Assignment 2

AE4462-17: Aircraft Emissions and Climate Effects

Henry Adam
Robin Kalderen
Feitong Wang

Delft University of Technology

Contents

0

Introduction

0.1. Report Objective

Physical interaction is a fundamental aspect of robotic manipulators, as robots inherently interact with the environment to manipulate it. In order to examine physical robotic interaction, this report characterizes and utilizes a simple 4DOF manipulator called the Edubot. First, its mobility is characterized and its forward kinematics are derived. Using the forward kinematics, the inverse kinematics are derived in order to command the robot into specific positions while maintaining joint limits. The inverse kinematics is tested by commanding it to various feasible and infeasible positions and tracing the TU Delft flame with its end-effector. After the inverse kinematics are confirmed with these tests, the Edubot's Jacobian is derived and examined to gain an understanding of its uses and limitations, particularly in singularity scenarios. Using the newfound Jacobian, a constant-velocity trajectory is commanded on the robot. After this base buildup, the robot is ready for its interaction tasks. The first two tasks are to pick and place different objects in the environment, one being a berry and the other being a PLA cube. Finally, the last task is to use the Edubot to wipe up a spill with a sponge and return the sponge to its original position.

0.2. A Note to the Grader on Code Structure

In our enthusiasm for the course, all three group members have created our own version of the code completed mostly individually and containing almost the entire assignment. As such, for each section of the assignment the group has between one and four different solutions programmed in one of three different programming languages. We realize that, although this shows our excitement for robotics, it does not make the grader's job easier, so we've structured the report to show our primary solutions for each section, followed by alternative solutions and methods that the grader can choose to ignore if they'd like. Each of these additional sections is marked as (Alternative) to signify to the grader that the section is ancillary.

1

Mobility and Workspace

Task 1.1

1.0.1. Robot parameters

The lengths of the 4-DoF robot's links, measured in simulation (Figure 1.1) and in real life (Figure 1.2), are shown in Table 1.1.

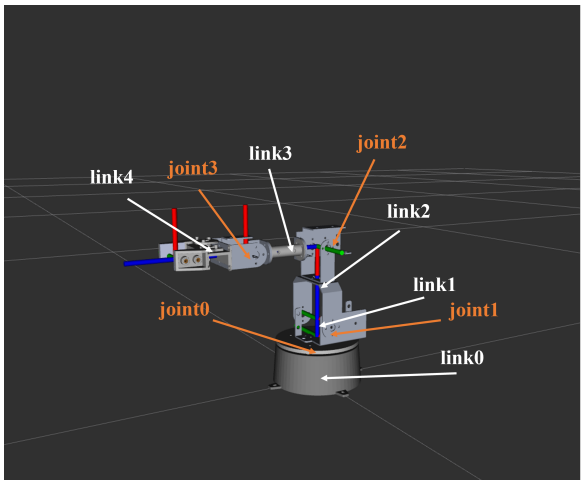


Figure 1.1: simulation robot arm

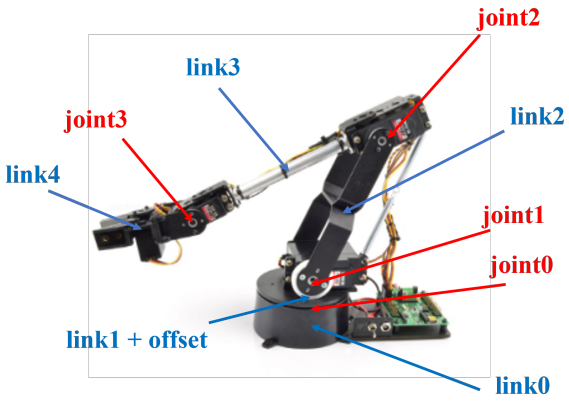


Figure 1.2: actual robot arm

Table 1.1: lengths of robot links in simulation and real life in [cm]

	simulation	real robot arm
link 0	0.05	0.045
link 1	0.02	0.025
joint 2's offset	0	0.017
link 2	0.0825	0.095
link 3	0.086	0.105
link 4	0.075	0.075

1.0.2. Mobility

Considering the general mobility, the end-effector of this 4-DoF manipulator can move in 3D space with limited rotational capability. It has three translational degrees of freedom (X, Y, Z) and one rotational degree of freedom around the Y-axis in base reference frame. Compared to a 6-DoF manipulator, its workspace is more constrained, meaning the end-effector cannot achieve arbitrary orientations.

The system provides full control over four degrees of freedom: three translations (X, Y, Z) and one rotation (pitch) around the Y-axis. The manipulator allows independent control of its position but has limited orientation control. Unlike a 6-DoF system, this 4-DoF manipulator cannot achieve arbitrary orientations, which may limit its ability to perform tasks requiring complex rotations.

Task 1.2

1.0.3. Forward kinematics

The mathematical expression for forward kinematics can be derived by applying transformation matrices to each adjacent joint and then multiplying these matrices together, as shown below. The translation and rotation directions of each frame are described in the coordinate frame of its preceding joint. The length of link i is denoted as l_i throughout the rest of the report.

From base (ground) to joint 0, the frame is risen by link 0 in +Z direction and rotate by q_0 about +Z axis:

$$\begin{bmatrix} \cos(q_0) & -\sin(q_0) & 0 & 0 \\ \sin(q_0 & \cos(q_0) & 0 & 0 \\ 0 & 0 & 1 & l_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

From joint 0 to joint 1, the frame is risen by link 1 in +Z direction and rotate by q_1 about +Y axis. In actual robot, there exists an additional offset due to that joint 1 and joint 0 do not perpendicularly coincide. The transformation matrix then is:

$$\begin{bmatrix} \cos(q_1) & 0 & \sin(q_1) & -\text{joint 2's offset} \\ 0 & 1 & 0 & 0 \\ -\sin(q_1) & 0 & \cos(q_1) & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.2}$$

From joint 1 to joint 2, the frame is translated by link 2 in +Z direction and flipped so +X axis points towards -Z direction in base frame and +Z axis points towards -X direction in base frame. Then it is rotated by q_2 in +Y axis.

$$\begin{bmatrix} \sin(q_2) & 0 & -\cos(q_2) & 0 \\ 0 & -1 & 0 & 0 \\ -\cos(q_2) & 0 & -\sin(q_2) & l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

From joint 2 to joint 3, the frame is first translated by link 3 in +Y direction and rotated 180 degree about +Z axis. Then it is rotated by q_3 in +Y direction:

$$\begin{bmatrix} -\cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & -1 & 0 & 0 \\ -\sin(q_3) & 0 & \cos(q_3) & l_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

From joint 3 to the end effector, the frame is translated by link 4 in +Z direction:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.5}$$

By multiplying these matrices, we have the transformation matrix from base frame to end effector attached frame:

$$\begin{bmatrix} \sin(q_1 - q_2 + q_3)\cos(q_0) & -\sin(q_0) & -\cos(q_1 - q_2 + q_3)\cos(q_0) & -\cos(q_0)\left(\text{offset}_2 + l_4\cos(q_1 - q_2 + q_3) - l_2\sin(q_1) + l_3\cos(q_1 - q_2)\right) \\ \sin(q_1 - q_2 + q_3)\sin(q_0) & \cos(q_0) & -\cos(q_1 - q_2 + q_3)\sin(q_0) & -\sin(q_0)\left(\text{offset}_2 + l_4\cos(q_1 - q_2 + q_3) - l_2\sin(q_1) + l_3\cos(q_1 - q_2)\right) \\ \cos(q_1 - q_2 + q_3) & 0 & \sin(q_1 - q_2 + q_3) & l_0 + l_1 + l_4\sin(q_1 - q_2 + q_3) + l_2\cos(q_1) + l_3\sin(q_1 - q_2) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

1.0.4. Measuring Joint Limits

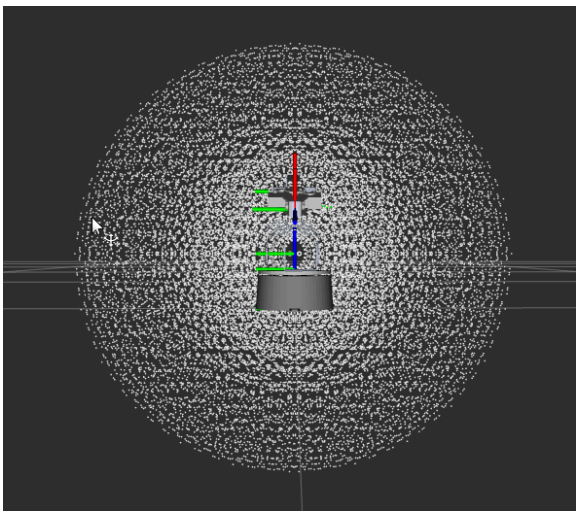
The actual joints of the robot arm have restricted movement due to physical limitations. The limits of each joint, measured with no actuators activated, are shown in Table 1.2.

Table 1.2: joint limits for acutal robot arm in [degree]

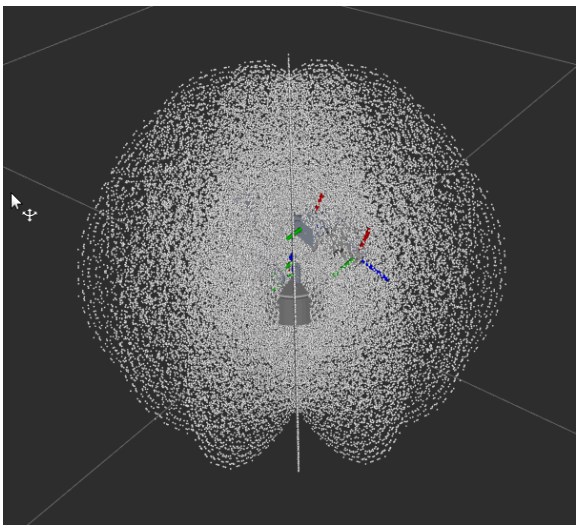
joint	max	min
1	107	-92
2	45	-87
3	68	-45
4	100	-100

1.0.5. Workspace Visualization Without Joint Limits

The workspace was visualized in RVIZ by iterating through a wide range of joint positions between these bounds in a nested loop. As shown in `Henrys_Code/visualizer.py`, the code calculates the forward kinematics at each joint position and compiles an array of Marker objects at each position, which it then publishes as a MarkerArray object to the topic `/workspace_markers`. With a fine enough resolution between points, the edges of the workspace are apparent. To visualize with no bounds, the bounds were set at -180deg to 180deg for all joints. There are two figures generated for the unbound workspace, Figure 1.3a shows a cross-section of the workspace in two dimensions, while Figure 1.3b shows an isometric view of the three-dimensional workspace.



(a) Two-dimensional cross-section of the robot workspace, with no constraints.



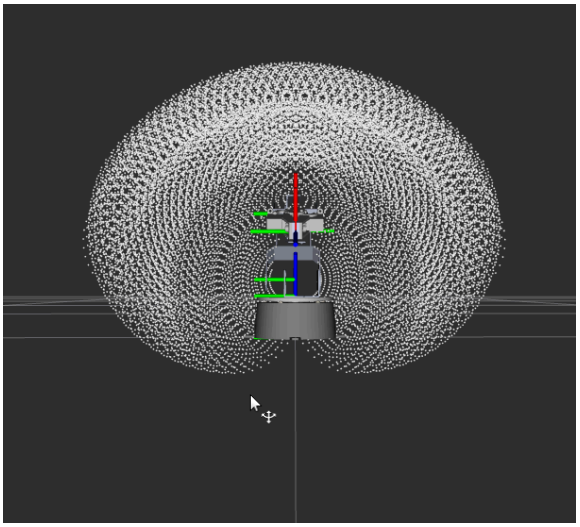
(b) Isometric view of three-dimensional workspace, with no constraints.

Figure 1.3: Comparison of robot workspace views.

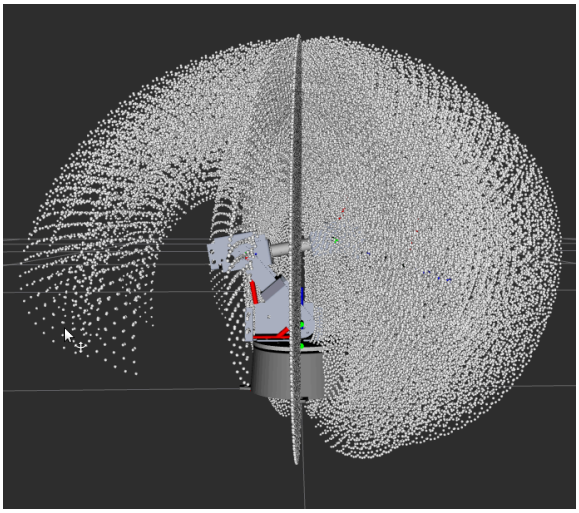
If no joint limits were in effect, the workspace of the robot would essentially be a sphere centering around the joint 1.

1.0.6. Workspace Visualization With Joint Limits

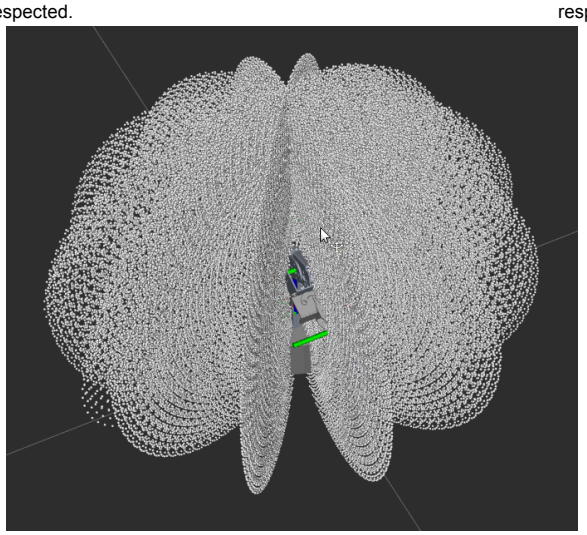
To visualize the workspace with bounds, the actual bounds shown in Table 1.2 were integrated into the `Henrys_Code/robot_arm.py` Edubot class, which enforces joint limits in `Henrys_Code/visualizer.py`. Figure 1.4a shows the 2D cross section of the constrained workspace in the yz plane, while Figure 1.4b shows a side view of the 3D workspace and Figure 1.4c shows an isometric view of the constrained 3D workspace.



(a) Two-dimensional cross-section of the robot workspace, with constraints respected.



(b) Side view of three-dimensional workspace, with constraints respected.



(c) Isometric view of three-dimensional workspace, with constraints respected.

Figure 1.4a shows that a two-dimensional cross-section of the workspace looks like a cardioid that pinches into the origin from the negative z-axis. In addition, the robot has significantly less ability to reach behind the robot, or in the base frame's +x direction.

It is obvious from all three workspace figures that, even when constraints are included, there are some points shown to be reachable that are not functionally reachable, either because they require the robot to pass through itself or go below the x-y plane, which would place them in the ground. These are shown as reachable because the joint bounds were measured only when all other joints were at 0deg deflection. Due to physical limitations on the robot such as its links, its base, the ground, and the wires

and springs that support it, the bounds of each joint change depending on the positions of the other joints. Thus, this workspace is only an estimate of the actual functional workspace, which would be much more constrained, especially near the robot base itself.

2

Inverse Kinematics

Task 2.1

2.1.1. Inverse Kinematics Functions

Three different methods of inverse kinematics were derived, each of which will be examined in this section.

Method 1: Newton-Raphson

The first method uses the Jacobian(see Section 3) with the Newton-Raphson method to iteratively find the inverse kinematics. The Newton-Raphson method, starting with the desired cartesian position and an initial guess(q_{init}), follows these steps:

- **Step 1: Analytically solve for** q_0

  Due to the geometry of the robot, for a given x and y position there is are only two possible q_0 angles that satisfy it. The preferred angle is where the point lies in front of the robot's arm(-x in the $j0$ frame), which is found by the equation

  $$q_0 = tan^{-1}(\frac{-y}{-x}) \tag{2.1}$$

  if this is outside of the j0 bound, the angle is flipped by 180deg so it can be reached "behind" the robot(+x in the j0 frame).

- **Step 2: Solve for** Δx**,** Δq**, and** q_{new}

  Δx can be solved for easily by subtracting the current position, calculated as the forward kinematics of the initial guess, from the desired position. Once Δx is found, Δq can be found using the equation:

  $$\Delta q = J^{-1}\Delta x \tag{2.2}$$

  where J^{-1} is the inverse of the Jacobian. Because it is only solving for the cartesian position and there are four joints, the pseudo-inverse is used instead, found using the Moore-Penrose Inverse. Then the joint positions are updated as such:

  $$q_{new} = q_{init} + \Delta q \tag{2.3}$$

  where q_{new} replaces q_{init} as the new initial guess. Each component of q_{new} is checked to make sure it is within the bounds of each joint, and if it is not it will be clipped so it stays within the possible range of motion of the robot.

- **Step 4: Check Forward Kinematics and Repeat**

Using q_{new}, determine the position of the end-effector with the forward kinematics function. If it is within a designated positional tolerance(default at 1μm, or a designated number of iterations have been reached(default at 1000), the loop will break and return the joint angles to achieve this position. If it is not within this tolerance or past this number of iterations, the process restarts at **Step 1** with q_{new} as the new initial guess.

In code, these steps can be seen in `Henrys_Code/robot_arm.py` and `Henrys_Code/robot_arm.cpp`, belonging to the `Edubot.inverse_kinematics_newton_raphson()` function in both.

Method 2: Analytical(Additional)
Alternatively, the inverse kinematics can be solved analytically if the x, y, z coordinates and the angle θ between the end effector and the xy-plane are given. The angle for each joint, q_0, q_1, q_2, q_3 is determined as follows:

$$q_0 = \text{atan2}(\frac{y}{x}) \tag{2.4}$$

where atan2 stands for two-argument arctangent. Here we define two additional variables to facilitate calculation:

$$L = \sqrt{x^2 + y^2} - l_3 \cdot \cos\theta \tag{2.5}$$
$$H = z - l_0 - l_1 - l_3 \cdot \sin\theta \tag{2.6}$$

then for q_2:

$$\sin q_2 = \frac{L^2 + H^2 - l_1^2 - l_2^2}{-2l_1 l_2} \tag{2.7}$$
$$\cos q_2 = \pm\sqrt{1 - \sin {q_2}^2} \tag{2.8}$$
$$q_2 = \text{atan2}(\frac{\sin q_2}{\cos q_2}) \tag{2.9}$$

here multiple solution may occur due to how we calculate $\cos q_2$. For q_1:

$$q_1 = \text{atan2}(\frac{l_2 \cos q_2}{l_1 - l_2 \sin q_2}) - \text{atan2}(\frac{L}{H}) \tag{2.10}$$

Finally for q_3:

$$q_3 = \theta - q_1 + q_2 \tag{2.11}$$

This part can be found in `Wangs_Code/Solu_Wang.py` within the `inverse_kinematics_analytical` function.

Method 3: Optimization-Based(Additional)
The last method of inverse kinematic solution is optimization. The package `scipy` provides a function `scipy.optimize.minimize()` that uses an objective function to optimize a set of input parameters in order to minimize the value of that objective function. For the inverse kinematics, the objective function is the squared error of the desired position and the actual position, given by the equation:

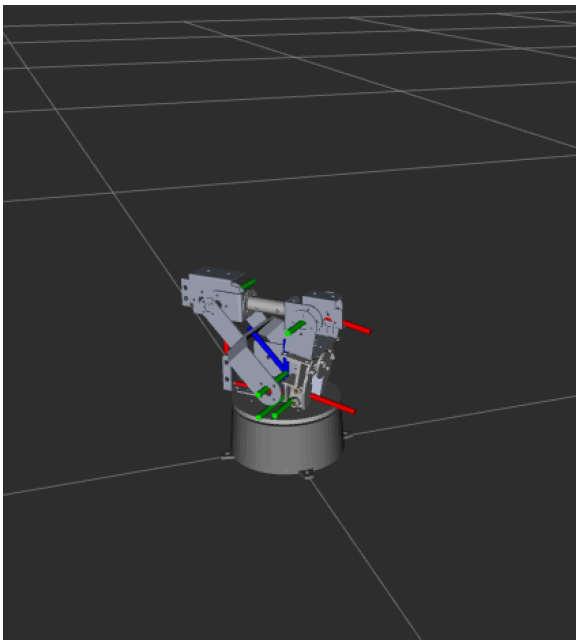
$$obj = (\bar{x}_{desired} - \bar{x}_{actual})^2 \tag{2.12}$$

The parameters are the joint angles, and the objective function calculates $\bar{x}_{actual}$ by finding the inverse kinematics from the current joint angles. Although this method does work well for the simpler and easily reachable positions, it can get caught in local minima when dealing with points further away from its starting position and it takes longer to solve. In addition, it provides much less information about the outcome, for example whether the solution is in a singularity or not. The optimization code can be seen in `Henrys_Code/robot_arm.py` within the `Edubot.inverse_kinematics_optimization()` function.

2.1.2. Testing The Inverse Kinematics

In order to check the inverse kinematics of the robot and its integration of the joint bounds, four points are chosen to run through the inverse kinematics algorithm in the file `Henrys_Code/IK_tester.py`:

1. **Point 1: [0.1, 0.1, 0.1]** Though this point would be reachable if the base joint j0 were unconstrained, the fact that the positive x direction is functionally ”behind” the robot makes this point unreachable with existing joint bounds. The inverse kinematics function does attempt to find the closest value using the Newton-Raphson method, and the eventual solution is seen in Figures 2.1a and 2.1b, which show the simulated and actual positions of the robot respectively.

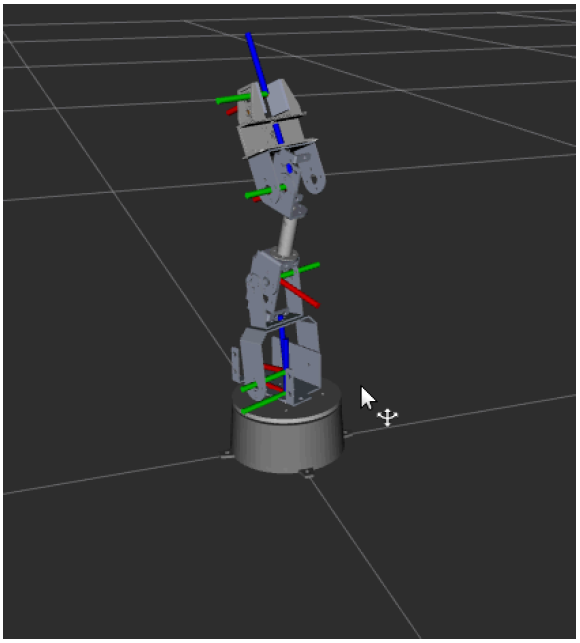


(a) Visualization of Point 1 in the simulator.

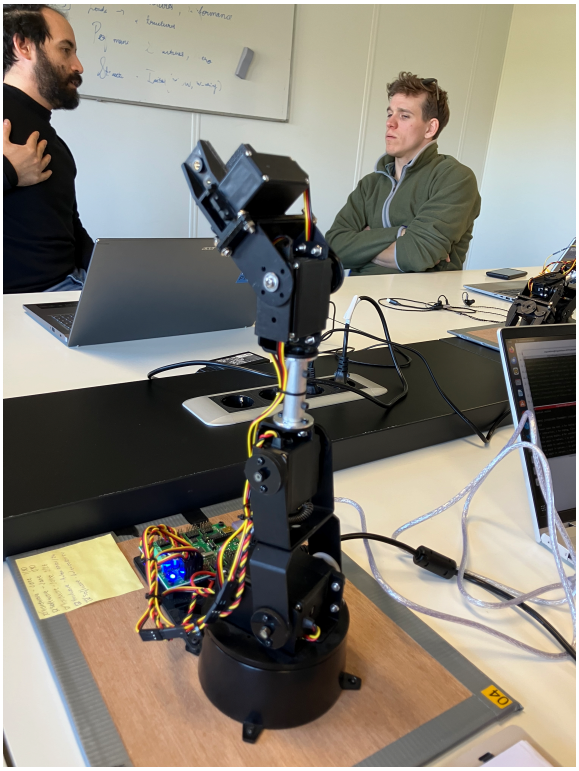
(b) Visualization of Point 1 on the robot itself.

The function returns the joint array [0.785 1.571 -0.786 -1.745], which results in the end-effector at the point [0.0642, 0.0642, 0.187].

2. **Point 2: [0.2, 0.1, 0.3]** Point 2, lying beyond the outer limits of the workspace of the robot, is also unreachable. Again, the inverse kinematics function finds the closest point in the reachable workspace, as shown in Figures 2.2a and 2.2b for the simulated and real life robot respectively.



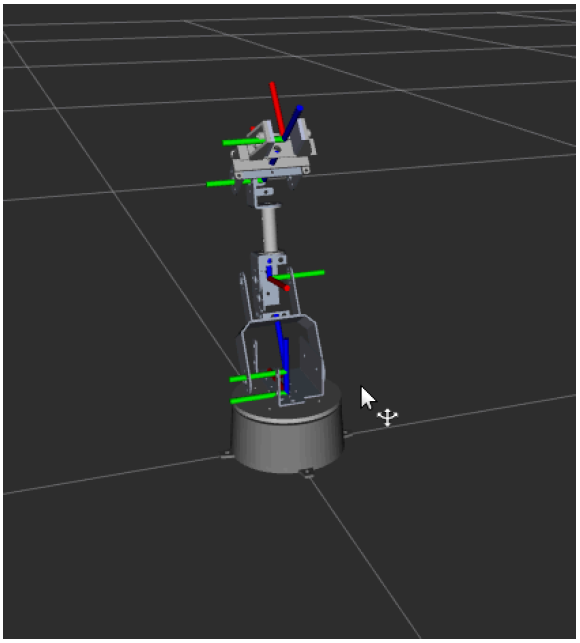
(a) Visualization of Point 2 in the simulator.



(b) Visualization of Point 2 on the robot itself.

The function returns the joint array [0.46 1.35 -1.06 -1.04], which results in the end-effector at the point [0.124, 0.0622, 0.235].

3. **Point 3: [0.0, 0.0, 0.3]** Point 3 is reachable, although it is in singularity position as it is aligned with j0, so there are infinitely many positions. The inverse kinematics function takes the solutions at $j_0 = 0$, which results in the position shown in Figures 2.3a and 2.3b.



(a) Visualization of Point 3 in the simulator.

(b) Visualization of Point 3 on the robot itself.

The function returns the joint array [0.0, 0.896, -0.333, 0.0449], which, since this is a reachable point, results in the end-effector at the point [0, 0, 0.3].

4. **Point 4: [0.0, 0.0, 0.07]** If there were no physical constraints on the robot, Point 4 would also be a reachable position. Unfortunately, though, the robot would have to pass through itself to get to this position, since the position is inside of the robot base. This position was not visualized on the actual robot as the inverse kinematics would put the gripper into the robot, but the simulated result can be seen in 3.1.

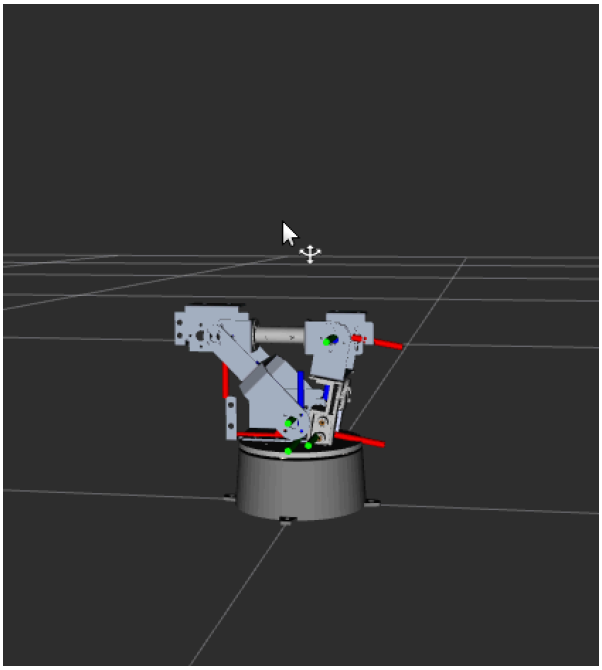
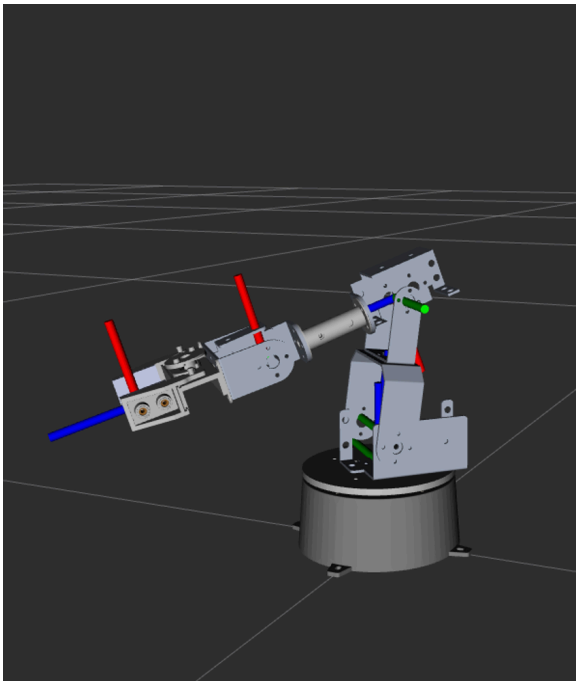


Figure 2.4: Point 4 visualized in the simulator.

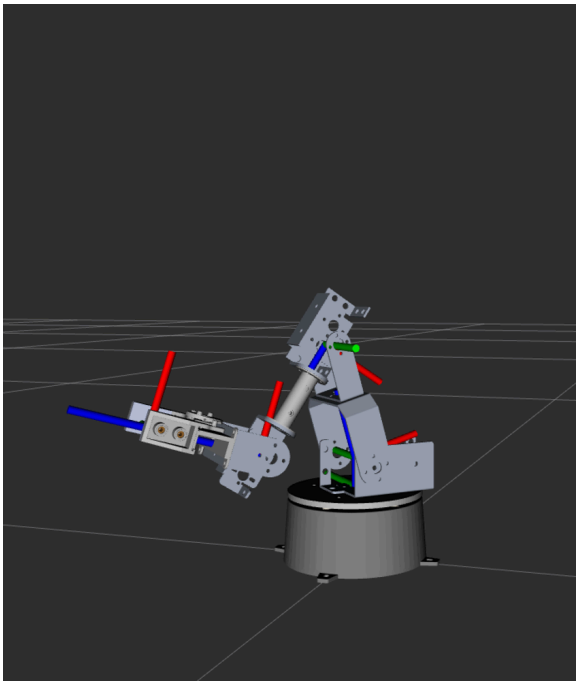
The function returns the joint array [0.0, 1.571, 0.681, -1.745], which results in the end-effector at the point [-0.037, 0.0, 0.095].

2.2. Elbow Up and Elbow Down Solutions

One important aspect of robot arms is that there are often multiple ways to reach the same position. In some cases, these are known as the ”Elbow Up” and ”Elbow Down” positions. To illustrate this on the Edubot, points 1 and 3 were recreated in the simulation with their elbow up and down solution. For point 1, joint bounds were ignored for the sake of clear visualization. Figures 2.5a and 2.5b show the up and down configurations for point 1.

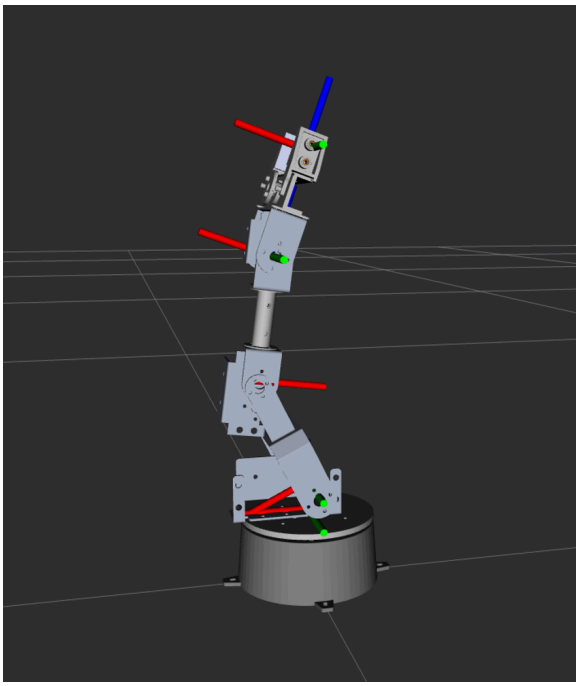


(a) Visualization of Elbow-Up solution to point 1 inverse kinematics.

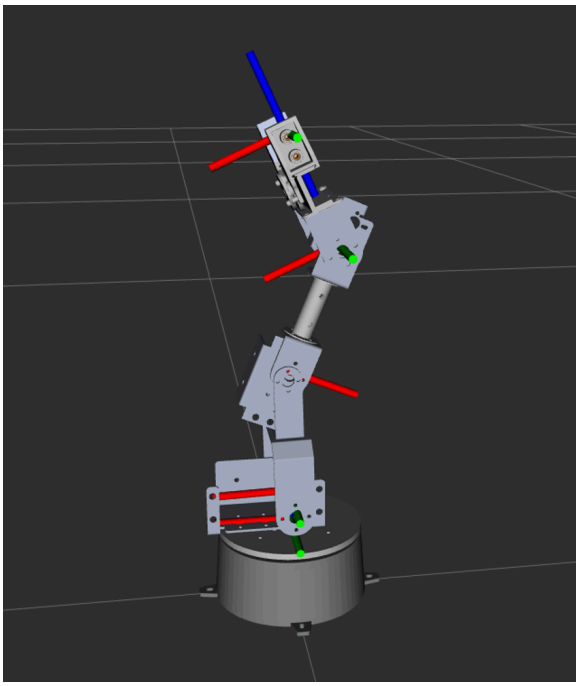


(b) Visualization of Elbow-Down solution to point 1 inverse kinematics.

Figures 2.6a and 2.6b show the same for point 3. It is important to note that, for point 3, arguably there is no elbow ”up” and ”down” solution as the base joint can be rotated to any position and it will be at the same point.



(a) Visualization of Elbow-Up solution to point 3 inverse kinematics.



(b) Visualization of Elbow-Down solution to point 3 inverse kinematics.

Depending on what you consider the ”elbow” in this scenario, the feasibility changes. If you consider j_2 the elbow, the placemenet of the spring and the limits of the joints on the edubot prevent any elbow down solutions. If, on the other hand, you consider j_3 the elbow, as the above figures do, both configurations are feasible, as the figures show.

2.3. Trajectory Following Task

The goal of this task is to trace out the TU Delft flame with the end-effector, which can then be recorded and post-processed to display the traced flame itself. The task can be broken down into four steps(all of the following code is in the `Henrys_Code` folder):

- **Step 1: Turn the outline into a cartesian trajectory**

  In order to do get a cartesian trajectory from the picture of the TU Delft flame, OpenCV is used to first turn the image to grayscale, then use Canny edge detection to find the pixel coordinates of the perimeter of the flame.

  After these pixel coordinates are found, they are turned into cartesian coordinates by specifying a lower bound for the end effector trajectory and scaling the flame to be 20cm tall.

  This step is seen in the `Edubot.trace_cartesian_trajectory()` function inside `robot_arm.py` file.

- **Step 3: Turn the cartesian trajectory into a joint trajectory**

  In order to turn this cartesian trajectory into a joint trajectory, each point of the cartesian trajectory is put through the inverse kinematics function and the joint trajectory is saved in a .csv file. This step is seen in the `Edubot.cartesian_to_joint_trajectory()` function inside `robot_arm.py` file.

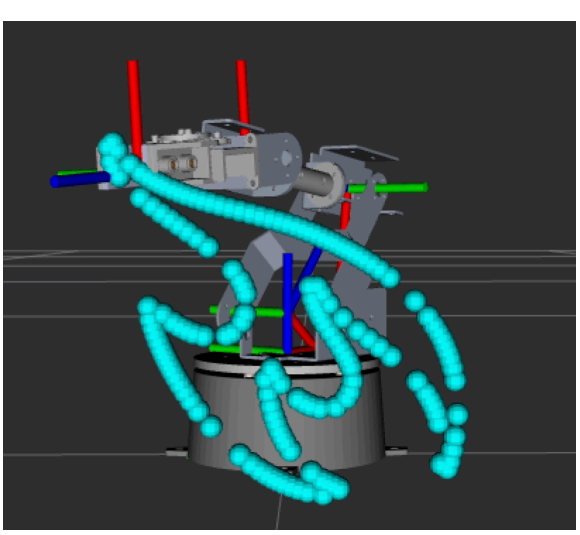
- **Step 4: Execute the joint trajectory**

  The last step on the actual robot is to iterate through this joint trajectory and actually move the robot arm to the specified positions. This was done by reading the .csv file with the pandas library, then creating a ROS timer object that will move from one joint position to the next every 0.1s. For the simulation, it also publishes a cyan marker for each position it moves to to visualize the trajectory. This can be seen in the `trajectory_follower.py` file.

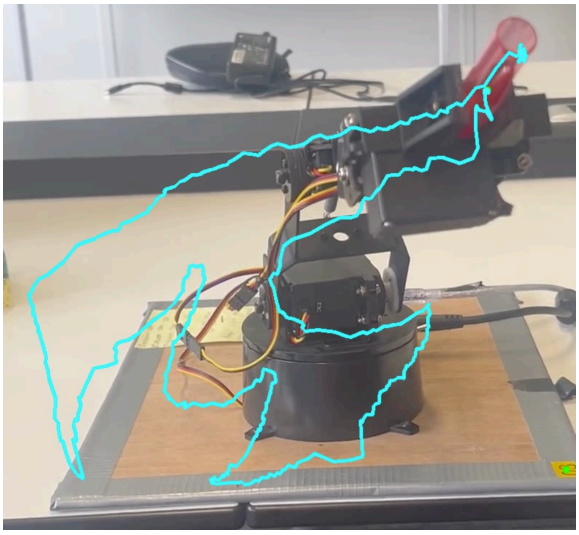
- **Step 5: Post-processing**

  For visualizing the trajectory on the physical robot, some post-processing had to be done on the video taken. OpenCV was used to stabilize the video then track a red pen cap being gripped by the end-effector to trace the trajectory. This can be seen in the `stabilize_and_trace_video.py` file.

The trajectory was carried out in the simulation and on the actual robot. The simulation can be seen in the `Videos/simulated_flame.mov` video, while the actual robot can be seen in the `Videos/actual_flame.mp4` video. The final drawings for the flame can be seen in Figure 2.7a for the simulator and Figure 2.7b for stabilized video of the actual robot.



(a) TU Delft Flame Traced by Edubot Inside RVIZ Simulation. See simulated_flame.mov for full video.



(b) Post-Processed TU Delft Flame Traced by Edubot in Real Life. See actual_flame.mp4 for full video.

Figure 2.7: Comparison of robot workspace views.

3

The Jacobian

3.1. Finding the Symbolic Jacobian

3.1.1. Derivative Definition Method

Another way of getting the Jacobian is by using its definition to derive it from the symbolic forward kinematics expression. The Python `sympy` symbolic math library supports derivative math, so it is easy to convert from the symbolic forward kinematics by conducting the partial derivatives as follows:

$$J = \begin{bmatrix} \frac{\partial x}{\partial q_0} & \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} \\ \frac{\partial y}{\partial q_0} & \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} \\ \frac{\partial z}{\partial q_0} & \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} \end{bmatrix} = \begin{bmatrix} J11 & J12 & J13 & J14 \\ J21 & J22 & J23 & J24 \\ J31 & J32 & J33 & J34 \end{bmatrix}$$

Solving the symbolic expression for each element of the Jacobian using the forward kinematics derived in Equation 1.6, it can be simplified and expressed as follows:

$$\begin{aligned} &S_0 = \sin q_0, \quad C_0 = \cos q_0, \quad S_1 = \sin q_1, \quad C_1 = \cos q_1, \\ &S_2 = \sin q_2, \quad C_2 = \cos q_2, \quad S_3 = \sin q_3, \quad C_3 = \cos q_3, \\ &T_1 = S_1 S_2 + C_1 C_2, \quad T_2 = S_1 C_2 - S_2 C_1. \end{aligned}$$

$$\begin{aligned} J_{11} &= 0.075(S_0 T_1 C_3 - S_0 T_2 S_3) + 0.105 S_0 T_1 - 0.095 S_0 S_1 + 0.006 S_0, \\ J_{12} &= -0.075(C_0 T_1 S_3 - C_0 T_2 C_3) + 0.105 C_0 T_2 - 0.105 C_0 T_1 + 0.095 C_0 C_1, \\ J_{13} &= -0.075(C_0 T_1 S_3 - C_0 T_2 C_3) - 0.105 C_0 T_1 + 0.105 C_0 T_2, \\ J_{14} &= -0.075(C_0 T_1 S_3 + C_0 T_2 C_3), \\ J_{21} &= 0.075(-C_0 T_1 C_3 + C_0 T_2 S_3) - 0.105 C_0 T_1 + 0.095 C_0 S_1 - 0.006 C_0, \\ J_{22} &= -0.075(S_0 T_1 S_3 - S_0 T_2 C_3) + 0.105 S_0 T_2 - 0.105 S_0 T_1 + 0.095 S_0 C_1, \\ J_{23} &= -0.075(S_0 T_1 S_3 - S_0 T_2 C_3) - 0.105 S_0 T_1 + 0.105 S_0 T_2, \\ J_{24} &= -0.075(S_0 T_1 S_3 + S_0 T_2 C_3), \\ J_{31} &= 0, \\ J_{32} &= 0.075(T_1 C_3 - T_2 S_3) + 0.105 T_1 - 0.095 S_1, \\ J_{33} &= 0.075(-T_1 C_3 + T_2 S_3) - 0.105 T_1, \\ J_{34} &= -0.075(-T_1 C_3 - T_2 S_3). \end{aligned}$$

This agrees with the expression found using the DH convention. The full derivation with the sympy toolbox can be seen in `Henrys_Code/robot_arm.py` within the `Edubot.get_jacobian()` function.

3.1.2. Numerical Jacobian Method(Alternative)

The Jacobian can also be computed using forward kinematics and numerical differentiation. The process is summarized as follows: First,the current end effector's position x, y, z, orientation θ, and joint angles q_0, q_1, q_2, q_3 are recorded; then we add a small increment δ to each joint angle at a time to obtain four new angle sets p_i, $i \in \{0, 1, 2, 3\}$. For example, p_1 stands for $\{q_0, q_1 + \delta, q_2, q_3\}$. For each angle sets, we apply forward kinematics and obtaining new positions x_i, y_i, z_i, where i stands for angle sets used. Thus we have the Jacobian for positions. For end effector's orientation, we notice:

$$\theta = q_1 - q_2 + q_3 \tag{3.1}$$

Together, we have the 4×4 Jacobian Matrix:

$$\begin{bmatrix} \frac{x_0-x}{\delta} & \frac{x_1-x}{\delta} & \frac{x_2-x}{\delta} & \frac{x_3-x}{\delta} \\ \frac{y_0-y}{\delta} & \frac{y_1-y}{\delta} & \frac{y_2-y}{\delta} & \frac{y_3-y}{\delta} \\ \frac{z_0-z}{\delta} & \frac{z_1-z}{\delta} & \frac{z_2-z}{\delta} & \frac{z_3-z}{\delta} \\ 0 & 1 & -1 & 1 \end{bmatrix} \tag{3.2}$$

This part can be found in `Wangs_Code/Solu_Wang.py` within the `cal_jacobian` function.

3.1.3. Denavit Hartenberg method

The Denavit-Hartenberg (DH) convention is a widely used robotics methodology for modeling serial manipulators' kinematic structure. It simplifies the mathematical representation of each joint and link by standardizing how coordinate frames are assigned and how transformations between them are described.

Each joint/link in a robotic arm is described using four parameters:

1. θ – Joint Angle

- **Definition**: Rotation about the z-axis from the previous x-axis to the current x-axis.
- **Variable for**: Revolute joints.
- **Constant for**: Prismatic joints.
- **Determination**: Measured as the angle around the z-axis needed to align the previous x-axis with the current one.

2. d – Link Offset

- **Definition**: Distance along the z-axis from the previous frame's origin to the point where the x-axes intersect.
- **Variable for**: Prismatic joints.
- **Constant for**: Revolute joints.
- **Determination**: Measured along the z-axis from one frame's origin to where the common normal intersects.

3. a – Link Length

- **Definition**: Distance along the x-axis from one z-axis to the next.
- **Always constant.**
- **Determination**: Measured along the x-axis between two successive z-axes.

4. α – Link Twist

- **Definition**: Angle about the x-axis needed to rotate the previous z-axis to align with the current z-axis.
- **Always constant.**
- **Determination**: Measured as the rotation required around the x-axis to align the z-axes.

Below is the Denavit-Hartenberg parameter table for the given robot configuration:

Link	a_i	d_i	α_i	θ_i
1	0	l_1	0	0
2	0	l_2	$-\frac{\pi}{2}$	q_1
3	l_3	0	0	$q_2 + \frac{\pi}{2}$
4	l_4	0	0	q_3
5	l_5	0	0	q_4

resulting in the following:

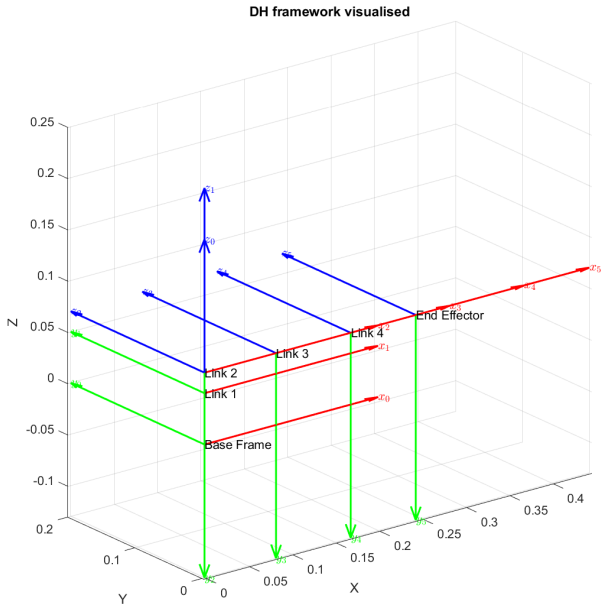


Figure 3.1: Point 4 visualized in the simulator.

Where O_0 is a right-handed base frame coordinate system that the rest of the robot moves with respect to. Red represents the $x-$ axis, green the $y-$ axis and blue the $z-$ axis. Using the values found for each of the parameters and the following formula:

$$T_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Using this, the transformation matrix T_5^0 is found symbolically:

$$T_5^0 = \begin{bmatrix} \cos(q_1)\cos(\sigma_3) & -\cos(q_1)\sin(\sigma_3) & -\sin(q_1) & \cos(q_1)\sigma_2 \\ \sin(q_1)\cos(\sigma_3) & -\sin(q_1)\sin(\sigma_3) & \cos(q_1) & \sin(q_1)\sigma_2 \\ -\sin(\sigma_3) & -\cos(\sigma_3) & 0 & l_1 + l_2 - \sigma_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} \sigma_1 &= l_4\sin(\sigma_4) + l_5\sin(\sigma_3) + l_3\sin\left(\frac{\pi}{2} + q_2\right) \\ \sigma_2 &= l_4\cos(\sigma_4) + l_5\cos(\sigma_3) + l_3\cos\left(\frac{\pi}{2} + q_2\right) \\ \sigma_3 &= \frac{\pi}{2} + q_2 + q_3 + q_4 \\ \sigma_4 &= \frac{\pi}{2} + q_2 + q_3 \end{aligned} \tag{3.4}$$

It should be noted that due to the different coordinate frames used, equation 3.3 differs from 1.6.

$$J = \begin{bmatrix} \sin(q_1)\,\sigma_1 & -\cos(q_1)\,\sigma_2 & -\cos(q_1)\,(0.075\,\sigma_5 + \sigma_6) & -0.075\,\sigma_5\,\cos(q_1) \\ -\cos(q_1)\,\sigma_1 & -\sin(q_1)\,\sigma_2 & -\sin(q_1)\,(0.075\,\sigma_5 + \sigma_6) & -0.075\,\sigma_5\,\sin(q_1) \\ 0 & \sigma_1 & 0.075\,\sigma_3 + \sigma_4 & 0.075\,\sigma_3 \end{bmatrix}$$

where

$$\begin{aligned} \sigma_1 &= 0.075\,\sigma_3 + \sigma_4 + 0.083\,\sin(q_2) \\ \sigma_2 &= 0.075\,\sigma_5 + \sigma_6 + 0.083\,\cos(q_2) \\ \sigma_3 &= \sin(q_2 + q_3 + q_4) \\ \sigma_4 &= 0.086\,\sin(q_2 + q_3) \\ \sigma_5 &= \cos(q_2 + q_3 + q_4) \\ \sigma_6 &= 0.086\,\cos(q_2 + q_3) \end{aligned} \tag{3.5}$$

Finally, the Jacobian is found:

$$J = \begin{bmatrix} \frac{\partial x}{\partial q_0} & \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} \\ \frac{\partial y}{\partial q_0} & \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} \\ \frac{\partial z}{\partial q_0} & \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} \end{bmatrix} = \begin{bmatrix} J11 & J12 & J13 & J14 \\ J21 & J22 & J23 & J24 \\ J31 & J32 & J33 & J34 \end{bmatrix} \tag{3.6}$$

$$\begin{aligned} J_{11} &= \sin(q_1)\,(0.075\sin(q_2 + q_3 + q_4) + 0.086\sin(q_2 + q_3) + 0.083\sin(q_2)) \\ J_{12} &= -\cos(q_1)\,(0.075\cos(q_2 + q_3 + q_4) + 0.086\cos(q_2 + q_3) + 0.083\cos(q_2)) \\ J_{13} &= -\cos(q_1)\,(0.075\cos(q_2 + q_3 + q_4) + 0.086\cos(q_2 + q_3)) \\ J_{14} &= -0.075\cos(q_1)\cos(q_2 + q_3 + q_4) \\ J_{21} &= -\cos(q_1)\,(0.075\sin(q_2 + q_3 + q_4) + 0.086\sin(q_2 + q_3) + 0.083\sin(q_2)) \\ J_{22} &= -\sin(q_1)\,(0.075\cos(q_2 + q_3 + q_4) + 0.086\cos(q_2 + q_3) + 0.083\cos(q_2)) \\ J_{23} &= -\sin(q_1)\,(0.075\cos(q_2 + q_3 + q_4) + 0.086\cos(q_2 + q_3)) \\ J_{24} &= -0.075\sin(q_1)\cos(q_2 + q_3 + q_4) \\ J_{31} &= 0 \\ J_{32} &= 0.075\sin(q_2 + q_3 + q_4) + 0.086\sin(q_2 + q_3) + 0.083\sin(q_2) \\ J_{33} &= 0.075\sin(q_2 + q_3 + q_4) + 0.086\sin(q_2 + q_3) \\ J_{34} &= 0.075\sin(q_2 + q_3 + q_4) \end{aligned} \tag{3.7}$$

Again it should be noted that different coordinate frames cause different values Jacobians.

3.2. Examining the Jacobian at Infeasible Positions

An infeasible position can cause the Jacobian to become singular or nearly singular, meaning it loses rank (e.g., the determinant is zero for a square Jacobian). This happens because, at or near the boundary of the robot's workspace—or at certain configurations like singularities—the columns of the Jacobian become linearly dependent. Physically, this corresponds to the robot losing dexterity: it can no longer move the end-effector freely in all directions.

3.3. Constant Velocity Movement

Given a desired direction and distance, the function will use the predetermined small step size ds as desired constant Cartesian velocity $\dot{\mathbf{x}}$. The current joint angles $\mathbf{q}$ are continuously updated. These joint values are then substituted into the robot's Jacobian matrix, and its pseudo inverse is computed numerically. This numerical approach is used in place of a symbolic pseudo inverse due to the high computational cost of calculating a symbolic pseudo inverse.

With the Jacobian pseudo inverse and the constant Cartesian velocity $\dot{\mathbf{x}}$, the joint angular velocities $\dot{\mathbf{q}}$ are computed using the relation:

$$\dot{\mathbf{x}} = J\dot{\mathbf{q}} \quad \Rightarrow \quad \dot{\mathbf{q}} = J^{-1}\dot{\mathbf{x}} \tag{3.8}$$

Here $\dot{\mathbf{q}}$ is published as small joint angle increment to keep the end effector moving at constant speed. This process is repeated iteratively until the end effector Achieved the planned trajectory. The robot's performance can be found in `Videos/Constant_Speed.mov`. In this video, the end effector moves sequentially in z, x, y.

4

Pick and Place

4.1. Basic Pick and Place

Once the forward and inverse kinematics are established, we can command the robot arm to reach any position within its reachable workspace by sending appropriate joint commands. However, a major challenge arises due to discrepancies between the simulation model and the actual robot arm, such as variations in arm lengths, joint offsets, motor drift, and other factors. To address this, we implemented a teleoperation system that continuously listens for keyboard inputs and sends commands to the robot arm while also monitoring its current state to track its position in Cartesian space. In this way we were able to map real life positions to robot's work space with minimal error. The teleopration code can be found in `Henrys_Code/teleoperator.py`.

Another challenge occurs during the trajectory of moving the end effector to the object's original position and returning to the home position after placing the object. During these movements, the end effector might accidentally collide with the object. To solve this issue, we divide the trajectory into multiple stages: first, the arm moves horizontally, followed by a vertical motion when picking the object, and similarly, when returning the arm to the home position, it first moves vertically and then horizontally. This approach reduces the likelihood of collisions during the operation.

This part can be found in `Wangs_Code/Solu_Wang.py` within the `pick_and_place` function. Robot's performance in real life can be found in `Videos/Pick_Place.mov` and `Videos/Pick_Place_reverse.mov`

4.2. Berry Task

The main challenge in this task, aside from the two previously mentioned, is to ensure the berry remains intact during the picking process. To address this, we programmed the gripper to move slowly and estimated the berry size to adjust the gripper's openness accordingly. If the gripper is equipped with a force sensor, advanced control methods could further enhance the process by providing feedback for more delicate handling.

This part also used the `pick_and_place` function mentioned above but with some modification. Robot's performance can be found in `Videos/PickBerry.mov`.

4.3. Wiping Task

In addition to the challenges discussed in Section 4.1, the primary issue here is designing an effective wiping program that ensures the end effector stays within the workspace during the wiping motion. Furthermore, the end effector's orientation is constrained due to the limited DoF. Increasing the number of DoF would provide greater flexibility in movement, allowing for more precise and adaptable wiping actions. Robot's performance can be found in `Videos/Wipe.mov`.

This part can be found in `Wangs_Code/Solu_Wang.py` within the `wipe` function.

5

Conclusion

This report explores the forward and inverse kinematics solutions, trajectory tracking tasks, and Jacobian matrix computation for a 4-DoF robot arm, along with real-world applications. Various inverse kinematics methods, including the Newton-Raphson iterative method, analytical solutions, and optimization-based approaches, were introduced and validated through test cases. The feasibility and singularity of the robot under different constraints were analyzed, considering the impact of ”elbow-up” and ”elbow-down” configurations on the available solutions.

The report also details the execution of the TU Delft flame trajectory tracking task, covering image processing for trajectory extraction, joint-space trajectory computation, motion command execution, and post-processing visualization. This process validated the accuracy of the inverse kinematics solutions and demonstrated the feasibility of trajectory control for the robot.

In the Jacobian matrix computation section, three methods were explored: the Denavit-Hartenberg (DH) parameter approach, symbolic differentiation, and numerical differentiation. Their derivations were explained, and results were compared to evaluate their effectiveness. These methods provide a solid foundation for future applications in motion planning, force control, and singularity analysis.

Furthermore, practical applications were implemented using the inverse kinematics solutions, including pick-and-place tasks, berry picking, and surface wiping. During these implementations, several challenges were encountered, and effective solutions were developed to improve task execution.

Overall, this study establishes a comprehensive framework for the kinematic modeling and control of a 4-DoF robot arm. Future work could focus on optimizing motion planning algorithms and exploring more complex trajectory tracking tasks to further enhance the robot’s practical applications.