Kinematics and Jacobian Analysis of SHER-3.0

Botao Zhao

July 27, 2025

Contents

1 Introduction

The purpose of this document is to show detailed derivation of Jacobian matrix and Jacobian pseudo-inverse of SHER-3.0 (Eye Robot 3.0). Two separate documents covering the kinematic derivation of the delta platform and the roll-tilt mechanism are available at `https://github.com/zhaob5/delta_kinematics` and `https://github.com/zhaob5/roll_tilt_mechanism`. If you haven't read them yet, **I strongly encourage you to review those two documents**, as a lot explanations here build upon concepts introduced there. Some related papers and scripts can be found on this github page: `https://github.com/zhaob5/Kinematics-and-Jacobian-Analysis-of-SHER-3.0`

2 System Overview

The base of SHER-3.0 is a delta mechanism, which has been described in detail in my previous work. Mounted on top of the delta stage is an arm with roll and tilt functionality. The tilt mechanism was designed with a four-bar linkage configuration, as illustrated in Fig.(1) and Fig.(2).

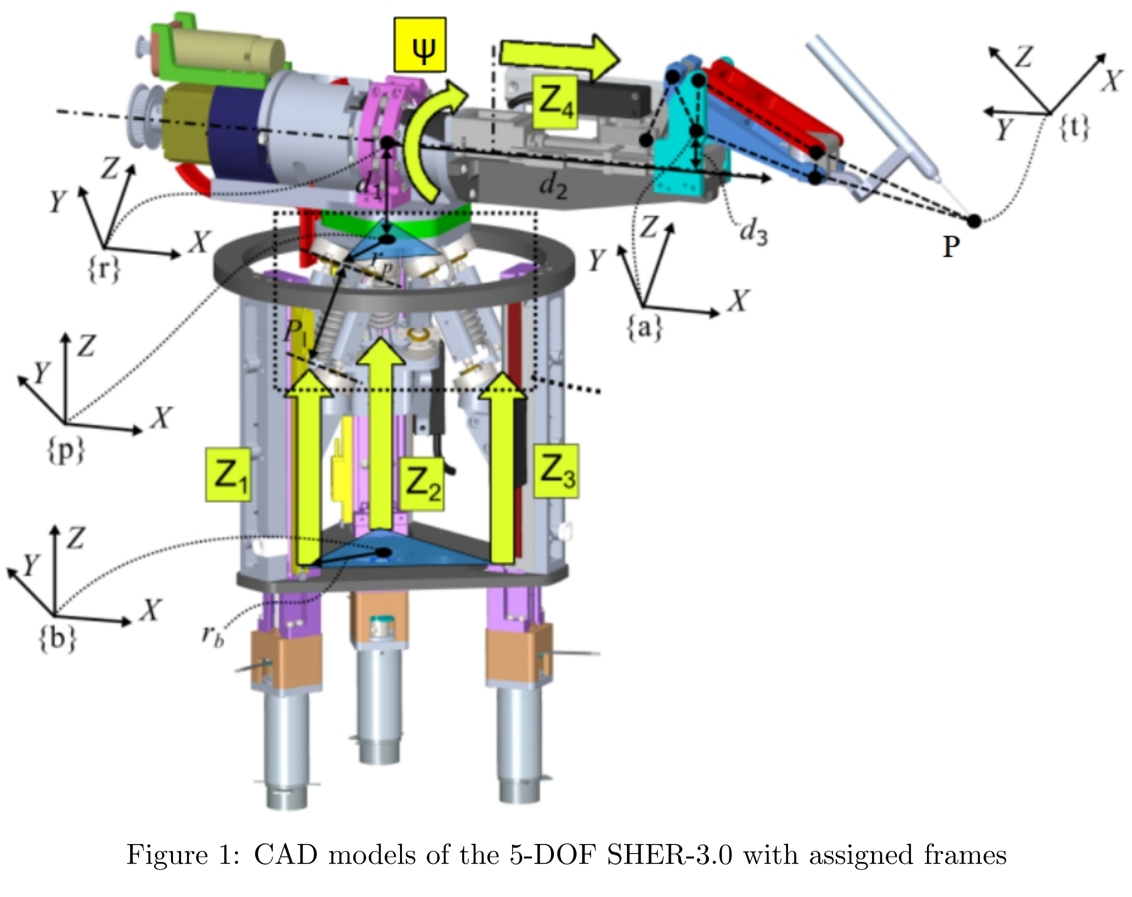

Figure 1: CAD models of the 5-DOF SHER-3.0 with assigned frames

3 Jacobian Matrix of SHER-3.0

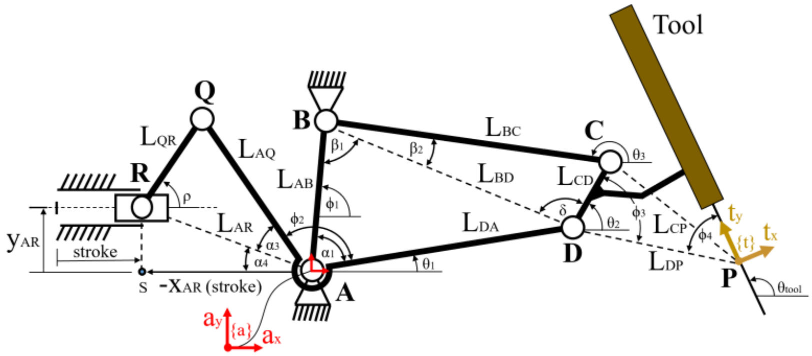

Figure 2: Side view of the tilt mechanism

3.1 Homogeneous Transformation from Base Frame to Tool Frame

Note that the coordinate frames $\{a\}$ and $\{t\}$ shown in Fig.1 and Fig.2 have different orientations. In this document, I will use the frame definition shown in Fig.2 to maintain consistency with my previous documentation. If a different frame is chosen, the resulting expressions will differ accordingly.

Since the delta platform provides only translational motion along x, y, z, its homogeneous transformation matrix has the following form:

$$T_{bp} = \begin{bmatrix} \mathbf{I} & \mathbf{r} \\ \mathbf{0} & 1 \end{bmatrix} \tag{1}$$

where $\mathbf{r} = [r_x, r_y, r_z]^T$ is the position vector from base frame $\{b\}$ to platform frame $\{p\}$.

In *Kinematics of the Roll-Tilt Mechanism in SHER-3.0*, we derived:

$$T_{at} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & P_x \\ \sin(\theta) & \cos(\theta) & 0 & P_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$T_{pa} = \begin{bmatrix} 1 & 0 & 0 & d_2 \\ 0 & -\sin(\psi) & -\cos(\psi) & -\sin(\psi)d_3 \\ 0 & \cos(\psi) & -\sin(\psi) & \cos(\psi)d_3 + d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Thus, the transformation from base frame to tool frame is:

$$\begin{aligned} T_{bt} &= T_{bp}T_{pa}T_{at} \\ &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & P_x + d_2 + r_x \\ -\sin(\theta)\sin(\psi) & -\cos(\theta)\sin(\psi) & -\cos(\psi) & -\sin(\psi)P_y - \sin(\psi)d_3 + r_y \\ \sin(\theta)\cos(\psi) & \cos(\theta)\cos(\psi) & -\sin(\psi) & \cos(\psi)P_y + \cos(\psi)d_3 + d_1 + r_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4}$$

3.2 Spacial Jacobian

A brief recap of the **Spatial Jacobian**: it maps joint velocities to the tool tip velocity, $\mathbf{v}_{tool} = [\dot{x}, \dot{y}, \dot{z}, \dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z]^T$, expressed in the base frame $\{b\}$.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}_{\{b\}} = \mathbf{J_s} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix} \tag{5}$$

Since q_1, q_2, q_3 only generate translational motion, the upper-left 3×3 block of the Spatial Jacobian $\mathbf{J_s}$ corresponds to the Jacobian of the delta platform, $\mathbf{J_d}$, which was derived in *Kinematics of a 3-PUU Delta-Style Parallel Robot*:

$$\mathbf{J_d} = \begin{bmatrix} x_{l_1} & y_{l_1} & z_{l_1} \\ x_{l_2} & y_{l_2} & z_{l_2} \\ x_{l_3} & y_{l_3} & z_{l_3} \end{bmatrix}^{-1} \begin{bmatrix} z_{l_1} & 0 & 0 \\ 0 & z_{l_2} & 0 \\ 0 & 0 & z_{l_3} \end{bmatrix} \tag{6}$$

Similarly, the lower-right 3×2 block of the Spatial Jacobian is only related to q_4 and q_5 since only q_4 and q_5 can provide rotational velocity.

Now things become a bit more complex. As previously mentioned, due to the nature of the four-bar linkage, q_5 induces both rotational and translational motion, and q_4 contributes to linear velocity as well, since the tool tip is offset from its axis of rotation.

Let's break this down and construct the Spatial Jacobian one by one.

Recall in *Kinematics of the Roll-Tilt Mechanism in SHER-3.0*, we derived the tool tip velocity in the delta platform frame $\{p\}$ as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}_{\{p\}} = \begin{bmatrix} \dot{P}_{x\{a\}} \\ -\sin(\psi)\dot{P}_{y\{a\}} - \cos(\psi)\dot{\psi}(P_{y\{a\}} + d_3) \\ \cos(\psi)\dot{P}_{y\{a\}} - \sin(\psi)\dot{\psi}(P_{y\{a\}} + d_3) \\ \dot{\psi} \\ -\cos(\psi)\dot{\theta}_2 \\ -\sin(\psi)\dot{\theta}_2 \end{bmatrix} \tag{7}$$

where

$$\begin{aligned} \dot{P}_{x\{a\}} &= -\sin(\theta_1)L_{DA}\dot{\theta}_1 - \sin(\theta_2 - \phi_3)L_{DP}\dot{\theta}_2 \\ \dot{P}_{y\{a\}} &= \cos(\theta_1)L_{DA}\dot{\theta}_1 + \cos(\theta_2 - \phi_3)L_{DP}\dot{\theta}_2 \\ \theta_1 &= 180^\circ - (\phi_2 + \alpha_3 + \alpha_4) \\ \theta_2 &= \beta_1 + \phi_1 - \delta \end{aligned} \tag{8}$$

and previously, we also derived the angular velocity of each link:

$$\begin{aligned} \dot{\theta}_1 &= -\frac{L_{ID}}{L_{DA}}\dot{\theta}_2 \\ \dot{\theta}_2 &= \frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}}\dot{s} \end{aligned} \tag{9}$$

Plug eq.(9) into eq.(8):

$$
\begin{aligned}
\dot{P}_{x\{a\}} &= \left(\sin(\theta_1)\frac{L_{DA}L_{JQ}}{L_{AQ}L_{JR}} - \sin(\theta_2 - \phi_3)\frac{L_{DP}L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}}\right)\dot{s} \\
\dot{P}_{y\{a\}} &= \left(-\cos(\theta_1)\frac{L_{DA}L_{JQ}}{L_{AQ}L_{JR}} + \cos(\theta_2 - \phi_3)\frac{L_{DP}L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}}\right)\dot{s}
\end{aligned}
\tag{10}
$$

Let:

$$
\begin{aligned}
A &= \left(\sin(\theta_1)\frac{L_{DA}L_{JQ}}{L_{AQ}L_{JR}} - \sin(\theta_2 - \phi_3)\frac{L_{DP}L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}}\right) \\
B &= \left(-\cos(\theta_1)\frac{L_{DA}L_{JQ}}{L_{AQ}L_{JR}} + \cos(\theta_2 - \phi_3)\frac{L_{DP}L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}}\right)
\end{aligned}
\tag{11}
$$

Since $\dot{\psi} = \dot{q}_4$ and $\dot{s} = \dot{q}_5$, we can rewrite eq.(7) as:

$$
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}_{\{p\}} = \begin{bmatrix} 0 & A \\ -\cos(\psi)(P_{y\{a\}} + d_3) & -\sin(\psi)B \\ -\sin(\psi)(P_{y\{a\}} + d_3) & \cos(\psi)B \\ 1 & 0 \\ 0 & -\cos(\psi)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \\ 0 & -\sin(\psi)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \end{bmatrix} \begin{bmatrix} \dot{q}_4 \\ \dot{q}_5 \end{bmatrix}
\tag{12}
$$

Because there is no rotation between the base frame $\{b\}$ and the delta platform frame $\{p\}$, the velocity of the tool tip expressed in $\{b\}$ is simply the sum of the delta platform's velocity (in $\{b\}$) and the tool tip's velocity in fixed $\{p\}$ frame.

With that, we now have all the components needed to construct the Spatial Jacobian:

$$
\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}_{\{b\}} = \begin{bmatrix} \mathbf{J_d}_{[1,1]} & \mathbf{J_d}_{[1,2]} & \mathbf{J_d}_{[1,3]} & 0 & A \\ \mathbf{J_d}_{[2,1]} & \mathbf{J_d}_{[2,2]} & \mathbf{J_d}_{[2,3]} & -\cos(\psi)(P_{y\{a\}} + d_3) & -\sin(\psi)B \\ \mathbf{J_d}_{[3,1]} & \mathbf{J_d}_{[3,2]} & \mathbf{J_d}_{[3,3]} & -\sin(\psi)(P_{y\{a\}} + d_3) & \cos(\psi)B \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -\cos(\psi)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \\ 0 & 0 & 0 & 0 & -\sin(\psi)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix}
\tag{13}
$$

Since $\psi = q_4$, the Spacial Jacobian is:

$$
\mathbf{J_s} = \begin{bmatrix} \mathbf{J_d}_{[1,1]} & \mathbf{J_d}_{[1,2]} & \mathbf{J_d}_{[1,3]} & 0 & A \\ \mathbf{J_d}_{[2,1]} & \mathbf{J_d}_{[2,2]} & \mathbf{J_d}_{[2,3]} & -\cos(q_4)(P_{y\{a\}} + d_3) & -\sin(q_4)B \\ \mathbf{J_d}_{[3,1]} & \mathbf{J_d}_{[3,2]} & \mathbf{J_d}_{[3,3]} & -\sin(q_4)(P_{y\{a\}} + d_3) & \cos(q_4)B \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -\cos(q_4)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \\ 0 & 0 & 0 & 0 & -\sin(q_4)\frac{L_{DA}L_{JQ}}{L_{ID}L_{AQ}L_{JR}} \end{bmatrix}
\tag{14}
$$

3.3 Body Jacobian

Similar to the Spatial Jacobian, the **Body Jacobian** also maps joint velocities to the tool tip velocity, but expressed in the tool frame $\{t\}$.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \end{bmatrix}_{\{t\}} = \mathbf{J_b} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix} \tag{15}$$

In the traditional formulation of robot kinematics, **twists** (which combine angular and linear velocity) are often used in conjunction with exponential coordinates and homogeneous transformation matrices. When switching between spatial and body frames, the adjoint transformation is typically applied as:

$$\mathbf{V_{body}} = \mathbf{Ad}_{(\mathbf{T_{bt}^{-1}})} \mathbf{V_{spacial}} \tag{16}$$

where $\mathbf{Ad}_{(\mathbf{T_{bt}^{-1}})}$ is the adjoint matrix of the inverse transformation from the base frame $\{b\}$ to the tool frame $\{t\}$. This standard adjoint takes both rotation and translation into account:

$$\mathbf{Ad}_{(\mathbf{T})} = \begin{bmatrix} \mathbf{R} & 0 \\ [\mathbf{p}]\mathbf{R} & \mathbf{R} \end{bmatrix} \tag{17}$$

Here, $\mathbf{R}$ is the rotation matrix from one frame to another, $\mathbf{p}$ is the position vector between origins, and $[\mathbf{p}]$ is the skew-symmetric matrix of $\mathbf{p}$.

However, in our case, we are not using twist representation or exponential coordinates directly. Instead, we are dealing with raw velocity vectors: stacked linear and angular velocity components in the form $[\mathbf{v}; \omega]$. These do not contain the full twist structure and lack the translational offset context needed for the full adjoint to apply correctly.

As such, the traditional adjoint would incorrectly mix rotational and translational effects when applied to our representation. To transform raw velocity vectors between frames, only the rotation part should be applied to each component separately. For example, $\mathbf{v}_t = \mathbf{R_{tb}} \mathbf{v}_b$ and $\omega_t = \mathbf{R_{tb}} \omega_b$.

Thus, we define a simplified adjoint-like mapping $\mathbf{Ad}^{\dagger}_{(\mathbf{T_{tb}})}$ using only the rotation matrix:

$$\mathbf{Ad}^{\dagger}_{(\mathbf{T_{tb}})} = \begin{bmatrix} \mathbf{R_{tb}} & 0 \\ 0 & \mathbf{R_{tb}} \end{bmatrix} \tag{18}$$

where $\mathbf{R_{tb}} = \mathbf{R_{bt}}^T = \mathbf{T_{bt}}[:\mathbf{3}, :\mathbf{3}]^T$, and $\mathbf{T_{bt}}$ is already been derived in eq.(4).

Finally, we get the expression of the **Body Jacobian**:

$$\mathbf{J_b} = \mathbf{Ad}^{\dagger}_{(\mathbf{T_{tb}})} \mathbf{J_s} \tag{19}$$

4 Jacobian Pseudo-inverse of SHER-3.0

In robotic kinematics, the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{6\times n}$ maps joint velocities $\dot{\mathbf{q}} \in \mathbb{R}^n$ to end-effector velocity $\mathbf{v} \in \mathbb{R}^6$, and Jacobian inverse does the opposite thin as:

$$\begin{aligned} \mathbf{v} &= \mathbf{J}\dot{\mathbf{q}} \\ \dot{\mathbf{q}} &= \mathbf{J}^{-1}\mathbf{v} \end{aligned} \tag{20}$$

This works well when the number of joints $n = 6$, making $\mathbf{J}$ square and (if full rank) invertible. However, in our case, we only have **5 joints**, so the Jacobian is a 6×5 matrix. That means it's **not directly invertible** and no exact solution may exist for a general 6D velocity command.

To resolve this, we use the **Moore-Penrose pseudo-inverse** of the Jacobian, denoted $\mathbf{J}^\dagger = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T$, to find a **least-squares optimal** solution for the joint velocities $\dot{\mathbf{q}}$ that best approximates the desired end-effector velocity:

$$\dot{\mathbf{q}} = \mathbf{J}^\dagger\mathbf{v} \tag{21}$$

This gives us the joint velocity vector that minimizes the squared error between the actual and desired end-effector motion. It essentially "projects" the 6D desired velocity into the 5D space of achievable motions.

4.1 Spacial Jacobian Pseudo-inverse

Since the Moore-Penrose pseudo-inverse is a general-purpose mathematical tool that can be applied to any non-square Jacobian matrix, it works for both Spatial and Body Jacobians.

Thus,

$$\mathbf{J_s}^\dagger = (\mathbf{J_s}^T\mathbf{J_s})^{-1}\mathbf{J_s}^T \tag{22}$$

$$\dot{\mathbf{q}} = \mathbf{J_s}^\dagger\mathbf{v_s} \tag{23}$$

4.2 Body Jacobian Pseudo-inverse

Similarly,

$$\mathbf{J_b}^\dagger = (\mathbf{J_b}^T\mathbf{J_b})^{-1}\mathbf{J_b}^T \tag{24}$$

$$\dot{\mathbf{q}} = \mathbf{J_b}^\dagger\mathbf{v_b} \tag{25}$$

5 Validation with SolidWorks Motion Analysis

To validate the kinematic calculations presented in this document, I used **SolidWorks** to build a simplified CAD model of the SHER-3.0 mechanism. The assembly was constructed without incorporating any built-in kinematic constraints or control logic—only individual parts and their mechanical relationships were defined.

A motion simulation was then performed with **SolidWorks Motion Study**. Since SolidWorks Motion Simulation does not provide instantaneous velocity outputs at a specific configuration, I set the total simulation time to 0.5 seconds and observed the velocity values at time t=0.

At the initial moment, the model has not yet moved significantly, so the velocity at t=0 effectively represents the instantaneous velocity of the system at the given joint configuration. This allows for a valid comparison between the simulation results and the analytical velocity computed using the Jacobian-based method described in this document.

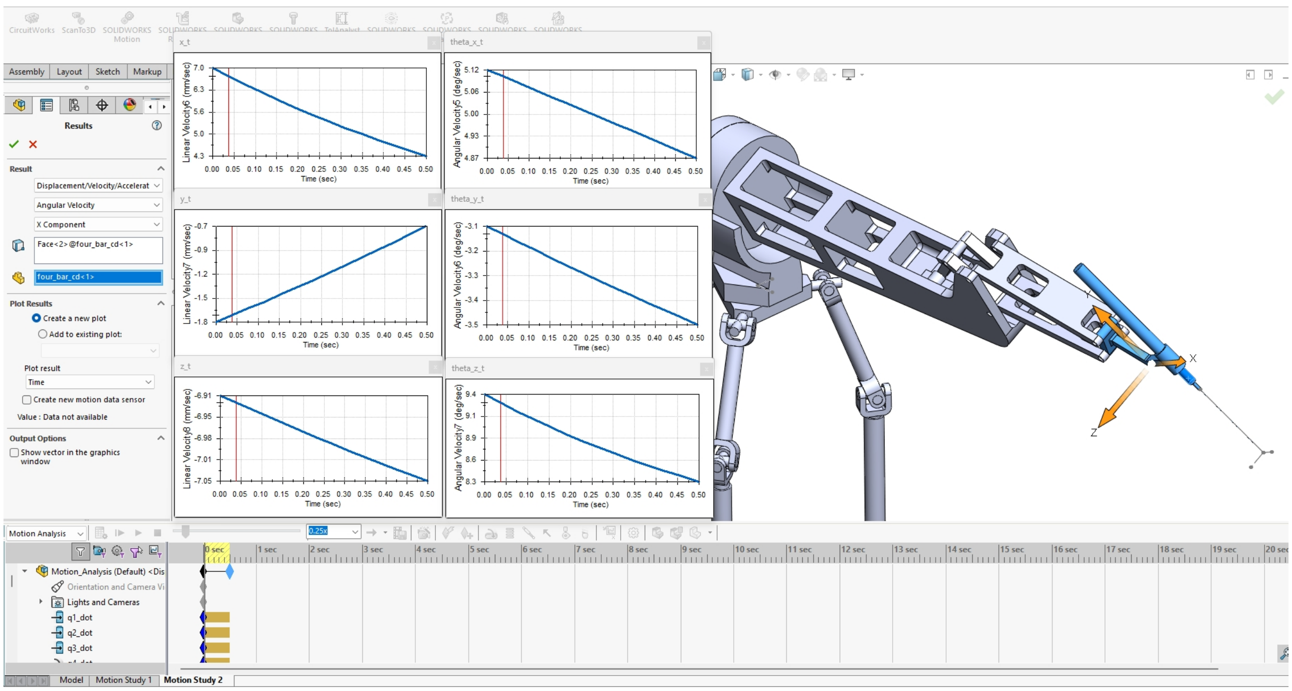

Figure 3: Motion Simulation Setup in SolidWorks

To test the model with arbitrary values, consider the following joint positions and velocities:

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{bmatrix} = \begin{bmatrix} 1\ mm \\ 2\ mm \\ 3\ mm \\ \frac{\pi}{6}\ rad \\ 10\ mm \end{bmatrix}, \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix} = \begin{bmatrix} 4\ mm/s \\ 5\ mm/s \\ 6\ mm/s \\ 1\ rad/s \\ 10\ mm/s \end{bmatrix}$$

5.1 Spacial Jacobian and Jacobian Pseudo-inverse

The simulated results, such as the position, orientation, and velocity of the end-effector, closely matched the analytical results computed using Python code (attached in the appendix) based on the method described in this document, confirming the correctness of the kinematic model.

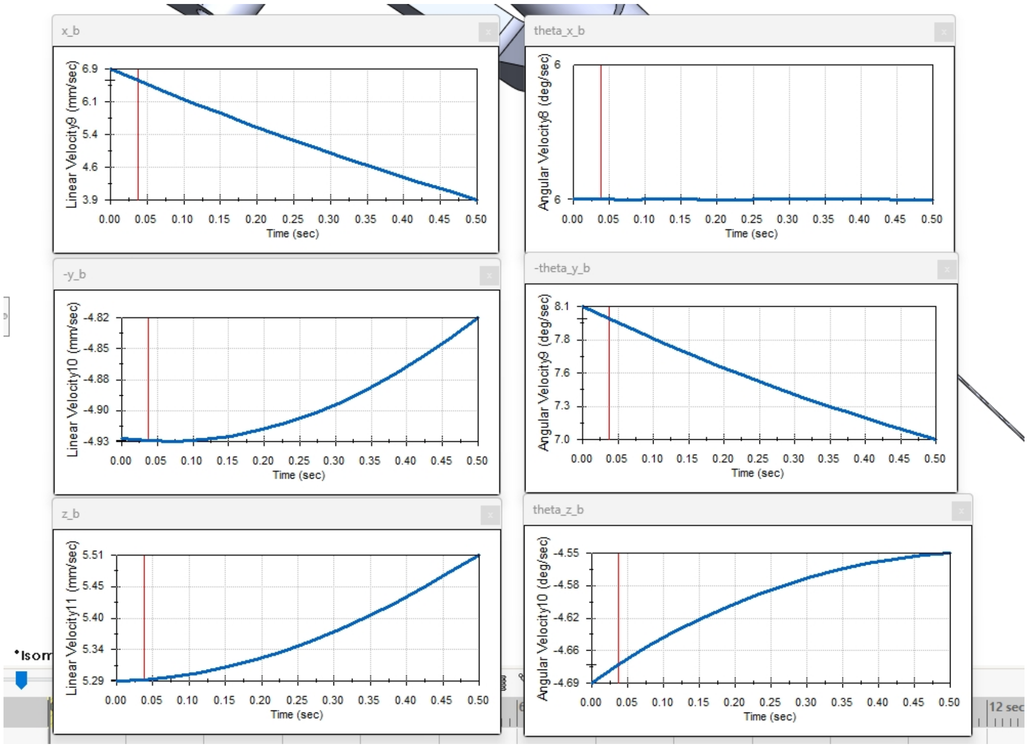

Figure 4: Tool Tip Velocity in Base Frame (SolidWorks Motion Simulation)

```
In [21]: runfile('G:/My Drive/Botao/Jobs/Delta_Robot/JacobianVerification.py', wdir='G:/My Drive/Botao/Jobs/Delta_Robot')

In base frame:

x_dot_b =  6.87802679981911 mm/s
y_dot_b =  4.93238197732434 mm/s
z_dot_b =  5.28504434847777 mm/s
theta_x_dot_b =  6.00000000000000 deg/s
theta_y_dot_b =  -8.14832999948954 deg/s
theta_z_dot_b =  -4.70444051865119 deg/s

Joint Velocity from Jacobian Pseudo-Inverse:
q_dot =
⎡3.99999999999976 ⎤
⎢                 ⎥
⎢4.99999999999823 ⎥
⎢                 ⎥
⎢5.99999999999941 ⎥
⎢                 ⎥
⎢0.104719755119696⎥
⎢                 ⎥
⎣10.0000000000024 ⎦
```

Figure 5: Tool Tip Velocity in Base Frame & Jacobian Pseudo-inverse Validation (Python Script)

5.2 Body Jacobian and Jacobian Pseudo-inverse

Similarly, the simulated results matched the analytical results as shown below:

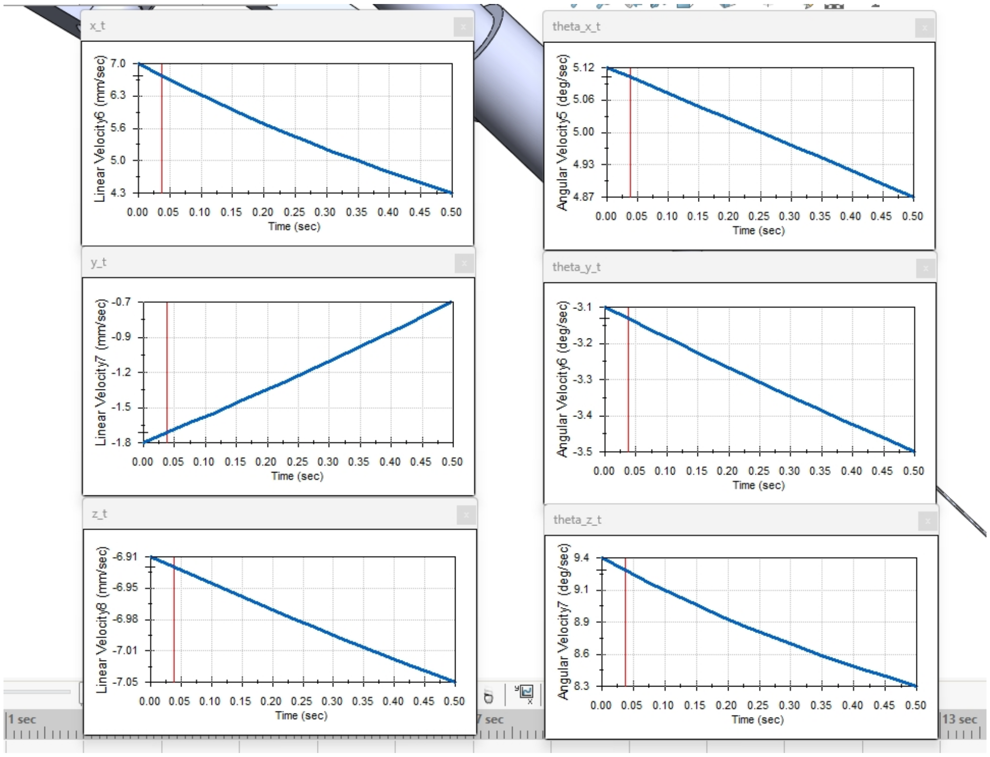

Figure 6: Tool Tip Velocity in Tool Frame (SolidWorks Motion Simulation)

```
In [22]: runfile('G:/My Drive/Botao/Jobs/Delta_Robot/JacobianVerification.py', wdir='G:/My Drive/Botao/Jobs/Delta_Robot')

In tool frame:

x_dot_t =  6.97252562855088 mm/s
y_dot_t =  -1.77386034483160 mm/s
z_dot_t =  -6.91409026777029 mm/s
theta_x_dot_t =  5.12488415034004 deg/s
theta_y_dot_t =  -3.12018628379677 deg/s
theta_z_dot_t =  9.40888103730238 deg/s

Joint Velocity from Jacobian Pseudo-Inverse:
q_dot =
[3.99999999999912 ]
[                 ]
[4.99999999999957 ]
[                 ]
[5.99999999999927 ]
[                 ]
[0.104719755119721]
[                 ]
[9.99999999999841 ]
```

Figure 7: Tool Tip Velocity in Tool Frame & Jacobian Pseudo-inverse Validation (Python Script)

6 Disclaimer

The simulations and Jacobian-based calculations presented in this document are based on my personal understanding of the kinematic structure of SHER-3.0. Due to the complexity of the Jacobian—especially in systems involving multiple frames and mixed translational and rotational joints—there may be discrepancies between analytical results and simulation outputs. While the SolidWorks motion simulation validates the general correctness of the method, it is not guaranteed to match perfectly due to potential simplifications, numerical errors, or modeling assumptions. I encourage readers to focus on the methodology and reasoning behind the derivations rather than relying solely on the final equations. Please independently verify the results before applying them to any critical applications. If you spot any errors, have suggestions, or would like to discuss the topic further, feel free to reach out to me at: bzhao17@alumni.jh.edu.

7 Appendix: Python Scripts

7.1 Jacobian & Jacobian Pseudo-inverse Validation

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Jul 27 15:24:01 2025

@author: Botao
"""

import numpy as np
from sympy import Matrix, zeros, pprint

################## Joint Parameters: ##############################
q1 = 1  # Prismatic
q2 = 2  # Prismatic
q3 = 3  # Prismatic
q4 = 30*np.pi/180  # Revolute
q5 = 10  # Prismatic

################## Joint Velocities: ##############################
q1_dot = 4  # mm/s
q2_dot = 5  # mm/s
q3_dot = 6  # mm/s
q4_dot = 6*np.pi/180  # rad/s
q5_dot = 10  # mm/s
q_dot = Matrix([
        [q1_dot],
        [q2_dot],
        [q3_dot],
        [q4_dot],
        [q5_dot]
])

################## Delta Platform: ###################################

# 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])


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

r = np.array([x, y, z])

l1 = -L1 - rb1 + r + rp1
l2 = -L2 - rb2 + r + rp2
l3 = -L3 - rb3 + r + rp3


################ Roll-Tilt Mechanism: ###########################

s = q5
s_dot = q5_dot
psi = q4 # roll angle

# Fixed Parameters
x_AR_max = 60.48
y_AR = -10
L_AQ = 32.5
L_QR = 42
L_AB = 28
L_BC = 78.5
L_CD = 15
L_DA = 73.5
phi1 = np.pi / 2
phi2 = np.pi * 138 / 180
L_DP = 94.49
phi3 = np.pi * 103.93 / 180
phi4 = np.pi * 31.06 / 180
L_T = 50
```

```python
d1 = 56.01 # Distance between the center frame of the delta platform to point A
d2 = 192.7
d3 = 25.5

# Angles in radians
phi1 = np.pi / 2
phi2 = np.pi * 138 / 180
phi3 = np.pi * 103.93 / 180
phi4 = np.pi * 31.06 / 180

# Position Analysis
x_AR = x_AR_max - s
L_AR = np.sqrt(x_AR**2 + y_AR**2)
alpha3 = np.arccos((L_AR**2 + L_AQ**2 - L_QR**2) / (2 * L_AR * L_AQ))
alpha4 = np.pi - np.arctan2(y_AR, -x_AR)
alpha = alpha3 + alpha4

theta1 = np.pi - (phi2 + alpha3 + alpha4)
alpha_1 = phi1 - theta1
L_BD = np.sqrt(L_AB**2 + L_DA**2 - 2 * L_AB * L_DA * np.cos(alpha_1))
beta_1 = np.arccos((L_AB**2 + L_BD**2 - L_DA**2) / (2 * L_AB * L_BD))

delta = np.arccos((L_CD**2 + L_BD**2 - L_BC**2) / (2 * L_CD * L_BD))
theta2 = phi1 + beta_1 - delta

# Velocity Analysis
L_JR = np.tan(alpha)*x_AR - y_AR
L_JQ = np.sqrt((L_JR+y_AR)**2 + x_AR**2) - L_AQ

rho_dot = s_dot / L_JR
theta1_dot = -L_JQ / L_AQ * rho_dot

beta_2 = np.arccos((L_BC**2 + L_BD**2 - L_CD**2) / (2 * L_BC * L_BD))
L_IA = L_AB * np.sin(beta_1 + beta_2) / np.sin(alpha_1 + beta_1 + beta_2)
L_ID = L_IA - L_DA

theta2_dot = -L_DA / L_ID * theta1_dot

# Position in {a} frame:
px_a = np.cos(theta1)*L_DA + np.cos(theta2 - phi3)*L_DP
py_a = np.sin(theta1)*L_DA + np.sin(theta2 - phi3)*L_DP

# Position in {p} frame:
px_p = px_a + d2
py_p = -np.sin(psi)*py_a - np.sin(psi)*d3
pz_p = np.cos(psi)*py_a + np.cos(psi)*d3 + d1

# Position in {b} frame:
px_b = px_a + d2 + x
py_b = -np.sin(psi)*py_a - np.sin(psi)*d3 + y
pz_b = np.cos(psi)*py_a + np.cos(psi)*d3 + d1 + z


################# Spacial Jacobian ###############

J_l = zeros(3,3)

J_l[0,0] = l1[0]
J_l[0,1] = l1[1]
```

```
J_l[0,2] = l1[2]
J_l[1,0] = l2[0]
J_l[1,1] = l2[1]
J_l[1,2] = l2[2]
J_l[2,0] = l3[0]
J_l[2,1] = l3[1]
J_l[2,2] = l3[2]

J_z = zeros(3,3)

J_z[0,0] = l1[2]
J_z[1,1] = l2[2]
J_z[2,2] = l3[2]

J_d = J_l.inv() * J_z


A = np.sin(theta1)*L_DA*L_JQ/(L_AQ*L_JR) - np.sin(theta2-phi3)*L_DP*L_DA*L_JQ/(L_ID*L_AQ*L_JR)
B = -np.cos(theta1)*L_DA*L_JQ/(L_AQ*L_JR) + np.cos(theta2-phi3)*L_DP*L_DA*L_JQ/(L_ID*L_AQ*L_JR)

JacobianTip = zeros(6,5)

JacobianTip[:3, :3] = J_d

JacobianTip[1,3] = -np.cos(q4)*(py_a + d3)
JacobianTip[2,3] = -np.sin(q4)*(py_a + d3)
JacobianTip[3,3] = 1

JacobianTip[0,4] = A
JacobianTip[1,4] = -np.sin(q4)*B
JacobianTip[2,4] = np.cos(q4)*B
JacobianTip[4,4] = -np.cos(q4)*L_DA*L_JQ/(L_ID*L_AQ*L_JR) # Should there be a "-" in front?
JacobianTip[5,4] = -np.sin(q4)*L_DA*L_JQ/(L_ID*L_AQ*L_JR)

Js = JacobianTip  # 6x5 spatial Jacobian in base frame


################ Body Jacobian ##################

def adjoint_pseudo(T):
    R = T[:3, :3]
    return Matrix.vstack(
        Matrix.hstack(R, Matrix.zeros(3)),
        Matrix.hstack(Matrix.zeros(3), R)
    )

# Tbt is known
theta = np.pi/2 - phi4 - phi3 + theta2
Tbt = Matrix([
    [np.cos(theta), -np.sin(theta), 0, px_a+d2+x],
    [-np.sin(theta)*np.sin(psi), -np.cos(theta)*np.sin(psi), -np.cos(psi), -np.sin(psi)*py_a-np.sin(psi)*d3+y],
    [np.sin(theta)*np.cos(psi), np.cos(theta)*np.cos(psi), -np.sin(psi), np.cos(psi)*py_a+np.cos(psi)*d3+d1+z],
    [0, 0, 0, 1]
    ])  # 4x4 transformation from base to tool
Rbt = Tbt[:3, :3]

# Compute Ad_Tbt¹
Ttb = Tbt.inv()
Ad_Ttb = adjoint_pseudo(Ttb)
```

```python

# Compute body Jacobian
Jb = Ad_Ttb * Js

V_tool_t = Jb*q_dot
V_tool_b = Js*q_dot

# print("\nIn base frame:\n")
# print("x_dot_b = ", V_tool_b[0], "mm/s")
# print("y_dot_b = ", V_tool_b[1], "mm/s")
# print("z_dot_b = ", V_tool_b[2], "mm/s")
# print("theta_x_dot_b = ", V_tool_b[3]*180/np.pi, "deg/s")
# print("theta_y_dot_b = ", V_tool_b[4]*180/np.pi, "deg/s")
# print("theta_z_dot_b = ", V_tool_b[5]*180/np.pi, "deg/s\n")


print("\nIn tool frame:\n")
print("x_dot_t = ", V_tool_t[0], "mm/s")
print("y_dot_t = ", V_tool_t[1], "mm/s")
print("z_dot_t = ", V_tool_t[2], "mm/s")
print("theta_x_dot_t = ", V_tool_t[3]*180/np.pi, "deg/s")
print("theta_y_dot_t = ", V_tool_t[4]*180/np.pi, "deg/s")
print("theta_z_dot_t = ", V_tool_t[5]*180/np.pi, "deg/s\n")



################ Jacobian Inverse #################

Js_pseudo_inv = (Js.T * Js).inv() * Js.T
Jb_pseudo_inv = (Jb.T * Jb).inv() * Jb.T

q_dot_verify_s = Js_pseudo_inv * V_tool_b
q_dot_verify_b = Jb_pseudo_inv * V_tool_t

print("Joint Velocity from Jacobian Pseudo-Inverse:")
print("q_dot =")
pprint(q_dot_verify_s) # The results should match your q_dot values
```