FOUNDATION OF ROBOTICS (ROB-GY6003)
PROJECT 2 REPORT

Rushi Bhavesh Shah (rs7236)

1. Introduction

Serial manipulators are the most common industrial robots which are designed as a series of links connected by motor-actuated joints that extend from a base to an end-effector. Kuka IIWA 14 robot is one such serial manipulator with 7 revolute joints. IIWA stands for "Intelligent Industrial Work Assistant". This project aims to build all the necessary various functionalities for this and similar robot manipulators. This second project aims at building advanced functions (like inverse kinematics, joint control and joint trajectory generation, End-effector control or resolved rate control and Impedance control & Trajectory generation) for this robot.

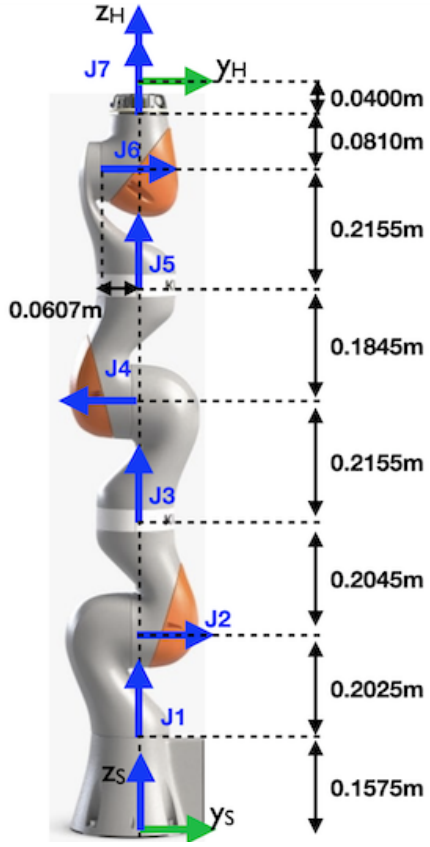

Fig1 : Kuka IIWA 14 robot

In the Fig1, J1, J2, J3, J4, J5, J6 and J7 are all the revolute joints of the robot which rotate about the axis perpendicular to the plane of the joint. This project utilizes robotic python libraries called Pinocchio and Meshcat to display the robot and for other visualization purposes.

2. Methodology

Entire project has been divided into four questions where each question signifies a particular task which needs to be completed.

2.1 Inverse Kinematics

This section aimed at writing two functions for calculating Inverse Kinematics of the given robot.

(i) *compute_IK_position* : This function gets a desired end-effector 3D position (in spatial frame) and returns a vector of joint angles that solves the inverse kinematics problem without considering any singularities.

The algorithm used for calculating inverse kinematics is as follows :

1. Start with initial joint positions.
2. Calculate the forward kinematics with the initial joint positions to get current end-effector position.
3. Calculate the error by subtracting desired end-effector positions and current end-effector positions.
4. Calculate the error in the joint by using following equation

$$\Delta\theta = J(\theta_n)^{-1} . (x^d_e - f(\theta_n)) \qquad (2.1)$$

5. Calculate the new values of joint angles, θ_{n+1} for next iteration.
6. Continue this multiple times till the output stabilizes.

7. Stop when the error is less than or equal to minimum allowance.

Among the given 10 values of *desired_end_effector_positions*, following values don't have possible I_K solutions:
1. [0.38339707,-0.95646426, 1.01716708]
6. [-0.15173455 0.26826613 1.31846463]
7. [-0.79793568 0.42569676 1.29349058]
8. [-0.48681569 -0.75169487 0.7639032]

The IK values calculated have a deviation of about 0.1 to 0.5 from the desired values.

(ii) *compute_IK_position_nullspace* : This function gets a desired end-effector 3D position (in spatial frame) and returns a vector of joint angles that solves the inverse kinematics problem and additionally uses joint redundancy (i.e. the null space) to try and keep the joints close to the following configuration [1,1,-1,-1,1,1,1].

An algorithm similar to the previous function has been adapted for this function as well. However, to eliminate the singularity conditions, a modified equation for calculating the new values of joint angles is used, i.e.

$$\Delta\theta_i = J(\theta_n)^{-1} . (x^d_e - f(\theta_n)) + (I - J^+J).(\theta_d - \theta_i) \quad (2.2)$$

Here, the $(I - J^+J).(\theta_d - \theta_i)$ is the null space term that helps in eliminating the singularity by using J^+, i.e the pseudo-inverse of the Jacobian, over J^{-1} i.e. the normal inverse of the Jacobian. But singularity arises when the Jacobian is non-invertible and hence the need to use pseudo inverse arises.

2.2 Joint control and Joint trajectories generation

This section asks to write a function *get_point_to_point_motion* that returns a desired position and velocity and takes as input the total motion duration T, the desired initial position and the desired final position. The generated trajectory needs to ensure that at t=0 and t=T both the velocity and acceleration are 0. This function can be used to interpolate between desired positions in both joint and end-effector space. Also, a *robot_controller* function is written to move the robot from its initial configuration to reach the first goal (displayed in pink) at t=5 and the second goal ((in yellow) at t=10 by interpolating joint positions using the function *get_point_to_point_motion*. Also the resulting joint simulated and desired positions and velocities are plotted along with the resulting end-effector positions and velocities.

As the first step towards providing a solution for this problem, I found desired joint angle positions using *compute_IK_position_ nullspace* and got *joint_angles_goal1* and *joint_angles_goal2*. To proceed with the next parth, wherein I had to define *get_point_to_point_motion* function, I used the trajectory equation

$$\theta_{des} = \theta_{init} + \left(\frac{10}{T^3}t^3 + \frac{-15}{T^4}t^4 + \frac{6}{T^5}t^5\right) * (\theta_{init} - \theta_{init}) \quad (2.3)$$

$$d\_\theta_{des} = \left(\frac{30}{T^3}t^2 + \frac{-60}{T^4}t^3 + \frac{30}{T^5}t^4\right) * (\theta_{init} - \theta_{init}) \quad (2.4)$$

to get the desired joint positions of the robot. In the equation, T is the total trajectory time and t is the instantaneous time. On similar lines, the derivative of this equation will give the *desired joint velocities* for each instant. Hence, the function *get_point_to_point_motion* will contain both equations (2.3) and (2.4) and give two outputs *desired_joint_positions* and *desired_joint_velocities.*

Furthermore, definition of the *robot_controller* function starts with initializing the variable $t = 0$ and then declaring it as global, which is used in the

simulate_robot function from the *robot_visualizer.py* file, that accepts the *robot_controller* function as its first input parameter. *t* iterates through a loop till it reaches the value of *T* with a step size of *0.001*.

Later on, *t* is equated to *a* or else the same variable will be iterated in both the loops and it will have overlapping values. Hence, to avoid that *t* is equated to *a*. Later on, *desired_joint_positions* and *desired_joint_velocities* are calculated using *get_point_to_point_motion* function. Now, since the robot, in the first half of its motion i.e. till t = 5 sec, has to go till the pink ball, the input arguments for the *get_point_to_point_motion* function will be the one where all joint angles are zero and the goal position will be the position of the pink ball. For the second half, i.e. where time i.e. *a > 5,* similar sequence of instructions is ran, except that the input arguments for the *get_point_to_point_motion* function will change from the position of the pink ball as the initial position and position of the yellow ball as the goal position. Then, the desired *joint_torques* are calculated using the PD control and defined P and D gains within the equation.

Further, the two different plot are formed:

a) Desired vs Actual velocities
b) Desired vs Actual Velocities

All the graphs have two plots each, which depict the two loops, one from 0 to 5 sec and the other one from 6 to 10 sec.

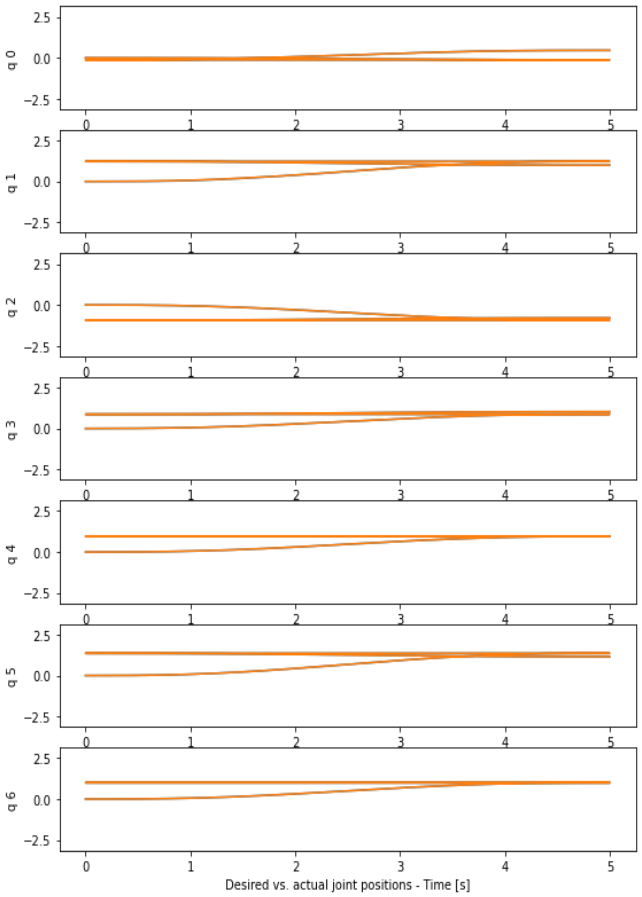

Fig 1 : Desired vs Actual Positions

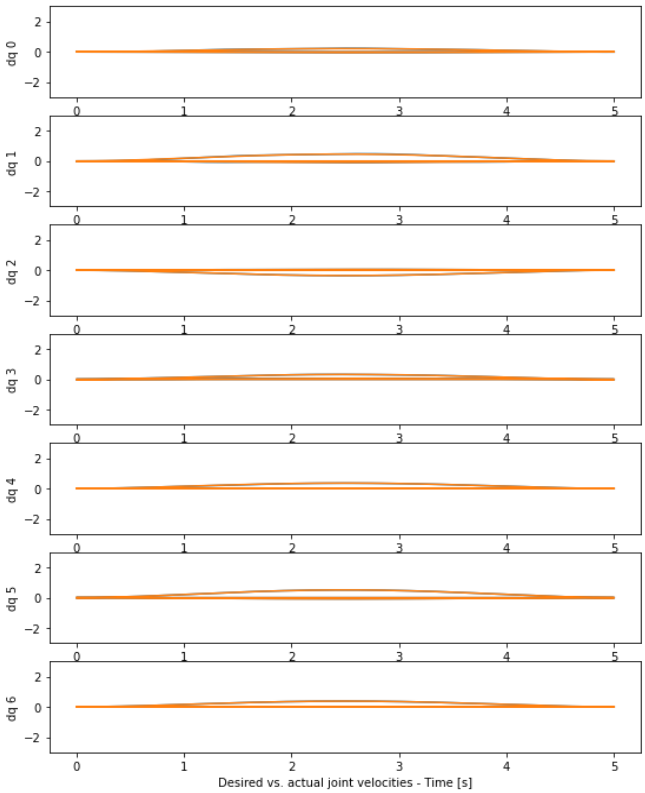

Fig 2 : Desired vs Actual Velocities

2.3 End-Effector Control

As in 2.2, the robot is supposed to go from its initial configuration to the desired end-effector positions (in spatial coordinates) [0.7, 0.2,0.7] in 5 seconds and then to the configuration [0.3, 0.5,0.9] during the following 5 seconds. A *robot_controller2* function has to be written to move the robot from its initial configuration to the first goal (reaching at t=5) and the second goal (t=10) by interpolating the desired end effector positions and directly mapping end-effector error to desired joint velocities (i.e. use P gains equal to 0 in joint space and do resolved-rate control).

For this, a similar approach as 2.2 has been adapted, except that the input parameter to the *get_point_to_point_motion* function will be the end effector positions. For t < 5, the first input parameter will be the end effector position when all the joint angles are zero. The output from this will be *desired_end_effector_positions* and *desired_ end_effector_velocities.* Further, desired joint positions can be calculated using the *compute_IK_position_nullspace* function and the required jacobian is calculated using the *joint+_positions* as the input. Then the desired joint velocities are found using the equation :

$$d\_\theta_{des} = J^{+}.(P(x_{des}(t) - x) + d\_x_{des}) \qquad (2.5)$$

Where J^{+} is the pseudo inverse using **Tikhonov Regularization**. Similar sequence of instructions id followed for time from 5 to 10 sec.

2.4 Impedance Control and Gravity Compensation

As in section 2 and 3, the robot has to go from its initial configuration to the desired end-effector positions (in spatial coordinates) [0.7, 0.2,0.7] in 5 seconds and then to the configuration [0.3, 0.5,0.9]$ during the following 5 seconds.

In the previous sections, a gravity compensation controller was running "in the background" in addition to the control law you were computing. In this question, we remove this and implement a complete impedance controller with gravity compensation. Here, we have a function *robot_visualizer.rnea(q,dq,ddq)* which implements the Recursive Newton Euler Algorithm (RNEA).

The *robot_controller3* function is to be implemented as an impedance controller with gravity compensation (add a small amount of joint damping, using a joint-space D gain of 0.1). Use this controller to move the robot from its initial configuration to the first goal (reaching at t=5) and the second goal (t=10) by interpolating the desired end effector positions as in the previous questions.

$$\tau = J^{T}\left(K(p_0(t) - p_{measured}) + D(\dot{p}_0(t) - \dot{p}_{measured})\right)$$

Here, the above equation has to be used in order to find the torque. The sequence of instructions will be similar as the previous two sections, but the *measured_velovities* is the product of the Jacobian and the joint_velocities and *measured_positions* is the fk of joint_positions.

Gravity compensation is found by putting dq and dqq = in the *robot_visualizer.rnea(q,dq,ddq) function.*