Introduction to Robotics Lab 3

Karin Dyer, Saibernard Yogendran

1 Methods

1.1 Forward Velocity Kinematics

To calculate the velocity kinematics of the Franka PANDA Arm, we can analyze the robot using the DH table (Table 1) extracted from the symbolic representation of the frames for each joint (Lab 1).

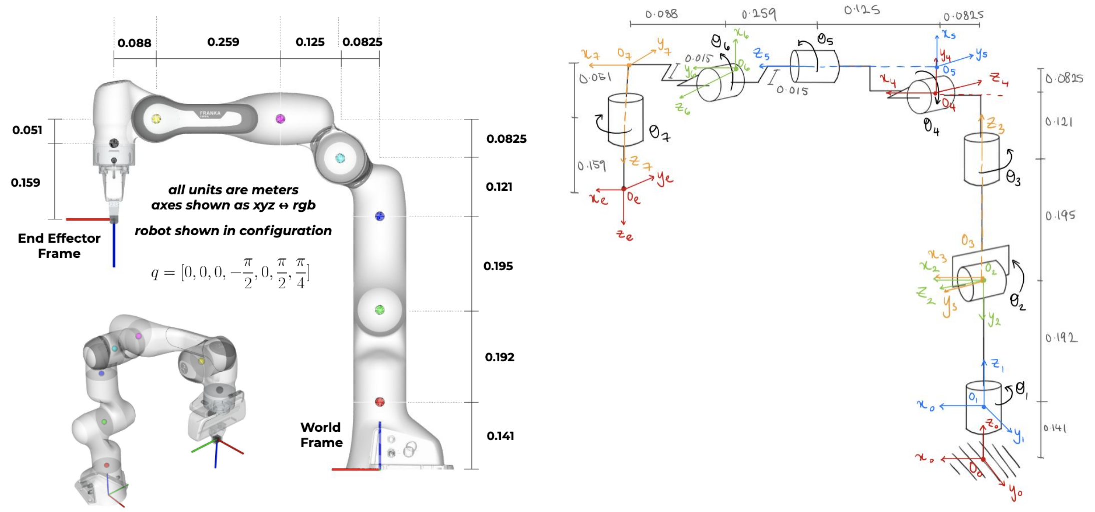

0.088
0.259
0.125
0.0825
0.051
0.159
End Effector Frame
all units are meters
axes shown as xyz ↔ rgb
robot shown in configuration
$q = [0, 0, 0, -\frac{\pi}{2}, 0, \frac{\pi}{2}, \frac{\pi}{4}]$
0.0825
0.121
0.195
0.192
0.141
World Frame
x_7 O_7 y_7
θ_6 x_6 y_6 O_6 z_6
0.015
z_5 θ_5
0.015
x_5 y_4 y_5 O_5
x_4 z_4 O_4 θ_4
θ_7 z_7 y_e x_e O_e z_e
z_3 θ_3
O_3 x_3 x_2 O_2 z_2 y_3 y_2 θ_2
z_1 θ_1 O_1 x_0 z_0 y_1
x_0 O_0 y_0

Figure 1: Symbolic representation of FRANKA Emika Panda Arm

Table 1: DH table

Link	a_i	α_i	d_i	θ_i
1	0	0	0.141	0
2	0	$-\frac{\pi}{2}$	0.192	θ_1
3	0	$\frac{\pi}{2}$	0	θ_2
4	0.825	$\frac{\pi}{2}$	0.316	θ_3
5	0.825	$\frac{\pi}{2}$	0	$\theta_4 - \pi$
6	0	$\frac{\pi}{2}$	0.384	θ_5
7	0.088	$-\frac{\pi}{2}$	0	$\pi - \theta_6$
8	0	0	0.210	$\theta_7 - \frac{\pi}{4}$

The transformation matrix for each joint, T_1^0, T_2^0, T_3^0, T_4^0, T_5^0, T_6^0, T_7^0, T_e^0 with respect to the world frame is then computed from the A_i matrices and the equation:

$$T_n^0 = A_1 A_2 A_3 \ldots A_n$$

Where:

$$A_i = \begin{bmatrix} C\theta_i & -S\theta_i C\alpha_i & S\theta_i S\alpha_i & a_i C\theta_i \\ S\theta_i & C\theta_i C\alpha_i & -C\theta_i S\alpha_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3: A$_i$ matrix layout

For each transformation matrix, the translation matrix o_i^0 is extracted from the first 3 rows of the fourth column of the correlated transformation matrix. For frames with origins off the joint center, the translation matrices must be shifted by multiplying with the translation between the frame origin and the joint center. The rotation matrices R_i^0 were extracted from the first 3×3 matrix in the transformation matrix.

The linear velocity jacobian was calculated from the equation:

$$J_{vi} = \hat{z}_{i-1} \times (o_n^0 - o_{i-1}^0)$$

Where $\hat{z}_{i-1}$ is the third column of R_i^0.

The linear velocity jacobian is a 3×7 matrix of the linear velocities in the x, y, and z direction for each joint with respect to the world frame, $[J_2, J_3, J_4, \cdots, J_e]$.

The angular velocity jacobian was calculated from the equation:

$$J_\omega = \hat{z}_{i-1}$$

The forward velocity jacobian is a 6×7 matrix of J_v and J_ω stacked vertically.

The forward velocity of the end effector is found from the equation:

$$\text{velocity} = \begin{bmatrix} \vec{V} \\ \vec{\omega} \end{bmatrix} = \begin{bmatrix} \dot{x}_v \\ \dot{y}_v \\ \dot{z}_v \\ \dot{x}_\omega \\ \dot{y}_\omega \\ \dot{z}_\omega \end{bmatrix} = \begin{bmatrix} J_{2v} & J_{3v} & J_{5v} & J_{6v} & J_{7v} & J_e \\ J_{2\omega} & J_{3\omega} & J_{5\omega} & J_{6\omega} & J_{7\omega} & J_e \end{bmatrix} \begin{bmatrix} \dot{\vec{q}}_1 \\ \dot{\vec{q}}_2 \\ \dot{\vec{q}}_3 \\ \dot{\vec{q}}_4 \\ \dot{\vec{q}}_5 \\ \dot{\vec{q}}_6 \\ \dot{\vec{q}}_7 \end{bmatrix}$$

Where the forward velocity is a 6×1 matrix of the linear and angular velocity of the end effector with respect to the base frame.

1.2 Inverse Velocity Kinematics

The inverse kinematics uses the inverse Jacobian and the end effector velocity to find q_{dot} which is a 7×1 matrix of the individual joint velocities.

1.3 Code (Saibernard's)

calcJacobian:

- Initialize constants to store link parameters and positions of joint centers relative to the corresponding joint frame.
- For the given configuration of q, calculate the corresponding A matrices of each link.
- For each joint, post multiply the homogeneous transformation matrices to obtain T^0_n and take the dot product of T^0_n and the position of the joint center relative to the joint frame to obtain the position of the joint center with respect to the world frame.
- For each joint, extract the rotational matrix R^0_n and the translational matrix o^0_n from the transformation matrix T^0_n.
- To find the linear velocity, calculate the Jacobian relative to each joint by taking the cross product of the z rotation of R^0_{n-1} and $(o^0_e - o^0_{n-1})$.
- The linear velocity Jacobian is a 1×6 array of each joint Jacobian stacked horizontally.
- To find the angular velocity, calculate the Jacobian relative to each joint by taking the z rotation of R^0_{n-1}.
- The angular velocity Jacobian is a 1×6 array of each joint Jacobian stacked horizontally.
- The velocity Jacobian is a 6×7 matrix of the linear and angular velocity Jacobians stacked vertically.

IK_velocity:

- Initialize q_{dot} (dq) as a 1×7 matrix.
- Extract the velocity Jacobian by calling calcJacobian(q_in).
- Extract the velocity by horizontally stacking the linear (v_in) and angular (omega_in) velocity inputs. Reshape to be a 6×1 vector.
- Store the indexes of nan values within the velocity vector.
- Delete the Jacobian row corresponding to the index of nan values by using isnan function and picking out the rows which has nan values.
- Delete the velocity element corresponding to the index of nan values by using isnan function and picking out the rows which has nan values..
- Take the least squares solution of the updated Jacobian and velocity matrices to find dq. Reshape dq to be a 1×7 vector.

2 Evaluation

2.1 Testing Process

For testing, we first tested scenarios where the robot starts in the zero configuration, $q = [0,\ 0,\ 0,\ \ -\frac{\pi}{2},\ 0, \frac{\pi}{2},\ \frac{\pi}{4}]$, and only one joint moves. We could verify the calculated velocity by comparing with the physics formula: $v\ =\ \omega r$. As we know the radius (the orthogonal distance from the robot base to the joint) from Figure 1, and the joint velocity input, the calculated linear velocity should match the code's calculated velocity. This test was done for joints where the end effector linear velocities were in only one direction (joints 1, 3, 5, and 7). The consistency between the computed and theoretical velocities validated the jacobian calculations in the function.

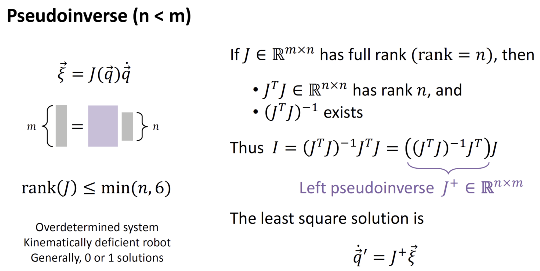
We then used the visualize.py simulation to check the joint velocity with the expected velocity vector direction relative to the world frame. Consistency between the visualized and calculated vector directions further validated the jacobian calculations. Triangulating across three different configurations ensures the jacobian works in multiple scenarios.

Table 2: Joint Configurations for Visualize.py Checks

Configuration 1	Configuration 2	Configuration 3
$q\ =\ [0,\ 0,\ 0, -\frac{\pi}{2},\ 0,\ \frac{\pi}{2},\ \frac{\pi}{4}]$	$q\ =\ [\frac{\pi}{2},\ 0,\ \frac{\pi}{4}, -\frac{\pi}{2}, -\frac{\pi}{2}, \frac{\pi}{2},\ 0]$	$q\ =\ [0,\ 0, -\frac{\pi}{2}, -\frac{\pi}{4},\ \frac{\pi}{2},\ \pi,\ \frac{\pi}{4}]$

Each check consisted of setting the target joint velocity to 1 with the other joint velocities set to 0.

We also tested cases where there are singularities. Singularities occur when the robot loses one or more degrees of freedom. Geometrically, this occurs when any two wrist axes align (Joint 1, 2, and 3). Singularities can be found mathematically when the determinant of the linear velocity Jacobian = 0 for a square Jacobian. As the linear velocity Jacobian is a 3 × 7 matrix, we can use kinematic decoupling to look at just the 'wrist', joints 1, 2, and 3, to obtain a square Jacobian matrix. If the determinant of this square Jacobian is zero, a singularity is occurring. When singularity occurs, there will be one solution or zero solutions. In this case, if we set joint 2 = 0, we will get a unique solution of the end effector, provided other joint velocities are not zero. This means that the robot lose one degree of freedom and this provided a unique solution.



Pseudoinverse (n < m)

$$\vec{\xi} = J(\vec{q})\dot{\vec{q}}$$

$$\text{rank}(J) \leq \min(n, 6)$$

Overdetermined system
Kinematically deficient robot
Generally, 0 or 1 solutions

If $J \in \mathbb{R}^{m \times n}$ has full rank ($\text{rank} = n$), then

- $J^T J \in \mathbb{R}^{n \times n}$ has rank n, and
- $(J^T J)^{-1}$ exists

Thus $I = (J^T J)^{-1} J^T J = \underbrace{\left((J^T J)^{-1} J^T\right)}_{\text{Left pseudoinverse } J^+ \in \mathbb{R}^{n \times m}} J$

The least square solution is

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

To test the inverse kinematics code, we implemented the follow.py code and tested trajectory tracking for position and orientation. We tracked a line, a circle, an ellipse, and a figure eight. For tracking a line along the z axis, the equation for the desired end effector position is xdes = x0 + [0, 0, L*f*t] where f is the frequency in Hz of the trajectory and t is the time since the start. The equation for the desired end effector velocity is vdes = [xdes/t]. For tracking an ellipse along the z-y plane, the equation for the desired end effector position is xdes = [0, ry*sin(f*t), rz*cos(f*t)] where ry and rz is the radius in meters of the y and z portions of the ellipse respectively. The equation for the desired end effector velocity is the time derivative of xdes: vdes = [0, -ry*f*cos(f*t), rz*f*sin(f*t)]. Tracking a circle can be obtained with the condition ry = rz. We also changed the velocity of the end effector to investigate the impact of velocity on the tracking quality.

We did not conduct hardware testing due to the difficulties in obtaining the linear velocities of the end effector for trajectory tracking.

2.2 Testing Results

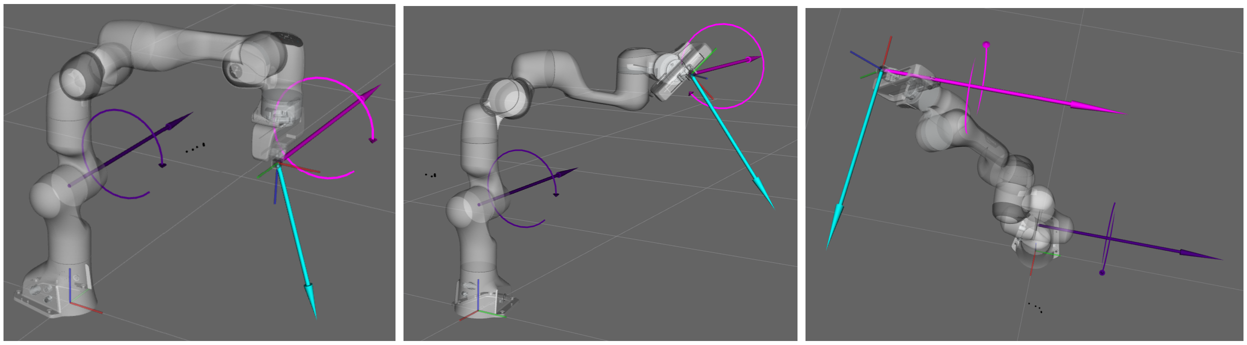
With the robot starting in the zero position, each joint velocity was set to 1 rad/s with the remaining joint velocities set to zero. The computed linear velocities were consistent with the theoretical velocities from the equation $v = \omega r$ which is the end effector linear velocity with respect to each joint. r is the perpendicular distance from the end effector to the axis of rotation. Thus, r can be r_x, r_y, or r_z depending on the axis of rotation of the moving joint. Results are shown in Table 3 with the perpendicular radius in bold.

Table 3: Computed and Theoretical Linear Velocity for Joint # = 1

Joint Number	Computed Linear Joint Velocity	Theoretical Linear Joint Velocity wrt base frame	r_x	r_y	r_z
1	[0, 0.55, 0]	0.55 in y direction	**0.5545**	0	0.522
3	[0, 0.55, 0]	0.55 in y direction	**0.5545**	0	0.189
5	[0, 0.21, 0]	0.21 in y direction	0.347	0	**0.21**
7	[0, 0, 0]	0 in x direction	**0**	0	0.159

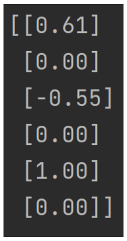
The visualize checks for joint 2 and joint 4 are shown below (Figure 4 and 5). The world frame axes for x, y, and z are shown in red, green, and blue respectively. Figures 4 and 5 compare the simulated linear end effector velocity (in cyan) with the calculated linear end effector velocity. The angular end effector velocity is shown in magenta, and matches the direction of the input joint velocity with respect to the world frame. Similarly, the remaining joints were also tested with the following code and compared with the visualization (visualize.py). All seven joints were consistent between the calculated velocities and the visualized velocity vector directions.

Joint 2: $q_{dot} = [0, 1, 0, 0, 0, 0, 0]$

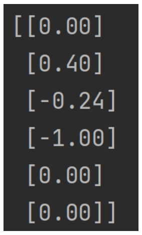


Configuration 1 Configuration 2 Configuration 3

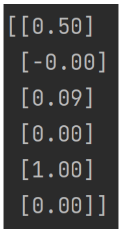
Calculated joint velocity: Calculated joint velocity: Calculated joint velocity:



```
[[0.61]
 [0.00]
 [-0.55]
 [0.00]
 [1.00]
 [0.00]]
```



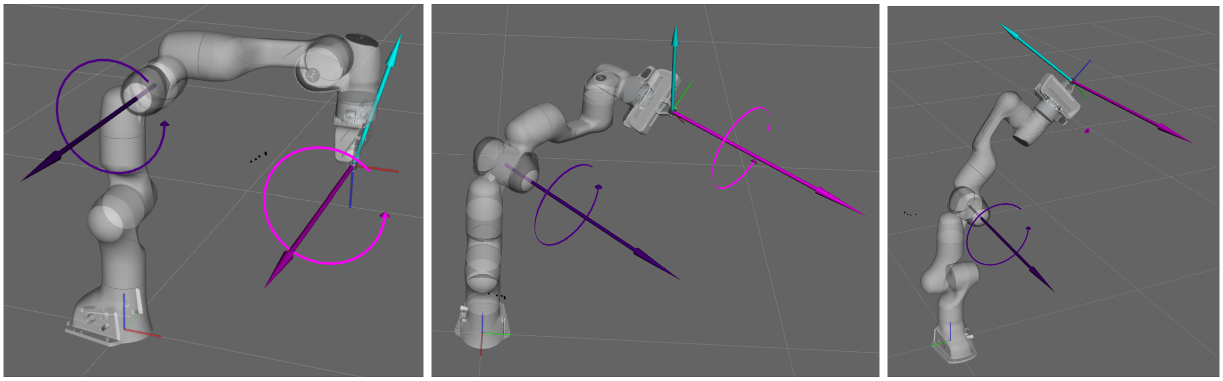
```
[[0.00]
 [0.40]
 [-0.24]
 [-1.00]
 [0.00]
 [0.00]]
```



```
[[0.50]
 [-0.00]
 [0.09]
 [0.00]
 [1.00]
 [0.00]]
```

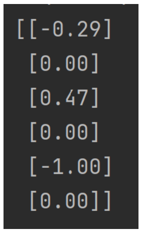
Figure 4: Visualize.py configuration checks for Joint 2

Joint 4: $q_{dot} = [0, 0, 0, 1, 0, 0, 0]$

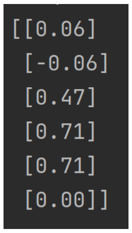


Configuration 1 Configuration 2 Configuration 3

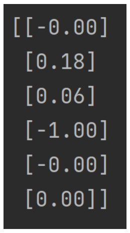
Calculated joint velocity: Calculated joint velocity: Calculated joint velocity:



```
[[-0.29]
 [0.00]
 [0.47]
 [0.00]
 [-1.00]
 [0.00]]
```



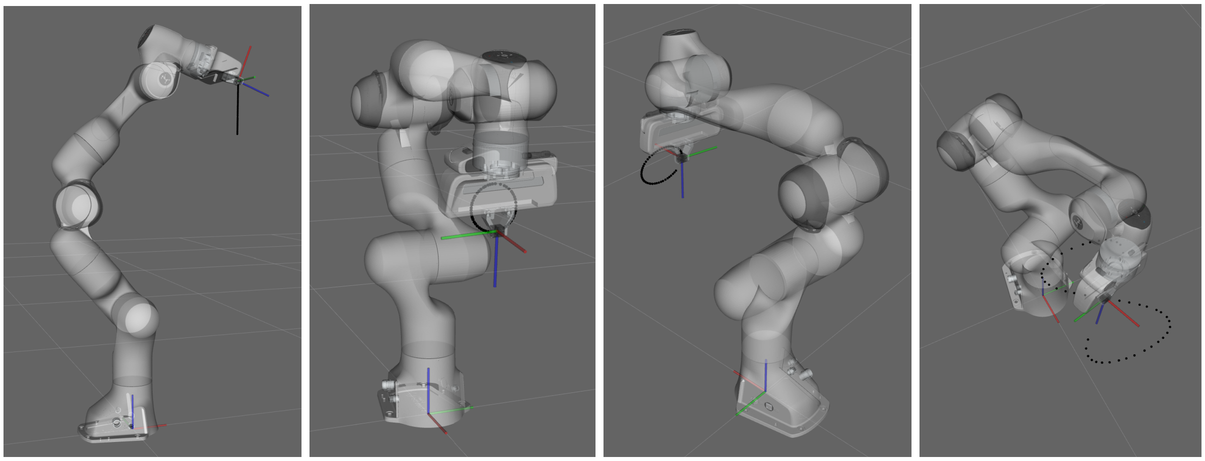
```
[[0.06]
 [-0.06]
 [0.47]
 [0.71]
 [0.71]
 [0.00]]
```



```
[[-0.00]
 [0.18]
 [0.06]
 [-1.00]
 [-0.00]
 [0.00]]
```

Figure 5: Visualize.py configuration checks for Joint 4

The resulting trajectories from implementing the follow.py code are shown in Figure 6.



Line Circle Ellipse Eight

Figure 6: Robot tracking various trajectories using follow.py

Successful tracking of a circular and elliptical trajectory verifies that the inverse kinematics velocity and ellipse velocity equations are correct. Figure 7 shows the successful tracking of an ellipse trajectory in two different orientations.

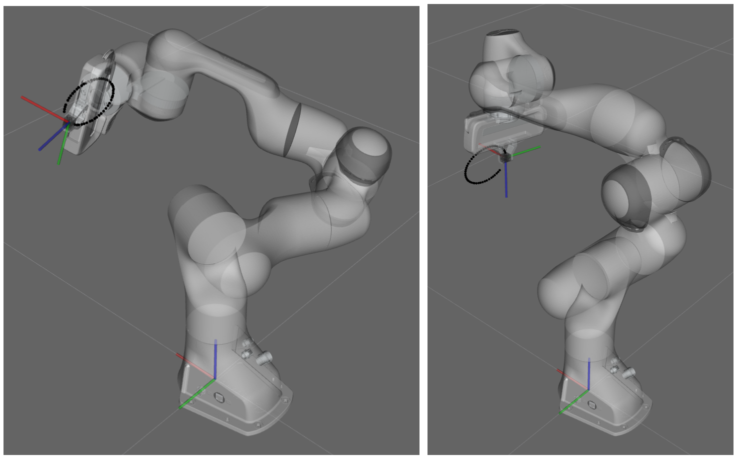


Figure 7: Robot tracking ellipse trajectory in different orientations

Figure 8 shows the tracking of a figure eight in three different orientations. It is worth noting that in the third configuration, the trajectory path and the robot arm collide (see Analysis). Additionally, decreasing the velocity of the end effector for the figure eight trajectory yielded a smoother trajectory.

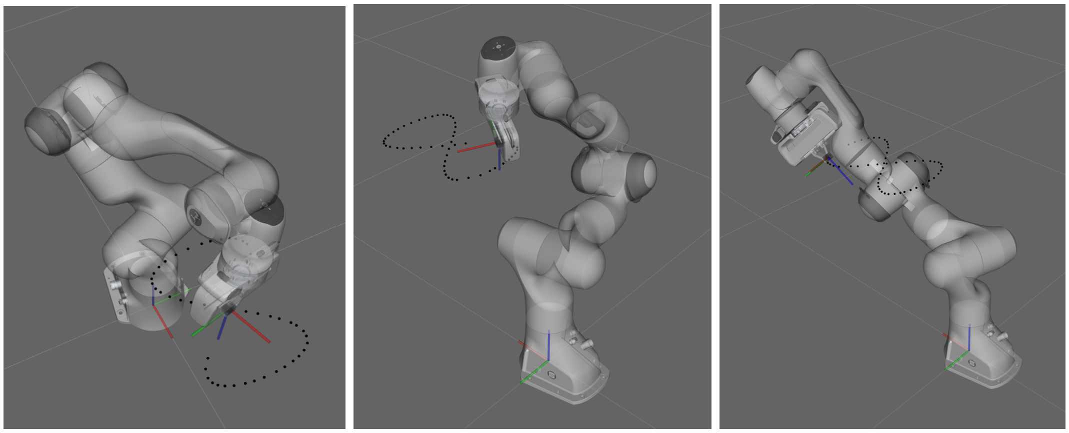


Figure 8: Robot tracking figure eight trajectory in different orientations

3 Analysis

In the cases where only one joint moves, the end effector velocity was consistent with the theoretical velocity for joints where the end effector linear velocities were in only one direction (see Testing Process). Table 3 shows that the calculated end effector velocity and the theoretical end effector velocity match. In comparing the computed velocity vector directions with the simulation in visualize.py (Figure 4 and 5), testing of each of the seven joints confirmed that the jacobian code is correct. The inverse velocity kinematics successfully tracked the desired trajectories of a line, circle, ellipse, and figure eight.

For trajectory tracking, the robot was good at tracking a line, circle, ellipse, and figure eight when starting in the zero position of $q = [0,\ 0,\ 0, -\frac{\pi}{2},\ 0,\ \frac{\pi}{2},\ \frac{\pi}{4}]$. However, when investigating the robots ability to track trajectories in different configurations, there were cases when the trajectories collided with the robot arm (Figure 8). This is problematic if the target trajectory is intended to avoid an external obstacle, as the robot arm will collide with the obstacle the trajectory is trying to avoid.

Velocity control may be used in circumstances where speed and time is important such as transporting goods. In an industry setting, if a robot is moving parcels or picking up and placing delicate objects, velocity control is prioritized over position control. Position control is important in circumstances where accuracy is needed for fitting precision parts such as fitting a windshield or door to a car on a factory line. Automotive lines use a combination of velocity and position control on the assembly line. Velocity control is used to transport the windshield to the car assembly, then position control is used to fit the windshield accurately in the car.