Fun Robo Mini Project 2

Sam Wisnoski, Bill Le, Satchel Schiavo

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

0. Introduction

In this project we conceptualized and implemented the inverse kinematics of a 5 DOF robot manipulator. We learned about various methods to solve for each individual arm angle to result in a specific end-effector behavior, before implementing two solutions in the Viz tool: One analytical, one numerical. We programmed two solutions and tested both with various different end-effector locations to verify the two solutions. Below you will see our process for solving the two methods, as well as our implementation in Python with code snippets, and finally videos of our viz tools functioning.

1. Derivation of the Analytical IK Equations

The goal of analytical inverse kinematics is to produce a closed-form solution that calculates the joint positions for the arm when given an end effector (EE) pose. This section will outline the methodology to calculate those joint positions for a 5-DOF arm system.

1.1 Calculating θ_1

In the 5-DOF arm system, we concluded that only θ_1 impacts the yaw (rotation around the z-axis in the origin frame) of the EE. We came to this conclusion because when looking at the arm from top-down view, we notice that all joints and the origin can be connected in a straight line, as shown in **Figure 1**.

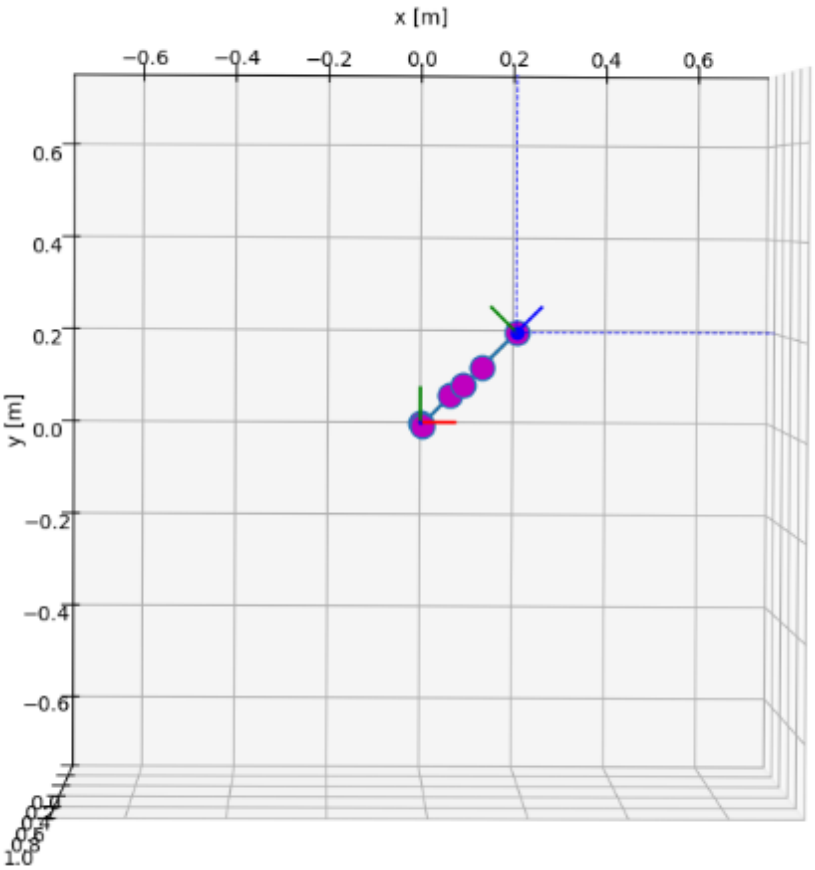

Figure 1: Top-down view of 5-DOF arm.

Using this fact, we can calculate for θ_1 using the following equation.

$$\theta_1 = \arctan(\frac{EE_y}{EE_x})$$

It is important to note that there are actually two solutions for θ_1 as the tangent function's period is π. Hence, the full answer for θ_1 is as follows.

$$\theta_1 = \arctan(\frac{EE_y}{EE_x}) \quad \text{or} \quad \theta_1 = \arctan(\frac{EE_y}{EE_x}) + \pi$$

In code, this is simply written with a single line:

```
def calc_inverse_kinematics(self, EE: EndEffector, soln=0):
    """
    Calculate inverse kinematics to determine the joint angles based on end-effector position.

    Args:
        EE: EndEffector object containing desired position and orientation.
        soln: Optional parameter for multiple solutions (not implemented).
    """
    ########################################

    # calculating theta_1
    self.theta[0] = atan2(EE.y, EE.x)
```

1.2 Calculating P_{wrist}

To calculate θ_2 and θ_2, we need to calculate the position of the wrist of the arm. In our 5-DOF case, the wrist is labeled below, in **Figure 2**.

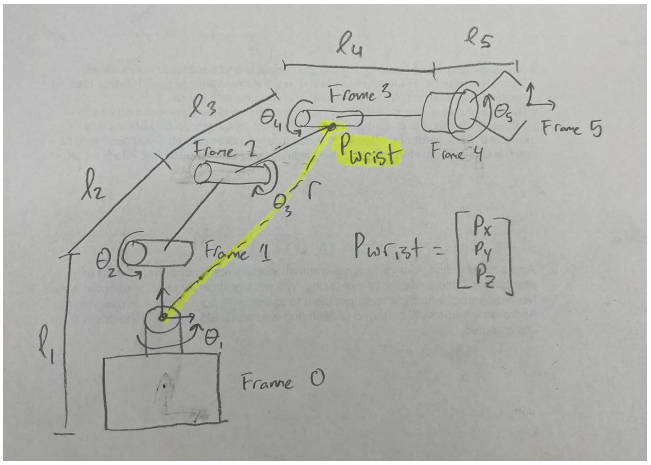

Figure 2: The six different frames of our robot with the wrist labeled.

Notice from **Figure 2** that the z-axis of frame 3 (wrist frame) and frame 5 (EE frame) lies on the same line (parallel and intersecting). Hence, we know that the displacement between the two frames is along the z-axis of frame 5 (and frame 3). We know that this displacement is $l5$. Therefore, we can calculate the position of frame 3 as follows.

$$P_{wrist} = P_{EE} - {}^0R_5\hat{k}$$

where ${}^0R_5$ is the rotation between the origin frame and EE frame and $\hat{k}$ is the unit vector representing the z-axis of the origin frame.

$$\hat{k} = [0, 0, 1]^T$$

Before we can complete this calculation, we need to figure out what ${}^0R_5\hat{k}$ is. For this project, we are going to use the **ZYX Euler Convention** to describe the rotation of the EE. Therefore, ${}^0R_5\hat{k}$ can be calculated as follows.

$${}^0R_5\hat{k} = R_z(EE_{rotz}) \cdot R_y(EE_{roty}) \cdot R_x(EE_{rotx})$$

We can see this implemented in python here:

```python
k = np.transpose(np.array([[0, 0, 1]]))
rotx = np.array(
    [
        [1, 0, 0],
        [0, cos(EE.rotx), -sin(EE.rotx)],
        [0, sin(EE.rotx), cos(EE.rotx)],
    ]
)
roty = np.array(
    [
        [cos(EE.roty), 0, sin(EE.roty)],
        [0, 1, 0],
        [-sin(EE.roty), 0, cos(EE.roty)],
    ]
)
rotz = np.array(
    [
        [cos(EE.rotz), -sin(EE.rotz), 0],
        [sin(EE.rotz), cos(EE.rotz), 0],
        [0, 0, 1],
    ]
)

# calculating r_06 using euler ZYX convention
r_06 = rotz @ roty @ rotx
t_35 = (self.l4 + self.l5) * r_06 @ k

# calculating p_wrist
p_wrist_x = EE.x - t_35[0]
p_wrist_y = EE.y - t_35[1]
p_wrist_z = EE.z - t_35[2]
```

1.3 Simplifying to the 2-DOF Case

Now, since we have calculated the wrist position, we can exclude frames 4 and 5, and solely analyze the arm from frame 0 to 3. First, we recognize that the link between frame 0 and frame 1 is stationary regardless of any changes in any joint position. Therefore, the system from frame 0 to 3 is essentially a 2-DOF case within a plane.

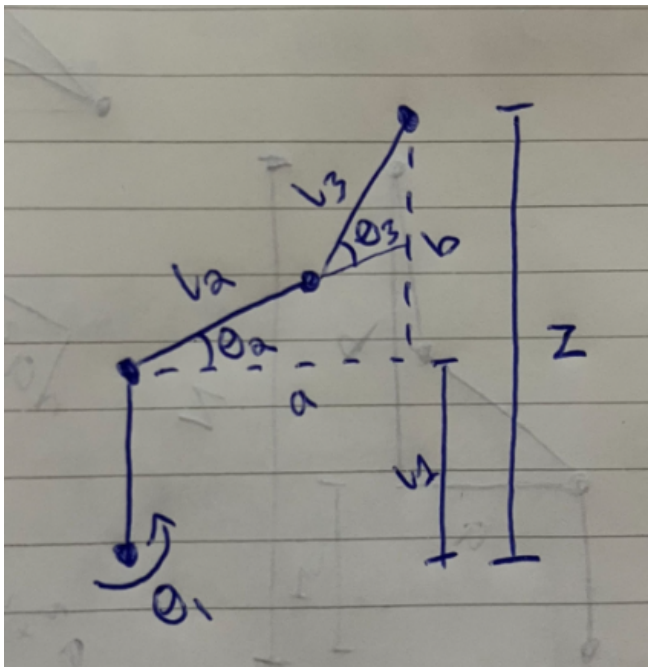

Figure 3: Simplifying the 5-DOF's arm frame 0 to frame 3 to a 2-DOF case.

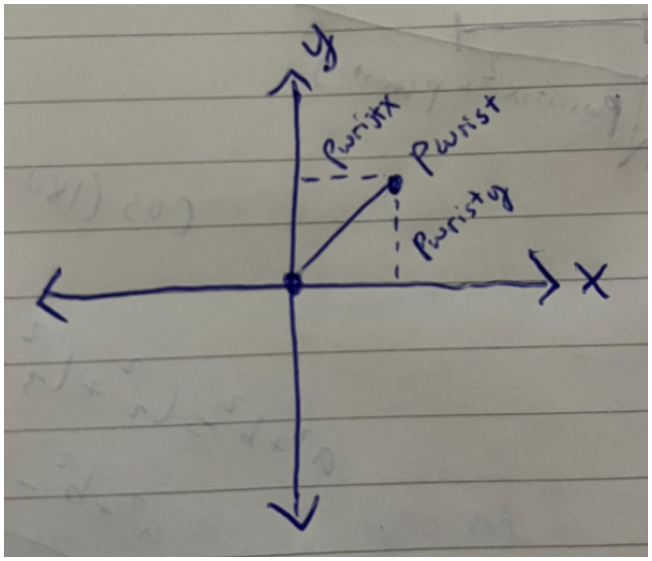

Figure 4: Top-down view of 2-DOF system.

We can find expressions for a and b (referenced in **Figure 3**) as follows.

$$a = \sqrt{(P_{wristx})^2 \cdot (P_{wristy})^2}$$
$$b = z - l_1$$

In python, we calculate a and b as rx and ry.

```
# calculating new x and y for 2-DOF solution
rx = sqrt(p_wrist_x**2 + p_wrist_y**2)
ry = p_wrist_z - self.l1
```

We can see these equation expressed in relation to the our frames in **Figure 5**.

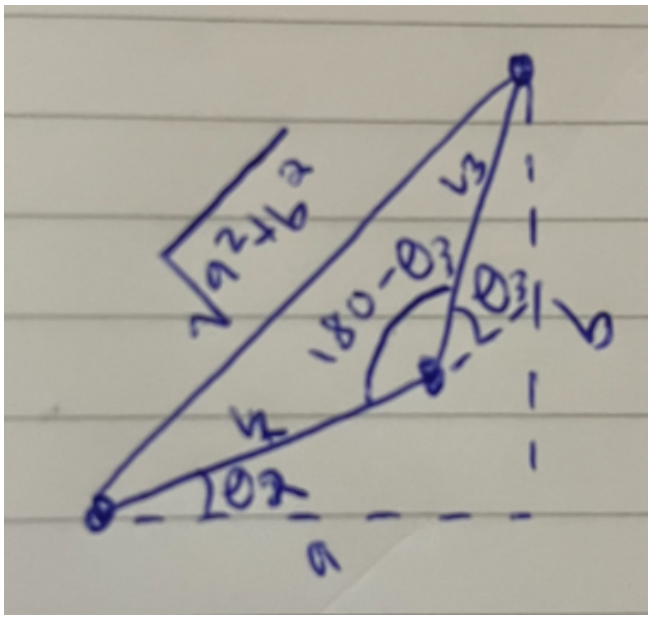

Figure 5: Break down of angles and distances required for 2-DOF analytical IK derivation of θ_2

We can use this information to derive θ_3 using a and b.

$$c^2 = a^2 + b^2 - ab\cos(\theta)$$
$$a^2 + b^2 = (1_2)^2 + (l_3)^2 - 2l_2l_3\cos(180 - \theta_3)$$
$$\cos(180 - \theta) = -\cos(\theta)$$
$$a^2 + b^2 = (1_2)^2 + (l_3)^2 + 2l_2l_3\cos(\theta_3)$$
$$\theta_3 = \pm\arccos(\tfrac{a^2+b^2-(l_2)^2-(l_3)^2}{2\cdot l_2\cdot l_3})$$

The final equation for θ_3 includes a plus/minus sign to account for the fact that the cosine function is mirrored around the y-axis. It physically correlates to the elbow up and elbow down scenarios.

Now, let's derive θ_2 using our newly derived θ_3. We use placeholder variables here to make our equation easier to read.

$$s = l_3 \cdot \cos(\theta_3)$$
$$r = l_3 \cdot \sin(\theta_3)$$
$$\alpha = \arctan(\tfrac{r}{l_2+s})$$
$$\gamma = \arctan(\tfrac{x}{y})\ \theta_2 = \gamma - \alpha$$

There is another solution for θ_2 which represents the elbow-up scenario as shown below.

$$\theta_2 = \gamma + \alpha$$

We solve for θ_2 and θ_3 in python as shown below:

```
# elbow down
if soln == 0:
    self.theta[2] = acos(
        (rx**2 + ry**2 - self.l2**2 - self.l3**2) / (2 * self.l2 * self.l3)
    )
# elbow up
else:
    self.theta[2] = -acos(
        (rx**2 + ry**2 - self.l2**2 - self.l3**2) / (2 * self.l2 * self.l3)
    )

# calculate theta 3
alpha = atan2(
    self.l2 * sin(self.theta[2]), self.l2 + self.l3 * cos(self.theta[2])
)
gamma = atan2(ry, rx)
self.theta[1] = (gamma - alpha) - (np.pi / 2)
self.theta[2] = -self.theta[2]
```

1.4 Using ${}^3R_5$ to solve for θ_4 and θ_4

Now with θ_1, θ_2, and θ_3, we will calculate the H matrix from frame 0 to frame 3 using DH parameters calculated in Mini-project 1.

${}^0H_3 = {}^0H_1\ {}^1H_2\ {}^2H_3$

From the resulting H matrix, we will take the first three rows and columns, which is the rotation matrix, ${}^0H_3$. From section Calculating P_{wrist}, we have also calculated ${}^0R_5$. We can use both of these information to calculate for ${}^3R_5$ as follows.

$${}^0R_5 = {}^0R_3\ {}^3R_5$$
$$({}^0R_3)^T\ {}^0R_5\ {}^3R_5$$

Note here that we were able to use $({}^0R_3)^T$ to represent the inverse because rotation matrices are orthogonal.

We also happen to know what each index of the ${}^3R_5$ matrix is due to the DH parameters. Their expansions are shown below.

$${}^3R_{3.5} = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 \\ \sin(\theta_4) & \cos(\theta_4) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$${}^{3.5}R_4 = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$${}^4R_5 = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 \\ \sin(\theta_5) & \cos(\theta_5) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$^3R_5 = {}^3R_{3.5} \cdot {}^{3.5}R_4 \cdot {}^4R_5 = \begin{bmatrix} \sin(\theta_4)\cos(\theta_5) & -\sin(\theta_4)\sin(\theta_5) & \cos(\theta_4) \\ -\cos(\theta_4)\cos(\theta_5) & \cos(\theta_4)\sin(\theta_5) & \sin(\theta_4) \\ -\sin(\theta_5) & -\cos(\theta_5) & 0 \end{bmatrix}$$

From this we can solve for θ_4 by doing the following:

$$\theta_4 = \arctan(\frac{^3R_5(2,3)}{^3R_5(1,3)}) \quad \text{or} \quad \theta_4 = \arctan(\frac{^3R_5(2,3)}{^3R_5(1,3)}) + \pi$$

We can solve for θ_5 by doing the following:

$$\theta_5 = \arctan(\frac{^3R_5(3,1)}{^3R_5(3,2)}) \quad \text{or} \quad \theta_5 = \arctan(\frac{^3R_5(3,1)}{^3R_5(3,2)}) + \pi$$

In python, we first need to calculate the H matrices, and then we can derive θ_4 and θ_5 and shown above:

```
self.calc_DH_matrices()
r_03 = (self.DH[0] @ self.DH[1] @ self.DH[2])[:3, :3]
r_35 = np.transpose(r_03) @ r_06

# calculate theta 4 and 5
self.theta[3] = atan2(r_35[1][2], r_35[0][2])
self.theta[4] = atan(r_35[2][0] / r_35[2][1])
```

1.5 Multiple Solutions

We can observe that there are multiple solutions for each θ_i. Here is a list of each θ_i and their possible solutions.

$$\theta_1 = \arctan(\frac{EE_y}{EE_x}) \quad \text{or} \quad \theta_1 = \arctan(\frac{EE_y}{EE_x}) + \pi$$

$$\theta_2 = \gamma - \alpha \quad \text{or} \quad \theta_2 = \gamma + \alpha$$

$$\theta_3 = \arccos(\frac{a^2+b^2-(l_2)^2-(l_3)^2}{2 \cdot l_2 \cdot l_3}) \quad \text{or} \quad \theta_3 = -\arccos(\frac{a^2+b^2-(l_2)^2-(l_3)^2}{2 \cdot l_2 \cdot l_3})$$

$$\theta_4 = \arctan(\frac{^3R_5(2,3)}{^3R_5(1,3)}) \quad \text{or} \quad \theta_4 = \arctan(\frac{^3R_5(2,3)}{^3R_5(1,3)}) + \pi$$

$$\theta_5 = \arctan(\frac{^3R_5(3,1)}{^3R_5(3,2)}) \quad \text{or} \quad \theta_5 = \arctan(\frac{^3R_5(3,1)}{^3R_5(3,2)}) + \pi$$

Not all of these solutions are valid. One of the clear problems that we have is solutions involving tangents. The reason why tangent produces multiple solutions is due to the nature of its inverse function. Even though the unit circle has a period of 2π, the inverse tangent function only has a period of π which leads it to miss possible solutions. One way to tackle this is using the **atan2** function provided in many languages. This function takes in the y and x values and will return the correct angle. Therefore, this reduces the amount of possible solutions down to only two: elbow up and elbow down.

1.6 Recalculating θ's to Match Servo Orientation

There are two things we have to account for with regards to our servo behavior for it to match the behavior of our 5-DOF arm. Firstly, the orientation of θ_3 is the opposite of that of θ_2 and θ_4. To account for this, we simply reverse its direction by doing the following.

$$\theta_3 = -\theta_3$$

Secondly, when θ_2 is set to zero, its x-axis actually points in the z-direction of the origin frame (shifted +90 degrees). Therefore, we need to shift our calculation backwards.

$$\theta_2 = \theta_2 - \frac{\pi}{2}$$

1.7 Strategies for Selecting Valid Solutions

Besides using **atan2** as mentioned in the previous sections, to select valid solutions, we employed two strategies: 1.) enforcing joint limits and 2.) FPK to check resultant EE with desired EE. Enforcing joint limits is rather trivial in terms of implementation so we won't discuss it here. With regards to FPK to check resultant versus desired EE pose, we simply took the difference between the two poses, summed them up, and checked if they were below a certain threshold.

These two strategies can be shown in python below:

```python
# enforce joint limits
if (
    self.theta[0] < (-2 * np.pi / 3)
    or self.theta[0] > (2 * np.pi / 3)
    or self.theta[1] < (-np.pi / 2)
    or self.theta[1] > (np.pi / 2)
    or self.theta[2] < (-2 * np.pi / 3)
    or self.theta[2] > (2 * np.pi / 3)
    or self.theta[3] < (-5 * np.pi / 9)
    or self.theta[3] > (5 * np.pi / 9)
    or self.theta[4] < (-np.pi / 2)
    or self.theta[4] > (np.pi / 2)
):
    print([degrees(i) for i in self.theta])
    self.theta = [0, 0, 0, 0, 0]
    print("HERE")

# check that it is giving the right EE location
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])

estimated_EE = self.T_cumulative[5] @ np.array([0, 0, 0, 1])
diff = estimated_EE - np.array([EE.x, EE.y, EE.z, 1])
if np.linalg.norm(diff) > 0.05:
    print("HERE!")
    self.theta = [0, 0, 0, 0, 0]
########################################

self.calc_robot_points()
```

1.8 Visualizer Screenshot Video - Analytical

The last item for this section was is to test our Analytical solution using the python visualizer. We successfully demonstrated the ability of our solution by presenting four different poses and two solutions for each pose. Additionally, we present an invalid solution due to joint limits.

The video can be found at: https://www.youtube.com/watch?v=dgRZzHpuan0.

2. Numerical Inverse Kinematics

2.1 Jacobian and Inverse Jacobian Matrix Computation

In Mini-Project 1, we went into excruciating detail about how to compute the Jacobian and inverse Jacobian matrices. If you are interested in reading more in detail about the computation of Jacobian's, please read our Mini-Project 1 report here: https://acrobat.adobe.com/id/urn:aaid:sc:VA6C2:d360267f-5120-40b5-ac69-63d3e0b898a1.

In this report, we wanted to focus more on our Numerical solution to Inverse Kinematics and how Jacobians were implemented. However, as required by the rubric, we will first give a brief overview of how to calculate a Jacobian matrix.

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

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

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:

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

The above z vectors give us the angular velocities, but in order to calculate the full Jacobian, we also need the linear velocities, which require the r vectors. The r vectors are 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.

As such, to calculate the value for $\overrightarrow{r_i}$, we need:

$$\overrightarrow{r_i} = \overrightarrow{d_5^0} - \overrightarrow{d_{i-1}^0}$$

Lastly, we can finally find our linear Jacobians using the formula:

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

All of the above equations can be squeezed into a few lines of code, given that we already have functions for calculating the DH matrices.

```python
def calc_pseduojacobian(self):
    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])
    J_l = []
    J_w = []

    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_w.append(z)
        J_l.append(np.cross(z, r))

    # Construct the Jacobian matrix
    J_l.append([0, 0, 0])  # Adding a zero row for completeness
    J_w.append([0, 0, 0])  # Adding a zero row for completeness

    jacobian_l = np.transpose(np.array(J_l))
    jacobian_w = np.transpose(np.array(J_w))
    jacobian = np.concatenate((jacobian_l, jacobian_w), axis=0)
```

As seen above, we create a list of H matrices using our DH table function. Then, we use the rotation matrix to find the linear velocity and the entire matrix to find the linear velocity. We then iterate over different frames to compose our entire Jacobian. Lastly, we add the linear to our angular velocity. This is important since our numerical solver needs to solve for both position and orientation.

Now that we have our Jacobian, we can derive the inverse Jacobian by applying some basic principles of linear algebra. 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.

This, as well, can be done with a few lines of code, as seen below.

```
# Calculate pseudo-Jacobian matrix using the transpose method
if np.linalg.matrix_rank(jacobian) < min(jacobian.shape):
    # Handle near singularity or underdetermined system
    print(
        "Warning: Jacobian matrix is not full rank. Adjusting for pseudo-inverse calculation."
    )
    # Use a regularized pseudo-inverse calculation
    new_jacobian = np.transpose(jacobian) @ np.linalg.inv(
        jacobian @ np.transpose(jacobian) + np.eye(jacobian.shape[0]) * 1e-5
    )
else:
    # Calculate pseudo-Jacobian matrix using the transpose method
    new_jacobian = np.transpose(jacobian) @ np.linalg.inv(
        jacobian @ np.transpose(jacobian)
    )

return new_jacobian
```

First, we check if the Jacobian is full rank. If the Jacobian is near a singularity or is not invertible, it will add a small term to help normalize the Jacobian. In our case, the Jacobian will always add this term since our regular Jacobian is non-invertible. This helps increase the accuracy of the pseduo-Jacobian as well as the numerical solver.

Now that we understand how to calculate the Jacobian, we can derive a numerical solution to the Inverse Kinematics Problem.

2.2 Solving Numerical IK with the Newton-Raphson Method

Now that we have our Jacobian Matrix, we can work on solving the Numerical Inverse Kinematics. The Newton-Raphson Method is an iterative method, where we take an initial guess, compute error, move in the desired direction, and then repeat until our error is very small. The exact steps of the Newton-Raphson Method are as follows:

1. Take an initial guess. In our case, our initial guess will be the current position of our EE.

2. Calculate the error by subtracting the desired distance and orientation from the current distance and orientation (calculated via Forward Kinematics).

3. If the error is less than our tolerance, ϵ, then we have successfully reached our desired position. We can return our thetas and move to our calculated position.

4. If the error is greater than our tolerance, then we add our error multiplied by the Jacobian to our current thetas. This will move us closer to our desired position. Then, return to step two and repeat.

When we implement this to work in python, we use the class variables self.theta and ee.x, ee.y, etc to repeatedly make these iterations. In each loop, we recalculate the Jacobian based on the new thetas, apply it to the error, and add it to our existing thetas. Then, we recalculate the error, and check if it's within tolerance using a while loop. Once we are within tolerance or have iterated 50 times, we move to our new position by calling self.calc_forward_kinematics().

Below is an implementation of this method in code:

```python
def calc_numerical_ik(self, EE: EndEffector, tol=0.01, ilimit=50):
    """Calculate numerical inverse kinematics based on input coordinates."""

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

    # define initial guess
    original_points = [
        self.ee.x,
        self.ee.y,
        self.ee.z,
        self.ee.rotx,
        self.ee.roty,
        self.ee.rotz,
    ]
    epsilon = 0.001
    desired_coords = [EE.x, EE.y, EE.z, EE.rotx, EE.roty, EE.rotz]
    points = original_points

    # calc initial variable
    error_var = [
        des_i - theta_i for des_i, theta_i in zip(desired_coords, points)
    ]  # 1x6 matrix
    i = 0
    while (sum(np.absolute(error_var)) / 6 >= epsilon) & (i < 3000):
        pseudojacobian = self.calc_pseduojacobian()  # 6x5 matrix
        new_points = pseudojacobian @ np.transpose(np.array(error_var))
        self.theta = self.theta + new_points  # 1x5 matrix \
        self.calc_robot_points()
        points = [
            self.ee.x,
            self.ee.y,
            self.ee.z,
            self.ee.rotx,
            self.ee.roty,
            self.ee.rotz,
        ]
        error_var = [
            des_i - theta_i for des_i, theta_i in zip(desired_coords, points)
        ]
        i += 1

    [EE.x, EE.y, EE.z, EE.rotx, EE.roty, EE.rotz] = original_points
    ########################################

    # Recompute robot points based on updated joint angles
    self.calc_forward_kinematics(self.theta, radians=True)
```

We ran into a decent amount of trouble with implementing this, and had to debug significantly, including rewriting our pseudo-Jacobian function, but eventually got it up and running!

2.3 Visualizer screenshot video - Numerical

After implementing the numerical inverse kinematics, we used the visualization tool to test its performance in comparison to the analytical IK. We tested a variety of positions in the video, including a scenario where our numerical IK runs into a local minima. The video can be found at: https://www.youtube.com/watch?v=vMdYZhNd9Ww.

In the video, there are two errors. First, since the visualizer does not place limits on joints, some of the joint angles are calculated to be ridiculously high. While the numerical method does compute the correct solution, it fails to properly normalize the joint angles. This could be fixed with a small change to the visualizer or our code, such as the limit enforcer code shown in section 1.7.

Secondly, the Newton-Raphson method can occasionally lock itself into a local minima. The Newton-Raphson method is a very similar method to gradient descent, attempting to minimze the error. In a 'local minima', moving in any direction will raise error, even if the EE is not at the correct destination. To combat this problem, we add a limit of 50 iterations to prevent the method from cycling infinitely. To further combat this, we could use way points between the start and end position, reducing the distance traveled between calculations. This is more likely to occur the further the initial guess is from the desired location, and in testing, happens roughly once every ten runs.

3. Inverse kinematics Observation and Comparison

Now that we have completed both analytical and numerical solutions, we can compare both.

In general, both solutions were effective calculating the inverse kinematics. However, the analytical solution was both faster (shorter processing time) and more reliable. The numerical model has a chance of running into local minima, had varying processing time based on number of iterations, and would give a best guess if it was fed a false solution. In contrast, you would be able to tell instantly if the analytical solution was given a "false" value, and always found the exact point.

The downside of the analytical solution is that it must be calculated exactly for each robot, whereas the numerical solution is a more generally usable theorem that can be used for a large variety of robo-appendages. As such, neither the analytical or numerical solution is **better**, but they both have different pros and cons.

4. Individual reflections

Individual reflections on the following questions:

- What did you learn from this? What did you not know before this assignment?
- What was the most difficult aspect of the assignment?
- What was the easiest or most straightforward aspect of the assignment?
- How long did this assignment take? What took the most time (PC setup? Coding in Python? Exploring the questions?)?
- What did you learn about arm inverse kinematics that we didn't explicitly cover in class?
- What more would you like to learn about arm inverse kinematics?

Bill's Reflection:

From this mini-project, I learned how to derive the analytical inverse kinematics for the 5-DOF arm. I got practice calculating the wrist's position, which we then used to calculate θ_2 and θ_3 by applying the solution of the 2-DOF arm learned in class. We then used the knowledge we learned about rotation matrices in mini-project 1 to eventually solve for θ_4 and θ_5 . I did not know any of this before this assignment. We also learned how to numerically solve the inverse kinematics for the arm (which could be applied to arms of any degree and configuration). Mathematically, it was less rigorous than the analytical solution. Something super useful I learned that I did not know before was the strengths and weaknesses of the analytical versus numerical solution. It puts into context how and when each should be used.

The most difficult aspect of the assignment for me was understanding and implementing the whole of the analytical solution. The steps involved to reach the solution are rather long. Therefore, when an error occurred in code, it was hard to figure out which part was implemented incorrectly.

The easiest part of the assignment was implementing the 2-DOF arm solution to solve for θ_2 and θ_3 . This was mainly due to the fact that we went over this before in class together through the simpler 2-DOF arm example.

The assignment took me roughly twelve hours. I spent three hours deriving and understanding the analytical solution. Then, I spent another two hours implementing it in the visualizer. The next five hours was spent implementing the code on hardware. We ran into a lot of issues regarding remapping the controls and getting the code to run correctly. Because we had to push our code and pull it on the raspberry pi every time we made a change, testing took up a lot more time than we expected. When we finally got the code to work, we realized that there are inconsistencies between the visualizer and the physical reality. We spent some more time trying to figure out how to fix this. Due to time constraints, we stopped short. The last two hours were spent writing the report. The class covered most of what we needed to know to implement inverse kinematics. The one thing I had to figure out on my own was how to deal with the fact that the servo motors' rotation would switch every other motor. However, Kene did touch upon this in one of the classes.

The thing that I'm most interested in right now is to figure out how to make the visualizer tool and the physical reality of the robot more consistent. I would imagine that for an application like this, it is essential that they produce very similar results. Else, there is really no point in testing on the visualizer tool at all.

Sam's Reflection:

In this mini-project (and the lessons leading up to it), I learned how to derive and implement the analytical and numerical solutions for the 5-DOF robot arm we have been using in class.

Pretty much everything I learned from this project was something I didn't know. While I am familiar with the principles that these methods use (general trigonometry, linear algebra, gradient descent), I have never used the principles in the space of robotics or to do something at this level of complexity before. Additionally, from mini-project 1, I was already familiar with DH tables, Jacobian/Inverse Jacobian matrices, and other principles we had built these methods up from.

The most difficult aspect of this assignment was the conversion from the viz tool to the hardware. While the derivation of the solutions was difficult, I found it more difficult to rewrite the entire framework of our functions to fit with the hardware. Additionally, once we had the hardware working, we still ran into issues with calibration and connection. This was frustrating more than anything, as it was hard to tell where to even begin looking for errors.

The most straightforward part of this assignment was understanding the numerical solution. Satchel and I had written out functioning pseudocode in a matter of minutes, and the much more difficult part was implementing the known solution into code.

I spent about 8 hours outside of class working on this project, and all the allotted class time working on this project. The majority of this time (probably about 5 hours) was spent writing the report, while the other three were spent messing with the hardware. The most time consuming part was again the report, followed closely by attempting to integrate with hardware.

I learned more about different ways to increase the accuracy of the numerical Newton-Raphson method, such as adding waypoints. I also did brief research into other ways of calculating the numerical solution.

I would like to learn more about different joints and how to calculate their motions (such as ball joint).

Satchel's Reflection:

I learned a lot about conceptualizing the relationship between a given end-effector position and the arm angles, as well as coding methods to solve. I didn't know a whole lot about using analytical methods to solve for an equation as complex as a 5DOF arm, but I learned during this assignment!

The most difficult aspect of the assignment was bug-fixing for the analytical methods of solving for IK thetas. Specifically, understanding how to bug fix for IK thetas - and understanding that there was some tolerancing for different positions. Initially we thought our Viz tool was bugged as we would test super similar points only to find no variation in theta angles. However, once we checked in with CAs and found a better way to test through checking between two known end-effector positions, we were able to quickly debug and complete.

Although it wasn't the quickest aspect of this assignment, the most straightforward part was deriving the inverse kinematics numerically. The implementation in the Python Viz tool took a lot longer than I would've liked, but actually creating an algorithm off of the scaffolding provided in class was pretty simple. It was actually really cool to reason out the different steps in numerically solving for theta.

This assignment took about 10 hours provided in class and about 2-3 hours outside of class. Although the initial implementation of numerical IK solver was difficult to conceptualize and took a few class periods of trial and error to understand, the longest part was the hardware implementation. It took multiple classes to set up and took longer due to the confusing errors we encountered, and having less reference for errors unlike the viz tool, where we could program in checks to see what part of the code wasn't working. We also were not sure how much of the error was outside of our control and a hardware issue, and as such struggled.

What did you learn about arm inverse kinematics that we didn't explicitly cover in class?

One thing I learned about arm inverse kinematics that we didn't explicitly cover in class is conceptualizing that IK cannot compute super specific points and are limited by joint angles. I discovered this when attempting to debug the Viz tool by testing between two different end-effector poses that were ever so slightly off (still greater than the tolerance) and found that it didn't find different angles.

Using different numerical methods to calculate arm inverse kinematics definitely seemed like an interesting extension of the class. I would be really interested in seeing what the industry standard for IK calculations is, as well as comparing different methods to find a solution (Newton's method, Secant, etc). How many practical applications actually use analytical methods to solve, if any?