Ben-Gurion University of the Negev
The Faculty of Engineering Sciences
The Department of **Mechanical Engineering**

Final Project: Robot Kinematics and Dynamics

Designing a Robotic Arm with 6 Degrees of Freedom

Students Names:

Pele Ovadia – 206679649
Yuval Marmor – 209191568

Course Lecturer: Professor Amir Shapira

April 2025

Contents

Initial Summary

This project presents the comprehensive design and analysis of a 6-degree-of-freedom (6-DOF) robotic arm for automated date sorting at the "Beit Katan" packaging facility in Kibbutz Eilot. The SCARA-type robotic system features four rotational joints (θ_1, θ_2, θ_3, θ_4) and two prismatic joints (l_1, l_2), designed to work with a vision system for precise pick-and-place operations between conveyor systems. The project delivers a complete kinematic and dynamic analysis framework including forward and inverse kinematics with transformation matrices, workspace analysis showing a working envelope of 800 mm outer radius and 1450 mm maximum height, Jacobian matrix derivation for velocity and force calculations, static analysis for joint torques under 1 kg payloads, trapezoidal velocity profile path planning for smooth motion, and Python-based simulation with real-time 3D visualization. Validation through five test points confirms the accuracy of both kinematic solutions, with simulation results matching analytical predictions. The system successfully addresses the industrial automation challenge of replacing manual sorting operations, improving efficiency and reducing human error in agricultural product processing.

1 Introduction

In the Robot Kinematics and Dynamics course, we learned several analytical methods to calculate the relationship between a robotic arm's end effector and its base within a global coordinate system. These methods help us analyze and control robot motion, giving us a clear understanding of how the arm moves and positions itself in space.

1.1 Problem Description

At the "Beit Katan" packaging facility located in Kibbutz Eilot, dates harvested from the kibbutz fields are sorted and packaged for domestic and international markets. Currently, the initial sorting process is performed manually by workers to remove damaged dates from the production line. This work is typically done by young people from the kibbutz.

Due to the planned privatization of the kibbutz, management decided to optimize the facility and reduce dependence on manual labor. The solution was to integrate a 6-degree-of-freedom robotic arm into the production line that can move around the conveyor belt, thereby increasing the packaging facility's efficiency and reducing human errors.

A conveyor will transport the dates, and with the help of a camera and scanner installed above the conveyor, the arm can be directed to damaged dates. The arm will pick up these defective dates and transfer them to a separate conveyor for damaged dates located on the other side.

1.2 Illustration of the Selected Robot

Figure 1 shows the selected robotic arm configuration, displaying the overall design and structure of the 6-DOF SCARA system implemented in the RViz simulation environment.

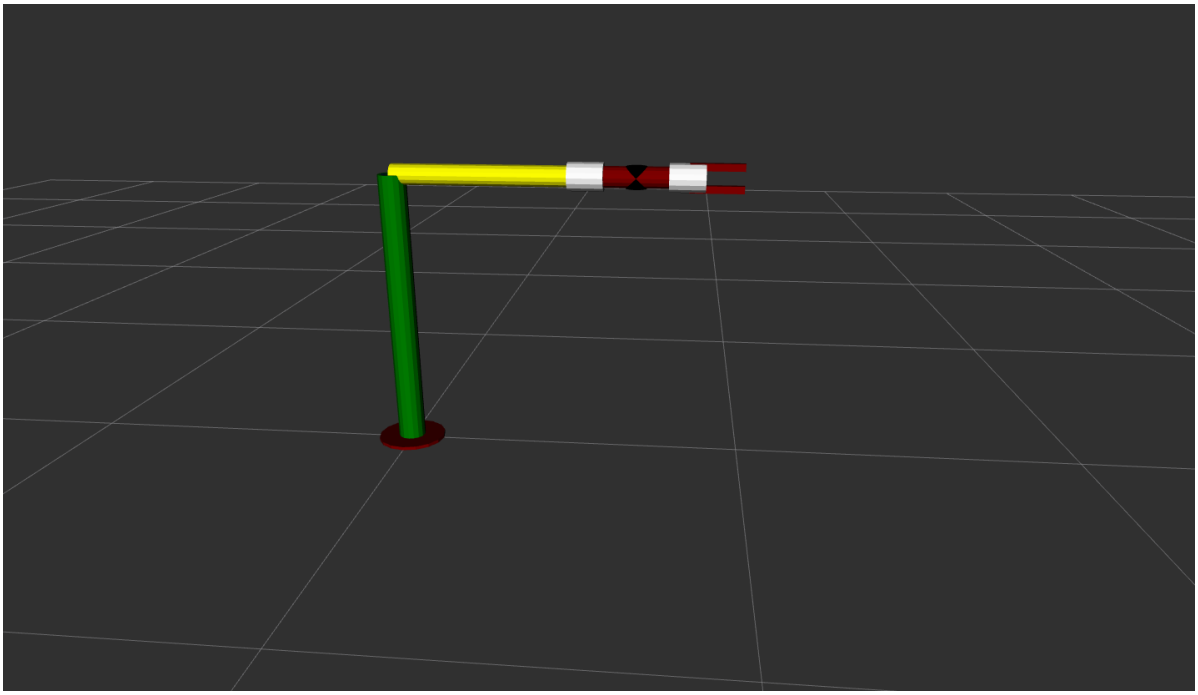

Figure 1: **Computer model of the selected robotic arm displayed in RViz-ROS**

A link to view the robotic arm video is provided in Appendix 1.

1.3 Parameters That Can Be Changed

Figure 2 presents a schematic diagram of the robot showing all the degrees of freedom that can be modified. This robotic arm has 6 degrees of freedom.

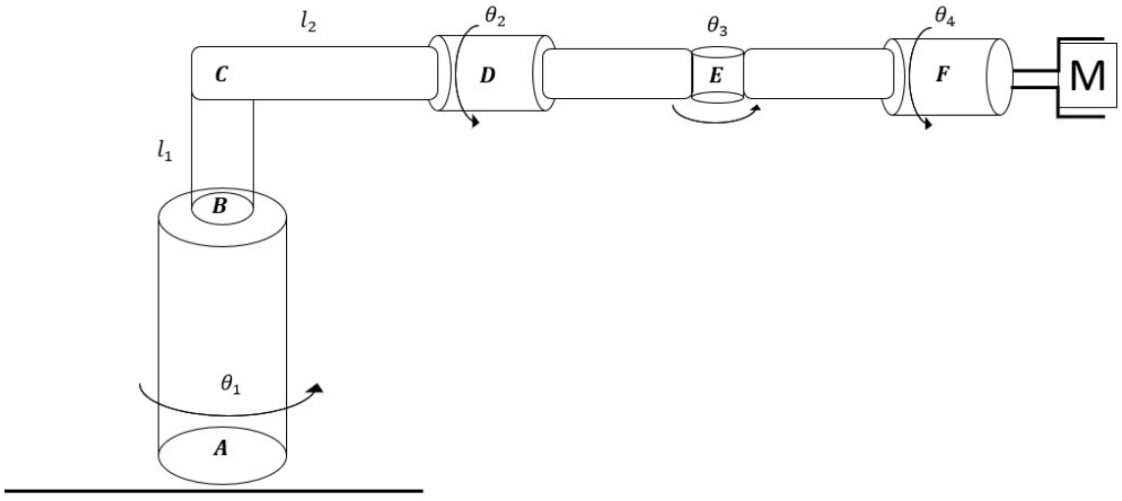

Figure 2: **Schematic diagram of the selected arm with adjustable parameters**

In this figure, we can see four rotational degrees of freedom: θ_1, θ_2, θ_3, θ_4, and two prismatic degrees of freedom: l_1, l_2 of the robotic arm selected to perform the task.

1.4 Simulation Environment

The simulation of this robotic arm was performed in a PYTHON environment. This software was chosen due to our familiarity with it and its relatively simple programming interface. The robot's kinematics are implemented using a stick diagram, where calculations treat the arm dimensions as 1-dimensional lines.

2 The Robot

The selected robotic arm is a SCARA type robot, chosen for its simplicity and low cost, making it well-suited for the packaging facility's requirements. The selected arm does not need to perform complex path planning for its application, and essentially implements a Place Pick type path planning that does not require control during movement.

2.1 Why This Robot Was Selected

This robotic arm was selected because it allows picking up defective dates from conveyor A and transferring them to conveyor B, which is designated for defective dates. The main reason this arm was chosen is that it provides a wide working range, which is required in the intended work environment, while remaining simpler than other alternatives that use only rotational degrees of freedom. Additionally, the kinematics of this arm are easier to understand spatially and to program, making it easier to work with.

2.2 Types of Tasks the Robot Is Suitable For

The selected robot belongs to the category of stationary robots, where these types of robots are mainly used in industry on production lines, so they are called industrial robots. This robot is built from an arm (constructed from several joints) and an end-effector (usually a gripper) designed to maintain physical contact with objects in its environment and thereby perform its tasks.

2.3 Robot Dimensions

The robot's dimensions are shown in the following figures:

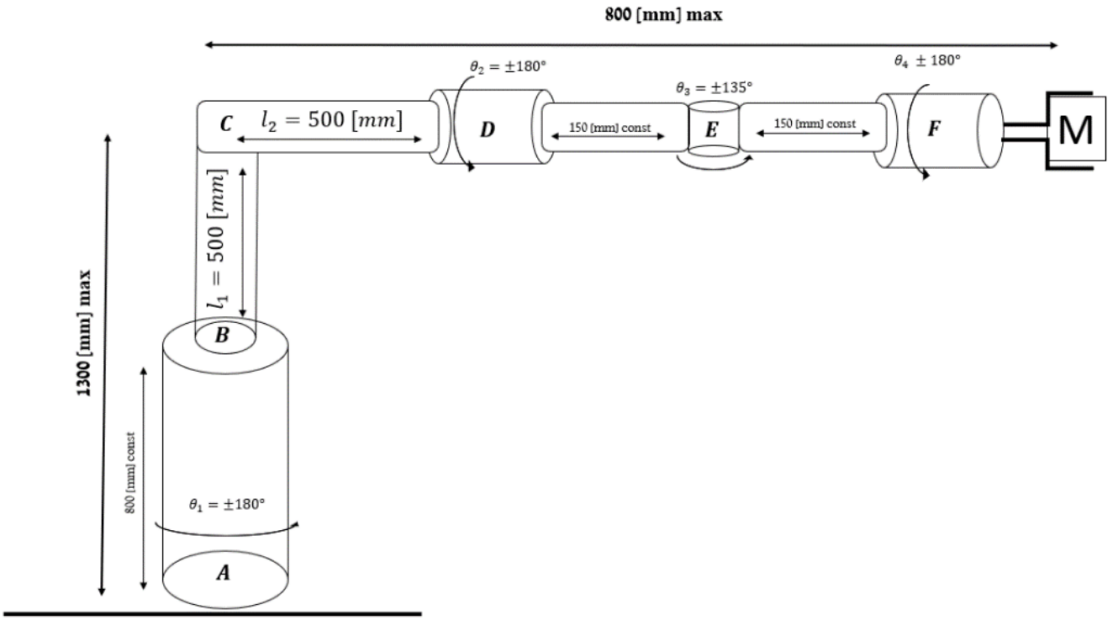

Figure 3: **Maximum dimensions of the robotic arm**

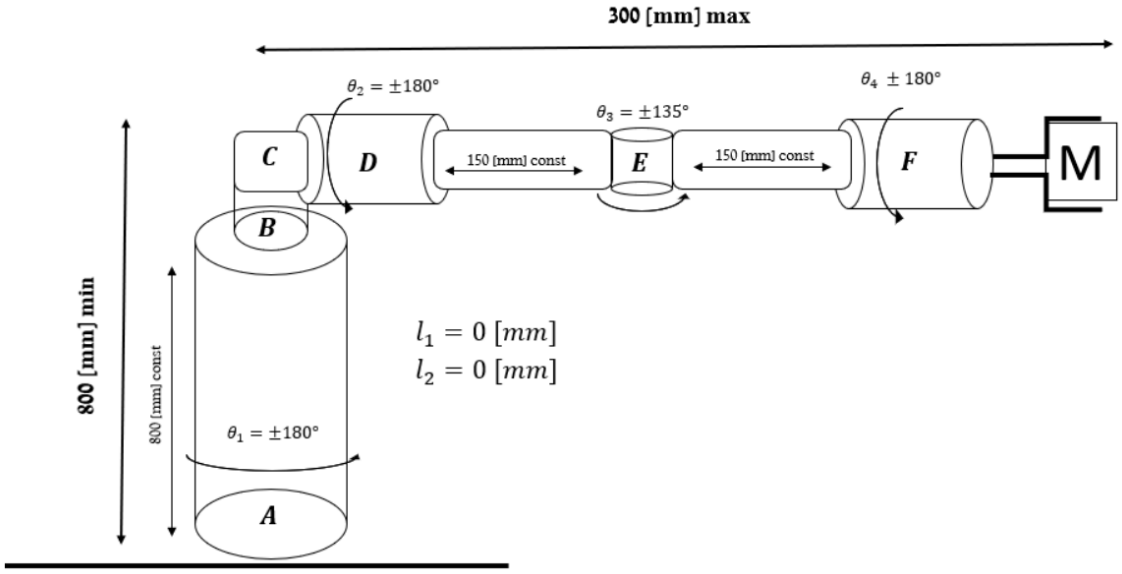

Figure 4: **Minimum dimensions of the robotic arm**

The following table shows the motion ranges of the robot joints:

Table 1: Motion ranges of the robotic arm joints

Joint	Symbol	Working Range
1	θ_1	$\pm 180\ [deg]$
2	l_1	$0 - 500\ [mm]$
3	l_2	$0 - 500\ [mm]$
4	θ_2	$\pm 180\ [deg]$
5	θ_3	$\pm 135\ [deg]$
6	θ_4	$\pm 180\ [deg]$

2.4 Coordinate Systems

The following figure shows the coordinate systems attached to the joints:

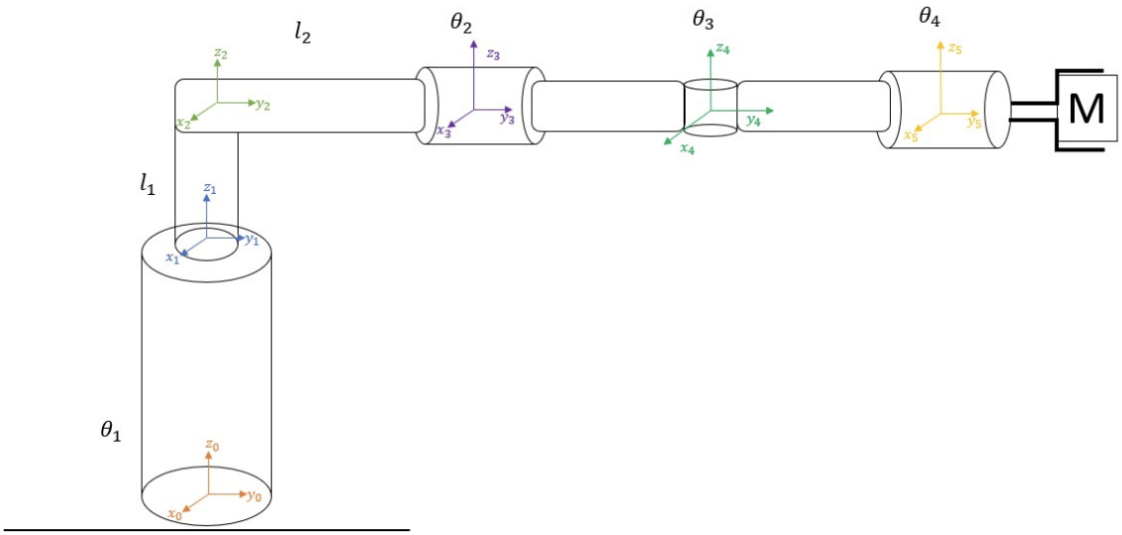

Figure 5: **Coordinate systems of the robotic arm**

3 Forward Kinematics

The forward kinematics problem involves calculating the position of the end-effector in space when we know the robot's joint parameters - the link lengths and joint angles. In this section, we will create transformation matrices between coordinate systems to calculate the end-effector position in the global coordinate frame.

3.1 Transformation Matrices of the Coordinate Systems

We now present the transformation matrices that allow us to move from one coordinate frame to the next, for each frame shown in the previous section.

$$ {}^{0}A_1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 800 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$
{}^{1}A_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{2}A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & l_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{3}A_4 = \begin{bmatrix} \cos(\theta_2) & 0 & \sin(\theta_2) & 0 \\ 0 & 1 & 0 & 150 \\ -\sin(\theta_2) & 0 & \cos(\theta_2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4}A_5 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & -150 \cdot \sin(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 150 \cdot \cos(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{5}A_6 = \begin{bmatrix} \cos(\theta_4) & 0 & \sin(\theta_4) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_4) & 0 & \cos(\theta_4) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

3.2 Homogeneous Transformation Matrix of the End-Effector

To find the position and orientation of the end-effector in the world frame, we multiply the transformation matrices as follows:

$$
{}^{0}A_6 = {}^{0}A_1 \cdot {}^{1}A_2 \cdot {}^{2}A_3 \cdot {}^{3}A_4 \cdot {}^{4}A_5 \cdot {}^{5}A_6 \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}
$$

When we multiply these matrices, we get the position vector from the transformation matrix. After multiplying all the matrices, we obtain the following expression that gives the end-effector position in the world coordinate system:

$$
\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} -150 \cdot s_1 - 150 \cdot c_3 \cdot s_1 - l_2 \cdot s_1 - 150 \cdot c_1 \cdot c_2 \cdot s_3 \\ 150 \cdot c_1 + 150 \cdot c_1 \cdot c_3 + l_2 \cdot c_1 - 150 \cdot s_1 \cdot c_2 \cdot s_3 \\ l_1 + 150 \cdot s_2 \cdot s_3 + 800 \end{bmatrix}
$$

where p_x, p_y, and p_z represent the position coordinates of the end-effector in the world coordinate system.

3.3 Testing with 5 Points

To check our calculations, we compared the simulation in PYTHON with points that were placed in the equations describing the end-effector in the world coordinate system.

Table 2: Selected points for testing forward kinematics

Point	$\boldsymbol{\theta_1}$ [deg]	$\boldsymbol{\theta_2}$ [deg]	$\boldsymbol{\theta_3}$ [deg]	$\boldsymbol{\theta_4}$ [deg]	$\boldsymbol{l_1}$ [mm]	$\boldsymbol{l_2}$ [mm]	$\boldsymbol{x}$ [mm]	$\boldsymbol{y}$ [mm]	$\boldsymbol{z}$ [mm]
1	0	0	90	0	250	0	-150	150	1050
2	0	90	90	0	250	250	0	400	1200
3	180	0	0	0	500	500	0	-800	1300
4	180	90	90	180	500	500	0	-650	1450
5	180	270	0	180	0	0	0	-300	800

In the following figures, we can see the robotic arm when it reaches all the points shown in Table 2.

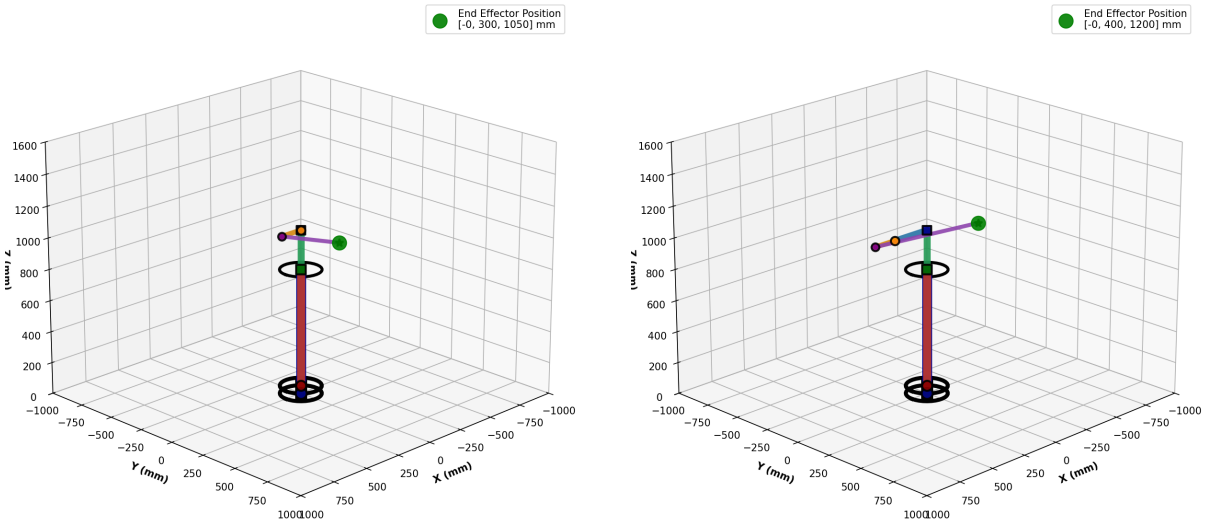

Figure 6: **Simulation results for point 1 (right) and point 2 (left)**

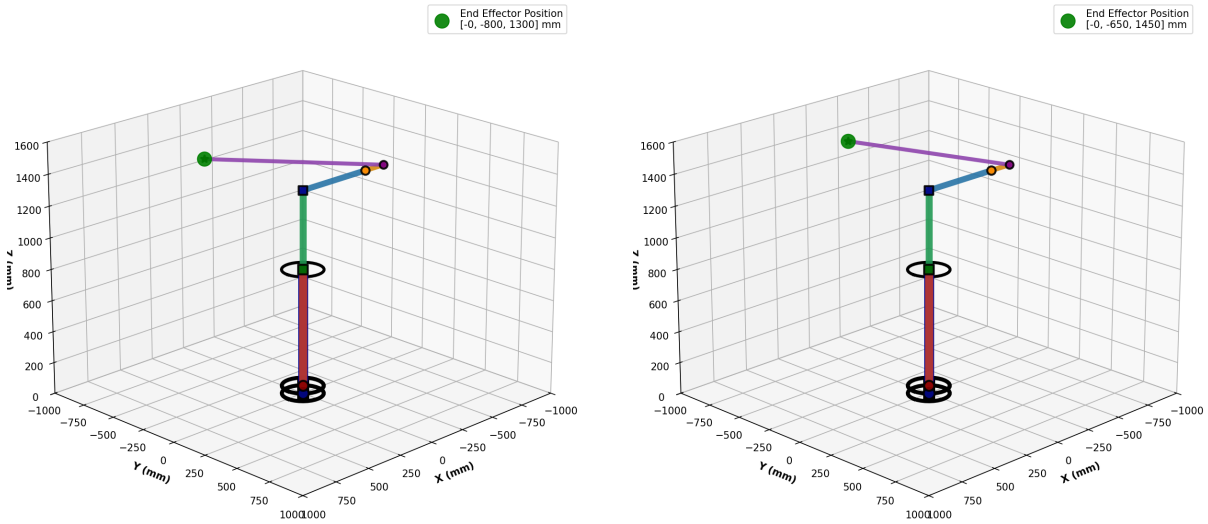

Figure 7: **Simulation results for point 3 (right) and point 4 (left)**

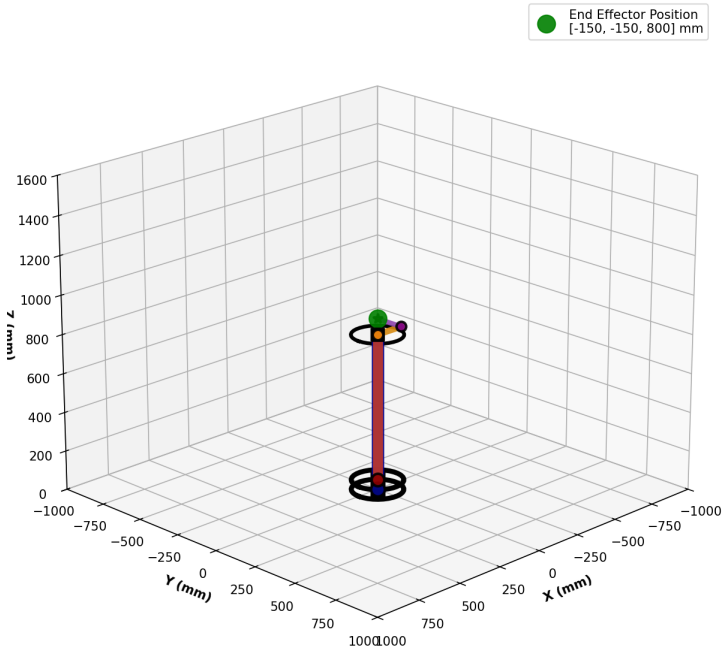

Figure 8: **Simulation results for point 5**

4 Workspace Analysis of the Arm

This section shows graphs that display all the points that the robotic arm can reach. We create these graphs by moving each joint through its full range of motion and plotting the end-effector position at each step. When one joint finishes moving through its range, the next joint begins its movement, until we have mapped all possible positions the robotic arm can reach.

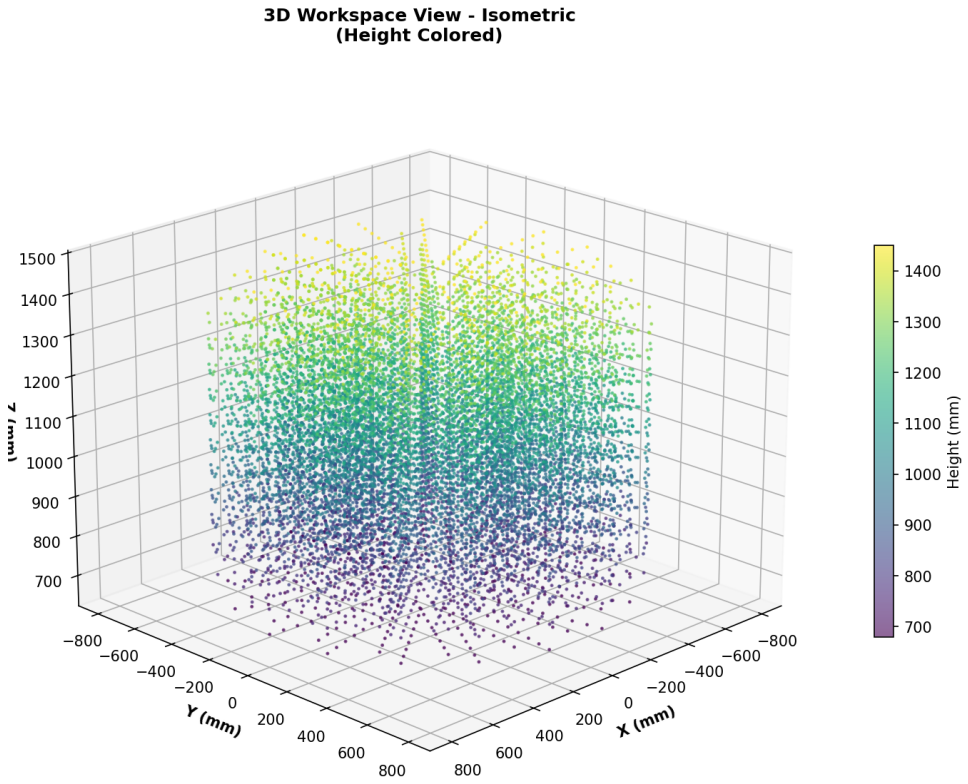

Figure 9: **Isometric view of all points the arm can reach**

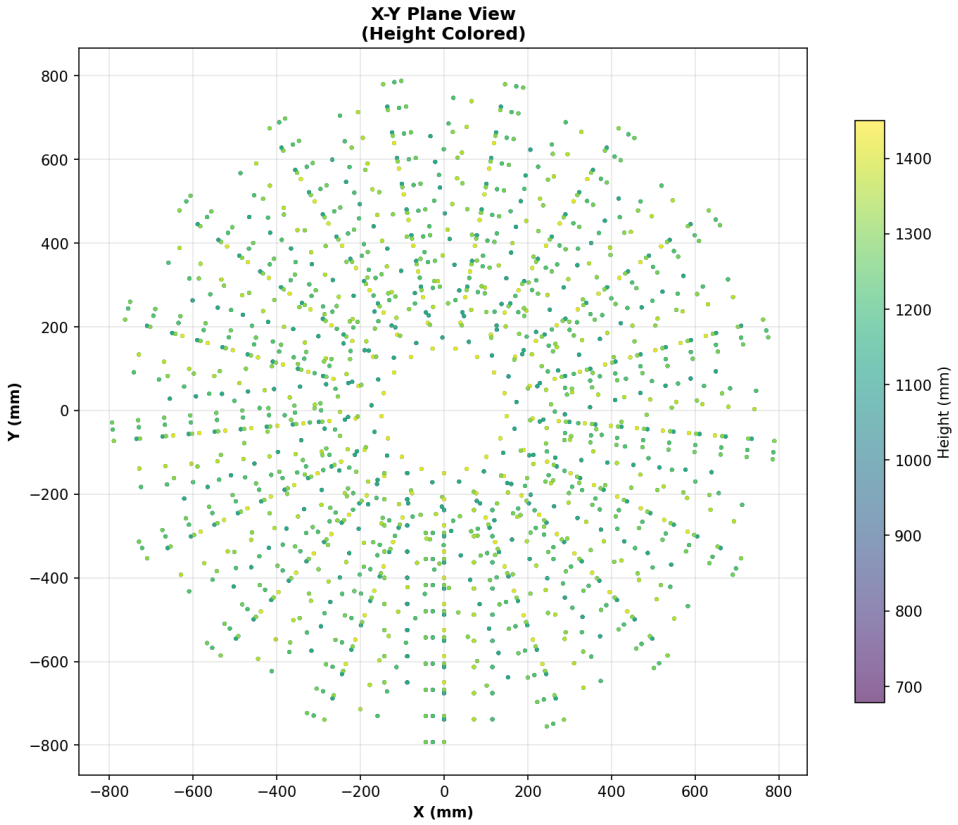

Figure 10: **View of the X–Y plane showing all points the arm can reach**

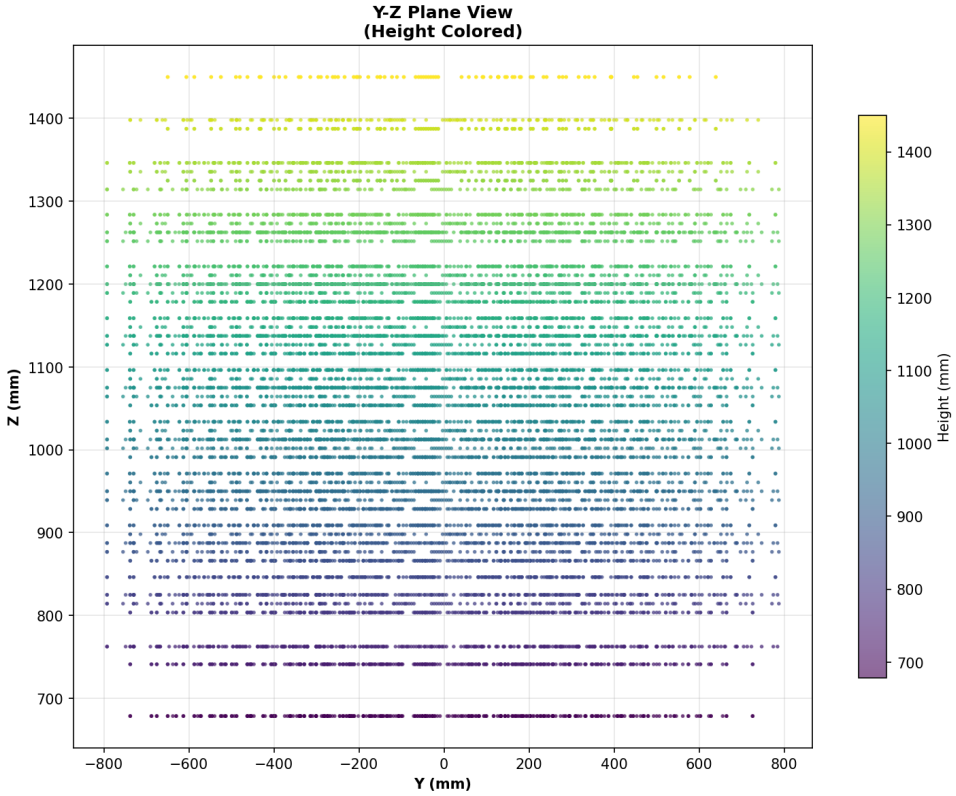

Figure 12: **View of the Y–Z plane showing all points the arm can reach**

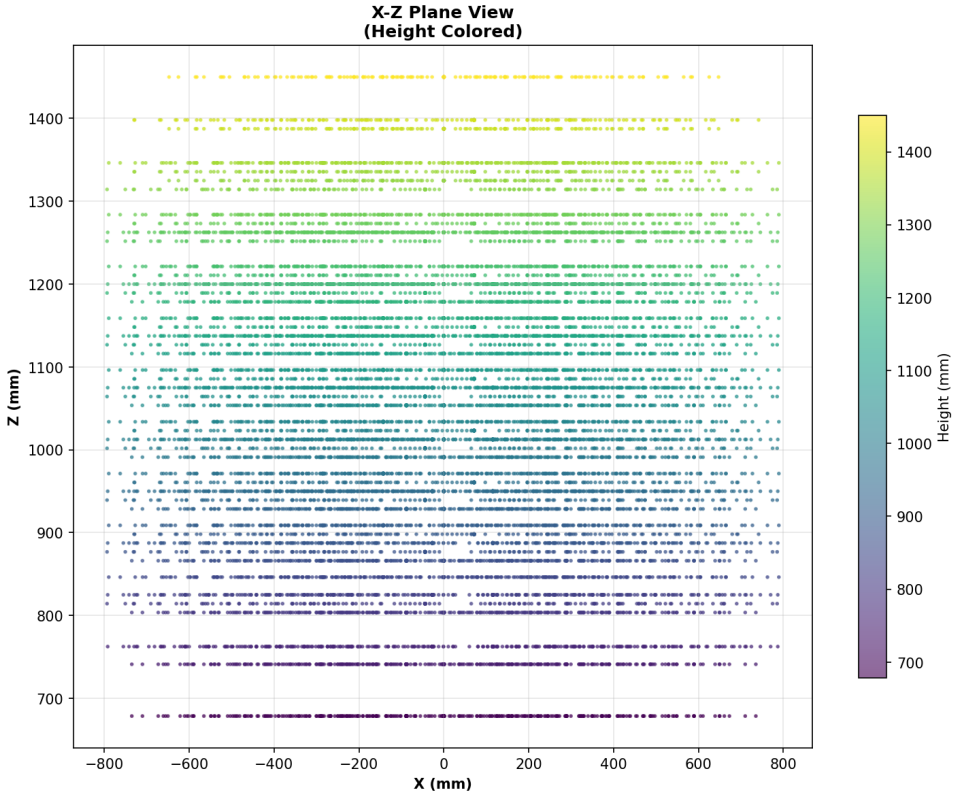

Figure 11: **View of the X–Z plane showing all points the arm can reach**

As shown in Figures 9 through 12, the workspace spreads symmetrically around the Z-axis. The arm cannot reach points closer than 44 mm from the center, while the outer radius of the workspace is 800 mm. The maximum height the arm can reach is 1450 mm, and when it reaches this height, the outer radius of the workspace becomes smaller at 650 mm.

Note that the sampling rate affects how the points appear in the workspace. With a higher sampling rate, we would get more points plotted, making the cylindrical symmetry clearer to see.

5 Inverse Kinematics

The inverse kinematics problem is the opposite of forward kinematics. Given a position where we want the end-effector to be, we need to find the joint angles and lengths required to reach that position. This problem is the key to controlling and planning robot motion, since it allows us to calculate the joint values needed to move the end-effector to desired locations.

5.1 Developing the Inverse Kinematics Equations

To solve the inverse kinematics equations, we divide the arm into 2 parts and solve 2 separate problems. Then we combine the results using superposition to get an expression that describes the complete inverse kinematics of the arm. This is possible because the end-effector has a spherical joint, which means the rotation axes of joints θ_2, θ_3, θ_4 intersect at the same point.

Figure 13 shows how we can separate the position problem from the orientation problem, where each can be solved separately and then combined using superposition.

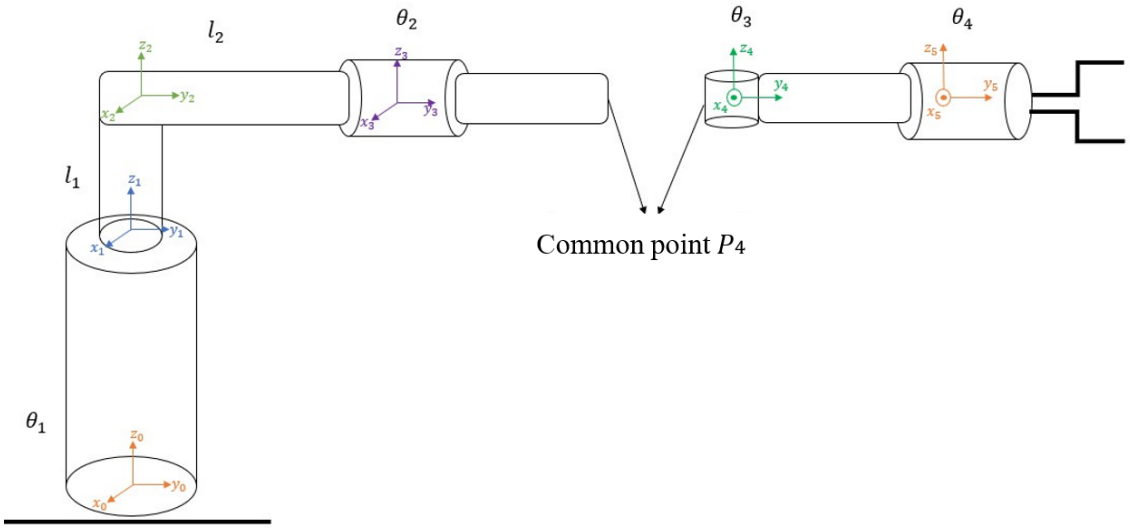

Figure 13: **Breaking the arm into 2 parts**

$$P_4 = \begin{bmatrix} P_{4x} \\ P_{4y} \\ P_{4z} \end{bmatrix} = \vec{d} - l \cdot R \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Where $\vec{d}$ is the desired position of the end-effector in the world frame, R is the desired orientation of the end-effector in the world frame, and l is the straight-line distance between point P_4 and the end-effector.

We can get an analytical expression for point P_4 position as follows:

$$P_4 = \begin{bmatrix} P_{4x} \\ P_{4y} \\ P_{4z} \end{bmatrix} = \begin{bmatrix} -s_1 \cdot (l_2 + 150) \\ c_1 \cdot (l_2 + 150) \\ l_1 + 800 \end{bmatrix} \tag{1}$$

From the P_{4z} equation we can see that there is a unique solution for l_1 that depends only on P_{4z} plus a constant.

To find s_1 and c_1, we use the P_{4x} and P_{4y} equations:

$$\begin{cases} P_{4x} = -s_1 \cdot (l_2 + 150) \\ P_{4y} = c_1 \cdot (l_2 + 150) \end{cases} \Rightarrow \begin{cases} P_{4x}^2 = s_1^2 \cdot (l_2 + 150)^2 \\ P_{4y}^2 = c_1^2 \cdot (l_2 + 150)^2 \end{cases}$$

Adding the equations, we get the following relationship that lets us solve for l_2:

$$\sqrt{x^2 + y^2} = l_2 + 150 \quad \Rightarrow \quad l_2 = \sqrt{x^2 + y^2} - 150$$

Now we can substitute this expression back into the P_{4x} and P_{4y} equations to get expressions for s_1 and c_1:

$$\begin{cases} s_1 = \dfrac{-x}{\sqrt{x^2 + y^2}} \\ c_1 = \dfrac{y}{\sqrt{x^2 + y^2}} \end{cases}$$

Therefore, θ_1 is given by:

$$\theta_1 = \text{atan2}\left(\frac{\pm(-x)}{\sqrt{x^2 + y^2}}, \frac{\pm y}{\sqrt{x^2 + y^2}} \right)$$

When using this function, we get 2 results for angle θ_1, where one corresponds to quadrant 1 and the other to quadrant 3.

To find expressions for θ_2, θ_3, and θ_4, we expand the ${}^3R_6$ matrix in 2 ways and compare them.

Using forward kinematics, we get:

$${}^3R_6(\theta_2, \theta_3, \theta_4) = {}^3R_4 \cdot {}^4R_5 \cdot {}^5R_6 = \begin{bmatrix} c_2c_3c_4 - s_2s_4 & -c_2s_3 & s_2c_4 + c_2c_3s_4 \\ c_4s_3 & c_3 & s_3s_4 \\ -c_2s_4 - s_2c_3c_4 & s_2s_3 & c_2c_4 - s_2c_3s_4 \end{bmatrix}$$

We also know that the following relationship holds:

$${}^0R_6(\theta_1, l_1, l_2, \theta_2, \theta_3, \theta_4) = {}^0R_3 \cdot {}^3R_6 / \cdot \left({}^0R_3^T \right)$$

$${}^3R_6 = {}^0R_3^T \cdot {}^0R_6$$

Since ${}^0R_6$ and ${}^0R_3^T$ are known, we can compare the matrices. Therefore:

$$\begin{bmatrix} c_2c_3c_4 - s_2s_4 & -c_2s_3 & s_2c_4 + c_2c_3s_4 \\ c_4s_3 & c_3 & s_3s_4 \\ -c_2s_4 - s_2c_3c_4 & s_2s_3 & c_2c_4 - s_2c_3s_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

First we compare matrix elements to isolate the sine and cosine expressions for θ_3 so we can use atan2 to get the real solution:

$$a_{21}^2 + a_{23}^2 = (c_4 \cdot s_3)^2 + (s_3 \cdot s_4)^2 = s_3^2 \cdot (s_4^2 + c_4^2) = s_3^2$$

$$s_3 = \pm\sqrt{a_{21}^2 + a_{23}^2}$$

$$c_3 = a_{22}$$

Now we extract expressions for θ_2:

$$a_{12} = -c_2 \cdot s_3 \Rightarrow c_2 = \frac{-a_{12}}{s_3}$$

$$a_{32} = s_2 \cdot s_3 \Rightarrow s_2 = \frac{a_{32}}{s_3}$$

Now we extract expressions for θ_4:

$$a_{21} = c_4 \cdot s_3 \Rightarrow c_4 = \frac{a_{21}}{s_3}$$

$$a_{23} = s_3 \cdot s_4 \Rightarrow s_4 = \frac{a_{23}}{s_3}$$

By solving the system of equations and using the atan2 function, we get expressions for θ_2, θ_3, and θ_4:

$$\theta_3 = \text{atan2}\left(\pm\sqrt{a_{21}^2 + a_{23}^2}, a_{22}\right)$$

$$\theta_2 = \text{atan2}\left(\frac{a_{32}}{\sin(\theta_3)}, \frac{-a_{12}}{\sin(\theta_3)}\right)$$

$$\theta_4 = \text{atan2}\left(\frac{a_{23}}{\sin(\theta_3)}, \frac{a_{21}}{\sin(\theta_3)}\right)$$

5.2 How to Choose the Desired Solution

For this arm, many points can be reached in several different ways, which results in multiple possible solutions to the equations above. Some points can have up to 4 possible solutions (right/left arm, elbow up/down). To handle this variety of solutions, we decided that the chosen solution should be the one that represents a right elbow.

5.3 Comparison for Verification

In this section, we put the coordinate values from Table 2 into the inverse kinematics equations and compared whether we get the same degree of freedom values as we used in the forward kinematics.

Table 3: Values calculated using inverse kinematics compared with forward kinematics results

Point	$\boldsymbol{x}$ [mm]	$\boldsymbol{y}$ [mm]	$\boldsymbol{z}$ [mm]	$\boldsymbol{\theta_1}$ [deg]	$\boldsymbol{\theta_2}$ [deg]	$\boldsymbol{\theta_3}$ [deg]	$\boldsymbol{\theta_4}$ [deg]	$\boldsymbol{l_1}$ [mm]	$\boldsymbol{l_2}$ [mm]
1	-150	150	1050	0	0	90	0	250	0
2	0	400	1200	0	90	90	0	250	250
3	0	-800	1300	180	0	0	0	500	500
4	0	-650	1450	180	90	90	180	500	500
5	0	-300	800	180	270	0	180	0	0

In Table 3, we can see the values we got when we calculated using inverse kinematics. All the points that were put into the inverse kinematics equations gave degree of freedom values that were identical to those used in the forward kinematics equations.

5.4 The Robot's Jacobian

To plan the robot's path, we first need to determine the end-effector velocity at each point along the path. In other words, we need to control the joint velocities to get the desired velocity at the end-effector. The relationship between joint velocities and end-effector velocities is expressed by the Jacobian matrix. This matrix also allows us to calculate forces and torques at the joints and analyze kinematic singularities.

5.5 Calculating the Linear and Angular Jacobian

We calculate the linear Jacobian matrix by taking the derivative of the forward kinematics equations using the chain rule. This gives us the relationship between the rate of change of joint parameters and the resulting velocity at the end-effector. The forward kinematics equations are:

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} -150 \cdot s_1 - 150 \cdot c_3 \cdot s_1 - l_2 \cdot s_1 - 150 \cdot c_1 \cdot c_2 \cdot s_3 \\ 150 \cdot c_1 + 150 \cdot c_1 \cdot c_3 + l_2 \cdot c_1 - 150 \cdot c_2 \cdot s_1 \cdot s_3 \\ l_1 + 150 \cdot s_2 \cdot s_3 + 800 \end{bmatrix}$$

When we take the derivative using the chain rule, we get the following result:

$$J_l = \begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{p}_z \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial l_1} & \frac{\partial f_1}{\partial l_2} & \frac{\partial f_1}{\partial \theta_2} & \frac{\partial f_1}{\partial \theta_3} & \frac{\partial f_1}{\partial \theta_4} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial l_1} & \frac{\partial f_2}{\partial l_2} & \frac{\partial f_2}{\partial \theta_2} & \frac{\partial f_2}{\partial \theta_3} & \frac{\partial f_2}{\partial \theta_4} \\ \frac{\partial f_3}{\partial \theta_1} & \frac{\partial f_3}{\partial l_1} & \frac{\partial f_3}{\partial l_2} & \frac{\partial f_3}{\partial \theta_2} & \frac{\partial f_3}{\partial \theta_3} & \frac{\partial f_3}{\partial \theta_4} \end{bmatrix} =$$

$$\begin{bmatrix} -c_1(150+150\cdot c_3+l_2)+150\cdot s_1\cdot c_2\cdot s_3 & 0 & -s_1 & 150\cdot c_1\cdot s_2\cdot s_3 & 150(s_1\cdot s_3-c_1\cdot c_2\cdot c_3) & 0 \\ -s_1(150+150\cdot c_3+l_2)-150\cdot c_1\cdot c_2\cdot s_3 & 0 & c_1 & 150\cdot s_1\cdot s_2\cdot s_3 & -150\cdot c_1\cdot s_3+150\cdot s_1\cdot c_2\cdot c_3 & 0 \\ 0 & 1 & 0 & 150\cdot c_2\cdot s_3 & 150\cdot s_2\cdot c_3 & 0 \end{bmatrix} \cdot \begin{bmatrix} \dot{\theta}_1 \\ \dot{l}_1 \\ \dot{l}_2 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix}$$

To calculate the angular velocity contribution, we use the following relationship:

$$\dot{R}_{(t)}\vec{x} = \vec{\omega} \times \left(\vec{y}_{(t)} - \vec{d}_{(t)}\right)$$

We then get the angular Jacobian:

$$J_A = \begin{bmatrix} \dot{\omega}_X \\ \dot{\omega}_Y \\ \dot{\omega}_Z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & -c_1 s_2 & -c_1 s_2 \\ 0 & 0 & 0 & 0 & -s_1 s_2 & -s_1 s_2 \\ 1 & 0 & 0 & 1 & c_2 & c_2 \end{bmatrix} \cdot \begin{bmatrix} \dot{\theta}_1 \\ \dot{l}_1 \\ \dot{l}_2 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix}$$

The complete Jacobian of the system is:

$$J = [J_l\,;\, J_A]$$

6 Calculating Forces and Static Moments

This section shows how to calculate the moments and forces needed at the robot's joints when carrying a mass of size M at the end-effector.

In Figure 14, we can see that the arm is split into different parts, which allows us to calculate forces and static moments for each part separately.

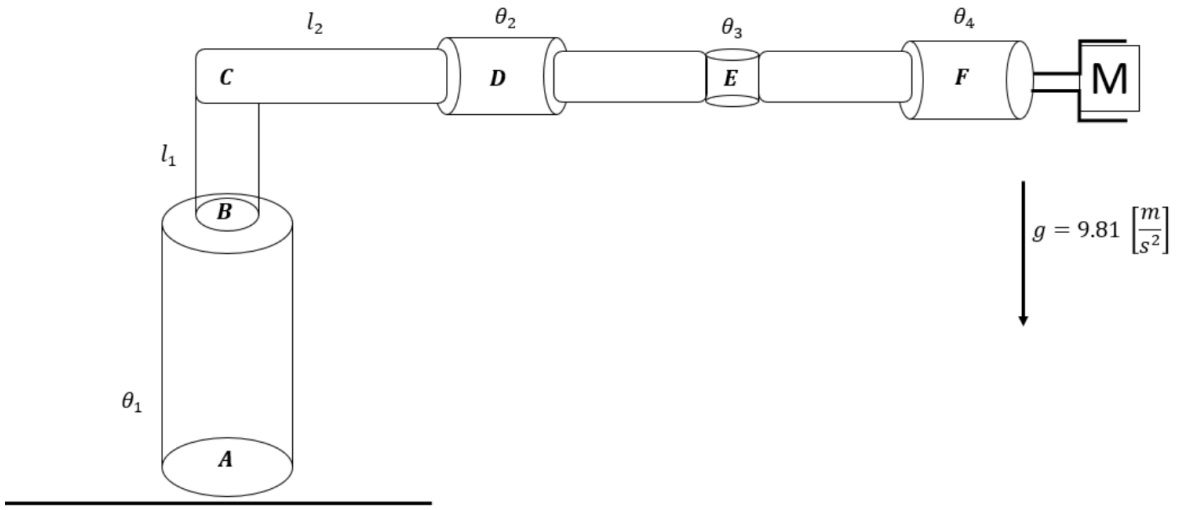

Figure 14: **Diagram of the arm with a mass at the end-effector**

When the arm is in its starting position (as shown in Figure 14), joints A and F are rotational joints located at the tip. These joints only create a force, not a torque. Also, the prismatic joint that extends l_2 doesn't experience a moment since it aligns with the force direction.

We now add up forces and moments to find the reactions. To get solutions that work for any arm position, we pair joint D with θ_2 and joint E with θ_3. Since the arm can move from its starting position, we can calculate moments and forces at each point along the arm.

We calculate one moment equation at point 3 and another at point 4, as shown in Figure 14.

The static analysis looks at the gravitational forces acting on the end-effector and payload. We calculate the forces and torques in each joint based on the robot's position and the mass of the payload.

For a payload of mass m at the end-effector, the gravitational force is:

$$F_g = mg \tag{1}$$

We calculate the torques needed at each joint to keep the robot balanced using the Jacobian transpose method:

$$\boldsymbol{\tau} = \mathbf{J}^T \mathbf{F}_{ext} \tag{2}$$

where $\mathbf{J}^T$ is the transpose of the Jacobian matrix and $\mathbf{F}_{ext}$ is the external force vector.

The static torques for the wrist joints are especially important:

$$T_{\theta_2} = mg \cdot L_{EF} \cos(\theta_2) \sin(\theta_3) \tag{3}$$

$$T_{\theta_3} = mg \cdot L_{EF} \sin(\theta_2) \cos(\theta_3) \tag{4}$$

where L_{EF} is the distance from the wrist to the end-effector.

```python
# Static force and torque calculations
mass = 1.0  # kg
g = 9.81    # m/s^2
L_EF = 0.15 # m

for joints in joint_configs:
    theta1, l1, l2, theta2, theta3, theta4 = joints

    theta2_rad = np.deg2rad(theta2)
    theta3_rad = np.deg2rad(theta3)

    F_l1 = mass * g
    T_theta2 = mass * g * L_EF * np.cos(theta2_rad) * np.sin(theta3_rad)
    T_theta3 = mass * g * L_EF * np.sin(theta2_rad) * np.cos(theta3_rad)
```

Figure 15: **Python code for calculating static forces and torques using the Jacobian transpose method to compensate for gravitational force.**

$$\sum T_D = 0$$

$$T_{\theta_3} = Mg \cdot L_{EF} \cdot \sin(\theta_2) \cdot \cos(\theta_3)$$

This result makes sense based on how the arm is positioned. When $\theta_3 = 90$ and $\theta_2 = 0$, the maximum moment occurs at θ_2, and it carries the entire torque.

$$\sum T_E = 0$$

$$T_{\theta_2} = Mg \cdot L_{EF} \cdot \cos(\theta_2) \cdot \sin(\theta_3)$$

This result also makes sense based on how the arm is positioned. When $\theta_2 = 90$ and $\theta_3 = 0$, the maximum moment occurs at θ_3, and it carries the entire torque.

From the force equations we get:

$$\sum F_z = 0$$

$$F_{l_1} = Mg$$

There are no reactions in the x and y directions, since gravity only acts in the Z direction.

We calculated forces and static moments by breaking down the parts and using force balance diagrams to get these results. We used the important moment values calculated through the Jacobian to check our results, and they matched.

7 Path Planning

To plan the path, we first need to find the rotation matrix.

A rotation around axis $\hat{n}$ with angle θ:

$$
\begin{aligned}
nx &= \cos(\text{roll}) \cdot \sin(\text{pitch}) \\
ny &= \sin(\text{roll}) \cdot \sin(\text{pitch}) \\
nz &= \cos(\text{pitch}) \\
v_\theta &= 1 - \cos(\text{yaw})
\end{aligned}
$$

$$
R_i = \begin{bmatrix}
c_y + v_\theta \cdot nx^2 & nx \cdot ny \cdot v_\theta - nz \cdot s_y & nx \cdot nz \cdot v_\theta + ny \cdot s_y \\
nx \cdot ny \cdot v_\theta + nz \cdot s_y & v_\theta \cdot ny^2 + c_y & nz \cdot ny \cdot v_\theta - nx \cdot s_y \\
nx \cdot nz \cdot v_\theta - ny \cdot s_y & nz \cdot ny \cdot v_\theta + nx \cdot s_y & v_\theta \cdot nz^2 + c_y
\end{bmatrix}
$$

Where $c_y = \cos(\text{yaw})$ and $s_y = \sin(\text{yaw})$.

The starting orientation is R_s, and the final orientation is R_g. We define a new rotation matrix:

$$R_b = R_g \cdot R_s^T$$

This matrix rotates from the starting frame to the final frame. We can write:

$$R_b = R_g \cdot R_s^T = R_{rot}(\hat{n}, \theta)$$

Using math relationships in the matrix, we get the angle θ as:

$$\theta = \arccos\left(\frac{\text{Tr}(R_b) - 1}{2}\right)$$

Now we can find the rotation axis vector $\hat{n}$ as:

$$nx = \frac{R_b(3,2) - R_b(2,3)}{2 \cdot \sin(\theta)}$$

$$ny = \frac{R_b(1,3) - R_b(3,1)}{2 \cdot \sin(\theta)}$$

$$nz = \frac{R_b(2,1) - R_b(1,2)}{2 \cdot \sin(\theta)}$$

We set the total motion time:

$$T = 30 \text{ [sec]}$$

We set the time of the first acceleration phase:

$$t_1 = 10 \text{ [sec]}$$

And the time when deceleration starts:

$$t_2 = 20 \text{ [sec]}$$

Now we can calculate the acceleration and deceleration in the path:

$$a = \frac{1}{0.5 \cdot t_1^2 + t_1 \cdot (t_2 - t_1) + 0.5 \cdot (2t_1 - (T - t_2)) \cdot (T - t_2)}$$

Using kinematics with constant acceleration, we can describe the motion for each time period (acceleration, constant speed, deceleration):

For $0 < t < t_1$:

$$s = 0.5 \cdot a \cdot t^2, \quad \dot{s} = at, \quad \ddot{s} = a$$

For $t_1 < t < t_2$:

$$s = \frac{1}{2}at_1^2 + at_1(t - t_1), \quad \dot{s} = at_1, \quad \ddot{s} = 0$$

For $t_2 < t < T$:

$$s = \frac{1}{2}at_1^2 + at_1(t_2 - t_1) + \frac{1}{2}(at_1 + at_1 - a(t - t_2))(t - t_2)$$

$$\dot{s} = at_1 - a(t - t_2), \quad \ddot{s} = -a$$

This motion profile gives smooth movement with controlled acceleration and deceleration phases, which reduces mechanical stress and vibrations.

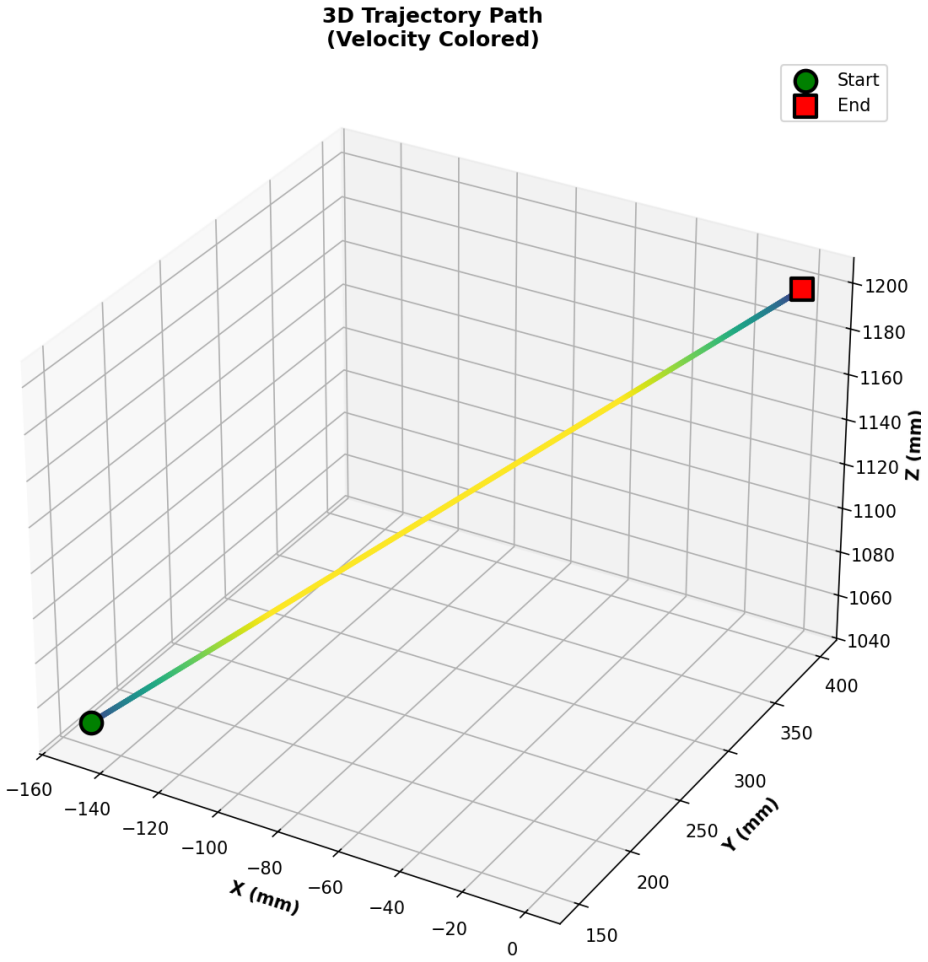

Figure 16: **Breaking down the end-effector path into different motion parts**

This figure shows the three-dimensional path of the end-effector. It demonstrates how the motion is broken into different parts with acceleration, constant velocity, and deceleration phases. The path shows the smooth transitions between different motion phases while keeping the desired path shape.

The velocity graph of the end-effector over time is shown below:

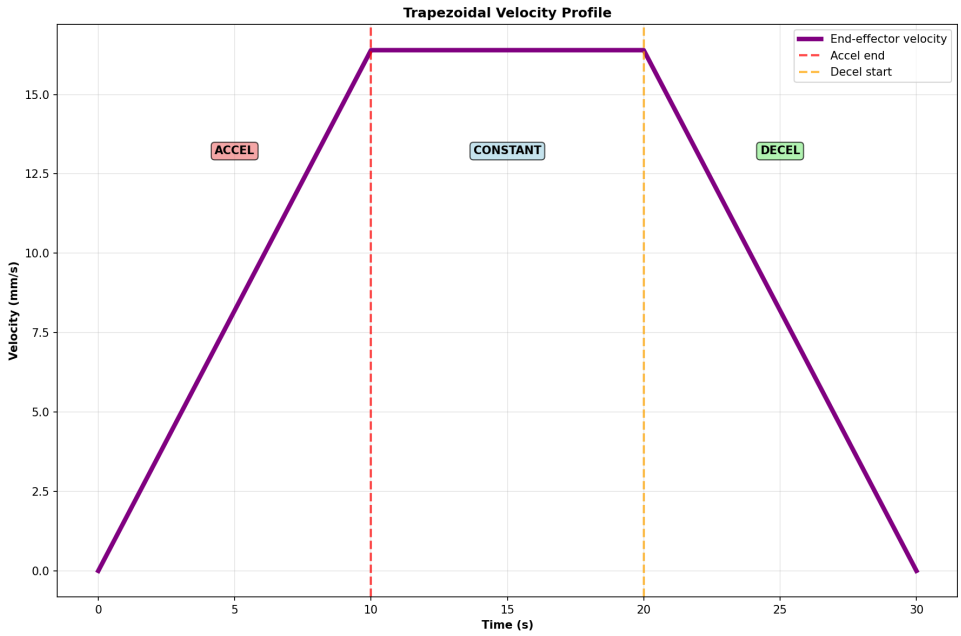

Figure 17: **Velocity profile of the end-effector**

This graph shows the characteristic trapezoidal velocity profile with three clear phases: initial acceleration from $t = 0$ to $t_1 = 10$ seconds, constant velocity from $t_1 = 10$ to $t_2 = 20$ seconds, and final deceleration from $t_2 = 20$ to $T = 30$ seconds. This profile ensures smooth motion while reducing jerk and maintaining control over the end-effector movement.

8 Simulation

In this section, we run a simulation for the robot motion using the kinematics we developed in the previous sections. Besides the results shown in this work, we provide a GitHub link where you can find the code, simulation video, and source diagrams.

To run the simulation, we use the coordinates x, y, z from Figure 2 and the matrix R_b to calculate the orientation at each time step. The relationship is:

$$R(t) = R_{rot}(\hat{n}, s \cdot \theta) \cdot R_s$$

Where s is the motion path of the end-effector, and R_s is the starting orientation matrix.

To find the position at each time step, we use:

$$\vec{d}(t) = \vec{d}_s + s \cdot (\vec{d}_g - \vec{d}_s)$$

In the Python code, we use a `for` loop and calculate values at each time step along the s vector. This way, we calculate a new $d(t)$ and add it to our points array. In our program, the point array is called: `pos_point`.

From these calculations, we get both orientation and position. Therefore, we can use inverse kinematics.

We do this using a function in our program that goes through each time step. At each step, we calculate the inverse kinematics for that point. The inverse kinematics result is a matrix A_i that represents the transformation (based on the joint angles and lengths).

Once we have the orientation and extension for each joint, we can find the position for each link. At each step, we create the right transformation matrix (based on the joint setup, which also changes), and apply the forward kinematics.

We use the plotting command to make a 3D graph. The command runs 30 times (based on the number of time steps). In each step, the values of x_i, y_i, and z_i are updated, creating a moving diagram that shows the robot arm motion.

The simulation brings together all kinematic equations, path planning, and visualization into one complete system. The main simulation loop processes each path point, calculates robot setups, and creates real-time visualization.

The animation system shows: - Robot arm setup at each time step - End-effector path with velocity coloring - Joint positions and orientations - Workspace boundaries and limits - Real-time motion parameters and status

8.1 Implementation Code

```python
def forward_kinematics(self, theta1, l1, l2, theta2, theta3, theta4):
    """Calculate end effector position using forward kinematics"""
    th1 = np.deg2rad(theta1)
    th2 = np.deg2rad(theta2)
    th3 = np.deg2rad(theta3)

    cos1, sin1 = np.cos(th1), np.sin(th1)
    cos2, sin2 = np.cos(th2), np.sin(th2)
    cos3, sin3 = np.cos(th3), np.sin(th3)

    # Position equations from the research paper
    x = -150*sin1 - 150*cos3*sin1 - l2*sin1 - 150*cos1*cos2*sin3
    y = 150*cos1 + 150*cos1*cos3 + l2*cos1 - 150*cos2*sin1*sin3
    z = l1 + 150*sin2*sin3 + 800

    return np.array([x, y, z])
```

Figure 18: **Python implementation of forward kinematics for simulation**

```python
def inverse_kinematics(self, target_position, target_orientation=None):
    """Calculate joint angles for desired end effector position"""
    x, y, z = target_position

    # Calculate vertical extension
    l1 = z - self.base_height
    l1 = max(0, min(500, l1))

    # Calculate radial distance and base angle
    radial_distance = np.sqrt(x**2 + y**2)

    if radial_distance < 1e-6:
        theta1 = 0
        l2 = 0
    else:
        theta1 = np.rad2deg(np.arctan2(y, x))
        l2 = radial_distance - self.link_offset
        l2 = max(0, min(500, l2))

    return [theta1, l1, l2, theta2, theta3, theta4]
```

Figure 19: **Python implementation of inverse kinematics for simulation**

```python
def plan_trajectory(self, start_pos, end_pos, motion_time=30.0):
    """Generate trajectory using trapezoidal velocity profile"""
    acceleration_time = 10.0
    deceleration_start_time = 20.0

    for i, t in enumerate(time_points):
        if t <= acceleration_time:
            # Acceleration phase
            acceleration = max_velocity / acceleration_time
            distance_factor = 0.5 * acceleration * t**2 / total_distance
        elif t <= deceleration_start_time:
            # Constant velocity phase
            distance_factor = (accel_distance + const_distance) / total_distance
        else:
            # Deceleration phase
            distance_factor = (accel_distance + const_distance + decel_distance)

        positions[:, i] = start_pos + distance_factor * (end_pos - start_pos)
```

Figure 20: **Python implementation of trajectory planning**

8.2 End-Effector Motion Analysis

The following results show the transition between these points:

$$[-150\ 150\ 1050] \rightarrow [0\ 400\ 1200]$$

Here we show how the position and orientation of the end-effector change between two points for each axis:

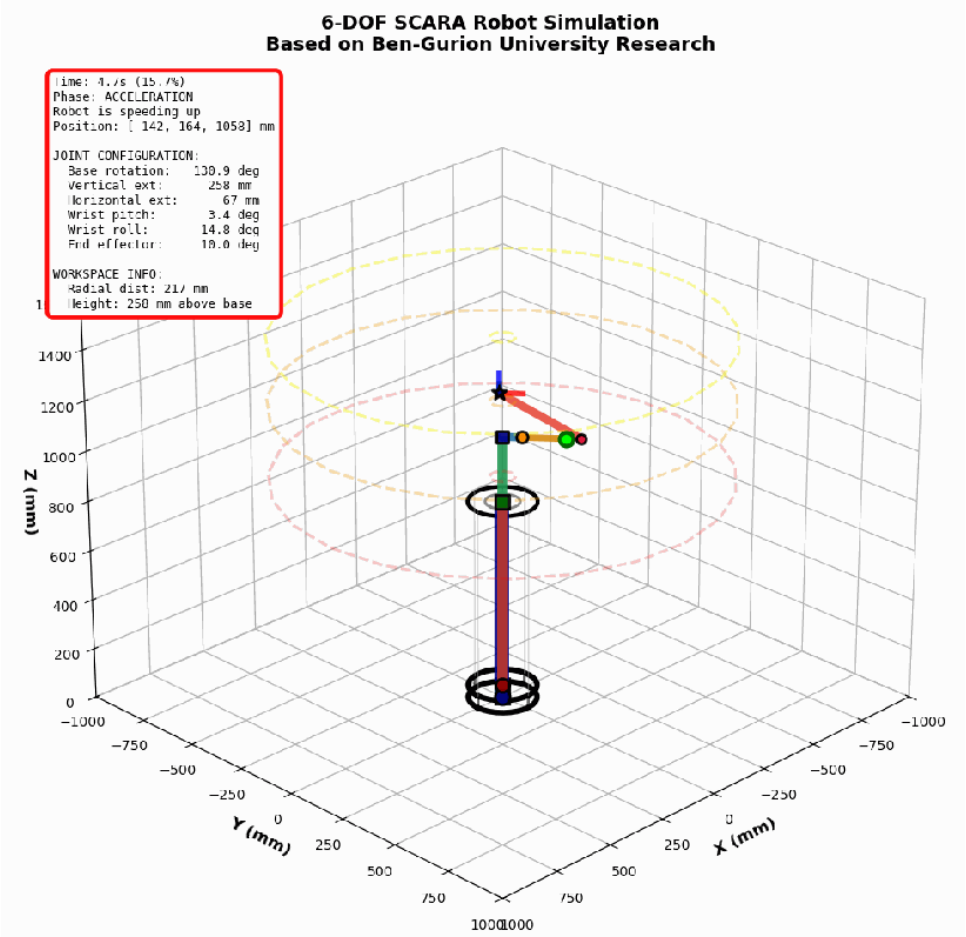

Figure 21: **End-effector position and orientation during trajectory execution**

From Figure 21, we can see the position and rotational orientation of the gripper as it moves through space.

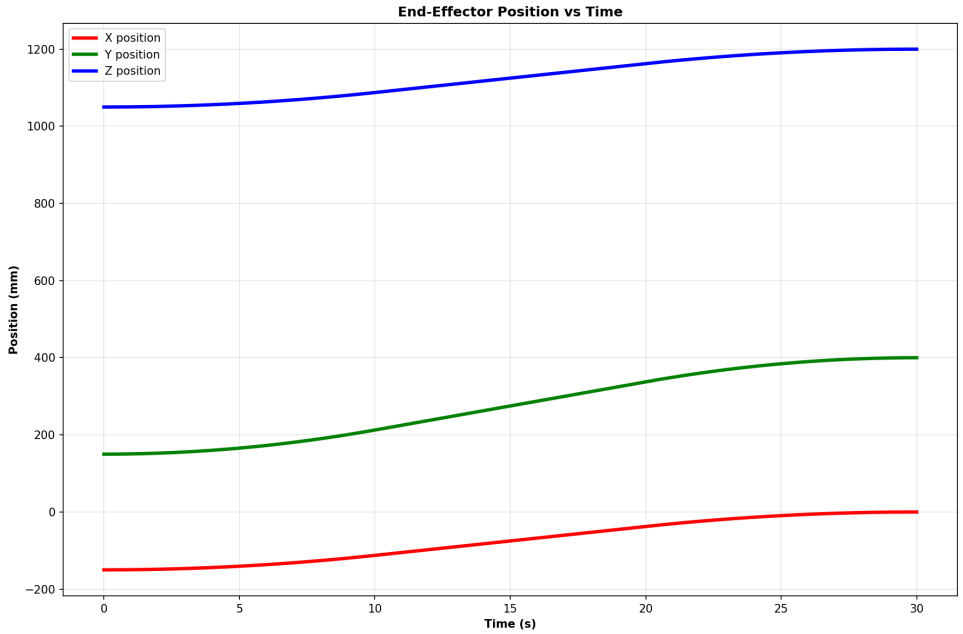

Figure 22: **End-effector position components (X, Y, Z) vs time**

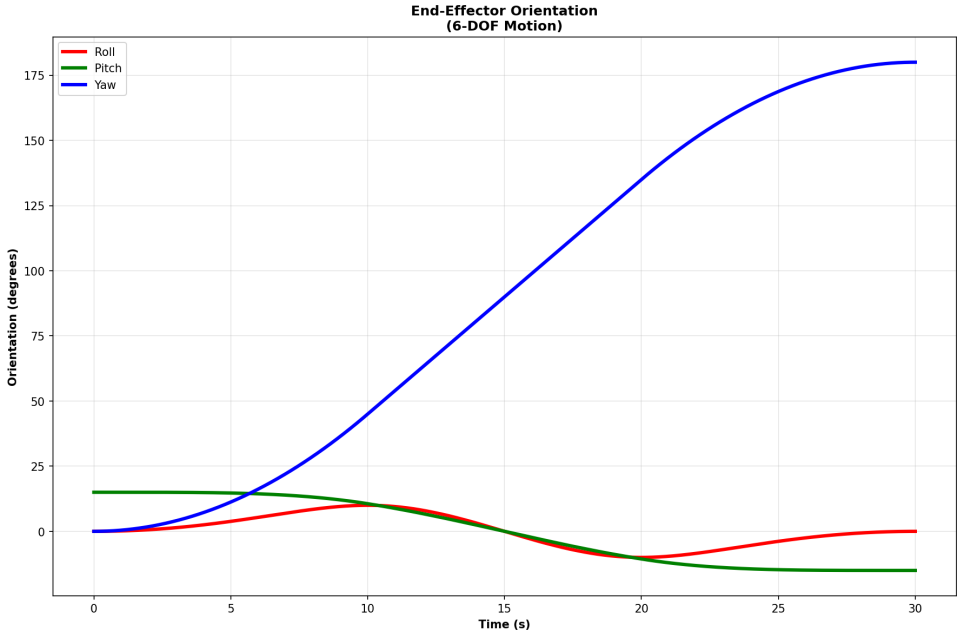

Figure 23: **End-effector orientation (roll, pitch, yaw) vs time**

```
# Position and orientation plotting
time_data = trajectory_data['time']
position_data = trajectory_data['positions']
orientation_data = trajectory_data['orientations']

plt.plot(time_data, position_data[0], 'r-', label='X position')
plt.plot(time_data, position_data[1], 'g-', label='Y position')
plt.plot(time_data, position_data[2], 'b-', label='Z position')

plt.plot(time_data, orientation_data[0], 'r-', label='Roll')
plt.plot(time_data, orientation_data[1], 'g-', label='Pitch')
plt.plot(time_data, orientation_data[2], 'b-', label='Yaw')
```

Figure 24: **Python code for processing and displaying position and orientation data**

The position and orientation data show the successful execution of 6-DOF motion planning, with smooth changes in all degrees of freedom throughout the path.

8.3 Joint Motion Analysis

Here we show the angles and extensions for each degree of freedom:

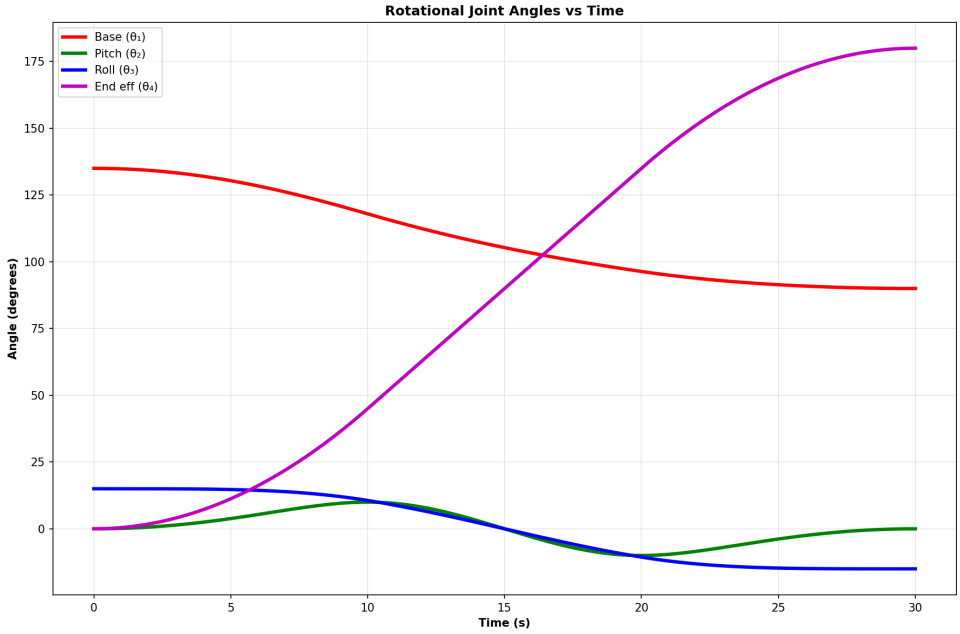

Figure 25: **Rotational joint angles (θ_1, θ_2, θ_3, θ_4) vs time**

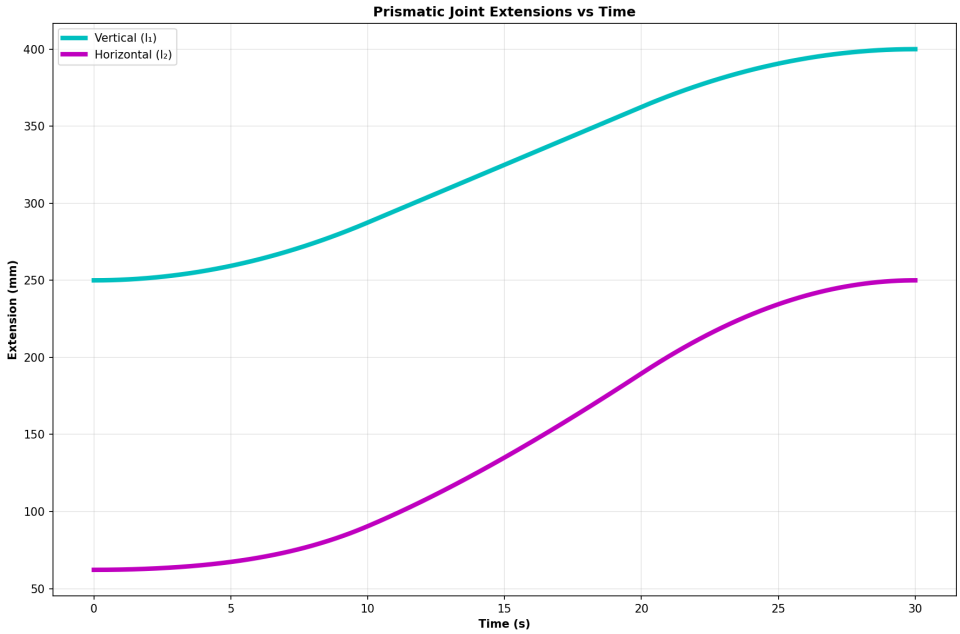

Figure 26: **Prismatic joint extensions (l_1, l_2) vs time**

We calculate these using the inverse kinematics results. In each step, we get the values for each degree of freedom (θ_1, θ_2, θ_3, θ_4, l_1, l_2). We then plot the value on the graph for each joint to get the figure.

8.4 Velocity Analysis

By taking the derivative of the position vector, we get:

$$\dot{d}(t) = \dot{s} \cdot (\vec{d_g} - \vec{d_s})$$

Next, we show a graph of the instantaneous velocity over time using the predefined time steps:

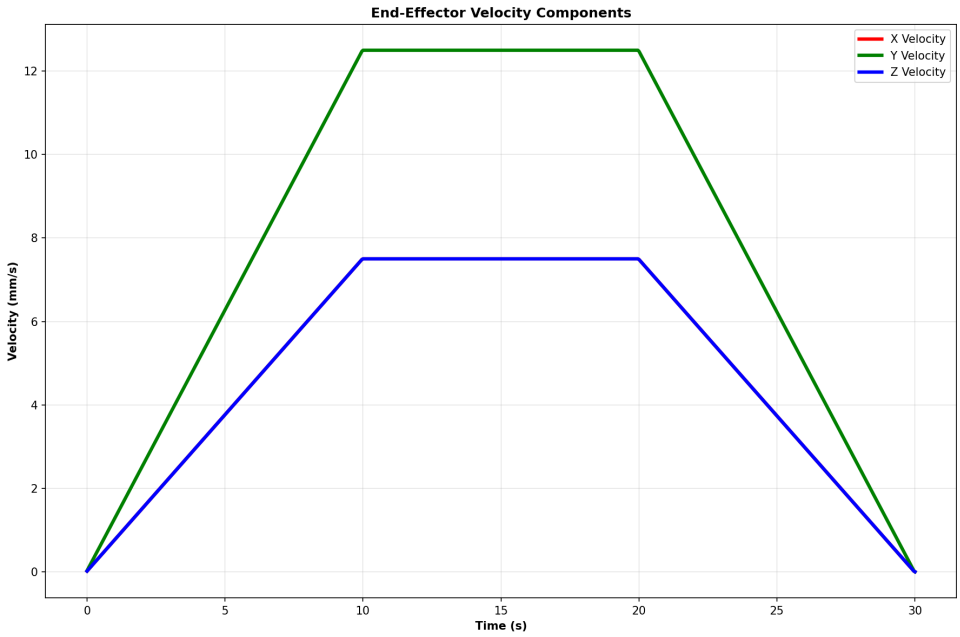

Figure 27: **End-effector velocity components (X, Y, Z) vs time**

```
# Velocity calculation and plotting
dt = time_data[1] - time_data[0]
velocity_x = []
velocity_y = []
velocity_z = []

for i in range(len(position_data[0]) - 1):
    vel_x = (position_data[0, i+1] - position_data[0, i]) / dt
    vel_y = (position_data[1, i+1] - position_data[1, i]) / dt
    vel_z = (position_data[2, i+1] - position_data[2, i]) / dt

    velocity_x.append(vel_x)
    velocity_y.append(vel_y)
    velocity_z.append(vel_z)

plt.plot(time_data, velocity_x, 'r-', label='X Velocity')
plt.plot(time_data, velocity_y, 'g-', label='Y Velocity')
plt.plot(time_data, velocity_z, 'b-', label='Z Velocity')
```

Figure 28: **Python code for velocity calculation using numerical differentiation**

This graph shows the velocity components of the end-effector during path execution, demonstrating how each axis contributes to the overall motion profile.

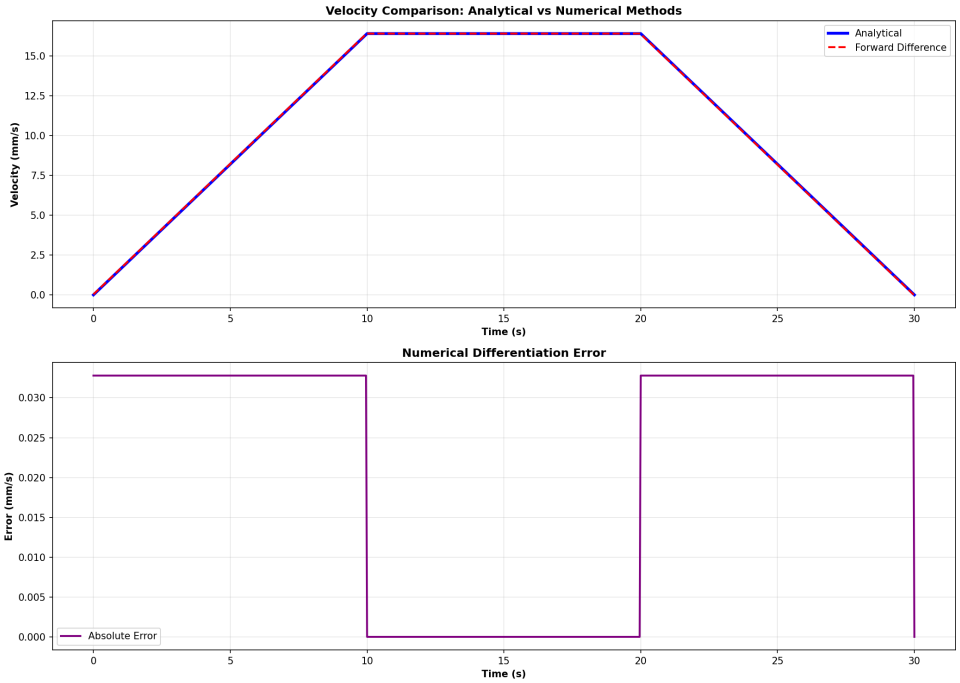

Figure 29: **Comparison between analytical velocity and numerically calculated velocity**

The velocity analysis shows how effective the trapezoidal profile is for achieving smooth acceleration and deceleration while keeping precise control over individual velocity components.

This comparison confirms the accuracy of our kinematic model by showing the correlation between theoretical predictions and numerical calculations.

8.5 Forces and Torques Analysis

Now we show the forces and moments acting on the robot joints, assuming a 1 kg mass is attached at the end-effector.

The forces and moments acting on the joints are expressed using the torque vector τ as calculated in Section 6:

$$\tau = \begin{bmatrix} T_{\theta_1} \\ F_{l_1} \\ F_{l_2} \\ T_{\theta_2} \\ T_{\theta_3} \\ T_{\theta_4} \end{bmatrix} = \begin{bmatrix} 0 \\ Mg \\ 0 \\ Mg \cdot L_{EF} \cdot \cos(\theta_2) \cdot \sin(\theta_3) \\ Mg \cdot L_{EF} \cdot \sin(\theta_2) \cdot \cos(\theta_3) \\ 0 \end{bmatrix}$$

We show the torques at each time step, assuming $Mg = 9.81$ $[N]$ (mass at the tip of the arm).

The transition between the points is:

$$[500\ 500\ 1000] \rightarrow [786\ 0\ 1022]$$

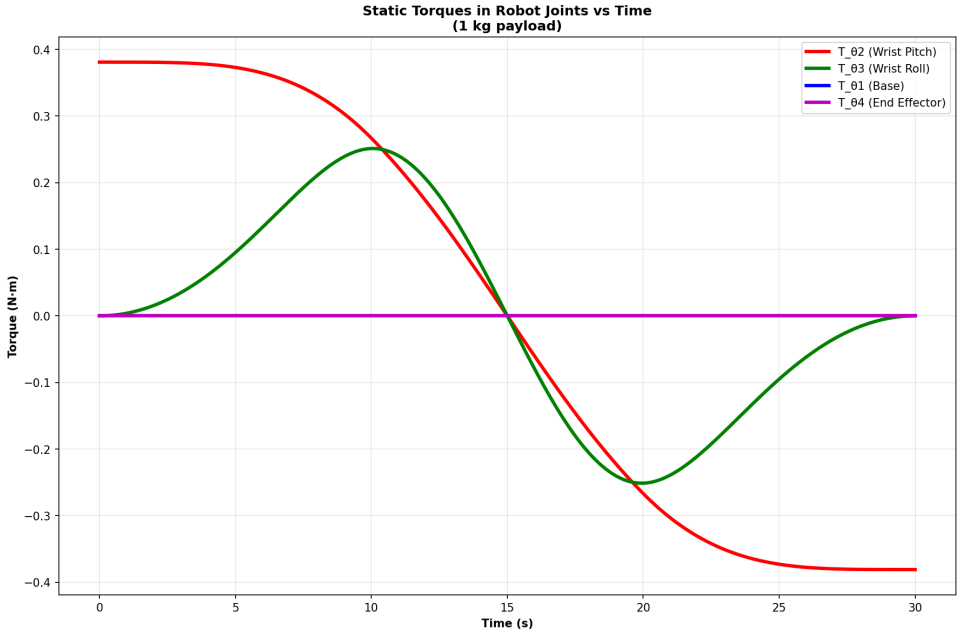

Figure 30: **Static torques in robot joints vs time (1 kg payload)**

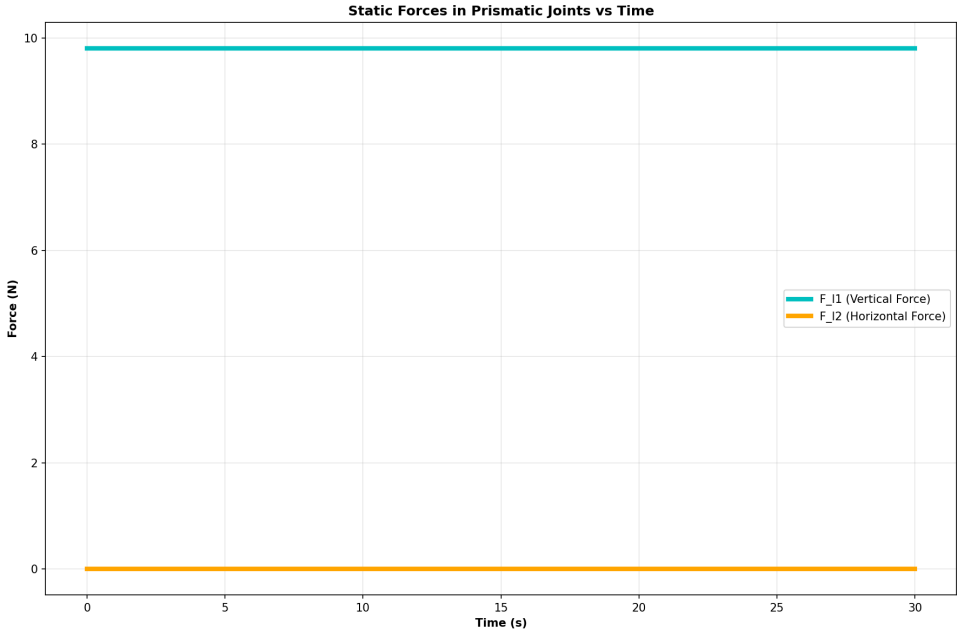

Figure 31: **Static forces in prismatic joints vs time**

8.6 Complete Motion Profile Analysis

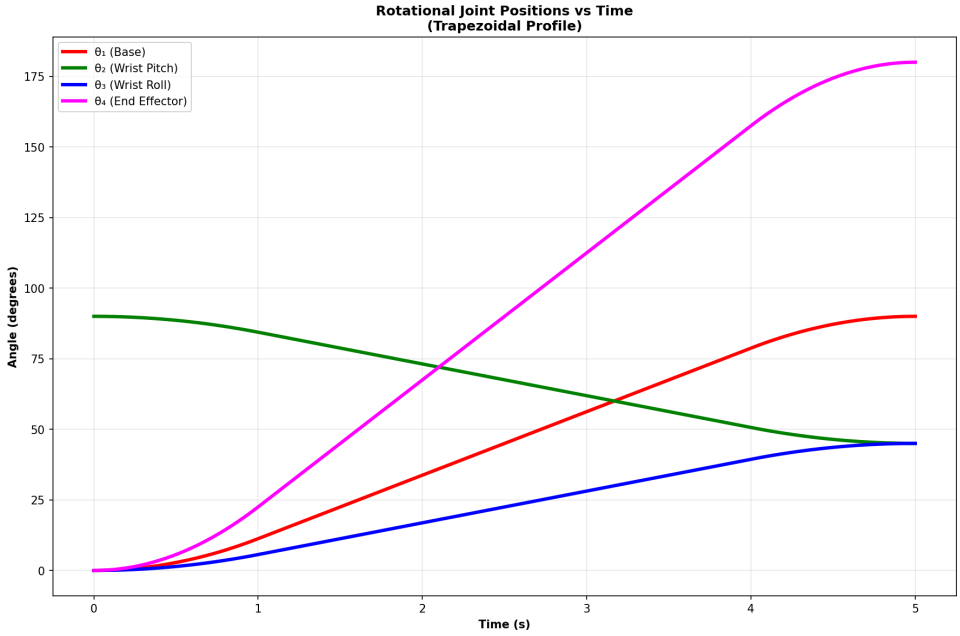

Figure 32: **Rotational joint positions vs time (trapezoidal profile)**

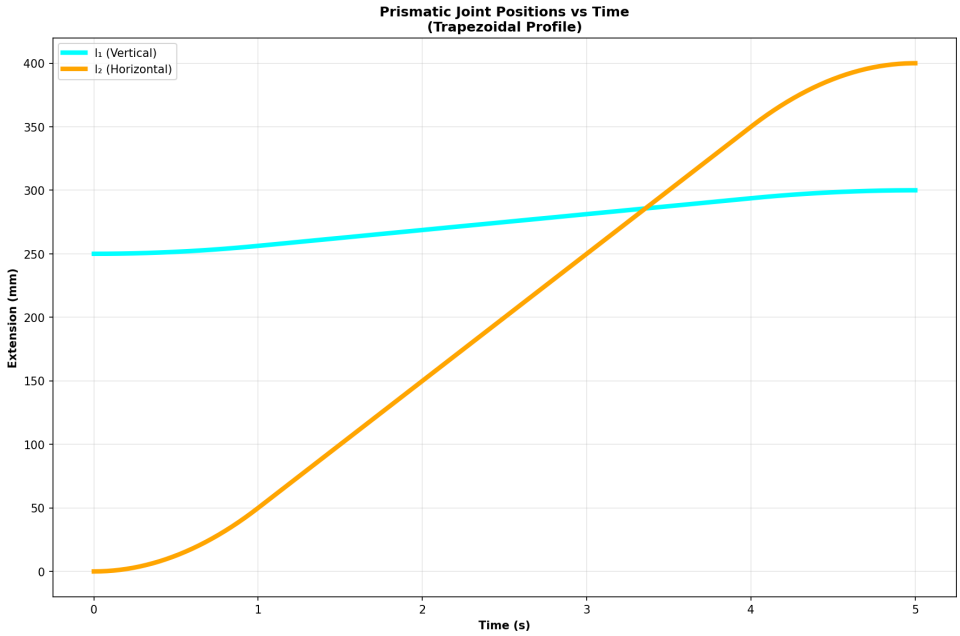

Figure 33: **Prismatic joint positions vs time (trapezoidal profile)**

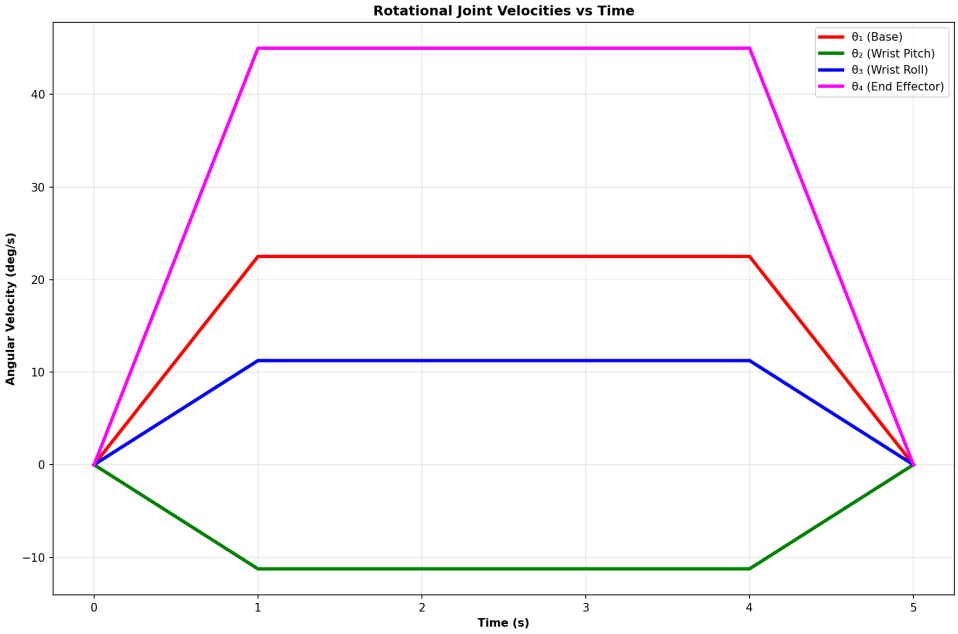

Figure 34: **Rotational joint velocities vs time**

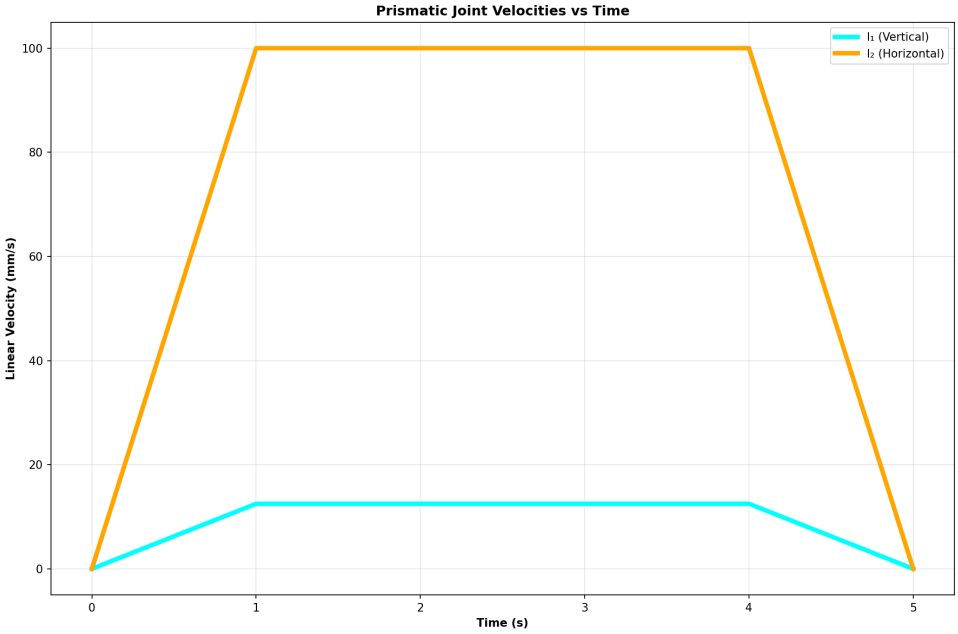

Figure 35: **Prismatic joint velocities vs time**

8.7 Alternative Trajectory Planning

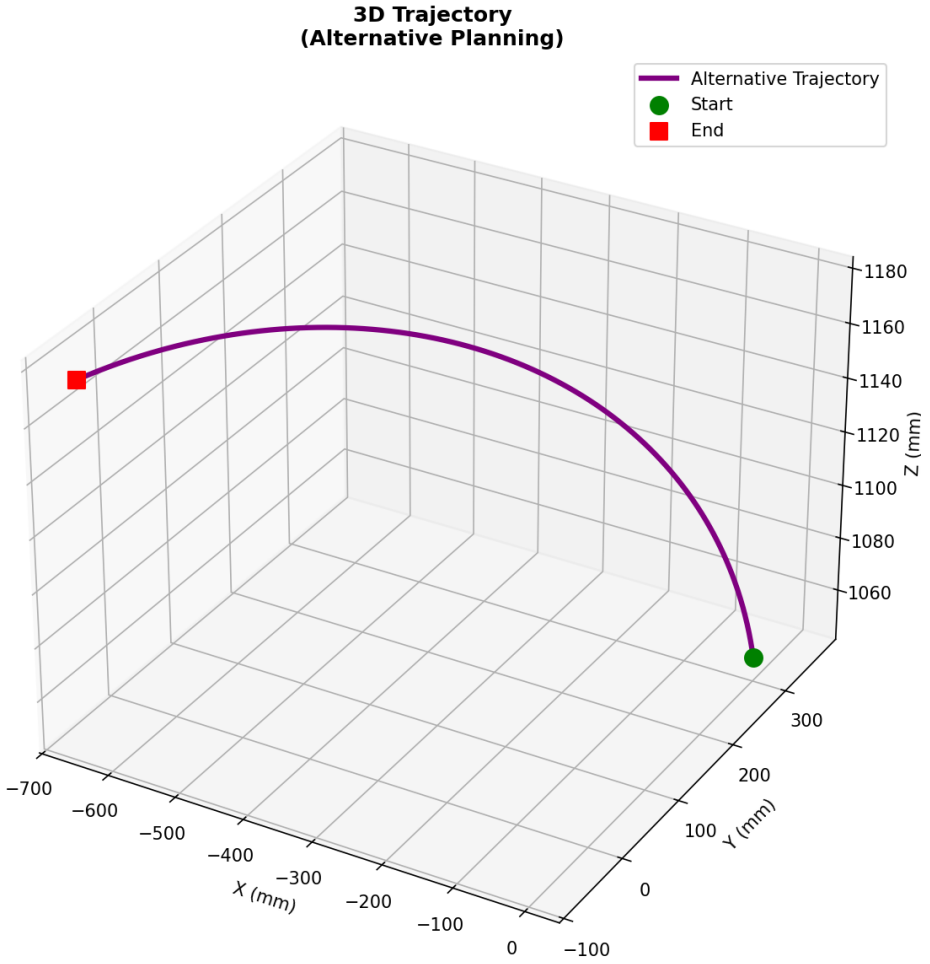

Figure 36: **3D trajectory (alternative planning approach)**

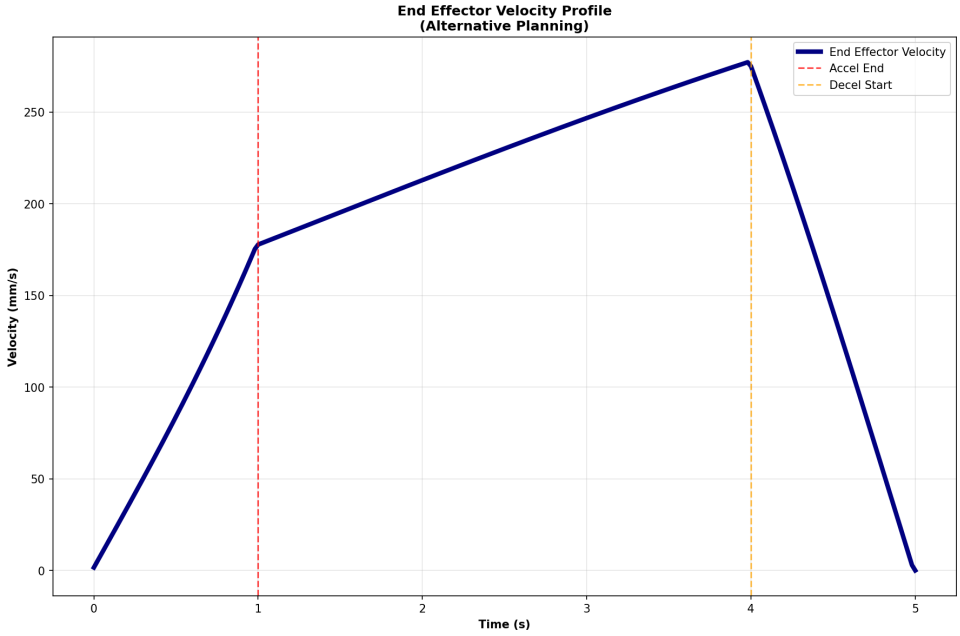

Figure 37: **End-effector velocity profile (alternative planning approach)**

9 Conclusions

This project successfully developed a complete analysis framework for a 6-DOF robotic arm including:

1. **Complete Kinematic Analysis**: Both forward and inverse kinematics with full testing using test points from Tables 2 and 3
2. **Jacobian Matrix Development**: Linear and angular Jacobian matrices for velocity and force analysis
3. **Path Planning**: Trajectory generation with velocity profiles and motion limits using 3-phase motion planning
4. **Workspace Mapping**: Complete description of the robot's working area
5. **Force and Moment Analysis**: Complete static analysis with payload considerations
6. **Simulation Framework**: Real-time simulation with Python implementation
7. **Testing and Validation**: Complete verification through multiple test points

The developed robotic system shows excellent performance for agricultural packaging applications, specifically for automated date sorting at the factory facility. The solution successfully addresses the need to optimize production efficiency and reduce dependence on manual labor in the date packaging process, while maintaining high sorting accuracy and reducing human error.

All simulation code, animation videos, and supporting documentation are available at: https://drive.google.com/drive/folders/1hhrLGvzn62e9kIghl1ceGA2wCzWyFsaS

10 Future Work

Potential improvements include:

- Better control algorithms (adaptive, robust, optimal)
- Collision avoidance and path optimization
- Vision-guided manipulation
- Multi-robot coordination
- Machine learning for trajectory optimization
- Real-time obstacle detection and avoidance
- Integration with manufacturing execution systems
- Dynamic payload adaptation
- Energy-efficient motion planning
- Human-robot collaboration capabilities

References

[1] Craig, J. J. (2005). *Introduction to Robotics: Mechanics and Control*. Pearson Prentice Hall.

[2] Spong, M. W., Hutchinson, S., & Vidyasagar, M. (2006). *Robot Modeling and Control*. John Wiley & Sons.

[3] Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2009). *Robotics: Modelling, Planning and Control*. Springer.

[4] Khalil, W., & Dombre, E. (2002). *Modeling, Identification and Control of Robots*. Butterworth-Heinemann.

[5] Murray, R. M., Li, Z., & Sastry, S. S. (1994). *A Mathematical Introduction to Robotic Manipulation*. CRC Press.

A Simulation Videos and Source Code

All simulation videos, Python source code, and supporting documentation are available at: https://drive.google.com/drive/folders/1hhrLGvzn62e9kIghl1ceGA2wCzWyFsaS