5.1 Methods

Forward velocity kinematics:

The Jacobian matrix maps the angular velocity of the joints to end effector linear and angular velocity. Intuitively, the columns of a Jacobian matrix represent the contribution of each joint on the end effector's linear and angular velocity. Hence, the first column of the Jacobian matrix is the contribution of the first joint on the linear and angular velocity of the end effector and so on for every other joint. The calculation of the forward linear and angular Jacobian for the panda arm is as follows:

1. We calculated the linear velocity Javcobian of the Panda arm as follows,:
    a. We can calculate the linear velocity Jacobian J_v for the ith joint using the equation stated below. As per the convention of frames defined by us in Lab1, each joint rotates about the z-axis. Hence, z_{i-1} is the unit vector of the rotation matrix in the z direction for the ith joint expressed in the base coordinate frame. o_n is the position vector of the effector in the world frame and o_{i-1} is the position vector for the ith joint in the world frame.

$$J_{v_i} = z_{i-1} \times (o_n - o_{i-1})$$

    b. Since all the joints in the Panda Arm are revolute joints, the contribution of each joint to the end effectors linear velocity will be equivalent to:

$$V = \omega \times r$$

    c. To calculate the linear velocity Jacobian matrix J_v, we needed the information of each joint's z-axis unit vector from the rotation matrix, end effector's position and the position of each joint in the world frame. For this, we used our implementation of forward kinematics from Lab1 and defined a new function where we returned each joints transformation matrix in the world frame T_{i-1}^0 to get the appropriate information. Iterating over all the joints and using the equation above, we were able to calculate the linear velocity Jacobian for the Panda arm.
2. For the rotational velocity Jacobian J_ω, we used the equation below.

$$J_{\omega_i} = z_{i-1}$$

    a. Each joint's contribution to the end effector's rotational velocity would be the unit vector of the rotation matrix in the z direction for the ith joint expressed in the base coordinate frame.
    b. Since the Panda arm has all revolute joints and each joint rotates about its z-axis.

$$J_\omega = [R_1^0 Z_1 \;\; R_2^0 Z_2 \;\; \dots \;\; R_7^0 Z_7]\dot{q}$$
$$\text{where } \dot{q} = [\dot{\theta_1} \;\; \dot{\theta_2} \;\; \dots \;\; \dot{\theta_7}]$$

    c. We acquire z_{i-1} from our implementation of the forward kinematics from Lab1 in the similar fashion as explained above while calculating the linear velocity Jacobian.

Inverse velocity kinematics:

The inverse velocity kinematics solution is based on the concept of pseudo inverse for N>6 as the jacobian is a non square matrix of dimension 6*7 for panda arm as it has 7 joints

The inverse velocity kinematics solution takes the end effector linear and angular velocities as input for a particular state of the robot and gives the required joint velocities needed to achieve the end effector velocity.

The code that we implemented for solving the Inverse velocity kinematics takes into account whether a particular component of end effector velocity is constrained or not. We first stack v_in and ω_in in a column and then check which components are not Nan and store the index of those rows. Using these indexes we then take the corresponding rows from our Jacobian matrix calculated in calcJacobian function.

For example if velocity_x= NaN. Then our Jacobian would be a 5x7 matrix with the first row not considered (the row corresponding to velocity_x) and the end effector velocity vector would be 5x1.

In the scenario of solving inverse velocity kinematics for the Panda arm, there are more variables than the number of equations. This results in two cases: a case when there exists no joint velocity which will exactly produce the given target velocity and another case when if a solution exists then there would be infinitely many solutions.

- The solution to case 1 is to produce a least squares solution, as there is no exact solution.
- In case 2, the way to choose a solution from infinitely many solutions is to take one that gives a minimum L2 norm of joint velocities.

Mathematically the solution of the inverse velocity kinematics problem can be calculated as:

$$\dot{q}^{'} = J^{+}\xi$$

where J^{+} is the pseudo inverse of Jacobian and ξ is the desired end effector velocity vector

Here we calculate J^{+} , by first taking JJ^{T} which is a 6*6 square matrix, whose inverse $(JJ^{T})^{-1}$ is defined. Now we know, that $I = JJ^{T}(JJ^{T})^{-1}$. Grouping terms together ,we get $I = J[J^{T}(JJ^{T})^{-1}]$. Hence $I = JJ^{+}$ where $J^{+} = J^{T}(JJ^{T})^{-1}$.

If there is no exact solution for the set of equations, then $\dot{q}^{'}$ is a solution which gives the minimum least square error.

$$\dot{q}^{'} = \arg\min_{\dot{q}} ||\xi - J\dot{q}||_2^2$$

Also, if there are infinitely many solutions then the solution space can be expressed as:

$$\dot{q}^{'} = J^{+}\xi + (I - JJ^{+})b, b \in \mathbb{R}^{7}$$

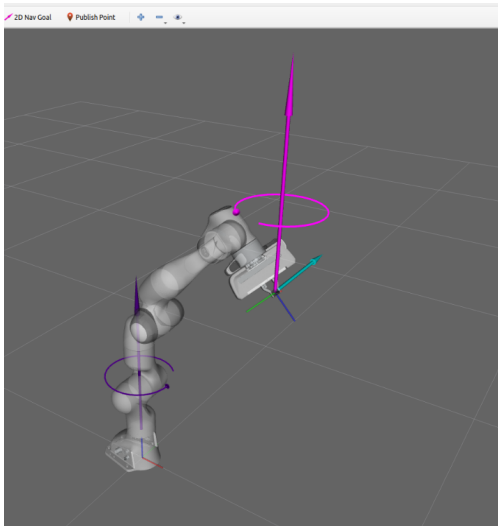
Now, $\dot{q}^{'} = J^{+}\xi$ would be the solution with the least L2 norm.

For getting the solution in our code we have used the function `numpy.linalg.lstsq` provided in numpy module and returned the joint velocity vectors.

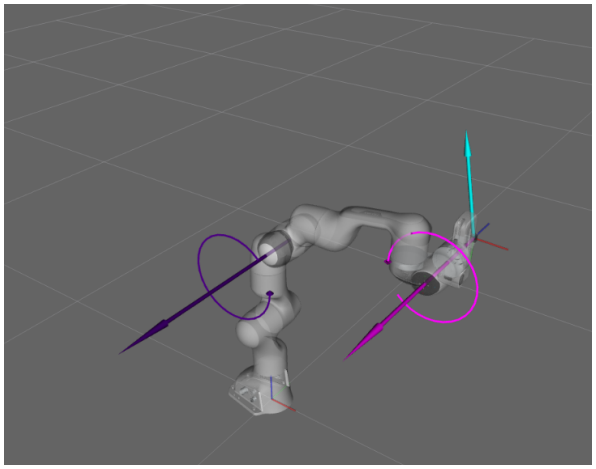
5.2 EVALUATION:

Testing process for forward kinematics:
To evaluate the correctness of our forward velocity kinematics implementation, we used the visulaize.py script to visually check the linear and angular velocity of the end effector if only one joint was rotated at a certain time. We used the right hand rule to verify the end effector linear and angular velocity in the default configurations and also new configurations added by us. Following are three new configurations each with different joint rotating and the resulting linear and angular velocity of the end effector.

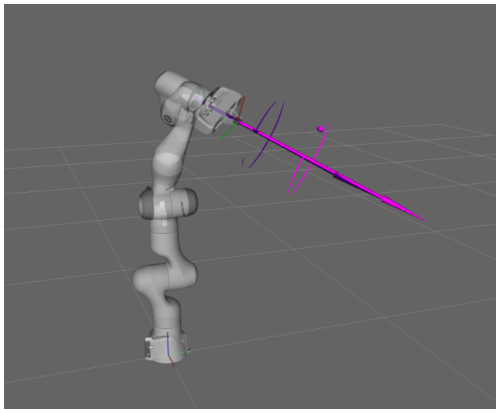
Configuration 1: [0, 0, 0, -pi/4, 0, pi/2, pi/2], Joint in consideration: 1



Configuration 2: [0, 0, 0, -pi/2, pi/2, pi/2, pi/4], Joint in consideration: 4



Configuration 3: [0, 0, 0, -pi/5, pi/4, pi/2, pi/4], Joint in consideration: 7



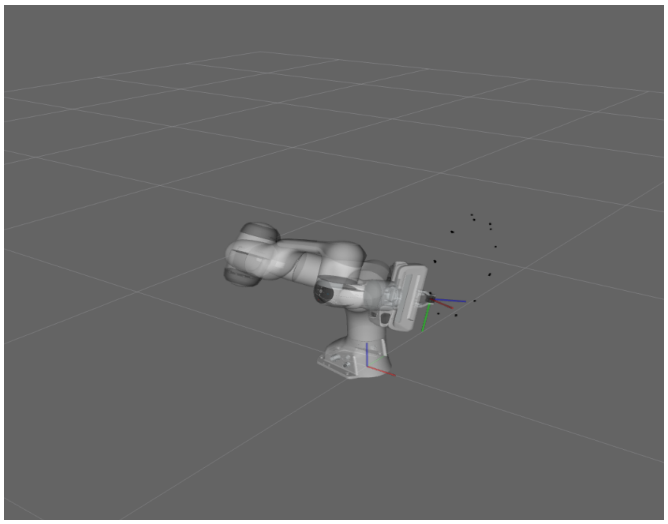
Testing process inverse kinematics:

Suppose we have an end effector velocity, if a solution exists for the joint velocities it would be infinitely many solutions. This makes the manual testing of our inverse velocity code non-trivial. Suppose, we give a certain joint velocity $\dot{\theta}$ and we use the Jacobian to calculate the end effector velocity ξ. Now, if we calculator the joint velocity using our inverse kinematics solution from the same ξ we cannot expect to get the same $\dot{\theta}$ due to infinitely many solutions of joint velocities.

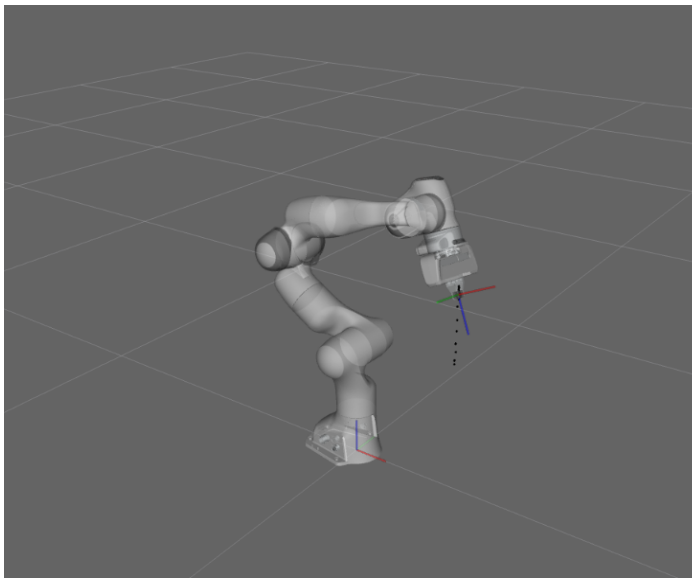
Hence we resort to a visual interpretation of our solution using follow.py. We make sure that the arm is following the trajectories (ellipse,eight and line) and if we are getting expected behaviour or not.

We code the equation of ellipse as $y = r_x cos(f_y * t), z = r_z sin(f_z * t)$ where f_y,f_z are frequency in rad/s and the desired velocity being the derivative of position.
We code the equation of line with simple harmonic motion as $z = Lsin(2 * \pi * f_z * t)$where f_z is in Hz and the velocity being the derivative of the position.

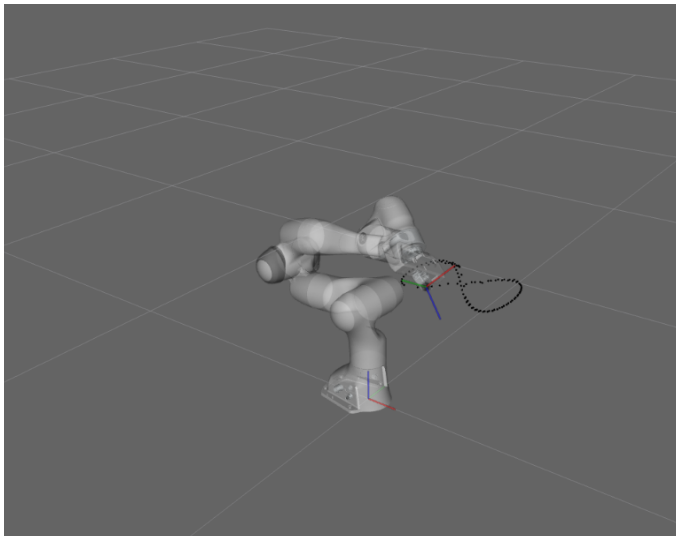
Panda Arm for different trajectories



Ellipse trajectory followed by the Panda arm



Line trajectory followed by the Panda arm



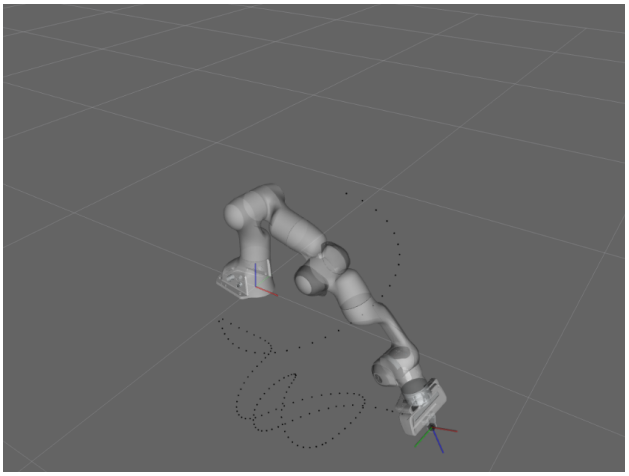
Eight trajectory followed by the Panda arm

5.3 Analysis

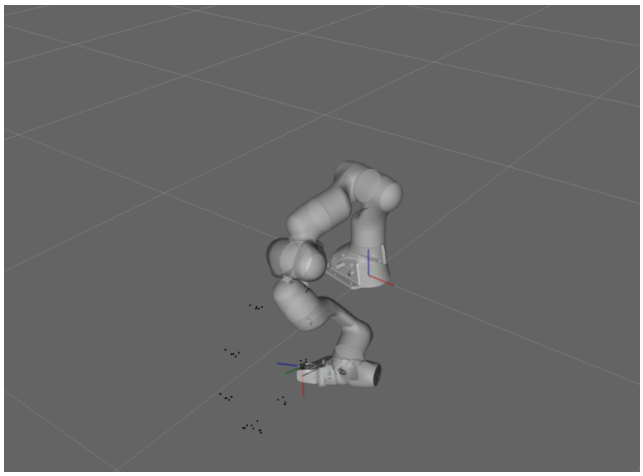
5.3.1 Trajectory tracking

When Kp =20, the arm is able to follow the given trajectory for a limited amount of times after which the robot joint limits get exceeded and as there are no constraints in the inverse velocity solver.

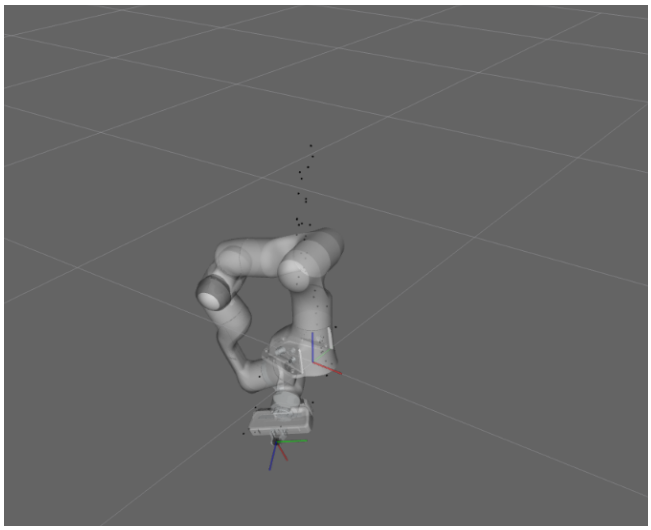
The proportional term in the controller acts like an error correction term. While following a certain trajectory, if the end effector has not reached the desired position, Kp will add a term proportional to the difference between current state and desired state with the velocity we wanted at that position to make the robot reach the position we want. Hence, Kp acts as a feedback to form a closed loop controller. When we do Kp=0, we are removing this feedback and the control law becomes an open loop controller. This is very sensitive to noise and errors. As the controller loses its sense of correctness in position, the error in position of the desired trajectory accumulates overtime and results in erroneous trajectory following as depicted in the pictures below for the eight, ellipse and the line trajectory respectively..



Eight trajectory followed by the Panda arm when Kp=0



Ellipse trajectory followed by the Panda arm when Kp=0



Line trajectory followed by the Panda arm when Kp=0

5.3.2 Joint trajectories

No, the robot does not follow the same path in configuration space everytime the end effector completes a cycle along the tracked trajectory. This is due to the simplicity of the controller which accounts for error solely on the grounds of the difference between desired and current position of the robot. One more plausible reason for this is that the solution of the inverse velocity kinematics does not take into consideration the joint constraints of the robot. This leads to the robot not being able to reach the desired position which in turn increases the error accumulated by the controller and leads to a jerky motion of the robot arm.

If implemented on a real arm, the motion of the arm would not be predictable and would have less repeatability. Moreover, we would not be able to program the robot for repeated motions as the arm's configuration space would change every time the end effector completes a cycle along the tracked trajectory. Also, due to the jerky motion of the robot arm, the actuator might get damaged.

5.3.3 Target End effector pose

When we control the world frame linear velocity and angular of the end effector, we can design a planning and control pipeline to reach the desired position and orientation of the end effector smoothly. For this, given a target position and orientation, we first use a planning algorithm to output a trajectory and velocities at each point taking in consideration the obstacles and the robot constraints. Using this trajectory and velocities, we can calculate the joint velocities using our inverse velocity implementation and use a suitable controller to track them with appropriate tolerances making sure that our end effector position and orientation is reached properly.

5.3.4 IK velocity issues

The panda arm would have issues following the given velocities when it would be in the neighbourhood of a singularity position. We referred to this link for gaining a mathematical intuition regarding issues with the naive least squares solution used in our inverse velocity solver. When the robot is in the neighbourhood of a singularity position, the output joint velocities change by a very high amount. This can probably be solved by thresholding the joint velocities given by our inverse velocity solver to a certain amount or we can change the way we frame the optimization objective by using a damped least squares method.