Forward Kinematics

Introduction

This lab uses the dimensions of the UR3 robot used in lab and calculates the position of the end effector using forward kinematics concepts. Forward kinematics is the process of finding the position and velocity of the end effector of the robot, given the dimensions of each joint and their axes of rotation, all relative to the body frame (in this case). Some of the skills learned during this lab include learning to use different packages in python to find the symbolic solution of the kinematics equations.

Method

Before any programming can begin, the M matrix, ω_1 through ω_6, and q_1 through q_6 needed to be calculated. To start building the M matrix, Figure 1 can be used to calculate the rotation matrix from the body frame to the frame of the end effector. The more tedious part of building the M matrix is calculating the translation from the origin of the body frame to the origin of the end effector frame. Figure 2 was used to do these calculations in millimeters, and then converted to meters in the python code. After doing all this, the M matrix created is

$$\begin{bmatrix} 0 & -1 & 0 & .390 \\ 0 & 0 & -1 & .401 \\ 1 & 0 & 0 & .2155 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

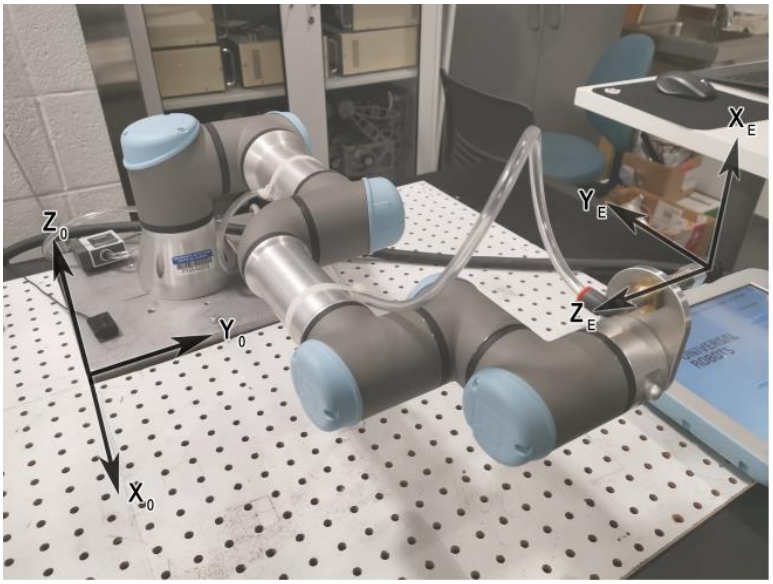

Figure 1: Zero configuration of the UR3, with the coordinate frames of the base and the end effector labeled.

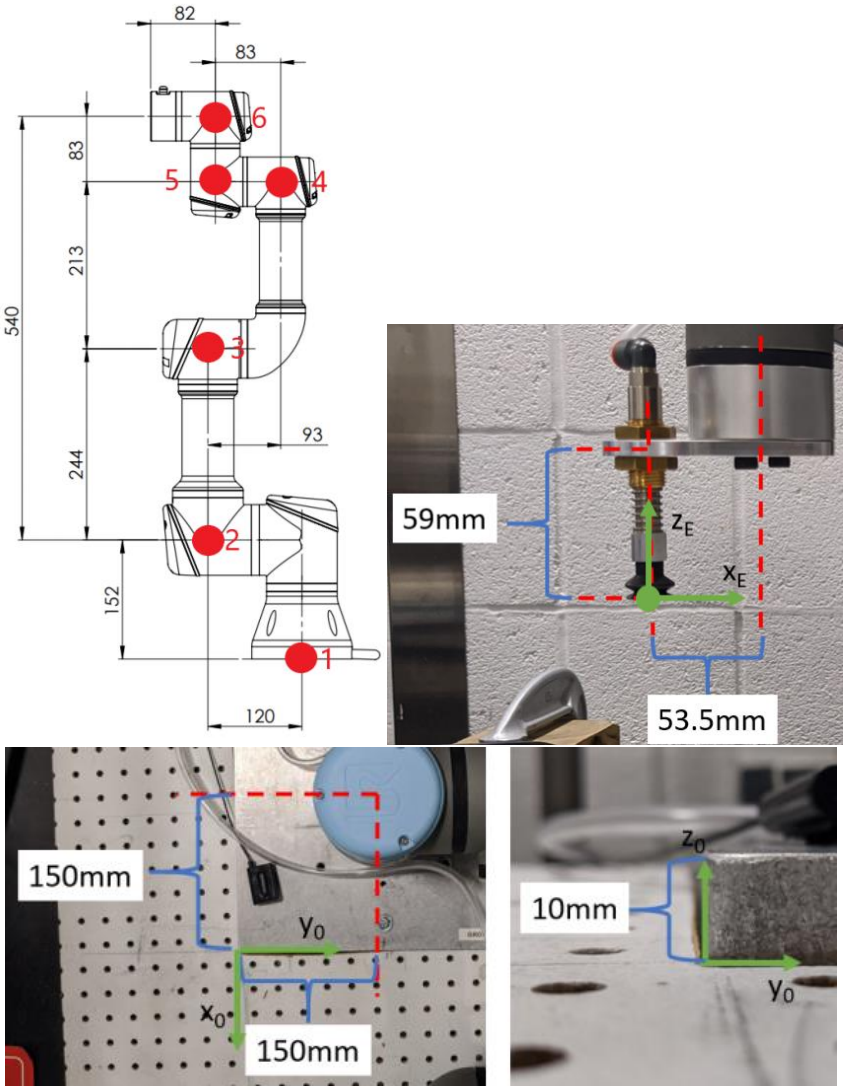

Figure 2: Labeled dimensions of the UR3 robotic arm and its setup in lab, with the coordinate frames defined in the lab.

The next step is to define the ω and q matrices for each joint. Figure 3 shows the rotation axis of each joint, which is used to construct the ω matrix in terms of the body frame. The q matrix is created using Figure 2's dimensions. For the sake of making the code more organized, the ω and q matrix for each joint is transposed and listed in a larger 2D matrix, such that the first row corresponds to the first joint and the second row corresponds to the second joint, and so on and so forth. These matrices end up being $\omega = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ $and\ q = \begin{bmatrix} -.150 & .150 & .010 \\ -.150 & .270 & .162 \\ .094 & .270 & .162 \\ .307 & .177 & .162 \\ .307 & .260 & .162 \\ .390 & .260 & .162 \end{bmatrix}$.

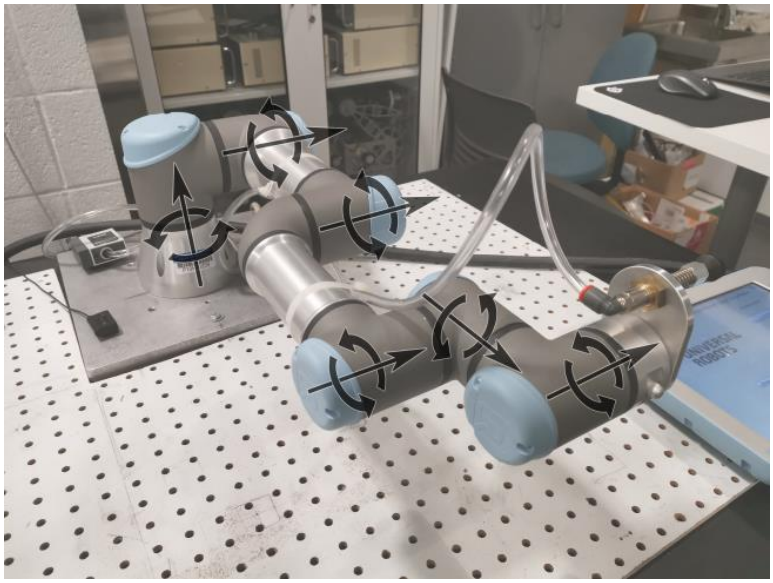

Figure 3: Zero configuration of the UR3, with the axes of rotation for each joint labeled.

I began my implementation of the code by programming the symbolic solution. Each joint needs to have the velocity vector calculated by taking the cross product of – ω and q, and then formatted into a square matrix, multiplied by its rotation angle, and exponentiated. Considering how this is a fairly repetitive and lengthy task, I created a helper function that would take in ω, q, and a placeholder for the rotation angle for a joint and return an exponential matrix in the form $e^{[S_i\theta_i]}$. Some of the issues that were encountered during this step included remembering to transpose the ω and q matrices, making them 3x1 instead of 1x3, before taking the cross product. The v matrix created was then transposed because I felt the code was easier to read when accessing a 1D matrix. Another issue was that the numpy function to get the exponential matrix wasn't working because the matrix was being multiplied by a sympy symbol for θ_i. Solving this was simple enough, the matrix needed to be created using sympy.Matrix() and then exponentiated using sympy's exp() function. The final version of the helper function created is shown in Figure 4. The result of each joint's cross product is listed below Figure 4.

```
def buildSymExponential(w, q, theta):
    temp_w = np.transpose(w) * -1          # transpose for cross product
    q = np.transpose(q)            # transpose for cross product

    v = np.cross(temp_w, q)
    v = np.transpose(v)            # transpose to make temp matrix look cleaner
    temp = sym.Matrix(
            [[0, -w[2], w[1], v[0]],
            [w[2], 0, -w[0], v[1]],
            [-w[1], w[0], 0, v[2]],
            [0, 0, 0, 0]]) * theta

    e = temp.exp()        # create exponential matrix
    return e
```

Figure 4: buildSymExponential() function that takes ω, q, and a placeholder for the rotation angle for a joint and returns an exponential matrix in the form $e^{[S_i\theta_i]}$.

$$v_1 = \begin{bmatrix} .15 \\ .15 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} -.162 \\ 0 \\ .094 \end{bmatrix}, v_3 = \begin{bmatrix} -.162 \\ 0 \\ .094 \end{bmatrix}, v_4 = \begin{bmatrix} -.162 \\ 0 \\ .307 \end{bmatrix}, v_5 = \begin{bmatrix} 0 \\ .162 \\ -.26 \end{bmatrix}, v_6 = \begin{bmatrix} -.162 \\ 0 \\ .39 \end{bmatrix}$$

The larger function for the symbolic solution is passed the M matrix, ω, and q and returns the transformation matrix T and a matrix of all the exponential matrices, e. The function is fairly straightforward and with more time could be cleaned up with a couple for loops and some formatting. The first portion creates a sympy symbol θi and passes it to buildSymExponential, along with its ω and q matrix. I spent a little bit of time trying to find a way to subscript the number identifying θ but was having trouble and decided it wasn't important enough to spend a lot of time on. Each exponential could probably have been done inside a for loop but I also didn't want to waste too much time trying to append them to the larger e matrix in the format I wanted. The second part of the function organizes the e matrices and uses a for loop to multiply them all together, finishing by multiplying it all by the M matrix. This function is shown in Figure 5.

```
def symbolicSolution(M, w, q):
    # e^[S1]θ1
    theta1 = sym.Symbol('\u03F41')
    e1 = buildSymExponential(w[0], q[0], theta1)
    # e^[S2]θ2
    theta2 = sym.Symbol('\u03F42')
    e2 = buildSymExponential(w[1], q[1], theta2)
    # e^[S3]θ3
    theta3 = sym.Symbol('\u03F43')
    e3 = buildSymExponential(w[2], q[2], theta3)
    # e^[S4]θ4
    theta4 = sym.Symbol('\u03F44')
    e4 = buildSymExponential(w[3], q[3], theta4)
    # e^[S5]θ5
    theta5 = sym.Symbol('\u03F45')
    e5 = buildSymExponential(w[4], q[4], theta5)
    # e^[S6]θ6
    theta6 = sym.Symbol('\u03F46')
    e6 = buildSymExponential(w[5], q[5], theta6)

    e = [e1, e2, e3, e4, e5, e6]
    T = e[0]
    for i in range(1, 6):
        T = np.matmul(T, e[i])
    T = np.matmul(T, M)

    return T, e
```

Figure 5: symbolicSolution() function that takes M, ω, and q and returns the transformation matrix T and a matrix of all the exponential matrices as a function of θ.

Running the symbolicSolution() matrix with the previously defined M, ω, and q matrices produces the exponential matrices shown below, for each joint in terms of θ.

$$e_1 =$$

$$\begin{bmatrix} .5e^{I\theta_1} + .5e^{-I\theta_1} & .5Ie^{I\theta_1} - .5Ie^{-I\theta_1} & 0 & -.5(-.15 + .15I)e^{I\theta_1} - .15 - I(.5I(-.15 + .15I) + .15I)e^{-I\theta_1} \\ -.5Ie^{I\theta_1} + .5Ie^{-I\theta_1} & .5e^{I\theta_1} + .5e^{-I\theta_1} & 0 & .5I(-.15 + .15I)e^{I\theta_1} + .15 + (.5I(-.15 + .15I) + .15I)e^{-I\theta_1} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e_2 =$$

$$\begin{bmatrix} .5e^{I\theta_2} + .5e^{-I\theta_2} & 0 & -.5Ie^{I\theta_2} + .5Ie^{-I\theta_2} & .5(.15 + .162I)e^{I\theta_2} - .15 - .5(-.15 + .162I)e^{-I\theta_2} \\ 0 & 1 & 0 & 0 \\ .5Ie^{I\theta_2} - .5Ie^{-I\theta_2} & 0 & .5e^{I\theta_2} + .5e^{-I\theta_2} & .5I(.15 + .162I)e^{I\theta_2} + .162 + .5I(-.15 + .162I)e^{-I\theta_2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e_3 =$$

$$\begin{bmatrix} .5e^{I\theta_3} + .5e^{-I\theta_3} & 0 & -.5Ie^{I\theta_3} + .5Ie^{-I\theta_3} & .5(-.094 + .162I)e^{I\theta_3} + .094 - .5(.094 + .162I)e^{-I\theta_3} \\ 0 & 1 & 0 & 0 \\ .5Ie^{I\theta_3} - .5Ie^{-I\theta_3} & 0 & .5e^{I\theta_3} + .5e^{-I\theta_3} & .5I(-.094 + .162I)e^{I\theta_3} + .162 + .5I(.094 + .162I)e^{-I\theta_3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e_4 =$$

$$\begin{bmatrix} .5e^{I\theta_4} + .5e^{-I\theta_4} & 0 & -.5Ie^{I\theta_4} + .5Ie^{-I\theta_4} & .5(-.307 + .162I)e^{I\theta_4} + .307 - .5(.307 + .162I)e^{-I\theta_4} \\ 0 & 1 & 0 & 0 \\ .5Ie^{I\theta_4} - .5Ie^{-I\theta_4} & 0 & .5e^{I\theta_4} + .5e^{-I\theta_4} & .5I(-.307 + .162I)e^{I\theta_4} + .162 + .5I(.307 + .162I)e^{-I\theta_4} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e_5 =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & .5e^{I\theta_5} + .5e^{-I\theta_5} & .5Ie^{I\theta_5} - .5Ie^{-I\theta_5} & .5(-.26 - .162I)e^{I\theta_5} + .26 - .5(.26 - .162I)e^{-I\theta_5} \\ 0 & -.5Ie^{I\theta_5} + .5Ie^{-I\theta_5} & .5e^{I\theta_5} + .5e^{-I\theta_5} & -.5I(-.26 + .162I)e^{I\theta_5} + .162 - .5I(.26 - .162I)e^{-I\theta_5} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$e_6 =$$

$$\begin{bmatrix} .5e^{I\theta_6} + .5e^{-I\theta_6} & 0 & -.5Ie^{I\theta_6} + .5Ie^{-I\theta_6} & .5(-.39 + .162I)e^{I\theta_6} + .39 - .5(.39 + .162I)e^{-I\theta_6} \\ 0 & 1 & 0 & 0 \\ .5Ie^{I\theta_6} - .5Ie^{-I\theta_6} & 0 & .5e^{I\theta_6} + .5e^{-I\theta_6} & .5I(-.39 + .162I)e^{I\theta_6} + .162 + .5I(.39 + .162I)e^{-I\theta_6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

After completing the symbolic solution, it was time to try to plug values into the sympy symbols for each joint angle to get a numerical solution. This ended up being difficult for me because I haven't used sympy's library of functions before and was running into issues with numpy functions not being able to operate on sympy matrices. Rather than learning how to use an entirely new package, I chose to create separate functions to numerically solve the kinematics equations using numpy like I've been used to all semester. The code to build the exponential matrix and solve for the transformation matrix is almost the same as it was for the symbolic solution, and is shown in Figure 6.

A last small function was created that takes in an array of thetas in radians and computes the numerical solution, then extracts the location of the end effector. The end effector's position is shown in the T matrix as the rightmost column. One of the issues I ran into at the end of programming was that I was forgetting to convert the test angles given to radians, but that was just a quick and simple fix. The results of the test configurations are shown below, with distances in meters.

Configuration	*Theta Values*	*X*	*Y*	*Z*
1	(35, -35, 25, -20, -90, 0)	0.2084	0.6006	0.2583
2	(10, -25, 35, -45, -90, 10)	0.4002	0.4122	0.1656

```
def buildExponential(w, q, theta):
    temp_w = np.transpose(w) * -1          # transpose for cross product
    q = np.transpose(q)            # transpose for cross product

    v = np.cross(temp_w, q)
    v = np.transpose(v)            # transpose to make temp matrix look cleaner
    temp = np.array([[0, -w[2], w[1], v[0]],
                [w[2], 0, -w[0], v[1]],
                [-w[1], w[0], 0, v[2]],
                [0, 0, 0, 0]])

    e = expm(temp*theta)           # create exponential matrix
    return e

def numericalSolution(M, w, q, thetas):
    e1 = buildExponential(w[0], q[0], thetas[0])    # e^[S1]θ1
    e2 = buildExponential(w[1], q[1], thetas[1])    # e^[S2]θ2
    e3 = buildExponential(w[2], q[2], thetas[2])    # e^[S3]θ3
    e4 = buildExponential(w[3], q[3], thetas[3])    # e^[S4]θ4
    e5 = buildExponential(w[4], q[4], thetas[4])    # e^[S5]θ5
    e6 = buildExponential(w[5], q[5], thetas[5])    # e^[S6]θ6

    e = [e1, e2, e3, e4, e5, e6]
    T = e[0]
    for i in range(1, 6):
        T = np.matmul(T, e[i])
    T = np.matmul(T, M)

    return T, e
```

Figure 6: buildExponential() and numericalSolution() functions that solve the forward kinematics equations given an array of thetas in radians.

Results

One of the biggest challenges of this lab was trying to learn to use sympy after being used to using numpy for numerical solutions this whole time. Part of what made it so difficult was that by not using sympy, I wasn't sure if the functions numerically did what I was expecting them to do. Knowing the difference between numpy.multiply(), the * operator, and np.matmul() was also difficult at the beginning of the semester and I was hesitant to convert to using an unfamiliar python package. For this lab, I stuck with only finding the T and e matrices using sympy and not using those matrices to plug values into my symbols. This makes it difficult to be confident that my symbolic solutions are correct because the numerical calculations are done completely separately. A quick visual inspection of my symbolic T matrix makes me believe it would be accurate but I also wouldn't be too surprised if there are some errors in it because I used numpy.matmul() on a sympy matrix object.

Errors that would be clearly seen in my numerical solution to the test configurations would most likely be caused my miscalculations in the q matrix or the last column of the M matrix. These values are the distance coordinates in the body frame. The listed verification configurations and solutions were within a rounding error of my solutions when testing my code, so I'm confident any error would have been caused by some small difference in measurement.

Conclusion

This lab wasn't too complicated overall, just somewhat tedious when it came to measuring the distances between joints accurately. The first steps to completing it focus around the physical robot and its dimensions, measuring distances and determining rotation axes for each joint and the base and tool coordinate frames. After creating those matrices, the math was very similar to the homework we have been doing lately to create the transformation matrix T.

I feel that the most valuable part of this lab was beginning to get familiar with sympy's library of functions. For the next lab, I aim to go into programming using sympy's functions to avoid worrying about converting between numpy and sympy matrices. I think doing that would make substituting values into my symbols go more smoothly. Being able to solve for solutions symbolically is useful on its own and I'm glad that this lab gave me an excuse to start learning how to do that in python.

References

Block, D., Holm, J., Xu, J., Fan, Y., Cui, H. and Chen , Y. (n.d.). *ECE470 Introduction to Robotics Lab Manual.* 2nd ed.