University of Camerino
School of Science and Technology
Master Degree in Computer Science

Autonomous and Collaborative Robotics: Simulating multi-robot collaboration

Supervisor
Prof. Alessandro Marcelletti

Students
Francesco Finucci

Academic Year 2024-2025

Abstract

This project presents the development of a multi-robot system consisting of a robotic arm, a webcam and a human that simulates a robotic car. The system simulates a portual drop-off scenario where the car transports a container to the robotic arm, which then picks it up and places it back onto the ship. The collaboration between the robots is achieved using computer vision for object detection and mathematical algorithms for trajectory planning and coordination. The document details the system architecture, algorithms, implementation, and experimental results.

Contents

Part I

PROLOGUE

1

Introduction

1.1 MOTIVATION

In recent years, the development of autonomous robotic systems has gained significant attention in both research and industrial applications. The ability of multiple robots to collaborate efficiently can lead to improvements in automation, logistics, and smart manufacturing. One practical scenario where such collaboration is essential is in automated transportation and handling of goods, such as in port operations.

This project explores a multi-robot system where we simulate two different arms working together to transport and manipulate containers. The motivation behind this study is to design a system that integrates **computer vision** for perception and **mathematical algorithms** for motion planning, enabling smooth coordination between the two robots. By simulating a portual drop-off scenario, this work aims to demonstrate how **robotic collaboration** can improve efficiency and precision in automated handling tasks.

1.2 PROBLEM STATEMENT

The challenge addressed in this project is the **autonomous coordination of a robotic arm** in a controlled environment. The system simulates a logistics scenario where containers, represented by 3D-printed LEGO bricks, must be picked up and placed in predefined locations. While the initial plan included a robotic car for transportation, due to technical limitations, the process of moving the bricks to and from the robotic arm is performed manually.

Several technical challenges must be tackled, including **effective communication** between system components, **accurate object detection** to identify the bricks' position, and **reliable motion planning** to ensure precise grasping and placement. The system must be robust enough to handle variations in object placement and respond dynamically to real-time feedback.

1.3 OBJECTIVES

The primary objective of this project is to develop a functional multi-robot system capable of collaborating in a simulated portual drop-off scenario. This involves implementing a **computer vision pipeline** for detecting and tracking the container, designing an efficient **motion planning strategy**, and ensuring **real-time communication** for synchronized execution. Additionally, the system's **performance and reliability** will be evaluated through experimental testing.

1.4 OVERVIEW OF APPROACH

The project consists of several key stages. The system simulates a logistics scenario where a container, represented by a 3D-printed LEGO brick, is manually placed within the robotic arm's workspace. The robotic arm is equipped with a vision system to detect the brick and determine its position. Once identified, the arm uses **inverse kinematics** to precisely grasp the container and reposition it at a predefined location.

To achieve seamless execution, a combination of **image processing techniques**, **machine learning models** for object detection, and **mathematical algorithms** for motion planning is employed. The control logic ensures accurate perception and movement, allowing the system to respond dynamically to variations in object placement.

1.5 STRUCTURE OF THE DOCUMENT

This document is structured as follows:

- **Chapter 2**: Background information on multi-robot systems, computer vision, and motion planning.
- **Chapter 3**: System architecture, including hardware and software components.
- **Chapter 4**: Mathematical and algorithmic foundations of the project.
- **Chapter 5**: Implementation and integration of different subsystems.
- **Chapter 6**: Experimental results, analyzing the system's performance and limitations.
- **Chapter 7**: Conclusion with a summary of findings and potential future improvements.
- **Appendices**: Additional materials such as code snippets, experimental data, and hardware schematics.

2

Background and Related Work

2.1 MULTI-ROBOT COLLABORATION IN LOGISTICS

The use of multi-robot systems in logistics has been extensively studied due to the increasing demand for automation in warehouses, factories, and transportation hubs. The ability of multiple robots to work together efficiently can enhance productivity, reduce human intervention, and optimize resource allocation.

In logistics, robots often perform tasks such as **object transportation, sorting, and palletizing**. Coordinating multiple robots requires efficient task allocation strategies, real-time communication, and robust motion planning. Several approaches exist for achieving multi-robot collaboration, including **centralized coordination**, where a central controller assigns tasks, and **decentralized coordination**, where robots make independent decisions based on shared information.

Recent advancements in **swarm robotics** and **reinforcement learning** have further improved coordination strategies, enabling robots to dynamically adapt to changes in the environment. Applications of such systems can be seen in **automated guided vehicles (AGVs)**, robotic sorting systems in fulfillment centers, and port automation, where autonomous vehicles transport cargo containers efficiently.

2.2 COMPUTER VISION IN ROBOTICS

Computer vision plays a crucial role in enabling robots to perceive and interact with their surroundings. In robotics, vision-based systems are used for object detection, localization, and recognition, allowing robots to make informed decisions based on visual data.

A key challenge in computer vision for robotics is achieving **robust real-time perception**. This often involves processing images from cameras using techniques such as **feature extraction, edge detection, and deep learning-based object recognition**. In particular, convolutional neural networks (CNNs) have revolutionized robotic vision by significantly improving the accuracy of object detection and scene understanding.

For this project, computer vision is essential for the robotic car to detect and locate the container and for the robotic arm to determine

precise grasping points. By integrating **OpenCV-based image processing techniques** and **machine learning models**, the robots can perform reliable perception tasks even in dynamic environments.

2.3 TRAJECTORY PLANNING AND KINEMATICS

Trajectory planning is fundamental in robotics to ensure smooth and efficient motion. It involves computing a sequence of movements that allow a robot to reach a target location while avoiding obstacles and optimizing travel time.

For mobile robots like the robotic car, trajectory planning relies on algorithms such as **A* search, RRT (Rapidly-exploring Random Tree), and Dijkstra's algorithm**. These methods help determine optimal paths by evaluating possible routes and selecting the most efficient one. Additionally, motion control techniques, such as **PID controllers** and **model predictive control (MPC)**, ensure that the car follows the computed trajectory accurately.

For the robotic arm, trajectory planning is based on **inverse kinematics**, which calculates joint angles required to move the end effector to a desired position. Algorithms such as **Jacobian-based methods** and **iterative numerical solvers** are commonly used to solve inverse kinematics problems.

The integration of these techniques enables precise and coordinated movement, ensuring that both robots work together effectively in the simulated portual drop-off scenario.

2.4 RELEVANT RESEARCH AND IMPLEMENTATIONS

Several studies and industrial implementations have explored the use of multi-robot systems in automated logistics. Research on **multi-agent systems** has provided insights into how robots can collaborate autonomously while minimizing conflicts and optimizing efficiency.

In industrial settings, companies like **Amazon Robotics** have pioneered the use of mobile robots for warehouse automation, where fleets of autonomous robots transport and sort packages. Similarly, **Boston Dynamics** has developed robotic arms capable of manipulating objects with high precision using advanced computer vision and motion planning techniques.

Academic research has contributed to the development of **deep reinforcement learning-based coordination**, where robots learn optimal strategies through trial and error in simulated environments. Additionally, **graph-based planning approaches** have been used for optimizing robot navigation in structured environments. By studying these approaches, this project aims to incorporate effective methodologies for real-world applications in collaborative robotic systems.

Part II

IMPLEMENTATION & DOCUMENTATION

3

System Architecture

In the following chapter we are going to talk about the system architecture, the used technologies and programming languages.

3.1 OVERVIEW OF THE MULTI-ROBOT SYSTEM

We start by taking a look at the general architecture used for the ROS nodes. We can see the nodes in Figure 3.1 (you may need to zoom the image)

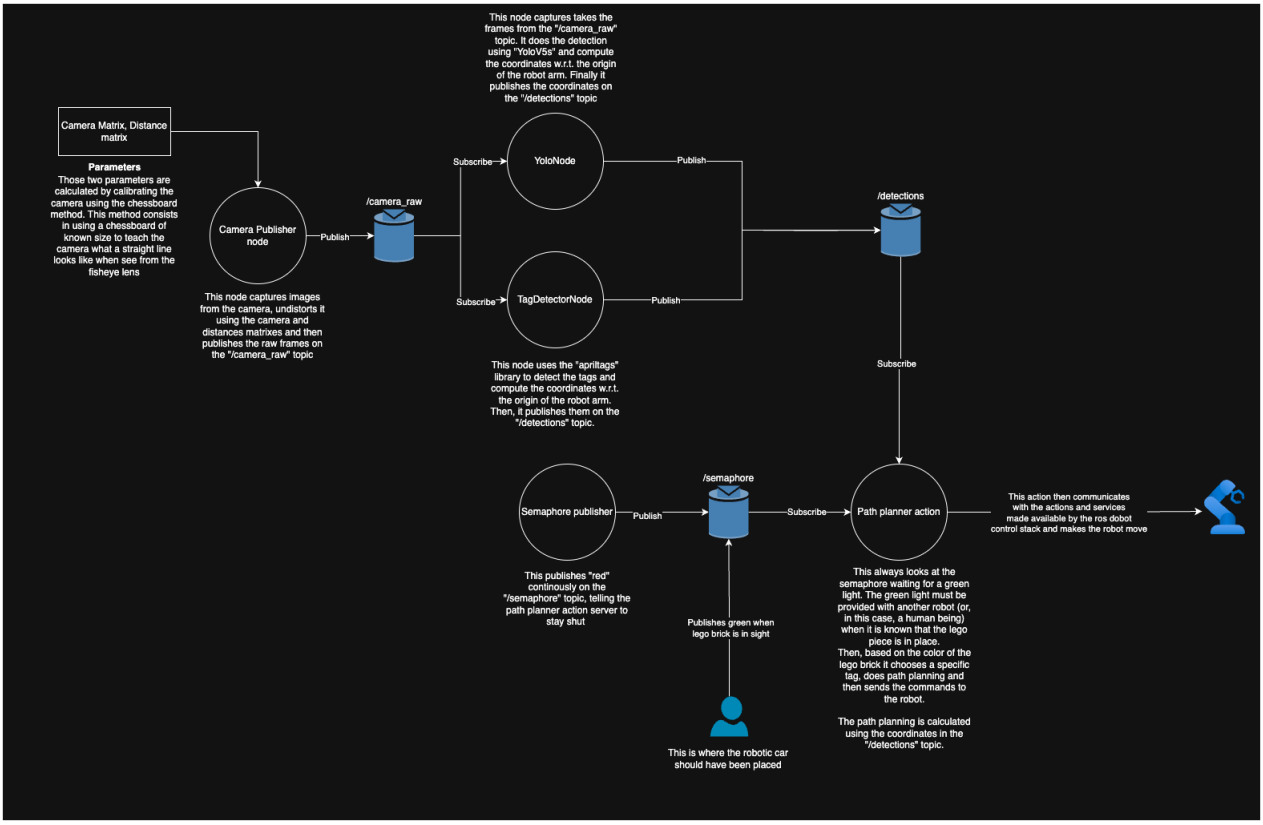

FIGURE 3.1: ROS Architecture

The architecture is compartmentalized and organized in order to assign to each and every single node a specific task. The first node uses the camera parameters and connects to a Logitech Webcam to extract

frames and publish it on a topic called **/camera_raw**. This topic contains the undistorted version of the image using the Camera and Distance matrices. The details of the undistortion will be discussed in Chapter **??**.

Once the frames are published on the topic we then use two different nodes, the first one is called **YoloNode** and it uses the **YoloV5s** model (the smaller version of the YoloV5 model) to compute classification and detection. More about that in Chapter **??**. The other node, called **TagDetectorNode**, uses the **apriltag** library to detect apriltags [1] that are placed on the table and are used to simulate the landing spots for the brick that is picked up by the robotic arm. Finally, both of those nodes publish the coordinates that are calculated w.r.t. the coordinates system of the arm on the **/detections** topic.

[1]AprilTags are a family of visual fiducial markers used in computer vision to provide precise location and orientation information. These markers are detected using camera systems, and their position and rotation are calculated relative to the camera's viewpoint, making them useful for robotic navigation, localization, and object tracking. The markers are robust to variations in lighting and scale, offering reliable tracking even in dynamic environments

Then, a semaphore (simulated using the **/semaphore** topic) handles the communication between the human operator and the robotic arm. The way it is done is that an action node listens to the semaphore and triggers the action when a green light is detected on the semaphore. The green light is published manually once the brick in in place and ready to be picked up. When the brick isn't there the red light is published continously telling the arm to stay shut. Once the action starts, it computes the path planning and then the robotic arm's control stack is called thus starting the arm movements.

3.1.1 *Robotic Car*

Initially, the robotic car that was being used was the Adeept Pi car PRO. It was later substituted and simulated by a human operator since the car itself was not working.

3.1.2 *Robotic Arm*

The robotic arm used in this project is the **Dobot Magician**, a versatile and cost-effective robotic arm developed by Dobot. It is designed for educational and research purposes, making it an ideal choice for prototyping and experimentation in robotic applications. The arm is equipped with multiple degrees of freedom (DOF), which allows it to perform a variety of tasks such as object manipulation, picking, and placing.

Key features of the Dobot Magician include:

- A **high precision control system** capable of sub-millimeter accuracy.
- An **integrated gripper** for object grasping and manipulation.
- A **modular design** that allows for easy customization with additional tools, such as a suction cup or pen holder.
- Compatibility with **various programming environments**, including ROS, Python, and Blockly.

The Dobot Magician's ability to perform precise movements and its ease of integration into existing systems make it a suitable choice for this project, where it collaborates with the robotic car to transport and manipulate containers in the simulated portual drop-off task.

3.1.3 *Communication and Synchronization*

For communication between the various nodes and the robotic arm, **Wi-Fi** was used to establish a reliable and fast network connection. The system is designed to facilitate real-time data exchange and coordination between the robotic car and arm through the use of **ROS topics**.

In this architecture, each robot and node in the system publishes and subscribes to specific topics, enabling them to share sensor data, status updates, and control commands.

The use of Wi-Fi ensures that data transmission occurs without the need for physical wiring, providing flexibility in positioning and scalability of the system.

3.2 HARDWARE COMPONENTS

The system is built using a combination of off-the-shelf and custom components to achieve the required functionality. The hardware setup includes:

- A **Raspberry Pi 4B+** as the central computing unit, providing sufficient processing power for controlling the robotic arm and car, running ROS, and handling computer vision tasks.
- A **Logitech webcam** for capturing visual data, which is used for object detection and localization in the task environment.
- The **Dobot Magician** robotic arm, which is used to pick up and place containers.
- **3D-printed LEGO bricks** that are used as objects to be manipulated and moved by both the robotic arm and the robotic car. These bricks serve as the containers in the task, representing the objects that need to be transferred between the robots.

This combination of hardware ensures a compact and efficient system that can handle the robotic operations required in the project.

3.3 SOFTWARE STACK

The software stack is designed to interface with the hardware components, manage communication, and control the robots. The stack includes:

3.3.1 *Operating System and Middleware (e.g., ROS)*

The operating system used in this project is **Ubuntu**, which serves as a stable and reliable environment for running various robotics software packages. On top of Ubuntu, the **Robot Operating System (ROS)** is used as the middleware. ROS provides a framework for building and managing robotic applications, offering several tools and libraries for:

- Real-time communication between various hardware and software components.

- Integration with sensors, actuators, and algorithms.
- A modular and scalable design for robotic systems.

ROS acts as the backbone of the system, ensuring smooth communication and coordination between the robotic arm, car, and other nodes.

3.3.2 *Programming Languages and Libraries*

The primary programming language used for the development of this system is **Python**, which is widely used in robotics due to its simplicity and flexibility. The libraries connected to ROS are essential for handling hardware interfaces, control logic, and communication. Key libraries and tools include:

- **ROS Python API** for interacting with ROS nodes and topics;
- **Magician ROS 2 control stack** for motion planning and control of the robotic arm. The stack available at the following GitHub repository: https://github.com/jkaniuka/magician_ros2/tree/main and also as Citation [Kan23]. The only modification that was made is about the **dobot_driver** node. Since it originally used the USB serial connection to communicate to the Robot it had to be converted into a Wi-Fi driver for the needs of this project;
- **OpenCV** for image processing and object detection, using the Logitech webcam;
- **NumPy and SciPy** for numerical computations and algorithms.

These tools, in combination with ROS, allow for efficient and robust development of the robotic system.

[Kan23] Kaniuka, "System sterowania robota manipulacyjnego Dobot Magician na bazie frameworka ROS2"

4

Mathematical and Computer Vision Algorithms

4.1 COMPUTER VISION FOR OBJECT DETECTION

4.1.1 *Camera Calibration and Image Processing*

Proper calibration of the camera is essential for accurate object detection. This involves determining the camera's intrinsic and extrinsic parameters to correct for distortions and improve spatial accuracy. Image processing techniques, such as **thresholding, edge detection, and filtering**, are applied to enhance object visibility before recognition.

The technique that was used is OpenCV's camera calibration using the chessboard method. The way of calibrating the camera using a chessboard is the most common way to remove distortions from the frames captured by the camera. It works by taking a number of photos of said chessboard in the scene like in Figure 4.1

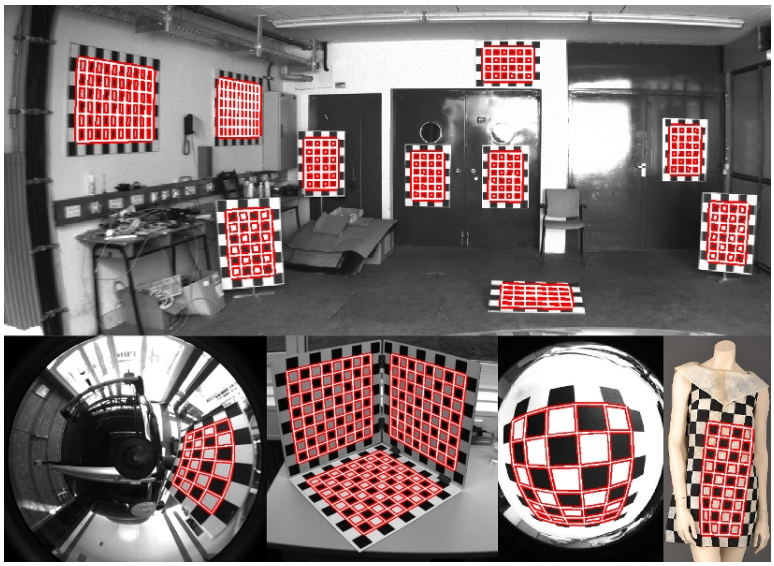

FIGURE 4.1: Example of chessboard pictures

This way, we are teaching the software to recognize a straight line when there is distortion of maybe a fish-eye lens. To achieve that, we need to compute a projection of the point in the 3D world inside a 2D frame. This is done by using the following equation

$$s\mathbf{m}' = A[R|t]\mathbf{M}' \tag{4.1}$$

or

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4.2}$$

where

- (X, Y, Z) are the coordinates of a 3D point in the world coordinate space;
- (u, v) are the coordinates of the projection point in pixels;
- A is a camera matrix, or a matrix of intrinsic parameters. In Equation 4.2, this matrix is expressed in terms of focal lenghts and principal points;
- (c_x, c_y) is a principal point that is usually at the image center;
- f_x and f_y are the focal lengths expressed in pixel units;
- s is a **scaling factor** that ensures the homogeneous coordinates remain valid.

In Figure 4.2 we can see how the **pinhole camera model** can be represented graphically

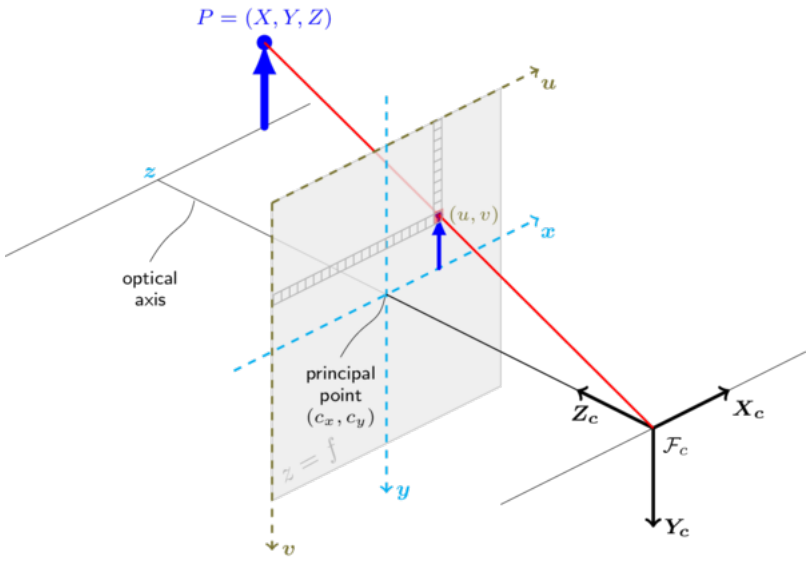

FIGURE 4.2: Illustration of the pinhole camera model

Thus, if an image from the camera is scaled by a factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The **matrix of intrinsic parameters** does not depend on

the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed (in case of zoom lens). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of a still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to a coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = \frac{x}{z}$$

$$y' = \frac{y}{z}$$

$$u = f_x \cdot x' + c_x$$

$$v = f_y \cdot y' + c_y$$

Real lenses usually have some distortion, mostly radial distortion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = \frac{x}{z}$$

$$y' = \frac{y}{z}$$

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$

where

$$r^2 = x'^2 + y'^2$$

$$u = f_x \cdot x'' + c_x$$

$$v = f_y \cdot y'' + c_y$$

k_1, k_2, k_3, k_4, k_5, and k_6 are radial distortion coefficients. p_1 and p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV.

Figure 4.3 shows two common types of radial distortion: barrel distortion (typically $k_1 > 0$) and pincushion distortion (typically $k_1 < 0$).

FIGURE 4.3: Different distortions

In the functions used for the undistortion the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2, [k_3, [k_4, k_5, k_6]]),$$

That is, if the vector contains four elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed. Thus, they also belong to the intrinsic camera parameters. And they remain the same regardless of the captured image resolution. If, for example, a camera has been calibrated on images of $320x240$ resolution, absolutely the same distortion coefficients can be used for $640x480$ images from the same camera while f_x, f_y, c_x, and c_y need to be scaled appropriately.

4.1.2 *3D Object Detection using YOLOV5s*

YOLOv5 is a model in the **You Only Look Once (YOLO)** family of computer vision models. *YOLOv5* is commonly used for detecting objects. *YOLOv5* comes in four main versions: small (s), medium (m), large (l), and extra large (x), each offering progressively higher accuracy rates. Each variant also takes a different amount of time to train.

Looking at Figure 4.4, we can see that the goal is to produce an object detector model that is very performant (Y-axis) relative to it's inference time (X-axis)

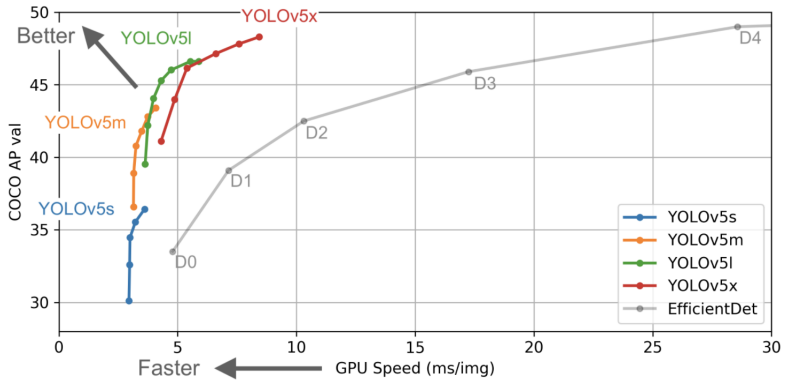

FIGURE 4.4: Yolo's speed vs efficiency

We can see that all variants of *YOLOv5* train faster than EfficientDet. The most accurate *YOLOv5* model, *YOLOv5x*, can process images multiple times faster with a similar degree of accuracy than the *EfficientDet D4* model.

We will not go into the depths of Yolo as the can be found in the YoloV5 paper [Joc+22], but let's take a look at the general architecture.

[Joc+22] Jocher et al., *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*

Object detection, a use case for which YOLOv5 is designed, involves creating features from input images. These features are then fed through a prediction system to draw boxes around objects and predict their classes As we can see in Figure 4.5

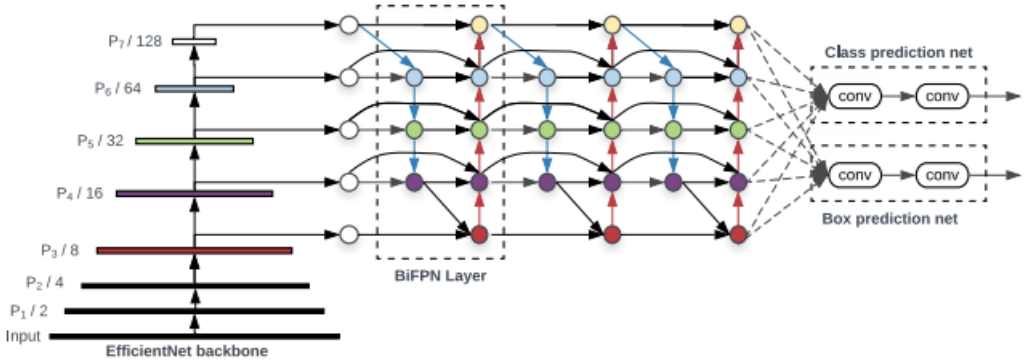

FIGURE 4.5: Object detection general architecture

The YOLO model was the first object detector to connect the procedure of predicting bounding boxes with class labels in an end to end differentiable network.

The YOLO network consists of three main pieces.

- **Backbone**: A convolutional neural network that aggregates and forms image features at different granularities;
- **Neck**: A series of layers to mix and combine image features to pass them forward to prediction,
- **Head**: Consumes features from the neck and takes box and class prediction steps.

In Figure 4.6 we can see the complete YoloV5 architecture including an explanation of every involved component

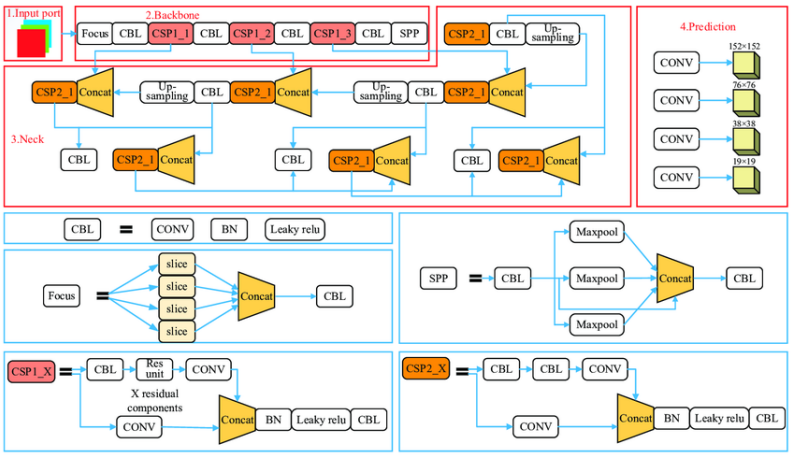

FIGURE 4.6: YoloV5 architecture

4.1.3 *Apriltags detection*

Developed at the *University of Michigan*, **AprilTag** is like a 2D barcode or a simplified QR Code. It contains a numeric ID code and can be used for location and orientation. AprilTag is a type of visual fiducial, or fiducial marker, containing information and designed for easy recognition. Apriltags are divided into families that can be used for a number of different purpouses, some of those families can be seen in Figure 4.7

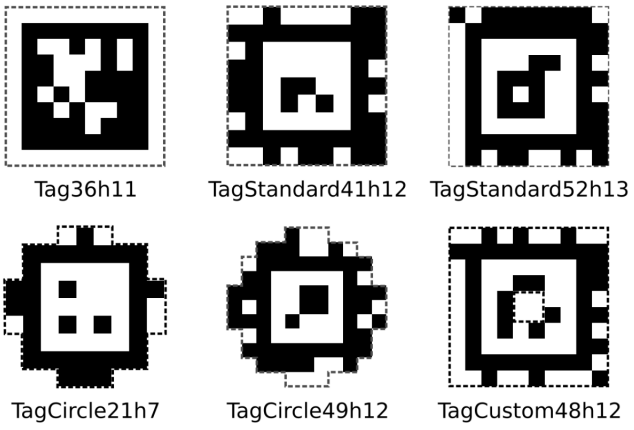

FIGURE 4.7: A sample of different AprilTag families

Each apriltag has a numeric ID associated that allow us to recognize the tag in the camera scene. One important factor of apriltags is that when they are printed and insterted into the real world we must know their real width and height so that we can then estimate the pixel to millimiters ratio and estimate real world distance. A tag must be measured like what we can see in Figure 4.8

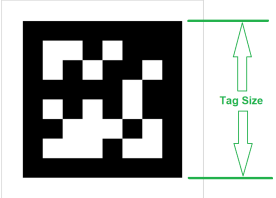

FIGURE 4.8: Figure demonstrating the tag size measurement

Beyond ID code, apriltag's SDK also provides **pose** data, namely position and orientation (rotation) from the **camera's point of view**. Imagine a laser beam pointing straight outward from the center of the camera lens. Its 3-dimensional path appears (to the camera) as a single point, indicated by the green star in Figure 4.9. You can see that the center of the AprilTag (yellow star) is offset from that "laser beam".

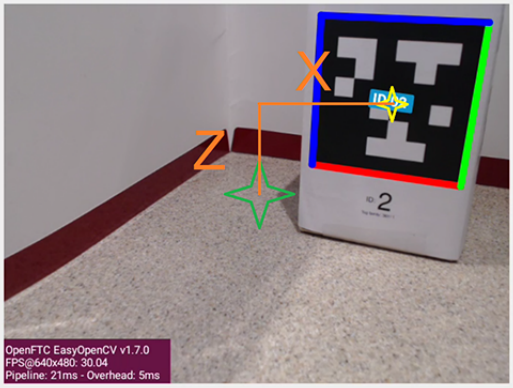

FIGURE 4.9: Apriltag pose estimation

That **translation offset** can break down into three traditional components (X, Y and Z distances), along axes at 90 degrees to each other:

- X distance (horizontal orange line) is from the center, to the right;
- Y distance (not shown) is from the lens center, outwards;
- Z distance (vertical orange line) is from the center, upwards.

The SDK provides these distances in the real world, not just reporting how many pixels on the screen. You can also see that the AprilTag's flat face is not parallel to the plane of the camera. That rotation offset can break down into three angles about the X, Y and Z axes.

In summary, the SDK evaluates the AprilTag image and performs "**pose estimation**", providing an estimated X, Y and Z **distance** between the tag and the camera, along with an estimated **angle** of rotation around those axes. A closer or larger AprilTag can yield a more accurate estimate of pose.

To provide good pose estimates, each RC phone camera or webcam **requires calibration data**, for a specific resolution. The SDK contains such data for a limited number of webcams and resolutions.

More about apriltags can be found at [Ols11].

[Ols11] Olson, "AprilTag: A robust and flexible visual fiducial system"

4.1.4 *Inverse Kinematics of the Robotic Arm*

The inverse kinematics (IK) problem for the Dobot Magician aims to calculate the joint angles required for a robotic arm to reach a specific position in three-dimensional space. The robotic arm consists of several links, and we must determine the angles that the arm's joints should take to position the end effector at a desired location (x, y, z). The primary goal is to compute four angles: θ_1, θ_2, θ_3, and θ_4, which correspond to the joint angles for the base, shoulder, elbow, and end effector rotation respectively.

For the Dobot Magician, the arm consists of two main segments:

- The **rear arm** with a length of 135 mm;
- The **forearm** with a length of 147 mm.

The inverse kinematics solution can be broken down into several steps:

- **Step 1**: Calculate the Base Rotation Angle (θ_1). The base rotation angle θ_1 is determined by the projection of the end effector's position onto the xy-plane. The angle is given by the arctangent of the ratio of the y-coordinate to the x-coordinate:

$$\theta_1 = \text{atan2}(y, x) \tag{4.3}$$

  This angle defines the rotation needed for the base joint to align the arm with the desired target along the xy-plane.

- **Step 2**: Calculate the Shoulder Angle (θ_2).

  Next, we calculate the shoulder angle θ_2, which involves two main components: the angle between the arm's direction and the z-axis, and the angle at the shoulder joint. The angle θ_2 is determined by the following:

  First, compute the angle β between the arm's direction and the vertical z-axis:

$$\beta = \text{atan2}\left(z, \sqrt{x^2 + y^2}\right) \tag{4.4}$$

  Then, compute the angle ψ, which is the angle between the two arm segments using the law of cosines:

$$\psi = \cos^{-1}\left(\frac{r^2 + d^2 - l^2}{2rd}\right) \tag{4.5}$$

  where $r = \sqrt{x^2 + y^2 + z^2}$, $d = 135$ (rear arm length), $l = 147$ (forearm length).

  Finally, the shoulder angle θ_2 is determined by the following equation:

$$\theta_2 = 90^\circ - (\beta + \psi) \quad \text{if} \quad \beta \geq 0 \tag{4.6}$$

  or

$$\theta_2 = 90^\circ - (\psi - |\beta|) \quad \text{if} \quad \beta < 0 \tag{4.7}$$

- **Step 3**: Calculate the Elbow Angle (θ_3). The elbow angle θ_3 is determined using the law of cosines, applied to the triangle formed by the rear arm, the forearm, and the line of sight from the base to the target. The angle θ_3 is given by:

$$\theta_3 = \cos^{-1}\left(\frac{r^2 - d^2 - l^2}{2dl}\right) \tag{4.8}$$

  This angle represents the relative motion between the rear arm and the forearm. The elbow angle θ_3 is adjusted by subtracting 90 degrees to ensure proper orientation:

$$\theta_3 = \theta_3 - 90^\circ \tag{4.9}$$

- **Step 4**: Calculate the End Effector Rotation Angle (θ_4). Finally, the end effector rotation angle θ_4 is simply the desired angle r (in degrees), as specified by the user:

$$\theta_4 = r$$

  This angle controls the orientation of the end effector relative to the arm.

These angles are computed in degrees and represent the necessary joint positions for the arm to reach the target position (x, y, z).

By solving the inverse kinematics equations, we can efficiently control the Dobot Magician's movements in 3D space, ensuring that it reaches the desired target position while maintaining the correct joint orientations.

4.2 POSE ESTIMATION W.R.T TO THE ROBOTIC'S ARM ORIGIN

Now, the mathematical problem switches to calculating the distances of the elements located on the table w.r.t the robotic's arm origin. The Dobot Magician is controlled by giving to the control stack the coordinates of the end effector, like what we can observe in Figure 4.10

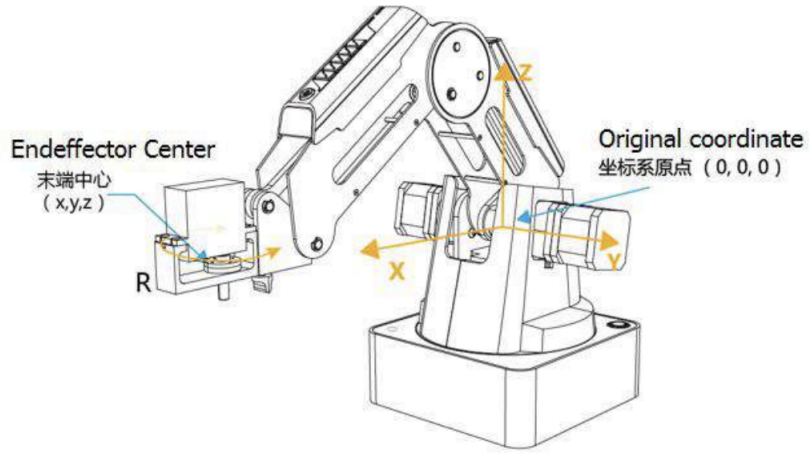

FIGURE 4.10: Coordinate system of the Dobot Magician

But up until this point all of the pose estimation that we made was w.r.t. the **camera coordinate system**. What we need to do now is to take the pose data that we got and translate it so that it is aligned with the robotic arm.

4.2.1 *Alignment of the coordinate systems*

First, two *control tags* were chosen. The first one is **Tag ID 0** and the second one is **Tag ID 4**. *Tag 0* is used as the center of our coordinate system on the table, meaning that we calculate all of the coordinates of other tags or objects by using *tag 0* as our $(0, 0)$ coordinates center. *Tag 4*, on the other end, is used to align the robotic arm. *Tag 4* is, in fact, always positioned at coordinates

$$x = 150,\ y = -150$$

w.r.t. to the robotic arm. This is later used to compute the error between the camera pose estimation and the actual coordinates needed by the arm to pick the 3d lego.

Tag 0 is then offsetted so that when the arm is aligned with *Tag 4*, then the center of *Tag 0* is perfectly aligned with the robotic arm's center. So we offset *Tag 0* by subtracting his relative position w.r.t. the arm

$$x'_0 = x_0 - 150,\ y'_0 = y_0 - 150$$

All of the computations are made only using the **center** of the tags so that the computed distances are **invariant by rotation**, meaning that no matter how the tag is positioned on the table the distance will always be the same.

4.2.2 *Conterting pixels measurements into millimiters and vice-versa*

Since the Dobot Magician accepts coordinates only in millimiters, while pose estimation techniques return us pixel distance from the camera, we need to convert all of the distances from pixels to millimiters. The pixel size of the tag is calculated like follows

$$\text{tag\_0\_width\_pixels} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.10}$$

and

$$\text{tag\_0\_height\_pixels} = \sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2} \tag{4.11}$$

To do so, we use *Tag 0* once again. Knowing its real world size of $30mm$ and knowing its pixel size thanks to our pose estimation techniques we can calculate the pixels to millimiters ratio as

$$r = tag\_0\_size\_mm / tag\_0\_side\_pixel\_dim \tag{4.12}$$

Then we can compute distances in millimiters by just multiplying distances in pixels by r like

$$d_{mm} = d_{px} * r.$$

Alternatively, we can invert the equation to go back to pixels from mm

$$d_{px} = \frac{d_{mm}}{r}.$$

4.2.3 *Computing the error of the camera*

Since we already know that *Tag 4's* position is always ($x = 150$, $y = -150$), we are able to compute the error between its actual position and the estimated position. The way we compute the error is to compute the **dilation factor** between the real coordinates and the estimated one.

Let (x'_4, y'_4) the coordinates calculated by the apriltag pose detector and $(x_4 = 150, y_4 = -150)$ the real and known coordinates. Since we know, by looking at the camera matrix calibration parameters, that there is a dilation between the estimation and the real world we compute our **dilation factor** as

$$k_x = x'_4/|x_4| \tag{4.13}$$

$$k_y = y'_4/|y_4| \tag{4.14}$$

Then, each time we make an estimation we can invert this equation and calculate the actual coordinates by dividing by $K = (k_x, k_y)$. For example, if the have the pose estimation of *Tag 3* Given that the coordinates are represented as $X_3 = (x_3, y_3)$ for the correct coordinates, and $X'_3 = (x'_3, y'_3)$ for the estimated coordinates, the vectorial form of the equation to obtain X_3 can be written as:

$$X_3 = \frac{X'_3}{K} \tag{4.15}$$

where:

$$X_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}, \quad X'_3 = \begin{pmatrix} x'_3 \\ y'_3 \end{pmatrix}, \quad K = \begin{pmatrix} k_x \\ k_y \end{pmatrix} \tag{4.16}$$

This implies:

$$\begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = \begin{pmatrix} \frac{x'_3}{k_x} \\ \frac{y'_3}{k_y} \end{pmatrix} \tag{4.17}$$

Thus, the division by K is applied element-wise to the components of X'_3, where k_x and k_y are the dilation factors in the x- and y-directions, respectively.

4.2.4 *Applying the calculations in real time*

The mathematics just discussed above are then applied each and every time we make pose estimation, both with YOLO and apriltags. Those are the steps:

- **Step 0**: We align the robot, align tags 0 and 4, compute the **pixels to mm ratio** r and compute the **dilation vector** K. This is a preliminary step;
- **Step 1**: we estimate the object's pose (another tag or the lego brick) w.r.t. the camera coordinates system;

- **Step 2**: We compute the distance along the x and y axis w.r.t the *Tag 0* origin. This is done by just subtracting *Tag 0*'s coords w.r.t to the camera from the detected object's coords w.r.t. to the camera. This effectively translates the origin and puts it at *Tag 0*'s center. Let's recall that *Tag 0*'s is already offsetted to be aligned with the robotic arm's center;
- **Step 3**: We convert the coordinates from pixels to millimiters using r;
- **Step 4**: We divide the new found coordinates by the **dilation vector** K in order to correct the error and to find the real world coordinates of the estimated object;
- **Step 5**: We communicate the coordinates to the robotic's arm so that they can later be used for path planning;
- **Step 6**: We reapeat it all again from Step 1, as long as the application is still running.

4.3 TRAJECTORY GENERATION

Trajectory generation is performed by using the estimated coordinates of the Lego block and the estimated coordinates of the landing AprilTag. There are three different variations of Lego blocks: red, white, and black. Each block is assigned a unique tag ID, for example:

- Red block = 3
- Black block = 2
- White block = 1

Once a Lego block is detected, the robot's movement is planned by first directing it to the block's location. The robot will then pick up the block and transport it to the designated AprilTag landing spot corresponding to the tag ID. The trajectory is computed to ensure smooth and efficient movements from the block's detection point to the assigned landing position.

5

Implementation Details

5.1 SYSTEM INTEGRATION

The system integration phase ensures seamless communication between hardware and software components, enabling coordinated operation. The primary components involved in the system include:

- **Robotic Arm:** The Dobot Magician robotic arm is responsible for executing pick-and-place tasks.
- **Vision System:** A Logitech webcam captures real-time images for object detection and pose estimation.
- **Processing Unit:** A Raspberry Pi 4B+ processes vision data and controls the robotic arm.
- **Software Frameworks:** ROS (Robot Operating System) on top of Ubuntu 22.04 LTS are utilized for motion planning and execution.

5.1.1 *Data Flow and Communication*

The integration follows a structured data flow:

1. The vision system captures images of the workspace and transmits them to the processing unit.
2. Image processing algorithms detect objects and estimate their positions.
3. The processed data is forwarded to the robotic control system.
4. MoveIT generates motion plans based on object locations.
5. The robotic arm executes the planned movements to manipulate objects.

Communication between components is facilitated using ROS topics and services, ensuring real-time data exchange and synchronization.

5.1.2 *Synchronization and Coordination*

To maintain operational efficiency, synchronization mechanisms are implemented:

- **ROS Topics:** Used for continuous data streaming between sensors and the robotic controller.
- **Action Servers:** Manage goal-based execution, ensuring coordinated movement.
- **Feedback Loops:** Provide real-time status updates for error correction and refinement.

The integration process guarantees that each component functions cohesively, ensuring reliability and accuracy in the system's operation.

5.2 SOFTWARE IMPLEMENTATION

5.2.1 *Robotic Arm Control*

The robotic arm is controlled using the https://github.com/jkaniuka/magician_ros2/tree/m and also as Citation control stack, which provides planning and execution capabilities. The control system integrates with ROS to send and receive commands.

dobot_bringup - System Initialization and Configuration

This package is responsible for launching and configuring the control stack using YAML files. It ensures that all necessary parameters and dependencies are loaded correctly before operation.

dobot_control_panel - RQT-based Control Interface

This package provides an RQT plugin that enables users to interact with the Dobot Magician. It allows control of both the robotic arm and an optional sliding rail, providing a graphical interface for command execution.

dobot_description - URDF and Mesh Descriptions

This package contains the Unified Robot Description Format (URDF) and mesh files necessary for the visualization and simulation of the Dobot Magician in ROS environments.

dobot_diagnostics - Alarm State Monitoring

A package designed for aggregation and analysis of alarm states in the Dobot Magician. It helps diagnose errors and system malfunctions.

dobot_driver - Low-Level Communication Interface

This package provides a low-level Python interface to communicate with the Dobot Magician via Wifi using UDP. It handles direct command execution and communication protocols.

This was originally written as a **serial USB** driver and was converted into a wifi driver by me. In order to achieve that the implementation of 3 classes was needed: **Interface**, **Message** and **Parsers**.

Let's see them one after the other.

- **Interface**:

  The 'Interface' class initializes a UDP socket to communicate with the Dobot Magician robotic arm. It inherits from 'threading.Thread' to enable multi-threading and uses a 'threading.Lock' to ensure thread safety. The constructor takes an IP address and an optional port (defaulting to 8899), creating a UDP socket for network communication.

```
    class Interface:
        def __init__(self, ip, port=8899):
            threading.Thread.__init__(self)
            self.lock = threading.Lock()
            self.ip = ip
            self.port = port

            self.sock = socket.socket(socket.AF_INET,
    socket.SOCK_DGRAM)

```

Listing 5.1: Initializing the class

  The following function attempts to send a message via UDP, converting the message into bytes and handling potential failures with retries. It sends the message, waits for a response with a timeout, and parses the received data. If no response is received within the timeout, it retries up to a maximum number of attempts with a delay between retries. If all attempts fail, it prints an error message; otherwise, it returns parsed response parameters if available.

```
    def send(self, msg, max_retries=3, retry_delay=1):
        #self.lock.acquire()

        request_package = msg.package()
        message = bytes([byte for byte in request_package])
        attempt = 0
        success = False

        while attempt < max_retries and not success:
            try:
                # Send the message
                self.sock.sendto(message, (self.ip, self.
    port))
                #print(f"Message sent to {self.ip}:{self.
    port}")

                # Receive the response
                self.sock.settimeout(5)  # Set a timeout of
     1 second for the response
                response, addr = self.sock.recvfrom(1024)
    # Buffer size of 1024 bytes
                #print(f"Received response: {response.hex()
    } from {addr}")

                # Parse the response
```

```
                parsed_response = self.parse_response(response)
                success = True
                #self.lock.release()
            except socket.timeout:
                print("No response received within the timeout period.")

             # If the message was not sent successfully, retry
            if not success:
                attempt += 1
                time.sleep(retry_delay)

        if not success:
            print("Failed to send message after multiple retries.")
        else:
            if parsed_response is None:
                pass
            else:
                return parsed_response.params

```

Listing 5.2: Send class

The next function is, instead, used to send a UDP message without waiting for a response. It converts the message into bytes and attempts to send it, retrying if necessary up to a specified number of attempts with a delay between retries. If all attempts fail, it prints an error message. This is used when you don't need a response from the Robot, for example if you want to start the Homing procedure

```
    def send_only(self, msg, max_retries=3, retry_delay=1):
        #self.lock.acquire()
        request_package = msg.package()
        message = bytes([byte for byte in request_package])

        attempt = 0
        success = False

        while attempt < max_retries and not success:
            try:
                # Send the message
                self.sock.sendto(message, (self.ip, self.port))
                #print(f"Message sent to {self.ip}:{self.port}")
            except socket.timeout:
                print("No response received within the timeout period.")
                #sock.close()

             # If the message was not sent successfully, retry
            if not success:
                attempt += 1
                time.sleep(retry_delay)

        if not success:
            print("Failed to send message after multiple retries.")

```

Listing 5.3: Send only class

This function is responsible for parsing UDP responses received from the server. It is used in the 'send' function after receiving a response to extract relevant parameters. If the response can be successfully parsed using 'Message.read_udp(response)', the

parsed message is returned; otherwise, an error message is printed, and ‘None‘ is returned. This ensures that the calling function can determine whether the response is valid and process it accordingly.

```
    def parse_response(self, response):
        parsed_message = Message.read_udp(response)
        if parsed_message:
            #print("Parsed message:", parsed_message.params
    )
            return parsed_message
        else:
            print("Failed to parse the message.")
            return None

```

Listing 5.4: Perse response class

Finally, we have functions that define the UDP headers and body of the message that we want to send based on the communication protocol described by the Dobot Magician documentation.

```
    def get_pose(self):
        #print("hello")
        request = Message([0xAA, 0xAA], 2, 10, False, False
    , [], direction='out')
        return self.send(request)

    def reset_pose(self, manual, rear_arm_angle,
    front_arm_angle):
        request = Message([0xAA, 0xAA], 2, 11, True, False,
     [manual, rear_arm_angle, front_arm_angle], direction='
    out')
        return self.send(request)

```

Listing 5.5: Perse response class

- **Message**:

  The ‘Message‘ class is designed to handle communication packets by structuring and parsing parameters based on the direction of the message. When an object is created, it initializes fields like the header, length, ID, read/write flag, queue status, and parameters. If the direction is set to "in," it stores the given parameters as raw data and then processes them. If set to "out," it takes structured parameters and converts them into raw data for transmission. This ensures the message format is correctly interpreted for both sending and receiving operations.

```
    class Message:
        def __init__(self, header, length, id, rw,
    is_queued, params, direction='in'):
            self.header = header
            self.length = length
            self.id = id
            self.rw = rw
            self.is_queued = is_queued
            self.raw_params = []
            self.params = []

            if direction == 'in':
                self.raw_params = params
                self.params = self.parse_params('in')
            elif direction == 'out':
                self.params = params
                self.raw_params = self.parse_params('out')

```

Listing 5.6: Initializing the message class

The `Message` class includes static methods to handle checksum validation and parsing of received messages.

The `calculate_checksum` method computes a checksum by summing the payload bytes, taking the modulo 256, and applying two's complement to ensure data integrity. The formula for the checksum is:

$$C = (256 - (\sum P \mod 256)) \mod 256$$

where P represents the payload bytes.

The `verify_checksum` method checks whether the sum of the payload and checksum results in zero modulo 256, confirming message correctness. The verification formula is:

$$(\sum P + C) \mod 256 = 0$$

The `parse` method extracts message components from a raw byte sequence, including the header, length, ID, control flags, read/write status, queue status, parameters, and checksum. It then verifies the checksum for specific message IDs (10, 20, or 13) before returning a valid `Message` object or `None` if verification fails.

```
    @staticmethod
    def calculate_checksum(payload):
        r = sum(payload) % 256
        # Calculate the two's complement
        check_byte = (256 - r) % 256
        return check_byte

    @staticmethod
    def verify_checksum(payload, checksum):
        a = sum(payload) % 256
        is_correct = True if (a + checksum) % 256 == 0 else
     False
        return is_correct

    @staticmethod
    def parse(message):
        bytes = list(message)

        header = bytes[0:2]
        length = bytes[2]
        id = bytes[3]
        control = bytes[4]
        rw = (control & 1) == 1
        is_queued = ((control & 2) >> 1) == 1
        params = bytes[5:-1]
        checksum = bytes[-1]

        if id==10 or id==20 or id==13:
            verified = Message.verify_checksum([id] + [
    control] + params, checksum)

            if verified:
                return Message(header, length, id, rw,
    is_queued, params)
            else:
                return None
        else:
            return None

```

Listing 5.7: Calculating checksums and parsing

The *read_udp* method processes incoming UDP data by validating its structure before parsing it into a Message object.

First, it checks if the message is long enough to contain at least a header and length byte. Then, it verifies that the header matches the expected value ($b'\xaa\xaa'$). After extracting the length byte, it ensures that the total message length matches the expected value. If any validation fails, the function prints an error message and returns None.

If the structure is valid, it extracts the payload and checksum, reconstructs the full message, and then passes it to the Message.parse method for further processing.

```
    @staticmethod
    def read_udp(data):
        # Ensure the message is long enough to contain a
    header and length
        if len(data) < 3:  # Header (2 bytes) + Length (1
    byte)
            print("Invalid length")
            return None

        # Parse the header
        header = data[:2]
        if header != b'\xaa\xaa':
            print("Invalid header")
            return None

        # Parse the length (3rd byte)
        length = data[2]
        if len(data) != 3 + length + 1:  # Header + Length
    + Payload + Checksum
            print("Invalid message length")
            return None

        # Extract the payload and checksum
        payload = data[3:3 + length]
        checksum = data[3 + length]

        # Reconstruct the full message
        full_message = header + bytes([length]) + payload +
     bytes([checksum])

        # Parse using existing logic
        return Message.parse(full_message)

```

Listing 5.8: Reading udp messages

The parse_params method is used within the Message constructor to process parameters based on the message direction. It selects the appropriate parser from a predefined set of parsers indexed by message ID.

For incoming messages, the method first checks whether parsers exist for the given message ID. If so, it selects the appropriate parser based on whether the message is read-only ($rw = 0$) or write ($rw = 1$) and whether it is queued ($is\_queued = 1$). If no suitable parser is found, it returns an empty list; otherwise, it processes the raw parameters.

For outgoing messages, the function ensures that parsing is performed only for write operations ($rw = 1$). If a parser is available, it processes the parameters before they are included in the message.

```
    @staticmethod
    def parse_params(self, direction):
        message_parsers = parsers[self.id]

        if direction == 'in':
            if message_parsers is None:
                return None

            parser = None
            if self.rw == 0 and self.is_queued == 0:
                parser = message_parsers[0]
            elif self.rw == 1 and self.is_queued == 0:
                parser = message_parsers[0]
            elif self.rw == 1 and self.is_queued == 1:
                parser = message_parsers[2]

            if parser is None:
                return []

            return parser(self.raw_params)
        elif direction == 'out':
            if message_parsers is None:
                return []

            parser = None
            if direction == 'out' and self.rw == 1:
                parser = message_parsers[3]

            if parser is None:
                return []

            return parser(self.params)

```

Listing 5.9: Setting the parsers

- **Parser**:

  This Python script contains a collection of parsers designed to handle communication specifications. The parsers are organized by message IDs, each corresponding to different device functions, such as real-time pose, alarms, homing functions, end effectors, and more.

  Each entry in the `parsers` dictionary contains four values:

  - **Response parser for getters:** (direction=in, read-write=0, isQueued=0)
  - **Response parser for setters:** (direction=in, read-write=1, isQueued=0)
  - **Response parser for setters with queuing:** (direction=in, read-write=1, isQueued=1)
  - **Request parser for setters:** (direction=out, read-write=1, isQueued=0/1)

  The parsers utilize the `struct` module to convert byte arrays into structured data types such as integers, floats, and strings. The approach is concise, scalable, and can easily accommodate future changes to the parsing logic. The following is an example of how parsers are defined

```
    # Real-time pose
    10: [lambda x: struct.unpack('<' + 'f' * 8, bytearray(x
    )), None, None, None],
```

```
    11: [None, None, None, lambda x: list(struct.pack('<Bff
    ', *x))],
    13: [lambda x: struct.unpack('<f', bytearray(x))[0],
    None, None, None]
```

Listing 5.10: Setting the parsers

dobot_end_effector - End-Effector Control Services

A set of service servers allowing control over different types of end effectors that can be attached to the Dobot Magician.

dobot_homing - Homing Procedure Execution

A package that executes the homing procedure for the Dobot Magician's stepper motors, ensuring the arm starts from a known position.

dobot_kinematics - Kinematics Computation

This package implements forward and inverse kinematics for the Dobot Magician, validating trajectory feasibility for motion planning.

dobot_motion - Motion Control Action Server

A package that contains an action server responsible for controlling the motion of the Dobot Magician, supporting joint-interpolated and linear motion.

dobot_msgs - ROS Message Definitions

Defines the ROS messages used by the control stack for structured communication between nodes.

dobot_state_updater - State Monitoring

Contains a node that regularly retrieves information about the state of the robot, including joint angles and TCP (Tool Center Point) position.

dobot_visualization_tools - Visualization Utilities

Provides visualization tools, such as trajectory and range markers, in RViz for monitoring the robot's motion and workspace.

5.2.2 *Vision Processing Pipeline*

The vision processing pipeline handles object detection and pose estimation. It receives image data from the camera, processes it, and extracts relevant information for object manipulation.

Camera Publisher

The '*CameraPublisher*' class is a ROS2 node designed to capture images from a camera and publish them on the '*/camera/image_raw*' topic. The class inherits from the 'Node' class and is initialized with the node name

‘*dobot_cv*‘. Upon initialization, a publisher is created to publish images on the ‘*/camera/image_raw*‘ topic with a queue size of 10. Additionally, a timer is set up to call the ‘*publish_frame*‘ method every 5 seconds, which results in a publishing rate of approximately 500 Hz. The class also creates a ‘*CvBridge*‘ object to convert OpenCV images to ROS2 Image messages. Furthermore, the ‘*cv2.VideoCapture*‘ function is used to initialize a video capture object, which interfaces with the default camera (index 0) to capture frames.

The class loads camera calibration parameters from two pickle files: ‘*cameraMatrix.*pkl‘ for the camera matrix and ‘*dist.pkl*‘ for the distortion coefficients. These calibration parameters are used to compute an optimal new camera matrix using the ‘*cv2.getOptimalNewCameraMatrix*‘ method. This matrix and the distortion coefficients are then employed in the ‘*cv2.initUndistortRectifyMap*‘ method to generate undistortion maps. These maps allow the class to correct any distortion present in the captured images.

The camera’s properties are configured with a default resolution of 640x480 for both width and height. The frame rate is configured using the MJPEG codec through the ‘*cv2.VideoWriter_fourcc*‘ method.

```
class CameraPublisher(Node):
    def __init__(self, width=640, height=480):
        super().__init__('dobot_cv')
        self.publisher = self.create_publisher(Image, '/camera/
    image_raw', 10)
        self.timer = self.create_timer(5, self.publish_frame)  #
     Publish at 500 Hz
        self.bridge = CvBridge()
        self.cap = cv2.VideoCapture(0, cv2.CAP_V4L2)

        # Load the camera matrix from pickle and set as
    parameter
        with open("/home/finucci/magician_ros2_control_system_ws
    /src/dobot_cv/camera_params/cameraMatrix.pkl", "rb") as f:
            self.camera_matrix = pickle.load(f)
        with open("/home/finucci/magician_ros2_control_system_ws
    /src/dobot_cv/camera_params/dist.pkl", "rb") as f:
            self.dist = pickle.load(f)

        newCameraMatrix, self.roi = cv2.
    getOptimalNewCameraMatrix(self.camera_matrix, self.dist, (
    width, height), 1, (width, height))

        self.map1, self.map2 = cv2.initUndistortRectifyMap(
            self.camera_matrix, self.dist, None, newCameraMatrix
    , (width, height), cv2.CV_32FC1
        )
        # Set camera properties (optional)
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)
        self.cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc
    ('M', 'J', 'P',
```

Listing 5.11: Initializing the camera publisher

The ‘*publish_frame*‘ method is responsible for capturing frames from the camera, applying undistortion, and publishing the processed frames to the ROS2 topic. Within the method, the ‘*cap.read()*‘ function is used to capture a frame from the camera. If the frame is successfully captured (i.e., ‘ret‘ is ‘True‘), the frame undergoes undistortion using the ‘*cv2.remap*‘ function, which applies the undistortion maps generated earlier. The frame is then converted from an OpenCV image to a ROS2

Image message using the ‘*CvBridge*‘ class, specifically the ‘*cv2_to_imgmsg*‘ method with the "bgr8" encoding. This message is then published on the ‘*/camera/image_raw*‘ topic using the publisher. After publishing the frame, a log message indicating the successful publishing is generated. If the frame capture fails (i.e., ‘ret‘ is ‘False‘), a warning log is generated to indicate the failure.

The ‘*__del__*‘ method is a destructor that ensures the release of the camera resource when the ‘*CameraPublisher*‘ object is destroyed. It checks whether the camera is still open using the ‘*cap.isOpened()*‘ method, and if it is, the ‘*cap.release()*‘ method is called to release the camera resource.

```
    def publish_frame(self):
        ret, frame = self.cap.read()
        if ret:
            frame = cv2.remap(frame, self.map1, self.map2,
    interpolation=cv2.INTER_LINEAR)
            # Convert OpenCV image to ROS2 Image message
            msg = self.bridge.cv2_to_imgmsg(frame, encoding="
    bgr8")
            self.publisher.publish(msg)

            # Log the frame publishing
            self.get_logger().info("Published a new frame")
        else:
            # Log the failure to capture frame
            self.get_logger().warn("Failed to capture frame from
     camera")

    def __del__(self):
        if self.cap.isOpened():
            self.cap.release()
```

Listing 5.12: Publishing the frames

Camera processor

The *CameraProcessor* class is designed to handle the detection and processing of AprilTags from a camera feed. Upon initialization, it sets up a publisher to send tag data on the */tag_detections* topic and creates a *CvBridge* object for converting between OpenCV images and ROS2 Image messages.

The detector is initialized with several parameters, including the family of tags to detect (defaulted to "*tag36h11*"), the tag size in meters (defaulted to 0.03), and an option to estimate the tag's pose (*estimate_tag_pose* set to *True* by default). Additionally, the class stores the camera matrix (loaded from a *cameraMatrix.pkl* file) and extracts important camera parameters like focal length (f_x, f_y) and the principal point (c_x, c_y), which are stored in *self.camera_params*.

The class also interacts with the Roboflow API to load a pre-trained model for object detection. The model is used to perform detection on the frames and includes parameters for confidence and overlap thresholds, ensuring the detections are reliable.

The *process_frame_callback* method is triggered upon receiving a new frame from the */camera/image_raw* topic. This method processes the frame to detect and handle AprilTags, providing information about the detected tags.

```
class CameraProcessor(Node):
```

```
    def __init__(self, families = "tag36h11", tag_size = 0.03,
    estimate_tag_pose = True):
        super().__init__('dobot_cv')
        self.publisher = self.create_publisher(String, '/
    tag_detections', 10)  # Publisher for tag data

        self.bridge = CvBridge()

        # Initializing the detector
        self.at_detector = Detector(
            families=families,
            nthreads=1,
            quad_decimate=1.0,
            quad_sigma=0.8,
            refine_edges=1,
            decode_sharpening=0.5,
            debug=0,
        )

        self.tag_size = tag_size  # Tag size in meters
        self.tag_size_mm = tag_size * 1000
        self.estimate_tag_pose = estimate_tag_pose

        with open("/home/finucci/magician_ros2_control_system_ws
    /src/dobot_cv/camera_params/cameraMatrix.pkl", "rb") as f:
            self.camera_matrix = pickle.load(f)

        # Extract fx, fy, cx, cy
        fx = self.camera_matrix[0, 0]
        fy = self.camera_matrix[1, 1]
        cx = self.camera_matrix[0, 2]
        cy = self.camera_matrix[1, 2]

        # Log and store camera parameters
        self.camera_params = (fx, fy, cx, cy)

        # Roboflow API key and project details
        rf = roboflow.Roboflow(api_key="ncH2Wguo0cJ3wV8FZ7sv")
        project = rf.workspace().project("brick3")
        self.model = project.version("2").model
        # Set confidence and overlap thresholds (optional)
        self.model.confidence = 50  # Minimum confidence
    threshold (50%)
        self.model.overlap = 25  # Maximum overlap between
    bounding boxes (25%)


        self.tag_0_center_offsetted = None
        self.tag_4_center = None

        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.process_frame_callback,
            10
        )
```

Listing 5.13: Initializing the camera processor

Apriltags detection

In this section, we describe the process of handling incoming frames, detecting tags, and calculating relative positions.

The **process_frame_callback** method starts by logging a message indicating the processing of a frame. The frame, received as a ROS2 *Image* message, is then converted to an OpenCV image using the *bridge.imgmsg_to_cv2* method, with the encoding "*bgr8*". In case of an exception, an error message is logged.

The image is then converted to grayscale using *cv2.cvtColor*, and the *at_detector.detect* method is used to detect tags in the image. The detection process estimates the pose of the tags and uses the camera parameters and tag size to refine the detection.

```
    def process_frame_callback(self, msg):
        self.get_logger().info("Processing frame...")

        try:
            # Convert ROS2 Image message to OpenCV image
            frame = self.bridge.imgmsg_to_cv2(msg, "bgr8")

            # Log the frame received
            self.get_logger().info("Received a new frame")
        except Exception as e:
            self.get_logger().error(f"Failed to process the
    image: {str(e)}")

        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Detect tags
        detections = self.at_detector.detect(
            img=gray_frame,
            estimate_tag_pose=self.estimate_tag_pose,
            camera_params=self.camera_params,
            tag_size=self.tag_size,
        )
```

Listing 5.14: Setting up the frames

The method then looks for a detection with a *tag_id* of 0 to use as the origin. If such a tag is found, its center is stored in *tag_0_center*, and the offset is calculated by subtracting 150 pixels from both the x and y coordinates of the center. The pixel dimensions of the tag are also computed by calculating the horizontal and vertical distances between its corners, using *np.linalg.norm*. The average pixel size is calculated assuming the tag is square-shaped, and the ratio of pixels to millimeters is computed using the tag size in millimeters and the average pixel size.

```
         # Find the detection with tag_id == 0 to use as the
    origin
        tag_0_center = None

        for detection in detections:
            if detection.tag_id == 0:
                tag_0_center = tuple(map(float, detection.center
    ))
                self.tag_0_center_offsetted = (
                    tag_0_center[0] - 150,
                    tag_0_center[1] - 150
                )
                # Calculate the pixel size of the tag (in pixels
    )
                tag_0_corners = np.array(detection.corners,
    dtype=np.float64)
                tag_0_width_pixels = np.linalg.norm(
    tag_0_corners[0] - tag_0_corners[1])  # Horizontal distance
    (in pixels)
                tag_0_height_pixels = np.linalg.norm(
    tag_0_corners[1] - tag_0_corners[2])  # Vertical distance (
    in pixels)

                # Average pixel size (assuming the tag is square
    -shaped)
                tag_0_side_pixel_dim = (tag_0_width_pixels +
    tag_0_height_pixels) / 2.0

                # Calculate the pixels to mm ratio
                self.pixels_to_mm_ratio = self.tag_size_mm /
    tag_0_side_pixel_dim
```

```
21                 break
```

Listing 5.15: Estimating tag 0

The method then searches for a detection with *tag_id* 4. If the reference tag (tag 0) has been found and the offset is defined, the center of tag 4 is recorded. The relative position of tag 4 to the origin (tag 0) is calculated by subtracting the offset and converting the result to millimeters using the *pixels_to_mm_ratio*. The scaling factors *factor_x_mm* and *factor_y_mm* are then determined based on the relative position, assuming that the tags are aligned along the x and y axes.

Finally, the *tag_data_list* is initialized to store the tag data.

```
1         for detection in detections:
2             if detection.tag_id == 4 and self.
    tag_0_center_offsetted is not None:
3                 self.tag_4_center = tuple(map(float, detection.
    center))
4
5                 relative_position_mm_off = (
6                     -(self.tag_4_center[1] - self.
    tag_0_center_offsetted[1]) * self.pixels_to_mm_ratio,
7                     -(self.tag_4_center[0] - self.
    tag_0_center_offsetted[0]) * self.pixels_to_mm_ratio
8                 )
9
10                self.factor_x_mm = abs(relative_position_mm_off
    [0]) / 150
11                self.factor_y_mm = abs(relative_position_mm_off
    [1]) / 150
12
13                break
14
15        tag_data_list = []
```

Listing 5.16: Using tag 4 for error detection

The following code snippet calculates the relative positions of detected tags, excluding the reference tag (tag 0), based on an inverted axis system.

If *self.tag_0_center_offsetted* is not *None*, the code proceeds to compute the relative coordinates of other tags by iterating over the list of detections. Each detection that is not *tag_id = 0* is processed by first extracting the tag's ID and its center coordinates.

The relative position of the tag in millimeters is calculated by subtracting the offset of the reference tag (tag 0) from the current tag's center. The results are multiplied by the *pixels_to_mm_ratio* to convert from pixels to millimeters. The axes are inverted by negating the x and y values.

Next, the relative position in millimeters is further adjusted by dividing by *factor_x_mm* and *factor_y_mm*, respectively, to refine the scaling. The coordinates are then rounded to two decimal places.

The tag's data, including its ID and the calculated x and y coordinates, is appended to *tag_data_list*, which is a list storing the processed information for each tag.

```
1         if self.tag_0_center_offsetted is not None:
2             # Compute the relative coordinates for other tags
    based on the inverted axis system
3             for detection in detections:
4                 if detection.tag_id != 0:
5                     tag_id = detection.tag_id
```

```
                    center = tuple(map(float, detection.center))
      # The center of the tag
                    relative_position_mm = (
                        -(center[1] - self.
    tag_0_center_offsetted[1]) * self.pixels_to_mm_ratio,
                        -(center[0] - self.
    tag_0_center_offsetted[0]) * self.pixels_to_mm_ratio
                    )

                    relative_position_mm_off = (
                        relative_position_mm[0] / self.
    factor_x_mm,
                        relative_position_mm[1] / self.
    factor_y_mm
                    )

                    # Format the coordinates to 2 decimal places
                    x = relative_position_mm_off[0]
                    y = relative_position_mm_off[1]

                    # Add tag data to the list
                    tag_data_list.append({
                        "tag_id": str(tag_id),
                        "x": round(x, 2),
                        "y": round(y, 2)
                    })
```

Listing 5.17: Calculating tags coordinates

Lego bricks detection

The **process_lego_brick** method processes a given frame to detect LEGO bricks using a pre-trained model from Roboflow. The method first saves the incoming frame temporarily as an image file, "*temp_frame.jpg*", and then performs inference on it using the model's *predict* method.

For each predicted object in the returned *predictions*, the method extracts the bounding box's coordinates (x, y, width, height), class name, and confidence level. These values are used to calculate the bounding box coordinates (x_1, y_1 for the top-left corner and x_2, y_2 for the bottom-right corner), and the center of the bounding box is computed as the average of the x and y coordinates.

The method returns a list of detections, where each detection contains the tag ID (as *class_name*), the center of the bounding box, and the corner coordinates of the bounding box (x_1, y_1, x_2, y_2).

```
def process_lego_brick(self, frame):
        detections = []
        # Save the frame temporarily as a file for Roboflow
    inference
        cv2.imwrite("temp_frame.jpg", frame)

        # Perform inference on the saved frame
        prediction = self.model.predict("temp_frame.jpg")

        # Parse predictions and draw bounding boxes
        for prediction_item in prediction.json()["predictions"]:
            x, y, width, height = (
                prediction_item["x"],
                prediction_item["y"],
                prediction_item["width"],
                prediction_item["height"],
            )
            class_name = prediction_item["class"]
            confidence = prediction_item["confidence"]

            # Calculate bounding box coordinates
            x1 = int(x - width / 2)
```

```
            y1 = int(y - height / 2)
            x2 = int(x + width / 2)
            y2 = int(y + height / 2)

            center = [(x1 + x2)/2, (y1 + y2)/2]

            # Add the corresponding detection to detections2
            detections.append({
                "tag_id": class_name,
                "center": center,
                "corners": [x1, x2, y1, y2]
            })

        return detections
```

Listing 5.18: Detecting lego bricks

Here we have the method of calculating the relative coordinates of detected LEGO tags based on the camera's reference point, represented by *self.tag_0_center_offsetted*. If this reference point is defined, the method calculates the relative position of each detected tag by first extracting the *tag_id* and *center* from the list of detections.

The axes are swapped and inverted, and the difference between the tag's center coordinates and the reference point is calculated. This difference is then multiplied by *self.pixels_to_mm_ratio* to convert the position from pixels to millimeters. The relative position is further adjusted by dividing by *self.factor_x_mm* and *self.factor_y_mm* to normalize the coordinates for the specific scaling factors.

The final relative position is then rounded to two decimal places and added to the *tag_data_list*, where the coordinates (x, y) for each tag are stored, along with its *tag_id*.

```
lego_detections = self.process_lego_brick(frame)

    if self.tag_0_center_offsetted is not None:
            # Compute the relative coordinates for other tags based on the inverted axis system
            for detection in lego_detections:
                tag_id = detection['tag_id']
                center = tuple(map(float, detection['center'])) # The center of the tag

                # Swap and invert axes
                relative_position_mm = (
                    -(center[1] - self.tag_0_center_offsetted[1]) * self.pixels_to_mm_ratio,
                    -(center[0] - self.tag_0_center_offsetted[0]) * self.pixels_to_mm_ratio
                )

                relative_position_mm_off = (
                    relative_position_mm[0] / self.factor_x_mm,
                    relative_position_mm[1] / self.factor_y_mm
                )

                # Format the coordinates to 2 decimal places
                x = relative_position_mm_off[0]
                y = relative_position_mm_off[1]

                # Add tag data to the list
                tag_data_list.append({
                    "tag_id": str(tag_id),
                    "x": round(x, 2),
                    "y": round(y, 2)
                })
```

Listing 5.19: Computing the distance between legos and tag 0

5.2.3 *Pick and place action*

The *PickAndPlace* class is designed to handle the pick-and-place functionality for a robotic arm, utilizing multiple ROS2 components. It integrates a semaphore subscriber, tag detection subscription, action client for point-to-point (PTP) movement, and a service client for controlling the gripper.

The *semaphore_subscriber* listens for messages on the */semaphore* topic. Upon receiving a message, it triggers the *semaphore_callback* method, which can be used for synchronizing the pick-and-place operation based on semaphore signals.

The *tag_subscription* listens to the */tag_detections* topic, receiving information about the detected tags. This data helps the system determine the locations where objects should be placed.

The *msg_buffer* stores incoming messages for later processing. This buffer is used to ensure the system can handle messages asynchronously and process them in the correct order.

The action client communicates with a point-to-point action server to control the robotic arm's movement. The arm moves from one point to another based on the provided coordinates or positions, facilitating the pick-and-place operation.

The gripper service client sends requests to control the gripper's actions (e.g., open or close). It waits for the *dobot_gripper_service* to be available before making any service calls, ensuring the gripper is ready to perform tasks.

The *tasks_list* holds the tasks that need to be completed in the pick-and-place process. The *goal_num* variable tracks the current goal number, providing a means of managing task execution sequences.

```
class PickAndPlace(Node):

    def __init__(self):
        super().__init__('dobot_cv')

        # Semaphore subscriber
        self.green_light_detected = False
        self.semaphore_subscriber = self.create_subscription(
            String,
            '/semaphore',
            self.semaphore_callback,
            10,
            callback_group=ReentrantCallbackGroup()
        )

        # Tag detections subscription
        self.tag_subscription = self.create_subscription(
            String,  # Replace with your actual message type
            '/tag_detections',
            self.tag_callback,
            10  # QoS depth
        )

        # Store messages in a buffer
        self.msg_buffer = []

        # Action client for PointToPoint
        self._action_client = ActionClient(self, PointToPoint, '
    PTP_action', callback_group=ReentrantCallbackGroup())

        # Gripper service client
        self.cli = self.create_client(
            srv_type=GripperControl,
```

```
            srv_name='dobot_gripper_service',
            callback_group=ReentrantCallbackGroup()
        )
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Gripper service not
    available, waiting again...')
        self.req = GripperControl.Request()

        self.tasks_list = []
        self.goal_num = 0
```

The *tag_callback* method processes incoming messages related to tag detections. It is designed to store these messages in a buffer and ensures the buffer maintains a maximum size of 10 messages.

When a new message is received, it is appended to the *msg_buffer*, and a log entry is made indicating that the message has been added. If the buffer exceeds 10 messages, the oldest message is removed to keep the buffer size consistent. This approach ensures that only the most recent detections are retained.

```
def tag_callback(self, msg):
    # Add received message to the buffer
    self.msg_buffer.append(msg)
    self.get_logger().info('Adding message to the detections
    buffer...')
    if len(self.msg_buffer) > 10:  # Keep the buffer size to the
     last 10 messages
        self.get_logger().info('Removing message from the
    detections buffer...')
        self.msg_buffer.pop(0)
```

5.3 SEMAPHORE CALLBACK AND TASK EXECUTION

The *semaphore_callback* method listens for incoming semaphore messages and triggers the execution of the pick-and-place tasks based on the semaphore state. When a "green" light is detected, it calls the *process_tag_detections* method to calculate the means of tag detections and proceeds with the task execution. If a "red" light is detected, it waits for the green light to proceed.

```
def semaphore_callback(self, msg):
    if msg.data.lower() == "green":
        self.get_logger().info("Green light detected! Starting
    pick-and-place...")

        if self.goal_num > 0:
            self.goal_num = 0
        self.green_light_detected = True

        self.process_tag_detections()
        self.execute()

    elif msg.data.lower() == "red":
        self.get_logger().info("Red light detected. Waiting for
    green light...")
```

The *process_tag_detections* method calculates the mean positions of detected tags from the message buffer. It processes data from different Lego bricks (red, black, white) and calculates the means of the x and y coordinates for each tag. This data is then used to create a list of tasks that the robot will execute.

```
def process_tag_detections(self):
    # Calculate means from the message buffer
    mean_values = self.calculate_means(self.msg_buffer)
```

```
    self.log_means(mean_values)

    # Extract mean coordinates for each tag
    mean_x_red_brick = mean_values.get('red-lego', {}).get('mean_x', None)
    mean_y_red_brick = mean_values.get('red-lego', {}).get('mean_y', None)
    # Similar for other bricks and tags...

    self.tasks_list = [
        ["move", [mean_x_red_brick, mean_y_red_brick, 10.0, 65.0], 1, 1.0, 1.0],
        ["gripper", "open", False],
        ["move", [mean_x_red_brick, mean_y_red_brick, -8.0, 65.0], 1, 0.1, 0.1],
        # More tasks for other bricks...
    ]
```

The *calculate_means* method iterates over the message buffer, deserializes the received data, and computes the mean coordinates for each tag by aggregating the x and y values. The computed mean values are returned as a dictionary.

```
def calculate_means(self, msgs):
    tag_data = {}
    for msg in msgs:
        try:
            msg_data_str = msg.data.replace("'", '"')
            msg_data = json.loads(msg_data_str)

            for tag in msg_data:
                tag_id = tag['tag_id']
                x = tag['x']
                y = tag['y']

                if tag_id not in tag_data:
                    tag_data[tag_id] = {'sum_x': 0, 'sum_y': 0, 'count': 0}

                tag_data[tag_id]['sum_x'] += x
                tag_data[tag_id]['sum_y'] += y
                tag_data[tag_id]['count'] += 1
        except Exception as e:
            self.get_logger().warning(f"Error processing message: {e}")

    mean_values = {}
    for tag_id, data in tag_data.items():
        mean_values[tag_id] = {
            'mean_x': data['sum_x'] / data['count'],
            'mean_y': data['sum_y'] / data['count'],
        }

    return mean_values
```

The *log_means* method logs the calculated mean values for each tag.

```
def log_means(self, mean_values):
    for tag_id, means in mean_values.items():
        self.get_logger().info(f"Tag ID: {tag_id}, Mean X: {means['mean_x']:.2f}, Mean Y: {means['mean_y']:.2f}")
```

The *execute* method is responsible for executing tasks based on the green light signal. It iterates through the tasks in the *tasks_list* and executes them. If the task type is "gripper," it calls the gripper service; if the task type is "move," it sends the movement goal to the robot.

```
def execute(self):
    self.get_logger().info(f"Green Light Detected: {self.green_light_detected}")

    if not self.green_light_detected:
```

```
        self.get_logger().info("Waiting for green light to start execution...")
        return

    if self.goal_num >= len(self.tasks_list):
        self.get_logger().info("All tasks completed. Shutting down.")
        self.green_light_detected = False
        rclpy.shutdown()
        sys.exit()

    self.get_logger().info(f'*** TASK NUM ***: {self.goal_num}/{len(self.tasks_list)-1}')
    task = self.tasks_list[self.goal_num]
    self.get_logger().info(f"Executing task: {task}")

    if task[0] == "gripper":
        self.send_request(*task[1:])
        self.goal_num += 1
        self.timer = self.create_timer(0.1, self.timer_callback, callback_group=ReentrantCallbackGroup())

    elif task[0] == "move":
        self.send_goal(*task[1:])
        self.goal_num += 1

    else:
        self.get_logger().error(f"Unknown task type: {task[0]}")
        self.goal_num += 1  # Prevent getting stuck
```

Part III

RESULTS & CONCLUSIONS

6

Experiments and Results

6.1 EXPERIMENTAL SETUP

The one in Figure 6.1 is the experimental setup with the webcam attached to the Raspberry, the robot, the legos and the apriltags

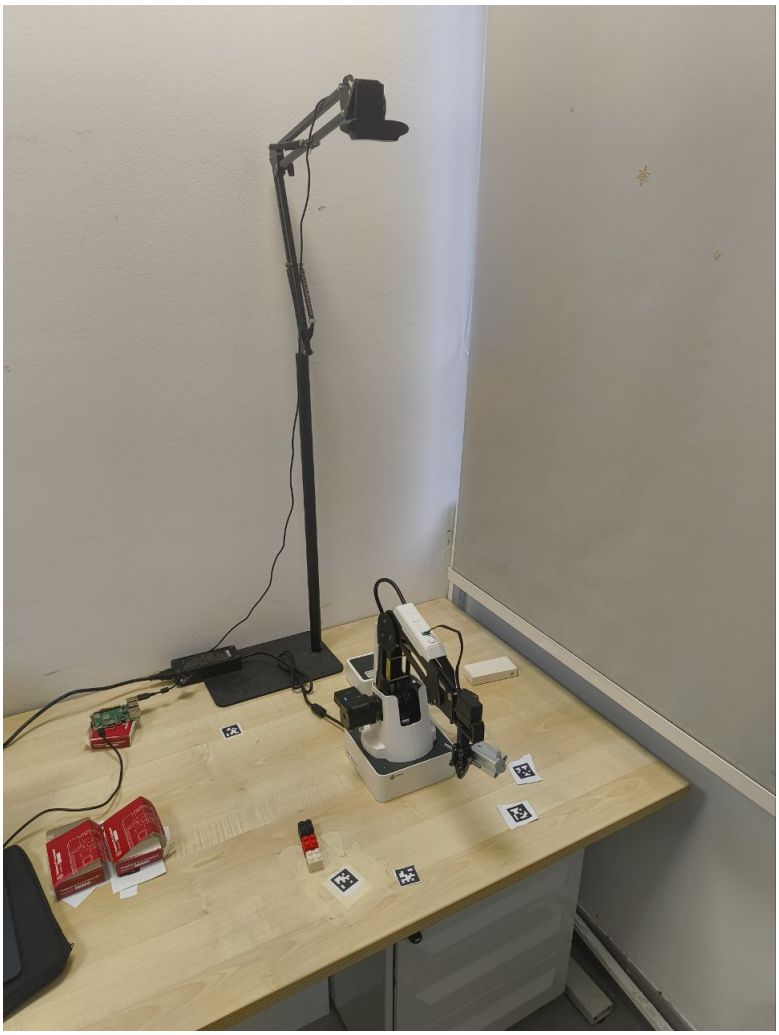

FIGURE 6.1: experimental setup

The test starts with the lego blocks in the initial position on the right side of the robotic arm as we can see in Figure 6.2 At the end the lego blocks

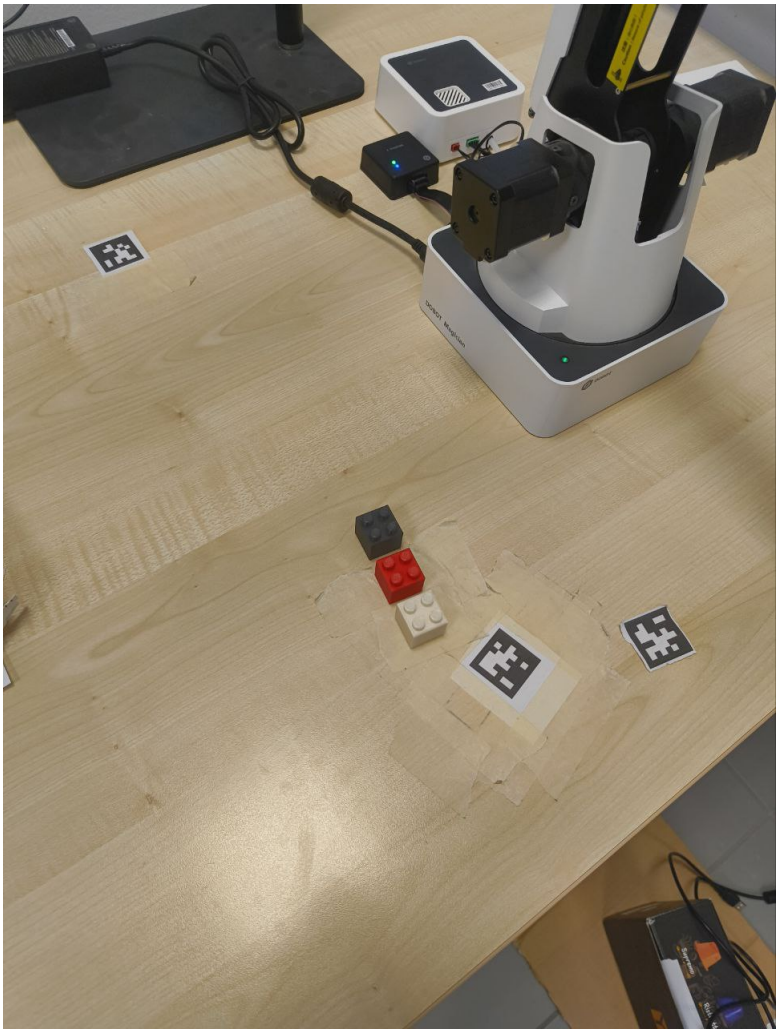

FIGURE 6.2: Initial position

are then placed on the corresponding apriltags as we can see in Figure 6.3

FIGURE 6.3: Final position

6.2.1 *Accuracy of Object Detection*

To train the Deep Learning model we captured *640x480* photos directly from the logitech webcam and the i annotated it using roboflow as we can see in Figure 6.4

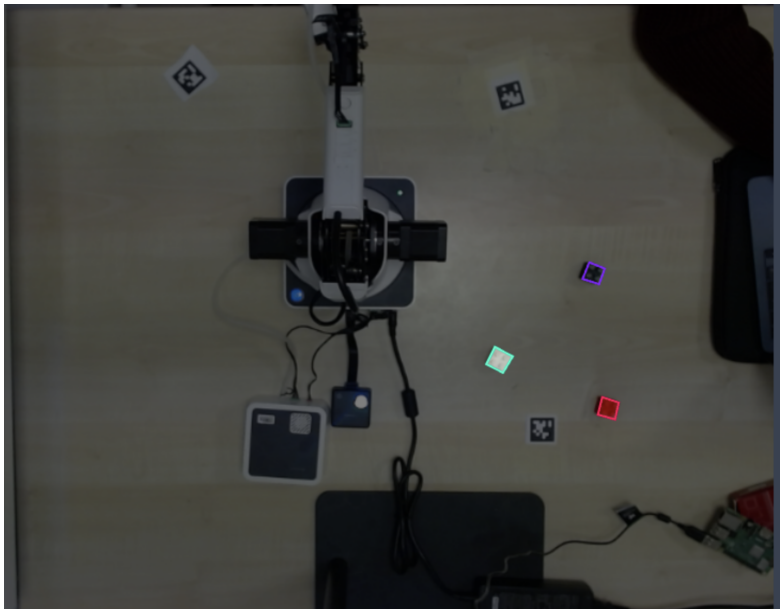

FIGURE 6.4: Annotated photos

The accuracy of the object detection is very high

FIGURE 6.5: Precision percentages

And also the training graphs look promising too

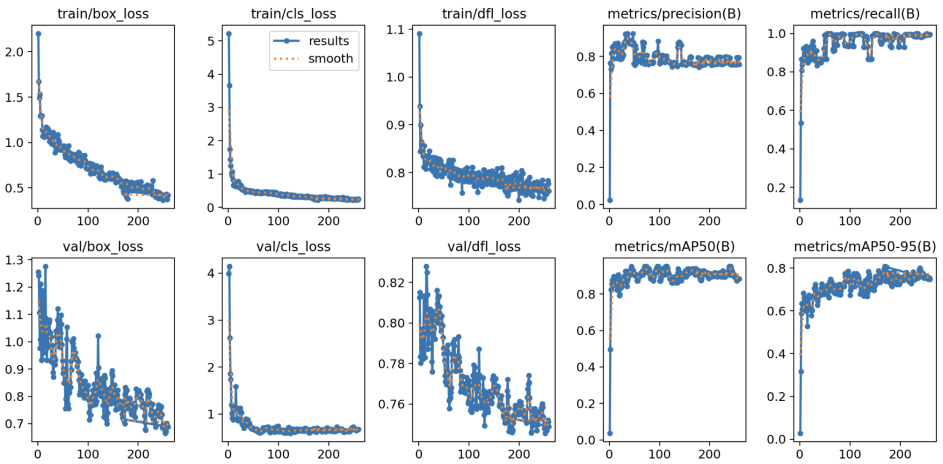

FIGURE 6.6: Training graphs

6.2.2 *Performance of Motion Planning*

The motion planning is static and just consists in the robot going from one point to the next one, so of course the performance of it is very high.

6.2.3 *Time Efficiency in Task Execution*

When using WiFi, the time efficiency of task execution tends to decrease significantly due to slower data transfer speeds compared to wired connections. This can result in delays in communication, especially when large amounts of data need to be transferred or real-time processing is required. The limited bandwidth and potential interference from other devices on the network contribute to these slowdowns, impacting the overall system performance.

6.2.4 *Error Analysis and Improvements*

The size of the bricks poses a significant challenge. If the robotic arm isn't perfectly calibrated, it may struggle to pick up the bricks due to their large size. This inaccuracy can arise from several factors, including misalignment, imperfect grasping mechanisms, or slight deviations in the arm's movement. To improve the system's reliability, adjustments to the

arm's precision and calibration, as well as a reconsideration of the brick size or grasping strategy, may be necessary.

6.3 DISCUSSION OF RESULTS

The results obtained from the robotic system are promising. The system is capable of picking up the Lego bricks and placing them on the corresponding tags with reasonable accuracy. However, there are areas for improvement, particularly regarding the precision of the arm's movements. Despite being functional, the robotic arm's precision could be enhanced to ensure more reliable placement of the bricks.

As discussed in the previous sections, one key challenge lies in the size of the Lego bricks. The larger size of the bricks makes it more difficult for the robotic arm to handle them with perfect accuracy. Even small misalignments or imperfections in the arm's movement can lead to failure in properly placing the bricks. This issue highlights the importance of improving the arm's calibration and overall precision.

In terms of time efficiency, the process is somewhat slow, particularly when relying on Wi-Fi for communication. The speed of task execution can be optimized by reducing latency in the system, potentially through alternative communication protocols or optimizing the control loop for quicker response times. Additionally, further refinement in the software and hardware components could contribute to faster execution and better overall performance.

7

Conclusion and Future Work

7.1 CONCLUSION

The project focused on simulating the operation of a robotic arm tasked with picking up a container from the dock and loading it onto a ship. Throughout the project, computer vision and mathematics played crucial roles in ensuring accurate detection and movement planning. The robotic arm was able to detect and track containers using visual inputs, and mathematical algorithms were employed to calculate the necessary trajectories and movements, providing a detailed simulation of a real-world task.

The integration of computer vision algorithms allowed the robotic system to recognize key objects, such as containers and docking locations, enabling it to perform operations with a high degree of automation and precision. The mathematical models provided the framework for optimizing the arm's movements, ensuring efficiency and accuracy in the task execution. This project demonstrated how combining computer vision with mathematical principles, such as inverse kinematics and trajectory planning, could simulate complex robotic operations and serve as the basis for future advancements in automation and robotics.

7.2 LIMITATIONS

Despite the progress made, there were several limitations within the current simulation. One of the primary challenges was the accuracy of the detection system. While the system could detect the containers and the docking locations, the detection was not always spot-on. The algorithm faced difficulties in handling variations in lighting, angles, and object occlusions, which impacted the precision of the recognition and, in turn, the arm's movement planning.

Furthermore, the current simulation was based on a single robotic arm, and while it was capable of completing the task, there were situations where the task could be expedited with the use of additional robotic arms. The single robot setup made the task more time-consuming, as it had to handle the entire process of picking up and placing the container on its own.

7.3 POTENTIAL ENHANCEMENTS

There are several potential enhancements that could improve the system's performance and extend its capabilities:

- **Improved Detection Algorithms:** The detection system could be enhanced by integrating more advanced computer vision techniques, such as deep learning-based object recognition or stereo vision, which would allow for more accurate and robust container detection even under challenging conditions.
- **Multiple Robot Coordination:** A second robot could be integrated into the system to work alongside the first, enabling parallel execution of tasks. This would allow for faster container handling and the ability to coordinate movements between the two robots, improving overall efficiency and reducing completion times. Communication protocols could be introduced to facilitate real-time coordination and decision-making between the robots.
- **Dynamic Trajectory Planning:** Implementing real-time dynamic trajectory planning could help the robot adjust its movements on the fly based on unexpected obstacles or changes in the environment. This would allow the system to better adapt to unforeseen situations and ensure smoother operations.
- **Advanced Feedback Systems:** Adding feedback systems such as force sensors and cameras in the robotic arm could enable it to fine-tune its grip on the container, ensuring a safer and more precise loading process.

With these enhancements, the system could evolve into a more robust, adaptable, and efficient robotic solution for container handling in real-world applications. The ability to coordinate multiple robots and improve the accuracy of detection and movement would bring the project closer to practical use in industries such as shipping and logistics.

Bibliography

[Joc+22] Glenn Jocher et al. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. Version v7.0. Nov. 2022. DOI: 10.5281/zenodo.7347926. URL: https://doi.org/10.5281/zenodo.7347926 (cit. on p. 17).

[Kan23] Jan Kaniuka. "System sterowania robota manipulacyjnego Dobot Magician na bazie frameworka ROS2". Bachelor's thesis. WEiTI, 2023. URL: https://gitlab-stud.elka.pw.edu.pl/robotyka/rpmpg_pubs/-/raw/main/student-theses/jkaniuka-bsc-23-twiki.pdf (cit. on p. 12).

[Ols11] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407 (cit. on p. 19).