Fun Robo Mini Project 1

Sam Wisnoski, Bill Le, Satchel Schiavo

Github: https://github.com/titut/arm-kinematics-module/tree/main

AI Disclaimer: Sam used Chatgpt for LaTeX formatting. No other AI was used.

Introduction

For this project, we have the following three tasks:

1. Problem 1:
   (a) Derive the DH parameters and table for the 5DOF robot platform
   (b) Derive the FPK equations and implement in software with verification in the Viz tool
2. Problem 2:
   (a) Derive the inverse jacobian matrix and implement a resolved-rate motion control (RRMC) with verification in the Viz tool
3. Problem 3:
   (a) Implement the resolved-rate motion control (RRMC) through gamepad control of the 5DOF robot hardware

Problem 1: Reference Frames and DH Table

Our first step in deriving a DH table for our 5DOF robot was to create a diagram of the robot arm with the coordinate frames clearly defined. We also labeled the links and joints accordingly.

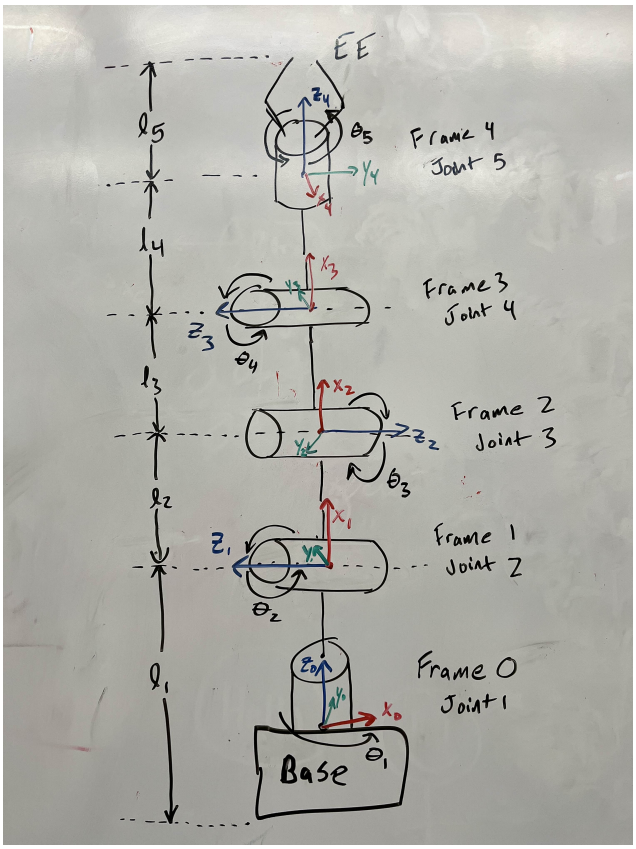

Figure 1: A diagram of our 5DOF with labeled reference frames, joints, and links.

From the diagram in **Figure 1**, we were able to derive the DH parameters found in **Table 1**, allowing us to calculate the forward kinematics equations for the 5DOF robot platform.

Frame #	θ_i	d_i	a_i	α_i
Frame 0	θ_1	l_1	0	$\pi/2$
Frame 0.5	$\pi/2$	0	0	0
Frame 1	θ_2	0	l_2	π
Frame 2	θ_3	0	l_3	π
Frame 3	θ_4	0	l_4	0
Frame 3.5	$-\pi/2$	0	0	$-\pi/2$
Frame 4	θ_5	l_5	0	0

Table 1: DH Table for our 5DOF robot.

From our DH table, we can then derive the FPK equations. From each line of our DH table, we can find the transformation from reference frame H_{i-1} to H_i.

$$^{0}H_{0.5} = \begin{bmatrix} & & 0 \\ R_{Z(\theta_1)}R_{X(\alpha_1)} & & 0 \\ & & l_1 \\ 0 & & 1 \end{bmatrix}$$

$$^{0.5}H_{1} = \begin{bmatrix} & & 0 \\ R_{Z(\pi/2)} & & 0 \\ & & 0 \\ 0 & & 1 \end{bmatrix}$$

$$^{1}H_{2} = \begin{bmatrix} & & l_2 \\ R_{Z(\theta_2)}R_{X(\pi)} & & 0 \\ & & 0 \\ 0 & & 1 \end{bmatrix}$$

$$^{2}H_{3} = \begin{bmatrix} & & l_3 \\ R_{Z(\theta_3)}R_{X(\pi)} & & 0 \\ & & 0 \\ 0 & & 1 \end{bmatrix}$$

$$^{3}H_{3.5} = \begin{bmatrix} & & l_4 \\ R_{Z(\theta_4)} & & 0 \\ & & 0 \\ 0 & & 1 \end{bmatrix}$$

$$^{3.5}H_{4} = \begin{bmatrix} & & 0 \\ R_{Z(-\pi/2)}R_{X(-\pi/2)} & & 0 \\ & & 0 \\ 0 & & 1 \end{bmatrix}$$

$$^{4}H_{5} = \begin{bmatrix} & & 0 \\ R_{Z(\theta_5)} & & 0 \\ & & l_5 \\ 0 & & 1 \end{bmatrix}$$

Our full transformation from frame H_0 to H_5 can be found by multiplying each of the transformations together, effectively translating from our base frame to our end effector.

$$^{0}H_{0.5} \cdot {}^{0.5}H_{1} \cdot {}^{1}H_{2} \cdot {}^{2}H_{3} \cdot {}^{3}H_{3.5} \cdot {}^{3.5}H_{4} \cdot {}^{4}H_{5} = {}^{0}H_{5} = {}^{0}H_{EE}$$

From our $^{0}H_5$ frame transformation, we can extract our FPK equations. Defined below, $^{0}R_{EE}$ is our rotation matrix from the base frame to the end effector frame, and $^{0}P_5$ is our position translation matrix from the base frame to the end effector frame.

$$^{0}H_{5} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad ^{0}R_{5} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \qquad ^{0}P_{5} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

In order to calculate ${}^0H_5$, and therefore our position and rotation matrices, we can use the MATLAB toolbox "DH Table Solver", linked here. This software gives us an easy way to compute our H matrices from DH tables. The implementation of the code is shown in **Figure 2**.

```
clear all

syms t1 t2 t3 t4 t5 l1 l2 l3 l4 l5

% create matrix representing DH table with order of [theta, alpha, r, d]

matrix =   [t1, pi/2, 0, l1;
            pi/2, 0, 0, 0;
            t2, pi, l2, 0;
            t3, pi, l3, 0;
            t4, 0, l4, 0;
            -pi/2, -pi/2, l4, 0;
            t5, 0, 0, l5];

h_matrix = simplify(DH_HTM(matrix, "r"))
```

Figure 2: Code using MATLAB toolbox "DH Table Solver" to create a transformation matrix based on given DH parameters.

When we run this code, we are returned a matrix with the following values:

$$ {}^0H_5 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^0R_5 = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \qquad {}^0P_5 = \begin{bmatrix} x \\ y \\ z \end{bmatrix} $$

Where:

$$
\begin{aligned}
r_{11} &= \cos(\theta_2 - \theta_3 + \theta_4)\cos(\theta_1)\cos(\theta_5) - \sin(\theta_1)\sin(\theta_5) \\
r_{12} &= -\cos(\theta_5)\sin(\theta_1) - \cos(\theta_2 - \theta_3 + \theta_4)\cos(\theta_1)\sin(\theta_5) \\
r_{13} &= -\sin(\theta_2 - \theta_3 + \theta_4)\cos(\theta_1) \\
r_{21} &= \cos(\theta_1)\cos(\theta_5) - \cos(\theta_2 - \theta_3 + \theta_4)\sin(\theta_1)\sin(\theta_5) \\
r_{22} &= \cos(\theta_1)\sin(\theta_5) + \cos(\theta_2 - \theta_3 + \theta_4)\cos(\theta_5)\sin(\theta_1) \\
r_{23} &= -\sin(\theta_2 - \theta_3 + \theta_4)\sin(\theta_1) \\
r_{31} &= \sin(\theta_2 - \theta_3 + \theta_4)\cos(\theta_5) \\
r_{32} &= -\sin(\theta_2 - \theta_3 + \theta_4)\sin(\theta_5) \\
r_{33} &= \cos(\theta_2 - \theta_3 + \theta_4) \\
x &= -\cos(\theta_1)\Big(l_4\sin(\theta_2 - \theta_3 + \theta_4) - l_4\cos(\theta_2 - \theta_3 + \theta_4) + l_5\sin(\theta_2 - \theta_3 + \theta_4) + l_2\sin(\theta_2) + l_3\sin(\theta_2 - \theta_3)\Big) \\
y &= -\sin(\theta_1)\Big(l_4\sin(\theta_2 - \theta_3 + \theta_4) - l_4\cos(\theta_2 - \theta_3 + \theta_4) + l_5\sin(\theta_2 - \theta_3 + \theta_4) + l_2\sin(\theta_2) + l_3\sin(\theta_2 - \theta_3)\Big) \\
z &= l_1 + l_4\cos(\theta_2 - \theta_3 + \theta_4) + l_5\cos(\theta_2 - \theta_3 + \theta_4) + l_4\sin(\theta_2 - \theta_3 + \theta_4) + l_2\cos(\theta_2) + l_3\cos(\theta_2 - \theta_3)
\end{aligned}
$$

While it's nice to be able to calculate our values using MATLAB, it's much more useful to calculate them directly in python, which allows us to test on the visualizer and communicate with the robot. However, it means we have to calculate each of the matrices on our own rather than using a prebuilt function. To do this, we can use a simple homemade python function to calculate the matrices based on the input of a DH table.

For our first step, we input the values from the DH table (**Table 1**) we've created into the code. To do this, we have a list storing only theta values. The index of the element in the list represents the row of the DH table that the theta value belongs to. Similarly, we have these lists for d, a, and alpha values.

Then, we create a transformation matrix from each of the indexes of the lists mentioned prior. Importantly, in our DH tables, there were two transformations between reference frames that required two rows of the DH table (from frame 0 to frame 1 and from frame 3 to frame 4). To account for this, we check for index 0 and 3. We multiply the current transformation matrix that results from the current index with the extra row of DH table value that followed it. Our resulting code is shown in **Figures 3 and 4**.

```python
def calc_DH_matrices(self):
    """Calculates all DH Matrices of the system"""
    # DH table parameters
    theta_i_table = [
        self.theta[0],
        self.theta[1],
        self.theta[2],
        self.theta[3],
        self.theta[4],
    ]
    d_table = [self.l1, 0, 0, 0, self.l5]
    r_table = [0, self.l2, self.l3, self.l4, 0]
    alpha_table = [np.pi / 2, np.pi, np.pi, 0, 0]
    for i in range(self.num_dof):
        if i == 0:
            self.DH[i] = self.DH_matrix(
                theta_i_table[i], d_table[i], r_table[i], alpha_table[i]
            ) @ self.DH_matrix(np.pi / 2, 0, 0, 0)
        elif i == 3:
            self.DH[i] = self.DH_matrix(
                theta_i_table[i], d_table[i], r_table[i], alpha_table[i]
            ) @ self.DH_matrix(-np.pi / 2, 0, 0, -np.pi / 2)
        else:
            self.DH[i] = self.DH_matrix(
                theta_i_table[i], d_table[i], r_table[i], alpha_table[i]
            )
```

Figure 3: Code to calculate all matrices of the the 5DOF robot. Note that this function only works for specifically the 5DOF robot since we define the parameters internally.

```python
def DH_matrix(self, theta, d, r, alpha):
    """Calculates DH matrix based on given arguments"""
    return np.array(
        [
            [
                cos(theta),
                -sin(theta) * cos(alpha),
                sin(theta) * sin(alpha),
                r * cos(theta),
            ],
            [
                sin(theta),
                cos(theta) * cos(alpha),
                -cos(theta) * sin(alpha),
                r * sin(theta),
            ],
            [0, sin(alpha), cos(alpha), d],
            [0, 0, 0, 1],
        ]
    )
```

Figure 4: Code to create a transformation matrix based on given DH parameters.

Lastly, we can verify these equations by implementing them in our visualization tool. A link to a video of our FPK equations working in the visualization tool can be found here.

The code in Figures 3 and 4, as well as their implementation into the visualizer can be found at our Github, linked here.

Problem 2: Reference Frames and DH Table

Our next task is to derive the inverse Jacobian matrix and implement a resolved-rate motion control (RRMC) for the same 5DOF robot.

The first step to deriving the inverse Jacobian matrix is to derive the generalized Jacobian, characterized as:

$$\begin{bmatrix}\overrightarrow{v}\\ \overrightarrow{\omega}\end{bmatrix} = J(\overrightarrow{\theta})\dot{\overrightarrow{\theta}}\text{; where } J = \begin{bmatrix}J_{v_1} & J_{v_2} & ... & J_{v_n}\\ J_{\omega_1} & J_{\omega_2} & ... & J_{\omega_n}\end{bmatrix}$$

To find each of our linear Jacobian velocities (J_{v_i}), we can use the formula:

$$J_{vi} = \overrightarrow{z_i} \times \overrightarrow{r_i}$$

To find each of our angular Jacobian velocities (J_{ω_i}), we can use the formula:

$$J_{\omega_i} = \overrightarrow{z_i}$$

To find our linear and angular velocities, we need to derive both the r and z vectors for each frame. Let's start with our z vectors, which are given by the equation:

$$\overrightarrow{z_i} = {}^0R_{i-1} \times \hat{k}\text{, where } \hat{k} = [0,0,1]^T$$

For our 5 DOF robot arm, with a total of eight frames (including intermediate frames), we have the following seven z vectors:

$\overrightarrow{z_{0.5}} = {}^0R_0 \times \hat{k} \rightarrow$ frame 0 to frame 0:

$$\begin{bmatrix}0\\0\\0\end{bmatrix}$$

$\overrightarrow{z_1} = {}^0R_{0.5} \times \hat{k} \rightarrow$ frame 0 to frame 0.5:

$$\begin{bmatrix}sin(\theta_1)\\-cos(\theta_1)\\0\end{bmatrix}$$

$\overrightarrow{z_2} = {}^0R_1 \times \hat{k} \rightarrow$ frame 0 to frame 1:

$$\begin{bmatrix}sin(\theta_1)\\-cos(\theta_1)\\0\end{bmatrix}$$

$\overrightarrow{z_3} = {}^0R_2 \times \hat{k} \rightarrow$ frame 0 to frame 2:

$$\begin{bmatrix}-sin(\theta_1)\\cos(\theta_1)\\0\end{bmatrix}$$

$\overrightarrow{z_{3.5}} = {}^0R_3 \times \hat{k} \rightarrow$ frame 0 to frame 3:

$$\begin{bmatrix}sin(\theta_1)\\-cos(\theta_1)\\0\end{bmatrix}$$

$\overrightarrow{z_4} = {}^0R_{3.5} \times \hat{k} \rightarrow$ frame 0 to frame 3.5:

$$\begin{bmatrix}-sin(\theta_2-\theta_3+\theta_4)\cdot cos(\theta_1)\\-sin(\theta_2-\theta_3+\theta_4)\cdot sin(\theta_1)\\cos(\theta_2-\theta_3+\theta_4)\end{bmatrix}$$

$\overrightarrow{z_5} = {}^0R_4 \times \hat{k} \rightarrow$ frame 0 to frame 4:

$$\begin{bmatrix} -sin(\theta_2 - \theta_3 + \theta_4) \cdot cos(\theta_1) \\ -sin(\theta_2 - \theta_3 + \theta_4) \cdot sin(\theta_1) \\ cos(\theta_2 - \theta_3 + \theta_4) \end{bmatrix}$$

Remember, we only want rotation to the frame $i - 1$, which is why the frames stop at 4; our end effector constitutes our fifth and final frame.

After computing the z vectors, we have already found our angular Jacobians, and are halfway to finding our linear Jacobians. Next, we need to find the r vectors for each frame. This is the x, y and z distances between each of our frames and the end effector frame (frame 5). To find each, we simply compute variations our DH table, starting from the first term to the end effector. For each next vector, we remove the top row (or bottom frame) of our DH table. So for example, 0-5 r is the full DH table, and 0.5-5 r is the DH table with the first row removed.

$\overrightarrow{r_{0.5}} = \overrightarrow{d_5^0} - \overrightarrow{d_0^0} \rightarrow$ frame 0 to frame 5:

$$\begin{bmatrix} -cos(\theta_1) \cdot (l4 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l5 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l2 \cdot sin(\theta_2) + l3 \cdot sin(\theta_2 - \theta_3)) \\ -sin(\theta_1) \cdot (l4 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l5 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l2 \cdot sin(\theta_2) + l3 \cdot sin(\theta_2 - \theta_3)) \\ l1 + l4 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l5 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l2 \cdot cos(\theta_2) + l3 \cdot cos(\theta_2 - \theta_3) \end{bmatrix}$$

$\overrightarrow{r_1} = \overrightarrow{d_5^0} - \overrightarrow{d_{0.5}^0} \rightarrow$ frame 0.5 to frame 5:

$$\begin{bmatrix} -l4 \cdot sin(\theta_2 - \theta_3 + \theta_4) - l5 \cdot sin(\theta_2 - \theta_3 + \theta_4) - l2 \cdot sin(\theta_2) - l3 \cdot sin(\theta_2 - \theta_3) \\ l4 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l5 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l2 \cdot cos(\theta_2) + l3 \cdot cos(\theta_2 - \theta_3) \\ 0 \end{bmatrix}$$

$\overrightarrow{r_2} = \overrightarrow{d_5^0} - \overrightarrow{d_1^0} \rightarrow$ frame 1 to frame 5:

$$\begin{bmatrix} l4 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l5 \cdot cos(\theta_2 - \theta_3 + \theta_4) + l2 \cdot cos(\theta_2) + l3 \cdot cos(\theta_2 - \theta_3) \\ l4 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l5 \cdot sin(\theta_2 - \theta_3 + \theta_4) + l2 \cdot sin(\theta_2) + l3 \cdot sin(\theta_2 - \theta_3) \\ 0 \end{bmatrix}$$

$\overrightarrow{r_3} = \overrightarrow{d_5^0} - \overrightarrow{d_2^0} \rightarrow$ frame 2 to frame 5:

$$\begin{bmatrix} l3 \cdot cos(\theta_3) + l4 \cdot cos(\theta_3 - \theta_4) + l5 \cdot cos(\theta_3 - \theta_4) \\ l3 \cdot sin(\theta_3) + l4 \cdot sin(\theta_3 - \theta_4) + l5 \cdot sin(\theta_3 - \theta_4) \\ 0 \end{bmatrix}$$

$\overrightarrow{r_{3.5}} = \overrightarrow{d_5^0} - \overrightarrow{d_3^0} \rightarrow$ frame 3 to frame 5:

$$\begin{bmatrix} cos(\theta_4) \cdot (l4 + l5) \\ sin(\theta_4) \cdot (l4 + l5) \\ 0 \end{bmatrix}$$

$\overrightarrow{r_4} = \overrightarrow{d_5^0} - \overrightarrow{d_{3.5}^0} \rightarrow$ frame 3.5 to frame 5:

$$\begin{bmatrix} l5 \\ 0 \\ 0 \end{bmatrix}$$

$\overrightarrow{r_5} = \overrightarrow{d_5^0} - \overrightarrow{d_4^0} \rightarrow$ frame 4 to frame 5:

$$\begin{bmatrix} 0 \\ 0 \\ l5 \end{bmatrix}$$

Wow! That was a lot of terms. But now we can finally find our linear Jacobians using the formula:

$$J_{v_i} = \overrightarrow{z_i} \times \overrightarrow{r_i}$$

Substituting in our r and z vectors, we find:

$$
\begin{aligned}
J_{v_{0.5}} &= \overrightarrow{z_{0.5}} \times \overrightarrow{r_{0.5}} & J_{\omega_{0.5}} &= \overrightarrow{z_{0.5}} \\
J_{v_1} &= \overrightarrow{z_1} \times \overrightarrow{r_1} & J_{\omega_1} &= \overrightarrow{z_1} \\
J_{v_2} &= \overrightarrow{z_2} \times \overrightarrow{r_2} & J_{\omega_2} &= \overrightarrow{z_2} \\
J_{v_3} &= \overrightarrow{z_3} \times \overrightarrow{r_3} & J_{\omega_3} &= \overrightarrow{z_3} \\
J_{v_{3.5}} &= \overrightarrow{z_{3.5}} \times \overrightarrow{r_{3.5}} & J_{\omega_{3.5}} &= \overrightarrow{z_{3.5}} \\
J_{v_4} &= \overrightarrow{z_4} \times \overrightarrow{r_4} & J_{\omega_4} &= \overrightarrow{z_4} \\
J_{v_5} &= \overrightarrow{z_5} \times \overrightarrow{r_5} & J_{\omega_5} &= \overrightarrow{z_5}
\end{aligned}
$$

For our final Jacobian Matrix to be:

$$
J = \begin{bmatrix} J_{v_{0.5}} & J_{v_1} & J_{v_2} & J_{v_3} & J_{v_{3.5}} & J_{v_4} & J_{v_5} \\ J_{\omega_{0.5}} & J_{\omega_1} & J_{\omega_2} & J_{\omega_3} & J_{\omega_{3.5}} & J_{\omega_4} & J_{\omega_5} \end{bmatrix}
$$

Now that we have our Jacobian Matrix, we can calculate the inverse Jacobian! The inverse Jacobian allows us to define a desired end effector velocity and calculate the appropriate joint velocities to achieve the end effector velocity.

To derive the inverse Jacobian, we can apply some basic linear algebra principles. Since our Jacobian has seven columns and six rows, we cannot find the exact inverse, but we can find the best approximation using the formula:

$$
J^+ = J^T \left(J \, J^T \right)^{-1}
$$

where J^+ is the "pseudoinverse" of J, providing the closest possible approximation of our inverse Jacobian. Now that we have our inverse Jacobian, we can simply plug in our desired velocity vector to calculate our needed individual joint velocities.

$$
\dot{\theta} = J^+ \cdot \overrightarrow{v}
$$

The last step for this section is to test our results using the visualizer. To do this, we can make our calculations directly in python and visualize them accordingly.

Following the method outlined above, we will first calculate the Jacobian by finding the z and r vectors for each reference frame. To get our z vectors, we take the rotation portion of the T_cumulative matrix and multiply it by the k unit vector. To get the r value, we simply subtract the current frames position with the end-effector position. In python, we combine the "intermediate" frames with their parent frames.

Now that we have our Jacobian, we need to find its inverse and multiply it to the end-effector desired velocity. Again, because the Jacobian in this case is not a square matrix, it's not possible to find its actual inverse. We use the following formulas to find the closest possible estimates for joint velocites:

$$
J^+ = J^T \left(J \, J^T \right)^{-1}; \qquad \dot{\theta} = J^+ \cdot \overrightarrow{v}
$$

The resulting code from this method is shown in **Figure 5**.

```
for i in range(4):
    k = np.array([0, 0, 1])
    rot_matrix = self.T_cumulative[i][:3, :3]
    z = rot_matrix @ k
    r = (self.points[5] - self.points[i])[:3]
    J.append(np.cross(z, r))

jacobian = np.transpose(np.array([J[0], J[1], J[2], J[3], [0, 0, 0]]))
new_jacobian = np.transpose(jacobian) @ np.linalg.inv(
    jacobian @ np.transpose(jacobian)
)

theta_dot = new_jacobian @ vel
theta_dot = theta_dot / 10 / (np.max(np.absolute(theta_dot)) + 1)
self.theta = self.theta + theta_dot
```

Figure 5: Code to find Jacobian Matrix, inverse Jacobian Matrix, and joint velocities.

An important part we want to point to is the normalization we applied to theta_dot on the second to last line shown above. When the joint values are too close to singularity, we get theta_dot values that are unnaturally high. This will cause our robot to rotate much quicker than it is capable of. To combat this, we divided the theta_dot values by the biggest theta_dot value among them, essentially capping the maximum possible angular change. You'll also notice we added the biggest theta_dot value by 1. This ensures that even when the biggest theta_dot value happens to be zero, our code will not crash.

The implementation of Forward Velocity Kinematics in the 5-dof robot can be found in the Hiwonder Github at scripts/hiwonder.py from line 101 to line 193. The code is basically identical to the arm-kinematics-module; we only changed the values which we scaled theta_dot by. Our implementation of the code can be found at our Github.

Our very last step for this section is to test the code in the visualizer. Luckily, it worked easily with some minor debugging. A video of our visualizer with FVK/RRMC can be found here.

Problem 3: Real Life Robots

For the final part of this project, our goal is to implement the resolved-rate motion control (RRMC) we derived in section 2 through gamepad control of the 5DOF robot hardware.

We had a few hiccups in this section, including but not limited to:

1. Robot Hardware being incompatible
2. Needing to re-frame our DH tables to match the orientation of the motors
3. Minor coding difficulties regarding syntax and variable names
4. Improper scaling of negative velocities
5. Controller drift making the robot difficult to control

However, after persevering through these issues, we were eventually able to control the robot arm seamlessly. A video demonstration of us practicing with the robot arm can be found linked here.

Individual Reflections

To cap everything off, each of us wrote individual reflections based on the following questions:

1. **What did you learn from this?**
2. **What did you not know before this assignment?**
3. **What was the most difficult aspect of the assignment?**
4. **What was the easiest or most straightforward aspect of the assignment?**
5. **How long did this assignment take?**
6. **What took the most time (PC setup? Coding in Python? Exploring the questions?)?**
7. **What did you learn about arm forward kinematics that we didn't explicitly cover in class?**
8. **What more would you like to learn about arm forward kinematics?**

Each reflection has it's own page dedicated to it below.

Bill's Reflection

The two main things I learned from this assignment were (1) the Denavit–Hartenberg convention and (2) applying the Jacobian matrix to do forward velocity kinematics. I had some prior knowledge of reference frames from QEA 2. But the class did not teach any systematic approaches to relate multiple frames together. In this assignment, we learned about the DH tables which allowed us to relate reference frames in a more structured and efficient way. With regards to the Jacobian matrix, prior to this assignment, I would not be able to explain how to calculate the desired change in angle of each joint given the desired change in end-effector velocity. Now, I can. I also understand singularities and why it is important to avoid it. We've also applied in our code a very simple normalization method to stop the theta_dot values from reaching quantities that are too large.

The most difficult aspect of this project was understanding and implementing the Forward Velocity Kinematics. The problem for me was keeping track of all the variables. It was easy to mix up between different vectors such as z, k, and r. It was also initially unclear how we were supposed to calculate r (the distance between the current point and the end-effector).

The easiest part of the assignment was implementing the Forward Position Kinematics into code. This was because a part of the implementation was already given to us with the T_cumulative variable and for loops. Our only job was to simply plug in the DH parameters and create the transformation matrix.

I spent a total of about eight to ten hours on the assignment. Setting up the DH tables and doing the mathematics took the majority of the time of about four hours. Another two hours was spent on doing the implementation of the mathematics. We had trouble with the robot (uncorrect gamepad mapping and arm not moving) which took us another one to two hours to debug. Last but not least, an hour was spent making sure we had all the links to the github and videos, and writing up the report/individual reflection.

The only thing I wished we covered more in class are strategies to combat singularities. Currently, with the normalization method, it only stops the robot from going crazy. However, once the robot is stuck in the singularity, there's nothing we can do to get it out of it other than telling it to go home. Of course, we can come up with our own solution. But, I would love to know what the industry standards are to deal with this problem since it seems to be a pretty common occurrence.

Satchel's Reflection

The three main things I learned in this mini-project were D-H convention and the concept of singularities, as well as working code implementation. I actually had worked with Jacobians before in a Special Topics in Mathematics class, so I understood them fairly well before this project. However, the idea of connecting a lot of these math concepts to actual robot implementation was new to me, and I struggled a lot with understanding how D-H convention was useful. I didn't realize how important aligning all three axes were, and especially struggled before we introduced the .5 frames to align them in a single transformation. Singularities also were a confusing topic - mathematically it made sense why it would occur as the denominator approaches zero, but I had never worked with them in a physical setting before. Safeguarding our code from singularities was a really fun and creative process, although I did wish we could implement a more robust solution, as our robot needs to be reset after encountering a near-singularity. Finally, this project gave me good experience with Python, which was really needed. As a MechE, I didn't have a ton of experience with it beforehand, so there was quite a learning curve, but Bill was able to really help me identify where the code and math aligned, and I actually enjoyed it once I got the hang of it.

I did not really know anything about robotics convention before this class, and learning both D-H convention and Forward Kinematics were really interesting. Previously I had a lot of experience using multi-dimensional functions to simulate motion, but I feel like in FunRobo it's much more about trying to establish some 'controlled motion' - a lot of the new math in this course was all anchored around trying to get the robotic arm into a position that was easily manipulatable by us.

The most difficult part of this assignment was deriving the D-H tables. My whole team spent a class deriving them, found out we were wrong, individually spent time making them, came together again, realized a few days later we were wrong again, and then spent another whole class block making sure we had them completed. The nuances of D-H tables, especially in finding more complex parts like the r vector or rotation matrix for multi-linked arms, required an understanding both the convention and why it's useful that we developed throughout the project.

The easiest or most straightforward part of this assignment to me was using the length and angle of every frame to compute the Jacobians. As I think I mentioned above, I have had a lot of experience with Jacobians in other classes, and mapping it into matlab actually helped me feel more secure in my knowledge of Funrobo.

This assignment took around 10 hours outside of class. I spent around 4 hours deriving the D-H tables and associated math, around 2 hours debugging and implementing in Python, and around 4 hours writing the report. The longest part of this assignment was really just conceptualizing it until the point I could explain it to the person next to me, and really understanding not just how we compute everything but why we do it as well.

Outside of class, I learned a lot about using normalization to shut down the robot arm when it approaches a singularity. I also learned about increasing the acceptable velocity threshold to combat controller drift, which were both really novel and creative ideas to combat singularities. I would really like to learn more about how we can use forward kinematics for arms with multiple end effectors, instead of just one. Or possibly how we could compute a multi-armed robot's joints together, in order to coordinate its end-effectors.

Sam's Reflection

I learned a lot from this project! The three main things I learned were (1) the true way to make DH tables, why they are useful, and how to convert them into H matrices. (2) How to convert DH tables into Jacobians, and how to use the inverse jacobian to find joint velocities. (3) How to implement code in python that allows us to control the robot.

Before this assignment, I had a loose understanding of all three of these ideas. However, I had little experience implementing them, and understood them more mathematically than conceptually. Doing this project built upon my previous knowledge and allowed me to fully understand how each of these concepts worked.

For me, the most difficult part of the assignment was understanding the DH tables. I derived the DH matrix on my own, then as a team, then as a team again, and then on my own again before I finally got the correct table and actually understood the full implementation. Notably, the write-up for the assignment also was challenging, moreso because of time.

The easiest part of the assignment for me was understanding the reference frames. I had plenty of knowledge from QEA3 and high school physics. Although I had some original confusions about certain orientations, I was able to understand it after a short conversation with Kene.

I spent all of the given class time working on the project, as well as about 10-15 hours outside of class working. The derivation of the matrices and the writeup took the longest time.

Outside of class, I learned about how to normalize matrices in order to ensure proper arm kinematics. I spent a bit of time studying singularities and other creative ways to avoid them for other types of robots, such as: setting joint limits, using smaller angle movements, or slowing the robot proportionally as it nears a singularity. I would love to learn more about different joints and how to calculate their motions.

Appendix

Below are some code snippets for the functions we implemented. For a full description, please check our github linked in the project submission.

```
# Precompute cumulative transformations to avoid redundant calculations
self.calc_DH_matrices()
self.T_cumulative = [np.eye(4)]
for i in range(self.num_dof):
    self.T_cumulative.append(self.T_cumulative[-1] @ self.DH[i])
```

Figure 6: Calculating T_cumulative.

```
# Calculate the robot points by applying the cumulative transformations
for i in range(1, 6):
    self.points[i] = self.T_cumulative[i] @ self.points[0]
```

Figure 7: Calculating points from T_cumulative.

```
# Link lengths
self.l1, self.l2, self.l3, self.l4, self.l5 = 0.155, 0.099, 0.095, 0.055, 0.105
```

Figure 8: Link Lengths

```
def _handle_event(self, event):
    """Handles individual gamepad events and updates internal state."""
    code_map = {
        "ABS_X": ("abs_x", event.state),
        "ABS_Y": ("abs_y", event.state),
        "ABS_RY": ("abs_z", event.state),
        # "ABS_RZ": ("abs_rz", event.state),
        #'BTN_WEST': ('MOBILE_BASE_FLAG', bool(event.state)),
        "BTN_TR": ("ARM_FLAG", bool(event.state)),
        "BTN_WEST": ("ARM_J1_FLAG", bool(event.state)),
        "BTN_EAST": ("ARM_J2_FLAG", bool(event.state)),
        "BTN_SOUTH": ("ARM_J3_FLAG", bool(event.state)),
        "BTN_NORTH": ("ARM_J4_FLAG", bool(event.state)),
        "BTN_START": ("ARM_J5_FLAG", bool(event.state)),
        "BTN_TL": ("ARM_EE_FLAG", bool(event.state)),
        "BTN_SELECT": ("ARM_HOME", bool(event.state)),
    }

    if event.code in code_map:
        setattr(self, code_map[event.code][0], code_map[event.code][1])
```

Figure 9: Fixed gamepad mapping

```
self.theta = [radians(i) for i in self.joint_values]
```

Figure 10: Turning theta from degrees to radians

```python
def set_arm_velocity(self, cmd: ut.GamepadCmds):
        """Calculates and sets new joint angles from linear velocities.

        Args:
            cmd (GamepadCmds): Contains linear velocities for the arm.
        """
        vel = [cmd.arm_vx, cmd.arm_vy, cmd.arm_vz]

        ########################################################################

        self.theta = [radians(i) for i in self.joint_values]

        self.calc_DH_matrices()
        T_cumulative = [np.eye(4)]
        for i in range(5):
            T_cumulative.append(T_cumulative[-1] @ self.DH[i])

        J = []

        points = [
            np.array([0, 0, 0, 1]),
            np.array([0, 0, 0, 1]),
            np.array([0, 0, 0, 1]),
            np.array([0, 0, 0, 1]),
            np.array([0, 0, 0, 1]),
            np.array([0, 0, 0, 1]),
        ]
        for i in range(1, 6):
            points[i] = T_cumulative[i] @ points[0]

        for i in range(4):
            k = np.array([0, 0, 1])
            rot_matrix = T_cumulative[i][:3, :3]
            z = rot_matrix @ k
            r = (points[5] - points[i])[:3]
            J.append(np.cross(z, r))

        jacobian = np.transpose(np.array([J[0], J[1], J[2], J[3], [0, 0, 0]]))
        new_jacobian = np.transpose(jacobian) @ np.linalg.inv(
            jacobian @ np.transpose(jacobian)
        )

        # get thetalist_dot
        thetalist_dot = new_jacobian @ vel
        # normalize thetalist_dot and scale it by a factor
        thetalist_dot = thetalist_dot / (np.max(np.absolute(thetalist_dot)) + 0.01) / 6
        # turn thetalist_dot into degrees
        thetalist_dot = [degrees(theta) for theta in thetalist_dot]
```

Figure 11: Full implementation of hiwonder forward velocity kinematics