WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Lab 4 – Velocity Kinematics

SUBMITTED BY
Nicolas Graham
Pranav Jain
Fiona Prendergast

Date Submitted : 09.19.2024
Date Completed: 09.26.2024
Course Instructor: Prof. Agheli
Lab Section: RBE 3001 A'24

Abstract

This report covers the implementation of Velocity kinematics (VK) on the OpenManipulator-X robot arm by using the Jacobian matrix in MATLAB. In lab 3 we focused on using the Jacobian to calculate the forward velocity kinematics, calculated inverse velocity kinematics, executed velocity-based commands for trajectory following with real time adjustments. Successfully computed task space velocities form joint velocities, detected singularity's and avoided these configurations, accurate trajectory tracking though velocity-based control, validated results using live, 3d and 2d plotting.

Introduction

In Labs 1, 2, and 3, we established communication with the robotic arm, implemented forward kinematics, and developed inverse kinematics and trajectory generation. In Lab 4, we focus on velocity kinematics, known as differential kinematics. The goal of this lab was to implement the Jacobian matrix to calculate task-space velocities from joint velocities.

The Jacobian plays an important role in velocity kinematics as it gives us linear mapping between joint and task-space velocities. It translates joint movements into EE velocities, both linear and angular. When using the Jacobian, it is important to detect singularities, these are configurations where the EE position can be achieved by an infinite number of joint configurations, leading to loss of control and unpredictable motion.

Methodology

1) Calculate the Jacobian

There are two ways to calculate the Jacobian, one using differentiation, and one using the cross product. For our manual calculations we decided to write out the method by hand and use MATLAB for the rest.

For the partial differentiation method, the first step was to calculate the homogeneous transformation matrix from the base of the robot to the end effector. The next step was to use the position, the last column in the HT matrix and take the partial derivative with respect to each of the joint variables to solve for the upper half of the Jacobian matrix. Each column represents one joint of the robot arm and to solve for each row the X, Y, Z position of the end effector are solved by partial derivatives with each theta value respectively.

For the cross-product method, the HT matrix from the base to the end effector is also used as P_e and for each joint respectively the HT matrix from the base to that joint only is also calculated. The difference between the position matrix for the joint (P_i) and P_e are taken and the cross product with the last column of the transformation matrix (Z_i) for each of the joints is found for the upper half of the Jacobian matrix.

2) Implement Jacobian calculation in MATLAB as jacob3001()

We implemented a method for both the differentiation method and the cross-product method, this way we can decide which one we prefer, and which works better in our application.

3) **Validate your Jacobian calculation**
   Verified the Jacobian calculation by checking known configurations against the results from Jacobian.
4) **Calculate the forward velocity kinematics in MATLAB**

Given the joint velocity's using the formula v = J*q we computed task space velocity's and implemented fdk3001() function which translates joint velocities to EE linear and angular velocities. These results were validated by comparing the expected vs actual velocities when running through controlled movements.

5) **Live plot of the task-space velocity vector**

Using the same vertices as the previous lab, an arbitrary triangle was created in the task space. This involved calculating trajectories for each cartesian component x, y z separately and applying inverse kinematics to convert these into joint angles. The plot_arm() method in the Model() class was extended to display the velocity vector of the end-effector by using the linear velocity matrix of fdk3001() method, where the length of the vector was proportional to the magnitude of the speed of the end-effector.

6) **Discover and avoid singularities**

A new method named isSingularity() is created to calculate the determinant of the Jacobian in real time to make sure that the arm was never close to a singularity, returning true if the end-effector of the robotic arm is close to a singularity. This method is called inside the run_trajectory() method which stops the robotic arm from moving and displays an error.

7) **Extra Credit: Velocity-based motion planning in task space**

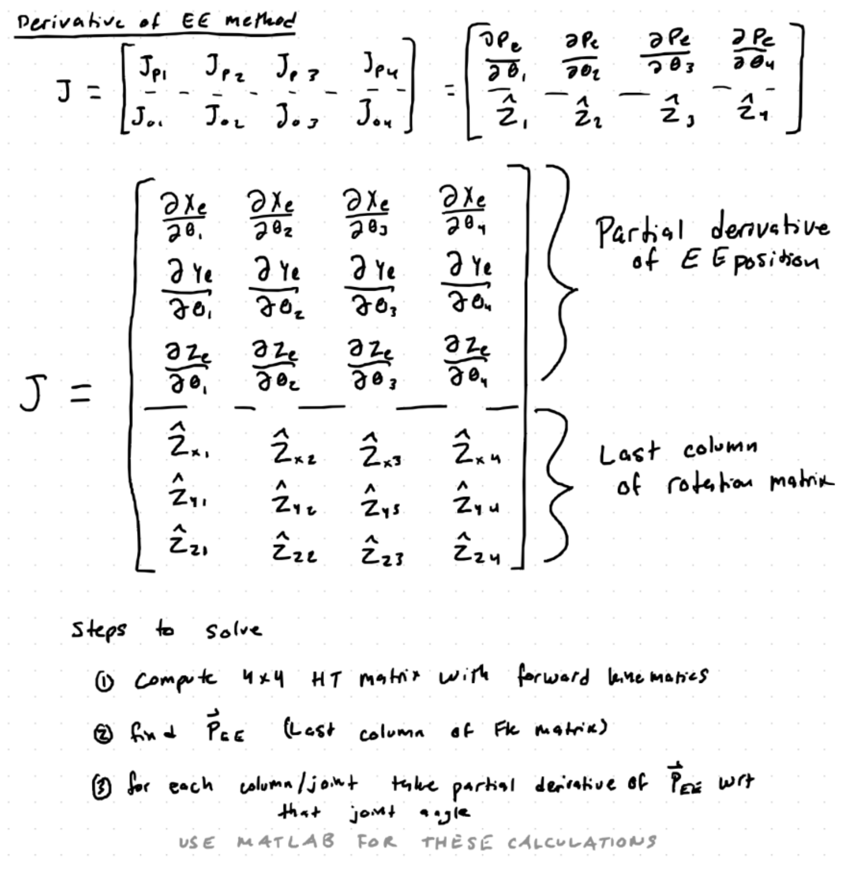
The same vertices are used again to do velocity control. A unit vector is created towards the target position which is multiplied with an arbitrary speed of 34000 to get the linear velocity vector. The Jacobian for the current pose is also calculated and pseudo-inverse of top half of the Jacobian is calculated. This pseudo-inverse is multiplied by the linear velocity vector to get the desired velocity. To reduce any jitter in the robotic arm, a low-pass filter is added to the desired velocity. The desired velocity is multiplied by a small change in time of 0.02 and the next position of the robotic arm is calculated and sent to via servo_jp(). This process is repeated until the robotic arm is within a tolerance of 2.1 away from the target position. This method is repeated 3 times for each side of the triangle.

Results

1) Calculate the forward velocity kinematics

The forward velocity kinematics can be calculated with the differentiation method and with the cross-product method. In this lab, two functions were created to calculate the forward velocity kinematics by using each method respectively. Since the calculations are so math intensive, the outline of the steps was written out shown in figures 1 and 2 below.

The MATLBAB class FK was created and the file Lab4_part1 has more in-depth calculations for each type of Jacobian calculation. For the sake of space and being only tangentially related to the rest of the lab past the code implementation we did not include those calculations step by step in this lab document.



Derivative of EE method

$$J = \begin{bmatrix} J_{p1} & J_{p2} & J_{p3} & J_{p4} \\ \hline J_{o1} & J_{o2} & J_{o3} & J_{o4} \end{bmatrix} = \begin{bmatrix} \frac{\partial P_e}{\partial \theta_1} & \frac{\partial P_e}{\partial \theta_2} & \frac{\partial P_e}{\partial \theta_3} & \frac{\partial P_e}{\partial \theta_4} \\ \hline \hat{z}_1 & \hat{z}_2 & \hat{z}_3 & \hat{z}_4 \end{bmatrix}$$

$$J = \begin{bmatrix} \frac{\partial x_e}{\partial \theta_1} & \frac{\partial x_e}{\partial \theta_2} & \frac{\partial x_e}{\partial \theta_3} & \frac{\partial x_e}{\partial \theta_4} \\ \frac{\partial y_e}{\partial \theta_1} & \frac{\partial y_e}{\partial \theta_2} & \frac{\partial y_e}{\partial \theta_3} & \frac{\partial y_e}{\partial \theta_4} \\ \frac{\partial z_e}{\partial \theta_1} & \frac{\partial z_e}{\partial \theta_2} & \frac{\partial z_e}{\partial \theta_3} & \frac{\partial z_e}{\partial \theta_4} \\ \hline \hat{z}_{x1} & \hat{z}_{x2} & \hat{z}_{x3} & \hat{z}_{x4} \\ \hat{z}_{y1} & \hat{z}_{y2} & \hat{z}_{y3} & \hat{z}_{y4} \\ \hat{z}_{z1} & \hat{z}_{z2} & \hat{z}_{z3} & \hat{z}_{z4} \end{bmatrix} \begin{matrix} \Big\} \text{Partial derivative of EE position} \\ \Big\} \text{Last column of rotation matrix} \end{matrix}$$

Steps to solve

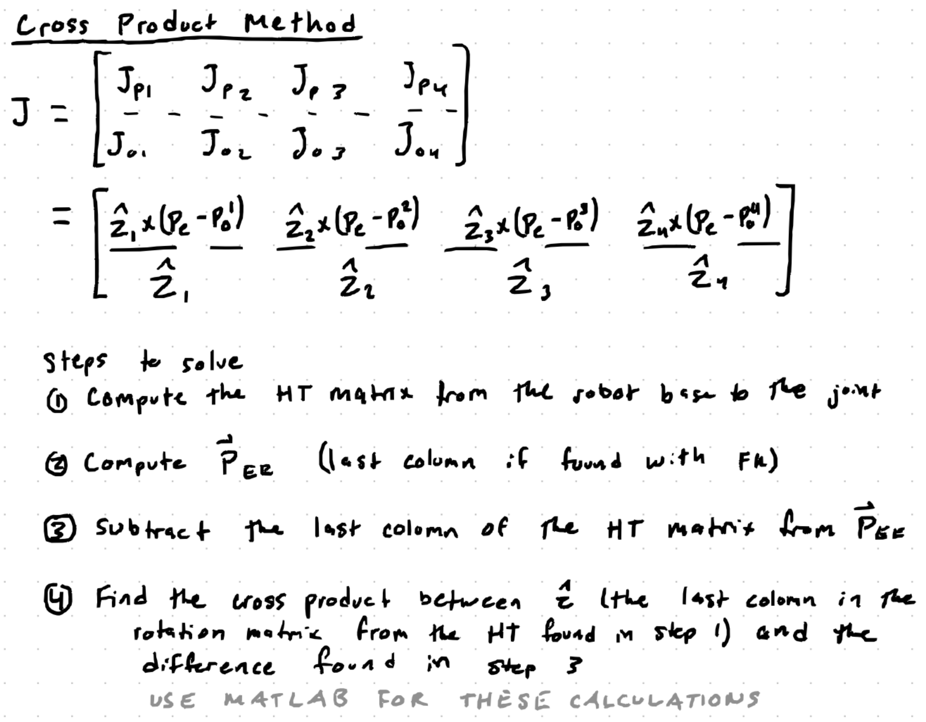
① Compute 4x4 HT matrix with forward kinematics

② find $\vec{P}_{EE}$ (Last column of FK matrix)

③ for each column/joint take partial derivative of $\vec{P}_{EE}$ wrt that joint angle

USE MATLAB FOR THESE CALCULATIONS

Figure 1: Handwritten method outline for calculating the Jacobian matrix for the 4 DOF robot arm using the partial derivative method.



Cross Product Method

$$J = \begin{bmatrix} J_{p1} & J_{p2} & J_{p3} & J_{p4} \\ \hline J_{o1} & J_{o2} & J_{o3} & J_{o4} \end{bmatrix}$$

$$= \begin{bmatrix} \hat{z}_1 \times (P_e - P_0^1) & \hat{z}_2 \times (P_e - P_0^2) & \hat{z}_3 \times (P_e - P_0^3) & \hat{z}_4 \times (P_e - P_0^4) \\ \hline \hat{z}_1 & \hat{z}_2 & \hat{z}_3 & \hat{z}_4 \end{bmatrix}$$

steps to solve

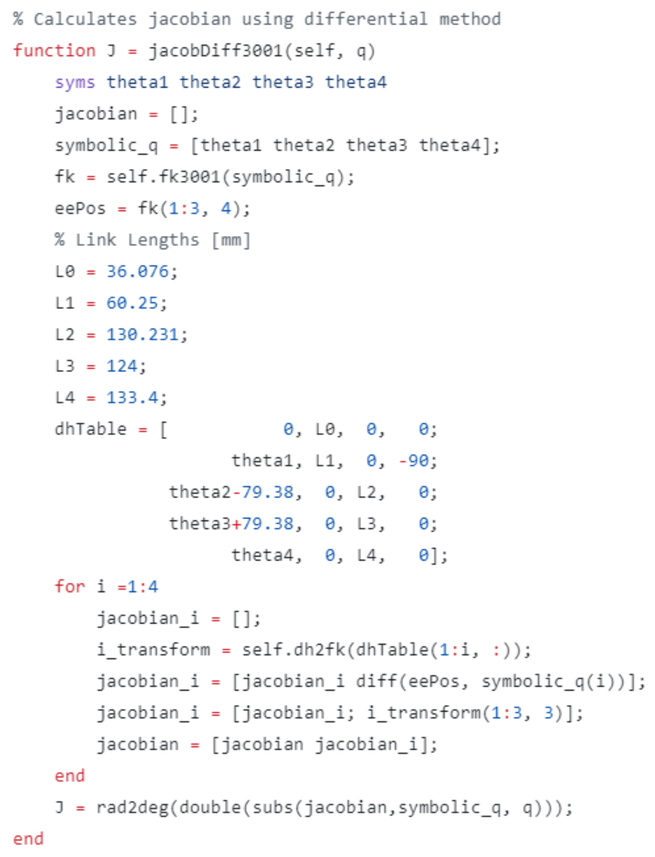
1. Compute the HT matrix from the robot base to the joint
2. Compute $\vec{P}_{EE}$ (last column if found with FK)
3. Subtract the last column of the HT matrix from $\vec{P}_{EE}$
4. Find the cross product between $\hat{z}$ (the last column in the rotation matrix from the HT found in step 1) and the difference found in step 3

USE MATLAB FOR THESE CALCULATIONS

Figure 2: Handwritten method outline for calculating the Jacobian matrix for the 4 DOF robot arm using the cross-product method.

2) Implement Jacobian calculation in MATLAB as jacob3001()

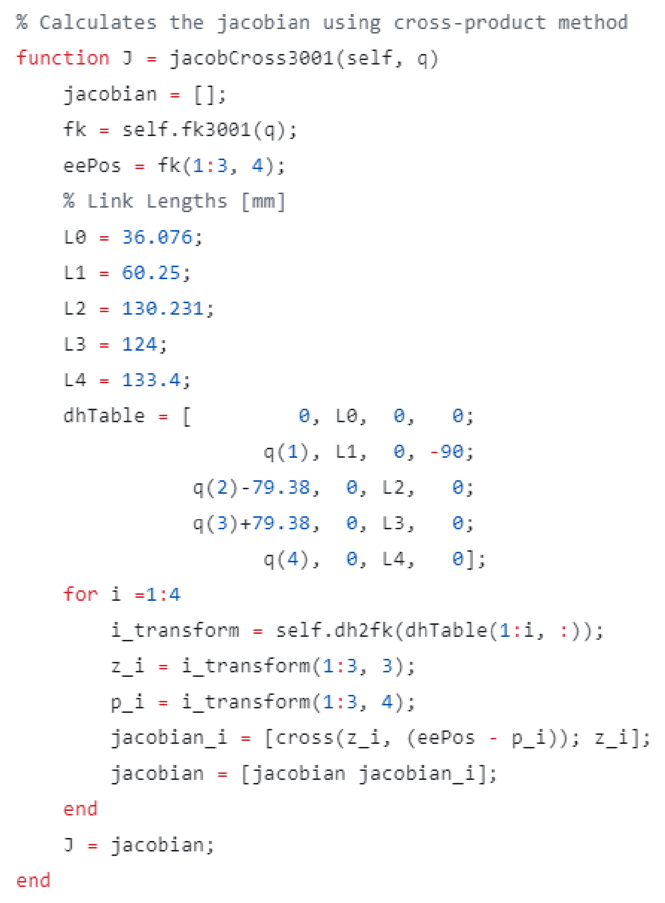
The implementation of the jacobian calculations in matlab were done through two matlab methods: jacobDiff3001() and jacobCross3001(). The methods are shown and described in the figures below.



```
% Calculates jacobian using differential method
function J = jacobDiff3001(self, q)
    syms theta1 theta2 theta3 theta4
    jacobian = [];
    symbolic_q = [theta1 theta2 theta3 theta4];
    fk = self.fk3001(symbolic_q);
    eePos = fk(1:3, 4);
    % Link Lengths [mm]
    L0 = 36.076;
    L1 = 60.25;
    L2 = 130.231;
    L3 = 124;
    L4 = 133.4;
    dhTable = [           0, L0,  0,   0;
                     theta1, L1,  0, -90;
               theta2-79.38,  0, L2,   0;
               theta3+79.38,  0, L3,   0;
                     theta4,  0, L4,   0];
    for i =1:4
        jacobian_i = [];
        i_transform = self.dh2fk(dhTable(1:i, :));
        jacobian_i = [jacobian_i diff(eePos, symbolic_q(i))];
        jacobian_i = [jacobian_i; i_transform(1:3, 3)];
        jacobian = [jacobian jacobian_i];
    end
    J = rad2deg(double(subs(jacobian,symbolic_q, q)));
end
```

Figure 3: MATLAB code function written to calculate the Jacobian matrix for the robot arm using differentiation.

Figure 3 shows the calculations for the differential method. Here, the top half of the Jacobian is calculated by finding the differentiating the symbolic end-effector position by its corresponding theta value. For the lower half of the Jacobian, as all the joints are revolute joints, the zth column of the transformation matrix is determined. Using the dhTable, the 0th to ith transformation matrix is calculated and the zth column (3rd column) is the corresponding angular velocity.



```matlab
% Calculates the jacobian using cross-product method
function J = jacobCross3001(self, q)
    jacobian = [];
    fk = self.fk3001(q);
    eePos = fk(1:3, 4);
    % Link Lengths [mm]
    L0 = 36.076;
    L1 = 60.25;
    L2 = 130.231;
    L3 = 124;
    L4 = 133.4;
    dhTable = [         0, L0,  0,   0;
                     q(1), L1,  0, -90;
              q(2)-79.38,  0, L2,   0;
              q(3)+79.38,  0, L3,   0;
                     q(4),  0, L4,   0];
    for i =1:4
        i_transform = self.dh2fk(dhTable(1:i, :));
        z_i = i_transform(1:3, 3);
        p_i = i_transform(1:3, 4);
        jacobian_i = [cross(z_i, (eePos - p_i)); z_i];
        jacobian = [jacobian jacobian_i];
    end
    J = jacobian;
end
```

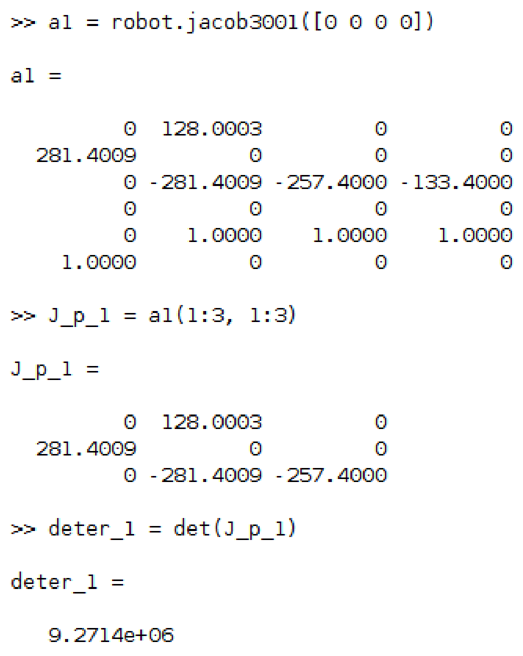
Figure 5: MATLAB code function written to calculate the Jacobian matrix for the robot arm using the cross product method.

This shows the calculations for the cross-product method. Here, the top half of the Jacobian is calculated by z_i - (eePos – p_i) as all the joints are revolute joints. For the lower half of the Jacobian, as all the joints are revolute joints, the zth column of the transformation matrix is determined. Using the dhTable, the 0th to ith transformation matrix is calculated and the zth column (3rd column) is the corresponding angular velocity.

To increase the calculation speed, the Jacobian is converted into a matlabfunction to be used in other parts of the lab and in the future.

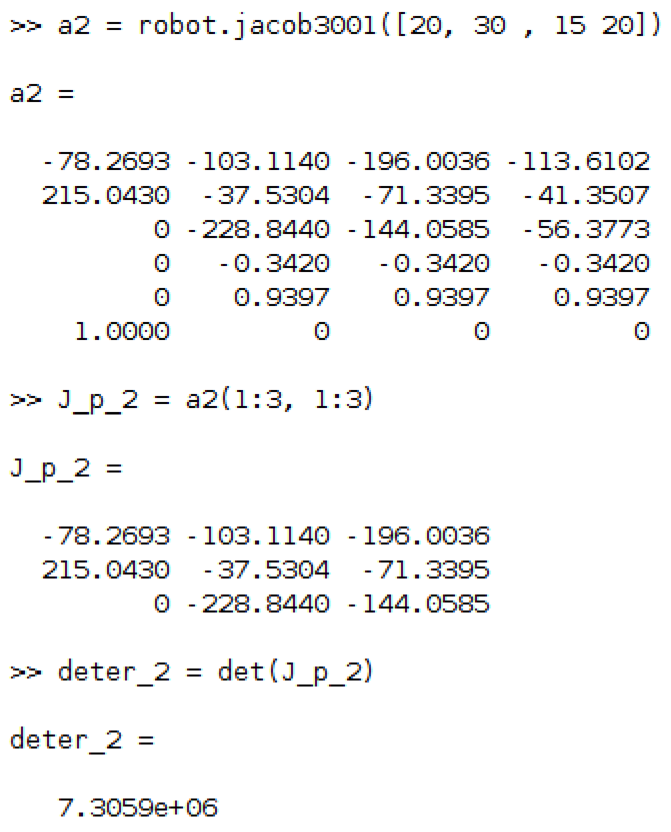
3) Validate your Jacobian calculation

In order to verify the Jacobian calculation, the jacob3001 function was run for several joint configurations that are known to produce singularities. For instance, at the configuration [0, -10.62, -79.38, 0], the determinant of the first three columns of the upper half of the Jacobian was found to be near zero, indicating a singularity. This result was expected and confirms the accuracy of the Jacobian matrix. Other Jacobians have a non-zero determinant, suggesting that these points aren’t at singularity.



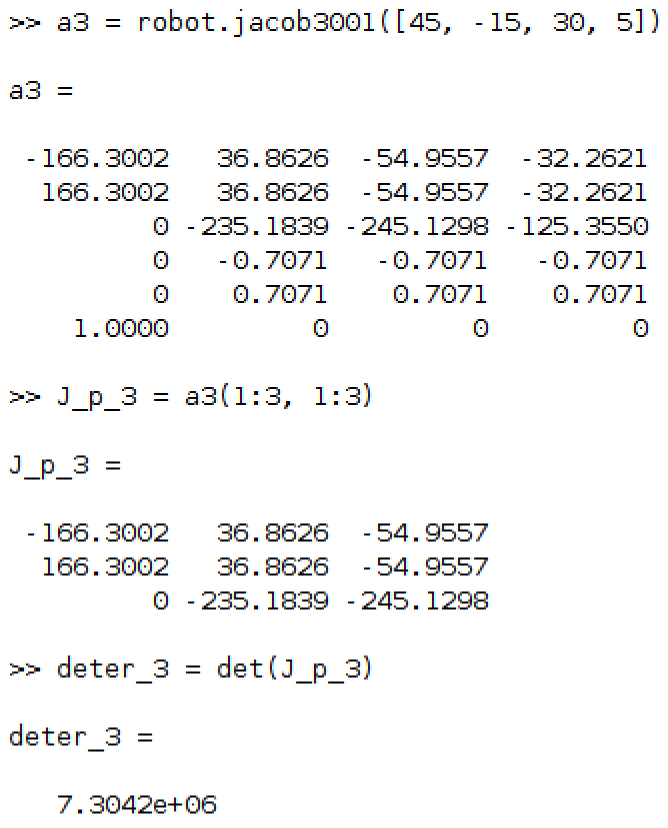
```
>> a1 = robot.jacob3001([0 0 0 0])

a1 =

         0  128.0003         0         0
  281.4009         0         0         0
         0 -281.4009 -257.4000 -133.4000
         0         0         0         0
         0    1.0000    1.0000    1.0000
    1.0000         0         0         0

>> J_p_1 = a1(1:3, 1:3)

J_p_1 =

         0  128.0003         0
  281.4009         0         0
         0 -281.4009 -257.4000

>> deter_1 = det(J_p_1)

deter_1 =

   9.2714e+06
```

Figure 5: Command window output from running jacob3001 and finding its determinant. The input for the jacob3001 method is the joint space values representing [0, 0, 0, 0].



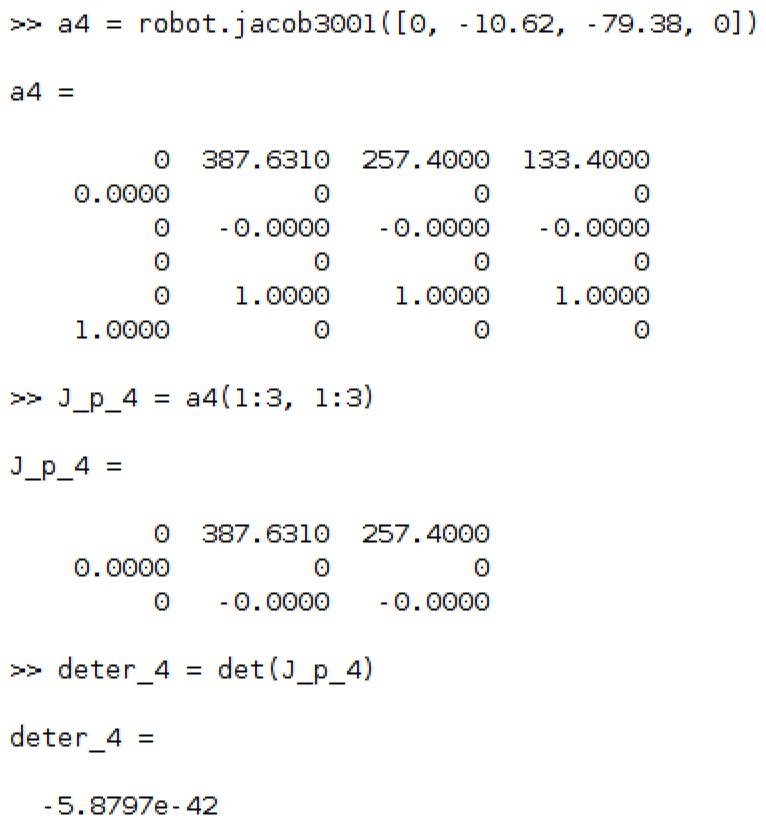
```
>> a2 = robot.jacob3001([20, 30 , 15 20])

a2 =

  -78.2693 -103.1140 -196.0036 -113.6102
  215.0430  -37.5304  -71.3395  -41.3507
         0 -228.8440 -144.0585  -56.3773
         0   -0.3420   -0.3420   -0.3420
         0    0.9397    0.9397    0.9397
    1.0000         0         0         0

>> J_p_2 = a2(1:3, 1:3)

J_p_2 =

  -78.2693 -103.1140 -196.0036
  215.0430  -37.5304  -71.3395
         0 -228.8440 -144.0585

>> deter_2 = det(J_p_2)

deter_2 =

   7.3059e+06
```

Figure 6: Command window output from running jacob3001 and finding its determinant. The input for the jacob3001 method is the joint space values representing [20, 30, 15, 20].



```
>> a3 = robot.jacob3001([45, -15, 30, 5])

a3 =

 -166.3002   36.8626  -54.9557  -32.2621
  166.3002   36.8626  -54.9557  -32.2621
         0 -235.1839 -245.1298 -125.3550
         0   -0.7071   -0.7071   -0.7071
         0    0.7071    0.7071    0.7071
    1.0000         0         0         0

>> J_p_3 = a3(1:3, 1:3)

J_p_3 =

 -166.3002   36.8626  -54.9557
  166.3002   36.8626  -54.9557
         0 -235.1839 -245.1298

>> deter_3 = det(J_p_3)

deter_3 =

   7.3042e+06
```

Figure 7: Command window output from running jacob3001 and finding its determinant. The input for the jacob3001 method is the joint space values representing [45, -15, 30, 5].



```
>> a4 = robot.jacob3001([0, -10.62, -79.38, 0])

a4 =

         0  387.6310  257.4000  133.4000
    0.0000         0         0         0
         0   -0.0000   -0.0000   -0.0000
         0         0         0         0
         0    1.0000    1.0000    1.0000
    1.0000         0         0         0

>> J_p_4 = a4(1:3, 1:3)

J_p_4 =

         0  387.6310  257.4000
    0.0000         0         0
         0   -0.0000   -0.0000

>> deter_4 = det(J_p_4)

deter_4 =

  -5.8797e-42
```

Figure 8: Command window output from running jacob3001 and finding its determinant. The input for the jacob3001 method is the joint space values representing [0, -10.62, -79.38, 0].

Here, the robotic arm is completely stretched out upwards which is a singularity point. The first column of the Jacobian is zero, which suggests that theta1 or the joint1 doesn't affect the end-effector velocity at that certain pose. As the robotic arm is at singularity point, the determinant is zero or very close to zero in this case.

4) Calculate the forward velocity kinematics in MATLAB

The forward velocity kinematics were successfully calculated using the Jacobian matrix derived in the earlier parts of the lab. The fdk3001 function, which computes the task-space velocities from joint velocities, was implemented in MATLAB. The accuracy of this function was validated by comparing the computed velocities with expected outcomes during the robot's movement. The results showed that the function accurately computed both linear and angular velocities of the end-effector, aligning with theoretical predictions. The output of the function was verified by using known matrices provided by lab staff, so they were a good indicator by sign off about the validity of the calculations and the accuracy of the function.

5) Live plot of the task-space velocity vector

The real-time plotting of the task-space velocity vector was implemented and visualized as the robot followed a trajectory through three arbitrary vertices in its workspace. Figures 9, 10, and

11 show the 3D live plots with the velocity vector of the end-effector as it moved between these vertices. The direction and magnitude of the velocity vectors were consistent with the planned trajectory, confirming that the forward velocity kinematics were correctly implemented. Additionally, Figure 13 presents subplots of the end-effector's velocity separated into X, Y, and Z components, as well as the magnitude of these velocities plotted against time. These plots demonstrate that the robot's motion followed the expected pattern, with the velocity vector changing direction and magnitude as the robot moved between the vertices.

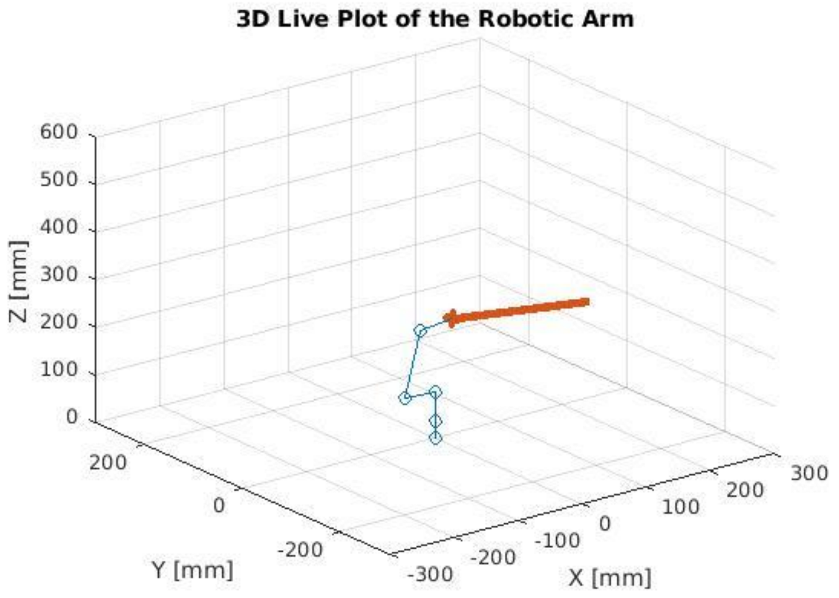


Figure 9: 3D Live plot with the velocity vector of the end-effector of the robotic arm while it is moving from the 1st to 2nd vertices.

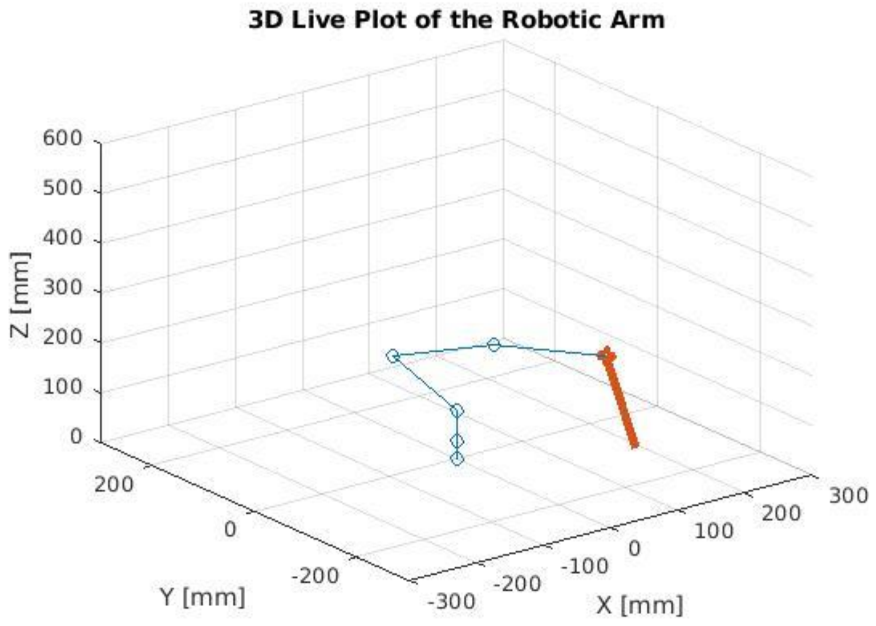


Figure 10: 3D Live plot with the velocity vector of the end-effector of the robotic arm while it is moving from the 2nd to 3rd vertices.

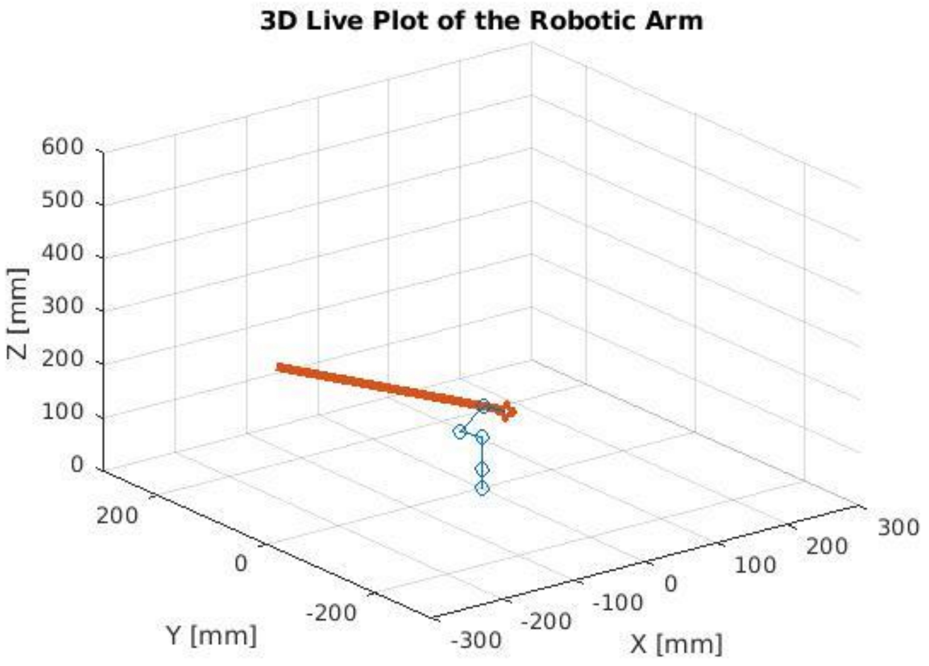


Figure 11: 3D Live plot with the velocity vector of the end-effector of the robotic arm while it is moving from the 3rd to 1st vertices.

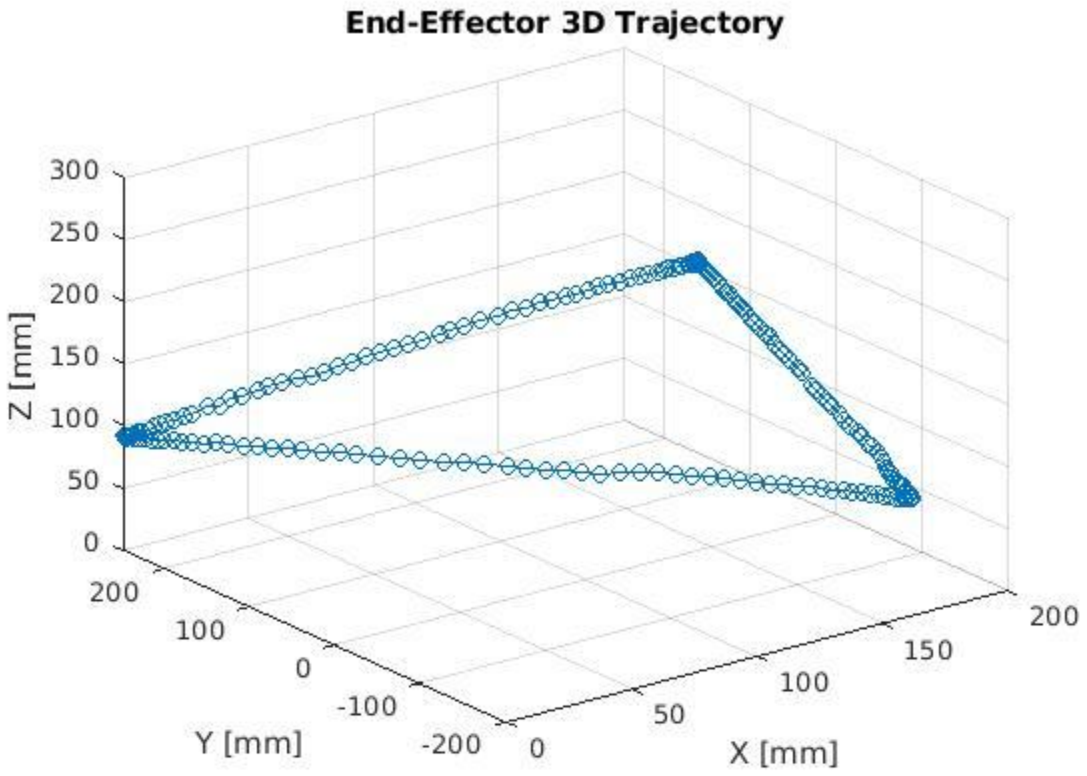


Figure 12: 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle in the robot arm's workspace.

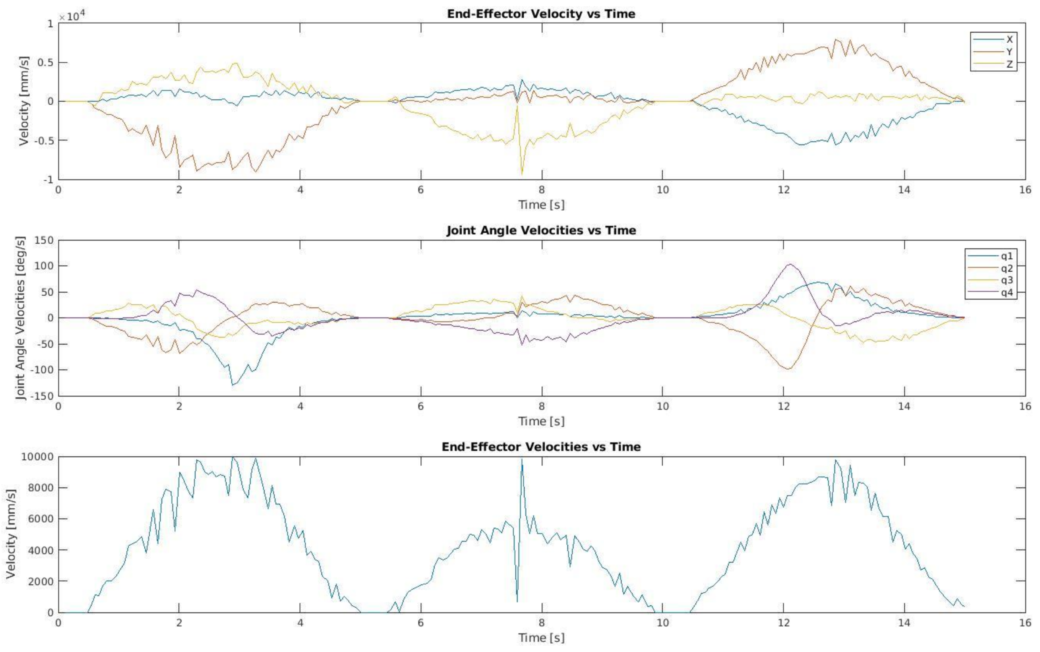


Figure 13: Three subplots of the end-effector velocity separated into X, Y, Z axis, joint angle velocities, and the magnitude

of the end-effector velocities plotted against time, over the whole movement of the robot arm through three arbitrary poses.

6) Discover and avoid singularities

The isSingularity function was developed to detect singularities in real-time as the robot arm moved through its workspace. The function monitored the determinant of the 3x3 submatrix of the Jacobian and successfully identified when the robot was approaching a singularity. Figure 14 illustrates the robot's end-effector position as it moved from the center of the workspace to a singularity position, while Figure 15 shows the corresponding plot of the Jacobian's determinant over time. As expected, the determinant approached zero as the robot neared the singularity, triggering the emergency stop function to prevent the robot from entering an unsafe configuration. This confirmed the effectiveness of the singularity detection and avoidance mechanism.

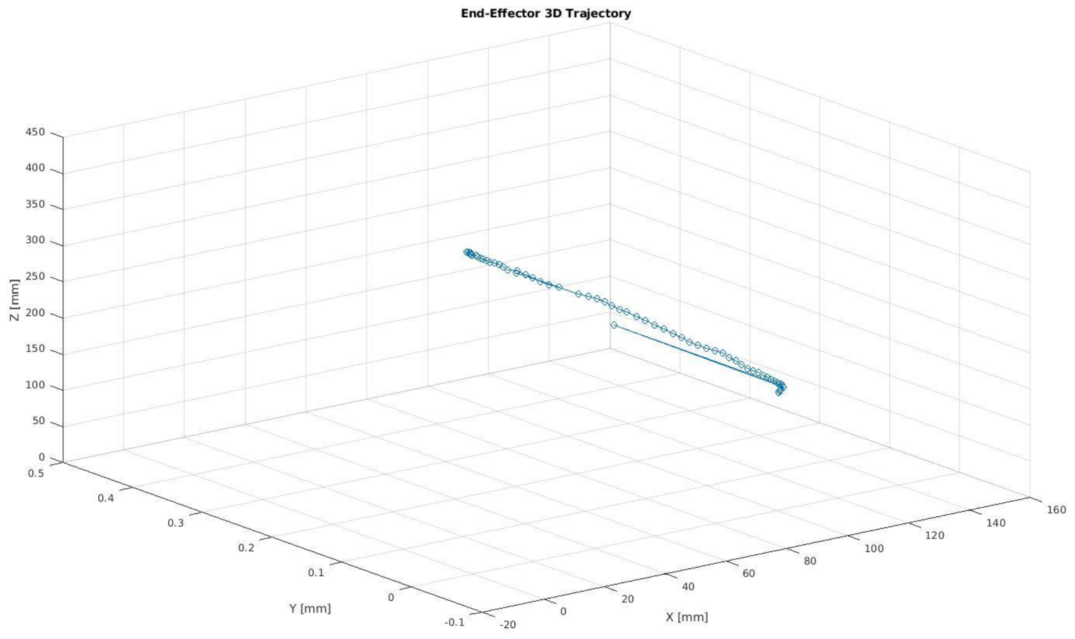


Figure 14: 3D plot of the end effector position as the robot arm was moved from the center of the workspace of [162, 0, 42, 90] to a singularity position of [0, 0, 450, -90].

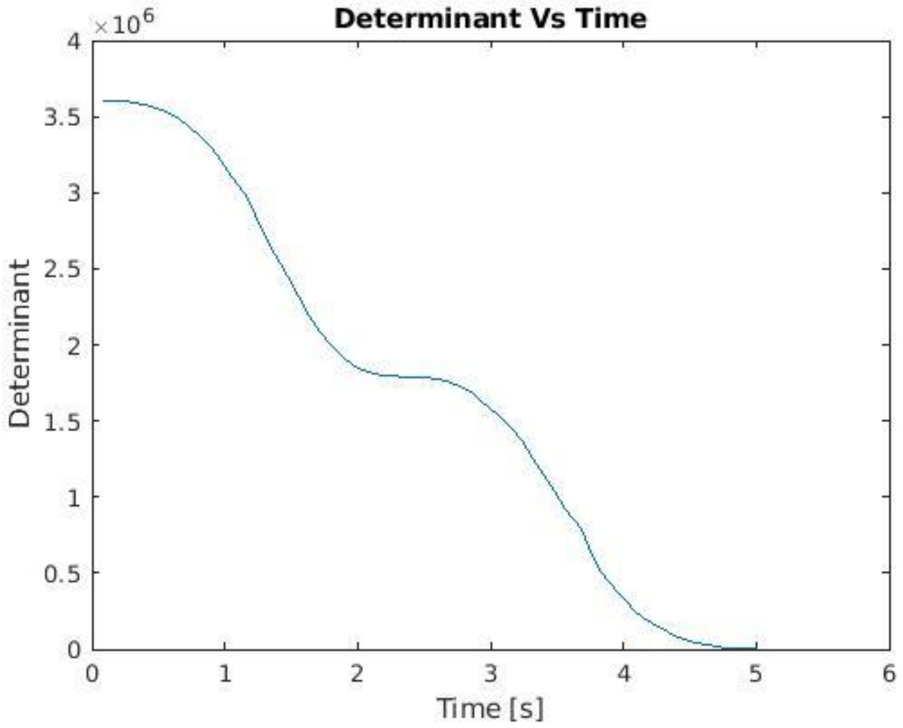


Figure 15: A plot of the determinant of the Jacobian against time as the robot arm was moved from the center of the workspace of [162, 0, 42, 90] to a singularity position of [0, 0, 450, -90].

7) Extra Credit: Velocity-based motion planning in task space

For the extra credit task, velocity-based motion planning was implemented to move the robot's end-effector at a constant speed along the triangle path defined by three vertices. The end-effector's position, velocity, and acceleration were plotted against time, as shown in Figures 16, 17, and 18. While the robot followed the planned trajectory, the motion was less smooth compared to traditional trajectory planning methods. The presence of slight jitter and errors indicated that the real-time calculations required for velocity control introduced delays, resulting in less precise movement. Despite these limitations, the implementation demonstrated the feasibility of velocity-based motion control, though it is not recommended as the primary method for tasks requiring high precision.

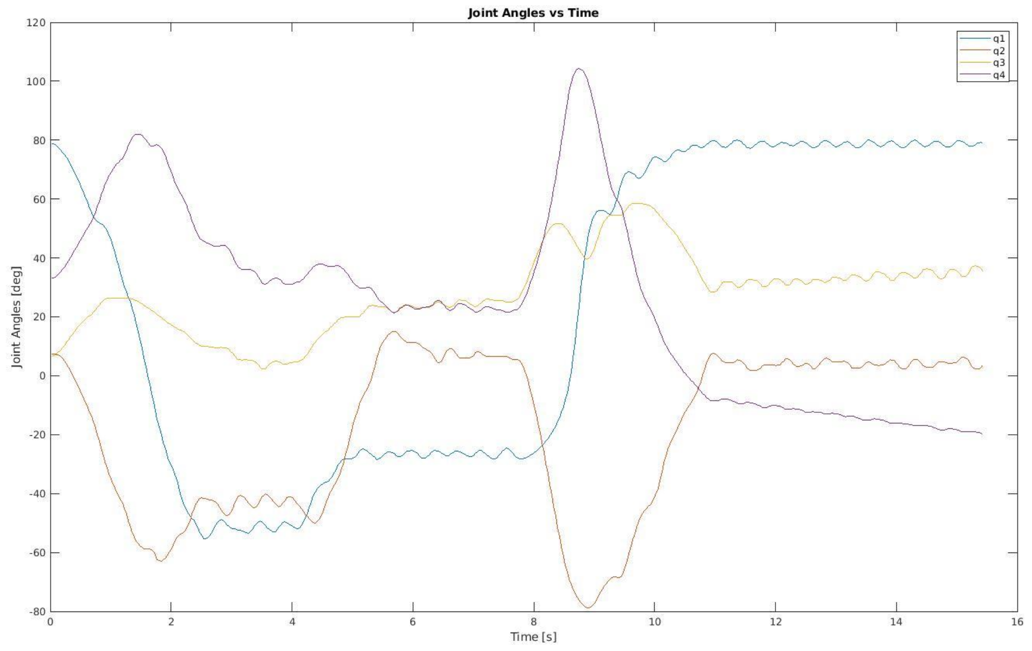


Figure 16: A plot of the joint angles of the robot arm against time over the whole movement of the robot arm through three arbitrary poses.

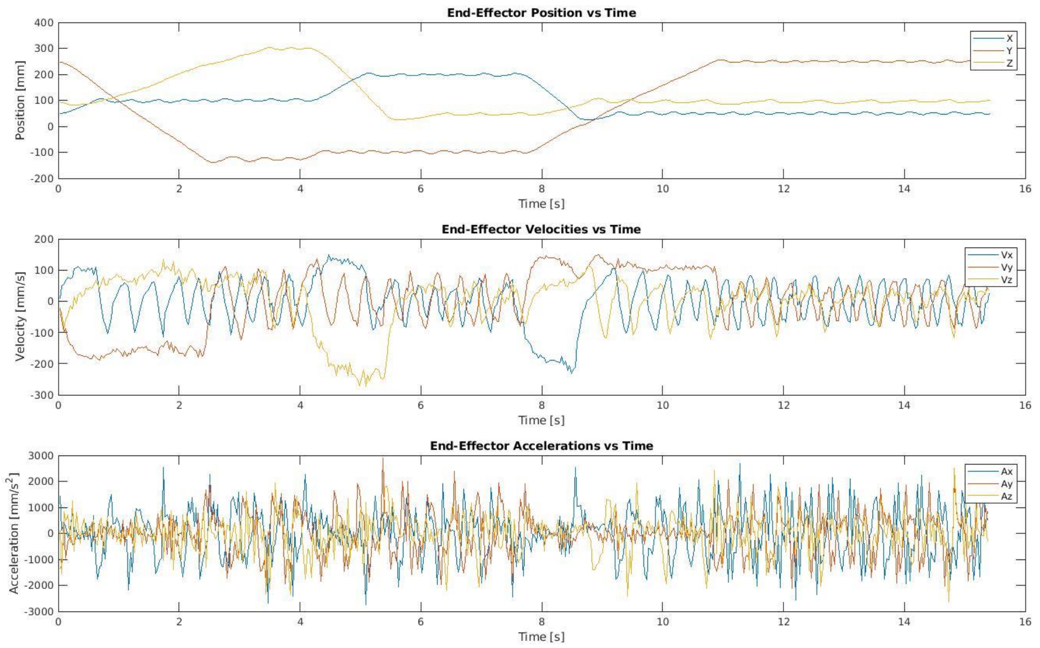


Figure 17: Three subplots of the end effector position, velocity, and acceleration separated into X, Y, and Z positions plotted against time, over the whole movement of the robot arm through three arbitrary poses.

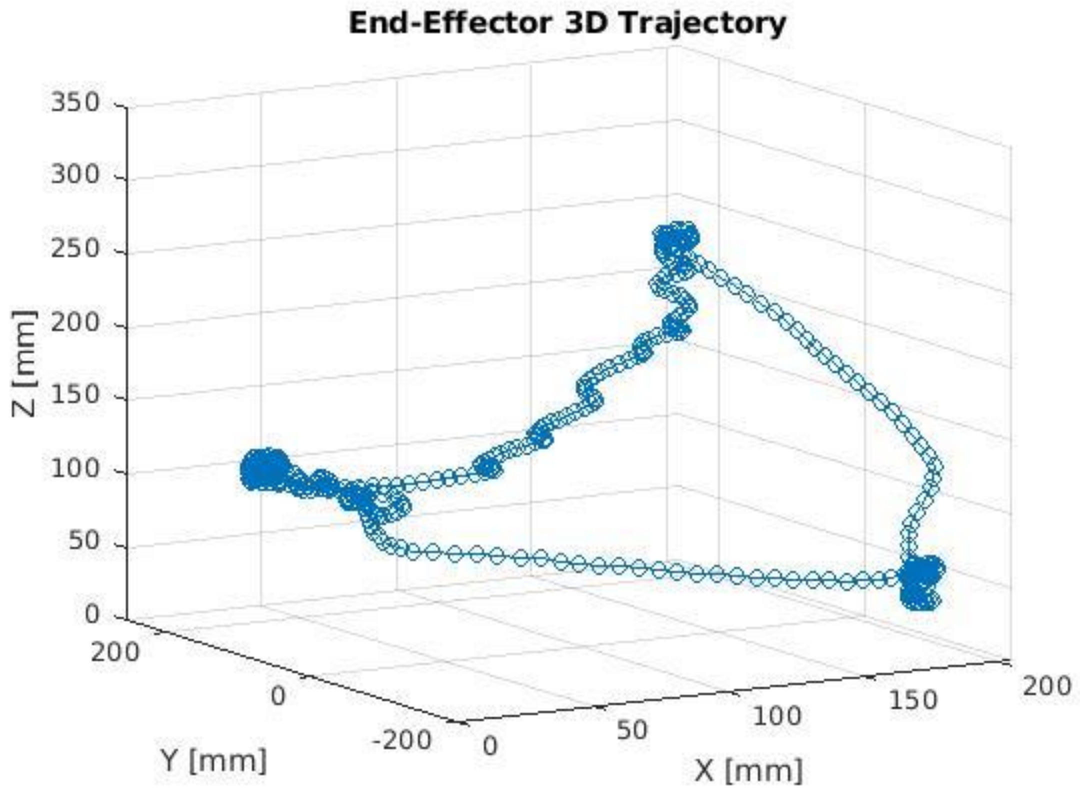


Figure 18: 3D plot of the end effector position as the robot arm was moved between the three arbitrary poses chosen to form a triangle in the robot arm's workspace using velocity control in MATLAB.

Discussion

The primary reason for this lab was to implement and validate the velocity kinematics for the robot arm using the Jacobian matrix. Throughout the lab, the task space velocities were calculated and plotted from the joint space velocities and singularities in the robot arm workspace were also explored. Based on the results above, the key findings are discussed below.

Jacobian Calculation and Validation

The Jacobian 6x4 matrix was found for the OpenManipulator X arm in two ways, using the partial derivative of the end effector positioning and by the cross-product method. The results show consistency across the two methods and confirm the accuracy of the calculations. The singularity verification was also an important piece to verifying the accuracy and effectiveness of the matrix. By finding the determinant of the matrix at different points to check for singularities, it was determined that the calculation could accurately determine what configuration was a known singularity. This means that the Jacobian is a reliable method to calculate the velocity kinematics and to check for singularities before moving the robot into an unpredictable joint configuration.

Forward Velocity Kinematics

The forward velocity kinematics were calculated and implemented with the function fdk3001. Based on the results from the live plotting of the velocity vector in the robot arm task space through the predefined trajectory, it is reasonable to conclude that the function is a reliable implementation of forward velocity kinematics. The vectors' magnitudes and directions matched those that would be present based on the predetermined trajectory when observing the end effectors behavior between the three points.

Singularity Detection and Avoidance

Another extremely important part of the lab was the calculation and detection of singularities during the trajectory of the robot arm. Since configurations of the robot arm in singularities can result in unpredictable behavior from the robot, the calculations and avoidance of singularities is an extremely valuable part of the lab to avoid reaching singularities when looking to complete other tasks with the robot arm. The Matlab function isSingularity() was written to detect singularities in real time as the robot arm is in motion, which prevents the robot from reaching positions where control could be compromised. The implementation was tested and was able to successfully avoid singularity configurations and is therefore a useful tool to aid the control of the robot arm.

Velocity Based Motion Control

For the extra credit part of the lab, velocity-based motion planning in the task space was implemented in the attempt to move the end effector of the robot at a constant speed. Since the control of the robot by using the run_trajectory function developed in the previous lab and used and modified in this lab has proven so successful, it would be hard for the new method to live up to the smooth trajectory motion from the joint space. Consistent with this, while the results showed that the robot arm did follow the planned trajectory, it did so with some jerkiness and errors not seen in the other trajectory planning functions. Since there are so many calculations involved in real time to make the task space velocity control work properly, there is bound to be some slight delays from the code that result in inconsistencies in the movement of the end effector and less than perfectly smooth motion. To reduce the jerkiness of the robot, a low-pass filter was added to make the movement as smooth as possible. As a result, although the implementation does prove to be effective, it will not be the main method of trajectory planning used to control the motion of the robot arm going forward because of the lack of smooth motion.

Overall, the lab successfully demonstrated the application of velocity kinematics and the importance of singularity detection in robotic motion. The methods developed here lay the groundwork for applying the methods written and tested in this lab and the previous lab and applying them to the task laid out in the final project.

Conclusion

This lab effectively demonstrated the application of velocity kinematics through the implementation and validation of the Jacobian matrix for the OpenManipulator-X robot arm. The Jacobian matrix was used to calculate the task space velocities from the joint space and identify singularity configurations to ensure the movement of the robot arm was consistent and controlled. The methods developed in this lab, including the calculation of the Jacobian matrix, forward velocity kinematics, and singularity detection, overall proved to be reliable and accurate.

The forward velocity kinematics, implemented through the fdk3001 function, accurately translated joint velocities into end-effector velocities, as evidenced by the live plotting results. The successful avoidance of singularities through real-time detection further highlighted the importance of these calculations in maintaining safe and effective robot arm motion. The extra credit task of implementing the velocity-based motion was further evidence of the successful calculation from the Jacobian matrix, but that the implementations vary in effectiveness as shown by the plots above.

Overall, the lab provided solid evidence of a foundational understanding and implementation of forward velocity kinematics and the calculations with the Jacobian matrix. Going forward into the final lab of the term, the methods tested and proved to be reliable and good control of the robot arm. These matlab methods will be extremely useful for completing the final task of moving the end effector to different locations to pick up colored balls and sorting them into specific positions. Being able to direct the end effector with precision over time and with calculated trajectory will be essential for completing this task and can be done using the methods described in this lab above.