

Perception and Decision Making in Intelligent Systems
Homework 4

310551003 陳嘉慶

1. About Task 1(10%):

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

- (You can paste the screenshot of your code and explain it)

I write 2 function for getting A matrix and Jacobian matrix:

1. According to joint parameters (DH_params) and joint information(q), we can get our transform matrix between each joint from joint 1 to joint 7.
2. As Task 1 is forward kinematic, we use A matrix dot product with transform matrix from 1 to 7 and get the result of corresponding 7D pose of robot end-effector.

```
trans_mtx, Amtx_list = params_to_Amatrix(DH_params, q)
A = np.dot(A, trans_mtx)
jacobian = Amatrix_to_Jmatrix(trans_mtx, Amtx_list)
```

The transform matrix is given as below:

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

Where all highlighted parameters are given by robot parameters(dh_params), substitute the matrix one by one and we can have all transformed matrices.

```
def params_to_Amatrix(dh_params, theta_list):
    T_mtx_total = []
    final_mtx = np.identity(4)
    for i in range(len(theta_list)):
        alpha = dh_params[i]['alpha']
        a = dh_params[i]['a']
        d = dh_params[i]['d']
        theta = theta_list[i]
        trans_mtx = np.array([[np.cos(theta), -np.sin(theta), 0, a],
                               [np.sin(theta) * np.cos(alpha), np.cos(theta) * np.cos(alpha), -np.sin(alpha), -d *
np.sin(alpha)],
                               [np.sin(theta) * np.sin(alpha), np.cos(theta) * np.sin(alpha), np.cos(alpha), d *
np.cos(alpha)],
                               [0, 0, 0, 1]])
        final_mtx = np.dot(final_mtx, trans_mtx)
        T_mtx_total.append(final_mtx)
    return final_mtx, T_mtx_total
```

For the Jacobian matrix, every column of matrix (J as below) can be generated from below matrix which represent each joint:

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

Perception and Decision Making in Intelligent Systems

Homework 4

310551003 陳嘉慶

Where R_{i-1}^0 , d_{i-1}^0 and d_n^0 are come from transform matrix calculated from above part 1:

$$T_i^0 = \begin{bmatrix} R_i^0 & d_i^0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, combine all of J columns, we can have Jacobian matrix.

```
def Amatrix_to_Jmatrix(trans_mtx, Amtx_list):
    # initial parameter
    r_dot = np.array([[0, 0, 1]]).T
    Jmatrix = np.zeros((6, 7))
    d07 = np.array([trans_mtx[:3, 3]]).T

    for i in range(len(Amtx_list)):

        r0i = Amtx_list[i][:3, :3]
        d0i = np.array([Amtx_list[i][:3, 3]]).T
        r0i_dot = np.dot(r0i, r_dot)

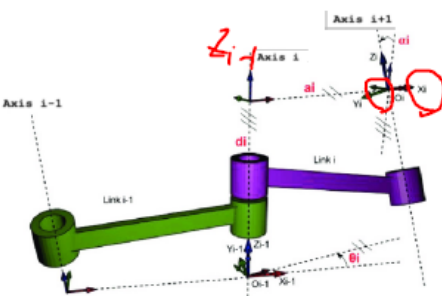
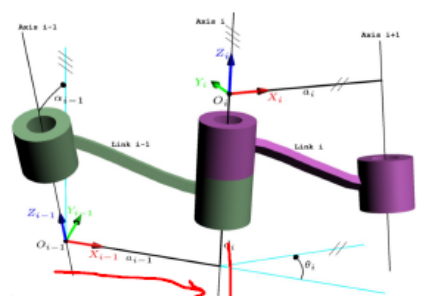
        d_diff = d07 - d0i
        t0i = np.cross(r0i_dot, d_diff, axis=0)

        Jmatrix[0, i], Jmatrix[1, i], Jmatrix[2, i] = t0i[0], t0i[1], t0i[2]
        Jmatrix[3, i], Jmatrix[4, i], Jmatrix[5, i] = r0i_dot[0], r0i_dot[1], r0i_dot[2]
        # print("Jmatrix: ", Jmatrix)

    return Jmatrix
```

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

The different between D-H convention and Craig's convention are shown as below:

Classic D-H Convention	Modified D-H Convention
 <p>Constraints</p> <ul style="list-style-type: none"> the X_i-axis is perpendicular to the Z_{i-1} the X_i-axis intersects Z_{i-1} the origin of joint i is at the intersection of X_i and Z_i 	 <p>Constraints</p> <ul style="list-style-type: none"> the Z_i-axis is perpendicular to the X_{i-1} the Z_i-axis intersects X_{i-1} the origin of joint i is at the intersection of X_i and Z_i

The most different between them are Classic D-H Convention will use Z_{i-1} to check on Z_i , but Modified D-H will use Z_i to check on Z_{i-1} . Both of convention will have their constraint, Classic D-H X_i -axis is perpendicular to the Z_{i-1} , and X_i -axis intersects Z_{i-1} , but modified one is reversed.

Perception and Decision Making in Intelligent Systems
Homework 4

310551003 陳嘉慶

1.3 Complete the D-H table in your report following D-H convention (5%)

i	d	$\alpha(\text{rad})$	a	$\theta_i(\text{rad})$
1	d1	$\pi/2$	0	θ_1
2	0	$-\pi/2$	0	θ_2
3	d3	$\pi/2$	a3	θ_3
4	0	$\pi/2$	-a4	θ_4
5	d5	$\pi/2$	0	θ_5
6	0	$\pi/2$	a6	θ_6
7	d7	0	0	θ_7

2. About Task 2 (15%)

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

- (You can paste the screenshot of your code and explain it)

Below is my main code written at your_ik:

1. Initial parameter, get dh_params from fk function, get new_pose matrix from get_matrix_from_pose provided by bullet_utils.py
2. Get current_pose, current_jacobian matrix by input tmp_q, also convert current_pose to matrix form.
3. Calculate transformation matrix between target_pose and current_pose, convert transformation matrix to 6d pose.
4. Evaluate error by norm of new_pose and current_pose to determine stop iteration.
5. According to error to adjust step_size.
6. Calculate delta_q by multiple inverse jacobian and delta_pose
7. Update tmp_q with step_size of delta_q for next loop
8. Check whether tmp_q exceed limit by check_limit function.

```
verbose = False
dh_params = get_panda_DH_params()
new_pose_mtx = get_matrix_from_pose(new_pose)
if verbose:
    progress = tqdm(total=max_iters)
for i in range(0, max_iters):
    current_pose, current_jacobian = your_fk(robot, dh_params, tmp_q)
    # inv_jacobian = current_jacobian.T
    inv_jacobian = np.linalg.pinv(current_jacobian) # best
    current_pose_mtx = get_matrix_from_pose(current_pose)
    delta_pose_mtx = np.dot(new_pose_mtx, np.linalg.inv(current_pose_mtx))
    delta_pose = get_pose_from_matrix(delta_pose_mtx, 6)
    err = np.linalg.norm(new_pose - current_pose, ord=2)
    if err > 1: # all test
        step_size = 0.1
    else: # all test
        step_size = 0.012 # 0.012
```

```

if verbose:
    progress.update(1)
    progress.set_description("error: {:.4f}".format(err))
if err < stop_thresh:
    break
delta_q = np.dot(inv_jacobian, delta_pose)
tmp_q = tmp_q + step_size * delta_q
tmp_q = check_limit(tmp_q, joint_limits)

```

check whether tmp_q exceed provided joint limit, if so, replace it with limit value.

```

def check_limit(q_value, limit):
    low_mask = q_value < limit[:, 0]
    high_mask = q_value > limit[:, 1]
    q_value = np.where(low_mask, limit[:, 0], q_value) # + 0.01
    q_value = np.where(high_mask, limit[:, 1], q_value) # - 0.01
    return q_value

```

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

I think the problem is the understanding of transformation between each joint, their coordinates and relationship. I found many resources with professor slides to understand them and practice it by examples. For implementation, I found some of test case need to have large step at some points, that's why I added dynamic step size according to error in order to move a larger step to reach converge.

2.3 Bonus! Do you also implement other IK methods instead of pseudo inverse method? How about the results? (5% bonus)

I tried transpose method instead of inverse jacobian matrix, as provided by HW4 resource, there are easy way to implement IK by just replace inverse jacobian matrix by Transpose jacobian matrix.

Jacobian transpose algorithm sets the step in the Cartesian space proportional to the distance to the goal, but Jacobian pseudoinverse accounts for the curvature of the space in addition.

The result of both Jacobian is shown as below:

<pre> - Testcase file : ik_testcase_easy.json - Mean Error : 0.001243 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_medium.json - Mean Error : 0.001489 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_medium_2.json - Mean Error : 0.001509 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_hard.json - Mean Error : 0.001843 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 ===== - Your Total Score : 40.000 / 40.000 ===== </pre>	<pre> - Testcase file : ik_testcase_easy.json - Mean Error : 0.001308 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_medium.json - Mean Error : 0.001465 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_medium_2.json - Mean Error : 0.001502 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 - Testcase file : ik_testcase_hard.json - Mean Error : 0.001905 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 10.000 / 10.000 ===== - Your Total Score : 40.000 / 40.000 ===== </pre>
Figure 2.1 Result of pseudoinverse Jacobian matrix	Figure 2.2 Result of pseudoinverse Jacobian matrix

310551003 陳嘉慶

Obviously, the result of transpose Jacobian matrix have larger mean error than pseudoinverse one, but actually both methods can still pass all the test cases. In addition, transpose method will need a larger step size compare to pseudoinverse.

3. About manipulation.py (25%)

This script uses the **your_ik()** function to control the robot and complete the hanging task. Besides, in the “**grasping**” sub-task, you need to annotate some key points to find the grasping pose via the **template pose matching method**, we have provided a basic implementation to do so using your annotated keypoints (see **get_src2dst_transform_from_kpts()** in **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%)

- (Hint: this function may be similar to some code in HW1)

From my observation, the **get_src2dst_transform_from_kpts()** is very similar to our own ICP implementation from HW1. It utilizes SVD method to re-center all the points first and produce matrix H. Finally, to find out transform matrix by rotation and translation matrix gather from SVD function.

For **template_gripper_transform**, it gets transform matrix of mug and convert it to 7D pose. Then, robot can reach corresponding pose by **your_ik** function we done at part 2.

3.2 What is the minimum number of keypoints we need to ensure the program runs properly? Why? (5%)

3 is possible solution. I test the robot, 3 key points is enough for robot to hang the mug on the hook(refer to below figure or .mp4). It is the less requirement for matching, also to identify the location of hook in order to reach it. However, it still has some error of matching, the program will run properly if the we have more points (let's said 4 or 5).



Perception and Decision Making in Intelligent Systems
Homework 4

310551003 陳嘉慶

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

- (You don't need to implement it. Of course, if you implement the **improved version** and compare the results with the original version, **you will get another 5 points!!!**)

Similar to HW1, we can divide to global ICP and final tune ICP for adjust the accuracy of matching accuracy. On the other hands, sample more points of object can also increase accuracy, but implementation efficiency will be lower.

3.4 Record the manipulation process in “manipulation.mp4” (5%)

Please find attachment.

Discussion:

In addition, I also try both inverse and transpose Jacobian matrix for devil test case, it passed successfully. And obviously pseudo inverse Jacobian matrix have less error than transpose one as expected, but still, both of them can pass devil test case.

<pre>===== Task 2 : Inverse Kinematic ===== - Testcase file : ik_testcase_devil.json - Mean Error : 0.001573 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 40.000 / 40.000 ===== - Your Total Score : 40.000 / 40.000 =====</pre>	<pre>- Testcase file : ik_testcase_devil.json - Mean Error : 0.001618 - Error Count : 0 / 100 - Your Score Of Inverse Kinematic : 40.000 / 40.000 ===== - Your Total Score : 40.000 / 40.000 =====</pre>
Figure 3.1 Result of pseudoinverse Jacobian matrix	Figure 3.2 Result of pseudoinverse Jacobian matrix