1. About task 1 (15%)

1.1 Briefly explain how you implement your_fk() function (3%) (You can paste the screenshot of your code and explain it)

Initialize Base Position:

- Set up the transformation matrix A based on the initial base_pos, which initializes the robot arm's base position. This matrix will be used to iteratively apply each joint transformation.
- Initialize a 6x6 Jacobian matrix jacobian to later compute the linear and angular velocity mappings.

```
# Initialize the transformation matrix of the base position
base_pose = list(base_pos) + [0, 0, 0]  # Initial base position with zero rotation
A = get_matrix_from_pose(base_pose)  # 4x4 transformation matrix
jacobian = np.zeros((6, 6))  # 6x6 Jacobian matrix
```

Verify Input Length:

- Ensure that the DH parameters and joint angles q each contain 6 values, matching the robot arm's configuration.

```
# Ensure that both DH_params and q contain 6 values
assert len(DH_params) == 6 and len(q) == 6, f'Both DH_params and q should contain 6 values,\n' \
                                            f'but received len(DH_params) = {len(DH_params)}, len(q) = {len(q)}'
```

Initialize Position and Orientation Storage:

- Store the base position and z-axis orientation, which will serve as the starting point for calculating the Jacobian matrix later.

```
# Store each joint's position and orientation (z-axis direction)
positions = [A[:3, 3]]  # Store the base position
orientations = [A[:3, 2]]  # Store the base z-axis direction
```

Define DH Transformation Matrix Generator:

- Use the dh_transform() function to generate the transformation matrix for each joint. This function calculates the transformation matrix based on each joint's DH parameters, linking each joint's position and orientation.

```python
# Define a function to generate the DH transformation matrix
def dh_transform(a, d, alpha, theta):
    """
    Generate the transformation matrix for a single joint based on DH parameters.
    """
    return 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]
    ])
```

Apply Each Joint's DH Transformation Matrix Iteratively:

- For each joint, generate the transformation matrix T using its DH parameters and joint angle q[i], and update the cumulative matrix A by multiplying it with T.
- Simultaneously, store the position and z-axis orientation of each joint, which will be used to compute the Jacobian matrix later.

```python
# Iteratively apply the DH transformation matrix for each joint
for i, params in enumerate(DH_params):
    # Generate the transformation matrix using the current joint angle q[i] and corresponding DH parameters
    T = dh_transform(params['a'], params['d'], params['alpha'], q[i])
    A = A @ T  # Update the cumulative transformation matrix

    # Store the current joint's position and z-axis direction for Jacobian computation
    positions.append(A[:3, 3])
    orientations.append(A[:3, 2])
```

Calculate the Jacobian Matrix:

- For calculating the Jacobian matrix, retrieve the end-effector position end_effector_pos.
- Iterate to calculate each column of the Jacobian matrix:
  - The linear velocity part is obtained by taking the cross product of the joint's z-axis vector and the difference between the end-effector and joint positions.
  - The angular velocity part is simply the joint's z-axis orientation.

```python
# Calculate the Jacobian matrix
end_effector_pos = positions[-1]  # Position of the end effector
for i in range(6):
    # Linear velocity component: cross product of z-axis vector with the position difference to the end effector
    jacobian[:3, i] = np.cross(orientations[i], (end_effector_pos - positions[i]))
    # Angular velocity component: directly take the z-axis direction
    jacobian[3:, i] = orientations[i]
```

Orientation Adjustment:

- Apply an adjustment matrix adjustment to ensure the final end-effector pose aligns with a specific orientation requirement, and apply it to the final rotation matrix part.

```
# Adjustment - do not modify
adjustment = np.asarray([[ 0, -1,  0],
                         [ 0,  0,  0],
                         [ 0,  0, -1]])
A[:3, :3] = A[:3, :3] @ adjustment
```

Return Final Pose and Jacobian Matrix:

- Extract the 7-dimensional pose (position and orientation) from the final transformation matrix A, and return it along with the computed Jacobian matrix.

```
pose_7d = np.asarray(get_pose_from_matrix(A, 7))

return pose_7d, jacobian
```

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

The **Denavit-Hartenberg (D-H) convention** and **Craig's Modified D-H Convention** both define transformations between coordinate frames of robotic joints, but they differ in how they assign these frames and the resulting transformation matrices.

1. **Frame Assignment**:
    - **D-H Convention**: Assigns the frame of each joint along the previous joint's z-axis. The transformations focus on aligning each joint's frame using four parameters: θ, d, a, and α.
    - **Modified D-H Convention**: Places the coordinate frame of each link along its own z-axis instead of the previous joint's z-axis. This leads to slightly different parameter definitions and can simplify certain link-to-link transformations, particularly for common robotic configurations.

2. **Transformation Matrix**:

- In the **D-H Convention**, the transformation from one frame to the next always follows the pattern: rotate around z_{i-1}, translate along z_{i-1}, then translate along x_i, and finally rotate around x_i.
- In the **Modified D-H Convention**, this order changes to: rotate around z_i, translate along z_i, translate along x_i, and finally rotate around x_i. This difference in sequence can make Craig's convention preferable for certain complex manipulations and robot structures.

In summary, while both conventions achieve forward kinematics, Craig's Modified D-H convention provides greater flexibility for frame assignment, which can simplify the math in some robotic configurations.

1.3 Complete the D-H table in your report following D-H convention (10%)

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

D-H table example format.

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

- Testcase file : fk_test_case_easy.json
- Your Score Of Forward Kinematic : 3.333 / 3.333, Error Count :    0 /  300
- Your Score Of Jacobian Matrix   : 3.333 / 3.333, Error Count :    0 /  300

- Testcase file : fk_test_case_medium.json
- Your Score Of Forward Kinematic : 3.333 / 3.333, Error Count :    0 /  100
- Your Score Of Jacobian Matrix   : 3.333 / 3.333, Error Count :    0 /  100

- Testcase file : fk_test_case_hard.json
- Your Score Of Forward Kinematic : 3.333 / 3.333, Error Count :    0 /  100
- Your Score Of Jacobian Matrix   : 3.333 / 3.333, Error Count :    0 /  100

====================================================================================
- Your Total Score : 20.000 / 20.000
====================================================================================
```

Result

2. About task 2 (10% + 5% bonus)

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

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

1. **Setting Joint Limits:**

   The program first defines the joint angle limits for each joint and stores them in the joint_limits array to ensure that angles stay within these bounds during updates.

```
joint_limits = np.asarray([
    [-3*np.pi/2, -np.pi/2],  # joint1
    [-2.3562, -1],           # joint2
    [-17, 17],               # joint3
    [-17, 17],               # joint4
    [-17, 17],               # joint5
    [-17, 17]                # joint6
])
```

2. **Initializing Joint Angles:**

   The initial joint angles of the robot arm are obtained using p.getJointStates, and the first six joint angles are extracted and stored in tmp_q as the starting angles.

```
# Get the initial joint angles of the robotic arm
num_q = p.getNumJoints(robot_id)
q_states = p.getJointStates(robot_id, range(0, num_q))
tmp_q = np.asarray([x[0] for x in q_states][2:8])  # Initial angles for the 6 joints
```

3. **Retrieving DH Parameters:**

   The DH parameters of the robot are retrieved using the get_ur5_DH_params() function, which will be used in the forward kinematics calculations.

```
# Retrieve DH parameters
DH_params = get_ur5_DH_params()
```

Setting Step Rate:

The step_rate is set to control the increment in angle updates. This helps adjust the rate of change, preventing overly large updates that may cause instability.

```
# Set step rate parameter to control update increments
step_rate = 0.1
```

Iteratively Updating Joint Angles:

The main iterative loop begins here. During each iteration, the program calculates the current pose of the end-effector and compares it with the target pose to reduce the difference.

- Calculate Current Pose and Jacobian Matrix: Using the your_fk function, the program computes the current end-effector pose current_pose and the Jacobian matrix J by passing in the DH parameters, current joint angles tmp_q, and base position base_pos.
- Calculate Position and Rotation Differences: The differences between the current and target positions (delta_pos) and orientation (delta_rot_vec) are computed, which together form delta_x.

```
for i in range(max_iters):
    # Use your_fk to obtain the current end-effector pose and Jacobian matrix
    current_pose, J = your_fk(DH_params, tmp_q, base_pos)  # Pass DH_params, tmp_q, and base_pos

    delta_pos = new_pose[:3] - current_pose[:3]
    delta_rot = R.from_quat(current_pose[3:]).inv() * R.from_quat(new_pose[3:])
    delta_rot_vec = delta_rot.as_rotvec()

    delta_x = np.hstack((delta_pos, delta_rot_vec))
```

Check if Target Pose is Achieved:

If the difference in position and orientation (delta_x) is below the threshold stop_thresh, the iteration stops, and the current joint angles tmp_q are returned.

```
# Check if the change in position and orientation is below the threshold
if np.linalg.norm(delta_x) < stop_thresh:
    return list(tmp_q)
```

Calculate and Update Joint Angles:

Using the pseudo-inverse of the Jacobian matrix, the joint angle increment delta_q is calculated and applied to the current

```
# Calculate joint angle update and apply it
delta_q = step_rate * (pinv(J).dot(delta_x))
tmp_q += delta_q
```

Limit Joint Angles:

After each update, the joint angles are clamped to stay within the bounds defined in joint_limits to ensure they remain within the robot's allowable range of motion.

```
# Clamp joint angles within their limits
for j in range(len(tmp_q)):
    tmp_q[j] = np.clip(tmp_q[j], joint_limits[j][0], joint_limits[j][1])
```

Return Final Joint Angles:

If the target pose is reached within the maximum number of iterations, the list of final joint angles tmp_q is returned.

```
return list(tmp_q)
```

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

Precision in Pose Control
In the calculation process, it is essential to ensure that the robot's end-effector accurately reaches the target position and orientation, as even a small error can lead to a deviation in the final outcome. To maintain precision, I set a stop_thresh threshold. When the end-effector's deviation falls below this threshold, the iteration stops, preventing endless calculations and ensuring sufficient accuracy in the result.

Iteration Speed and Efficiency
This code uses a loop to iteratively adjust the joint angles, gradually bringing the end-effector closer to the target pose. However, if the iteration speed is too slow, it could impact the program's efficiency. To address this, I introduced the step_rate parameter to control the increment of each update, balancing accuracy and efficiency.

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

In addition to using the pseudo-inverse method for intuitive inverse kinematics, I also researched deep learning-based IK solutions. After training, these methods can quickly predict joint configurations to adapt to different target positions and orientations. However, at this stage, I have not yet been able to successfully implement the deep learning-based IK method.

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

- Testcase file : ik_test_case_easy.json
- Mean Error : 0.001733
- Error Count :   0 / 300
- Your Score Of Inverse Kinematic : 13.333 / 13.333

- Testcase file : ik_test_case_medium.json
- Mean Error : 0.001371
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

- Testcase file : ik_test_case_hard.json
- Mean Error : 0.001133
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

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

result

3.About task 3 (5%)

This part uses the your_ik() function to control the robot and complete the block insertion task.

```
Test: 1/10
WARNING:tensorflow:From /home/morris/pdm-f24/hw3/ravens/ravens/agents/transporter.py:167: set_learning_phase (from tensorflow.python.keras.backend) is deprecated and will be removed after 2020-10-11.
Instructions for updating:
Simply pass a True/False value to the `training` argument of the `__call__` method of your layer or model.
W1109 15:47:43.973576 139827093787008 deprecation.py:323] From /home/morris/pdm-f24/hw3/ravens/ravens/agents/transporter.py:167: set_learning_phase (from tensorflow.python.keras.backend) is deprecated
d will be removed after 2020-10-11.
Instructions for updating:
Simply pass a True/False value to the `training` argument of the `__call__` method of your layer or model.
2024-11-09 15:47:44.595401: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 74317824 exceeds 10% of free system memory.
2024-11-09 15:47:44.626434: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 74317824 exceeds 10% of free system memory.
2024-11-09 15:47:44.653078: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 37748736 exceeds 10% of free system memory.
2024-11-09 15:47:44.656948: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 37748736 exceeds 10% of free system memory.
2024-11-09 15:47:44.666475: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 37748736 exceeds 10% of free system memory.
Total Reward: 1.0 Done: True
Test: 2/10
Total Reward: 1.0 Done: True
Test: 3/10
Total Reward: 1.0 Done: True
Test: 4/10
Total Reward: 1.0 Done: True
Test: 5/10
Total Reward: 1.0 Done: True
Test: 6/10
Total Reward: 1.0 Done: True
Test: 7/10
Total Reward: 1.0 Done: True
Test: 8/10
Total Reward: 1.0 Done: True
Test: 9/10
Total Reward: 1.0 Done: True
Test: 10/10
Total Reward: 1.0 Done: True
==================================================================================
- Your Total Score : 10.000 / 10.000
==================================================================================
```

In the test results, for each test (from Test: 1/10 to Test: 10/10), the model achieved a Total Reward of 1.0 and showed Done: True, indicating that the model successfully completed and met the objective in all 10 tests.

The final total score is 10.000 / 10.000, representing a perfect performance by the model in this task.

3.1 Compare your results between your_ik function and pybullet_ik

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

- Testcase file : ik_test_case_easy.json
- Mean Error : 0.001733
- Error Count :   0 / 300
- Your Score Of Inverse Kinematic : 13.333 / 13.333

- Testcase file : ik_test_case_medium.json
- Mean Error : 0.001371
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

- Testcase file : ik_test_case_hard.json
- Mean Error : 0.001133
- Error Count :   0 / 100
- Your Score Of Inverse Kinematic : 13.333 / 13.333

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

your_ik

```
============================ 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

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

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

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

pybullet_ik

When comparing the results of your_ik and pybullet_ik, we can see that both achieved a perfect total score of 40.000, with no errors (Error Count of 0 for each test case). The precision is also very close for both methods, but there are slight differences in the mean error across some test cases. Here's a detailed comparison:

1. **ik_test_case_easy.json**
   - your_ik: Mean Error = 0.001733

    - pybullet_ik: Mean Error = 0.001188
    - Result: pybullet_ik shows a slightly smaller error than your_ik.
2. **ik_test_case_medium.json**
    - your_ik: Mean Error = 0.001371
    - pybullet_ik: Mean Error = 0.001459
    - Result: In this case, your_ik has a slightly smaller error than pybullet_ik.
3. **ik_test_case_hard.json**
    - your_ik: Mean Error = 0.001133
    - pybullet_ik: Mean Error = 0.001016
    - Result: pybullet_ik shows a slightly smaller error than your_ik.

Summary

- Both methods are very close in precision and achieved a perfect score.
- pybullet_ik has slightly lower mean errors in the easy and hard test cases, while your_ik performs slightly better in the medium difficulty case.