MEAM520, University of Pennsylvania

LAB4

Jacobians and Velocity Kinematics

Venkata Gurrala
Sheil Sarda

November 6, 2020

Table of Contents

Introduction

The objective of this lab was to determine the forward and inverse velocity kinematic relations for the robot given certain inputs. Once these relations are determined, it was evaluated with various inputs to create desired trajectories. Practical issues such as singularities are also addressed.

Method

Determining the Jacobian

The most fundamental matrices involved in determining the velocity kinematics of the lynx robot is the Jacobian. This matrix has a size of 6 x n where n is the number of joints that the robot has. Each row represents the corresponding joint's velocity in the following order:

$$v_x, v_y, v_z, \omega_x, \omega_y, \omega_z$$

The linear velocity Jacobian and angular velocity Jacobians were determined before appending them together to form the final Jacobian. In the python code developed, a function was created to form the Jacobian and the matrix was initialized as a 6 x 6 zero matrix before filling in the elements of the Jacobian. This was done to ensure that the Jacobian stayed as a 6 x 6 matrix.

Linear velocity Jacobian

The equation represents the general form of the linear velocity Jacobian:

$$J_v(\vec{q}) = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \cdots & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \cdots & \frac{\partial y}{\partial q_n} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \cdots & \frac{\partial z}{\partial q_n} \end{bmatrix}$$

Since the Lynx robot only consists of revolute joints, the following equation can be used to find each element of the linear velocity Jacobian:

$$J_{v_i} = \hat{z}_{i-1} \times (\vec{o}_n - \vec{o}_{i-1})$$

In this equation, the $\hat{z}_{i-1}$ represents the orientation of the previous joint's z axis with respect to the world frame. This can be found in the third column of the joint's rotation matrix with respect to the 0th frame. However, this can also be simplified for each joint using geometric intuition for joints 0-3 (figure 1). For joint 0 which is the base of the robot, the z axis will never deviate from the positive z direction, so it has the following z:

$$z_0 = \begin{vmatrix} 0 \\ 0 \\ 1 \end{vmatrix}$$

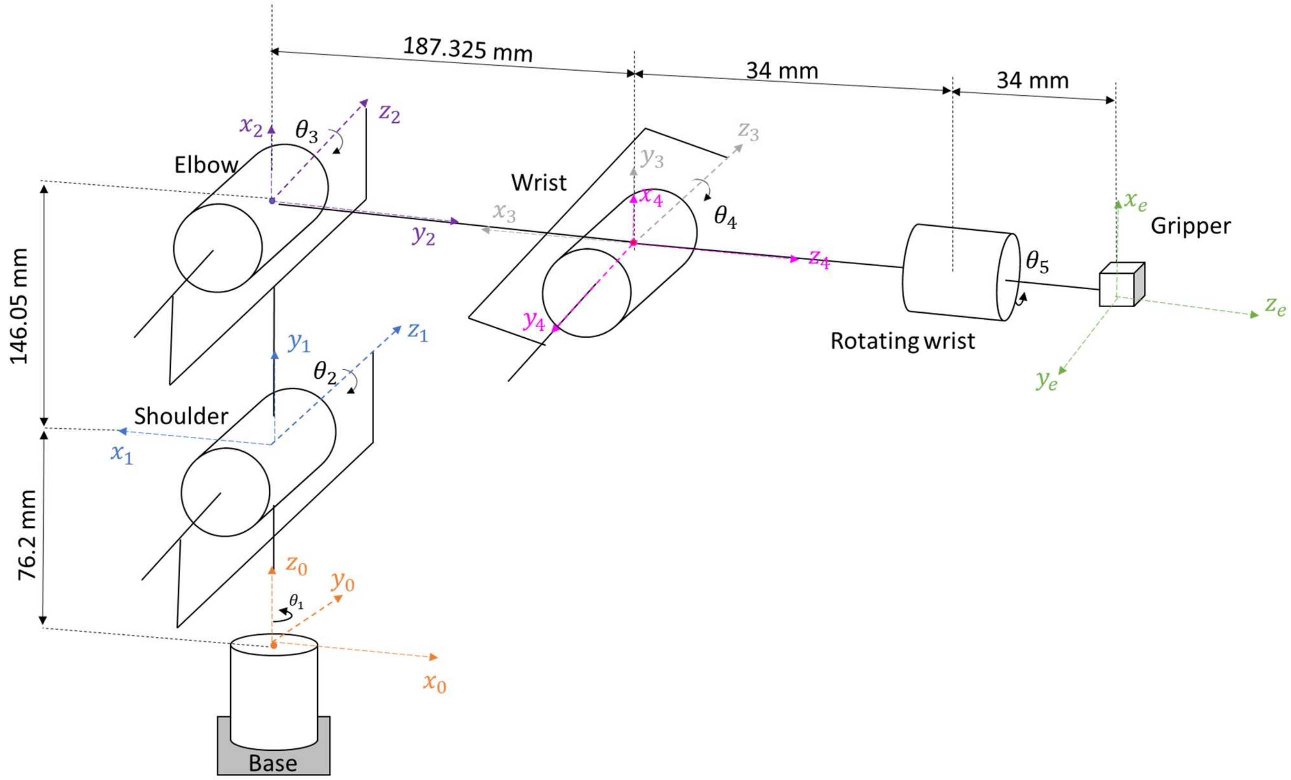

Figure 1. Understanding the geometric intuition behind choosing the z vectors

For joints 1,2 and 3, the z axes point in the same direction no matter what q input is given (figure 1). Another valid assumption is their z vector is a function of angle θ_1. Therefore, the z matrix is found to be the following:

$$z_{1,2,3} = \begin{vmatrix} -\sin(\theta_0) \\ cos(\theta_0) \\ 0 \end{vmatrix}$$

The z matrix for 4 and e are less intuitive to derive so it would be easier to obtain them the 3rd column of the transformation matrix outputted by the calculateFK function. The rotating wrist and the end effector will always have their z vector pointed in the same direction so the same z vector can be used for both joints:

$$z_{4,e} = \begin{vmatrix} z_e & * \, x_0 \\ z_e & * \, y_0 \\ z_e & * \, z_0 \end{vmatrix}$$

To determine the linear velocity Jacobian, the origin of the end effector and the origin of the previous joint is required. These inputs can be found from the Jointpositions variable outputted by the calculateFK function.

Angular velocity Jacobian

This Jacobian has the following form:

$$J_\omega(q) = [\rho_1\hat{\mathbf{z}} \quad \rho_2\mathbf{R}_1^0\hat{\mathbf{z}} \quad \rho_3\mathbf{R}_2^0\hat{\mathbf{z}} \quad \cdots \quad \rho_n\mathbf{R}_{n-1}^0\hat{\mathbf{z}}]$$

Since the Lynx robot has only rotational joints, ρ_n is 1 for all joints. While calculating the linear velocity Jacobian J_v, the z matrices were already formed with respect to the 0th frame. This makes is easy to simply substitute the values of z for the angular velocity Jacobian. The simplified angular velocity Jacobian will have the following form where n represents the joint:

$$J_\omega = \begin{vmatrix} z_{0,x} & z_{1,x} & z_{2,x} & \cdots & z_{n,x} \\ z_{0,y} & z_{1,y} & z_{2,y} & \cdots & z_{n,y} \\ z_{0,z} & z_{1,z} & z_{2,z} & \cdots & z_{n,z} \end{vmatrix}$$

Forward kinematic velocity

This function is responsible to return a linear velocity v and angular velocity ω in the x, y and z directions, given the current position q, joint velocity, $\dot{q}$ along with the specified joint. The is done by calculating the Jacobian using the Jacobian function using the current q and joint of interest. The following equation can then be used to find the linear velocity matrix where the Jacobian is a 6 x 6 matrix:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = J * \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix}$$

Inverse kinematic velocity

In determining the inverse kinematic velocity of the robot, the first step would be to determine the Jacobian using the current position q and the joint of interest. It is possible that some linear and angular velocities are not constrained and defined as NaN (Not a number). When the user specifies this in the linear velocity and angular velocity inputs, the rows of the corresponding Jacobian are removed before calculating the $\dot{q}$. The following equation can be used to determine the joint velocity $\dot{q}$:

$$\begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{bmatrix} = J^{-1} * \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Evaluation

Testing FK

The code which is responsible to output the forward kinematic velocity was tested by setting q to [$\pi/2$,0,0,0,0,0] and setting $\dot{q}$ to [2,0,0,0,0,0]. This would ensure that only joint 0 was moving and the end effector's velocity can be understood using geometric intuition. The output velocity vector was then used as an input to the IK velocity function which outputted the following $\dot{q}$: V=[2,0,0,0,0,0] which shows that both IK and FK velocity codes work as expected. In addition to the them matching, the output linear velocity was found to be [-361,361,0] which is reasonable since both x and y magnitudes are the same and the negative sign of the x and positive sign of the y direction. This is true since the q position given was (pi/4).

Singularities

A singularity implies that for a desired end-effector linear velocity, there exists no $\dot{q}$ which fulfills $v = J * \dot{q}$, i.e. instantaneous endpoint motion is impossible in one direction. A singularity is found when the Jacobian loses rank, i.e. rank(J(q)) < 5 (number of controllable end-effector DOFs).

It is also worth noting that since the Lynx is a 5DOF robot, there are always axis of rotations which are not feasible to rotate the end-effector around. However, there are also cases when joints are in singular alignment, requiring careful motion planning to avoid.

When solving for the $\dot{q}$ at or around a singularity, there can be several issues that can arise. It becomes impossible to take the inverse of the Jacobian at a singularity since the matrix is no longer full rank. Further, if this inverse is taken when around singular positions, the resulting $\dot{q}$ results in undefined behavior, yielding extremely high and unexpected values.

Thus, an alternate method should be used to determine the closest solution that satisfies the linear and angular velocity constraints. The least-squares error solution to solving for dq around a singularity can be computed using the Moore-Penrose pseudo-inverse operation (implemented in `NumPy` as `np.pinv(Jacobian)`).

An example of a singularity for the Lynx robot:

$$q = [0, 0, -\pi/2, 0, 0, 0]$$

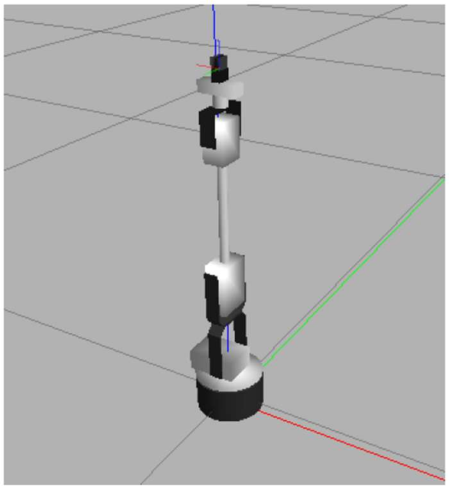

Figure 2. Arm in vertical position

Unconstrained velocities

If the user specifies that some linear or angular velocities are unconstrained (NaN), the inverse kinematic velocity function is designed to remove the corresponding rows of the Jacobian which will make it a non-square matrix. The Moore-Penrose pseudo inverse can still be used to overcome this problem when solving for the joint velocities.

End effector trajectory

The trajectory of the end effector was plotted after the $\dot{q}$ was set to a constant velocity: [0.01,0.01,0.01,0.01,0.01,0.01]. This $\dot{q}$ was added in a for loop to the zero configuration and the end effector position was tracked over 100 seconds. The following plots show the end effector's trajectory:

3D view:

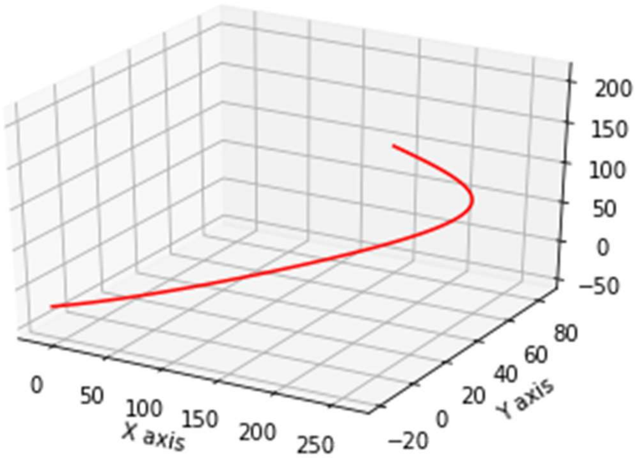

Figure 3. 3D trajectory when all joints are increasing their angle at a constant rate

Top view:

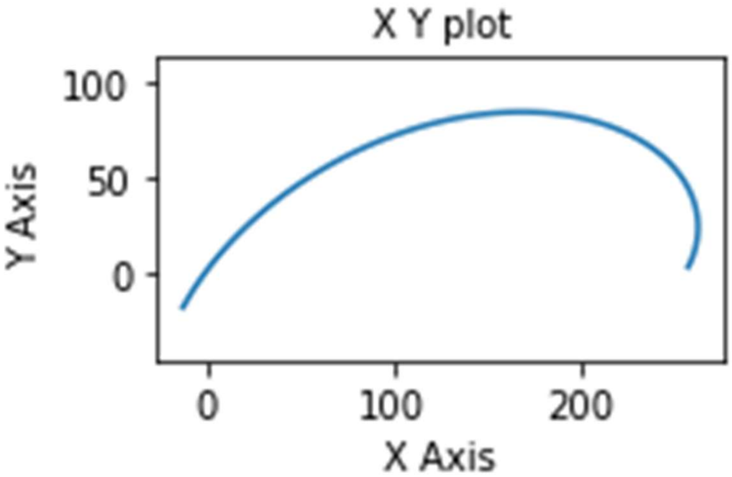

Figure 4: Top view of trajectory

Side view:

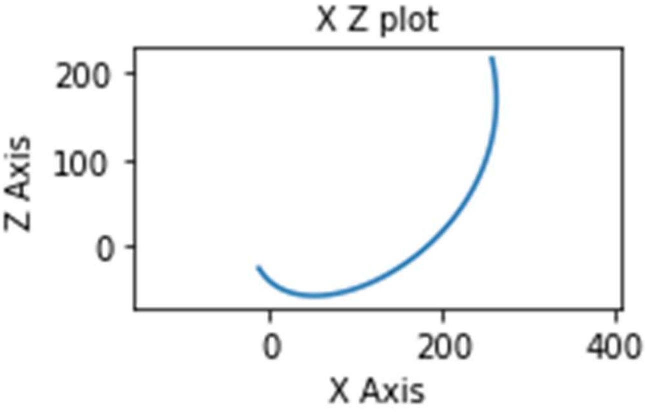

Figure 5: Side view of trajectory

Front view:

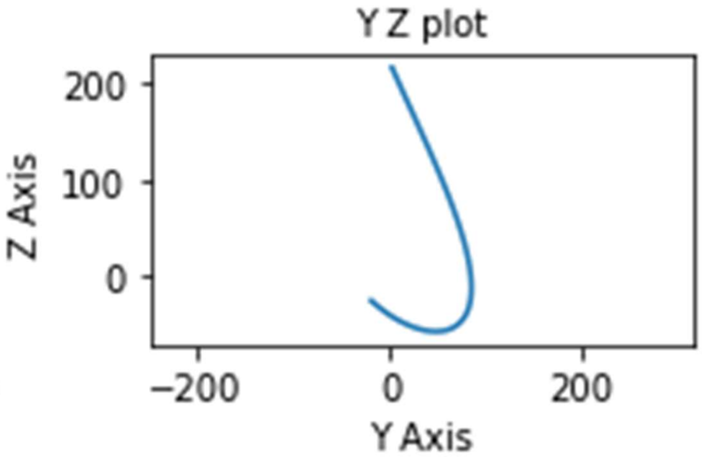

Figure 6: Front view of trajectory

The results of the trajectory make sense considering that by increasing each joints angle, the robot naturally comes closer to itself and the base of the robot while the arm moves towards the y axis. This is seen from the top view of the robot. The 3D plot shows how the height of the end effector is decreasing while spiraling towards the robot.

Linear trajectory:

A linear trajectory can be drawn with the end effector by specifying a linear velocity in one direction while keeping every other linear and angular velocity value at 0. This would ensure that the velocity of the end effector is only in one direction. The initial position was set to the zero configuration and the inverse kinematic velocity function was used to determine the initial $\dot{q}$ using the given linear and angular velocities. In the for loop, the position q is updated by adding $\dot{q}$ to the current q. Once the new position q is determined, it is used along with the v and omega values to determine a new $\dot{q}$ value. The resulting plots of the end effector are shown below:

3D plot:

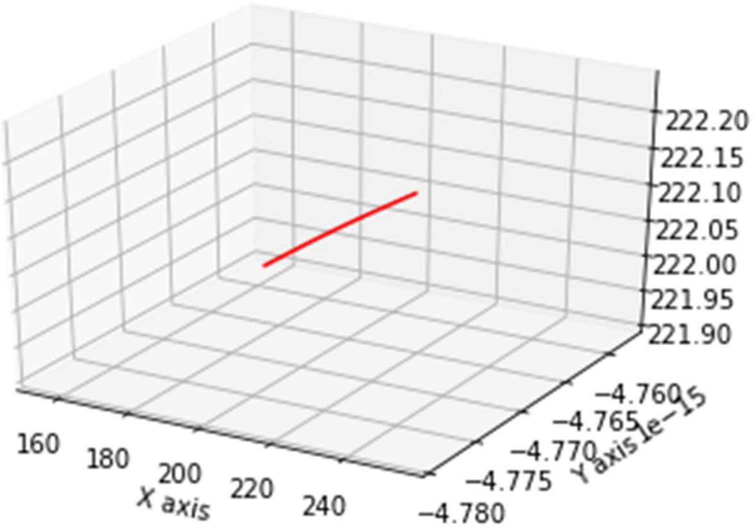

Figure 7: Linear trajectory shown with 3D plot

Top view:

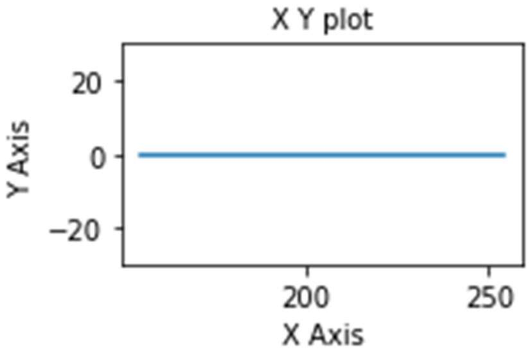

Figure 8: Top view shows linear trajectory with no error

Side view:

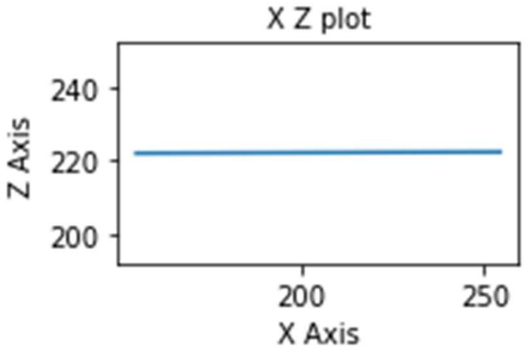

Figure 9: Side view of linear trajectory shows no error

Front view:

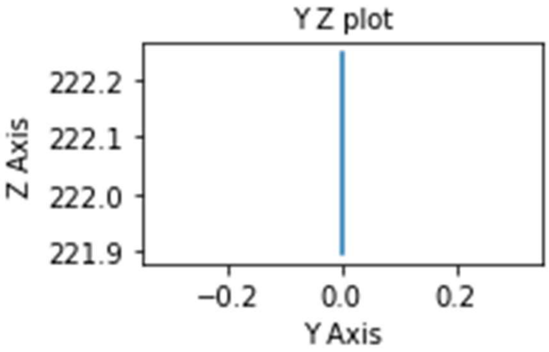

Figure 10: Front view of linear trajectory shows no error

Circular trajectory:

To have the robot follow a circular trajectory, 4 linear trajectories were combined together. The initial linear velocity was (0,1,-1). The velocity was interpolated to make it more smooth between each trajectory. The angular velocity w was left unconstrainted. The IK_Velocity function was used to determine the $\dot{q}$ given the initial q, v and ω. Once this was found, a for loop was built in which the $\dot{q}$ was added to the zero position and was recalculated with each iteration using the new position over several iterations. The new end effector position was found in each iteration and was plotted in figures 11-14. From the 3D view (figure 11), the circle is shown to not lie completely onto a plane as it has an error of about 3-4 mm. Since the circle was drawn with a bias in the negative y direction, the robot had lost some dexterity with the end effector to keep the circle on the same plane. Since the Lynx robot consists of many rotational joints, it would cause the end effector to trace out a radius when the base is moved. Since all the other joints are also revolute joints, they also trace a radius with the end effector. So, if a circle is to be drawn on the z-y plane, it becomes difficult to be able to keep in on a single plane. Each joint must be constantly adjusted to “cancel out” the circular trajectory caused by the rotational joint.

3D plot:

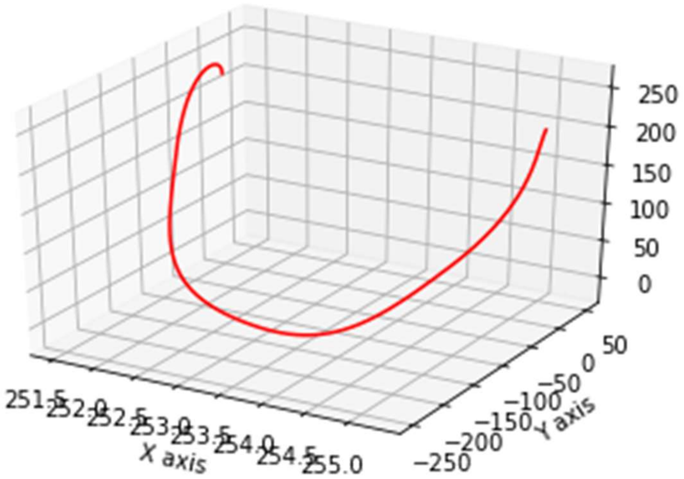

Figure 11: 3D plot of circular trajectory shows some error (3-4 mm)

Top view:

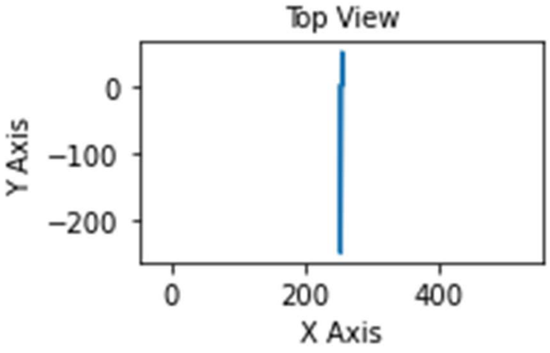

Figure 12:Top view of circular trajectory shows little error

Side view:

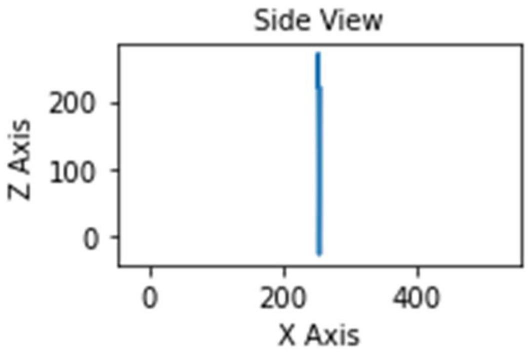

Figure 13: Side view of circular trajectory shows little error

Front view:

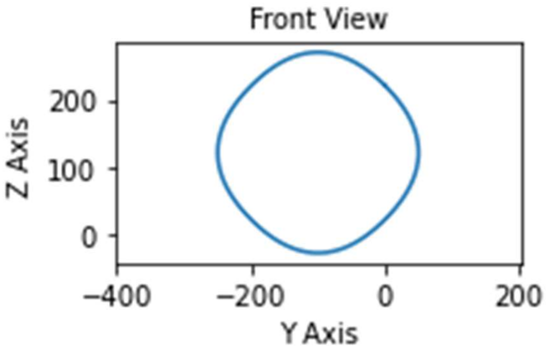

Figure 14: Front view of circle drawn

If it is desired to change the orientation of the gripper, the angular velocity ω can be manipulated. Specifically, controlling joint 3 using FK and IK velocities will help specify orientation of the gripper. The angular velocity will help the joint become more dexterous and reach more specific points in the space with a specific orientation. In the example trajectories, the angular velocity is set to 0 when drawing the line. However, when drawing the circle, the angular velocity in the x and z position was left unconstrained while setting the y angular velocity to 0. This would enable it to draw the circle on the top view plane. This proves that the end effector's orientation can be controlled by using the angular velocity constraints effectively.

Showing the effect of singularities:

Even though a least squares method was used to determine a solution for velocity with singular Jacobians, the solution chosen may not always be the one that the user is looking for. The effect of the singularity will still be easily noticed by the user. Figure 15 shows a trajectory that passes through a singularity. The instability of the position can be noticed at point (0,0,255) which is where the singularity is present.

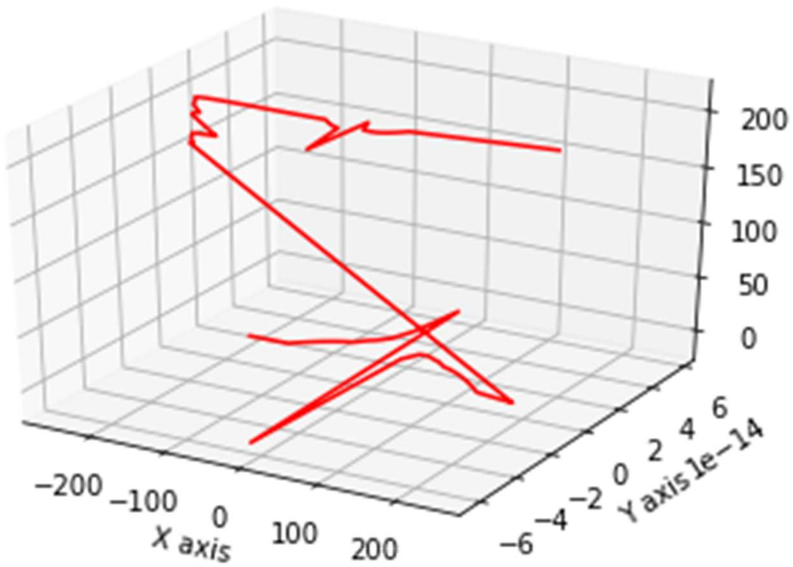

Figure 15: Trajectory of a linear velocity in the negative x direction showing where position can become unstable

Analysis

In this lab, we were able to determine a kinematic formulation of the various velocities of the Lynx robot. We also found cases where singularities are found, and we dealt with them by using a least squares solution method to find the closest possible solution. By using this solution method, we eliminated the need to evaluate the column rank of every Jacobian in each iteration which would save computational time. The Jacobians were also calculated in a method to avoid calculating rotation matrices for the intermediate joints which saved computation time. These changes help to create a more efficient calculation of the velocity trajectories.

The results obtained from the plots show that the functions responsible to calculate Jacobians, forward kinematic velocities and inverse kinematic velocities worked correctly. As mentioned previously, the robot is successful in drawing a circular trajectory, but it encounters some error as shown by the 3D plot (figure 11). The rotational joints will have to counteract each other's circular trajectories to draw a circle on a plane. Depending on the location in the workspace where this circle is drawing, the circle may not always lie on a plane since not every point is reachable. The robot also struggles to smoothly travel through singularities which cause, unstable velocities and trajectories based on figure 15.

By running experiments using python's plotting function, we found that velocity could be used to find a trajectory of the end effector, but it required more intuition to direct compared to using position control. For example, when defining a circular trajectory, it would have been simpler to define a circle using the workspace variables (x, y, z) before using inverse kinematics to plan a motion. Defining a position would give more control to the user about the end effectors location that the velocity did not. After running more experiments with the same code used to make a circle, the velocities were increased to find any key differences between position and velocity control. In general, we found that using velocity control could often overshoot the position that we were expecting. In general, using velocity to control the robot would be advantageous if it is important to the user to be able to control the velocity during a trajectory. By controlling the robot through velocities, some positional control is lost but it may not matter for some applications where there are fewer obstacles to avoid that are harder for the robot to encounter.

Generally, the Lynx robot was able to successfully follow the desired trajectories with little error. The effect of singularities is still present during certain trajectories. While this can be alleviated by choosing a more appropriate mathematical solution, the more intuitive way to avoid this problem is to use position control rather than velocities to guide it to the desired location.

Understanding the code:

CalcJacobian:

- Two variables, Jacv and Jacw are initiated in lines 2-3 which will be filled with linear velocity Jacobian and angular velocity Jacobian elements respectively
- Lines 8-11 create the z matrix which consists of all the z orientation values
- Line 13 calculates the joint position values as well as the final transformation matrix using the forward kinematics function
- The last two joints' z vectors are recorded into the z matrix in lines 14-15
- A for loop is created to calculate the linear velocity Jacobian using the Joint positions
- If the user specified a joint less than the end effector's joint, their respective columns of the z vector are kept as 0 in line 25.
- The angular velocity Jacobian is found by transposing the z matrix calculated in line 26

FK_Velocity:

- If the user specifies a joint that is out of bounds, it will return 0 vectors for v and w (line 7)
- The FK velocity is found by multiplying the Jacobian with the dq provided by the user (line 10)
- The final vector is split between linear and angular velocities (lines 11-12)

IK_Velocity:

- The velocity vector is created by combining the v and w vectors (line 7)
- If the user specified any non-constrained variables, their indexes are found (line 8)
- Line 9 deletes the rows of the Jacobian which represents the velocities that the user does not want to constrain
- Line 10 deletes the rows of the velocity vector that the user does not want to constrain
- Line 11 performs the pseudoinverse calculation of the new Jacobian matrix and multiplies it with the new velocity vector