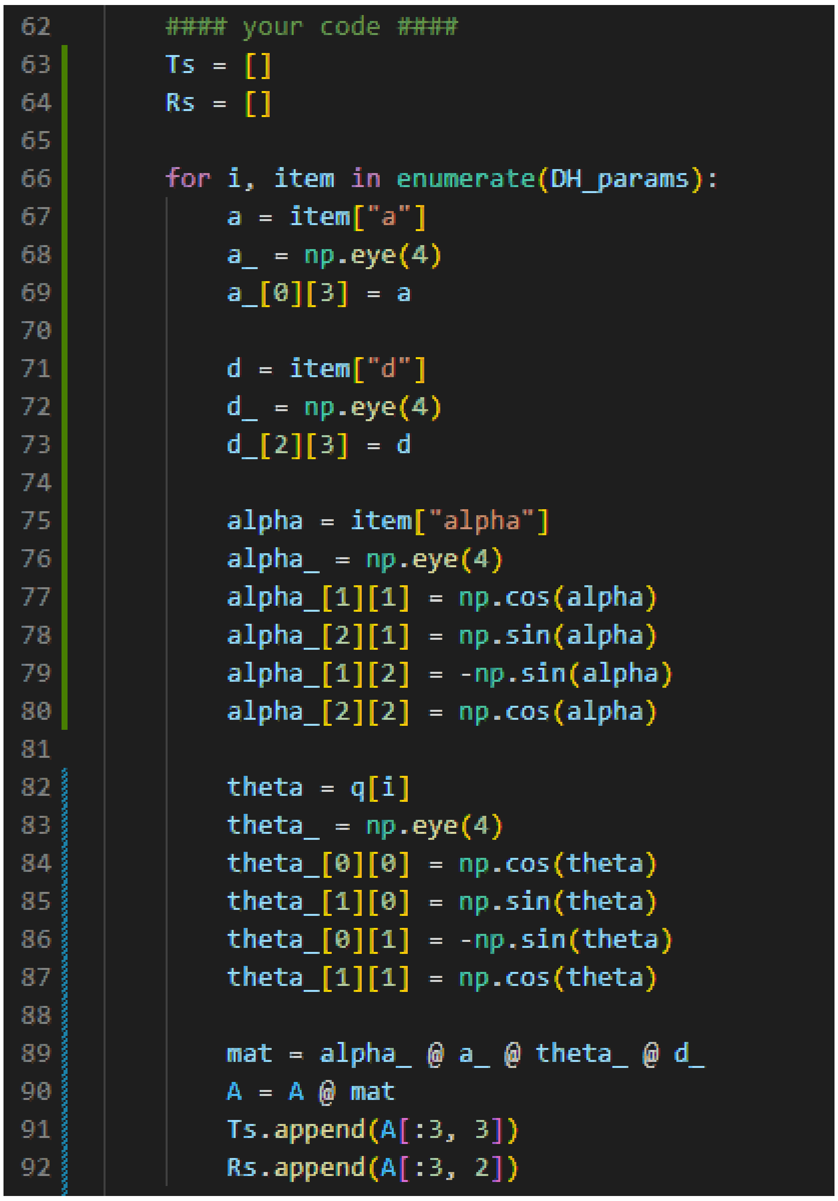
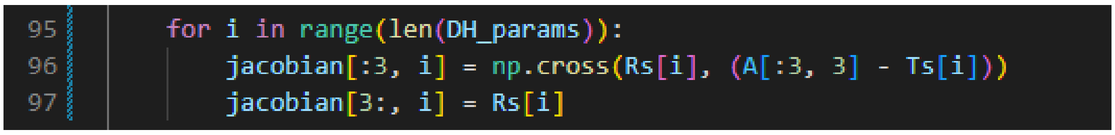
1. About task 1

1.1 Briefly explain how you implement your_fk() function (3%)



```python
    #### your code ####
    Ts = []
    Rs = []

    for i, item in enumerate(DH_params):
        a = item["a"]
        a_ = np.eye(4)
        a_[0][3] = a

        d = item["d"]
        d_ = np.eye(4)
        d_[2][3] = d

        alpha = item["alpha"]
        alpha_ = np.eye(4)
        alpha_[1][1] = np.cos(alpha)
        alpha_[2][1] = np.sin(alpha)
        alpha_[1][2] = -np.sin(alpha)
        alpha_[2][2] = np.cos(alpha)

        theta = q[i]
        theta_ = np.eye(4)
        theta_[0][0] = np.cos(theta)
        theta_[1][0] = np.sin(theta)
        theta_[0][1] = -np.sin(theta)
        theta_[1][1] = np.cos(theta)

        mat = alpha_ @ a_ @ theta_ @ d_
        A = A @ mat
        Ts.append(A[:3, 3])
        Rs.append(A[:3, 2])
```

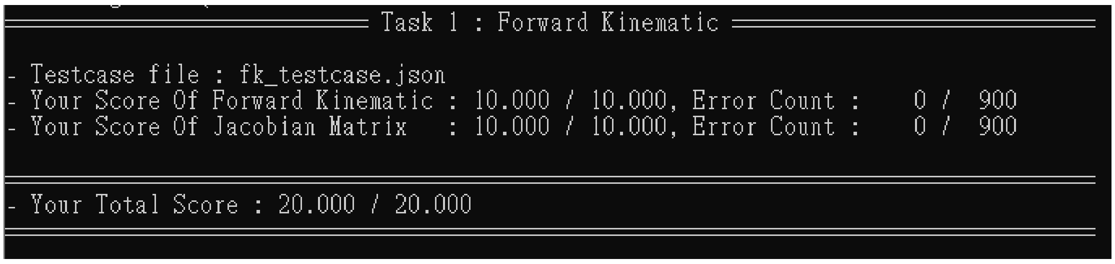
In the very beginning, I use 2 list to store the rotation and translation for calculating the Jacobian. Then, I iterate all the D-H parameter. In each iteration, I construct the 4 transformation matrix for the 4 parameters and multiplied them together to get the transform of this joint. After all, I multiplied A matrix with it to get the transformer from joint 0 to this joint. And save the rotation and translation information into the list.



```python
    for i in range(len(DH_params)):
        jacobian[:3, i] = np.cross(Rs[i], (A[:3, 3] - Ts[i]))
        jacobian[3:, i] = Rs[i]
```

For the Jacobian matrix, I follow the slides to construct it by using the data I saved in the previous iteration.

Result:



```
=================================== Task 1 : Forward Kinematic ===================================

- Testcase file : fk_testcase.json
- Your Score Of Forward Kinematic : 10.000 / 10.000, Error Count :    0 /  900
- Your Score Of Jacobian Matrix   : 10.000 / 10.000, Error Count :    0 /  900
==================================================================================================
- Your Total Score : 20.000 / 20.000
==================================================================================================
```

1.2 What is the difference between D-H convention and Craig's convention? (2%)

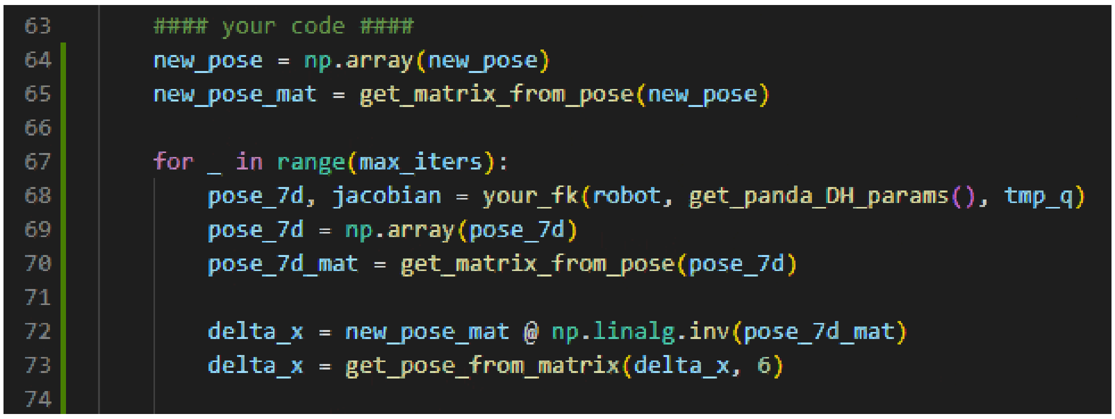
Craig's convention is the modified D-H convention. The coordinates of Oi is put on the axis-i, rather than axis-i+1

1.3 Complete the D-H table in your report following D-H convention

	d	α(rad)	a
1	d1	$\pi/2$	0
2	0	$-\pi/2$	0
3	d3	$\pi/2$	a3
4	0	$\pi/2$	a4
5	d5	$\pi/2$	0
6	0	$-\pi/2$	a6
7	d7	0	0

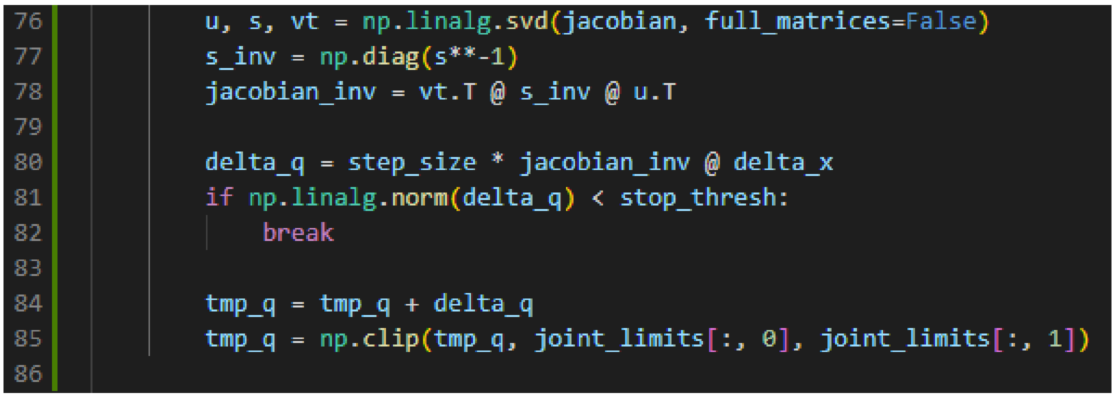
2. About task 2

2.1 Briefly explain how you implement your_ik() function (5%)



```python
#### your code ####
new_pose = np.array(new_pose)
new_pose_mat = get_matrix_from_pose(new_pose)

for _ in range(max_iters):
    pose_7d, jacobian = your_fk(robot, get_panda_DH_params(), tmp_q)
    pose_7d = np.array(pose_7d)
    pose_7d_mat = get_matrix_from_pose(pose_7d)

    delta_x = new_pose_mat @ np.linalg.inv(pose_7d_mat)
    delta_x = get_pose_from_matrix(delta_x, 6)

```

In the very beginning, I transform the 7d target pose into matrix form. Then, use for loop to run for at most max_iters iterations. In the iteration, I first calculate the pose and Jacobian with the current tmp_q. Then, transform the 7d current pose into matrix form, and calculate the transform from current pose to target pose by matrix multiplication. After that, we can get the 6d transform, delta_x.



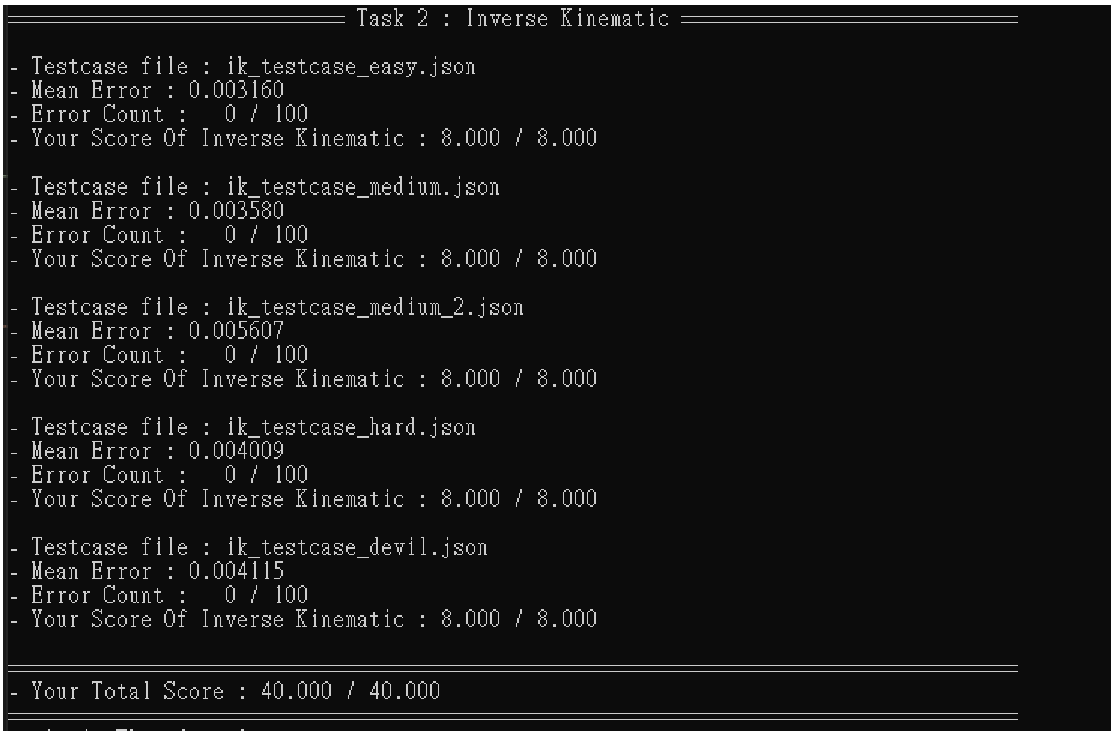
```python
    u, s, vt = np.linalg.svd(jacobian, full_matrices=False)
    s_inv = np.diag(s**-1)
    jacobian_inv = vt.T @ s_inv @ u.T

    delta_q = step_size * jacobian_inv @ delta_x
    if np.linalg.norm(delta_q) < stop_thresh:
        break

    tmp_q = tmp_q + delta_q
    tmp_q = np.clip(tmp_q, joint_limits[:, 0], joint_limits[:, 1])

```

Then, I need to calculate the pseudo-inverse of Jacobian. I use the SVD-decomposition to calculate it. It would be equal to $V\Sigma^{-1}U^T$. After that, calculate the delta_q and check if it is less than the threshold. And add it to the tmp_q. After all, clamp it to avoid exceed the joint limitation.

2.2 What problems do you encounter and how do you deal with them? (5%)

I face a big problem in the ik_testcase_hard. In the last 20 poses of that test case, my robot arm would twist in a strange angle. I found that the temp_q would exceed the joint limit in those situations. So, I add the np.cilp() function the let the temp_q not exceed the limit. In the end, it works.

Result:



```
======================= Task 2 : Inverse Kinematic =======================

- Testcase file : ik_testcase_easy.json
- Mean Error : 0.003160
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 8.000 / 8.000

- Testcase file : ik_testcase_medium.json
- Mean Error : 0.003580
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 8.000 / 8.000

- Testcase file : ik_testcase_medium_2.json
- Mean Error : 0.005607
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 8.000 / 8.000

- Testcase file : ik_testcase_hard.json
- Mean Error : 0.004009
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 8.000 / 8.000

- Testcase file : ik_testcase_devil.json
- Mean Error : 0.004115
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 8.000 / 8.000

===========================================================================
- Your Total Score : 40.000 / 40.000
===========================================================================
```

3. About manipulation.py

3.1 Briefly explain how get_src2dst_transform_from_kpts() function works for pose matching and how template_gripper_transform work for gripper pose estimation in manipulation.py. (5%)

get_src2dst_transform_from_kpts() is used to calculate the optimal transform from source points to destination points.

template_gripper_transform is the transform of robot arm when grasping a object in robot arm's coordinate. In line 685, it times with template2init. template2init is the transformation matrix of the object to grasp. Multiply them together would get the transformation matrix of grasping that object in world coordinate. For line766, gripper_key_trans is the object transform being hang in robot arm's coordinate. So, it would times the inverse of template_gripper_transform to get the object's world coordinate.

3.2 What is the minimum number of keypoints we need to ensure the program runs properly? Why?

3. Since it is PnP problem. PnP problem needs at least 3 points to solve the homography. However, 3 points are usually not enough to produce an accurate result. In my experiment, I usually need 4 or more points to produce a good result.

3.3 Briefly explain how to improve the matching accuracy (5% + 5% bonus)

Annotate more points.