REPORT: SW INTERVIEW CHALLENGE FOR CANDIDATES

This report explains in short some of the decisions I made to solve the tasks which include control flow of the important steps. It also includes the reasoning behind potential compromises I made based on my understanding of the task, limitation of the current solution and how to improve on those with a proposed alternative solution.

Configuring the robot:

A part of the robot's description in the task was a little bit unclear to me. As described, the articulated arm robot has 2 joints and with only 2 joints it's possible to move in a 2D plane only. From a 3D printing robot, I'd expect it to move it's end-effector freely on the XY plane, To move on Z-axis a 3rd joint will be required. So, for this specific solution, I used the 2 joints (RR) to operate on the XY plane and added an extra example project where I implemented movement on 3D space with 3 joints (RRP).

The current solution is a robot with 2 revolute joints which operate on an XY plane. The robot is configured with Denavit–Hartenberg (DH) parameters where rotation axes Z of both joints are parallel. The description is in the "robot_description.txt" file in the same program directory. The path description is also there in the "path_point.txt" file which contains 4 lines, 2 co-ordinates in each line as the task description specifies. 4 lines makes a task space to create a rectangle within the robot's workspace. To execute the program (main.py) from cmd/terminal run the following command:

"python main.py robot_description.txt path_point.txt"

Inside the program:

The "read_robot_description()" method reads the description from "robot_description.txt" and checks if the description is valid, for example, the data type of the fields. Then, a robot is created and stored in the variable "robot_created" with the method "create_robot()". There are configurations for the robot's initial pose. Then path description is read from "path_point.txt" with method "read_path_description()" and stored in the variable "work_path_points". There are 3 major parts in generating the trajectory for the whole path. step 1: from initial position to work_start position, step 2: from work_start position to work_end position, step 3: from work_end position to initial position. The initial pose is set in configuration during initialization of the robot, "work_start" and "work_end" is generated from the "path_point.txt" with defined step_size. "step_size" is defined as 'x'th division of unit 'meter'. So, for creating path with 1mm step size along trajectory with method "find_point2point_trajectory()", step_size will have a value of 1000 (1m/1000 = 1mm).

For each value in trajectory, the joint states are calculated with inverse kinematics. If all the points are within the workspace of the robot and reachable considering the joint rotation limitation then all the joint states for all positions are saved in the "solved_joint_states" variable.

In the next stage, those joint state values are used to check if the velocity conditions are satisfied. Velocity condition is satisfied if the entire moves from point to point are below the maximum angular velocity limit. If all of the above conditions are satisfied then final execution starts. Which

includes the rotation of the motor and printing the joints as required in the task. For convenience, a graphic representation shows the movement of the robot besides printing the values.

Limitations and improvement scope:

- The plotting is resource intensive when step size is 1 mm (1/1000) and slows down my computer. Please use a value of less than 100 throughout the program for the each call of function **find_point2point_trajectory()**. If you want to have a precision of 1 millimeter step size then execute the program with the plotting turned off by commenting out "robot.plot(pose)" inside the function execute_joint_poses(). However, in a powerful PC the program might work fine. The program was tested on a laptop with Intel Core i5 u-series processor with 6 GB RAM.
- This implementation isn't tested to run on real motors. The "motor.py" class is kept at the basic stage. However, this can easily be modified to run on real servo motors. Based on the specification servos can be controlled using a Pulse-Width Modulation (PWM) signal. For practical implementation, the "motor.py" class will have methods to calculate PWM to convert the duty cycle times to percentages by dividing the position's duty cycle by the total PWM Period for all positions of servos. In a real implementation, the servo can send its position values from the encoder readings and the program can check in real-time if each position is reached for each joint before going to the next position. With the checking in place, the system will be robust. Also the value of j2 for printing in "d j0 j1 j2" format is kept at 0.0 because j2 is practically inactive as joint but stays as end-effector.
- PID(Proportional-Integral-Derivative) controller will be part of the solution to run a real motor. The P, I, D values will be calculated with the feedback loop from the motor. The values will be adjusted to run the motor smoothly.
- Movement along the z-axis isn't implemented as the description mentions about 2 joints. To, build the robot I declared third joint j2 with zero value, which can be replaced with a prismatic joint (RRP) to move along the z-axis to go up & down for printing. IN FACT, I IMPLEMENTED A SOLUTION AS SUGGESTED ON A REAL HARDWARE AS A REQUIREMENT OF ROBOTICS COURSE WORK FOR MY MAJOR IN FACTORY AUTOMATION IN MASTER DEGREE STUDIES AT TAMPERE UNIVERSITY. The project is inside RMMCP folder in the link: https://github.com/prasun-biswas/Robotics/tree/master/RMMCP . This project was created with LEGO and BrickPy to satisfy similar requirements of Point-to-point movement of the task "SW interview challenge for candidates" with REAL motors. The solution was developed on Robot Operating System (ROS: 1). The program has 2 files ran on 2 separate ROS Nodes, "planner.cpp" computes the joint states with inverse-kinematics using the KDL library in ROS, "task.py" executes the trajectory on the robot. PID control was implemented for the motor to run smoothly. That robot moves in the XYZ plane and draws lines on paper with a real pen. The pen is attached to the 3rd joint(prismatic) and goes down on Z-axis to draw lines from point to point on XY plane.

*** Please have a look at my project in the link above too. It is possible to build a robust solution to run on a real system combining strategies implemented in the link above & this current solution. ***