Actuators and Mechanisms

1

A conveyor belt inside a candy factory carries colourful hard candy of quasi-spherical shape (e.g. Skittles, M&Ms etc.). The candy is assumed to be stationary with respect to the conveyor, which moves at a constant velocity which cannot be altered. The conveyor belt is approximately 20cm wide and the candy appears randomly along its width. Assume the weight and shape of the candy is known. You are asked to implement a robotic system in addition to the conveyor belt to extract a candy of any given colour and place it in a separate basket. The system should be able to perform the task as fast as possible. Ensure that you are designing a minimum viable system to solve the task, hence, avoid for example redundancy in your manipulator design as well as your actuator and sensor choices. Propose a robotic system to perform the task. In your answer, address the following. (Recommended answer is up to 50 words per sub-question.)

a

Required degrees of freedom of your manipulator in Cartesian space. [report - 2 pts]

The workspace requires 3 degrees of freedom ; x, y, z. As the candy moves along the conveyor belt in x, they vary in position along y, and require to be picked up in the z axis.

b

Manipulator topology (serial/parallel) and design. You are free to reference commonly utilised designs and robot types. Also state a reasonable choice of end- effector.[report - 2 pts]

A parallel Cartesian manipulator with 3 DOF, as it has high accuracy required for small targets such as candy, and high speed capabilities for the conveyor belt. The workspace is relatively small (20 cm + basket width * y * z), so this suits the parallel manipulator. Furthermore, the end effector stays parallel to the conveyor belt, allowing for consistent picking across the belt. A reasonable choice of end effector is a 3 fingered gripper, as it adapts to the candy's shape to form a solid grip

c

Choice of actuators (e.g. stepper motors, AC motors, brushed or brushless DC motors, pneumatics, hydraulics etc.) and transmission. This will vary greatly with your manipulator design.[report - 2pts]

The actuators used should be AC induction servomotors, as they offer position and velocity feedback through embedded sensors. The motor stator is 3-coil windings (geometrically spaced by 120^0) with 3-phases currents (electrically spaced by 120^0). The power input of this motor is the stator, not the rotor as in the case of DC motors. Induction motors can also be easily controlled in terms of speed, as speed is proportional to frequency of the power supply. AC induction motors are also very robust, and brush-less, suitable for long life span in factory setting.

d

Choice of sensors (both, required for driving the manipulator as well as determining the correct object).[report - 2 pts]

To determine the correct object, a camera and computer vision algorithm should be used. The object being picked each time is very similar, so the algorithm will be straightforward to train to a high accuracy. To drive the manipulator, a potentiometer rotary position sensor should be used, this measures displacement in a rotary fashion. Speed feedback is not required as the speed of the AC Induction motor is proportional to the frequency of the input power supply.

e

Discuss a component of the proposed system (e.g. joints, actuators, sensors) which would be prone to wear from repeated task execution. How could the wear be minimised?[report - 3 pts]

An induction AC brushless servo motor with rotary position sensors is a no contact system, thus is very However in an AC motor, the armature windings can burn out and short circuit due to size reduction, overloading, inadequate protection circuitry, or excess heat. Also, over time, bearings or lubrication fatigue can cause failure in this motor type.

f

Assume, only candy with the previously specified colour above a certain weight threshold (e.g. $>$ 5 grams) should be placed in the basket. How could you modify your system to be able to consider this additional factor without modifying the conveyor? [report - 4 pts]

Modify the computer vision algorithm to only identify a certain colour as the target candy. To only choose candy $>$ 5 grams, the system should utilise strain gauges in the base to measure the weight of the robot before and after picking the candy, if the weight does not increase by $>$ 5 grams, then the candy is dropped into a rejected bin, and the process repeats.

Robot Dynamics

2

Complete the following tasks by filling in the "cw3q2/iiwa14DynStudent.py" python class code template, to compute the dynamic components for the KUKA LBR iiwa14 manipulator. A simple code breakdown in the report is required for all subquestions. In the cw3q2 folder you can find three files.

- *iiwa14DynStudent.py: This is the class template for the questions below.*
- *iiwa14DynBase.py: This class includes common methods you may need to call in order to solve the questions below. You should not edit this file.*
- *iiwa14DynKDL.py: This class provides implementations to the questions below in KDL in order to check your own solutions. You should not edit this file.*

a

Write a script to compute Jacobian at the centre of mass for the iiwa14 manipulator. [report - 2pts, code - 3 pts]

The jacobian at the centre of mass of a link is defined as as follows:

$$\mathbf{J}_{P_j}^{(l_i)} = \begin{cases} \mathbf{z}_{j-1}, & \text{for a prismatic joint} \\ \mathbf{z}_{j-1} \times (\mathbf{p}_{l_i} - \mathbf{p}_{j-1}), & \text{for a revolute joint} \end{cases}$$

$$\mathbf{J}_{O_j}^{(l_i)} = \begin{cases} 0, & \text{for a prismatic joint} \\ \mathbf{z}_{j-1}, & \text{for a revolute joint} \end{cases}$$

As the Kuka iiwa consists of revolute joints only, the jacobian at the centre of mass is calculated thusly. The forward kinematics at the centre of mass function is used at the link's centre of mass to calculate the jacobian J.

```
def get_jacobian_centre_of_mass(self, joint_readings, up_to_joint=7):
        """Given the joint values of the robot, compute the Jacobian
   matrix at the centre of mass of the link.
        Reference - Lecture 9 slide 14.

        Args:
            joint_readings (list): the state of the robot joints.
            up_to_joint (int, optional): Specify up to what frame you want
    to compute the Jacobian.
            Defaults to 7.

```

```
        Returns:
            jacobian (numpy.ndarray): The output is a numpy 6*7 matrix describing the Jacobian matrix defining at the
            centre of mass of a link.
        """
        assert isinstance(joint_readings, list)
        assert len(joint_readings) == 7

        # Your code starts here ----------------------------
        # T07 * ROT * TRANS
        jacobian = np.zeros((6,7)) #INitialise the jacobian
        T0Gi = self.forward_kinematics_centre_of_mass(joint_readings, up_to_joint)
        p    = T0Gi[0:3,3]
        T = []
        for i in range(up_to_joint):
            T.append(self.forward_kinematics(joint_readings, i))
            Tr = T[i]
            z_prev = Tr[0:3,2]
            p_prev = Tr[0:3,3]

            jacobian[0:3,i] = np.cross(z_prev, (p - p_prev))
            jacobian[3:6, i] = z_prev

        # Your code ends here ----------------------------

        assert jacobian.shape == (6, 7)
        return jacobian
```

Listing 1: 2a code

b

Fill in the appropriate class method to compute the dynamic component $\mathbf{B}(\mathbf{q})$ *for the iiwa14 manipulator.*[report - 2 pts, code 8 pts]

The Inertia matrix $\mathbf{B}(\mathbf{q})$ as a function of the joint positions is defined as:

$$\mathbf{B}(\mathbf{q}) = \sum_{i=1}^{n} (m_{l_i} \mathbf{J}_P^{(l_i)^T} \mathbf{J}_P^{l_i} + \mathbf{J}_O^{(l_i)^T} \mathbf{J}_{l_i} \mathbf{J}_O^{(l_i)}) = [b_{ij}]_{n \times n}$$

```
def get_B(self, joint_readings):
        """Given the joint positions of the robot, compute inertia matrix B.
        Args:
            joint_readings (list): The positions of the robot joints.

```

```
        Returns:
            B (numpy.ndarray): The output is a numpy 7*7 matrix describing the inertia matrix B.
        """
        B = np.zeros((7, 7))

      # Your code starts here ------------------------------
        #Lecture 9slide 16
        #B(q) = [b_ij]_nxn
        #Inertia matrix2
        #R0G = []
        #Jli = []

        for i in range(1,8):
            #i+1
            #Get transformation R from T
            R0G = self.forward_kinematics_centre_of_mass(joint_readings, i)[0:3, 0:3]

            Ioli = np.zeros((3,3))
            for j in range(3):
                Ioli[j,j] = self.Ixyz[i-1, j]
            Ioli = np.matmul(np.matmul(R0G,Ioli),R0G.T)

            mli = self.mass[i - 1]
            Jcm = self.get_jacobian_centre_of_mass(joint_readings, i)
            Jpli = Jcm[0:3, :]
            Joli = Jcm[3:6, :]


            B += mli * np.matmul(Jpli.T , Jpli) + np.matmul(np.matmul(Joli.T,Ioli),  Joli)

        # Your code ends here ------------------------------

        return B
```

Listing 2: 2b code

c

Fill in the appropriate class method to compute the dynamic component $C(q, \dot{q})\dot{q}$ for the iiwa14 manipulator.[report - 2pts, code - 6 pts]

The $(i, j)^{th}$ element of the matrix $C(\mathbf{q}, \dot{\mathbf{q}})$ is defined as:

$$c_{ij} = \sum_{k=1}^{n} h_{ijk}\dot{q}_k$$

Where:

$$h_{ijk} = \frac{\partial b_{ij}(\mathbf{q})}{\partial q_k} - \frac{1}{2}\frac{\partial b_{jk}(\mathbf{q})}{\partial q_i}$$

and satisfies:

$$\sum_{j=1}^{n} c_{ij}\dot{q}_j = \sum_{j=1}^{n}\sum_{k=1}^{n} h_{ijk}\dot{q}_k\dot{q}_j$$

```python
def get_C_times_qdot(self, joint_readings, joint_velocities):
        """Given the joint positions and velocities of the robot, compute Coriolis terms C.
        Args:
            joint_readings (list): The positions of the robot joints.
            joint_velocities (list): The velocities of the robot joints.

        Returns:
            C (numpy.ndarray): The output is a numpy 7*1 matrix describing the Coriolis terms C times joint velocities.
        """
        assert isinstance(joint_readings, list)
        assert len(joint_readings) == 7
        assert isinstance(joint_velocities, list)
        assert len(joint_velocities) == 7

        # Your code starts here ------------------------------
        #L9S19
        # C = h_ijk qdot_k
        B = self.get_B(joint_readings)
        C = np.zeros((7,7))

        delta = 0.001 # small delta to approximate pde well
        for i in range(7):
            for j in range(7):
                for k in range(7):
                    q_k_copy = np.copy(joint_readings)
                    q_k_copy[k] += delta
                    bij = self.get_B(q_k_copy.tolist() ) # do this sum elementwise!

                    q_i_copy = np.copy(joint_readings)
                    q_i_copy[i] += delta
                    #bjk = self.get_B(joint_readings[i] + delta )
                    bjk = self.get_B(q_i_copy.tolist() )

                    dbij_dq = (bij[i,j] - B[i,j]) / delta
                    dbjk_dq = (bjk[j,k] - B[j,k]) / delta

                    C[i,j] = (dbij_dq - (0.5 * dbjk_dq)) * joint_velocities[k]
                    #sum_j (sum_k(h_ijk *q_dot_k *q_dot_j))
```

```
                    #print(str(type(C)))
        C = np.matmul(C , np.array(joint_velocities))
        print(C.shape)
        # Your code ends here ------------------------------

        assert isinstance(C, np.ndarray)
        assert C.shape == (7,)
        return C
```

Listing 3: 2c code

d

Fill in the appropriate class method to compute the dynamic component $\mathbf{g}(\mathbf{q})$ *for the iiwa14 manipulator.* [report - 2 pts, code 5 pts]

The dynamic component $\mathbf{g}(\mathbf{q})$ can be calculated as follows:

$$\mathbf{g}(\mathbf{q}) = -\sum_{i-1}^{n} m_{l_i} \mathbf{g}_0^T \mathbf{p}_{l_i}$$

Where m_{l_i} is the mass of the link l_i, $\mathbf{p_{li}}$ is the position vector of the centre of mass of link l_i and $\mathbf{g}_0^T = [0, 0, -g]^T$ of z in the base frame is vertical axis pointing up.

```
def get_G(self, joint_readings):
        """Given the joint positions of the robot, compute the gravity
    matrix g.
        Args:
            joint_readings (list): The positions of the robot joints.

        Returns:
            G (numpy.ndarray): The output is a numpy 7*1 numpy array
    describing the gravity matrix g.
        """
        assert isinstance(joint_readings, list)
        assert len(joint_readings) == 7

        # Your code starts here ------------------------------
        #l9s17
        G = np.zeros((7,))
        g = [0,0,self.g]
        g = np.array(g)
        delta = 0.001
        for i in range(7):
            G_i = 0
            for j in range(7):
                mli = self.mass[j]
                #change in pe = force
                J_l = self.get_jacobian_centre_of_mass(joint_readings, j
    +1)[0:3,i]
```

```
                J_l = J_l.reshape(3,1)
                G_i += mli * np.matmul(np.array(g), J_l)

            G[i] = G_i

        # Your code ends here ------------------------------

        assert isinstance(g, np.ndarray)
        assert G.shape == (7,)
        return G
```

Listing 4: 2d code

3

Describe the Huygens-Steiner theorem. Your answer should include a description of its hypothesis and of its derivation. Explain why it is important in robotics applications.[report - 7 pts]

The Huygens-Steiner theorum is the Parallel-Axis theorum can be used to determine the moment of inertia or the second moment of area of a rigid body about any axis, given the body's moment of inertia about a parallel axis through the object's centre of gravity and the perpendicular distance between the axes.
We may assume that the moment of inertia of a body relative to the z axis, in cartesian co-ordinates is:

$$I_{cm} = \int (x^2 + y^2) dm$$

Where:

- Moment of inertia at the centre of mass: I_{cm}
- Elementary mass: dm
- Cartesian coordinates: (x, y)

The moment of inertia relative to the axis z', distance D from the centre of mass along the x-axis, is thus [1];

$$I = \int ((x - D)^2 + y^2) dm$$

Expand the brackets;

$$I = \int (x^2 + y^2) dm + D^2 \int dm - 2D \int x dm$$

As the centre of mass lies at the origin, the integral in the final term = 0.

$$I = I_{cm} + mD^2$$

Which states that the mass moment of inertia defined wrt. a non-centroidal co-ordinate system is equal to the mass moments of inertia defined wrt. the axis of centroidal co-ordinates plus the product of the mass m and the square distances D^2 between the axis [2]. This is useful in robotics as often many co-ordinate systems are used for different robotic joints, with links that have centre of masses not in the origin of the other co-oridnate systems. Thus the parallel axis theorem allows for linking up the co-ordinate systems.

4

Define the forward and inverse dynamics problems, and highlight their main applications. Describe what are the difficulties of each problem. [report - 8 pts]

Forward Dynamics problems refer to the use of kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters. The main applications of forward dynamics is mathematically modelling the kinematic behaviour of a robot manipulator to determine position/end effector of a robot given joint and link parameters. Aswell as modelling the relations between joint velocities and end effector velocities. The difficulties in dynamics are the identification and decoupling of singularities, as singular configurations should be avoided in robot trajectories.

Inverse Dynamics problems compute the joint parameters that achieve a specified position of the end effector. The main applications of inverse dynamics is determining joint and link parameters given the kinematic behaviour of the robot. Difficulties in inverse dynamics include highly non-linear problems as there can be many values of joint variables that are required to achieve a given robot pose. Thus many transformations exist and this makes the discipline highly computationally demanding. Also, redundancy in inverse kinematics problems means there can be more than one robot configuration to achieve a certain robot pose.

5

In this question, you are tasked with computing the joint accelerations throughout the trajectory defined in the bagfile "cw3q5/bag/cw3q5.bag" using dynamic components. You should also plot the computed joint accelerations. You only need to edit the "cw3q5.py" file, and the corresponding launch file. Note that a coding template is not provided for this question. For the dynamic components, you can use either your own implementation from Q2, or the corresponding KDL class. Synchronising ROS rate, bagfile reading, rviz executing the trajectory, and the function calling can prove tricky. Thus, adding artificial delays in parts of the script is allowed.

a

Load the bag from the bagfile. What type of message does the bagfile contain? How many messages does the bagfile have, and what is the content of the messages? [report - 3 pts, code 4 pts]

The bagfile "cw3q5.bag" contains 1 message, of type trajectory_msgs/JointTrajectory, and topic /iiwa/EffortJointInterface_trajectory_controller/command. The types messages are joint_names, points, position, velocity and acceleration [3]; and can be accessed through the JointTrajectory() or JointTrajectoryPoint() messages.

The function 'traj' imports the messages joint names, points and corresponding positions and velocities from the bag file, and assigns them to 'joint_traj'.

```
def traj(self):
        '''Import from bagfile and publish the trajectory to the appropriate topic to see
        the robot moving in simulation.'''
        #Open bagfile
        joint_traj  = JointTrajectory()
        #args  in trajectory_msgs joint_traj are ['header', 'joint_names', 'points']
        #Header
        joint_traj.header.stamp = rospy.Time.now()
        #Get bag path
        path = rospack.get_path('cw3q5') + '/bag/cw3q5.bag'
        #Open bag
        bag = rosbag.Bag(path)
        #Read messages in bagfile from specific topic
        for topic, msg, t in bag.read_messages(topics=['/iiwa/EffortJointInterface_trajectory_controller/command']):
            #trajectory_msgs/JointTrajectoryPoint has attributes positions, velocities, accelerations, effort, duration time_from_start

            # Get joint names
            for name in msg.joint_names:
                #Joint names
                joint_traj.joint_names.append(name)
                #Points
            for points in msg.points:
                points_obj = JointTrajectoryPoint()
                for i in range(7):
                    #positions
                    points_obj.positions.append(points.positions[i])
                    #velocities
                    points_obj.velocities.append(points.velocities[i])
                    #accelerations
                    points_obj.accelerations.append(points.accelerations[i])
                    #duration
                    points_obj.time_from_start = points.time_from_start
```

```
                joint_traj.points.append(points_obj)
        #Close bag
        bag.close()
        return joint_traj
```

Listing 5: 5a code

b

Is this a problem of forward or inverse dynamics?[report - 3 pts]

This is a problem of inverse kinematics, as we have the joint trajectories, and the joint parameters such as position, velocity and acceleration are being calculated.

c

Publish the trajectory to the appropriate topic to see the robot moving in simulation.[report - 2 pts, code 3 pts]

The listing below shows the ROS node being initiated, joint trajectories are then taken from the bagfile via the function 'traj', and then publishing the appropriate topic via the rospy publisher. Artificial delays are used to help synchronise ROS rate, bagfile reading and RVIZ execution.

```
if __name__ == '__main__':
    try:
        counter = 0
        counter +=10
        #Initialise node
        rospy.init_node('cw3', anonymous=True)


        iiwa_planner = iiwaTrajectoryPlanning()
        rospack = rospkg.RosPack()
        #rospy.spin()

        #Get joint_traj from bagfile
        joint_trajectory  = JointTrajectory()
        trajectory_publisher = rospy.Publisher('/iiwa/
   EffortJointInterface_trajectory_controller/command', JointTrajectory,
   queue_size= 5)
        #Calculate Joint trajectories
        joint_traj  = iiwa_planner.traj() #TODO

        #Artificial delay
        rospy.sleep(1)
```

```
        #Publish joint trajectories
        trajectory_publisher.publish(joint_traj)
        #Artificial delay
        rospy.sleep(1)

        #Calculate Acceleration
        #Subscribe to the topic to get postion and velocities.
        state_subscriber = rospy.Subscriber('/iiwa/joint_states',
    JointState, iiwa_planner.acceleration)# subscriber

        plt.ion
        plt.show()
        rospy.spin()


    except rospy.ROSInterruptException:
        pass
```

Listing 6: 5c code

d

Subscribe to the appropriate topic, and calculate the joint accelerations throughout the trajectory using dynamic components.[report - 3 pts, code - 12 pts]

$$\mathbf{B}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) + \mathbf{J}^{\mathbf{T}}\mathbf{F}_{\mathbf{hard}} = \tau$$

As the inertia matrix $\mathbf{B}(\mathbf{q})$ is always invertible, we can rearrange for acceleration $\ddot{\mathbf{q}}$:

$$\ddot{\mathbf{q}} = \mathbf{B}(\mathbf{q})^{-1}(\tau - \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} - \mathbf{g}(\mathbf{q}))$$

Where $\mathbf{J}^{\mathbf{T}}\mathbf{F}_{\mathbf{hard}} = \tau_{\mathbf{ext}} = \mathbf{0}$ external torques on the robot are equal to zero. The acceleration is calculated via the function shown in the code listing below.

```
def acceleration(self,joint_state): #kdl in inputs?'####################
        '''Subscribe to the appropriate topic,
        and calculate the joint accelerations throughout
        the trajectory using dynamic components'''
        #Import position, velocity, torque
        #print(joint_state)
        #SUbscribe to the joint_traj message and get position, velcoities,
    and torques

        q = np.array(joint_state.position)
        q_dot = np.array(joint_state.velocity)
        tau = np.array(joint_state.effort)

        #Get B using KDL incase of any mistakes in Q2
```

```
        B = Iiwa14DynamicKDL.get_B(Iiwa14DynamicKDL(), q)
        #Get C_times_q_dot
        C_qdot = Iiwa14DynamicKDL.get_C_times_qdot(Iiwa14DynamicKDL(), q , q_dot)
        #Get G
        G = Iiwa14DynamicKDL.get_G(Iiwa14DynamicKDL(), q)
        #Calculate accleration q_ddot
        q_ddot = np.matmul(np.linalg.inv(B), (tau - C_qdot - G))
        q_ddot = q_ddot.reshape((7,))


        stamp = joint_state.header.stamp
        time = stamp.secs + stamp.nsecs *1e-9
        plt.plot(time, q_ddot[:,0], 'k-')
        plt.plot(time, q_ddot[:,1], 'r+')
        plt.plot(time, q_ddot[:,2], 'b*')
        plt.plot(time, q_ddot[:,3], 'g-')
        plt.plot(time, q_ddot[:,4], 'k*')
        plt.plot(time, q_ddot[:,5], 'r*')
        plt.plot(time, q_ddot[:,6], 'g*')
        plt.legend()
        plt.title('Joint Accelerations')
        plt.xlabel('Time')
        plt.ylabel('Acceleration')
        plt.draw
        plt.pause(1e-5)
```

Listing 7: 5d Acceleration function code

e

Plot the joint accelerations as a function of time in your python script.[report - 5 pts, code - 5 pts]

The output of the acceleration function "q_ddot" is plotted in the figure below, plotting 7 joint accelerations, each with joint number and acceleration of said joint. See the code listing in 5d.

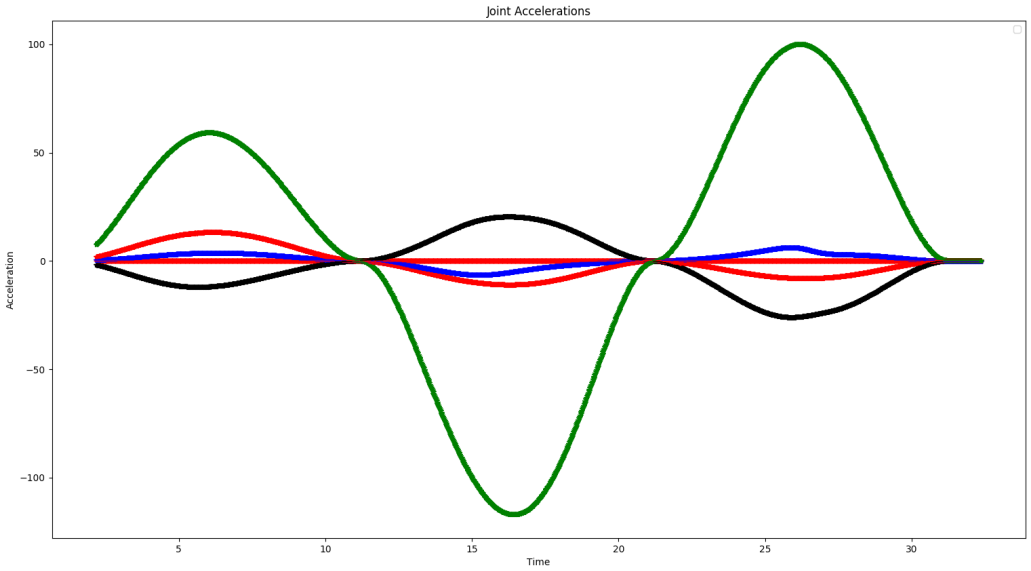

Figure 1: Joint Accelerations

References

[1] L. Page *et al.*, "Introduction to theoretical physics," 1928.

[2] S. Staicu, *Dynamics of Parallel Robots*. Springer, 2019.

[3] Ros.docs.org, "Ros Docs,"

Appendix

```
<?xml version="1.0"?>
<launch>
    <arg name="rviz" default="true"/>
    <arg name="robot_description" default="$(find iiwa_description)/urdf/iiwa14.urdf.xacro" />
    <!-- Loads the world. -->
    <include file="$(find iiwa_gazebo)/launch/iiwa_world.launch">
        <arg name="hardware_interface" value="hardware_interface/EffortJointInterface" />
        <arg name="robot_name" value="iiwa" />
        <arg name="model" value="iiwa14" />
        <arg name="paused" value="false"/>
        <arg name="use_sim_time" value="true"/>
        <arg name="gui" value="false"/>
        <arg name="debug" value="false"/>
    </include>

    <!-- Spawn controllers - it uses a JointTrajectoryController -->
    <group  ns="iiwa">
        <include file="$(find iiwa_control)/launch/iiwa_control.launch">
            <arg name="hardware_interface" value="hardware_interface/EffortJointInterface" />
            <arg name="controllers" value="joint_state_controller EffortJointInterface_trajectory_controller" />
            <arg name="robot_name" value="iiwa" />
            <arg name="model" value="iiwa14" />
        </include>
    </group>

    <node name="static_tf_pub_world_to_gazebo_world" pkg="tf2_ros" type="static_transform_publisher" args="0 0 0  0 0 0 1 map world" />


    <node name = "cw3" pkg = "cw3q5" type = "cw3q5.py" output = "screen"/>

    <group if="$(arg rviz)">
    <node name="$(anon rviz)" pkg="rviz" type="rviz" respawn="false" args="-d $(find cw3q5)/config/iiwa14.rviz" output="screen"/>
    </group>

</launch>
```

Listing 8: Question 5 launch file