ROBT 403 Laboratory Report 2

Daniil Filimonov, Abdirakhman Onabek, Kir Smolyarchuk

GitHub:

https://github.com/Meneralllka/FailTeamLab2/tree/master

Disclaimer

Since we had to use a lot of packages from other github repositories and then upload it to ours, we would kindly ask you to navigate to the path shown in the figure below to get directly to the code that we have written. Overall, there are two files – one for the step message, another for the sine. Those were enough to finish all the tasks.

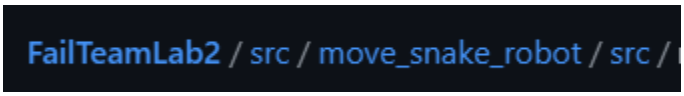

Figure 1. Path to the code

Also, it would be important to mention that code provided to us was written partially on *Python 2*, which is not supported by default in *ROS* Noetic. Hence, we had to install *Python 2* manually, make it first in the priority list of python versions in *Ubuntu,* and then we also had to install missing libraries.

Last but not least, since the software changed, it became impossible to change PID constants in the *RViz* or *Gazebo* for the practical robot. Hence, Zhanat told to abandon **Task 4.**

1. Task 1 & 2 – Listen and Publish, Step Response

This code (figure 2) is an implementation of a *ROS* node called "rotate" that acts as a publisher for controlling a Dynamixel motor. It sends commands to the `/joint2/command` topic using a `Float64` message variable. Right now, it's sending a step response (figure three) with the value `1.0`, meaning the motor should instantly jump to this position.

```cpp
#include "ros/ros.h"
#include "std_msgs/Float64.h"

//for sin()
#include <math.h>

int main(int argc, char **argv) {
    ros::init(argc, argv, "rotate");

    ros::NodeHandle nh;

    ros::Publisher pub2 = nh.advertise<std_msgs::Float64>("/joint2/command", 100);

    ros::Rate loop_rate(10);

    ros::Time startTime = ros::Time::now();

    while (ros::ok()) {
        // Create a message to publish
        std_msgs::Float64 msg_to_send;

        // Calculate elapsed time
        ros::Duration elapsed = ros::Time::now() - startTime;
        double time = elapsed.toSec();

        // Parameters for the sine wave

        // Compute the sine wave value
        msg_to_send.data = 1.0

        // Publish the message
        pub1.publish(msg_to_send);



        // Log the published value
        ROS_INFO("Publishing step value: %f", msg_to_send.data);

        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

Figure 2. Publisher Code

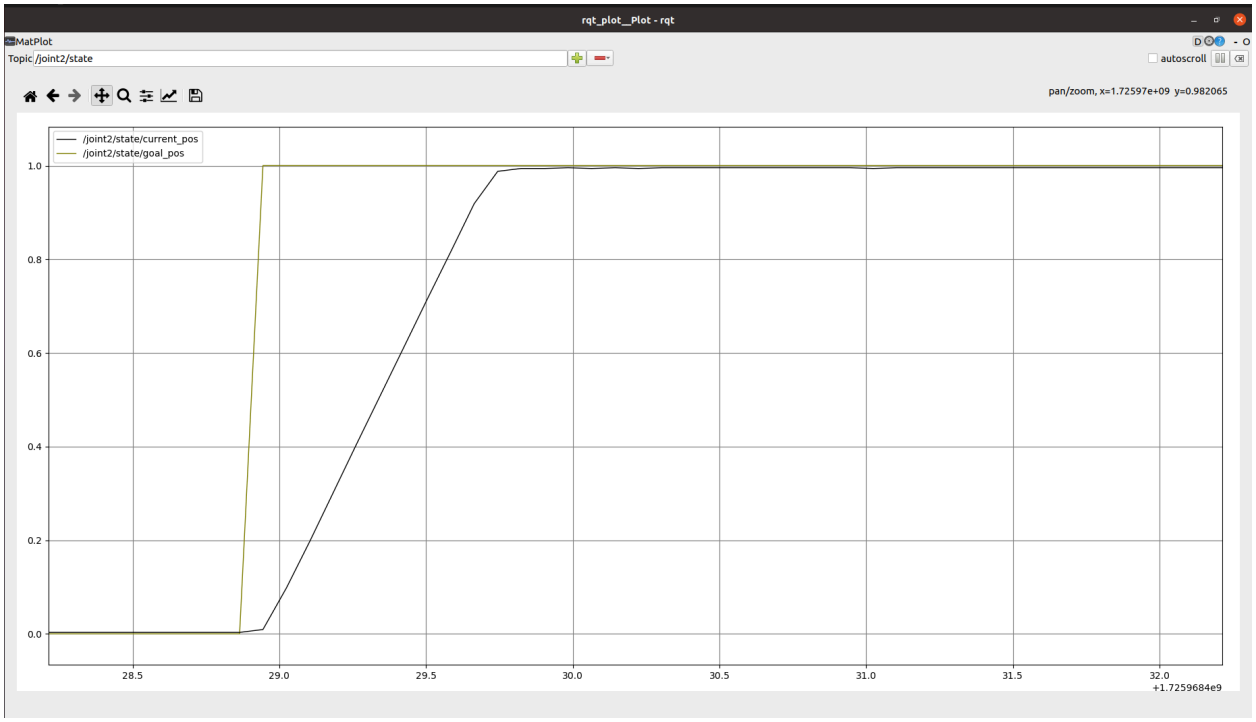

Figure 3. Step Response

As could be seen on the figure above, it takes some time for the motor to achieve the desired position. Technically, this time could be minimized by changing the PID constants, which will lead to the faster response, but to the higher energy consumption.

2. Tasks 3 & 5 – Sine Wave Response, Snake

This task was similar to the previous, but here we were observing the sine wave response of a particular motor. The *ROS* node now publishes commands to five different topics corresponding to the joints or actuators of a Dynamixel motor setup: `/joint2/command`, `/joint4/command`, `/joint6/command`, `/end/command`, and `/motortom2m/command`. The node uses multiple publishers (`pub1` to `pub5`) to send `Float64` messages.

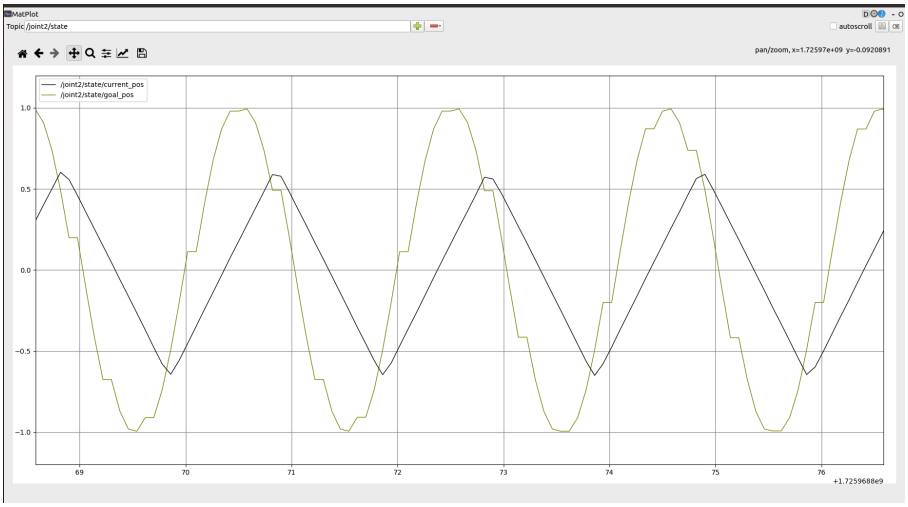

Figure 4. Sin response, 0.5 Hz

The key change in this version is the introduction of a sine wave function to control the motor joints. Two `Float64` message variables (`msg_to_send` and `msg_to_send2`) are used to generate and publish sine wave values. The `msg_to_send` variable sends a positive sine wave value to some of the joints, while `msg_to_send2` sends a negative sine wave value to others, essentially creating synchronized but opposite movements. These values are computed based on the elapsed time since the program started, with an amplitude of 1.0 and a frequency of 0.3 Hz. Initially, we set the frequency to 0.5 Hz, which was too fast for the system to follow the reference (figure 4). Then, we have decreased the frequency to 0.1 Hz, and achieved a more stable following of the goal

position by the motor (Figure 5). For the final demonstration of the snake movement, we have set frequency to 0.3 Hz.

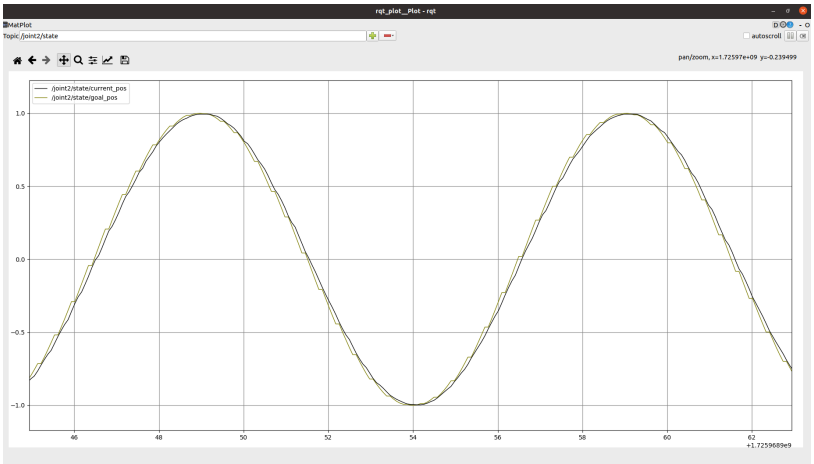

Figure 5. Sin response, 0.1 Hz

```cpp
ros::Publisher pub2 = nh.advertise<std_msgs::Float64>("/joint2/command", 100);
ros::Publisher pub3 = nh.advertise<std_msgs::Float64>("/joint4/command", 100);
ros::Publisher pub4 = nh.advertise<std_msgs::Float64>("/joint6/command", 100);
ros::Publisher pub5 = nh.advertise<std_msgs::Float64>("/end/command", 100);
ros::Publisher pub1 = nh.advertise<std_msgs::Float64>("/motortom2m/command", 100);

ros::Rate loop_rate(10);

ros::Time startTime = ros::Time::now();

while (ros::ok()) {
    // Create a message to publish
    std_msgs::Float64 msg_to_send;
    std_msgs::Float64 msg_to_send2;

    // Calculate elapsed time
    ros::Duration elapsed = ros::Time::now() - startTime;
    double time = elapsed.toSec();

    // Parameters for the sine wave
    double amplitude = 1.0;  // Amplitude of the sine wave
    double frequency = 0.3;  // Frequency of the sine wave in Hz

    // Compute the sine wave value
    msg_to_send.data = amplitude * sin(2 * M_PI * frequency * time);
    msg_to_send2.data = -amplitude * sin(2 * M_PI * frequency * time);

    // Publish the message
    pub1.publish(msg_to_send2);
    pub2.publish(msg_to_send);
    pub3.publish(msg_to_send2);
    pub4.publish(msg_to_send);
    pub5.publish(msg_to_send2);
```

Figure 6. Code for the tasks 3 & 5

Final video demonstration could be found in the github with name *Snake.mp4*

3. Conclusion

In this lab report, we implemented control of a Dynamixel motor through *ROS* nodes, focusing on step and sine wave responses. Adapting code from *Python 2* to *ROS Noetic* involved manually installing *Python 2* and required libraries.

For Tasks 1 and 2, we created a *ROS* node "rotate" to generate a step response. While the motor reached the desired position, adjusting PID constants could reduce response time but increase energy consumption.

Tasks 3 and 5 involved generating sine wave responses across multiple joints, achieving stable movement by adjusting frequency from an initial 0.5 Hz to 0.1 Hz. This demonstrated the need for parameter tuning in multi-joint motor systems.

Overall, this lab exercise provided valuable insights into *ROS*-based motor control. The video "Snake.mp4" demonstrates the effective implementation of these techniques, with room for future improvements in control algorithms and multi-actuator configurations.