A Robot Manipulation Framework

I. Introduction

In this wok, implemented both forward kinematics (FK) and inverse kinematics (IK) functions to determine joint angles and velocities for controlling a 6-degree-of-freedom robot arm. The PyBullet simulation environment was employed to validate the functionality of these kinematic functions. Specifically, manipulated a UR5 robot arm, utilizing the simulation to perform a block insertion task as part of the verification process.

II. Forward kinematics function implementation

Function Implementation:

The forward kinematics problem is formulated as follows: given the joint angles, the system can calculate the position of the end-effector point in Cartesian space as well as its velocities. This is achieved by computing the transformation matrix between each joint using the Denavit-Hartenberg (DH) convention. The transformation matrix of J_{i-1} to J_i is:

$$\begin{bmatrix} C(\theta_n) & -S(\theta_n)C(\alpha_n) & S(\theta_n)S(\alpha_n) & r_nC(\theta_n) \\ S(\theta_n) & C(\theta_n)C(\alpha_n) & -C(\theta_n)S(\alpha_n) & r_nS(\theta_n) \\ 0 & S(\alpha_n) & C(\alpha_n) & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By the transformation matrix we could compute T_0^1 to T_5^6, by multiplying the matrix we could get the transform from the base to the 6th joint T_6.

To compute the velocities of the end-effector, solve the forward Jacobian problem by computing the Jacobian Matrix, since all the joints of the UR5 arms are all revolute joint (Joint angle is the variable), the Jacobian Matrix is

$$J = \begin{bmatrix} J_v \\ J_w \end{bmatrix} = \begin{bmatrix} R_{i-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times (d_n^0 - d_{i-1}^0) \\ R_{i-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{bmatrix}$$

With the transformation matrix and Jacobian Matrix, implemented the function as the following code shown as follow:

```
def your_fk(DH_params : dict, q : list or tuple or np.ndarray,
base_pos) -> np.ndarray:
    # robot initial position [0,0,0,0,0,0]
    base_pose = list(base_pos) + [0, 0, 0]
```

```
    assert len(DH_params) == 6 and len(q) == 6, f'Both DH_params and q
should contain 6 values,\n' \
                                                f'but get len(DH_params) =
{DH_params}, len(q) = {len(q)}'

    # initialize
    A = get_matrix_from_pose(base_pose) # a 4x4 matrix, type should be
np.ndarray
    jacobian = np.zeros((6, 6)) # a 6x6 matrix, type should be
np.ndarray

    # foward kinematics
    T6 = np.eye(4)
    transforms = []
    for i, parameters in enumerate(DH_params):
        a, d, alpha = parameters['a'], parameters['d'],
parameters['alpha']
        theta = q[i]
        # transform matrix
        transform = np.array([[np.cos(theta),-
np.sin(theta)*np.cos(alpha),
np.sin(theta)*np.sin(alpha),a*np.cos(theta)],
                              [np.sin(theta),
np.cos(theta)*np.cos(alpha),-
np.cos(theta)*np.sin(alpha),a*np.sin(theta)],
                              [0, np.sin(alpha), np.cos(alpha), d],
                              [0, 0, 0, 1]])
        transforms.append(transform)
        T6 = T6.dot(transform)

    A = A.dot(T6)

    # foward jacobian
    # https://automaticaddison.com/the-ultimate-guide-to-jacobian-
matrices-for-robotics/
    transform = np.eye(4)
    d = T6[0:3,3]
    for i in range(6):
```

```
        J_R = transform[0:3,0:3]
        J_D = transform[0:3,3]
        Jv = cross(J_R.dot([0,0,1]),(d-J_D))
        Jw = J_R.dot([0,0,1])
        jacobian[0:3,i] = Jv
        jacobian[3:6,i] = Jw
        transform = transform.dot(transforms[i])

    # adjustment
    adjustment = np.asarray([[ 0, -1,  0],
                             [ 0,  0,  0],
                             [ 0,  0, -1]])
    A[:3, :3] = A[:3,:3] @ adjustment
    pose_7d = np.asarray(get_pose_from_matrix(A,7))

    return pose_7d, jacobian
```

In this function, the Denavit-Hartenberg (DH) model is established, following the Classic DH Conventions. The base pose is defined with specific base position coordinates, while the rotation is set to (0, 0, 0). With the base pose, initialize a base matrix $\boldsymbol{A}$ and the Jacobian matrix.

Then compute the transformation matrix between each joint and saved them into the list "transforms". Compute T_6 by multiply each transform in transforms. The final pose matrix is computed by $\boldsymbol{A}$ dot T_6.

Then calculate the Jacobian matrix, each column of the matrix could be solved by computing J_V and J_w. In the end, the function returns the adjusted 7D pose and the calculated Jacobian matrix.

Difference between classic and modify conventions

It is important to note that the difference between the classic D-H conventions and the modified D-H conventions, as known as Craig's conventions. The classic conventions set the coordinates of j_i on the axis of j_{i+1} (next joint). In contrast, Craig's conventions put the axis on the current axis itself. The D-H model the work is shown in the following section.

D-H model examples (DH convention)

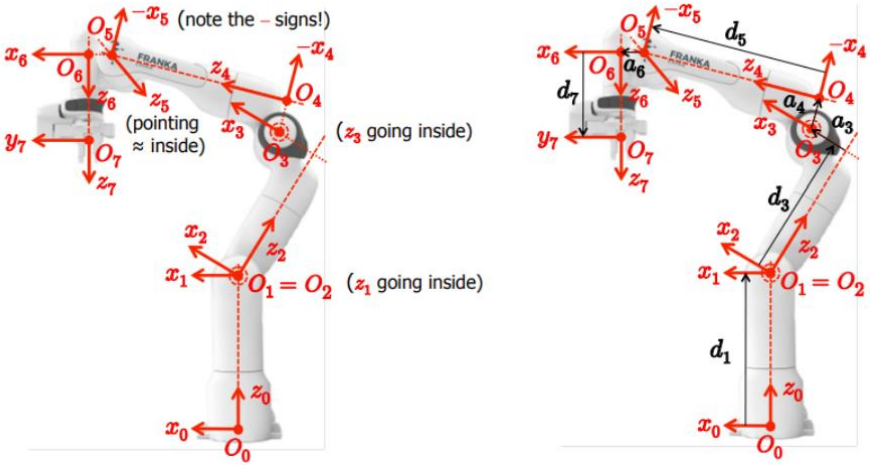

Fig (1). Example robot arm coordinate systems.

i	d	α(rad)	a	θ(rad)
1	d_1	$-\pi/2$	0	θ_1
2	0	$\pi/2$	0	θ_2
3	d_3	$-\pi/2$	a_3	θ_3
4	0	$\pi/2$	a_4	θ_4
5	d_5	$-\pi/2$	0	θ_5
6	0	$\pi/2$	a_6	θ_6
7	d_7	0	0	θ_7

Table (1). D-H table example format.

III. Inverse Kinematic function implementation

Function Implementation:

Various methods exist to address the inverse kinematic (IK) problem, including algebraic and geometric approaches. However, many of these methods involve complex mathematical operations. In practical applications, the pseudo-inverse method is frequently employed due to its simplicity and effectiveness.

The pseudo-inverse method involves calculating the pseudo-inverse of the Jacobian matrix. The Jacobian matrix represents the relationship between changes in joint angles and changes in end-effector position. The pseudo-inverse is used to approximate the inverse of the Jacobian matrix, enabling the calculation of joint angles that achieve a desired end-effector position.

This method is particularly useful when dealing with redundant or over-constrained robotic systems, where there may be more joints than necessary to achieve a specific task. The pseudo-inverse provides a way to find a solution even when an exact solution does not exist, offering practical advantages in real-world

applications. The inverse kinematics function code is shown as follows:

```
def your_ik(robot_id, new_pose : list or tuple or np.ndarray,
               base_pos, max_iters : int=1000, stop_thresh :
float=.001):
    joint_limits = np.asarray([
            [-3*np.pi/2, -np.pi/2], # joint1
            [-2.3562, -1],          # joint2
            [-17, 17],              # joint3
            [-17, 17],              # joint4
            [-17, 17],              # joint5
            [-17, 17],              # joint6
        ])

    # get current joint angles and gripper pos, (gripper pos is fixed)
    num_q = p.getNumJoints(robot_id) # numbers of joint
    q_states = p.getJointStates(robot_id, range(0, num_q)) # get joint
states

    tmp_q = np.asarray([x[0] for x in q_states][2:8]) # current joint
angles 6d

    dh_params = get_ur5_DH_params()
    iteration = 0
    step_size = 0.05
    IK_flag = True

    while(IK_flag and (iteration <= max_iters)):

        # compare the current cartesian poses to the target cartesian
poses
        pose_7d, jacobian = your_fk(dh_params, tmp_q, base_pos) #
current poses and jacobian
        delta = get_matrix_from_pose(new_pose) @
np.linalg.inv(get_matrix_from_pose(pose_7d))
        delta_x = pose_7d_to_6d(get_pose_from_matrix(delta))

        # Pseudo Inverse Problem
```

```
        J = jacobian
        Jt = np.transpose(jacobian)
        delta_theta = step_size * Jt @ np.linalg.inv(J@Jt) @ delta_x

        # updating current joint angle
        tmp_q += delta_theta

        # Check Joint Limitation
        for i, joint_limit in enumerate(joint_limits):
            if(tmp_q[i] < joint_limit[0]):
                tmp_q[i] = joint_limit[0]
            elif(tmp_q[i] > joint_limit[1]):
                tmp_q[i] = joint_limit[1]

        # Threshold to stop
        norm = np.linalg.norm(delta_x)
        if(norm >= stop_thresh):
            IK_flag = True
        else:
            IK_flag = False

        iteration += 1

    if(iteration >= max_iters):
        print("Fail to find IK solution with |delta x| = %d" % norm)

    return list(tmp_q) # 6 DoF
```

In the beginning, joint limits for each joint are defined in the “joint_limits” array. This array represents the allowable range of motion for each joint in radians. The function starts by obtaining the current joint angles (tmp_q) from the robot simulation using PyBullet.

Inside the IK loop, the current joint angles are used to calculate the forward kinematics (FK) using the FK function. The FK provides the current Cartesian pose and the Jacobian matrix representing the sensitivity of the end-effector position to changes in joint angles.

The difference between the target Cartesian pose (new_pose) and the current pose is calculated as delta. This difference is then transformed into a 6D vector

(delta_x) using the pose_7d_to_6d function.

The function then solves the IK problem using the pseudo-inverse method. It calculates the pseudo-inverse of the Jacobian matrix and multiplies it by the Cartesian pose difference (delta_x) to obtain the change in joint angles (delta_theta).

The new joint angles (tmp_q) are updated by adding the change in joint angles. These updated joint angles are then checked against the defined joint limits to ensure they fall within the allowed range.

The process iterates until the solution converges or until the maximum number of iterations is reached. The convergence is determined by checking if the norm of the Cartesian pose difference is below a specified threshold.

Encountered Problem while implementation:

In the initial implementation, I overlooked incorporating a crucial convergence condition for the Cartesian pose difference. This omission hindered the solver's ability to converge effectively.

To rectify this, I introduced a convergence condition based on the norm of delta_x, ensuring that the difference between the current and target Cartesian poses is below a specified threshold. This condition is vital for preventing the solver from running indefinitely and ensures that joint angles are adjusted until the end-effector closely matches the target pose.

With this enhancement, the inverse kinematics solver now iteratively refines joint angles until the specified convergence threshold is met. This improvement promotes a more reliable and stable solution, enabling the robot to accurately achieve and maintain the desired end-effector pose.

IV. Transport network manipulation pipeline

After finishing the previous kinematics functions, this work go through a block insertion task that the robot will grasp a block and insert it into a fixture using a method called Transporter Network (using ravens environment). Then compare the simulation results between the implemented IK function and the PyBullet IK function.

The manipulation framework contains 4 sub-tasks:

1. to prepick: move the robot arm to a prepick pose (above the block)
2. grasping: move the robot arm downward to grasp the block using suction cup
3. to preplace: move the robot arm to a preplace pose (above the fixture)
4. insertion: insert the block into the fixture

Results:

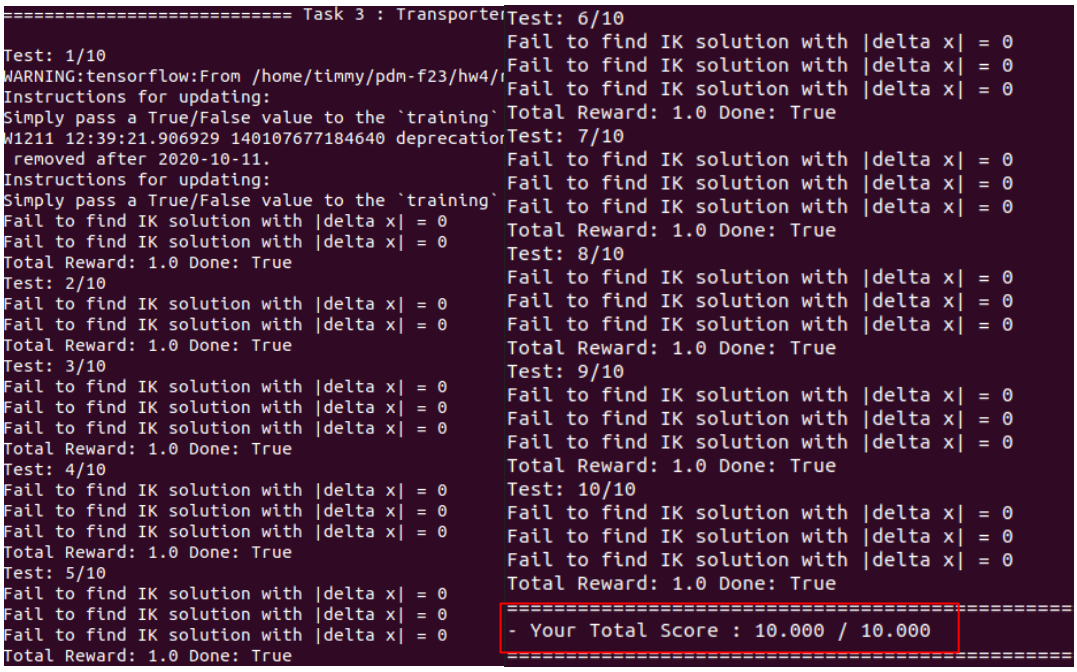

Fig (2). Result using implemented IK function

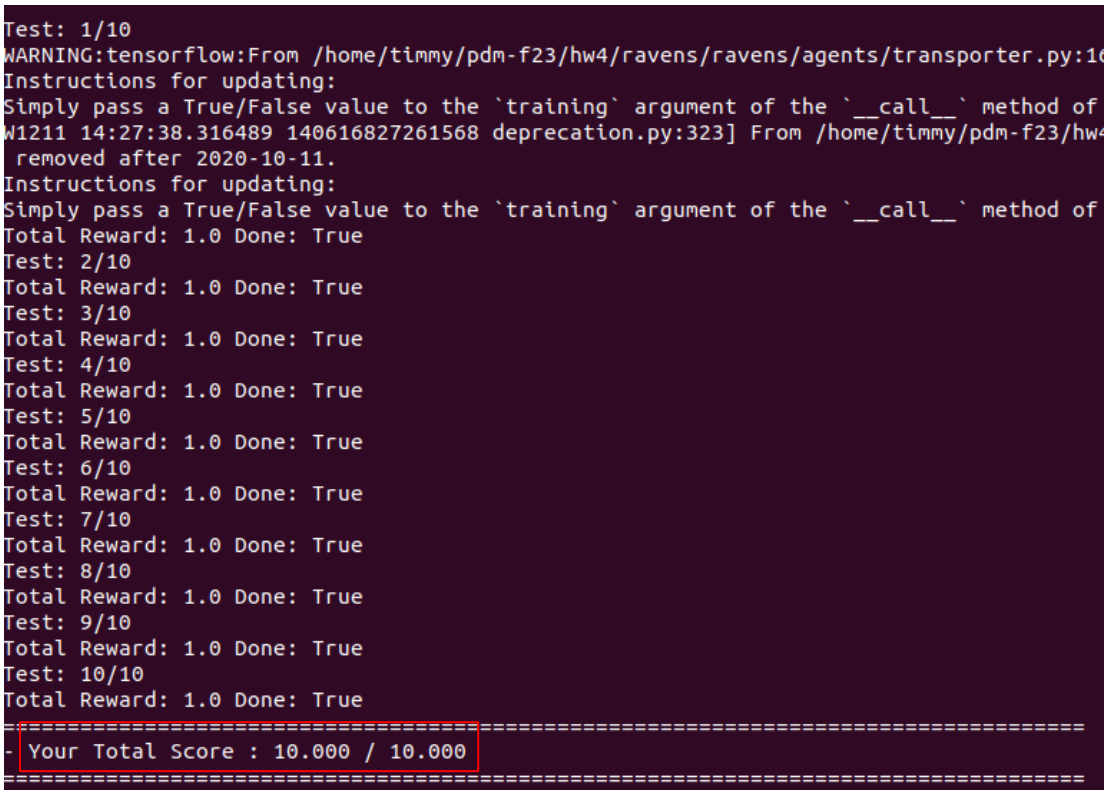

Fig (3). Result using PyBullet IK function

The results indicate that both inverse kinematics (IK) functions successfully completed the manipulation tasks across 10 test cases. However, during execution,

PyBullet's built-in function exhibited faster performance. Occasionally, the implemented IK function produced errors indicating the inability to find a valid IK solution. This suggests potential challenges or limitations in the implemented IK algorithm, leading to occasional failure in achieving the desired end-effector poses. Further optimization and refinement may be necessary to enhance the robustness and reliability of the implemented IK solver.