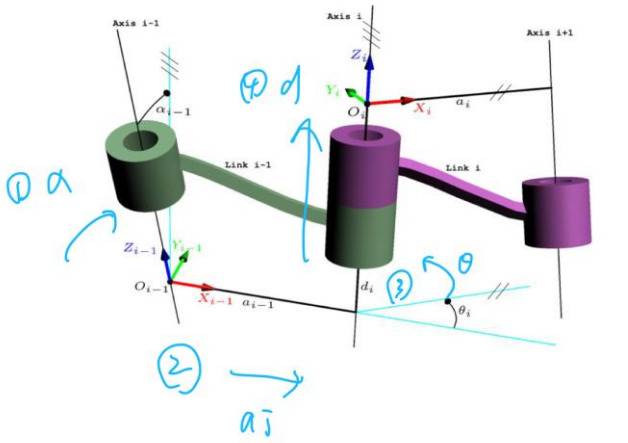
1. About task 1

1.1 What forward kinematics do is mapping from joint space to cartesian space. That is, given the angle of joints, we want to know where these joints (or the end effector) are in 3d space.

Using “Modified D-H Conventions”, we know that the axes of “joint i-1’s coordinate system” can match the axes of “joint i’s coordinate system” if we first rotate it (joint i-1’s coordinate system) around its x-axis by “alpha” degree, translate along its x-axis by “a” units, rotate around its z-axis by “theta” degree, and then translate along its z-axis by “d” units. (“alpha”, “a”, “theta” and “d” are D-H parameters.)



Therefore, the transformation matrix, with which a point which is originally in “joint i’s coordinate system” can be transformed into “joint i-1’s coordinate system”, is as the following.

$$T_{i-1}^{i} = T_{Rx}(\alpha_i) \times T_x(a_i) \times T_{Rz}(\theta_i) \times T_z(d_i)$$

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

In my “your_fk()”, I use the input, the angles of 7 joints, D-H parameters, to calculate the position (and orientation) of the end effector by using the above transformation matrix.

First of all, this part of codes has been provided in hw4 template. The robot is located at its initial position and is in the “base pose coordinate system” (the origin is located at the variable “base_pos”). The variable “A” here is a transformation matrix which can change the robot form "base_pose" coordinate system to "world" coordinate system (the origin is located at (0, 0,

0)). Jacobian is a matrix which records how much the position of the end effector will change when each joint changes a little bit (6-by-7 matrix, 6 -> 6d pose, 7 -> 7 joints).

```
46    # robot initial position
47    base_pos = robot._base_position            # tuple (x, y, z) = (-0.2, 0.13, 0.6)
48    base_pose = list(base_pos) + [0, 0, 0, 1]  # [0, 0, 0, 1] -> quaternion, list [x, y, z, 0, 0, 0, 1] = [-0.2, 0.13, 0.6, 0, 0, 0, 1],
49
50
51    assert len(DH_params) == 7 and len(q) == 7, f'Both DH_params and q shoul (parameter) DH_params: dict
52                                               f'but get len(DH_params) = {DH_params}, len(q) = {len(q)}'
53
54    # A: transformation matrix which can change the robot form "base_pose" coordinate to "world" coordinate
55    A = get_matrix_from_pose(base_pose) # a 4x4 matrix, type should be np.ndarray
56    jacobian = np.zeros((6, 7)) # a 6x7 matrix, type should be np.ndarray
```

To calculate the end effector position and Jacobian matrix, first of all I compute transformation matrix which change a point from “joint i’s coordinate system” into “joint i-1’s coordinate system” iteratively (by using the above formula). By multiplying these matrices together, we can have transformation matrices which can change a point from “joint i’s coordinate system” into “world coordinate system”. (I store them in List “Ts”).

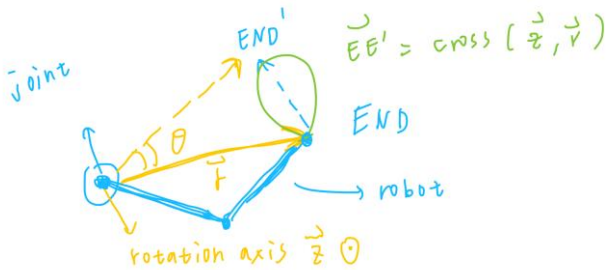
```
72    Ts = []
73    for i, theta in enumerate(q):     # compute transformation mation form joint 1
74
75        a = DH_params[i]['a']
76        d = DH_params[i]['d']
77        alpha = DH_params[i]['alpha']
78
79        Trx = np.array([[1, 0, 0, 0],
80                        [0, math.cos(alpha), -math.sin(alpha), 0],
81                        [0, math.sin(alpha), math.cos(alpha), 0],
82                        [0, 0, 0, 1]])
83
84        Tx = np.array([[1, 0, 0, a],
85                       [0, 1, 0, 0],
86                       [0, 0, 1, 0],
87                       [0, 0, 0, 1]])
88        Trz = np.array([[math.cos(theta), -math.sin(theta), 0, 0],
89                        [math.sin(theta), math.cos(theta), 0, 0],
90                        [0, 0, 1, 0],
91                        [0, 0, 0, 1]])
92        Tz = np.array([[1, 0, 0, 0],
93                       [0, 1, 0, 0],
94                       [0, 0, 1, d],
95                       [0, 0, 0, 1]])
96
97        T = Trx @ Tx @ Trz @ Tz
98        A = A @ T
99
100       Ts.append(A)
```

The end effector is the origin of joint 7. Its position in world coordinate system is the translation part of the i-to-world transformation matrix and its orientation is the rotation part of the i-to-world transformation matrix. Therefore, the variable “pose_7d” is what we want.

```
111   pose_7d = np.asarray(get_pose_from_matrix(A))   # shape = (7,), (pos_x, pos_y, pos_z, rot_x, rot_y, rot_z, rot_w)
112
```

As for the Jacobian matrix, what we want to know is how much the position and the orientation will change when a specific joint changes a little bit. The change of end effector position is the cross product of the rotation axis and the vector pointing from the joint to the end effector. The change of end effector orientation is the rotation axis. These vectors are expressed in world

coordinate system. The rotation axis is the third column of the rotation part of i-to-world transformation matrix and the joint position is the translation part of i-to-world transformation matrix. Simply using them to calculate the cross product for each joint, then we can get Jacobian matrix.




```
115     # Jacobian
116     for i in range(7):
117         T = Ts[i]
118         axis = T[0:3, 2]
119         r = pose_7d[0:3] - T[0:3, 3]    # end point - origin of joint i
120         jacobian[0:3, i] = cross(axis, r)
121         jacobian[3:6, i] = axis
122
```

1.2 The difference between D-H convention and Craig's convention:

(1) Coordinates of Oi is put on axis i+1 in D-H convention, while coordinates of Oi is put on axis i in Craig's convention.

(2) The order of the matrix multiplication is: theta -> d -> a -> alpha in in D-H convention, while the order of the matrix multiplication is: alpha -> a -> theta -> d in Craig's convention.

(3) The Xi axis is perpendicular to the Zi-1 axis in D-H convention, while the Zi axis is perpendicular to the Xi-1 axis in Craig's convention.

1.3 D-H table

i	d	α (rad)	a	θ_i (rad)
1	d_1 (0.333)	$-\frac{\pi}{2}$	0	θ_1
2	0	$\frac{\pi}{2}$	0	θ_2
3	d_3 (0.3160)	$-\frac{\pi}{2}$	a_3 (0.0825)	θ_3
4	0	$\frac{\pi}{2}$	$-a_4$ (-0.0825)	θ_4
5	d_5 (0.3840)	$-\pi/2$	0	θ_5
6	0	$-\pi/2$	a_6 (0.088)	θ_6
7	d_7 (0.107)	0	0	θ_7

2. About task 2

2.1 What inverse kinematics do is mapping from joint space to cartesian space. That is, given the position and the orientation of the end effector, we want to know what values of the joint angle can make the end effector there.

In my “your_ik()” function, first an initial set of joint angles is given (code line 52, 53) and I used my “your_fk()” function to calculate the end effector position and orientation when the robot followed these joint angles (code line 84). Then I updated the joint angles using the following two formula so that we can generate a set of joint angles which make the end effector position and orientation closer to the target position and orientation (code line 108).

$$\boldsymbol{q^{i+1} = q^{i} - \alpha_i \nabla \mathbf{L}(\mathbf{q^i})}$$

$$\nabla L(q) = \begin{bmatrix} \frac{\partial L}{\partial q_1} \\ \frac{\partial L}{\partial q_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial q_1}(f_1(q) - x_1^d) + \frac{\partial f_2}{\partial q_1}(f_2(q) - x_2^d) \\ \frac{\partial f_1}{\partial q_2}(f_1(q) - x_1^d) + \frac{\partial f_2}{\partial q_2}(f_2(q) - x_2^d) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \frac{\partial f_2}{\partial q_1} \\ \frac{\partial f_1}{\partial q_2} & \frac{\partial f_2}{\partial q_2} \end{bmatrix} \begin{bmatrix} f_1(q) - x_1^d \\ f_2(q) - x_2^d \end{bmatrix}$$

$$= \boldsymbol{J^T(q)\,(f(q) - x^d)}$$

This procedure (having a set of joint angles -> compute end effector position and orientation with this set of joint angles -> update joint angles) will be run iteratively, until the updated joint angles can make the end effector position and orientation close enough to our target position and orientation or until the maximum iteration is reached.

```
50      # get current joint angles and gripper pos, (gripper pos is fixed)
51      num_q = p.getNumJoints(robot.robot_id)
52      q_states = p.getJointStates(robot.robot_id, range(0, num_q))
53      tmp_q = np.asarray([x[0] for x in q_states[:7]]) # current joint angles 7d (You only need to modify this)
54                                                       # tmp_q.shape = (7, )
55      gripper_pos = robot.get_gripper_pos() # current gripper position 2d (Don't touch or modify this)
56
```

```
82      for _ in range(max_iters):
83          # q: theta1-theta7, list
84          tmp_pose_7d, jacobian = your_fk(robot, DH_params, tmp_q)
85          tmp_pose_6d = np.asarray(pose_7d_to_6d(tmp_pose_7d))
86
87          if(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) < stop_thresh):
88              break
89
90          if(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) > 0.1):
91              step = 0.06
92          elif(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) > 0.08):
93              step = 0.05
94          elif(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) > 0.05):
95              step = 0.04
96          elif(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) > 0.01):
97              step = 0.03
98          elif(np.linalg.norm(tmp_pose_7d - np.asarray(new_pose)) > 0.003):
99              step = 0.01
100         else:
101             step = 0.005
102
103         T_tmp = get_matrix_from_pose(tmp_pose_6d)
104         T_new = get_matrix_from_pose(new_pose)
105         T_delta = T_tmp @ scipy.linalg.inv(T_new)
106
107         # tmp_q  =  tmp_q - step * jacobian.transpose() @ (tmp_pose_6d - np.asarray(pose_7d_to_6d(new_pose)))
108         tmp_q  =  tmp_q - step * jacobian.transpose() @ (get_pose_from_matrix(T_delta, 6))
109
110         for i, is_greater in enumerate(tmp_q > joint_limits[:, 1]):
111             if(is_greater):
112
113                 tmp_q[i] = (joint_limits[i, 1] + joint_limits[i, 0])/2
114                 if(i-1>0):
115                     tmp_q[i-1] = (joint_limits[i-1, 1] + joint_limits[i-1, 0])/2
116                 if(i+1<7):
117                     tmp_q[i+1] = (joint_limits[i+1, 1] + joint_limits[i+1, 0])/2
118
119         for i, is_smaller in enumerate(tmp_q < joint_limits[:, 0]):
120             if(is_smaller):
121
122                 tmp_q[i] = (joint_limits[i, 1] + joint_limits[i, 0])/2
123                 if(i-1>0):
124                     tmp_q[i-1] = (joint_limits[i-1, 1] + joint_limits[i-1, 0])/2
125                 if(i+1<7):
126                     tmp_q[i+1] = (joint_limits[i+1, 1] + joint_limits[i+1, 0])/2
```

2.2 The major problem I encountered is that it's hard to pass the testcases.
At the beginning, I followed the above formula literally. I subtracted new pose from the current pose, and used it to computed the joint angles to be updated. However, this method has a serious limitation. When the orientations of the new pose and current pose are like "roll/pitch/yaw of new pose = -179 degrees and roll/pitch/yaw of current pose = 179 degrees", the actual differences of the orientation are 2 degrees. If we just simply subtract the new pose from the current one, it will result in a difference of 358 degrees. It made my IK hard to converge. Thus, according to TA's advice, instead of directly subtracting two pose, I used transformation matrix to compute DELTA_X (code line 103-105, 107). I transformed the two pose into two transformation matrix and multiplied the current transformation matrix by the inverse matrix of the new matrix. It generated the matrix which represented the difference of these two poses. Then I transformed the difference matrix into pose and used it to computed the joint angles to be updated.
Also, in order to pass the testcases, I also defined the stepsize dynamically. The more difference the two poses have, the larger the stepsize is (code line 90-101). Besides, when the joint exceeds its joint limit, I set this joint and its previous and next joint angle as the average of their joint limit (code line 90-101). I also

modified the max iteration to a larger number (in my case, 3000).
After doing all of these, finally I passed all of the available testcases.

3. About manipulation.py

3.1 What was done in “get_src2dst_transform_from_kpts()” function is to calculate the transformation matrix which can transform the source reference frame into destination reference frame so that the source key points can match the destination key points as well as possible. The way that it calculated the transformation matrix is similar to what we did in Hw1 (the way to calculate the best transformation matrix in local ICP). First, it computed the centroid of source points (SP) and destination points (DP). Second, it subtracted the centroid from SP and DP (the result are SP’ and DP’). Then it does singular value decomposition. What was gotten from singular value decomposition can be used to compute the best rotation matrix and the best rotation matrix can be used to compute the best transition matrix. At the end, the best rotation matrix and the best transition matrix were put together to form a transformation matrix.
The difference with what I did in Hw1 is that it used DP’ @ SP’.transpose to do singular value decomposition, instead of SP’ @ DP’.transpose. Thus, the best rotation matrix became U @ VT, instead of V @ UT. Actually, it results in the same result.

The variable “template_gripper_transform” records where the robot should grip the cup in template reference frame (the first picture we add annotated points). After it used the function “get_src2dst_transform_from_kpts()” to calculate the transformation matrix (template2init) from template reference frame into initial reference frame (code line 620), “template2init@ template_gripper_transform” became the transformation matrix which records where the robot should grip the cup in initial reference frame (code line 688, 724).

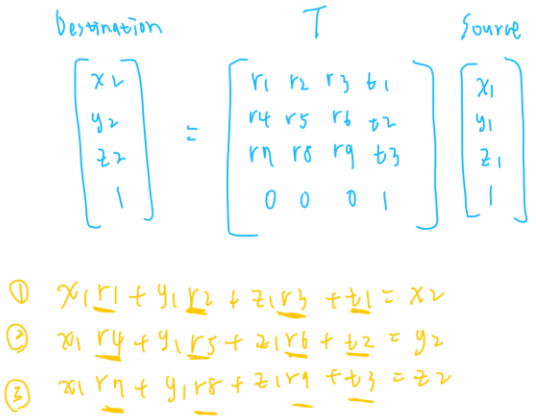
```
620     template2init = get_src2dst_transform_from_kpts(template_3d_kpts_homo, template_extrinsic, init_3d_kpts_homo, init_extrinsic)
```

```
687         # grasping pose
688         gripper_grasping_trans = template2init @ template_gripper_transform
689         gripper_grasping_pose = get_pose_from_matrix(gripper_grasping_trans)
690
```

```
723     # go to the grasping pose
724     robot_dense_action(robot, obj_id, gripper_grasping_pose, grasp=False, resolution=action_resolution)
725
```

3.2 I think the minimum number of key points we need is THREE, because the object is in a 3D world. As long as three points in different picture can match each other, then the only transformation matrix can be decided (these three points should not colinear or coplanar with each other).

Also, we can think of it as solving the transformation matrix from the source to the destination. We have three equations to solve and each equation has three unknown variables (we only need to solve the rotation part because the translation part can directly be solved when we know the rotation part). Thus, the minimum number of key points is THREE.



3.3 In function “get_src2dst_transform_from_kpts()”, the transformation matrix was only calculated once. Maybe one improvement can be “iteratively calculate the transformation matrix until the source points are close enough to the destination points” as what we did in Hw1.

3.4 Please find the attached video!