WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Lab 3 – INVERSE KINEMATICS AND TRAJECTORY GENERATION

SUBMITTED BY
Nicolas Graham
Pranav Jain
Fiona Prendergast

Date Submitted: 09.19.2024
Date Completed: 09.11.2024
Course Instructor: Prof. Agheli
Lab Section: RBE 3001 A’24

Abstract

This Report covers the implementation of inverse kinematics (IK) and trajectory generation techniques on the OpenManipulator-X robot arm using MATLAB. The Objective of this lab was to use MATLAB to compute the joint angle values of the 4 DOF robot arm, when given the desired position of the end effector (EE) in physical space. This lab was focused on calculating inverse kinematics, generating smooth trajectories using cubic and quintic polynomials, and integrating these calculations to control the OpenManipulator-X robot arm. The motion of the robot arm was visualized using 3D Plots, and the implemented methods were verified by guiding the OpenManipulator-X robot arm through various target positions. These results are critical for precise control of the OpenManipulator-X robot motion in future tasks.

Introduction

In Lab 3, the concepts of inverse kinematics (IK) and trajectory generation are explored using the OpenManipulator-X robot arm and MATLAB. The goal of the lab is to calculate and implement algorithms to control the robot arm’s end effector as it moves between different points in space. The lab builds of the previous labs (Lab 1 and 2) where the forward kinematic (FK) and control methods were developed and tested in MATLAB.

Inverse Kinematics is the math used to convert given points in the task space as [X, Y, Z, α] and convert it to the joint variables needed to get the end effector to that point [q1, q2, q3, q4]. There are two methods that can be used to calculate inverse kinematics. One method is to take an algebraic approach by translating the forward kinematics (the last column in the 4x4 homogeneous transformation matrix holds the position of the end effector) into equivalent polynomial equations to solve for the joint values. The other method is a geometric approach to find the joint values by inspection of the kinematic chain and the use of the law of cosines.

In the lab, to calculate, implement and test the IK mathematics and MATLAB functions, the robot arm is moved between different poses and data about the joint and end effector position are collected and plotted over time to visualize the motion of the robot arm in 3D space. This report describes the methods used to implement the methods and collect the data and the graphical representations of the robot arm movement are discussed. The inverse kinematics of the robot arm are vital to being able to move to specific locations within the workspace and are the groundwork for future labs and being able to manipulate the end effector to complete detailed tasks.

Methodology

Calculating Inverse Kinematics

The first step of Lab 3 was to calculate the inverse kinematics of the 4-DOF robot arm. The geometric approach, as described in the introduction, was used because with a 4DOF robot the algebraic method could result in infinite solutions. First the joint and task spaces were defined according to the convention followed in the previous labs, shown in Figure 1 and 2. The lengths of the links, labeled 0-4, are shown in Figure 3 in millimeters.

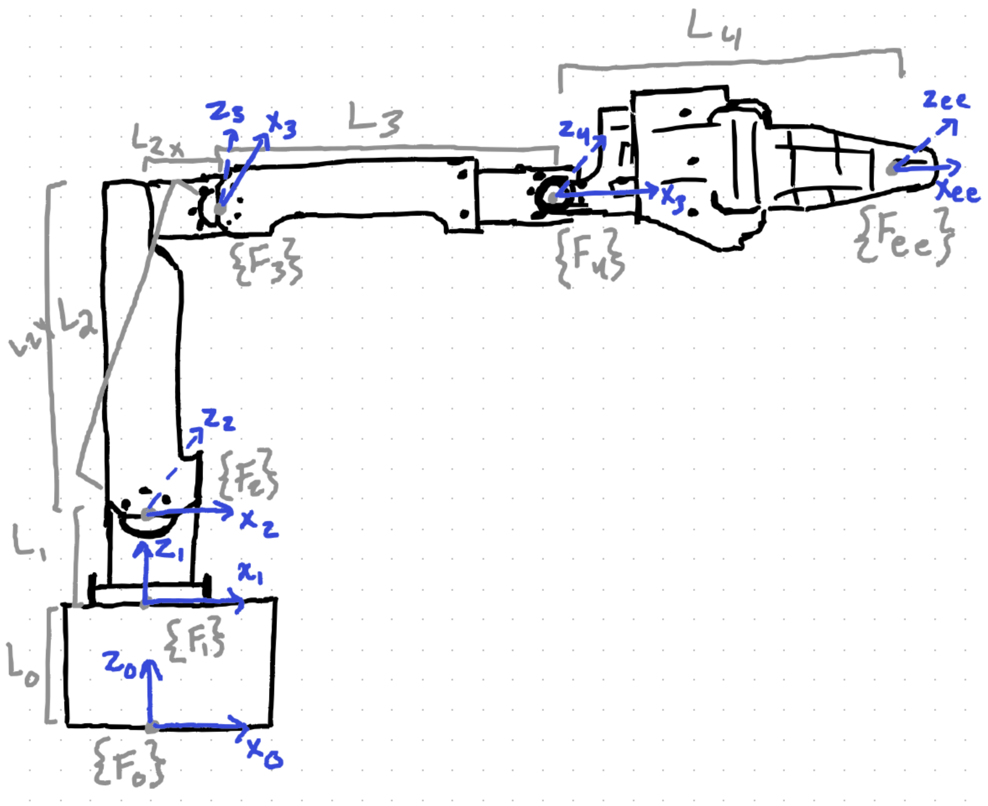


Figure 1: Diagram of the OpenManipulator-X robot arm in the home position (all of the joint angles set to 0 degrees) with the frames 0-4 and end effector frame assigned, and the link lengths labeled.

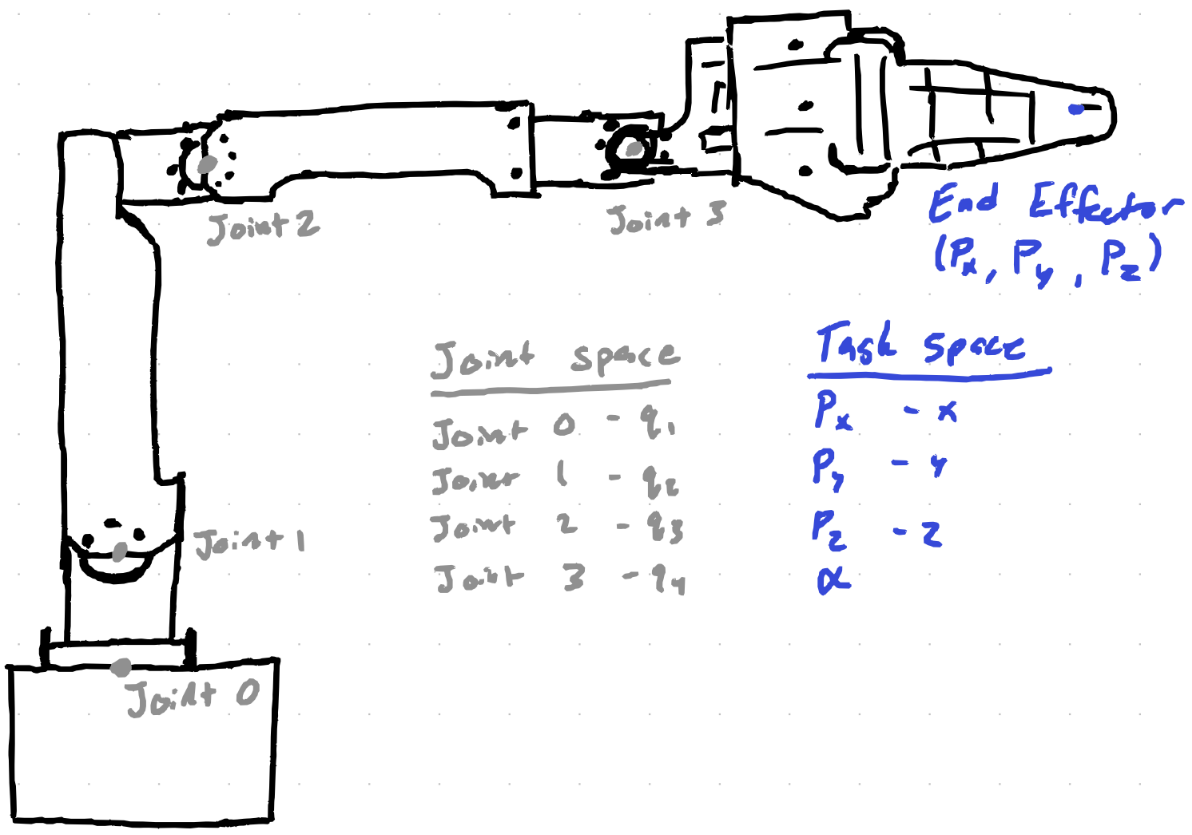


Figure 2: Diagram of the 4-DOF robot arm with the joint space (q1, q2, q3, q4) and task space (X, Y, Z, α) variables labeled.

Link	Lengths [mm]
L0	36.076
L1	60.25
L2x	24.0
L2y	128.0
L2	130.231
L3	124
L4	133.4

Figure 3: Table of the lengths of each of the robot links in millimeters from the lengths given in Lab 1. L2 was calculated using L2x and L2y and the Pythagorean theorem.

Θ_1 was found first by creating triangles with known side lengths from the robot base to the end effector and with the law of cosines plugging Sin(Θ_1) and Cos (Θ_1) into the MATLAB function atan2. The other inverse trigonometric functions cannot be used in MATLAB because there can be two angles (positive and negative) that are both valid solutions, but only atan2 allows MATLAB to find all of the valid solutions. The diagram and steps to find Θ_1 are shown in figure 4 in results.

Θ_2 was more complex but was found using the same method of forming triangles and applying the law of cosines. Two intermediate angles, alpha and beta, were defined and found by using the atan2 function and were then summed to calculate Θ_2. Since link 2 is not completely vertical in the 0 position, the 10.62-degree angle of link 2 needed to be accounted for in the calculations for Θ_2. The angle was calculated using trig functions in the previous lab. For visual clarity in figure 5, joints 2, 3, and 4, were drawn in a new frame rotated from frame 0 so that the x-axis is along the plane from the base of the robot to the end effector. Since all of the links and joints shown in figure 5 are in the same plane, the frame does not change the calculations at all. The calculations for Θ_2 are shown in figure 7 in results.

Similarly, Θ_3 was found by the law of cosines and imputing the expressions for Sin and Cos of the angle into the atan2 MATLAB function. Again, the slight offset of Θ_2 needs to be accounted for with the directionality of the joint value, so as shown in the diagram in figure 8 with the calculations for Θ_3, the value of Θ_3 must be found by subtracting 79.38-degrees from the result of the atan2 function.

MATLAB Implementation of Inverse Kinematics

To implement the inverse kinematic calculations made for the robot arm in MATLAB, a function called ik3001() was created in the Robot class with input on an array p which stored a 1x4 array with the [X, Y, Z, α] task space and end effector orientation values.

The function used the equations derived above with the inverse kinematics to solve for Θ_1, Θ_2, Θ_3, and Θ_4 in that order and return the four joint values [q1, q2, q4, q4] in a 1x4 joint space array. The function atan2 was used throughout ik3001 since it allows both the positive and negative values to be calculated and returned in the function.

Since the robot arm has 4 DoF, theoretically there could be 16 possible arm configurations for any given point input into the ik3001, so a check function, called checkIKangles(), was implemented to find the most accurate configuration of the arm to the point given in task space. CheckIKangles iterates over each of the possible configurations and by using fk3001 finds the set of joint angles with the least amount of error to the given [X, Y, Z, α].

Validating Inverse Kinematics

Even though the checkIKangles function was written and implemented, the inverse kinematics function and calculations still needed to be properly verified before implementing them in any real problem application. To validate the inverse kinematics function, three arbitrary task space points were chosen so that a non-coplanar triangle is formed between the three points within the workspace. Then the ik3001 function was used to get the joint angles [q1, q2, q3, q4] for each of the vertices.

The robot arm was programmed to move between the three points to travel the path of a complete triangle. As the arm moved, the joint angle values [q1, q2, q3, q4] and the end effector position [X, Y, Z, α] were recorded over time and stored in a data array.

A 3D model plot of the robot arm movement over time was generated with the data collected during the robot arm movement. The triangle seen in Figure 16 has a warped shape but does distinctly resemble a triangle.

Trajectory Planning in Joint space

To implement trajectory planning, a new class named Traj_Planner() was created. A new method cubic_traj() was added to this class. This method uses the start and end times, points and velocities to find the coefficients of a polynomial function that maps the movement from the initial to final point. A run_trajectory() method was also created in the Robot class that converted these coefficients to functions of time and plot the robotic arm's trajectory as it moves from the initial to final position.

Trajectory Planning in Task Space

Using the task space trajectory generation, the robot followed a predefined path based on task coordinates rather than joint angles. This involved calculating trajectories for each cartesian component x, y z separately and applying inverse kinematics to convert these into joint angles. To do this, run_trajectory() method was extended to allow trajectory planning in the task space. As alpha also was variable between the points, a trajectory for alpha was also calculated using cubic_traj() function.

Quintic Trajectory Planning

For more advanced and smooth movement quintic trajectories were employed, allowing control over initial and final accelerations in addition to positions and velocities. This further refined the robots motion ensuring smoother stops and starts. A new method called quintic_traj() was created to do this type of trajectory planning.

Extra credit: Circular Trajectory

As an extension, the robot was programmed to follow a circular trajectory not confined to the primary planes. This involved generating a 3D circle using parametric equations and employing quintic polynomials to ensure smooth execution. However, challenges in accurately generating the circle and issues with points on the trajectory appearing to be out of bounds prevented full functionality. Despite this, we believe that our function is on the right path to correctly setting a circular trajectory. We believe that our current function does work partially to generate a trajectory, but something has gone wrong in our process. With more time to debug and rework, we could achieve consistent results.

Results

Inverse Kinematics Calculations

To calculate the inverse kinematics of the robot arm, detailed drawings of the joints, links and end effector were drawn to make the geometric calculations easier to visualize. Figure 4 and 5 show drawings of the entire 4 DOF robot arm with labeled joints, links and intermediate lengths and a partial drawing of links 2 and 3 respectively. The drawings were an extremely important step to be able to calculate the joint variables with inverse kinematics.

Figures 6-9 show the handwritten calculations to solve for Θ_1, Θ_2, Θ_3, Θ_4 in each figure respectively. Since alpha is given in the inverse kinematics calculation and the joint 3 value (Θ_3) is calculated by adding together two intermediate angles (α_1 and β_1 shown in Figures 5 and 7), the set of possible joint angle configurations for the robot arm contains 16 solutions.

Not all the configurations of joint angles will result in the desired end effector position, but it is necessary to note that with the use of atan2 in the calculations, there are multiple solutions for the joint angle configuration to consider. The MATLAB function ik3001 needed to take the multiple potential joint configurations into account so that the output of the function would be the most accurate to the input task space coordinates and wrist angle.

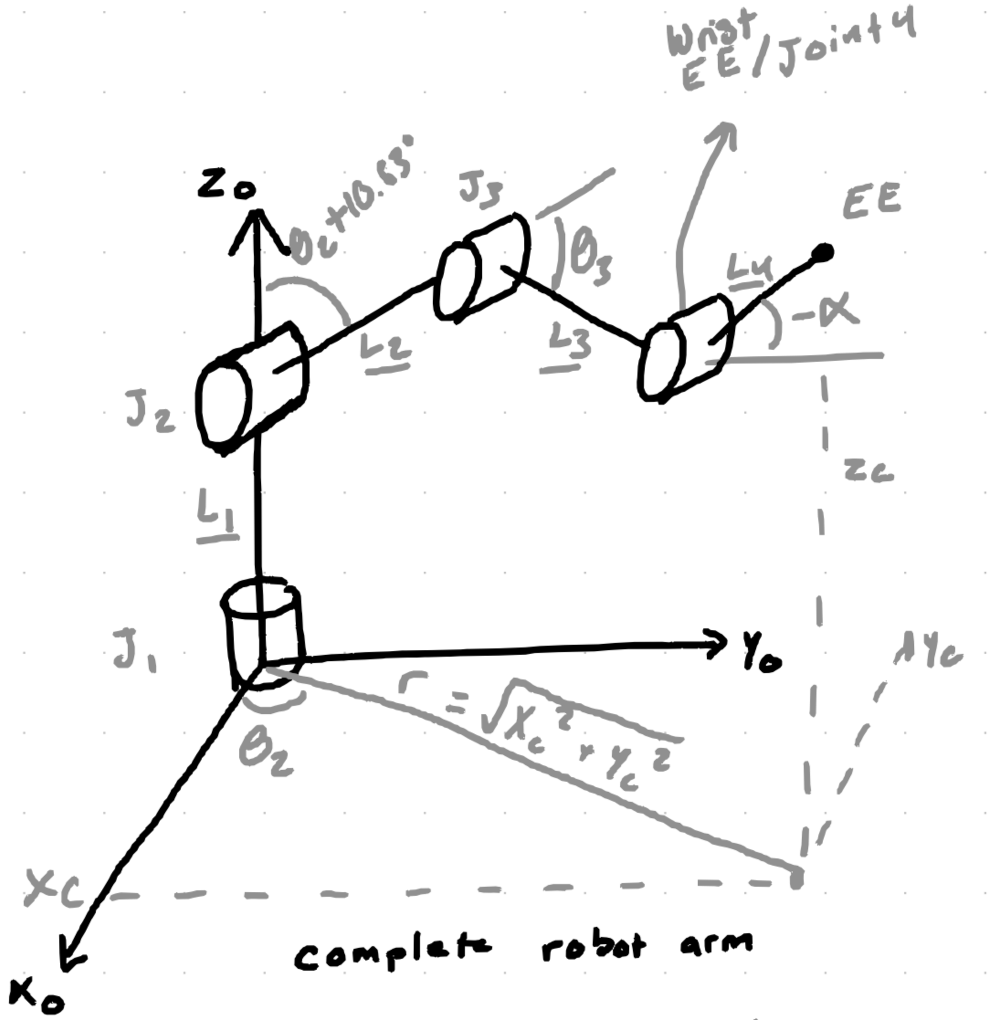


Figure 4: Stick diagram of the 4-DOF robot arm in frame 0 with joints 1-4, links 1-4 and joint values (Θ_1 , Θ_2 , Θ_3, Θ_4) all labeled

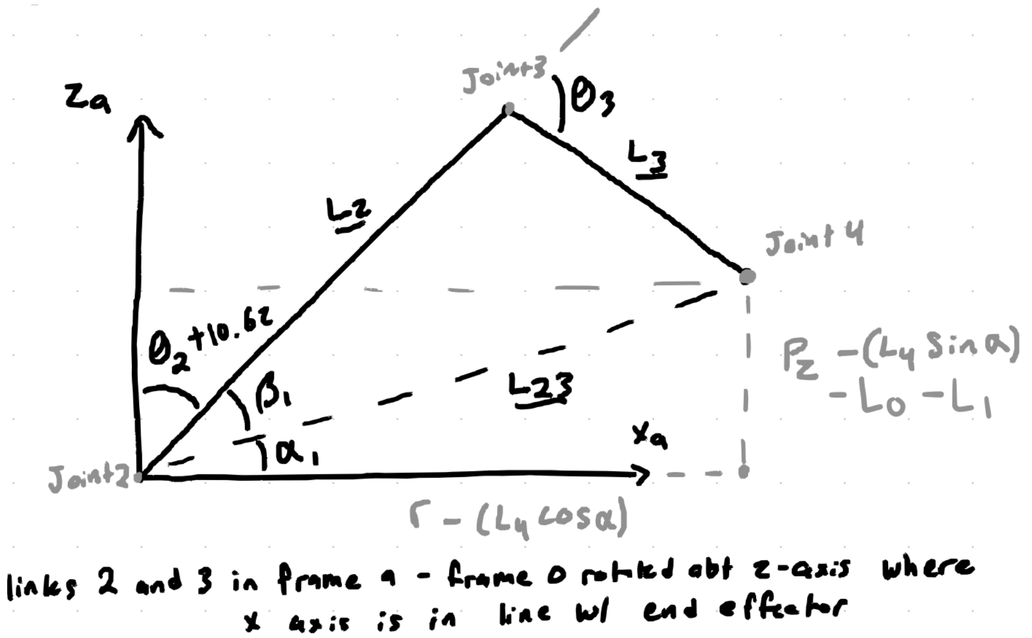

Za
Joint3
θ3
L3
L2
Joint 4
θ2+10.62
β1
L23
Pz −(L4 sin α) −L0 −L1
α1
Xa
Joint2
r − (L4 cos α)
links 2 and 3 in frame a - frame 0 rotated abt z-axis where x axis is in line w/ end effector

Figure 5: Simplified drawing of joints 2-4, and links 2 and 3 out of the 4-DOF robot arm. The links have been redrawn in a frame {a} (which is just frame 0 rotated about the Z-axis so X is in line with the end effector) with the Xa and Ya axes labeled. Angles Θ_2 and Θ_3 are labeled as well as intermediate angles α_1 and β_1 and intermediate links L_{23} and the X and Z values of the joint 4 position in frame {a}.

ANGLE 1

$$\cos(\theta_1) = \frac{x_c}{r} \qquad \sin(\theta_1) = \sqrt{1 - \left(\frac{x_c}{r}\right)^2}$$

$$\theta_1 = \operatorname{atan2}\left(\pm\sqrt{1 - \left(\frac{x_c}{r}\right)^2},\ \frac{x_c}{r}\right)$$

Figure 6: Handwritten calculations to solve for joint angle 1 or Θ_1 which in the case of the 4 DOF robot arm is also the joint value q1.

$$L_{23} = \sqrt{(r - L_4\cos\alpha)^2 + (P_z - L_4\sin\alpha - L_1 - L_0)^2}$$

ANGLE 2

$$\cos(\beta_1) = \frac{L_2^2 + L_{23}^2 - L_3^2}{2\cdot L_2\cdot L_{23}} = B \qquad \sin(\beta_1) = \sqrt{1 - B^2}$$

$$\cos(\alpha_1) = \frac{r - (L_4\cos(\alpha))}{L_{23}} = C \qquad \sin(\alpha_1) = \sqrt{1 - C^2}$$

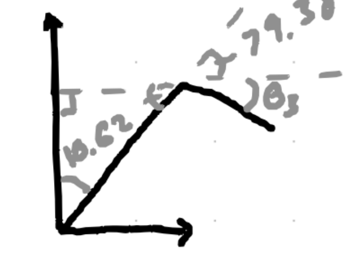
$$\beta_1 = \text{atan2}(\pm\sqrt{1 - B^2}, B)$$

$$\alpha_1 = \text{atan2}(\pm\sqrt{1 - C^2}, C)$$

$$\theta_2 = 90 - 10.63 - (\beta_1 + \alpha_1)$$

Figure 7: Handwritten calculations to solve for joint angle 2 or Θ_2 by utilizing the intermediate angles called α_1 and β_1 drawn in Figure 5 as well as the calculation for the intermediate length L_{23}.

ANGLE 3



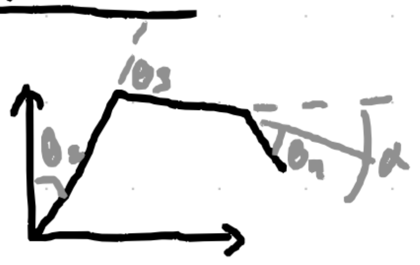
$$\cos(\theta_3) = \frac{L_2^2 + L_3^2 - L_{23}^2}{2\cdot L_2\cdot L_3} = A$$

$$\sin(\theta_3) = \sqrt{1 - A^2}$$

$$\theta_3 = \text{atan2}(\pm\sqrt{1 - A^2}, A) - 79.38$$

Figure 8: Handwritten calculations to solve for joint angle 3 or Θ_3 and an intermediate diagram of the similar triangles created to be used in the calculation.

ANGLE 4



$$\theta_4 = \alpha - (\theta_2 + \theta_3)$$

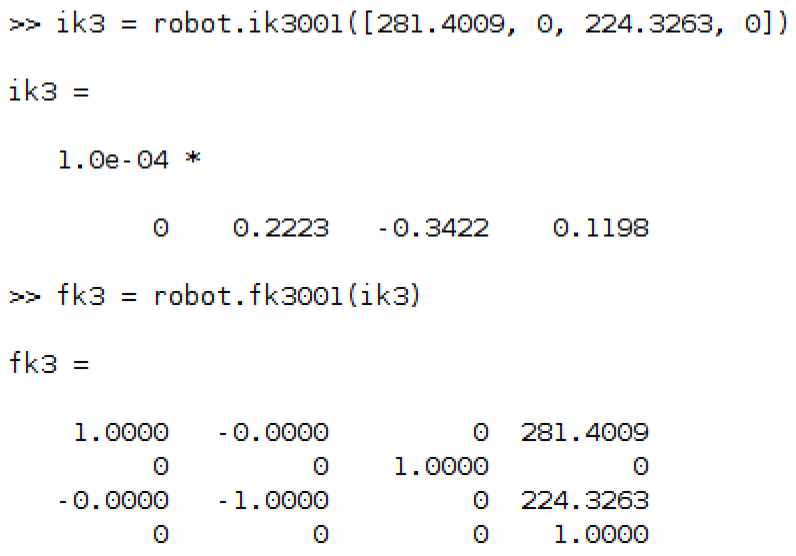
Figure 9: Handwritten calculations to solve for joint angle 4 or Θ_4 and an intermediate diagram of the similar triangles created to be used in the calculation.

Validating Inverse Kinematics Function

Verification for the inverse kinematics function (ik3001 function) written in MATLAB involved testing the function and verifying the results match the joint inputs to the already tested and verified forward kinematics function fk3001.

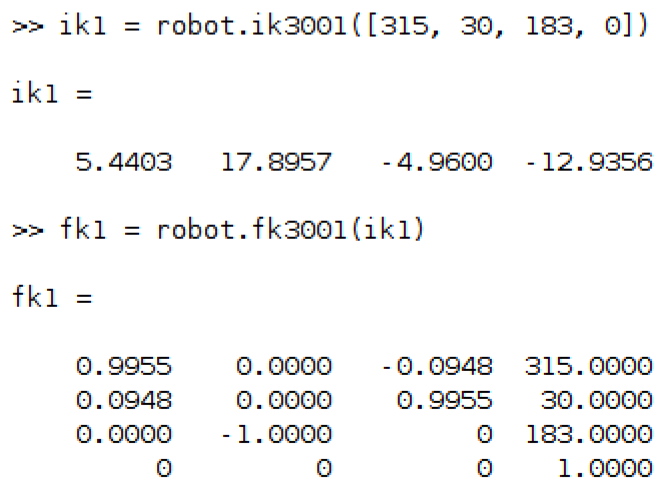
First the Zero / Home position was tested. The function ik3001 was run with the input of [281.4009, 0, 224.3263, 0] (because these are the coordinates measured by the measured_js function when the robot arm is in the home position with all the joints set to 0) and the results were displayed in the command window. Then fk3001 was run with the output from IK function and the 4x4 homogeneous transformation matrix was displayed in the command window as well. The output of the two functions is shown below in Figure 10. The function can be verified because the last column of the transformation matrix corresponds to the input of the ik3001 function. This means that the inverse kinematics accurately computes the joint angles needed to move the end effector to the input task space coordinates.

Two other arbitrary poses were tested to verify the ik3001 function in the same way as the home position. The input values were [315, 30, 183, 0] with the output from both functions shown in Figure 11 and [100, -125, 300, 0] with the output from both functions shown in Figure 12.



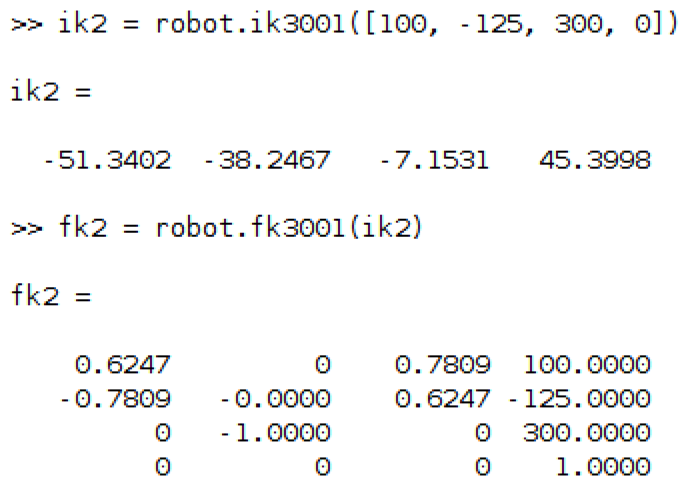
```
>> ik3 = robot.ik3001([281.4009, 0, 224.3263, 0])

ik3 =

   1.0e-04 *

         0    0.2223   -0.3422    0.1198

>> fk3 = robot.fk3001(ik3)

fk3 =

    1.0000   -0.0000         0  281.4009
         0         0    1.0000         0
   -0.0000   -1.0000         0  224.3263
         0         0         0    1.0000
```

Figure 10: Command window output from running ik3001 and fk3001 with the IK output. The input for the ik3001 function is the coordinate task space values representing the home position [281.4009, 0, 224.3263, 0].



```
>> ik1 = robot.ik3001([315, 30, 183, 0])

ik1 =

    5.4403   17.8957   -4.9600  -12.9356

>> fk1 = robot.fk3001(ik1)

fk1 =

    0.9955    0.0000   -0.0948  315.0000
    0.0948    0.0000    0.9955   30.0000
    0.0000   -1.0000         0  183.0000
         0         0         0    1.0000
```

Figure 11: Command window output from running ik3001 and fk3001 with the IK output for an arbitrary pose of the robot for verification. The input for the ik3001 function [315, 30, 183, 0].



```
>> ik2 = robot.ik3001([100, -125, 300, 0])

ik2 =

  -51.3402  -38.2467   -7.1531   45.3998

>> fk2 = robot.fk3001(ik2)

fk2 =

    0.6247         0    0.7809  100.0000
   -0.7809   -0.0000    0.6247 -125.0000
         0   -1.0000         0  300.0000
         0         0         0    1.0000
```

Figure 12: Command window output from running ik3001 and fk3001 with the IK output for an arbitrary pose of the robot for verification. The input for the ik3001 function [100, -125, 300, 0].

The verification of the IK method was not purely mathematical. The ik3001 was also tested moving through a triangle created by the end effector moving between three arbitrarily chosen poses with the end effector inside the workspace. The photo of the physical robot and the corresponding 3D model arm plot for each of the three vertices are shown below in Figures 13-15 for each vertex respectively.

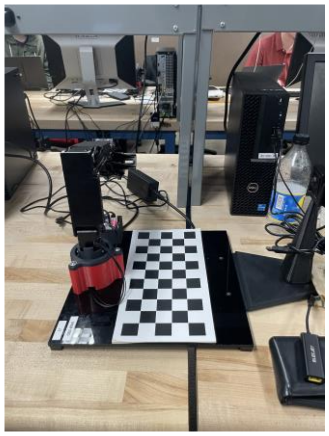
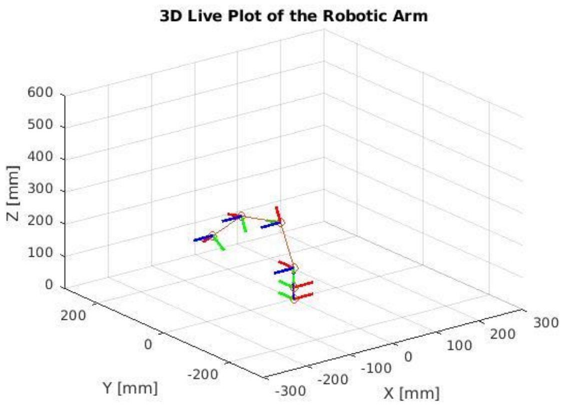
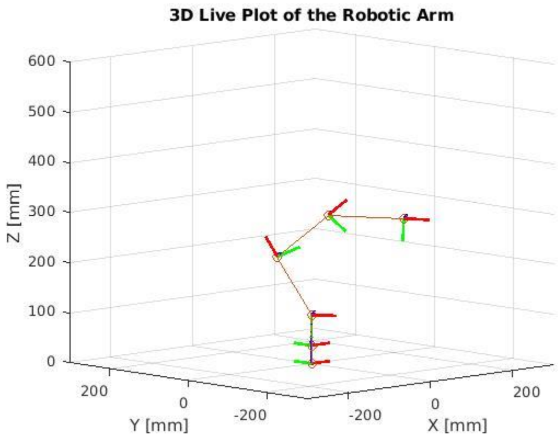



Figure 13: Photo and 3D live plot image of the 4-DOF in the position of the first vertex of the triangle formed by three arbitrary poses of the robot arm. The X, Y, Z, and α of the end effector are [0, 250, 100, 45].



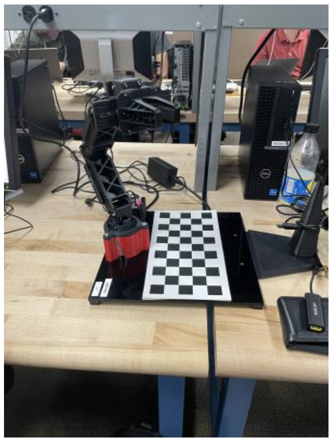


Figure 14: Photo and 3D live plot image of the 4-DOF in the position of the second vertex of the triangle formed by three arbitrary poses of the robot arm. The X, Y, Z, and α of the end effector are [100, -125, 300, 0].

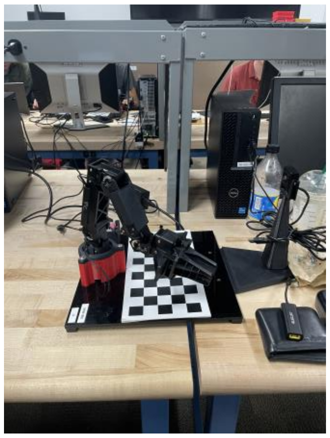
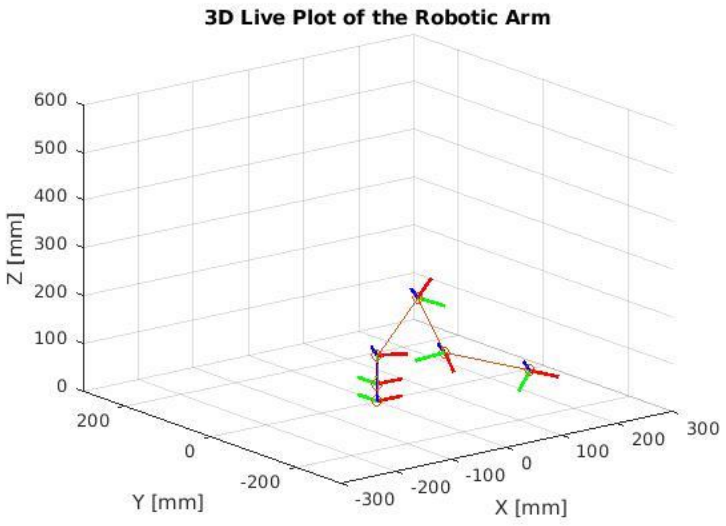



Figure 15: Photo and 3D live plot image of the 4-DOF in the position of the third vertex of the triangle formed by three arbitrary poses of the robot arm. The X, Y, Z, and α of the end effector are [200, -100, 50, 20].

The end effector was moved between the three vertices of the triangle as data was collected for the end effector position and the joint angles over time as the robot arm moved from vertex to vertex. The 3D plot of the motion of the end effector is shown in Figure 16 below. The path the end effector makes between the three vertices of the triangle is not a perfectly planar path. Each side of the triangle is curved quite a bit and the whole path shape appears curved like a triangular slice out of the surface of a sphere.

The joint angles as well as the X, Y, Z coordinates of the end effector were also collected over time as the robot arm moved between poses in the task space The two plots showing the joint angles over time and the end effector over time are shown below in Figure 17. The joint angle and end effector position values over time show smooth curves which indicates relatively smooth motion of the robot arm and end effector itself.

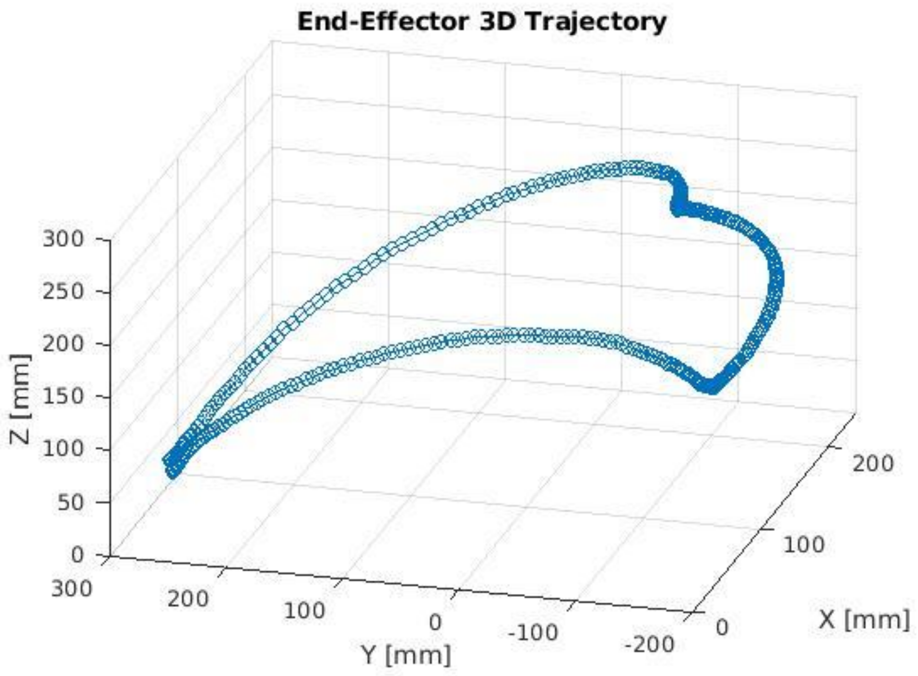


Figure 16: 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle in the robot arm's workspace.

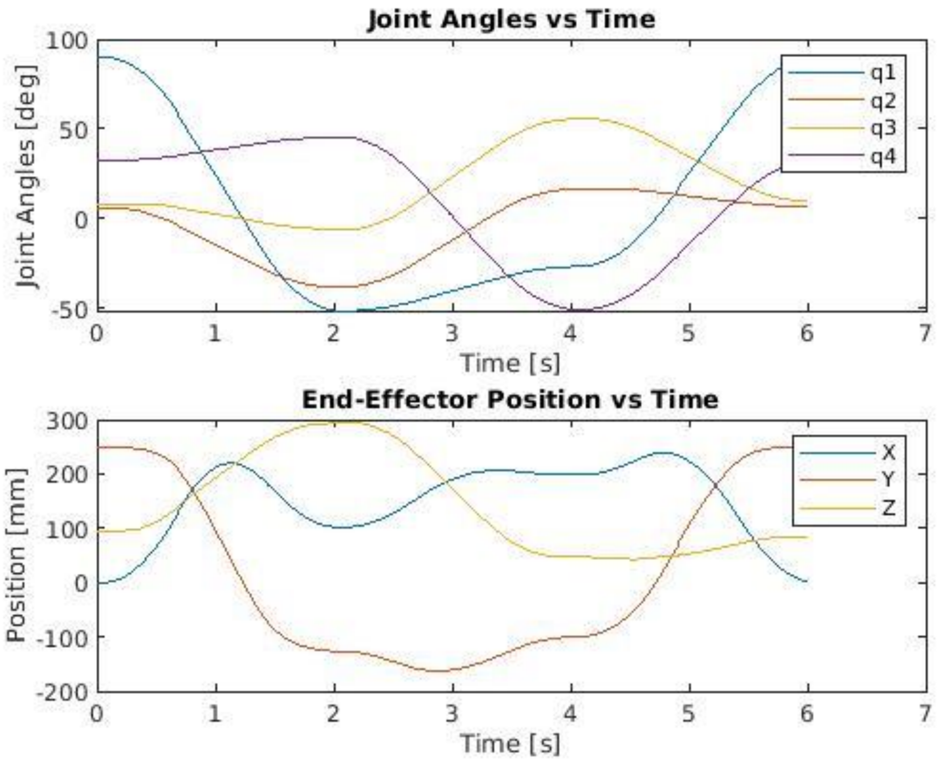


Figure 17: Two subplots of the joint angles of the robot arm plotted against time and the end effector position separated into X, Y, and Z position plotted against time over the whole movement of the robot arm through three arbitrary poses.

Trajectory Planning in Joint space

There was a new MATLAB class created called traj_planner() with methods cubic_traj() and quintic_traj() that solve for 3rd and 5th order polynomials between two via-points respectively. In the Robot class in MATLAB, there was a method called run_trajectory() written that can follow a trajectory based on the given trajectory coefficients and the current time.

The run_trajectory() method was used to move between the three poses in the triangle that was traversed in the inverse kinematic verification section of the lab. The same data that was collected in Figures 16-17 was collected when using the run_trajectory() method. The below graph shows the 3D plot of the motion of the end effector while moving between vertices of the triangle with run_trajectory(), figure 18. Each path between the vertices is somewhat curved slightly less than the 3D plot from Figure 16, but the curves were much less smooth. At the vertices of the triangle, there was a lot of correction and movement in the end effector as can be seen in the top right of the 3D plot in Figure 18 at the vertex of the triangle.

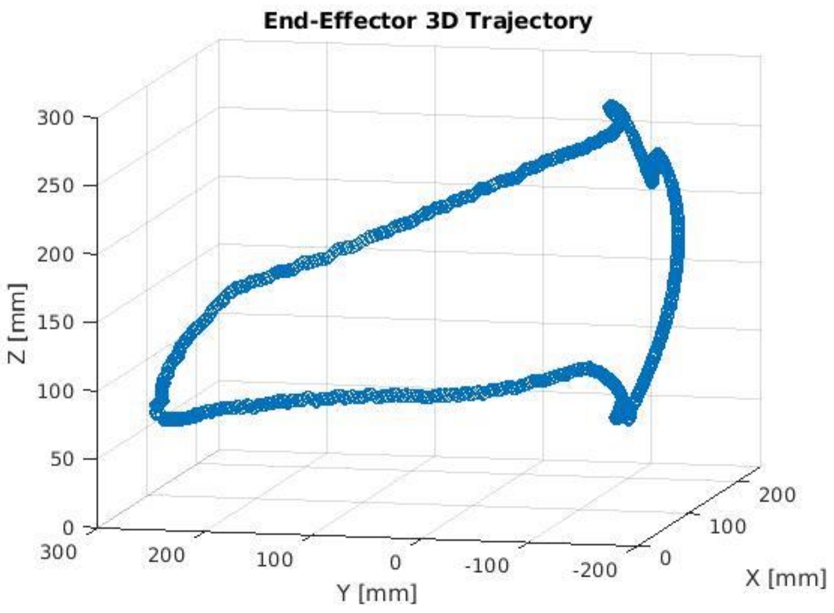


Figure 18: 3D plot of the end effector position as the robot arm was moved using the run_trajectory() method between the three arbitrary poses chosen to form a triangle in the robot arm's workspace.

The joint angle position and end effector position were recorded as the robot arm was moved between the three sides of the triangle, as well as the end effector velocity and acceleration in the X, Y, and Z directions. Figure 19 shows the joint angle positons for each of the four joints of the robot arm plotted against time. Figure 20 below shows the end effector position, velocity, and acceleration (in the X, Y, and Z directions) each graphed in a subplot against time.

The joint angles over time follow a similar shape to the joint angle paths shown in figure 17, but less smooth curves and more jumps in the otherwise predictable joint angle line over time.

The plots in figure 20 also seem less smooth when compared to the X, Y, Z, end effector position values over time when measured with the ik3001 function. There are some bumps in the position paths over time similarly random like in the graph in figure 19. The velocity and acceleration plots for the each of the x, y, and z directions over time increase each have significantly more noise than the positon plot. There doesn't seem to be any pattern to the noise although in the velocity graph, there is a section of the plot that seems more unstable than the rest of the plot just before 30s. The sudden change in velocity could relate to the jump at the vertex that can be seen in the 3D plot of the ende effector in Figure 18.

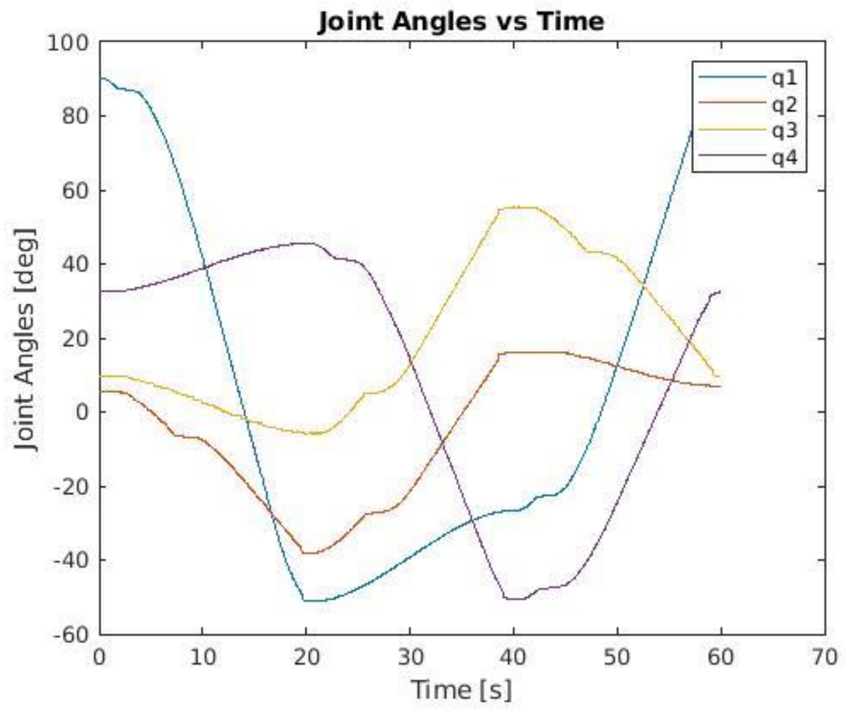


Figure 19: A plot of the joint angles of the robot arm against time over the whole movement of the robot arm through three arbitrary poses.

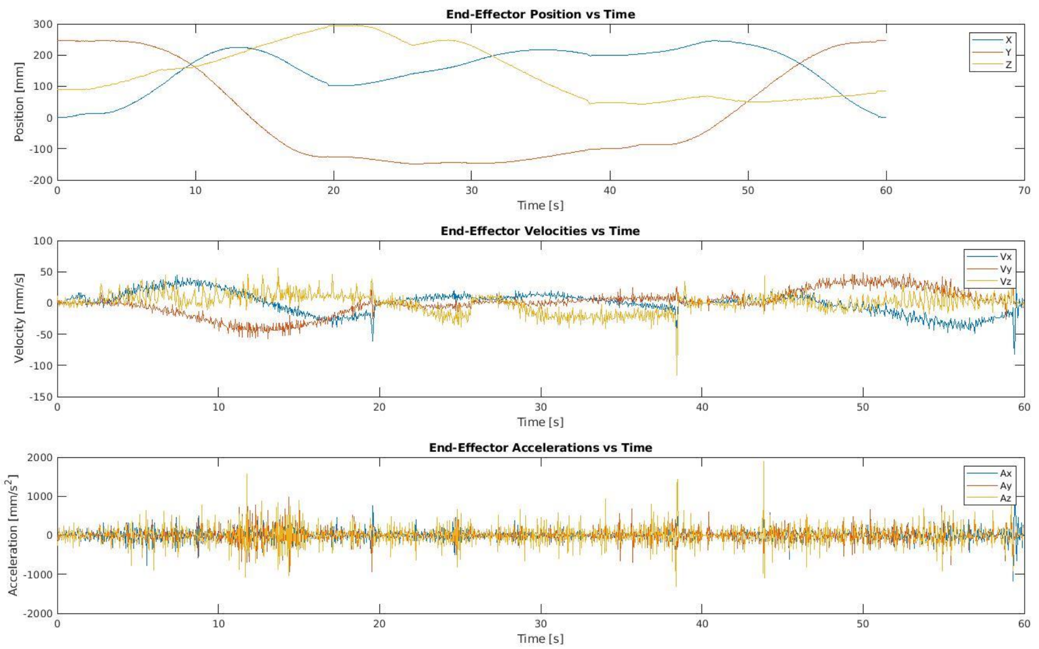


Figure 20: A plot of the end effector position separated into X, Y, and Z positions plotted against time over the whole movement of the robot arm through three arbitrary poses.

Trajectory Planning in Task Space

The cubic polynomial trajectory planning in the run_trajectory function was extended to operate in task space, allowing the robot to follow a smooth path between the same triangular set of points previously used in joint space. The run_trajectory() function was modified to handle task space trajectories, converting Cartesian positions into joint angles via the ik3001() function in real time.

Once again, the same arbitrary points were used to form the same triangle used in the previous sections and the same 3D model and joint and end-effector subplots for the , y and z positions (in mm) vs time (in seconds), x, y and z velocities (in mm/s) vs time (in seconds), x, y, and z accelerations (in mm/s^2) vs time (in seconds) were plotted from the data collected as the robot arm moved between the 3 points with the extended run_trajectory function.

The 3D plot of the task space trajectory shown in figure 21 revealed that the robot's end-effector followed a more precise and smoother path compared to the joint space trajectory. This was evident from the smoother curves observed in the 3D plot, indicating that cubic interpolation in task space provided better path fidelity when moving between points.

The plot of the joint angles of the robot over time shown in figure 22 showed consistent and gradual transitions between each vertex of the triangle. The position plots displayed smoother transitions with minimal overshoot or oscillation, which suggests that the cubic trajectory planning effectively mitigated abrupt changes in motion.

The end effector position, velocity, and acceleration plots over time revealed that the motion in task space was much smoother than the motion using the run_trajectory() method in the joint space. The position plot of the end effector over time shown as a subplot in figure 22 shows very smooth curves as the end effector moved through the workspace.

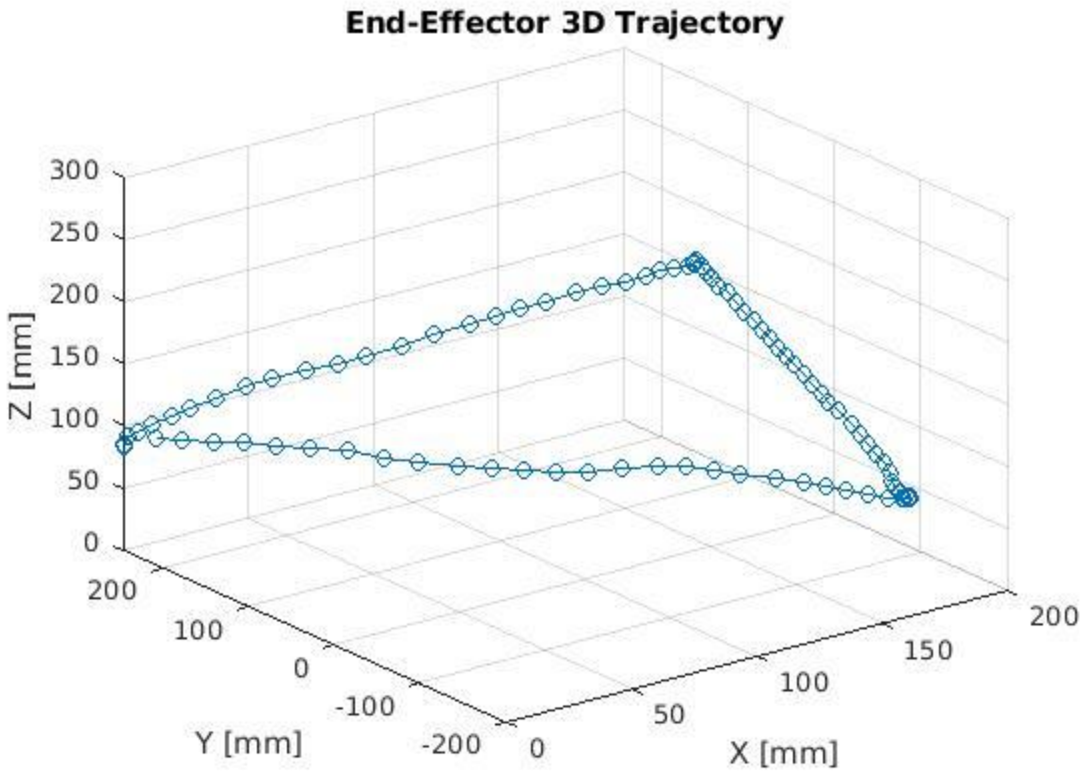


Figure 21: 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle in the robot arm's workspace using the extended task space run_trajectory method in MATLAB.

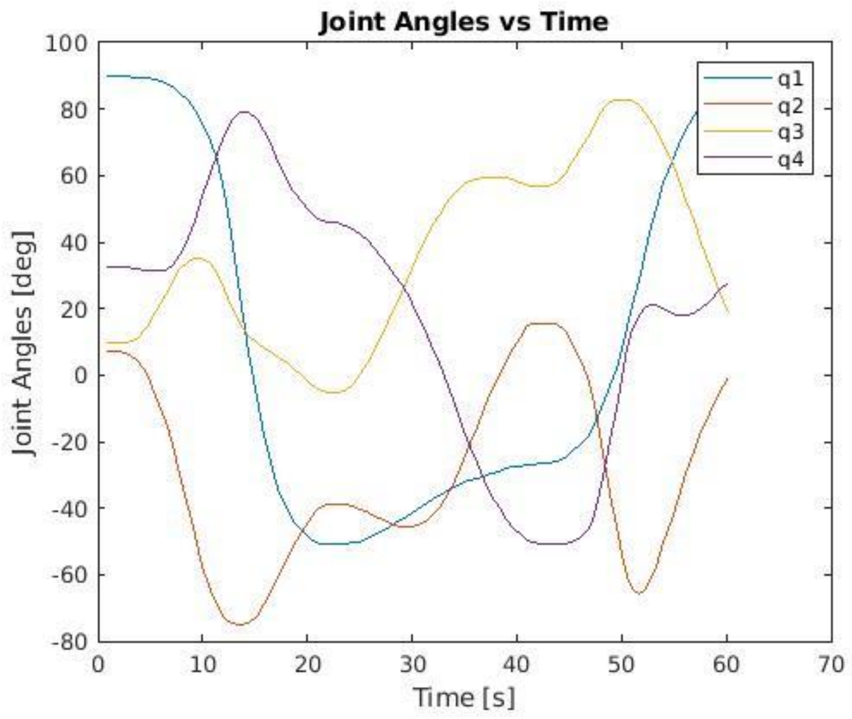


Figure 22: A plot of the joint angles of the robot arm against time over the whole movement of the robot arm through three arbitrary poses.

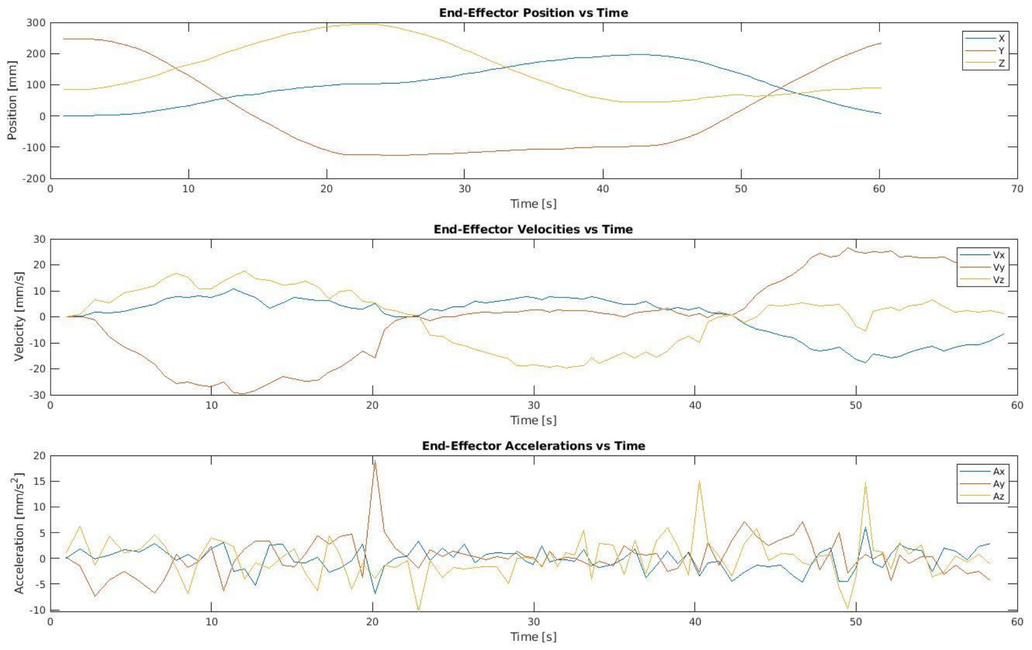


Figure 23: Three subplots of the end effector position, velocity, and acceleration separated into X, Y, and Z positions plotted against time, over the whole movement of the robot arm through three arbitrary poses.

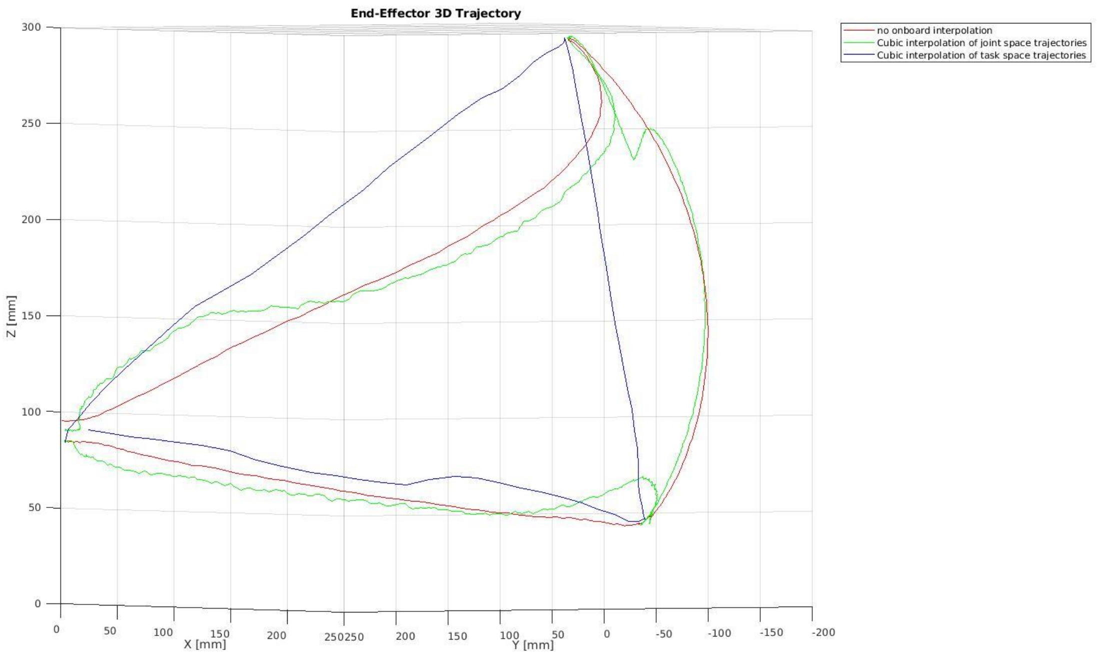


Figure 24: Front view of the 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle without onboard interpolation, cubic interpolation of the joint space trajectory, and cubic interpolation of the task space trajectory in the robot arm's workspace.

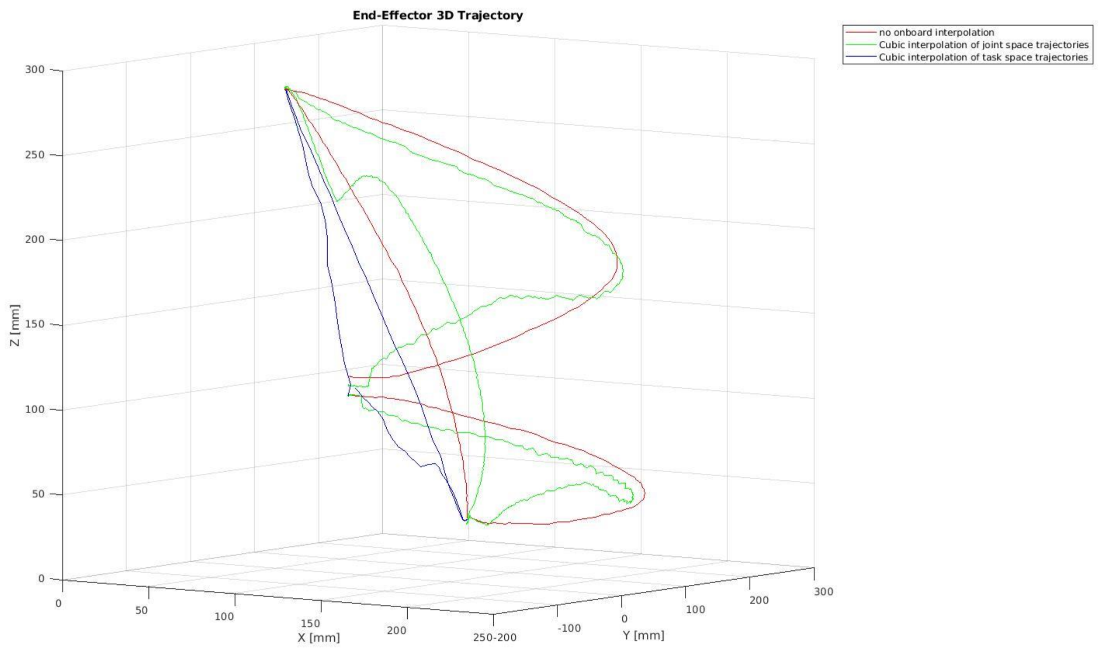


Figure 25: Side view of the 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle without onboard interpolation, cubic interpolation of the joint space trajectory, and cubic interpolation of the task space trajectory in the robot arm's workspace.

Quintic Trajectory Planning

Finally, the same process as above was repeated with the quintic trajectory planning instead of the cubic trajectory planning. The plots as outlined above can be seen below. Figure 26 shows the 3D plot of the end effector position over time as the robot arm moved between the three vertices of the triangle. Figure 27 shows the joint angles for each of the joints in the robot arm plotted over time. Figure 28 shows the end effector position, velocity, and acceleration in the x, y, and z planes in (mm) plotted against time.

The quintic trajectory was much smoother than the cubic. As seen in the 3D plot in Figure 26, the triangle was much more stable than in Figure 18 when run_trajectory() was run with the cubic_traj method, even though the triangle with the quintic trajectory planning has significantly fewer via-points.

The joint angle plot over time shown in figure 27 looks very similar to the previous smooth joint angle plots and the end effector position plot as a subplot in figure 28 also has smooth curves like previous plots. These smooth lines and the lack of jumps and random error and noise indicate the effectiveness and precision of the quintic trajectory planning method and its superiority to the cubic method.

The velocity and acceleration plots also seen in figure 28 show a much reduced level of noise and random spikes and changes in the X, Y, or Z value compared to the cubic and even task space run_trajectory functions (plots seen in figures 20 and 23 respectively).

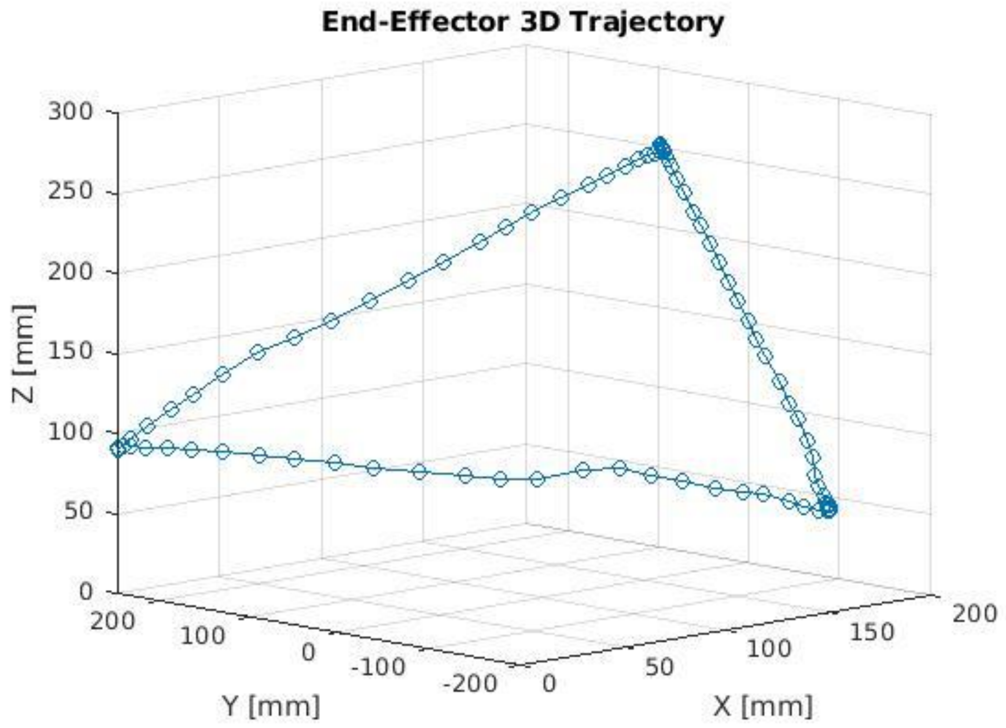


Figure 26: 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle in the robot arm's workspace.

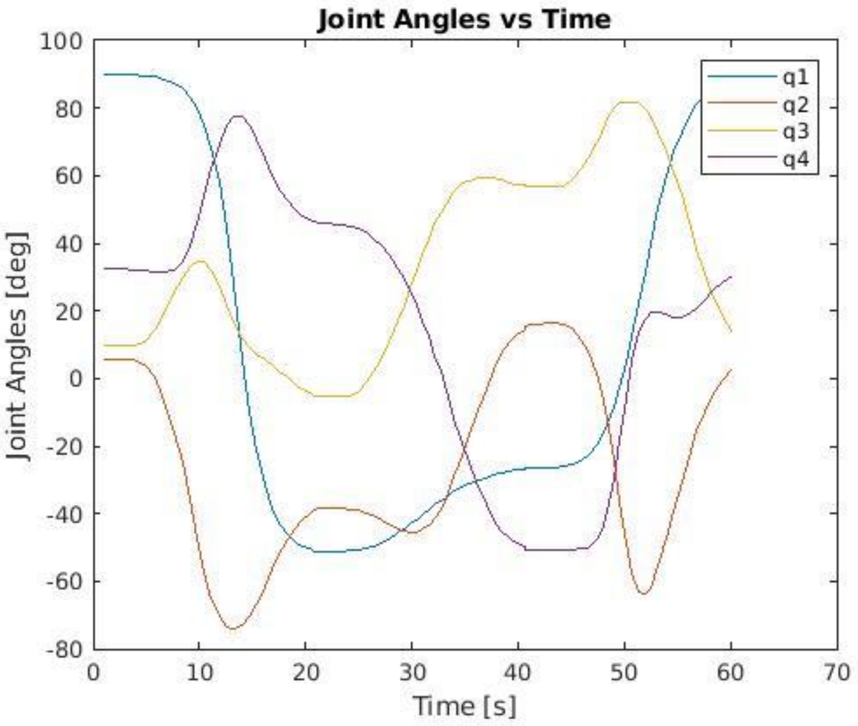


Figure 27: A plot of the joint angles of the robot arm against time over the whole movement of the robot arm through three arbitrary poses.

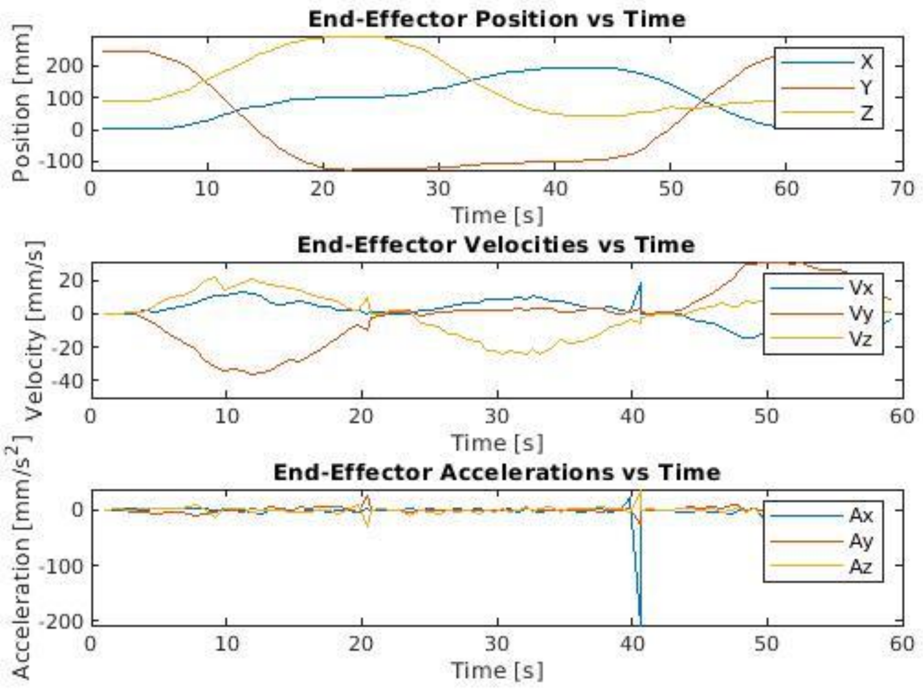


Figure 28: Three subplots of the end effector position, velocity, and acceleration separated into X, Y, and Z positions plotted against time, over the whole movement of the robot arm through three arbitrary poses.

Discussion

The implementation of inverse kinematics (IK) and trajectory generation for the robot arm was successful, demonstrating both accurate end effector positioning and smooth motion control.

The geometric approach to IK allowed us to solve for the joint angles required to achieve the desired end effector pose. By directly computing each angle step by step we were able to account for the robot's specific link geometry and constraints the use of MATLAB’s atan2 function ensured that both positive and negative solutions were considered, allowing the robot to reach all accessible poses.

Validation through forward kinematics: we validated our IK implementation by comparing our calculated joint angles against forward kinematics outputs. This cross-verification confirmed that the robot could reliably move to target positions with minimal error checking the effectiveness of our IK calculations.

One challenge was managing multiple valid configurations of each pose. Our checkIKangles() function successfully filtered these solutions selecting the most accurate set of joint angles based on the smallest deviation from the desired end effector.

The trajectory generation methods implemented in the Traj_Planner class, specifically the cubic and quintic polynomial methods, provided a smooth and controlled way to move the robot between points. When comparing the cubic and quintic trajectories, the quintic trajectory showed superior performance in terms of smoothness and control. The quintic trajectory reduced the mechanical stress on the robot’s joints by ensuring smoother acceleration and deceleration, which is particularly beneficial for high-precision tasks or when operating at higher speeds.

Both task space and joint space trajectories were tested. Joint spaces allowed for direct control over each angle, while task space required additional IK calculations at each point

The robot's ability to accurately follow a triangular path validated our trajectory generation and IK functions. Plots of the joint angles and end effector position showed smooth and continuous movement with minor deviations.

Another challenge was getting the correct time for the run_trajectory() method as it continuously used subs() method to substitute the time in the joint angles and the co-ordinate position functions. Moreover, while running in the task space, the method had to continuously call upon ik3001() method which was relatively slower than the rest of the functions. Hence, we had to increase the move_time from one point to another to reduce this error. Though, this didn’t completely omit the error, it significantly reduced it; however, it also meant that the arm slowed down causing some jerking motion to take place. This means that the jerkiness that can be seen in the 3D graphs of the end effector position could be a result of timing in the code. For further optimization, the timing code could be reworked and tested to achieve a smoother and more reliable run_trajectory function for both the cubic and quintic methods.

Conclusion

This lab demonstrates the application of inverse kinematic and trajectory generation techniques used with the 4-DOF OpenManipulator-X robot arm. The functions developed in the lab allowed for precise computation of the necessary joint values for each of the 4 robot arm joints to effectively move the robot arm, so the end effector reliably reaches the input task space coordinates and wrist angle. The calculations and methods written lay the foundation for future tasks that require specific and precise movements of the end effector arm to specified locations to solve various tasks.

The lab involved the creation of several key methods for solving the inverse kinematics based on the function input and two types of trajectory generation. The ik3001() method was implemented in the Robot MATLAB class to calculate the joint angles necessary for a given task space position and orientation. Additionally, cubic and quintic trajectory planning methods in a new class called Traj_planner were developed to ensure smooth joint motion between waypoints, both in joint space and task space. These methods were integrated with the existing forward kinematics function fk3001() to verify the accuracy of the inverse kinematics solution.

A comprehensive testing and validation process was conducted. In order to test, the robot arm was commanded to move between three arbitrary poses selected to create a triangular path for the end effector. The motion of the robot pose was visualized through a 3D stick model (developed in Lab 2), and the joint angles and end-effector positions were plotted over time and analyzed. The results from these visualizations are crucial for understanding the robot arm's behavior and for verification will be valuable in future task that need to use trajectory planning and inverse kinematics calculations to facilitate efficient and precise control of the end effector within the robot's workspace.