Perception and Decision Making in Intelligent Systems

Homework 3: A Robot Manipulation Framework

109350008 張詠哲

1. About task 1

1.1 implementation

```
# Accumulate forward kinematics transformations
transformation = np.eye(4)  # initialize
positions = []
z_axes = []

for i, param in enumerate(DH_params):
    theta = q[i]
    a = param['a']
    d = param['d']
    alpha = param['alpha']

    A_i = 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]
    ])

    positions.append(transformation[:3, 3])
    z_axes.append(transformation[:3, 2])

    transformation = transformation @ A_i

A = A @ transformation

# Jacobian matrix
for i in range(6):
    z = z_axes[i]

    p = transformation[:3, 3] - positions[i]
    jacobian[:3, i] = np.cross(z, p)
    jacobian[3:, i] = z

##############################################
```

Above code is the implementation. For each joint, a transformation matrix AA_i is calculated based on the D-H parameters (d, a, α, θ) and is sequentially multiplied into the overall transformation matrix transformation to determine the end-effector's position and orientation. The transformation matrix of each joint follow the bellowing equation.

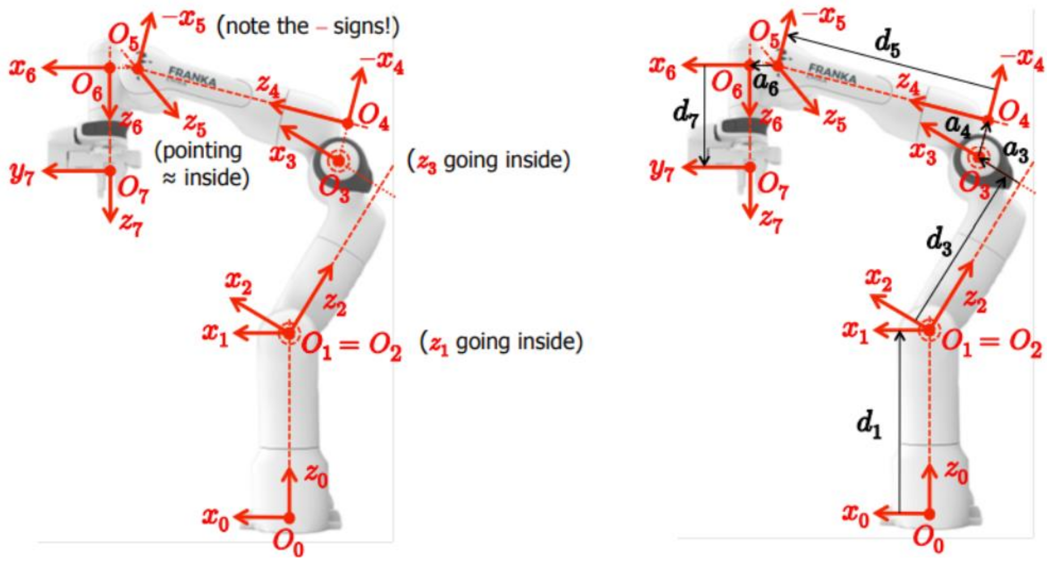
$$T_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, storing each joint position and z-axis for Jacobian calculation, and finally computing the Jacobian matrix for the manipulator.

1.2 Difference of convention D-H & modified D-H

The main difference between standard D-H convention and Craig's modified D-H convention is the coordinate frame assignment. In the standard D-H convention, the z-axis is aligned with the joint axis, and the x-axis points from one link to the next and is perpendicular to the z-axis of that link. While in Craig's modified D-H convention, the x-axis is aligned along the shortest distance between the two successive z-axes, simplifying calculations for certain types of robotic configurations.

1.3 Fill the table



Index	d	α	a
1	d_1	$\frac{\pi}{2}$	0
2	0	$\frac{\pi}{2}$	0
3	d_3	$\frac{\pi}{2}$	a_3
4	0	$-\frac{\pi}{2}$	$-a_4$
5	d_5	$\frac{\pi}{2}$	0
6	0	$\frac{\pi}{2}$	a_6
7	d_7	0	0

2. About task 2

2.1 implementation

```
# pseudo-inverse
target_pose = np.asarray(new_pose[:6])

for _ in range(max_iters):
    current_pose, jacobian = your_fk(get_ur5_DH_params(), tmp_q, base_pos)

    error = target_pose - current_pose[:6]
    if np.linalg.norm(error) < stop_thresh:
        break

    dq = np.dot(pinv(jacobian), error)
    tmp_q += dq
```

As above code shows that I first get the end effector pose and Jacobian matrix by FK which is implemented in task 1. Then, apply the pseudo-inverse method helps adjust q q to bring the end effector closer to the target pose with each iteration until the error is below a threshold.

2.2 What problems do you encounter and how do you deal with them

It's hard to understand the mathematical derivation of IK, so I asked ChatGPT for help and did some online research.

2.3 Bonus

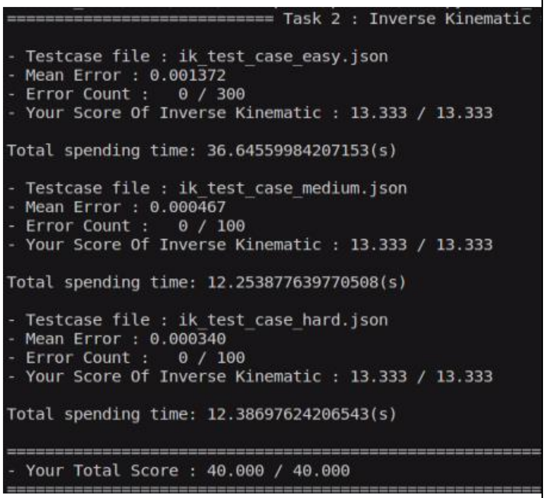
```
# LM (Levenberg-Marquardt)
damping = 0.001
def cost_func(q):
    current_pose, _ = your_fk(get_ur5_DH_params(), q, base_pos)
    return (new_pose[:6] - current_pose[:6]).flatten()

result = least_squares(cost_func, tmp_q, method='lm', ftol=stop_thresh, max_nfev=max_iters, xtol=damping)
tmp_q = result.x
```

I also implement the **Levenberg-Marquardt (LM)** method, which helps address the limitations of the pseudo-inverse approach. The LM method provides better results in terms of stability and convergence, particularly when dealing with near-singularities or target poses.

3. About task 3

3.1 Compare your results between your_ik function and pybullet_ik

Following figures are the execution time of yor_ik (LM method) and pybullet_ik

<table>
<tr>
<td>
============================ Task 2 : Inverse Kinematic

- Testcase file : ik_test_case_easy.json
- Mean Error : 0.001372
- Error Count :   0 / 300
- Your Score Of Inverse Kinematic : 13.333 / 13.333

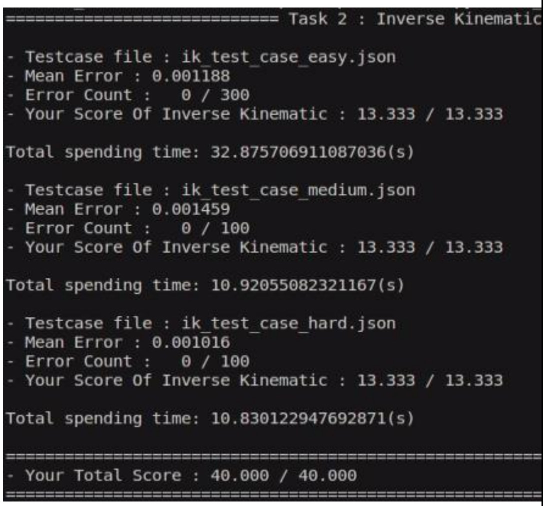
Total spending time: 36.64559984207153(s)

- Testcase file : ik_test_case_medium.json
- Mean Error : 0.000467
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

Total spending time: 12.253877639770508(s)

- Testcase file : ik_test_case_hard.json
- Mean Error : 0.000340
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

Total spending time: 12.38697624206543(s)

==
- Your Total Score : 40.000 / 40.000
==</td>
<td>
============================ Task 2 : Inverse Kinematic

- Testcase file : ik_test_case_easy.json
- Mean Error : 0.001188
- Error Count :   0 / 300
- Your Score Of Inverse Kinematic : 13.333 / 13.333

Total spending time: 32.875706911087036(s)

- Testcase file : ik_test_case_medium.json
- Mean Error : 0.001459
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

Total spending time: 10.92055082321167(s)

- Testcase file : ik_test_case_hard.json
- Mean Error : 0.001016
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

Total spending time: 10.830122947692871(s)

==
- Your Total Score : 40.000 / 40.000
==</td>
</tr>
<tr>
<td>Your_ik</td>
<td>Pybullet_ik</td>
</tr>
</table>

We can see that the pybullet_ik is faster than my implementation (Levenberg Marquardt method) on each task. And I don’t find how pybullet implement their IK…