UNIVERSITÀ DEGLI STUDI DI PADOVA

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di laurea magistrale in Control System Engineering

"Computer vision - based robotics for the automation: simulation and control of a robotic arm in the automotive field"

Tutore universitario:
Angelo Cenedese

Laureando:
Luigi Laratta

Tutore aziendale:
Stefano Rivella

Anno Accademico 2022/2023
Data di laurea: 24 Ottobre 2023

Abstract

This thesis project is driven by the goal of acquiring the necessary competencies to develop a dedicated framework for the simulation and control of a 6-degree-of-freedom robotic arm, Lite 6 of UFactory. The framework leverages ROS (Robot Operating System) and computer vision algorithms to enhance the capabilities of the robotic arm.

The project, developed as part of a curricular internship, comprises multiple phases aimed at equipping the researcher with essential skills in computer vision and robotics. These competencies are crucial for the realization of a system that seamlessly integrates with the test framework developed by Reply Concept Quality. The key phases include the development of the ROS architecture for controlling the robotic arm within a simulation environment, the integration of the simulated architecture with the physical robotic platform, the integration of computer vision techniques in the ROS architecture to allow the movement of the robot according to the objects detected by the camera and the creation of an user-friendly remote control interface through APIs, utilizing Python and Flask, to command the robotic arm.

The project results are in line with the objectives set at the outset, contributing to the development of a versatile and accessible framework that underscores the synergy between computer vision and robotics in the realm of industrial automation.

Abstract in lingua italiana

Questo progetto di tesi ha l'obiettivo di acquisire le competenze necessarie per sviluppare un framework dedicato alla simulazione e al controllo di un braccio robotico a 6 gradi di libertà, Lite 6 di UFactory. Il framework sfrutta l'architettura ROS (Robot Operating System) e algoritmi di computer vision per migliorare le capacità del braccio robotico.

Il progetto, sviluppato nell'ambito di un tirocinio curriculare, comprende diverse fasi volte a dotare il ricercatore di competenze essenziali in computer vision e robotica. Queste competenze sono fondamentali per la realizzazione di un sistema che si integri perfettamente con il framework di test sviluppato dall'azienda Reply Concept Quality. Le fasi principali comprendono lo sviluppo dell'architettura ROS per il controllo del braccio robotico all'interno di un ambiente di simulazione, l'integrazione dell'architettura simulata con la piattaforma robotica fisica, l'integrazione di tecniche di computer vision nell'architettura ROS per consentire il movimento del robot in funzione degli oggetti rilevati dalla camera e la creazione di un'interfaccia di controllo remoto user-friendly attraverso API, utilizzando Python e Flask, per comandare il braccio robotico.

I risultati del progetto sono in linea con gli obiettivi fissati all'inizio, contribuendo allo sviluppo di un framework versatile e accessibile che sottolinea la sinergia tra computer vision e robotica nel campo dell'automazione industriale.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The integration of computer vision with robotics has revolutionized the field of automation, enabling the development of advanced systems capable of perceiving and interacting with their environment. This new hybrid approach offers loads of benefits when it comes to talk about flexibility, intelligence and adaptability in regards to automation systems. Moreover, it opens up several new perspectives for the development of a broad array of projects taking advantage of a robotic arm, such as pick-and-place tasks, assembly operations and quality control inspections in the automotive industry [1].

This thesis is developed on the basis of a curricular internship, which purpose is to acquire the computer vision and robotics skills that are necessary for the creation of a system that can be integrated into the test framework developed by the Reply Concept Quality company. Overall, this dissertation focuses on the creation of an intuitive and user-friendly framework that enables seamless interaction with the robot and facilitates the development of projects that leverage its capabilities.

In particular, we will deal with the development and integration of the ROS (Robot Operating System) architecture for the control of a robotic arm, UFactory Lite 6 [4], in both simulation and real-world environments. The primary objectives of this work include the creation of an intuitive remote control interface using APIs in order to make trajectory recording and playback and the incorporation of computer vision techniques within the ROS framework for object detection and localization. An external camera is employed to capture the visual scene from which the position of specific objects is extracted. These object positions are then used to guide the robotic arm, enabling it to execute precise movements.

The remainder of this dissertation is structured as follows:

- Chapter 2 presents a general overview of the key concepts of robotics and computer vision, explaining how these two disciplines complement each other aiming to improve the robot abilities to perceive and understand its surroundings;
- Chapter 3 describes materials and methods employed in the development of the project, including hardware, software tools and the algorithms used;

- Chapter 4 deals with the initial setup and the steps followed in order to start working with both robotic arm and external camera;
- Chapter 5 explains the general robot kinematics (forward and inverse kinematics), focusing then on the specific case of the Lite 6 robotic arm;
- Chapter 6 illustrates the design and implementation of an application programming interface using Flask, in order to remotely control the robotic arm. In particular, we implement a web page where APIs are called to record robot trajectories and to playback them.
- Chapter 7 describes the integration of computer vision techniques within the ROS architecture. It discusses the implementation of object detection algorithms, the utilization of camera data and the development of algorithms to enable the robot to autonomously move towards the detected objects.
- Chapter 8 concludes the thesis by summarizing the findings, highlighting the contributions and outlining potential future directions for research and development.

Through the realization of this thesis, we aim to make significant contributions to the automation of tasks in the automotive field, by providing an accessible and user-friendly framework that harnesses the power of Computer vision and robotics. The ultimate objective is to pave the way for enhanced efficiency, precision, and versatility in industrial automation.

This thesis is accompanied by the "Tirocinio Reply" folder, which contains all the materials used to accomplish the final objectives. Specifically, the implemented code, that will be presented in detail in the following chapters, is located in the "project" folder, whose path is the following: Tirocinio Reply/ros/catkin_ws/src/xarm_ros. This code serves as a practical implementation of the concepts and theories discussed in the document, enabling readers to observe the actual execution and outcomes of the proposed methodologies.

Chapter 2

State of the art

Robotics is an interdisciplinary field that has witnessed remarkable advancements in recent years. It involves the design, construction, operation and application of robots to automate various tasks. The growing interest in robotics is driven by the need to improve efficiency, safety and productivity in industries and everyday life. Over the years, robotics has evolved from simple, pre-programmed machines to sophisticated systems capable of autonomous decision-making and learning [5].

One of the key breakthroughs that revolutionized the field of robotics is the introduction of computer vision. Computer vision is a subfield of artificial intelligence and computer science that focuses on enabling machines to interpret and understand visual information from the surrounding environment. By providing robots with the ability to "see" and interpret the world through cameras and sensors, computer vision has opened new avenues for automation and interaction with the physical world. The combination of the two disciplines has significantly enhanced automation capabilities. By incorporating computer vision into robotic systems, robots can perceive, analyze and respond to their surroundings in real-time. This integration has led to various applications in automation, revolutionizing industries such as manufacturing, logistics, healthcare, agriculture and more.

Computer vision enables robots to identify and locate objects within their environment. This capability is crucial for tasks such as pick-and-place operations in manufacturing, sorting items in warehouses or assisting visually impaired individuals in daily activities. With computer vision, robots can navigate autonomously in dynamic environments. They can build maps of their surroundings, detect obstacles and plan optimal paths. This is particularly valuable in applications such as autonomous vehicles, exploration robots and service robots operating in complex environments. Moreover, computer vision facilitates natural human-robot interaction through gesture and facial expression recognition. This enables robots to understand and respond to human cues, making them more intuitive and user-friendly for tasks such as customer service, healthcare assistance and collaborative manufacturing. In manufacturing and production, computer vision-equipped robots can conduct precise quality control and inspection tasks. They can identify defects, measure tolerances and ensure products meet specific standards, thereby improving

production efficiency and reducing errors.

While the integration of robotics and computer vision has shown promising results, several challenges remain. These include real-time processing of large visual datasets, robustness to environmental variations and the ethical implications of automation in various industries. Future research in this area aims to address these challenges and further advance the capabilities of robotic systems in automation. In conclusion, the combination of robotics and computer vision has opened up exciting possibilities in automation, enabling robots to perceive and interact with the world in ways previously unattainable. The integration of these fields holds great potential for transforming industries and enhancing the quality of human-robot interactions.

Chapter 3

Materials and methods

In this chapter we provide a description of the materials and methods employed in this project. The key components of our experimental setup include software and hardware tools: the ROS architecture, Python IDE, a six degrees of freedom robotic arm (UFactory Lite 6), a linux PC for ROS and a Logitech Streamcam.
The camera calibration, image capture process, preprocessing techniques, trajectroy recording process, detection, localization and pose estimation methods are presented herein, but they will be described in detail in the following chapters where we will develop the work of this thesis.

Initially, we focus on software and hardware tools, describing ROS, the robotic arm and the streamcam, while in the second part we show methods and algorithms applied to solve specific problems and achieve the desired goal.

3.1 ROS: Robot Operating System

ROS (Robot Operating System) [7] is a free and open-source software framework for building and developing robot applications. It provides a collection of libraries, tools and conventions to simplify the development of complex robotic systems. ROS provides a flexible and modular architecture that allows developers to build complex robot systems by combining and integrating software components called nodes. Nodes can communicate with each other over a publish-subscribe messaging system called ROS topic system. It also provides a range of other features, including support for various programming languages (such as C++, Python and Java), tools for visualization and simulation, libraries for sensor and actuator control and a powerful ecosystem of third-party libraries and packages. Overall, it has become a popular choice for developing robot applications due to its flexibility, modularity and the strong community support it provides.
The ROS architecture has been designed and divided into three sections or levels of concepts:

- the Filesystem level
- the Computation Graph level

- the Community level

In the first level, a group of concepts are used to explain how ROS is internally formed, the folder structure and the minimum number of files that it needs to work. In the second level, communication between processes and systems happens. In this section, you can see the concepts and mechanisms that ROS has to set up to systems, handle all the processes and communicate with more than a single computer, and so on.
The third level is the Community level, which comprises a set of tools and concepts to share knowledge, algorithms and code between developers.

The main goal of the ROS Filesystem is to centralize the build process of a project while at the same time provide enough flexibility and tooling to decentralize its dependencies. Similar to an operating system, a ROS program is divided into folders and these folders have files that describe their functionalities:

- packages, which form the atomic level of ROS. A package has the minimum structure and content to create a program within ROS;

- package manifests, which provide information about a package, licenses, dependencies, compilation flags, and so on. A package manifest is managed with a file called package.xml;

- metapackages, used to aggregate several packages in a group;

- metapackage manifests, which are similar to a normal package, but with an export tag in XML. It also has certain restrictions in its structure;

- message (msg) types: a message is the information that a process sends to other processes. ROS has a lot of standard types of messages;

- service (srv) types: service descriptions define the request and response data structures for services provided by each process in ROS.

The workspace is a folder which contains packages, those packages contain our source files and the environment or workspace provides us with a way to compile those packages. It is useful when you want to compile various packages at the same time and it is a good way to centralize all of our developments.

In the workspace, each folder is a different space with a different role:

- the source space (src folder), where you put packages, projects, clone packages, and so on. One of the most important files in this space is CMakeLists.txt. The src folder has this file because it is invoked by *cmake* when you configure the packages in the workspace. This file is created with the *catkin_init_workspace* command;

- the build space (build folder), where cmake and catkin keep the cache information, configuration and other intermediate files for our packages and projects;

- development space (devel folder), used to keep the compiled programs. This is used to test the programs without the installation step. Once the programs are tested, you can install or export the package to share with other developers.

Regarding the Computation Graph level, ROS creates a network where all the processes are connected. Any node in the system can access this network, interact with other nodes, see the information that they are sending and transmit data to the network. The basic concepts in this level are nodes, the master, messages, services, actions and bags, all of which provide data to the graph in different ways and are explained in the following list:

- nodes, which are processes where computation is done. If you want to have a process that can interact with other nodes, you need to create a node with this process to connect it to the ROS network;
- the master, which provides the registration of names and the lookup service to the rest of the nodes. It also sets up connections between the nodes. If you don't have it in your system, you can't communicate with nodes, services, messages, and others;
- messages: nodes communicate with each other through messages. A message contains data that provides information to other nodes. ROS has many types of messages, and you can also develop your own type of message using standard message types. Each message must have a name to be routed by the ROS network. When a node is sending data, we say that the node is publishing a topic. Nodes can receive topics from other nodes by simply subscribing to the topic. It's important that topic names are unique to avoid problems and confusion between topics with the same name;

Figure 3.1: ROS message

- services: when you publish topics, you are sending data in a many-to-many fashion, but when you need a request or an answer from a node, you can't do it with topics. Services give us the possibility of interacting with nodes. Also, services must have a unique name;

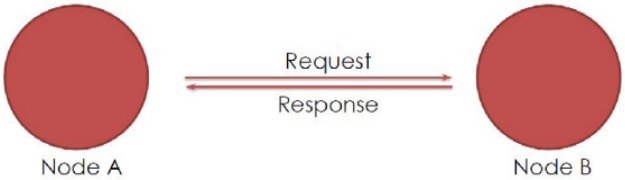


Figure 3.2: ROS service

- actions, which have a client-to-server communication relationship with a specified protocol. The actions use ROS topics to send goal messages from a client to the server. After receiving a goal, the server processes it and can give information back to the client. This information includes the status of the server, the state of the current goal, feedback on that goal during operation and finally a result message when the goal is complete;

Figure 3.3: ROS action

- bags, which are formats to save and play back the ROS message data. Bags are an important mechanism to store data, such as sensor data, that can be difficult to collect but is necessary to develop and test algorithms.

3.2 UFactory Lite 6 robotic arm

The UFactory Lite 6 [4]is a desktop robotic arm with six degrees of freedom. It can rotate around its base, lift objects up and down, extend its arm forward and backward and also move its wrist and gripper. It has a reach of 440 mm, a payload capacity of 660 g and a speed of 500 mm/s. The robotic arm is controlled by a software interface that allows users to program movements and actions or control the arm manually using a joystick or keyboard commands. It is compatible with a wide range of peripherals, including cameras, sensors and grippers, which can be attached to the arm using a modular system. This makes it a versatile tool for a variety of applications, such as pick-and-place tasks, 3D printing and machine vision. Overall, it is a powerful and flexible robotic arm that offers users a high level of control and precision for different industrial and research applications.

The robotic arm workspace refers to the area within the extension of the links. The figures and the table below show its working range and dimension

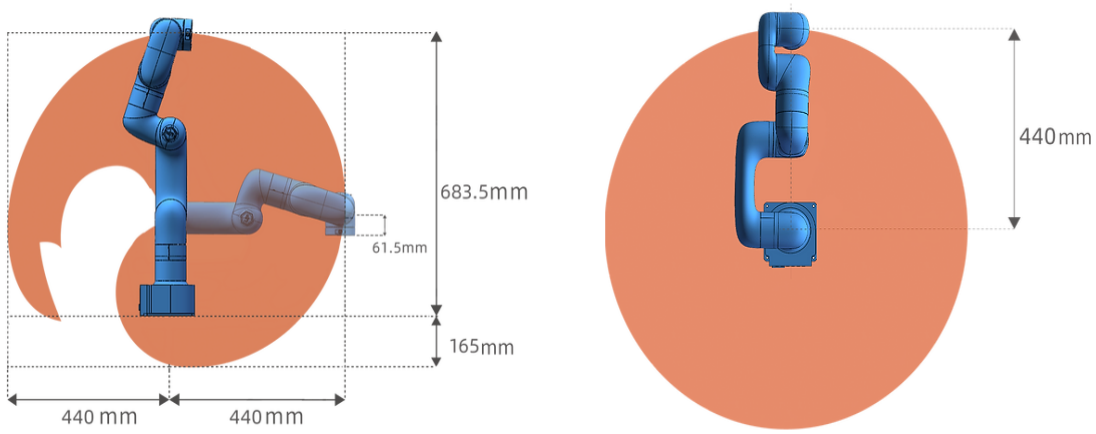


Figure 3.4: Robotic arm working range

Maximum Speed	180°/s
Joint 1	±360°
Joint 2	±150°
Joint 3	-3.5° ∼ 300°
Joint 4	±360°
Joint 5	±124°
Joint 6	±360°

Table 3.1: Range of various motion parameters of the robotic arm

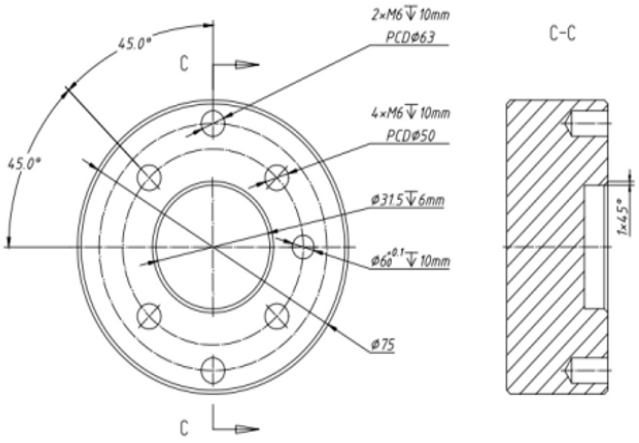


Figure 3.5: Robotic arm dimension

3.3 Logitech Streamcam

The Logitech Streamcam [6] is a portable webcam employed for capturing images and videos. The camera is mounted on a fixed frame and positioned to have a clear view of the workspace. It has a resolution of 1920x1080 pixels and a frame rate of 30 frames per second.

3.4 Trajectory recording and playback

One of the purposes of this project consists of recording and storing a desired path or motion of the robotic arm in order to replay it later. It allows the robot to reproduce a specific sequence of movements accurately, which is useful in applications where repetitive or precise tasks need to be performed.

To execute this process using the Lite 6 robotic arm, the following steps are needed:

- choose the appropriate recording mode for trajectory capture: manual recording mode where the arm is physically moved while recording or software recording mode where an external palnner like Moveit! permits to define the trajectory programmatically;
- begin the trajectory recording mode. This could involve pressing a specific button or initiating the recording process through the control software. The robotic arm will start capturing the movements that we perform or program during this recording phase;
- physically manipulate the robotic arm to perform the desired path or motion that we want to record. We can also program the arm to move through a series of waypoints or execute a pre-designed motion sequence;
- once the desired trajectory is completed, stop the recording process. This can be done by pressing a stop button or ending the recording mode through the control software;
- store the recorded trajectory. In the specific case of this robot, the recorded trajectory is saved in the Control box;
- to execute the recorded trajectory, we can trigger the correct mode and the arm will then follow the stored trajectory, replicating the same sequence of movements that were recorded earlier.

3.5 Camera calibration

In order to use the camera, first we need to calibrate it. Camera calibration involves a chessboard pattern to determine the intrinsic and extrinsic parameters of the camera. In the following, we can see how to do this process using chessboard images:

- consider a physical chessboard pattern (a square grid of black and white squares) with a known number of corners;
- capture multiple images of the chessboard pattern using the camera that requires calibration. It is recommended to capture images from various angles and positions, covering different parts of the image frame, as shown in the following pictures;

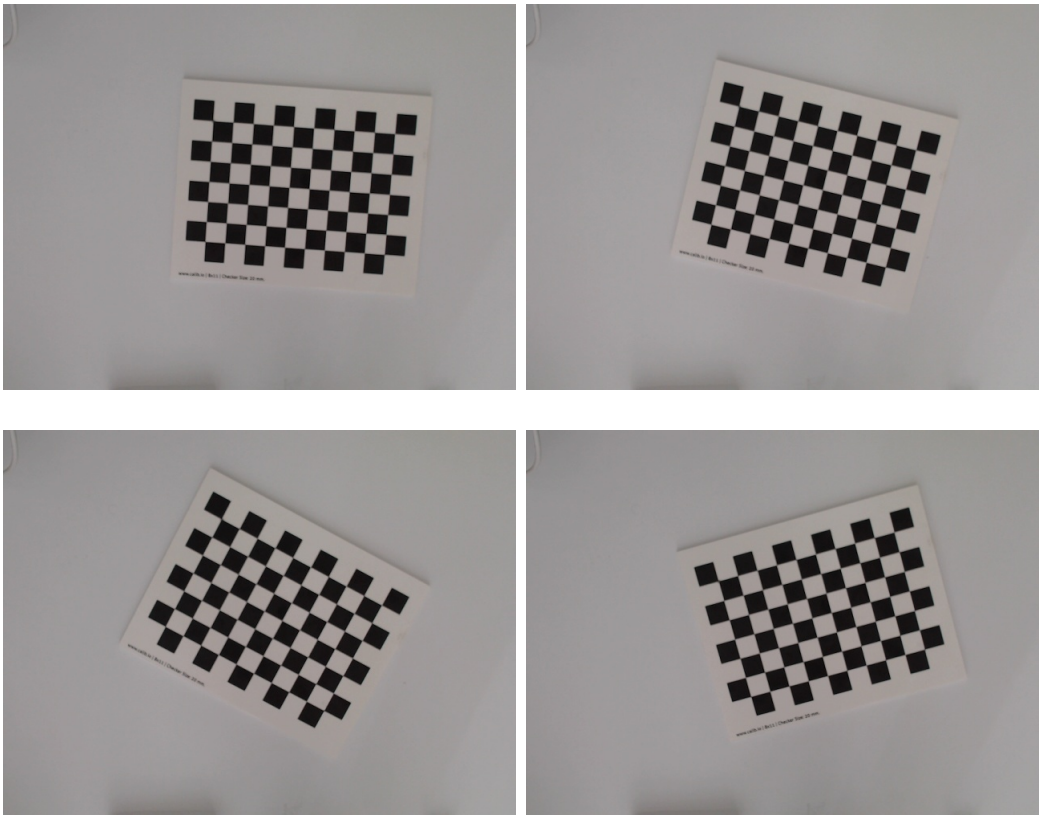


Figure 3.6: Chessboard captures

- in each captured image, detect the corners of the chessboard pattern using a corner detection algorithm, such as the Harris corner detector or the Shi-Tomasi corner detector. The algorithm should identify the coordinates of the internal corners of the chessboard squares;
- extract the corner coordinates from each image, storing them as corresponding 2D points in the image plane;
- assign world coordinates to the 3D points of the chessboard corners. For example, assuming a flat chessboard lying on a plane, the world coordinates can be assigned as the (x, y) positions of the corners on the chessboard plane.
- use the extracted 2D image points and corresponding 3D world points to estimate the camera's intrinsic and extrinsic parameters. This process typically involves solving a set of equations using algorithms like Direct Linear Transform (DLT) or Zhang's method, but in our case we use a built function of OpenCV;
- the intrinsic parameters of the camera include the focal length, principal point, and distortion coefficients. These parameters describe the camera's internal characteristics such as its lens properties and distortion effects;
- the extrinsic parameters represent the camera's position and orientation in the 3D world. They include the rotation matrix and translation vector, which

define the camera's pose relative to the chessboard.

3.6 Calibration validation

In order to validate the calibration process, we consider the camera matrix equation

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} R & | & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & | & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \tag{3.1}$$

$$= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

because it includes the previous estimated parameters. This equation allows to calculate the 2D pixel coordinates of a given 3D real point. In the following table, all the equation symbols are defined.

Symbols	Definition
s	Scaling factor
$\begin{bmatrix} u \\ v \end{bmatrix}$	Pixel coordinates
K	Camera matrix
f_x	Focal length along x coordinate
f_y	Focal length along y coordinate
c_x	Camera principal point along x coordinate
c_y	Camera principal point along y coordinate
R	Rotation matrix
t	Translation vector
$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$	Real-world coordinates

Table 3.2: Definition of the camera matrix equation symbols

The validation process is characterized by two steps:

- the first step is to validate the previous equation (from real-world coordinates to pixel coordinates): given a 3D real point of coordinates (0, 0, 0), we obtain one of the inner corner of the chessboard, as shown in the following figure.

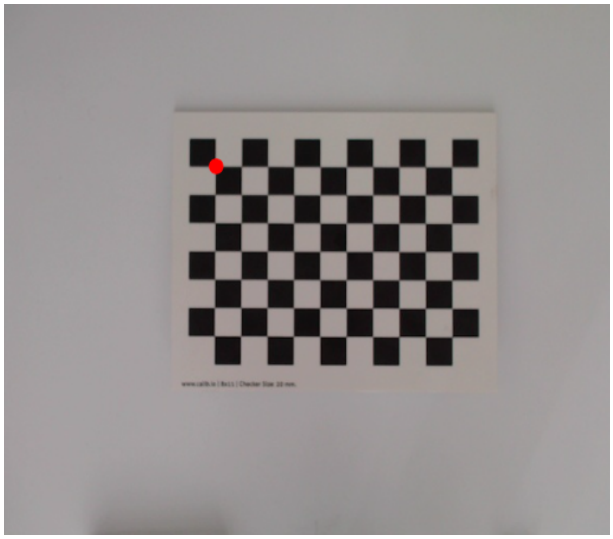


Figure 3.7: Result of the camera matrix equation validation

- the second one is to validate the inverse equation (from pixel coordinates to real-world coordinates): given two points on the captured image, such as the two highlighted in the following figure,

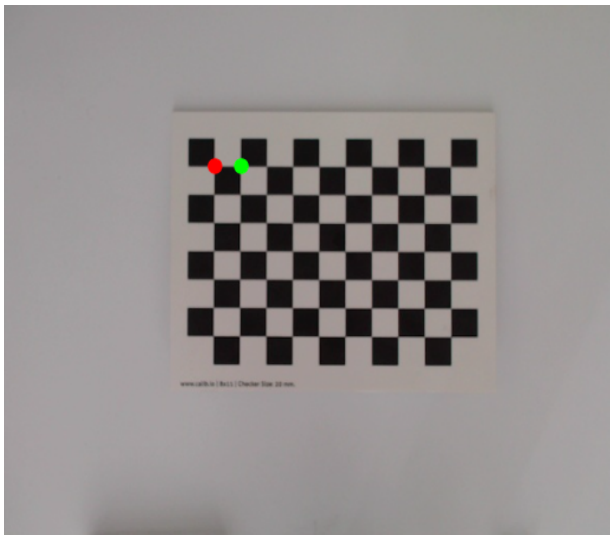


Figure 3.8: Points for the inverse camera matrix equation validation

the distance between the corresponding real points is equal to the square side of the chessboard.
In order to determine the inverse equation, we start from the equation 3.1 and reduce it,

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{3.2}$$

because the 3D real points are on the plane, therefore their z coordinate is

equal to 0. As a consequence, we can get the inverse camera matrix equation in this way:

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} su \\ sv \\ s1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix}^{-1} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} su \\ sv \\ s \end{bmatrix} \tag{3.3}$$

3.7 Marker detection

The marker detection is another important procedure to be done for the final goal. On the plane below the camera, some ArUco markers are placed and used as reference points or landamarks in the localization and coordination between camera, robot and objects.

Figure 3.9: ArUco markers

Once the camera captures an image of the plane, using OpenCV's ArUco module, the detection process is executed in this way:

- the captured image is preprocessed depending on image quality and lighting conditions;
- the resulting image is convert to grayscale;
- initialize the ArUco dictionary using the selected dictionary;

- detect markers by calling an ArUco function (*aruco.detectMarkers()*);

- obtain the marker corners and marker IDs from the detection results.

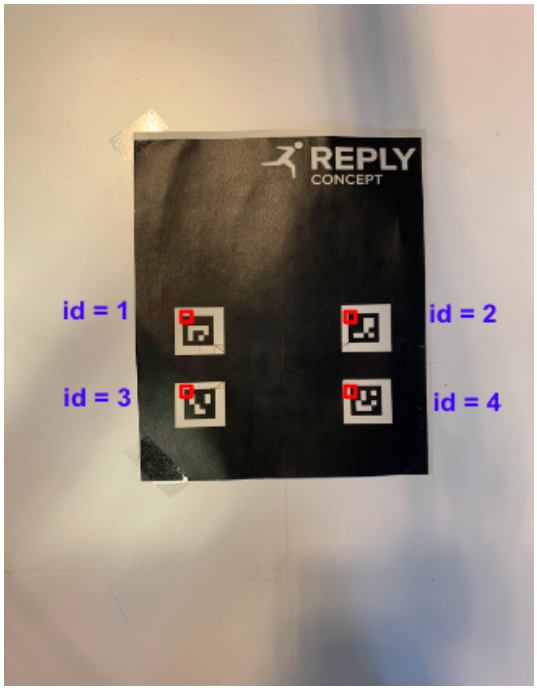


Figure 3.10: ArUco marker detection

3.8 Object detection and localization

Object detection localization refer to the processes of identifying and determining the precise location of some objects within an image. Given an image as the following

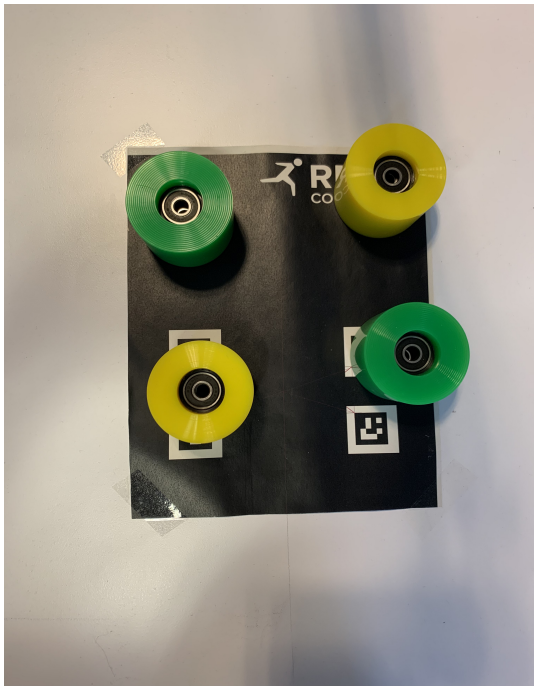


Figure 3.11: Input image for object detection and localization

we perform object detection based on color thresholds in the HSV color space. Secondly, we estimate objects position. We suppose to detect only green and yellow objects. The following steps explain how to do this process:

- we define lower and upper color ranges in the HSV color space. These color ranges are used to create binary masks for isolating the objects of interest;
- the input image is converted from the BGR color space to the HSV color space using the OpenCV function *cv2.cvtColor()*;
- based on the defined color ranges, we create binary masks by thresholding the image using *cv2.inRange()*. These masks separate the regions of the image that correspond to the desired colors;
- morphological operations are applied to the binary masks to remove noise and smooth the edges, obtaining cleaner object regions;
- using *cv2.findContours()*, the function detects contours in the binary masks obtained from the previous step. It separately finds contours for green and yellow objects;
- for each contour found, we calculate the center of mass using the moments of the contour. If the center of mass is not close to any previously detected center (within a defined threshold), it is considered as a new center, otherwise we compute their mean because they refer to the same object;
- finally, we draw circles at the detected centers of the green and yellow objects on the input image using *cv2.circle()*.

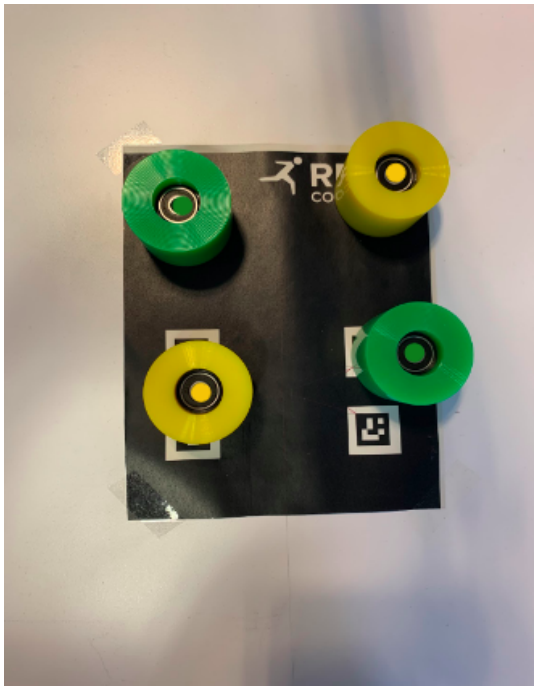


Figure 3.12: Object detetction and localization result on the input image

Chapter 4

Initial setup

In order to start working in the word of ROS and the use of the robot, it is essential to carry out a correct initial configuration. In this chapter, we deal with all the steps necessary to prepare the work environment and ensure correct integration between ROS, the robot and the resources needed for development. We start by installing ROS on the Linux PC. This framework allows us to fully exploit the potential of robotics and automation. We explore the system requirements, necessary packages and step-by-step procedures for a successful installation. Next, we focus on setting up the robot itself. We explain the necessary hardware components, connections and configurations to ensure proper communication and control of the robot via ROS. Then, we cover the topic of robot repository installation. This step allows us to access all the specific resources, libraries and packages for our robot, which will be essential for the development of our project. In the end, we describe the OpenCV and ArUco module installation in order to integrate computer vision with the robotic arm.
Preparing the working environment correctly is the first step to ensure the success of our robotics activities.

4.1 ROS installation

In this section, we provide a step-by-step guide to install ROS on Ubuntu. First of all, we have to setup our computer to accept software from packages.ros.org

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
   lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list
   '
```

and set up our keys

```
$ sudo apt install curl # if you haven't already installed curl
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/
   ros.asc | sudo apt-key add -
```

Now, we can update packages and install ROS

```
$ sudo apt update
$ sudo apt install ros-<distro>-desktop-full # replace <distro>
   with your desired ROS version (e.g. noetic)
```

For this project, the melodic distribution is used.
It is convenient if the ROS environment variables are automatically added to the bash session every time a new shell is launched:

```
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

If we have more than one ROS distribution installed, */.bashrc* must only source the *setup.bash* for the version we are currently using. If we just want to change the environment of our current shell, instead of the above, we can type:

```
$ source /opt/ros/melodic/setup.bash
```

Before using many ROS tools, we need to initialize *rosdep*. It enables to easily install system dependencies for source we want to compile and is required to run some core components in ROS. If we have not yet installed *rosdep*, do so as follows.

```
$ sudo apt install python-rosdep
```

With the following, we can initialize *rosdep*.

```
$ sudo rosdep init
$ rosdep update
```

Now, ROS is installed on the Ubuntu machine. We can verify the installation by running the command *roscore* in a terminal window. This should start the ROS core, which is the central component of the ROS system.

4.2 Robot installation

In order to install the Lite 6 robotic arm, we need to make the hardware and software connections. Starting from the first one, we define the workspace and fix the robot base. Then, we connect the DC input cable to the integrated control box of the robotic arm and the AC power cable to power supply. Adjust the voltage selector switch according to local voltage. At this point, we connect the Emergency

Stop Switch, LAN cable to the robotic arm and check the power and motion status indicators. The connection between PC and router is more stable if it is via Network cable. When we have connected all parts, it should look like as the following figure.

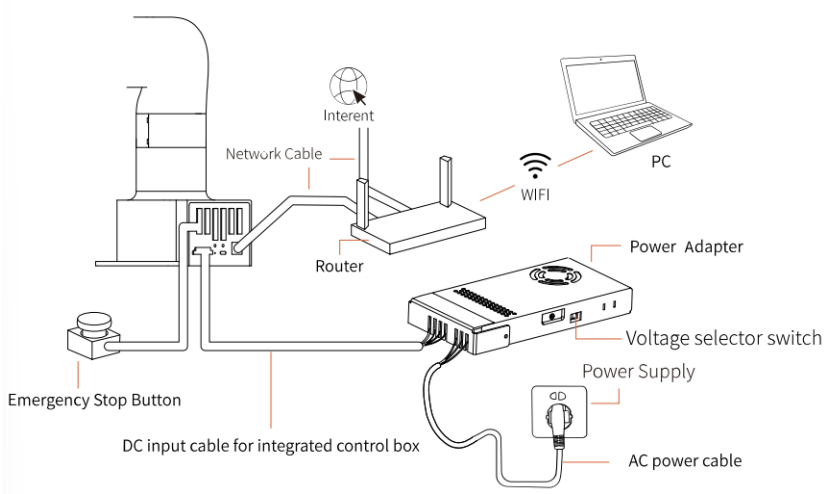


Figure 4.1: Hardware connection

Regarding the software connection, we configure the IP address of the PC: the PC and the integrated Control Box have to be in the same LAN environment. The network segment of integrated Control Box is 192.168.1.xxx, which means that the IPV4 network segment of the PC must be between 192.168.1.1 and 192.168.1.255 (the end number of IP address shall be different from that of the Control Box); the factory IP address of the Lite 6 has marked on the integrated Control Box.

Now, we have the following two ways to communicate with the robotic arm:
1. If we use a browser to access xArm Studio, we can communicate with the robot as follows:

- Open the browser
- Enter the IP address of the control box: 18333 in the search bar. For example, if the IP of control box is 192.168.1.185, enter 192.168.1.185:18333 in the search bar.

2. If we access xArm Studio software (developed and provided by UFactory for the operation and control of their robots, including the Lite 6 robotic arm), we can communicate with the robotic arm through the following steps:

- Download xArm Studio
  xArm Studio download address:
  https://store-ufactory-cc.myshopify.com/pages/download-xarm
- Install xArm Studio software
- Open the xArm Studio software, and enter the IP address of the control box in the search box.

All installation steps have been completed, therefore the robotic arm is ready to use.

4.3 UFactory Lite 6 repository installation

In order to work with the robotic arm using ROS, we need to install the relative repository. First of all, we create and initialize the ROS workspace,

```
$ mkdir -p $HOME/ros/catkin_ws/src
$ /bin/bash -c "source /opt/ros/melodic/setup.bash && cd $HOME/ros/
   catkin_ws/src && catkin_init_workspace && cd .. && catkin_make"
$ rosdep update
```

set up *.bashrc* file

```
$ echo "" >> $HOME/.bashrc
$ echo "source $HOME/ros/catkin_ws/devel/setup.bash" >> $HOME/.
   bashrc
$ echo "" >> $HOME/.bashrc
```

and check *catkin_make*

```
$ /bin/bash -ci "cd $HOME/ros/catkin_ws; catkin_make"
```

For installing the repository of the robot, its recommended to clone the github repo or every folder with roslaunch packages in HOME/ros/catkin_ws/src folder.

```
$ cd $HOME/ros/catkin\_ws/src
$ git clone https://github.com/xArm-Developer/xarm_ros.git --
   recursive
```

We update the package

```
$ cd $HOME/ros/catkin_ws/src/xarm_ros
$ git pull
$ git submodule sync
$ git submodule update --init --remote
```

and install ROS-packages for the Lite 6 robotic arm:

```
$ sudo apt-get update
$ sudo apt-get install -y mongodb-server-core \
    ros-melodic-warehouse-ros \
    ros-melodic-warehouse-ros-mongo \
```

```
    ros-melodic-trac-ik-kinematics-plugin \
    ros-melodic-joint-state-publisher-gui \
    ros-melodic-moveit-visual-tools
```

Update Ubuntu ROS-package

```
$ sudo apt update -qq
```

and install other dependent packages

```
$ cd $HOME/ros/catkin_ws
$ rosdep update
$ rosdep install --from-paths . --ignore-src --rosdistro melodic -y
```

Build the code

```
$ cd $HOME/ros/catkin_ws
$ catkin_make
```

and source the setup script

```
$ echo "source $HOME/ros/catkin_ws/devel/setup.bash" >> $HOME/.
   bashrc
```

Skip above operation if you already have that inside your HOME/.bashrc. Then do

```
$ source $HOME/.bashrc
```

Now, we can simulate the robotic arm in RViz

```
$ roslaunch xarm_description lite6_rviz_display.launch
```

and run the demo in Gazebo, an open-source simulator. It is designed to create three-dimensional virtual environments in which robot models, objects and realistic scenarios can be simulated. Gazebo offers a wide range of features and tools to support the development and experimentation of robots and complex robotic systems.

```
$ roslaunch xarm_gazebo lite6_beside_table.launch
```

For running this demo, the 'table' is needed. We can go inside the Gazebo simulator,

navigate through the model database for 'table' item, drag and place the 3D model inside the virtual environment and it will then be downloaded locally.

Additionally, we can execute the MoveIt motion planner, which is a powerful tool designed to generate and optimize motion plans for the robot. MoveIt is instrumental in tasks such as trajectory planning, collision avoidance and path execution, making it an essential component in robotic applications for tasks like grasping, manipulation and navigation.

- To run Moveit motion planner along with Gazebo simulator, supposing that no arm gripper is needed, first run:

```
$ roslaunch xarm_gazebo lite6_beside_table.launch
```

  then in another terminal:

```
$ roslaunch lite6_moveit_config lite6_moveit_gazebo.launch
```

  If we have a satisfied motion planned in Moveit, hit the "Execute" button and the virtual arm in Gazebo will execute the trajectory.

- To run Moveit! motion planner to control the real robotic arm: first make sure the arm and the controller box are powered on, then execute:

```
$ roslaunch lite6_moveit_config realMove_exec.launch robot_ip
   :=<your controller box LAN IP address>
```

If any error occured during the launch, you can play with the robot in Rviz and execute the sucessfully planned trajectory on real arm. Moreover, we can see all the ROS topics by using the following command:

```
$ rostopic list
```

and the ROS graph

```
$ rqt_graph
```

which plot the relationship between each node and the topics that they communicate with.

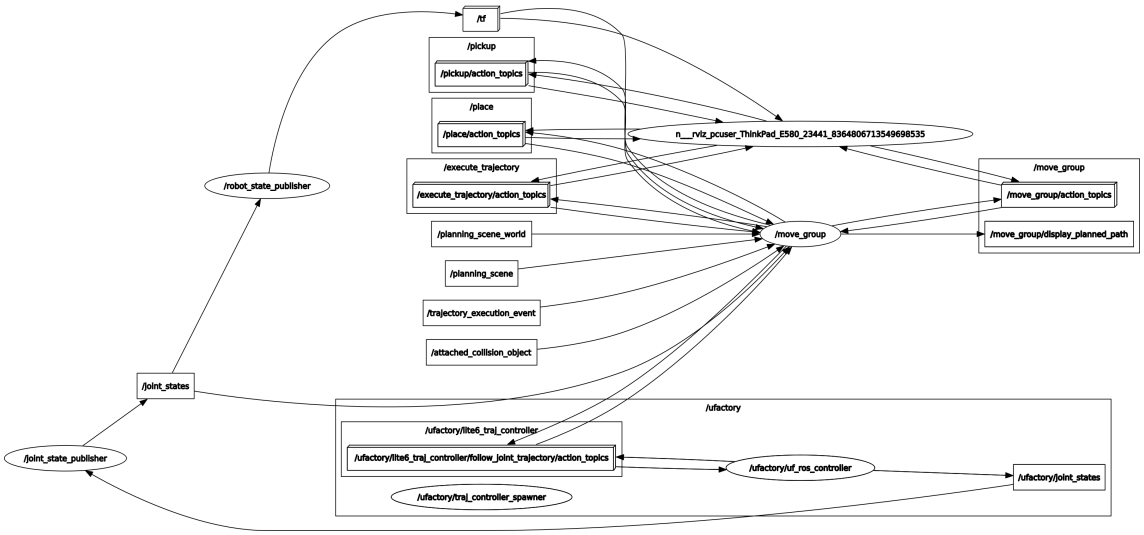


Figure 4.2: Rosgraph

4.4 OpenCV installation

OpenCV (Open Source Computer Vision Library) is an open-source computer vision library [3] used for a very wide range of applications, including medical image analysis, stitching street view images, surveillance video, detecting and recognizing faces, tracking moving objects, extracting 3D models, and much more.

To install the latest OpenCV version, we perform the following steps:

- install the required dependencies:

```
$ sudo apt install build-essential cmake git pkg-config libgtk
   -3-dev \ libavcodec-dev libavformat-dev libswscale-dev
   libv4l-dev \ libxvidcore-dev libx264-dev libjpeg-dev libpng
   -dev libtiff-dev \ gfortran openexr libatlas-base-dev
   python3-dev python3-numpy \ libtbb2 libtbb-dev libdc1394
   -22-dev
```

- clone the OpenCV's and OpenCV contrib repositories:

```
$ mkdir ~/opencv_build && cd ~/opencv_build
$ git clone https://github.com/opencv/opencv.git
$ git clone https://github.com/opencv/opencv_contrib.git
```

- once the download is complete, create a temporary build directory, and switch to it:

```
$ cd ~/opencv_build/opencv
$ mkdir build && cd build
```

Set up the OpenCV build with CMake:

```
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \
    -D CMAKE_INSTALL_PREFIX=/usr/local \
    -D INSTALL_C_EXAMPLES=ON \
    -D INSTALL_PYTHON_EXAMPLES=ON \
    -D OPENCV_GENERATE_PKGCONFIG=ON \
    -D OPENCV_EXTRA_MODULES_PATH=~/opencv_build/opencv_contrib/
   modules \
    -D BUILD_EXAMPLES=ON ..
```

- start the compilation process:

```
$ make -j8
```

Modify the *-j* flag according to the processor. If you do not know the number of cores in your processor, you can find it by typing *nproc*.

- install OpenCV with:

```
$ sudo make install
```

- to verify whether OpenCV has been installed successfully, we can type the following command and we should see the OpenCV version:

```
$ pkg-config --modversion opencv4$
```

4.5 ArUco module installation

The ArUco module is a computer vision library used for marker-based augmented reality applications. It allows us to detect and track ArUco markers. They can be printed and placed in the real world and a camera can then track these markers to superimpose virtual objects or information onto the real world.

To install the ArUco module in OpenCV, we can follow these steps:

- make sure we have OpenCV installed, otherwise we can install it using *pip*:

```
pip install opencv-python
```

- install the ArUco module

```
pip install opencv-contrib-python
```

Chapter 5

Robot kinematics

Before performing any task with the robot, we want to investigate its kinematics. Robot kinematics deals with the study of motion and positioning of robot manipulators without considering the forces and torques involved. It focuses on understanding and describing the geometry, structure and movement of robotic systems. In particular, robot kinematics involves analyzing the relationship between the joint angles and the resulting position and orientation of the robot's end-effector (the tool or device attached to the robot's arm). It aims to determine how the robot's joints move and how the end-effector moves in response to those joint motions.
Robot kinematics can be divided into two main areas: forward kinematics and inverse kinematics.

5.1 Forward kinematics

Forward Kinematics involves calculating the position and orientation of the end-effector based on the given joint angles. This process is essential for determining where the robot's end-effector will be located in the workspace for a given set of joint values. Forward kinematics can be visualized as a mapping from joint space to Cartesian space.

5.1.1 Lite 6 forward kinematics

In this project, we don't use an end-effector, therefore the forward kinematics is used to calculate the pose of the last link based on joint angles. In order to achieve this aim, we create a ROS action client (see **move_arm_publisher.py** file) which subscribes to the */ufactory/lite6_traj_controller/follow_joint_trajectory* topic of *FollowJointTrajectoryAction* type.
Then, we define a goal to send to the action server as you can see in the following code.

```
# Creates the SimpleActionClient, passing the topic and its type
arm_client = actionlib.SimpleActionClient("/ufactory/
    lite6_traj_controller/follow_joint_trajectory",
    FollowJointTrajectoryAction)
```

```
# Waits until the action server has started up and started
# listening for goals.
arm_client.wait_for_server()
# Creates a goal to send to the action server.
arm_goal = FollowJointTrajectoryGoal()
arm_goal.trajectory.joint_names = ['joint1', 'joint2','joint3' ,'
    joint4', 'joint5', 'joint6']
# Create a trajectory point
point = JointTrajectoryPoint()
# Store the desired joint values
point.positions = joint_values
# Set the time it should in seconds take to move the arm to the
# desired joint angles
point.time_from_start = rospy.Duration(10)
# Add the desired joint values to the goal
arm_goal.trajectory.points.append(point)
# Define timeout values
exec_timeout = rospy.Duration(10)
prmpt_timeout = rospy.Duration(5)
# Send a goal to the ActionServer and wait for the server to
# finish performing the action
arm_client.send_goal_and_wait(arm_goal, exec_timeout, prmpt_timeout
    )
```

As result, the robot moves and reaches a certain position based on the specified joint values.
Now, we want to understand what is the pose of the reference frame attached to the robot's last link, after finishing the previous motion. This pose is defined with respect to the robot base frame. We create another ROS node (see **link6_coord_listener.py** file) which subscribes to the */rviz_moveit_motion_planning_display/robot_interaction_interactive_marker_topic/feedback* topic of *InteractiveMarkerFeedback* type.

```
def coordinateCallback(data):
print("Link 6 pose:\n")
rospy.loginfo("Position x: %f, y: %f, z: %f \n", data.pose.position
    .x, data.pose.position.y, data.pose.position.z)
rospy.loginfo("Orientation x: %f, y: %f, z: %f, w: %f \n", data.
    pose.orientation.x, data.pose.orientation.y,data.pose.
    orientation.z, data.pose.orientation.w)

def coordinate_listener():
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("/rviz_moveit_motion_planning_display/
    robot_interaction_interactive_marker_topic/feedback",
    InteractiveMarkerFeedback, coordinateCallback)
rospy.spin()
```

The *coordinateCallback* function is executed whenever a new message is received on the previous topic. It takes the received data as input and prints the position and

orientation information of the link 6.

If during the execution of this file, the following warnings occur

[WARN] Controller failed with error GOAL_TOLERANCE_VIOLATED:
[WARN] Controller handle reports status ABORTED

we have to set the value of the "stopped_velocity_tolerance" parameter to 0 inside the xarm_ros/xarm_controller/config/lite6_traj_controller file.

5.2 Inverse kinematics

Inverse Kinematics deals with the calculation of joint angles required to achieve a desired position and orientation of the end-effector. Inverse kinematics is crucial for motion planning and control. It is more complex than forward kinematics, as it involves solving mathematical equations or numerical methods to find the joint angles.

5.2.1 Lite 6 inverse kinematics

In the case of the Lite 6 robotic arm, we don't need to solve mathematical equations to derive the inverse kinematics, because this is already implemented in the robot. As a consequence, we want to verify if, given any pose for the last link of the robot with respect to the base frame, the robot is able to move and reach it. To implement this, we create a ROS node (see **moveToPose2.py** file) where the following function is defined.

```
def move_to_goal_pose(x, y, z, roll, pitch, yaw):
    #print("Starting setup")
    # DisplayTrajectory publisher to publish trajectories for RVIZ
    #display_trajectory_publisher = rospy.Publisher('/move_group/
    # display_planned_path', moveit_msgs.msg.DisplayTrajectory)

    #print("Wiaiting for RVIZ...")
    #rospy.sleep(10)
    #print("Starting tutorial")

    # name of the reference frame for the robot
    #print("Reference frame for the robot: %s"
    # % move_group.get_planning_frame())
    # name of the end-effector link for the group
    #print("Refernce frame for the group: %s"
    # % move_group.get_end_effector_link())
    # list of all groups in the robot
    #print("Robot groups:")
    #print(robot.get_group_names())

    print("Plannning to a pose goal ...")
    target_pose = geometry_msgs.msg.Pose()
    target_pose.position.x = x
```

```
    target_pose.position.y = y
    target_pose.position.z = z

    qx = np.sin(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) - np.cos(
   roll/2) * np.sin(pitch/2) * np.sin(yaw/2)
    qy = np.cos(roll/2) * np.sin(pitch/2) * np.cos(yaw/2) + np.sin(
   roll/2) * np.cos(pitch/2) * np.sin(yaw/2)
    qz = np.cos(roll/2) * np.cos(pitch/2) * np.sin(yaw/2) - np.sin(
   roll/2) * np.sin(pitch/2) * np.cos(yaw/2)
    qw = np.cos(roll/2) * np.cos(pitch/2) * np.cos(yaw/2) + np.sin(
   roll/2) * np.sin(pitch/2) * np.sin(yaw/2)

    target_pose.orientation.x = qx
    target_pose.orientation.y = qy
    target_pose.orientation.z = qz
    target_pose.orientation.w = qw

    move_group.set_pose_target(target_pose,"link6")
    move_group.set_goal_tolerance(0.0005)
    plan = move_group.plan()
    #print("Waiting while RVIZ displays the plan ...")
    #rospy.sleep(5)

    #print("Visualizing the plan")
    #display_trajectory = moveit_msgs.msg.DisplayTrajectory()

    #display_trajectory.trajectory_start = robot.
    #get_current_state()
    #display_trajectory.trajectory.append(plan)
    #display_trajectory_publisher.publish(display_trajectory)

    #print("Waiting while plan is visualized (again)...")
    #rospy.sleep(5)

    #move_group.set_max_velocity_scaling_factor(1)
    move_group.go(wait=True)
    print("The robot has reached the pose goal!")
    move_group.stop()
    # It is always good to clear your targets after planning
    # with poses.
    move_group.clear_pose_targets()
    rospy.sleep(5)
```

This function initializes a *geometry_msgs.msg.Pose* object and sets its position and orientation based on the provided inputs. It sets the pose target for the robot's last link using *move_group.set_pose_target()* and a goal tolerance (0.0005) to determine how close the robot needs to be to the target pose to consider it successful. Then, it plans a trajectory to the target pose using *move_group.plan()* and stores the trajectory in the *plan* variable. It executes the planned trajectory using *move_group.go(wait=True)*, causing the robot arm to move to the desired pose. After reaching the goal pose, it stops the motion and clears the pose targets.
Once this file is executed, we can see that the robotic arm succeeds to reach the desired pose.

Chapter 6

Trajectory recording and playback

In this chapter we describe the recording and playback steps of the robot's trajectory using rosservice calls and, subsequently, functions defined in a ROS node, according to the procedure seen in Section 3.4. This involves utilizing the capabilities provided by the ROS ecosystem to record and store the robot's movements as well as replaying those recorded trajectories. In the second part, we develop an application programming interface in which APIs allow us to interact with the robot and control its movements remotely. By integrating API calls into a web application, we provide a user-friendly interface for users to command the robot, execute pre-defined trajectories and customize its behavior according to their specific needs. This allows for seamless control and automation of the robot's actions through the web page that we create.

6.1 Trajectory recording and playback via rosservice calls

We start this section introducing the different modes of the Lite 6 robotic arm:

- Mode 0: Position control mode
  The robotic arm can execute a series of motion commands (joint motion, linear motion, circular motion, etc.) automatically planned by the control box, which is also the mode that the control box enters by default after startup;

- Mode 1: Servoj mode
  The robotic arm can accept joint position commands sent at a fixed high frequency. The robotic arm responds immediately after receiving each command and executes at the maximum speed. If the user can complete the planning of the motion trajectory with smooth speed and acceleration and map it to the joint space, the servoj mode can replace the planning of the control box, and let robot execute the user's own or third-party (such as ROS Moveit!) planning algorithm;

- Mode 2: Joint teaching mode
  The robotic arm will enter the zero gravity mode, and the user can freely drag the links of the robotic arm to complete the teaching function. If the drag teaching is completed, switch back to mode 0.
- Mode 4: Joint velocity control mode
- Mode 5: Cartesian velocity control mode

The robotic arm is also characterized by motion states which are divided in:

- states that the control box can set (*set_state()*)
  - State 0: Start motion
    In this state, the robotic arm can normally respond to and execute motion commands. If the robotic arm recovers from an error, power outage, or stop state (state 4), remember to set the state to 0 before continuing to send motion commands. Otherwise the commands sent will be discarded;
  - State 3: Paused state
    Pause the currently executing motion and resume the motion at the interruption by setting state 0 again;
  - State 4: Stop state
    Terminates the current motion and clears the cached subsequent commands. Need to set state 0 to continue the motion.
- states that the control box can get (*get_state()*)
  - State 1: In motion
    The robotic arm is executing motion commands and is not stationary;
  - State 2: Standby
    The control box is already in motion ready state, but no motion commands are cached for execution;
  - State 3: Pausing
    The robotic arm is set to pause state, and the motion commands buffer may not be empty;
  - State 4: Stopping
    This is the state entered by default upon power-on. Stop and on commands can be executed until state is set to 0;
  - State 5: System reset
    The user just enters the state after the mode switch or changes some settings (such as TCP offset, sensitivity, etc.). The above operations will terminate the ongoing movement of the robotic arm and clear the cache commands, which is the same as the STOP state.

For example, if we want to enable the joint teaching mode, we have to use the following commands:

```
$ rosservice call /ufactory/set_mode 2
$ rosservice call /ufactory/set_state 0
```

while to enable the servoj mode:

```
$ rosservice call /ufactory/set_mode 1
$ rosservice call /ufactory/set_state 0
```

In order to record a robot trajectory (no longer than 5 minutes) directly by using rosservice calls, the following terminal commands are needed

```
$ rosservice call /ufactory/set_mode 1 (1 if you want to move the
   robot by using rviz or 2 if you want to move it manually)
$ rosservice call /ufactory/set_state 0 (DO NOT set STOP state(4)
   during recording or saving process)

$ rosservice call /ufactory/set_recording  1 (to start recording
   trajectory)

# MOVE THE ROBOT ACCORDING TO THE DESIRED MODE

$ rosservice call /ufactory/set_recording  0 (to finish recording
   trajectory)

$ rosservice call /ufactory/save_traj 'my_recording.traj' (give
   your desired name with the suffix '.traj') 15 (timeout)
```

recalling that all the active services can be list with the ROS command

```
$ rosservice list
```

After the recording step, we can run the saved trajectory in this way:

```
$ rosservice call /ufactory/set_mode 0
$ rosservice call /ufactory/set_state 0
$ rosservice call /ufactory/play_traj 'my_recording.traj' 1 (repeat
    times)  1 (speed-up factor: 1x,2x or 4x speed)
```

6.2 Trajectory recording and playback via ROS node

Based on the previous commands, we implement a ROS node (**gesture.py**) to record and playback trajectories for the robotic arm, as well as returning the arm to its

home position.
After asking the user for different arguments to perform various actions,

```
if __name__ == "__main__":
    if len(sys.argv) <= 1 or len(sys.argv) >= 5:
        print("Wtire one of the following cases (after the node
   name):")
        print("1. playRecord trajectoryName")
        print("2. action(recording or recordingAndPlay or
   recPlayHome) mode(1 or 2) trajectoryName")
        print("3. home")
    else:
        if len(sys.argv) == 3:
            name = str(sys.argv[2])
            playRecord(name)
        elif len(sys.argv) == 4:
            name = str(sys.argv[3])
            mode1 = int(sys.argv[2])
            action = str(sys.argv[1])
            if action == 'recording':
                startRecording(mode1)
                # MOVE THE ROBOT
                finishRecording(name, mode2=0)
            elif action == 'recordingAndPlay':
                startRecording(mode1)
                # MOVE THE ROBOT
                finishRecording(name, mode2=0)
                playRecord(name)
            elif action == 'recPlayHome':
                startRecording(mode1)
                # MOVE THE ROBOT
                finishRecording(name, mode2=0)
                playRecord(name)
                go_home()
            else:
                print("Error in the written sequence")
        elif len(sys.argv) == 2:
            go_home()
```

the code calls the corresponding functions based on the provided inputs:

- *startRecording(mode1)*:
  This function performs the necessary operations to start recording a trajectory using the robotic arm. First of all, we initialize the rospy node and makes it anonymous. Then, we create service proxy objects (*client_state*, *client_mode*, *client_rec*, *client_motion* and *client_clear_err*) to call specific ROS services related to robot control. We wait for the required services to become available using *rospy.wait_for_service()*, log a warning message indicating that the wait is over and call the service to clear any existing errors on the robot. Subsequently, we create a *SetAxisRequest* object to enable motion control and sends the request using *client_motion.call(srv_enable)*. After setting the mode (the provided "mode1" argument) and state of the robot, we call the

corresponding services. Finally, we create a *SetInt16Request* object to start recording a trajectory by setting its data field to 1 and call the service using *client_rec.call(srv_recording)*.

```
def startRecording(mode1):
    rospy.init_node('lite6_traj_plays', anonymous=True)
    client_state = rospy.ServiceProxy("ufactory/set_state",
   SetInt16)
    client_mode = rospy.ServiceProxy("ufactory/set_mode",
   SetInt16)
    client_rec = rospy.ServiceProxy("/ufactory/set_recording",
   SetInt16)
    client_motion = rospy.ServiceProxy("ufactory/motion_ctrl",
   SetAxis)
    client_clear_err = rospy.ServiceProxy("ufactory/clear_err",
    ClearErr)

    rospy.wait_for_service("ufactory/motion_ctrl")
    rospy.wait_for_service("ufactory/set_state")
    rospy.wait_for_service("ufactory/set_mode")
    rospy.wait_for_service("ufactory/save_traj")
    rospy.wait_for_service("ufactory/clear_err")
    rospy.logwarn("out of wait")

    # Clear errors
    srv_clearerr = ClearErrRequest()
    client_clear_err.call(srv_clearerr)

    # Motion enable
    srv_enable = SetAxisRequest()
    srv_enable.id = 8
    srv_enable.data = 1
    client_motion.call(srv_enable)
    rospy.sleep(1.0)

    #Setting the mode and the state
    srv_mode = SetInt16Request()
    srv_mode.data = mode1
    srv_state = SetInt16Request()
    srv_state.data = 0
    client_mode.call(srv_mode)
    client_state.call(srv_state)

    rospy.loginfo("Setting mode state...")
    rospy.sleep(1.0)

    srv_recording = SetInt16Request()
    srv_recording.data = 1  # to start recording
    client_rec.call(srv_recording)
```

- *finish_recording(name, mode2)*:
  This function executes several operations to finish the recording of a trajectory, save it and update a JSON file with the trajectory name. We start

initializing the rospy node, creating service proxy objects to call specific ROS services and waiting for the required services to become available, as before. Then, we create a *SetInt16Request* object to stop the recording of the trajectory by setting its data field to 0 and call the corresponding service using *client_rec.call(srv_recording)*. After that, a *SetStringRequest* object is created to specify the name of the trajectory file to be saved. We call the service to save the trajectory using *client_save.call(srv_saving)* and read the content of a JSON file. We append the name of the saved trajectory to the loaded data and call the service to clear any existing errors on the robot. In conclusion, we create *SetInt16Request* objects to set the mode and state of the robot using the provided "mode2" argument and call the corresponding services.

```
def finishRecording(name, mode2):
    rospy.init_node('lite6_traj_plays', anonymous=True)
    client_state = rospy.ServiceProxy("ufactory/set_state",
   SetInt16)
    client_mode = rospy.ServiceProxy("ufactory/set_mode",
   SetInt16)
    client_rec = rospy.ServiceProxy("/ufactory/set_recording",
   SetInt16)
    client_save = rospy.ServiceProxy("/ufactory/save_traj",
   SetString)
    client_clear_err = rospy.ServiceProxy("ufactory/clear_err",
    ClearErr)

    rospy.wait_for_service("ufactory/motion_ctrl")
    rospy.wait_for_service("ufactory/set_state")
    rospy.wait_for_service("ufactory/set_mode")
    rospy.wait_for_service("ufactory/save_traj")
    rospy.wait_for_service("ufactory/clear_err")

    srv_recording = SetInt16Request()
    srv_recording.data = 0 # to finish recording
    client_rec.call(srv_recording)

    rospy.loginfo("Saving trajectory")
    srv_saving = SetStringRequest()
    srv_saving.str_data = name + '.traj'
    srv_saving.timeout = 30.0
    client_save.call(srv_saving)

    # Save file name in json file called "trajectories.json"
    #filename = "./src/xarm_ros/project/scripts/
    # trajectories.json"
    filename = "trajectories.json"
    lst = srv_saving.str_data
    with open(filename, "r") as file:
        # load data from the file
        data = json.load(file)

    data.append(lst)

    with open(filename, "w") as file:
```

```
        json.dump(data,file)

    #Clear errors and change mode
    srv_clearerr = ClearErrRequest()
    client_clear_err.call(srv_clearerr)
    srv_mode = SetInt16Request()
    srv_mode.data = mode2
    srv_state = SetInt16Request()
    srv_state.data = 0
    client_mode.call(srv_mode)
    client_state.call(srv_state)

    rospy.sleep(3.0)
```

- *playRecord(name)*:
  This function performs a series of operations to run a recorded trajectory. After the initialization of the node, the creation of the service proxy objects and waiting for the required services to become available, we create a *SetAxisRequest* object to enable motion control and send the request using *client_motion.call(srv_enable)*. Then, we create *SetInt16Request* objects to set the mode and state of the robot to 0 (assuming it represents the desired mode and state) and call the corresponding services (*client_mode.call(srv_mode)* and *client_state.call(srv_state)*). After that, we create a *PlayTrajRequest* object to specify the trajectory file to be played, the repeat times and the speed factor. We call the service to play the trajectory using *play_client.call(play_srv)*, the service to clear any existing errors on the robot and the services to change the mode and state of the robot back to 0 using *client_mode.call(srv_mode)* and *client_state.call(srv_state)*.

```
def playRecord(name):
    rospy.loginfo("Run the recorded trajectory")
    rospy.init_node('lite6_traj_plays', anonymous=True)
    play_client = rospy.ServiceProxy("ufactory/play_traj",
   PlayTraj)
    client_state = rospy.ServiceProxy("ufactory/set_state",
   SetInt16)
    client_mode = rospy.ServiceProxy("ufactory/set_mode",
   SetInt16)
    client_motion = rospy.ServiceProxy("ufactory/motion_ctrl",
   SetAxis)
    client_clear_err = rospy.ServiceProxy("ufactory/clear_err",
    ClearErr)

    rospy.wait_for_service("ufactory/motion_ctrl")
    rospy.wait_for_service("ufactory/set_state")
    rospy.wait_for_service("ufactory/set_mode")
    rospy.wait_for_service("ufactory/play_traj")
    rospy.wait_for_service("ufactory/clear_err")
    rospy.logwarn("out of wait")

    # Clear errors
    srv_clearerr = ClearErrRequest()
```

```
    client_clear_err.call(srv_clearerr)

    # Motion enable
    srv_enable = SetAxisRequest()
    srv_enable.id = 8
    srv_enable.data = 1
    client_motion.call(srv_enable)
    rospy.sleep(1.0)

    #while not rospy.is_shutdown():
    srv_mode = SetInt16Request()
    srv_mode.data = 0
    client_mode.call(srv_mode)
    srv_state = SetInt16Request()
    srv_state.data = 0
    client_state.call(srv_state)

    rospy.loginfo("Setting mode state...")
    rospy.sleep(3.0)

    play_srv = PlayTrajRequest()
    play_srv.traj_file = name + '.traj'
    play_srv.repeat_times = 1
    play_srv.speed_factor = 1
    if not play_client.call(play_srv):
        rospy.logerr("Failed to call service play_traj")

    rospy.loginfo("Trajectory performed")

    # Clear errors and change mode
    srv_clearerr = ClearErrRequest()
    client_clear_err.call(srv_clearerr)
    srv_mode = SetInt16Request()
    srv_mode.data = 0
    client_mode.call(srv_mode)

    srv_state = SetInt16Request()
    srv_state.data = 0
    client_state.call(srv_state)
```

- *go_home()*:
  This function is only used to move robot arm to its home position. The first part of the code is equal to the previous ones. Subsequently, we set mode and state of the robot and we create a *MoveRequest* object to specify the parameters for moving the robot to the home position, including velocity (*mvvelo*), acceleration (*mvacc*) and time (*mvtime*). Then, we call the service to move the robot using *client_home.call(srv_home)*. In the end, we clear any existing errors and change the mode and state of the robot back to 0.

```
def go_home():
    rospy.loginfo("Go home")
    rospy.init_node('lite6_traj_plays', anonymous=True)
    client_home = rospy.ServiceProxy("ufactory/go_home", Move)
```

```
    client_state = rospy.ServiceProxy("ufactory/set_state",
SetInt16)
    client_mode = rospy.ServiceProxy("ufactory/set_mode",
SetInt16)
    client_clear_err = rospy.ServiceProxy("ufactory/clear_err",
    ClearErr)
    client_motion = rospy.ServiceProxy("ufactory/motion_ctrl",
SetAxis)

    rospy.wait_for_service("ufactory/motion_ctrl")
    rospy.wait_for_service("ufactory/go_home")
    rospy.wait_for_service("ufactory/set_state")
    rospy.wait_for_service("ufactory/set_mode")
    rospy.wait_for_service("ufactory/clear_err")
    rospy.logwarn("out of wait")

    # Clear errors
    srv_clearerr = ClearErrRequest()
    client_clear_err.call(srv_clearerr)

    # Motion enable
    srv_enable = SetAxisRequest()
    srv_enable.id = 8
    srv_enable.data = 1
    client_motion.call(srv_enable)
    rospy.sleep(1.0)

    srv_mode = SetInt16Request()
    srv_mode.data = 0
    client_mode.call(srv_mode)
    srv_state = SetInt16Request()
    srv_state.data = 0
    client_state.call(srv_state)

    rospy.loginfo("Setting mode state...")
    rospy.sleep(1.0)

    srv_home = MoveRequest()
    srv_home.mvvelo = 0.6
    srv_home.mvacc = 2.5
    srv_home.mvtime = 0
    client_home.call(srv_home)

    rospy.sleep(5.0)

    # Clear errors and change mode
    srv_clearerr = ClearErrRequest()
    client_clear_err.call(srv_clearerr)
    srv_mode = SetInt16Request()
    srv_mode.data = 0
    client_mode.call(srv_mode)

    srv_state = SetInt16Request()
    srv_state.data = 0
    client_state.call(srv_state)
```

6.3 Trajectory recording and playback via web page

We can proceed with the creation of an application interface to be able to call the previous functions directly from the web page. In order to do that, we need first to install Flask.

6.3.1 Flask installation

Flask is a web application framework [2] or simply a web framework which represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on. To install Flask, the latest version of Python is recommended. (Flask supports Python 3.7 and newer)
First of all we have to create an environment: a project folder and a venv folder within:

```
$ mkdir myproject
$ cd myproject
$ python3 -m venv venv
```

Then activate the corresponding environment:

```
$ . venv/bin/activate
```

and the shell prompt will change to show the name of the activated environment. Within it, use the following command to install Flask:

```
$ pip install Flask
```

Flask is now installed.

6.3.2 Application web page

After the previous installation, we can develop the web page that allows us to make API calls. They are requests made by one software application to another through a defined set of rules and protocols. An API call specifies the desired action or information that the calling application wants to obtain from the API provider. The latter processes the request and returns the requested data or performs the requested action, allowing the two applications to communicate and exchange information effectively.
Therefore, this web page will serve as an interface to interact with the APIs and control the desired functionalities of the system.

We create a a python file (**application.py**) in which we import the Flask class and create an instance of it which will be our WSGI application. Then, we use the *route()* decorator to tell Flask what URL should trigger our function.
We start creating the home page by using HTML language.

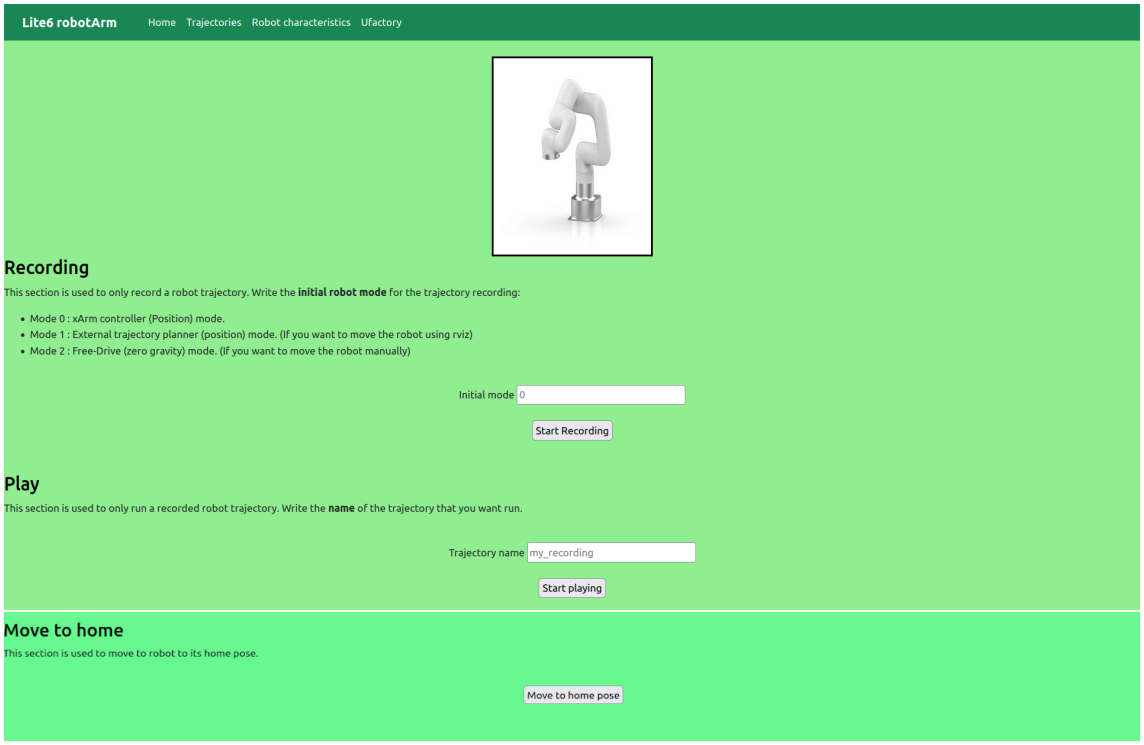


Figure 6.1: Home page

The home page is divided in three sections: Recording, Play and Move to home.

- The first section permits to record a robot trajectory: after setting the initial robot mode, we press on "Start recording" button and the following API is called.

```
# API to only record the robot trajectory
@app.route("/start_recording", methods=["GET", "POST"])
def start_recording():
    initial_mode = int(request.form.get("recStart_mode"))
    startRecording(mode1=initial_mode)
    return render_template("record_page.html")
```

This function recalls *startRecording(mode1)* defined in the **gesture.py** node and it returns a html page where we can specify the trajectory name and the final robot mode. After moving the robot according to the initial mode, we press on "Finish recording" button and the following API call is executed.

```
# API to finish recording the robot trajectory
```

```
@app.route("/finish_recording", methods=["GET", "POST"])
def finish_recording():
    traj = request.form.get("recTraj_name")
    final_mode = int(request.form.get("recFinal_mode"))
    finishRecording(name=traj,mode2=final_mode)
    return render_template("record2_page.html")
```

It calls the *finishRecording(name, mode2)* function and returns the template we want to display in the user's browser.

- In the second section, we can run a recorded robot trajectory by simply writing its name. Once the "Start playing" button is pressed, another API call is made

```
# API to run a recorded trajectory
@app.route("/play", methods=["GET", "POST"])
def traj_play():
    traj = request.form.get("playTraj_name")
    playRecord(name=traj)
    return render_template("play_page.html")
```

during which the *playRecord(name)* function executes the corresponding robot motion.

- The last section is used only to move the robot to its home pose through the following API call.

```
# API to move the robot to the home pose
@app.route("/move_to_home", methods=["GET", "POST"])
def move_to_home():
    go_home()
    return render_template("home_page.html")
```

In the application interface we also create the recorded trajectories page

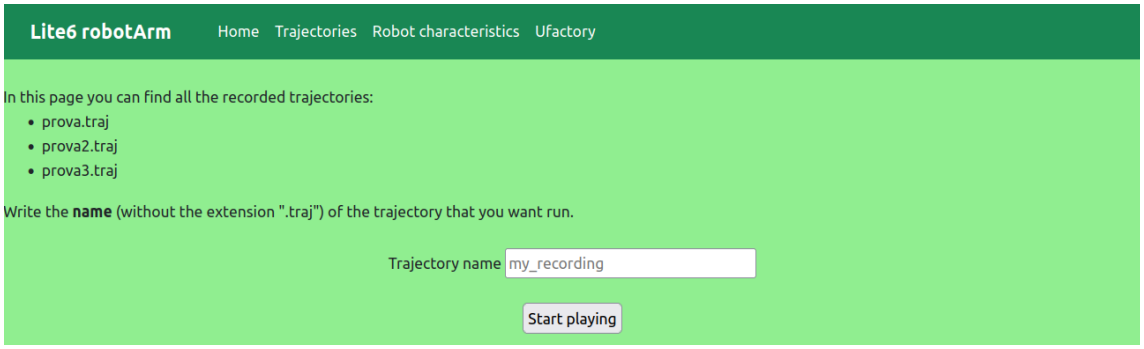
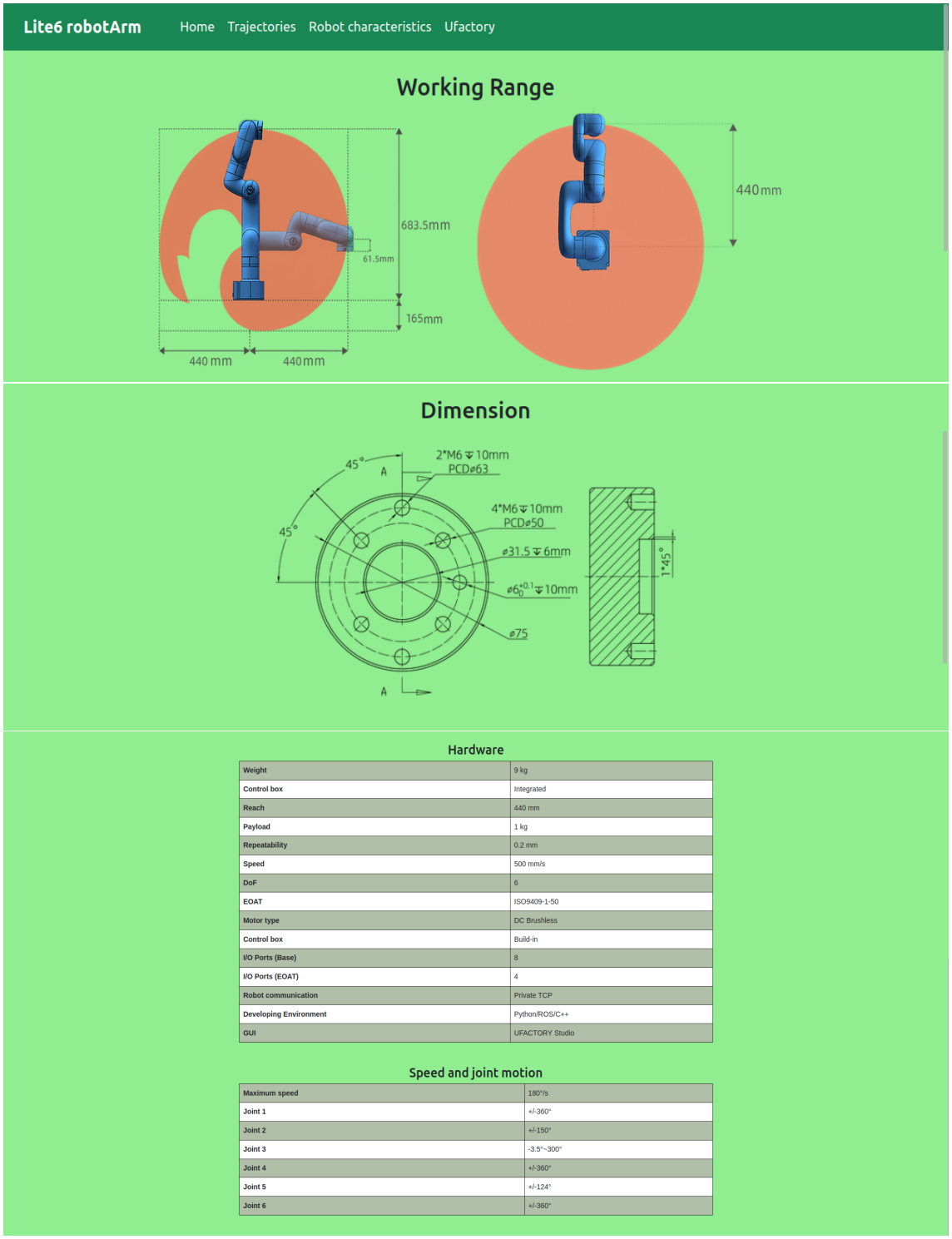


Figure 6.2: Recorded trajectories page

where we can find all the trajectories and we can directly run one of them. In the end, we introduce also the robot characteristics page



Hardware

Weight	9 kg
Control box	Integrated
Reach	440 mm
Payload	1 kg
Repeatability	0.2 mm
Speed	500 mm/s
DoF	6
EOAT	ISO9409-1-50
Motor type	DC Brushless
Control box	Build-in
I/O Ports (Base)	8
I/O Ports (EOAT)	4
Robot communication	Private TCP
Developing Environment	Python/ROS/C++
GUI	UFACTORY Studio

Speed and joint motion

Maximum speed	180°/s
Joint 1	+/-360°
Joint 2	+/-150°
Joint 3	-3.5°~300°
Joint 4	+/-360°
Joint 5	+/-124°
Joint 6	+/-360°

Figure 6.3: Robot characteristics page

In order to run this application, we need to write the following command lines

in the terminal:

```
$ flask run
$ export FLASK_APP=application.py
$ flask run
```

This launches a very simple builtin server. Therefore, if we head over to *localhost:5000* (where 5000 is the port number) we can see our home page. If we want to make a specific API call, we use *localhost:5000/* followed by the name of the function (Example: *localhost:5000/go_home*).

The server is only accessible from our own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary python code on your computer. If you have the debugger disabled or trust the users on your network, we can make the server publicly available simply by adding *–host=0.0.0.0* to the command line:

```
$ flask run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

Finally, by enabling debug mode, the server will automatically reload if code changes, and will show an interactive debugger in the browser if an error occurs during a request.

Chapter 7

Computer vision and robotics

This chapter illustrates how to integrate computer vision with robotics. Our focus lies in implementing the procedure leading to the object detection and localization, followed by developing intelligent algorithms to empower the robotic arm with the ability to autonomously move towards the detected objects. This goal is achieved through the implementation of a ROS network and the use of a class (Operations class) which provides a collection of useful methods for performing calculations related to pixel coordinates and coordinate transformations between different reference frames. The class methods will be explained in detail when called in the different nodes.

7.1 Image capture

As already seen in Section 3.5, one fundamental and crucial step to use the camera is the calibration process in order to achieve accurate and reliable results. This involves capturing a series of images (from various angles and positions) of a chessboard pattern. The code which implements this step is in the **chessboardCaptures.py** file.

```
cap = cv2.VideoCapture(0) # 0 is the camera index
count = 0 # initialize the counter
while True:
    ret, frame = cap.read() # read a frame from the camera
    cv2.imshow('frame', frame)

    # check for spacebar key press
    if cv2.waitKey(1) & 0xFF == ord(' '):
        count = count + 1
        # Save the frame in the "Captures" folder with a
        # unique filename
        filename = 'capture{}.jpg'.format(count)
        filepath = os.path.join('../Captures', filename)
        cv2.imwrite(filepath, frame)
        print("Capture {} saved!".format(count))

    # If the user presses the 'q' key, quit the program
    elif cv2.waitKey(1) & 0xFF == ord('\n'):
        break
```

```
cap.release() # release the camera
cv2.destroyAllWindows() # destroy all windows
```

This code sets up the video capture using *cv2.VideoCapture(0)*, which accesses the default camera (if you have multiple cameras, you can change the index accordingly) and a loop is initialized to continuously read frames from it. Inside this loop, the code waits for user input. If the spacebar key is pressed, the following actions are performed: the count of captured frames is incremented, the frame is saved as an image file in the "Captures" directory using *cv2.imwrite(filepath, frame)* and a message is printed to the console to indicate that the capture has been saved. If the user presses the Enter key, the loop breaks and the program terminates. After exiting the loop, the code releases the camera using *cap.release()* and closes all the OpenCV windows with *cv2.destroyAllWindows()*. This is essential to release the camera resources properly and close the display windows.

7.2 Camera calibration

After the previous step, we create the **chessboard** node, which is responsible to send the captured chessboard images to the **calibration** node in order to make the camera calibration. The communication between these two nodes takes place through a ROS service called *chessboard_captures*.

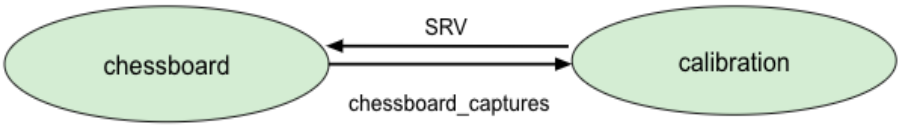


Figure 7.1: chessboard_captures service

In the **chessboard** node, we start initializing the ROS node and the service, as we can see in the following code.

```
# Initialize the ROS node and service
rospy.init_node('chessBoard')
img_srv = rospy.Service('chessboard_captures', ChessBoardCaptures,
    get_images)

# Initialize the OpenCV bridge
bridge = CvBridge()

# Spin the ROS node to keep it running
rospy.spin()
```

Then, the *get_images* function is assigned as the callback for the service. The latter

is designed to handle requests for chessboard captures. Furthermore, the OpenCV bridge is initialized to ensure smooth communication between OpenCV and ROS, as it provides functions for converting images between the two environments. After that, the script enters a loop using *rospy.spin()*, which is essential to keep the ROS node running continuously. This allows the node to listen for incoming service requests and respond accordingly.
The callback function is the core of the script, responsible for processing the image captures and constructing the response.

```
def get_images(request):
    # Get the path to the folder containing the images
    folder_path = request.folder_path

    # Get a list of image file paths in the folder
    image_paths = sorted(glob.glob(folder_path + '/*.jpg'))

    # Convert each image to a ROS image message and add it to
    # the response
    images = []
    for image_path in image_paths:
        cv_image = cv2.imread(image_path)
        ros_image = bridge.cv2_to_imgmsg(cv_image, encoding='bgr8')
        images.append(ros_image)

    # Return the list of ROS image messages as the response
    return ChessBoardCapturesResponse(images)
```

When invoked, this function expects a request object that contains the *folder_path*, representing the location of the images to be processed.
First, the code uses the glob module in Python to obtain a sorted list of image file paths within the specified folder. The *glob.glob()* function enables pattern matching, allowing the script to fetch all image files with a ".jpg" extension in the specified directory.
Next, the function iterates through the list of image paths and reads each image using the OpenCV library's *cv2.imread()* function. For each image, the code then employs the ROS CvBridge to convert the image from the OpenCV format to a ROS image message.
As the images are processed, they are appended to a list named *images*, which will store the chessboard captures as ROS image messages. In conclusion, the function constructs a *ChessBoardCapturesResponse* object, likely a custom ROS message type that can encapsulate the list of ROS image messages as the response to the service request.

On the other hand, the **calibration** node acts as client to request chessboard images and compute the calibration parameters through the *computeCalibrationParams()* function.

```
def computeCalibrationParams():
```

```
    get_images = rospy.ServiceProxy('chessboard_captures',
ChessBoardCaptures)

    # Call the service to get the images
    folder_path = './src/xarm_ros/project/Captures' # path as
    # request message
    response = get_images(folder_path)

    # CAMERA CALIBRATION PART
    # Defining the dimensions of checkerboard
    chessboard_size = (10,7) # (number of inner points in a row,
    # number of inner points in a column)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
    30, 0.001)

    # Creating vector to store vectors of 3D points for each
    # chessboard image
    objpoints = [] # 3D points in real world space
    # Creating vector to store vectors of 2D points for each
checkerboard image
    imgpoints = [] # 2D points in image plane

    # Defining the world coordinates for 3D points
    objp = np.zeros((np.prod(chessboard_size), 3), dtype=np.float32
)
    objp[:,:2] = np.mgrid[0:chessboard_size[0], 0:chessboard_size
[1]].T.reshape(-1, 2)*2 # 2cm is the dimension of the squares
    # in the chessboard

    gray_img = np.zeros((480,640,3), dtype=np.uint8)
    bridge = CvBridge()

    # Process the images in the response
    for image in response.images:
        cv_image = bridge.imgmsg_to_cv2(image, desired_encoding='
passthrough')
        #img = cv2.imread(cv_image)
        gray_img = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray_img,
chessboard_size, None)
        # If desired number of corner are detected, we refine the
        # pixel coordinates and display
        them on the images of chessboard
        if ret == True:
            objpoints.append(objp)
            # refining pixel coordinates for given 2d points.
            corners2 = cv2.cornerSubPix(gray_img, corners, (11,11)
,(-1,-1), criteria)
            imgpoints.append(corners2)

            # Draw and display the corners
            img = cv2.drawChessboardCorners(cv_image,
chessboard_size, corners2, ret)
```

```
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
   imgpoints, gray_img.shape[::-1], None, None)

    return mtx, dist, rvecs, tvecs, cv_image
```

This function establishes a connection to the service using *rospy.ServiceProxy()* to create a proxy for calling the service. After connecting to the service, it requests images of the chessboard from the specified folder path. The response is stored in the variable *response*, which contains a list of ROS image messages representing the captured chessboard images.
Next, it proceeds to define the dimensions of the chessboard and create the world coordinates corresponding to the 3D points on the chessboard (*objp*). These are essential inputs for the camera calibration process. Each image in the response is converted from a ROS image message to an OpenCV image using the *imgmsg_to_cv2()* method. The OpenCV library is then used to find the corners of the chessboard in the image using *cv2.findChessboardCorners()*. If the corners are successfully detected, the object points and refined image points are appended to separate lists (*objpoints* and *imgpoints*, respectively). Subsequently, the function proceeds to perform the camera calibration using *cv2.calibrateCamera()*, which returns the following parameters:

- *ret*: the value of the root mean square error (reprojection error) after calibration. A lower value indicates a better calibration;
- *mtx*: the 3x3 camera matrix (intrinsic parameters) which contains parameters such as focal length, optical center and other internal camera factors;
- *dist*: the vector of distortion coefficients. They describe the distortion of the camera lens;
- *rvecs*: a vector of rotation vectors. Each element of *rvecs* is a rotation vector representing the orientation of the camera in the coordinate system of the 3D world;
- *tvecs*: a vector of translation vectors. Each element of *tvecs* is a translation vector representing the position of the camera in the coordinate system of the 3D world.

The **calibration** node is also a server to handle another ROS service called *getCalibrationParams*, which is responsible for providing camera calibration parameters and the last chessboard image to the service client, **calibrationValidation_and_markerDetection** node, which performs camera calibration validation and marker detection.
To implement the *getCalibrationParams* service server, we define the following function

```
def get_params(mtx, dist, rvecs, tvecs, img):
    params_srv = rospy.Service("getCalibrationParams",
   CalibrationParams, params_callback)
    rospy.spin()
```

which takes the camera calibration parameters (*mtx*, *dist*, *rvecs*, *tvecs*) and the last chessboard image (*img*) as inputs.
The callback function *params_callback* is set as the handler, ensuring that this function is executed whenever a request is made to the service.

```
def params_callback(request):
    req = request.params
    bridge = CvBridge()

    if req == "params":
        array_mtx = mtx.flatten().astype(np.float32)
        array_dist = dist.flatten().astype(np.float32)
        array_rvec = rvecs[-1]
        array_tvec = tvecs[-1]
        ros_image = bridge.cv2_to_imgmsg(img, encoding='bgr8')
    return  CalibrationParamsResponse(array_mtx, array_dist,
   array_rvec, array_tvec, ros_image)
```

When the service is called, it expects a request that contains certain parameters. After extracting the necessary field from the request, the function checks it. If it is correct, it proceeds to prepare the response message to be sent back to the service caller.
The calibration parameters and the chessboard image are used to create appropriate response variables: *array_mtx*, *array_dist*, *array_rvec*, *array_tvec*, and *ros_image*. In particular, the calibration parameters are flattened and converted to the appropriate data types (*np.float32*) to facilitate serialization, the last element of the *rvecs* and *tvecs* lists is taken and assigned respectively to *array_rvec* and *array_tvec* (this is done because we are interested only in the last element of the two lists) and the *img* (OpenCV image) is converted to a ROS image message using the *cv2_to_imgmsg()* method, ensuring compatibility with the ROS communication framework. In the end, the function returns a *CalibrationParamsResponse* message to the service caller, containing the calibration parameters and the image.

7.3 Calibration validation

The calibration validation process is implemented in the **calibrationValidation_and_markerDetection** node. As a first step in achieving this goal, we need to retrieve the camera calibration parameters from the **calibration** node. This is done through the *getCalibrationParams()* function which initiates a service client (*params_client*) to request the needed parameters from the ROS service named *getCalibrationParams*. The following figure shows the communication between the two involved nodes.

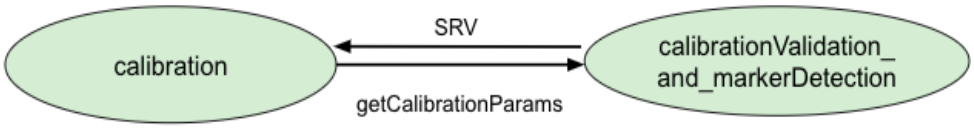


Figure 7.2: getCalibrationParams service

The response contains the camera matrix (*mtx*), distortion coefficients (*dist*), rotation vector (*rvec*), translation vector (*tvec*) and the chessboard image (*img*). The latter is converted from ROS image format to OpenCV format using the CvBridge library.

```
def getCalibrationParams():
        params_client = rospy.ServiceProxy('getCalibrationParams',
   CalibrationParams)
        message_request = "params"
        response = params_client(message_request)

        mtx = np.array(response.mtx).reshape((3,3))
        dist = np.array(response.dist).reshape((1,5))
        rvec = np.array(response.rvec).reshape((3,1))
        tvec = np.array(response.tvec).reshape((3,1))
        img = response.image
        # Initialize CvBridge object for converting between OpenCV
        # and ROS image formats
        bridge = CvBridge()
        img = bridge.imgmsg_to_cv2(img, desired_encoding='
   passthrough')

        return mtx, dist, rvec, tvec, img
```

After receiving the desired parameters, we set up an object (*obj*) of the Operations class and initialize the camera attributes with the retrieved calibration parameters through the *set_camera_attributes()* method.

```
class Operations:

    mtx = None # Camera matrix
    dist = None # Distortion parameter
    rvec = None # Rotation vector
    tvec = None # Translation vector
    chess_image = None
    scalingFactor = 0

    def __init__(self):
        pass

    def set_camera_attributes(self, mtx, dist, rvec, tvec):
        Operations.mtx = mtx
        Operations.dist = dist
        Operations.rvec = rvec
```

```
        Operations.tvec = tvec
```

The *obj* object is used to execute class functions for the validation process according to the two steps described in the Section 3.6.
Considering the camera matrix equation

$$\begin{bmatrix} su \\ sv \\ s \end{bmatrix} = s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} R & | & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & | & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \tag{7.1}$$

$$= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

- the first step is to validate the real-world-to-pixel coordinates transformation. In order to do that, we define the *pixelCoordsValidation(mtx, dist, r_wc, t_wc, chess_image*) function which calculates the pixel coordinates for three 3D points in the world space $(0, 0, 0)$, $(2, 0, 0)$, and $(0, 2, 0)$ using the camera calibration parameters. In particular, the second and third point are used to create the word reference frame. Then, the function draws points and lines on the chessboard image corresponding to these calculated pixel coordinates as we can see in the following figure

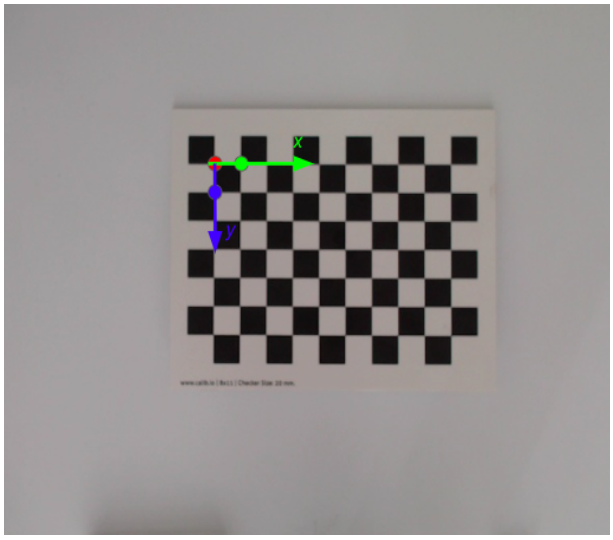


Figure 7.3: World reference frame

where the z axis enters the plane. As we can see from the figure, the obtained result is satisfactory. In fact, the 3D point $(0, 0, 0)$ corresponds to one of the

inner angles while the other two $(2, 0, 0)$ and $(0, 2, 0)$ are transformed in the green and blue points. This is an expected result since the square side on the chessboard is 2 (cm).

```
def pixelCoordsValidation(mtx, dist, r_wc, t_wc, chess_image):
    # According the formula s * [u, v, 1]' = K * [R|t] *
    # [X, Y, Z, 1]' , we can validate the camera calibration
    # considering a point in the 3D world space of coordinates
    # (0, 0, 0). If we obtain one of the inner corner of the
    # chessboard we can state that the camera is well
    # calibrated. Moreover, I choose other 2 points, (2,0,0)
    # and (0,2,0) to create the world reference frame.
    point3D_list = ([0,0,0], [2, 0, 0], [0, 2 , 0])
    pixel2D_list = []
    for i in point3D_list:
        x = i[0]
        y = i[1]
        z = i[2]
        pixel_coords, scalingFactor = obj.
   getPixelCoord_from_3Dcoord(x, y, z)
        print("Real 3D coordinates:", i)
        print("Pixel coordinates:", pixel_coords)
        pixel2D_list.append(pixel_coords)

    for j in pixel2D_list:
        # Draw a point at the (x=u, y=v) coordinates
        cv2.circle(chess_image, (j[0],j[1]), 5, (0, 0, 0), -1)

    # Define the color and thickness of the line
    color_green = (0, 255, 0)  # Green
    color_red = (0, 0, 255)  # Red
    thickness = 2

    # Draw the lines on the image
    cv2.line(cv_image, (pixel2D_list[0][0], pixel2D_list[0][1])
   , (pixel2D_list[1][0], pixel2D_list[1][1]), color_green,
   thickness)
    cv2.line(cv_image, (pixel2D_list[0][0], pixel2D_list[0][1])
   , (pixel2D_list[2][0], pixel2D_list[2][1]), color_red,
   thickness)

    print("In the image you can see the world reference frame:"
   )
    print("Green line: x axis")
    print("Red line: y axis")
    print("z axis enter on the screen")

    cv2.imshow("img", chess_image)
    cv2.waitKey(0)
    return (pixel2D_list[0][0], pixel2D_list[0][1]), (
   pixel2D_list[1][0], pixel2D_list[1][1]), scalingFactor
```

Within the previous function, the computation of pixel coordinates is carried

out through the *getPixelCoord_from_3Dcoord()* method of the Operations class.

```
def getPixelCoord_from_3Dcoord(self, x, y, z):
        point3D = np.array([x, y, z , 1]).reshape(4,1)
        # Computation of Rotation matrix
        self.R_wc,_ = cv2.Rodrigues(self.rvec)
        # Computation of the Roto-translation mtx
        rotoTranslmtx = Operations.compute_rotoTransl_mtx(self,
    self.R_wc, self.tvec)
        # Computation of the vector containing the pixel
        # coordinates s*[u, v, 1]
        pixel_2Dcoord = np.dot(np.dot(self.mtx, rotoTranslmtx),
    point3D)
        scalingFactor = pixel_2Dcoord[2]
        self.scalingFactor = scalingFactor
        pixel_2Dcoord = pixel_2Dcoord/ self.scalingFactor
        # Pixel coordinates
        u = pixel_2Dcoord[0]
        v = pixel_2Dcoord[1]
        pixel_coordinates = np.array([u,v])
        return pixel_coordinates, scalingFactor
```

It takes three input parameters: x, y and z, representing the 3D coordinates of a point in the real-world space. The function creates 4x1 NumPy array *point3D* with the provided coordinates and an additional 1 appended at the end. This last value is necessary for performing the homogeneous coordinate transformation. Then, we compute the rotation matrix R_{wc} from the rotation vector *rvec* using the *cv2.Rodrigues()* function. After combining properly the rotation matrix R_{wc} and the translation vector *tvec* to create the roto-translation matrix, we perform the transformation of the 3D point to pixel coordinates on the camera image plane. This is done by computing the matrix product of the camera matrix and the roto-translation matrix with the 3D point (Equation 7.1). Subsequently, we extract the scaling factor which is used to normalize the pixel coordinates and u and v from the normalized *pixel_2Dcoord* array. In the end, we store the scaling factor in the class attribute for potential future use and return the computed pixel coordinates as a NumPy array [u, v], along with the scaling factor.

- The second step consists in validating the accuracy of the inverse transformation formula, specifically from pixel coordinates to real-world coordinates.

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} su \\ sv \\ s1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$
$$\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{bmatrix}^{-1} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} su \\ sv \\ s \end{bmatrix} \tag{7.2}$$

This process is performed by considering two points: the origin and the end-

point of one axis of the world reference frame.
We define the *realCoordsValidation(point1, point2)* function

```
def realCoordsValidation(point1, point2):
    # In order to validate the inverse formula, we consider
    # 2 points:
    # - point 1: origin of the world ref frame
    # - point 2: endpoint of one axis of the world ref frame
    # We compute the corresponding real-world 3D coords
    # (with z = 0) and calculate the distance between them.
    # This distance has to be equal to 2 (cm), named the
    # dimension of the square side in the chessboard
    point_3Dcoord = obj.get3Dcoord_from_pixelCoord(point1[0],
   point1[1])
    point2_3Dcoord = obj.get3Dcoord_from_pixelCoord(point2[0],
   point2[1])
    dist = obj.get_distance(point_3Dcoord, point2_3Dcoord)
    dist = float(dist[0])
    print("The distance between the origin of the world ref
   frame and the axis endpoint is")
    print(dist)
```

in which the *get3Dcoord_from_pixelCoord()* method of the Operations class, that we will describe later, is called to obtain the 3D real-world coordinates for the two points passed as inputs. The function then calculates the Euclidean distance between the two obtained 3D coordinates using the *get_distance(point_3Dcoord, point2_3Dcoord)* method. This represents the physical distance between the origin and the endpoint of the reference axis. By comparing the calculated distance between two reference points in the real-world space to the known dimension of the chessboard's square side, the function provides a way to validate that the pixel-to-real-world coordinate conversion is functioning as expected and accurately represents the physical world measurements.

The class method *get3Dcoord_from_pixelCoord()* converts pixel coordinates from the camera's image plane into corresponding real-world 3D coordinates, according to the Equation 7.2.

```
def get3Dcoord_from_pixelCoord(self, u, v):
        # This function returns the pixel coordinates starting
        # from the 3D coordinates of a real point (real point
        # has z coord = 0)
        pixelPoint = np.array([u, v, 1]).reshape(3,1)
        pixelPoint = pixelPoint * self.scalingFactor[0]
        inv_mtx = np.linalg.inv(self.mtx)
        self.R_wc,_ = cv2.Rodrigues(self.rvec)
        modified_rotoTranslmtx = Operations.
   compute_modified_rotoTransl_mtx(self, self.R_wc, self.tvec)
        inv_modifRotoTransmtx = np.linalg.inv(
   modified_rotoTranslmtx)
```

```
        point3Dcoord = np.dot(np.dot(inv_modifRotoTransmtx,
   inv_mtx), pixelPoint)
        x = point3Dcoord[0]
        y = point3Dcoord[1]
        real_coordinates = np.array([x, y])
        return real_coordinates
```

It takes two parameters, u and v, which represent the pixel coordinates of a point in the camera's image plane. Then, a pixel coordinate vector *pixelPoint* is created by concatenating the input values along with a constant value of 1. The resulting vector is multiplied by the *scalingFactor* attribute stored in the Operations class to scale the pixel coordinates. This is applied to account for any scaling factor used during the pixel-to-3D coordinate conversion.
Next, we compute the inverse of the camera matrix *mtx* (intrinsic calibration matrix) using *np.linalg.inv(self.mtx)* and the rotation matrix R_{wc} from the rotation vector *rvec* using the OpenCV function *cv2.Rodrigues()*.
Then, we compute the modified roto-translation matrix through the following class method.

```
def compute_modified_rotoTransl_mtx(self, R, t):
        # select the first column of R
        firstColumn = R[:,0].reshape(3,1)
        # select the second column of R
        secondColumn = R[:,1].reshape(3,1)
        modifiedRotoTranslmtx = np.concatenate((firstColumn,
   secondColumn, t), axis=1)
        return modifiedRotoTranslmtx
```

The resulting matrix is simplified with respect to the one obtained in the previous step because we assume that the real-world points lie on a plane with a known depth (z coordinate) of 0.

After that, the inverse of the modified roto-translation matrix is computed and the real-world 3D coordinates are obtained by performing a series of matrix multiplications:

- The inverse modified roto-translation matrix is pre-multiplied with the inverse camera matrix;
- The result is then post-multiplied by the scaled pixel coordinate vector.

The resulting 3D point coordinates are extracted from the matrix, with the x and y values representing the real-world coordinates of the point in the plane (with z equal to 0).
Finally, the calculated real-world coordinates are returned by the function.

7.4 Marker detection

After validating the camera calibration, we can carry out the marker detection. This process involves using computer vision techniques to identify specific visual patterns or markers in images or video streams captured by a camera. These markers are designed to be easily recognizable and distinguishable from their surroundings because they serve as reference points that help the robot to determine its own position and orientation.
The **calibrationValidation_and_markerDetection** node implements this process. In order to do that, this node needs a frame captured by the camera, in which the ArUco markers are present (Figure 3.9). Therefore, we create a node called **camera** which acts as server to provide the desired captured image through the following service.

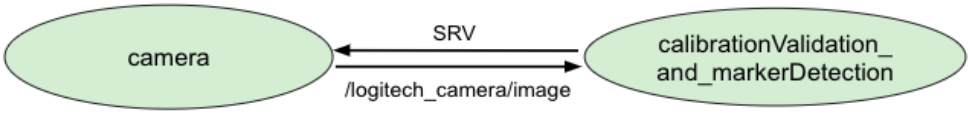


Figure 7.4: /logitech_camera/image service

We start describing the service server in which the *frame_publisher()* function initializes two service publishers using the */logitech_camera/image* and */logitech_camera/image2* service names. Both publishers use the *get_image* callback function to handle service requests. At the moment, we only consider the first service.

```
def frame_publisher():
    # Initialize publishers
    image_pub = rospy.Service('/logitech_camera/image', CameraFrame
   , get_image)
    image_pub2 = rospy.Service('/logitech_camera/image2',
   CameraFrame, get_image)
    # Spin the ROS node to keep it running
    rospy.spin()

if __name__ == '__main__':
    # Initialize ROS node
    rospy.init_node('camera', anonymous=True)
    frame_publisher()
```

The callback function takes a ROS service request as its argument and initializes a video capture object named *cap* using OpenCV's *VideoCapture* class. The argument 0 specifies that the default camera (usually the first camera available) should be used. Optionally, we can also set a specific resolution for the captured frames. Then, a *CvBridge* object is created to enable the conversion between OpenCV images and ROS image messages.

If the received request *req* is equal to the string "frame", the function captures a single frame from the camera using *cap.read()*. The returned values *ret* (a boolean indicating if the frame was captured successfully) and *frame* (the captured frame) are assigned. If the frame capture was successful (ret is True), the frame is converted into a ROS image message, *ros_image*, and assigned to the variable *image*.
In the end, the function returns a ROS service response using the *CameraFrameResponse* class, containing the image message.

```
def get_image(request):
    req = request.message

    # Initialize video capture from Logitech camera
    cap = cv2.VideoCapture(0)

    # Set camera resolution
    #cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    #cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    # Initialize CvBridge object for converting between OpenCV
    # and ROS image formats
    bridge = CvBridge()

    if req == "frame":
        ret, frame = cap.read()
        if ret:
            # Convert theimage to a ROS image message and add it
            # to the response
            #cv_image = cv2.imread(frame)
            ros_image = bridge.cv2_to_imgmsg(frame, encoding='bgr8'
   )
        image = ros_image
        # Return the list of ROS image messages as the response
        return CameraFrameResponse(image)
```

Now, we consider the service client where the marker detection process takes place. The focal function is *getMarkerCoordinates()* which fetches camera frames, detects ArUco markers, extracts inner corner coordinates and returns the detected marker corner coordinates along with the image.

```
def getMarkerCoordinates():
    get_image = rospy.ServiceProxy('/logitech_camera/image',
   CameraFrame)
    message = "frame"
    response = get_image(message)
    img = response.image

    # Initialize CvBridge object for converting between OpenCV and
    # ROS image formats
    bridge = CvBridge()
    marker_image = bridge.imgmsg_to_cv2(img, desired_encoding='
   passthrough')
```

```
    corner1 = 0
    corner2 = 0
    corner3 = 0
    corner4 = 0

    # Define the Aruco dictionary to use
    aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_250)

    # Define the Aruco parameters
    aruco_params = aruco.DetectorParameters_create()

    # Detect the Aruco markers in the frame
    corners, ids, rejectedImgPoints = aruco.detectMarkers(
   marker_image, aruco_dict, parameters=aruco_params)

    # Draw the marker on the frame
    aruco.drawDetectedMarkers(cv_image, corners, ids)

    # Select and draw a circle on the inner corner of each marker
    # Check if any markers were detected
    if ids is not None:
        # Get the index of the first detected marker
        index = np.squeeze(np.where(ids == 1))
        # Check if the first marker was detected
        if index.size > 0:
            # Get the corners of the first marker
            marker_corners = corners[index[0]]
            # Get the right-bottom corner of the first marker
            corner = marker_corners[0][2]
            corner1 = (int(corner[0]), int(corner[1]))
            # Draw a circle on the right-bottom corner of the
            #first marker
            cv2.circle(marker_image, (int(corner[0]), int(corner
   [1])), 3, (0, 0, 255), -1)

    if ids is not None:
        # Get the index of the second detected marker
        index = np.squeeze(np.where(ids == 2))
        # Check if the second marker was detected
        if index.size > 0:
            # Get the corners of the second marker
            marker_corners = corners[index[0]]

            # Get the left-bottom corner of the second marker
            corner = marker_corners[0][3]
            corner2 = (int(corner[0]), int(corner[1]))
            # Draw a circle on the left-bottom corner of
            # the second marker
            cv2.circle(marker_image, (int(corner[0]), int(corner
   [1])), 3, (0, 0, 255), -1)

    if ids is not None:
        # Get the index of the third detected marker
        index = np.squeeze(np.where(ids == 3))
        # Check if the third marker was detected
        if index.size > 0:
```

```
            # Get the corners of the third marker
            marker_corners = corners[index[0]]
            # Get the up-right corner of the third marker
            corner = marker_corners[0][1]
            corner3 = (int(corner[0]), int(corner[1]))
            # Draw a circle on the up-right corner of
            # the third marker
            cv2.circle(marker_image, (int(corner[0]), int(corner
    [1])), 3, (0, 0, 255), -1)

    if ids is not None:
        # Get the index of the fourth detected marker
        index = np.squeeze(np.where(ids == 4))
        # Check if the fourth marker was detected
        if index.size > 0:
            # Get the corners of the fourth marker
            marker_corners = corners[index[0]]

            # Get the up-left corner of the fourth marker
            corner = marker_corners[0][0]
            corner4 = (int(corner[0]), int(corner[1]))
            # Draw a circle on the up-letf corner of
            # the fourth marker
            cv2.circle(marker_image, (int(corner[0]), int(corner
    [1])), 3, (0, 0, 255), -1)

    cv2.imshow("img", marker_image)
    cv2.waitKey(0)

    return corner1, corner2, corner3, corner4, marker_image
```

This function starts by setting up a service proxy named *get_image* to call the service named */logitech_camera/image* using the message "frame" as a request to obtain a camera frame response. The obtained image data is then converted from the ROS image message format to an OpenCV image format using the *CvBridge* object.
The function proceeds to perform marker detection using the ArUco library. In particular, it initializes the ArUco dictionary with its parameters and then it detects the markers in the received image, extracting their corners and IDs (Figure 3.10). For each detected marker, the function:

- draws the marker outlines on the image using the *aruco.drawDetectedMarkers()* function;
- extracts the corners of the marker and specifically the coordinates of its inner corners;
- draws small circles at the detected inner corners using the *cv2.circle()* function.

In the end, the function returns the coordinates of the detected inner corners (*corner1, corner2, corner3, corner4*), along with the modified image containing the drawn markers and circles.

At this stage, we implement the following function in the same node.

```
def getCentralPointCoords(corner1, corner2, corner3, corner4,
    marker_image):
     midpoint_u = corner1[0] + (np.sqrt((corner1[0]-corner2[0])**2 +
     (corner1[1]-corner2[1])**2))/2
     midpoint_v = corner1[1] + (np.sqrt((corner1[0]-corner3[0])**2 +
     (corner1[1]-corner3[1])**2))/2
     midpoint = (round(midpoint_u), round(midpoint_v))

     cv2.circle(marker_image, (int(midpoint[0]), int(midpoint[1])),
    3, (0, 0, 255), -1)

     print("Coordinates of the central point:")
     print("Midpoint:", midpoint)

     # convert midpoint from tuple to array
     midpoint = np.array([midpoint[0], midpoint[1]])

     cv2.imshow("img", marker_image)
     cv2.waitKey(0)
     cv2.destroyAllWindows()

     # compute the translation vector t_wp
     t_wp = obj.get3Dcoord_from_pixelCoord(midpoint[0], midpoint[1])
     t_wp = np.append(t_wp, [[0]], axis=0)
     obj.set_t_wp(t_wp)
     return midpoint
```

It calculates the coordinates of the central point within the detected ArUco markers identified by their inner corner coordinates. The central point is determined by finding the midpoint between the four corner coordinates. The following figure shows the result.

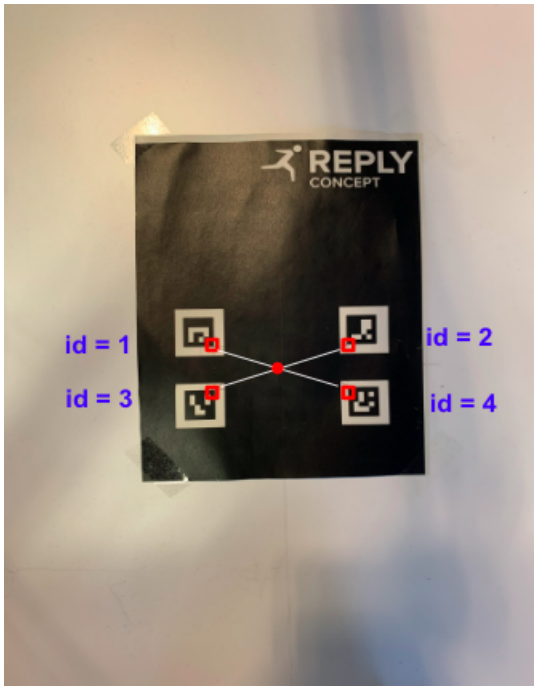


Figure 7.5: Detected ArUco markers with the midpoint

The calculation of the midpoint coordinates is fundamental to compute the translation vector t_{wp} of the plane frame with respect to the world reference frame using the *get3Dcoord_from_pixelCoord()* function. The plane reference frame is the one centered on the midpoint

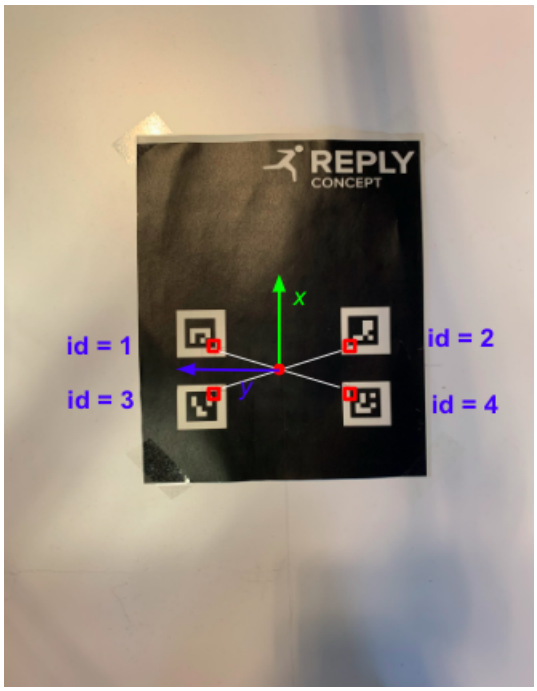


Figure 7.6: Plane reference frame

with z axis exiting the plane.

7.5 Object detection and localization

Object detection and localization are fundamental tasks in the field of computer vision. The ability to accurately identify and precisely locate objects within images or video streams is paramount for numerous domains, including autonomous driving, surveillance, robotics and augmented reality, among others.
In this section, we will explore the underlying principles behind these two tasks which are implemented in the **object_detection_and_localization** node. The latter relies on some parameters provided by the **calibrationValidation_and_markerDetection** node and, additionally, it requires the image of the plane, which is provided by the **camera** node. This image is captured and transmitted once some objects have been randomly placed on the plane as in the Figure 3.11. In the following, we can see how the three nodes mentioned above communicate with each other.

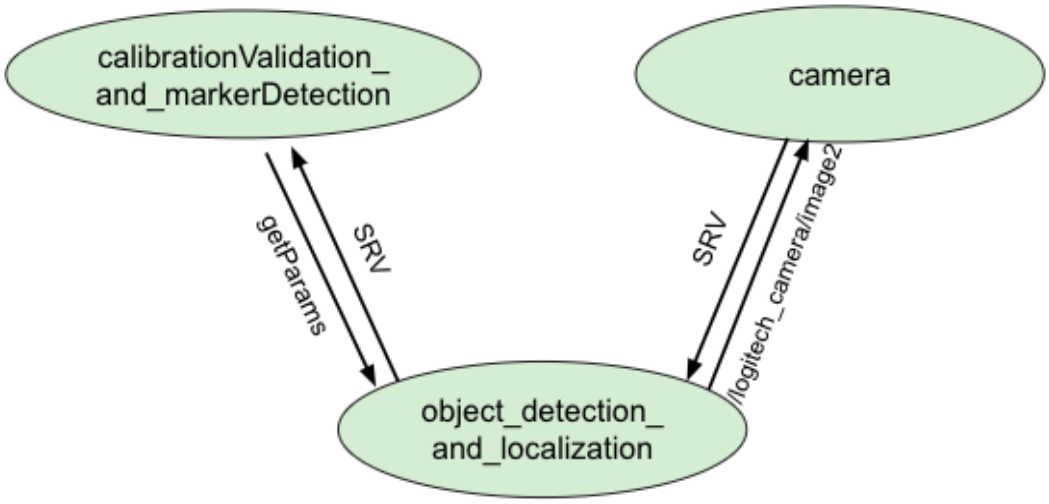


Figure 7.7: /logitech_camera/image2 and getParams services

In light of our understanding, the **object_detection_and_localization** node acts as client towards the other nodes through two functions. The first one is *get_params()*, which plays a crucial role in retrieving the necessary parameters using a service called *getParams*. This function serves as a client to the service and its purpose is to gather specific calibration and transformation parameters that are required for the two desired tasks.

```
def get_params():
    params_client = rospy.ServiceProxy('getParams', Params)
    message_request = "params"
    response = params_client(message_request)

    mtx = np.array(response.mtx).reshape((3,3))
    dist = np.array(response.dist).reshape((1,5))
    r_wc = np.array(response.rvec).reshape((3,1))
    t_wc = np.array(response.tvec).reshape((3,1))
    scaling_factor = response.scaling_factor
    midpoint = np.array(response.midpoint).reshape((2,1))
```

```
    return mtx, dist, r_wc, t_wc, scaling_factor, midpoint
```

The function establishes a service client named *params_client* and it prepares a message request to query the service for the required parameters. Then, it sends the message request to the service and captures the response which contains several parameters: camera matrix, distortion coefficients, rotation vector from camera to world frame, translation vector from camera to world frame, scaling factor and the midpoint's pixel coordinates.

The second function is *getImage()* wich serves the purpose of acquiring an image from the camera using another ROS service named */logitech_camera/image2*.

```
def getImage():
    get_image = rospy.ServiceProxy('/logitech_camera/image2',
   CameraFrame)

    message = "frame"
    response = get_image(message)

    img = response.image
    # Initialize CvBridge object for converting between OpenCV and
    # ROS image formats
    bridge = CvBridge()
    cv_image = bridge.imgmsg_to_cv2(img, desired_encoding='
   passthrough')

    return cv_image
```

This function establishes a service client which is configured to communicate with the service. Then, a message request with the content "frame" is prepared. This message informs the service to provide a camera frame. The function sends the message request to the service using the *get_image* client and captures the response, which contains an image in ROS format. In the end, the ROS image is converted to an OpenCV-compatible format using the *imgmsg_to_cv2()* function from the *CvBridge* object and returned.

The server of the *getParams* service is the **calibrationValidation_and_markerDetection** node, in which we implement the following function.

```
def get_params(mtx, dist, r_wc, t_wc, scaling_factor, midpoint):
    parameters_srv = rospy.Service("getParams", Params,
   params_callback)
    rospy.spin()
```

After initializing the ROS service with the provided matrices, values and parameters, it enters a loop to keep the node running and listening for incoming service requests. The *params_callback()* function is invoked when the service is requested

and it checks for a specific request message ("params"). If the request matches, it converts various matrices and values (such as camera matrix, distortion parameters, rotation and translation vectors, midpoint coordinates and scaling factor) into flattened numpy arrays of *float32* type and then packages them into a *ParamsResponse* object to be sent back as a response to the service request.

```
def params_callback(request):
    req = request.message

    if req == "params":
        array_mtx = mtx.flatten().astype(np.float32)
        array_dist = dist.flatten().astype(np.float32)
        array_r_wc = r_wc.flatten().astype(np.float32)
        array_t_wc = t_wc.flatten().astype(np.float32)
        array_midpoint = midpoint.flatten().astype(np.float32)
        return  ParamsResponse(array_mtx, array_dist, array_r_wc,
   array_t_wc, array_midpoint, scaling_factor)
```

Regarding the */logitech_camera/image2* service, the relative server is the **camera** node which has been describe in the previous section.

Once all the requisite elements have been obtained, we create an instance of the Operations class to use *set_camera_attributes()* and *set_scaling_factor()* methods to set the camera attributes and the scaling factor. Before proceeding with the implementation of the function that allows us to determine the position of the midpoint in the base reference frame, we show all the reference frames and the transformations between them.

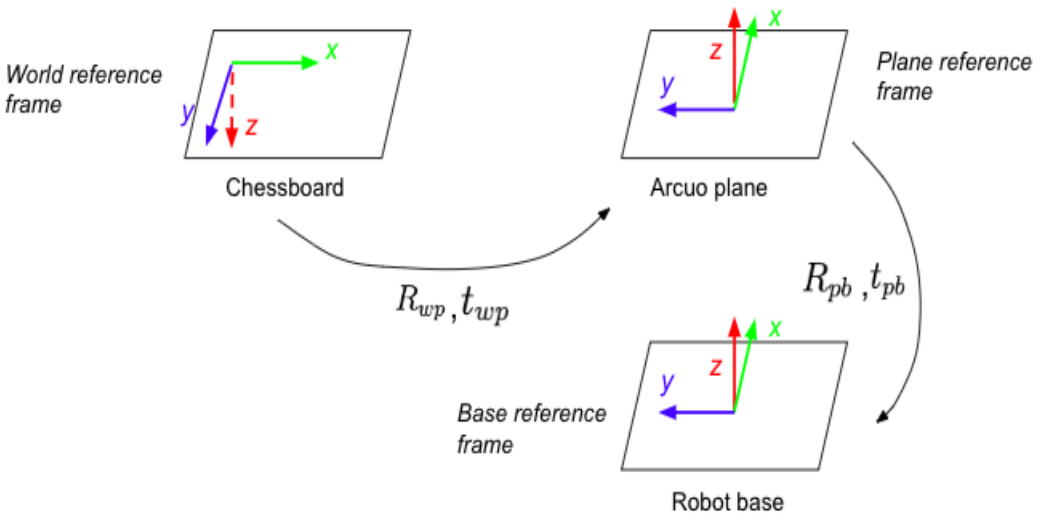


Figure 7.8: Reference frames and relative transformations

The base reference frame is centered on the robot base. The following table defines all the transformations symbols.

Symbols	Definition	Value
R_{wp}	Rotation matrix of the plane wrt world frame	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$
t_{wp}	Translation vector of the plane wrt the world frame	Computed in Sec 7.4
R_{pb}	Rotation matrix of the base wrt the plane frame	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
t_{pb}	Translation vector of the base wrt the plane frame	$\begin{bmatrix} -0.3 \\ 0 \\ 0 \end{bmatrix}$

Table 7.1: Definition and value of the transformations symbols

The translation vector t_{pb} indicates the distance between the robot base centre and the midpoint within the ArUco markers. Its value has negative x because the vector is defined with respect to the plane reference frame.

As anticipated, we can now compute the midpoint position through the following function. The primary purpose of it is to validate the transformations between different reference frames and, in particular, verify if the computed midpoint is exactly 30 cm from the base frame.

```
def midpoint_position_in_baseFrame(u, v):
    # This function is used to verify if the midpoint seen by
    # the camera is exactly at x = 0.3 from the base frame.
    # Therefore, it is used to validate the trasformations between
    # the different ref frames.
    # Moreover, it is used to set the translation vector t_wp.
    midpoint_3Dcoord = operation_obj.get3Dcoord_from_pixelCoord(u,
   v) # pixel coordinates of the midpoint among the markers
    midpoint_3Dcoord = np.array(midpoint_3Dcoord).reshape(2,1)

    null_value = np.array([[0]])
    midpoint_3Dcoord = np.append(midpoint_3Dcoord, null_value, axis
   =0)
    operation_obj.set_t_wp(midpoint_3Dcoord)

    midpoint_planeframe = operation_obj.
   getCoord_planeframe_from_worldframe(operation_obj.R_wp,
   operation_obj.t_wp, midpoint_3Dcoord)
    midpoint_planeframe = midpoint_planeframe.reshape((3,1))
    print("Coords of the midpoint with respect to the plane frame:
   \n")
    print(midpoint_planeframe)

    midpoint_baseframe = operation_obj.
   get3DCoord_baseframe_from_planeframe(operation_obj.R_pb,
   operation_obj.t_pb, midpoint_planeframe)
    print("Coordinates of plane frame's origin with respect to the
```

```
    base frame: \n")
     print(midpoint_baseframe)
```

The function takes the midpoint's pixel coordinates in the camera's image as the inputs and it starts by calculating the 3D coordinates of the midpoint using the *get3Dcoord_from_pixelCoord()* method of the Operations class. This calculated 3D coordinate is then extended with a null value to create a 3D coordinate in the format required for the transformation calculations.
Next, we apply transformations to determine the midpoint's coordinates in the plane reference frame and subsequently in the base reference frame. These transformations involve using the rotation and translation matrices shown in the previous table. The result, *midpoint_baseframe*, represents the coordinates of the midpoint in the base reference frame.

After having confirmed the accuracy of the previous step's outcome, we shift our focus to the function that enables the calculation of object positions, given the image provided by the camera.

```
def object_position(img):
    # Define the range of color to threshold in HSV
    lower_green = np.array([40, 50, 50])
    upper_green = np.array([80, 255, 255])
    lower_yellow = np.array([20, 100, 100])
    upper_yellow = np.array([30, 255, 255])
    lower_red = np.array([150, 100, 100])
    upper_red = np.array([190, 255, 255])

    # Convert the image from the RGB color space to the HSV
    # color space
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # Create a binary mask by thresholding the image based on the
    # desired color range
    green_mask = cv2.inRange(hsv, lower_green, upper_green)
    yellow_mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    # Apply morphological operations to the binary mask to remove
    # noise and smooth the edges
    kernel = np.ones((5,5), np.uint8)
    green_mask = cv2.erode(green_mask, kernel, iterations=1)
    green_mask = cv2.dilate(green_mask, kernel, iterations=1)

    yellow_mask = cv2.erode(yellow_mask, kernel, iterations=1)
    yellow_mask = cv2.dilate(yellow_mask, kernel, iterations=1)

    # Find contours in the filtered binary mask
    contours1, hierarchy = cv2.findContours(green_mask,cv2.
   RETR_TREE,cv2.CHAIN_APPROX_SIMPLE) # green contours
    contours2, hierarchy = cv2.findContours(yellow_mask,cv2.
   RETR_TREE,cv2.CHAIN_APPROX_SIMPLE) # yellow contours
```

```
    centers_list = [] # list of all computed centers
    greenCenters_list = []
    yellowCenters_list = []

    # Define the threshold for grouping close centers
    center_threshold = 2.5

    # In the following two cycles, I do the same procedure for
    # both colors. First of all, I compute the center for each
    # contour.
    # It can happen that the algorithm determines two or more
    # centers very closer. For each center, I compute the distance
    # between it and other centers. If it is less than a
    # particular threshold, I consider them as a unique center,
    # computing their mean.

    # Initialize the list of centers found
    centers = []
    close_centers = []
    # Draw a circle at the center of mass of each green object
    for cnt in contours1:
        M = cv2.moments(cnt)
        if M['m00'] > 0:
            cx = int(M['m10']/M['m00'])
            cy = int(M['m01']/M['m00'])
            #cv2.circle(img, (cx, cy), 5, (0, 0, 0), -1)
            # Check if the current center is close to a previous
            # center
            close_center = False
            for center in centers:
                distance = np.sqrt((cx-center[0])**2 + (cy-center
[1])**2)
                if distance < center_threshold:
                    close_center = True
                    close_centers.append([cx, cy])
                    break
            # If the current center is not close to any previous
            # center, add it to the list of centers
            if not close_center:
                centers.append([cx, cy])

    # Draw a circle at each center of mass
    for center in centers:
        greenCenters_list.append(center)
        cv2.circle(img, (center[0], center[1]), 5, (0, 255, 0), -1)

    # Draw a circle at the center of mass for any close centers
    #if len(close_centers) > 0:
    #    x_mean = int(np.mean([center[0]
    # for center in close_centers]))
    #    y_mean = int(np.mean([center[1]
    # for center in close_centers]))
    #    greenCenters_list.append([x_mean, y_mean])
    #    cv2.circle(img, (x_mean, y_mean), 5, (0, 0, 0), -1)

    # Show the frame with the detected objects
```

```
    #cv2.imshow('Camera frame', img)

    print("The center coordinates of the green objects are: \n")
    print(greenCentres_list)

    # Initialize the list of centers found
    centers = []
    close_centers = []
    # Draw a circle at the center of mass of each yellow object
    for cnt in contours2:
        M = cv2.moments(cnt)
        if M['m00'] > 0:
            cx = int(M['m10']/M['m00'])
            cy = int(M['m01']/M['m00'])
            #cv2.circle(img, (cx, cy), 5, (0, 120, 150), -1)
            # Check if the current center is close to a previous
            # center
            close_center = False
            for center in centers:
                distance = np.sqrt((cx-center[0])**2 + (cy-center
    [1])**2)
                if distance < center_threshold:
                    close_center = True
                    close_centers.append([cx, cy])
                    break
            # If the current center is not close to any previous
            # center, add it to the list of centers
            if not close_center:
                centers.append([cx, cy])

    # Draw a circle at each center of mass
    for center in centers:
        yellowCenters_list.append(center)
        cv2.circle(img, (center[0], center[1]), 5, (0, 120, 150),
    -1)

    # Draw a circle at the center of mass for any close centers
    #if len(close_centers) > 0:
    #    x_mean = int(np.mean([center[0]
    # for center in close_centers]))
    #    y_mean = int(np.mean([center[1]
    # for center in close_centers]))
    #    yellowCenters_list.append([x_mean, y_mean])
    #    cv2.circle(img, (x_mean, y_mean), 5, (0, 0, 0), -1)

    # Show the frame with the detected objects
    #cv2.imshow('Camera frame', img)

    print("The center coordinates of the yellow objects are: \n")
    print(yellowCenters_list)

    # Show the frame with the detected objects
    cv2.imshow('Camera frame', img)
    cv2.waitKey(0)
    #cv2.destroyAllWindows()
```

```
    return greenCenters_list, yellowCenters_list
```

This function starts defining the HSV color ranges for the desired colors (green and yellow) using lower and upper bounds. Subsequently, the input image is converted from the RGB color space to the HSV color space. This color space transformation allows the creation of binary masks through thresholding within the specified color ranges. These masks effectively separate the regions of the image containing the desired colors.
Following this step, we apply morphological operations (erosion and dilation) to the binary masks to reduce noise and smooth the mask edges. After that, the function identifies contours in the binary masks using the *cv2.findContours()* function. Each contour corresponds to a detected object.
For each detected contour, the center of mass (centroid) is computed using the contour moments. In cases where multiple contours are close to each other, their centers are grouped together trough the *center_threshold*. Circles are then drawn around the centers of the detected green and yellow objects on the input image.
Finally, the function returns lists that hold the coordinates of the detected centers for both green and yellow objects.

The resulting image with the marked object centers is like Figure 3.12 where it can be observed that a few points (in this case only one) are not exactly at the center of their respective objects. The accuracy of the object centers can be influenced by several factors in the computer vision system. Some of them include:

- object position relative to the camera: the precise position of objects in relation to the camera plays a crucial role. Variations in distance, angle and orientation can affect the accuracy of center calculations;

- color mask definitions: the definition of color masks used in the detection process can impact the accuracy. Fine-tuning the color thresholds and filters may be necessary to distinguish objects effectively, especially in cases with similar or varying colors;

- lighting conditions: ambient lighting conditions at the time of image capture can introduce variations in object appearance. Changes in brightness, shadows or reflections can challenge the detection algorithm and affect the accuracy of center calculations;

- thresholding parameters: the choice of thresholding parameter can impact the accuracy of the calculated centers. Adjusting these parameter may be required to achieve more precise results.

- camera calibration: a poorly calibrated camera can cause geometric distortions in images, affecting localization accuracy;

- image resolution: it can impact accuracy. High-resolution images can allow for more precise localization of object details.

The next important stride involves transforming the object centers from pixel coordinates to well-defined coordinates with respect to the base reference frame. This conversion is essential as the robot employs the base reference frame for its movements.

```
def objectPosition_in_baseFrame(point_list):
    # This function takes the list of center positions and
    # compute them with respect to the base frame
    # and returns them.
    centresBaseFrame_list = [] # list of centres with respect to
    # the base frame
    for point in point_list:
        u = point[0]
        v = point[1]
        objPosition_3Dcoord = operation_obj.
   get3Dcoord_from_pixelCoord(u, v)
        objPosition_3Dcoord = np.array(objPosition_3Dcoord).reshape
   (2,1)
        null_value = np.array([[0]])
        objPosition_3Dcoord = np.append(objPosition_3Dcoord,
   null_value, axis=0)
        objPosition_planeframe = operation_obj.
   getCoord_planeframe_from_worldframe(operation_obj.R_wp,
   operation_obj.t_wp, objPosition_3Dcoord)
        objPosition_planeframe = objPosition_planeframe.reshape
   ((3,1))
        objPosition_planeframe = objPosition_planeframe/100 # in
        # order to match the units of measure between plane
        # and base ref frames
        objPosition_baseframe = operation_obj.
   get3DCoord_baseframe_from_planeframe(operation_obj.R_pb,
   operation_obj.t_pb, objPosition_planeframe)
        centresBaseFrame_list.append(objPosition_baseframe)
    return centresBaseFrame_list
```

This function iterates through each point of the list taken as input (object centers list) and computes the real-world 3D coordinates of the object's position based on its pixel coordinates. The coordinates are then transformed from the world frame to the plane frame and further to the base frame using the provided transformation matrices. The resulting positions are stored in a list and returned. Additionally, the units of measure between the plane and base reference frames are matched by dividing the computed positions by 100.

7.6 Robot motion

The final crucial step involves transmitting the previously computed object centers to the node responsible for maneuvering the robot, **moveToPose**. The service which allows this operation is the following.

Figure 7.9: move_to service

As we can understand from the previous figure, the **object_detection_and_localization** node acts as a server through the implementation of the *get_coords* function

```
def get_coords(goal_poses):
    moveTo_srv = rospy.Service('move_to', MoveTo, coords_callback)
    rospy.spin()
```

which sets up the service and utilizes the *coords_callback()* function to respond to service requests with the appropriate goal poses.

```
def coords_callback(request):
    req = request.message
    if req == "coords":
        array_goal_pose = goal_poses.flatten().astype(np.float32)
        return  MoveToResponse(array_goal_pose)
```

If the request message is "coords", the callback function flattens the provided *goal_poses* array and returns it as a response.

In order to make the final objective more engaging and distinctive, we allow the user to select which type of objects the robot should move towards: green, yellow or all.

```
value = input("Write 1 (green obj), 2 (yellow obj) or 3 (all obj)
   to move the robot towards green, yellow or all objects: ")
    if str(value) == "1":
        goal_poses = np.array(greenCentersBaseFrame_list)
        get_coords(goal_poses)
    elif str(value) == "2":
        goal_poses = np.array(yellowCentersBaseFrame_list)
        get_coords(goal_poses)
    elif str(value) == "3":
        goal_poses = np.array(complete_list)
        get_coords(goal_poses)
    else:
        print("Wrong input!")
```

Based on the user choice, the *get_coords()* function is called with the corresponding goal positions.

In the client node, we start by retrieving the list of goal poses through the following function.

```
def get_coords():
    moveTo_client = rospy.ServiceProxy('move_to', MoveTo)
    message_request = "coords"
    response = moveTo_client(message_request)
    goal_poses = np.array(response.goal_poses)
    return goal_poses
```

In particular, it creates a client for the *move_to* service and prepares a request message with the content "coords" to be sent to the previous server. After receiving the response, it extracts the *goal_poses* attribute from the response, which contains an array of goal pose values.

After that, we initialize the ROS node and MoveIt! interfaces are created for both the robot and the MoveGroupCommander.

```
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_to_pose', anonymous=True)
# instantiate a MoveGroupCommander object
robot = moveit_commander.RobotCommander()
# instantiate a RobotCommander object
move_group = moveit_commander.MoveGroupCommander("lite6")
```

Subsequently, we iterate over the list of goal poses retrieved and processed from the service response.

```
for i in range(0, len(goal_poses), 3):
        # consider the single goal pose
        goal_pose = [goal_poses[i], goal_poses[i+1], goal_poses[i
   +2]]
        print("")
        print("The robot moves to the following goal pose: \n")
        print(goal_pose)
        x = float(goal_pose[0])
        y = float(goal_pose[1])
        z = 0.15 # Based on the object and end-effector's height
        move_to_goal_pose(x, y ,z)
        go_gome()
```

For each goal pose, we first extract the x and y coordinates. Next, we set a fixed value for the z-coordinate, taking into account both the height of the objects and that of the custom-made wooden end-effector.

Figure 7.10: Custom-made end-effector

This is built to better verify the accuracy of the robot's achieved position after its movements. In the end, we call the *move_to_goal_pose()* function with the extracted x, y and z coordinates and the *go_home()* function.

```
def move_to_goal_pose(x, y, z):
    print("Plannning to a pose goal ...")
    target_pose = geometry_msgs.msg.Pose()
    target_pose.position.x = x
    target_pose.position.y = y
    target_pose.position.z = z

    qx = 1
    qy = 0
    qz = 0
    qw = 0
    target_pose.orientation.x = qx
    target_pose.orientation.y = qy
    target_pose.orientation.z = qz
    target_pose.orientation.w = qw

    move_group.set_pose_target(target_pose,"link6")
    move_group.set_goal_tolerance(0.0005)
    plan = move_group.plan()

    move_group.go(wait=True)
    print("The robot has reached the pose goal!")
    move_group.stop()

    move_group.clear_pose_targets()
    rospy.sleep(5)
```

This function creates a *target_pose* object of type *geometry_msgs.msg.Pose()* and

assigns the given values x, y and z to its position attributes. This defines the position where the robot should move. The orientation of the target pose is defined using the quaternions. Since we are not concerned about the robot reaching the objects with a specific orientation, we define it in a way that the robot accomplishes its goal. Then, we use *move_group.set_pose_target()* to set the desired pose goal for the robot's movement, specifying that it is associated with "link6". Moreover, we set a tolerance for how close the robot needs to reach to the goal pose using *move_group.set_goal_tolerance()*. After planning the trajectory, we execute it by calling *move_group.go(wait=True)*, where the parameter *wait=True* means the function will wait until the movement is completed before proceeding. In conclusion, we stop the robot's movement and clear the pose targets.

The result of this first operation can be seen in the following figures.

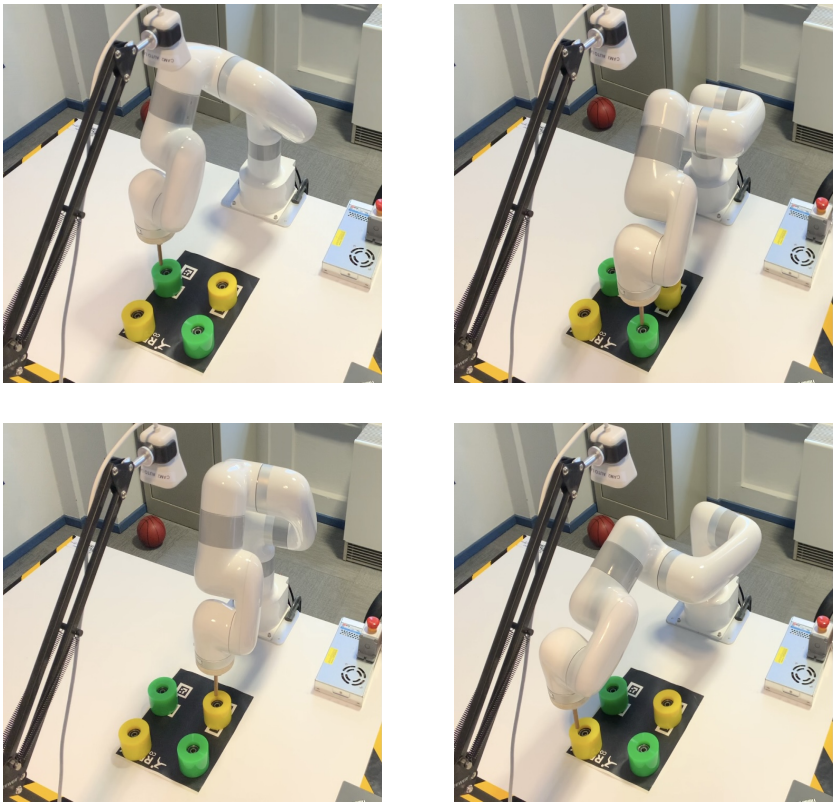


Figure 7.11: Robot goal poses

The *go_home()* function is defined to move the robot back to its home position.

```
def go_gome():
    print("Plannning to  home pose ...")

    target_pose = geometry_msgs.msg.Pose()
    target_pose.position.x = 0.087
    target_pose.position.y = 0
    target_pose.position.z = 0.1542
```

```
    roll = m.pi
    pitch = 0.0
    yaw = 0.0

    qx = np.sin(roll/2)*np.cos(pitch/2)*np.cos(yaw/2) - np.cos(roll
   /2)*np.sin(pitch/2)*np.sin(yaw/2)
    qy = np.cos(roll/2)*np.sin(pitch/2)*np.cos(yaw/2) + np.sin(roll
   /2)*np.cos(pitch/2)*np.sin(yaw/2)
    qz = np.cos(roll/2)*np.cos(pitch/2)*np.sin(yaw/2) - np.sin(roll
   /2)*np.sin(pitch/2)*np.cos(yaw/2)
    qw = np.cos(roll/2)*np.cos(pitch/2)*np.cos(yaw/2) + np.sin(roll
   /2)*np.sin(pitch/2)*np.sin(yaw/2)

    target_pose.orientation.x = qx
    target_pose.orientation.y = qy
    target_pose.orientation.z = qz
    target_pose.orientation.w = qw

    move_group.set_pose_target(target_pose,"link6")
    move_group.set_goal_tolerance(0.0005)

    plan = move_group.plan()
    move_group.go(wait=True)
    print("The robot has reached the home pose!")

    move_group.stop()
    move_group.clear_pose_targets()
```

It creates the *target_pose* using the *geometry_msgs.msg.Pose()* class. The position coordinates (x, y and z) are set to specific values, provided by the xArm Studio software of UFactory. Regarding the orientation, we set the Euler angles (*roll, pitch,* and *yaw*) which are used to calculate the quaternions. Subsequently, we set the pose target and the goal tolerance as in the previous function. Then, the trajectory plan is generated and the robot is instructed to execute it. Finally, the robot's motion is stopped and the target pose information is cleared from the MoveIt group.

The following figures show the result of this function.

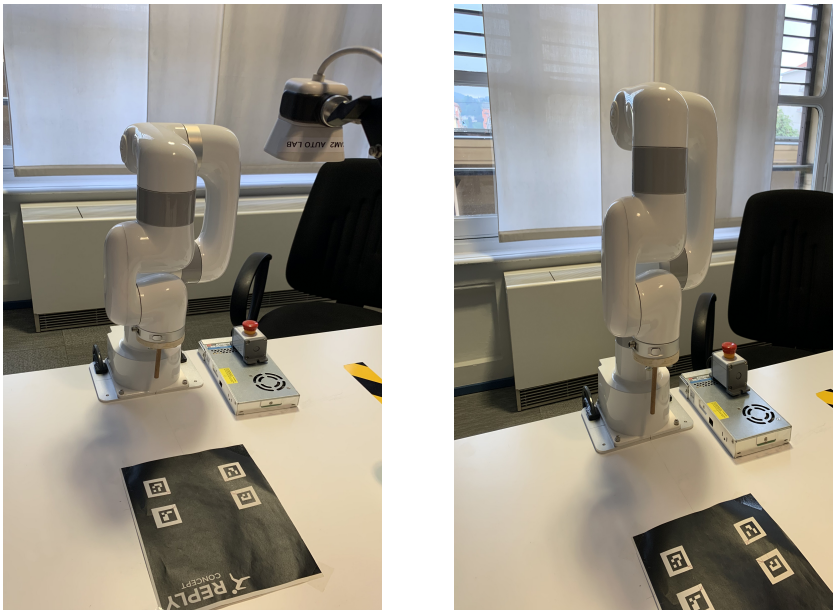


Figure 7.12: Robot home pose

Chapter 8

Conclusions

In this thesis we have successfully achieved the predefined objectives, marking a significant step forward in the integration of computer vision with robotics. The development of an intuitive and user-friendly framework for the UFactory Lite 6 robotic arm, controlled through the ROS architecture, has paved the way for seamless interaction with the robot in various applications, especially for those who will use it in the automotive field.

One of the primary accomplishments of this work is the creation of a framework that simplifies the utilization of the robotic arm, offering a versatile platform for a wide range of projects. This framework not only facilitates trajectory recording and playback but also provides the capability to incorporate computer vision techniques. By integrating object detection and localization within the ROS framework, this work demonstrates the potential of the system to perceive and interact with its environment intelligently.

The successful deployment of an external camera for capturing the visual scene and extracting object positions is a crucial component of the project. This achievement enables the robotic arm to execute precise movements towards detected objects, exemplifying the potential of this technology in real-world applications.

The outcomes of this thesis project hold promise for various industries, particularly in the field of automation and robotics. The combination of computer vision and robotics provides a powerful toolset for enhancing flexibility, intelligence and adaptability in automation systems. This work serves as a testament to the potential of such integration and opens up new avenues for innovation and development.

As we move forward, further refinements and enhancements to the framework can be explored, expanding its capabilities and applications. The synergy between computer vision and robotics continues to be a dynamic and evolving field, and this work is a valuable contribution to its advancement. With the foundation laid by this thesis, the future looks promising for the integration of advanced robotic systems into a wide array of projects and industries, driving innovation and efficiency in automation.

References

[1] Peter I Corke, Witold Jachimczyk, and Remo Pillat. *Robotics, vision and control: fundamental algorithms in MATLAB*, volume 73. Springer, 2011.

[2] Flask. `https://flask.palletsprojects.com/en/2.3.x/`.

[3] OpenCV. `https://opencv.org/`.

[4] UFactory Lite 6 robotic arm. `https://www.ufactory.cc/lite-6-collaborative-robot/`.

[5] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Force control*. Springer, 2009.

[6] Logitech Streamcam. `https://www.logitech.com/it-it/products/webcams/streamcam.960-001281.html`.

[7] ROS: Robot Operating System. `https://www.ros.org/`.