Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Using the DH parameter table you derived earlier, create individual transformation matrices about each joint. In addition, also generate a generalized homogeneous transform between base_link and gripper_link using only end-effector(gripper) pose.

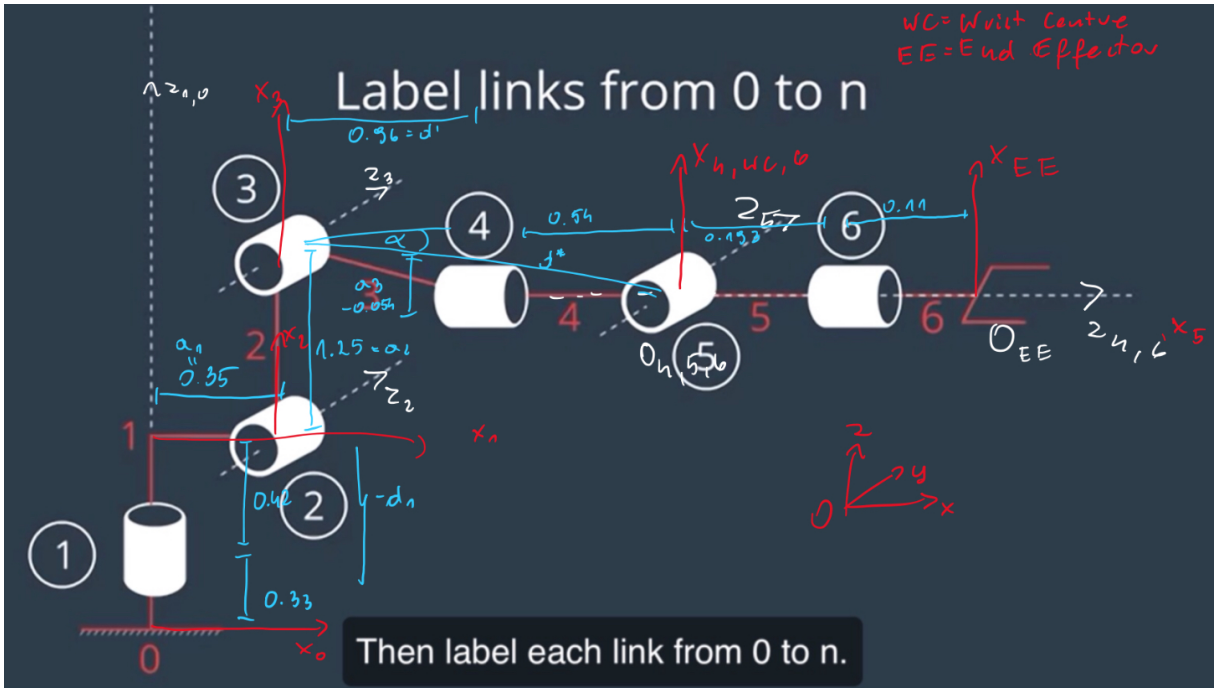

Figure 1: Link and origin frames' assignments

Links	alpha(i-1)	a(i-1)	d(i-1)	theta(i)
T_1^0	0	0	0.75	q1
T_2^1	- pi/2	0.35	0	-pi/2 + q2
T_3^2	0	1.25	0	q3
T_4^3	- pi/2	-0.054	1.5	q4
T_5^4	pi/2	0	0	q5

Links	alpha(i-1)	a(i-1)	d(i-1)	theta(i)
T_6^5	-pi/2	0	0	q6
T_{EE}^6	0	0	0.303	0

$$T_1^0 = \begin{bmatrix} \cos(q_1) & -\sin(q_1) & 0 & 0 \\ \sin(q_1) & \cos(q_1) & 0 & 0 \\ 0 & 0 & 1 & 0.75 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_2^1 = \begin{bmatrix} \sin(q_2) & \cos(q_2) & 0 & 0.35 \\ 0 & 0 & 1 & 0 \\ \cos(q_2) & -\sin(q_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^2 = \begin{bmatrix} \cos(q_3) & -\sin(q_3) & 0 & 1.25 \\ \sin(q_3) & \cos(q_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_4^3 = \begin{bmatrix} \cos(q_4) & -\sin(q_4) & 0 & -0.054 \\ 0 & 0 & 1 & 1.5 \\ -\sin(q_4) & -\cos(q_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_5^4 = \begin{bmatrix} \cos(q_5) & -\sin(q_5) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(q_5) & \cos(q_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_6^5 = \begin{bmatrix} \cos(q_6) & -\sin(q_6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(q_6) & -\cos(q_6) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{EE}^6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.303 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The homogeneous transform matrix from base link can be constructed from the input data:

$$R_T = R_z \cdot R_y \cdot R_x \cdot R_{gripper} = \begin{bmatrix} \sin(r_x)\sin(r_z) + \sin(r_y)\cos(r_x)\cos(r_z) & -\sin(r_x)\sin(r_y)\cos(r_z) + \sin(r_z)\cos(r_x) & \cos(r_y)\cos(r_z) \\ -\sin(r_x)\cos(r_z) + \sin(r_y)\sin(r_z)\cos(r_x) & -\sin(r_x)\sin(r_y)\sin(r_z) - \cos(r_x)\cos(r_z) & \sin(r_z)\cos(r_y) \\ \cos(r_x)\cos(r_y) & -\sin(r_x)\cos(r_y) & -\sin(r_y) \end{bmatrix}$$

$$T = \left[\begin{array}{ccc:c} & R_T & & \begin{matrix} P_x \\ P_y \\ P_z \end{matrix} \\ \hdashline 0 & 0 & 0 & 1 \end{array}\right]$$

Figure 2: Homogenous Transform Matrix

3. Decouple Inverse Kinematics problem into Inverse Position Kinematics and inverse Orientation Kinematics; doing so derive the equations to calculate all individual joint angles.

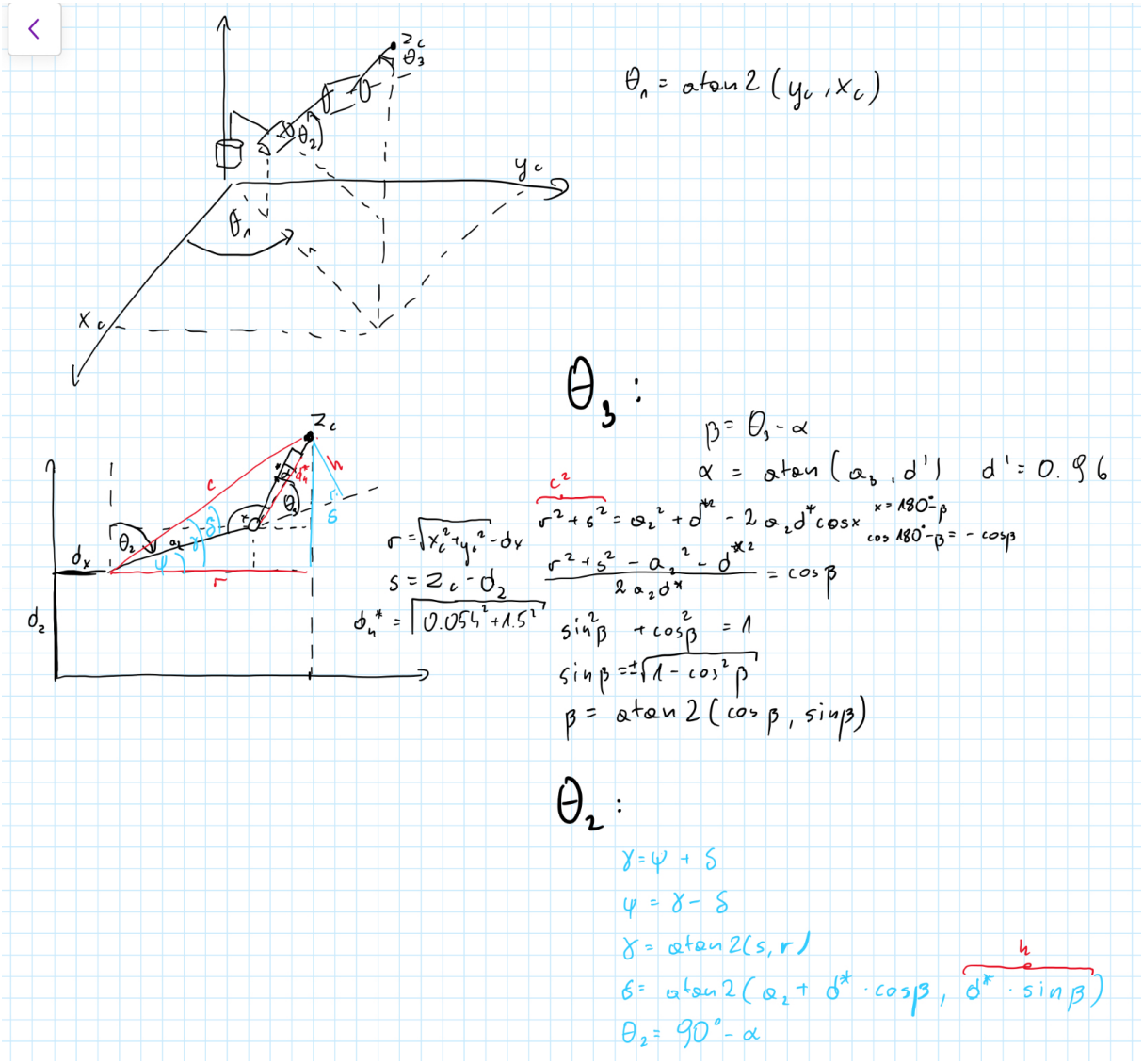

Figure 3: Solution for θ_{1-3}

θ_{4-6} is computed by solving $R^3_{EE} = (R^0_3)^{-1} \cdot R^0_{EE} = (R^0_3)^T \cdot R^0_{EE}$. Since

$$R^3_{EE} = \begin{bmatrix} -\sin(q_4)\sin(q_6) + \cos(q_4)\cos(q_5)\cos(q_6) & -\sin(q_4)\cos(q_6) - \sin(q_6)\cos(q_4)\cos(q_5) & -\sin(q_5)\cos(q_4) \\ \sin(q_5)\cos(q_6) & -\sin(q_5)\sin(q_6) & \cos(q_5) \\ -\sin(q_4)\cos(q_5)\cos(q_6) - \sin(q_6)\cos(q_4) & \sin(q_4)\sin(q_6)\cos(q_5) - \cos(q_4)\cos(q_6) & \sin(q_4)\sin(q_5) \end{bmatrix}$$

and $(R^0_3)^T \cdot R^0_{EE}$ can be computed from input and θ_{1-3} we have:

```
for t5_flip in [1, 0]:
    # Theta5 has an alternative solution at -Theta5
```

```
    theta5 = (1 - 2*t5_flip) * atan2(sinq5, cosq5)

    sgn = sin(theta5)
    sinq4 = R3_EE[2, 2]/sgn
    cosq4 = -R3_EE[0, 2]/sgn
    theta4 = atan2(sinq4, cosq4)

    sinq6 = -R3_EE[1, 1]/sgn
    cosq6 = R3_EE[1, 0]/sgn
    theta6 = atan2(sinq6, cosq6)
```

Mutliple solutions

Both $\sin \beta$ from figure 3 and θ_5 can have more than one solution. For an alternative solution to θ_5, the solution at $-\theta_5$ effectively forces a 180 dg offset to θ_4 and θ_6, and the arm ends up in a "flipped wrist" pose. Additionally each joint can be rotated by $\pm 360^\circ$, as long as it meets the joint specifications. In my program I consider all possible solutions and choose the most cost-effective configuration with regard to transition time between joint poses. The algorithm is explained in detail in the next section.

Project Implementation

I've implemented the solution in `inverse_kinematics.py` in the same directory. For dealing with multiple solutions (regarding the two possible solutions for θ_3, θ_5 or rotation by $\pm 360^\circ$) I compute them all (`SolveIK`, `SolveIKLimits`). Then I discard the solutions that are impossible because of joint constraints specified in the urdf file (`SolveIKLimits`). Later I compute the cost of each solution by taking the maximum time that joint would travel from a previous position to a new position using joint velocities specified in the urdf model. I select the solution with the smallest cost.

Without the cost computation, if any solution is selected, the arm would often just rotate the last few joints in place to an alternative configuration that yields the same EE position. I've found that using cost and computing maximum instead of a sum in the cost computation improves the solution greatly, since the joints can operate in parallel. Even though the solution where the third joint is closer to the floor would not be very useful in the given world configuration, I've found that the cost model would be able to efficiently use it from time to time.

By considering the alternative solutions the robot no longer performs rotations in place, although some unnecessary rotations are still part of the motion plan. This should be optimized in the motion planner or in postprocessing.

My code always assumes there is a solution, I've not worked the cases where the EE position would be out of range or the computations would lead to a division by zero. Especially $\theta_5 = 0$ leads to an infinite number of solutions for θ_4 and θ_6.

Results

I've evaluated the motion planner commands and IK-FK output during simulation. The error between desired and computed EE position is small, less than 0.00001 on average. The robot is able to follow the trajectory perfectly. See the last section of `computations.ipynb` for code.

The robot is usually able to pick up the can, I believe the only times it fails is due to a time lag while grasping.

The simulation would stutter and follow the trajectory in small increments, even though it works smoothly in demo mode. This can be seen on a video. I'm not sure what the reason for that would be.

Sidenotes

I've used a quaternion to a rotation matrix implementation I've found online to be able to work on the solution on a computer without ROS installed.

I've found Jupyter notebook's interactive widgets useful for testing forward kinematics (see "computations.ipynb" Forward Kinematics Playground).

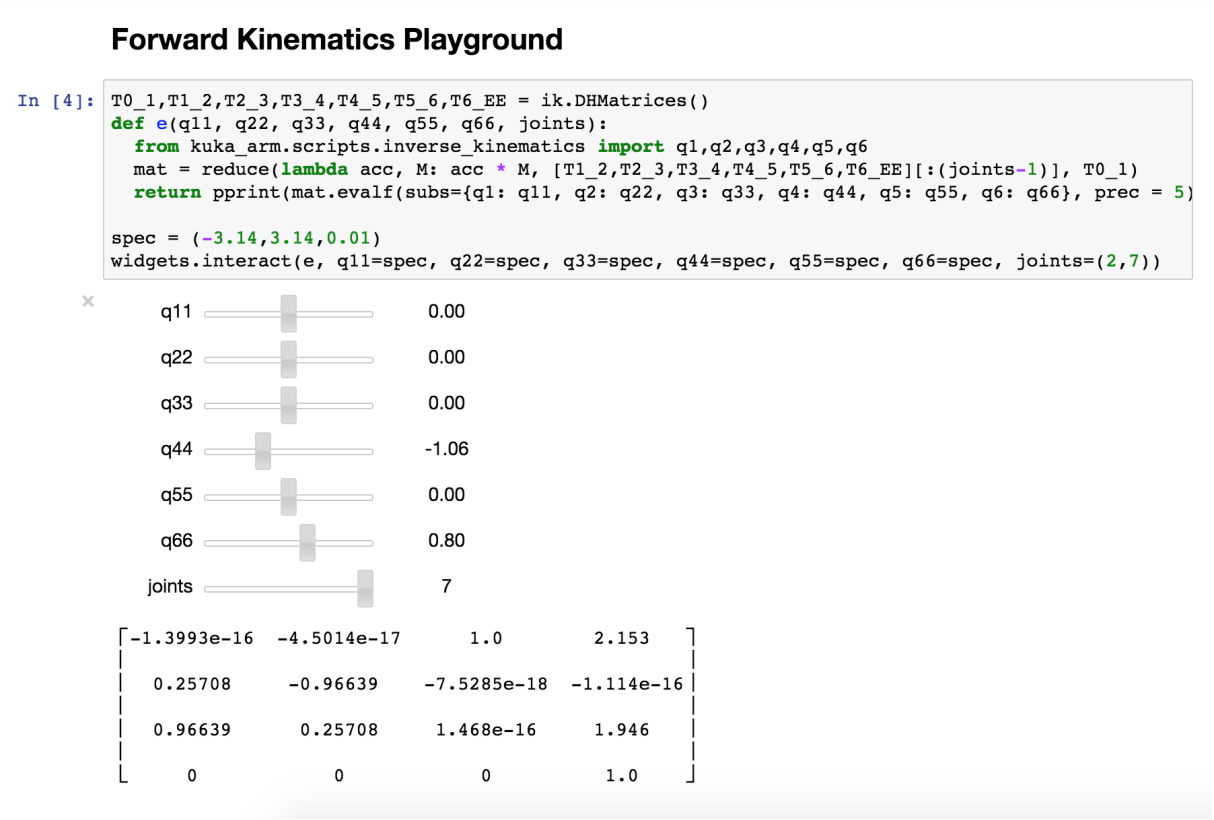

Figure 4: playground