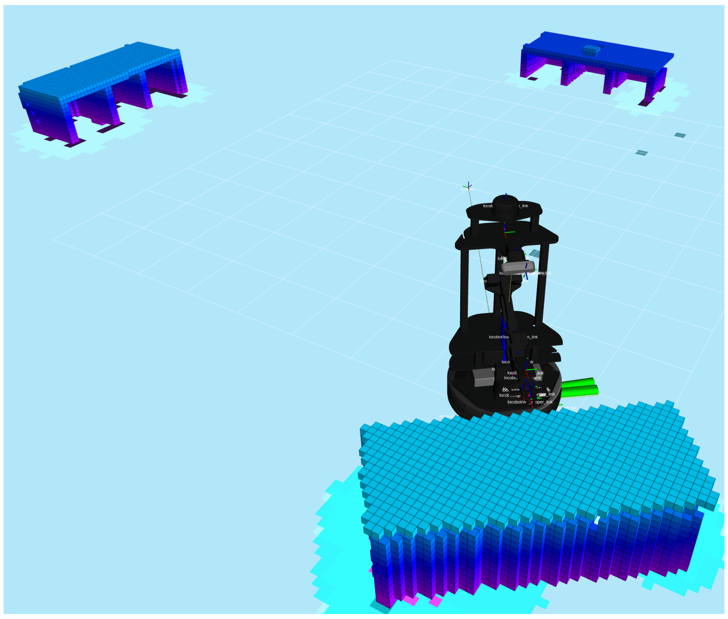
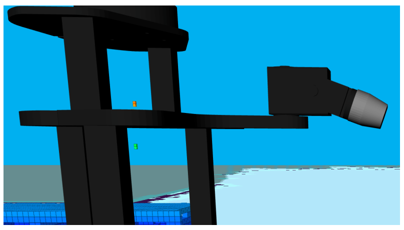
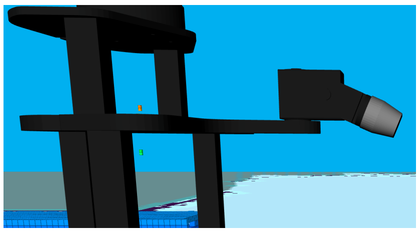


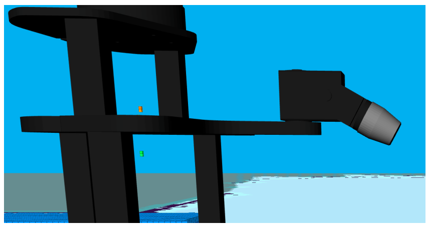
Figure 7.7: LoCoBot reached Location 2 pose.



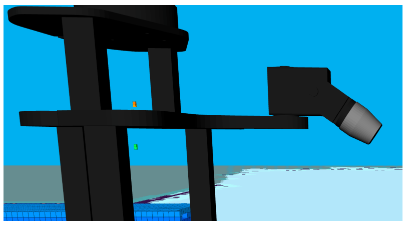
(a)



(b)



(c)



(d)

Figure 7.8: After the desired location is reached, the Communication Manager starts moving the camera to locate the object. Four different positions corresponding to four tilt angles are used and shown here: $tilt = 0.3$ (a), $tilt = 0.4$ (b), $tilt = 0.5$ (c), $tilt = 0.6$ (d) .

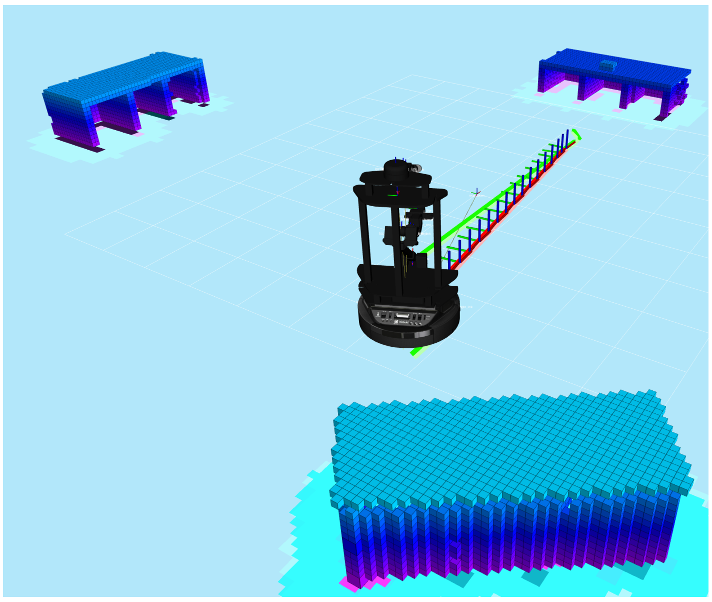


Figure 7.9: During this simulation, the object has been positioned in Location 1. For this reason, from Location 2 it is not possible to find the desired item, and as a result, the next point is reached.

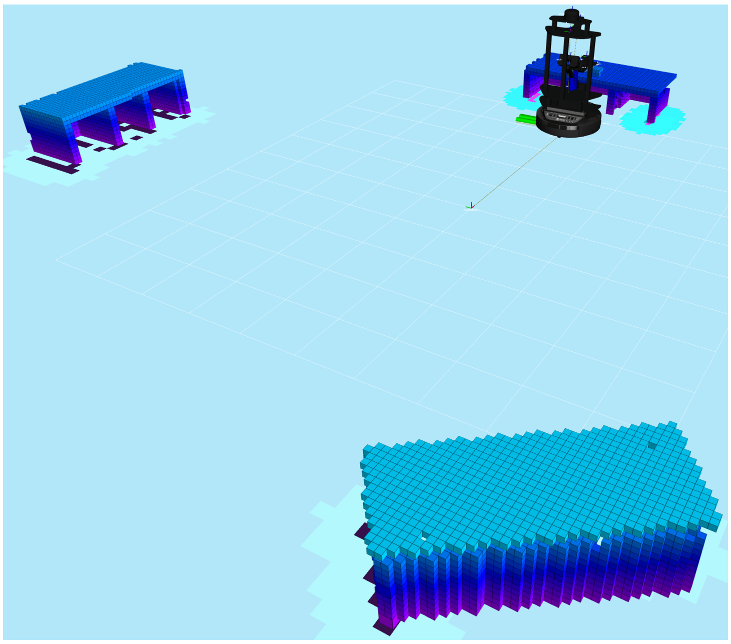


Figure 7.10: LoCoBot reached Location 1. Again, the Communication Manager will move the camera trying to locate the requested item.

Depending on the result of the object detection done from the last known location, the robot would start the pick-and-place pipeline or it would return to its home position. In the worst case, that is the one where all locations must be explored before finding the object, the entire search phase requires about **2.5 minutes** to be completed, and a total distance covered of about 5 meters. The main overhead is due to the poor simulation performances of the computer used, while the mobile paths computation can be neglected and the time required to process a single image with the trained YOLOv5 network is of **0.309 seconds**, leading to a total of **2.5 seconds** spent processing the images.

7.2.2 Pick-and-place routine

In this section, the main steps executed to complete the grasping pipeline are reported. In this case, it is also important to point out that the objects to grasp must have dimensions, shape and weight compatible with the gripper and robotic arm properties. In order to test whether the pick-and-place pipeline described in Chapter 5 works well, and to satisfy all the constraints that the desired object must meet, the same object used to evaluate the search pipeline has been used. Figure from 7.11 to 7.5 describe how the entire routine is executed; details are provided in the captions.

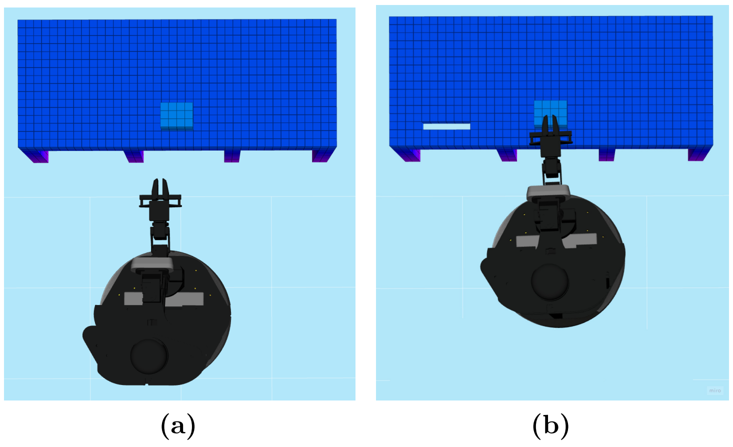


Figure 7.11: After the search phase, if the requested item is found, the grasp and pre-grasp poses are computed and a motion plan to the LoCoBot Extended Planner node is requested. As a result, the mobile base is moved to reach the pose resulting from the plan. The Figure shows how the base is repositioned (b) with respect to the arrival pose (a).

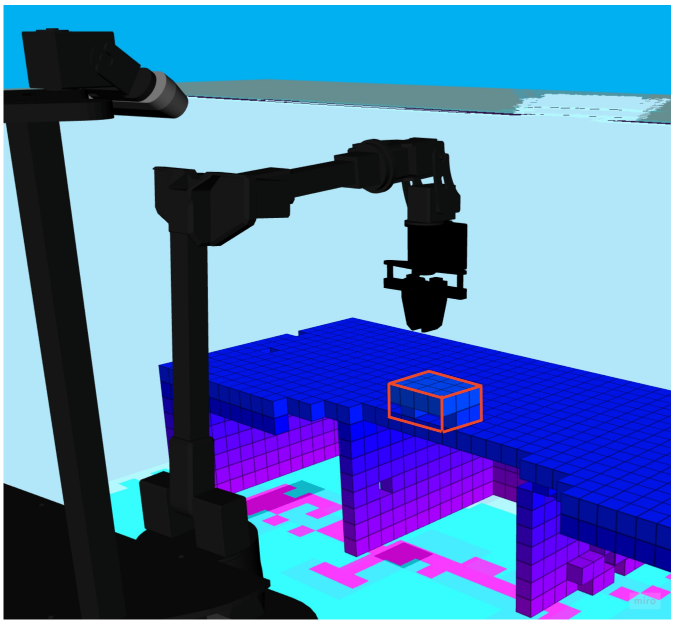


Figure 7.12: Pre-grasp pose reached as a result of the execution of the planned trajectory. Here the voxels occupied by the object are marked in red. Notice that, due to the voxel size, the occupied space is greater than the real one.

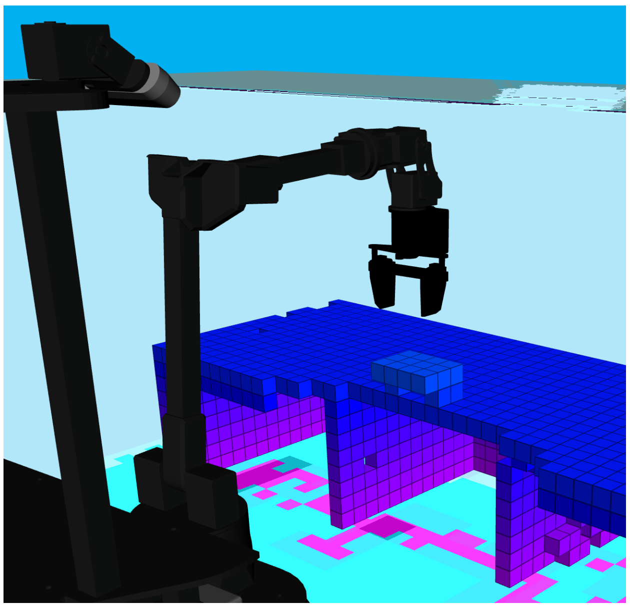


Figure 7.13: After having reached the pre-grasp pose, the gripper is opened and an object is attached to the mobile manipulator's fingers.

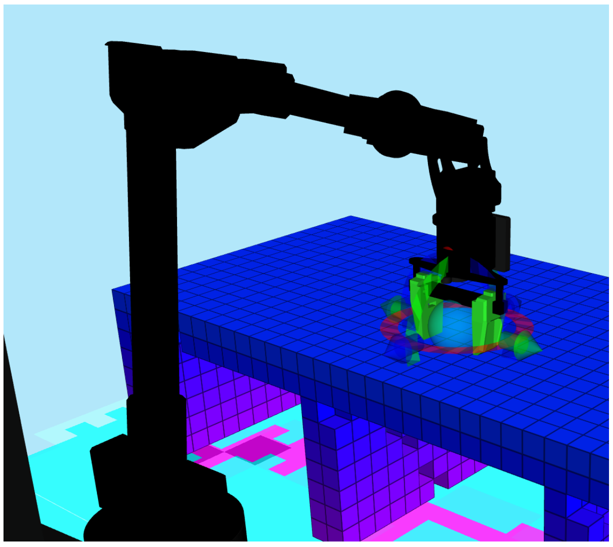


Figure 7.14: In this figure it is possible to see the object attached to the gripper and the grasp position reached. Moreover, as a result of attaching the object, the occupancy grid around the object is cleared.

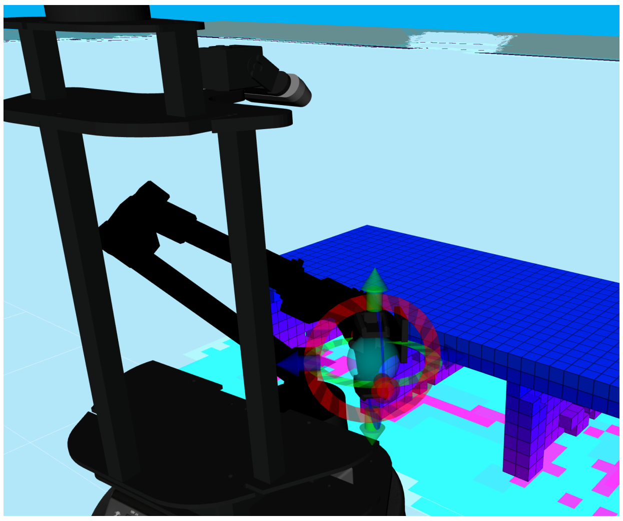


Figure 7.15: After the success of the grasping actions, the arm is taken to the arm_secure_pose: this is done by requesting a plan to the Arm Handler node by specifying the name corresponding to the desired pose.

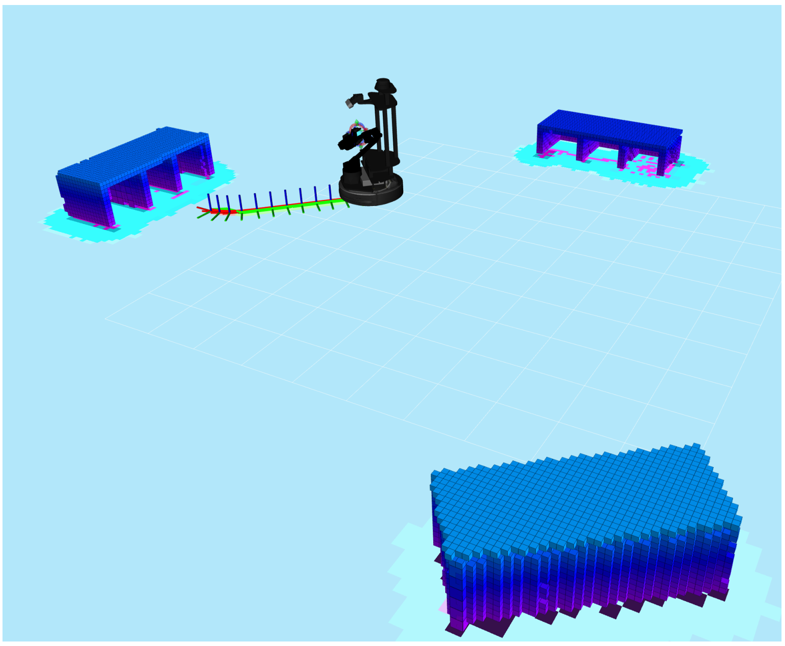


Figure 7.16: Once the object has been grasped and the manipulator has reached the arm_secure_pose, the Communication Manager retrieves the depot's position and requests a plan for the mobile base.

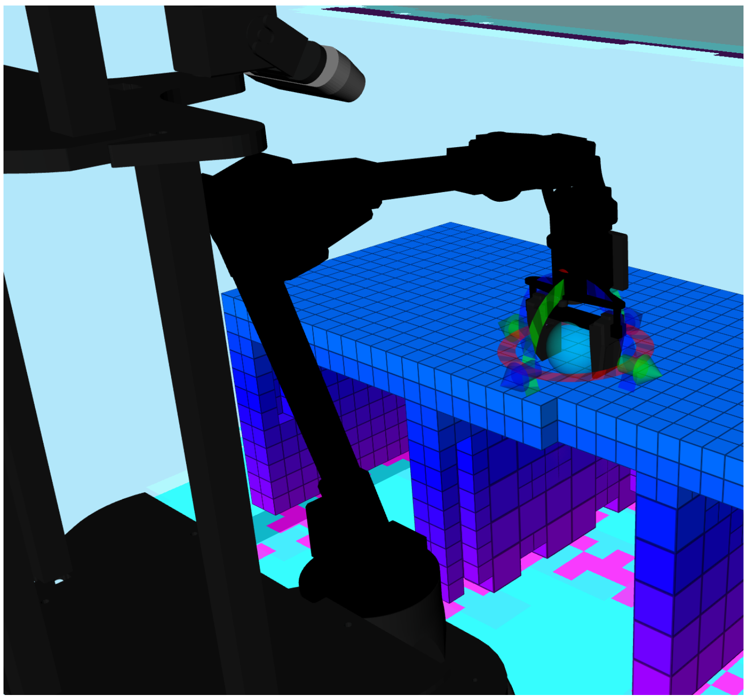


Figure 7.17: Once the depot position is reached, the Arm Handler node requests a plan to reach the placing pose of the end-effector and open the gripper.

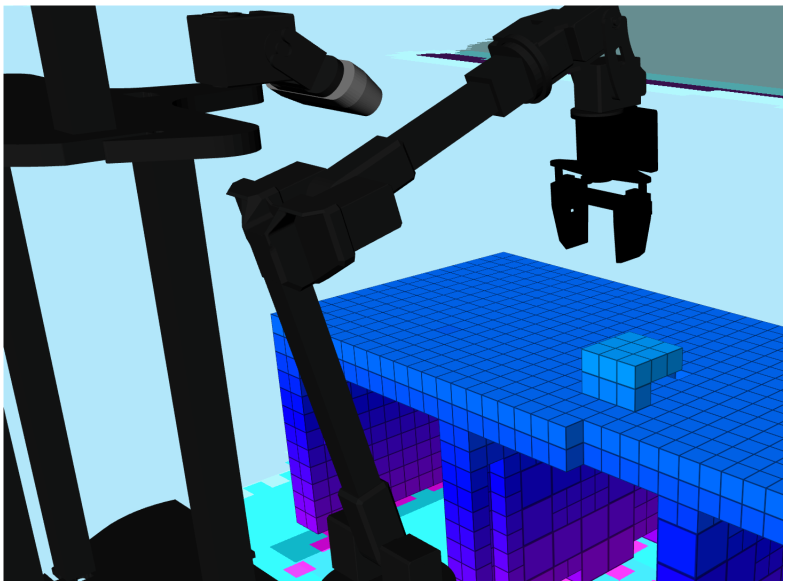


Figure 7.18: After the gripper opens, the object is detached from the robot's gripper. This leads to have the voxels present in the planning scene again.

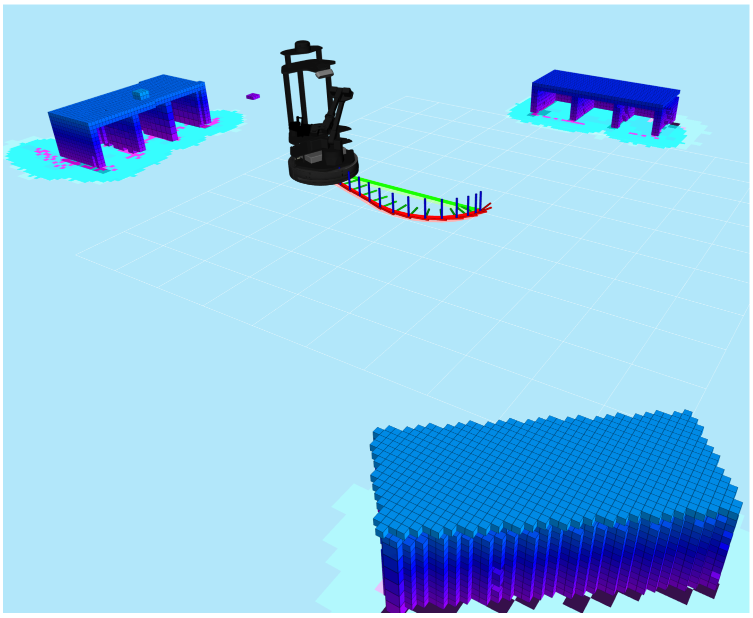


Figure 7.19: After having successfully retracted the arm to reach the arm_-secure_pose, the robot reaches again the Home location waiting for other requests.

The performances of the pick-and-place pipeline are again measured in terms of the time needed to complete this task. The main impact is due to the time given to the planner to find a suitable trajectory. During the experiments conducted in simulation, the considered planner was the **OMPL RRTstar**. A total of **30 seconds** for each plan and a maximum of **5 attempts** are available. The RRTstar planner takes all the planning time to find a solution, even if a suitable trajectory is found in less time. The total amount of time results to be mainly affected by the number of planning attempts that are actually performed: the LoCoBot Extended planner usually requires all the available attempts (5) before trying a suitable plan, while the Arm Handler, that controls only the arm's joints, requires from 1 to 2 attempts. Since the entire pick-and-place pipeline requires one plan from the LoCoBot Extended Planner node and two from the Arm Handler (one to reach the grasp pose starting from the pre-grasp pose, and the other one to place the object on the drawer), in the worst case a total of about **2.5 minutes** are necessary.

Chapter 8

Experiments on the real robot and results validation

This chapter illustrates how the code tested by running different simulations has been adapted, in order to make a real mobile manipulator able to search for and safely handle objects present in a workspace shared with humans. A description of the real environment used for testing, together with the assumptions made are deepened hereafter.

8.1 Environment description

The mobile manipulator, for which the software architecture described in Chapter 5 has been developed, is intended to work in indoor environments shared with humans, in which objects are stored on shelves, drawers and cabinets. A perfect example of such a context is the Robotic Laboratory at the Department of Electronics and Telecommunications Engineering (DET) at Politecnico di Torino, the place where tests described in this chapter have been conducted. As can be seen from Figure 8.1, it is an indoor and semi-structured environment, where robots and people work together, and it is characterized by the presence of obstacles (both static and dynamic) and, above all, storage places that can contain any kind of objects.

The first assumption that has been made regards the robot's starting position: keeping in mind that the mobile manipulator considered has to work as a human assistant, it is supposed to stand right next to the place where a person works. For this reason, how it is shown in Figure 8.2, the starting and ending location of the LoCoBot used here is right next to a person's desk, in a position where the person's movements are not hindered.

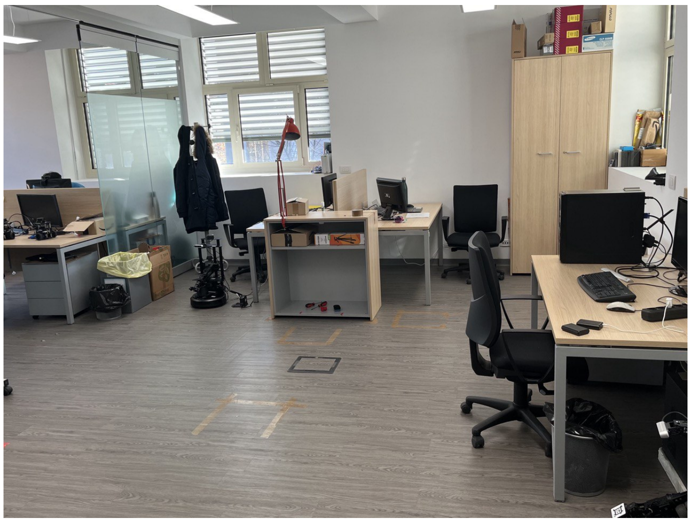


Figure 8.1: Real environment where experiments have been done.

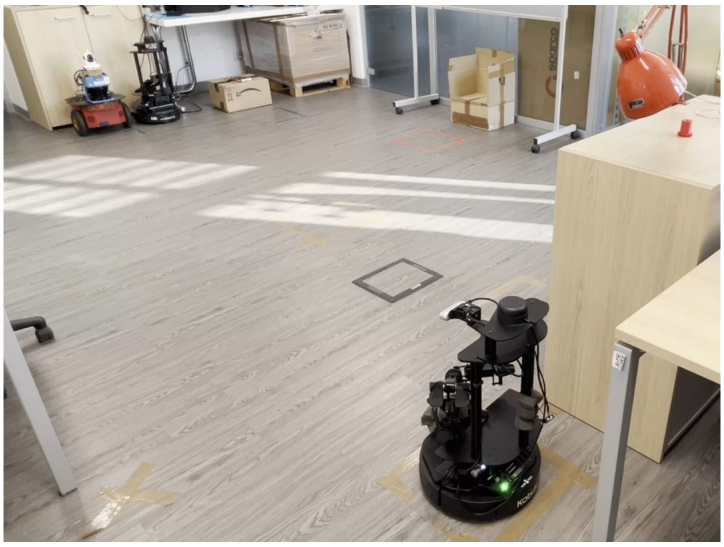


Figure 8.2: Starting position of the LoCoBot next to the person's desk.

As it has been done in the simulation environment, some well-known spots have been defined; a brief description of them and of their uses follows:

1. **home location** is the position where the robot stands right next to the desk of the person it is assisting. This spot is the one where the searching phase starts and also the one where the mobile manipulator goes back once it has completed the request it received;
2. **dataset location** is right in front of a cabinet containing some objects the

person can ask for. The name of this location highlights that is the point from where images building the dataset described in Chapter 6 have been taken;

3. **general location** is another spot containing objects the robot is able to recognize;
4. **depot location** is the location of a container inside which grasped objects are released.

Table 8.1 reports the positions and orientations with respect to the world reference frame of the locations described above.

Location's name	Position [m]	Orientation [deg]
Home	(0, 0, 0)	(0, 0, 0)
Dataset	(0.6, -0.8, 0)	(0, 0, 180)
General	(4.34, 1.68, 0)	(0, 0, 0)
Depot	(2.86, -2.21, 0)	(0, 0, -90)

Table 8.1: Table describing the pose of the real locations used for testing.

8.1.1 Obstacle avoidance

To solve the problem of obstacle avoidance, an occupancy map created using Octomap was used (like during the simulations). It is important to notice that the occupancy map generated is not stored; hence, every time a new run starts the occupancy grid is empty and it is updated as the experiment continues. Although this aspect may be redundant, it allows to deal with new static obstacles that were not present before. From Figure 8.3, it is possible to see the presence of a large free area in the center of the laboratory, while the main obstacles appear at the edges.

It is important to point out that the obstacle grid is crucial while picking objects from a cabinet: it is a closed space where different objects are present, hence the motion planning must take into consideration the boundaries and generate a suitable trajectory. For these reasons, the occupancy grid generated is fed to the motion planners used.

8.1.2 SLAM problem

In order to correctly navigate the environment it is necessary to build a map of it and to localize the robot. To solve the SLAM problem during the real experiments, the RTAB-map used during simulations has been substituted with the Slam Toolbox. The main reason is that, since the first package uses pictures

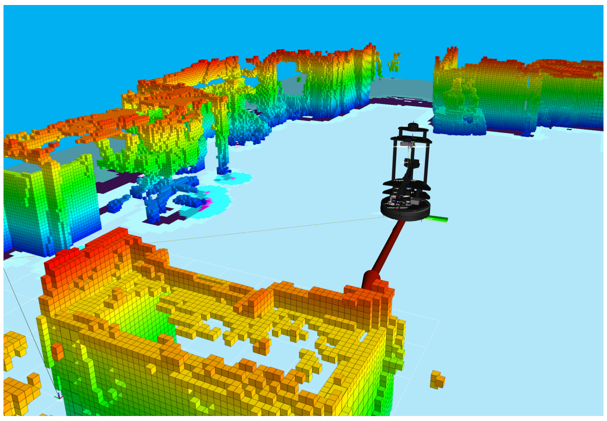
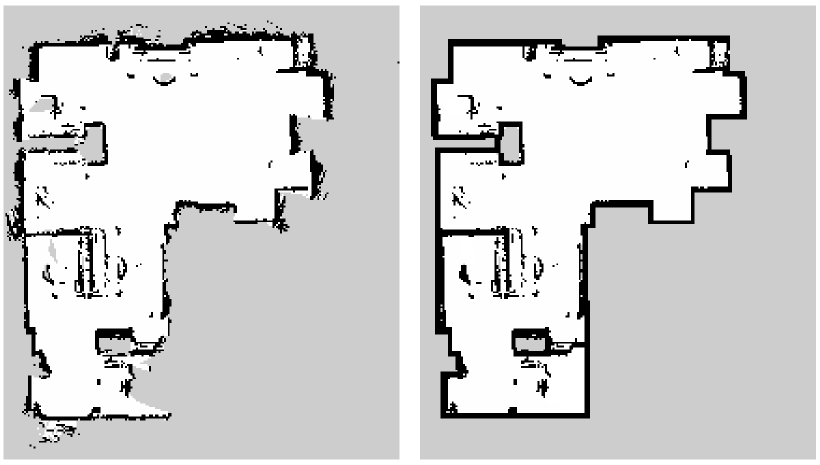


Figure 8.3: Occupancy grid of the real environment.

to build a map and localize the robot, the entire process resulted to be really slow. Moreover, the real camera is quite different from the simulated one, and this results in the presence of a lot of noise, especially for far points, that would require to visit the same locations several times to refine the map enough. In order to solve these problems, instead of using both the camera and the lidar, only the latter has been used by the Slam Toolbox. The Lidar appears, overall, more precise and it presents less noise, hence the map obtained appear to be cleaner; Figure 8.4a shows the map used to navigate. Although the map obtained is more than suitable, it has been post-processed to delete all the noise (Figure 8.4b).



(a) **(b)**

Figure 8.4: Map of the real environment.

Every time a new run starts, the refined map is loaded and used to localize the LoCoBot. It is also important to point out that if new obstacles appear, or those that were present while creating the map have been removed, the map is updated accordingly, but these changes are not permanently stored.

Regarding the initial position of the robot with respect to the map, from Table 8.1 it is possible to understand that the Home Location coincides with the origin of the map. This aspect is related to the fact that when the robot is turned on the Slam Toolbox needs to know its starting location, that has been fixed to be coincident with the origin of the map frame. The created map could also be used by other robots, simply setting their correct starting positions.

8.2 Robots' descriptions

As previously explained, the URDF file contains the description of the robot in terms of joint positions and limits, link lengths, and defines the reference frame characterizing the robot. It is important to point out that the LoCoBot Extended Planner node works with a modified model of the robot, which includes the three fake joints added in order to have a plan for the mobile manipulator as a unique holistic system. On the contrary, the robot's movements are possible thanks to two different controllers realised by the manufacturer, one for the base and the other for the arm, hence it is not possible to directly execute the planned trajectory. To overcome this problem two descriptions of the same robot are loaded at runtime:

1. the vanilla description is used together with the controllers to correctly govern the entire system;
2. the extended one is used to let the LoCoBot Extended Planner node to find a suitable trajectory for the holistic system.

Thanks to these two descriptions it is possible to both control the robot using the already-developed controllers and request a plan to the MoveIt! planner, which is then split into two parts, one to be sent to the arm controller and the other to the one of the base. It is important to point out that a crucial step to let these two descriptions exist contemporary is the definition of two different namespaces. Here the *locobot* namespace refers to the virtual robot, the extended one, and the *locobot_real* contains all the parameters and topics belonging to the real robot.

8.2.1 Mobile Base space occupancy

Another important change made with respect to the settings used to run the simulations regards the geometric model representing the mobile base. Indeed, the cylinder modelling the base is used by:

1. the mobile_base package to generate the inflation layers on which the global and local costmaps needed by the navigation stack are generated,
2. the LoCoBot Extended Planner to find a suitable collision-free trajectory.

Using the default values of height and radius, reported in Table 8.2 under the "*old*" column, the motion planner for the holistic system is able to find a collision-free trajectory, but the planned pose of the mobile base cannot be effectively reached. This is essentially due to the fact that the motion planner considers the occupancy grid to avoid collisions, but it may place the base right next to an obstacle, without considering a minimum distance from them; such a distance should correspond to the size of the inflation layer of the costmaps. As a result, it happens that the mobile base's goal position falls within the inflation layer itself, hence is considered as forbidden and unreachable by the navigation stack, leading to a movement failure.

In order to solve this problem, the extended robot description belonging to the *locobot* namespace has been further modified by changing the radius of the cylinder representing the collision region of the mobile base; the new parameters are reported in Table 8.2 under the "*new*" column. It is possible to notice that only the radius has been increased. Since the mobile_base package uses the robot description belonging to the *locobot_real* namespace, coming from the default and unchanged URDF file, the inflation layers have a default size corresponding to the normal space occupancy of the mobile base. On the contrary, the motion planner used for the entire mobile manipulator uses the extended robot description, thus the one with the augmented mobile base's space occupancy: in this way, the collision-free trajectory planned places the mobile base right next to an obstacle, but since the collision shape is larger, the position of the real mobile base will be far from the obstacle, outside the inflation layer. Figure 8.5 shows the comparison between the occupancy space of the mobile base coming from the default robot description (8.5a) and the augmented one (8.5b).

Parameter	Old Value [m]	New Value [m]
height	0.10938	0.10938
radius	0.178	0.3

Table 8.2: Parameters of the mobile base's space occupancy.

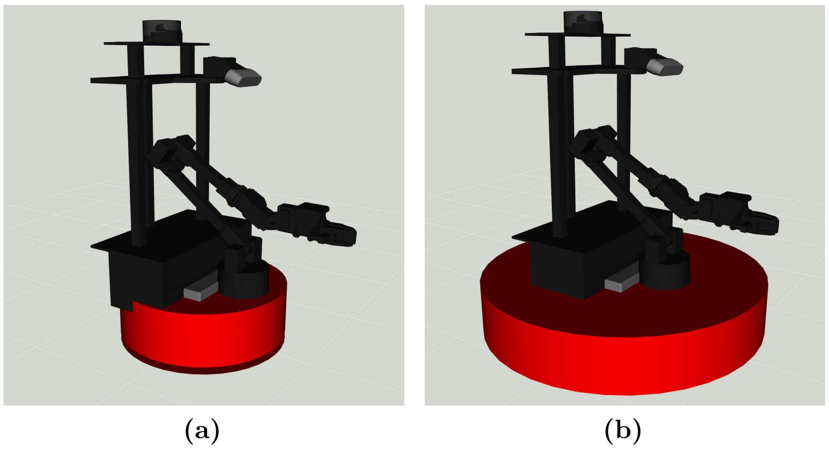


Figure 8.5: Comparison between the default mobile base collision region (a) and the augmented one (b).

8.3 Searching phase

As soon as the human sends a request to find a specific object, the search starts. Here the focus is to point out the main aspects of this phase tested on the real robot, and to comment the results obtained. Figure 8.6 shows the position of the known locations, while Figures from 8.7 to 8.12 depict all the steps executed by the LoCoBot while searching for an object. Details are provided, as usual, directly in the captions.

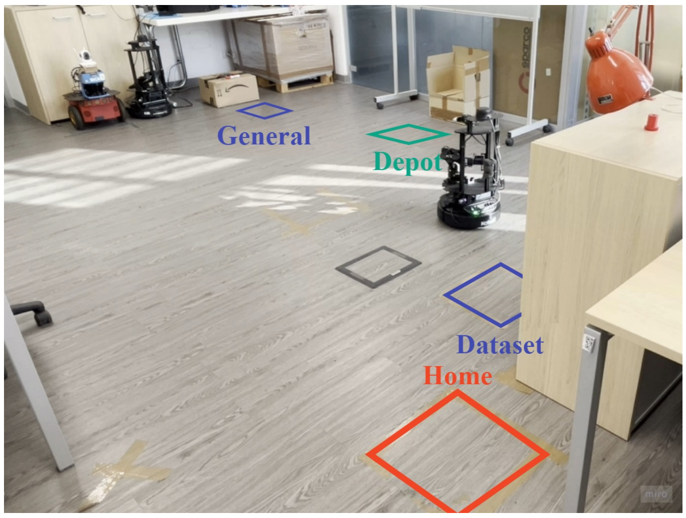


Figure 8.6: View of the known locations in the laboratory environment.

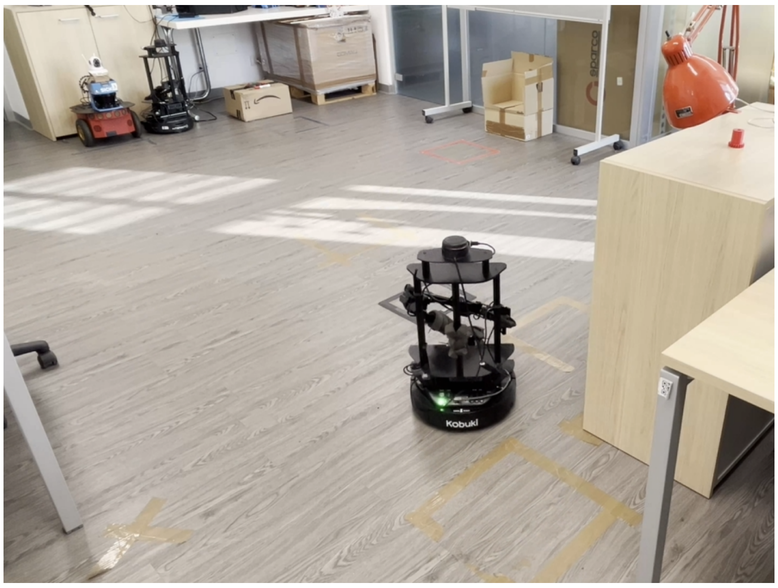


Figure 8.7: After the Communication Manager receives a request, it extracts the first known location where it is possible to find the requested item. In this case, the first place is the Dataset Location, hence its coordinates are extracted and the control is demanded by the Base Handler that computes a path to reach the target.

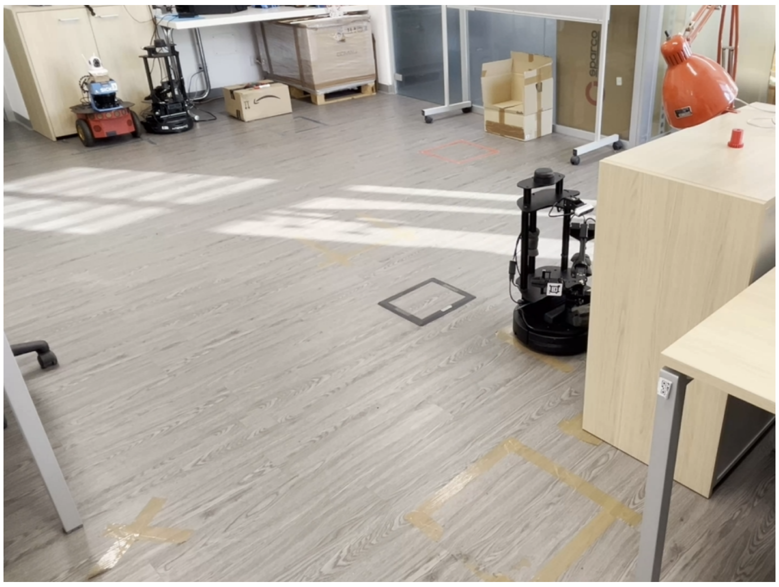


Figure 8.8: The LoCoBot reaches the target location. Here the object detection task is executed, involving the Grasping Point Generator node to compute the predictions using the trained YOLOv5 network.

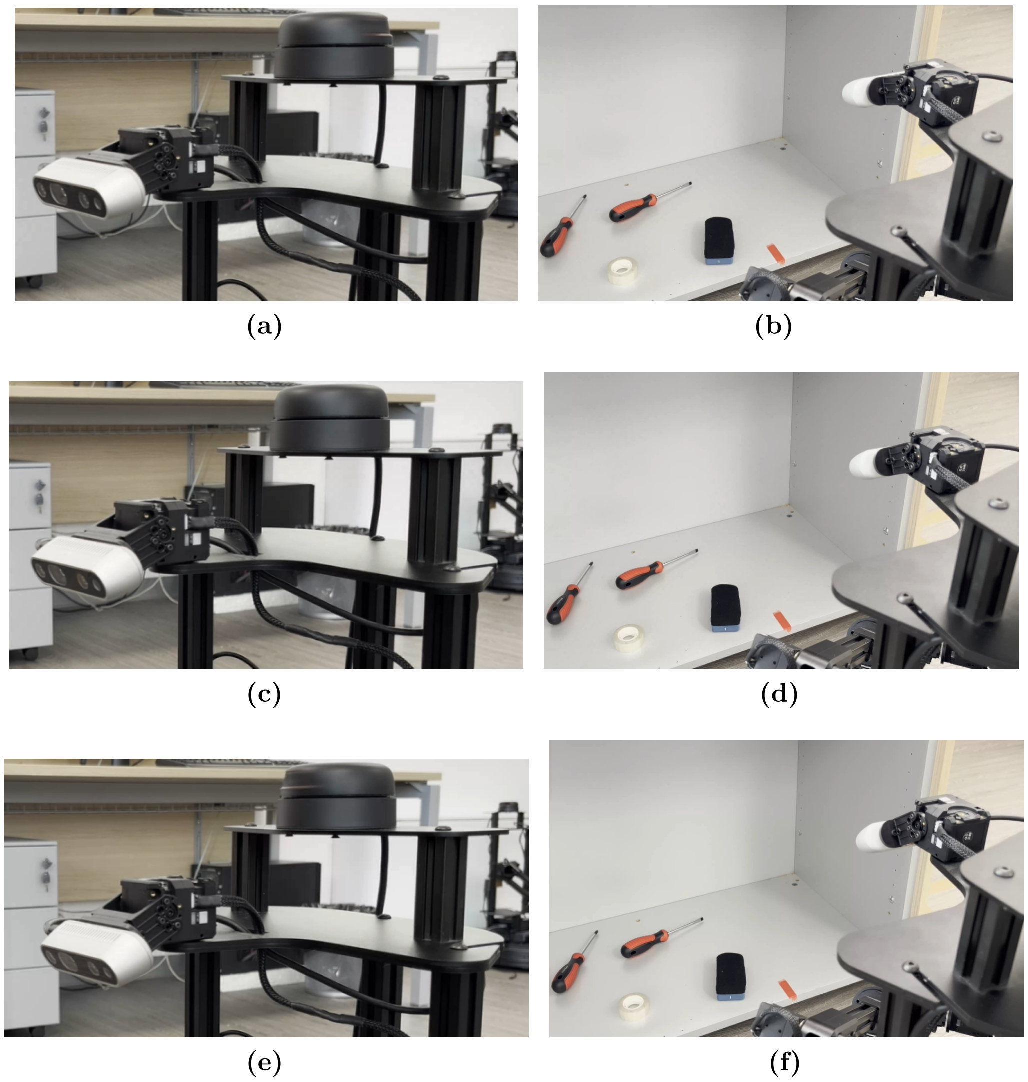


Figure 8.9: Front and rear views of the camera movements during the searching phase. In order to show the complete searching phase, the requested item is a 3D-printed red gear that has been removed from any of the corresponding known locations.

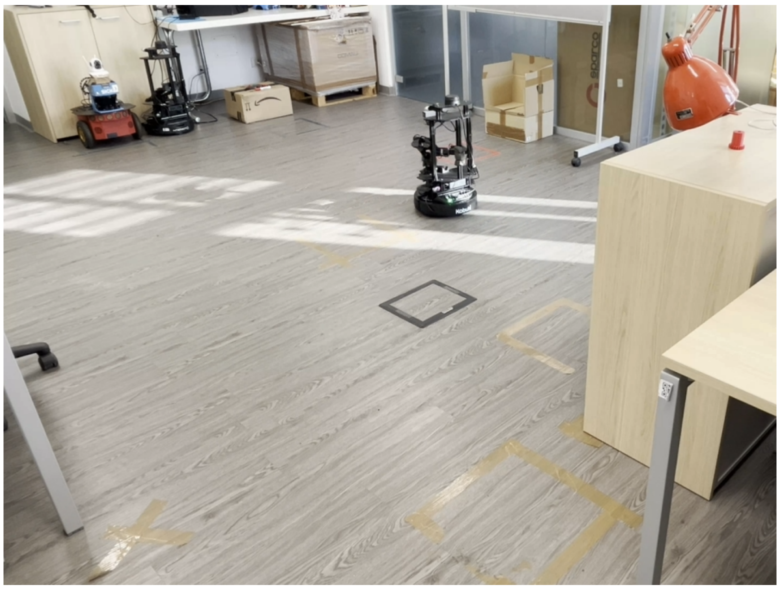


Figure 8.10: The requested object has not been found, The communication Manager extracts the second known location, that is the General Location, and its coordinates, and sends them again to the Mobile Handler.

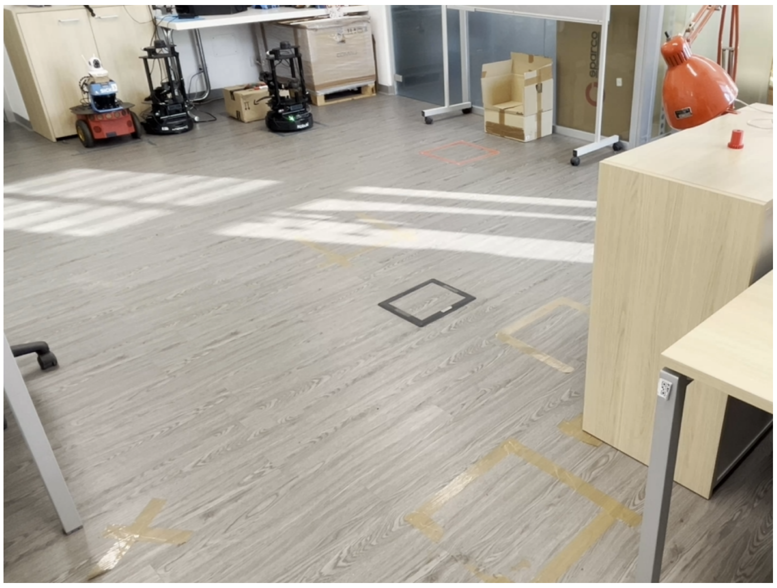


Figure 8.11: The General Location is reached and the object detection pipeline starts again.

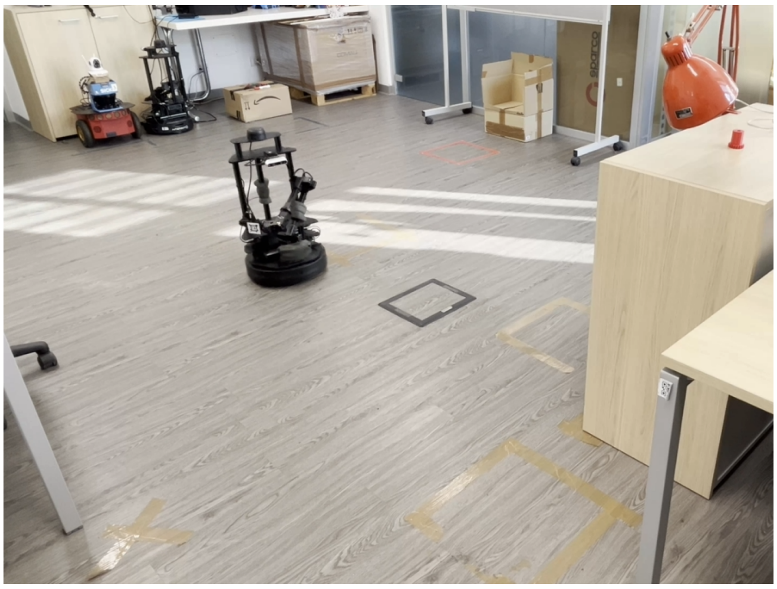


Figure 8.12: The target item was not found either in the last known point. The Communication Manager recognizes the termination condition of the searching phase and requests a path to return to the Home Position.

The entire searching phase shown in the figures is executed in about **1 minute**, and the major contribution is given by the base movements needed to go from one place to another. Regarding the object detection, 4 different camera positions are considered, with a *tilt* angle that goes from 0.3 *rad* to 0.6 *rad*, with a step equal to 0.1 *rad*, and each prediction requires about **0.6 seconds**. It is possible to notice a slight difference with respect to the time needed to compute a prediction during simulations, and the main reason is that more nodes run during the real experiment, due to the presence of two different descriptions of the same robot. What is important to notice is that the trained YOLO v5 network is used on a distributed system, where no GPUs are present, highlighting the lightweight of this CNN.

8.4 Pick-and-place pipeline

The second part of the tests performed on the real robot regards the pick-and-place pipeline. The objective of this part was to understand the real speed of the entire pipeline and the limitations of the grasping actions. Two different planners belonging to the OMPL are used:

1. the RRTstar planner is used to obtain a complete trajectory considering the mobile manipulator as a holistic system. The number of attempts available was set to 5, while the maximum planning time was set to 30 seconds;
2. the RRTConnect algorithm is instead used by the Arm Handler to find suitable trajectories for the arm only. In this case, the total number of planning attempts was 5, while the maximum time to find a solution was set to 10 seconds.

The values of the settings reported above are based on some experiments conducted before having the possibility to test the entire behaviour of the robot. Figures from 8.13 to 8.22 show the entire pick-and-place pipeline, with details in the captions.

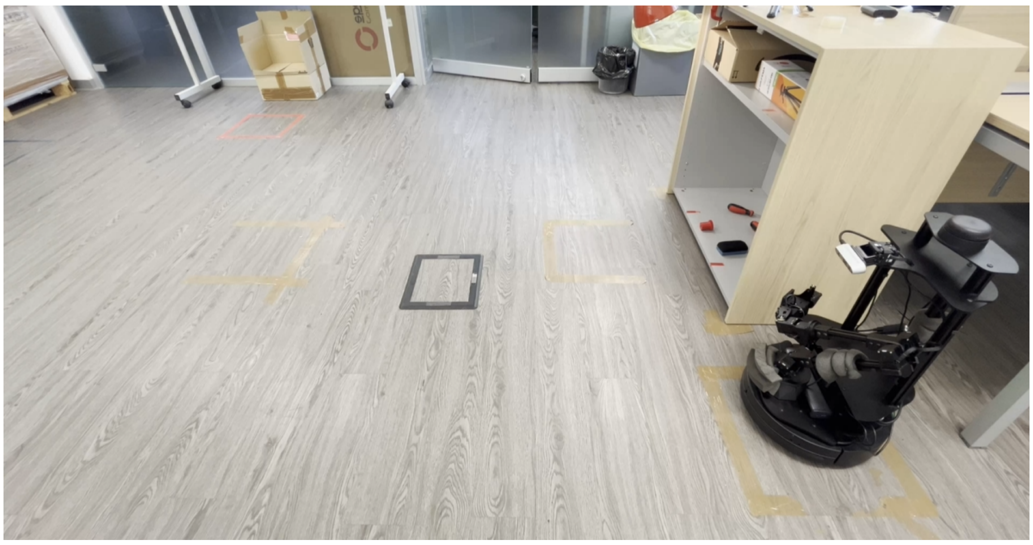


Figure 8.13: The LoCoBot always starts from the Home Position, right next to the desk of the human it is assisting. It is possible to notice how the arm is retracted in order to not exceed the collision shape of the mobile base, thus not generating unexpected collisions.

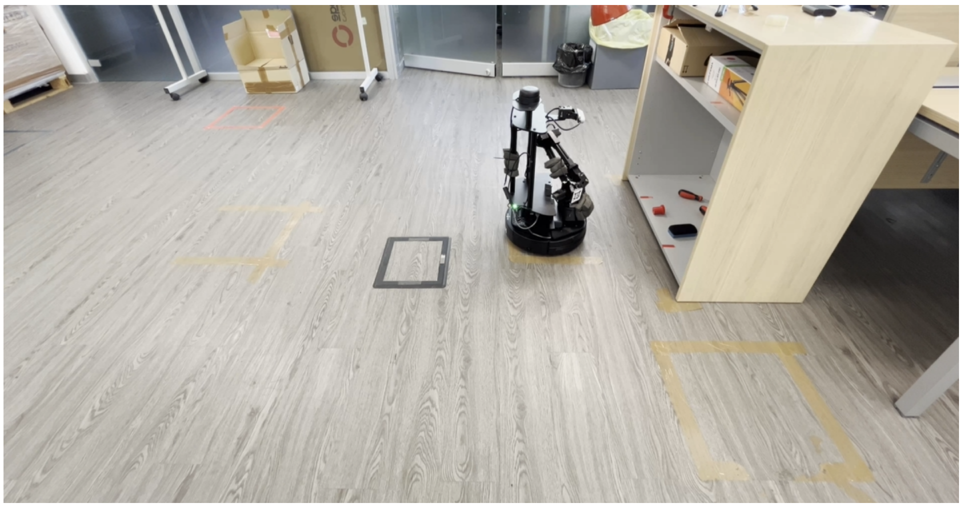


Figure 8.14: After a request is received, the searching phase starts, and the mobile manipulator goes to the first known location. Here the Object detection phase starts, and if the requested item is present in the current location, the LoCoBot Extended Planner node is called to find a suitable trajectory to reach the pre-grasp pose.

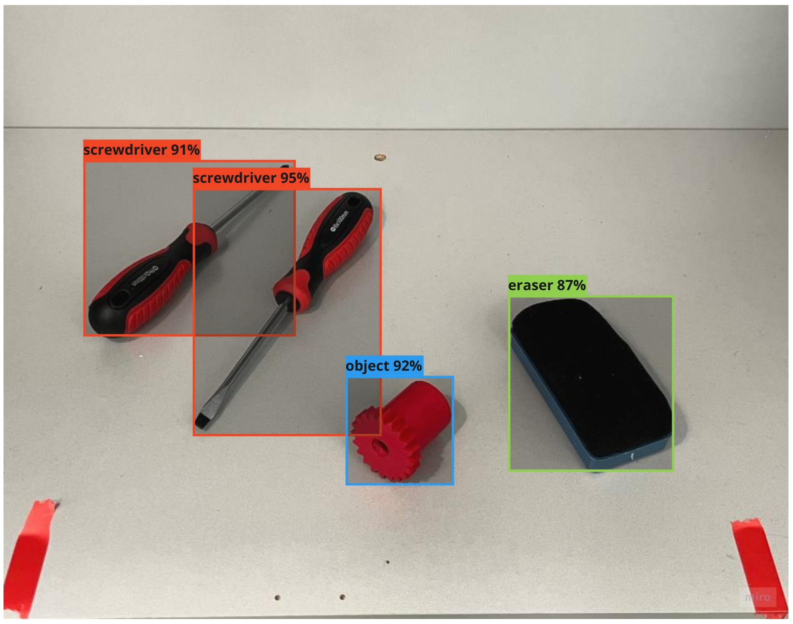


Figure 8.15: Result of the object detection task. It is possible to observe the bounding boxes computed and the predictions' confidence.

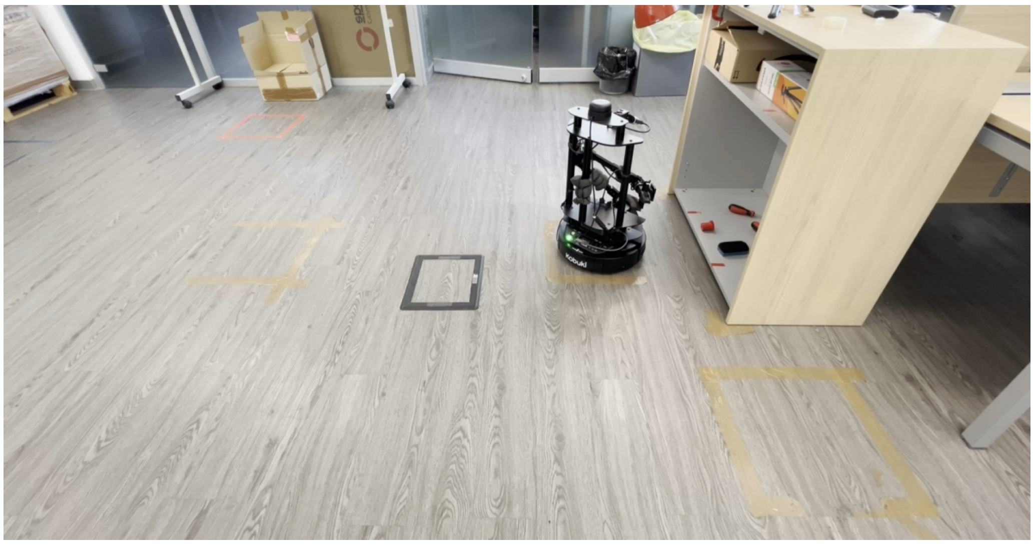


Figure 8.16: After the planner has found a suitable trajectory, the mobile base final pose is extracted and assigned to the Base Handler node. As a result, the mobile base is moved to a different position with respect to the one where the object detection pipeline started.

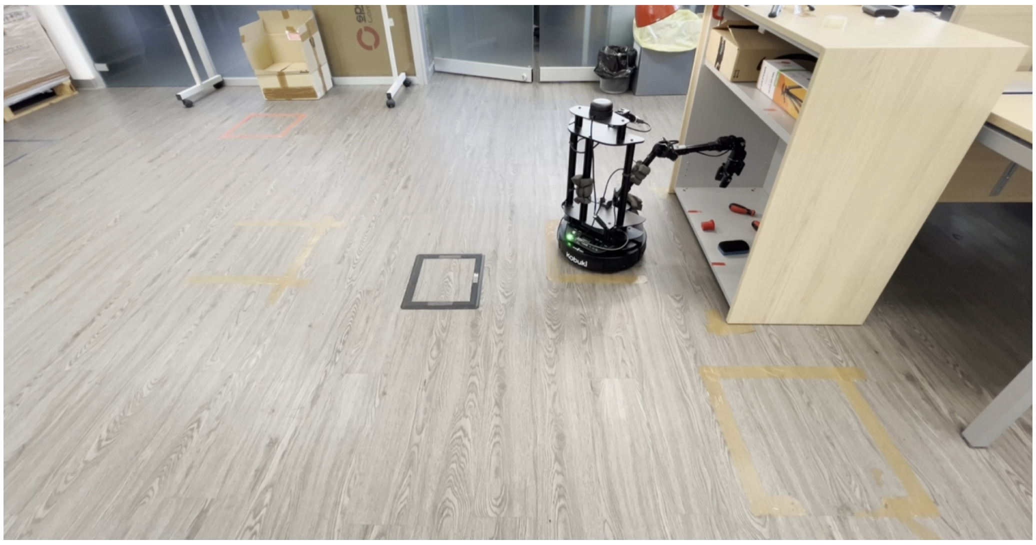


Figure 8.17: After the mobile base has reached the correct position, the trajectory computed for the arm is executed and the pre-grasp pose is reached. In this figure, it is possible to see the pre-grasp pose right above the object and the gripper open.

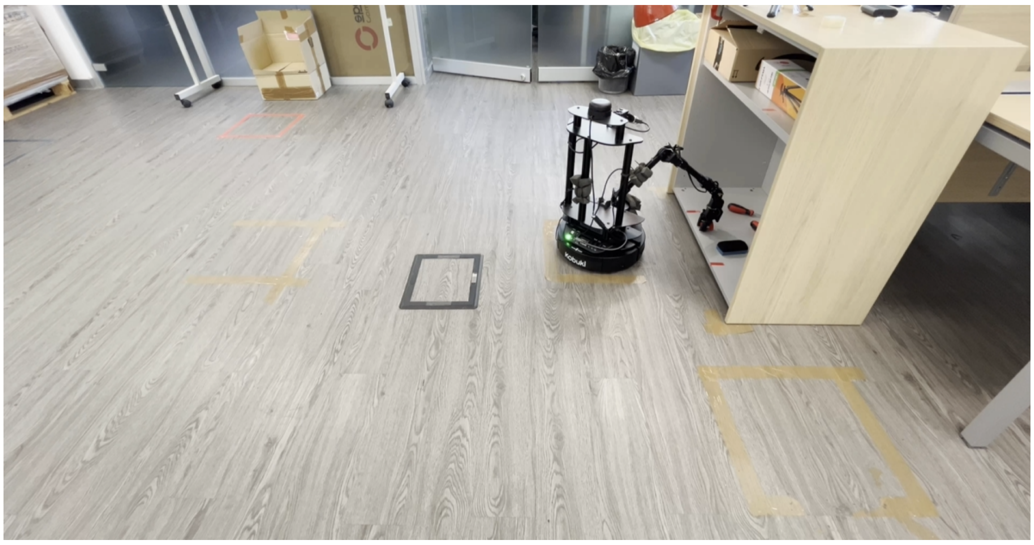


Figure 8.18: Once the pre-grasp pose is reached, the Communication Manager requests to the Arm Handler a plan to reach the grasp pose. When in this pose, the gripper is closed to grasp the object.

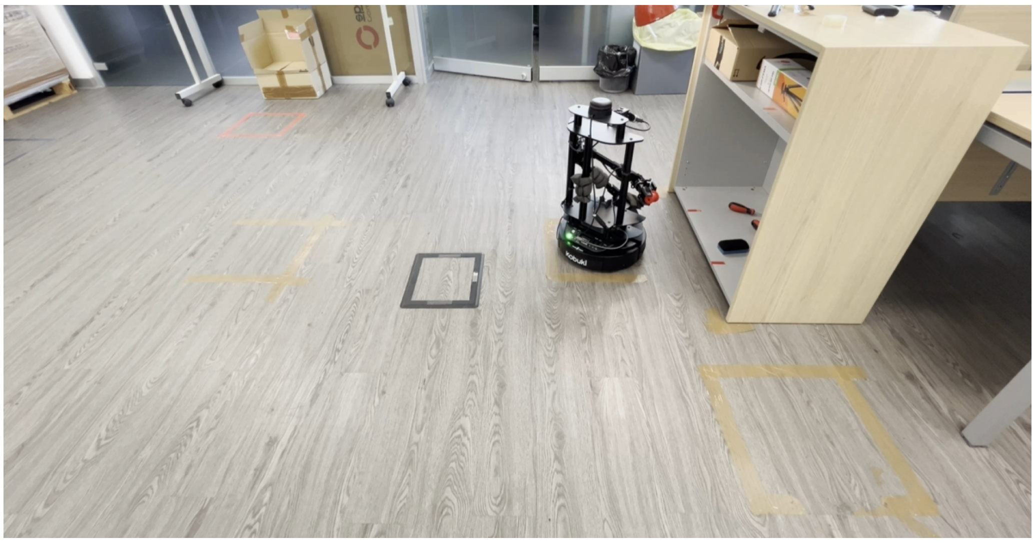


Figure 8.19: After the object has been grasped, the Arm Handler plans a trajectory to go back to the arm_default_pose. This latter pose reduces any risk of collision with the humans, ensuring their safety.

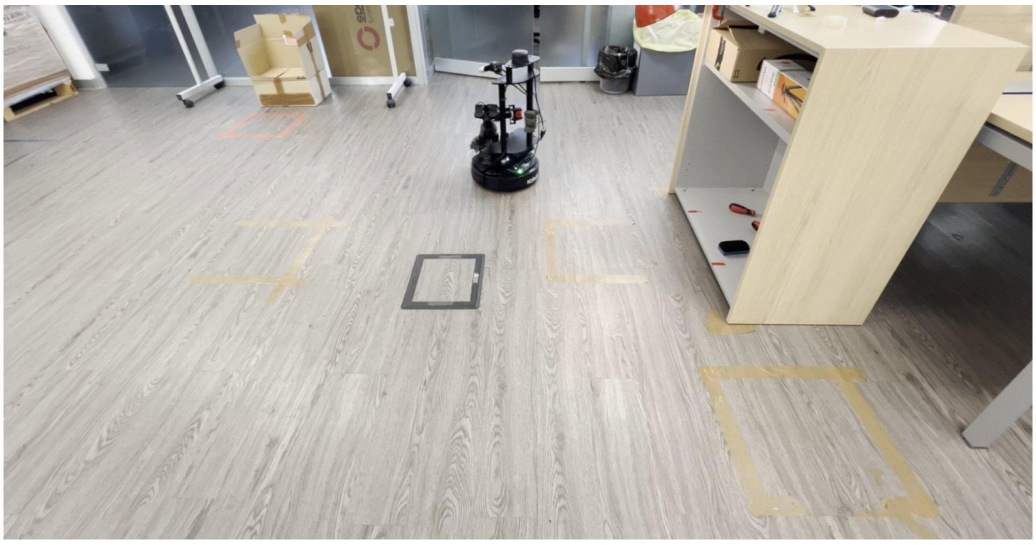


Figure 8.20: After the success of the grasp action, the mobile manipulator approaches the depot location to release the object.

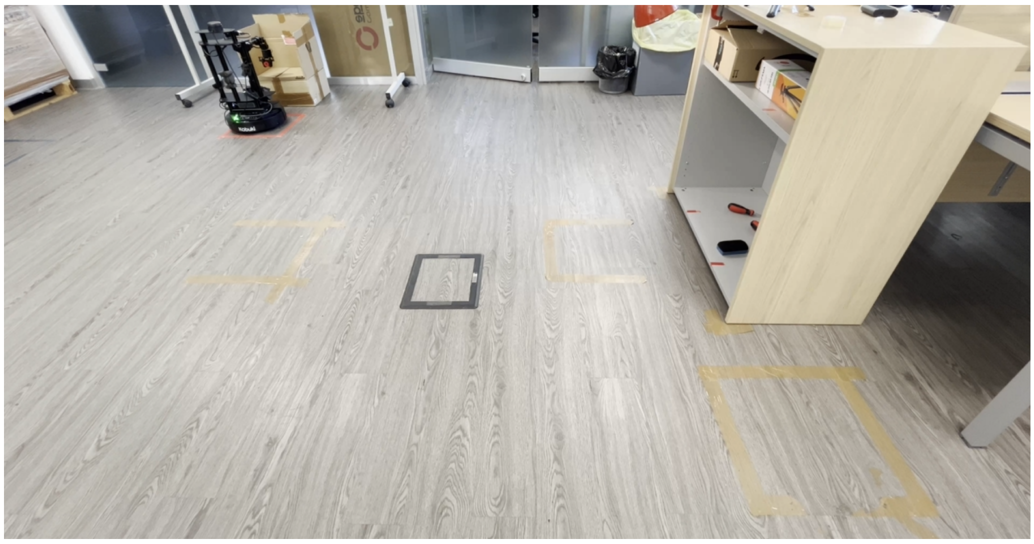


Figure 8.21: Once there, the robotic arm reaches the placing position and the gripper is opened.

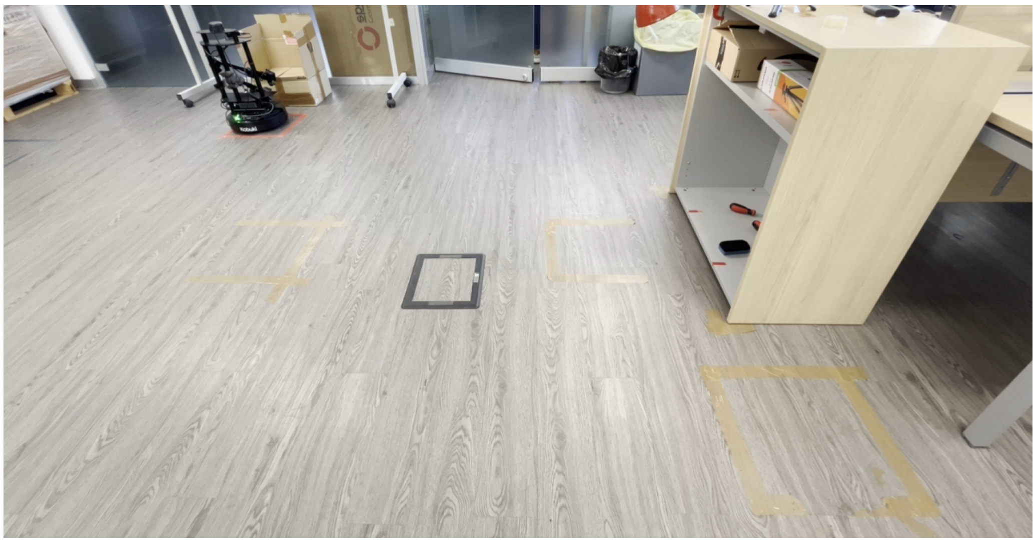


Figure 8.22: As last step, the arm goes back to the arm_secure_pose and the LoCoBot will return to its Home position.

The entire pipeline is executed in about **2.5 minutes**. Again, the main contributions is due to the navigation, while the object detection task appears to be negligible, although it is executed on a GPU-free system, and it is requested twice to mitigate the effect of the noise affecting the depth camera; the impacts of this task, considering also the delay introduced to reduce the noise effect, is of about **10 seconds** for 8 predictions (4 camera positions and predictions are requested twice). It is also important to point out that the maximum planning time given to the holistic planner has been decreased to **15 seconds**, since, if a trajectory is available, the RRTstar algorithm is able to find it in less than that. On the contrary, this algorithm uses the entire amount of planning time available, even if a suitable trajectory is found in few seconds. Differently from RRT*, the RRTConnect algorithm used by the Arm Handler to find paths related only to the manipulator stops searching as soon as a plan is found: in this way, the 10 seconds available to this planner are seldom totally used.

8.4.1 Dangerous part feedback

After the object detection phase and before the grasp action, if the requested item belongs to the hard-coded list of dangerous ones, the Dangerous Part Feedback Getter node takes into action. The user receives a picture representing the scene seen by the LoCoBot, together with an indication of the object that is going to be grasped and the computed grasping point. Figure 8.23 shows the image received by the human: it contains the bounding box of the requested item, and a red point indicating the grasping point computed.

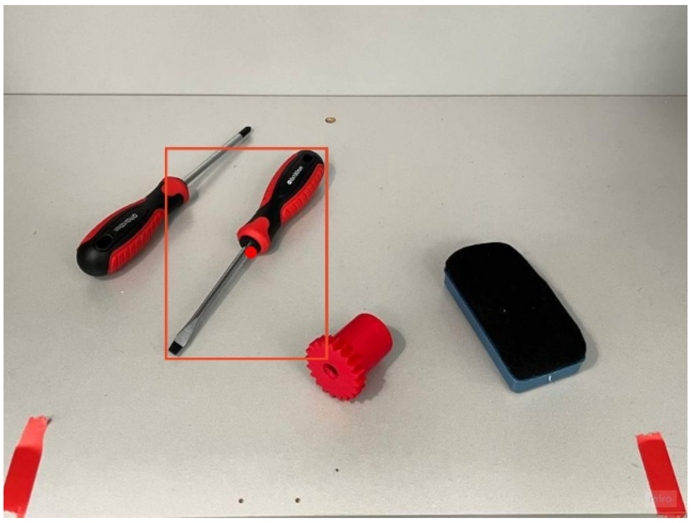


Figure 8.23: Interface through which the human operator can indicate the grasping point.

Once the image above appears, the person can click on the most dangerous point that should be grasped: the callback "get_feedback", realized using Python and the OpenCV library, saves the coordinates of the clicked point. As a result, the mean point between the predicted grasp point and the one indicated by the human is computed (Figure 8.24), and will be used as an effective grasping point.

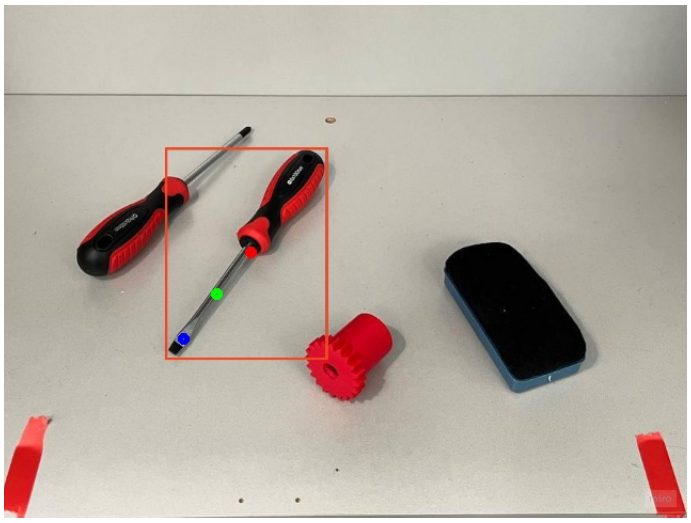


Figure 8.24: Visualization of the point clicked by the user (blue) and the real grasping point that will be used (green).

This interface is intuitive and easy to use, and allows to correctly handle dangerous objects. Regarding the grasping performances, the properties of the used gripper lead to some major issues with some objects in the dataset: long objects with the center of mass located near to one end of the object, like the screwdrivers, are seldom successfully grasped, while others with a regular shape and the center of mass near to the real grasping point are picked correctly most of the times.

Chapter 9

Conclusions and Future works

This thesis work gave me the possibility to explore the state-of-the-art regarding mobile manipulators and the robotic research fields in general, to deepen the knowledge of what ROS is and how it works, and which are the packages that perfectly suit a specific need. In addition, the need to create a custom dataset and to select, train and use a custom CNN gave me the possibility to further deepen my knowledge about AI and its applications. Problem-solving skills have been honed, particularly regarding how a robot should think in order to correctly behave in environments shared with humans, and which actions it should take to emulate them.

The objective of this thesis was to develop a software architecture which aims at improving the overall performance of a mobile manipulator, in order to make it more autonomous and flexible, and to make it able to behave as a smart assistant for humans. In order to achieve these objectives, the code structure has been organized as a communication graph, where each node is a ROS one and it is in charge of controlling a specific task or part of the robot. In addition, such an architecture allows to easily expand or modify the robot's behaviour without having to make heavy changes. It is also oriented towards a distributed settings, allowing to use the different computational resources present on the different computers, a characteristic that has been already exploited to ease the computations demanded by the on-board PC and to use a CNN for object detection. This latter feature gave the robot more autonomy and flexibility, making it able to recognize different objects in cluttered scenes, without having to mark them with specific labels. Moreover, the use of the reduced (small) version of YOLOv5 leads to predictions that are done in less than a second, speeding up the recognition process. To better exploit the manipulability space of a mobile manipulator, a holistic planner has been realized and used: this allows to obtain motion plans leading the gripper to poses that would not be reachable keeping the arm's origin frame fixed. The entire pick-and-place pipeline benefits from this planner, since less recovery behaviours and base movements are required. In addition, it is no longer needed to fix a base's pose from which the grasping task starts; instead, it is possible to define a location from where the robot starts searching for a given object, and if it is present in the current scene, the holistic planner will automatically find a suitable position for the mobile base from where the grasping task can be accomplished.

Working with the LoCoBot, some limitations that may regard also other mobile manipulators have been detected. First of all, the performances and the computational resources of the on-board PC are crucial: although some problems can be overcome by exploiting the distributed settings provided by ROS, some tasks must be executed on the on-board system. Controllers and data coming from sensors must be handled by that hardware, hence low performances may limit the entire process. A further limitation of the robot that has been used is the RGB-D camera it mounts, especially the depth sensor: it is affected by a lot of noise, forcing to take several depth frames before obtaining a reliable one. Regarding the mobile base and the arm's movements, they result to be highly inaccurate, leading to have to perform several attempts before executing a successful grasp. For what concerns the grasping abilities of the robot, besides the inaccurate positioning of the gripper itself, the parallel fingers system presents lots of limitations: too thin objects are not grasped correctly, and items with a non-uniform mass distribution tend to fall down.

As future works, the overall performance of the robot can be improved by adding some recovery behaviours, especially when a pick task fails. Regarding the dangerous part handling, to use a human in the loop is a first step towards the training of the robot using IML algorithms, making it completely autonomous also in this task. To further improve the grasping results, besides changing the parallel fingers gripper with a three fingers one or even a humanoid hand, it would be possible to use the point cloud coming from the depth camera to further refine the 3D pose estimation of an object, and use this information to train a NN able to generate the best grasping point that also considers the orientation of the object, and not only its position. Finally, considering the mobile manipulator as a holistic system for the motion plan resulted to be really useful to exploit the total abilities of this type of robot, but some advancements can be made in this direction, by designing a custom planner for mobile manipulators.

Bibliography

[1] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Modellistica, Pianificazione E Controllo.* The Mcgraw-Hill, 2008.

[2] Comau. *Racer Robotic Manipulator from Comau.* URL: `https://www.comau.com/it/competencies/robotics-automation/robot-team/racer-7-1-4/`.

[3] Martin Hägele, Klas Nilsson, J Norberto Pires, and Rainer Bischoff. «Industrial robotics». In: *Springer handbook of robotics* (2016), pp. 1385–1422.

[4] J. Sankari and R. Imtiaz. «Automated guided vehicle(AGV) for industrial sector». In: *2016 10th International Conference on Intelligent Systems and Control (ISCO).* 2016, pp. 1–5. DOI: `10.1109/ISCO.2016.7726962`.

[5] B. Stouten and A.J. de Graaf. «Cooperative transportation of a large object - development of an industrial application». In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004.* Vol. 3. 2004, 2450–2455 Vol.3. DOI: `10.1109/ROBOT.2004.1307428`.

[6] Amazon. *Scout Delivery Robot.* URL: `https://www.aboutamazon.com/news/transportation/meet-scout`.

[7] Boston Dynamics. *Spot Legged Robot from Boston Dynamics.* URL: `https://bostondynamics.com/products/spot`.

[8] Juan Angel Gonzalez-Aguirre, Ricardo Osorio-Oliveros, Karen L Rodrıguez-Hernández, Javier Lizárraga-Iturralde, Rubén Morales Menendez, Ricardo A Ramırez-Mendoza, Mauricio Adolfo Ramırez-Moreno, and Jorge de Jesús Lozoya-Santos. «Service robots: Trends and technology». In: *Applied Sciences* 11.22 (2021), p. 10702.

[9] Robotnik. *RB-KAIROS+ MOBILE MANIPULATOR.* URL: `https://robotnik.eu/products/mobile-manipulators/rb-kairos/`.

[10] Sanjoy Das, Indrani Das, Rabindra Nath Shaw, and Ankush Ghosh. «Advance machine learning and artificial intelligence applications in service robot». In: *Artificial Intelligence for Future Generation Robotics.* Elsevier, 2021, pp. 83–91.

[11] Jerry Alan Fails and Dan R Olsen Jr. «Interactive machine learning». In: *Proceedings of the 8th international conference on Intelligent user interfaces.* 2003, pp. 39–45.

[12] Mayank Mittal, David Hoeller, Farbod Farshidian, Marco Hutter, and Animesh Garg. «Articulated object interaction in unknown scenes with whole-body mobile manipulation». In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE. 2022, pp. 1647–1654.

[13] Luigi Tagliavini, Lorenzo Baglieri, Giovanni Colucci, Andrea Botta, Carmen Visconte, and Giuseppe Quaglia. «DOT PAQUITOP, an Autonomous Mobile Manipulator for Hospital Assistance». In: *Electronics* 12.2 (2023), p. 268.

[14] Kinova. *Kinova Gen3 Lite robotic arm.* URL: `https://www.kinovarobotics.com/product/gen3-lite-robots`.

[15] Thushara Sandakalum and Marcelo H Ang Jr. «Motion planning for mobile manipulators—a systematic review». In: *Machines* 10.2 (2022), p. 97.

[16] Siddhant Saoji and Jan Rosell. «Flexibly configuring task and motion planning problems for mobile manipulators». In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA).* Vol. 1. 2020, pp. 1285–1288. DOI: `10.1109/ETFA46521.2020.9212086`.

[17] Yugen You et al. «Design and Implementation of Mobile Manipulator System». In: *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER).* 2019, pp. 113–118. DOI: `10.1109/CYBER46603.2019.9066594`.

[18] Adeel Javaid. «Understanding Dijkstra's algorithm». In: *Available at SSRN 2340905* (2013).

[19] Nicola Castaman, Enrico Pagello, Emanuele Menegatti, and Alberto Pretto. «Receding horizon task and motion planning in changing environments». In: *Robotics and Autonomous Systems* 145 (2021), p. 103863.

[20] Heiko Engemann, Shengzhi Du, Stephan Kallweit, Patrick Cönen, and Harshal Dawar. «Omnivil—an autonomous mobile manipulator for flexible production». In: *Sensors* 20.24 (2020), p. 7249.

[21] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. «Path Planning with Modified a Star Algorithm for a Mobile Robot». In: *Procedia Engineering* 96 (2014). Modelling of Mechanical and Mechatronic Systems, pp. 59–69. ISSN: 1877-7058. DOI: `https://doi.org/10.1016/j.proeng.2014.12.098`. URL: `https://www.sciencedirect.com/science/article/pii/S187770581403149X`.

[22] Alireza Rastegarpanah, Hector Cruz Gonzalez, and Rustam Stolkin. «Semi-Autonomous Behaviour Tree-Based Framework for Sorting Electric Vehicle Batteries Components». In: *Robotics* 10.2 (2021). ISSN: 2218-6581. URL: `https://www.mdpi.com/2218-6581/10/2/82`.

[23] Da Hu, Hai Zhong, Shuai Li, Jindong Tan, and Qiang He. «Segmenting areas of potential contamination for adaptive robotic disinfection in built environments». In: *Building and Environment* 184 (2020), p. 107226. ISSN: 0360-1323. DOI: `https://doi.org/10.1016/j.buildenv.2020.107226`. URL: `https://www.sciencedirect.com/science/article/pii/S0360132320305977`.

[24] Sachin Chitta, E. Gil Jones, Matei Ciocarlie, and Kaijen Hsiao. «Mobile Manipulation in Unstructured Environments: Perception, Planning, and Execution». In: *IEEE Robotics Automation Magazine* 19.2 (2012), pp. 58–71. DOI: `10.1109/MRA.2012.2191995`.

[25] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. «ARA*: Anytime A* with Provable Bounds on Sub-Optimality». In: *Proceedings of (NeurIPS) Neural Information Processing Systems*. Dec. 2003, pp. 767–774.

[26] Lieping Zhang, Xiaoxu Shi, Yameng Yi, Liu Tang, Jiansheng Peng, and Jianchu Zou. «Mobile Robot Path Planning Algorithm Based on RRT_-Connect». In: *Electronics* 12.11 (2023). ISSN: 2079-9292. URL: `https://www.mdpi.com/2079-9292/12/11/2456`.

[27] Steven M. LaValle. «Rapidly-exploring random trees : a new tool for path planning». In: *The annual research report* (1998). URL: `https://api.semanticscholar.org/CorpusID:14744621`.

[28] Sertac Karaman and Emilio Frazzoli. «Sampling-based algorithms for optimal motion planning». In: *The international journal of robotics research* 30.7 (2011), pp. 846–894.

[29] John Schulman et al. «Motion planning with sequential convex optimization and convex collision checking». In: *The International Journal of Robotics Research* 33.9 (2014), pp. 1251–1270.

[30] Dmitry Berenson, James Kuffner, and Howie Choset. «An optimization approach to planning for mobile manipulation». In: *2008 IEEE International Conference on Robotics and Automation*. 2008, pp. 1187–1192. DOI: `10.1109/ROBOT.2008.4543365`.

[31] Kyshalee Vazquez-Santiago, Chun Fan Goh, and Kenji Shimada. «Motion Planning for Kinematically Redundant Mobile Manipulators with Genetic Algorithm, Pose Interpolation, and Inverse Kinematics». In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. 2021, pp. 1167–1174. DOI: `10.1109/CASE49439.2021.9551546`.

[32] Ming Yao, Haigang Deng, Xianying Feng, Peigang Li, Yanfei Li, and Haiyang Liu. «Global Path Planning for Differential Drive Mobile Robots Based on Improved BSGA* Algorithm». In: *Applied Sciences* 13.20 (2023). ISSN: 2076-3417. DOI: `10.3390/app132011290`. URL: `https://www.mdpi.com/2076-3417/13/20/11290`.

[33] Ander Iriondo, Elena Lazkano, Loreto Susperregi, Julen Urain, Ane Fernandez, and Jorge Molina. «Pick and Place Operations in Logistics Using a Mobile Manipulator Controlled with Deep Reinforcement Learning». In: *Applied Sciences* 9.2 (2019). ISSN: 2076-3417. DOI: `10.3390/app9020348`. URL: `https://www.mdpi.com/2076-3417/9/2/348`.

[34] Shuxin Xie, Lining Sun, Zhenhua Wang, and Guodong Chen. «A speedup method for solving the inverse kinematics problem of robotic manipulators». In: *International Journal of Advanced Robotic Systems* 19.3 (2022), p. 17298806221104602.

[35] Hasan Danaci, Luong A. Nguyen, Thomas L. Harman, and Miguel Pagan. «Inverse Kinematics for Serial Robot Manipulators by Particle Swarm Optimization and POSIX Threads Implementation». In: *Applied Sciences* 13.7 (2023). ISSN: 2076-3417. DOI: `10.3390/app13074515`. URL: `https://www.mdpi.com/2076-3417/13/7/4515`.

[36] Rongrong Liu, Florent Nageotte, Philippe Zanne, Michel de Mathelin, and Birgitta Dresp-Langley. «Deep reinforcement learning for the control of robotic manipulation: a focussed mini-review». In: *Robotics* 10.1 (2021), p. 22.

[37] Lin Jiang, Baiyan Liu, Liangcai Zeng, Xinyuan Chen, Jie Zhao, and Jihong Yan. «Research on the omni-directional mobile manipulator motion planning based on improved genetic algorithm». In: *2009 IEEE International Conference on Automation and Logistics*. 2009, pp. 1921–1926. DOI: `10.1109/ICAL.2009.5262620`.

[38] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. «The Open Motion Planning Library». In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012). `https://ompl.kavrakilab.org`, pp. 72–82. DOI: `10.1109/MRA.2012.2205651`.

[39] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. «STOMP: Stochastic trajectory optimization for motion planning». In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 4569–4574. DOI: `10.1109/ICRA.2011.5980280`.

[40] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. «CHOMP: Gradient optimization techniques for efficient motion planning». In: *2009 IEEE International Conference on Robotics and Automation*. 2009, pp. 489–494. DOI: `10.1109/ROBOT.2009.5152817`.

[41] Jianbo Su and Wenlong Xie. «Motion planning and coordination for robot systems based on representation space». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 41.1 (2010), pp. 248–259.

[42] Keping Liu, Jilei Sui, Ning Yue, and Shuaishi Liu. «Path planning method of mobile manipulator based on the representation space». In: *2016 IEEE International Conference on Mechatronics and Automation*. 2016, pp. 322–326. DOI: 10.1109/ICMA.2016.7558582.

[43] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. «Probabilistic roadmaps for path planning in high-dimensional configuration spaces». In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[44] D. Hsu, J.-C. Latombe, and R. Motwani. «Path planning in expansive configuration spaces». In: *Proceedings of International Conference on Robotics and Automation*. Vol. 3. 1997, 2719–2726 vol.3. DOI: 10.1109/ROBOT.1997.619371.

[45] James Ward and Jayantha Katupitiya. «Mobile Manipulator Motion Planning Towards Multiple Goal Configurations». In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2006, pp. 2283–2288. DOI: 10.1109/IROS.2006.282633.

[46] Ruben Seyboldt, Christian Frese, and Angelika Zube. «Sampling-based Path Planning to Cartesian Goal Positions for a Mobile Manipulator Exploiting Kinematic Redundancy». In: *Proceedings of ISR 2016: 47st International Symposium on Robotics*. 2016, pp. 1–9.

[47] Jesse Haviland, Niko Sünderhauf, and Peter Corke. «A holistic approach to reactive mobile manipulation». In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 3122–3129.

[48] Syed Sahil Abbas Zaidi, Mohammad Samar Ansari, Asra Aslam, Nadia Kanwal, Mamoona Asghar, and Brian Lee. «A survey of modern deep learning based object detection models». In: *Digital Signal Processing* 126 (2022), p. 103514. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2022.103514. URL: https://www.sciencedirect.com/science/article/pii/S1051200422001312.

[49] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. «Rich feature hierarchies for accurate object detection and semantic segmentation». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[50] Ross Girshick. «Fast R-CNN». In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 1440–1448. DOI: 10.1109/ICCV.2015.169.

[51] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. «Mask r-cnn». In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 2961–2969.

[52] Weikuan Jia, Yuyu Tian, Rong Luo, Zhonghua Zhang, Jian Lian, and Yuanjie Zheng. «Detection and segmentation of overlapped fruits based on optimized mask R-CNN application in apple harvesting robot». In: *Computers and Electronics in Agriculture* 172 (2020), p. 105380. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2020.105380. URL: https://www.sciencedirect.com/science/article/pii/S0168169919326274.

[53] Yipeng Yang, Zhaoting Li, Xinghu Yu, Zhan Li, and Huijun Gao. «A trajectory planning method for robot scanning system uuuusing mask R-CNN for scanning objects with unknown model». In: *Neurocomputing* 404 (2020), pp. 329–339. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2020.04.059. URL: https://www.sciencedirect.com/science/article/pii/S0925231220306263.

[54] Michael Danielczuk, Matthew Matl, Saurabh Gupta, Andrew Li, Andrew Lee, Jeffrey Mahler, and Ken Goldberg. «Segmenting unknown 3D objects from real depth images using mask R-CNN trained on synthetic point clouds». In: *arXiv preprint arXiv:1809.05825* 16 (2018).

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[56] Zeyad Farisi et al. «Motion Control System of IoT Intelligent Robot Based on Improved ResNet Model». In: *Journal of Sensors* 2023 (2023).

[57] Karen Simonyan and Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». In: *arXiv preprint arXiv:1409.1556* (2014).

[58] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You only look once: Unified, real-time object detection». In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 779–788.

[59] Afdhal Afdhal, Khairun Saddami, Sugiarto Sugiarto, Zahrul Fuadi, and Nasaruddin Nasaruddin. «Real-Time Object Detection Performance of YOLOv8 Models for Self-Driving Cars in a Mixed Traffic Environment». In: *2023 2nd International Conference on Computer System, Information Technology, and Electrical Engineering (COSITE).* 2023, pp. 260–265. DOI: 10.1109/COSITE60233.2023.10249521.

[60] Huanlong Liu, Dafa Li, Bin Jiang, Jianyi Zhou, Tao Wei, and Xinliang Yao. «MGBM-YOLO: a faster light-weight object detection model for robotic grasping of bolster spring based on image-based visual servoing». In: *Journal of Intelligent & Robotic Systems* 104.4 (2022), p. 77.

[61] Caner Sahin, Guillermo Garcia-Hernando, Juil Sock, and Tae-Kyun Kim. «A review on object pose recovery: From 3D bounding box detectors to full 6D pose estimators». In: *Image and Vision Computing* 96 (2020), p. 103898. ISSN: 0262-8856. DOI: `https://doi.org/10.1016/j.imavis.2020.103898`. URL: `https://www.sciencedirect.com/science/article/pii/S0262885620300305`.

[62] Douglas Morrison, Peter Corke, and Jürgen Leitner. «Closing the loop for robotic grasping: A real-time, generative grasp synthesis approach». In: *arXiv preprint arXiv:1804.05172* (2018).

[63] Yuhang Yang, Wei Zhai, Hongchen Luo, Yang Cao, Jiebo Luo, and Zheng-Jun Zha. «Grounding 3D Object Affordance from 2D Interactions in Images». In: *arXiv preprint arXiv:2303.10437* (2023).

[64] Zhanpeng He, Nikhil Chavan-Dafle, Jinwook Huh, Shuran Song, and Volkan Isler. «Pick2Place: Task-aware 6DoF Grasp Estimation via Object-Centric Perspective Affordance». In: *arXiv preprint arXiv:2304.04100* (2023).

[65] Harish Ravichandar, Athanasios S Polydoros, Sonia Chernova, and Aude Billard. «Recent advances in robot learning from demonstration». In: *Annual review of control, robotics, and autonomous systems* 3 (2020), pp. 297–330.

[66] *ROS homepage.* URL: `https://www.ros.org/`.

[67] *Gazebo homepage.* URL: `https://gazebosim.org/home`.

[68] rviz. *Home page of the rviz ROS package wiki.* URL: `http://wiki.ros.org/rviz`.

[69] MoveIt! *Example of the Rviz graphic interface used to interact with a robotic arm with MoveIt!* URL: `https://ros-planning.github.io/moveit_tutorials/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html`.

[70] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. «OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees». In: *Autonomous Robots* (2013). Software available at `https://octomap.github.io`. DOI: `10.1007/s10514-012-9321-0`. URL: `https://octomap.github.io`.

[71] RTAB-Map. *Homepage of the RTAB-Map ROS package wiki.* URL: `http://wiki.ros.org/rtabmap_ros`.

[72] Navstack. *Homepage of the Navstack ROS package wiki.* URL: `http://wiki.ros.org/navigation`.

[73] costmap_2d. *Homepage of the costmap_2d ROS package wiki.* URL: `http://wiki.ros.org/costmap_2d`.

[74] global_planner. *Homepage of the global_planner ROS package wiki.* URL: http://wiki.ros.org/global_planner.

[75] teb_local_planner. *Homepage of the teb_local_planner ROS package wiki.* URL: http://wiki.ros.org/teb_local_planner.

[76] MoveIt! *Homepage of the MoveIt! ROS package wiki.* URL: http://moveit.ros.org.

[77] MoveIt! *Visualization of the rviz interface for the MoveIt! package.* URL: https://moveit.ros.org/assets/images/screens/3d_visualizer.png.

[78] MoveIt! *Visualization of the main concepts regarding the MoveIt! package.* URL: https://moveit.ros.org/documentation/concepts/.

[79] move_base. *Homepage of the move_base ROS package wiki.* URL: http://wiki.ros.org/move_base.

[80] Trossen Robotics. *Specification page of the LoCoBot WX250s.* URL: https://docs.trossenrobotics.com/interbotix_xslocobots_docs/index.html.

[81] Trossen Robotics. *WidowX 250 6-DOF Robotic Arm mounted on the LoCoBot WX250s.* URL: https://www.trossenrobotics.com/widowx-250-robot-arm-6dof.aspx.

[82] Intel. *Intel RealSense D435 Depth Camera.* URL: https://www.intelrealsense.com/depth-camera-d435/.

[83] Itseez. *Open Source Computer Vision Library.* https://github.com/itseez/opencv. 2015.

[84] Glenn Jocher. *Ultralytics YOLOv5.* Version 7.0. 2020. DOI: 10.5281/zenodo.3908559. URL: https://github.com/ultralytics/yolov5.

[85] Renjie Xu, Haifeng Lin, Kangjie Lu, Lin Cao, and Yunfei Liu. «A Forest Fire Detection System Based on Ensemble Learning». In: *Forests* 12 (Feb. 2021), p. 217. DOI: 10.3390/f12020217.

[86] Roboflow. *Roboflow website home page.* URL: https://roboflow.com/.

Voglio esprimere la mia gratitudine alla Professoressa Marina Indri e ai PhD Fiorella Sibona e Pangcheng David Cen Cheng, per avermi dato questa opportunità e per avermi seguito lungo tutto il percorso.

Ringrazio tutti i colleghi che mi sono stati accanto e che hanno alleviato le interminabili ore di lezione di questi anni.

Ringrazio tutti i ragazzi di HKN che ho incontrato in questi ultimi anni per la conoscenza e le esperienze condivise.

Ringrazio Edoardo, Michele, Tano, Manuel, Davide, Claudio, Luca, Mattia, Giorgia, Lorenza e tutti i miei amici, per tutte le risate, le uscite, le feste e la sangria che hanno accompagnato questo percorso.

Voglio ringraziare Lorenzo e Ilaria per il loro supporto, la loro ispirazione, i consigli e le serate passate insieme.

Ringrazio Valeria, per essermi sempre vicina, per spronarmi, supportarmi e spingermi a dare il massimo sempre, anche nei momenti più bui.

Ringrazio la mia famiglia tutta, anche coloro che guardano da lassù, per aver sempre creduto in me e per essere stati i miei punti di riferimento.

Un ringraziamento speciale va a mia sorella, i miei genitori e i miei nonni, miei pilastri, per avermi cresciuto, educato e per avermi insegnato ciò che è davvero importante nella vita.

E infine, ringrazio me stesso, per tutti i sacrifici fatti, per la determinazione, la costanza e la passione dimostrati e che mi hanno portato fino a questo punto.