POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Development of a multi-environment platform composed by a robot and an autonomous guided vehicle

Supervisor

Prof. Marina INDRI

Candidate

Antonio Paolo PASSARO

Supervisor at Comau S.p.A.

Ing. Stefano PESCE

October 2020

Acknowledgements

Alla mia famiglia

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, robotics has spread to many areas with various types of applications and is increasingly present both in everyday life and in the companies. At the beginning, robots were used only in factories for the execution of repetitive jobs and tasks to replace human operators, today instead thanks to technological evolution and artificial intelligence they are able to interact with people, helping and supporting them in daily life [1]. In fact, they are widely used in the form of household appliances, tools and collaborative assistants and assume different appearances depending on the task and the field of application [1]. In addition to their use in almost all sectors nowadays, they are increasingly present in companies and industry in the form of manipulators, mobile robots and mobile manipulators thanks to their characteristics of versatility, adaptability and accuracy in the execution of the tasks. Within the industries, robots are widely used as programmable manipulators to perform well-defined tasks such as the handling of parts and materials, assembly of mechanical/electronic parts, welding, painting or cutting [2]. In addition to the common manipulators, automated guided vehicles (AGVs) and mobile robots are also very popular within automation processes. The automated guided vehicles allow the movement of pieces from one area to another within the environment following pre-established paths. Mobile robots, on the other hand, represent an evolution of AGVs thanks to their ability to move autonomously both in static and dynamic environments with or without the presence of obstacles. The AGVs execute the automatic transport from one area to another by means of a cable guide or of a laser guide based on laser sources and reflectors inside the environment in such a way as to follow predetermined paths [3]; on the contrary, mobile robots are completely autonomous and all sensors and devices are integrated on board. In general they are equipped with laser sensors, cameras and proximity/distance sensors in order to acquire information on the surrounding environment and execute the algorithms that determine the movement based on different localization and navigation techniques. Today, AGVs are widely used but are less flexible than

the mobile robots because it is necessary to introduce changes and infrastructures within the environment in which they operate [3].

Today, however, the concept of Industry 4.0 is increasingly common. Industry 4.0 is based on the concept of having fully automated systems that are composed of interconnected machines and robotic systems that interact with each other by exchanging information and performing diagnostic and maintenance functions [4][5]. The goal therefore is to have totally independent industries and production plants (smart factories) in order to independently identify critical issues within the production process and adopt the best strategies. Furthermore, in these types of automated systems it is possible to adapt the production to individual customers in such a way as to create different products that satisfy the different characteristics and needs of the customer [5][6]. Another feature of industry 4.0 is that of having production plants with the lowest possible energy consumption thanks to the use of efficient and intelligent control systems that allow to reduce waste [6].

The goal of this project was to create an autonomous system that manages the communication between an anthropomorphic robot and an autonomous guided vehicle (AGV) in order to perform the coordinated movement of both the robots and execute a specific pick and place task. The integration of anthropomorphic robots with an autonomous guided vehicle is very close to the concept of Industry 4.0 with the goal to create a system with an independent communication between several devices without the presence of human beings. The anthropomorphic robot considered in the system is the robot e.DO. e.DO is a modular educational robot developed by Comau S.p.A composed by six joints and a spherical wrist. It is equipped with a raspberry pi controller based on ROS that manages the motion with an open source control logic. Comau provides also a very easy to use Android application that allows the users to connect to the robot and executes the desired trajectory, and a lot of resources to support the users in developing their projects. The e.DO robot was designed as an educational robot for teaching robotics and programming and for performing small automated tasks defined by users. In this thesis, however, its integration with other external electronic devices and other robotic systems within the same environment has been studied. Since the AGV was not available during the development of the project it has been performed only a gazebo simulation of the entire system with a differential drive mobile robot as collaborative assistant for e.DO in order to have a complete proof of concept. The task considered in this project consists of releasing an object grasped by e.DO inside one of the two boxes positioned on the base of the AGV, which represent two different containers for the objects. In particular, to coordinate the movement between the two robots, a camera and a laser rangefinder have been used. The camera was exploited in order to correct the trajectory of the AGV and to identify the point where to release the object, while the laser scanner has been used to measure the AGV distance from e.DO and to check its position. The whole system

is managed by an external raspberry pi controller that communicates with the robot and with the autonomous guided vehicle. In this way it is possible to manage more electrical devices on the same platform in order to move the robotic arm combined with the AGV. In particular it has been developed a specific ROS package, which runs on the raspberry pi controller, that manages the two robots and synchronizes all the movements in order to accomplish the task. The package developed in fact allows to interact with the ROS package edo_core provided by Comau which runs on the e.DO raspberry and with the Autonomous Guided Vehicle. The thesis work was developed in collaboration with Comau S.p.A.

The second chapter of the thesis is a background chapter, where the main topics covered in the thesis work are presented. The main concepts of robotics are explained considering both the position kinematics and the differential kinematics of a manipulator and a brief introduction to mobile robots is provided. In addition, the characteristics and the different types of sensors used in robotic systems were illustrated with particular reference to vision sensors and the various visual servoing techniques. The third chapter describes the robot e.DO from its software and hardware structure point of view and shows the study of the position direct kinematics and the relative singularities. The fourth chapter instead describes the whole project development process. Starting from a general description of the system, the communication methods adopted to interact with the robots are illustrated, as well as the different approaches adopted for the execution of the task and the techniques used for the vision system. Finally, the development of the ROS package is described, highlighting the software features of which it is composed. The fifth chapter describes the simulation of the system in Gazebo. In particular, it describes how the Gazebo simulation environment works, the definition of the models of the two robots for the simulation and the main concepts of the Moveit platform used for trajectory planning. At the end the simulation of the system and the obtained results are shown. The sixth chapter shows the real implementation of the developed package on the external raspberry pi, in particular the results of the experimental tests obtained by testing the part of the architecture related to the movement of e.DO. Finally, the last chapter proposes possible future extensions of the developed work and some possible improvements.

Chapter 2

Background

2.1 Robotics

A robot is a machine that interacts with the surrounding environment by performing various actions. It can be seen as a system composed of a mechanical structure that includes mechanical arms, tools and/or wheels and mechanisms capable of moving the whole physical structure such as motors [2]. In addition, a robot or a robotic system is also equipped with some sensors that allow to obtain information both on the status of the robot and on the external environment in order to regulate its movement. The robot control system manages the execution of all robot actions in order to perform predefined tasks. Robots are classified into two main categories:

- Manipulators
- Mobile Robots

A manipulator is a particular type of robot whose mechanical structure is made of a chain of rigid or flexible bodies called links connected by devices called joints that allow the movement of the manipulator. The mechanical structure is an open kinematic chain if the chain of links that connects the ends is unique, otherwise it is a close kinematic chain [2]. There are two types of joints:

1. revolute joint
2. prismatic joint

The revolute joint allows a rotational movement with respect to the two links while the prismatic one determines a translation. Each joint corresponds to a degree of freedom in an open kinematic chain. The working space of the robot is defined by the set of points in the space that the robot can reach with the end effector. A manipulator consists of an arm and a wrist. The manipulator arm is

made of the first three links of the mechanical structure which allow the positioning of the end effector in the space, while the wrist is formed by the last links which allow to define the orientation too [2]. In particular, there are different types of manipulators depending on the type of joints of the arm. In this thesis work the category of anthropomorphic manipulators has been considered. In fact, in these robots the arm is composed of three revolute joints in which the axis of the first one is perpendicular to the axes of the other two. For the wrist, on the other hand, the typical structure is the spherical one with three revolute joints whose axes intersect at one point [2].

2.1.1 Pose of a rigid body

The pose of a rigid body in the space is defined by the position and the orientation with respect to a reference frame. Considering the fixed reference frame O-xyz and the reference frame O'-x'y'z' integral with the rigid body shown in Figure 2.1:

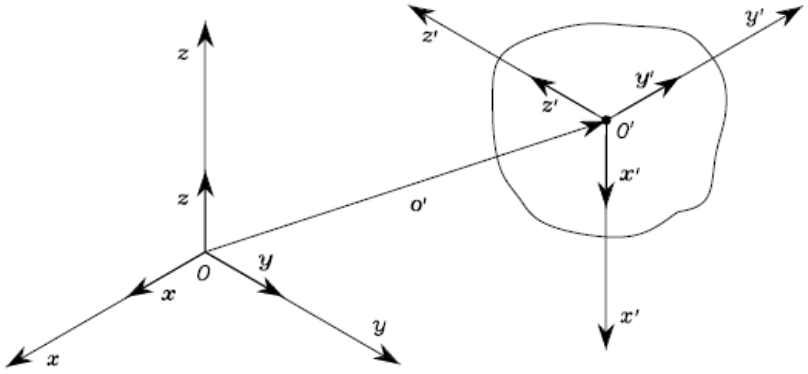

Figure 2.1: Position and orientation of a rigid body [2]

The position is defined by: $o^{'} = o_x^{'}x + o_y^{'}y + o_z^{'}z$ [2]

$$o^{'} = \begin{bmatrix} o_x^{'} \\ o_y^{'} \\ o_z^{'} \end{bmatrix} \tag{2.1}$$

while the orientation is defined by expressing the versors of the reference frame O'-x'y'z' with respect to the reference frame O-xyz:

$$\begin{aligned} x^{'} &= x^{'}_{x}x + x^{'}_{y}y + x^{'}_{z}z \\ y^{'} &= y^{'}_{x}x + y^{'}_{y}y + y^{'}_{z}z \\ z^{'} &= z^{'}_{x}x + z^{'}_{y}y + z^{'}_{z}z \end{aligned} \tag{2.2}$$

In general, given a fixed reference frame O-xyz and a reference frame O'-x'y'z' rotated with respect to O-xyz, it is possible to obtain the rotation matrix R:

$$R = \begin{bmatrix} x^{'}_{x} & y^{'}_{x} & z^{'}_{x} \\ x^{'}_{y} & y^{'}_{y} & z^{'}_{y} \\ x^{'}_{z} & y_{z} & z_{z} \end{bmatrix} \tag{2.3}$$

The rotation matrix represents the rotation around the origin of the two reference frame until they overlap [2]. Moreover, thanks to the rotation matrix it is also possible to transform the coordinates of a point P between the two reference frames that have the same origin [2]. Given the points p in O-xyz and p' in O'-x'y'z':

$$p = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}, p' = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} \tag{2.4}$$

$$p = Rp' \tag{2.5}$$

The main property of a rotation matrix is that it is an orthonormal matrix (and hence its transpose is equal to the inverse). The product of multiple rotation matrices represents the concatenation of multiple rotations. The product is defined by the post-multiplication of the rotation matrices if the individual rotations are defined with respect to the previous one [2]:

$$R = R_1 R_2 R_N \tag{2.6}$$

while it is defined by the pre-multiplication if the single rotations refer to the initial reference frame:

$$R = R_N R_2 R_1 \tag{2.7}$$

Given a fixed reference frame O_0 and a mobile reference frame O_l defined by a translation vector o_1^0 and by a rotation R_1^0 with respect to O_0 as shown in Figure 2.2:

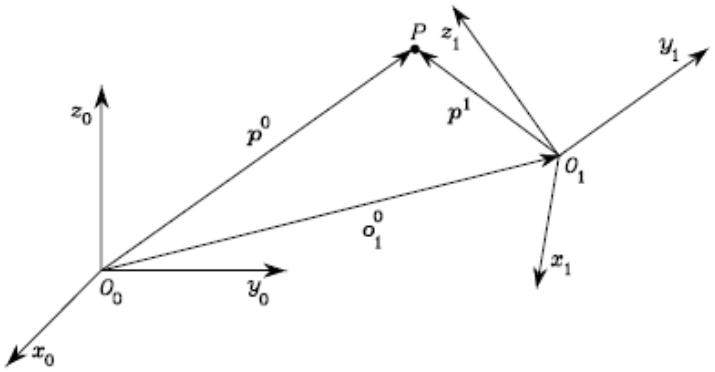

Figure 2.2: Fixed and mobile reference frames [2]

given p^0 and p^1 the vectors with respect to the point P in the two reference frames and $\tilde{p}^0$ and $\tilde{p}^1$ the corresponding homogeneous coordinates:

$$\tilde{p}^0 = \begin{bmatrix} p_x^0 \\ p_y^0 \\ p_z^0 \\ 1 \end{bmatrix}, \tilde{p}^1 = \begin{bmatrix} p_x^1 \\ p_y^1 \\ p_z^1 \\ 1 \end{bmatrix} \tag{2.8}$$

it is possible to define the overall transformation matrix T which describes the roto-translation of the local reference with respect to the fixed reference frame [2]:

$$T_1^0 = \begin{bmatrix} R_1^0 & o_1^0 \\ 0^T & 1 \end{bmatrix} \tag{2.9}$$

$$\tilde{p}^0 = T_1^0 \tilde{p}^1 \tag{2.10}$$

Compared to a rotation matrix, the transformation matrix T is not orthonormal and its inverse is defined by [2]:

$$(T_1^0)^{-1} = T_0^1 = \begin{bmatrix} R_0^1 & -R_0^1 o_1^0 \\ 0^T & 1 \end{bmatrix} \tag{2.11}$$

$$\tilde{p}^1 = T_0^1 \tilde{p}^0 \tag{2.12}$$

Furthermore, the same properties seen before for the rotation matrices are valid for the composition of multiple transformation matrices.

2.1.2 Positions Kinematics

The posture of the manipulator depends on the position and orientation of the links of the kinematic chain and is therefore determined by the vector of the joint variables q:

$$q = \begin{bmatrix} q_1 \\ q_2 \\ q_N \end{bmatrix} \tag{2.13}$$

The goal of the direct kinematics is to determine the position and orientation of the end-effector in the space respect to the position of the joints of the robot. In fact, by solving the direct kinematic it is possible to obtain the transformation matrix $T_p^0(q)$. Given R_0 the reference frame fixed on the base of the robot and R_p the reference frame related to the end-effector, the transformation matrix $T_p^0(q)$ represents R_p into the fixed reference frame R_0 [2].

$$T_p^0(q) = \begin{bmatrix} R_p^0(q) & t_p^0(q) \\ 0^T & 1 \end{bmatrix} \tag{2.14}$$

where $R_p^0(q)$ is the rotational matrix of R_p respect to R_0 and $t_p^0(q)$ is the vector that represents the position of the origin of R_p respect to R_0.

By defining R_p considering the directions of interest:

$$T_p^0(q) = \begin{bmatrix} n_p^0(q) & s_p^0(q) & a_p^0(q) & t_p^0(q) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

with $a_p^0(q), s_p^0(q)$ and $n_p^0(q)$ representing the versors k, j and i respectively.

Position and orientation of the end-effector can be also described in a more simple way by the vector p in the operational space [2]:

$$p = \begin{bmatrix} x1 \\ x2 \\ x3 \\ \alpha 1 \\ \alpha 2 \\ \alpha 3 \end{bmatrix} \tag{2.16}$$

with the position vector $x = \begin{bmatrix} x1x2x3 \end{bmatrix}$ and the vector $\alpha = \begin{bmatrix} \alpha 1\alpha 2\alpha 3 \end{bmatrix}$ with the parameters (angles) that define the orientation.

The transformation matrix $T_p^0(q)$ is obtained by the product of the transformation matrices between the single reference frames associated to each link of the robot. Considering a robot with six degrees of freedom and neglecting the end-effector, there are six links and hence six reference frames:

$$T_6^0(q) = T_1^0(q1) \cdot T_2^1(q2) \cdot T_3^2(q3) \cdot T_4^3(q3) \cdot T_5^4(q5) \cdot T_6^5(q6) \tag{2.17}$$

The Denavit-Hartenberg conventions allow to define the single local reference frames and to compute the transformation matrices thanks to the DH parameters: d_i, θ_i, a_i, α_i and x_i, Fig. 2.3. The parameters define position and orientation of all the reference frames with respect to the previous one. The parameters a_i and α_i are constant and depend on the physical links while one of the other two is variable, depending on the type of the joint. If the robot is composed by all revolute joints the variable parameter is θ_i [2].

Conventions

Given a robot with n joints, n+1 local reference frames R_i associated with each arm are defined. The joint g_i then connects the arm b_{i-1} with the arm b_i.

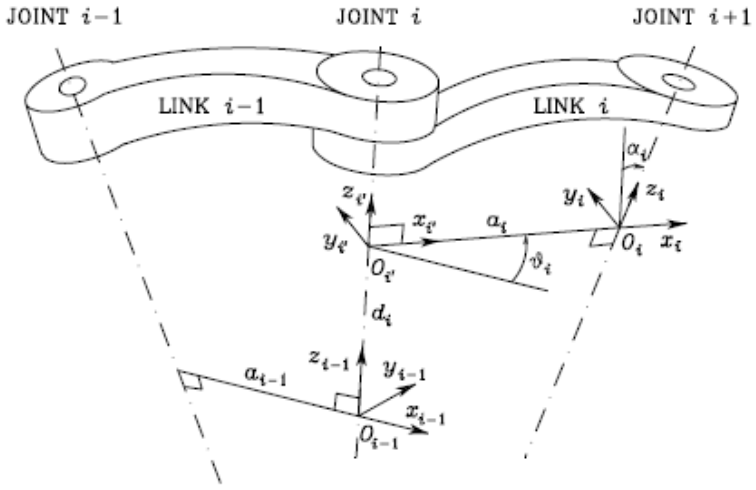

Figure 2.3: DH Conventions and parameters [2]

For the definition of the reference frame R_i are defined the following rules [2]:

- The axis z_i corresponds to the movement axis of the joint g_{i+1}
- The origin of the reference frame is defined at the intersection of z_i with the common normal of the axes z_{i-1} and z_i
- x_i is defined along the common normal of the axes z_{i-1} and z_i
- The axis y_i completes the base using the right hand rule

For the reference frame R_0 the origin and the x axis are chosen arbitrarily. Instead for the reference frame R_n, the axis x_n is normal to the axis z_{n-1} while the origin and the axis z_n are not defined in a unique way [2]. Furthermore, for the

definition of the DH parameters, it is defined an intermediate reference system $R_{i'}$ whose origin $O_{i'}$ is positioned at the intersection point of z_{i-1} with the common normal between the axes z_{i-1} and z_i. The axis $z_{i'}$ is defined as the axis z_{i-1} while $x_{i'}$ is defined as x_i [2]. Then the DH parameters are defined [2]:

- d_i: It is represented by the coordinate of $O_{i'}$ along the axis z_{i-1}
- θ_i: It represents the angle relative to the rotation around the axis $z_{i'}$ which causes x_{i-1} to coincide with x_i
- a_i: It is defined by the distance between O_i and $O_{i'}$
- α_i: It represents the angle relative to the rotation around the axis x_i which leads z_{i-1} to coincide with z_i

.

The final transformation between the two reference frames R_i and R_{i-1} is obtained by the composition of two transformation matrices [2]. The matrix $T_{i'}^{i-1}$ which describes the rototranslation of $R_{i'}$ with respect to R_{i-1}:

$$T_{i'}^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

and the matrix $T_i^{i'}$ which describes the rototranslation of R_i with respect to $R_{i'}$:

$$T_i^{i'} = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.19}$$

The final transformation matrix between R_i and R_{i-1} is then:

$$T_i^{i-1}(q_i) = T_{i'}^{i-1} T_i^{i'} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.20}$$

with $q_i = \theta_i$

The workspace is given by the region of the points in the space that the center of the end-effector reference frame can reach. It is given by the set of data points of the position vector $x = x(q)$ with $q_{i,m} <= q_i <= q_{i,M}$ (min and max values of the joints). The inverse kinematics, instead, has the aim of determining the joint variables q_i corresponding to a certain position in the space of the end-effector [2].

$$p, T_p^0(q) \rightarrow q \tag{2.21}$$

2.1.3 Differential kinematics

Differential kinematics concerns the relationship between the velocity of the joints and the velocity of the end-effector. In particular, there are two types of velocity of the end-effector in the operational space [2]:

1. geometric velocity
2. analytical velocity

The geometric velocity is described by the vector $v \in R_6$:

$$v = \begin{bmatrix} \dot{x}^T \\ \omega^T \end{bmatrix} \tag{2.22}$$

The vector v defines the linear velocity $\dot{x}$ and the angular velocity ω of the end-effector. The analytical velocity is defined by the vector $\dot{p} \in R_6$:

$$\dot{p} = \begin{bmatrix} \dot{x} \\ \dot{\alpha}^T \end{bmatrix} \tag{2.23}$$

$\dot{\alpha}^T$ is the time-derivative of the vector α of the angles used to represent the end-effector orientation. The two velocity vectors are related to the velocities of the joints through the relations [2]:

$$\begin{aligned} v &= J_g(q)\dot{q} \\ \dot{p} &= J_a(q)\dot{q} \end{aligned} \tag{2.24}$$

given:

1. $\dot{q}$: vector of the joints velocities
2. $J_g(q) \in R_6^n$: geometric Jacobian
3. $J_a(q) \in R_6^n$: analytical Jacobian

The geometric Jacobian is defined by [2]:

$$J_g(q) = \begin{bmatrix} J_{L,1} & J_{L,2} & J_{L,3} & J_{L,4} & J_{L,5} & J_{L,6} \\ J_{A,1} & J_{A,2} & J_{A,3} & J_{A,4} & J_{A,5} & J_{A,6} \end{bmatrix} \tag{2.25}$$

with:

- $J_{L,i}$: defines the influence of the i-th joint on the end effector linear velocity
- $J_{A,i}$: defines the influence of the i-th joint on the end effector angular velocity

The analytic Jacobian, instead, is obtained directly by computing the derivative with respect to time of the position vector p defined before in 2.16.

Singularities

The configurations of the manipulator for which the geometric Jacobian is not full rank are called singular configurations. These kinematics singularities can be obtained by solving the equation: $det(J_g(q_s)) = 0 \rightarrow q_s$ [2]. In these points some cartesian velocities cannot be obtained and the velocities at the joints can result in zero cartesian velocities. Furthermore, small velocities of the end-effector can be associated to large joint speeds in a neighborhood of a singular configuration [2]. Considering a manipulator with six degrees of freedom and with a spherical wrist, the geometric Jacobian can be also written as:

$$J_g(q) = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} \tag{2.26}$$

with $J_{ij} \in R^{3,3}$
By fixing the origin of the end-effector reference frame in the intersection of the wrist axes, it's obtained the decoupling of the singularities of the arm and of the wrist:

$$J_{12} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{2.27}$$

$$det(J_g) = det(J_{11})det(J_{22}) \tag{2.28}$$

The arm singolarities are obtained by solving the equation $det(J_{11}) = 0$ while solving $det(J_{22}) = 0$ are obtained the wrist singolarities.

2.1.4 Introduction to Mobile Robots

A mobile robot has the characteristic of moving independently in the surrounding environment in which it is located. The environment can include different obstacles which can be fixed/mobile and can be known or not. In order to move autonomously in the space, a mobile robot is equipped with a power supply system without cables (such as batteries), a control system that can be integrated or remote and various sensors/actuators that allow to acquire information and perform movements. The control system of the mobile robot typically consists of an embedded system integrated on the base that executes the control algorithms that allow the movement thanks to the information provided by the sensors. A mobile robot can be described as a rigid body composed by a locomotion system that allows the movement in the space. There are two main types of locomotion systems: the wheeled robots, characterized by a system of wheels, and the legged robots, characterized by a series of rigid bodies connected by joints that allow the movement [2]. The locomotion system based on wheels is the most used. In particular, there are different types of wheels [2]:

- Fixed: characterized by an horizontal rotation axis passing through the center and orthogonal to the plane of the wheel.
- Steering: characterized by an additional vertical rotation axis passing through the center with respect to the fixed ones, in order to change the orientation.
- Caster: they are similar to the steering wheels but the vertical axis doesn't pass through the center in order to maintain the direction of the motion.
- Omnidirectional: they are fixed wheels characterized by the presence of rollers on the final part whose rotation axis is inclined with respect to the wheel.

In general, regardless of the type of wheel, they can be active or passive and depending on how they are combined there are different types of kinematic structures [2]:

- Differential drive vehicle: it is equipped with two fixed active wheels characterized by the same rotation axis and a passive central caster wheel.
- Synchronized traction vehicle: it is equipped with three steering wheels having the same orientation which are actuated by two motors, one for the traction and one for the orientation.
- Tricycle: it consists of two axes: a rear axis with two fixed wheels and a front axis with a steering wheel. The traction is determined by the two fixed wheels while the motor that drives the front wheel defines the orientation.

- Automobile: it is similar to the tricycle with the difference that the front axis consists of two steering wheels.

Figure 2.4 shows the structure of a differential drive vehicle.

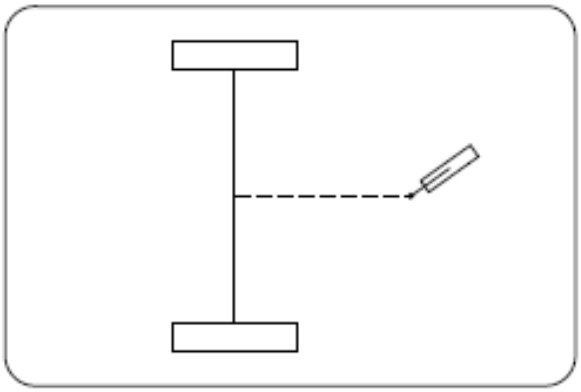

Figure 2.4: Differential drive vehicle [2]

2.2 Sensors and visual servoing

Sensors represent a fundamental part for a robot and for robotic systems in general as thanks to them it is possible to acquire different types of information necessary to perform the tasks of the robot. In particular, there are two macro categories of sensors [2]:

- proprioceptive sensors: they allow to acquire information on the status of the robot by measuring quantities such as position and speed of the joints.
- heteroceptive sensors: they allow to acquire information on the surrounding environment.

The main characteristics of a sensor are [2]: the resolution, represented by the maximum variation of the input that does not provide a variation of the output, and the accuracy which is determined by the correlation of the values acquired by the sensor with respect to a standard. Other features are repeatability, stability, error/offset and noise [2]. The most important proprioceptive sensors are the position sensors (encoder, resolver) and the velocity sensors (such as the dynamo). For the category of heteroceptive sensors, on the other hand, there are force sensors that allow to measure the force applied to a body (such as the strain gauge), proximity/distance sensors used mostly by mobile robots for the detection of obstacles (infrared, ultrasound and laser) and vision sensors.

Vision sensors

Vision sensors are essential for mobile robots to acquire information on the surrounding environment but are also used in industrial robotics to acquire geometric information in the environment in which the robot operates [2]. The vision system can consist of one or more cameras. If the system is composed by more cameras, it is possible to obtain some information which allows to determine the 3D characteristics of the object [2]. Considering a vision system with a single camera, there are two types of configuration [2]:

1. fixed configuration - camera-to-hand: the camera is positioned on a fixed base and therefore has always the same field of view.

2. mobile configuration - camera-in-hand: the camera is positioned on the robot and the field of view changes according to the movement performed.

The main disadvantage of the fixed configuration is that the robot could position itself between the camera and the object of interest during movement while in the mobile configuration the accuracy is very variable due to the changes of the field of view [2]. In particular, there are two main techniques that exploit the information provided by the vision system to coordinate the movement of the robot [2]:

- Look-and-move technique: it consists of trajectory planning based on visual information.

- Visual Servoing: defines the control action on the basis of an error defined between the desired pose and the current one both in the space of the images and through a 3D reconstruction.

Visual servoing approaches

Visual servoing approaches are grouped into two categories [2]:

1. PBVS (Position based visual servoing): based on the reconstruction of the 3D pose of the object starting from the visual information. A cartesian error is generated with respect to the desired pose and the robot is guided with a cartesian controller. Multiple cameras are used to have 3D information or only one camera is employed but knowing the 3D information of the object [2][7].

2. IBVS (Image based visual servoing): the error is calculated directly from the features extracted from the 2D plane of the image and it is performed a movement in order to see the features in the desired position in the image. It is mainly used with a single camera positioned on the wrist. Since the error is expressed in the image plane it is necessary to associate the changes of the

extracted features with incremental changes of the velocity of the robot. This relationship is approximated with the interaction matrix (Image Jacobian) which depends on the camera parameters and on the calculated features [2][7].

2.3 ROS - Robot Operating System

ROS is a software framework, a collection of tools and libraries, that allows to design robotics software applications. It provides the standard services of an operating system and different functionalities such as hardware abstraction, device drivers, communication between processes over multiple machines and tools for testing [8][9]. The most important feature of ROS is how the applications run and communicate each others, allowing to build complex system with an abstraction of the hardware below. In particular, ROS allows to create packages containing programs, called nodes, and provides a way to connect a network of nodes with a central node master [10][8]. The master node manages all the ROS nodes connected to the

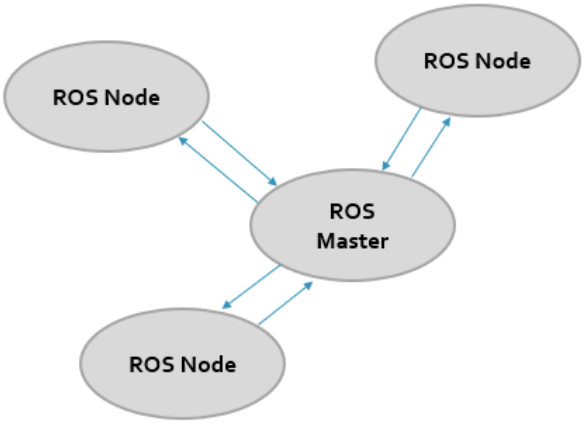

Figure 2.5: ROS nodes and master

network allowing parallel execution and different types of communication so that the nodes can exchange messages with each other, Fig. 2.5. All the nodes connect to the master before to run in order to register the information on the types of the messages they publish and subscribe. Then the master provides all the information to perform the connection and the communication with other nodes which publish and subscribe to the same messages. The nodes can be run on multiple devices and can communicate in different ways. The communication protocol used in a ROS is TCPROS, which is based on the standard TCP/IP sockets [11]. The Parameter Server is part of the master node and allows to store all the data with a defined

key in a central location accessible by the nodes [11].

ROS is based on the concept of code reuse allowing sharing and collaborations. It has been designed in order to be easy to integrate with other robot software frameworks and to be easy to test. However it only runs on Unix systems. ROS is language independent and the ROS framework is easy to implement with the programming languages Python and C++, thanks to the use of the "roscpp" and "rospy" libraries. In particular it means that all the nodes can be written in any language, and so for example it is possible to have two nodes written with two different programming languages that communicates with each other, thanks to the ROS communication layer which is below the "language level". The ROS nodes are grouped into packages and stacks, many of which are available on a github repository, Fig. 2.6. The package contains ROS nodes, messages, services, configuration files and dependent libraries, while a stack contains one or more packages. The package manifest provides information about the package such as name, version, description, license information, dependencies, and other meta information. Instead, the metapackages are particular kinds of packages which represent a group of related other packages [11].

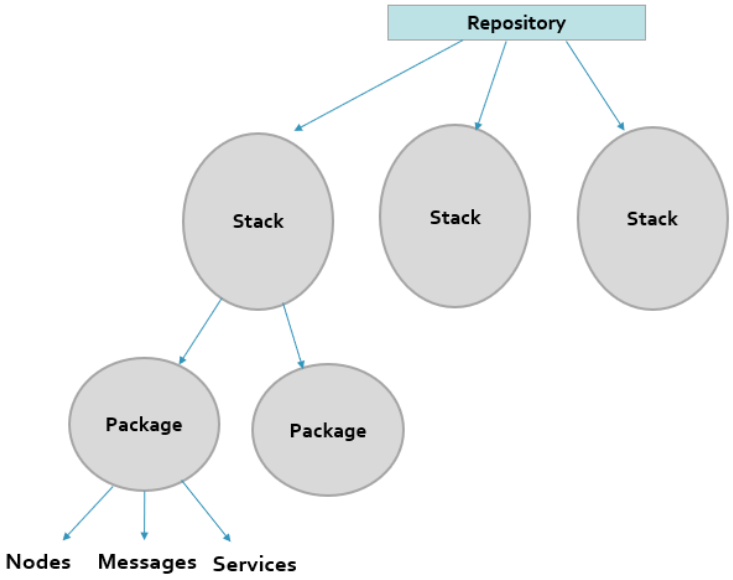

Figure 2.6: ROS packages

ROS provides three main communication tools [10]:

1. Topics: It allows to send message streams between nodes.
2. Services: It allows to create a synchronous client/server communication between nodes.

3. Actions: It provides an asynchronous client/server architecture based on topics.

The main way to create the network is to define publisher/subscriber connections with other nodes which communicate through different types of messages belonging to different topics, Fig. 2.7. There are some standard messages provided by the core packages, but they can be defined by the users [8]. The topic is the name of a specific stream of messages. So different nodes communicate with each other by publishing/subscribing messages of topics. There may be multiple publishers and subscribers to a specific topic. The topic messages are data structures that may also contain a header field which gives the timestamp and the frame of reference. However, in a distributed system with request/reply interactions, the publish/subscribe model based on topics is not the best way to communicate. For this reason the request/reply interaction is performed via services which are based on two type of messages: one message for a request and one for a reply. Then one node offers a service with a specific name and a client sends a request message and waits for a reply [11].

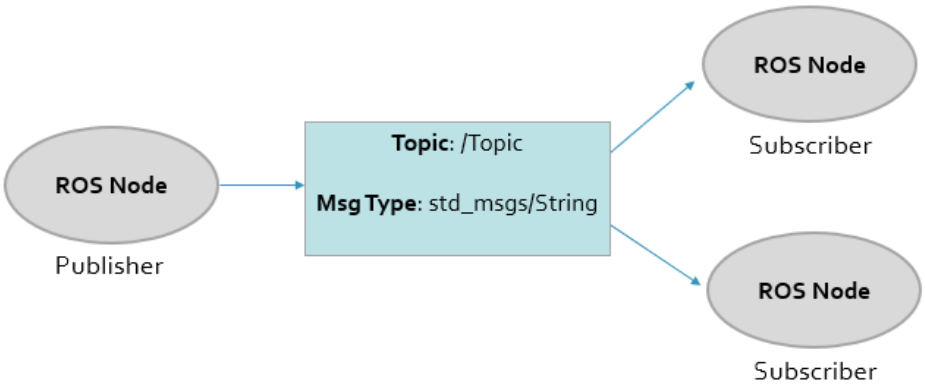

Figure 2.7: ROS publishers/subscribers

ROS main commands

- roscore: The command starts running the master node, the ROS parameter server and the Rosout logging node.

- rosrun: It allows to run a node, with the syntax "rosrun <package><executable>"

- rosnode: It shows the information about the current nodes, including publications, subscriptions and connections. It allows also to kill a running node with the syntax "rosnode kill node".
- rostopic: The command allows to obtain information about topics and also to publish messages with the syntax "rostopic pub topic message".

Example of a python publisher

[12]

```
import rospy
from std_msgs.msg import String
def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The function rospy.init_node() allows to initialize the node receiving in input its name. The function rospy.Publisher("chatter", String, queue_size=10) defines that the node publishes to the topic "chatter" with messages of type String while the function rate = rospy.Rate(10) creates a Rate object in order to loop at desired rate, in this case 10 times per second. In the while loop is checked the flag returned by the function rospy.is_shutdown() which states if the node should exit or not, and it is published a new message with the function pub.publish(hello_str). At the end of the loop there is a call to the function rospy.sleep() which allows to maintain the desired rate defined before [12].

Example of a python subscriber

[12]

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

The function rospy.Subscriber("chatter", String, callback) allows the node to subscribe to the topic “chatter” specifying the callback that must be invoked each time a new message is received [12]. The function rospy.spin(), instead, allows to keep the node running until the node is stopped. Moreover, compared with the roscpp library, the callback functions have their own threads [12].

Chapter 3

e.DO

3.1 Overview

e.DO is a modular educational robot developed by Comau S.p.A. The robot is a manipulator composed by six joints, more in details it is an anthropomorphic manipulator with spherical wrist. Inside the base, e.DO contains a raspberry pi controller with ROS installed that manages the motion with an open-source control logic. Thanks to the possibility to add some instruments on the sixth axis such us grippers or pens it's possible to make e.DO execute different tasks like simple pick and place movements to handle automated activities [13][14].

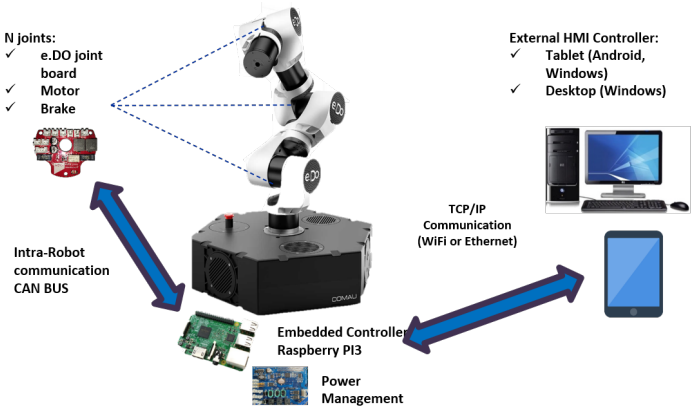

Figure 3.1: e.DO description [13]

The embedded controller Raspberry pi is based on Raspbian Jessie O.S. and has inside the ROS Kinetic Kame. The USB RosSerial allows the communication between the raspberry pi and all the joints on the robotic arm, that are considered as independent nodes from the ROS point of view. Moreover there is also the

possibility to communicate with external devices with the USBs, WiFi and Ethernet interfaces in order to connect and control the robot throught the PC or remote app/tablet and execute the desidered trajectory, Fig. 3.1.

3.2 Software Architecture

The architecture of the e.DO controller is showed in Figure 3.2.

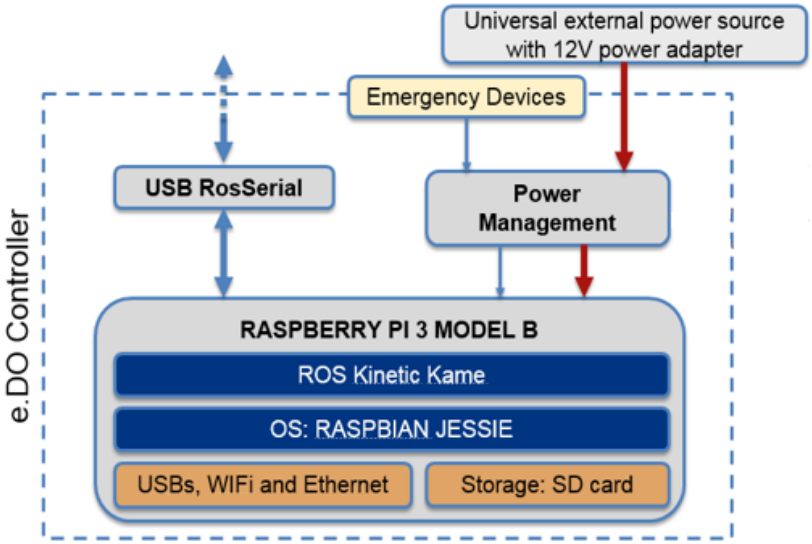

Figure 3.2: e.DO software architecture [13]

Inside the robotic arm there is a CAN network that allows the communications between all the joints, that are independent ROS nodes, and the raspberry pi. The six joints are equipped with print circuit boards with ChibiOS/RT (GPL3) and KebiOS that runs the applications.

The raspberry pi runs different ROS nodes which subscribes/publishes messages on different topics. The most important are: ROS e.DO Algorithm that manages the planning of the motion through the proper algorithm "ORL", ROS State Machine that manages the state of the robot in terms of positions and others variables related to the joints, and the ROS Bridge that manages the communication with the App. e.DO Algorithm, once subscribed to topics related to movement requests of the robot coming from external devices, publishes messages for the joints with a frequency of 10 ms that contain the correct values in terms of position, velocity and accelerations. This is possible thanks to the ORL library inside the kernel that provides the translation between positions of the joints and current/acceleration values to provide. The six joints that are subscribed to the specific topic, check

with a rate of 1ms if a new message is available thank to a microinterpolator and provide the values of current and acceleration to the actuators with a PID control.

3.3 Hardware Description

e.DO is a modular robot that is available in different version and configurations [15]. There are two types of joints: three big joints corresponding to the first three axes (Ax1 Ax2 Ax3) and three small joints for the lasts three axes (Ax4 Ax5 Ax6), each of these with a separate board, motor and an encoder sensor. Each axis also has the ability to rotate in both the directions [15]. The maximum payload supported by the robot is 1kg and it can apply a torque of 4N/m. For this reason, the robot is suitable for very simple pick and place movements and for automatic activities, e.g. sorting. The motion units of the six axes are composed of DC motors and a print circuit board with ChibiOS/RT (GPL3) for the control logic. Each unit is equipped with an integrated memory and the control is autonomous and independent from the others and can be configured in function of the goal. The motion units are differentiated into three big motion units with max speed of 38 deg/s and a static torque of 17.9Nm and three small motion units with max speed of 56 deg/s and static torque of 2.75Nm [15]. The hexagonal base structure holds the integrated raspberry pi with the sd memory card that stores the e.DO control logic. It is also equipped with an external USB port, one RJ45 Ethernet port, a DSub-9 Serial port that allows to connect with external devices and with an universal external power source with 12V power adapter [15]. There is also an external emergency stop button for emergency cases. Inside the robotic arm the CAN network connects and allows the communication between all the joints and the raspberry pi [13][16]. Tables 3.1 and 3.2 show the technical specifications and the motion units characteristics provided by Comau.

Specifications	Value
Number of axis	6
Max payload	1 Kg
Max reach	478 mm
Axis 1	+/-180° (38°/sec)
Axis 2	+/-113° (38°/sec)
Axis 3	+/-113° (38°/sec)
Axis 4	+/-180° (56°/sec)
Axis 5	+/-104° (56°/sec)
Axis 6	+/-2700° (56°/sec)
Total weight	11,1 Kg
Robot arm weight	5,4 Kg
Structure material	Ixef 1022

Table 3.1: Technical specifications provided by Comau [16] [17]

Specifications	Big	Small
Max Speed [deg/s]	38	56
Static Torque [Nm]	17.9	2.75
Max Torque [Nm]	20	3.16
Tilting Moment [Nm]	20	4.7
Weight [Kg]	0.88	0.42
Lenght [Mm]	114	90
Diameter [Mm]	85	70

Table 3.2: Motion units specifications provided by Comau [16] [17]

3.3.1 e.DO gripper

The considered end effector is a gripper with two fingers that is installed on the wrist of the robot and connected to the CAN network as the other motion units, Fig. 3.3. The presence of the prismatic joint allows a linear movement, to open and close the two fingers. Figures 3.4 and 3.5 show the gripper when it is open/closed with the relative measurements in mm. The technical specifications are shown in Table 3.3.

Figure 3.3: Gripper [18]

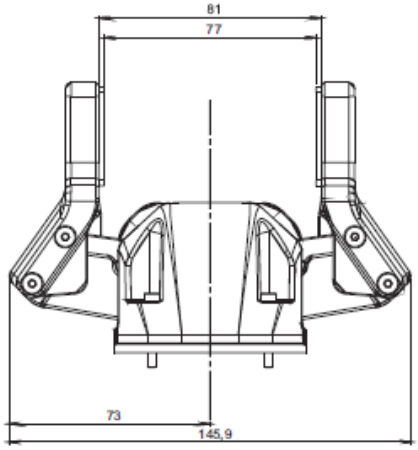

Figure 3.4: Gripper open (mm) [19]

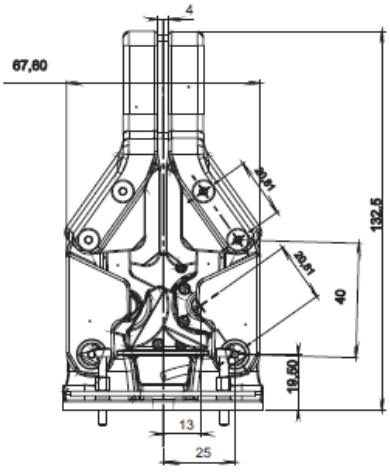

Figure 3.5: Gripper closed (mm) [19]

Description	Feature
Type	2-finger Gripper
Weight	200 g
Stroke	With neoprene fingertip cover 77 mm
Payload	400 g
Opening/closing maximum speed	80 mm/s
Clamping force	<65 N
On-board sensors	Encoder
Comau Part No.	CR82442100

Table 3.3: Technical specifications provided by Comau [15]

3.4 e.DO ROS Workspace

The e.DO ROS workspace is composed by a source folder that contains the source code of the catkin packages, the build folder that is the space for build all the packages in the source space and finally the development folder for the built target. The source folder contains two packages:

- edo_core _msg
- edo_core _pkg

edo_core _msg is the package that contains all the ROS messages associated to the ROS topics, while edo_core _pkg is the package developed by Comau that contains all the ROS nodes that allow to control the motion of e.DO and to manage the communication with external devices, like the Android application. The edo_core _pkg is composed by the config folder that contains the different configurations of e.DO, a launch folder for the launch file that sets all the parameters when the system is started, the CMakeList file, the include folder and the source folder that contains the C++ source code of all the ROS nodes that runs on the raspberry. As mentioned before, the most important nodes are: "edo_algorithms", "edo_state_machine", "edo_bridge" and "edo_recovery".

e.DO Algorithm is the main node that controls the motion of e.DO and manages the main functionalities that are: calibration, feedback signal, errors notification and communication with the State Machine node, Fig. 3.6. Particularly important is the calibration of the robot; in fact, at each startup e.DO needs to calibrate all the joints in order to set the initial reference of the encoders. The calibration can be performed by the users with the Android app and is the first operation that is done when starting up the application.

The most important function of e.DO Algorithm is to manage the move requests from the users in order to move the robot according to a defined target point in

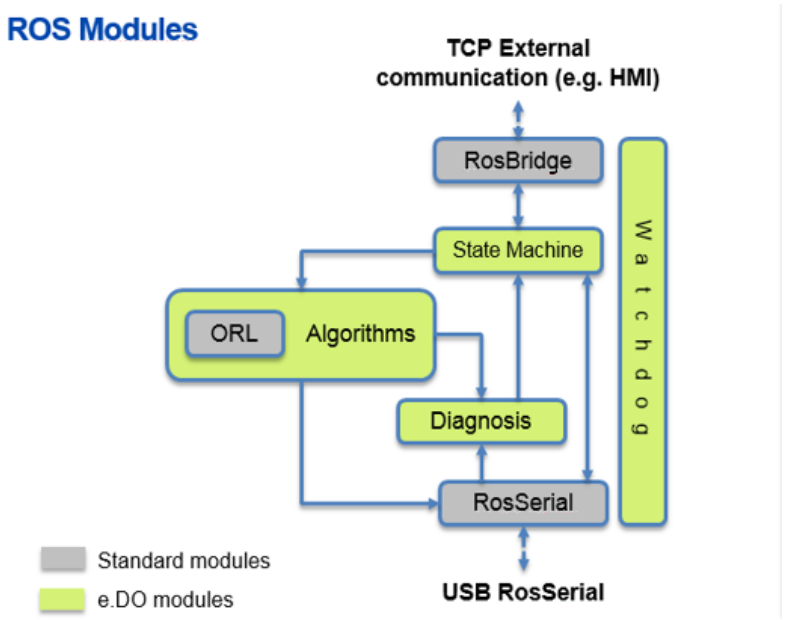

Figure 3.6: e.DO ROS nodes and communications [13]

the space or respect to a specific joints position. In particular, e.DO Algorithm subscribes to the topic "bridge_move" and receives messages of the type "MoveCommand" that represent movement requests sent by the users through the Android app or from other control nodes. The messages are analyzed and transmitted to the "ORL" library, the private Comau library that contains the kinematics algorithms that allows to control e.DO providing the positions of the joints with respect to a specific target point and the current/acceleration values for the motors. Then, after the translation from the library, e.DO Algorithm publishes messages for the joints with a frequency of 10 ms with the correct values of position, velocity and accelerations to achieve. The structure of the MoveCommand message is defined in the package edo_core _msg and is composed of different fields that are showed in Tables 3.4 - 3.7:

Type	Description
uint18	move_command
uint18	move_type
uint18	override
uint18	delay
uint18	remote_tool
uint18	cartesian linear speed
Point	target
Point	via
Frame	tool
Frame	frame

Table 3.4: MoveCommand message structure

Type	Description
uint18	data_type
CartesianPose	cartesian_data
uint64	joint_mask
float32[]	joints_data

Table 3.5: Point message structure

Type	Description
float32[]	[x,y,z,a,e,r]
string	config_flag

Table 3.6: CartesianPose message structure

Type	Description
float32[]	[x,y,z,a,e,r]

Table 3.7: Frame message structure

The move_command field can be of different types and depending on the action request can be "MOVE", "JOG", "CANCEL" or "PAUSE". The type MOVE represents a request of a movement to a target position starting from the current one. The type JOG increments one of the values of the joints, CANCEL terminates the execution of the current action, and PAUSE stops the execution of the action.

The field move_type, instead, specifies the type of movement to perform. There are three types of movements that can be performed:

- JOINT: Linear movement in the joint space to the target position
- LINEAR: Linear movement in the cartesian space to the target position
- CIRCULAR: Type of movement of the last release of the package that allows to perform a circular movement

The field Target is a message of type Point and represents the target joint values or the target cartesian position to reach. In the Via Point field it is possible to specify an intermediate point during the motion before reaching the target. The Override, instead, is a percentage value that represents the velocity of the joints during the movement, with the value 100 associated to the maximum velocity of the motors. By analyzing the structure of the Point message, the data_type can be JOINT if it is performed a movement in the joint space or CARTESIAN if in the cartesian space. In the first case the target position is a vector of joint angular positions while in the second case is a vector that contains the cartesian coordinates and the pose of the end-effector. The joint_mask defines how many and which joints to move during the movement. Each joint is associated to a binary value that can be 1 if the joint is activated or 0 if not. Then, the final binary number is converted into the corresponding integer number [20].
In Tables 3.8 and 3.9 are showed the codes associated to each field in the messages.

Type	Code
MoveCommand:	
JOGMOVE	74
JOGSTOP	83
MOVE	77
PAUSE	80
CANCEL	67
MoveType:	
JOINT	74
LINEAR	76
CIRCULAR	67

Table 3.8: MoveCommand and MoveType codes

The State Machine node manages the state of the robot and communicates with e.DO Algorithm, while the ROS Bridge node manages the communication with the App and external devices. e.DO Recovery, instead, allows to record all the data relative to the movements performed, such us position, velocity and current of the

Type	Code
DataType:	
JOINT	74
CARTESIAN	80
MaskType:	
JOINT_MASK6	63
JOINT_MASK7	127
JOINT_MASK_EXT	64

Table 3.9: DataType and MaskType codes

motors, and to store them in a txt file on the raspberry internal memory [20]. The State Machine node subscribes to the topic bridge_move and publishes messages on the topics machine_move and machine_state. In fact, each time a new message is received on the topic bridge_move, the node updates and publishes its status on the topic machine_state and transmits the message to e.DO Algorithm on the topic machine_move [20]. The possible states of e.DO are:

- INIT: e.Do initial state when powered on
- NOT CALIBRATE/CALIBRATE: States before and after the calibration procedure
- MOVE: When the message received is a Move
- JOG: When the message received is Jog
- ERROR: When an internal error occurs
- BRAKED: When a critical fault occurs and the motors are powered off
- COMMAND: When a new message is received

The codes associated to each state are showed in Table 3.10.

Type	Code
INIT	0
NOT CALIBRATE/CALIBRATE	1-2
MOVE	3
JOG	4
ERROR	5
BRAKED	6
COMMAND	-

Table 3.10: e.DO States

3.5 e.DO Kinematics

e.DO belongs to the category of anthropomorphic robots with a spherical type wrist structure, Fig. 3.7. It is composed by six joints, hence 6 degrees of freedom, where the first three belong to the arm and the lasts to the wrist. The workspace is a portion of the sphere of large dimension and it is a very typical structure used in the industries because allows to make easily many types of movements [2].

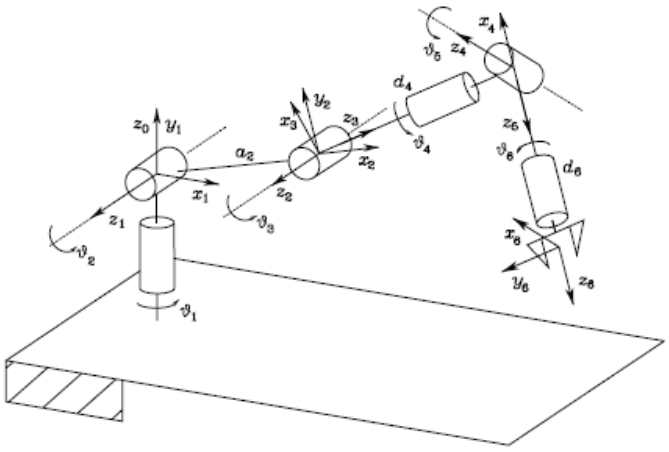

Figure 3.7: Anthropomorphic arm with spherical wrist [2]

n	$a_i(mm)$	$\alpha_i(rad)$	$d_i(mm)$	$\theta_i(rad)$
1	0	$\pi/2$	0	θ_1
2	210.50	0	0	θ_2
3	0	$\pi/2$	0	θ_3
4	0	$-\pi/2$	268.00	θ_4
5	0	$\pi/2$	0	θ_5
6	0	0	174.50	θ_6

Table 3.11: DH Parameters

Considering the DH parameters in Table 3.11, the transformation matrices for the arm are:

$$T_1^0(q_1) = \begin{bmatrix} \cos q_1 & 0 & -\sin q_1 & 0 \\ \sin q_1 & 0 & -\cos q_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

$$T_2^1(q_2) = \begin{bmatrix} \cos q_2 & -\sin q_2 & 0 & a_2 \cos q_2 \\ \sin q_2 & \cos q_2 & 0 & a_2 \sin q_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$T_3^2(q_3) = \begin{bmatrix} \cos q_3 & -\sin q_3 & 0 & a_3 \cos q_3 \\ \sin q_3 & \cos q_3 & 0 & a_3 \sin q_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

while for the wrist:

$$T_4^3(q_4) = \begin{bmatrix} \cos q_4 & 0 & -\sin q_4 & 0 \\ \sin q_4 & 0 & -\cos q_4 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$T_5^4(q_5) = \begin{bmatrix} \cos q_5 & 0 & \sin q_5 & 0 \\ \sin q_5 & 0 & -\cos q_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

$$T_6^5(q_6) = \begin{bmatrix} \cos q_6 & -\sin q_6 & 0 & 0 \\ \sin q_6 & \cos q_6 & 0 & 0 \\ 0 & 0 & 1 & d_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

Considering the previous transformation matrices and the DH parameters in Table 3.11, it is obtained the final transformation that determines the position and orientation of the last reference frame of the 6$^{\text{th}}$ link respect to the base reference frame:

$$T_6^0(\underline{q}) = \begin{bmatrix} R_6^0(\underline{q}) & t_6^0(\underline{q}) \\ 0^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{n}_6^0(\underline{q}) & \mathbf{s}_6^0(\underline{q}) & \mathbf{a}_6^0(\underline{q}) & \mathbf{t}_6^0(\underline{q}) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

with:

$$\mathbf{t}_6^0 = \begin{bmatrix} a_2c_1c_2 + d_4c_1s_{23} + d_6(c_1(c_{23}c_4s_5 + s_{23}c_5) + s_1s_4s_5) \\ a_2s_1c_2 + d_4s_1s_{23} + d_6(s_1(c_{23}c_4s_5 + s_{23}c_5) - c_1s_4s_5) \\ a_2s_2 - d_4c_{23} + d_6(s_{23}c_4s_5 - c_{23}c_5) \end{bmatrix} \tag{3.8}$$

$$\mathbf{n}_6^0 = \begin{bmatrix} c_1(c_{23}(c_4c_5c_6 - s_4s_6) - s_{23}s_5c_6) + s_1(s_4c_5c_6 + c_4s_6) \\ s_1(c_{23}(c_4c_5c_6 - s_4s_6) - s_{23}s_5c_6) - c_1(s_4c_5c_6 + c_4s_6) \\ s_{23}(c_4c_5c_6 - s_4s_6) + c_{23}s_5c_6 \end{bmatrix} \tag{3.9}$$

$$\mathbf{s}_6^0 = \begin{bmatrix} c_1(-c_{23}(c_4c_5s_6 + s_4c_6) + s_{23}s_5s_6) + s_1(-s_4c_5s_6 + c_4c_6) \\ s_1(-c_{23}(c_4c_5s_6 + s_4c_6) + s_{23}s_5s_6) - c_1(-s_4c_5s_6 + c_4c_6) \\ -s_{23}(c_4c_5s_6 + s_4c_6) - c_{23}s_5s_6 \end{bmatrix} \tag{3.10}$$

$$\mathbf{a}_6^0 = \begin{bmatrix} c_1(c_{23}c_4s_5 + s_{23}c_5) + s_1s_4s_5 \\ s_1(c_{23}c_4s_5 + s_{23}c_5) - c_1s_4s_5 \\ s_{23}c_4s_5 - c_{23}c_5 \end{bmatrix} \tag{3.11}$$

3.5.1 Singolarities

Considering the geometric Jacobian:

$$J_g(q) = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} \tag{3.12}$$

with $J_{ij} \in R^{3,3}$
and fixing the origin of the end-effector reference frame in the intersection of the wrist axes, it's obtained the decoupling of the singularities of the arm and wrist:

$$J_{12} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \tag{3.13}$$

$$det(J_g) = det(J_{11})det(J_{22}) \tag{3.14}$$

By solving the equation $det(J_{11}) = 0$ are obtained the arm singularities while solving $det(J_{22}) = 0$ are obtained the wrist singolarities:

- Wrist singularities: $q_5 = 0, \pi$
- Arm singularities: $q_3 = 0, \pi$ (elbow singularity) and $(a_2c_2 + a_3c_{23}) = 0$

with :

$$J_{22} = \begin{bmatrix} z_3 & z_4 & z_5 \end{bmatrix} \tag{3.15}$$

$$J_{11} = \begin{bmatrix} -s_1(a_2c_2 + a_3c_{23}) & -c_1(a_2s_2 + a_3s_{23}) & -a_3c_1s_{23} \\ c_1(a_2c_2 + a_3c_{23}) & -s_1(a_2s_2 + a_3s_{23}) & -a_3s_1s_{23} \\ 0 & a_2c_2 + a_3c_{23} & a_3c_{23} \end{bmatrix} \tag{3.16}$$

Chapter 4

Project development

4.1 System description

The considered pick and place task consists of releasing an object grasped by e.DO in a double box positioned on the base of the AGV. The two boxes on the base represent two containers associated to correctly processed pieces and production wastes. Each box is associated to a specific color that is defined by the user in the configuration step. The object to be released in the correct box is a small ball which can be easily grasped and released by the gripper in a possible real implementation, while for the gazebo simulation are considered both a small ball and a coke can, which are object standard models already available in the environment. In order to coordinate the movement between the two robots and improve the accuracy on the release position, it has been used a camera sensor. For the real implementation the camera is connected directly to the external raspberry pi controller, while for the Gazebo simulation it has been used a gazebo plugin that simulates the behaviour of the camera and that publishes the images with a fixed rate. In particular, the camera has the dual purpose of correcting the trajectory of the AGV and allowing the identification of the point where to release the object (ball/coke can) in the double box with different colors. In the gazebo simulation it has been used also a laser scan sensor to measure the AGV distance from e.DO and hence to check if the AGV is positioned correctly in the target release zone. With the aim of having the best field of vision both to recognize the AGV and to coordinate the release of the piece in the right box, two types of position for the camera have been analyzed:

1. Camera positioned on the wrist of e.DO
2. Camera positioned on the e.DO base or in front of the two robots

By positioning the camera on the base of e.DO or in front of the robots so that the gripper and the AGV are visible in the camera's field of view, it is possible to

recognize if the AGV has arrived in the release zone and also to coordinate the release movement on the correct box. On the contrary, positioning the camera on the wrist it is possible only to coordinate the movement by sliding the e.DO wrist on the boxes, but it is more difficult to recognize the AGV. For this reason, in this case the introduction of the laser sensor is required. Figure 4.1 provides a description of the overall system:

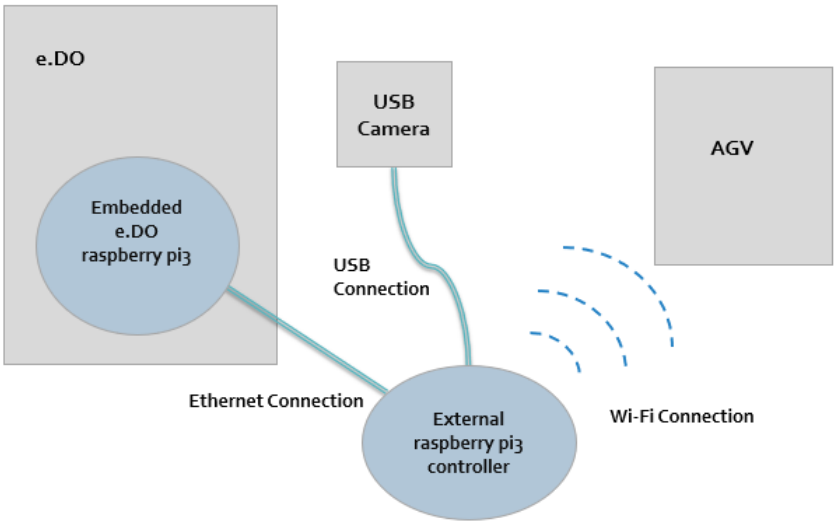

Figure 4.1: System description and connections

The external raspberry pi represents the core of the system. It runs Ubuntu Mate 18.04.2 O.S. with ROS Melodic installed in the base version. The ROS package inside the workspace manages all the movements to perform in order to accomplish the task. The raspberry pi is connected to the camera sensor through the USB cable while an ethernet connection allows the communication with the raspberry pi embedded in the base of e.DO. The communication with the AGV takes place via a Wifi connection where both the raspberry and the AGV are connected.

For the gazebo simulation, however, the raspberry has not been used because the simulation of the entire system is very heavy from the computational point of view and requires more performing resources. For this reason, the system was simulated on another much more performing machine which allowed to test all the functionality of the ROS package in a simulated environment. Figure 4.2 describes

the system in the Gazebo environment.

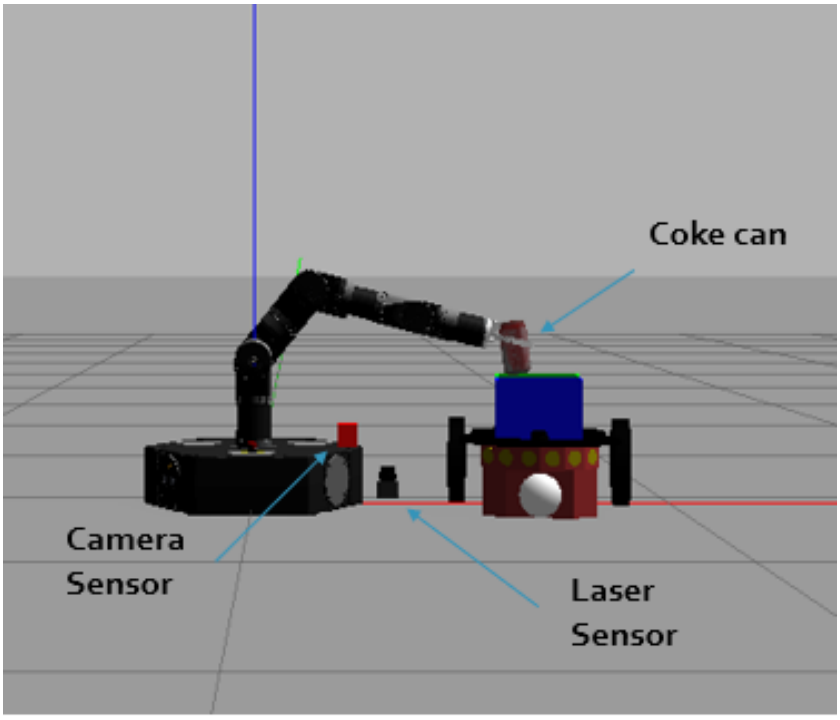

Figure 4.2: System in the Gazebo world

4.2 Communication with e.DO

With the goal to integrate the robot e.DO with external devices, two different types of communication between the raspberry pi controller and the raspberry pi embedded in the e.DO base have been analyzed:

1. Communication with ROS
2. Comunication with Python ROS Bridge library

The first type of communication is based on ROS and consists on running ROS on multiple machines in order to obtain a unique autonomous system with different ROS nodes on the different machines that communicate each other as running on the same machine. The second way to communicate is based on the “roslibpy“ library, that is a python library that allows to interact with the ROS framework.

4.2.1 Communication with ROS

This first way to communicate is based on creating a ROS communication between the devices, with one device that run the ROS master and the others that connect to the master in the network [21]. In this case the raspberry pi inside e.DO runs the ROS master and all the remote nodes on the external raspberry connect to the master through an ethernet cable between the two raspberry pi.
The first step is to connect the two raspberry to the same local network and get the ip addresses. Then it is necessary to check if they have the same version of ROS installed or versions that are compatible in terms of type of messages and dependencies. The last step is to edit a bashrc file to indicate the ip address of the computer running the ROS master and the ip address of the local computer [21].

```
#!/bin/bash
source /home/raspberry/catkin_ws/devel/setup.bash
sudo ifconfig eth0 10.42.0.10
export ROS_MASTER_URI=http://10.42.0.49:11311
export ROS_IP=10.42.0.10
sleep 5
rosrun edo_core_pkg edo_agv_node
sleep 2
wait
```

The previous configuration script that runs on the raspberry pi controller, sequentially:

1. Set the local ethernet ip address to 10.42.0.10
2. Set the ip address of the ROS master, in this case it is the master that runs on e.DO
3. Set the local Ros ip address
4. Run the node “edo_agv_node” on the current raspberry pi

The edo_agv_node in this case is a node that reads the state of the robot e.DO and prints the messages of the topic “machine_state” on the current console.

4.2.2 Comunication with Python ROS Bridge library

The Python ROS Bridge library allows to create python programs that are able to interact with the ROS framework. The library is based on WebSockets in order to connect to the rosbridge and provides the main ROS functionalities and different services such us publishing, subscribing, service calls and actionlib [22]. Moreover, it doesn't require a ROS environment and provides the possibility to write python programs in different platforms other than Linux [22].

The next functions in the python code allow to connect with the ROS framework:

```
1    client = roslibpy.Ros(host='10.42.0.49', port=9090)
2    client.run()
```

In particular, the first function creates a connection with ROS that runs on the computer with the local ip address "10.42.0.49", that is the ip address of the ROS master. The second one starts the connection with a call to the function run() that starts the event loop in the background [23].

Also for this type of communication it has been used an ethernet cable to connect the external raspberry pi, that runs the python program, and the robot e.DO that runs the ROS master and the other nodes of the edo_core _package.
As an example, the following part of a python program starts a ROS node and publishes messages in a while loop based on the value of a flag variable. The message is "MovementCommand" of the topic "bridge_move". The move_joint() function publishes a new "MovementCommand" message on the topic "bridge_move" with a particular position of the joints.

```
client = roslibpy.Ros(host='10.42.0.49', port=9090) # Cable
#client = roslibpy.Ros(host='192.168.12.1', port=9090) # Wi-Fi
MovementCommand = roslibpy.Topic(client, '/bridge_move', '
    edo_core_msgs/MovementCommand')
listener= roslibpy.Topic(client,'machine_state','edo_core_msgs/
    MachineState')
client.run()
command = {}
command_template = {  # Standard Command Template
     "move_command": 0,
     "move_type": 0,
     "ovr": 0,
     "delay": 0,
     "remote_tool": 0,
     "cartesian_linear_speed": 0.0,
     "target": {
          "data_type": 0,
          "cartesian_data": {"x": 0.0, "y": 0.0, "z": 0.0, "a": 0.0, "e
    ": 0.0, "r": 0.0, "config_flags": ''},
          "joints_mask": 0,
          "joints_data": [0]},
     "via": {
          "data_type": 0,
          "cartesian_data": {"x": 0.0, "y": 0.0, "z": 0.0, "a": 0.0, "e
    ": 0.0, "r": 0.0, "config_flags": ''},
          "joints_mask": 0,
          "joints_data": [0]},
     "tool": {"x": 0.0, "y": 0.0, "z": 0.0, "a": 0.0, "e": 0.0, "r":
    0.0},
     "frame": {"x": 0.0, "y": 0.0, "z": 0.0, "a": 0.0, "e": 0.0, "r":
    0.0}}
def move_joint(ovr=30, j1=0.0, j2=0.0, j3=0.0, j4=0.0, j5=0.0, j6
    =0.0, j7=0.0):
     command = command_template
     command['move_command'] = MoveCommand.EXE_MOVE.value
     command['move_type'] = MoveType.JOINT.value
     command['ovr'] = ovr
     command['target']['data_type'] = DataType.E_MOVE_POINT_JOINT.
    value
     command['target']['joints_mask'] = MaskType.JOINT_MASK7.value
     command['target']['joints_data'] = [j1, j2, j3, j4, j5, j6, j7]
     MovementCommand.publish(roslibpy.Message(command))
while(1):
     if(flag):
          move_joint(100,40.0,0.0,35.0,20.0,90.0,90.0,0.0)
     else:
          move_joint(100,0.0,0.0,0.0,0.0,0.0,0.0,0.0)
```

4.3 Autonomous Guided Vehicle

Since the AGV was not available during the development phase it has been performed a gazebo simulation with a differential drive mobile robot that interacts with the robot e.DO. The mobile robot has been created developing the URDF file that describes and contains the model of the robot for the simulation in the Gazebo world. The model has been created in such a way to be similar to the common Pioneer 3-DX mobile robot, Fig. 4.3. The Pioneer 3-DX is a small differential drive mobile robot that is widely used in research laboratories or for indoor environment. The robot is equipped in the standard version with a front sonar, one battery, encoder sensors for the wheel and with an on board micro controller. The characteristics of the robot that make it very popular are: versatility, reliability and durability. For this reason is the preferred platform for advanced intelligent robotics [24].

Figure 4.3: Pioneer 3-DX [24]

As it is possible to see in Figure 4.4, the mobile robot built in Gazebo is equipped with two boxes positioned on the base and that represents the two containers associated with correctly processed pieces and production wastes.

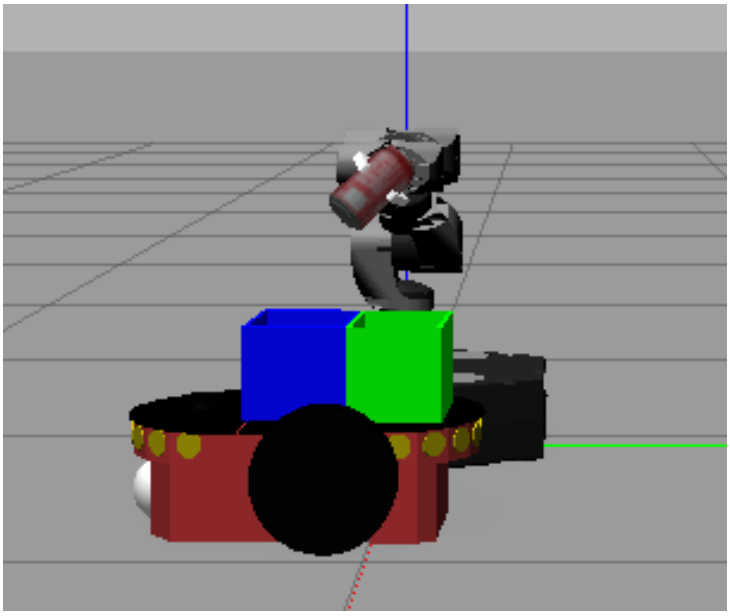

Figure 4.4: Mobile Robot in Gazebo

4.4 Task assumptions

During the development phase some assumptions have been made. First of all, when the system starts, the AGV reaches e.DO in a known release zone and positions itself in lateral position at a predefined point in the space so that the two boxes on the base are clearly visible from the camera. Moreover, the movement of the AGV is managed and planned separately from the system. The second assumption is that e.DO starts the movement from a known joint configuration called "INIT position" and the object is already kept by the gripper. For this reason, it is possible to identify two main areas and three points in space that allow to define the Point-to-Point movements of e.DO from one area to another in order to release the piece in the right AGV box. The points and the zones are shown in Table 4.1.
The e.DO zone is the area where the robot e.DO is located when it is ready to

Zone	Point
e.DO zone	Init point
Release zone	e.DO target point
	AGV target point

Table 4.1: Zones and points of the workspace

release the object, while the Release zone is the area inside the workspace of e.DO that the AGV reaches when the system starts. The Init point is the point in the space of the e.DO zone that corresponds to the initial position of the end-effector before the start of the movement. The AGV target point is the position in the space where the AGV is positioned as soon as it arrives in the release area. The pose of the AGV has the same orientation with respect to the camera so that the AGV and the boxes are perfectly visible when it arrives and position itself in a lateral position. The e.DO target point instead is the first point in which e.DO starts to move from the Init point when the system starts. This point refers to the point of the space belonging to the Release zone positioned above the boxes and in the center of the AGV.

4.5 State machine

The state machine shown in Figure 4.5 describes at a high level the evolution of the system, starting from checking the status of the two robots up to the completion of the task.

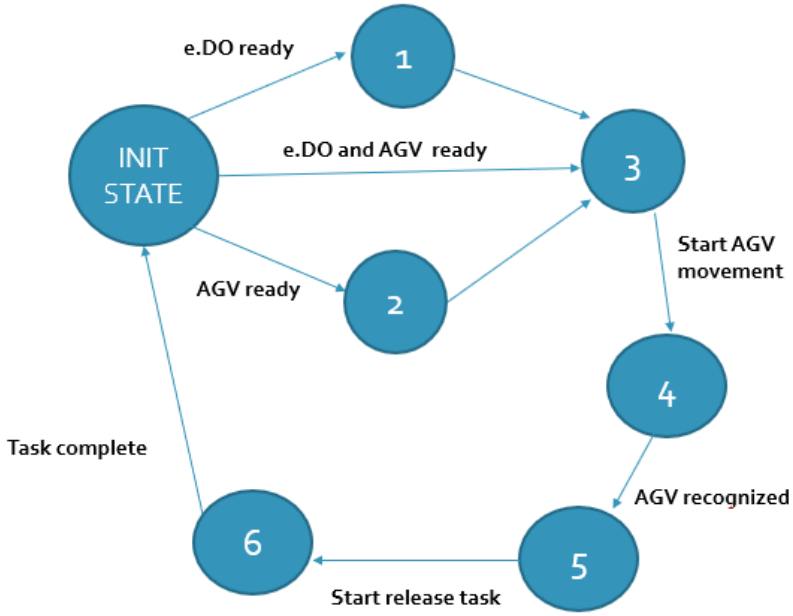

Figure 4.5: System state machine

The states are:

- INIT: It is the initial state of the system in which it waits for the two robots to be ready to perform the task. In other words, it waits until e.DO has grabbed the piece to be released and is ready in the Init position and the AGV is ready to move to the release zone.
- 1: State in which e.DO is ready while the AGV is not-ready yet. The system waits until the AGV is ready.
- 2: State in which the AGV is ready to move to the release zone while e.DO is still moving. The system waits e.DO until it is ready in the Init position with the object kept by the gripper.
- 3: State in which both e.DO and the AGV are ready to execute the task.

- 4: In this state the AGV is moving and the system waits until it reaches the target position in the Release zone.
- 5: State in which the AGV is arrived in the target point and it has been recognized from the vision system.
- 6: It is the final state in which e.DO is moving to perform the task. The system waits until the task is completed and the object has been released in the box.

4.6 Activity and path planning

The main task of e.DO is to take pieces and put them inside one of the two boxes positioned on the base of the AGV. For the thesis work, only the piece release operation is considered. The piece release operation can be divided into a series of elementary moves that allow to complete the task. In particular, it is possible to identify three elementary moves:

1. Move the gripper with the piece inside towards the e.DO target point in the space corresponding to the center of the AGV above the boxes.
2. Move the gripper above the correct target box
3. Open the gripper and drop the piece into the box

The three moves are performed consecutively in an orderly manner and define the whole path that the robot must follow. The first move is associated with a path in the joint space that corresponds to a linear PTP movement from the Init initial configuration of e.DO Q_0 to the final configuration Q_f corresponding to the e.DO target point in space, Fig. 4.6. For this type of path, it has been chosen a movement in the joint space in order to avoid any kinematic singularities being the longest path of the task. Furthermore, this movement in the joint space doesn't allow to assign a particular path of the end effector and the resulting cartesian movement is of a non-linear type. For the second move it is necessary to consider the physical constraints determined from the shape of the boxes during the motion. In order to avoid collisions, the move is associated with a path in the cartesian space that corresponds to a linear PTP movement between the e.DO target point and the point in the space above the target box, Fig. 4.7. Since the e.DO target point is referred to the center of the AGV above the boxes, it is possible to identify a linear movement along the longitudinal axis above the boxes

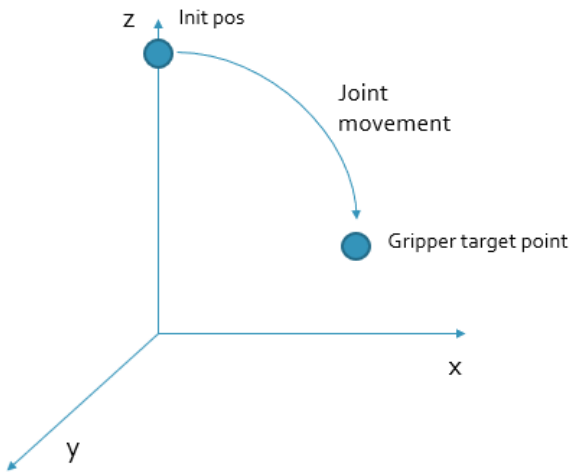

Figure 4.6: Joints movement

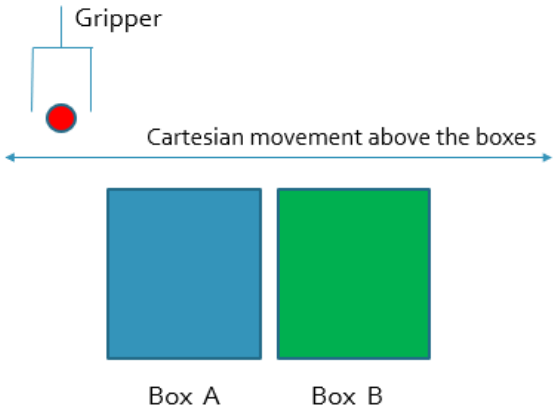

Figure 4.7: Cartesian movement

where the gripper moves until it reaches the right box. The trajectory related to the two moves was not planned but the Comau library was directly exploited. The library allows the generation of the trajectory both with planning in the joint space and in the cartesian space. In particular, by publishing a MovementCommand message on the topic /bridge_move, it is possible to move e.DO in the space both with a linear movement in the joint space and with a linear cartesian movement

of the gripper. When planning the trajectory in the operational space, starting from the planned trajectory $p(t)$ it is necessary to calculate the corresponding samples of position, velocity and acceleration of the joints through the inverse kinematics. The Comau library allows this calculation directly through an inverse kinematics function implemented in the cartesian movement. So the first move is associated to a MovementCommand type message in the joint space that contains the references of the joints with respect to the target e.DO point, while the second move is associated to a MovementCommand message corresponding to a cartesian movement of the gripper with respect to the coordinates of the target box, Fig. 4.8.

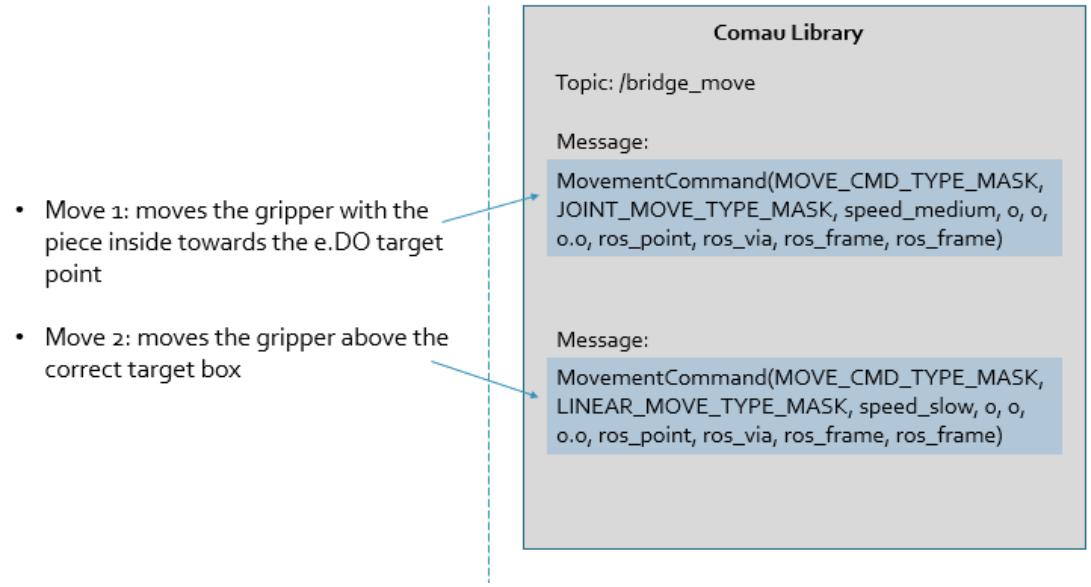

Figure 4.8: Comau movements

4.7 Gripper positioning exploiting visual information from the camera

Once the gripper is ready and positioned in the e.DO target point right above the boxes of the AGV, a cartesian linear movement is executed along a straight line between the e.DO target point and the point in the space above the target box. For this purpose the camera allows to coordinate the movement of the gripper through the identification of the point in the space above the target box in which the object can be safely released. In order to do that, image processing operations are performed thanks to which it is possible to detect the position of the target box and/or the position of the e.DO gripper in the image 2D plane. Depending on the type of the camera configuration and on the image processing techniques, different strategies have been adopted. The implemented strategies exploit the information extracted from the 2D image in order to move and position the gripper on the target box, thanks to the functions provided by the Comau library, as mentioned before.

4.7.1 Camera-to-hand configuration

Considering the camera-to-hand configuration, it has been implemented an image-based visual servoing strategy (IBVS) that use the cartesian type MovementCommand message of the topic /bridge_move in order to move the gripper in the direction of the target box. In this configuration the camera is positioned on the base of e.DO or it is arranged sideways in front of the two robots in such a way that the gripper and the boxes on the AGV are clearly visible when the AGV is located in lateral position. In particular, three approaches with three different type of image processing techniques have been evaluated. The image processing techniques differ in the operations that allow the detection of the gripper and/or the object to release. Given the image provided by the camera, the first approach is based on the real time detection of the gripper and the target box. Instead, the other two approaches are based on the detection of the target box and the object to be released instead of the gripper. This choice is justified by the fact that the position of the object in the image provides a reliable information on the current position of the gripper and above all, it makes the identification process much easier since the geometry of the gripper is complex and therefore complicated to identify. Regardless of the type of approach used, when the target box and the gripper/or the object have been identified, the pixel coordinates in the image plane are calculated. Finally, it is computed the error between the two pixel coordinates on the width dimension of the camera that is used for the control action. In fact, based on the errors computed in the image processing phase, the control action increments the position of the gripper with a cartesian movement until the camera

sees the gripper ready to release the object on the top of the correct box. Figure 4.9 describes the general control scheme and Figure 4.10 shows the image processing steps:

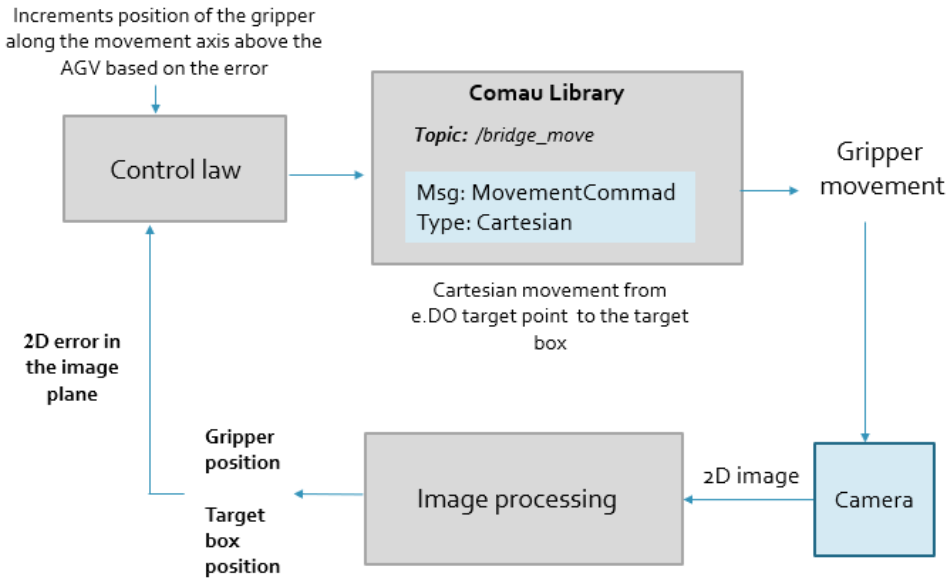

Figure 4.9: Camera-to-hand general control scheme

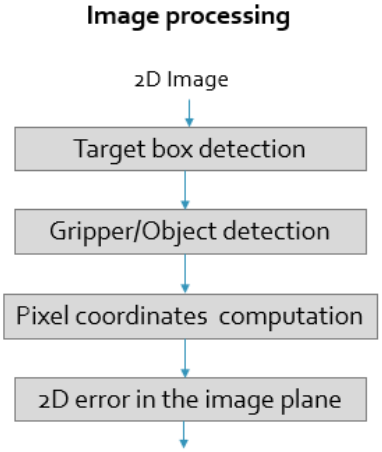

Figure 4.10: Image processing steps

4.7.2 Camera-in-hand configuration

Considering the camera-in-hand configuration, instead, the same IBVS approach is used but with the camera located on the wrist. More in details, the camera is positioned below the wrist of e.DO in such a way to see the boxes of the AGV under the current position of the gripper. In this case the image processing functions are based only on the identification of the target box. Once the target box has been identified, the pixel coordinates in the image plane and the error with respect to the center of the camera are calculated. Then, the control action increments the position of the gripper with a cartesian movement until the target box is positioned at the center of the camera. The control scheme and the image-processing steps are described in Figures 4.11 and 4.12:

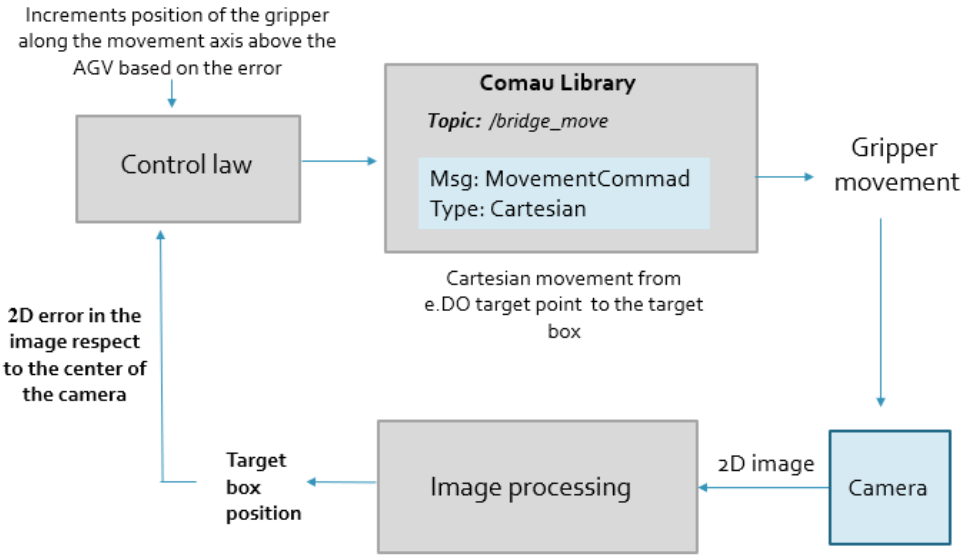

Figure 4.11: Camera-in-hand general control scheme

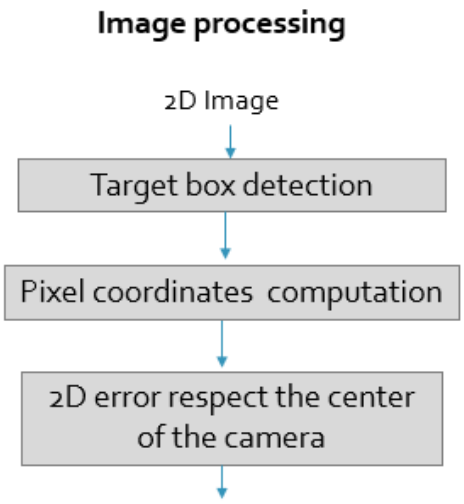

Figure 4.12: Image processing steps

4.8 Edo-AGV_Integration ROS Package

A specific ROS package has been developed which implements all the functionalities of the system. The ROS package contains different nodes and a finite state machine that manages all the nodes and the evolution of the system. In developing the package and writing the code, a modular approach was used in order to have a structured environment divided into independent modules that can be easily reused and/or modified for future developments. In particular, the developed structure includes a management code associated to each robot that manages its communication and movements and a code that manages the state of the entire system. The code was written as modular and generic as possible, thanks to the use of global variables and generic parameters which are defined and set during the configuration phase. Python was chosen as the programming language in writing the different nodes as it is an interpreted language and very simple to use. Its use simplified the development phase by reducing the complexity of the code and the time required for the testing phase, since it was not necessary to recompile the nodes after each change. Therefore the package consists of three main nodes and a state machine that manages the entire system:

- State Machine node
- e.DO Manager node
- AGV Manager node
- System state node

The package directory contains:

1. src folder : containing all the python nodes of the system
2. param folder : containing the configuration file with all the parameters for the nodes
3. launch folder : containing a launch file that launches the nodes and registers the parameters of the configuration file to the ROS parameter server
4. package.xml file

The complete representation of the workspace is described in Figure 4.13.
In order to recognize the AGV when it arrives and is positioned in front of the

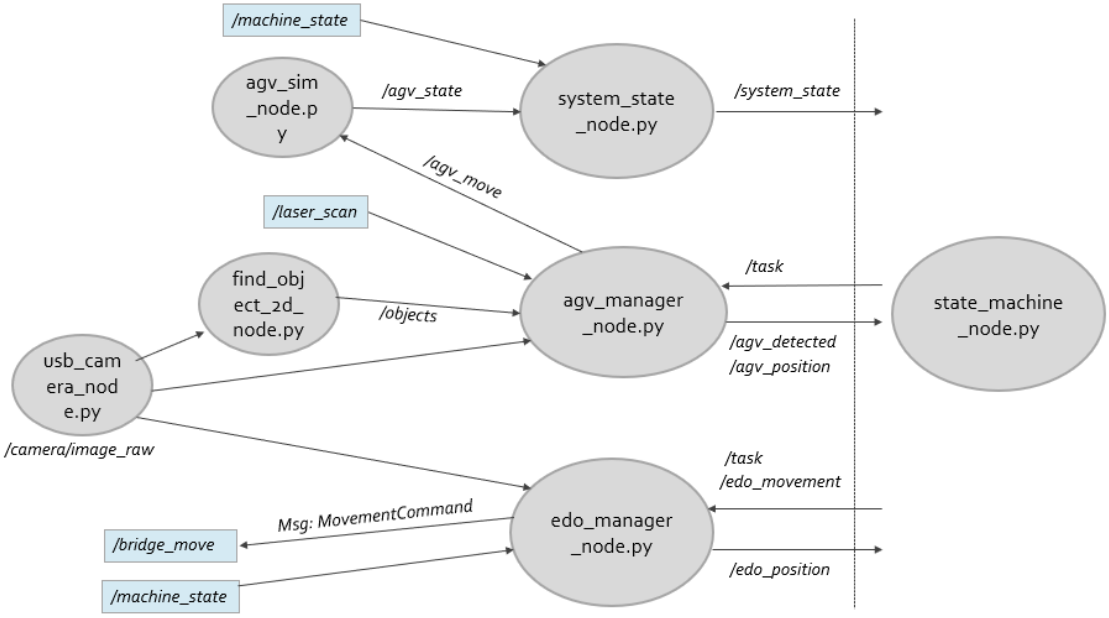

Figure 4.13: ROS workspace

camera, the node find_object_2d_node.py of the ROS package find _object _2d was used. This node allows to recognize objects in the image provided by the camera that have been previously saved in a learning phase. The usb_camera_node.py is a ROS node that allows to interface to the usb camera connected to the raspberry and publishes the captured images as Image messages of the sensor_msgs library. Since the AGV was not available during the development phase, the node agv_sim_node.py was created which allows to simulate the actions and the movements of the AGV during the gazebo simulation. As it is possible to see in Figure 4.13, the agv_manager _node subscribes also to the topic /laser_scan that is the topic used by the laser plugin in the gazebo simulation. Thanks to the information provided by the laser it is possible to measure the AGV distance from e.DO and to check if the AGV is correctly positioned.

4.8.1 Smach State Machine

The SMACH ROS package was used for the implementation of the state machine; such a package provides various tools and efficient structures for the implementation of a robust state machine that integrates very well with the ROS environment. Thanks to the use of this package, a more complex finite state machine has been implemented than the one in Figure 4.5. The state machine allows the synchronization of the two robots and manages all the movements and the actions that must be performed to complete the task. It is composed of different states in which different actions are performed, starting from waiting for the two robots until they are ready and until the execution of all the operations necessary to complete the task. The implemented state machine is described in Figure 4.14 thanks to the use of the smach_viewer package. The package allows to displays the smach state machine that is running on the system.
The states are:

1. INIT: Initial state of the system which monitors the topic /system_state. When it receives the message "Ready", the state changes to SYS_READY.

2. SYS_READY: In this state both e.DO and the AGV are ready to perform the task. The state monitors the topic /task and waits for an external "Start" message before to start the task. Then the state changes to WAIT_AGV.

3. WAIT_AGV: This state waits the AGV until it reaches the target position in the Release zone and is recognized from the vision system. The state monitors the topic /agv_detected and it changes when the message "True" is received, meaning that the AGV has been recognized.

4. WAIT_AGV_TARGET_POS: In this state the AGV has been detected and its position is corrected. The state waits until the AGV is correctly positioned in a lateral position with respect to the center of the camera, in order to let the boxes clearly visible. When a "Ready" message on the topic /agv_position is received, the state changes to AGV_READY.

5. AGV_READY: This state starts the movement of e.DO toward its target position with the AGV ready and positioned correctly in the target point. The state publishes the message "move_to_target" on the topic /edo_movement in order to start the movement.

6. WAIT_EDO_TARGET_POS: The state monitors the topic /edo_position and waits for a "Ready" message from the edo manager node meaning that the movement has been completed and e.DO has reached the e.DO target point.

7. EDO_TARGET_POS: The state starts the movement of the gripper on the correct box by publishing the message "move_gripper" on the topic /edo_movement.

8. WAIT_GRIPPER: The state monitors the topic /edo_position until it receives the message "Ready" meaning that the movement has been completed and the gripper is positioned on the target box.

9. GRIPPER_READY: The state starts the release movement on the target box by publishing the message "release_object" on the topic /edo_movement.

10. WAIT_RELEASE: The state monitors the topic /edo_position and waits for a "Ready" message published by the edo manager when the movement is complete and the object has been released.

11. END_TASK: It is the final state, indicating that the task is completed. The state publishes the message "Task_completed" on the topic /task for the manager nodes of the two robots. Then e.DO returns in the INIT position and the AGV starts to move to the Initial zone.

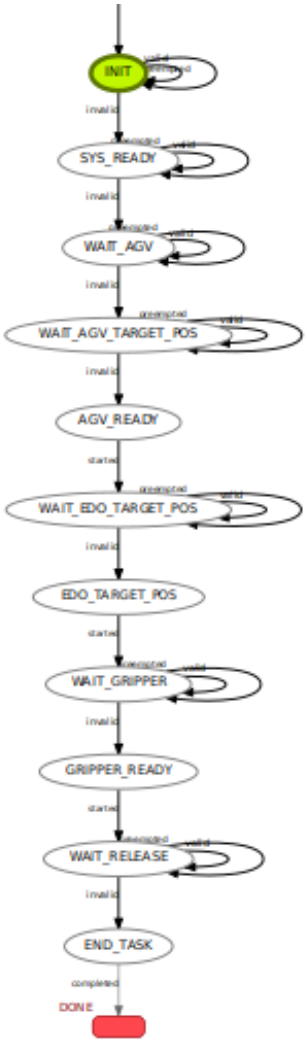

Figure 4.14: Smach state machine

4.8.2 System state node

The System state node manages the state of the system as a whole considering the state of the single robots. In particular, the node subscribes to the topic /machine_state and /agv_state and publishes the overall state of the system on the topic /system_state. /machine_state is the topic on which e.DO publishes its state while the AGV publishes the state on the topic /agv_move. In particular, when a MachineState message is received on the topic /machine_state, the code associated with the current state of e.DO is interpreted as in Table 3.10, while

the AGV publishes its state with a String message on the topic /agv_state. So, considering the current state of both the robots, the node publishes with a rate of 10 Hz the overall state of the system with a String message on the topic /system_state. The possible messages are shown in Table 4.2:

State of the system
System _Ready
Not-ready
Agv _movement
Edo _movement
System _error
System _busy

Table 4.2: System states

4.8.3 Edo Manager node

Edo manager is the ROS node that manages communication with e.DO and controls its movements during the execution of the task, Fig. 4.15. The communication between the node and e.DO occurs mainly through two topics made available by the edo_core_package package:

- /machine_state
- /bridge_move

/machine_state is the topic on which e.DO publishes its state and is therefore used to determine the current state of e.DO and synchronize the various movements during the task. /bridge_move instead, is the topic used to send movement requests to e.DO both in the joint space and in the cartesian space.
The communication with the state machine takes place via three main topics: /task, /edo_position and /edo_movement.
The state machine sends movement requests to the e.DO manager node via the topic /edo_movement, while the topic /edo_position is used by the node to signal the completion of the movement to the state machine. The topic /task instead is the topic in which the state machine sends the message "Task_complete" to signal to the node that the task is complete and e.DO can return to the Init position. Therefore, the node subscribes to the topics: /edo_movement, /machine_state, /task and publishes messages on the topics /edo_position and /bridge_move.

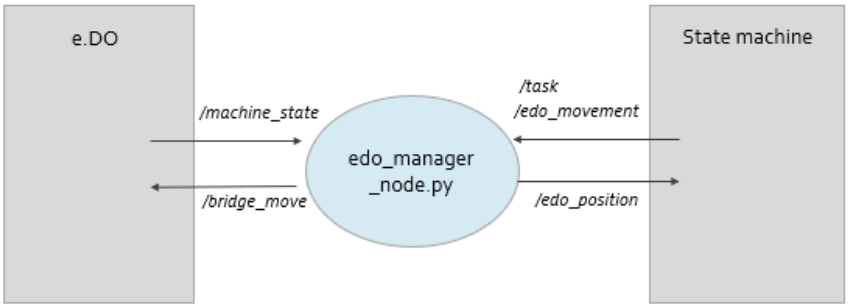

Figure 4.15: Edo manager

In particular, the state machine sends the movement requests to the node with different types of String messages on the topic /edo_movement which are:

- move_to_target
- move_gripper
- release_object

Then the callback function of the topic /edo_movement interprets and manages each move request by invoking the functions that allows to perform the movement. move _to_target is the request related to the movement of the gripper towards the e.DO target point, move_gripper refers to the movement of the gripper on the target box exploiting the information of the camera, while release_object represents the request of the piece release operation when the gripper is correctly positioned on the target box. As previously mentioned in sections 4.6 and 4.7, the move_to_target and release_object requests are associated with a MovementCommand message in the joint space, while the move_gripper request is relative to the execution of a cartesian movement using the camera information with cartesian type MovementCommand messages. In order to do that, the Edo Manager node subscribes to the topic /camera/image_raw related to the camera images published by the camera_node, and controls the positioning of the gripper.
The callback of the topic /camera/image_raw, in fact, implements the different strategies discussed in section 4.7 considering both the camera configurations and the different image-processing techniques. In particular, the OpenCV library was used, which is one of the most famous libraries employed in vision systems and represents the state of the art for image processing. Each frame is then processed to identify the gripper or the object and the target box according to the used strategy. Considering the camera-to-hand configuration, different transformations

are applied to the image that allow to identify the gripper/object and the target box. In general, regardless of the type of the object to be identified, an approach similar to the one described in [25] has been implemented. For that purpose the following transformations and functions provided by the OpenCV library are applied sequentially [26]:

1. cvtColor() [27]: Method used to convert the captured image to another scale of colors, in this case from BGR (blue, green, red) to HSV (hue, saturation, value).

2. inRange() [27]: Method that allows to filter the colors in the image between a min and max threshold that corresponds to the object.

3. findContours() [28]: Method used to find contours in the filtered image.

4. contourArea() [29]: Method used to calculate the area of the contours and therefore to verify if the area of the contour is in the range specified of the object.

5. moments() [29]: Method used to get the coordinates of the contour.

For the first approach, the transformation chain is used to identify the gripper and the target box. The same transformations have been used for the second approach with the difference that the position of the gripper is estimated by identifying the object enclosed. Then the transformations are used in order to allow the identification of the object enclosed in the gripper and the target box. Finally, the last approach uses the chain of transformations for the identification of the target box while using a cascade classifier [30] for the detection of the object to be released. In particular the function detectMultiscale() [31] of the OpenCV library is used allowing to recognize objects in the image that are previously defined in the configuration of the classifier. Once the objects are identified in the frame, the error between the found coordinates is calculated and a cartesian movement is applied to align the gripper to the box. On the other hand, considering the camera-in-hand configuration, the transformation chain is applied only to identify the target box and the error is calculated between the found coordinates and the center of the box. Then a cartesian movement of the gripper is performed so that the box is centered inside the camera. Therefore, once the error has been calculated, regardless of the type of approach, a cartesian MovementCommand message is published on the topic /bridge_move as in section 4.7.

When the requested movement has been completed, a "Ready" message is published on the topic /edo_position to inform the state machine. For this purpose the node controls the completion of the movement through the callback of the topic /machine_state, which is suitably activated. In fact, when a MachineState

message is received, the node monitors the transition from the Ready state to the Move state and until it returns to the Ready state, where the completion message is published.

4.8.4 AGV Manager node

The AGV manager node is the ROS node that manages the communication with the AGV and controls its movements. The main functions of the node are:

- Recognition of the AGV when it arrives in the Release zone
- Correction of the AGV position with respect to the camera

The node subscribes to the topics /objects, /camera/image_raw, /task, /laser_scan and publishes instead on the topics /agv_move, /agv_detected, /agv_position. The arrival of the AGV in the Release zone and its positioning in front of the camera are recognized thanks to the information published by the find_object_2d node on the topic /objects. In fact, the find_object_2d node performs a real time recognition of the AGV in the image provided by the camera and publishes with a fixed rate the code of the recognized object (in this case the AGV) and its coordinates inside the image. The AGV Manager node then, using the information published on the topic /objects, checks it that the AGV has been recognized and monitors its position within the image until the coordinates become constant. This means that the AGV has reached the target point; the message "True" is then published on the topic /agv_detected to inform the state machine. Furthermore, when the AGV is recognized and for the duration of the position monitoring, the node checks the AGV distance from the camera thanks to the information published by the laser. If the distance constraint associated with the target point is not respected, an error message is published on the topic /agv_move. When the AGV has been identified and reported to the state machine, the node performs the correction of the position to let the AGV be positioned in the center of the camera with the boxes clearly visible. The correction of the position is implemented in the callback associated with the camera topic /camera/image_raw. The current position of the AGV in the frame is estimated with the position of the wheel, which is identified using the functions provided by the OpenCV library used in subsection 4.8.3. Then, once the position has been identified, the error is calculated with respect to the center of the camera on the width dimension and a message is published on the topic /agv_move containing the information on the movement to be performed. This message will be managed by the ROS node on the AGV responsible for the motion of the robot. When the positioning at the center of the camera is completed, a "Ready" message is published on the topic /agv_detected to inform the state machine. Finally, when the message "Task_completed" is received

on the topic /task, the node forwards the message to the management node on the AGV by publishing the message "complete" on the topic /agv_move.

4.9 Other packages and Libraries

During the development phase different ROS package have been used:

- edo_core _package
- SMACH
- opencv _vision
- find_object _2d _

edo_core _package is the package provided by Comau that controls the motion of e.DO (see section 3.4). The SMACH package contains a ROS-independent Python library to build hierarchical state machines [32]. The package allows to design complex state machines with maintainable and modular code [32]. Moreover, thanks to the easy python syntax it allows fast prototyping [32]. opencv _vision is a package that allows to interface ROS with OpenCV [33] providing a wrapper of the OpenCV library for ROS. It provides the package cv_bridge that allows to convert the ROS Image messages to OpenCV images and the package image_geometry that contains a set of methods to interpret the images geometry [33]. Finally, the package find_object _2d _allows to use the "Find-Object" application in ROS. The application detects the objects from the images captured by the camera and publishes messages on a ROS topic with id and position of the object in the image [34][35].

Chapter 5

Gazebo simulation

5.1 Introduction

The entire system has been simulated in Gazebo with a differential drive mobile robot as collaborative assistant for e.DO. The simulation made it possible to test all the functionalities of the ROS package in a simulated environment. Gazebo is a very popular 3D simulator that allows to create robots and testing control algorithms for robotic systems in a very realistic scenario for both indoor and outdoor environments [36]. Gazebo is based on OGRE (Open Source Graphics Rendering Engines) for the graphics, to offer a very realistic rendering for environments [36]. It also provides a lot of sensors such as 2D/3D cameras, lasers and contact/force sensors and several plugins that allow to simulate movements. It provides a model editor for creating robots in the SDF format (Simulation Description Format) and a series of robot models already available for the simulation [36]. The integration with the ROS framework takes place thanks to the ROS packages gazebo_ros_pkgs, which provide an interface for the simulation in Gazebo using the services made available by ROS. These packages also allow to use the URDF format as if it were SDF with only the addition of some elements within the XML file [37]. The URDF (Unified Robotic Description Format) is an XML file format used in ROS to describe all elements of a robot, specifying its dynamic and kinematic properties [38]. More in details, the robot is described with a tree structure where the nodes, called 'links', are the rigid parts of the robot and the connections between them are the joints. Of particular importance is the Xacro file, which is an XML file format that is based on functions called macros that simplify the creation of the robot description thanks to a structured text for the URDF file. To view an URDF file in Gazebo it is necessary to make some changes to the file by introducing different tags associated with the links and joints of the robot in order to be compatible with the simulation environment and equivalent respect to the SDF file. The main

difference is that the basic URDF file only describes the kinematic and dynamic characteristics of the robot while the SDF file provides a complete description of both the robot and its pose within the environment. In particular, in order to simulate the robot in Gazebo, it is necessary to make the following changes to the URDF file [38]:

- Configuration of the inertia element within each link
- Introduction of the gazebo element for each joint
- Introduction of the gazebo element for each link
- Introduction of the gazebo element for the robot element
- Introduction of the world link to attach the robot to the world

There are several ways to start gazebo and then load the generated robot model within the simulation environment [39]. The method used is based on ROS and in particular on the roslaunch command, which is the main tool used to launch ROS nodes. It starts Gazebo, open a world model and spawn the e.DO robot into the simulated environment storing URDF files in ROS packages [39].

5.2 e.DO simulation

The model already available on the Comau github repository [40] has been used to simulate e.DO. In particular, the package eDO_description [41] was used, which is a fork of the official package released by Comau [42] that contains the official URDF model of e.DO. The package consists of several files including the launch file for uploading the model to the ROS parameter server, the mesh files and a series of xacro files that generate the URDF model: The edo.urdf.xacro file is the main file that includes all the macros and the URDF model contained in the edo.xacro file. In particular, the edo.xacro file and the macros utilities.xacro and materials.xacro contain all the characteristics of the robot in terms of dynamic and kinematic properties and describe the graphic and physical aspects. edo.gazebo.xacro contains all the elements and the tags needed for the simulation and to get the robot URDF properly working in Gazebo. The transmission.xacro file, instead, allows to control the joints of the robot specifying the controller and the hardware interfaces. The edo_gazebo package of the github repository [43] was used for the Gazebo simulation. The package contains the launch files to simulate e.DO in Gazebo with or without gripper based on the package mentioned above [41] and the yaml files for configuring the controllers that allow the movement. For the motion planning into the simulated environment, the Moveit [44] platform was used, not having the Comau library available. In particular, the packages eDO_moveit [45] and

edo_gripper_moveit [46] available on the github repository [47] were used, which allow to plan and execute the trajectory of the robot and the gripper using the planning provided by Moveit. Moreover, the edo_gazebo package mentioned before has been specifically configured to be used with Moveit jointly with the package eDO_moveit [45]/edo_gripper_moveit [46], by providing the controllers and yaml files that allow to perform the planned trajectory.

5.2.1 MoveIt

Moveit is an open-source platform for the creation and manipulation of different types of robots. It allows to plan the trajectory in the environment with a collision control to avoid obstacles [44]. It allows to solve the inverse/direct kinematics and to perform joint trajectories with different types of controllers and hardwarte interfaces. It also integrates very well with Gazebo and ROS control in order to have a complete development platform [44]. The Rviz Motion Planning plugin allows to perform an interactive trajectory planning by dragging the end effector to the desired position and to view the planned trajectory in the space. MoveIt setup assistant [48] is the main tool of MoveIt that allows to initialize and configure the robot in order to plan and execute the desired trajectories. It provides a graphical user interface for configuring the robot for use with MoveIt. Starting from the URDF file, it generates the SRDF (Semantic Robot Description Format) file and other configuration files for use with the MoveIt pipeline [48]. In the configuration step it is possible to specify the virtual/fixed joints, planning groups, fixed poses of the robot and the informations of the sensors used for the 3D perception. Furthermore, at the end of the configuration phase, it is possible to choose to automatically generate the compatible robot model for the simulation in Gazebo and to define different types of controllers belonging to the ROS control [49] package to simulate the controllers for the joints. move_group is the main ROS node of MoveIt which provides a set of ROS actions and services for the users [50], Fig. 5.1. Moveit provides three ways to access the services and features offered by the move_group node [50]:

1. move_group_interface package to interface to the node in C++
2. moveit_commander package to iterface with python
3. Rviz Motion Planning plugin with the GUI

The move_group node is configured through the ROS parameter server where it obtains the URDF model of the robot, the SRDF file created through the Moveit Setup Assistant and other information that includes joint limits, kinematics, motion planning and perception [50]. In general, the move_group node communicates via

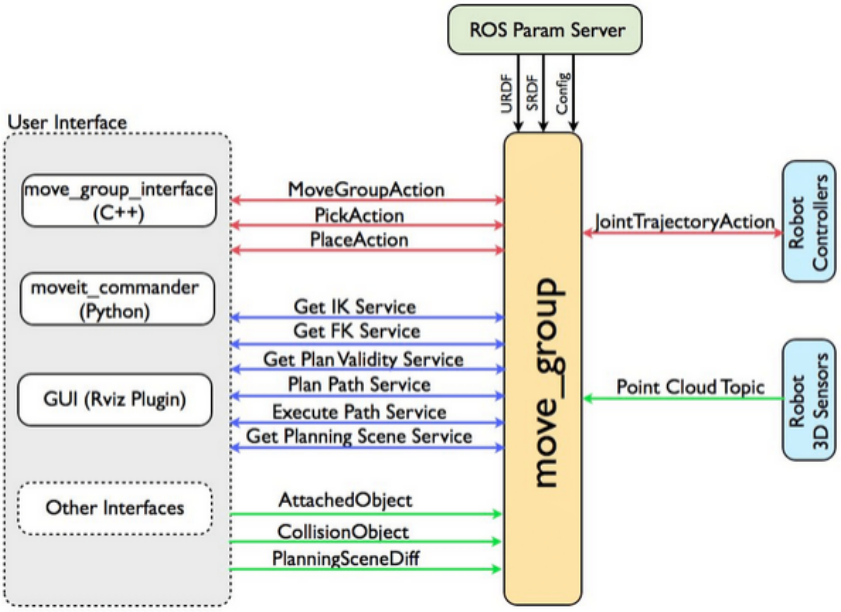

Figure 5.1: MoveIt move_group [50]

ROS topics and actions, obtaining information on the current state of the robot and other general information and sending commands to the controllers through the FollowJointTrajectoryAction interface, that is a ROS action interface [50]. MoveIt uses and communicates with various motion planners through a plugin interface based on a ROS Action/Service provided by the move_group node. The default motion planners for move_group are based on OMPL (Open Motion Planning Library) [51]. The motion plan requests ask to the motion planner to move the robot to a defined target pose or to a different position of the joints. Then the move_group node generates the desired trajectory in order to move the robot [50].

5.2.2 Movement Control

The ros_control package [49] and the gazebo_ros_control plugin [52] that simulates the controllers of the ros_control package [49] are used to simulate the controllers that actuate the joints in Gazebo. The plugin is set inside the file edo.gazebo.xacro and its purpose is to parse the transmission tags associated with each joint of the transmission.xacro file and to load the hardware interfaces and controller manager [52]. The launch file of the edo_gazebo package starts the controllers and loads the yaml file to the parameter server which contains all the PID gains and the controllers setting associated with the hardware interfaces.

In particular, two different types of controllers have been analyzed and used to actuate the e.DO joints and perform the task:

1. position_controllers/JointTrajectoryController
2. position_controllers/JointPositionController

The first type of controller is already provided in the edo_gazebo package and is used as a single controller for all the joints that performs the trajectories planned with MoveIt. This type of controller is associated with the hardware interface "hardware_interface/PositionJointInterface". The second type of controller was used to create a position controller associated with each joint which allows to change its position. The joint position controllers, associated with the hardware interface "PositionJointInterface", receive a goal position and move each of the joints. For the movement of the gripper, on the other hand, the controllers position_controllers/JointPositionControllers made available by the edo_gazebo package have been used, which are associated respectively to the four joints: left/right finger and left/right base. Finally, the edo_gazebo package also defines a controller of type joint_state_controller associated with the hardware interface "JointStateInterface" and that publishes the joint states of the arm.

In relation to the two types of controllers used to actuate the joints, two approaches have been adopted for the execution of the trajectory described in section 4.6. The first approach is based on the use of the first type of controller for the execution of the trajectory generated by MoveIt both for the movement of e.DO towards the e.DO target point and for the positioning of the gripper on the target box. In order to do that, the python package moveit_commander [53] was used to interface with the ROS node move_group of MoveIt provided by the packages eDO_moveit and edo_gripper_moveit. The python package allows to plan and execute the trajectory both in the joint space and in the cartesian space. Therefore, exploiting the functions provided by the package and considering that e.DO is in the initial configuration before to execute the task, it was performed a joints movement with respect to the e.DO target point and a cartesian movement towards the target box above the AGV. The second approach is based instead on the use of the position controllers associated to the different joints. In this case, MoveIt was not used for planning the trajectory, but a reference position was directly assigned to the various joints. Therefore, for the first movement, the position of the joints relative to the e.DO configuration in the e.DO target point was assigned, while the cartesian movement on the target box was approximated by the movement of the first joint until the gripper is positioned on the target box. This approach is justified by the fact that the size of the boxes is very small compared to e.DO, thus making the resulting movement of the gripper almost cartesian. So the edo_manager node described in 4.8.3 has been modified in order to have both

the commands that allow movement within the simulation environment and the commands of the package edo_for the real movement.

5.3 AGV simulation

For the simulation of the AGV, the URDF model of the differential drive mobile robot has been created following the indications described in [54]. The built URDF model is contained in a xacro file which contains also all the tags required for the simulation in Gazebo. For the construction of the boxes positioned above the robot, the model editor made available by Gazebo has been used. Considering the structure of the mobile robot described in the URDF model, the boxes have been built in the editing environment and subsequently converted into links connected later to the URDF model. The differential_drive_controller plugin was used to allow the movement within the simulated environment, Fig. 5.2. The differential drive plugin is a model plugin that provides a basic controller for differential drive robots in Gazebo [55]. The plugin subscribes to the topic /cmd_vel where it receives messages of type Twist and publishes location information on the topic /odom. The plugin drives the robot model according to the velocities of the Twist messages received and as the robot moves, it publishes odometry information to the topic /odom [55]. To move the robot, a specific python script has been

```
<gazebo>
    <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <leftJoint>joint_left_wheel</leftJoint>
      <rightJoint>joint_right_wheel</rightJoint>
      <wheelSeparation>0.4</wheelSeparation>
      <wheelDiameter>0.2</wheelDiameter>
      <torque>0.1</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <robotBaseFrame>link_chassis</robotBaseFrame>
    </plugin>
  </gazebo>
```

Figure 5.2: Differential drive plugin [55]

developed that controls the mobile robot in order to go to the desired position in correspondence of the AGV target point.

5.4 Simulation of the task

A camera and a laser sensor have been introduced into the simulated environment. The camera has been positioned on the base of e.DO, while the laser sensor has been placed on the ground near e.DO in order to check if there are objects near the robot, and hence to check if the AGV has arrived or not thanks to the information provided by the camera. Then the camera is used to recognize the AGV and to coordinate the release movement, while the laser sensor is used to measure the distance of the AGV from e.DO in order to check if the AGV is positioned correctly in the target Release Zone. Therefore, to introduce the camera and the laser sensor, two extra links have been created into the environment, associated to the plugins provided by Gazebo [55] for the simulation of sensors:

- camera controller plugin
- gazebo_ros_head_hokuyo_controller plugin

The camera controller plugin provides a ROS interface for the simulation of the RGB camera by publishing the CameraInfo and Image ROS messages of the sensor_msgs package [55]. As shown in Figure 5.3, when setting the plugin it is possible to specify the update rate of the images, the dimension of the camera frame, the format and the ROS topics where the images and the camera info are published. The gazebo_ros_head_hokuyo_controller plugin simulates the laser range sensor by broadcasting LaserScan messages of the package sensor_msgs [55]. In the plugin setting described in Figure 5.4 are specified all the properties of the sensor, such us resolution, min/max angles, ranges and the ROS topic where the plugin publishes the information provided by the laser.

In addition, in order to perform the task inside the simulated environment it has been used the gazebo_grasp_plugin available on the github repository [56]. The plugin allows to grab/drop objects with the gripper in order to avoid strange behaviors of the robot or that the object may slip off the robot hand [57]. In particular, the plugin fixes the object to grab to the gripper in order to avoid problems with physics engines of Gazebo [57]. The object is grasped as soon as two opposing forces are applied by the gripper links on the object, making the object fixed to the robot hand. As soon as the gripper opens or the two forces are no more applied, the object is detached again [57].

5.4.1 Configuration phase

Before to simulate the system and test the functionalities of the Edo-AGV_Integration package developed, it is necessary to perform an initial configuration phase to set up and initialize the system. In particular, the configuration phase consists of the following steps:

```
<gazebo reference="camera_link">
  <sensor type="camera" name="camera1">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <!-- Noise is sampled independently per pixel on each frame.
             That pixel's noise value is added to each of its color
             channels, which at that point lie in the range [0,1]. -->
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>0.0</updateRate>
      <cameraName>rrbot/camera1</cameraName>
      <imageTopicName>image_raw</imageTopicName>
      <cameraInfoTopicName>camera_info</cameraInfoTopicName>
      <frameName>camera_link</frameName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
</gazebo>
```

Figure 5.3: Camera plugin [55]

1. Definition of the Release zone and of the known e.DO target point and AGV target point
2. Definition of the reference values for the object detection functions
3. Setting of the find_object_2d recognition node to recognize the AGV

In order to define the e.DO target and AGV target points, the Rviz Motion

```
<gazebo reference="hokuyo_link">
  <sensor type="gpu_ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>false</visualize>
    <update_rate>40</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.570796</min_angle>
          <max_angle>1.570796</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.10</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <!-- Noise parameters based on published spec for Hokuyo laser
             achieving "+-30mm" accuracy at range < 10m.  A mean of 0.0m and
             stddev of 0.01m will put 99.7% of samples within 0.03m of the true
             reading. -->
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_gpu_laser.so">
      <topicName>/rrbot/laser/scan</topicName>
      <frameName>hokuyo_link</frameName>
    </plugin>
  </sensor>
</gazebo>
```

Figure 5.4: Laser plugin [55]

Planning plugin made available by MoveIt was directly exploited. Then, by moving the gripper within the environment provided by the graphical interface, it is possible to determine the e.DO target point within the workspace, so that it can be reached by e.DO avoiding singularity points. Consequently, the AGV target point has been defined and the camera has been positioned in order to have the same orientation of the AGV when it is in lateral position. For the definition of the reference values for the object detection functions and to set up the find_object_2d node, the release scene was recreated by positioning the AGV in front of the camera at the AGV target point. In particular, it has been created a python script that exploits the OpenCV functions described in subsection 4.8.3 in order to identify the range of colors and the area of the object to be identified within the image provided by the camera. Then, the reference values used by the edo_manager node to identify the object to be released and the target box have been set. The graphical interface provided by the application was used instead to set the find_object_2d

node, in order to capture the image in which the AGV is present and extract its characteristics. Then, an initial learning phase was performed in order to recognize the characteristics of the AGV in the image provided by the camera during the recognition phase, Fig. 5.5.

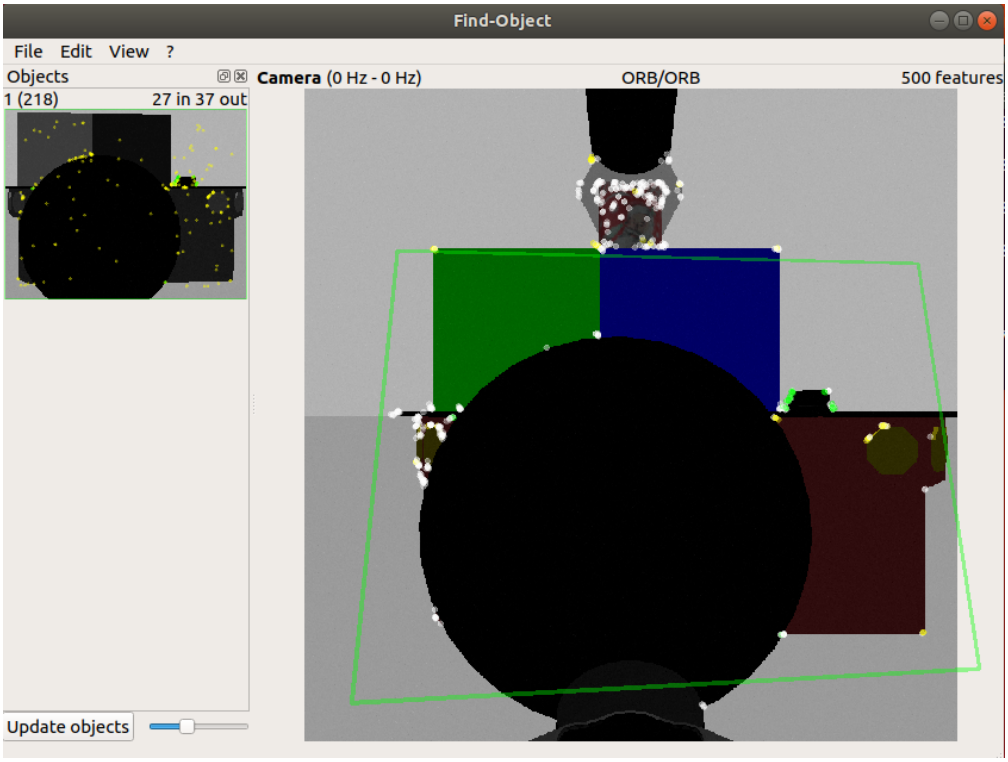

Figure 5.5: Learning phase of the Find Object application [35][34]

5.5 Results

In the simulation tests, only the configuration of the camera on the base was considered, using the first two object detection approaches based on the functions provided by the OpenCV library, as described in subsection 4.8.3, being the considered object a simple small ball. In fact, the latest approach which is based on the use of a cascade classifier [30] for the detection of the object, was designed to recognize objects with more complex characteristics that require the use of deep learning techniques for their identification and the use of neural networks for their recognition within the image. Regarding the final position of the AGV within the Release zone, a pose described by a rotation of -90 degrees around the axis z and by a translation of about 70 cm relative to the x axis with respect to the e.DO reference frame was supposed.

In Figures 5.6 - 5.11, the system states during the simulation are shown. In particular, Fig. 5.6 and Fig. 5.7 describe the arrival of the AGV in the Release zone, before and after the correction of the position at the center of the camera. Fig. 5.8 and Fig. 5.9, instead, show the positioning of the gripper at the e.DO target point and on the target box. After the movement of the gripper on the target box, the object is released and e.DO returns to the Init position, Fig. 5.10 and Fig. 5.11.

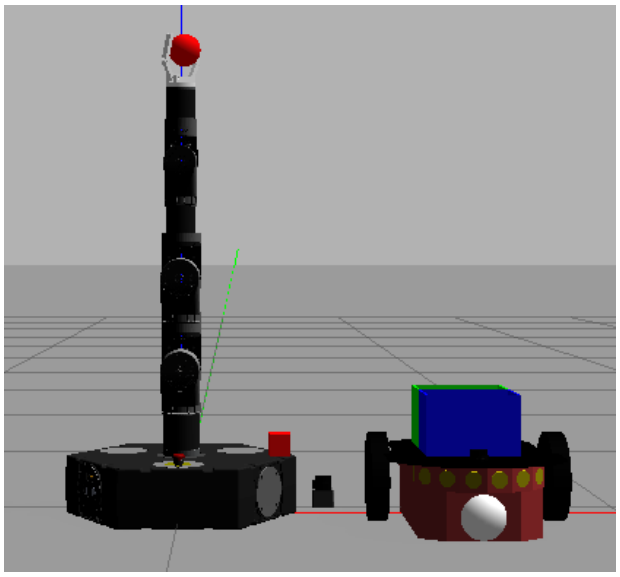

Figure 5.6: AGV arrived in the Release Zone

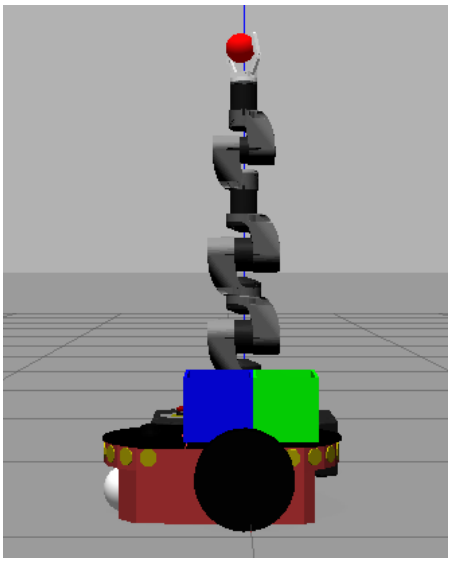

Figure 5.7: AGV correctly positioned

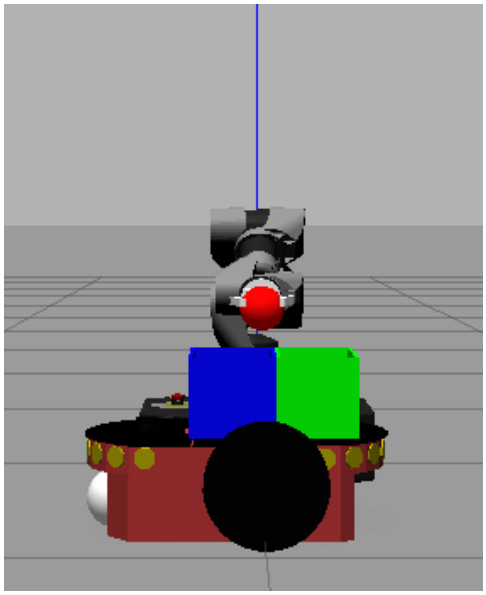

Figure 5.8: e.DO positioned at the e.DO target point

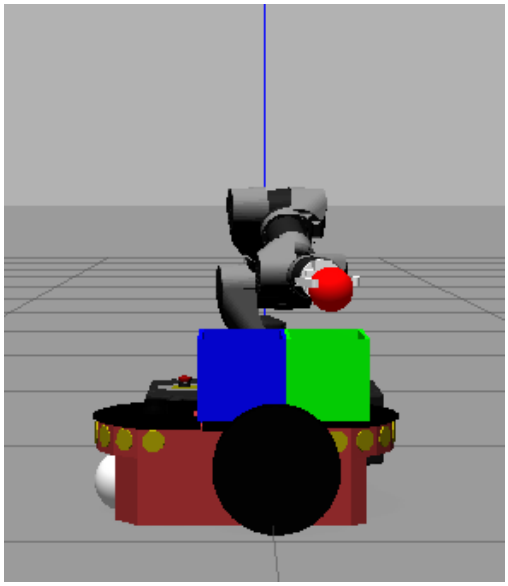

Figure 5.9: Gripper positioned on the target box

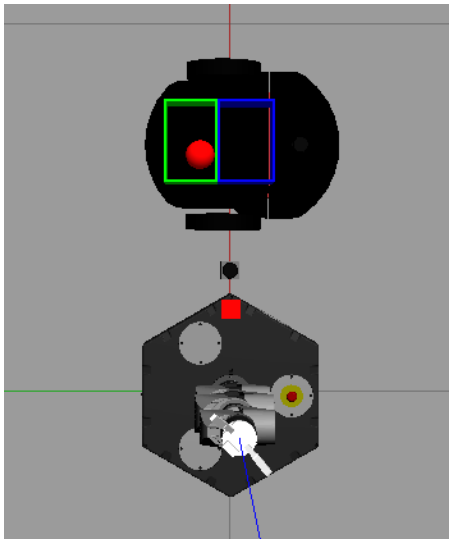

Figure 5.10: Object released in the target box

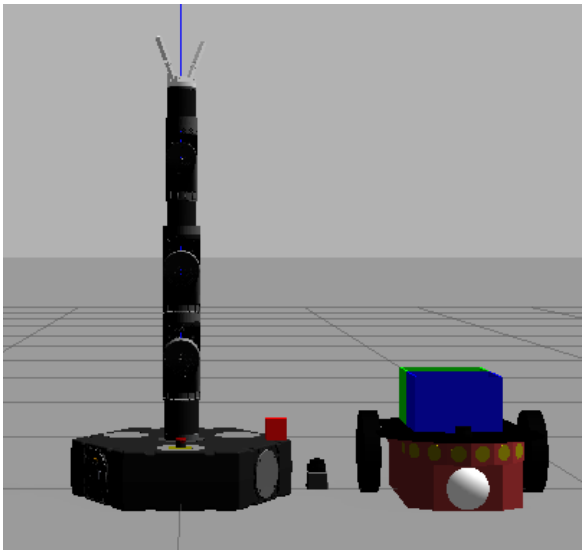

Figure 5.11: e.DO in Init position after the completion of the task

5.5.1 Joint Position Controllers

Object Detection

Figure 5.12 shows the identification of the object and the target box while Figure 5.13 describes the evolution over time of the error using the position controllers. In particular, using a frame rate of 2 frames per second and a fixed increment of 1 cm, the error decreases and converges quickly to the value of 10/15 pixel. Using the topics provided by Gazebo, the position signals of all the joints and the end-effector/object coordinates have been saved during the execution of the task, Fig. 5.15 - 5.20, Fig. 5.14.

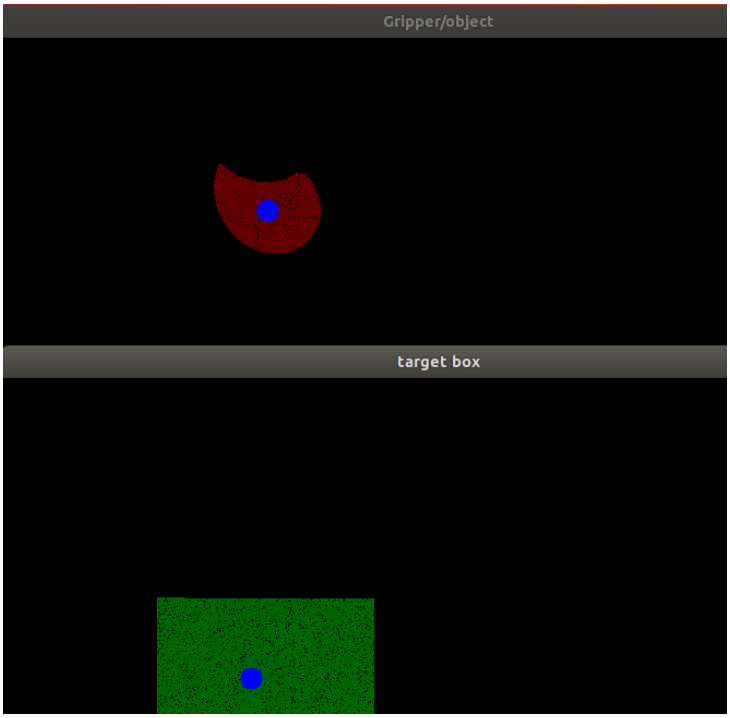

Figure 5.12: Object and target box detection (simulation)

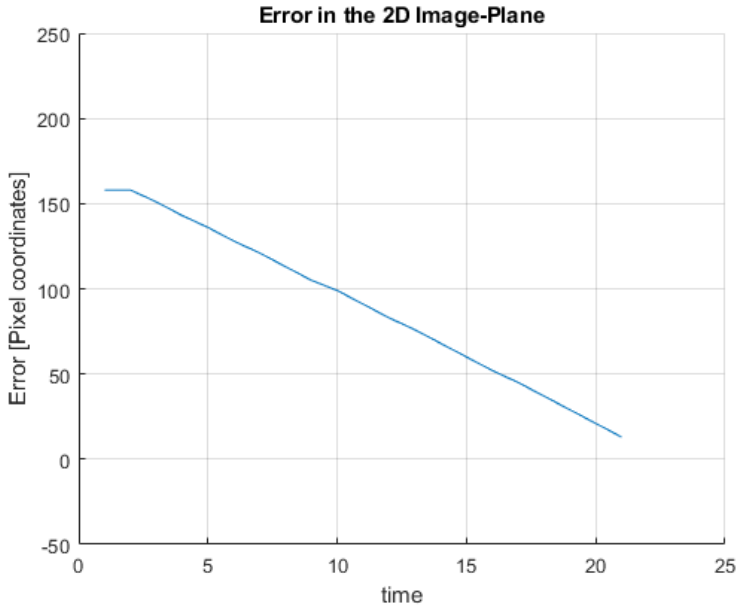

Figure 5.13: Error in the 2D image plane with object detection (simulation)

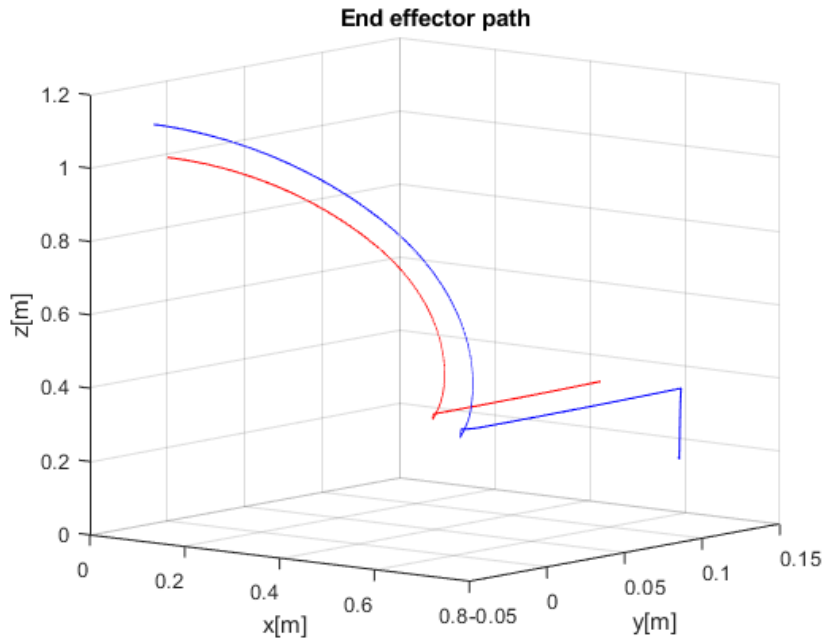

Figure 5.14: End effector and Object paths with object detection

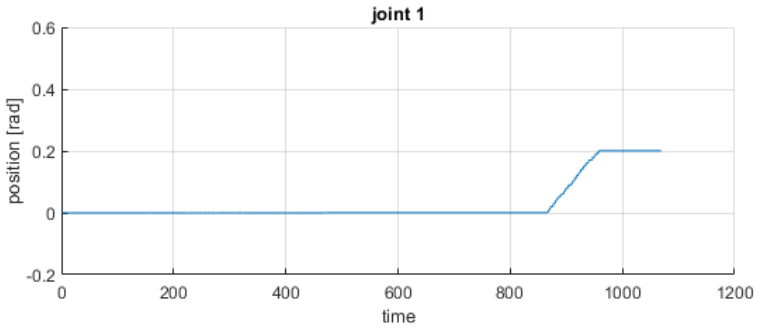

Figure 5.15: Joint 1 - object detection

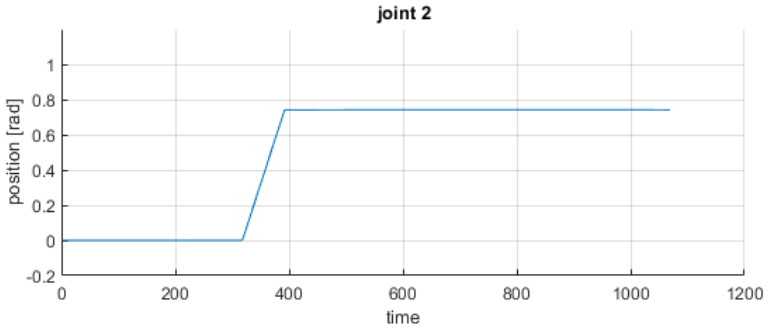

Figure 5.16: Joint 2 - object detection

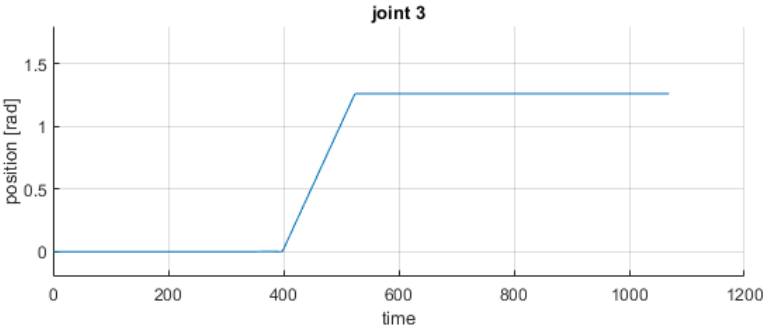

Figure 5.17: Joint 3 - object detection

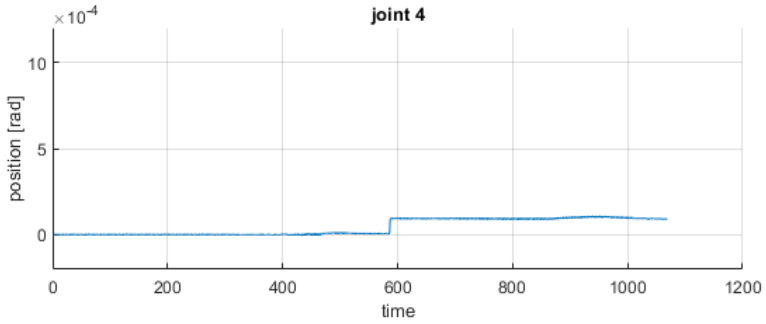

Figure 5.18: Joint 4 - object detection

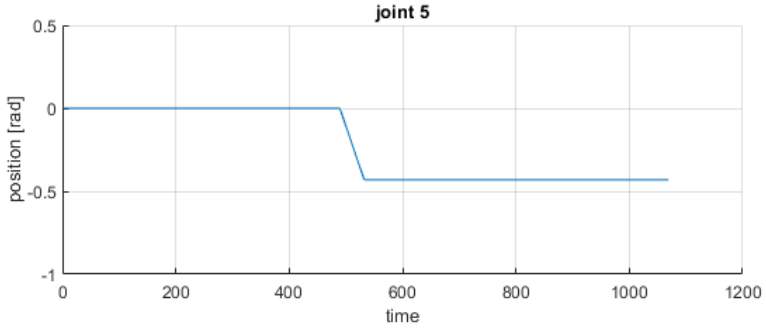

Figure 5.19: Joint 5 - object detection

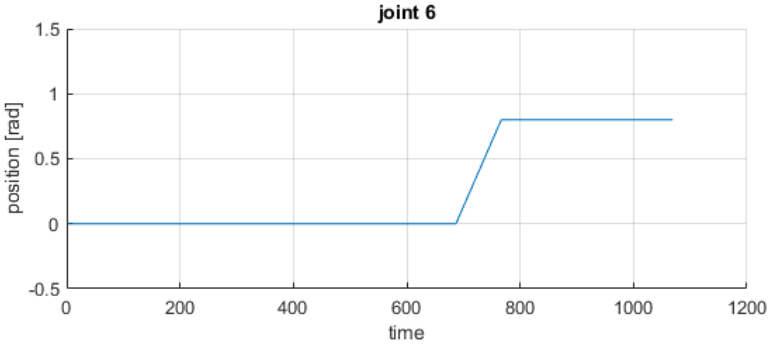

Figure 5.20: Joint 6 - object detection

As can be seen in Figures 5.15 - 5.20, the joint position signals are characterized by a constant initial and final part and a linear intermediate part associated to the variation of the position until the value associated to the e.DO target point is reached.

Gripper Detection

Using the gripper detection approach, Figure 5.21 shows the identification of the gripper and the target box, while the error is shown in Figure 5.21. Using the same frame rate and increment value, the error has the same time evolution and converges to the same value of the object detection approach. Figure 5.23 shows the end-effector/object paths and Figures 5.24 -5.29 show the joint position signals. From the charts, it is possible to see that the position signals of the joints have the same time evolution as before.

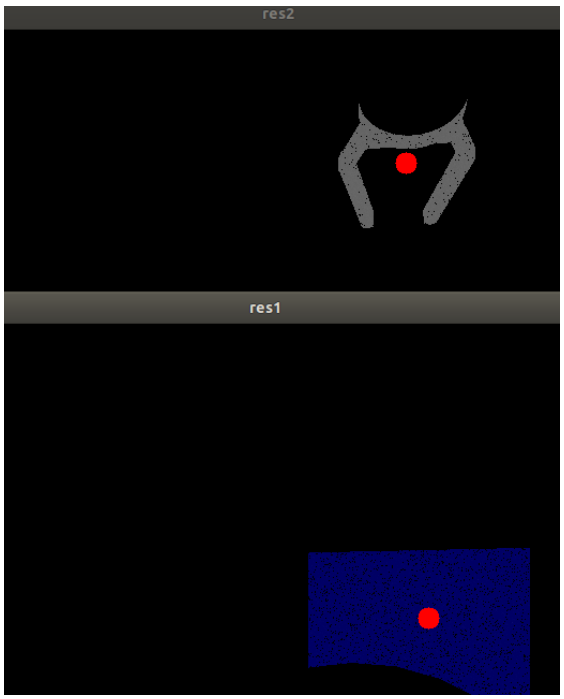

Figure 5.21: Gripper and target box detection (simulation)

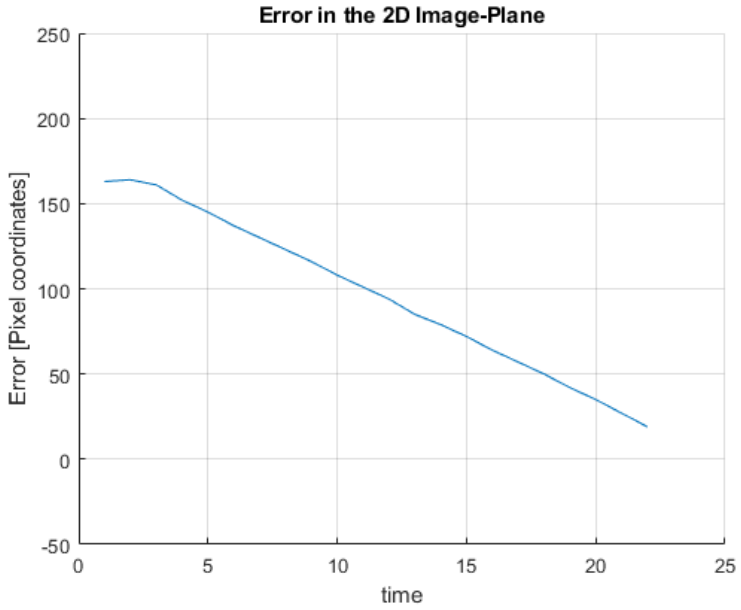

Figure 5.22: Error in the 2D image plane with gripper detection (simulation)

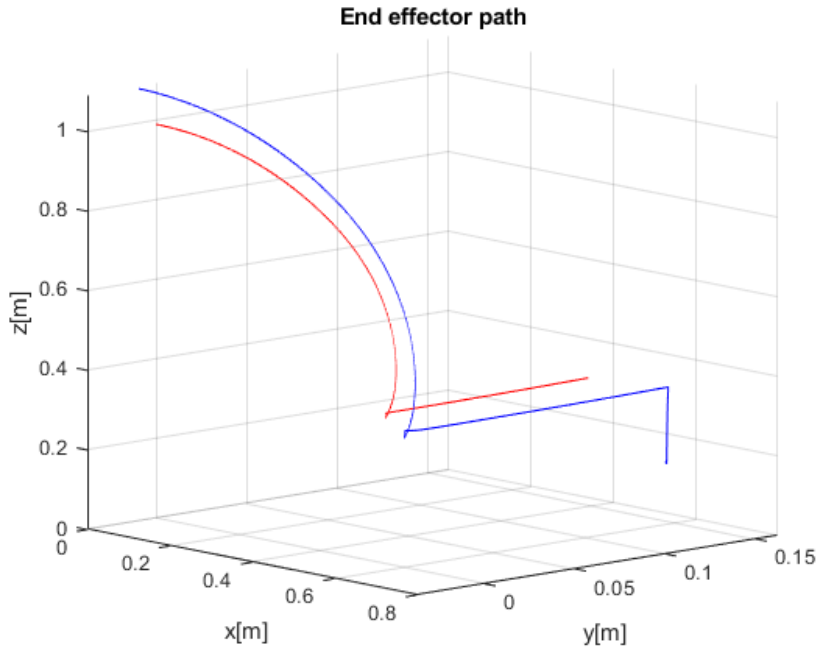

Figure 5.23: End effector and Object paths with gripper detection

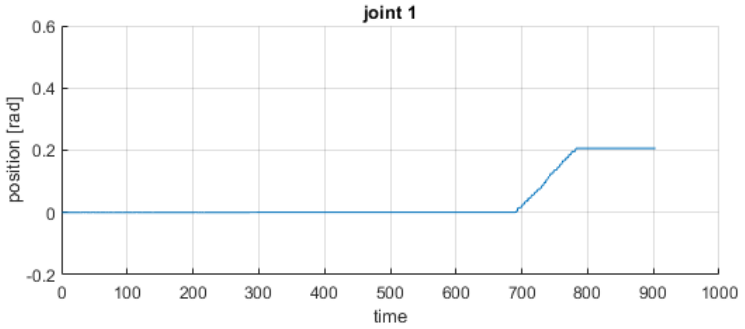

Figure 5.24: Joint 1 - gripper detection

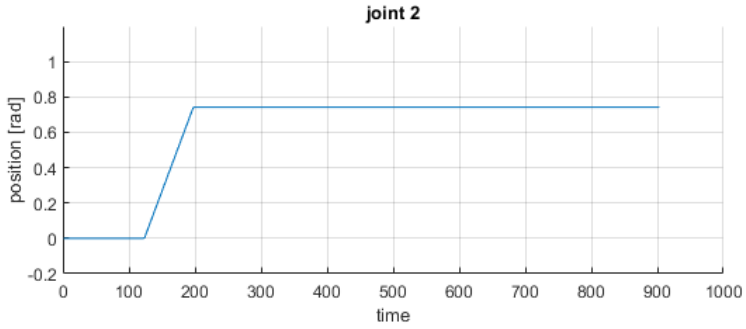

Figure 5.25: Joint 2 - gripper detection

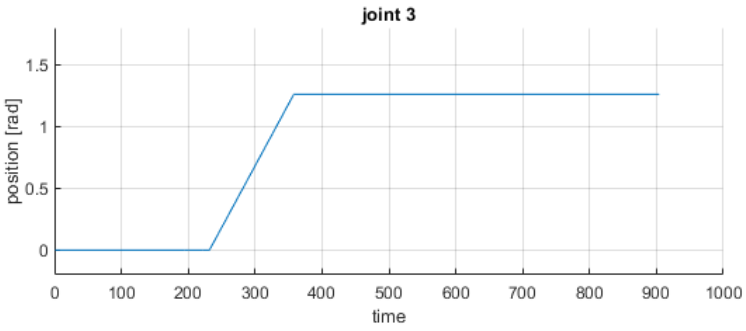

Figure 5.26: Joint 3 - gripper detection

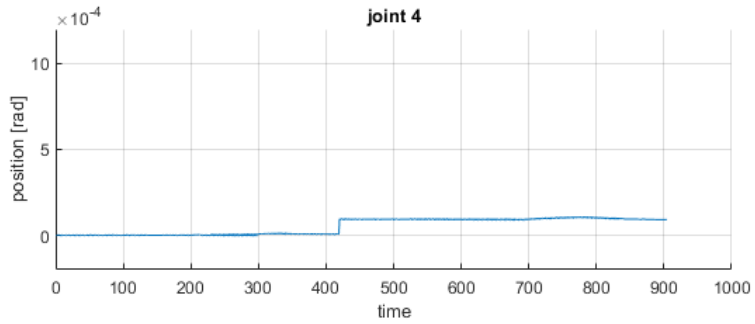

Figure 5.27: Joint 4 - gripper detection

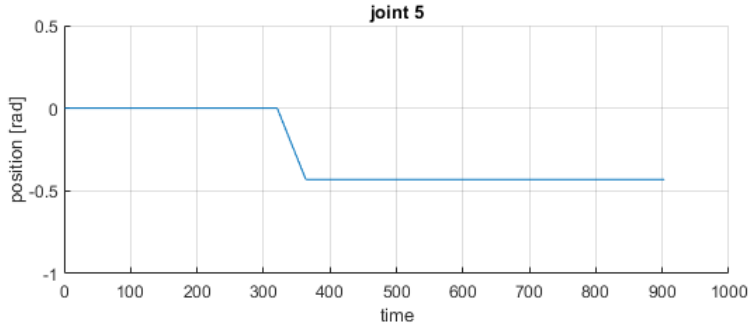

Figure 5.28: Joint 5 - gripper detection

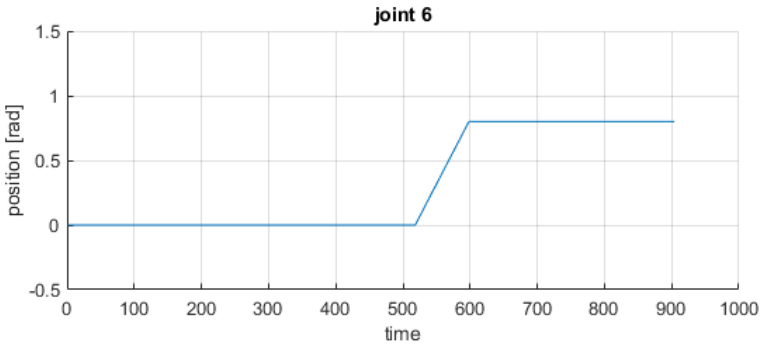

Figure 5.29: Joint 6 - gripper detection

5.5.2 MoveIt - Joint Trajectory Controller

Trajectory with Gripper Detection approach

Figure 5.30 describes the time evolution of the error using the joint trajectory controller for the execution of the trajectory planned by MoveIt. Also in this case, a frame rate of 2 frames per second and a cartesian increment of 1 cm have been used. In particular, the error has a much smoother time evolution and decreases more slowly until to reach the final value of about 20 pixel. Figure 5.31 shows the end-effector path during the execution of the task and Figures 5.32 - 5.37 show the position signals of the joints.

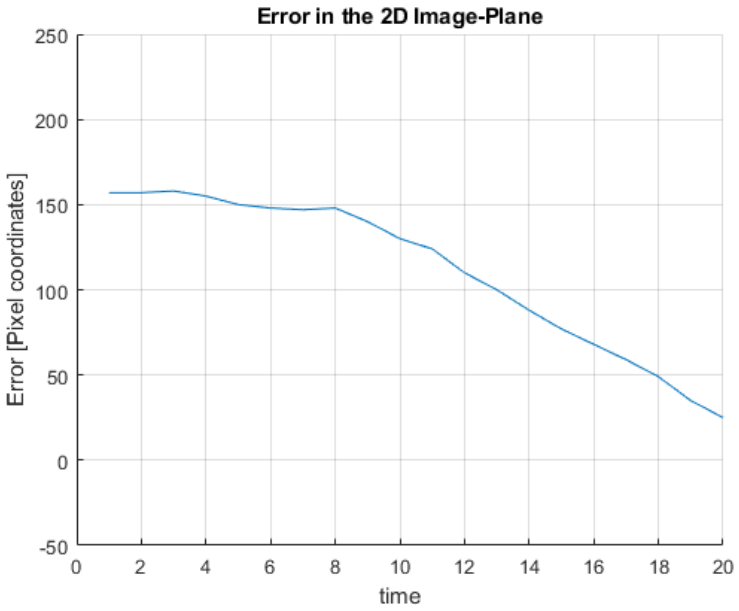

Figure 5.30: Error in the 2D image plane with gripper detection (MoveIt)

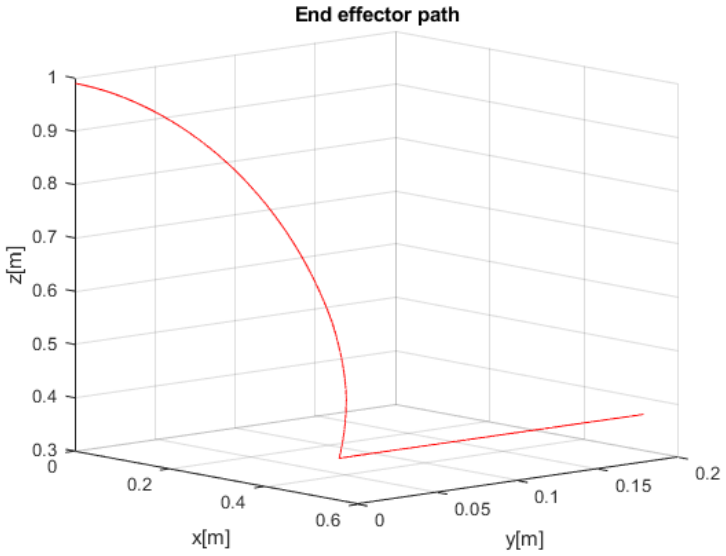

Figure 5.31: End effector path with gripper detection (MoveIt)

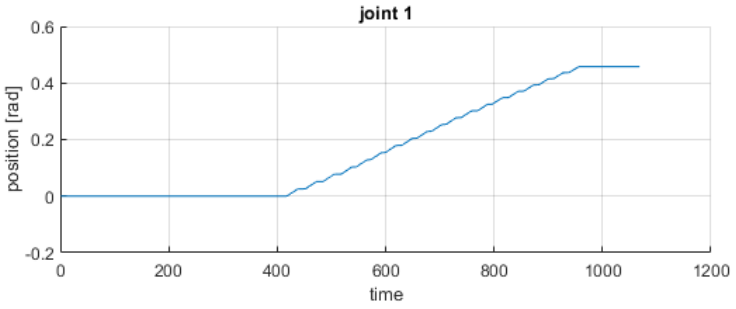

Figure 5.32: Joint 1 - MoveIt

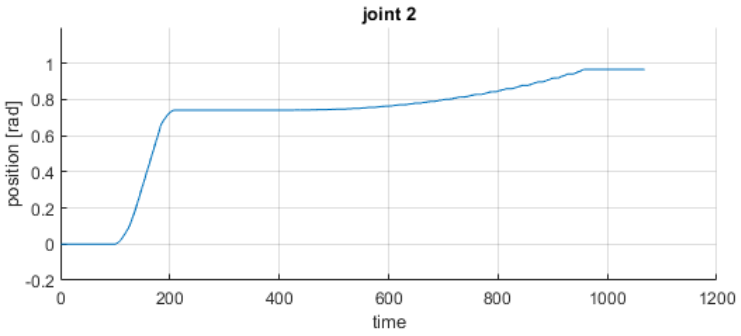

Figure 5.33: Joint 2 - MoveIt

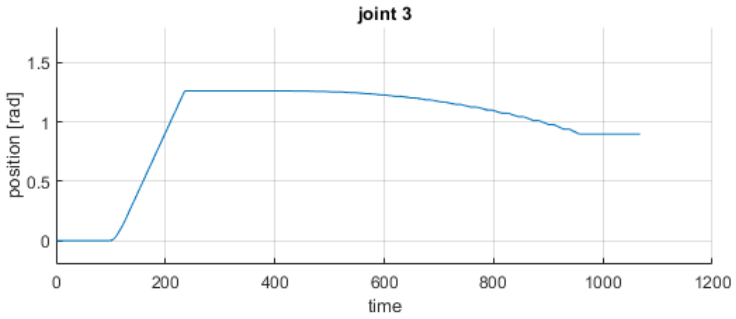

Figure 5.34: Joint 3 - MoveIt

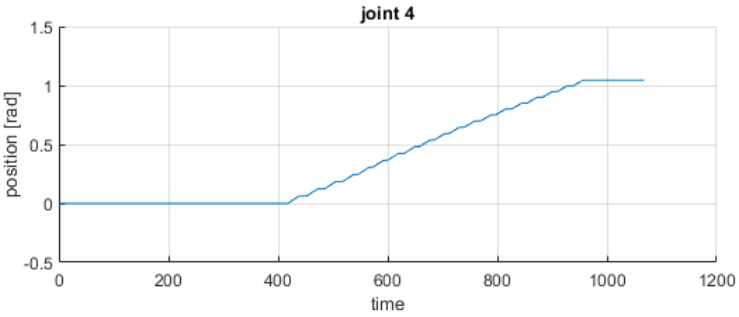

Figure 5.35: Joint 4 - MoveIt

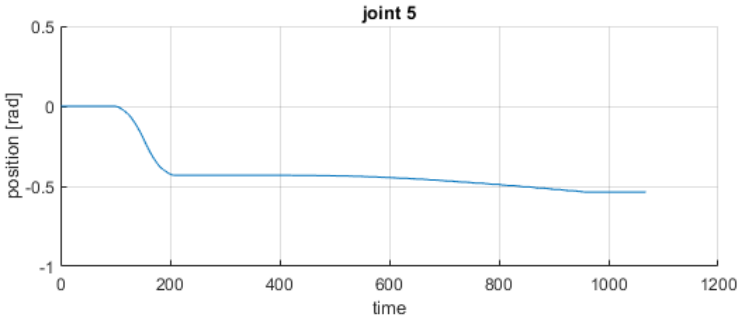

Figure 5.36: Joint 5 - MoveIt

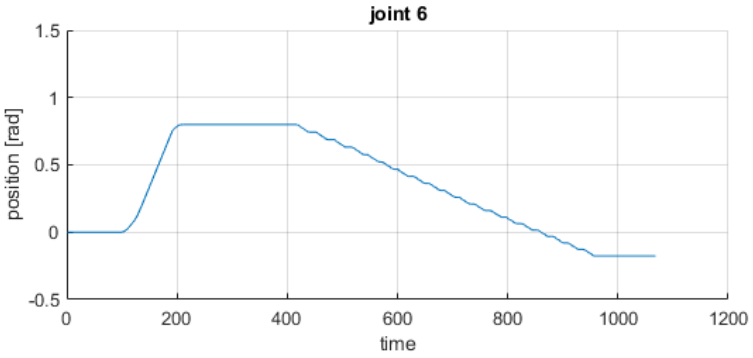

Figure 5.37: Joint 6 - MoveIt

As can be seen from the charts shown in Figures 5.32 - 5.37, the joint position signals have a much smoother time evolution with respect to the signals obtained using the position controllers associated to each joint.

Chapter 6

Experimental Tests

The part of the architecture related to the movement of e.DO has been experimentally tested, as well as the release of the piece inside the box using the USB camera connected to the raspberry. The raspberry pi runs Ubuntu Mate 18.04.2 O.S. with ROS Melodic installed in the base version. It is connected to the camera sensor through the USB cable, while the ethernet connection allows the communication with the e.DO raspberry pi. The camera used for the project is the Logitech C270 HD webcam (Fig. 6.1). The camera has a full HD 720p screen at 30 frames per second; it allows to capture very clear and defined images and it is also equipped with a microphone for audio recording with noise reduction. The technical specifications are shown in Table 6.1. The system was tested with the camera-to-hand configuration and with the object detection approach. In order to do that, two boxes (yellow/blue) of dimension 17x14x12 cm and a small ball were used (Fig. 6.2 and Fig. 6.3). The ball has been used because it is both easy to grasp and to identify using the vision system.

Before testing the ROS package on e.DO, the same initial configuration phase of the simulation in Gazebo was performed, and the same release scene was recreated. As in the Gazebo simulation, the following transformation matrix T was assumed to be known, which determines the pose of the AGV (in this case the boxes) with respect to the e.DO reference frame inside the Release zone:

$$T = \begin{bmatrix} 0 & 1 & 0 & 70 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

The matrix defines a rotation of -90 degrees around the z axis and a translation of about 70 cm relative to the x axis with respect to the e.DO reference frame. For the definition of the known points, the e.DO Android application was used to move the gripper towards such points and save the values according to the position

of the boxes. To define the reference values of the object detection functions, the python script was executed to identify the color range and the area of the object (target box/ ball) within the image provided by the camera. Finally, the calculated values have been used to set the reference values of the ROS node that manages the movement of e.DO. Since only the part of the architecture related to the movement of e.DO (eDO Manager ROS node) has been tested, it was not necessary to configure the ROS node find_object_2d to recognize the AGV.

Figure 6.1: Logitech c270 [58]

Specification	Value
Max Resolution	720p/30fps
Focus type	fixed
Integrate microphone	mono
FoV	60°
USB Cable	1,5 m

Table 6.1: Technical specification [58]

Figure 6.2: AGV Boxes

Figure 6.3: Ball

Figure 6.4 shows the system in the real implementation with the external raspberry pi connected to e.DO with the ethernet cable e to the camera with the USB cable.

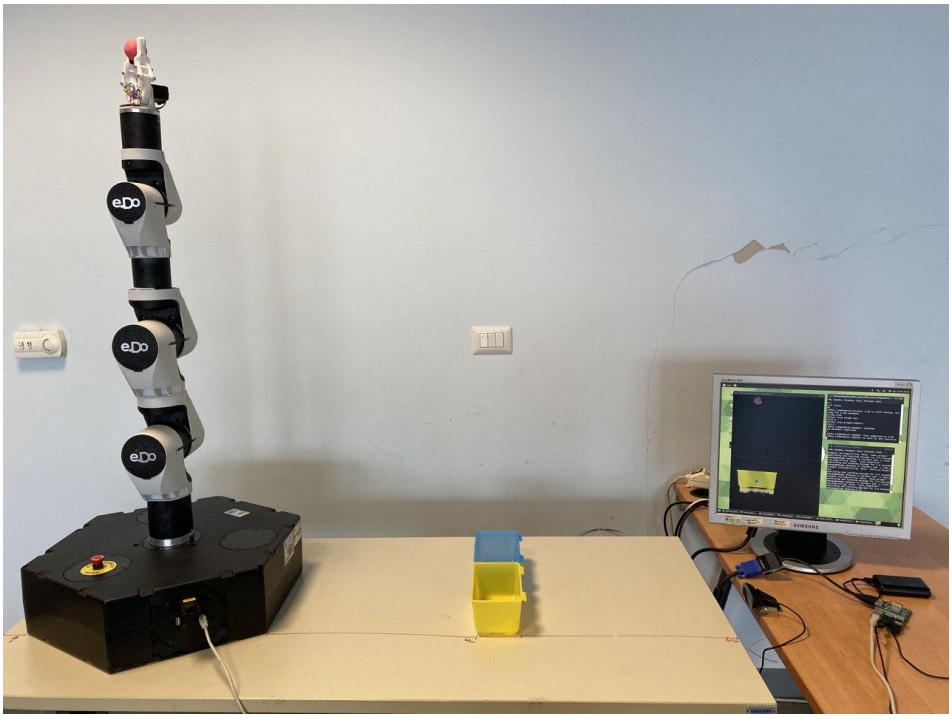

Figure 6.4: System in the real implementation

6.1 Results

A series of preliminary experimental tests have been performed in order to find the best values for the camera frame rate and for the value of the fixed increment associated to the cartesian movement on the box. After such tests, it was found that the best pair of values is 2 frames per second with a fixed increment of 0.35mm, in order to have a frame rate that matches the movement of e.DO. However, the resulting movement turned out to be a bit jerky both by setting the "fast" and "medium" speed during the cartesian movement. For this reason, an additional ROS node has been created and added to the package, which monitors the e.DO topic /machine_state in order to republish the camera images only when e.DO has completed its movement. Then, using the created ROS node, the resulting movement of the gripper was found to be approximately linear with different fixed increment values. In this case, a fixed increment of 0.1 mm proved to be the most

suitable value. Using the topic /usb_joint_state, the position signals of all the joints have been saved during the execution of the task. The values represent the measurements of the encoders of the six joints and are shown in Figures 6.7 - 6.12.

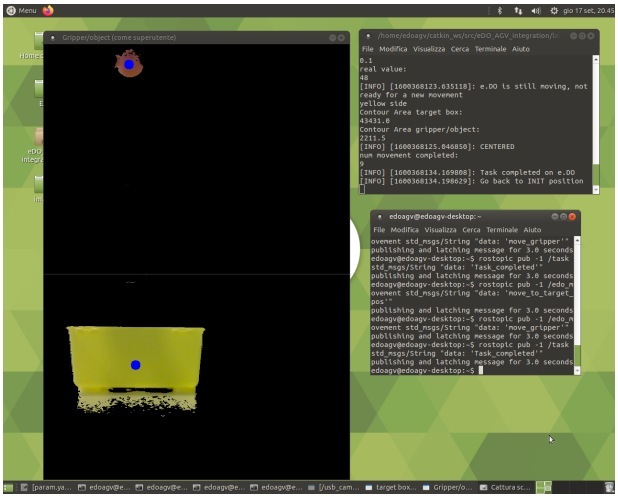

Figure 6.5: Object and target box detection

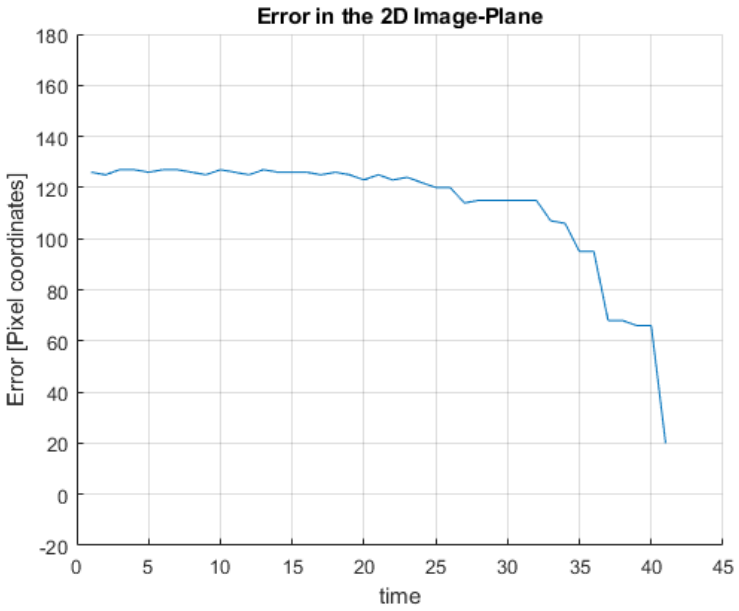

Figure 6.6: Error in the 2D image plane

Figure 6.5 shows the ball and the target box detection, whereas Figure 6.6 shows the variation of the error during the positioning of the gripper on the target box. As it is possible to see in Figure 6.6, the error converges much more slowly compared to the Gazebo simulation, until it reaches the final value of about 20 pixel. This could be due to the fact that e.DO performs the cartesian movement with a fixed increment of 0.1 mm and a "medium" speed corresponding to 50% of the motors speed, and the e.DO manager node and the image-processing functions are executed on the external raspberry connected to e.DO with the ethernet cable. Despite this, the error converges to almost the same value reached in simulation, which represents the acceptable limit threshold.

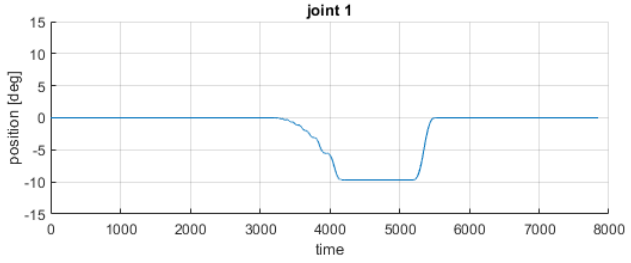

Figure 6.7: e.DO Joint 1

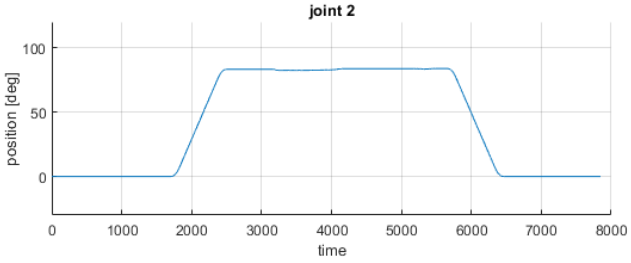

Figure 6.8: e.DO Joint2

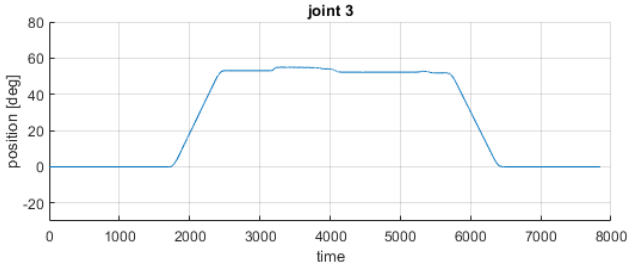

Figure 6.9: e.DO Joint 3

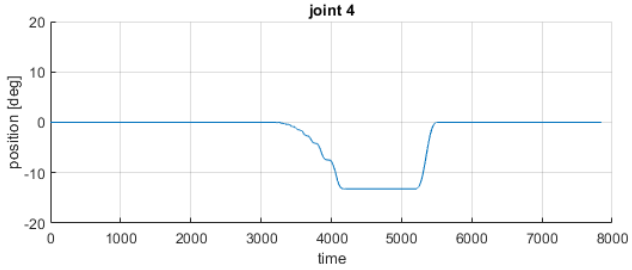

Figure 6.10: e.DO Joint 4

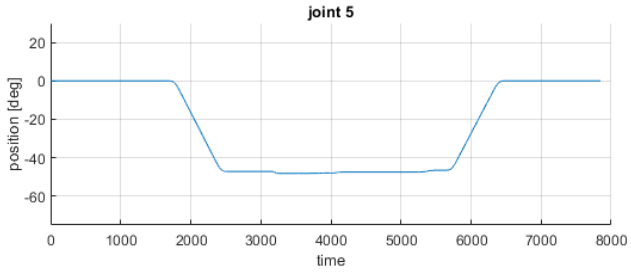

Figure 6.11: e.DO Joint 5

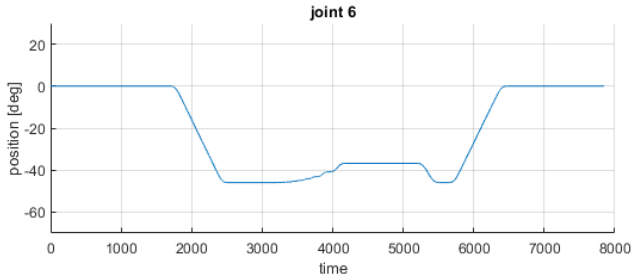

Figure 6.12: e.DO Joint 6

From the plots shown in Figures 6.7 - 6.12, it is possible to notice that the joint position signals are characterized by a constant final part equal to the initial one. This is due to the fact that the data acquisition process has been extended until the return of e.DO in the Init initial configuration.

Figure 6.13 shows the gripper positioned in the e.DO target point, while Figure 6.14 shows the ball released in the target box when the task is completed.

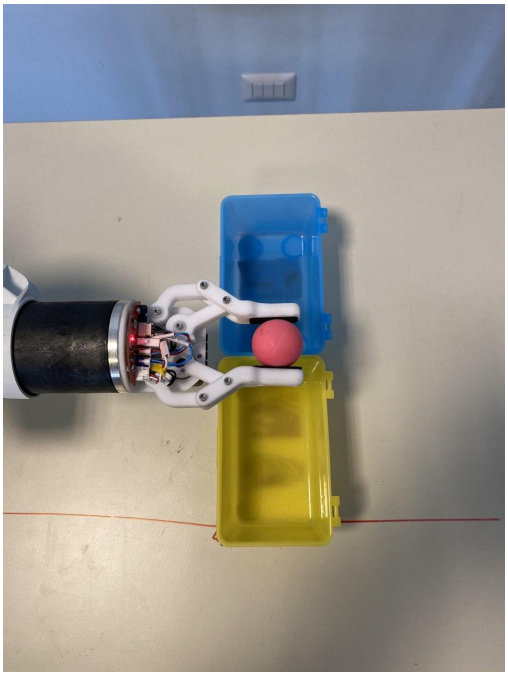

Figure 6.13: e.DO gripper positioned at the e.DO target point

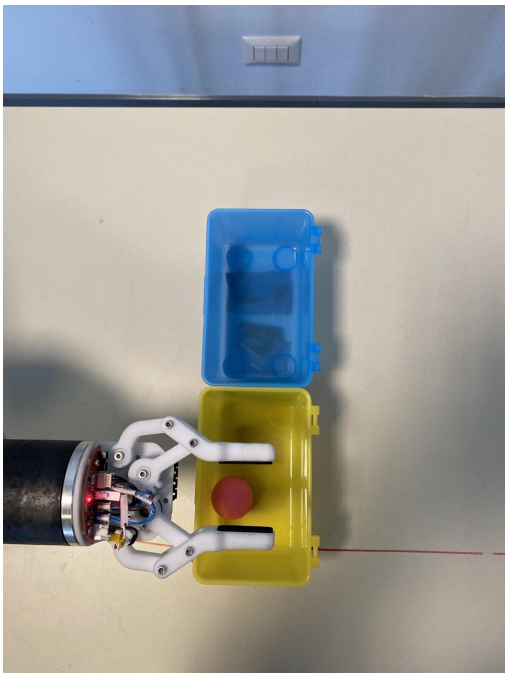

Figure 6.14: Task completed - Ball released

Figure 6.15, finally shows the usage of the raspberry resources during the execution of the ROS package. In particular, it is possible to see that the package (the Edo manager node) uses about 50% of the CPU and about 60% of memory. This can be a good result considering that the image-processing operations are very computationally expensive.

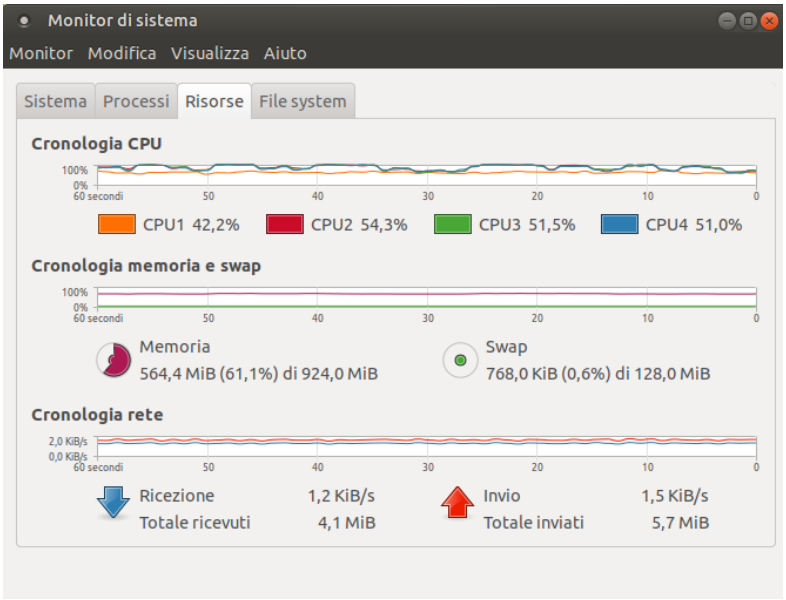

Figure 6.15: Raspberry resources usage

Chapter 7

Conclusions

In this thesis work, it has been created a system that is able to manage the communication between two robots and coordinate their movement to perform a specific task. In order to do that, the ROS Edo-AGV_integration package was developed, capable of interacting with the robots and managing their movements. The developed package is executed by an external raspberry pi, which represents the heart of the system and capable of connecting to both the robots. The thesis work was developed following different phases. In the first phase, the different strategies for integrating anthropomorphic robots with an autonomous guided vehicle and other devices have been analyzed. The second phase was focused on the study of possible ways to communicate with the robot and on the implementation of a program that manages the communication between the external raspberry pi and e.DO. Then, the possible strategies to adopt for the vision system have been analyzed, considering different configurations for the camera. The third phase involved the development of the ROS package Edo-AGV_integration aimed at managing the synchronization between the two robots and executing the release task, according to the given specifications. The subsequent phase focused on the simulation of the system in the Gazebo environment. First the different ways to simulate the two robots have been studied and then all the necessary changes inside the software architecture of the package have been done, in order to test all the functionalities of the package into the simulated environment. The final part involved the real implementation of the developed package on the external raspberry pi. In particular, the part of the architecture related to the movement of e.DO has been tested, as well as the release of the piece inside the box using the USB camera connected to the raspberry.

The developed package allows to interact with the ROS package edo_core provided by Comau, which runs on the e.DO raspberry, and with the Autonomous Guided Vehicle. Regarding the vision system, with the camera-to-hand configuration the strategies implemented have obtained good results both in the Gazebo

simulation and in the real implementation, and approximately converge to the same error in the positioning of the gripper. One of the main limitations of the system is represented by the fact that the release zone and the pose of the AGV inside the release zone are known to the system, and are set as parameters in the configuration phase. Some possible improvements and future developments of the system may concern:

- the technique used to recognize the AGV
- the extension of the vision system functionalities
- the planning of the AGV/mobile robot movement
- the creation of an interactive application for the system configuration

For the recognition of the AGV, the ROS package find_object_2d has been used that, thanks to a preliminary learning phase, is able to recognize objects within the frames captured by the camera. The use of the find_object_2d package can be replaced by implementing a detection technique similar to the one implemented for the recognition of the target box and for the gripper, exploiting the functions provided by the OpenCV library. Otherwise, if the structure of the AGV is not known or may vary, it is possible to use deep learning techniques in which, thanks to the training of neural networks, it is possible to recognize different types of AGVs within the images of the camera. In this case, it is necessary to provide an initial learning dataset containing the images of different types of AGV, and once the training is completed, the saved network can be used for the real time recognition. This type of approach can be very interesting and flexible if it is necessary to recognize different types of AGV that can interact with e.DO, and therefore adopt different strategies depending on the type of AGV recognized. Regarding the execution of the release task inside the target box, two configurations for the camera have been adopted and different image-based approaches able to identify the target box and the gripper/object have been implemented. A possible extension might be to adopt a position-based approach able to determine the pose of the target box, and then move e.DO according to the cartesian error, or to use a look and move technique able to calculate the coordinates of the box and move e.DO to that point. For example, it is possible to use the visp_tracker package [59] which allows to track the object by providing the coordinates in the space with respect to the camera, knowing a priori the 3D model and the initial pose. Finally, other possible future developments might be to extend the functionalities of the system in order to manage and plan the movement of the AGV, or to create an application for the users with a graphic interface, that allows to define all the parameters of the system. For example, it is possible to define the known target points with respect to e.DO and the AGV, the type of control/detection to be performed and

the characteristics of the AGV. In this way it possible to simplify the preliminary configuration phase before running the system.

Bibliography

[1] URL: https://mostre.museogalileo.it/nexus/inex.php?c[]=49103.

[2] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics, Modelling, Planning and Control*. Springer, 2009.

[3] URL: https://www.logisticaefficiente.it/adept/magazzino/automazione/differenza-fra-agv-e-robot-mobile.html.

[4] URL: https://semplicecome.it/innovazione-robotica-applicazioni-nella-vita-quotidiana/.

[5] URL: https://www.economyup.it/innovazione/cos-e-l-industria-40-e-perche-e-importante-saperla-affrontare/.

[6] URL: https://www.innexhub.it/industrie-4-0-che-cose/.

[7] Chaiyapol Kulpate, Raman Paranjape, and Mehran Mehrandezh. «Precise 3D Positioning of a Robotic Arm Using a Single Camera and a Flat Mirror». In: *International Journal of Optomechatronics,* 2:3 (2008), pp. 205–232.

[8] URL: https://www.toptal.com/robotics/introduction-to-robot-operating-system.

[9] URL: http://wiki.ros.org/ROS/Introduction.

[10] URL: https://roboticsbackend.com/what-is-ros/.

[11] URL: http://wiki.ros.org/ROS/Concepts.

[12] URL: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29.

[13] Comau. URL: https://edo.cloud/the-robot/.

[14] Stefano Pesce. «Simulation and advanced control of a professional manipulator». MA thesis. Politecnico di Torino, 2018.

[15] Comau. *e.DO User Manual*. 2019. URL: https://edo.cloud/wp-content/uploads/2019/06/eDO-UserManual_en.pdf.

[16] Comau. URL: https://edo.cloud/edo-robot/.

[17] Comau. *e.DO 6 Axes Technical Sheet*. 2017. URL: https://edo.cloud/wp-content/uploads/2017/10/edo-6-axes-technical-sheet.pdf.

[18] Comau. URL: https://edo.cloud/extensions/.

[19] Comau. *e.DO Service Manual*. 2019. URL: https://edo.cloud/wp-content/uploads/2019/06/eDO-ServiceManual_en.pdf.

[20] Pietro Castelli. «Integration of anthropomorphic robots with high precision instrumentation». MA thesis. Politecnico di Torino, 2019.

[21] URL: https://roboticsknowledgebase.com/wiki/networking/ros-distributed/.

[22] URL: https://roslibpy.readthedocs.io/en/latest/readme.html#main-features.

[23] URL: https://roslibpy.readthedocs.io/en/latest/examples.html.

[24] URL: https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf.

[25] Rafael Custodio Cejas and Italo Guedes A. Silva. «Image processing tasks applied to robot vision system and path discovery(March2016)». In: *ASEE2016 Northeast Section Conference*. University of Bridgeport, Bridgeport, Connecticut 06604, USA, 2016.

[26] URL: https://docs.opencv.org/master/d2/d96/tutorial_py_table_of_contents_imgproc.html.

[27] URL: https://docs.opencv.org/master/df/d9d/tutorial_py_colorspaces.html.

[28] URL: https://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html.

[29] URL: https://docs.opencv.org/master/dd/d49/tutorial_py_contour_features.html.

[30] URL: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html.

[31] URL: https://docs.opencv.org/3.4/d1/de5/classcv_1_1CascadeClassifier.html#details.

[32] URL: http://wiki.ros.org/smach.

[33] URL: http://wiki.ros.org/vision_opencv.

[34] Labbé, M. *Find-Object*. http://introlab.github.io/find-object. 2011.

[35] URL: http://wiki.ros.org/find_object_2d.

[36] URL: http://gazebosim.org/.

[37] URL: http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros.

[38] URL: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[39] URL: http://gazebosim.org/tutorials?tut=ros_roslaunch&cat=connect_ros.

[40] URL: https://github.com/Comau.

[41] URL: https://github.com/Pro/eDO_description/tree/master/urdf.

[42] URL: https://github.com/Comau/eDO_description.

[43] URL: https://github.com/Pro/edo_gazebo.

[44] Ioan A. Sucan and Sachin Chitta. *"MoveIt"*. URL: https://moveit.ros.org/.

[45] URL: https://github.com/Pro/eDO_moveit.

[46] URL: https://github.com/Pro/edo_gripper_moveit.

[47] URL: https://github.com/Pro.

[48] URL: http://docs.ros.org/melodic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html.

[49] URL: https://wiki.ros.org/ros_control.

[50] URL: https://moveit.ros.org/documentation/concepts/.

[51] URL: https://ompl.kavrakilab.org/.

[52] URL: http://gazebosim.org/tutorials?tut=ros_control&cat=connect_ros.

[53] URL: https://ros-planning.github.io/moveit_tutorials/doc/move_group_python_interface/move_group_python_interface_tutorial.html.

[54] URL: https://www.theconstructsim.com/ros-projects-exploring-ros-using-2-wheeled-robot-part-1/#part1.

[55] URL: http://gazebosim.org/tutorials?tut=ros_gzplugins&cat=connect_ros.

[56] URL: https://github.com/JenniferBuehler/gazebo-pkgs.

[57] URL: https://github.com/JenniferBuehler/gazebo-pkgs/wiki/The-Gazebo-grasp-fix-plugin.

[58] URL: https://www.logitech.com/it-it/product/hd-webcam-c270#specification-tabular.

[59] URL: http://wiki.ros.org/visp_tracker.