Homework 4 Report

Name: 謝元碩

Student ID: 311512015

Task 1.

順向運動學的實作過程會在 task3 中說明。實作結果的終端機截圖如下：

```
(hw4) leo@leo-ASUSPRO:~/freshman/Sense/Sense_decision_hw4/PDMS-HW4$ python3 fk.py
pybullet build time: May 20 2022 19:43:01
============================ 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
================================================================================
```

Task 2.

逆向運動學的實作過程會在 task3 中說明。實作結果的終端機截圖如下：

```
- Testcase file : ik_testcase_hard.json
- Mean Error : 0.001836
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 10.000 / 10.000

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

Task 3. Questions

1. About task 1

```
A_ori = A

'''fk'''
for i in range(len(q)):
    a = DH_params[i]['a']
    d = DH_params[i]['d']
    alpha = DH_params[i]['alpha']
    theta = q[i]
    T = 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]])
    A = np.dot(A, T)

'''jacobian'''
P_eff = np.array(A[:3, 3]) # end effector position
for i in range(len(q)):
    a = DH_params[i]['a']
    d = DH_params[i]['d']
    alpha = DH_params[i]['alpha']
    theta = q[i]
    T = 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]])

    A_ori = np.dot(A_ori, T)
    Z = np.dot(A_ori[:3, :3], np.array([[0],[0],[1]])).flatten()

    # angular velocity
    jacobian[3, i] = Z[0]
    jacobian[4, i] = Z[1]
    jacobian[5, i] = Z[2]

    # linear velocity
    p_i = np.array(A_ori[:3, 3])
    p_i2e = P_eff - p_i
    J_linear = cross(Z, p_i2e)
    jacobian[0, i] = J_linear[0]
    jacobian[1, i] = J_linear[1]
    jacobian[2, i] = J_linear[2]
```

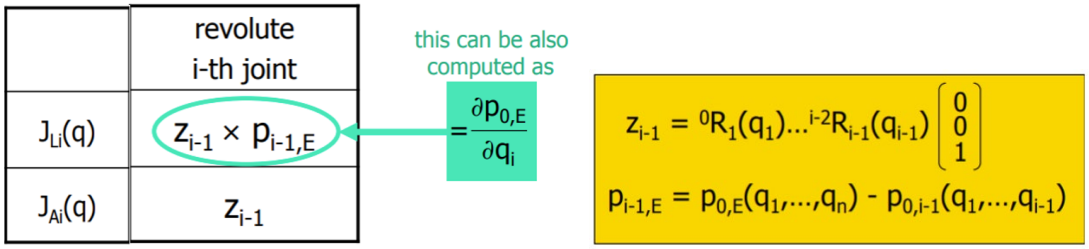
1.1

首先將作業提供的 DH model，以及每個時間點讀取到的 theta 值(q)，代入以下 modified DH convention 的轉移矩陣：

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

接著將原本的 A(base to world transformation matrix)，用迴圈不斷以 joint1~7 的轉移矩陣連乘，最後即可算出末端點的 pose。

Jacobian 的部分，先記錄前面算出的末端點座標，接著用幾何解的方式，同樣以迴圈分別計算出每個 joint 的線速度及角速度，使用幾何解公式如下：



在求出每個軸的線速度和角速度後，儲存到 jacobian 中，最後即可取得機械臂的 jacobian。

1.2

Classic D-H convention 的 Oi 定義在第 i+1 軸，其 a 和 alpha 值定義在第 i 到第 i+1 軸間；Craig's convention 的 Oi 定義在第 i 軸，其 a 和 alpha 值定義在第 i-1 到第 i 軸間。兩者所定義的 DH model 參數位置不同，其轉移矩陣的參數所帶入的 row 和連乘順序也都將不同。

1.3

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

2. About task 2

```python
dh_params = get_panda_DH_params()

iters = 0
alpha = 0.05
pre_Norm = 10
while(iters <= max_iters):

    pose, jacobian = your_fk(robot, dh_params, tmp_q)
    delta_T = get_matrix_from_pose(new_pose) @ linalg.inv(get_matrix_from_pose(pose))
    delta_x = get_pose_from_matrix(delta_T, 6)

    # update tmp_q with pseudo-inverse method
    j_t = np.transpose(jacobian)
    delta_q = alpha*j_t @ linalg.inv(jacobian@j_t) @ delta_x
    tmp_q = tmp_q + delta_q

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

    # check if it is out of threshold
    Norm = linalg.norm(delta_x)
    if(Norm < stop_thresh):
        break

    # check if the manipulator runs well
    if(Norm > pre_Norm):
        print("Diverge!!")
    pre_Norm = Norm
    iters+=1

if(iters > max_iters):
    print("Out of max iteration. Norm: ", Norm)
else:
    print("Succeed.")
```

2.1

這個部分的逆向運動學解法，主要是透過迭代不斷將 q 逼近理想 pose，而這裡我們使用順向運動學計算出的 jacobian 去反推 delta q。

在每一次迭代中，先以當下 q 計算出當下末端點的座標和 jacobian，接著透過理想點和當下點，以轉移矩陣的型態找出兩者之間的轉移矩陣，此步驟是為了求出兩點間的 delta x(逼近用)。

接著則使用 pseudo-inverse method，以類似反矩陣的方式反推 theta 的微量變化，公式如下：

$$\Delta\boldsymbol{\theta} = \alpha J^T(\boldsymbol{\theta})\left(J(\boldsymbol{\theta})J^T(\boldsymbol{\theta})\right)^{-1}\Delta\mathbf{x} = J^{\#}\Delta\mathbf{x}$$

此 delta theta 數值即為每次迭代用來更新原本 theta 的角度。此外，為避免關節超出活動範圍，我也利用程式提供的上下限，限制住機械臂關節角度。

理論上，若能夠順利逼近理想 pose，delta x 會有收斂的現象，因此若 delta x 順利收斂至閥值(stop_threshold)以內，代表逆向運動學計算成功，將結束此次迭代；反之若未收斂，代表已進入 singular 狀態，機械臂將永遠無法進入理想 pose。

2.2

一開始我認為使用上下限去對每個關節的角度設限應該對逆向運動學的計算不會有太大影響，所以並未使用上下限去對角度做限制。結果角度發散狀況有點嚴重，導致分數滿低的。後來加上限制後，收斂狀況相對正常許多。

此外，由於題目中的 max iteration 和 stop threshold 是給定的，pseudo-inverse method 必須在受限的參數中得最好的收斂，因此我嘗試調整 alpha 讓收斂變好。起初我將值都設在大概 0.001~0.01 間，發現總分僅有 30~39 左右，且在 hard 任務不斷顯示 delta x 發散了，效果不太好。後來經測試後使用 0.01~0.1 間的參數，讓逼近更快一些，收斂狀況改善，總分順利到達 40。

3. About manipulation.py

3.1

(1) get_src2dst_transform_from_kpts 主要針對不同座標系的 3D 座標進行座標轉換，與 HW1 的座標轉換類似。由於 pose matching 需透過不同視角的相機實現，此函式透過事前從不同視角選取的數個 key point，尋找兩個不同相機 3D 座標間之轉移矩陣，並利用此轉移矩陣，從其中一個相機的 3D 座標推算其在另一個相機的 3D 座標位置，接著扣掉另一個相機的 3D 座標(ground truth)，觀察其 matching error。函式中則是使用奇異值分解算法，以兩相機的 2D 座標做輸入，計算出 3D 座標系間的轉移矩陣。

(2) template_gripper_transform 代表 object frame 到 gripper frame 的轉移矩陣。取得此矩陣是為了取得其反矩陣：gripper frame 到 object frame 的轉移矩陣，利用這個矩陣才能夠求出物件在夾爪 frame 中的 pose 座標，進而得出物件在世界座標系的座標，以用在 robot_dense_action 函式中進行機械手臂夾取。

3.2

minimum number of keypoints：3。由於 matching 是作用在一個三維空間，因此至少需要 3 個座標點，才能實現轉移矩陣函式中的計算。

3.3

目前想到的改善方法，首先是從 keypoint 的部分下手，這裡簡單來說，就是將不同視角的 keypoint 選擇的更加精確，並且將點平均散佈在物件點雲上，如此可被動減少程式計算的 match 誤差。

接著，如 3.1 所說，我們透過 keypoint 產生不同視角 3D 座標系之間的轉移矩陣，並從中計算 match 誤差，因此只要改善或修正此轉移矩陣，就能夠讓不同座標系的座標趨近重合。這個部分，我認為可以嘗試修改 get_src2dst_transform_from_kpts 函式中的演算法，如可以改善 SVD 的算法或使用他種算法，尋找更適合的轉移矩陣。

3.4 影片檔已附在資料夾中