Design and Implementation of a Force Controller for a Haptic Device

PANTELIS MPONITSIS

Supervisor Professor: Kostas Vlachos

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science in *Computer Science and Engineering*

April 21, 2025

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNIVERSITY OF IOANNINA

Contents

Chapter 1

Introduction

Chapter 2

Kinematic Analysis

In robotic systems, it is essential to determine both the position and velocity of the end-effector in relation to the motor angles and their respective velocities. This knowledge allows for accurate control of the robot's motion, ensuring that the end-effector reaches its desired position and follows the intended trajectory. Furthermore, calculating the Jacobian matrix is crucial for determining the required joint torques (τ) in static conditions. These torques are necessary for the motors to apply the forces needed to achieve the specified motions and maintain the stability of the end-effector under various loads, within the framework of implementing Force Control. This chapter provides an in-depth analysis of kinematic concepts, including forward and inverse kinematics, differential kinematics, and methods for validating both numerical and symbolic solutions.

2.1 Forward Kinematics

Forward kinematics provides the coordinates of the end-effector given that all characteristic link lengths and the angles of the driven links are known.

The process for locating the end-effector's coordinates is relatively simple. First, the coordinates of points A and B are determined through direct trigonometric calculations. Next, the circle equation is used to identify the intersection points of two circles centered at A and B, with respective radii of l_2 and l_3 (as shown in Fig. 2.1).

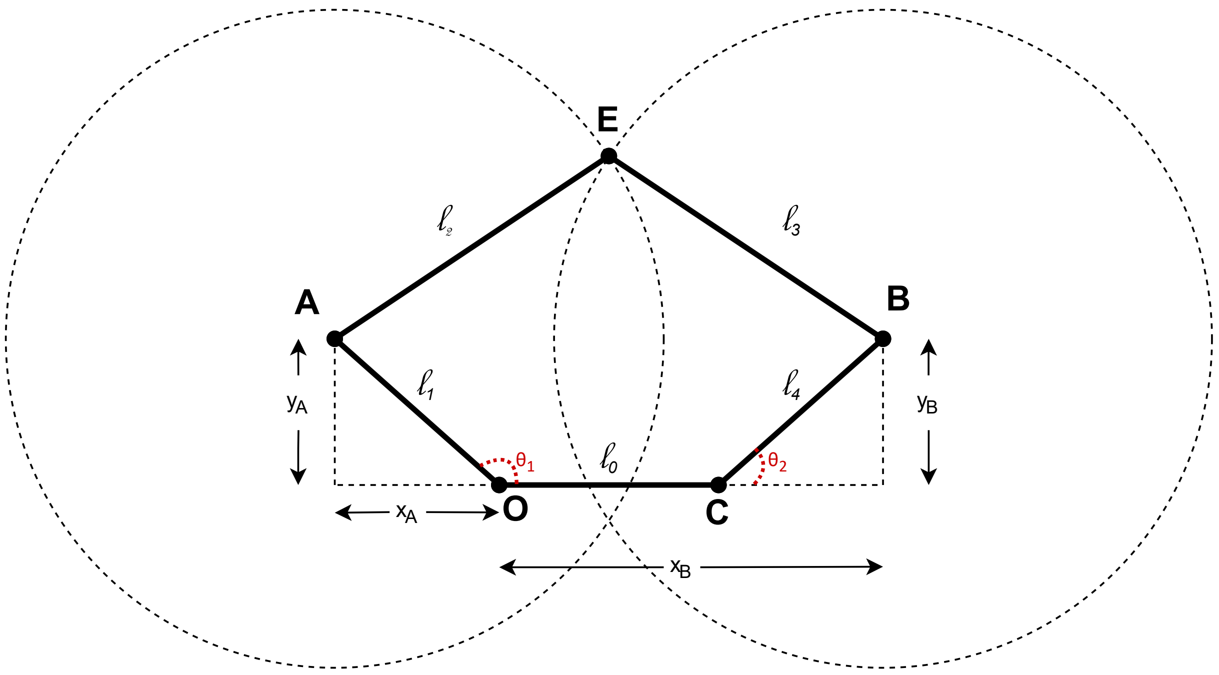


Figure 2.1: The positive solution of the intersection of the two circles provides the end-effector point.

Point O is considered the origin of the axes, and point E is the end-effector point. Using simple trigonometry, the coordinates of point A are derived as follows:

$$x_A = cos(\theta_1) * l_1 \quad \text{and} \quad y_A = sin(\theta_1) * l_1$$

In a similar way, the coordinates of point B are derived as follows:

$$x_B = l_0 + cos(\theta_2) * l_4 \quad \text{and} \quad y_B = sin(\theta_2) * l_4$$

Next, the above equations are used in the equations of the two circles.

For the circle centered at point A:

$$(x - x_A)^2 + (y - y_A)^2 = l_2^2 \Leftrightarrow$$

$$(x - cos(\theta_1) * l_1)^2 + (y - sin(\theta_1) * l_1)^2 = l_2^2$$

For the circle centered at point B:

$$(x - x_B)^2 + (y - y_B)^2 = l_2^2 \Leftrightarrow$$

$$(x - l_0 - cos(\theta_2) * l_4)^2 + (y - sin(\theta_2) * l_4)^2 = l_3^2$$

Applying the coordinates of the end-effector point to the equation of the circle centered at point A results in:

$$\begin{gathered}(x_\epsilon - x_A)^2 + (y_\epsilon - y_A)^2 = l_2^2 \Leftrightarrow \\ x_e^2 - 2 * x_\epsilon * x_A + x_A^2 + y_\epsilon^2 - 2 * y_\epsilon * y_A + y_A^2 - l_2^2 = 0\end{gathered} \tag{2.1}$$

In a similar way, applying the coordinates of the end-effector point to the equation of the circle centered at point B results in:

$$\begin{gathered}(x_\epsilon - x_B)^2 + (y_\epsilon - y_B)^2 = l_2^2 \Leftrightarrow \\ x_e^2 - 2 * x_\epsilon * x_B + x_B^2 + y_\epsilon^2 - 2 * y_\epsilon * y_B + y_B^2 - l_3^2 = 0\end{gathered} \tag{2.2}$$

Since the end-effector is the intersection point of the two circles, Eq. (2.1) and Eq. (2.2) are equal.

$$(2.1) = (2.2) \Leftrightarrow$$

$$x_\epsilon^2 - 2x_\epsilon x_A + x_A^2 + y_\epsilon^2 - 2y_\epsilon y_A + y_A^2 - l_2^2 = x_\epsilon^2 - 2x_\epsilon x_B + x_B^2 + y_e^2 - 2y_\epsilon y_B + y_B^2 - l_3^2 \Leftrightarrow$$

$$x_\epsilon = \frac{y_A - y_B}{x_B - x_A} * y_\epsilon + \frac{y_B^2 + x_B^2 - l_1^2 + l_2^2 - l_3^2}{2 * (x_B - x_A)} \Leftrightarrow$$

$$\boxed{x_\epsilon = a * y_\epsilon + b} \tag{2.3}$$

Subsequently, substituting Eq. (2.3) into Eq. (2.1) results in:

$$(a * y_\epsilon + b)^2 - 2 * x_A * (a * y_\epsilon + b) + x_A^2 + y_\epsilon^2 - 2 * y_\epsilon * y_A + y_A^2 - l_2^2 = 0 \Leftrightarrow$$

$$(a^2 + 1) * y_\epsilon^2 + (2 * a * b - 2 * a * x_A - 2 * y_A) * y_\epsilon + b^2 - 2 * b * x_A + l_1^2 - l_2^2 = 0 \Leftrightarrow$$

$$c * y_\epsilon^2 + d * y_\epsilon + e \Leftrightarrow$$

$$\boxed{y_\epsilon = \frac{-d \pm \sqrt{d^2 - 4 * c * e}}{2 * c}} \tag{2.4}$$

From the above expression for y_ϵ, the positive solution is retained.

Eq. (2.3) and Eq. (2.4) express the coordinates of end-effector point as a function of the motor angles θ_1 and θ_2.

2.2 Inverse Kinematics

Inverse kinematics determine the angles of the driven links when the characteristic link lengths and the coordinates of the end-effector are known.

The solution of Inverse kinematics can be obtained straightforwardly using the distance formula and the law of cosines.

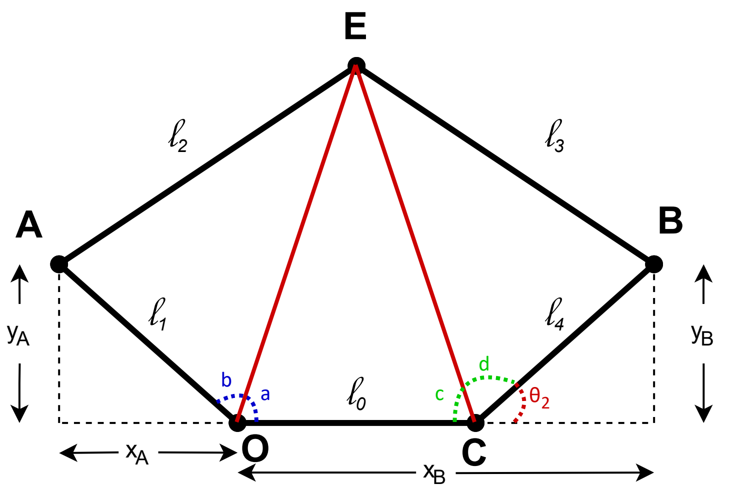


Figure 2.2: In the given diagram, two edges have been added that connect the end-effector point with points O and C.

Using the distance formula, the two new edges will be calculated.

$$OE = \sqrt{(x_\epsilon - x_O)^2 + (y_\epsilon - y_O)^2} \Leftrightarrow$$

$$OE^2 = x_\epsilon^2 + y_\epsilon^2$$

Similarly, for the other edge.

$$CE = \sqrt{(x_\epsilon - x_C)^2 + (y_\epsilon - y_C)^2} \Leftrightarrow$$

$$CE^2 = (x_\epsilon - l_0)^2 + y_\epsilon^2 \Leftrightarrow$$

$$CE^2 = x_\epsilon^2 - 2 * x_\epsilon * l_0 + l_0^2 + y_\epsilon^2$$

At this point, the law of cosines will be applied to triangle OEC in order to find angle a.

$$CE^2 = l_0^2 + OE^2 - 2 * l_0 * OE * cos(a) \Leftrightarrow$$

$$cos(a) = \frac{OE^2 - CE^2 + l_0^2}{2 * l_0 * OE} \Leftrightarrow$$

$$a = \arccos(\frac{x_\epsilon}{\sqrt{x_\epsilon^2 + y_\epsilon^2}})$$

Next, the law of cosines will be applied to triangle OEA in order to find angle b.

$$l_2^2 = l_1^2 + OE^2 - 2 * l_1 * OE * cos(b) \Leftrightarrow$$

$$cos(b) = \frac{l_1^2 - l_2^2 + x_\epsilon^2 + y_\epsilon^2}{2 * l_1 * \sqrt{x_\epsilon^2 + y_\epsilon^2}} \Leftrightarrow$$

$$b = \arccos(\frac{l_1^2 - l_2^2 + x_\epsilon^2 + y_\epsilon^2}{2 * l_1 * \sqrt{x_\epsilon^2 + y_\epsilon^2}})$$

Similarly to the first calculation, the law of cosines will be applied to triangle OEC again, but this time in order to find angle c.

$$OE^2 = CE^2 + l_0^2 - 2 * CE * l_0 * cos(c) \Leftrightarrow$$

$$cos(c) = \frac{2 * l_0^2 - 2 * x_\epsilon}{2 * \sqrt{(x_\epsilon - l_0)^2 + y_\epsilon^2} * l_0} \Leftrightarrow$$

$$c = \arccos(\frac{l_0 - x_\epsilon}{\sqrt{(x_\epsilon - l_0)^2 + y_\epsilon^2}})$$

Finally, the law of cosines will be applied to triangle CEB in order to find angle d.

$$l_3^2 = CE^2 + l_4^2 - 2 * CE * l_4 * cos(d) \Leftrightarrow$$

$$cos(d) = \frac{x_\epsilon^2 + y_\epsilon^2 + l_0^2 + l_4^2 - l_3^2 - 2 * x_\epsilon * l_0}{2 * \sqrt{(x_\epsilon - l_0)^2 + y_\epsilon^2} * l_4} \Leftrightarrow$$

$$d = \arccos(\frac{x_\epsilon^2 + y_\epsilon^2 + l_0^2 + l_4^2 - l_3^2 - 2 * x_\epsilon * l_0}{2 * \sqrt{(x_\epsilon - l_0)^2 + y_\epsilon^2} * l_4})$$

Having calculated the angles a, b, c, and d, the motor angles θ_1 and θ_2 are derived as follows:

$$\boxed{\theta_1 = a + b}$$

$$\boxed{\theta_2 = \pi \text{ - } (c + d)}$$

2.3 Validation of Inverse Kinematics

There is a very simple way to check if the Inverse Kinematics equations are correct.

Assuming that the motor angles θ_1 and θ_2 are 90°, Eq. (2.3) and Eq. (2.4) of Foward Kinematics are solved, and it follows that the coordinates of the end-effector point are:

$$x_\epsilon = 0.04\,\text{m} \quad \text{and} \quad y_\epsilon = 0.1931\,\text{m}$$

Now, the inverse process needs to be followed. The coordinates of the end-effector point are substituted into the Inverse Kinematics equations, and it follows that the motor angles θ_1 and θ_2 are:

$$\theta_1 = 1.572\,10\,\text{rad} = 90.07465°$$

$$\theta_2 = 1.569\,49\,\text{rad} = 89.92513°$$

It is observed that the motor angles θ_1 and θ_2 are approximately 90°, indicating that the Inverse Kinematics equations are correct. The small deviation from 90° is acceptable.

2.4 Representation of Passive Angles as a function of Motor Angles

The robotic system of the haptic device being analyzed contains two passive joints whose angles cannot be controlled by the two motors or measured by any sensor. These angles are θ_3 and θ_4. Therefore, they are unknown, and in order to be used in any calculation, they need to be expressed as functions of the known variables of the system, which include the links (l_0 - l_4) and the motor angles θ_1 and θ_1.

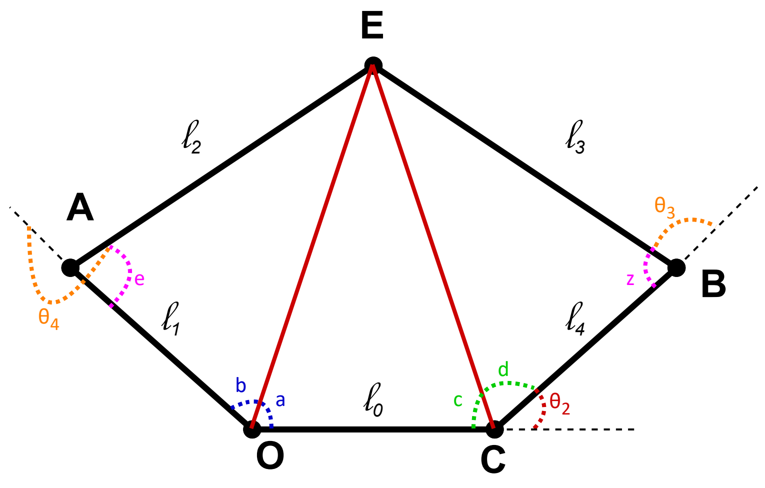


Figure 2.3: Points A and B represent the passive joints.

For the calculation of angles θ_3 and θ_4, it is easier to first calculate angles e and z.

The law of cosines will be applied to triangle OEA in order to find angle e.

$$OE^2 = l_1^2 + l_2^2 - 2 * l_1 * l_2 * cos(e) \Leftrightarrow$$

$$cos(e) = \frac{-x_\epsilon^2 - y_\epsilon^2 + l_1^2 + l_2^2}{2 * l_1 * l_2} \Leftrightarrow$$

$$e = \arccos(\frac{-x_\epsilon^2 - y_\epsilon^2 + l_1^2 + l_2^2}{2 * l_1 * l_2})$$

Next, the law of cosines will be applied to triangle CEB in order to find angle z.

$$CE^2 = l_3^2 + l_4^2 - 2 * l_3 * l_4 * cos(z) \Leftrightarrow$$

$$cos(z) = \frac{-x_\epsilon^2 - y_\epsilon^2 + 2 * x_\epsilon * l_0 - l_0^2 + l_3^2 + l_4^2}{2 * l_3 * l_4} \Leftrightarrow$$

$$z = \arccos(\frac{-x_\epsilon^2 - y_\epsilon^2 + 2 * x_\epsilon * l_0 - l_0^2 + l_3^2 + l_4^2}{2 * l_3 * l_4})$$

Having calculated the angles e and z, the passive angles θ_3 and θ_4 are derived as follows:

$$\boxed{\theta_3 = \pi \text{ - } z}$$

$$\boxed{\theta_4 = \pi + e}$$

2.5 Symbolic Solution of Differential Kinematics (Jacobian Matrix)

Differential kinematics provides the velocity of the end-effector point given that the velocities of the motor angles are known. Specifically, it is necessary to find the Jacobian matrix, which relates the velocity of the end-effector point to the velocities of the motor angles.

The straightforward solution for calculating the Jacobian matrix is to differentiate Eq. (2.3) and Eq. (2.4) with respect to θ_1 and θ_2 (numerical solution). However, since these equations are very complex and cannot be differentiated by hand, the Jacobian matrix must be computed using another method.

Thus, one approach is to decompose the robotic system into two new separate systems, whose end-effector points will have the same velocity. For each system, the position equations of the end-effector are easily found through homogeneous transformations, and then they are differentiated to obtain the velocities, which will be equal.

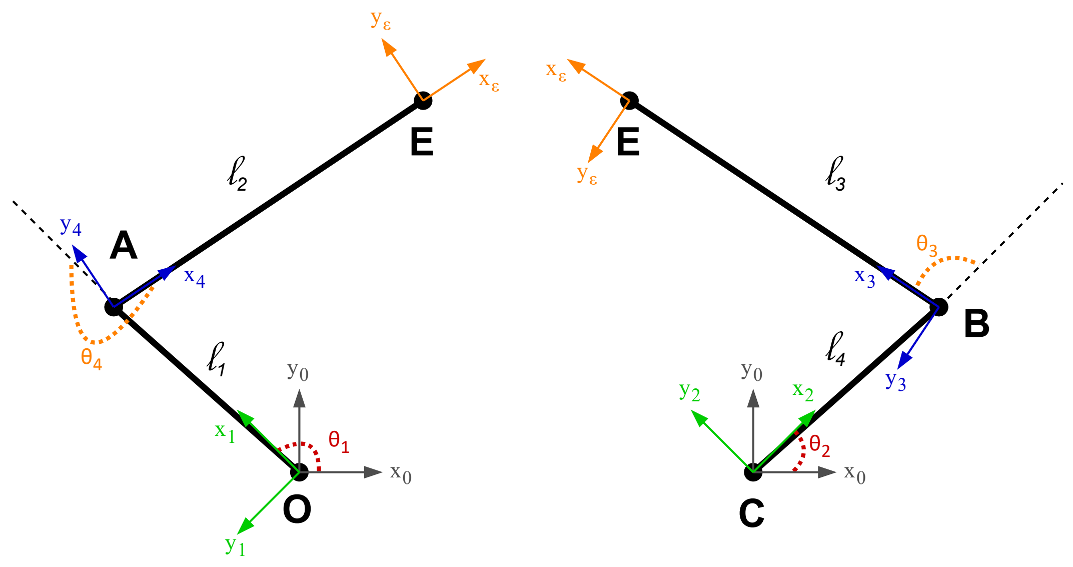


Figure 2.4: Two different robotic manipulators.

Through the combination of successive transformations T from coordinate frame 0 to coordinate frame E, the position of the end-effector point is calculated. Initially, this is performed for the left system.

$$
{}^{0}\mathbf{T}_1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^{1}\mathbf{T}_4 = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & l_1 \\ \sin(\theta_4) & \cos(\theta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^{4}\mathbf{T}_\epsilon = \begin{bmatrix} 1 & 0 & 0 & l_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{0}\mathbf{T}_4 = {}^{0}\mathbf{T}_1{}^{1}\mathbf{T}_4 = \begin{bmatrix} \cos(\theta_1)\cos(\theta_4) - \sin(\theta_1)\sin(\theta_4) & -\cos(\theta_1)\sin(\theta_4) - \sin(\theta_1)\cos(\theta_4) & 0 & \cos(\theta_1)l_1 \\ \sin(\theta_1)\cos(\theta_4) + \cos(\theta_1)\sin(\theta_4) & -\sin(\theta_1)\sin(\theta_4) + \cos(\theta_1)\cos(\theta_4) & 0 & \sin(\theta_1)l_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$^0\mathbf{T}_\epsilon = ^0\mathbf{T}_4^4\mathbf{T}_\epsilon = \begin{bmatrix} \cos(\theta_1+\theta_4) & -\sin(\theta_1+\theta_4) & 0 & (\cos(\theta_1)\cos(\theta_4)-\sin(\theta_1)\sin(\theta_4))l_2+\cos(\theta_1)l_1 \\ \sin(\theta_1+\theta_4) & \cos(\theta_1+\theta_4) & 0 & (\sin(\theta_1)\cos(\theta_4)+\cos(\theta_1)\sin(\theta_4))l_2+\sin(\theta_1)l_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where :

$$\cos(\theta_1+\theta_4) = \cos(\theta_1)\cos(\theta_4) - \sin(\theta_1)\sin(\theta_4)$$

$$\sin(\theta_1+\theta_4) = \sin(\theta_1)\cos(\theta_4) + \cos(\theta_1)\sin(\theta_4)$$

Differentiating the position equations of the end-effector point results in:

$$\dot{x_\epsilon} = -l_2\sin(\theta_1)\cos(\theta_4)\dot{\theta_1} - l_2\cos(\theta_1)\sin(\theta_4)\dot{\theta_4} - l_2\cos(\theta_1)\sin(\theta_4)\dot{\theta_1} - l_2\sin(\theta_1)\cos(\theta_4)\dot{\theta_4} - l_1\sin(\theta_1)\dot{\theta_1}$$

$$\dot{y_\epsilon} = l_2\cos(\theta_1)\sin(\theta_4)\dot{\theta_1} - l_2\sin(\theta_1)\sin(\theta_4)\dot{\theta_4} - l_2\sin(\theta_1)\sin(\theta_4)\dot{\theta_1} + l_2\cos(\theta_1)\cos(\theta_4)\dot{\theta_4} + l_1\cos(\theta_1)\dot{\theta_1}$$

Therefore, the Jacobian matrix is:

$$^0\mathbf{J}_u = \begin{bmatrix} -l_1\sin(\theta_1) - l_2(\sin(\theta_1)\cos(\theta_4)+\cos(\theta_1)\sin(\theta_4)) & -l_2(\sin(\theta_1)\cos(\theta_4)+\cos(\theta_1)\sin(\theta_4)) \\ l_1\cos(\theta_1) + l_2(\cos(\theta_1)\cos(\theta_4)-\sin(\theta_1)\sin(\theta_4)) & l_2(\cos(\theta_1)\cos(\theta_4)-\sin(\theta_1)\sin(\theta_4)) \end{bmatrix}$$

For simplicity, the above matrix will be written as:

$$^0\mathbf{J}_u = \begin{bmatrix} -\alpha_{00} & -\alpha_{01} \\ \alpha_{10} & \alpha_{11} \end{bmatrix}$$

Thus, the velocities of the end-effector point can be written as follows:

$$\dot{x_\epsilon} = -\alpha_{00}\dot{\theta_1} - \alpha_{01}\dot{\theta_4} \qquad \dot{y_\epsilon} = \alpha_{10}\dot{\theta_1} + \alpha_{11}\dot{\theta_4} \tag{2.5}$$

Subsequently, the transformations for the right system are calculated.

$$
{}^0\mathbf{T}_2 = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 & 0 \\ \sin(\theta_2) & \cos(\theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^2\mathbf{T}_3 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & l_4 \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^3\mathbf{T}_\epsilon = \begin{bmatrix} 1 & 0 & 0 & l_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^0\mathbf{T}_3 = {}^0\mathbf{T}_2^2\mathbf{T}_3 = \begin{bmatrix} \cos(\theta_2)\cos(\theta_3) - \sin(\theta_2)\sin(\theta_3) & -\cos(\theta_2)\sin(\theta_3) - \sin(\theta_2)\cos(\theta_3) & 0 & \cos(\theta_2)l_4 \\ \sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3) & -\sin(\theta_2)\sin(\theta_3) + \cos(\theta_2)\cos(\theta_3) & 0 & \sin(\theta_2)l_4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^0\mathbf{T}_\epsilon = {}^0\mathbf{T}_3^3\mathbf{T}_\epsilon = \begin{bmatrix} \cos(\theta_2+\theta_3) & -\sin(\theta_2+\theta_3) & 0 & (\cos(\theta_2)\cos(\theta_3) - \sin(\theta_2)\sin(\theta_3))l_3 + \cos(\theta_2)l_4 \\ \sin(\theta_2+\theta_3) & \cos(\theta_2+\theta_3) & 0 & (\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3))l_3 + \sin(\theta_2)l_4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Where :

$$
\cos(\theta_2+\theta_3) = \cos(\theta_2)\cos(\theta_3) - \sin(\theta_2)\sin(\theta_3)
$$

$$
\sin(\theta_2+\theta_3) = \sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)
$$

Differentiating the position equations of the end-effector point results in:

$$
\dot{x_\epsilon} = -l_3\sin(\theta_2)\cos(\theta_3)\dot{\theta_2} - l_3\cos(\theta_2)\sin(\theta_3)\dot{\theta_3} - l_3\cos(\theta_2)\sin(\theta_3)\dot{\theta_2} - l_3\sin(\theta_2)\cos(\theta_3)\dot{\theta_3} - l_4\sin(\theta_2)\dot{\theta_2}
$$

$$
\dot{y_\epsilon} = l_3\cos(\theta_2)\sin(\theta_3)\dot{\theta_2} - l_3\sin(\theta_2)\sin(\theta_3)\dot{\theta_3} - l_3\sin(\theta_2)\sin(\theta_3)\dot{\theta_2} + l_3\cos(\theta_2)\cos(\theta_3)\dot{\theta_3} + l_4\cos(\theta_2)\dot{\theta_2}
$$

Therefore, the Jacobian matrix is:

$$^{0}\mathbf{J}_u = \begin{bmatrix} -l_4\sin(\theta_2) - l_3(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)) & -l_3(\sin(\theta_2)\cos(\theta_3) + \cos(\theta_2)\sin(\theta_3)) \\ l_4\cos(\theta_2) + l_3(\cos(\theta_2)\cos(\theta_3) - \sin(\theta_2)\sin(\theta_3)) & l_3(\cos(\theta_2)\cos(\theta_3) - \sin(\theta_2)\sin(\theta_3)) \end{bmatrix}$$

For simplicity, the above matrix will be written as:

$$^{0}\mathbf{J}_u = \begin{bmatrix} -\beta_{00} & -\beta_{01} \\ \beta_{10} & \beta_{11} \end{bmatrix}$$

Thus, the velocities of the end-effector point can be written as follows:

$$\dot{x}_\epsilon = -\beta_{00}\dot{\theta}_2 - \beta_{01}\dot{\theta}_3 \qquad \dot{y}_\epsilon = \beta_{10}\dot{\theta}_2 + \beta_{11}\dot{\theta}_3 \tag{2.6}$$

At this point, the velocities $\dot{\theta}_3$ and $\dot{\theta}_4$ must be eliminated from the velocity equations of the end-effector point, because, as previously mentioned, these angles cannot be controlled by the motors.

From the relation of $\dot{x}_\epsilon$ in Eq. (2.5), solving for $\dot{\theta}_4$ results in:

$$\dot{\theta}_4 = -\frac{\dot{x}_\epsilon}{\alpha_{01}} - \frac{\alpha_{00}\dot{\theta}_1}{\alpha_{01}}$$

Next, substituting the expression for $\dot{\theta}_4$ into relation of $\dot{y}_\epsilon$ in Eq. (2.5) results in:

$$\dot{y}_\epsilon = -\frac{\alpha_{11}}{\alpha_{01}}\dot{x}_\epsilon + \frac{\alpha_{10}\alpha_{01} - \alpha_{11}\alpha_{00}}{\alpha_{01}}\dot{\theta}_1 \tag{2.7}$$

The same process is followed for Eq. (2.6)

$$\dot{\theta}_3 = -\frac{\dot{x}_\epsilon}{\beta_{01}} - \frac{\beta_{00}\dot{\theta}_2}{\beta_{01}}$$

$$\dot{y}_\epsilon = -\frac{\beta_{11}}{\beta_{01}}\dot{x}_\epsilon + \frac{\beta_{10}\beta_{01} - \beta_{11}\beta_{00}}{\beta_{01}}\dot{\theta}_2 \tag{2.8}$$

At this point, Eq. (2.7) and Eq. (2.8) are set equal to each other because, as previously mentioned, the end-effector points of the two systems have the same velocities.

(2.7) = (2.8) $\Leftrightarrow$

$$-\frac{\alpha_{11}}{\alpha_{01}}\dot{x_\epsilon} + \frac{\alpha_{10}\alpha_{01} - \alpha_{11}\alpha_{00}}{\alpha_{01}}\dot{\theta}_1 = -\frac{\beta_{11}}{\beta_{01}}\dot{x_\epsilon} + \frac{\beta_{10}\beta_{01} - \beta_{11}\beta_{00}}{\beta_{01}}\dot{\theta}_2 \Leftrightarrow$$

$$\boxed{\dot{x_\epsilon} = -\frac{\beta_{01}(\alpha_{10}\alpha_{01}-\alpha_{11}\alpha_{00})}{\beta_{01}\alpha_{01}-\alpha_{11}\beta_{01}}\dot{\theta}_1 + \frac{\alpha_{01}(\beta_{10}\beta_{01}-\beta_{11}\beta_{00})}{\beta_{11}\alpha_{01}-\alpha_{11}\beta_{01}}\dot{\theta}_2} \tag{2.9}$$

Substituting Eq. (2.9) into Eq. (2.7) results in:

$$\boxed{\dot{y_\epsilon} = (\frac{\alpha_{10}\alpha_{01}-\alpha_{11}\alpha_{00}}{\alpha_{01}} + \frac{\alpha_{11}\beta_{01}(\alpha_{10}\alpha_{01}-\alpha_{11}\alpha_{00})}{\alpha_{01}(\beta_{11}\alpha_{01}-\alpha_{11}\beta_{01})})\dot{\theta}_1 - \frac{\alpha_{11}(\beta_{10}\beta_{01}-\beta_{11}\beta_{00})}{\beta_{11}\alpha_{01}-\alpha_{11}\beta_{01}}\dot{\theta}_2} \tag{2.10}$$

Eq. (2.9) and Eq. (2.10) represent the symbolic solutions of the Differential Kinematics. Thus, the resulting Jacobian matrix is:

$${}^{0}\mathbf{J}_u = \begin{bmatrix} \gamma_{00} & \gamma_{01} \\ \gamma_{10} & \gamma_{11} \end{bmatrix}$$

$$\gamma_{00} = -\frac{\beta_{01}(\alpha_{10}\alpha_{01} - \alpha_{11}\alpha_{00})}{\beta_{01}\alpha_{01} - \alpha_{11}\beta_{01}} \qquad \gamma_{01} = \frac{\alpha_{01}(\beta_{10}\beta_{01} - \beta_{11}\beta_{00})}{\beta_{11}\alpha_{01} - \alpha_{11}\beta_{01}}$$

$$\gamma_{10} = (\frac{\alpha_{10}\alpha_{01} - \alpha_{11}\alpha_{00}}{\alpha_{01}} + \frac{\alpha_{11}\beta_{01}(\alpha_{10}\alpha_{01} - \alpha_{11}\alpha_{00})}{\alpha_{01}(\beta_{11}\alpha_{01} - \alpha_{11}\beta_{01})}) \qquad \gamma_{11} = -\frac{\alpha_{11}(\beta_{10}\beta_{01} - \beta_{11}\beta_{00})}{\beta_{11}\alpha_{01} - \alpha_{11}\beta_{01}}$$

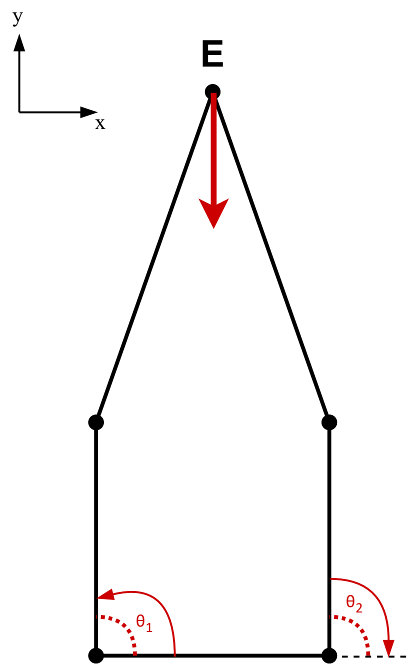
2.6 Validation of Kinematics and comparison of Numerical and Symbolic Solutions

To verify the correctness of the derived equations for the velocity of the end-effector point, two simple tests will be performed. In these tests, equal positive or negative velocity magnitudes will be applied to the two motor angles, and it will be checked, based on logic, whether the end-effector point moves in the correct direction.

Additionally, these tests will verify whether the symbolic solution of Differential Kinematics aligns with numerical solution.

Test 1:

Expected movement of the end-effector point in the negative direction of the y-axis.



For this test, the motor angles will be set to 90°. Additionally, a positive velocity will be assigned to angle θ_1, while a negative velocity of the same magnitude will be assigned to angle θ_2. These values are provided to the appropriate variables of the two Python programs presented earlier ((Symbolic solution) - (Numerical solution)).
The desired outcome, based on logic, is for the end-effector point to acquire a negative velocity along the y-axis, meaning it moves in the negative direction, while its velocity along the x-axis remains zero.

Angle	**Position**	**Velocity**
θ_1	90°	0.1745329252 rad/sec = 10 °/sec
θ_2	90°	- 0.1745329252 rad/sec = - 10 °/sec

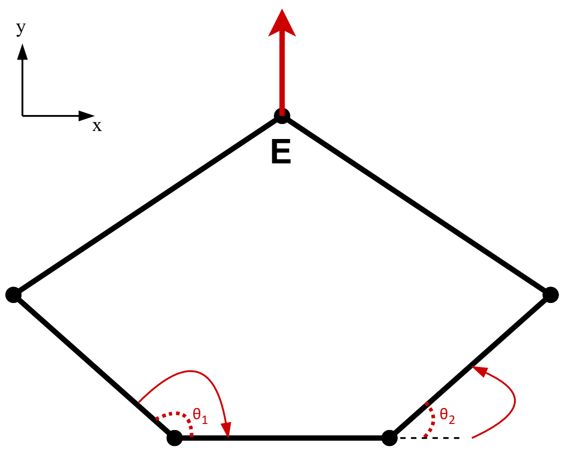
The results of the two programs are as follows:

Solution	**Velocity along the x-axis**	**Velocity along the y-axis**
Symbolic	- $0.20154 * 10^{-12}$ rad/sec $\approx$ 0 °/sec	- 0.0049365 rad/sec = - 0.282843 °/sec
Numerical	- $0.20153 * 10^{-12}$ rad/sec $\approx$ 0 °/sec	- 0.0049365 rad/sec = - 0.282843 °/sec

Initially, as observed, the two solutions align in terms of velocity values. Furthermore, as expected, the velocity along the y-axis is negative while along the x-axis it is approximately zero, confirming the correctness of the solutions.

Test 2:

Expected movement of the end-effector point in the positive direction of the y-axis.



For this test, motor angle θ_1 will be set to 135° and θ_2 will be set to 45°. Additionally, a positive velocity will be assigned to angle θ_2, while a negative velocity of the same magnitude will be assigned to angle θ_1. These values are provided to the appropriate variables of the two Python programs presented earlier ((Symbolic solution) - (Numerical solution)).

The desired outcome, based on logic, is for the end-effector point to acquire a positive velocity along the y-axis, meaning it moves in the positive direction, while its velocity along the x-axis remains zero.

Angle	**Position**	**Velocity**
θ_1	135°	- 0.6981317008 rad/sec = - 40 °/sec
θ_2	45°	0.6981317008 rad/sec = 40 °/sec

The results of the two programs are as follows:

Solution	**Velocity along the x-axis**	**Velocity along the y-axis**
Symbolic	- $0.3000100 * 10^{-12}$ rad/sec ≈ 0 °/sec	0.093028 rad/sec = 5.330112 °/sec
Numerical	- $0.3000030 * 10^{-12}$ rad/sec ≈ 0 °/sec	0.093028 rad/sec = 5.330112 °/sec

Initially, as observed, the two solutions align again in terms of velocity values. Furthermore, as expected, the velocity along the y-axis is positive while along the x-axis it is approximately zero, confirming the correctness of the solutions.

Chapter 3

PID Position Controller

A Proportional-Integral-Derivative (PID) position controller is a widely used control mechanism in robotic systems to accurately regulate the position of an end-effector. In this case, the objective of the PID controller is to drive the end-effector (point E) (Section 2.1) towards a desired target position while minimizing the position error. The controller continuously adjusts the actuator torques based on the current error, its rate of change, and its accumulated history, ensuring a stable and controlled motion.

This chapter examines the effect of each individual PID gain (Kp, Ki, and Kd) on the system's behavior. Specifically, by keeping two parameters constant while varying one, the impact of each gain on system performance will be studied separately. The primary focus is not to achieve the shortest possible movement time but to ensure a smooth and controlled motion trajectory.

To analyze the influence of the controller parameters, position and error plots will be presented in both the Cartesian space (describing the end-effector's position) and the actuator space (representing the motor angles). These plots provide valuable insights into how different gain values affect the system's response and overall motion stability.

3.1 Calculation of the Desired Motor Angles

Through the PID Controller, appropriate torques will be applied to the two motors to minimize the error, which is defined as the difference between the desired motor angles (formed by the joints controlled by the motors when the end-effector reaches and stabilizes at the target position) and the current motor angles at any given moment.

Therefore, given a desired final position of the end-effector, the corresponding desired motor angles must first be calculated. Since it is desirable to restrict the motion of the end-effector within the boundaries of the Cartesian space, ensuring that no unreachable target positions are assigned, the two following steps are followed.

Step 1: Find the final position (x , y coordinates) of end-effector point.

Regardless of the way the mechanism is used, the initial motor angles are set to 90 degrees, as these angles allow the joints to be positioned visually, without the need for any trigonometric tool.

Additionally, using forward kinematics (Section 2.1), given the known motor angles, the initial position of the end-effector can also be determined.

Therefore, the first step is to manually position the mechanism in its initial configuration and then move the end-effector to any desired position. Once it reaches and stabilizes at that position, its desired final position is computed and using sensors and forward kinematics.

Step 2: Find the desired motor angles.

The second step is to input the coordinates of the previously computed desired final position into the inverse kinematics equations (Section 2.2), thereby obtaining the desired motor angles.

Finally, some essential aspects that must be understood for the proper interpretation of the graphical representations presented in the following sections are as follows:

Initial position of end-effector point : $(x_0\ ,\ y_0) = (0.0400\ \text{m}\ ,\ 0.1931\ \text{m})$

Desired position of end-effector point : $(x_d\ ,\ y_d) = (0.0400\ \text{m}\ ,\ 0.0731\ \text{m})$

Initial position error of end-effector point : $(error_x\ ,\ error_y) = (0\ \text{m}\ ,\ 0.12\ \text{m})$

Initial position of motor angles : $(theta1_0\ ,\ theta2_0) = (90°\ ,\ 90°)$

Desired position of motor angles : $(theta1_d\ ,\ theta2_d) = (155.86°\ ,\ 24.14°)$

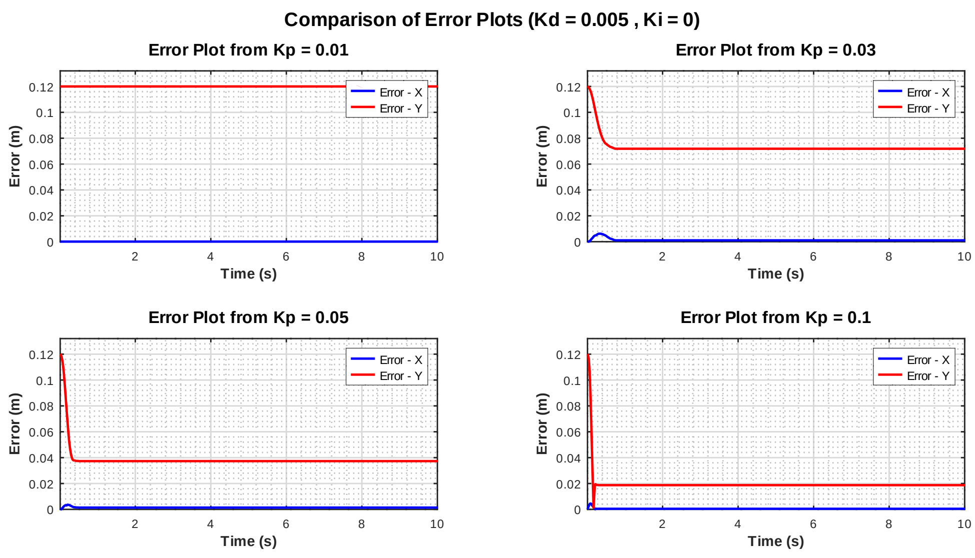
Initial position error of motor angles : $(error_1\ ,\ error_2) = (65.86°\ ,\ 65.86°)$

3.2 Analysis of Kp Gain

The proportional gain (Kp) is a fundamental component of the PID controller, directly influencing how the system responds to position errors. In the context of robotic motion control, Kp determines how forcefully the controller corrects deviations from the desired trajectory. A higher Kp results in stronger corrections, reducing the final position error, but it may also introduce oscillations or instability if not properly tuned.

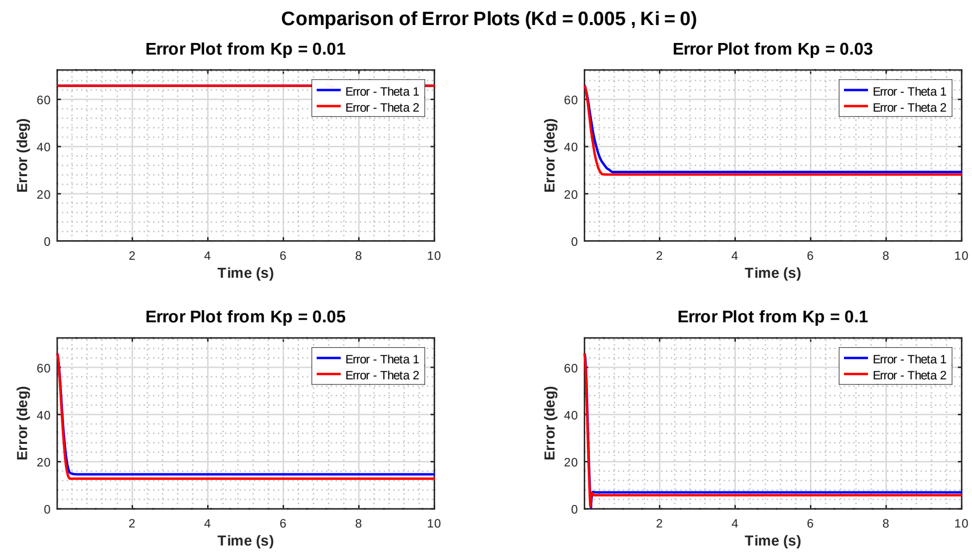
This section examines the impact of different Kp values on system performance. By keeping Ki and Kd constant, the response of the system is analyzed for varying proportional gain values, focusing on key performance metrics such as position accuracy, error convergence, and stability. The analysis is supported by error, position, and velocity plots, which illustrate the relationship between Kp and system behavior.

Graphical Representation of Position Error of End-Effector Point



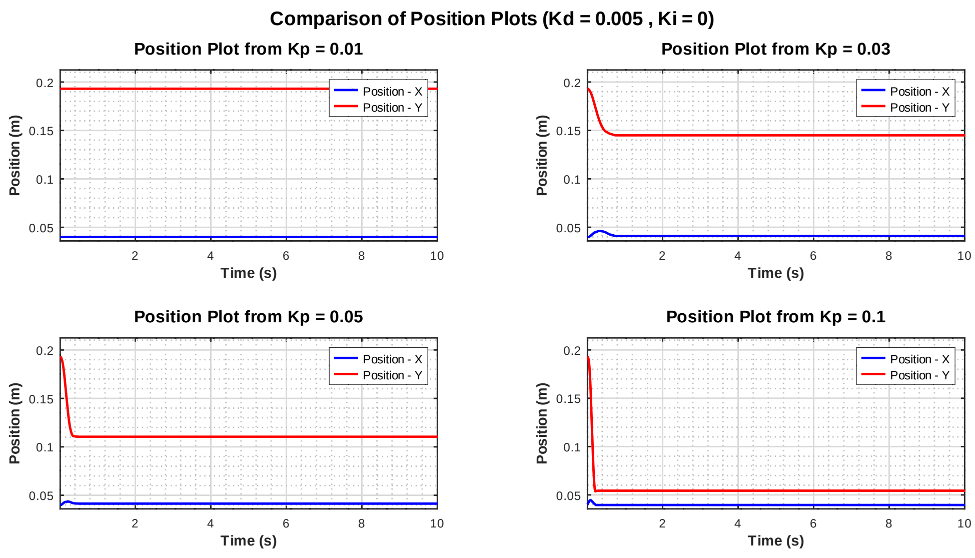
Based on the above graphical representations, it becomes evident that as Kp increases, the position error of end-effector decreases, allowing the system to reach the desired position more accurately and quickly. For low Kp values (e.g., 0.01), the error remains high, while higher values (e.g., 0.1) significantly reduce the steady-state error, particularly along the y-axis. The system responds faster with increasing Kp, however, excessive Kp values could introduce overshoot or oscillations, necessitating additional tuning, particularly of Kd, to ensure smoother convergence without instability.

Graphical Representation of Position Error of Motor Angles



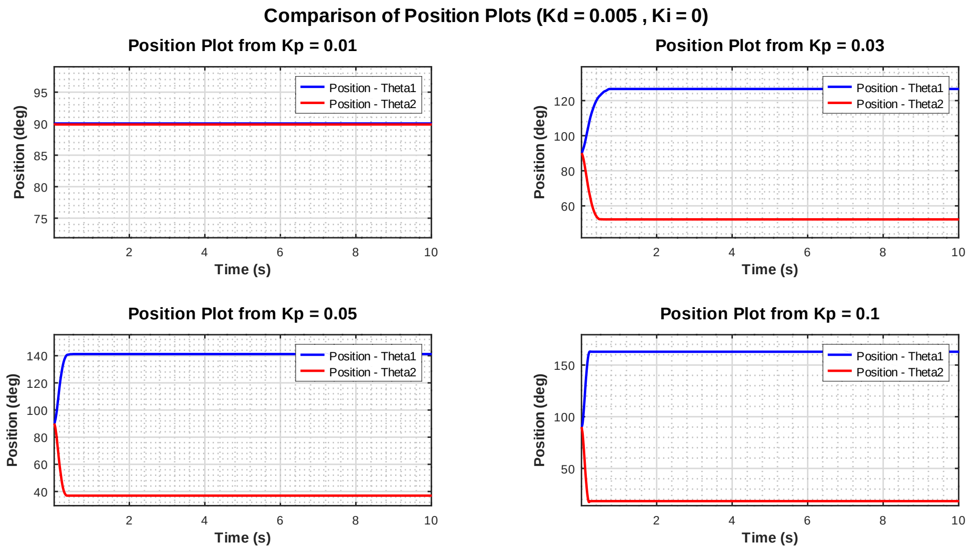
Similar to the position error of end-effector point, an increase in the Kp gain reduces the position error of the motor angles, leading to improved alignment with the desired angles. For low Kp values (e.g., 0.01), the error remains relatively high, indicating that the motors take longer to adjust their positions. Conversely, as Kp increases (e.g., 0.1), the system exhibits a significantly lower steady-state error, allowing motor angles to converge more efficiently to their target values.

Graphical Representation of Position of End-Effector Point



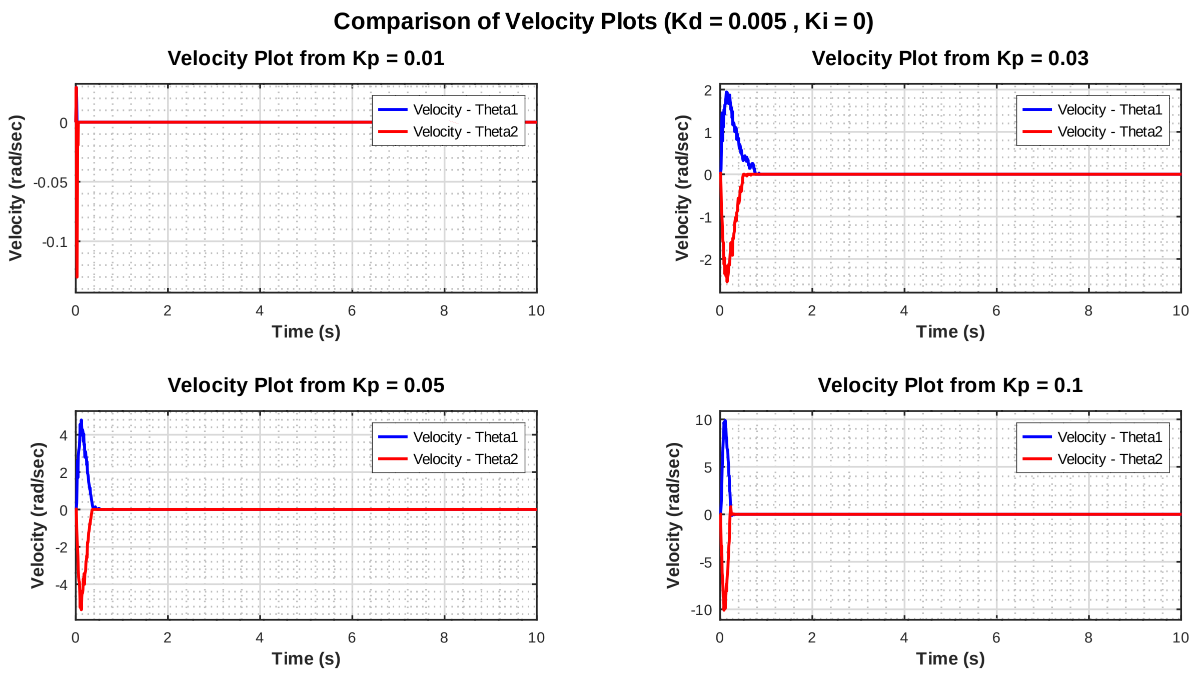
From the graphical representations above, it is clear that as Kp increases, the end-effector point reaches the desired position more rapidly and with greater accuracy. When Kp is low (e.g., 0.01), the system responds sluggishly, causing the end-effector to remain distant from its target position for a prolonged period. In contrast, as Kp increases (e.g., 0.03, 0.05, 0.1), the system achieves faster convergence toward the target position, particularly along the y-axis, where a significant reduction in settling time is observed. Moreover, for higher Kp values, the system stabilizes at the desired position much earlier, indicating enhanced response performance.

Graphical Representation of Position of Motor Angles



Based on the above graphical representations, it is evident that as Kp increases, motor angles converge more quickly to their desired values. For low Kp values (e.g., 0.01), motor angles remain close to their initial positions for a prolonged period, indicating a slow response. As Kp increases (e.g., 0.03, 0.05, 0.1), the system exhibits a much faster transition from the initial angles (90° for both motors) to the desired ones (155.86° and 24.14°). With higher Kp, the motion becomes significantly faster and more precise, reducing the time required for stabilization.

Graphical Representation of Velocity of Motor Angles



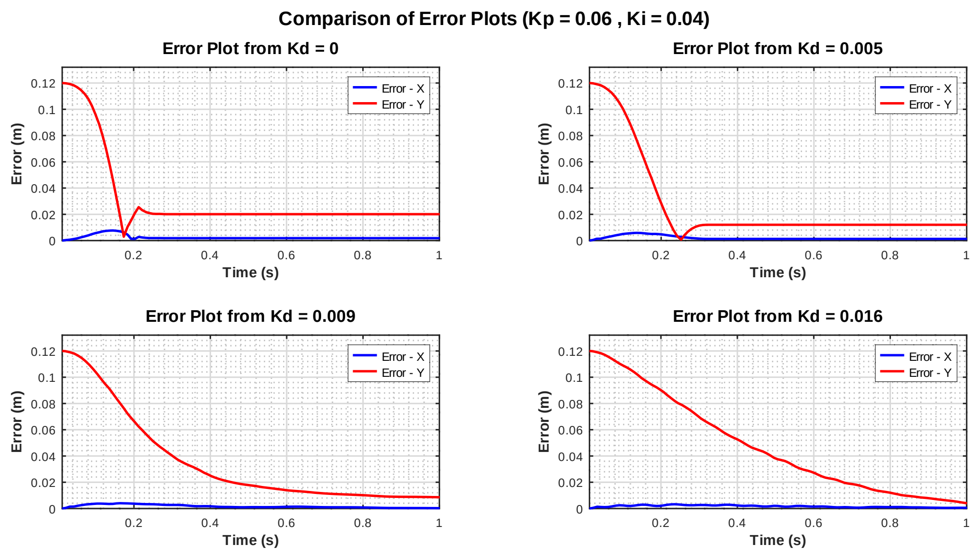
As Kp increases, motor velocities rise sharply, leading to a faster response. Low Kp values result in slow motor movement, while higher values (e.g., 0.1) cause higher peak velocities and quicker convergence. However, excessive Kp may introduce overshoot and instability, requiring proper Kd tuning for smoother motion.

3.3 Analysis of Kd Gain

The derivative gain (Kd) plays a crucial role in PID control, primarily improving the system's stability and response damping. In robotic motion control, Kd helps counteract rapid changes in error, reducing overshoot and minimizing oscillations that may arise from high Kp values. By predicting the system's future behavior based on the rate of error change, Kd enhances the smoothness of movement and prevents excessive corrections.

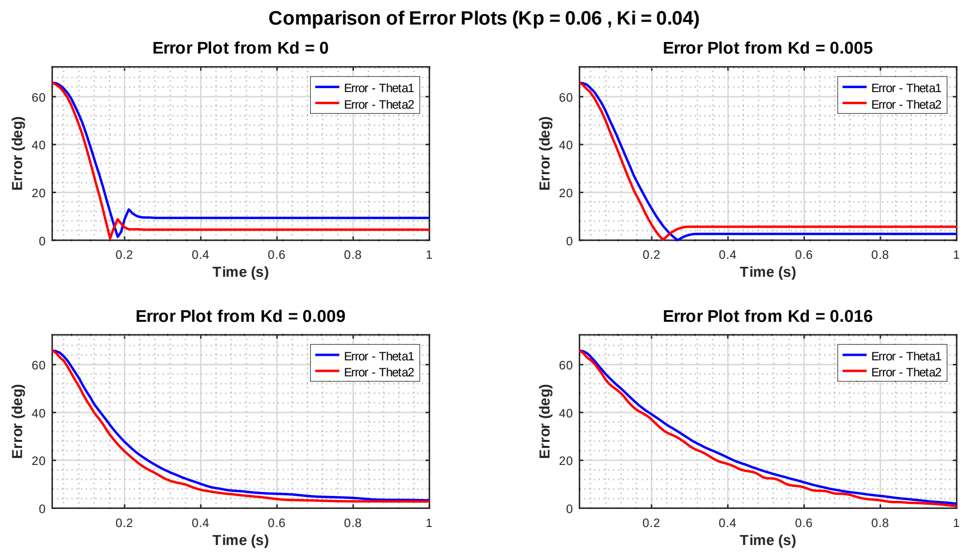
This section examines the effect of Kd on system performance while keeping Kp and Ki constant. The analysis explores how different Kd values influence error reduction, system damping, and velocity smoothness. To illustrate this, error, position, and velocity plots are presented, highlighting the relationship between Kd and the system behavior.

Graphical Representation of Position Error of End-Effector Point



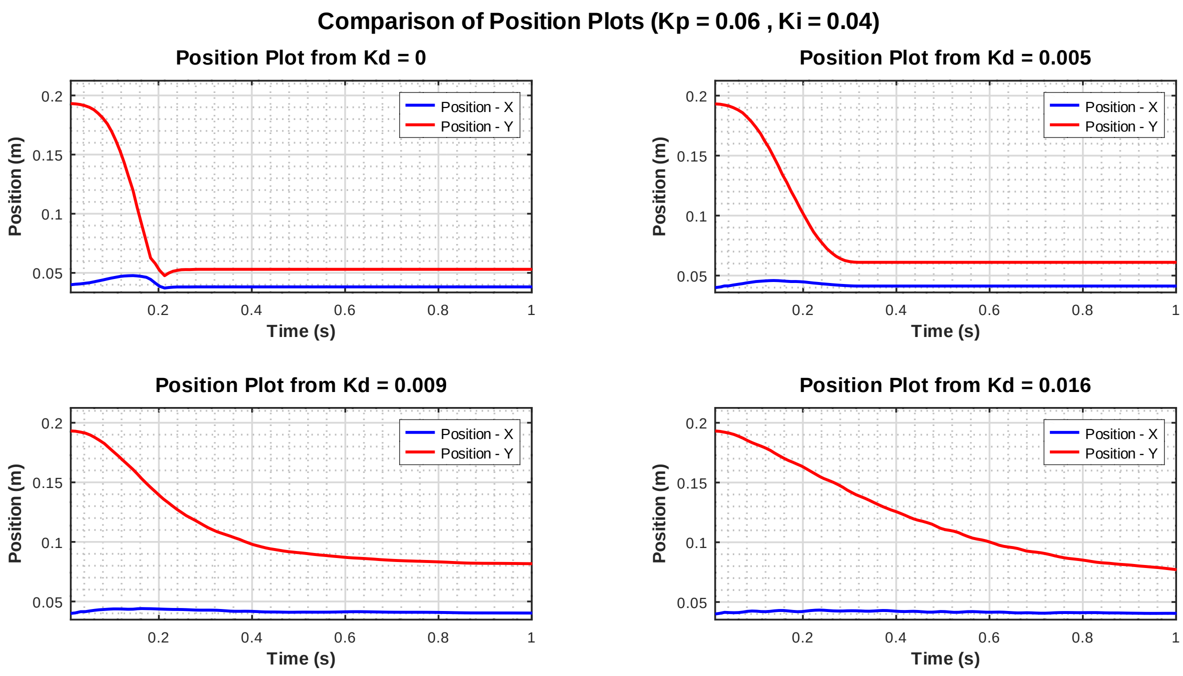
As the derivative gain (Kd) increases, the system exhibits improved damping, reducing overshoot and oscillations in the position error of the end-effector. At low Kd values (e.g., 0), significant oscillations and settling time are observed. Moderate Kd values (e.g., 0.005) effectively smooth the response, allowing the system to converge to the desired position faster. However, excessively high Kd values (e.g., 0.016) slow down the error reduction process, as the system becomes overly damped.

Graphical Representation of Position Error of Motor Angles



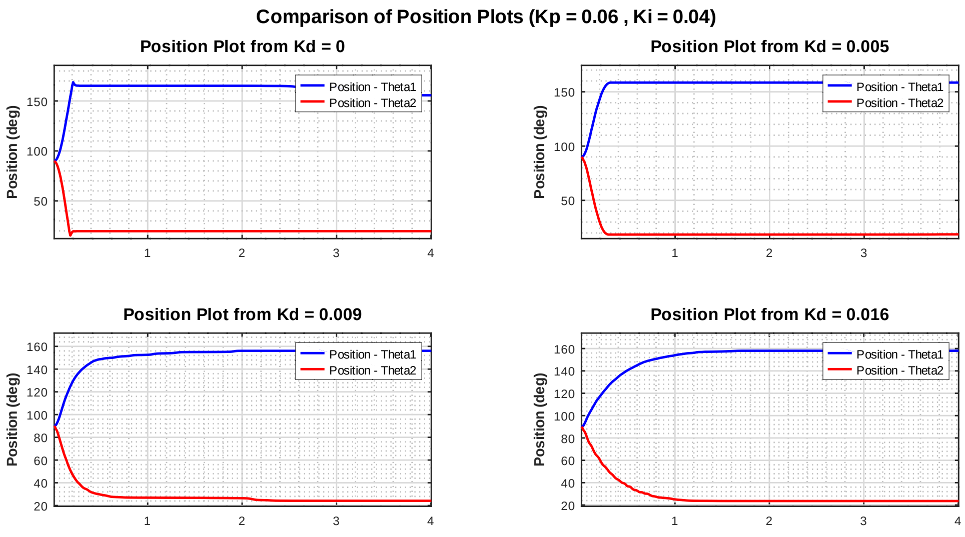
As the derivative gain (Kd) increases, the position error of motor angles experiences improved damping, effectively reducing oscillations and overshoot. When Kd = 0, significant fluctuations are observed before the angles reach stability. A moderate increase in Kd (e.g., 0.005) enhances the system's response, enabling faster convergence to the desired angles. However, excessively high values (e.g., 0.016) slow down the correction process, resulting in an overdamped system. Therefore, proper tuning of Kd is essential to achieve an optimal balance between error minimization and stability without introducing unnecessary delays.

Graphical Representation of Position of End-Effector Point



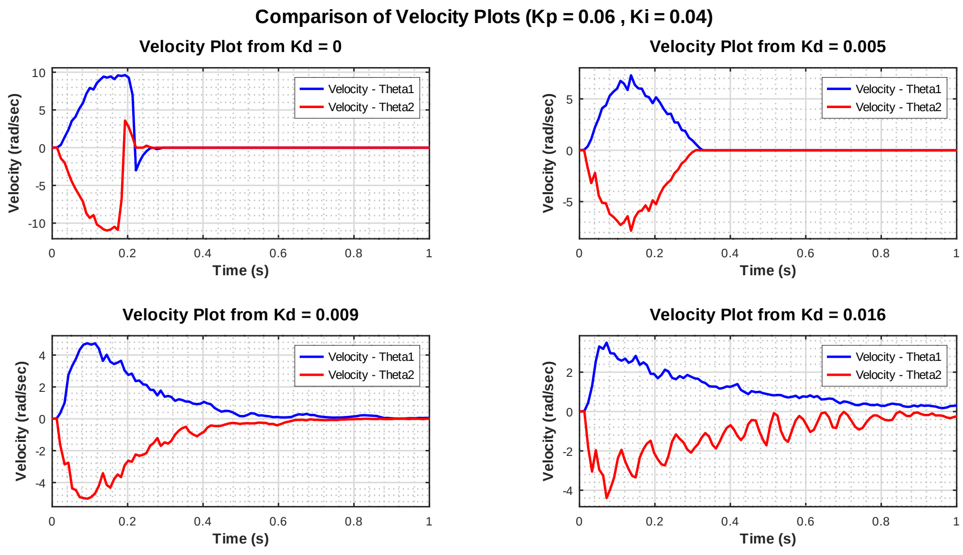
Similar to the previous cases, it is observed that as Kd increases, the movement of end-effector point becomes more stable, effectively reducing oscillations and overshoot. When Kd = 0, the system exhibits pronounced oscillations before reaching a steady state, whereas moderate values (e.g., 0.005) enhance stability and facilitate faster convergence. However, for higher values (e.g., 0.016), the system becomes overdamped, causing a slower approach to the desired position.

Graphical Representation of Position of Motor Angles



As the derivative gain (Kd) increases, motor angles exhibit improved damping, reducing oscillations and overshoot. When Kd is low (e.g., 0), motor angles reach the desired position with noticeable oscillations. In the other hand, with moderate Kd values (e.g., 0.005), the system stabilizes faster, achieving a smooth transition.

Graphical Representation of Velocity of Motor Angles



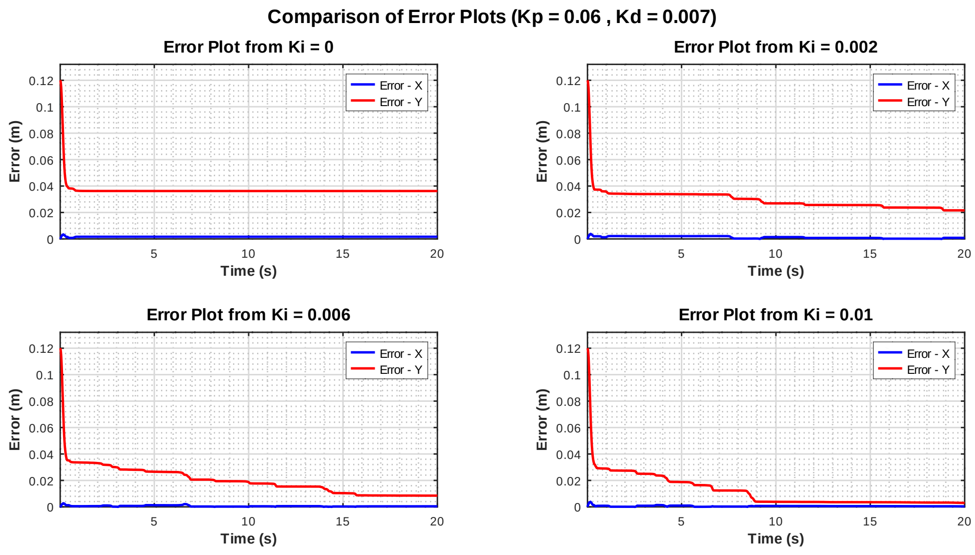
As the derivative gain (Kd) increases, the velocity response of motor angles becomes more stable, with reduced oscillations. When Kd is low (e.g., 0), the system exhibits high velocity peaks and oscillations, characteristic of an underdamped response. Moderate Kd values (e.g., 0.005) effectively suppress oscillations, resulting in a smoother velocity transition. However, excessively high Kd values (e.g., 0.016) lead to a slower response with small persistent oscillations, as the system becomes overdamped. Therefore, proper tuning of Kd is crucial to maintaining a balance between rapid response and smooth motion.

3.4 Analysis of Ki Gain

The integral gain (Ki) is a crucial component of the PID controller, primarily addressing steady-state errors that persist over time. In robotic motion control, Ki helps eliminate residual position errors by accumulating past errors and applying corrective actions based on their cumulative sum. This allows the system to achieve precise tracking of the desired trajectory, especially in the presence of external disturbances or model inaccuracies.

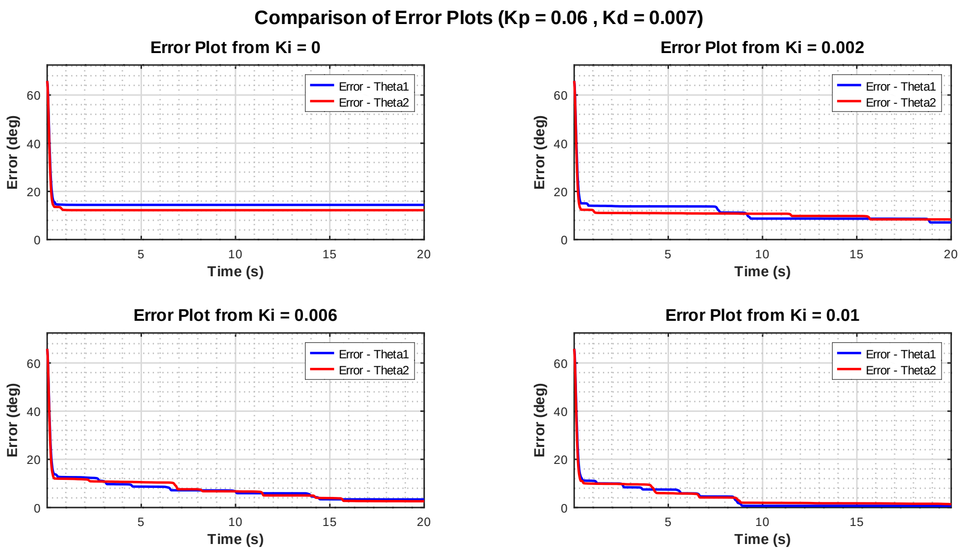
This section explores the impact of different Ki values on system performance. By keeping Kp and Kd constant, the system's response is analyzed for varying integral gain values, focusing on steady-state accuracy, error elimination, and stability. The analysis is supported by error, position, and velocity plots, providing insights into how Ki influences the system's ability to maintain the desired position over time.

Graphical Representation of Position Error of End-Effector Point



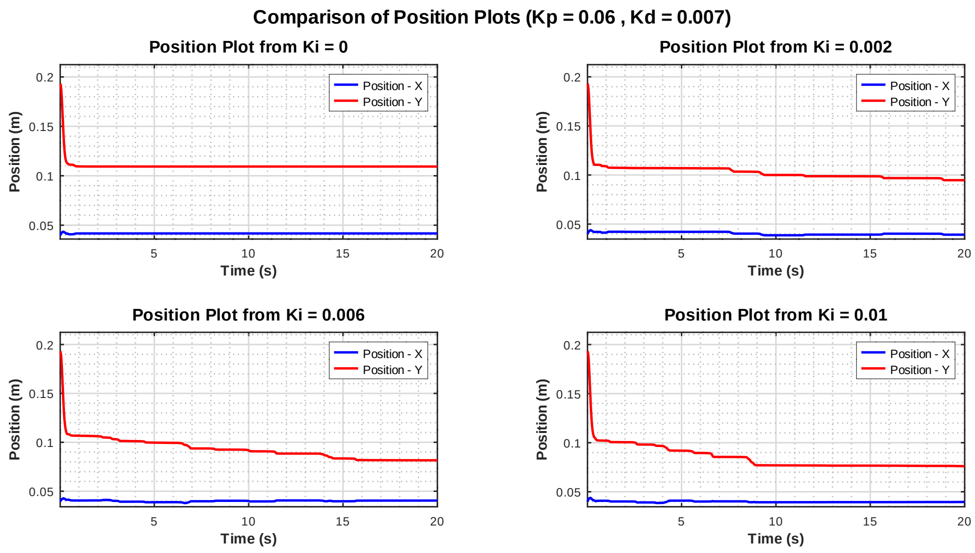
As the integral gain (Ki) increases, the steady-state error of end-effector position decreases, allowing the system to gradually correct accumulated deviations. At Ki = 0, a residual error remains, particularly along the y-axis. With moderate Ki values (e.g., 0.002), the system progressively reduces this error over time. Higher Ki values (e.g., 0.01) further improve error elimination but introduce slower convergence. Excessively high Ki may lead to instability or excessive overshooting, making precise tuning essential for maintaining balance between accuracy and stability.

Graphical Representation of Position Error of Motor Angles



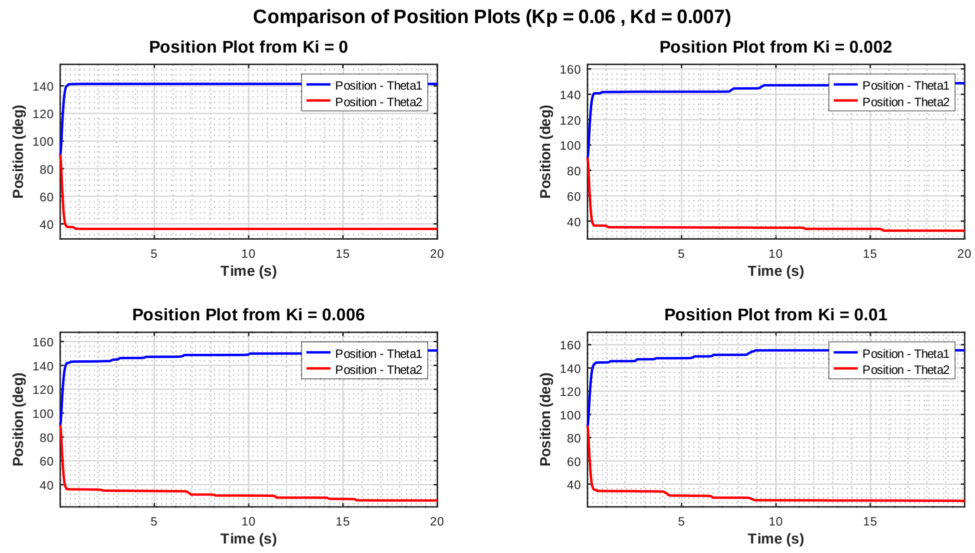
As Ki increases, the steady-state error in motor angles decreases, enabling the system to correct accumulated deviations. At Ki = 0, residual error prevents full convergence, while moderate values (e.g., 0.002) gradually reduce it. Higher Ki values (e.g., 0.01) improve correction but slow convergence.

Graphical Representation of Position of End-Effector Point



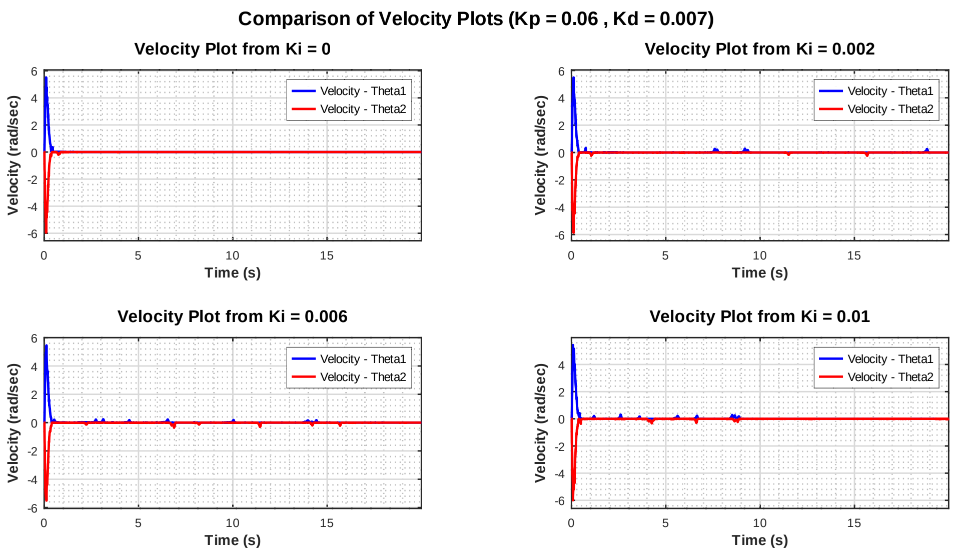
As the integral gain (Ki) increases, the position of end-effector gradually corrects accumulated deviations, reducing steady-state error. At Ki = 0, the system stabilizes but retains residual error. With moderate Ki values (e.g., 0.002), the position adjusts progressively, showing improved accuracy over time. Higher Ki values (e.g., 0.01) further enhance convergence to the desired position but slow the correction process.

Graphical Representation of Position of Motor Angles



As the integral gain (Ki) increases, motor angles gradually align more accurately with their desired values by compensating for accumulated errors. When Ki = 0, a steady-state error persists, preventing full correction. Moderate Ki values (e.g., 0.002) enhance the adjustment process over time, while higher values (e.g., 0.01) further improve convergence but slow stabilization. Excessively high Ki may introduce instability or oscillations, requiring careful tuning to balance accuracy and smooth motion.

Graphical Representation of Velocity of Motor Angles



As the integral gain (Ki) increases, the velocity response of motor angles becomes more stable and consistent over time. At Ki = 0, the velocity stabilizes quickly but leaves a steady-state error. With moderate Ki values (e.g., 0.002), small velocity adjustments occur to further correct deviations. Higher Ki values (e.g., 0.01) provide continuous error correction but slow down stabilization.

Appendix A

Symbolic Solution of Differential Kinematics (Jacobian Matrix) using Python

Below is the Python code that represents and calculates the symbolic solution of differential kinematics. Specifically, it calculates the velocities along the x and y axes of the end-effector point.

By examining the code, one can observe that the solutions of Forward Kinematics (Section 2.1) and the representation of angles θ_3 and θ_4 in terms of the motor angles (Section 2.4) are computed first, as these are used in the calculation of the velocities.

```
import sympy as sym
from sympy import diff, sin, cos, sqrt, acos, pi

theta1 = sym.Symbol('theta1')
theta2 = sym.Symbol('theta2')
theta1_dot = sym.Symbol('theta1_dot')
theta2_dot = sym.Symbol('theta2_dot')

l0 = sym.Symbol('l0')
l1 = sym.Symbol('l1')
l2 = sym.Symbol('l2')
l3 = sym.Symbol('l3')
l4 = sym.Symbol('l4')

a = ( ( sin(theta1)*l1 - sin(theta2)*l4 ) / ( l0 + \
cos(theta2)*l4  - cos(theta1)*l1 ) )

b = ( ( (sin(theta2))**2 * l4**2 + (l0 + cos(theta2)*l4)**2 \
- l1**2 + l2**2 - l3**2 ) / ( 2*(l0 + cos(theta2)*l4 \
- cos(theta1)*l1) ) )

c = ( (a**2) + 1 )
d = ( (2*a*b) - (2*a*cos(theta1)*l1) - (2*sin(theta1)*l1) )
e = ( b**2 - 2*b*cos(theta1)*l1 + l1**2 - l2**2 )

ye = ( ( -d + sqrt(d**2 - 4*c*e) ) / ( 2*c ) )
xe = ( (a*ye) + b )

theta3 = pi - acos( (-(xe)**2 - (ye)**2 + 2*xe*l0 \
- (l0)**2 + (l3)**2 + (l4)**2 ) / (2*l3*l4) )

theta4 = pi + acos( (-(xe)**2 - (ye)**2 + (l1)**2 \
+ (l2)**2 ) / (2*l1*l2) )

a00 = l1*sin(theta1) + l2*(sin(theta1)*cos(theta4) \
+ cos(theta1)*sin(theta4))

a01 = l2*(sin(theta1)*cos(theta4) + cos(theta1) \
*sin(theta4))

a10 = l1*cos(theta1) + l2*(cos(theta1)*cos(theta4) \
- sin(theta1)*sin(theta4))

a11 = l2*(cos(theta1)*cos(theta4) - sin(theta1) \
*sin(theta4))

b00 = l4*sin(theta2) + l3*(sin(theta2)*cos(theta3) \
+ cos(theta2)*sin(theta3))

b01 = l3*(sin(theta2)*cos(theta3) + cos(theta2) \
*sin(theta3))

b10 = l4*cos(theta2) + l3*(cos(theta2)*cos(theta3) \
- sin(theta2)*sin(theta3))

b11 = l3*(cos(theta2)*cos(theta3) - sin(theta2) \
*sin(theta3))

xe_dot = -( b01*( (a10*a01 - a11*a00) / (b11*a01 \
- a11*b01) ) )*theta1_dot + ( a01*( \
(b10*b01 - b11*b00) / (b11*a01 - \
a11*b01) ) )*theta2_dot

ye_dot = ( ( (a10*a01 - a11*a00) / (a01) ) + ( \
(a11*b01*(a10*a01 - a11*a00)) / (a01* \
(b11*a01 - a11*b01)) ) )*theta1_dot - \
( a11*( (b10*b01 - b11*b00) / (b11*a01 \
- a11*b01) ) )*theta2_dot
```

Appendix B

Numerical Solution of Differential Kinematics (Jacobian Matrix) using Python

As previously mentioned, the velocities along the two axes of the end-effector point can be easily found by differentiating the position equations obtained through Forward Kinematics (Section 2.1).

Below is a simple Python program that uses *diff* function provided by *sympy* library in Python. Variables f00, f01, f10 and f11 constitute the four elements of Jacobian matrix.

```
import sympy as sym
from sympy import diff, sin, cos, sqrt

theta1 = sym.Symbol('theta1')
theta2 = sym.Symbol('theta2')
theta1_dot = sym.Symbol('theta1_dot')
theta2_dot = sym.Symbol('theta2_dot')

l0 = sym.Symbol('l0')
l1 = sym.Symbol('l1')
l2 = sym.Symbol('l2')
l3 = sym.Symbol('l3')
l4 = sym.Symbol('l4')

a = ( ( sin(theta1)*l1 - sin(theta2)*l4 ) / ( l0 + \
cos(theta2)*l4 - cos(theta1)*l1 ) )

b = ( ( (sin(theta2))**2 * l4**2 + (l0 + \
cos(theta2)*l4)**2 - l1**2 + l2**2 - l3**2 ) / \
( 2*(l0 + cos(theta2)*l4 - cos(theta1)*l1) ) )

c = ( (a**2) + 1 )
d = ( (2*a*b) - (2*a*cos(theta1)*l1) - (2*sin(theta1)*l1) )
e = ( b**2 - 2*b*cos(theta1)*l1 + l1**2 - l2**2 )

ye = ( ( -d + sqrt(d**2 - 4*c*e) ) / ( 2*c ) )
xe = ( (a*ye) + b )

f00 = diff(xe,theta1)
f10 = diff(ye,theta1)
f01 = diff(xe,theta2)
f11 = diff(ye,theta2)

xe_dot = f00*theta1_dot + f01*theta2_dot
ye_dot = f10*theta1_dot + f11*theta2_dot
```

Appendix C

PID Position Controller using Python

Below is the Python code implementing the PID position controller analyzed in Chapter 3.

```
import nidaqmx
from nidaqmx.constants import LineGrouping
from nidaqmx.constants import EncoderType, AngleUnits
import time
from configure_encoder import configure_encoder
import math
from inverse_kinematics import inverse_kinematics
from forward_kinematics import forward_kinematics

# Device and channel configuration
device_name_1 = "Dev1"
analog_channel_1 = f"{device_name_1}/ao1"  # Analog Output channel AO1
analog_channel_2 = f"{device_name_1}/ao0"  # Analog Output channel AO0
digital_channel_name = f"{device_name_1}/port0/line0"  # Digital Output
                                     channel (P0.0)

device_name_2 = "Dev2"                        # Device name for the
                                     encoders
channel_name0 = f"{device_name_2}/ctr0"       # Counter 0 for the first
                                     encoder
channel_name1 = f"{device_name_2}/ctr1"       # Counter 1 for the second
                                     encoder
initial_angle = 90.0
pulses_per_rev = 9750
a0_input_term = "PFI0"
b0_input_term = "PFI1"
a1_input_term = "PFI2"
b1_input_term = "PFI3"
```

```
reading_task_0 = nidaqmx.Task()
reading_task_1 = nidaqmx.Task()
analog_task_a01 = nidaqmx.Task()
analog_task_a00 = nidaqmx.Task()
digital_task = nidaqmx.Task()

configure_encoder(reading_task_0, channel_name0, a0_input_term,
                                    b0_input_term, pulses_per_rev,
                                    initial_angle)
configure_encoder(reading_task_1, channel_name1, a1_input_term,
                                    b1_input_term, pulses_per_rev,
                                    initial_angle)

sampling_period = 1 / 200   # Define the sampling period: 200 Hz (5 ms per
                                    sample)

# Voltages to output
output_voltage_a00 = 0
output_voltage_a01 = 0
digital_enable = True

x_d = 0.0400
y_d = 0.0731
theta1 = theta2 = math.radians(initial_angle)
theta1_d, theta2_d = inverse_kinematics(x_d, y_d)

error1 = theta1_d - theta1
error2 = theta2_d - theta2

# PID controller gains
Kp = 0.06
Kd = 0.01
Ki = 0.04

# Initialize previous errors, integrals, and time for PID control
prev_error1 = error1
prev_error2 = error2
integral_error1 = 0
integral_error2 = 0
prev_time = time.time()

prev_theta1 = theta1
prev_theta2 = theta2

try:

reading_task_0.start()
reading_task_1.start()

# Add an analog output voltage channel for A01
analog_task_a01.ao_channels.add_ao_voltage_chan(
physical_channel=analog_channel_1,
min_val=-9.5,
max_val=9.5
)

# Add an analog output voltage channel for AO0
analog_task_a00.ao_channels.add_ao_voltage_chan(
physical_channel=analog_channel_2,
min_val=-9.5,
max_val=9.5
)

# Add a digital output channel
digital_task.do_channels.add_do_chan(
lines=digital_channel_name,
line_grouping=LineGrouping.CHAN_PER_LINE
)

digital_task.write(digital_enable, auto_start=True)

xe, ye = forward_kinematics(theta1, theta2)

output_volt_file = open("output_volt.txt", "w")
output_volt_file.write("V0, V1\n")  # Add header to the file

error_file = open("error.txt", "w")
error_file.write("error1, error2, time\n")  # Add header to the file

velocity_file = open("velocity.txt", "w")
velocity_file.write("velocity1, velocity2, time\n") # Add header to the
                                     file

position_error_file = open("position_error.txt", "w")
position_error_file.write("x_error, y_error, time\n") # Add header to the
                                     file

position_file = open("position.txt", "w")
position_file.write("xe, ye, time\n") # Add header to the file

position_deg_file = open("position_deg.txt", "w")
position_deg_file.write("theta1, theta2, time\n") # Add header to the file

error_time_start = time.time()

while time.time() - error_time_start < 20:

start_time = time.time()

# Calculate time elapsed since last loop
delta_time = start_time - prev_time

velocity1 = (theta1 - prev_theta1) / delta_time if delta_time > 0 else 0
velocity2 = (theta2 - prev_theta2) / delta_time if delta_time > 0 else 0
velocity_file.write(f"{velocity1:.5f}, {velocity2:.5f}, {time.time() -
                                    error_time_start:.5f}\n")

# Integral term for errors
integral_error1 += error1 * delta_time
integral_error2 += error2 * delta_time

# Derivative term for errors
if delta_time > 0:
d_error1 = (error1 - prev_error1) / delta_time
d_error2 = (error2 - prev_error2) / delta_time
else:
d_error1 = 0
d_error2 = 0

# PID Control calculations
T1 = Kp * error1 + Kd * d_error1 + Ki * integral_error1
T2 = Kp * error2 + Kd * d_error2 + Ki * integral_error2

# Convert torques to output voltages
output_voltage_a00 = 91.35 * T1
output_voltage_a01 = 91.35 * T2

# Voltage saturation [-10V, 10V]
output_voltage_a00 = max(min(output_voltage_a00, 9.5), -9.5)
output_voltage_a01 = max(min(output_voltage_a01, 9.5), -9.5)

# Write voltages to file
output_volt_file.write(f"{output_voltage_a00:.4f}, {output_voltage_a01:.4f}
                                    \n")

# Send voltages to the analog outputs
analog_task_a00.write(output_voltage_a00, auto_start=True)
analog_task_a01.write(output_voltage_a01, auto_start=True)

prev_theta1 = theta1
prev_theta2 = theta2

# Read angular positions from encoders
theta1 = math.radians(reading_task_0.read())
theta2 = math.radians(reading_task_1.read())
#print(f"Theta1: {theta1:.5f} rad, Theta2: {theta2:.5f} rad")
theta1_deg = math.degrees(theta1)
theta2_deg = math.degrees(theta2)
position_deg_file.write(f"{theta1_deg:.5f}, {theta2_deg:.5f}, {time.time()
                                    - error_time_start:.5f}\n")

prev_error1 = error1
prev_error2 = error2

# Update errors
error1 = theta1_d - theta1
error2 = theta2_d - theta2
error1_deg = math.degrees(error1)
```

```
error2_deg = math.degrees(error2)
print(f"error_theta1: {abs(error1_deg):.5f} deg, error_theta2: {abs(
                                    error2_deg):.5f} deg\n")
error_time = time.time() - error_time_start
error_file.write(f"{abs(error1_deg):.5f}, {abs(error2_deg):.4f}, {
                                    error_time:.5f}\n")

# Update position for logging
xe, ye = forward_kinematics(theta1, theta2)
position_file.write(f"{xe:.5f}, {ye:.5f}, {time.time() - error_time_start:.
                                    5f}\n")
print(f"X: {xe:.4f} m, Y: {ye:.4f} m")
error_x = x_d - xe
error_y = y_d - ye
print(f"error_x: {abs(error_x)*100:.5f} cm, error_y: {abs(error_y)*100:.5f}
                                     cm\n")
position_error_file.write(f"{abs(error_x):.5f}, {abs(error_y):.5f}, {time.
                                    time() - error_time_start:.5f}\n")

# Update time for next iteration
prev_time = start_time

# Maintain constant loop frequency
elapsed_time = time.time() - start_time
sleep_time = sampling_period - elapsed_time
if sleep_time > 0:
time.sleep(sleep_time)

print("---------------------------------")

output_voltage_a00 = 0
output_voltage_a01 = 0
analog_task_a01.write(output_voltage_a01, auto_start=True)
analog_task_a00.write(output_voltage_a00, auto_start=True)
digital_enable = False
digital_task.write(digital_enable, auto_start=True)

except KeyboardInterrupt:
output_voltage_a00 = 0
output_voltage_a01 = 0
analog_task_a01.write(output_voltage_a01, auto_start=True)
analog_task_a00.write(output_voltage_a00, auto_start=True)
digital_enable = False
digital_task.write(digital_enable, auto_start=True)
print("Stopped.")  # Handle program termination with Ctrl+C

finally:
# Stop and close the tasks
analog_task_a01.stop()
analog_task_a01.close()
analog_task_a00.stop()
analog_task_a00.close()
digital_task.stop()
digital_task.close()
reading_task_0.stop()
reading_task_0.close()
reading_task_1.stop()
reading_task_1.close()
print("Tasks stopped and closed.")
```