Robotics 2 – Coursework 1: Model

Authors: Ling Liu, Xinyue Zhang, Zhihan Wang, Zhuoxiaoyue Wang

Task A: Compute the initial D-H table of the robot arm

The Denavit-Hartenberg (D-H) parameter table was analysed by drawing the axes following the guidelines above. Each row corresponds to a joint or link of the robot. For each joint or link, four parameters need to be defined: d, θ, a, and α. The parameters are described as follows:

- a_i *: The distance from* $\hat{z}_{i-1}$ *to* $\hat{z}_i$ *measured along* $\hat{x}_{i-1}$.
- α_i *: The angle between* $\hat{z}_{i-1}$ *and* $\hat{z}_i$ *measured about* $\hat{x}_{i-1}$.
- d_i*: The distance from* $\hat{x}_{i-1}$ *to* $\hat{x}_i$ *measured along* $\hat{z}_i$.
- θ_i *: The angle between* $\hat{x}_{i-1}$ *and* $\hat{x}_i$ *measured about* $\hat{z}_i$.

We analysed the D-H table for each joint as follows:

1. From frame 0 to frame 1:

The distance from x_0 to x_1 is moved along z_0. There is no rotation or angle change in α or θ, and no distance is moved along x_0. Thus, the first row of the D-H table is $[l_0, 0,0,0]$.

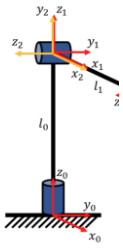


Figure 1: frame 0 and frame 1

2. From frame 1 to frame 2:

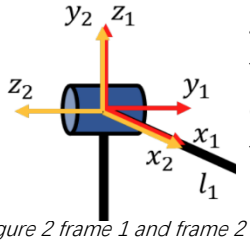


Figure 2 frame 1 and frame 2

The robot rotates, and the angle between $x_1$and z_2 is measured about x_1. The rotation is counterclockwise relative to the x_1 axis, using the right-hand rule, if you curl your fingers in the direction of rotation from z_1 to z_2 around x_1, your thumb points along the positive x_1 axis. This means the angle is **positive**, hence $\pi/2$. Thus, the second row the D-H table is $[0,0,0,\pi/2]$.

3. From frame 2 to frame 3:

The distance from z_2 to z_3 is moved along x_2. There is no rotation or angle change in α or θ, and no distance is moved along z_2 . Thus, the third row of the D-H table is $[0,0,l_1,0]$.

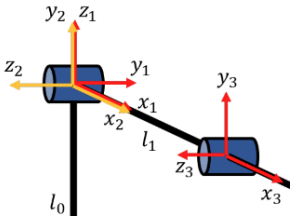
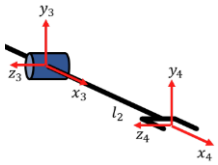


Figure 3 frame 2 and frame 3

4. From frame 3 to frame 4 (end effector):



Similarly, the distance from z_3 to z_4 is moved along x_3. Again, there is no rotation or angle change in a or θ, and no distance is moved along z_2 . Thus, the fourth row of the D-H table is $[0,0,l2,0]$.

Figure4: frame 3 and frame4

Table 1: D-H Table

i	di	θi	ai	αi
1	$l0$	0	0	0
2	0	0	0	$pi/2$
3	0	0	$l1$	0
4	0	0	$l2$	0

```python
class RobotKineClass():

    def __init__(self,link_lengths):

        self.ROSPublishers = set_joint_publisher()

        self.nj = 3    #number of joints
        self.links = link_lengths    # length of links

        ############################################ TASK 1
        #Define DH table for each link. DH_tab in R^njx4
        #d,theta,a,alpha
        # - d: Offset along the z-axis (link offset)
        # - theta: Angle around the z-axis (joint angle for revolute joints)
        # - a: Length along the x-axis (link length)
        # - alpha: Twist angle around the x-axis
        self.DH_tab = np.array([[self.links[0], 0, 0, 0],   # Offset along z-axis is l0, no rotation
                                [0, 0, 0, pi/2],            # Perpendicular twist of pi/2, no offset
                                [0, 0, self.links[1], 0],   # Link length is l1, no twist or offset
                                [0, 0, self.links[2], 0]])  # Link length is l2, no twist or offset
        self.joint_types = 'rrr'    # three revolute joints
```

Figure 5: Code for task A

Code Explanation:

Class Definition:

Robot KineClass(): It initializes the robot's Denavit-Hartenberg (D-H) table, the number of joints, link lengths, and joint types.

Constructor Method:

- ***__init__(self, link_lengths)*** *: The constructor initializes the properties of the robot kinematics class.*
- ***self.ROSPublishers = set_joint_publisher():*** *Setting up Ros Publishers, will send joint angles or positions to the robot*
- ***self.nj = 3*** *The robot has 3 joints*
- ***self.links = link_lengths:*** *This stores the length of the robot's links, passed as a parameter (link_lengths).*
- ***self.DH_tab:*** *The D-H table is defined as a n×4 matrix, use the D-H table analysed above.*

  Self.link[0]: self.links[0] refers to the first element of the self.links array, which stores the lengths of the robot's links.

Task B: Complete the code for the D-H matrix

The matrix $T_{i-1}^{i}$is a 4x4 homogeneous transformation matrix used to describe the rotation and translate from coordinate frame $i-1$ to coordinate frame i.

It consists of two main components:

Rotation Matrix R_{i-1}^{i}: Describes the rotation from coordinate frame $i-1$ to coordinate frame i.

Translation Vector P_{i-1}^{i}: Describes the translation of the origin from frame $i-1$ to frame i.

$$T_{i-1}^{i} = \begin{bmatrix} R_{i-1}^{i} & P_{i-1}^{i} \\ 0 & 1 \end{bmatrix}$$

R_{i-1}^{i} is a 3x3 rotation matrix; $P_{i-1}^{i}$is a 3x1 translation vector. The last row [0 0 0 1] is added to homogenize the matrix, allowing for easier matrix multiplication.

To transform from frame $i-1$ to frame i, we use the following transformations:

1. ***Rotation about the*** $\hat{z}_{i-1}$ ***axis by*** θ_i***:***

$$R_z(\theta_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. ***Translation along the*** $\hat{z}_{i-1}$ ***axis by*** d_i

$$T_z(d_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. *Translation along the* $\hat{x}_i$ *-axis by* a_i*:* *4. Rotation about the* $\hat{x}_i$ *-axis by* α_i

$$T_x(a_i) = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_x(\alpha_i) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combing the full transformation T_{i-1}^{i}, we multiply the matrices in the order:

$$T_{i-1}^{i} = Rz(\theta i) \cdot Tz(di) \cdot Tx(ai) \cdot Rx(\alpha i)$$

The final matrix incorporates the rotations and translations:

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

```python
#DH_params = parameters for link i
#d,theta,a,alpha
def DH_matrix(DH_params):
    d = DH_params[0]
    theta = DH_params[1]
    a = DH_params[2]
    alpha = DH_params[3]

    ############################################## TASK 2
    #DH-maxtrix function calculate the transformation matrix based on the Denavit-Hartenberg parameters.
    DH_matrix = np.array([
        [np.cos(theta), -np.sin(theta), 0, a],
        [np.sin(theta)*np.cos(alpha), np.cos(theta)*np.cos(alpha), -np.sin(alpha), -d*np.sin(alpha)],
        [np.sin(theta)*np.sin(alpha), np.cos(theta)*np.sin(alpha), np.cos(alpha), d*np.cos(alpha)],
        [0, 0, 0, 1]
    ])

    return DH_matrix

```

Figure 6: Code for Task B

Task C: Complete the code to compute forwards kinematics

The parameters of each row in the D-H table are substituted into the standard D-H transformation matrix formula. The transformation matrices of all joints are then multiplied to compute the final pose. $T_{\{0\}}^{\{n\}} = T_{\{0\}}^{\{1\}} \cdot T_{\{1\}}^{\{2\}} \cdot \ldots \cdot T_{\{n-1\}}^{\{n\}}$

```python
    #Computes Forward Kinematics. Returns 3x1 position vector
    def getFK(self,q):

        T_0_i_1 = np.identity(4)
        # Loop through each joint in the robot
        for i in range(self.nj):

            DH_params = np.copy(self.DH_tab[i,:])
            #print('q',q)
            #print(DH_params)
            if self.joint_types[i] == 'r': # Revolute joint: add the joint angle to theta (DH_params[1])
                DH_params[1] = DH_params[1]+q[i]

            elif self.joint_types[i] == 'p':# add the joint displacement to d (DH_params[0])
                DH_params[0] = DH_params[0]+q[i]

            T_i_1_i = DH_matrix(DH_params) #Pose of joint i wrt i-1

            ############################################## TASK 3 (replace np.eye(4) with the correct matrices)
            # Multiply the current transformation matrix with the accumulated base-to-(i-1) transformation
            T_0_i = np.matmul(T_0_i_1, T_i_1_i) #Pose of joint i wrt base

            # Update the base-to-(i-1) transformation matrix for the next iteration
            T_0_i_1 = T_0_i

        # Store the transformation matrix from the base to the last joint (n-1)
        T_0_n_1 = T_0_i
        # Get the D-H parameters for the end-effector from the last row of the D-H table
        DH_params = np.copy(self.DH_tab[self.nj, :])
        # Compute the transformation matrix from the last joint frame (n-1) to the end-effector frame (n)
        T_n_1_n = DH_matrix(DH_params)
        # Multiply the transformations to get the pose of the end-effector with respect to the base frame
        T_0_n = np.matmul(T_0_n_1, T_n_1_n)

        return T_0_n[0:3,3]
```

Figure 7: Code for Task C

Each matrix combines the transformation of the previous link with the transformation of the current link. The pose of each joint (three in total) of the robotic arm is calculated iteratively. The final result describes the position and orientation of the end effector relative to the base frame.

Code Explanation:

Initialisation:

T_0_i_1 is initialised as the identity matrix because the first joint in on the base frame.

Loop for each Joint: from line 246 to line 264

- *For each joint, the function updates the D-H parameters based on each row of the D-H Table.*
- *Copies the* i*-th row of the D-H table (self.DH_tab) into DH_params. This contains the D-H parameters for the current joint.*
- *In line 257, computes the transformation matrix* $\mathrm{T}_{\mathrm{i}-1}^{\mathrm{i}}$ *from frame* $\mathrm{i}-1$ *to frame* i *using the updated D-H parameters.*
- *In line 261, multiplies the previous transformation matrix* $\mathrm{T}_{0}^{\mathrm{i}-1}$ *with* $\mathrm{T}_{\mathrm{i}-1}^{\mathrm{i}}$ *to calculate* $\mathrm{T}_{0}^{\mathrm{i}}$ *which represents the pose of joint iii with respect to the base frame.*
- *In line 264, Updates* $\mathrm{T}_{0}^{\mathrm{i}-1}$ *to* $\mathrm{T}_{0}^{\mathrm{i}}$ *for the next iteration of the loop.*

End-Effector Transformation: from line 267

The last term is the transformation matrix from the last joint to the end effector T_{n-1}^{n}.This is because the end effector will often have individual parameters (such as tool length or angle) that may not be in the joint loop.

Task D: Checking if a point is in the workspace of the robot arm

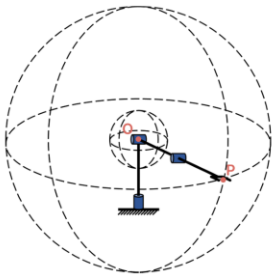


Figure 8: Workspace of the robot arm

The workspace of the robot arm refers to the region in which its end effector, denoted by point P, can reach. As shown in Figure 8, this workspace forms a spherical shell centred at point O. The radius of the outer sphere is given by $l_1 + l_2$, while the radius of the inner sphere is $l_1 - l_2$. The squared distance between point P and point O can be expressed as:

$$l_{PO}^2 = x_p^2 + y_p^2 + (z_p - l_0)^2$$

Given that l_{PO} should be smaller than the outer sphere radius and larger than the inner sphere radius, two equations can be derived:

$$x_p^2 + y_p^2 + (z_p - l_0)^2 \leq (l_1 + l_2)^2$$

$$x_p^2 + y_p^2 + (z_p - l_0)^2 \geq (l_1 - l_2)^2$$

```python
#Check if point is in WS. returns true or false
def checkInWS(self, P):
    xP, yP, zP = P
    l1, l2, l3 = self.links

    ############################################ TASK 4
    val = np.power(xP,2)+np.power(yP,2)+np.power(zP-l1,2) # Square length of PO
    r_max = l2+l3 # Radius of outer sphere
    r_min = l2-l3 # Radius of inner sphere

    inWS = True # Set the initial value of boolen inWS to be true

    if val > r_max**2. or val < r_min**2.: # Check if the end point P is outside the workspace
        inWS = False # Return False if outside

    return inWS # True if inside, false if outside
```

Figure 9: Code for Task D

Code Explanation:

The checkInWS function is used to determine whether a given point lies within the workspace of the robotic arm. The squared length of l_{PO} is represented as val, while the radius of the outer and inner spheres are denoted as r_max and r_min respectively. Based on the two equations derived above, if the squared length of l_{PO} exceeds the outer boundary (val > r_max**2) or the inner boundary (val < r_min**2), the function returns false, indicating that the point is outside the workspace. Otherwise, it returns true, confirming that the point is within the valid range.

Task E: Calculating inverse kinematics

This task aims to determine the joint angles corresponding to a given end-effector position. As shown in Figure 10, a single end-effector pose can have two possible configurations.

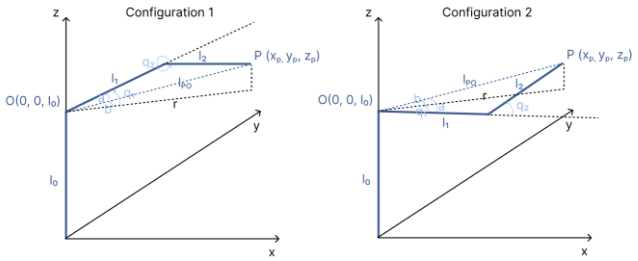


Figure 10: Two possible configurations for one end pose

Configuration 1:

The distance from P to O in the XY plane can be expressed as:

$$r_p = (x_p^2 + y_p^2)^{\frac{1}{2}}$$

Substitute this into the equation derived in task D, the squared length of l_{PO} is:

$$l_{PO}^2 = x_p^2 + y_p^2 + (z_p - l_0)^2$$

$$l_{PO}^2 = r_p^2 + (z_p - l_0)^2$$

According to the law of cosine, the squared length of l_{PO} can also be derived from l_1, l_2 and $\cos(q_2 - \pi)$.

$$l_{PO}^2 = l_1^2 + l_2^2 - 2l_1l_2\cos(q_2 - \pi) = l_1^2 + l_2^2 + 2l_1l_2\cos(q_2)$$

Therefore, q_2 can be calculated as follows:

$$l_1^2 + l_2^2 + 2l_1l_2\cos(q_2) = r_p^2 + (z_p - l_0)^2$$

$$\cos(q_2) = \frac{r_p^2 + (z_p - l_0)^2 - l_1^2 - l_2^2}{2l_1 l_2}$$

$$q_2 = -\arccos\left(\frac{r_p^2 + (z_p - l_0)^2 - l_1^2 - l_2^2}{2l_1 l_2}\right)$$

From Figure 10, q_1 can be separated into two parts: α and β. α can be simply derived as:

$$a = \arctan\left(\frac{z_p - l_0}{r_p}\right)$$

According to the law of cosine, the squared length of l_2 can be derived from l_{PO}, l_1 and $cos\beta$:

$$l_2^2 = l_{PO}^2 + l_1^2 - 2l_1 l_{PO}\cos(\beta)$$

$$l_2^2 = r_p^2 + (z_p - l_0)^2 + l_1^2 - 2l_1\left(r_p^2 + (z_p - l_0)^2\right)^{\frac{1}{2}}\cos(\beta)$$

$$\cos(\beta) = \frac{r_p^2 + (z_p - l_0)^2 + l_1^2 - l_2^2}{2l_1\left(r_p^2 + (z_p - l_0)^2\right)^{\frac{1}{2}}}$$

$$\beta = \arccos\left(\frac{r_p^2 + (z_p - l_0)^2 + l_1^2 - l_2^2}{2l_1\left(r_p^2 + (z_p - l_0)^2\right)^{\frac{1}{2}}}\right)$$

Therefore, q_1 is:

$$q_1 = \alpha + \beta = \arctan\left(\frac{z_p - l_0}{r_p}\right) + \arccos\left(\frac{r_p^2 + (z_p - l_0)^2 + l_1^2 - l_2^2}{2l_1\left(r_p^2 + (z_p - l_0)^2\right)^{\frac{1}{2}}}\right)$$

Based on the lab instruction, the expression for q_0 is:

$$q_0 = arctan2\left(\frac{y_p}{x_p}\right)$$

Configuration 2:

Similar to configuration 1, the joint angles can be calculated:

$$q_2 = -\arccos\left(\frac{r_p^2 + (z_p - l_0)^2 - l_1^2 - l_2^2}{2l_1 l_2}\right)$$

$$q_1 = \alpha - \beta = \arctan\left(\frac{z_p - l_0}{r_p}\right) - \arccos\left(\frac{r_p^2 + (z_p - l_0)^2 + l_1^2 - l_2^2}{2l_1\left(r_p^2 + (z_p - l_0)^2\right)^{\frac{1}{2}}}\right)$$

$$q_0 = arctan2\left(\frac{y_p}{x_p}\right)$$

Task F: Calculating inverse kinematics in Python

```
########################################### TASK 6
# Create array of length 3 and all elements are initialised to zero
# Two arrays are created because there are two possible configurations
q_a = np.zeros(3)
q_b = np.zeros(3)

# The joint angles are derived in task E
q_a[0] = np.arctan2(yP,xP)
q_b[0] = np.arctan2(yP,xP)

r = np.sqrt(np.power(xP,2)+np.power(yP,2)) # Distance between P and O in XY plane
z = zP-l1 # Distance between P and O along z axis

q_a[2] = -np.arccos((np.power(r, 2) + np.power(z, 2) - np.power(l2, 2) - np.power(l3, 2)) / 2 * l2 * l3)
q_b[2] = np.arccos((np.power(r, 2) + np.power(z, 2) - np.power(l2, 2) - np.power(l3, 2)) / 2 * l2 * l3)

q_a[1] = np.arctan2(z, r) + np.arccos((np.power(r, 2) + np.power(z, 2) + np.power(l2, 2) - np.power(l3, 2)) / (2 * l2 * np.sqrt(np.power(r, 2) + np.power(z, 2))))
q_b[1] = np.arctan2(z, r) - np.arccos((np.power(r, 2) + np.power(z, 2) + np.power(l2, 2) - np.power(l3, 2)) / (2 * l2 * np.sqrt(np.power(r, 2) + np.power(z, 2))))

# There are two possible configurations
q = [q_a, q_b]

Poses = [self.getFK(q_a), self.getFK(q_b)]
return q, Poses
```

Figure 11: Code for Task F

Code Explanation:

Two 1×3 arrays are generated because a single end-effector pose can correspond to two possible configurations, each defined by three joint angles. The expressions for the joint angles $\mathbf{q_0}$, $\mathbf{q_1}$ and $\mathbf{q_2}$ were derived in Task E. The final output, q, combines these two configurations into a 22×3 array, represented as [q_a, q_b].

Task G: Computing the Jacobian

The following forward kinematics equations show the position of the end effector:

$$x_p = (l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))\cos(q_0)$$
$$y_p = (l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))\sin(q_0)$$
$$z_p = l_0 + l_1 \sin(q_1) + l_2 \sin(q_1 + q_2)$$

The velocities of the end effector in task space can be calculated by differentiating the forward kinematics equations.

$$\dot{x_p} = -(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))\sin(q_0)\dot{q_0} - (\dot{q_0} l_1 sin(q1) + (\dot{q_1} + \dot{q_2}) l_2 sin(q_1 + q_2))cos(q_0)$$
$$\dot{y_p} = (l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))\cos(q_0)\dot{q_0} - (\dot{q_1} l_1 sin(q1) + (\dot{q_1} + \dot{q_2}) l_2 sin(q_1 + q_2))sin(q_0)$$
$$\dot{z_p} = (\dot{q_1} l_1 sin(q1) + (\dot{q_1} + \dot{q_2}) l_2 \cos(q_1 + q_2))$$

The Jacobian matrix serves as a bridge between the joint space (joint velocities) and the task space (end-effector velocities).

$$\dot{X} = J(q)\dot{q}$$

Where:

- $\dot{X}$ is the velocity vector of the end-effector in task space ($\dot{x_p}, \dot{y_p}, \dot{z_p}$).
- $\dot{q}$ is the velocity vector in joint space ($\dot{q_0}, \dot{q_1}, \dot{q_2}$).
- $J(q)$ is the Jacobian matrix, whose elements are partial derivatives of the forward kinematics equations:

$$\begin{bmatrix} \dot{x}_P \\ \dot{y}_P \\ \dot{z}_P \end{bmatrix} = \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} \\ J_{2,1} & J_{2,2} & J_{2,3} \\ J_{3,1} & J_{3,2} & J_{3,3} \end{bmatrix} \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}$$

For this robot, the Jacobian Matrix is as follows:

$$J(q) = \begin{bmatrix} -(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)) \sin(q_0) & (-l_1 \sin(q_1) - l_2 \sin(q_1 + q_2)) \cos(q_0) & -l_2 \sin(q_1 + q_2) \cos(q_0) \\ (l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)) \cos(q_0) & (-l_1 \sin(q_1) - l_2 \sin(q_1 + q_2)) \sin(q_0) & -l_2 \sin(q_1 + q_2) \sin(q_0) \\ 0 & l_1 \cos(q_1) + l_2 \cos(q_1 + q_2) & l_2 \cos(q_1 + q_2) \end{bmatrix}$$

```
353     # Computes Differential Kinematics
354     def getDK(self, q, q_dot):
355         q0, q1, q2 = q
356         l1, l2, l3 = self.links
357 
358         ############################################ TASK 7
359         self.Jacobian = np.array([[-(l1 * np.cos(q1) + l2 * np.cos(q1 + q2)) * np.sin(q0), (-l1 * np.sin(q1) - l2 * np.sin(q1 + q2)) *
    np.cos(q0), -l2 * np.sin(q1 + q2) * np.cos(q0)],
360                                   [(l1 * np.cos(q1) + l2 * np.cos(q1 + q2)) * np.cos(q0), (-l1 * np.sin(q1) - l2 * np.sin(q1 + q2)) *
    np.sin(q0), -l2 * np.sin(q1 + q2) * np.sin(q0)],
361                                   [0., l1 * np.cos(q1) + l2 * np.cos(q1 + q2), l2 * np.cos(q1 + q2)]])
362         x_dot = np.matmul(self.Jacobian, q_dot)
363         return x_dot
364 
```

Figure 12: Code for Task G

PID Tuning

The Proportional-integral-derivative (PID) controller is a widely used control mechanism that adjusts the output based on the error between the desired setpoint and the current state. It

consists of three main components: the proportional gain (Kp), the derivative gain (Kd), and the integral gain (Ki). It ensures precise and stable movement of the robotic arm, enabling the arm to accurately follow desired trajectories and handle dynamic changes in its environment.

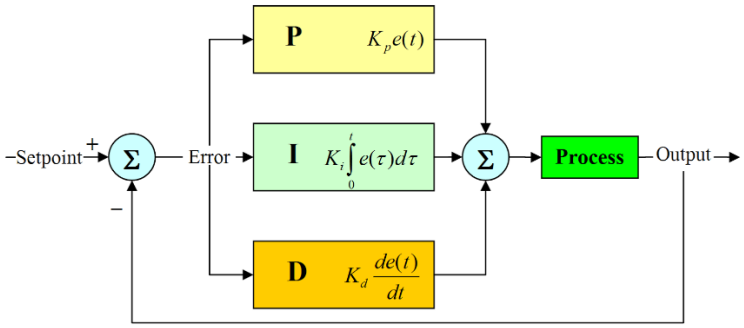


Figure 13: PID controller method formulation [1]

1. Proportional Gain (Kp):
    - **Theory:** The proportional term defines the strength of the controller's response to the current error. The error is the difference between the desired setpoint (position) and the current position.
    - **Effect:** Increasing Kp will **increase the response speed** of the system**,** and the plots to be generated more quickly. It will **reduce the rise time and initial-state error**. However, if Kp is too high, it can cause **overshoot and oscillations** which leads to instability. If the planned movement changes direction abruptly, this may cause large, sudden spikes in the plot, indicating instability or excessive responsiveness.
2. Derivative Gain (Kd):
    - **Theory:** The derivative term is related to the rate of change of the error which is calculated by dividing the difference between the current and previous errors by the sampling time. Increasing the derivative gain (Kd) by the error's derivative over time produces a damping effect.
    - **Effect:** Increasing Kd will **reduce overshoot and oscillations**, improving the stability of the system. This is particularly useful when the arm is approaching the desired position to ensure it settles smoothly. It also **reduces the settling time** and the overall speed of the system. However, too much Kd can slow down the system response.
3. Integral Gain (Ki):
    - **Theory:** The integral term of error accumulates over time, and Ki is useful to eliminate the steady-state error by adding a control effect proportional to the accumulated error, pushing the offset to zero.
    - **Effect:** Increasing Ki will help to eliminate steady-state error, **correcting any persistent offset** from the desired position. However, if Ki is too high, it can cause the system to become unstable and it might cause the arm to **oscillate or overshoot the target position**.

The visualisation of the changes corresponding to each Kp, Kd, and Ki is summarised in the following table:

Table 2: Effect of increasing PID gains on system response

Gain	Rise Time	Percent Overshoot	Settling Time	Steady State Error
Increase Kp	Decreases	Increases	Minimal Impact	Decreases
Increase Ki	Decreases	Increases	Increases	Great reduce
Increase Kd	Minimal Impact	Decreases	Decreases	Minimal Impact

The system response is described by the following characteristics:

Rise Time: measures how fast the system responds to change initially.

Percent Overshoot: The maximum peak the response reaches above the set point

Settling Time: The time required for the system to stabilize after an input change.

Steady State Error: The final difference between the desired set point and the actual system response after settling.

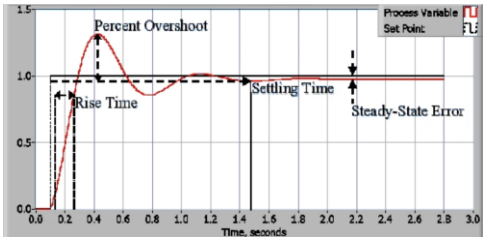


Figure 14: Graph demonstration of key characteristics of a damping system [2]

The goal of the PID tuning is to let the system respond to the changes and align with the planned trajectory smoothly and quickly without overshoot. The best practice is to ensure critical damping of the systems, that the rise time and settling time is short, with no steady-state error and minimal overshoot.

Task H: Tuning Controller Gains

No end-effector mass was assumed in task H, and the initial settings for Kp, Ki, and Kd were minimal (0.1 or 0). In this case, the robot was in stationary because with Kp, Ki, and Kd was very small, the control signal (output torque or force) is too weak to move the links.

```
<xacro:property name="ee_mass" value="0.01"/>
```

```
DESE3R:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 100

  # Position Controllers ---------------------------------------
  joint_0_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_0
    pid: {p: 0.1, i: 0, d: 0.0}
  joint_1_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_1
    pid: {p: 0.1, i: 0, d: 0.0}
  joint_2_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_2
    pid: {p: 0.1, i: 0, d: 0.0}
```

Figure 15: Code for PID tuning in file robot_model_gazebo.xacro (up) and controller_settings.yaml (down)

The joint position was visualised using the Quantitative Response Testing (RQT). To tune the PID controller for Joint 0, 1, and 2, the RQT plot of the set point (desired trajectory) and process value (actual response) was analysed to assess system performance.

joint_n_position_controller/state/process_value: The actual position of the joint n in real time

joint_n_position_controller/state/set_point: The desired position of the joint n (target)

Tuning joint 0

Initially, the proportional gain (Kp) was increased to 50, where noticeable oscillations was observed. This reflected that the system has become unstable, with the response overshooting and significant oscillation. To address these oscillations, the derivative gain (Kd) was gradually increased in intervals of 50. As Kd is increased, the system began to stabilize, and when Kd reaches 100, the system became critically damped.

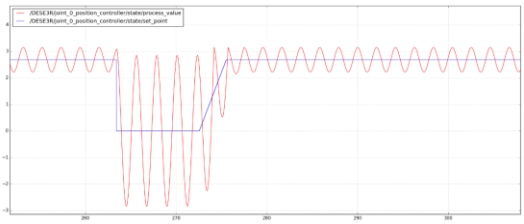


Figure 16: Joint 0 plots when Kp = 50, Kd, Ki = 0

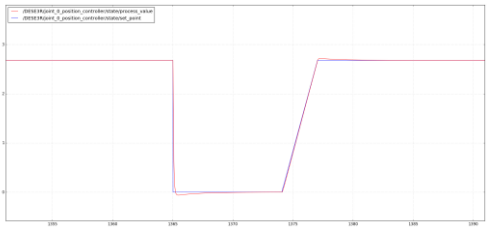


Figure 17: joint 0 plots when Kp = 50, Kd = 100, Ki = 0

The figure above indicates that overshooting is present at the initial state, and the settling time when the robot transitions can be reduced for better performance. To address the overshoot, the derivative gain (Kd) was increased, which helped dampen the oscillations and reduce the settling time. However, this also led to a drastic decrease in the system's response speed, as Kd tends to slow down the reaction to changes in the set point. To compensate for this reduction in speed and improve the response rate, the proportional gain (Kp) was subsequently increased.

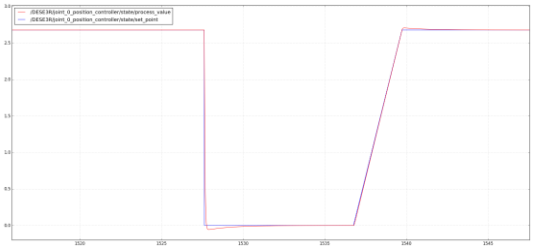


Figure 18: Joint 0 plots when Kp = 100, Kd =200, Ki = 0,

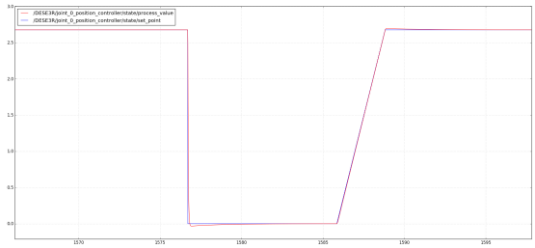


Figure 19: Joint 0 plots when Kp = 200 Kd=300 Ki = 0

The iterative tuning process continues as Kd was progressively increased to balance the overshoot and oscillations introduced by increasing Kp, while Kp was also adjusted to further enhance the speed of the operation. Through this iterative process, a balance was ultimately achieved at Kp = 200 and Kd = 400. As shown in fig, this tuning resulted in a well-balanced system where the trade-off between response speed and stability was optimized.

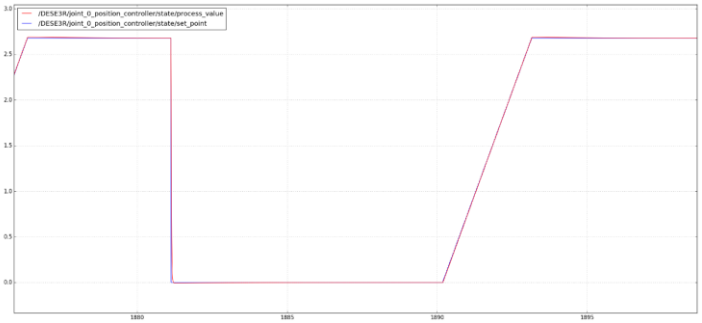


Figure 20: Joint 0 plots when Kp = 200, Kd = 400, Ki = 0

Tuning joint 1

When tuning Joint 1, it was observed that the initial offset is large, indicating a significant

difference between the set point and the process value. To mitigate this, proportional gain (Kp) was increased from 0 with 100 intervals. This aggressive tuning caused the initial offset to decrease, bringing the system closer to the desired set point at Kp = 500. However, the high Kp value also introduced large oscillations and system overshoot which made it difficult to visually determine whether the initial offset was still present.

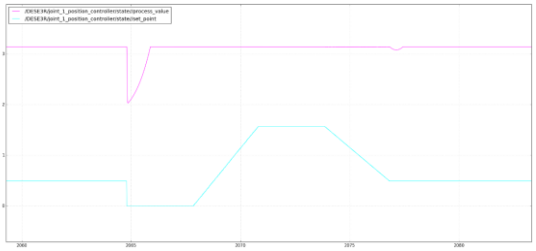


Figure 21: Joint 1 plots when Kp= 0.1, Kd = 0, Ki = 0

Figure 22: Joint 1 plots when Kp = 500, Kd = 0, Ki = 0

Kd values of 200, 400 and 800 were tested, and critical damping was achieved, but the initial offset remained present. To reduce the initial error, Kp was increased to 1000, and a further step was taken by increasing Kp to 2000, which successfully diminished the initial offset. However, this increase in Kp also introduced overshoots around the turning point of the desired path. This overshoot occurs because the higher Kp value caused the system to respond aggressively.

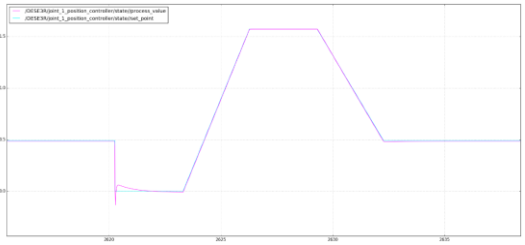


Figure 23: Joint 1 plots when Kp = 1000, Kd = 800, Ki = 0

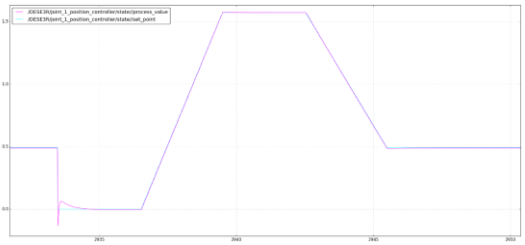


Figure 24: Joint 1 plots when Kp =2000, Kd = 800, Ki = 0

Figure 25: Joint 1 plots when Kp = 1600, Kd = 800, Ki = 400

To mitigate the overshoot, the integral gain (Ki) was introduced and increased to 400. The introduction of Ki helped addressing the steady-state error and further reduce the offset. To compensate for this offset minimization and prevent the system from becoming overly aggressive, the proportional gain (Kp) was lowered to 1600. This adjustment helped to strike a balance between maintaining a minimal offset and reducing the system's overshoot. The combination of increasing Ki and lowering Kp worked to stabilize the system by addressing both steady-state error and dynamic oscillations. This enhanced the performance with reduced overshoot and a more stable response.

Tuning joint 2

Joint 2 also faced a significant initial offset, so the derivative gain (Kd) is increased from 100 to 1000 to help mitigate this issue. An iterative approach similar to Joint 1 and Joint 0 was then taken to balance the proportional gain (Kp) and derivative gain (Kd) to find the best trade-off between response time, damping of the system, and minimizing oscillations and overshoot. Kd was gradually increased from 0 to 500 while adjusting Kp. The system achieved stable performance with a fast response when Kp = 1000 and Kd = 500. This combination effectively reduced both the initial offset and oscillations, maintaining stability and minimizing overshoot.

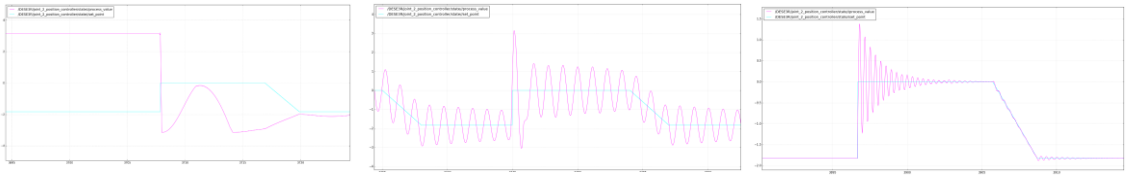


Figure 26, 27, 28: Joint 2 plots when Kp = 0.1, 100, 1000 (left to right), Kd = 0, Ki = 0

Ki (the integral gain) was increased to assess its effect on reducing the steady-state error, aiming to improve the system's accuracy over time. However, when Ki was increased to 100, new oscillations and disruptions were introduced into the system. This behaviour suggested that the integral accumulated error over time causes the system to become overly sensitive to past errors. As a result, undesirable instability and oscillatory behaviour was produced within the system.

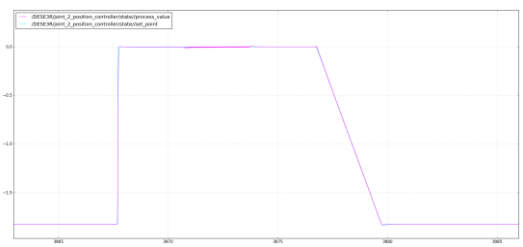


Figure 29: Joint 2 plots when Kp = 1000, Kd = 500, Ki = 0

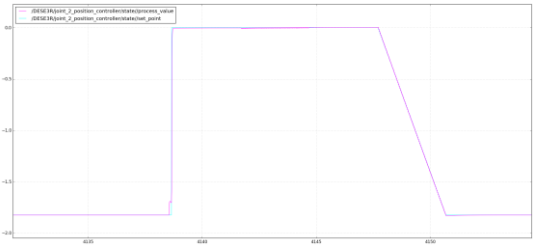


Figure 30: Joint 2 plots when Kp = 1000, Kd = 500, Ki = 100

The final values of Task H PID tuning is presented below:

Table 3: Final results of PID tuning

Joint	Kp	Ki	Kd
0	200	0	400
1	1600	400	800
2	1000	0	500

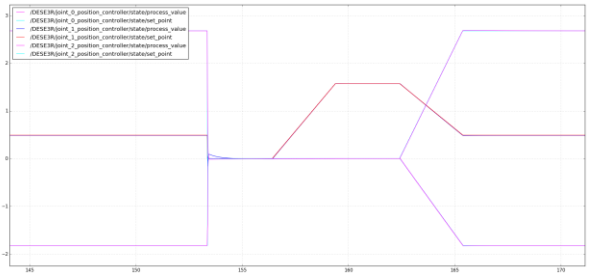


Figure 31: Joint 0, 1, 2 paths combined after PID tuning

The overall behaviour of three joints is demonstrated by Figure 31. This shows that the joints are able to follow their respective set points with no oscillation and a stable response. Although some slight overshooting is still present at the initial state, the system now operates with a balanced speed, quickly respond to the desired path, achieving a smooth and steady state at the set points.

Task I & J

To lift a 30kg object, the mass of the end effector was set to 30kg by modifying the ee_mass parameter in the robot_model_gazebo.xacro file.

```
16 <xacro:property name="ee_mass" value="30"/>
```

Figure 32: Code for Changing End Effector's Mass to 30

After applying a 30 kg mass, the end effector exhibited the following behaviours:

- It remained on the ground as the joint gains were insufficient to lift the weight.
- It displayed unstable and inconsistent movements due to the increased force acting on the end effector.

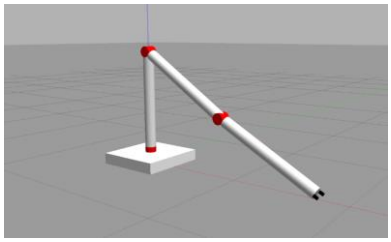


Figure 33: End Effector stays on the ground

A similar method was applied to tune the PID gains with the modified mass but turning Joint 2 first. Joint 1 without any gain on Joint 2 led to the unstable movement of the link, for example, overlapping of the two joints and the end effector staying on the ground.

Firstly, the Kp values for all three joints were adjusted simultaneously to establish a baseline. Given that each joint's behaviour influences the entire system, adjusting them together ensures system-wide stability before fine-tuning. This approach eliminates the need for redundant iterations, ensuring the process value (PV) closely follows the set point (SP).

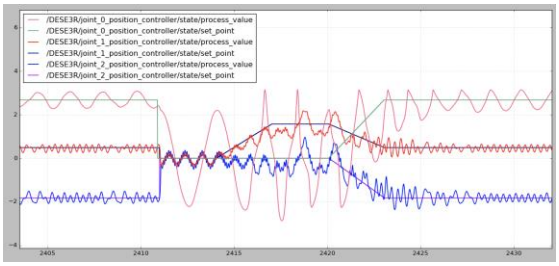


Figure 34

Table 4: Baseline Kp values for each joint

	Joint 0	Joint 1	Joint 2
Kp	500	10000	3100

Fine Tuning Joint 2

With the baseline Kp values founded, Joint 2 was selected for fine-tuning first, as it directly supports the end-effector and experiences the greatest torque variation due to the added 30 kg mass. By stabilizing Joint 2 first, potential disturbances to Joint 1 and Joint 0 were mitigated, making the following tuning steps more controlled.

Table 5: Stages of tuning joint 2

Stage	Kp	Ki	Kd
1	**3100**	0	0
2	**3100**	0	500
3	**3100**	0	1000

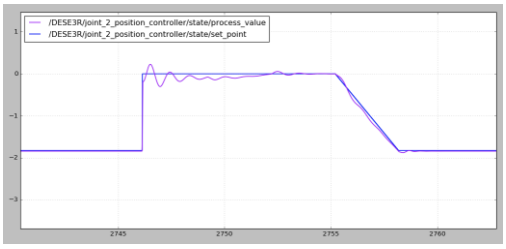


Figure 35: Joint 2 during Stage 2

In *Figure 35*, the steady-state error is already minimized, so increasing Kp and Ki is unnecessary. However, the oscillations during transient phases, indicates that further damping is needed to improve stability. To addressing the remaining oscillations, Kd was increased from 500 to 1000, providing stronger damping to smooth out the response.

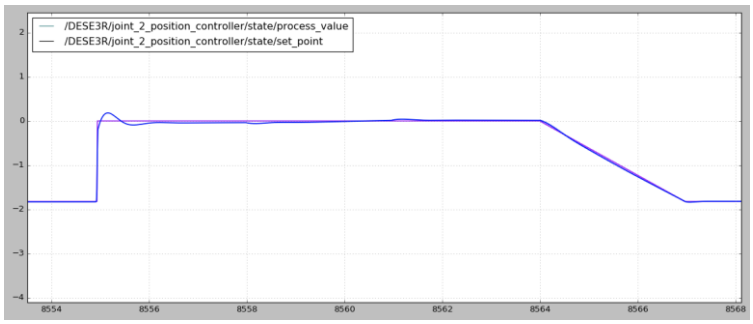


Figure 36: Final Graph representation for joint 2

Fine Tuning Joint 1

Tuning Joint 1 before Joint 0 ensures that the primary lifting and stabilization mechanisms are fully optimized before refining rotational movement. By doing this the impact of dynamic coupling was minimised.

Table 6: Stages of tuning joint 1

Stage	Kp	Ki	Kd
1	10000	0	0
2	10000	1000	0
3	10000	900	0
4	10000	900	500
5	10000	900	1000

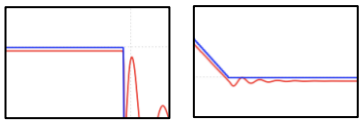


Figure 37: Steady-State Error during Stage 1

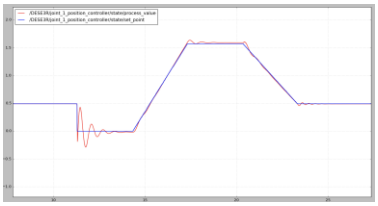


Figure 38: Joint 1 during Stage 3

Figure 37 shows a noticeable steady-state error where the PV did not completely converge to the SP. This indicates that while the Kp sufficient initial response, it was not enough to fully eliminate residual errors over time. To address this, Ki was introduced in Stage 2 and 3 to reduce steady-state error. The integral term accumulates past errors and applies a corrective action, helping the PV to reach and maintain the SP more accurately.

By utilizing the same method, Kd was increased to 1000 to eliminate the oscillations in *Figure 38*.

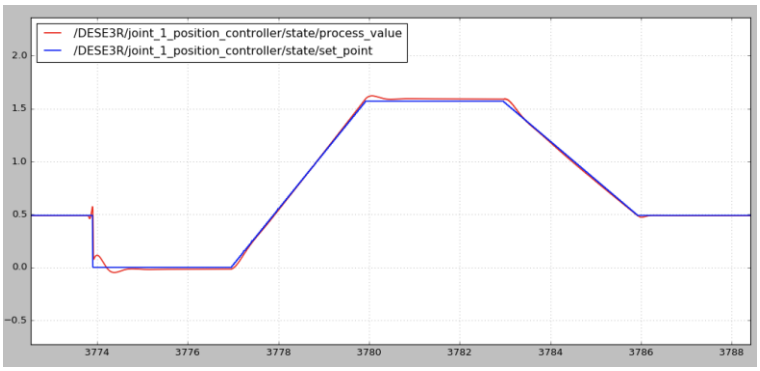


Figure 39: Final Graph representation for joint 1

Fine Tuning Joint 0

With Joint 1 and Joint 2 properly tuned, the final step of the tuning process focused on Joint 0,

as it primarily controls the rotational movement of the robotic arm. As a result, Joint 0 adjustments can be made without affecting the other joints.

Table 7: Stages of tuning joint 0

	Kp	Ki	Kd
1	500	0	0
2	500	0	500
3	500	0	1000

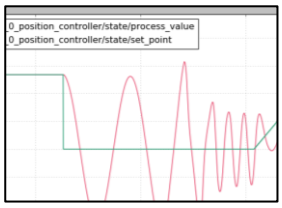


Figure 40: No Steady-State Error during Stage 1

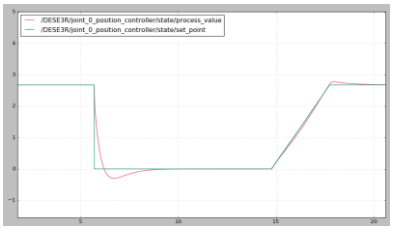


Figure 41: Joint 0 during Stage 2

There were significant oscillations observed from *Figure 40.* Comparing to the plot in *Figure 41,* the oscillations were visibly reduced by adding Kd introduced damping. Doubling Kd to 1000 further dampens the oscillations, improving trajectory tracking. The PV closely follows the SP with minimal overshoot, achieving a well-damped and stable response.

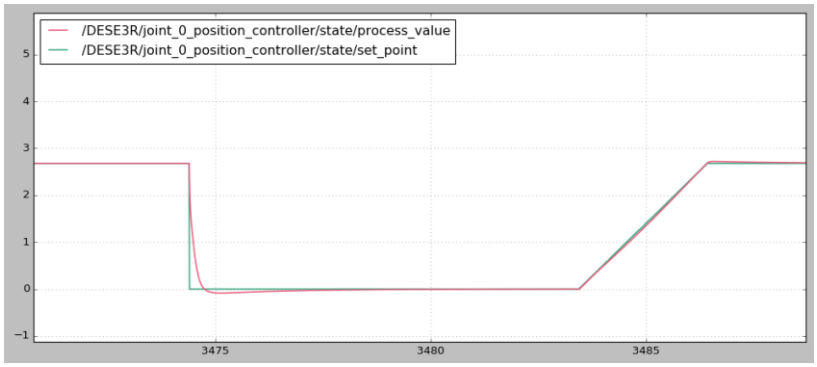


Figure 42: Final Graph representation for joint 0

Final Values

Table 8: Summary of PID values for loaded robot

	Kp	Ki	Kd
Joint 1	500	0	1000
Joint 1	10000	900	1000
Joint 2	3100	0	1000

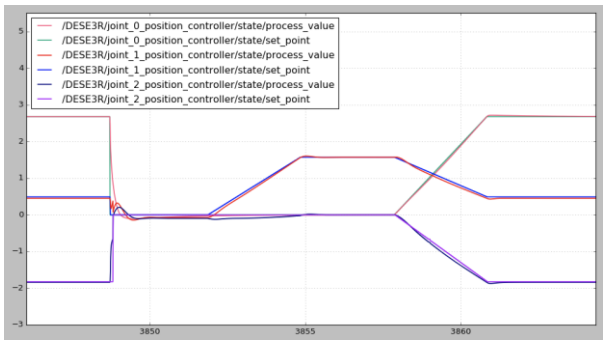


Figure 43: Final Graph representation for joint 0,1,2

For Kd, value of 1000 was used instead of higher value as it was the maximum value allowed in the dynamic reconfigure command.

Bibliography

Task H:

[1] I. Toledano, “PID Controller Explained: Types, Uses & Operations,” *Wattco*, May 21, 2024. https://www.wattco.com/2024/05/pid-controller-explained/

[2]“The PID Controller & Theory Explained,” *Ni.com*, 2024. https://www.ni.com/en/shop/labview/pid-theory-explained.html?srsltid=AfmBOopCLe9molj-talQLhF-_sgoXYWgA-0COvZF2NjC4tVUPc_7ipdp (accessed Jan. 31, 2025).