Lab 2: Jacobian and Inverse Velocity Kinematics
Lab Report

5.1 Method

For the inverse velocity kinematics define the tasks your solutions perform. Describe the approach you followed to solve both the velocity IK problem and to calculate the Jacobian.

In the inverse velocity kinematics section, our task is aiming at exploring the potential solutions for $\dot{q}$, which is the difference between different configurations of each joint with respect of time. We first analyze the matrix function which contains $\dot{q}$, and then use specific methods to solve the equation.

To solve the IK velocity problem, since the full manipulator Jacobian matrix is a 6x7 matrix, which is not a square matrix, we should first construct the pseudoinverse matrix, J^+, such that $\dot{q} = J^+\xi$, where $J^+ = J^T(JJ^T)^{-1}$. It could then transform into the equation: $J\dot{q} = \xi$. Then our goal becomes to analyze the solution cases (0/1/infinite), and to solve for $\dot{q}$. First, we have to get rid of the nan input, which is when the target velocities are unconstrained. Then we could determine the solution conditions of the equation by comparing the rank of these two matrices because inn the format of Ax=B, if Rank(A) = Rank(B), there must be only one solution; if Rank(A) < Rank(B), there will be infinite solutions. Lastly, if we only have one solution to solve, we would use linalg.solve function to solve for that particular answer, however, if there is no solution, or there are multiple solutions, we need to utilize the inalg.lstsq to compute the least square solutions that $\dot{q} = J^+\xi + (\mathrm{I} - J^+J)b$.

To calculate the full manipulator Jacobian matrix, we need both linear and angular velocity Jacobian matrices. We first used the equation $v = \omega \times r$, where $\omega = \dot{\theta}_i\widehat{Z_{i-1}}$, $r = O_n - O_{i-1}$, so that the linear velocity equation can be written as $J_{vi} = \widehat{Z_{i-1}} \times (O_n - O_{i-1})$ when it is a revolute joint, and $J_{vi} = \widehat{Z_{i-1}}$ when it is a prismatic joint. Since all the joints on the Panda Arm are revolute joint, we would apply the first equation to solve for the linear velocities. Here we used the elements in the transformation matrix to get the position of each joint from Lab1. For angular velocities of each joint, it is in the form of $\rho R_{n-1}^0\hat{Z}$, where R is the rotational matrix from each frame to the base frame and $\hat{Z}$ is basically the z axis of the rotating joints. After we got both the linear and angular velocities of the joints, the full manipulator matrix could be expressed as:

$J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix}$.

5.2 Evaluation

You should extensively test your solutions to ensure their correctness. In this section, document your testing process:

how can you be confident that your solutions are free of bugs? Additionally, you should include the screenshots of the robot tracking the line, circle, and eight trajectories using the follow.py script.

Before implementing our code into the simulation, we did some self-testing of the J matrix outcome to have a general idea if our solution makes sense in a numerical base. We could easily set input qs to some specific configurations and check if it returns with the correct Jacobian matrix. After conducting several rounds of such tests, we used the visualized tool provided in the lab instruction. With the visualize.py script, we could easily tell if the rotational motions of the end effector correspond correctly to the joint which is rotating, as well as the linear motion's direction, whether or not it follows the tangent line of the line between the rotating joints and the end effector.

We could go through several tests of various configurations and check the visualized linear velocity and angular velocity of the end effector. If our code passed all the visualized test and the results match what we expected, we could finally run the code on Gradescope for a final test to see how it works or if there is something else we need to fix.
In the IK velocity section, the process of testing our code is similar to what we did for FK Jacobian. This time we test out our result by running the follow.py script. We should observe the robot successfully tracking different trajectories that we pre-designed in the script if our code runs without bug. We could play with multiple designs and comment on how the robot did on following different trajectories. We could also use Gradescope to do the final check of our code to see if it could really pass those tests.

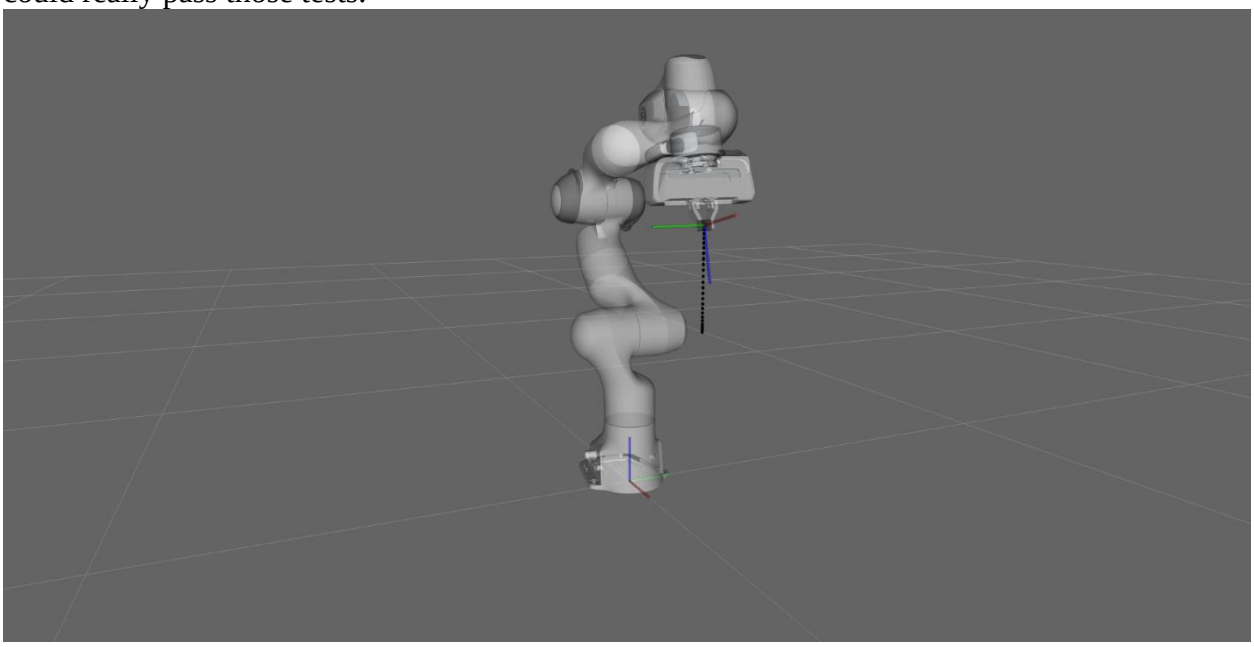

Figure 5.2.1 'Line' tracking trajectory

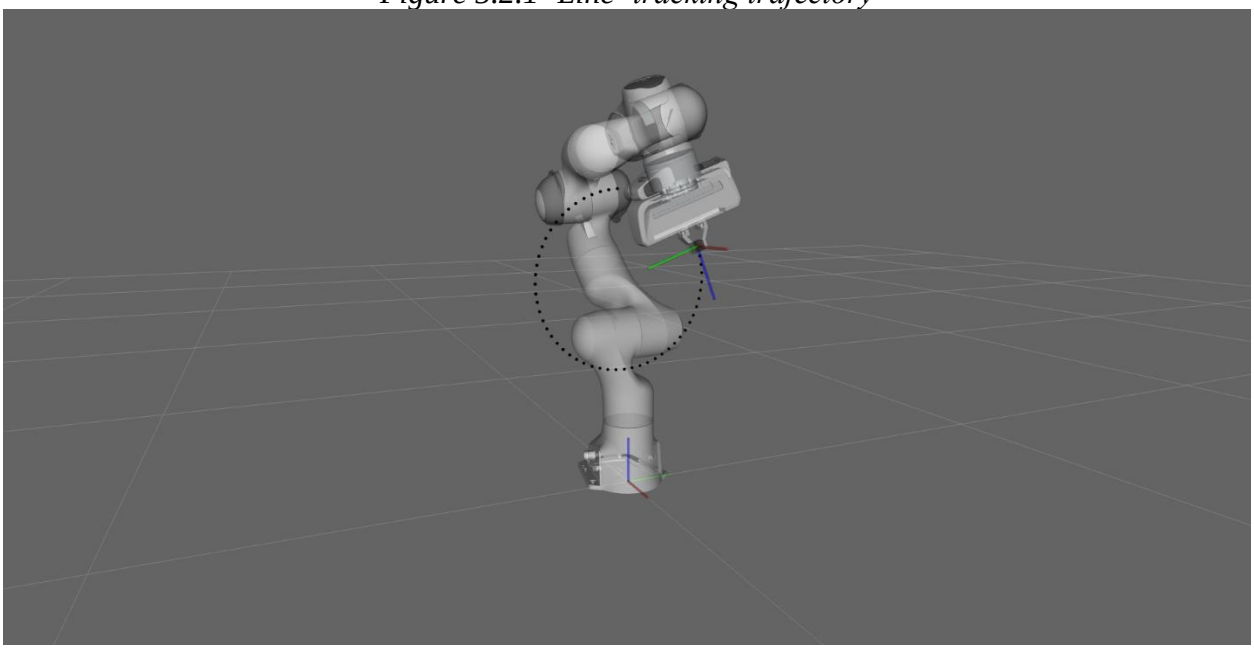

Figure 5.2.2 'Ellipse' tracking trajectory

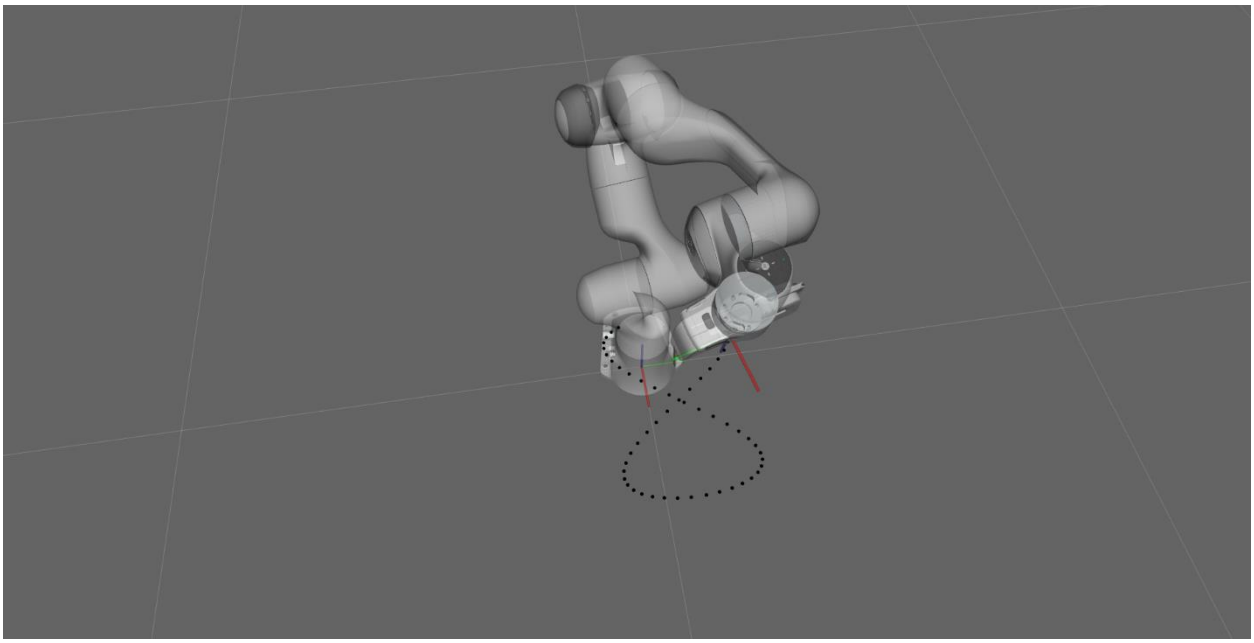

Figure 5.2.3 'Eight' tracking trajectory

5.3 Analysis

5.3.1 Trajectory Tracking

Does the arm follow the expected trajectories? What happens if you set kp to 0 in the trajectory tracking code? Give a justification explaining the behavior you see.

The end effector of the arm follows the expected trajectories that we designed, but the whole configurations of the arm did have some changes with tracking process proceeds. If we put a 0 to kp, the end effector no longer follows the designated trajectory strictly, for instance, when we applied the 'ellipse' tracking trajectory, the shape of the end effector's motion becomes skewed. What we also noticed was that setting kp to 0 would also affect the center position of the trajectory that the end effector followed. Still take 'ellipse' trajectory as an example, the center of the shape moves up a lot and kept changing when tracking went on.

One reasonable explanation we could come up with was that, though the callback function works at 1kHz to collect data about the configurations of the arm, we still cannot take it as the real-time configuration of the arm. Also, before sending out the command signal, it must do calculations of the desired position and velocity, desired end effector velocity to track the trajectory, and uses IK velocity to calculate joint velocities. During this time the robot would follow the last command motion and keep that angular and linear velocity, which is not quite right at that time spot before the new signal could be sent out. Thus, the proportional control parameter would help fix such dispensary between the 'desired motion' and the 'real-time motion' to keep the arm in track. By setting kp to 0, we basically get rid of the proportional control, so that the signal before modification resulted in the unexpected motion deviation of the end effector when tracking the trajectories.

5.3.2 Joint Trajectories

Let the robot track each trajectory in the follow.py script for about ten periods of the trajectory. Does the robot follow the same path in configuration space (joint angles) every time the end effector completes a cycle along the tracked trajectory? If not, what issues might this cause in a practical scenario?

The robot does not follow the same path in configuration space (joint angles) every time the end effector completes a cycle along the tracked trajectory, shown in Figure 5.3.2. The reason for this situation could be the nan input that we would like to get rid of, leaves the linear or angular velocities of joints not fully constrained at some configurations, or to say that the arm has different paths to choose in order to keep the end effector in track. In real scenarios, we would always want the arm to follow the same set of configurations of motions when completing different cycles so that we could monitor the motions and configurations of each joint better. It also might be an issue that if the arm could not follow the same configuration changes in each period, the working space of such arm could not be fixed and usually it takes up much more working spaces.

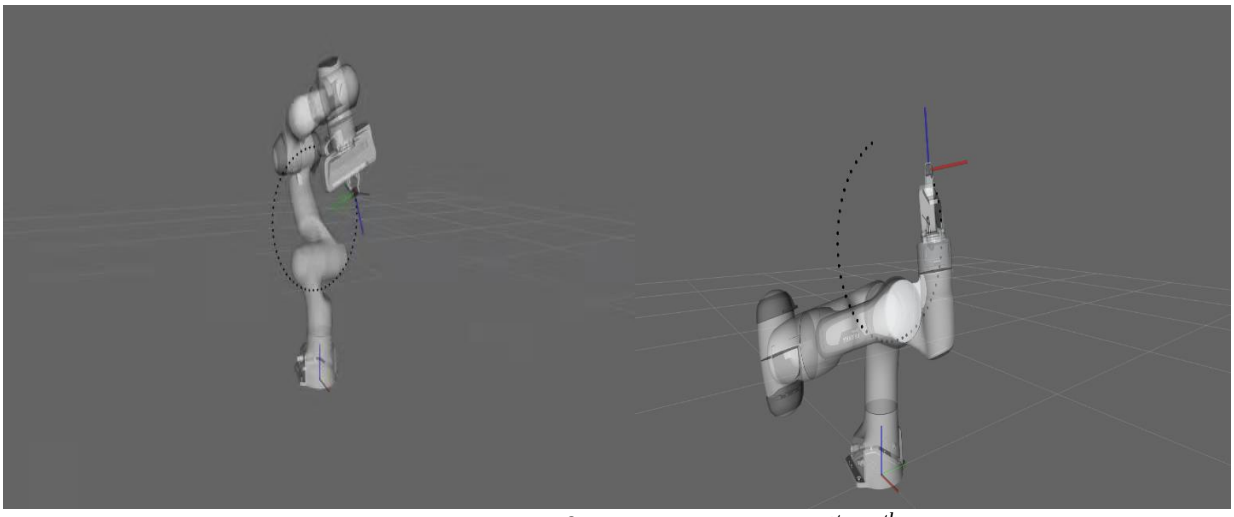

Figure 5.3.2 Robot configuration space in 1st/10th period

5.3.3 Target End Effector Pose

You can now control the world frame velocity of the end effector. Could you use this ability to move the end effector to a given target position in the world frame? Could you do the same for target orientation? Briefly describe what your approach might look like.

By knowing the position and world frame velocity of the end effector, we could solve for potential solutions of the linear and angular velocities of each joint using the IK velocity method. Once we get the Jacobian matrix of each single joint at different configurations, we could apply the FK velocity again to integrate each element in the Jacobian matrix with respect of the configuration parameter, q. Since the linear velocity is defined as the derivative of the position of each joint with respect of q, and multiply with the revolute speed, $\dot{q}$, we could easily do it with an opposite way to integrate the linear velocity with respect of the q between different poses to get to the targeted position of the end effector in the real-world frame. We can do the same for the target orientation since the angular velocity of the end effector could be controlled, we just need to set two different frames with different positions of the end effector and work out the rotation matrix between these two frames. With the rotation matrix as well as the Jacobian matrix of the end effector, we should be able to design the target orientation.

5.3.4 IK Velocity Issues

When might the Panda robot have trouble actually following the target joint velocities generated from your IK velocity solver? Describe how you might be able to modify the IK velocity solver to mitigate the impact of these issues.

Specifically, when there are too many nan inputs, and the coordinate of the end effector velocity was left unconstrained. In this case, the IK velocity solver we designed could have some trouble generating the target joint velocities for the arm to follow since it is hard to find the least squares solutions. One potential solution to mitigate such impact was that we could first get rid of those nan inputs before determining the solution situation of the equation, which we already implemented in this lab. Another mitigation could be using linear solver as much as possible. Since this issue was primarily caused by unconstrained joint paths, by getting the constrained values as much as we could, this problem could be mitigated.