RBE 500 Group Assignment #2

Joshua Gross, Arjan Gupta, Melissa Kelly

Problem 1

Create ROS Package for PD Controller

Preliminary Work with Gazebo

Before creating the ROS package for the PD controller we performed some reconnaissance.

When we executed `ros2 topic list`, we saw that the `/forward_effort_controller/commands` topic was part of it. We then executed the following command in an attempt to move the joints.

```
ros2 topic pub --once /forward_effort_controller/commands ...
    std_msgs/msg/Float64MultiArray "{data: [1, 1, 1]}"
```

However, this took no effect. We remember from the last group assignment (Part 1) that we executed a very similar command to make the joints move to a specific position, except in that case the topic we published to was `/forward_position_controller/commands`. This gave us a hint to the fact that Gazebo was preferring the position controller over the effort controller. Upon discovering the `controller_switch.cpp` file in the rrbot simulation files, we realized that we must 'activate' the effort controller in order to use it. Next, we did some more discovery using terminal commands. We executed

```
ros2 service list -t
```

This helped us see that `/controller_manager/switch_controller` was an available service we could use. Since we supplied the -t flag to this command, we could also see that `controller_manager_msgs/srv/SwitchController` was the type of message we had to use. To further take note of the message type, we executed

```
ros2 interface show controller_manager_msgs/srv/SwitchController
```

Here we could see that `activate_controllers` and `deactivate_controllers` were parameters of the message. Now we put together our findings and executed the following command that we constructed.

```
ros2 service call /controller_manager/switch_controller ...
    controller_manager_msgs/srv/SwitchController "{activate_controllers: ...
    ["forward_effort_controller"], deactivate_controllers: ...
    [forward_position_controller, forward_velocity_controller]}"
```

Upon doing this, we saw the the prismatic joint drop. This means something took effect. We now tried our initial command,

```
ros2 topic pub --once /forward_effort_controller/commands ...
    std_msgs/msg/Float64MultiArray "{data: [1, 1, 1]}"
```

Now we saw the robot in Gazebo move! The two revolute joints 'spun' in an anti-clockwise direction in the XY plane, whereas the prismatic joint did not do anything. With some more experimentation we realized:

- Acceleration due to gravity, approximated in magnitude to $9.8 m/s^2$ was acting on the prismatic joint. Therefore, in order to make it move upward we needed to publish an effort with an acceleration of anything less than $-9.8 m/s^2$.

Problem 1 continued on next page...

- In order to hold the prismatic joint in at any height, we needed to apply exactly $-9.8m/s^2$ acceleration. The negative sign is a consequence of the direction of motion we have defined in our URDF file.
- Since our joint masses are simply 1, we were able to publish a joint effort as simply -9.8, for example.
- We also realized that since there was no opposing force readily acting upon the revolute joints, if we executed an effort to them, that effort would keep pushing along the joints.
- In order to hold the revolute joints in a particular angle, we needed to specify that we are applying 0 effort. However this did not instantaneously stop the rotation of these joints, we noticed a de-acceleration effect before the joint came to a stop.
- As for implementing our package, this means that we will first need to execute a service call (by creating a client) to activate the right controller. As for executing efforts via the PD controller, we will need to account for gravity in case of the prismatic joint.

Preliminary Work for Controller

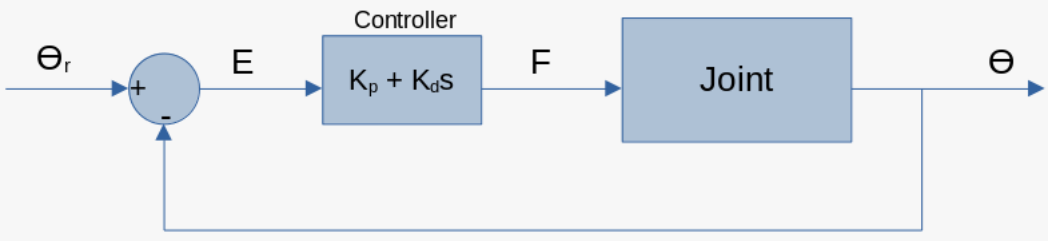

Figure 1: Standard PD controlled closed-loop system

Taking inspiration from how we have modeled PD controllers so far in this course, we can imagine the control for each joint as shown in Figure 1.

Here, the goal position, θ_r will be received via a service call. Our package will must contain a service for this purpose. The current position θ is received via the `/joint_states` topic. The error, E is calculated as $E = \theta_r - \theta$. This error is then with some untuned multiplier (gain) K_p to obtain the first term of our controller.

Next, we need to get the derivative of the error, $\dot{E}$, which could be expressed as

$$\dot{E} = \dot{\theta_r} - \dot{\theta}$$
$$\dot{E} = 0 - \dot{\theta}$$
$$\dot{E} = -\dot{\theta} = -v$$

Therefore the derivative of the error is the negative of v, the velocity of the joint. This velocity can be obtained from the `/joint_states` topic or it can be approximated by looking at the position in the last 'step' and the current step. Once we have $\dot{E}$, we use another untuned multiplier (gain) K_d to obtain the second term of our controller. Therefore, our controller is

$$F = K_p E + K_d \dot{E} = K_p E - K_d v$$

Where F is the resultant effort that we need to apply to the joint.

Writing the ROS Package

Now we create the package just as we've done in our past assignments of this course. Inside the package, we created a `scara_pd_controller.py` file. We created an empty Node in this file which gets called by the main function. We edited the package.xml and setup.py to make sure that we can build the basic package.

Next, we added a new custom interface by editing our rbe500 custom interface package from the previous package. This interface, which we named `ScaraRefPos.srv` will be used to make the service call to our Node to supply the reference position in order to get the controller started. This interface can be used for a service call. The interface has the 3 joint variables in its request portion and a simple acknowledgement boolean inside its response portion. We now import this message inside our main package's python file. The custom message is used to create the service within our package which will set the reference position for the SCARA robot. We define a callback for this service which sets the position as member variables, as well as a flag to indicate that we are ready to start our controller. After the service is created, we log a message to stdout that informs that we have created the service successfully.

Now we move on to creating the subscriber that receives the joint state information. We subscribe on the `/joint_states` topic and indicate that we intend to receive the `JointState` message from the `sensor_msgs` package. The queue for this subscriber is 100 messages long because information is published at a very high frequency on the joint states topic. We assign a callback for this subscription that firsts checks if we have a reference position to work with, and if not, simply notifies that we are awaiting the same. We added a print after the creation of the subscriber to inform that the creation was successful.