National Technical University of Athens
MSc in Control Systems & Robotics

Robotics Laboratory

# Kinematic Analysis, Design, and Simulation of the XArm7 7-DOF Robotic Manipulator in ROS for Path Following

Ronaldo Tsela

August 6, 2025

## Abstract

This report presents the kinematic analysis and modeling of the xArm redundant robotic manipulator with 7 degrees of freedom for the task of path following. We implement and evaluate two path-following algorithms: one for absolute 3-DOF position control using the inverse kinematic model, and another for full control-both orientation and position-using the inverse differential kinematic approach. Both algorithms are then ported in ROS 1 Noetic, implemented in Python, and evaluated through simulations using the xArm7 robotic manipulator simulation model provided in the Gazebo environment. This project introduced us to the ROS framework for robot manipulator-related designs.

**Keywords:** xArm7, robot, ROS, path following, waypoints, algorithm, position, orientation, control

# Contents

# 1 Objectives

The objective of this report is to study, analyze, and model the xArm 7 collaborative robotic manipulator, with a particular focus on path-following tasks. Following the theoretical analysis, two path-following control schemes are implemented in Python using ROS 1 and evaluated in the Gazebo simulation environment. The results obtained from the simulations are analyzed and discussed to assess the effectiveness of the proposed approaches, as well as their practical use.

# 2 Introduction

## 2.1 Cobots

We are living in an era where industrial automation is revolutionizing industries around the world by accelerating processes, performing tasks that most humans are physically unable to do, and optimizing production in terms of both speed and quality. The key feature of this industrial automation of-curse are autonomous machining tools and robots. Specifically, industrial robots are autonomous machines designed to perform tasks without direct human intervention. They are commonly used for repetitive, high-precision, or/ and hazardous tasks that would be difficult or dangerous for humans to handle [6, 3]. You can find these robots in production lines, such as in car manufacturing, packaging processes, and the management of heavy equipment. Common applications of industrial robots include welding, painting, material handling, and packaging, among many other applications and across multiple industrial domains [6].

Industrial robotic manipulators are typically large, fast, and highly powerful machines with a high payload capacity, a wide range of motion and capable of produce high forces and torques. Due to their size and capabilities, they are often enclosed in safety cages or barriers to protect human workers from potential hazards when they need to co-exist in the same environment [6].

Although robotic automation is widespread, and many industries benefit from automating several tasks and processes replacing human effort, the human factor remains essential in industrial settings. There are certain tasks that robots cannot perform independently, and thus requiring human collaboration [6]. This new trend of robot-human collaboration is commonly known as "human-in-the-loop".

This is where cobots come into play. The term "cobot" stands for "collaborative robot" [3]. These type of robots are specifically designed to work alongside humans in a shared workspace. Cobots are built with features that prioritize safety, flexibility, and ease of use as opposed to the classical large and "rigid" industrial robotic manipulators [6, 3]. They are equipped with force and torque sensors that enable them to detect and respond to contact with humans or objects, ensuring a safe working environment for them. Their usually lightweight and compact design allows for easy movement and reconfiguration, providing greater adaptability for a variety of tasks compared to large industrial robots. Common applications of cobots include assembly, pick-and-place, machine tending, quality inspection and of-curse in medical applications (e.g. The da Vinci Surgical System) [6].

Cobots are generally more affordable than industrial robots, with lower upfront costs. Their ease of programming and integration, combined with their flexibility and adaptability, helps reduce setup and training expenses. Additionally, cobots often eliminate the need for costly safety infrastructure, such as cages or barriers, which further lowers overall costs. These advantages make cobots a popular choice in the academic community for research and educational purposes [6].

## 2.2 Robot Operating System

The Robot Operating System (ROS) is an open-source framework for building and controlling robots. Despite its name, ROS is not an actual operating system but rather a collection of software libraries and tools that help developers create complex robot applications [2].

ROS is widely used in academia, research, and industry for developing robots ranging from mobile robots and manipulators to drones and autonomous vehicles. It facilitates rapid prototyping, algorithm testing, and integration, making it invaluable for robotics projects of all sizes. In settings where access to actual hardware is limited, ROS plays a crucial role in learning robotics concepts and testing algorithms in a simulated environment [2]. This flexibility makes it an excellent choice for both beginners and professionals.

ROS offers a wide range of features for robotic development, but its most important capability lies in its simulation environment extensions. Simulation environments are essential in robotics because testing on real hardware can be expensive, time-consuming, hazardous or even impossible in certain occasions. A characteristic simulator in ROS is the Gazebo, a free, open-source 3D physics-based tool maintained by Open Robotics. Gazebo enables developers to create virtual "worlds" containing robot models and obstacles, where sensors and actuators behave as they would in reality (publishing data on ROS topics, applying friction, etc.) [2]. This tool allow an entire robot, including its mechanic features, various sensor models, and the environment around it to be modeled and debugged entirely in software.

Simulations are particularly important for collaborative robots and human–robot interaction scenarios. Virtual "digital twins" of humans and robots can be integrated into a Gazebo scene to evaluate shared tasks, such as object handovers or navigation in a common workspace, under realistic conditions. For example, Araiza-Illan et al. developed a ROS-Gazebo testbench for human–robot object-handover scenarios , systematically simulating human interactions, environmental conditions, and sensor responses to validate the robot's control software [2]. Generally, simulation-based testing offers a fast, structured approach to exploring complex human–robot interactions within rich environment models. In practice, this allows developers to iterate cobot behaviors and safety protocols virtually—optimizing workflows and verifying safety measures before conducting physical trials.

## 2.3 The xArm 7 Cobot Kinematic Model

One such cobot system is the xArm 7 manipulator which is depicted in Fig. 1. It is a small design capable of manipulating a payload of 3.5 kg and with a workspace radius of 70 cm from its base. The end-effector can move with a maximum speed of 1 m/s. It features seven revolute joints which means a total of seven degrees of freedom making the manipulator a redundant type.

Currently, the xArm 7 is used not only in academic and research institutions but also in industrial settings in various production lines. One of the key reasons for its growing popularity—aside from its relatively low cost—is its compatibility with the widely adopted aforementioned ROS, a powerful software framework for developing robotic applications.

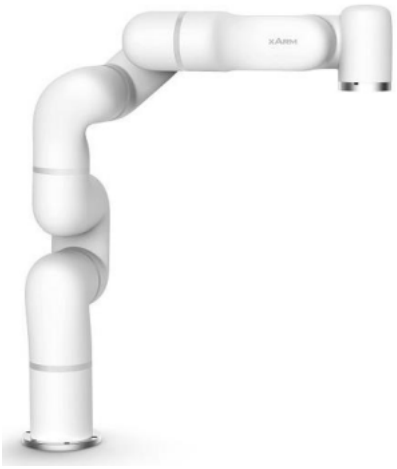


Figure 1: The xArm 7 cobot.

The geometric characteristics of the 7-DOF xArm7 robot manipulator are shown in Fig. 2, including the length of the respective links and placement of the reference frames according to the modified Denavit–Hartenberg (DH) convention. From this setup, we construct Table 1, which lists the DH parameters and subsequently used for deriving the homogeneous transformation matrices between successive links.

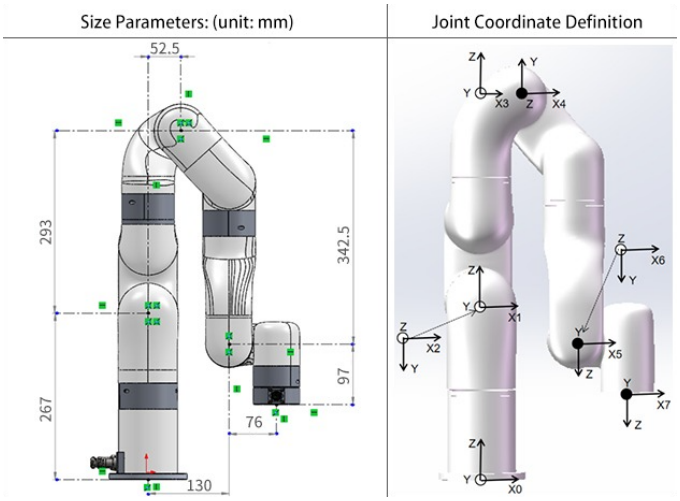



Figure 2: UFACTORY xArm7 robot manipulator. The reference frames are placed according to the alternative DH approach.

Table 1: Modified DH parameters for the UFACTORY xArm7 robot manipulator.

| Link $i$ | $\theta_i(rad)$ | $d_i(m)$ | $a_{i-1}(m)$ | $\alpha_{i-1}(rad)$ |
|---|---|---|---|---|
| 1 | $q_1$ | $l_1 = 0.267$ | 0 | 0 |
| 2 | $q_2$ | 0 | 0 | $-\pi/2$ |
| 3 | $q_3$ | $l_2 = 0.293$ | 0 | $\pi/2$ |
| 4 | $q_4$ | 0 | $l_3 = 0.0525$ | $\pi/2$ |
| 5 | $q_5$ | $l_4 \cos(\theta_1) = 0.343$ | $l_4 \sin(\theta_1) = 0.775$ | $\pi/2$ |
| 6 | $q_6$ | 0 | 0 | $\pi/2$ |
| 7 | $q_7$ | $l_5 \cos(\theta_2) = 0.097$ | $l_5 \sin(\theta_2) = 0.076$ | $-\pi/2$ |

Here, $\alpha_i$ and $a_i$ represent the twist angle and link length between $z_i$ and $z_{i+1}$, respectively, while $\theta_i$ and $d_i$ represent the joint angle and link offset, which are the joint variables for the revolute joints respectively around $Z_i$ and along $X_{i-1}$ to $X_i$ respectively, similarly to the standard DH form.

According to the modified DH convention [8], each successive transformation from link $i$ to link $i+1$ is defined by a homogeneous transformation matrix that depends on the joint displacement or rotation $q_i$ that guides the respective link, as follows:

$$A^i_{i+1}(q_i) = \text{Rot}(x, \alpha_i) \cdot \text{Tra}(x, a_i) \cdot \text{Rot}(z, \theta_i) \cdot \text{Tra}(z, d_i) \tag{1}$$

Based on Eq. 1 and the parameters listed in Table 1, we can compute the sequence of homogeneous transformation matrices $A^i_{i+1}$ for each link of the manipulator. Each transformation describes the pose of frame $i+1$ relative to frame $i$, following Craig's modified DH convention. Specifically, we obtain:

$$\begin{aligned}
A^0_1(q_1) &= Rot(x, q_1) \cdot Tra(z, l_1) \\
A^1_2(q_2) &= Rot(x, -\pi/2) \cdot Rot(z, q_2) \\
A^2_3(q_3) &= Rot(x, \pi/2) \cdot Rot(z, q_3) \cdot Tra(z, l_2) \\
A^3_4(q_4) &= Rot(x, \pi/2) \cdot Tra(x, l_3) \cdot Rot(z, q_4) \\
A^4_5(q_5) &= Rot(x, \pi/2) \cdot Tra(x, l_4 sin(\theta_1)) \cdot Rot(z, q_5) \cdot Tra(z, l_4 cos(\theta_1)) \\
A^5_6(q_6) &= Rot(x, \pi/2) \cdot Rot(z, q_6) \\
A^6_7(q_6) &= Rot(x, -\pi/2) \cdot Tra(x, l_5 sin(\theta_2)) \cdot Rot(z, q_7) \cdot Tra(z, l_5 cos(\theta_2))
\end{aligned}$$

and after simplifying the above equations we get the homogeneous transform matrices as follows:

$$A^0_1(q_1) = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^1_2(q_2) = \begin{bmatrix} c_2 & -s_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s_2 & -c_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^2_3(q_3) = \begin{bmatrix} c_3 & -s_3 & 0 & 0 \\ 0 & 0 & -1 & -l_2 \\ s_3 & c_3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A^3_4(q_4) = \begin{bmatrix} c_4 & -s_4 & 0 & l_3 \\ 0 & 0 & -1 & 0 \\ s_4 & c_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^4_5(q_5) = \begin{bmatrix} c_5 & -s_5 & 0 & l_4 \cdot s_{\theta_1} \\ 0 & 0 & -1 & -l_4 \cdot c_{\theta_1} \\ s_5 & c_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^5_6(q_6) = \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s_6 & c_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_7^6(q_7) = \begin{bmatrix} c_7 & -s_7 & 0 & l_5 \cdot s_{\theta_2} \\ 0 & 0 & 1 & l_5 \cdot c_{\theta_2} \\ -s_7 & -c_7 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where $s_i \equiv \sin(q_i)$, $c_i \equiv \cos(q_i)$, $s_{\theta_k} \equiv \sin(\theta_k)$, and $c_{\theta_k} \equiv \cos(\theta_k)$, with $i = 1, 2, \ldots, 7$ and $k = 1, 2$.

The forward kinematic model of the manipulator is derived from the sequential composition of homogeneous transformations, as expressed in Eq. 2. The complete form of this equation yields the pose (position and orientation) of the end-effector with respect to the base frame, given the joint variables $q_i$ where $i = 1, 2 \ldots, 7$.

$$T_{ee}^0(\boldsymbol{Q}) = \prod_{i=0}^{6} A_{i+1}^i(q_{i+1}), \tag{2}$$

where $\boldsymbol{Q} = [q_1, q_2, \ldots, q_7]$ the joint configuration vector that describes the robot manipulator. For the sake of readability, the full expression of the forward kinematics equation as a result of the computation presented by eq. (2) is omitted.

## 2.4 Path Following System

Robot manipulators can be programmed to perform a wide range of tasks. However the fundamental requirement for every robotic manipulator is the ability to move from an initial point to a designated final point. The transition between these points must be however governed by motion laws that control the robot's joints, enabling smooth and precise movements while respecting various constraints. These constraints may be inherent to the robot's physical structure, electronics, and mechanical components, or related to its operational environment and workspace layout. Additionally, user-defined requirements—such as limits on velocity, orientation, torque, and other parameters—must also be considered. This process is commonly known as trajectory following. The geometric description of the motion is referred as path and thus the task of following a locus of points in space is referred to as path-following [7, 1].

In this project, the objective, as previously stated, is to design a path-following scheme for the xArm 7 manipulator. Specifically, we aim to implement two control schemes: a simple 3-DOF position control of the end-effector, and a full kinematic control. However before delving into the design of these schemes, it is essential to define the system architecture that needs to be implemented.

We refer to the system that enables a robot manipulator to perform a constrained and controlled transition from one point to another as the Path Following System (PFS). A PFS is composed of two main components: the trajectory planning unit and the motion control unit, as illustrated in Fig. 3. The PFS interfaces directly with the robot system, which, at a high level of abstraction, consists of the robot device driver and the physical robotic device. All components in the system exchange data, as represented by the arrows in Fig. 3. Before presenting the design of the PFS for this project, it is important to first understand the structure and functionality of each unit in the system.

The process begins with user-specified trajectory requirements. Typically, the user defines parameters that describe the desired trajectory, such as the end-effector's position and orientation. Then the trajectory

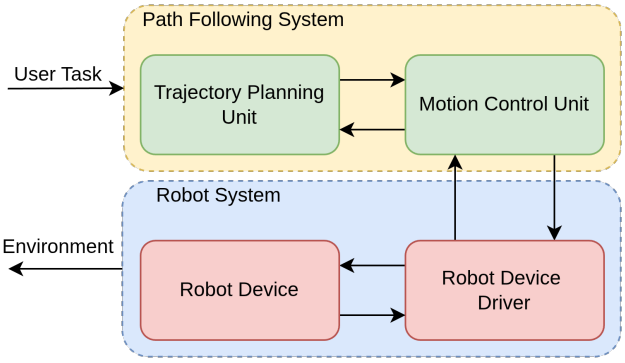



Figure 3: The path following system and the robot system modules.

planning unit generates a time-sequenced set of desired poses using interpolation methods that conform to these task requirements for each sampling time step. Here a point-to-point scheme is utilized, where the end-effector is requested to perform a motion between two points $A$ and $B$. Specifically, in point-to-point motion, the manipulator has to move from the initial to the final joint configuration in a given time $t_f$ through a constrained motion.

At this point we assumed that the trajectory planning involves transferring the end-effector between two given points $A$ and $B$. However, it is important to note that these points do not correspond to actual physical locations in space (not yet). Instead, they represent numerical values that can describe either a single Cartesian coordinate for the end-effector's position or an Euler angle for its orientation.

Formalizing the input to the trajectory planning system, by introducing the concept of a pose for the end-effector. A pose encapsulates both position and orientation information. Specifically a given pose $P_k$, where $k = 1, 2 \ldots, N$ for a series of $N$ poses that describe the entire task, can be expressed as follows:

$$P_k = \begin{bmatrix} r_k \\ \phi_k \end{bmatrix} \tag{3}$$

where:

$r_k = [x_k, y_k, z_k]^T$ represents the position of the end-effector in the Cartesian space.

$\phi_k = [\alpha_k, \beta_k, \gamma_k]^T$ represens the orientation in terms of Euler angles (roll, pitch, yaw).

Therefore for each of the Cartesian coordinates and the Euler angles, the trajectory planning process generates an appropriate path using either linear or cubic interpolation, as previously discussed.

Thus from the initial pose $P_k$ to the final pose $P_{k+1}$, the trajectory planning algorithm generates a series of intermediate pose vectors.

Once the desired trajectory is defined and these pose vectors are generated, they are fed into the motion control unit. Then the motion control unit generates a corresponding motion plan. This motion plan

consists of a series of control commands that are fed into the robot's controller to produce the desired motion. Specifically, the motion control unit takes the generated trajectory data from before and, using the current state of the robot and an appropriate kinematic model, computes a sequence of control commands. Essentially, the motion control unit is responsible for converting commands from operational space (Cartesian coordinates and orientations) into joint space (joint angles or displacements).

Keep in mind that these generated commands are not low-level control signals for the actuators, but rather higher-level abstractions of desired joint displacement. The robot's device driver, or controller, then translates these commands into actionable signals that subsequently drive the joint actuators.

As previously mentioned our objective is to design the PFS, which involves the development of both the trajectory planning and motion control units. Various approaches can be employed for designing each of these components, depending on task complexity, computational constraints, and accuracy requirements. Also the selected method often depends on the robot's configuration itself, workspace limitations, and required motion continuity.

In this project, we explore interpolation methods such as linear and cubic interpolation for trajectory planning. For the motion control unit, we focus on kinematic modeling using both inverse forward kinematics (for 3-DOF position control) and inverse differential kinematics (for full 7-DOF position and orientation control).

In the following sections, we present the design, implementation, and evaluation of each approach.

# 3 3-DOF Position Control

The first task consists of simply driving the robot manipulator from point $A$ to point $B$ following a straight line and actuating only joints $q_1, q_2$ and $q_4$ for end-effector position control only. The trajectory planning here uses a simple linear interpolation scheme while the motion control unit uses the inverse forward kinematics model of the robot manipulator but with only 3 actuated joints.

## 3.1 Trajectory Planning Unit

Starting from the linear interpolation scheme. Assuming the constraint that the motion of the end-effector from point $A$ to point $B$ implies linear motion, therefore the end effector must move along a straight line connecting these two points. To achieve this, we must generate a set of $n$ points that lie evenly on the line segment between $A$ and $B$-and containing $A$ and $B$ for smooth passing.

Assume the total time required to traverse the segment from point $A$ to point $B$ is $t_f$ and the sampling time between each generated point is constant and equal to $dt$. There is a relationship between the number of generated points, the total time and the sampling time given by:

$$t_f = (n-1) \cdot dt$$

The points must be evenly spaced along the linear path. Thus the time vector is a discrete representation as follows:

$$t(i) = i \cdot dt$$

for $i = 0, 1, \ldots, n-1$. Then the position of each interpolated point with the linear interpolation can then be calculated using the simple line formula:

$$x_e\big(i\big) = \alpha \cdot t(i) + b$$

Assuming that at the time instance $i = 0$, the end effector is at point $A$, and at the time instance $i = n-1$, it is at point $B$. Therefore, utilizing these boundary conditions, we derive the simple formula for linear interpolation as follows:

$$x_e(i) = (B - A)\frac{i}{n-1} + A \tag{4}$$

From the given two poses $P_A, P_B \in \mathbb{R}^6$ in this first task we only care about the position, which is the first 3 elements of these vectors. For each of the elements of the vector we employ eq. 4 as presented above and produce for each time instance $i$ a new intermediate value. The series of $k = 0, 1 \ldots, n-1$ interpolated $P_k \in \mathbb{R}^3$ vectors are the result of this trajectory planning scheme. These values are subsequently fed to the motion control unit.

### 3.2 Motion Control Unit

In this first approach, the motion control unit is fed with the set of pose vectors that describe the target trajectory in operation space. The goal here is to produce the joint positions required to reach each of these specific configuration such that the end-effector accurately follows the generated trajectory. This method involves mapping each pose from the trajectory to actual joint displacements by employing the inverse kinematic model as presented in the following relationship:

$$\boldsymbol{Q} \leftarrow f_K^{-1}(P)$$

where $\boldsymbol{Q} = \{q_1, q_2, \ldots, q_7\}$ the joint generalized displacement variables, $f_K^{-1}(.)$ is the inverse forward kinematics function and $P$ the target pose (here only position).

This approach is particularly suitable for tasks where precise positioning is crucial and where the inverse kinematics can be computed efficiently. It is commonly used in applications requiring repetitive or predictable motions, where computational simplicity and fast response are prioritized. The only drawback of this method is that it requires the analytical closed-form solution of the inverse forward kinematic problem, which in certain cases is very difficult to obtain and computationally expensive. For that reason in this first task we reduced the degrees of freedom of the robot manipulator to only 3 and targeted only the position control of the end-effector.

The actuated joints here are described by the generalized displacements $q_1$, $q_2$, and $q_4$, each of which respectively produce the required position motions for the end-effector. The rest of joint configuration is assumed to be set to constant values $q_3 = q_5 = q_7 = 0$ and $q_6 = 0.75$ rad. The next step is to compute the inverse kinematics for the reduced 3-DOF configuration.

Using eq.2 we can easily derive the relationship between the end-effector Cartesian position coordinates $P_x, P_y, P_z$ as function of the joint displacement:

$$P_x = \big(l_2 sin(q_2) + l_3 cos(q_2) + l_4 sin(q_4 - q_2 + \theta_1) + 0.73 l_5 sin(q_4 - q_2 + \theta_2) - 0.682 l_5 cos(q_4 - q_2 + \theta_2)\big) cos(q_1)$$
$$P_y = \big(l_2 sin(q_2) + l_3 cos(q_2) + l_4 sin(q_4 - q_2 + \theta_1) + 0.73 l_5 sin(q_4 - q_2 + \theta_2) - 0.682 l_5 cos(q_4 - q_2 + \theta_2)\big) sin(q_1)$$
$$P_z = l_1 + l_2 cos(q_2) - l_3 sin(q_2) - l_4 cos(q_4 - q_2 + \theta_1) - 0.682 l_5 sin(q_4 - q_2 + \theta_2) - 0.73 l_5 cos(q_4 - q_2 + \theta_2)$$

The equations from the forward kinematics that define the position of the end effector with respect the base frame of reference as provided above simplify to:

$$\begin{aligned} \frac{P_x}{c_1} &= s_2(l_2 - k_1 c_4 + k_2 s_4) + c_2(l_3 + k_1 s_4 + k_2 c_4) \\ \frac{P_y}{s_1} &= s_2(l_2 - k_1 c_4 + k_2 s_4) + c_2(l_3 + k_1 s_4 + k_2 c_4) \\ P_z - l_1 &= c_2(l_2 - k_1 c_4 + k_2 s_4) - s_2(l_3 + k_1 s_4 + k_2 c_4) \end{aligned}$$

or equally to:

$$\begin{bmatrix} \frac{P_x}{c_1} \\ \frac{P_y}{s_1} \\ P_z - l_1 \end{bmatrix} = \begin{bmatrix} l_2 - k_1 c_4 + k_2 s_4 & l_3 + k_1 s_4 + k_2 c_4 \\ l_2 - k_1 c_4 + k_2 s_4 & l_3 + k_1 s_4 + k_2 c_4 \\ -(l_3 + k_1 s_4 + k_2 c_4) & l_2 - k_1 c_4 + k_2 s_4 \end{bmatrix} \begin{bmatrix} s_2 \\ c_2 \end{bmatrix}$$

where,

$$k_1 = l_4 cos(\theta_1) + 0.681 l_5 sin(\theta_2) + 0.73 l_5 cos(\theta_2) = l_4 cos(\theta_1) + l_5 sin(\theta_2 + 0.82)$$
$$k_2 = l_4 sin(\theta_1) - 0.681 l_5 cos(\theta_2) + 0.73 l_5 sin(\theta_2) = l_4 sin(\theta_1) + l_5 sin(\theta_2 - 0.75)$$

Set:

$$l_2 - k_1 c_4 + k_2 s_4 = u_1$$
$$l_3 + k_1 s_4 + k_2 c_4 = u_2$$

then we get:

$$\begin{bmatrix} \frac{P_x}{c_1} \\ \frac{P_y}{s_1} \\ P_z - l_1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \\ u_1 & u_2 \\ -u_2 & u_1 \end{bmatrix} \begin{bmatrix} s_2 \\ c_2 \end{bmatrix}$$

Add $(\frac{P_x}{c_1})^2 + (P_z - l_1)^2$:

$$(\frac{P_x}{c_1})^2 + (P_z - l_1)^2 = u_1^2 + u_2^2 = l_2^2 + l_3^2 + k_1^2 + k_2^2 + 2c_4(l_3 k_2 - l_2 k_1) + 2s_4(l_3 k_1 + l_3 k_2)$$

Set $t_1 = l_2 k_2 + l_3 k_1$ and $t_2 = l_3 k_2 - l_2 k_1$. Then we get a linear combination of $sin(q_4)$ and $cos(q_4)$ which can be simplified by the following trigonometric identity:

$$Rsin(\omega + \phi) = Asin(\omega) + Bcos(\omega)$$
$$R = \sqrt{(A^2 + B^2)} \quad \phi = tan^{-1}(B/A)$$

Now set $\frac{1}{2}\left((\frac{P_x}{c_1})^2 + (P_z - l_1)^2 - l_2^2 - l_3^2 - k_1^2 - k_2^2\right) = L$, and then we get the aforementioned form:

$$L = t_1 sin(q_4) + t_2 cos(q_4)$$

where,

$$A = t_1 \quad B = t_2$$
$$R = \sqrt{t_1^2 + t_2^2} \quad \phi = tan^{-1}\left(\frac{t_2}{t_1}\right)$$

therefore:

$$L = \left(\sqrt{t_1^2 + t_2^2}\right) sin(q_4 + \phi)$$

which we solve for $sin(q_4 + \phi)$ and get:

$$sin(q_4 + \phi) = \frac{L}{\sqrt{t_1^2 + t_2^2}}$$

and,

$$cos(q_4 + \phi) = \pm\sqrt{1 - sin^2(q_4 + \phi)}$$

thus,

$$q_4 = atan2\Big(sin(q_4 + \phi), cos(q_4 + \phi)\Big) - tan^{-1}\left(\frac{t_2}{t_1}\right)$$

Using the $q_4$ calculated above we can then find $u_1, u_2$, where:

$$\begin{bmatrix} \frac{P_x}{c_1} \\ P_z - l_1 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \\ -u_2 & u_1 \end{bmatrix} \begin{bmatrix} sin(q_2) \\ cos(q_2) \end{bmatrix}$$

where,

$$\begin{bmatrix} sin(q_2) \\ cos(q_2) \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \\ -u_2 & u_1 \end{bmatrix}^{-1} \begin{bmatrix} \frac{P_x}{c_1} \\ P_z - l_1 \end{bmatrix}$$

We compute the inverted matrix:

$$\begin{bmatrix} u_1 & u_2 \\ -u_2 & u_1 \end{bmatrix} = \frac{1}{u_1^2 + u_2^2} \begin{bmatrix} u_1 & -u_2 \\ u_2 & u_1 \end{bmatrix}$$

Then we can find $sin(q_2)$ and $cos(q_2)$ respectively:

$$sin(q_2) = \frac{u_1 \frac{P_x}{c_1} - u_2(P_z - l_1)}{u_1^2 + u_2^2}$$
$$cos(q_2) = \frac{u_2 \frac{P_x}{c_1} + u_1(P_z - l_1)}{u_1^2 + u_2^2}$$

from where we finally get:

$$q_2 = atan2(sin(q_2), cos(q_2))$$

Finally, it is obvious that $q_1$ is computed as follows:

$$q_1 = atan2(P_y, P_x)$$

Note that there are multiple equally viable solutions for $q_1, q_2$ and $q_4$ respectively and not a single one.

### 3.3 Implementation

The user specifies the desired positions $P_A$ and $P_B$ in terms of the end-effector coordinates: $P_{Ax}, P_{Ay}, P_{Az}$, and $P_{Bx}, P_{By}, P_{Bz}$ respectively. Using a linear interpolation scheme, as previously described, $n-2$ intermediate points are generated for each coordinate axis, assuming that the first point corresponds to $P_A$ and the last to $P_B$. This process results in a sequence of $n$ evenly spaced positions along the straight-line path from $P_A$ to $P_B$. The procedure can be summarized by the following algorithm.

**Algorithm 1** Linear Interpolation Algorithm for Cartesian Coordinates

**Description:** Given two points $P_A, P_B \in \mathbb{R}^3$, this algorithm computes $n$ linearly interpolated points $P \in \mathbb{R}^{3\times n}$ between them, where $f_s$ is the sampling frequency.

$n \leftarrow f_s$
**for** $i \leftarrow 0$ to $n-1$ **do**
  $a = \frac{1}{n-1} \cdot i$
  $P(i) \leftarrow (P_B - P_A) \cdot a + A$
**end for**

Next, for each of the interpolated positions, the corresponding joint configuration is computed through the inverse kinematics model as presented in the theoretical part, solving for the values of $q_1, q_2$ and $q_4$ respectively. This process is described by the following algorithm:

**Algorithm 2** Inverse forward kinematic model for computing the joint displacement $q_1, q_2, q_4$ for the xArm 7 cobot.

**Description:** Given a set of points $P \in \mathbb{R}^{3\times n}$ this algorithm computes the joint values $q_1, q_2, q_3 \in \mathbb{R}^{1\times n}$ using the inverse kinematic model of the xArm 7 cobot. Here $n$ is the number of points that construct the path. Let $x, y, z, x', z', L, t_1, t_2, R, \phi, u_1, u_2, k_1, k_2$ be helper scalar values.

$k_1 \leftarrow l_4 \cos(\theta_1) + l_5 \sin(\theta_2 + 0.82)$
$k_2 \leftarrow l_4 \sin(\theta_1) + l_5 \sin(\theta_2 - 0.75)$
**for** $i \leftarrow 0$ to $n-1$ **do**
  $(x, y, z) \leftarrow P(i)$
  $q_1(i) \leftarrow \text{atan2}(y, x)$
  $x' \leftarrow \frac{x}{\cos(q_1(i))}$
  $z' \leftarrow z - l_1$
  $L \leftarrow \frac{1}{2}(x'^2 + z'^2 - l_2^2 - l_3^2 - k_1^2 - k_2^2)$
  $t_1 \leftarrow l_2 k_2 + l_3 k_1$
  $t_2 \leftarrow l_3 k_2 - l_2 k_1$
  $R \leftarrow \sqrt{t_1^2 + t_2^2}$
  $\phi \leftarrow \text{atan2}(t_2, t_1)$
  $\sin(q_4 + \phi) \leftarrow \frac{L}{R}$
  $\cos(q_4 + \phi) \leftarrow \sqrt{1 - \sin^2(q_4 + \phi)}$
  $q_4(i) \leftarrow \text{atan2}(\sin(q_4 + \phi), \cos(q_4 + \phi)) - \phi$
  $u_1 \leftarrow l_2 - k_1 \cos(q_4(i)) + k_2 \sin(q_4(i))$
  $u_2 \leftarrow l_3 + k_1 \sin(q_4(i)) + k_2 \cos(q_4(i))$
  $\sin(q_2) \leftarrow \frac{u_1 x' - u_2 z'}{u_1^2 + u_2^2}$
  $\cos(q_2) \leftarrow \frac{u_2 x' + u_1 z'}{u_1^2 + u_2^2}$
  $q_2(i) \leftarrow \text{atan2}(\sin(q_2), \cos(q_2))$
**end for**

Now that the set of joint configurations enabling the end-effector to follow the predefined path has been computed, the final step is to provide these configurations to the robot controller. The controller issues

new joint configuration commands to the robot at an update frequency of $f_s$. Since a total of $n = f_s$ interpolated configurations have been generated, the robot is required to execute exactly one command at each time instance, thereby covering the entire path in a synchronized manner, without additional checking logic required (at least not at this project).

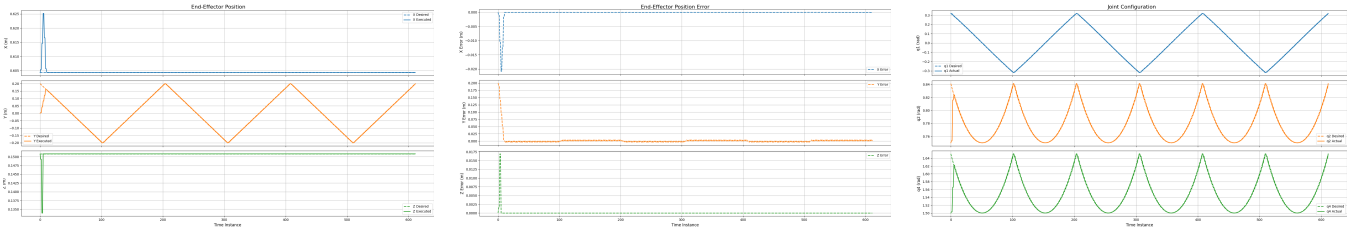
### 3.4 Evaluation

The developed algorithms were implemented in Python and integrated into the ROS 1 framework for controlling the digital model of the xArm 7 cobot. To evaluate the performance of the designed systems, we conducted a repetitive, periodic linear motion between two spatial points: $P_A = (0.6043, -0.2, 0.1508)$ and $P_B = (0.6043, 0.2, 0.1508)$. As can be observed the motion is performed in $y-$axis with 40 cm point-to-point distance.

The ROS environment is configured to operate at a control update frequency of $f_s = 100$ Hz. Consequently, the robot's joint actuators were expected to receive new command values at intervals of $1/f_s = 0.01$ seconds.

Upon execution, the robot successfully followed the predefined linear path between the two points. Although the desired behavior was clearly observable in the Gazebo simulation environment, we also recorded key variables during the execution of the simulation for a more quantitative evaluation of the system's performance. These measurements were taken over four complete execution cycles (from $P_A$ to $P_B$ and back to $P_A$ again). The corresponding results are illustrated in the following figures. Specifically, Fig.4a illustrates the comparison between the desired (interpolated) and actual position of the end-effector during execution. Fig.4b presents the corresponding position error along each of the three Cartesian axes. Lastly, Fig. 4c shows the desired versus actual joint configurations over time.

Specifically we observe a small deviation error in the motion along the $y$-axis. This error primarily arises from two factors. First, the xArm 7 robot is composed entirely of revolute joints, which makes it inherently difficult to follow a perfectly straight Cartesian path, particularly when constrained to planar motion. Second, the discrete nature of the control system—specifically, the finite sampling rate at which commands are issued and executed—can introduce deviations from the ideal continuous trajectory. These factors combined result in slight deviations from the intended straight-line path, as illustrated in the position error plot in Fig. 4b.



(a) End-effector position (b) Position error (c) Joint configuration

Figure 4: Evaluation results for the 3-DOF position control scheme.

# 4 7-DOF Position and Orientation Control

When all we need is a simple transfer from point $A$ to point $B$, a basic linear interpolation approach, as demonstrated in the previous section, is sufficient. This method ensures a straight-line trajectory between the two points. However, as we will see in this section, linear interpolation does not provide control over the speed of the end effector.

In many applications, it is important to maintain realistic velocity continuity while smoothly moving the end effector from point $A$ to point $B$. To achieve this, alternative methods—such as cubic interpolation—are used, which offer smoother transitions and velocity control [1].

Furthermore in this section, we present an algorithm that allows for full control of the robot—both in position and orientation. While the previously discussed approach enables precise point-to-point motions, it is not straightforward to extend it to full control of a 7 DOF manipulator, like the one at hand. Deriving an analytical closed-form solution for such a system is extremely challenging and often impractical.

Therefore, we will explore a more feasible approach that reduces manual derivation and leverages computational resources. Specifically, we will use inverse differential kinematic modeling to achieve the desired motion control.

## 4.1 Trajectory Planning Unit

Starting from the first component of the PFS, the trajectory planning unit. As aforementioned with the simplistic approach of linear interpolation used in the previous section, the velocity of the end effector remains constant from start to end. This is demonstrated by taking the time derivative of the previously mentioned formula (eq. (4)), resulting in a constant value that depends on the number of points $n$ and the constant sampling frequency $f_s = 1/dt$. Specifically the velocity is given by:

$$\dot{x}_e = \frac{B - A}{n - 1} f_s$$

A more refined approach is the use of cubic interpolation. Specifically, third-order interpolation ensures continuity in velocity and leads to smoother trajectories [1] from point-to-point motions. Assuming that the initial and final velocities of the end effector when moving from $A$ to $B$ are zero, the cubic interpolation is formulated as follows:

$$x_e(t) = a_o + a_1 t + a_2 t^2 + a_3 t^3$$
$$\dot{x}_e(t) = a_1 + 2a_2 t + 3a_3 t^2$$

At time instance $i = 0$, therefore $t(0) = 0 \cdot dt = 0$ the end-effector is at point $A$ and at time instance $n - 1$, thus $t(n-1) = (n-1) \cdot dt = t_f$ at point $B$ respectively, with zero initial and final velocities as mentioned

above, we apply these boundary conditions to construct the following linear system:

$$\begin{bmatrix} A \\ B \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Solving for the polynomial coefficients, the system can be re-written as:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix}^{-1} \cdot \begin{bmatrix} A \\ B \\ 0 \\ 0 \end{bmatrix}$$

From where we compute finally the required coefficients:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} A \\ 0 \\ 3\frac{B-A}{t_f^2} \\ 2\frac{A-B}{t_f^3} \end{bmatrix}$$

Therefore, the trajectory equation for each time step in case of the cubic interpolation between points $A$ and $B$ is given by:

$$x_e(i) = A + 3(B-A)\left(\frac{i}{n-1}\right)^2 + 2(A-B)\left(\frac{i}{n-1}\right)^3 \tag{5}$$

From the given the two poses $P_A, P_B \in \mathbb{R}^6$ in this second task we use both position and orientation elements of these vectors. For each of the elements of the vector we employ eq. (5) as presented above and produce for each time instance $i$ a new intermediate value. The series of $k = 0, 1 \ldots, n-1$ interpolated $P_k \in \mathbb{R}^6$ vectors are the result of this trajectory planning scheme. These values are used by the motion control unit to produce the appropriate commands for the robot driver as aforementioned.

## 4.2 Motion Control Unit

To fully control the robot manipulator, one possible way is to solve the inverse kinematics problem and derive a closed-form expression for each of the manipulator's joints, given the desired end-effector position and orientation as we did in the case of the 3-DOF simple position control approach. However, such an approach is quite complex—especially for the cobot used in this setup. An alternative control strategy involves solving the inverse differential kinematics problem, which is the method we adopted in this second approach.

Let $J$ denote the Jacobian matrix of the manipulator. The Jacobian acts as a transformation that maps the joint velocities $\dot{\boldsymbol{Q}}$ to the end-effector velocity vector $V$, which includes both linear and angular velocity components. This relationship is given by:

$$V = J \cdot \dot{\boldsymbol{Q}}$$

From this expression, we can compute the joint velocities as:

$$\dot{\boldsymbol{Q}} = J^{-1} \cdot V$$

Consequently, the joint configuration over time can be obtained through integration:

$$\boldsymbol{Q} - \boldsymbol{Q}_0 = \int_0^{\infty} J^{-1} \cdot V \, dt$$

However, this approach presents a significant limitation: the Jacobian matrix is not always invertible. In the case of a 7-degree-of-freedom (7-DoF) manipulator, the Jacobian is a non-square ($6 \times 7$) matrix, which makes classical matrix inversion impossible. To address this issue, we define the problem as an optimization problem and seek a least-squares solution. This leads to a formulation using the method of Lagrange multipliers [7].

Specifically given the end-effector velocity vector $V$ and the Jacobian matrix $J$, the goal is to find the joint velocities $\dot{\boldsymbol{Q}}$ that minimize a weighted quadratic cost while satisfying the kinematic constraint. The cost function, incorporating Lagrange multipliers $\lambda \in \mathbb{R}^6$, is defined as:

$$g(\dot{\boldsymbol{Q}}, \lambda) = \frac{1}{2} \dot{\boldsymbol{Q}}^T \boldsymbol{W} \dot{\boldsymbol{Q}} + \lambda^T (V - J\dot{\boldsymbol{Q}})$$

Here, $\boldsymbol{W}$ is a positive definite weighting matrix.

To find the optimal solution, we set the gradients of $g$ with respect to $\dot{\boldsymbol{Q}}$ and $\lambda$ to zero:

$$\frac{\partial g}{\partial \dot{\boldsymbol{Q}}} = \boldsymbol{W}\dot{\boldsymbol{Q}} - J^T\lambda = 0$$
$$\frac{\partial g}{\partial \lambda} = V - J\dot{\boldsymbol{Q}} = 0$$

Solving the first equation for $\dot{\boldsymbol{Q}}$ yields:

$$\dot{\boldsymbol{Q}} = \boldsymbol{W}^{-1} J^T \lambda$$

This expression will be further substituted into the constraint equation to solve for $\lambda$, and ultimately for $\dot{\boldsymbol{Q}}$.

Since $\boldsymbol{W}$ is symmetric and positive definite, its inverse $\boldsymbol{W}^{-1}$ exists. Substituting the expression for $\dot{\boldsymbol{Q}}$ into the constraint equation yields:

$$J\dot{\boldsymbol{Q}} = V = J\boldsymbol{W}^{-1} J^T \lambda$$

Solving again for the Lagrange multiplier vector $\lambda$ gives:

$$\lambda = (J\boldsymbol{W}^{-1} J^T)^{-1} V$$

Substituting this result into the earlier expression for $\dot{\boldsymbol{Q}}$, we obtain:

$$\dot{\boldsymbol{Q}} = \boldsymbol{W}^{-1} J^T (J\boldsymbol{W}^{-1} J^T)^{-1} V$$

This expression describes actually the weighted least-squares solution to the inverse differential kinematics problem, with weight matrix $\boldsymbol{W}$. If we choose our weights to be the identity matrix, showing no special preference to particular joints, then the expression simplifies to the well-known Moore-Penrose pseudoinverse Jacobian matrix as given bellow:

$$J^{+} = J^{T}(JJ^{T})^{-1}$$

Thus, the joint velocities become:

$$\dot{\boldsymbol{Q}} = J^{+} \cdot V$$

By iteratively updating the joint velocities using this formulation, we ensure smooth and continuous movement toward the target configuration, even in scenarios where exact inversion of the Jacobian is not feasible. The general update rule in discrete time is given by:

$$\dot{\boldsymbol{Q}}_{i+1} = \dot{\boldsymbol{Q}}_{i} + J^{+}V \cdot \Delta t$$

It is important to note that, since the system is implemented on a computer, computations are inherently discrete. This discretization, along with numerical integration of joint displacements, may introduce drift in the solution. As a result, the final pose of the end-effector may deviate from the desired target. Therefore to mitigate this, we incorporate the accumulated error into the control logic. Let $e$ denote the pose error between the desired and actual end-effector configurations:

$$e = P_d - P_a$$

where $P_d$ is the desired pose, and $P_a$ is the actual pose of the end-effector, both in $\mathbb{R}^6$. Taking the time derivative of this error form yields:

$$\dot{e} = \dot{P}_d - \dot{P}_a$$

Where $\dot{P}_d = V$ and $\dot{P}_a = J\dot{\boldsymbol{Q}}$. Substituting we obtain:

$$\dot{e} = V - J\dot{\boldsymbol{Q}}$$

Solving for $\dot{\boldsymbol{Q}}$ gives:

$$\dot{\boldsymbol{Q}} = J^{-1}(V - \dot{e})$$

To ensure that the error $e$ converges asymptotically to zero, we can define the following equation without loss of generality:

$$\dot{e} + Ke = 0$$

where $K$ is a positive definite gain matrix. The convergence rate depends on the eigenvalues of $K$—larger eigenvalues generally lead to faster convergence. However, due to the discrete-time implementation, there exists an upper bound beyond which the system may become unstable [7].

Finally, by substituting $\dot{e} = -Ke$ into the earlier equation, we arrive at the control law for the redundant 7-DOF manipulator:

$$\boldsymbol{Q}_{i+1} = \boldsymbol{Q}_{i} + J^{+}(V + Ke)\,\Delta t \tag{6}$$

This formulation enables robust control of the end-effector trajectory while compensating for pose errors introduced by numerical integration in discrete-time systems. The closed-loop control system presented by the above equation can also be visually described by the following Fig. 5.

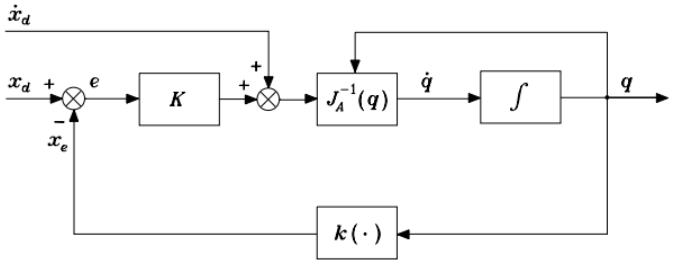



Figure 5: The closed-loop system for the 7-DOF Jacobian-based controller [7]

### 4.3 Implementation

Note that, to implement the above control approach, several computations are pre-required. Most notably, we need to compute the Jacobian matrix $J$ of the manipulator at each time step. Additionally, although not explicitly shown in the derivation above, we also require the computation of the end-effector velocity vector $V$, which includes both the linear velocity (due to translational motion) and the angular velocity (due to rotational motion) of the end-effector. These velocities can be derived from the desired pose trajectory through interpolation. Although linear velocity is straightforward to compute, angular velocity requires some additional effort!

Lets start from the first component. The first component required for implementing a path-following algorithm in this second approach is, of course, the Jacobian matrix. The Jacobian matrix is a $6 \times 7$ matrix structured as follows:

$$J(\boldsymbol{Q}) = \begin{bmatrix} J_{L1} & J_{L2} & J_{L3} & J_{L4} & J_{L6} & J_{L7} \\ J_{A1} & J_{A2} & J_{A3} & J_{A5} & J_{A6} & J_{A7} \end{bmatrix} \tag{7}$$

where each $J_{Li}, J_{Ai} \in \mathbb{R}^{3\times 1}$ represents a vector relating the motion of joint $q_i$ (with $i = 1, 2, \ldots, 7$) to the linear and angular velocity of the end-effector.

Since all joints are revolute, each Jacobian element is computed as:

$$J_{Li} = \hat{b}_{i-1} \times P_{i-1,E}, \qquad J_{Ai} = \hat{b}_{i-1} \tag{8}$$

where the vector $\hat{b}_{i-1}$ is given by:

$$\hat{b}_{i-1} = R^0_{i-1} \cdot \hat{b} \tag{9}$$

and the position vector $P_{i-1,E}$ of the end-effector with respect to frame $i-1$ is defined by:

$$\begin{bmatrix} P_{i-1,E} \\ 1 \end{bmatrix} = A^0_7(\boldsymbol{Q}) \cdot \hat{r} - A^0_{i-1}(q_1, q_2, \ldots, q_{i-1}) \cdot \hat{r} \tag{10}$$

with $\hat{r} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^T$.

Furthermore, due to the use of the (modified) DH representation convention and the fact that all joints are revolute, the directional vector $\hat{b}$ aligns with the $z$-axis, hence:

$$\hat{b} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{11}$$

Therefore, the Jacobian matrix for the differential kinematic model is finally explicitly structured as:

$$J(\boldsymbol{Q}) = \begin{bmatrix} \hat{b}_0 \times P_{0,7} & \hat{b}_1 \times P_{1,7} & \hat{b}_2 \times P_{2,7} & \hat{b}_3 \times P_{3,7} & \hat{b}_4 \times P_{4,7} & \hat{b}_5 \times P_{5,7} & \hat{b}_6 \times P_{6,7} \\ \hat{b}_0 & \hat{b}_1 & \hat{b}_2 & \hat{b}_3 & \hat{b}_4 & \hat{b}_5 & \hat{b}_6 \end{bmatrix} \tag{12}$$

Care must be taken here, since the homogeneous transformation matrices used in this formulation differ from those used in the forward kinematics through to the DH table. Specifically, each transformation matrix must be recomputed with a different configuration—shifting the joint displacement-dependent transformation to the beginning of the sequence. As such, each homogeneous transformation matrix here describes joint-to-joint motion.

$$\begin{aligned}
A_1^0(q_1) &= Rot(z, q_1) \cdot Tra(z, l_1) \cdot Rot(x, -\pi/2) \\
A_2^1(q_2) &= Rot(z, q_2) \cdot Rot(x, \pi/2) \\
A_3^2(q_3) &= Rot(z, q_3) \cdot Tra(z, l_2) \cdot Rot(x, \pi/2) \cdot Tra(x, l_3) \\
A_4^3(q_4) &= Rot(z, q_4) \cdot Rot(x, \pi/2) \cdot Tra(x, l_4 sin(\theta_1)) \\
A_5^4(q_5) &= Rot(z, q_5) \cdot Tra(z, l_4 cos(\theta_1)) \cdot Rot(x, \pi/2) \\
A_6^5(q_6) &= Rot(z, q_6) \cdot Rot(x, -\pi/2) \cdot Tra(x, l_5 sin(\theta_2)) \\
A_7^6(q_7) &= Rot(z, q_7)
\end{aligned}$$

Next, we compute the homogeneous transformations from joint $i$ to the base of the manipulator using the aforementioned transformation matrices:

$$\begin{aligned}
A_2^0 &= A_1^0(q_1) \cdot A_2^1(q_2) \\
A_3^0 &= A_2^0(q_1, q_2) \cdot A_3^2(q_3) \\
A_4^0 &= A_3^0(q_1, q_2, q_3) \cdot A_4^3(q_4) \\
A_5^0 &= A_4^0(q_1, q_2, q_3, q_4) \cdot A_5^4(q_5) \\
A_6^0 &= A_5^0(q_1, q_2, q_3, q_4, q_5) \cdot A_6^5(q_6) \\
A_7^0 &= A_6^0(q_1, q_2, q_3, q_4, q_5, q_6) \cdot A_7^6(q_7) \cdot \mathrm{Trans}_z(l_5 \cos(\theta_2))
\end{aligned}$$

Subsequently, we compute the axis of rotation for each joint, $\hat{b}_{i-1}$, determine the position vectors, and then compute the linear velocity components of the Jacobian. This leads to the full construction of the Jacobian matrix as presented.

Since $\hat{b}$ describes a revolution around the $z$-axis, equation (9) effectively extracts the third column (the $z$-axis direction) of the rotation matrix and the first three rows.

Additionally, the multiplication of a homogeneous transformation matrix $A_{i-1}^i(q_1, \ldots, q_i)$ with the vector $\hat{r}$ effectively yields the position vector by extracting the first three elements of the fourth column of the transformation matrix. These relationships simplify computations and make it more computationally-friendly.

Using these relationships and equations above, we now outline the step-by-step algorithm implemented for computing the Jacobian matrix for the xArm 7 cobot:

**Algorithm 3** Computing the Jacobian Matrix for the xArm 7 Cobot

**Description:** The transformation matrices $A_{i+1}^{i}$ represent the homogeneous transformations from frame $i$ to frame $i+1$, defined by joint variables $q_i$. The function $Rot(axis, \alpha)$ denotes a rotation matrix about the specified axis by angle $\alpha$, and $Tra(axis, d)$ denotes a translation along the specified axis by distance $d$. Vectors $b_i$ represent the joint axes expressed in the base frame, and $P_7^i$ the vectors from each joint origin to the end-effector origin in the base frame. The result is a $6 \times 7$ Jacobian matrix for the xArm 7 cobot.

$A_1^0 \leftarrow Rot(z, q_1) \cdot Tra(z, l_1) \cdot Rot(x, -\pi/2)$
$A_2^1 \leftarrow Rot(z, q_2) \cdot Rot(x, \pi/2)$
$A_3^2 \leftarrow Rot(z, q_3) \cdot Tra(z, l_2) \cdot Rot(x, \pi/2) \cdot Tra(x, l_3)$
$A_4^3 \leftarrow Rot(z, q_4) \cdot Rot(x, \pi/2) \cdot Tra(x, l_4 \sin(\theta_1))$
$A_5^4 \leftarrow Rot(z, q_5) \cdot Tra(z, l_4 \cos(\theta_1)) \cdot Rot(x, \pi/2)$
$A_6^5 \leftarrow Rot(z, q_6) \cdot Rot(x, -\pi/2) \cdot Tra(x, l_5 \sin(\theta_2))$
$A_7^6 \leftarrow Rot(z, q_7)$
**for** $i \leftarrow 2$ to $7$ **do**
    $A_i^0 \leftarrow A_{i-1}^0 \cdot A_i^{i-1}$
**end for**
$A_7^0 \leftarrow A_7^0 \cdot Tra(z, l_5 \cos(\theta_2))$
$b_0 \leftarrow [0, 0, 1]^T$
**for** $i \leftarrow 1$ to $6$ **do**
    $b_i \leftarrow A_i^0[:3, 2]$
**end for**
$P_7^0 \leftarrow A_7^0[:3, 3]$
**for** $i \leftarrow 1$ to $6$ **do**
    $P_7^i \leftarrow P_7^0 - A_i^0[:3, 3]$
**end for**
$J \leftarrow \begin{bmatrix} b_0 \times P_7^0 & b_1 \times P_7^1 & b_2 \times P_7^2 & b_3 \times P_7^3 & b_4 \times P_7^4 & b_5 \times P_7^5 & b_6 \times P_7^6 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 \end{bmatrix}$

The Jacobian matrix is evaluated at every new configuration $\boldsymbol{Q}$. However, the update rule, as presented in equation (6), requires not only the computation of the pseudo-inverse of the Jacobian matrix (as introduced earlier), but also the desired end-effector velocity vector augmented by the error compensation term $Ke$. This computation is performed at each time step of the $n = f_s$ total time steps defined.

Assuming that the user specifies to the path-following system the desired poses $P_A$ and $P_B$ in terms of the end-effector position coordinates and orientation. The position is provided in Cartesian coordinates, while for the orientation, we adopt a description using Euler angles (roll, pitch, and yaw). Thus the desired velocity and the error term are directly related to the interpolation scheme employed between these two poses as described earlier through a cubic approximation. Specifically by differentiating the trajectory interpolation expressions, we compute the velocities that the end-effector must follow at the intermediate poses.

Caution is needed at this point. A pose described by Cartesian coordinates and Euler rotation angles, as given by equation (3), when differentiated with respect to time, yields the linear velocity but not the

angular velocity directly. Specifically, we obtain:

$$\dot{P}_k = \begin{bmatrix} \dot{r}_k \\ \dot{\boldsymbol{\phi}}_k \end{bmatrix}$$

where $\dot{r}_k = v_k$ represents the linear velocity at the $k$-th intermediate sample, but in general $\dot{\phi}_k \neq \omega_k$, where $\omega_k$ is the true angular velocity of the end-effector [7]. Nonetheless, a relationship exists between the time derivatives of the Euler angles and the angular velocity of the end-effector, expressed as:

$$\omega = T_{zyx} \cdot \dot{\phi} \tag{13}$$

where,

$$T_{zyx} = \begin{bmatrix} \cos(\phi_y)\cos(\phi_z) & -\sin(\phi_z) & 0 \\ \cos(\phi_y)\sin(\phi_z) & \cos(\phi_z) & 0 \\ -\sin(\phi_y) & 0 & 1 \end{bmatrix} \tag{14}$$

is the transformation matrix that related the angles rate of change and orientation Euler angles roll, pitch and yaw with the actual angular velocity of the end-effector.

Regarding the error term—which encapsulates both position and orientation components—its computation is straightforward:

$$e = \begin{bmatrix} r_k - r_e \\ \phi_k - \phi_e \end{bmatrix}$$

Here, $r_k$ is the interpolated position vector, $\phi_k$ is the interpolated orientation vector, and $r_e$, $\phi_e$ denote the current position and orientation of the end-effector respectively, as provided by feedback to the controller and derived from the forward kinematic equation (2).

With these components in place, we now have all the necessary elements to finally implement the 7-DOF control algorithm, which is presented bellow:

**Algorithm 4** 7-DOF Path-Following Orientation and Control Algorithm

**Description:** Let $P_A, P_B \in \mathbb{R}^6$ be the initial and final pose vectors (position and orientation). Let $P, \dot{P} \in \mathbb{R}^{6\times(n+1)}$ denote the interpolated pose and its time derivative, respectively. Let $V \in \mathbb{R}^6$ be the end-effector spatial velocity, and $(a_0, a_1, a_2, a_3) \in \mathbb{R}^6$ the cubic interpolation coefficients for each DOF. Let $t$ be the time interval, $n$ the number of samples, $f_s$ the sampling frequency, $dt = 1/f_s$ the time step, and $t_f$ the final time. Let $J \in \mathbb{R}^{6\times7}$ be the geometric Jacobian, $Q \in \mathbb{R}^7$ the joint configuration, $r_{cur}, \phi_{cur} \in \mathbb{R}^3$ the current position and orientation of the end effector, and $e \in \mathbb{R}^6$ the pose error. $T \in \mathbb{R}^{4\times4}$ is a transformation matrix mapping angular velocity representation, $A_7^0(Q)$ is the forward kinematics mapping from joint space to task space, and $K \in \mathbb{R}^{7\times7}$ is a positive definite proportional gain matrix.

$t \leftarrow 0$
$dt \leftarrow 1/f_s$
$n \leftarrow f_s$
$dt \leftarrow 1/f_s$
$t_f \leftarrow n \cdot dt$
$(a_0, a_1, a_2, a_3) \leftarrow \left(P_A, 0, 3\frac{P_B - P_A}{t_f^2}, 2\frac{P_A - P_B}{t_f^3}\right)$
**for** $i \leftarrow 0$ to $n-1$ **do**
  $t \leftarrow i \cdot dt$
  $P(i) \leftarrow a_0 + a_1 t + a_2 t^2 + a_3 t^3$
  $\dot{P}(i) \leftarrow a_1 + 2a_2 t + 3a_3 t^2$
  $V[:3] \leftarrow \dot{P}[:3]$
  $V[3:] \leftarrow T \cdot \dot{P}[3:]$
  $r_{cur} \leftarrow A_7^0(Q(i))[:3, 3]$
  $\phi_{cur} \leftarrow to\_euler\_angles(A_7^0(Q(i))[:3, :3])$
  $e \leftarrow P(i) - [r_{cur} \quad \phi_{cur}]^T$
  $J \leftarrow J(Q(i))$
  $J^+ \leftarrow J^\top (JJ^\top)^{-1}$
  $\dot{Q} \leftarrow J^+(V + Ke)$
  $Q(i+1) \leftarrow Q(i) + \dot{Q} \cdot dt$
**end for**

## 4.4 Evaluation

The developed algorithms were again implemented in Python and integrated into the ROS framework to control the digital model of the xArm 7 cobot in Gazebo. To evaluate the performance of the designed control systems, we conducted a repetitive, periodic motion between two spatial points, similar to the first task. This time, however, the orientation of the end-effector, expressed in Euler angles, was also considered. To determine the end-effector's orientation at the target points, we utilized a previously implemented controller script and displayed the orientation as the end-effector passes through the desired positions. The poses defined for the two target points are: $P_A = (0.6043, -0.2, 0.1508, 3.1415, -0.0586, 0.3197)$ and $P_B = (0.6043, 0.2, 0.1508, 3.1415, -0.0586, 0.3197)$, respectively.

The ROS environment was configured to operate at a control update frequency of $f_s = 100$ Hz. Conse-

quently, the robot's joint actuators were expected to receive new command values at intervals of $1/f_s = 0.01$ seconds. The algorithm was initialized with a gain matrix $K = \text{diag}(2, 2, 2, 2, 2, 2)$ for the proportional term- feedback error.

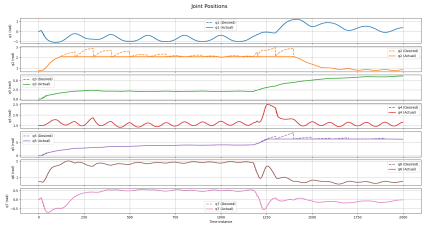
Upon execution of the control script, the robot began moving but exhibited irregular behavior. Initially, it deviated from the expected trajectory during the first few iterations. However, after this initial transient phase, the robot got its expected configuration and began following the desired trajectory between the two points, as observed visually in the Gazebo simulation. Various state variables were recorded during the simulation for quantitative analysis as well. These measurements were taken over ten iterations, and the results are shown in the following figures.

Specifically, Fig. 6a shows the comparison between the desired (interpolated) and actual position of the end-effector. Fig. 6c presents the corresponding position error along the axes. Fig. 6b illustrates the commanded versus actual orientation, while Fig. 6d shows the orientation error. Finally, Fig. 6e displays the desired versus actual joint configurations over time for all seven joints of the manipulator.

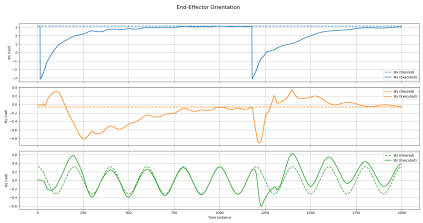
The aforementioned transient behavior is also depicted in these plots. Specifically, Fig. 6a shows that during the first seven cycles (a cycle is defined as the motion from $P_A$ to $P_B$ and back to $P_A$ here as well), the robot struggles to follow the desired path along the $x$- and $z$-axes. Although it tracks the $y$-axis position well, a significant deviation in orientation is observed, particularly along the $y$-axis, while better tracking is seen in the $x$- and $z$-axis rotations, as shown in Fig. 6b. After approximately time step 1400, the robot stabilizes, and both position and orientation errors begin to converge toward zero.

Interestingly, the robot closely follows the desired joint configuration, except for a noticeable deviation in joint $q_2$ during the stabilization period. Most of the observed errors can be attributed to the fact that the numerical integration is not perfectly synchronized with the control update period of $1/f_s$ at every iteration. We tried to compensate for this by interpolating the poses and their rate of change in real-time during simulation, rather than relying on precomputed interpolated values. However, no significant improvement is observed.

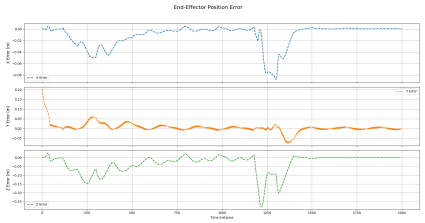
The time required for the robot to stabilize and for the position and orientation errors to converge to zero could be theoretically reduced by appropriately tuning the proportional gain matrix. Since deviations are primarily observed in the $x$- and $z$-axis positions and the $y$-axis orientation, it would be reasonable to start by increasing the corresponding elements in the gain matrix. An attempt in tuning the gains either resulted in no improvements at all or resulted in the simulation to crash. We found that one viable configuration is by setting the gains to $K = diag(4, 2, 4, 1, 2, 1)$ which however offered no notable differences.



(a) Position



(b) Orientation



(c) Position Error

(d) Orientation Error

(e) Joints Configuration

Figure 6: Evaluation results for the 7-DOF position control scheme.

# 5 The ROS Package

## 5.1 Description

The algorithms presented in this report are implemented in ROS 1 (Noetic) as a package named `robosys_path_followi`
The core functionality is in two Python scripts located in the `"scripts"` directory: `controller_3dof.py` and `controller_7dof.py`. Specifically the script `controller_3dof.py` implements a simplified path-following approach based on linear interpolation and a position-only control strategy using inverse forward kinematics, while `controller_7dof.py` implements a the full-control method based on the inverse differential kinematic model, as discussed in the theoretical section of this report.

Few supporting modules provide the core components required by both controllers implemented. The file `kinematics.py` implements the kinematic model of the xArm 7 cobot through the class `xArm7_kinematics`. Important functions within this class include `compute_angles(ee_position)`, which calculates the joint configuration for joints 1, 2, and 4 given an end-effector position, and `compute_jacobian(r_joints_array)`, which determines the Jacobian matrix for a given 7-element joint configuration array. Another file, `utils.py`, contains shared utility functions such as the elementary homogeneous transformation functions `Rot(axis, angle)` and `Tra(axis, displacement)`, which perform rotation and translation along or around a given axis. These are used for computing forward kinematics in `kinematics.py`. The file also includes interpolation functions and routines for converting angles into angular velocities, as described in the theoretical background.

Each script receives a high-level task description via a text file located in the `scripts` directory which is read internally. Specifically, `controller_3dof.py` uses `task_file_3dof.txt`, while `controller_7dof.py` uses `task_file_7dof.txt`. These files contain a single directive format:

$$\texttt{set\_pose:}\quad P_X,\ P_Y,\ P_Z,\ O_X,\ O_Y,\ O_Z$$

where $P_X$, $P_Y$, and $P_Z$ represent the target position in cartesian space, and $O_X$, $O_Y$, and $O_Z$ are the Euler angles specifying the orientation of the end-effector respectively. Although the 3-DOF controller does not use the orientation values, they must still be included in the file for consistency. All values must be 64-bit float compatible.

Both main scripts publish joint configurations to the ROS topic `/xar/joint_i_position_controller/command` at a frequency of 100 Hz. The rate is retrieved from the environment variable exposed by the node `/rate`. The ROS node instantiated in both scripts is named `controller_node`.

The launch files required to run both controllers are provided in the `launch` directory.

## 5.2 Installation

For the purposes of this project, implementation and evaluation are facilitated through a Docker image, which contains all necessary packages for installing and preparing the underlying system with the ROS 1

(Noetic) development environment. In the supporting material accompanying this report, we provide a Dockerfile that automatically configures the Docker image to include everything needed for development.

Users who wish to build the Docker image should follow the steps outlined in this subsection. Those who already have ROS 1 installed on their system can skip the Docker setup and proceed to the next subsection, including the package directly in their workspace, ready for launch.

The following steps for setting up the ROS 1 Docker image need to be executed only once per machine. The instructions below target Linux systems only; however, the process is similar for Windows users via the Windows Subsystem for Linux 2 (WSL-2), with only minor differences that require further investigation by the users themselves.

First, Docker must be installed. Run the following commands in your terminal. If you already have Docker installed omit this step.

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates curl software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyring
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io
```

Next, within the provided materials folder, set up the image using the following command:

```
sudo docker build -t ros-docker .
```

To initiate the newlly created image, we provide a shell script named `start-docker-image.sh`. You can find this script in the accompanying materials. Before executing the script, you must first grant it execution permissions:

```
chmod +x start-docker-image.sh
```

Next, run the script:

```
sudo ./start-docker-image.sh
```

Note that within the `external` folder your own packages, such as the `robosys_path_following` package provided in this implementation, are placed.

## 5.3 Usage

Start two separate instances (i.e., two terminal shells) of the Docker image using the `start-docker-image.sh` script. In one terminal, you can execute your packages (those stored in `external`), which are internally linked to the ROS workspace and in the other the ROS server (or Gazebo environment) is launched.

However before executing your packages for the first time, you must compile all the packages in the system:

```
source devel/setup.bash
rosdep install --from-paths src/external --ignore-src -r -y
catkin_make
```

Then, make the two core Python scripts executable:

```
cd ~/catkin_ws/src/external/robosys_path_following/scripts
chmod +x controller_3dof.py
chmod +x controller_7dof.py
```

You are now ready to run the scripts and observe the robot model moving in the simulated environment launched earlier. In the first terminal now, start the Gazebo simulator and load the xArm 7 cobot model. To launch the graphical environment execute:

```
roslaunch xarm_gazebo xarm7_pf.launch
```

In the second terminal launch the programs by simply executing:

```
roslaunch robosys_path_following path_following_3dof.launch
```

or

```
roslaunch robosys_path_following path_following_7dof.launch
```

An important note: if you wish to define a different task than the one used in this project, you can modify the corresponding task file. Use the following for the 3-DOF (position-only control):

```
~/catkin_ws/src/external/robosys_path_following/scripts/task_file_3dof.txt
```

Or, for the 7-DOF (full position and orientation control):

```
~/catkin_ws/src/external/robosys_path_following/scripts/task_file_7dof.txt
```

Additionally, there are some global constant variables within the main Python scripts that can be modified, related to the simulation:

```
C_USER_TASK_FILE
C_EVALUATION
C_EVALUATION_CYCLES
```

The `C_USER_TASK_FILE` variable specifies the task file name (actually path and name) to be used by the controller in order to define the desired path as determined by the user.

The `C_EVALUATION` variable is a boolean flag (`True` or `False`) that determines whether the program runs in evaluation mode. In this mode, the simulation executes a predefined number of cycles between the target poses and then generates plots of several evaluation metrics, as discussed in this report.

Finally, the `C_EVALUATION_CYCLES` variable defines the aforementioned number of cycles the controller will execute during evaluation mode.

By default, these variables are configured with the same values used to produce the results presented in this report, as presented in the previous sections.

# 6 Conclusions

This project gave us a valuable opportunity to connect theoretical concepts with practical applications by diving deeper into one of the most widely used tools in robotics: the Robot Operating System. Through an introductory exploration of ROS, we developed software-based models and gained a better understanding of how it supports both academic and industrial applications.

We then focused on the kinematic modeling and control of the xArm 7 collaborative robot, deriving and implementing two control schemes. The first method relies on the inverse kinematics model to directly compute joint configurations based on the robot's desired pose (position only). This approach, although dependent on obtaining closed-form solutions—which can be challenging for manipulators with redundant degrees of freedom as the xArm 7—offers high accuracy and is well-suited for tasks that require precise and repetitive movements. In our case we kept things simple and implemented only position control with 3-DOFs as requested.

The second method takes a more algorithmic approach by modeling the robot's kinematics in a simpler, computationally lighter way. This method utilized the pseudo-inversion of the Jacobian matrix to provide the inverse differential kinematic model. This control scheme operates in discrete time and uses integration, which naturally leads to some accumulated error over time. To address this, a proportional gain term to the controller, helping ensure the error converges to zero within a finite period is employed. This method is particularly beneficial for redundant robot manipulators, as it offers extremely useful features such as the ability to avoid singularities, decompose complex tasks into subtasks, and optimize motion for specific objectives.

Subsequently we implemented and tested these control algorithms using Python within the ROS framework, taking advantage of its runtime resources and tools. During testing in the Gazebo simulator, we observed that the inverse kinematics control method which follows a linearly interpolated path between two points, performs the task with reasonable accuracy. However, some deviation from the ideal path was noticeable, which we attribute to the discrete nature of the simulation and to numerical accuracy errors.

For the inverse differential kinematics approach, we noticed that the robot exhibited a relatively long settling time—requiring around 7 cycles to align with the desired trajectory. In theory, this behavior can be improved by properly tuning the proportional gains that scale the error term used in feedback control. While we did some initial experiments in this direction, a full analysis and tuning process were not deeply pursued.

There are certainly a few optimizations that could enhance the current implementation. One important improvement would be the addition of workspace limit checks to ensure that input commands are physically feasible—currently, this is left to the user's discretion assuming that the user is experienced and enters right values. Additionally, more careful tuning of the proportional gain could reduce the settling time significantly in the second approach. Developing an automated method for gain tuning would also be a valuable future enhancement and a quite interesting task for us.

Overall the objectives of the project—both the technical requirements and the personal learning goals—have been successfully met with its completion.

# References

[1] K. Tzafestas, "Robotics Laboratory Lectures", MSc in Control Systems and Robotics, Academic Year 2024–2025, National Technical University of Athens (unpublished lecture notes).

[2] ROS::Home, online : `https://www.ros.org/`, (Accessed May 2025).

[3] Claudio Taesi, Francesco Aggogeri, Nicola Pellegrini, *"COBOT Applications - Recent Advances and Challenges"*, MDPI, Robotics, 2023.

[4] Baris Akgun, Maya Cakmak, Jae Wook Yoo, Andrea Lockerd Thomaz, *"Trajectories and keyframes for kinesthetic teaching: a human-robot interaction perspective"*, ACM, HRI, 2012.

[5] Petar Kormushev, Sylvain Calinon, Darwin G. Galdwell, *"Imitation Learning of Positional and Force Skills Demonstrated via Kinesthetic Teaching and Haptic Input"*, Taylor & Francis, Advanced Robotics, 2011.

[6] Andrius Dzedzickis, Jurga Subaciute-Zemaitiene, Ernestas Sutinys, Urte Samukaite-Bubniene, Vytautas Bucinskas, *"Advanced Applicataions of Industrial Robotics: New Trends and Possibilities"*, MDPI, Applied Sciences, 2021.

[7] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villano, Giusepper Oriolo, "Robotics, Modelling, Planning and Control", Springer, (ISBN) 978-1-84628-641-4, 2009.

[8] John J. Craig, "Introduction to Robotics, Mechanics and Control", Pearson Education International, (ISBN) 0-13-123629-6, 2005.