CentraleSupélec

Pôle Projet P19, SG8 2023

Development of a small-scale ball-tossing robotic arm

Students:
Daniel Stulberg Huf
Emilio Hajjar
Luan Rocha do Amaral
Marina Daumas Carneiro
Tomas Gonzalez Villarroel
Yuanhao Dong

Advisor:
Maria MAKAROV

June 10, 2023

Contents

I Introduction

Throwing an object can be a useful strategy to exploit a robot's abilities and overcome its limitations. Assuming that the robot needs to place objects in specific locations and the objects are not fragile, throwing them can be an effective way to achieve the objective, even if the target is farther than the robot's maximum reach. This approach allows for faster task completion and reduces the complexity of the robot system.

Nevertheless, throwing presents inherent challenges stemming from intricate dynamics and various factors such as the initial conditions before the throw, the object's shape and mass, the unpredictable variable of wind, the specific point of the object gripped by the robot's gripper, and the influence of air resistance. These factors significantly impact the execution of successful throws.

This document represents a continuation of the previous year's work involving the same robotic arm (Openmanipulator-X Robotis RM-X52-TNM). The report is structured as follows: Section I provides an overview of the project's objective and the division of tasks within the group. Section II elucidates the methodology employed for computer vision techniques. Section III delves into the mathematical models utilized for controlling the robot's movements. Sections IV and V provide comprehensive descriptions of the simulation environment and the system employed for validating the computer vision and control models. Finally, section VI concludes the project by highlighting its key accomplishments and suggesting areas for future exploration.

Figure 1: Robot arm Openmanipulator-X Robotis RM-X52-TNM

I.1 Objectives

The goal of this project is to develop a system that enables the robot (fig. 1) to successfully launch a table tennis ball into a cup. In order to allow the robotic arm to detect the target cup in 3D space, it will be connected to a Zed2 stereo camera for depth sensing.

However, the laboratory with all the equipment is located on the Paris-Saclay campus and most of the members of this project study are settled in Rennes. As a consequence, the project was done remotely and the group focused on simulating the system and creating the mathematical model of the robot.

A new problem caused by the remote aspect of the project is the inability to access a stereo camera for depth sensing. Therefore, a new approach that uses laptop webcams is proposed. This is useful to make low-cost alternatives to the project, and comparing both remote and *in loco* performances.

II Computer vision

In this section, it will be presented the computer vision techniques employed for cup and robot identification, along with the mathematical model utilized to estimate the relative distance between these objects. This process constitutes the initial stage of the pipeline for ball targeting, as it is imperative to track the relative positions of the objects within the controlled environment, analyzing both cases when the input is in picture format, and also in video format. Subsequently, the obtained relative distances and angles are calculated, which will later be communicated to the control subsystem responsible for executing the tossing action.

II.1 Camera calibration

Prior to stepping into object identification using the camera and leveraging tools such as OpenCV, it is imperative to conduct camera calibration as a preliminary step. Camera calibration is essential for acquiring a comprehensive understanding of the camera's characteristics and optimizing its performance as a visual sensor.

The procedure involved in determining the parameters of a camera is known as camera calibration. It encompasses the acquisition of essential information, such as camera coefficients and parameters, to establish a precise correlation between a 3D point in the real world and its corresponding 2D projection (pixel) within the image captured by the calibrated camera. Camera calibration typically involves the retrieval of two distinct types of parameters.

1. **Internal parameters of the camera/lens system**. E.g. focal length, optical center, and radial distortion coefficients of the lens.
2. **External parameters** : This refers to the orientation (rotation and translation) of the camera with respect to some world coordinate system.

The goal of the calibration process is to find the 3×3 matrix K, the 3×3 rotation matrix R, and the 3×1 translation vector t using a set of known 3D points (X_w, Y_w, Z_w) and their corresponding image coordinates (u, v). The best way to perform calibration is to capture several images of an object or pattern of known dimensions from different view points. The checkerboard pattern is a very popular and efficient approach to calibration. The corners of squares on the checkerboard are ideal for localizing them because they have sharp gradients in two directions. In addition, these corners are also related by the fact that they are at the intersection of checkerboard lines. All these facts are used to robustly locate the corners of the squares in a checkerboard pattern.

Below, it is possible to observe a series of taken pictures of a checkerboard with the corresponding identification of its corners, which will then be used by the OpenCV library to determine the calibration parameters of the camera that took those pictures.

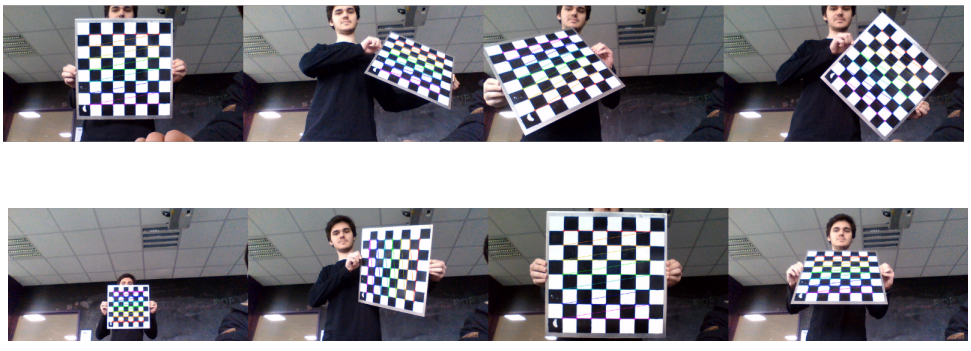


Figure 2: Camera calibration with checkerboard pattern

II.2 Cup identification

The initial phase of the computer vision process involved the identification of the cup, which serves as the target for the tossing action. This objective was achieved through the implementation of two distinct methods: Artificial Intelligence utilizing the YOLO detection system and pose estimation employing ArUco markers. Each of these methods will be elaborated upon, and a comparative analysis will be conducted to evaluate their respective performance.

II.2.1 The YOLO Algorithm

The YOLO algorithm, short for "You Only Look Once," is a neural network-based approach that enables real-time object detection. This algorithm is renowned for its efficiency and accuracy, making it a popular choice. It has found application in diverse domains such as traffic signal detection, pedestrian identification, parking meter recognition, and animal tracking. YOLO utilizes convolutional neural networks (CNNs) to achieve real-time object detection, with pre-trained models available in multiple versions capable of identifying up to 80 distinct object classes (among them the cup). As its name implies, the algorithm only requires a single forward propagation through the neural network to detect objects.

For the purpose of this project, two distinct versions of the YOLO algorithm were implemented. The **YOLOv3** version was employed for cup identification in static images, primarily chosen for its robust performance in such scenarios. On the other hand, **YOLOv3-spp** was utilized for real-time cup identification through video capture, mainly due to its enhanced capabilities for object recognition within shorter time frames. A quantitative comparison between the two YOLO versions can be seen in the table below, comparing the mean average precision, the floating point operations and the frames per second of each of the models.

Model	mAP	FLOPs	FPS
YOLOv3	57.9	140.69 Bn	20
YOLOv3-spp	60.6	141.45 Bn	20

Table 1: YOLO performance on the COCO Dataset [5]

The primary objective behind the implementation of the YOLO algorithm was to seamlessly integrate it with the Python-based OpenCV library. This integration aimed to facilitate the detection of cup borders, followed by a pixel-to-centimeter transformation enabling the determination of the cup's spatial position relative to the camera. Additionally, a supplementary endeavor was made to estimate the depth of the cup. It is worth noting that depth estimation typically relies on the triangulation of visual cues like parallax or texture gradient, which are not accessible when utilizing a single camera.

Assuming that the used camera was already calibrated and its parameters were properly obtained, the mathematical walk trough for estimating the cup real dimensions is presented below.

$$K = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Where K is the camera intrinsic matrix, fx and fy are the focal lengths along the x and y axes, respectively, and (cx, cy) is the principal point (optical center) of the camera. The 3D world coordinates, X, are represented as $X = [X_w, Y_w, Z_w]$, where X_w, Y_w, and Z_w represent the real-world coordinates of the cup. The 2D image coordinates, x, are represented as x = [u, v], where u and v represent the pixel coordinates of the cup in the image. To perform the transformation, homogeneous coordinates are used. The 3D world coordinates (X) and 2D image coordinates (x) are extended to homogeneous coordinates by appending a 1, resulting in $X_h = [X_w, Y_w, Z_w, 1]$ and $x_h = [u, v, 1]$. The transformation from pixel size to real size can be expressed using the following equation:

$$X_h = P \cdot Xx_h = K \cdot X_h \tag{2}$$

To obtain the transformed pixel coordinates, dehomogenization is applied by dividing the x_h coordinates by the third component ($x_h[2]$). The transformed pixel size of the cup can be obtained as:

$$\text{transformed\_pixel\_size} = [x_h[0]/x_h[2], x_h[1]/x_h[2]] \tag{3}$$

The identification of the cup in the controlled scenario is depicted in the picture below.

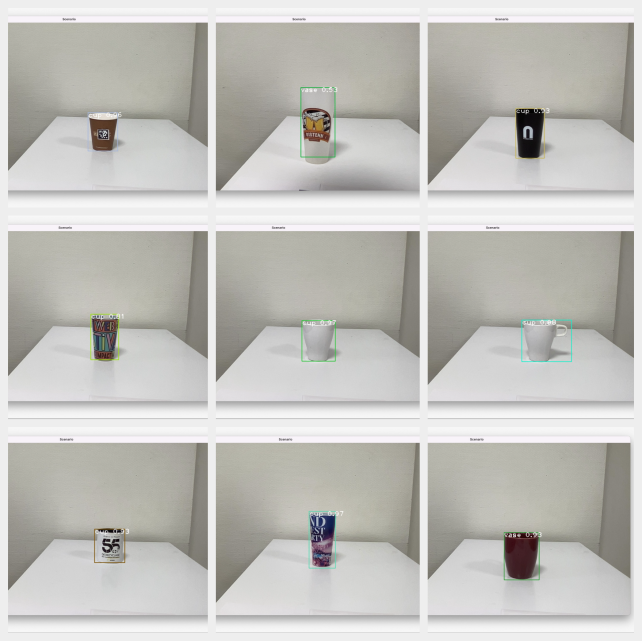


Figure 3: Different cup types identified by the YOLO algorithm

II.2.2 ArUco markers

Pose estimation involves establishing correspondences between real-world points and their two-dimensional image projections. To simplify this process, synthetic or fiducial markers like ArUco tags are commonly used. These markers offer a significant advantage: a single marker provides enough information (its four corners) to determine the camera pose. Also, the inner binary codification makes them specially robust, allowing the possibility of applying error detection and correction techniques. ArUco markers are square markers with a black border and an inner binary matrix that encodes their unique identifier (ID). The black border makes them easy to detect in images, while the binary codification enables identification and the use of error detection and correction techniques. The size of the marker determines the size of the internal matrix. For example, a 4x4 marker consists of 16 bits.

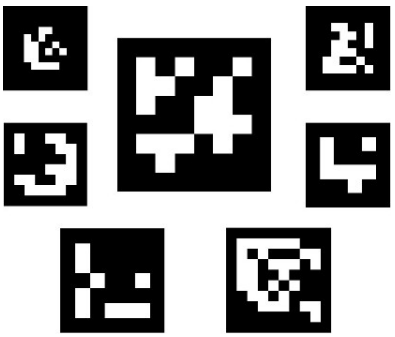


Figure 4: Examples of ArUco markers [1]

For this project, an ArUco marker was affixed to the outer surface of the cups being tested, as illustrated in Figure 5. While this approach of item identification may be considered more "artificial," it has demonstrated faster performance compared to the YOLO approach. Moreover, it is better suited for accurately measuring the relative distance and angle between the cup and the robot, as elaborated in the subsequent sections.

Figure 5: Cup containing ArUco marker

II.3 Robot identification

In this study, due to the unavailability of the physical robot for the participants involved in the 2-D Computer Vision task, the implementation of AI tools specifically designed for cup detection was not employed for robot detection. Consequently, to simulate the robot's position, ArUco markers were strategically positioned in the scenario. These markers served the purpose of estimating the distance between two distinct ArUco markers: one representing the base of the robot and the other fixed in the external surface of the cup. The forthcoming section will elaborate on the details of the distance estimation procedure.

II.4 Distance estimation

The majority of calculations regarding distance estimation through a monocular camera were made by using various advantages that ArUco markers provide. The following subsection will focus on the description of the 3D vectors that can be obtained by detecting these markers and the perspective transformations that arise from their usage.

II.4.1 Rotation matrix and translation vector

The functions used in OpenCV to estimate an ArUco marker's position follow the pinhole camera model. Therefore, a 3D scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$sm' = A[R|t]M' \tag{4}$$

or equivalently,

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{5}$$

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
- (cx, cy) is a principal point that is usually at the image center
- fx, fy are the focal lengths expressed in pixel units

Considering the camera matrix as a known constant from the previous calibration process, the equation can be reduced to

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \tag{6}$$

In this case, the world coordinates take as origin the point where the camera is placed. This is useful to show the point as a projection on a 2D image for measurement purposes, but in order to get the distance and angle between the robotic arm and the cup some perspective transformations will be needed.

II.4.2 Perspective Transformations

Once an ArUco marker has been identified, OpenCV describes its pose by using two vectors highly related to the previous subsection, which will be referred to as *tvec* and *rvec*, vectors which state a translation and a rotation from the camera, respectively. *tvec* is a 3-dimensional euclidean translation vector, while *rvec* is known as a Rodrigues rotation vector, and is described by using a unit vector and a rotation angle around it.

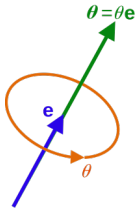


Figure 6: Rodrigues rotation vector

By using the Rodrigues method provided by the OpenCV library, *rvec* can be transformed into a rotation matrix R. This can be used in addition to the translation vector *tvec* to transform a 3D point expressed in one coordinated system to another one:

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{cam} = \begin{bmatrix} {}^{cam}R_{tag} & {}^{cam}t_{tag} \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{tag} \tag{7}$$

where:

- ${}^{cam}R_{tag}$ is the rotation matrix obtained by using the Rodrigues method on an *rvec* resulting from the ArUco tag detection
- ${}^{cam}t_{tag}$ is the translation vector *tvec* resulting from the ArUco tag detection.
- The vector with subscript *cam* is a 3D vector measured using the camera's coordinate system.
- The vector with subscript *tag* is a 3D vector measured using the ArUco marker's coordinate system.

By using this transformation it's possible to move from the camera's coordinated system to the one of a tag and viceversa.

Gathering the tools already discussed, there are notably two ways of estimating the distance and angle from the robotic arm to the cup: one is by using two markers, and the other one is by using one marker and relying on the AI detection of the cup's bounding box. Both approaches will be explained in the following subsections.

II.4.3 Distance and angle estimation between two ArUco markers

This method requires using two markers: one can be at the base of the robotic arm and the other one glued to a side of the cup.

The magnitude of the distance can be obtained by getting the norm of the difference between the *tvecs* resulting from the ArUco marker detection method.

$$d = |t_{arm} - t_{cup}| \tag{8}$$

where:

- t_{arm} is the *tvec* denoting the 3D vector from the camera to the marker present on the robotic arm.
- t_{cup} is the *tvec* denoting the 3D vector from the camera to the marker present on the cup.

In order to get the angle the cup is at, the arm's marker needs to be placed in such a way that the arm's main axis must be aligned to one of the marker's.

Once having decided the marker's placement, to get the angle it's easier to transform the perspective of t_{cup} to be represented using the arm's coordinate system, therefore allowing its trigonometric angle calculation.

$$\alpha = arccos(e_{arm} \cdot {}^{arm}t_{cup}) \tag{9}$$

where:

- e_{arm} is a unit vector of the arm's marker in the main direction of the arm
- ${}^{arm}t_{cup}$ is the normalized *tvec* denoting the 3D vector from the arm's marker to the cup's marker.

It's worth noting that this distance estimation can be improved by using information from the cup's marker. By having the measurement of the cup's radius, through perspective transformations it's possible to pinpoint a specific place inside the cup

Let Z be the axis that is normal to the cup's side surface. Then, the center of the cup can be found at $c_{cup} = (0, 0, r)^T$, with r radius of the cup at the height the marker is placed. Then,

$$^{cup}t_{center} = \begin{bmatrix} 0 \\ 0 \\ r \end{bmatrix} \tag{10}$$

By using the cup's *rvec* to get its Rotation matrix, a change of perspective allows finding that point using the arm's coordinate system.

The same procedure may be applied for any desired point inside the cup, given that enough information is available. For example, knowing the ball that will be thrown will reach the cup following a parabolic trajectory, the optimal point where the ball has to be thrown at can be estimated within a zone. However, this approach wasn't explored during this semester's version of the project.

II.4.4 Distance and angle estimation between one ArUco marker and the cup's AI detection

This procedure is similar to the one followed above, with some tradeoffs: it allows for greater flexibility in the cup's choice, as there's no need to place an ArUco marker on it, but some precision may be lost due to the high dependency on the precision of the bounding box in the cup's depth calculation.

The main difference is the calculation of t_{cup}. As there's no pose estimation information available, a way to calculate the depth the cup is at is to use the previously mentioned transformed pixel size of the cup. This can be multiplied by the focal length along the x axis (assuming a planar scene) to get an estimate of the cup's depth.

To get the X and Y coordinates of the cup in the camera's coordinated system, by using the intrinsic camera matrix it's possible to obtain the normalized device coordinates:

$$\begin{bmatrix} x_{ndc} \\ y_{ndc} \end{bmatrix} = \frac{1}{f_x} \begin{bmatrix} x_c cx \\ y_c cy \end{bmatrix} \tag{11}$$

And by dividing by the depth, it's possible to get the cup's 3D position using the camera's coordinate system:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \frac{x_{ndc}}{depth} \\ \frac{y_{ndc}}{depth} \\ depth \end{bmatrix} \tag{12}$$

Finally a perspective transformation is required to get the same vector in the arm marker's coordinate system, and from that point the same procedure described above can be used to get the distance and angle from the arm's marker to the cup.

II.5 3D vision

Part of the work in this project is based on a RGB-D sensor. Technically speaking, RGB-D means that a depth (D) measurement is added to the colour (RGB) camera. It solves the problem that conventional monocular cameras cannot sense depth. Currently, there are three main approaches to achieve this depth measurement: stereo vision, Time-of-Flight (ToF), and structured light. Initially, the project provided the ZED2 camera, which utilizes stereo vision for depth measurement. However, experiments revealed poor accuracy (approximately 1-3cm error), making it incapable of providing accurate depth information. Therefore, in subsequent development, we switched to Microsoft's Kinect v1 camera. It employs structured light for depth measurement, achieving accuracy within 3mm.

II.5.1 3D point cloud acquisition

A 3D point cloud is a collection of points in three-dimensional space that represents the geometry or shape of an object or scene. Each point in the point cloud has three coordinates (x, y, z) that define its position in the 3D space. With a 3D point cloud, it is possible to reconstruct objects and obtain highly accurate positional information. Firstly, we will explain how to obtain a 3D point cloud.

The Kinect v1 camera has two sensors: one for capturing color images and another for capturing depth images. The depth image is represented as a gray scale image, where each pixel value corresponds to the depth information of that point. We are going to generate a three-dimensional point cloud from this depth image. Additionally, since these pixels only contain one channel information, we need to align the depth image with the color image to obtain color information (registration). The implementation of these two steps is described below.

Based on the camera matrix K, we can establish the relationship between pixel coordinates and real-world coordinates as follows:

$$z\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}fx&0&cx\\0&fy&cy\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=K\begin{bmatrix}x\\y\\z\end{bmatrix} \tag{13}$$

Since neither fx, nor fy is zero, K is an invertible matrix, so we have:

$$K\begin{bmatrix}u\\v\\1\end{bmatrix}=zK^{-1}\begin{bmatrix}u\\v\\1\end{bmatrix} \tag{14}$$

This equation establishes the mapping from each pixel to the real-world coordinates. With the depth image obtained from the Kinect, we have the depth value (z) corresponding to the position of each pixel. By applying this transformation to all the pixels, we can obtain the three-dimensional point cloud of the scene.

The registration operation between the depth image and the color image involves transforming the pixel coordinates of the depth image to the pixel coordinate system of the color image. In other words, we aim to find a transformation matrix H such that:

$$x_h^c = Hx_h^d \tag{15}$$

where x_h^c and x_h^d represent the homogeneous coordinates of a point in the color image and the depth image, respectively. To find this transformation, we can first represent the points in the depth camera coordinate system in the coordinate system of the color camera:

$$X_h^c = {}^cT_d X_h^d \tag{16}$$

In the equation, X_h^c and X_h^d represent the homogeneous coordinates of a point in the color and depth camera coordinate systems, respectively. ${}^cT_d = \begin{bmatrix}{}^cR_d & {}^ct_d\\0&1\end{bmatrix}$ is the rigid transformation matrix, consisting of a 3x3 rotation matrix ${}^cR_d$ and a 3x1 translation vector ${}^ct_d$. The rotation matrix ${}^cR_d = [i_d, j_d, k_d]$ represents the coordinates of the depth camera coordinate system's principal axes in the color camera coordinate system. The translation vector ${}^ct_d = [o_x, o_y, o_z]^T$ represents the position of the origin of the depth camera coordinate system. Once we have the coordinates of a point in the color camera coordinate system, we can use equation 13 to obtain the corresponding pixel coordinates of that point. By performing this calculation for each point in the 3D point cloud, we can obtain the registered color 3D point cloud, as shown in Figure 7.

Figure 7: Point cloud of the scene

It should be noted that during the process of obtaining the point cloud and image registration, we utilized methods provided by Microsoft's SDK. The camera matrix K and ${}^cT_d$ are embedded in the C++ source files, which we cannot directly access. As a result, this posed some challenges in processing the point cloud data for further analysis with Python and in integrating into ROS system. We have found a way to exploit the camera on ROS, but this requires us to obtain these parameters through manual calibration. However, due to the presence of a filter on the depth camera, the captured images tend to be very dark. Therefore, when using the chessboard calibration method, an additional infrared light source is required to make the images brighter and enhance their quality.

II.5.2 Plane segmentation

In the point cloud scene, our main interest lies in identifying the object (in this case a cup) and segmenting it from the rest. If we were to apply the YOLO algorithm used for 2D images, we would only obtain a bounding box containing the cup, along with the surrounding pixels. While we could use other neural networks like Mask R-CNN for instance segmentation, they tend to be quite complex to construct and may not have readily available models. Therefore, in this case, we will use a rather traditional 3D point cloud segmentation method.

In real-world scenarios, objects are typically placed on a flat tabletop surface with no other objects around them. Hence, we can remove the tabletop to extract the point cloud of the object.

In our cases, assuming that the camera is fixed in its position, we start by selecting the workspace in the 2D image where the cup is located. Then, we extract the point cloud from this workspace, ensuring that it contains only the tabletop and the cup (Figure 8).

Figure 8: Point cloud containing only cup and tabletop

Once we have the point cloud of the region, we can apply the RANSAC (RANdom SAmple Consensus) algorithm to remove the tabletop. RANSAC is an iterative statistical algorithm used for model fitting and outlier removal. It was initially introduced by Fischler and Bolles in 1981 and is widely used in the fields of computer vision and computer graphics. The principle behind removing the plane using RANSAC is as follows:

1. Select three non-collinear points from the 3D point cloud of the scene and add them to the **inliers** set. Based on these three points, we can define a plane and its corresponding parameters, **params**.
2. Calculate the distance from all points in the scene's point cloud, excluding **inliers**, to the plane. If a point's distance to the plane is below a threshold **dist**, we consider it to be on the plane and add it to the **inliers** set.
3. Iterate through all points in the point cloud following the above steps. Count the number of points in **inliers**. If the count exceeds a threshold **t**, we consider the current plane as a possible correct plane. We further use the least squares method to find the plane and its corresponding parameters, **params**, that best fit the points in **inliers**. Calculate the average distance, **model_error**, from all points in the point cloud to the plane.
4. If **error** is smaller than the current minimum average distance, **best_error**, update **best_params** with **params** and update **best_error** with **error**.
5. Increase the iteration count and start the next iteration. Repeat the above steps until the maximum number of iterations is reached. Eventually, we obtain the parameters of the best-fit plane, **best_params**, from the current scene's point cloud.

It is important to note that setting a suitable threshold **t** is crucial; if it is too large, it may fail to find the possible correct plane, leading to the failure of the RANSAC algorithm. The pseudo-code is presented in Algorithm 1 and the effect of this algorithm is shown in Figure 9.

Figure 9: The result of plane segmentation

Algorithm 1 Remove plane using RANSAC

Input: pcd - Point cloud
k - Maximum number of iterations
t - The minimum number of inlier points to determine a plane
dist - The distance threshold to determine an inlier point
Output: best_params - The parameters of a plane

```
itr = 0
best_params = null
best_error = +∞
while itr < k do
  inliers = select_three_points(pcd)
  params = find_plane(inliers)
  for point in pcd\inliers do
    error = distance from point to current plane
    if error < t then
      inliers.add(point)
    end if
  end for
  if card(inliers) > t then
    params = find_plane(inliers)
    model_error = average distance from points to plane
    if model_error < best_error then
      best_params = params
      best_error = error
    end if
  end if
  itr ++
end while
return best_params
```

II.5.3 Noise reduction

During the process of obtaining depth maps, noise and invalid values may be generated due to sensor errors, specular reflections on object surfaces, and large depth variations at object edges. These points can interfere with subsequent calculations of object positions, so it is necessary to find a method to remove them.

Firstly, for points that fail to be captured, the Kinect sets their depth value to 0. Therefore, we can simply remove the points in the point cloud where $z = 0$. As for noise and outliers, common filtering methods include Gaussian filtering, statistical filtering, radius filtering, bilateral filtering, etc. In this case, we will use the radius filtering method.

The core idea of this method is to draw a fixed-radius sphere at each point and count the number of points within that sphere. If the count exceeds a given threshold, the point is retained; otherwise, it is considered an outlier and removed from the point cloud. Compared to other filtering methods, this algorithm is simpler, faster, and more suitable for handling edge points. The result is shown in Figure 10.

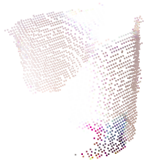


Figure 10: The result of filtering

II.5.4 Registration of two point sets for pose estimation

Now that we have obtained the point cloud of the cup, it is evident that this point cloud is incomplete, and we cannot determine the precise position and orientation of the cup. In order to compute the cup's pose, the idea is to match the generated model point cloud with the actual point cloud of the cup, i.e. find a rotation matrix R and translation vector t which satisfy the following equation:

$$X_{cup} = RX_{model} + t \tag{17}$$

where X_{cup} is a point in cup point cloud, X_{model} is its corresponding point in model point cloud. This model point cloud has its center at the origin of the coordinate system, and its principal axes coincide with the coordinate system's principal axes. If we know how the model is transformed into the position of the cup through a combination of rotation and translation, we will know the pose information of the cup. This process is also called registration. The key aspect of registration lies in determining the corresponding points between the cup point cloud and the model point cloud.

Since the obtained cup point cloud only contains three-dimensional coordinates and color information, we cannot deduct the corresponding point pairs between it and model. Therefore, we need to extract a higher-dimensional feature, which is known as feature descriptors. In this case, we have chosen Fast Point Feature Histograms (FPFH) as the descriptor for feature matching. The FPFH algorithm computes a histogram that captures the geometric properties of a point in relation to its neighboring points. This enables us to determine the type of a point accordingly and thus determine its corresponding point in another point cloud. With the FPFH features, we can apply an algorithm for point cloud registration. Here, we use a combination of RANSAC and ICP algorithms.

RANSAC is the algorithm we previously used for plane fitting. However, in this case, the model we want to fit is (17). The fitting process is different and as follows:

1. Randomly select three points from the model point cloud. Find the most similar points in the cup point cloud based on their FPFH features.
2. Once corresponding points are found, use the SVD decomposition method to estimate the transformation matrix between the point pairs.
3. Apply this transformation to all model points and calculate the distance errors between the transformed points and their corresponding points.
4. If the distance error for any point pair is smaller than a predefined threshold, consider it an inlier; otherwise, consider it an outlier. Count the number of inliers.
5. Repeat the above steps until the number of inliers exceeds a threshold in a fitting iteration or until the maximum number of iterations is reached. If the latter occurs, the solution is considered a failure.

Since this algorithm solves the model by randomly selecting three points during the implementation, we cannot guarantee that the results will provide a good point cloud match. We can improve the results by reducing the distance error, but this increases the risk of failure. Therefore, we choose to enhance the registration by applying the ICP (Iterative Closest Point) algorithm on top of the RANSAC alignment.

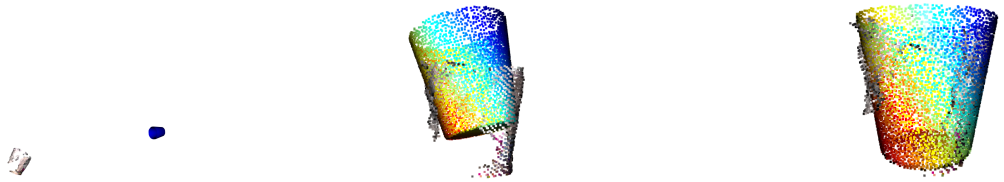
ICP is a commonly used point cloud registration algorithm. It iteratively searches for the best transformation that minimizes the distance between corresponding points in two point clouds. The algorithm consists of two main steps:

1. Apply the current transformation R and t to find the corresponding set $K = (p, q)$ between the target point cloud P and the source point cloud Q. The nearest neighbor search is used to find the corresponding points.
2. Perform calculations on K to update R and t in order to minimize the objective function $E(R, t)$.

By repeating these two steps iteratively, the algorithm achieves precise registration. The objective function is typically defined as:

$$E(R,t) = \Sigma_{(p,q)\in K} \|p - Rq - t\|^2 \tag{18}$$

Compared to the RANSAC algorithm, the ICP algorithm has a higher computational complexity, but it provides more accurate results. To combine the advantages of both, we first use RANSAC to compute an approximate transformation matrix. Then, we apply this transformation to the point cloud and use it as input for the more fine calculations of the ICP algorithm. This approach improves accuracy while ensuring efficiency. Figure 11 shows the results after applying the RANSAC algorithm and further refining with the ICP algorithm.



(a) Before registration (b) After RANSAC (c) After ICP

Figure 11: Registration with RANSAC and ICP

III Control of the robotic arm

III.1 Introduction to the Mathematical Model

The first important step in controlling any physical system is to acquire a complete and accurate mathematical model of the system. This model is useful for simulation of the system before implementing the system into a real-time application. In this section, the mathematical equations for the kinematic and dynamic model of the robotic manipulator are discussed. The kinematic analysis has been classified into two types, forward and inverse kinematics. The dynamic behavior of the manipulator describe the relationship between force and motion which represent respectively the input and the output of the system.

III.2 Kinematic Model

III.2.1 Direct Kinematic Model

Goal:

Express the Cartesian position of the gripper of the robot according to the angles of the joints of the robot.

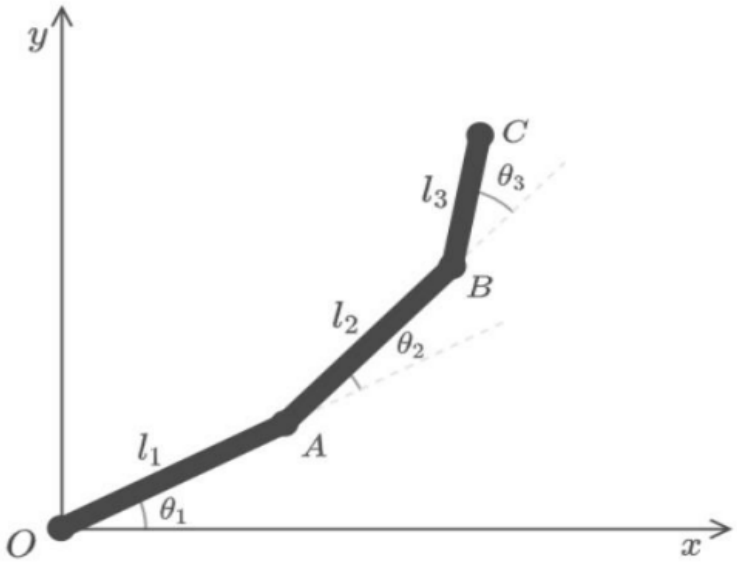


Figure 12

In other words we have to express x_3 and y_3 in function of $\theta_1, \theta_2, \theta_3$ and l_1, l_2, l_3.
By making projections, we have:

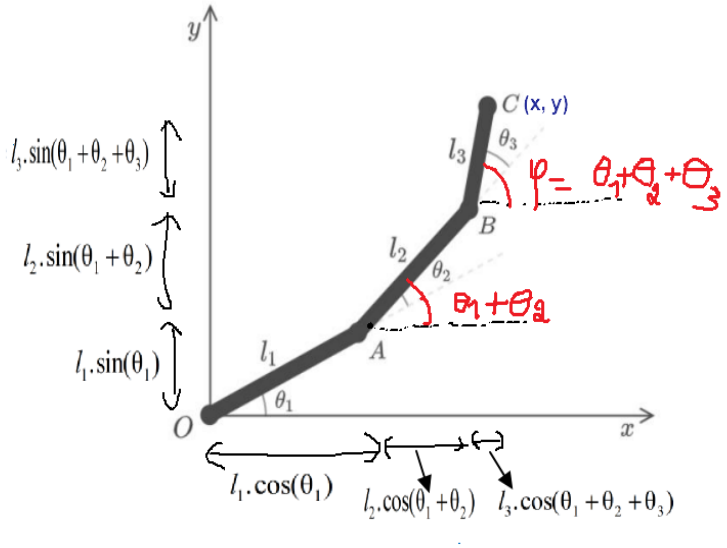


Figure 13

Consequently, we find that:

$$x = l_1.\cos(\theta_1) + l_2.\cos(\theta_1+\theta_2) + l_3.\cos(\theta_1+\theta_2+\theta_3)$$
$$y = l_1.\sin(\theta_1) + l_2.\sin(\theta_1+\theta_2) + l_3.\sin(\theta_1+\theta_2+\theta_3)$$

III.2.2 Inverse Kinematic Model

Goal:

Express the angles of the joints of the robot according to the Cartesian position of the gripper of the robot.

Note that:

$$x_B = x - l_3\cos(\varphi)$$
$$y_B = y - l_3\sin(\varphi)$$

By doing trigonometric operations, we find that:

$$\theta_2 = \cos^{-1}\left(\frac{({x_B}^2 + {y_B}^2) - (l_1^2 + l_2^2)}{2.l_1.l_2}\right)$$
$$\theta_1 = \sin^{-1}\left(\frac{(l_1 + l_2\cos(\theta_2))y_B - (l_2\sin(\theta_2))x_B}{x_B^2 + y_B^2}\right)$$
$$\theta_3 = \varphi - \theta_1 - \theta_2$$

III.3 Dynamical model

III.3.1 Robot Dynamics: Problem Formulation

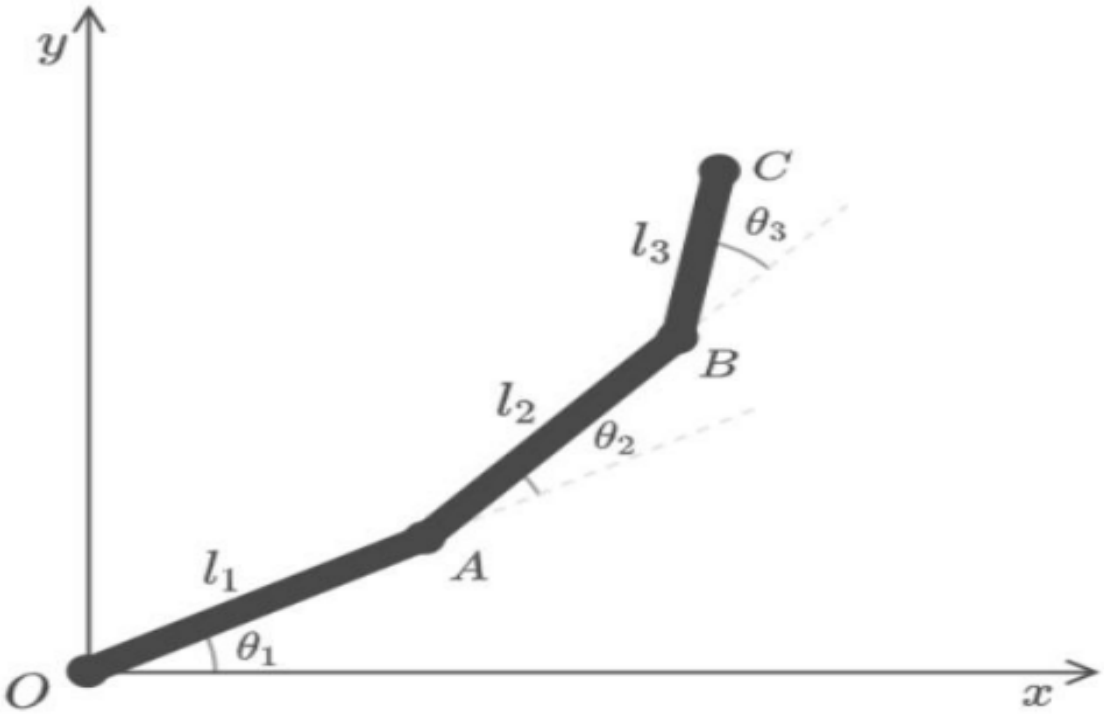


Figure 14: Robotic Arm

As mentionned previously, dynamic equations describe the relationship between force and motion. The forces (torques) which are applied to the joints by actuators are denoted by :

$$\tau = \begin{pmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{pmatrix}$$

These forces make the joint angles change when applied to the robot.

The position, velocity and acceleration of the joints are denoted:

$$q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

$$\dot{q} = \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix}$$

$$\ddot{q} = \begin{pmatrix} \ddot{q}_1 \\ \ddot{q}_2 \\ \ddot{q}_3 \end{pmatrix}$$

The main method that can be used for obtaining the dynamic model of the manipulator is the Euler Lagrangian approach [7] which gives an energy difference between the kinetic energy and potential energy. This approach is mostly used in deriving the complicated mathematical model of a robot manipulator Therefore, we have used the Euler -Lagrange method in this project.

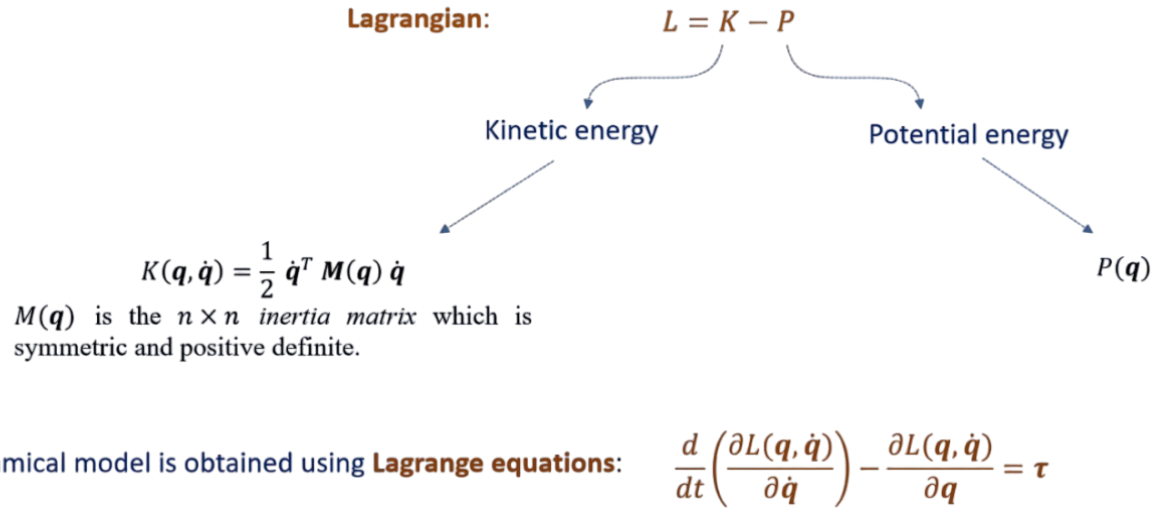


Figure 15: Lagrange Equation

$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = K(\boldsymbol{q}, \dot{\boldsymbol{q}}) - P(\boldsymbol{q}) = \frac{1}{2}\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}} - P(\boldsymbol{q})$

$\frac{\partial L(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \dot{\boldsymbol{q}}} = \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}}$

$\frac{d}{dt}\left(\frac{\partial L(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \dot{\boldsymbol{q}}}\right) = \frac{d}{dt}(\boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}}) = \dot{\boldsymbol{M}}(\boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}}$

$\frac{\partial L(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \boldsymbol{q}} = \frac{\partial K(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \boldsymbol{q}} - \frac{\partial P(\boldsymbol{q})}{\partial \boldsymbol{q}} = \frac{1}{2}\frac{\partial}{\partial \boldsymbol{q}}\left(\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}}\right) - \frac{\partial P(\boldsymbol{q})}{\partial \boldsymbol{q}}$

$\frac{d}{dt}\left(\frac{\partial L(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \dot{\boldsymbol{q}}}\right) - \frac{\partial L(\boldsymbol{q},\dot{\boldsymbol{q}})}{\partial \boldsymbol{q}} = \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \dot{\boldsymbol{M}}(\boldsymbol{q})\dot{\boldsymbol{q}} - \frac{1}{2}\frac{\partial}{\partial \boldsymbol{q}}\left(\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\dot{\boldsymbol{q}}\right) \frac{\partial P(\boldsymbol{q})}{\partial \boldsymbol{q}} = \boldsymbol{\tau}$

$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau}$

We can write the final equation in the following form:

$$\ddot{q} = M^{-1}(q).\,(\tau - \dot{q}.\,C(q, \dot{q}) - G(q))$$

with:

$q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$: Vector of joint coordinates.

$M(q) \in R^{3x3}$: Inertia Matrix.

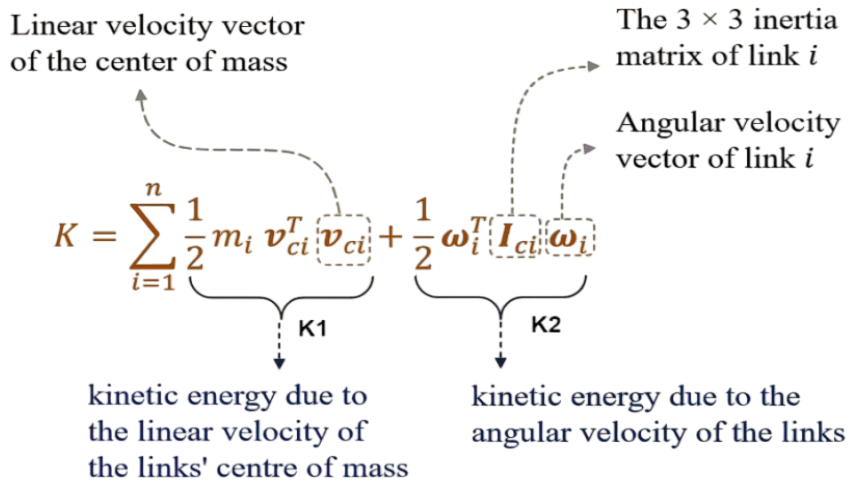
$\dot{q}.\,C(q, \dot{q}) \in R^3$: Vector of Coriolis and Centrifugal Forces.

$G(q) \in R^3$: Vector of Gravity Forces.

$\tau = \begin{pmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{pmatrix}$: Vector of Joint Torques.

III.3.2 Kinetic Energy Computation

The overall kinetic energy of the manipulator with n links is:



The objective is to calculate K and deduce the matrix M(q) since we have that:

$$K(q, \dot{q}) = \frac{1}{2} \dot{q}^t . M(q) . \dot{q}$$

We start by calculating the position of the center of masses of each link. To do that, let's note C1, C2, C3 the center of masses of link 1, 2 and 3 and :

$$l_{C1} = \frac{l_1}{2}$$
$$l_{C2} = \frac{l_2}{2}$$
$$l_{C3} = \frac{l_3}{2}$$

Hence,

$$x_{C1} = l_{C1} . \cos(q_1)$$
$$x_{C2} = l_1 . \cos(q_1) + l_{C2} . \cos(q_1 + q_2)$$
$$x_{C3} = l_1 . \cos(q_1) + l_2 . \cos(q_1 + q_2) + l_{C3} . \cos(q_1 + q_2 + q_3)$$

$$y_{C1} = l_{C1} . \sin(q_1)$$
$$y_{C2} = l_1 . \sin(q_1) + l_{C2} . \sin(q_1 + q_2)$$
$$y_{C3} = l_1 . \sin(q_1) + l_2 . \sin(q_1 + q_2) + l_{C3} . \sin(q_1 + q_2 + q_3)$$

From the given equations, we can determine the velocities of the center of masses by differentiating the positions of each center of mass.

By doing so, we can subsequently compute the first term, K1, of the K matrix.

After extensive analysis and derivation, we arrive at the following expression for K1:

$$K1(q, \dot{q}) = \frac{1}{2} [\dot{q}_1 \ \dot{q}_2 \ \dot{q}_3] . \begin{pmatrix} \alpha_1 & \beta & \gamma \\ \beta & \alpha_2 & \eta \\ \gamma & \eta & \alpha_3 \end{pmatrix} . \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}$$

with:

$$\alpha_1 = m_1 {l_{c1}}^2 + m_2 ({l_1}^2 + {l_{c2}}^2 + 2 l_1 l_{c2} \cos(q_2)) + m_3 ({l_1}^2 + {l_2}^2 + {l_{c3}}^2 + 2 l_1 l_2 \cos(q_2) + 2 l_2 l_{c3} \cos(q_3) + 2 l_1 l_{c3} \cos(q_2 + q_3))$$

$$\beta = m_2 ({l_{c2}}^2 + l_1 l_{c2} \cos(q_2)) + m_3 ({l_2}^2 + {l_{c3}}^2 + l_1 l_2 \cos(q_2) + 2 l_2 l_{c3} \cos(q_3) + l_1 l_{c3} \cos(q_2 + q_3))$$

$\gamma = m_3({l_{c3}}^2 + l_2 l_{c3}\cos(q_3) + l_1 l_{c3}\cos(q_2+q_3))$

$\alpha_2 = m_2{l_{c2}}^2 + m_3({l_2}^2 + {l_{c3}}^2 + 2l_2 l_{c3}\cos(q_3))$

$\eta = m_3({l_{c3}}^2 + l_2 l_{c3}\cos(q_3))$

$\alpha_3 = m_3{l_{c3}}^2$

Computation of K2

$$K2 = \frac{1}{2}\sum_{i=1}^{3} \dot{q}_i^t . I_C . \dot{q}_i$$

$$K2 = \frac{1}{2}\dot{q}_1^2 I_1 + \frac{1}{2}(\dot{q}_1^2 + \dot{q}_2^2)I_2 + \frac{1}{2}(\dot{q}_1^2 + \dot{q}_2^2 + \dot{q}_3^2)I_3$$

$$K2 = \frac{1}{2}\begin{bmatrix}\dot{q}_1 & \dot{q}_2 & \dot{q}_3\end{bmatrix} . \begin{pmatrix} I_1+I_2+I_3 & I_2+I_3 & I_3 \\ I_2+I_3 & I_2+I_3 & I_3 \\ I_3 & I_3 & I_3 \end{pmatrix} . \begin{bmatrix}\dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3\end{bmatrix}$$

Therefore we finally have:

$$K(\boldsymbol{q}, \dot{\boldsymbol{q}}) = K_1 + K_2 = \frac{1}{2}\dot{\boldsymbol{q}}^T \boldsymbol{M}(\boldsymbol{q})\,\dot{\boldsymbol{q}}$$

And

$$M(q) = \begin{pmatrix} \alpha_1+I_1+I_2+I_3 & \beta+I_2+I_3 & \gamma+I_3 \\ \beta+I_2+I_3 & \alpha_2+I_2+I_3 & \eta+I_3 \\ \gamma+I_3 & \eta+I_3 & \alpha_3+I_3 \end{pmatrix}$$

We will see later that M is symmetric definite positive.

NB:

I_1, I_2, I_3 are the moments of inertia of the links 1,2 and 3.

To calculate them, we can apply the parallel axis theorem. This theorem establishes a relationship between the moment of inertia of an object about an axis parallel to its center of mass and its moment of inertia about an axis passing through its center of mass.

The formula for the moment of inertia about an axis parallel to the center of mass of link i is given by: $I_i = I_{C_i} + m_i . {d_i}^2$

where:

- I_{C_i} is the moment of inertia of link i about an axis through its center of mass (I_{C_i} is the sum of the principle moments of inertia I_{xx}, I_{yy}, I_{zz} which are given in the robot documentation.
- m_i is the mass of link i.
- d_i is the perpendicular distance between the parallel axis and the center of mass of link i. Using the parallel axis theorem $d = \sqrt{\frac{I_{xx}+I_{yy}}{2}}$

III.3.3 Computation of C(q, q̇) and G(q)

Recall from 1.2.1 that

$$\ddot{q}.M(q) + \dot{q}.\,C(q,\dot{q}) + G(q) = \tau$$

$$\underbrace{M(q)\ddot{q} + \dot{M}(q)\dot{q} - \frac{1}{2}\frac{\partial}{\partial q}(\dot{q}^T M(q)\dot{q})}_{C(q,\dot{q})\dot{q}} + \underbrace{\frac{\partial P(q)}{\partial q}}_{g(q)} = \tau$$

D(q,q̇)

Computation of $\dot{\mathbf{q}}.\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$

$$\dot{q}.C(q,\dot{q}) = \dot{M}(q).\dot{q} - \frac{1}{2}.D(q,\dot{q})$$

By deriving M accordingly, we obtain that:

$$\dot{M}(q).\dot{q} = \begin{pmatrix} \dot{q}_1\alpha'_1 + \dot{q}_2\beta' + \dot{q}_3\gamma' \\ \dot{q}_1\beta' + \dot{q}_2\alpha'_2 + \dot{q}_3\eta' \\ \dot{q}_1\gamma' + \dot{q}_2\eta' \end{pmatrix}$$

With:

$$\alpha'_1 = -2m_2l_1l_{c2}\dot{q}_2\sin(q_2) - 2m_3l_1l_2\dot{q}_2\sin(q_2) - 2m_3l_2l_{c3}\dot{q}_3\sin(q_3) - 2m_3l_1l_{c3}(\dot{q}_2+\dot{q}_3)\sin(q_2+q_3)$$

$$\beta' = -m_2l_1l_{c2}\dot{q}_2\sin(q_2) - m_3l_1l_2\dot{q}_2\sin(q_2) - 2m_3l_2l_{c3}\dot{q}_3\sin(q_3) - m_3l_1l_{c3}(\dot{q}_2+\dot{q}_3)\sin(q_2+q_3)$$

$$\gamma' = -m_3l_2l_{c3}\dot{q}_3\sin(q_3) - m_3l_1l_{c3}(\dot{q}_2+\dot{q}_3)\sin(q_2+q_3)$$

$$\alpha'_2 = -2m_3l_2l_{c3}\dot{q}_3\sin(q_3)$$

$$\eta' = -m_3l_2l_{c3}\dot{q}_3\sin(q_3)$$

By developping the expression:

$$D(q,\dot{q}) = \frac{\delta}{\delta q}[\dot{q}^t.M(q).\dot{q}]$$

We obtain :

$$D(q,\dot{q}) = \begin{pmatrix} 0 \\ -2\,[\,m_2l_1l_{c2}(\dot{q}_1^2+\dot{q}_1\dot{q}_2)\sin(q_2) + m_3l_1l_2(\dot{q}_1^2+\dot{q}_1\dot{q}_2)\sin(q_2) + m_3l_1l_{c3}(\dot{q}_1^2+\dot{q}_1\dot{q}_2+\dot{q}_1\dot{q}_3)\sin(q_2+q_3)\,] \\ -2\,[\,m_3l_2l_{c3}(\dot{q}_1^2+2\dot{q}_1\dot{q}_2+\dot{q}_1\dot{q}_3+\dot{q}_2^2)\sin(q_3) + m_3l_1l_{c3}(\dot{q}_1^2+\dot{q}_1\dot{q}_2+\dot{q}_1\dot{q}_3)\sin(q_2+q_3) + m_3l_2l_{c3}(\dot{q}_2\dot{q}_3)\sin(q_3)\,] \end{pmatrix}$$

Hence, finally, we have:

$$\dot{\mathbf{q}}.\mathbf{C}(\mathbf{q},\dot{\mathbf{q}}) = \dot{\mathbf{M}}(\mathbf{q}).\dot{\mathbf{q}} - \frac{\mathbf{1}}{\mathbf{2}}.\mathbf{D}(\mathbf{q},\dot{\mathbf{q}})$$

Computation of $G(q) = \frac{\delta}{\delta q}P(q)$

We start by calculating the potential energy associated to the masses m1, m2 and m3 and then add them up to find the total potential energy of the system.

$$P_1 = m_1 \cdot g \cdot l_{c1} \cdot \sin(q_1)$$

$$P_2 = m_2 \cdot g \cdot (l_1 \cdot \sin(q_1) + l_{c2} \cdot \sin(q_1+q_2))$$

$$P_3 = m_3 \cdot g \cdot (l_1 \cdot \sin(q_1) + l_2 \cdot \sin(q_1+q_2) + l_{c3} \cdot \sin(q_1+q_2+q_3))$$

$$P = P_1 + P_2 + P_3$$

$$G(q) = \frac{\delta}{\delta q}P(q) = \begin{pmatrix} \frac{\delta P}{\delta q_1} \\ \frac{\delta P}{\delta q_2} \\ \frac{\delta P}{\delta q_3} \end{pmatrix} = \begin{bmatrix} (m_1.g.l_{c1} + m_2.g.l_1 + m_3.g.l_1).\cos(q_1) + (m_2.g.l_{c2} + m_3.g.l_2).\cos(q_1+q_2) + m_3.g.l_{c3}.\cos(q_1 + \\ (m_2.g.l_{c2} + m_3.g.l_2).\cos(q_1+q_2) + m_3.g.l_{c3}.\cos(q_1+q_2+q_3) \\ m_3.g.l_{c3}.\cos(q_1+q_2+q_3) \end{bmatrix}$$

III.3.4 Conclusion

In conclusion, the Lagrange equation enables the analysis of the robot manipulator arm's dynamics. By using the equation $\ddot{q} = M^{-1}\left[\tau - \dot{q}\cdot C(q,\dot{q}) - G(q)\right]$, we can determine the joint accelerations ($\ddot{q}$) based on the input torque (τ) and the current system state (q and $\dot{q}$).

The equation involves the inertia matrix M, which characterizes the arm's mass distribution. The term $C(q,\dot{q})$ represents the Coriolis and centrifugal forces arising from the arm's motion, while $G(q)$ accounts for gravitational forces. By rearranging the equation, we can isolate the joint accelerations and calculate them using known parameters.

Overall, the Lagrange equation provides a powerful framework for understanding and controlling the dynamics of the robot manipulator arm. It aids in tasks such as trajectory planning, motion control, and stability analysis by predicting joint accelerations in response to various inputs and system configurations.

III.4 Integration of the Model in Matlab/Simulink and Validation

In order to simulate the system and develop a control law, we integrated the dynamical model of the robotic arm in MATLAB and Simulink.

In the file `Dynamical_Model_And_Validation`, we defined all the robot's parameters, including joint masses, lengths, and inertia. We also constructed the Inertia matrix, Coriolis matrix, and Gravitational matrix according to the detailed specifications mentioned in the previous section.

Furthermore, we incorporated a friction term into the Lagrange equation to account for the joint friction [6]. This term is proportional to the absolute value of the joint velocity and the sign of the velocity. This is because the friction force acts in the opposite direction to the joint velocity, and its magnitude is proportional to the normal force between the joint surfaces, which is proportional to the joint force and thus the joint velocity:

$$Ff = \begin{bmatrix} -\text{sign}(\dot{q}_1)\cdot m_3\cdot(l_1^2\cdot\sin(q_1) + l_2^2\cdot\sin(q_1+q_2) + l_3^2\cdot\sin(q_1+q_2+q_3))\cdot|\dot{q}_1| \\ -\text{sign}(\dot{q}_2)\cdot m_3\cdot(l_2^2\cdot\sin(q_1+q_2) + l_3^2\cdot\sin(q_1+q_2+q_3))\cdot|\dot{q}_2| \\ -\text{sign}(\dot{q}_3)\cdot m_3\cdot l_3^2\cdot\sin(q_1+q_2+q_3)\cdot|\dot{q}_3| \end{bmatrix}$$

The Lagrange equation can be written in the following form:

$$\ddot{q} = M^{-1}(q).(\tau - \dot{q}.C(q,\dot{q}) - G(q) - Ff(q,\dot{q}))$$

To validate the equation, we performed tests to ensure the integrity of the Inertia Matrix. Using the Cholesky factorization in Matlab, we confirmed that the Inertia Matrix is symmetric and positive definite.

Furthermore, we calculated the matrix G for various angular positions, specifically focusing on the fully extended configuration of the arm. It was observed that the gravitational matrix reached its maximum value when the arm was fully extended (i.e q1 = q2 = q3 = 0)

Additionally, we explored the inverse model by inverting the system's behavior, taking the joint positions as inputs and obtaining the corresponding torque outputs. This analysis aimed to explore the system's limits and gain insights into its performance characteristics.

Finally, we integrated the dynamical equation in Simulink, allowing the creation of a block that represents the robot manipulator arm. This block takes torque as input and provides the angular positions of the joints as output. By integrating the previously detailed kinematic equations, we can also obtain the Cartesian positions of the joints.

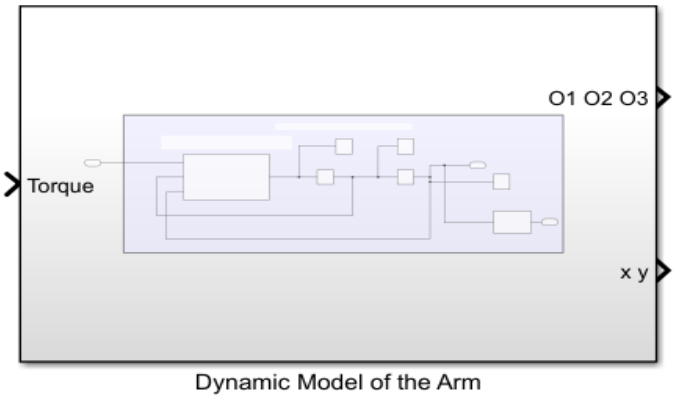


Figure 16: Robotic Arm Model

Please note that in Simulink, we used sometimes θ_1, θ_2, θ_3 to represent the angular position of the joints which we are noting q_1, q_2, q_3 throughout this presentation and in the matlab codes.

III.5 Torque Control of the Arm and Simulation

III.5.1 Linearization

To develop a control strategy for the robotic arm, it is essential to model it as a state space system. This involves linearizing the system to create a linear model that can be used for control design and analysis.

Linearization is the process of approximating the nonlinear dynamics of a system around an equilibrium point. It involves finding the linear relationship between the system's inputs, outputs, and states within a small region around the operating point.

We note:

$$U = \begin{pmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{pmatrix}$$

$$X = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix}$$

$$Y = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix}$$

To linearize the system, we must transform the dynamical model equation to multiple linear equations. To do that we pose that:

$$W = \frac{dq}{dt} = \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix}$$

.

Hence,

$$\frac{dq}{dt} = W = \begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

$$\frac{d\dot{W}}{dt} = M^{-1}(q).(U - C.W - G - Ff) = \begin{pmatrix} f_4 \\ f_5 \\ f_6 \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} X$$

Using the Jacobian matrix, we can derive the matrices A and B, which are essential components for formulating the state-space representation of the robotic system. The matrix A describes the dynamics of the system, while the matrix B represents the input-output relationship between the joint torques and the system states.

$$A = \begin{pmatrix} \frac{\delta f_1}{\delta q_1} & \cdots & \frac{\delta f_1}{\delta q_6} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_6}{\delta q_1} & \cdots & \frac{\delta f_6}{\delta q_6} \end{pmatrix}_{U_e, X_e}$$

$$B = \begin{pmatrix} \frac{\delta f_1}{\delta \tau_1} & \cdots & \frac{\delta f_1}{\delta \tau_3} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_6}{\delta \tau_1} & \cdots & \frac{\delta f_6}{\delta \tau_3} \end{pmatrix}_{U_e, X_e}$$

At the equilibrium point, the values of the inputs (torques) and states are denoted as U_e, X_e respectively. X_e is determined by solving the dynamical equation when it is set to zero, indicating a state of balance. On the other hand, U_e is obtained by solving the equation $U_e = G(q_0)$, , where q_0 represents the joint configuration at the equilibrium point. By substituting the equilibrium joint configuration into the gravitational matrix G(q), we can determine the torques U_e required to maintain the system in a state of equilibrium.

We finally have a linear state space system representing the non linear robot arm:

$$\dot{X} = AX + BU$$

$$Y = CX + DU$$

With:

$$A = \begin{pmatrix} \frac{\delta f_1}{\delta q_1} & \cdots & \frac{\delta f_1}{\delta q_6} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_6}{\delta q_1} & \cdots & \frac{\delta f_6}{\delta q_6} \end{pmatrix}_{U_e, X_e}$$

$$B = \begin{pmatrix} \frac{\delta f_1}{\delta \tau_1} & \cdots & \frac{\delta f_1}{\delta \tau_3} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_6}{\delta \tau_1} & \cdots & \frac{\delta f_6}{\delta \tau_3} \end{pmatrix}_{U_e, X_e}$$

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$D = \mathbf{0}_{3\times 3}$$

III.5.2 Control Design

With the state space model in place, we can design a robust control law to track desired angular positions and even Cartesian positions of the robot arm's gripper by using the kinematic model.

To achieve accurate trajectory tracking, we employ a Linear Quadratic Regulator (LQR) control strategy. LQR control is known for its robustness and ability to optimize system performance. By utilizing feedback techniques, the control law adjusts the torques applied to the joints to achieve the desired positions.

To simulate the system with the implemented control law, the MATLAB file `Linearization_SS_Control.m` must be executed. This file calls `Dynamical_Model_And_Validation.m`, which defines the robot arm's dynamical model. Then, it performs the previously discussed linearization process and incorporates the robust LQR control strategy.

When the execution is completed, we obtain the feedback control law:

$$U = -KX + LY_r$$

Here, U represents the applied torques, X denotes the state variables (Angular position and velocity), and Y_r corresponds to the reference trajectory (Angular or Cartesien). The control law is implemented in Simulink using the state space model.

By implementing the control law in Simulink (file `All_Simulations_.slx`), comprehensive system simulations can be performed. These simulations validate the effectiveness of the developed control strategy in achieving accurate position tracking and control of the robot arm.

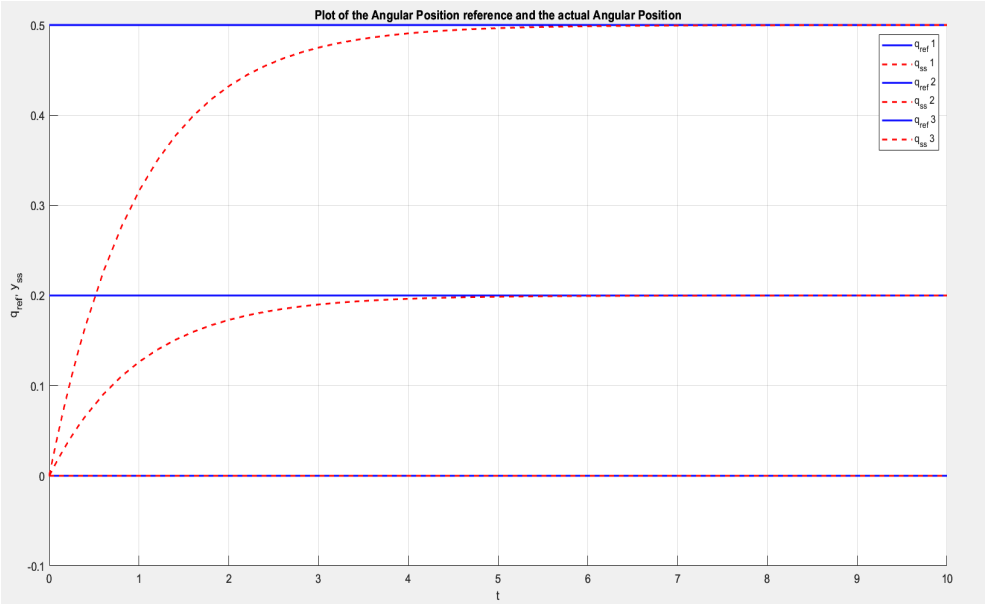


Figure 17: Angular Position of the Joints tracking their reference

The graph showcases the tracking performance of the robotic arm's joint positions in relation to their reference values. In this example, the joint position references are set as constants, as shown in the graph. The control performance is characterized by accurate tracking, with no static error observed. The system effectively follows the desired trajectory, resulting in precise control of the robot arm.

The objective is to control the non linear system since it is the closest to the real one. Hence, we applied the same control strategy to the block containing the dynamical model of the robot arm with no linearization or simplification. A few modification had to be made in the control law to achieve good and satisfactory reference tracking.

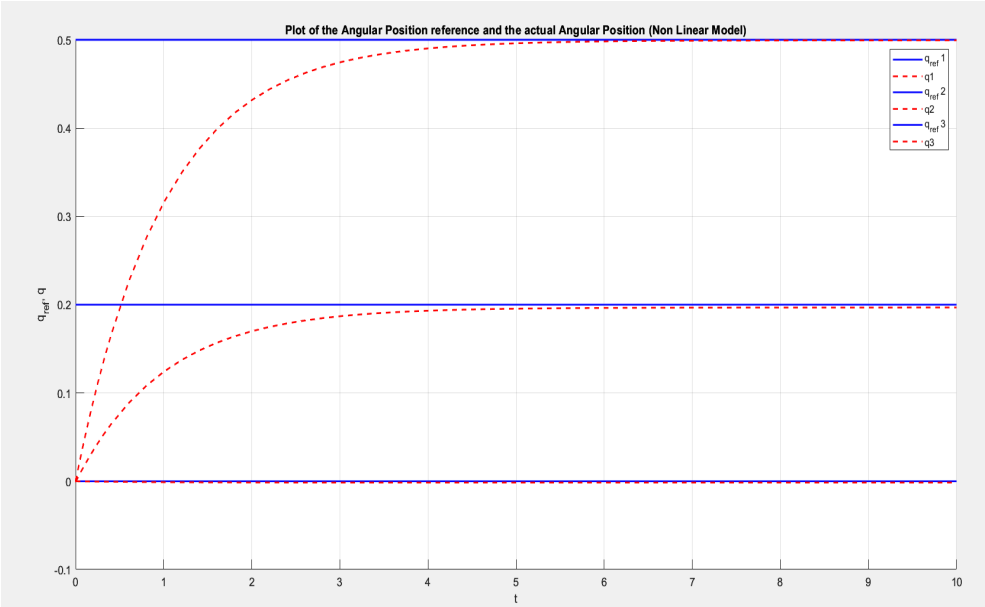


Figure 18: Angular Position of the Joints tracking their reference

The system demonstrates effective trajectory tracking, with only minimal static error. While the current control strategy performs adequately, there are opportunities for further improvement.

NB: Cartesian Position Tracking (Gripper)

In the Simulink file `All_Simulations_.slx`, you will find a block dedicated for the xy control.

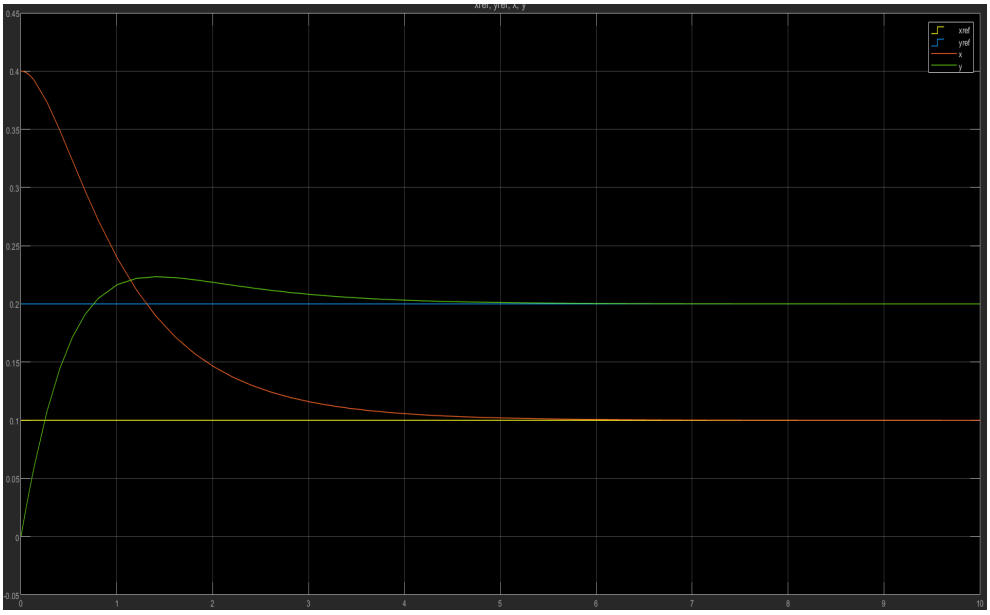


Figure 19: Angular Position of the Joints tracking their reference

The system exhibits effective trajectory tracking in Cartesian space. The desired trajectories for the end-effector position are accurately followed by the robotic arm. This indicates that the control strategy successfully translates the reference trajectories into appropriate joint configurations, allowing the robot to achieve the desired positions in Cartesian coordinates.

III.6 Trajectory planing

In relation to the work presented by the group previous to ours, the controllers provided by the manufacturer were used. However, the provided controller performs position control, i.e., it takes the robot's gripper to a certain

position with zero speed. In order to realize a trajectory, it was necessary to change the manufacturer's settings for the realization of the trajectory.

In our case, as we employ a distinct motor controller that operates based on torque, a unique approach becomes necessary. The trajectory planning involves two key steps: first, determining the desired throw position and velocity, and then generating the trajectory using a recursive algorithm.

III.6.1 Throw position and velocity

To establish the throw position, working in Cartesian coordinates proves to be more convenient. The movement of the object through the air is approximated as a projectile motion, assuming that the robot operates within a controlled environment with no wind and with low velocities.

When considering the projectile motion, multiple position and velocity combinations exist that can potentially accomplish the task, given a final position. Consequently, an optimization algorithm was proposed to identify the optimal solution for our specific problem. The objective of this optimization is to minimize the impact of the object when it reaches the target. To achieve this, the cost function employed in the optimization process is defined as the energy of the object at the moment of its release during the throw.

$$E = m \cdot g \cdot p_y + \frac{1}{2} m \cdot (v_x^2 + v_y^2) \tag{19}$$

Where p_x is the x coordinate of p, p_y is the y coordinate of p, and $v = \dot{p}$. Ignoring the constants, the cost function is:

$$J = g \cdot p_y + \frac{1}{2} (v_x^2 + v_y^2) \tag{20}$$

But, since the system is limited, we must provide the system's constraints on the optimization algorithm. The first constraint added is to limit the position of the throw. The throw is meant to be made using all of the joints, so the trajectory must be perform stating in the second quadrant of the Cartesian plane and finishing on the first quadrant, as shown in figure 20.

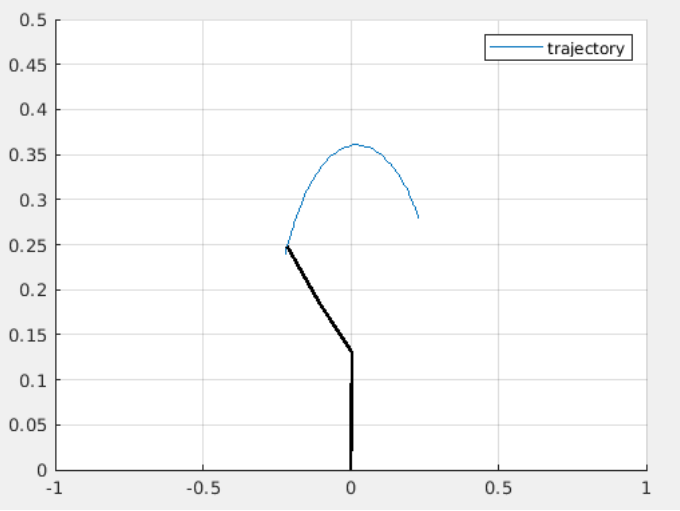


Figure 20: Trajectory of the throw

This implies that the throw position should be closer to the maximum reach of the robot. The second constraint is related to the maximum velocity of the thrown object. The ball's velocity has a lower bound of zero and an upper bound determined by the sum of the maximum linear velocities of each joint. The third constraint relates to the direction of the throw. Due to the chosen trajectory, the ball is thrown perpendicular to the robot's movement. Therefore, we have two types of constraints: inequality constraints and equality constraints. These constraints are formulated as follows:

$$\begin{bmatrix} -(p_x^2 + p_y^2) + L_{min}^2 \\ p_x^2 + p_y^2 - L_{max}^2 \\ -(v_x^2 + v_y^2) \\ v_x^2 + v_y^2 - V_{max}^2 \end{bmatrix} \leq 0 \rightarrow C < 0$$
$$\left[p_x \cdot v_x + p_y \cdot v_y\right] = 0 \rightarrow C_{eq} = 0 \tag{21}$$

The subsequent step involves utilizing the cost function and constraints within an optimization algorithm. As the problem encompasses non-linear optimization with non-linear constraints, MATLAB offers a suitable function called *fmincon* for solving such problems. Alternatively, if Python is chosen, the library *scilab.optimize* provides a *minimize* function that can be employed to address the same problem.

As already mentioned, the ball's trajectory is going to be considered as a projectile motion. Therefore, by using the dynamic equations it is possible to find the end time t_{end} and the v_x using the following equations.

$$t_{end} = \frac{v_y + \sqrt{v_y + 2 \cdot g \cdot p_y}}{g}$$
$$v_x = \frac{x_{end} - p_x}{t_{end}} \tag{22}$$

Using those equations, the optimization variable is defined as $x = \begin{bmatrix} p_x & p_y & v_y \end{bmatrix}^\top$, so the optimization problem is described as:

$$\underset{x \in \mathbb{R}^3}{minimize}\ J(x) \quad s.t. \quad \begin{cases} C(x) \leq 0 \\ C_{eq}(x) = 0 \end{cases} \tag{23}$$

III.6.2 Trajectory algorithm

After the optimization algorithm, the throw position is defined. Then we can use a trapezoidal velocity approach, as presented in figure 21. The control is executed in joint space, and due to the simplicity of the robot and the absence of obstacles along its path, generating the trajectory in joint space proves to be simpler and less computationally intensive, resulting in reduced processing time.

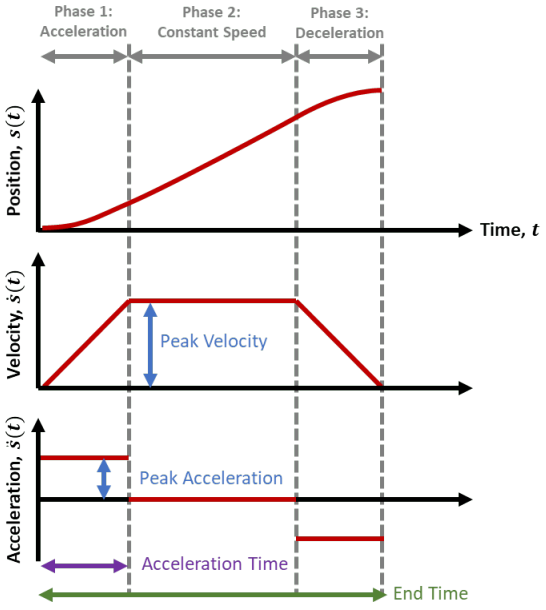


Figure 21: Trapezoidal profile

The initial step involves converting the throw position and velocity to joint space. To accomplish this, we require the throw position obtained through the optimization problem and the ϕ angle, which represents the orientation of the gripper as outlined in section III.2.2. The ϕ angle is simply defined as the angle of the p vector. Additionally, the angular velocity of the throw, ω_t, is calculated using the following equation:

$$\omega_t = \frac{\sqrt{v_x^2 + v_y^2}}{\sqrt{p_x^2 + p_y^2}} \tag{24}$$

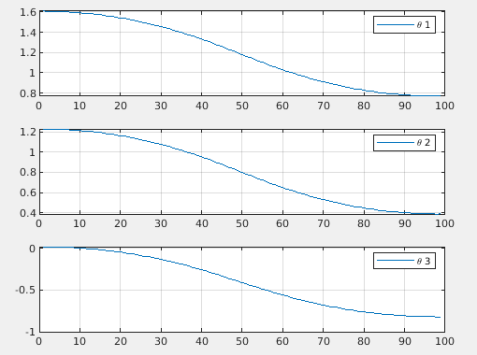
With that, it is just use the dynamic angular discrete motion equations defined as:

$$\begin{cases} \theta(k) = \theta(k-1) + \omega(k-1) \cdot t + \frac{\alpha}{2} Ts^2 \\ \omega(k) = \omega(k-1) + \alpha \cdot Ts \end{cases} \tag{25}$$

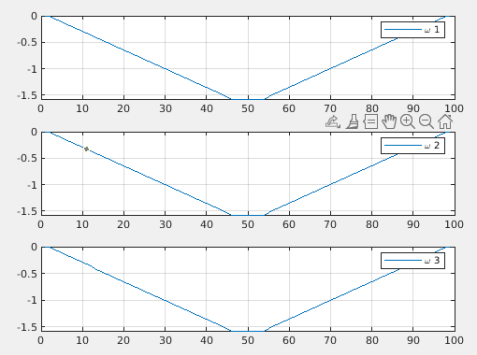
where Ts is the sample time of the discretization. Then the trajectory is created by the following algorithm:

1: $\theta_p \leftarrow inverse\_kinematic(p, \phi)$
2: $\omega \leftarrow \omega_t$
3: $\theta \leftarrow \theta_p$
4: **while** $\omega(first) \geq 0$ **do**
5: $\quad \theta \leftarrow [\theta(first) + \omega(first) * Ts + \frac{1}{2}\alpha \cdot Ts, \theta]$
6: $\quad \omega \leftarrow [\omega(first) + \alpha \cdot Ts, \omega]$
7: **end while**

This algorithm leads to a trajectory as seen in figure 20. The resulting trajectory has the same profile as shown in figure 21.



(a) Generated θ profile



(b) Generated ω profile

Figure 22: Motion profile of the throw

IV Simulation Environmnet

In the context of a remotely developed project, in order to test and validate algorithms it is necessary to have a reliable simulation environment. Moreover, utilizing a simulation environment prior to connecting algorithms to a physical arm can help prevent potential damages and increase the overall lifespan of the robot. The simulator chosen for this project was Gazebo, an open-source 3D robotics simulator known for its high-performance physics engine. One notable advantage of selecting Gazebo as the simulation environment is its large user/developer community, which provides valuable support and resources, making it easier to resolve any issues that may arise during simulation development.

IV.1 Robotic arm

Selecting an appropriate model to represent the robotic arm in the simulation is essential to guarantee that the results obtained when testing the developed control algorithms, mentioned in section III, are credible and reliable, but also to make it compatible with the physical arm. The second aspect is ensuring compatibility with the physical arm.

The selected model for representing the robotic arm in the simulation was provided by Robotis. It encompasses not only the physical aspects of the arm but also provides a wide range of functionalities. One key advantage of using the Robotis model is its compatibility with all the resources available for the real system. This includes subscribed topics and controllers, ensuring that the simulation environment replicates the behavior and functionalities of the physical arm accurately. By using the Robotis model, projects can be transitioned from simulation to the real system with minimal adjustments or modifications.

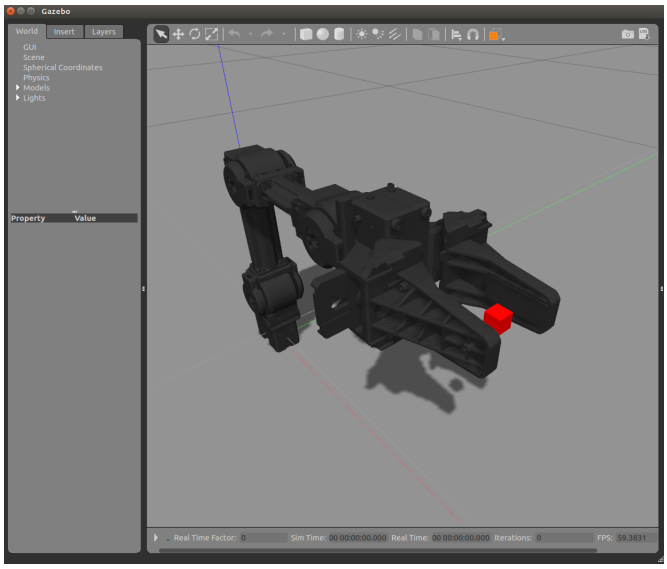


Figure 23: Robotis model

IV.2 Camera

The camera in the simulation is designed to be as simple as possible. It takes the form of a plain cube and it is equipped with a plugin that enables it to publish images on the topic */camera/image_raw*.

IV.3 Other objects

Two additional objects that are essential to the project are the table tennis ball and the target cup. These objects are defined by SDF (Simulation Description Format) models, allowing them to be easily spawned in the simulation at desired positions using a simple service. This eliminates the need to reinitialize the simulation after each throw. It also allows the user to define a new position for the cup for the next throw.

V System

In the simulated environment, throwing a ball involves multiple tasks and the exchange of information between various subsystems. To facilitate these communications, a range of different tools are utilized, depending on the specific characteristics of the data being sent and the subsystems involved.

The tasks related to throwing the ball in the simulation are divided into four main subsystems, each responsible for specific functions. These subsystems interact and exchange information to ensure the coordinated execution of the throwing process. Figure 24 illustrates the subsystems and the main communication exchanges between them.

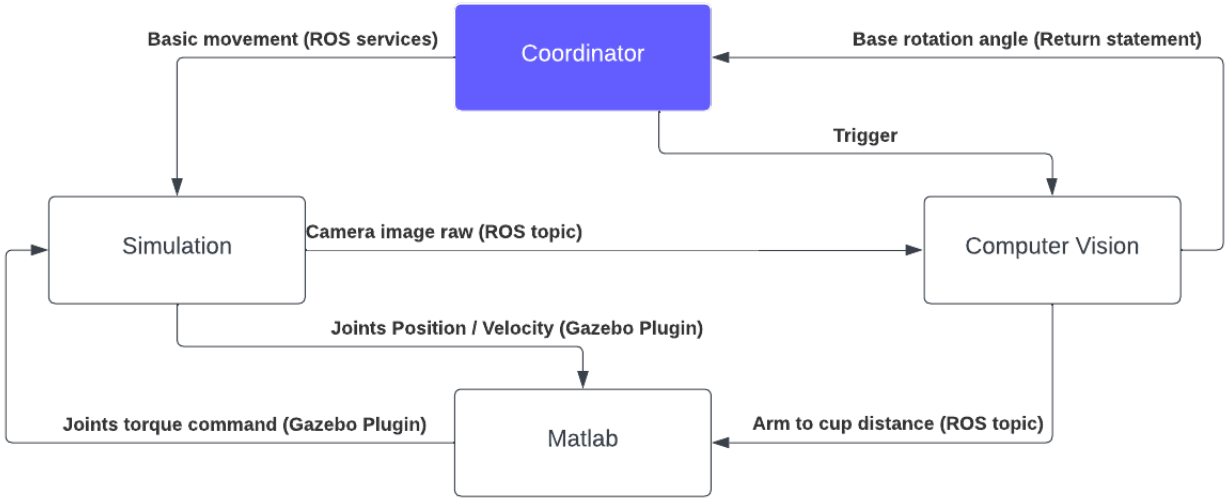


Figure 24: Subsystems

V.1 Robot Operating System (ROS) and Gazebo simulation

The choice of the arm simulation model had a direct impact in the choice of the ROS distribution used. The selected model was incompatible with ROS 2 distributions, leading to the adoption of Ubuntu 20.04 (Focal) and ROS Noetic Ninjemys, an older distribution. Although Noetic Ninjemys is expected to be deprecated sooner compared to ROS 2 distributions, it was preferred due to its richer availability of resources. ROS distributions have been in use for much longer with only minimal differences between them. However, the technological gap between ROS and ROS 2 made the adaptation of certain packages challenging, resulting in compatibility limitations. Migration to the latest ROS distribution would be recommended if all necessary packages become available for ROS 2 in the future. One notable consequence of this choice was the incompatibility of previous groups' work, particularly custom services used for arm movement, rendering them unusable.

V.2 Tasks and Subsystems

This section presents an overview of the subsystems involved in the project and their roles in achieving the overall project objectives. It is worth noting that for improved performance, one of the subsystems utilizing Matlab runs on a Linux operating system, while the remaining subsystems are installed in a Linux virtual machine.

Task	Parent subsystem	Exploitation system
Generate environment	Simulation	Linux
Identify cup and robot position	Computer vision	Linux
Prepare to throw	Coordinator	Linux
Calculate throw position and velocity	Matlab	Windows
Generate arm joints trajectory	Matlab	Windows
Send control instructions to simulated environment	Matlab	Windows
Simulate movement	Simulation	Linux
Prepare environment for new throw	Coordinator	Linux

Table 2: Organization of the tasks for each subsystem

V.2.1 Coordinator

The Coordinator subsystem serves as the main control unit within the project. It encompasses a loop that orchestrates all the steps required to perform a throw. In addition to its role as an organizer, the Coordinator subsystem performs two regular tasks: First one is preparing to throw by sending the instructions to perform the basic movements, such as picking up the ball, aligning the arm and the cup and going back to the initial position. The second one is preparing for a new throw. After each throw attempt, the Coordinator subsystem prompts the user if they would like to try another throw. If the answer is positive, the ball goes back to its initial position and the cup will be spawned on the coordinates indicated by the user.

V.2.2 Simulation

The simulation subsystem generates the simulated environment that interacts with all other subsystems. It models the physical behavior of the system and emulates the output of sensors that would be present in a real-life system. This includes providing simulated data for joint positions and camera streams. By simulating sensor outputs, the project can evaluate and validate the performance of algorithms and control strategies in a virtual environment.

The simulated environment consists of various components, including an empty background with gravity and no wind, the robotic arm, a camera, a ball, and a cup that can be moved around. ArUco markers are still to be fixed at the cup and the arm.

To address a known issue with the physics of grasping objects in the simulation, a plugin called the Gazebo Grasp Fix Plugin was installed. The plugin's details and usage instructions can be found in [4].

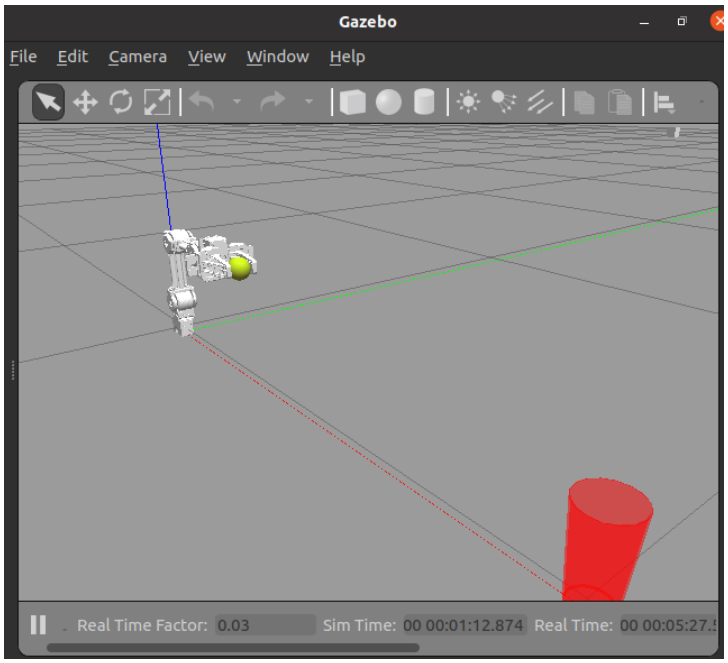


Figure 25: Gazebo simulated environment

V.2.3 Matlab

This subsystem includes all the Matlab and Simulink files. Those are responsible for the trajectory planning and control described in section III. Communication with the simulation subsystem is facilitated using specific blocks provided by the Robotics System Toolbox. Figure 26 illustrates the blocks comprising the "Gazebo interface", which enables torque commands to be sent to the simulated arm joints and retrieves position and velocity data from each joint. The "Pacer" component ensures synchronization of time between the Gazebo simulation environment and Simulink.

The communication with the Computer Vision subsystem is what triggers the trajectory generation and torque control. The data is received via ROS topic using functions provided by ROS toolbox.

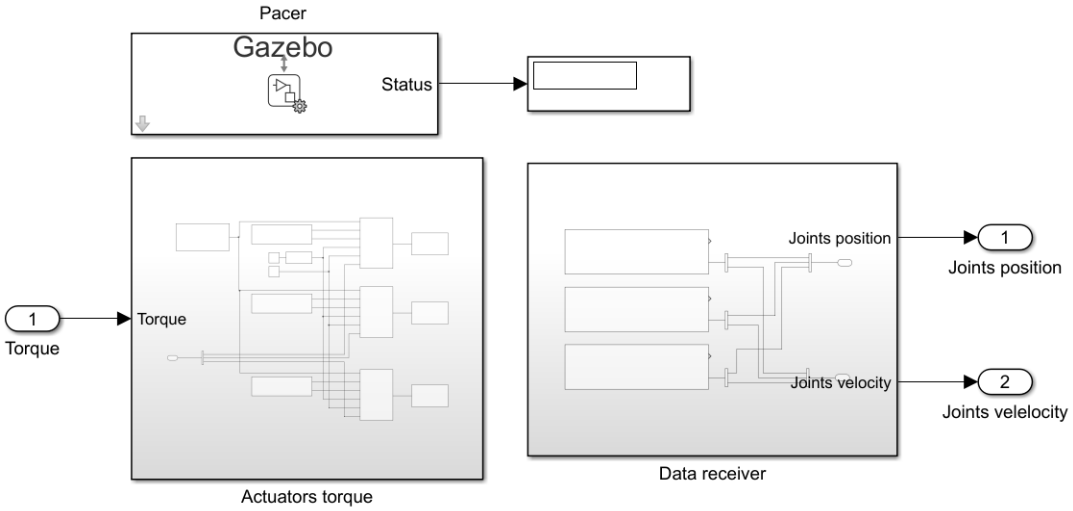


Figure 26: Gazebo interface

V.2.4 Computer Vision

This subsystem which encompasses all the programs involved in object detection and relative position calculation as described in Section II. An example image received from the simulation subsystem via a ROS topic is presented in Figure 27.

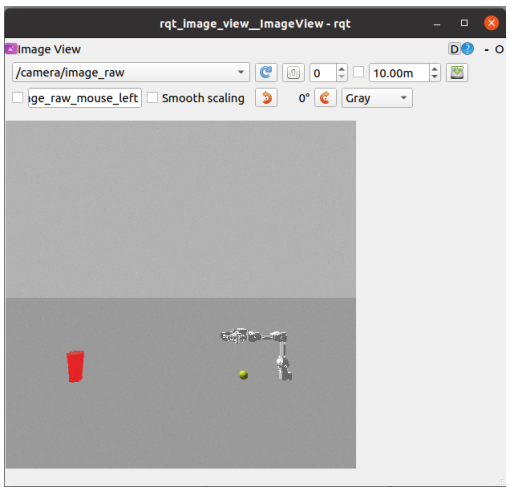


Figure 27: Published image topic

V.3 System Set Up and Use

To ensure easy access, all the necessary files are stored in a Gitlab group ([2]). Instructions on how to set up and run the project can be found on the README file in the Tossing-bot repository.

VI Results and Conclusion

For the Computer Vision 3-D task, a cup was used as the object of study. Additionally, we applied the same process to other objects such as building blocks. During testing, we found that at distances around 1 meter, the algorithm achieved distance errors within 5 mm and angle errors within 5 degrees for pose estimation. This result can be considered highly accurate. Considering that the OpenManipulatorX robot gripper has an opening range of 2-10 cm and a width of approximately 2 cm, it is fully capable of accurately grasping objects.

When it comes to the 2-D approach by Computer Vision, results have shown that the ArUco markers, when worked together with the Python OpenCV library, represent an excellent tool for measuring distances and angles in the real space with reasonable response time and accuracy. Although the results obtained by the YOLO approach have proven

to be worse, AI can still play a major role in this type of task. Some improvements that could be made regarding an AI-based implementation are improving the data base containing annotated data about the object to be identified, and also exploring newer approaches such as Lightweight OpenPose, a lightweight pose estimation model for Edge Machine Learning [3].

As for the control section, one suggestion for future work is to explore the implementation of a model predictive control (MPC) strategy for controlling the robot arm. MPC offers advantages such as the ability to incorporate constraints, allowing for more realistic and practical control. By imposing constraints on torque and position, the control strategy can better account for physical limitations of the system. Additionally, integrating an integrator component in the control system can help eliminate any residual static error. ensuring accurate trajectory tracking and potentially enabling better control on the real system. These advancements can contribute to improved overall system behavior and offer additional flexibility and robustness in controlling the robot arm.

For future projects it is recommended to validate that the model used in the control section is compatible with the one on the simulation. Due to computational power restrictions, we were not able to run the necessary tests to check the compatibility. One possibility is to try having Matlab subsystem running on a different computer to avoid overloading one machine.

Furthermore, given the functionality of the simulation, it becomes possible to utilize it for training a neural network to learn the dynamics of the throw and control the joints.

References

[1] **Detection of ArUco Markers**. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html. Accessed: 2023-06-04.

[2] **Project Gitlab**. https://gitlab-student.centralesupelec.fr/p19-arm. Accessed: 2023-06-06.

[3] **Project Gitlab**. OpenPose. Accessed: 2023-06-06.

[4] **The Gazebo grasp fix plugin**. https://github.com/JenniferBuehler/gazebo-pkgs/wiki/The-Gazebo-grasp-fix-plugin. Accessed: 2023-06-06.

[5] **YOLO: Real-Time Object Detection**. https://pjreddie.com/darknet/yolo/. Accessed: 2023-05-30.

[6] Aderajew Ashagrie, Ayodeji Olalekan Salau, and Tilahun Weldcherkos. **Modeling and control of a 3-DOF articulated robotic manipulator using self-tuning fuzzy sliding mode controller**. *Cogent Engineering*, 8(1):1950105, 2021.

[7] Zhangxi Zhou, Yuyao Zhang, and Yezhang Li. **Model Predictive Control Design of a 3-DOF Robot Arm Based on Recognition of Spatial Coordinates**, 2022.