
CAL POLY

**CALIFORNIA POLYTECHNIC STATE UNIVERSITY**
CENG - DEPARTMENT OF ELECTRICAL ENGINEERING

# EE 471- Vision Based Robotic manipulation

## Lab #2 Forward kinematics of Robot Manipulators

**Presented to:**
Siavash Farzan

**Presented By:**

Rodrigo Menchaca
Giovanni Dal Lago

OCT 12, 2025

**INTRODUCTION:**

In this lab, we expanded on the foundations established in Lab 0 and Lab 1 by developing and testing the forward kinematics of the OpenManipulator-X. Our goal was to compute the position and orientation of the end-effector from the robot's joint angles, using the Denavit–Hartenberg (DH) convention as the mathematical framework. As a pre-lab assignment, we analyzed the manipulator's kinematic chain and created a DH table to describe the geometry of each link. Then, during the lab we designed and implemented several Python functions—`get_dh_row_mat()`, `get_int_mat()`, and `get_fk()` to build the individual transformation matrices and combine them into the full FK matrix.
After verifying the correctness of our implementation through numerical tests, we developed additional functions, `get_current_fk()` and `get_ee_pos()`, to compute the end-effector's position in real time as the arm moved. Finally, we used our FK implementation to calculate and verify three joint-space waypoints forming a triangle within the robot's reachable workspace, which we then commanded the robot to follow sequentially.

**LAB PROCEDURE:**

## 1) Calculate the forward kinematics of the OpenManipulator-X arm

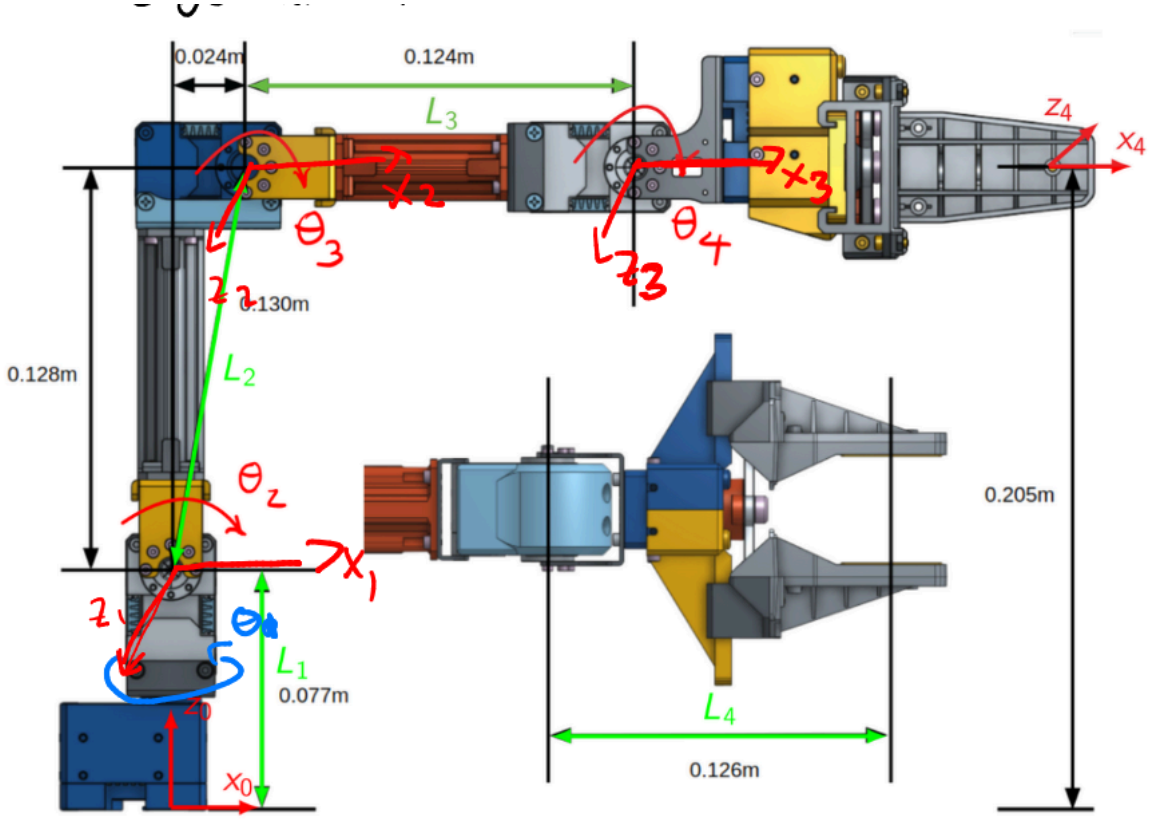


*Figure 0: Kinematic chain Diagram for Robot Manipulator*

| I | Theta | d | a | Alpha |
|---|---|---|---|---|
| **1** | Theta1 | L1 | 0 | 90 |
| **2** | Theta2 | 0 | 0 | 0 |
| **3** | Theta3 | 0 | L2 | 0 |
| **4** | Theta4 | 0 | L3 | 0 |

*Table 1: DH Parameters calculated in Prelab*

Using the Kinematic chain diagram we developed during our Prelab, the following DH table was developed. (Table 1) We ultimately ended using the provided DH table that was provided to us during Lab shown in Table 2.

| I | THETA | D | A | ALPHA |
|---|---|---|---|---|
| **1** | theta1 | L1 | 0 | $-\frac{\pi}{2}$ |
| **2** | $\theta 2 - (\frac{\pi}{2} - sin^{-1}(24/130))$ | 0 | L2 | 0 |
| **3** | $\theta 2 + (\frac{\pi}{2} - sin^{-1}(24/130))$ | 0 | L3 | 0 |
| **4** | theta4 | 0 | L4 | 0 |

*Table 2: Provided table used to perform Forward kinematics.*

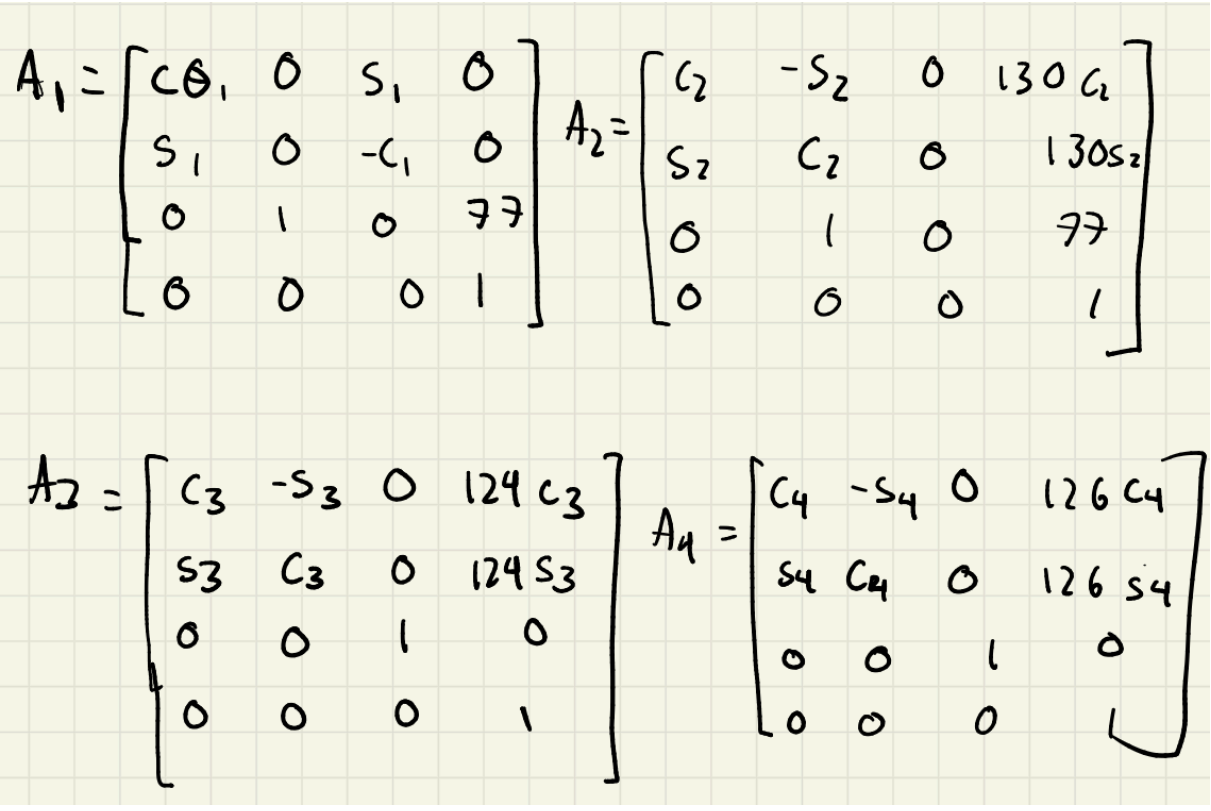


*Figure 1: Ai Matrices developed during prelab*

## 2) Implement the forward kinematics in Python

In order to implement our Forward Kinematics in python and the robot arm manipulator. We had to implement some helper functions that will assist us with this. The functions get_dh_row_mat(), get_int_mat(), get_fk() were developed to assist us with creating the Ai matrix based on a given DH table that was provided beforehand, calculate each of the Ai matrices based on the previously calculated matrix and develop the FK matrix once all the previously mentioned parameters were calculated. Once the functions were built, we used 3 different scenarios to make a quick sanity check. Shown in Figure 2, we can see that these values are pretty close to the expected outcomes. A rounding discrepancy was the cause for not completely resembling the expected values provided.

```
FK for [0,0,0,0]
Joint angles 1:  [[  1.      0.      0.    274.   ]
 [  0.      0.      1.     -0.   ]
 [  0.     -1.      0.    204.765]
 [  0.      0.      0.      1.   ]]

FK for [15,-45,-60,90]
Joint angles 2:  [[  0.933   0.25   -0.259  15.687]
 [  0.25    0.067   0.966   4.203]
 [  0.259  -0.966   0.    336.7  ]
 [  0.      0.      0.      1.   ]]

FK for [-90,15,30,-45]
Joint angles 3:  [[   0.       0.       1.       0.   ]
 [  -1.      -0.       0.    -269.932]
 [   0.      -1.       0.     106.519]
 [   0.       0.       0.       1.   ]]
PS C:\Users\mench\Downloads\ee471-codebase(1)>
```

*Figure 2: Sanity check results for FK function*

All of our resulting values were within 3-4 decimal places, and our position was at a logical position within the reachable area of the robot arm manipulator.

## 3) Interface forward kinematics with the robot

After the verification of our first 3 functions used to obtain our forward kinematics matrix. We continued to develop these functions to now obtain continuous real time matrices as our Robot arm moved into the desired position. To implement this, we first had to develop a new function called get_current_fk(). This function used the basis of our initial forward kinematics equation only with the addition of using the current joint readings by the use of the get_joint_readings() function. The new function continuously records the FK matrix for the current joint angle that is being read. Another function, called get_ee_pos() was developed to extract the end effector position once the current FK matrix was extracted. In our lab2_3.py script, we developed a code that would run both of these functions throughout the whole trajectory time of the robot arm and print the values in our terminal. We used [15,-45,-60,90] and [-90,15,30,-45] to extract the Forward kinematic matrix for the actual robot, instead of only our experimental values. Shown in Figure 3.1 and 3.2, we can see that the actual values align with the theoretical values quite closely.

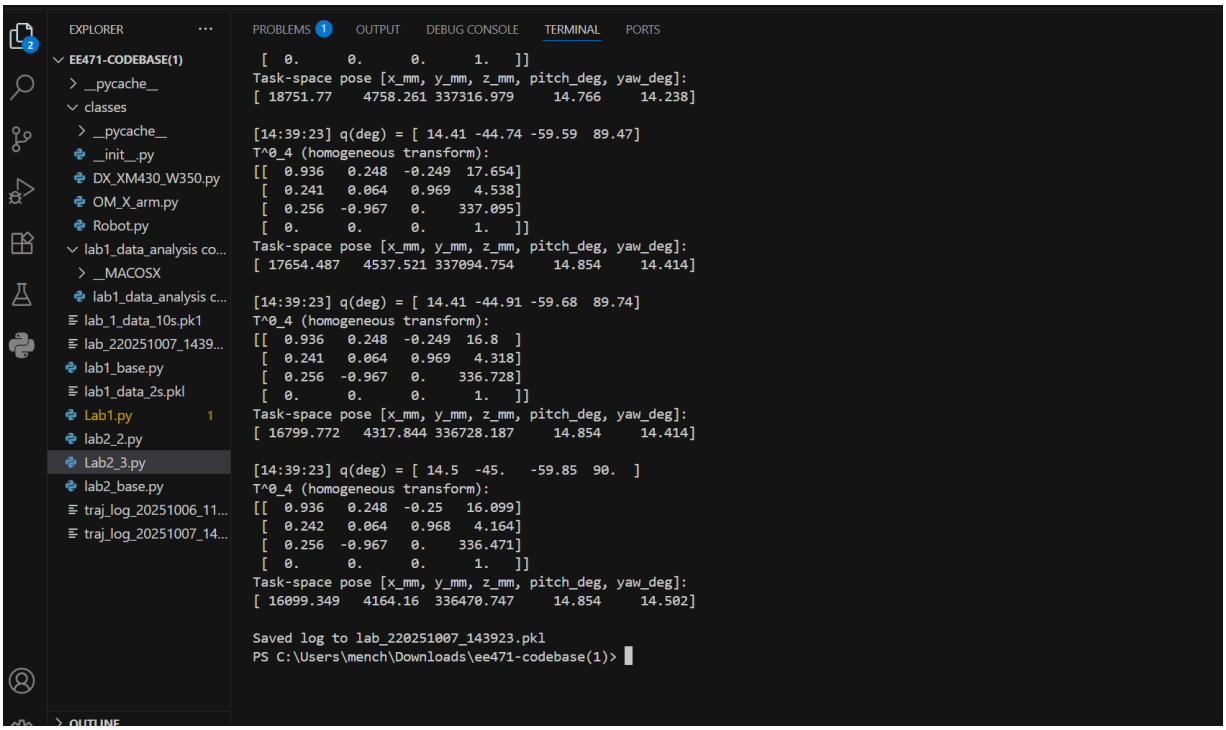


*Figure 3.1: FK results for [15,-45,60,90] joint angles*

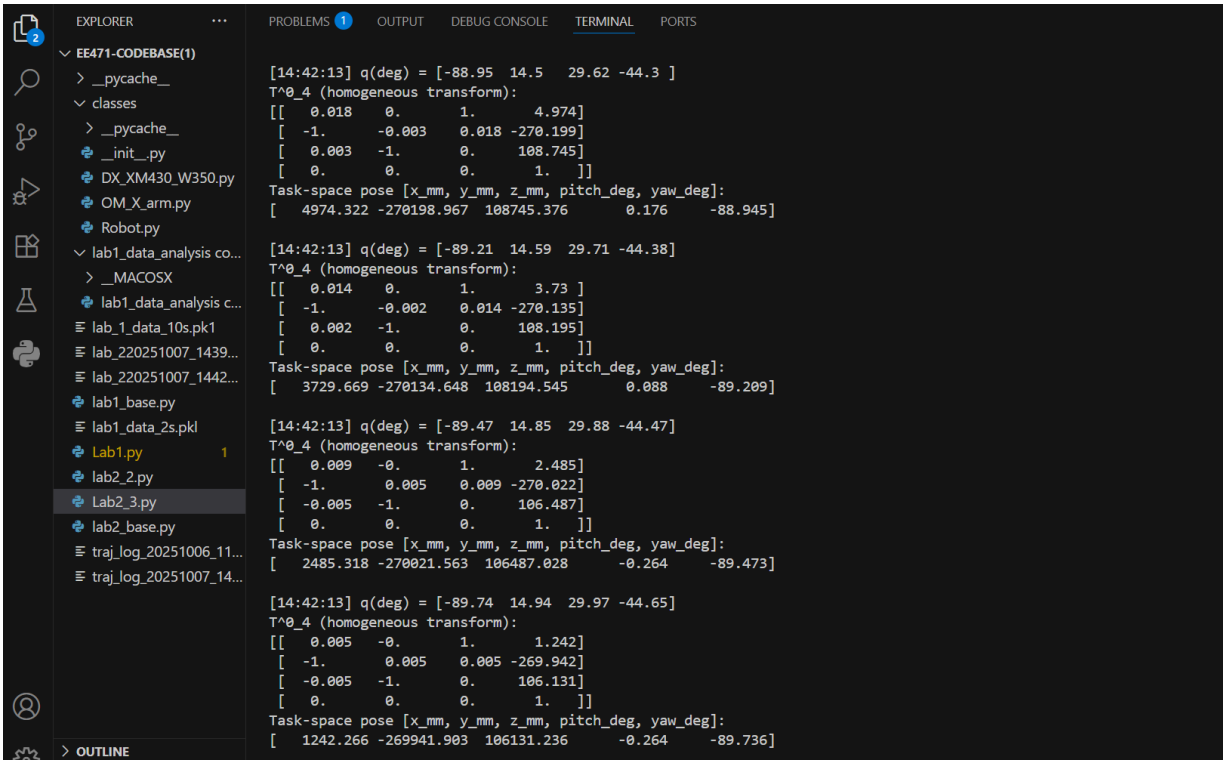


*Figure 3.2: FK results for [-90,15,30,-45] joint angles*

Overall, the results were the same with some discrepancies. These discrepancies were mostly caused by small differences between the angles being read and the actual angles. However, when comparing the resulting values in the Forward kinematics matrix, the difference was very minimal.(only a few decimal points)

**CODE IMPLEMENTED:**

```
# lab2_3.py
from classes.Robot import Robot
import numpy as np
import time
import pickle
from datetime import datetime

# pretty printing for matrices/arrays
np.set_printoptions(precision=3, suppress=True)

def main():
    seconds = 10.0           # trajectory duration [s]
```

```
    poll_dt = 0.02          # sampling interval [s]
    q_target = [-90,15,30,-45]  # degrees

    robot = Robot()
    try:
        # enable & set move time
        robot.write_motor_state(True)
        robot.write_time(seconds)

        # go home
        robot.write_joints([0, 0, 0, 0])
        time.sleep(seconds)

        # logs
        t_log, q_log = [], []

        # command target and sample during motion
        robot.write_joints(q_target)
        t0 = time.perf_counter()

        last_q = None
        deg_epsilon = 0.25   # print only when change exceeds this (deg)

        while True:
            t = time.perf_counter() - t0
            q_deg, qd_dps, I_mA = robot.get_joints_readings()

            # ensure it's plain floats for logging
            q_vec = np.asarray(q_deg, dtype=float).reshape(4)

            # log
            t_log.append(t)
            q_log.append(q_vec.tolist())

            # compute & print when angles change meaningfully
            if last_q is None or np.max(np.abs(q_vec - last_q)) > deg_epsilon:
```

```
                T = robot.get_fk(q_vec)                 # 4x4 homogeneous
                ee = robot.get_ee_pos(q_vec)            # [x_mm, y_mm, z_mm,
pitch_deg, yaw_deg]

                ts = time.strftime("%H:%M:%S")
                print(f"\n[{ts}] q(deg) = {np.round(q_vec,2)}")
                print("T^0_4 (homogeneous transform):")
                print(T)
                print("Task-space pose [x_mm, y_mm, z_mm, pitch_deg,
yaw_deg]:")
                print(np.round(ee, 3))

                last_q = q_vec

            # exit after duration
            if t >= seconds:
                break

            # sleep with clamp to maintain rate
            remaining = poll_dt - (time.perf_counter() - t0 - t)
            if remaining > 0:
                time.sleep(remaining)

        # one last reading at the end
        q_deg, _, _ = robot.get_joints_readings()
        t_log.append(time.perf_counter() - t0)
        q_log.append(np.asarray(q_deg, dtype=float).reshape(4).tolist())

        # save log
        data = {"t": t_log, "q_deg": q_log}
        fname = f"lab_2{datetime.now().strftime('%Y%m%d_%H%M%S')}.pkl"
        with open(fname, "wb") as f:
            pickle.dump(data, f)
        print(f"\nSaved log to {fname}")

    finally:
        robot.close()
```

```
    return t_log, q_log

if __name__ == "__main__":
    main()
```

Overall, the code successfully commanded the robot to move from its home position to the target joint angles while continuously logging data. The recorded results showed consistent behavior, with only minor discrepancies likely caused by small differences between the measured and actual joint angles during motion. Despite these small variations, the computed forward kinematics and end-effector positions remained very close to the expected values, with differences only noticeable within a few decimal points.

**CONCLUSION**

Through this experiment, we successfully implemented and validated the Forward Kinematics model for the OpenManipulator-X. Our Python functions produced accurate end-effector poses, and the differences between computed and measured positions were minimal, confirming the reliability of our approach. We observed that small discrepancies mainly arose from angle reading precision and floating-point rounding.

By testing the arm along a triangular trajectory, we verified that our FK correctly mapped joint configurations to spatial coordinates, both in static and dynamic conditions. Overall, we gained practical understanding of how FK links the robot's geometry to its motion and established the computational foundation needed for future work on inverse kinematics and trajectory control.