Kinematics of a 3-PUU Delta-Style Parallel Robot

Botao Zhao

June 12, 2025

Contents

1 Introduction

The purpose of this document is to give a quick understand of the kinematics of the delta platfrom of SHER-3.0 (Eye Robor 3.0) to those who are new to this project. It is assumed the reader has already taken the course RDKDC and familiar with some basic robotics and linear algebra knowledge. Some related papers and scripts can be found on this github page: `https://github.com/zhaob5/delta_kinematics`

2 System Overview

The Delta platform is a parallel manipulator known for high speed and accuracy, commonly used in pick-and-place applications. This tutorial focuses on a variation with a 3-PUU configuration, where each leg consists of a vertical **P**ristmatic actuator (active joint), and followed by two passive **U**niversal joints.

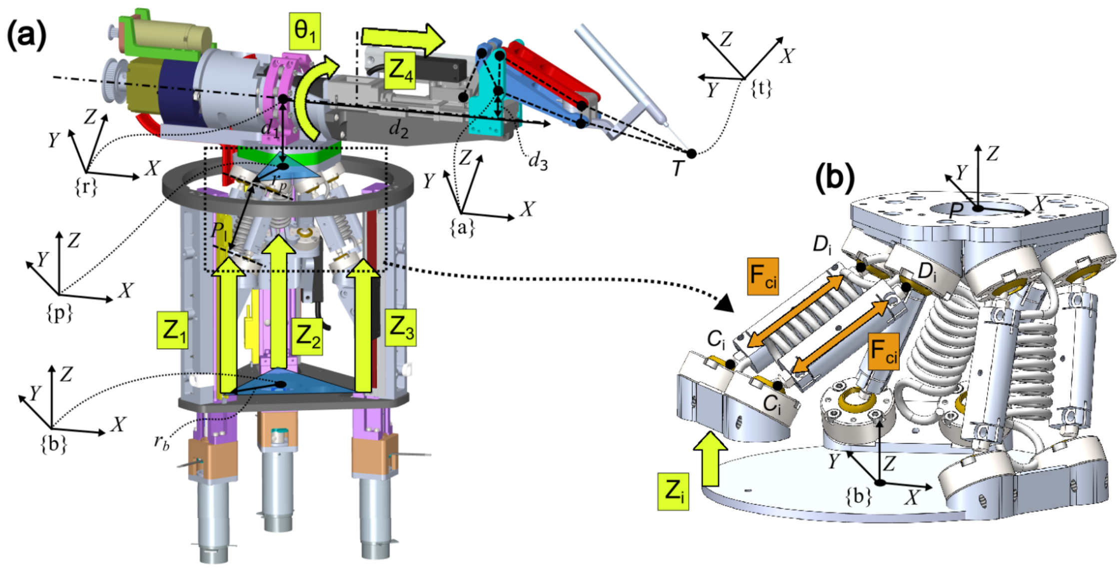

Figure 1: CAD models of (a) SHER-3.0, and (b) close up of the delta platform

We assume three identical legs connecting the base and moving platform, forming a closed kinematic loop.

2.1 Assumptions and Parameters

- Base joint circle radius: r_b
- Platform joint circle radius: r_p
- Angle between prismatic joint to base frame's x-axis: $\theta_{bi} \in \left\{\frac{\pi}{3}, \pi, \frac{5\pi}{3}\right\}$
- Angle between upper universal joint to moving frame's x-axis: $\theta_{pi} \in \left\{\frac{\pi}{3}, \pi, \frac{5\pi}{3}\right\}$
- Link length between universal joints: l
- Vector form of link: $\mathbf{l}_i = [x_{l_i}, y_{l_i}, z_{l_i}]^T$
- Vector form of prismatic joint: $\mathbf{L}_i = [0, 0, q_i]^T$
- End-effector position: $\mathbf{r} = [x, y, z]^T$

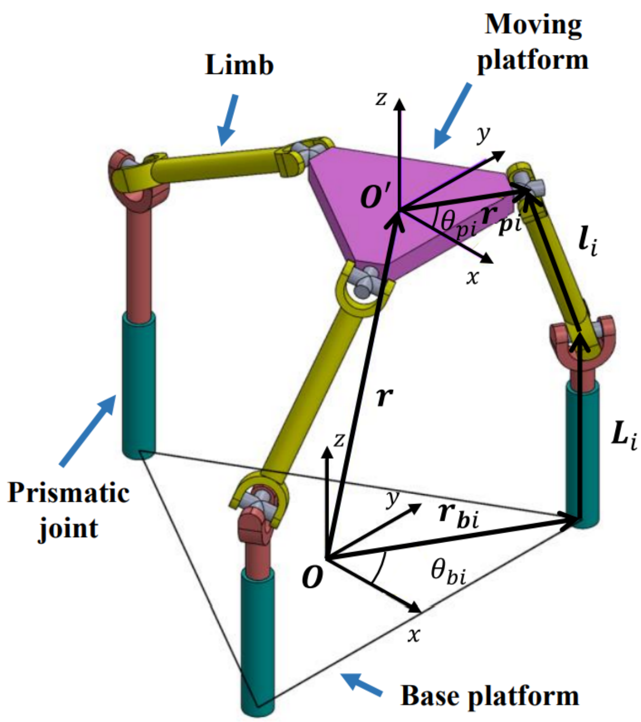

Figure 2: Simplified 3-D view of the delta platform.

3 Inverse Kinematics

For a closed-chain robot, the inverse kinematics is relatively straightforward to derive from its geometry or vector loop equations. Given a desired end-effector position $\mathbf{r} = [x, y, z]^T$, the task is to compute the required lengths of the prismatic actuators q_i for $i = 1, 2, 3$.

3.1 Vector Loop

As shown in Fig.2, the nominal length of the limbs is l_i. The joint length of each prismatic joints is q_i. The $i-$th vector loop closure equation is given by:

$$\mathbf{l}_i = -\mathbf{L}_i - \mathbf{r}_{bi} + \mathbf{r} + \mathbf{r}_{pi} \tag{1}$$

where $\mathbf{L}_i = [0, 0, q_i]^T$, $\mathbf{r}_{bi} = [r_{bi} \cos\theta_{bi}, r_{bi} \cos\theta_{bi}, 0]^T$ and $\mathbf{r}_{pi} = [r_{pi} \cos\theta_{pi}, r_{pi} \cos\theta_{pi}, 0]^T$.

3.2 Inverse Kinematics Equations

Write eq.(1) in matrix form:

$$\begin{bmatrix} x_{l_i} \\ y_{l_i} \\ z_{l_i} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ q_i \end{bmatrix} - \begin{bmatrix} r_b \cos(\theta_i) \\ r_b \sin(\theta_i) \\ 0 \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} r_p \cos(\theta_i) \\ r_p \sin(\theta_i) \\ 0 \end{bmatrix} \tag{2}$$

Since the length of each link is $l = \sqrt{x_{l_i}^2 + y_{l_i}^2 + z_{l_i}^2}$, eq.(2) can be written as:

$$x_{l_i}^2 + y_{l_i}^2 + z_{l_i}^2 = l^2 = (x + a_i)^2 + (y + b_i)^2 + (z - q_i)^2 \tag{3}$$

where a_i and b_i are just two constants: $a_i = -r_b \cos\theta_i + r_p \cos\theta_i$, $b_i = -r_b \sin\theta_i + r_p \sin\theta_i$
Reorganize eq.(3), we can get:

$$q_i = z - \sqrt{l^2 - (x + a_i)^2 - (y + b_i)^2} \tag{4}$$

This gives the prismatic actuator extension for each leg, where $i = 1, 2, 3$.

4 Forward Kinematics

Unlike serial robots (e.g., the UR5), the forward kinematics of a closed-chain robot is typically much harder to compute and can have multiple solutions. For example, if you try to directly substitute the known actuator values q_i back into eq.(3), you'll find that the resulting equations are coupled through x,y,z. The expressions contain quadratic cross-terms, making the system nonlinear and often analytically intractable. Additionally, multiple valid solutions exist, further complicating the problem.

The method described in this document uses some linear algebra tricks to eliminate as many of the quadratic cross-terms as possible. It may not be the most efficient or elegant way to solve forward kinematics for this delta platform. If you have a better idea or approach, feel free to improve this document by adding your method.

4.1 Rearranging the Vector Loop Equation

Since $\mathbf{L}_i$, $\mathbf{r}_{bi}$, $\mathbf{r}_{pi}$ are known, we can define $\mathbf{e}_i = \mathbf{L}_i + \mathbf{r}_{bi} - \mathbf{r}_{pi}$ and rewrite eq.(1) as:

$$\mathbf{l}_i = \mathbf{r} - \mathbf{e}_i \tag{5}$$

Then, dot-multiplying eq.(5) with itself on both side (rememebr $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b}$):

$$\mathbf{l}_i^T\mathbf{l}_i = (\mathbf{r} - \mathbf{e}_i)^T(\mathbf{r} - \mathbf{e}_i) \tag{6}$$

$$l^2 = \mathbf{r}^T\mathbf{r} - 2\mathbf{r}^T\mathbf{e}_i + \mathbf{e}_i^T\mathbf{e}_i \tag{7}$$

From eq.(7), we can get three equations for i =1, 2, 3:

$$l^2 = \mathbf{r}^T\mathbf{r} - 2\mathbf{r}^T\mathbf{e}_1 + \mathbf{e}_1^T\mathbf{e}_1 \tag{8}$$

$$l^2 = \mathbf{r}^T\mathbf{r} - 2\mathbf{r}^T\mathbf{e}_2 + \mathbf{e}_2^T\mathbf{e}_2 \tag{9}$$

$$l^2 = \mathbf{r}^T\mathbf{r} - 2\mathbf{r}^T\mathbf{e}_3 + \mathbf{e}_3^T\mathbf{e}_3 \tag{10}$$

Subtracting eq.(8) with eq.(9) and eq.(10) respectively, we will get:

$$0 = -2\mathbf{r}^T(\mathbf{e}_2 - \mathbf{e}_1) + (\mathbf{e}_2^T\mathbf{e}_2 - \mathbf{e}_1^T\mathbf{e}_1) \tag{11}$$

$$0 = -2\mathbf{r}^T(\mathbf{e}_3 - \mathbf{e}_1) + (\mathbf{e}_3^T\mathbf{e}_3 - \mathbf{e}_1^T\mathbf{e}_1) \tag{12}$$

Let $h_1 = (\mathbf{e}_2^T\mathbf{e}_2 - \mathbf{e}_1^T\mathbf{e}_1)/2$ and $h_2 = (\mathbf{e}_3^T\mathbf{e}_3 - \mathbf{e}_1^T\mathbf{e}_1)/2$, rearranging eq.(11) and eq.(12):

$$\mathbf{r}^T(\mathbf{e}_2 - \mathbf{e}_1) - h_1 = 0 \tag{13}$$

$$\mathbf{r}^T(\mathbf{e}_3 - \mathbf{e}_1) - h_2 = 0 \tag{14}$$

4.2 Forward Kinematics Equations

Since all of the elements in $(\mathbf{e}_2 - \mathbf{e}_1)$ and $(\mathbf{e}_3 - \mathbf{e}_1)$ are constants, let $\mathbf{e}_2 - \mathbf{e}_1 = [x_{21}, y_{21}, z_{21}]^T$ and $\mathbf{e}_3 - \mathbf{e}_1 = [x_{31}, y_{31}, z_{31}]^T$. Then eq.(13) and eq.(14) can be expressed as:

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} x_{21} \\ y_{21} \\ z_{21} \end{bmatrix} - h_1 = 0 \tag{15}$$

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} x_{31} \\ y_{31} \\ z_{31} \end{bmatrix} - h_2 = 0 \tag{16}$$

Then we get two equations with three unknowns x, y, z:

$$x_{21}x + y_{21}y + z_{21}z - h_1 = 0 \tag{17}$$

$$x_{31}x + y_{31}y + z_{31}z - h_2 = 0 \tag{18}$$

Rearranging eq.(17) and eq.(18):

$$x = \frac{(y_{31}z_{21}/y_{21} - z_{31})}{(x_{31} - y_{31}x_{21}/y_{21})}z + \frac{(h_2 - y_{31}h_1/y_{21})}{(x_{31} - y_{31}x_{21}/y_{21})} \tag{19}$$

$$y = \frac{(x_{31}z_{21}/x_{21} - z_{31})}{(y_{31} - x_{31}y_{21}/x_{21})}z + \frac{(h_2 - x_{31}h_1/x_{21})}{(y_{31} - x_{31}y_{21}/x_{21})} \tag{20}$$

Let $k_1 = \frac{(y_{31}z_{21}/y_{21} - z_{31})}{(x_{31} - y_{31}x_{21}/y_{21})}$, $k_2 = \frac{(h_2 - y_{31}h_1/y_{21})}{(x_{31} - y_{31}x_{21}/y_{21})}$, $k_3 = \frac{(x_{31}z_{21}/x_{21} - z_{31})}{(y_{31} - x_{31}y_{21}/x_{21})}$, $k_4 = \frac{(h_2 - x_{31}h_1/x_{21})}{(y_{31} - x_{31}y_{21}/x_{21})}$.

Then substituting eq.(19) and eq.(20) into eq.(8), where $\mathbf{e}_1 = [e_{1x}, e_{1y}, e_{1z}]^T$:

$$l^2 = \begin{bmatrix} k_1z + k_2 & k_3z + k_4 & z \end{bmatrix} \begin{bmatrix} k_1z + k_2 \\ k_3z + k_4 \\ z \end{bmatrix} - 2 \begin{bmatrix} k_1z + k_2 & k_3z + k_4 & z \end{bmatrix} \begin{bmatrix} e_{1x} \\ e_{1y} \\ e_{1z} \end{bmatrix} + \mathbf{e}_1^T\mathbf{e}_1 \tag{21}$$

Rearranging eq.(21):

$$z = \frac{-T_2 \pm \sqrt{T_2^2 - T_1T_3}}{T_1} \tag{22}$$

where $T_1 = k_1^2 + k_3^2 + 1$, $T_2 = k_1k_2 + k_3k_4 - e_{1x}k_1 - e_{1y}k_3 - e_{1z}$,
and $T_3 = k_2^2 + k_4^2 - 2e_{1x}k_2 - 2e_{1y}k_4 - l^2 + e_{1x}^2 + e_{1y}^2 + e_{1z}^2$

Since the delta platform must be above the base plane, when there are two different real solutions, only the one with positive z value is valid.

Thus, eq.(19), eq.(20), and eq.(22) represent the forward kinematic solutions.

5 Delta Platform Jacobian

Let $\dot{\mathbf{r}} = [\dot{x}, \dot{y}, \dot{z}]^T$ be the end-effector (delta platform) velocity, and $\dot{\mathbf{L}}_i = [0, 0, \dot{q}_i]^T$ the rate of change of each prismatic joint.

Rearranging eq.(1) and taking derivative to both side:

$$\dot{\mathbf{L}}_i = \dot{\mathbf{r}} - \dot{\mathbf{l}}_i - \dot{\mathbf{r}}_{bi} + \dot{\mathbf{r}}_{pi} \tag{23}$$

Since all of the elements in $\mathbf{r}_{bi}$ and $\mathbf{r}_{pi}$ are constants, $\dot{\mathbf{r}}_{bi} = \dot{\mathbf{r}}_{pi} = 0$.

Thus,

$$\dot{\mathbf{L}}_i = \dot{\mathbf{r}} - \dot{\mathbf{l}}_i \tag{24}$$

5.1 Time Derivative of Position Vector

Assuming we have a moving vector from point A to point B, $\mathbf{l} = B - A$, as shown below:

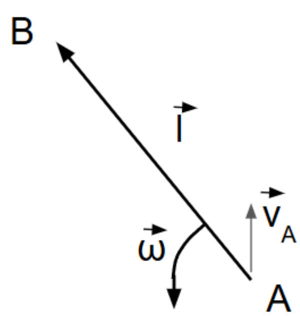

Figure 3: A moving vector $\mathbf{l}$.

Point A is moving at speed $\mathbf{v}_A$ and the vector itself has an angular velocity ω. The length of this vector is unchanged. The velocity of point B is:

$$\mathbf{v}_B = \mathbf{v}_A + \omega \times \mathbf{l}$$

Then:

$$\mathbf{v}_B - \mathbf{v}_A = \omega \times \mathbf{l}$$
$$\dot{B} - \dot{A} = \omega \times \mathbf{l}$$

Thus,

$$\dot{\mathbf{l}} = \omega \times \mathbf{l}$$

The time derivative of this moving vector is the angular velocity cross product itself.

5.2 Jacobian Matrix Derivation

Known the result of $\dot{\mathbf{l}} = \omega \times \mathbf{l}$, we can rewrite eq.(24) as:

$$\dot{\mathbf{L}}_i = \dot{\mathbf{r}} - (\omega_i \times \mathbf{l}_i) \tag{25}$$

Dot-multiplying $\mathbf{l}_i$ to both side:

$$\mathbf{l}_i \cdot \dot{\mathbf{L}}_i = \mathbf{l}_i \cdot \dot{\mathbf{r}} - \mathbf{l}_i \cdot (\omega_i \times \mathbf{l}_i) \tag{26}$$

Since $\mathbf{l}_i$ is perpendicular to $\omega_i \times \mathbf{l}_i$, and $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$:

$$\mathbf{l}_i^T \dot{\mathbf{L}}_i = \mathbf{l}_i^T \dot{\mathbf{r}} \tag{27}$$

$$\begin{bmatrix} x_{l_i} & y_{l_i} & z_{l_i} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{q}_i \end{bmatrix} = \begin{bmatrix} x_{l_i} & y_{l_i} & z_{l_i} \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{28}$$

$$z_{l_i} \dot{q}_i = x_{l_i} \dot{x} + y_{l_i} \dot{y} + z_{l_i} \dot{z} \tag{29}$$

expand eq.(29) with $i = 1, 2, 3$:

$$\begin{bmatrix} z_{l_1} & 0 & 0 \\ 0 & z_{l_2} & 0 \\ 0 & 0 & z_{l_3} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = \begin{bmatrix} x_{l_1} & y_{l_1} & z_{l_1} \\ x_{l_2} & y_{l_2} & z_{l_2} \\ x_{l_3} & y_{l_3} & z_{l_3} \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{30}$$

Thus we can obtain:

$$\dot{\mathbf{r}} = \mathbf{J} \dot{\mathbf{q}} \tag{31}$$

$$\dot{\mathbf{q}} = \mathbf{J}^{-1} \dot{\mathbf{r}} \tag{32}$$

where

$$\mathbf{J} = \begin{bmatrix} x_{l_1} & y_{l_1} & z_{l_1} \\ x_{l_2} & y_{l_2} & z_{l_2} \\ x_{l_3} & y_{l_3} & z_{l_3} \end{bmatrix}^{-1} \begin{bmatrix} z_{l_1} & 0 & 0 \\ 0 & z_{l_2} & 0 \\ 0 & 0 & z_{l_3} \end{bmatrix} \tag{33}$$

is the 3×3 Jacobian matrix of the delta platform.

6 Kinematics Simulation and Visualization

This work is part of my personal project, which is available on `https://github.com/zhaob5/delta_kinematics`. So far, it includes only the inverse and forward kinematics of the simulation. I plan to expand it with Jacobian analysis, and full SHER-3.0 visualization in the future, if time permitting. Feel free to follow the page for updates or contribute if you're interested.

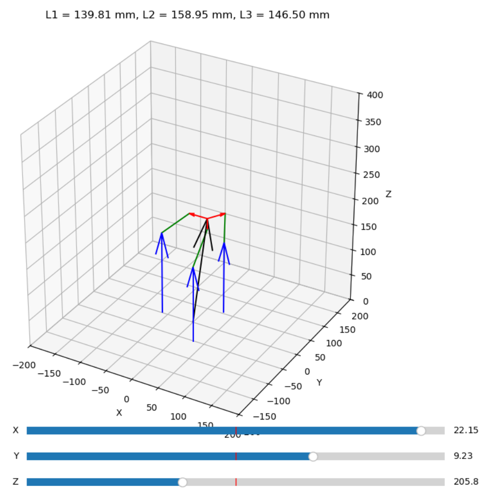

Figure 4: Inverse Kinematics Visualization

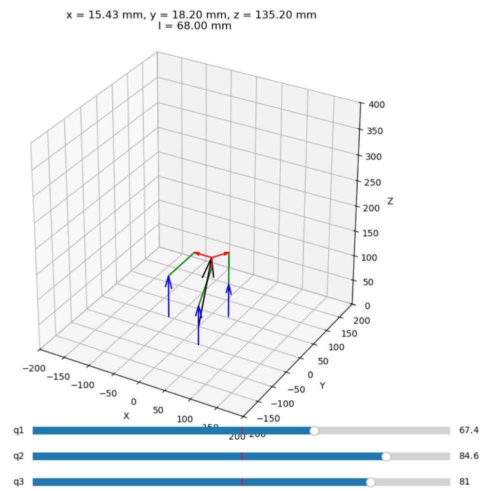

Figure 5: Forward Kinematics Visualization

7 Disclaimer

The calculations and methods presented in this document are based on my personal understanding and interpretation of the kinematics of closed-chain robots. While I've made every effort to ensure accuracy, there may still be errors or oversights. I strongly encourage you to verify the results independently before applying them in any critical context. If you have suggestions, corrections, or would like to discuss the topic further, feel free to reach out to me at: bzhao17@alumni.jh.edu.

8 Appendix: Python Scripts

8.1 Inverse Kinematics Visualization

```python
# # -*- coding: utf-8 -*-
"""
SHER-3.0 Delta Inverse Kinematics Visualization

Created on Wed Jun 11 17:11:23 2025

@author: Botao Zhao
"""
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider
import matplotlib.gridspec as gridspec

# Robot parameters
rp = 34.4773 # mm platform radius
rb = 60.6927 # mm base radius
l = 68  # mm rod length

# Base joint angles
theta1 = np.pi / 3
theta2 = np.pi
theta3 = 5 * np.pi / 3

# Base joint positions (fixed)
rb1 = np.array([np.cos(theta1)*rb, np.sin(theta1)*rb, 0])
rb2 = np.array([np.cos(theta2)*rb, np.sin(theta2)*rb, 0])
rb3 = np.array([np.cos(theta3)*rb, np.sin(theta3)*rb, 0])

# Platform offsets (relative to center)
rp1 = np.array([np.cos(theta1)*rp, np.sin(theta1)*rp, 0])
rp2 = np.array([np.cos(theta2)*rp, np.sin(theta2)*rp, 0])
rp3 = np.array([np.cos(theta3)*rp, np.sin(theta3)*rp, 0])

def calculate_actuator_lengths(x, y, z):
    # Platform joint positions in world frame
    p1 = np.array([x, y, z]) + rp1
    p2 = np.array([x, y, z]) + rp2
    p3 = np.array([x, y, z]) + rp3

    # Vectors from base to platform joints
    l1 = p1 - rb1
    l2 = p2 - rb2
    l3 = p3 - rb3

    # z_i = z - sqrt(l^2 - dx^2 - dy^2)
    def z_offset(vec):
        dx, dy = vec[0], vec[1]
        d_squared = dx**2 + dy**2
        if d_squared > l**2:
            return np.nan
```

```
        return z - np.sqrt(l**2 - d_squared)

    z1 = z_offset(l1)
    z2 = z_offset(l2)
    z3 = z_offset(l3)

    return z1, z2, z3

def update(val):
    x = slider_x.val
    y = slider_y.val
    z = slider_z.val

    z1, z2, z3 = calculate_actuator_lengths(x, y, z)

    ax.cla()

    # Replot everything
    p1 = np.array([x, y, z]) + rp1
    p2 = np.array([x, y, z]) + rp2
    p3 = np.array([x, y, z]) + rp3

    # l1 = [p1[0] - rb1[0], p1[1] - rb1[1], z1]
    # l2 = [p2[0] - rb2[0], p2[1] - rb2[1], z2]
    # l3 = [p3[0] - rb3[0], p3[1] - rb3[1], z3]

    # links
    ax.plot([rb1[0], p1[0]], [rb1[1], p1[1]], [rb1[2] + z1, p1[2]], 'g')
    ax.plot([rb2[0], p2[0]], [rb2[1], p2[1]], [rb2[2] + z2, p2[2]], 'g')
    ax.plot([rb3[0], p3[0]], [rb3[1], p3[1]], [rb3[2] + z3, p3[2]], 'g')

    ax.quiver(rb1[0], rb1[1], rb1[2], 0, 0, z1, color='b')
    ax.quiver(rb2[0], rb2[1], rb2[2], 0, 0, z2, color='b')
    ax.quiver(rb3[0], rb3[1], rb3[2], 0, 0, z3, color='b')

    ax.quiver(0, 0, 0, x, y, z, color='k')

    # Platform vectors
    ax.quiver(x, y, z, rp1[0], rp1[1], rp1[2], color='r')
    ax.quiver(x, y, z, rp2[0], rp2[1], rp2[2], color='r')
    ax.quiver(x, y, z, rp3[0], rp3[1], rp3[2], color='r')

    ax.set_xlim([-200, 200])
    ax.set_ylim([-200, 200])
    ax.set_zlim([0, 400])
    ax.set_box_aspect([1, 1, 1])
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")
    ax.set_title(f"L1 = {z1:.2f} mm, L2 = {z2:.2f} mm, L3 = {z3:.2f} mm")

    fig.canvas.draw_idle()

# Initial pose
```

```python
x0, y0, z0 = 0, 0, 225

# Set up figure with space for sliders
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 1, height_ratios=[6, 1])
ax = fig.add_subplot(gs[0], projection='3d')

slider_ax_x = plt.axes([0.25, 0.15, 0.65, 0.03])
slider_ax_y = plt.axes([0.25, 0.10, 0.65, 0.03])
slider_ax_z = plt.axes([0.25, 0.05, 0.65, 0.03])

slider_x = Slider(slider_ax_x, 'X', -25, 25, valinit=x0)
slider_y = Slider(slider_ax_y, 'Y', -25, 25, valinit=y0)
slider_z = Slider(slider_ax_z, 'Z', 150, 300, valinit=z0)

# Hook update
slider_x.on_changed(update)
slider_y.on_changed(update)
slider_z.on_changed(update)

# Initial draw
update(None)
plt.tight_layout()
plt.show()
```

8.2 Forward Kinematics Visualization

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 11 21:13:48 2025

@author: Botao
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider
import matplotlib.gridspec as gridspec

# Robot parameters
rp = 34.4773 # mm platform radius
rb = 60.6927 # mm base radius
l = 68  # mm rod length

# Base joint angles
theta1 = np.pi / 3
theta2 = np.pi
theta3 = 5 * np.pi / 3

# Base joint positions (fixed)
rb1 = np.array([np.cos(theta1)*rb, np.sin(theta1)*rb, 0])
rb2 = np.array([np.cos(theta2)*rb, np.sin(theta2)*rb, 0])
rb3 = np.array([np.cos(theta3)*rb, np.sin(theta3)*rb, 0])

# Platform offsets (relative to center)
rp1 = np.array([np.cos(theta1)*rp, np.sin(theta1)*rp, 0])
rp2 = np.array([np.cos(theta2)*rp, np.sin(theta2)*rp, 0])
rp3 = np.array([np.cos(theta3)*rp, np.sin(theta3)*rp, 0])

def update(val):
    q1 = slider_q1.val
    q2 = slider_q2.val
    q3 = slider_q3.val

    L1 = np.array([0,0,q1])
    L2 = np.array([0,0,q2])
    L3 = np.array([0,0,q3])

    e1 = L1 + rb1 - rp1
    e2 = L2 + rb2 - rp2
    e3 = L3 + rb3 - rp3

    h1 = ((e2[0]**2 + e2[1]**2 + e2[2]**2) - (e1[0]**2 + e1[1]**2 + e1[2]**2))/2
    h2 = ((e3[0]**2 + e3[1]**2 + e3[2]**2) - (e1[0]**2 + e1[1]**2 + e1[2]**2))/2

    x21 = (e2 - e1)[0]
    y21 = (e2 - e1)[1]
    z21 = (e2 - e1)[2]

```

```
    x31 = (e3 - e1)[0]
    y31 = (e3 - e1)[1]
    z31 = (e3 - e1)[2]

    k1 = (y31*z21/y21 - z31)/(x31 - y31*x21/y21)
    k2 = (h2 - y31*h1/y21)/(x31 - y31*x21/y21)
    k3 = (x31*z21/x21 - z31)/(y31 - x31*y21/x21)
    k4 = (h2 - x31*h1/x21)/(y31 - x31*y21/x21)

    T1 = k1**2 + k3**2 + 1
    T2 = k1*k2 + k3*k4 - e1[0]*k1 - e1[1]*k3 - e1[2]
    T3 = k2**2 + k4**2 -2*e1[0]*k2 - 2*e1[1]*k4 - l**2 + e1[0]**2 + e1[1]**2 + e1[2]**2

    z = (-T2 + np.sqrt(T2**2 - T1*T3))/T1
    x = k1*z + k2
    y = k3*z + k4

    l1 = np.array([x, y, z]) - e1

    ax.cla()

    # Replot everything
    p1 = np.array([x, y, z]) + rp1
    p2 = np.array([x, y, z]) + rp2
    p3 = np.array([x, y, z]) + rp3

    # l1 = [p1[0] - rb1[0], p1[1] - rb1[1], z1]
    # l2 = [p2[0] - rb2[0], p2[1] - rb2[1], z2]
    # l3 = [p3[0] - rb3[0], p3[1] - rb3[1], z3]

    # links
    ax.plot([rb1[0], p1[0]], [rb1[1], p1[1]], [rb1[2] + q1, p1[2]], 'g')
    ax.plot([rb2[0], p2[0]], [rb2[1], p2[1]], [rb2[2] + q2, p2[2]], 'g')
    ax.plot([rb3[0], p3[0]], [rb3[1], p3[1]], [rb3[2] + q3, p3[2]], 'g')

    ax.quiver(rb1[0], rb1[1], rb1[2], 0, 0, q1, color='b')
    ax.quiver(rb2[0], rb2[1], rb2[2], 0, 0, q2, color='b')
    ax.quiver(rb3[0], rb3[1], rb3[2], 0, 0, q3, color='b')

    ax.quiver(0, 0, 0, x, y, z, color='k')

    # Platform vectors
    ax.quiver(x, y, z, rp1[0], rp1[1], rp1[2], color='r')
    ax.quiver(x, y, z, rp2[0], rp2[1], rp2[2], color='r')
    ax.quiver(x, y, z, rp3[0], rp3[1], rp3[2], color='r')

    ax.set_xlim([-200, 200])
    ax.set_ylim([-200, 200])
    ax.set_zlim([0, 400])
    ax.set_box_aspect([1, 1, 1])
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")
    ax.set_title(f"x = {x:.2f} mm, y = {y:.2f} mm, z = {z:.2f} mm \n l = {np.linalg.norm(l1):.2f} mm")
```

```

    fig.canvas.draw_idle()

# Initial pose
q10, q20, q30 = 50, 50, 50

# Set up figure with space for sliders
fig = plt.figure(figsize=(10, 8))
gs = gridspec.GridSpec(2, 1, height_ratios=[6, 1])
ax = fig.add_subplot(gs[0], projection='3d')

slider_ax_q1 = plt.axes([0.25, 0.15, 0.65, 0.03])
slider_ax_q2 = plt.axes([0.25, 0.10, 0.65, 0.03])
slider_ax_q3 = plt.axes([0.25, 0.05, 0.65, 0.03])

slider_q1 = Slider(slider_ax_q1, 'q1', 0, 100, valinit=q10)
slider_q2 = Slider(slider_ax_q2, 'q2', 0, 100, valinit=q20)
slider_q3 = Slider(slider_ax_q3, 'q3', 0, 100, valinit=q30)

# Hook update
slider_q1.on_changed(update)
slider_q2.on_changed(update)
slider_q3.on_changed(update)

# Initial draw
update(None)
plt.tight_layout()
plt.show()
```