MEAM 5200 Lab 2

Kevin Paulose (63031616) and Tejendra Patel (35371691)

October 26, 2023

1 Method

Before computing the velocity kinematics, we need to compute the forward kinematics for the 7-DOF Panda robot arm (or any robotic arm), which involves determining the end-effector's pose (position and orientation) in the robot's coordinate system given the joint angles. We have done it as follows-

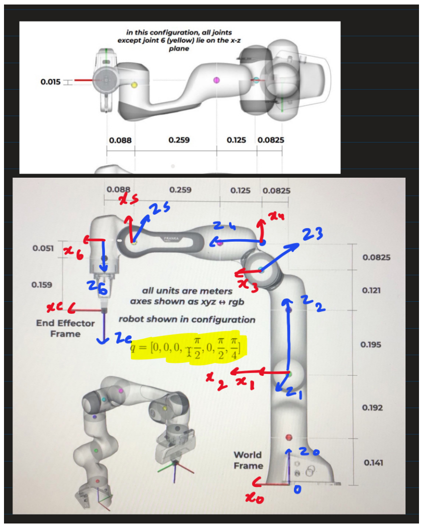


Figure 1: 7-DOF Franka Panda Arm joint orientations (a, α, d, θ) as per DH convention

Link	a_i	α_i	d_i	θ_i
1	0	$-\pi/2$	0.33	q_1
2	0	$\pi/2$	0	q_2
3	0.0825	$\pi/2$	0.316	q_3
4	0.0825	$\pi/2$	0	$q_4+\pi/2$
5	0	$-\pi/2$	0.384	q_5
6	0.088	$\pi/2$	0	$q_6-\pi/2$
7	0	0	0.21	q_7

Velocity Forward Kinematics-

$$\begin{pmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{pmatrix} = \begin{pmatrix} J_v \\ Jw \end{pmatrix} (\dot{q}) \tag{1}$$

For evaluating J_v-

$$Prismatic : J_v = \hat{Z}^0_{i-1} \tag{2}$$

$$Revolute : J_v = \hat{Z}_{i-1} \times (o^0_n - o^0_{i-1}) \tag{3}$$

$$T_e^0 = \begin{pmatrix} R & \mid & t \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & o_n^0 \\ \cdot & \cdot & \cdot & 1 \end{pmatrix} \tag{4}$$

$$T_{i-1}^0 = \begin{pmatrix} R & \mid & t \end{pmatrix} = \begin{pmatrix} R_{i-1}^0 & o_{i-1}^0 \\ 0 & 1 \end{pmatrix} \tag{5}$$

(3) is written considering all offsets as that calculated from Lab 1.

$$\hat{Z}_{i-1} = R_{i-1}^0 \hat{Z} \qquad where\ \hat{Z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{6}$$

$$J_v = (R_{i-1}^0 \cdot \hat{z}) \times (\vec{o}_n^0 - \vec{o}_{i-1}^0) \tag{7}$$

For evaluating J_w-

$$Prismatic : J_w = 0 \tag{8}$$

$$Revolute : J_w = \hat{Z}_{i-1} = \hat{R}_{i-1}^0 \hat{z} \tag{9}$$

Velocity Forward kinematics equation-

$$v = J(q) \cdot dq \tag{10}$$

Velocity Inverse Kinematics equation-

$$dq = J(q)^{-1} \cdot v \tag{11}$$

where $J(q)^{-1}$ is the pseudoinverse

1.1 Code Summary and Explanation

We used Tejendra Patel code to implement the Ellipse, the Line and Eight trajectory on hardware. Below is the code summary.

Using the CalculateFK.py file from Lab1 we added a function that generated a z_cap which is the Z axis orientation for all the joints which will be used in Calculating Jacobian.

```python
def get_axis_of_rotation(self, q):
    global z_cap                        # Making global to be used in other code
    z_cap=np.zeros((7,3))
    zth_axis= np.array([0,0,1])
    z_cap[0]=[0,0,1]
    for i in range(1,7):
        z_cap[i]= np.dot(R[i-1],zth_axis)          # Taking 3rd column of every rotation matrix
    return z_cap
```

For this, from the Rotation matrix for all the joints in forward kinematic of position was saved in a matrix and then doing $R_{i-1}^0 \cdot [0,0,1]^T$ we get z_cap which is the matrix of all the Z axis orientation of joints in ground frame.

1.1.1 Calculating Jacobian

For calculating jacobian using Eq(5) and Eq(6) for revolute joints we can get $\hat{Z}_{i-1}$ from above code and we already had Joint Orientation and End effector position from Transformation matrix in CalculateFK.py

So combining Linear Velocity Jacobian Jv and Angular Velocity Jacobian $J\omega$ to find Final Jacobian Matrix of 6x7 shape for our 7 DOF robot.

```python
import numpy as np
from lib.calculateFK import FK

def calcJacobian(q_in):
    fk = FK()
    joint_positions, T0e = fk.forward(q_in)
    z_cap = fk.get_axis_of_rotation(q_in)

    J = np.zeros((6, 7))
    J_v= np.zeros((7,3))
    J_omega= np.zeros((7,3))

    O_end_0=joint_positions[7]     # Taking end-effector position

    for i in range(0,7):
        J_v[i]=np.cross(z_cap[i],(O_end_0-joint_positions[i]))       # calculatin
    Jacobian for all revolute joints

    J_omega=z_cap

    J_=np.hstack((J_v,J_omega))

    J=np.transpose(J_)
    print(z_cap)
    return J
```

1.1.2 Velocity Forward Kinematics

Using Jacobian Matrix from above and dot product with vector of each joint velocity gives us the end effector Velcity in terms of linear velocity and angular velocity.

```python

import numpy as np
from lib.calcJacobian import calcJacobian

def FK_velocity(q_in, dq):
    J=np.round(calcJacobian(q_in),3)
    velocity = np.zeros((6, 1))
    velocity= np.dot(J,dq)

    return velocity
```

1.1.3 Velocity Inverse Kinematics

Using Jacobian Matrix from above we have 6x7 Jacobian which was not possible to inverse as seen in Eq(11). so we take pseudoinverse of J using least square method. Also in input there are cases of nan velocity of end effector for which we have to make it unconstrained so we removed that row from jacobian and velocity matrix.

```python
import numpy as np
from lib.calcJacobian import calcJacobian

def IK_velocity(q_in, v_in, omega_in):

    dq = np.zeros((1, 7))
    J=calcJacobian(q_in)
    v_in = v_in.reshape((3,1))
    omega_in = omega_in.reshape((3,1))

    velocity=np.vstack((v_in,omega_in))

    nan_indices = np.isnan(velocity)                        # Finding Nan indexes

    indices = np.where(nan_indices)[0]
    velocity=np.delete(velocity,indices,axis=0)
    J=np.delete(J,indices,axis=0)

    dq, _, _, _ = np.linalg.lstsq(J, velocity, rcond=None)          # Least square
    method for IK.
    return dq
```

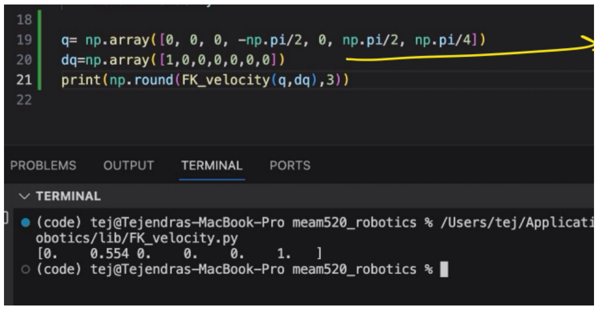
So now we have a code structure which can provide each joint input velocity for any end-effector velocities given which will be used below to give different trajectories.

2 Evaluation

2.1 Start in the zero configuration, assume only one joint moves. What do you expect the corresponding velocity of the end effector to be? Does your FK reflect this velocity?

We start by testing the code's correctness in the zero configuration, for our case the zero configuration is one where we calculated DH parameters which is: $q = [0,0,0,-\pi/2,0,\pi/2,\pi/4]$

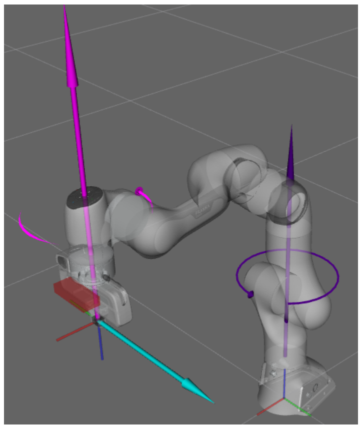
And we give input to first joint $dq = [1,0,0,0,0,0,0]$ and we check the expected output and compare with visualize.py file in gazebo and we got the correct result.



```
q= np.array([0, 0, 0, -np.pi/2, 0, np.pi/2, np.pi/4])
dq=np.array([1,0,0,0,0,0,0])
print(np.round(FK_velocity(q,dq),3))
```

```
(code) tej@Tejendras-MacBook-Pro meam520_robotics % /Users/tej/Applicati
obotics/lib/FK_velocity.py
[0.    0.554 0.    0.    0.    1.   ]
(code) tej@Tejendras-MacBook-Pro meam520_robotics %
```

(a) FK output



(b) Simulation output when moving joint 1

Figure 2: Comparing FK code and simulation output

So we can see that FK output and the simulation output is the same. Linear Velocity is along the y axis of ground frame and angular velocity along z axis.

2.2 Using geometric intuition, are there singularities and, if so, where are they? Are these configurations reflected in your mathematical expressions?

We expect geometric singularity in a 6-DOF system when atleast two joints' Z-axes are collinear. So, for the Franka Emika Panda Arm, since it is a 7-DOF there needs to be atleast three joints' Z-axes to coincide for a singularity to occur. However, since the joints in the Franka Panda arm are at offsets to each other, collinear Z-axes' case will never occur, and thus geometric intuition implies there will never be a case of singularity.

However, it's important to acknowledge that this simplification isn't entirely accurate. Singularities can arise due to various factors, including limitations in the reachable workspace and other methods of calculating the Jacobian, not solely through geometric considerations. While geometrically discerning singularities may not always be evident, there can be other types of singularities present in the system. There are ones because of Joint limits as well which is mentioned in Analysis part where our robot hit singularity when following trajectories.

For example lets take example for all angles zeros. If the end effector is in the following way, i.e. when it is at the end of its reachable workspace and it is given a velocity in an outward direction that is along z ground axis, a singularity case arises as the physical joint limits don't permit the robot arm to move in that direction.

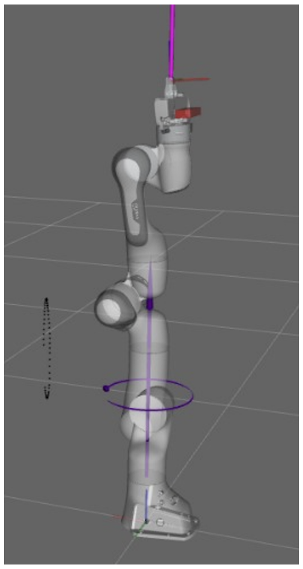


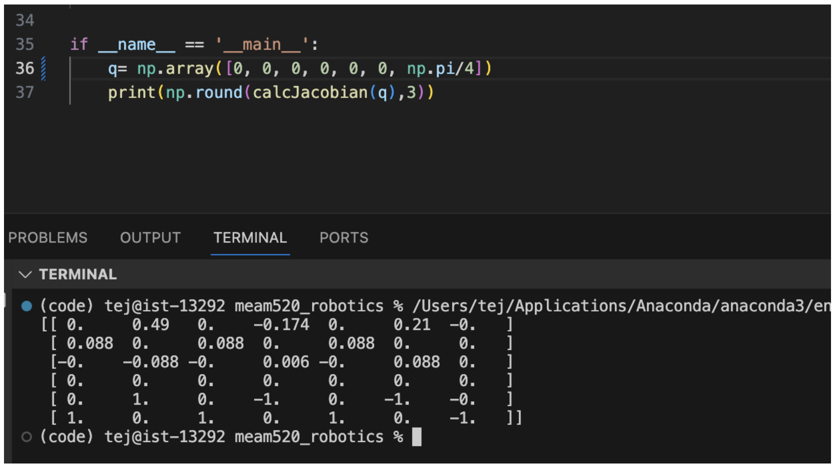
Figure 3: Robot arm at end of reachable workspace (zero configuration)

To prove mathematically that our robot reached singularity we can check using rank of the Jacobian.

If rank J < rank [J | v] : No solution exists
If rank J = rank [J | v] : A solution exists (maybe infinite solutions if n>6)

Here, if rank < 6 and Det(J) = 0, no solution exists

So when we give zero angles to all we see the one of the row of jacobian is zero which means that moving any joint the robot will never have velocity in that direction so if you give input of that velocity you will hit singularity which will have no solutions.


```
if __name__ == '__main__':
    q= np.array([0, 0, 0, 0, 0, 0, np.pi/4])
    print(np.round(calcJacobian(q),3))
```

```
(code) tej@ist-13292 meam520_robotics % /Users/tej/Applications/Anaconda/anaconda3/en
[[ 0.     0.49   0.    -0.174  0.     0.21  -0.   ]
 [ 0.088  0.     0.088  0.     0.088  0.     0.   ]
 [-0.    -0.088 -0.     0.006 -0.     0.088  0.   ]
 [ 0.     0.     0.     0.     0.     0.     0.   ]
 [ 0.     1.     0.    -1.     0.    -1.    -0.   ]
 [ 1.     0.     1.     0.     1.     0.    -1.   ]]
(code) tej@ist-13292 meam520_robotics %
```
Figure 4: Zero configuration with velocity given to first joint only

2.3 How well does your end effector track trajectories (which ones might you care about) when only position tracking is required? What if you also want to control orientation?

The Franka Panda arm has high-precision actuators and encoders which allows it to accurately track position trajectories. The Panda arm is capable of repeatability within a few millimeters due to which the arm kept repeating the same eight and ellipse trajectory with minimum error (video drive link attached in section 3).

But when only giving position control the end effector did follow the trajectory but because the orientation control was missing the robot drifted which is mentioned in details in 3.2 below.

For doing control of orientation we just have to give the orientation in velocity input vector for our inverse kinematics. Rightnow we gave nan values so just have to change that to given orientation.

3 Analysis

3.1 Discuss the Panda forward and inverse velocity kinematics in the context of your data and observations. Do your results make sense?

Yes, the results make sense. The forward velocity kinematics was observed correctly as expected from geometric intuition. As shown in the output photo in the Evaluation section, the simulation output velocity when only one joint is given a velocity was accurately observed.

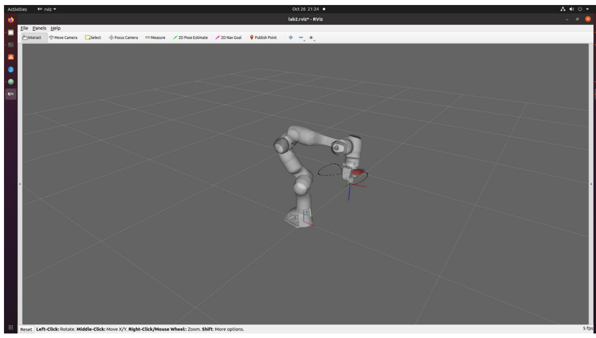
Further, inverse velocity kinematics was tested by making the robot arm follow two trajectories- making an eight-shape and making an ellipse. The code was successfully visualised in RViz and the same was observed in the robot arm.

- Ellipse Hardware Video: Ellipse trajectory tracking in Panda robot arm
- Eight Hardware Video: Eight trajectory tracking in Panda robot arm

3.2 What movements is the robot good at? What movements is it bad at? Under what circumstances might you want to use velocity control over position control (or vice versa)?

During our hardware implementation, we analyzed that the robot followed the straight line properly with no deviation. The tuned Kp gains were sufficient for it to maintain the desired position.

However, for trajectories like the figure-eight and ellipse, the robot initially followed the trajectory accurately. Nonetheless, the absence of orientation control and the absence of limitations on other joint positions and orientations caused the robot to drift in the null space, as evidenced by the initial and final positions of the robot when following the figure-eight trajectory below.


(a) Initially following correctly

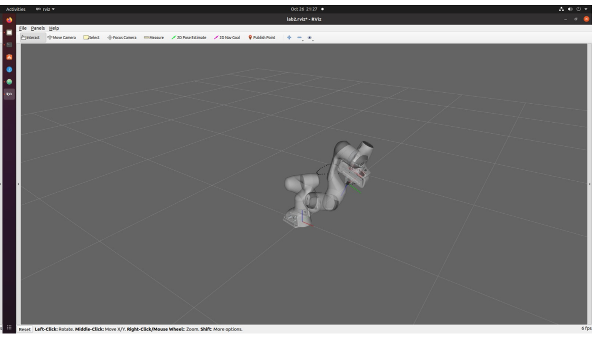

(b) Drifting of robot due to lack of orientation control

Figure 5: Simulation of Robot following Eight Trajectory

We can observe that the end-effector was able to follow the trajectory position thanks to the Kp gains that is the position is eight for both initial and final is same. However, due to the lack of orientation control, the other joints began drifting. Eventually, the robot encountered a singularity because of joint limits, which brought it to a halt.

Here is the video link that shows the robot encountering a singularity during the trajectory run, where it started drifting and ultimately stopped due to the joint limits.

- Ellipse Hardware Video : Panda arm Hitting Singularity Ellipse Trajectory
- Eight Hardware Video : Panda arm Hitting Singularity Eight Trajectory

When we require the robot to follow a continuous trajectory or engage in end-effector path planning, we often prefer velocity control over position control. This preference arises from the need to regulate the robot's speed, enabling it to stop at certain points or move swiftly through specific parts of the trajectory. However, for tasks like pick-and-place operations where only the initial and final positions and orientations matter, we can use position control without needing to concern ourselves with the intermediate velocities of the robot. Nevertheless, it's essential to take into account the maximum velocity at which the robot can move, considering the limitations imposed by its motors. In practice, we typically employ velocity control, allocating time for the robot to transition from one position to another, rather than simply providing discrete positions such as x1 and x2.