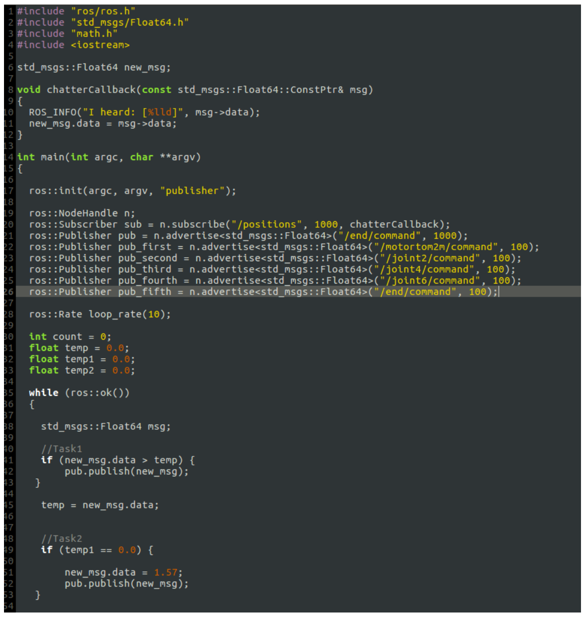
Basics Of Robot Manipulator Control in Joint Space

Moldir Zabirova, Valeriya Kostyukova, Daryn Kenzhebek

I. Task 1: Higher? Publish.

In the first task we are asked to create a node as a .cpp file to publish the command to move a joint if the incoming value is higher than the previous (Fig. 1).



```cpp
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include "math.h"
#include <iostream>

std_msgs::Float64 new_msg;

void chatterCallback(const std_msgs::Float64::ConstPtr& msg)
{
  ROS_INFO("I heard: [%lld]", msg->data);
  new_msg.data = msg->data;
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "publisher");

  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("/positions", 1000, chatterCallback);
  ros::Publisher pub = n.advertise<std_msgs::Float64>("/end/command", 1000);
  ros::Publisher pub_first = n.advertise<std_msgs::Float64>("/motortom2m/command", 100);
  ros::Publisher pub_second = n.advertise<std_msgs::Float64>("/joint2/command", 100);
  ros::Publisher pub_third = n.advertise<std_msgs::Float64>("/joint4/command", 100);
  ros::Publisher pub_fourth = n.advertise<std_msgs::Float64>("/joint6/command", 100);
  ros::Publisher pub_fifth = n.advertise<std_msgs::Float64>("/end/command", 100);

  ros::Rate loop_rate(10);

  int count = 0;
  float temp = 0.0;
  float temp1 = 0.0;
  float temp2 = 0.0;

  while (ros::ok())
  {

    std_msgs::Float64 msg;

    //Task1
    if (new_msg.data > temp) {
        pub.publish(new_msg);
   }

    temp = new_msg.data;


    //Task2
    if (temp1 == 0.0) {

        new_msg.data = 1.57;
        pub.publish(new_msg);
   }

```

Fig. 1. publisher.cpp code

This is done via "if-statement" and a temporary variable that keeps the current pose of a joint. If it is higher than the new value, it publishes it and updates the current position. Else, nothing happens. The values are sent in radians.

II. Task 2: Step Response in RQT.

In this task we published an angle in radians for moving end-effector and base joints. In rqt framework, we visualized this angle as a step response that a joint should reasch from its current position.

In the plot, we can see two graphs. One graph is a step function, or the goal position, and another is response, or joint's actual displacement (Fig. 2, 3).

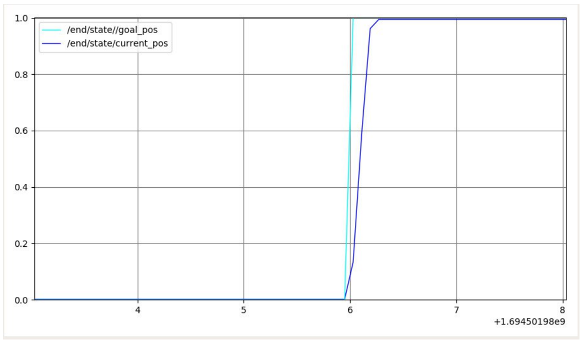


Fig. 2. Step Response for the End-Effector Joint

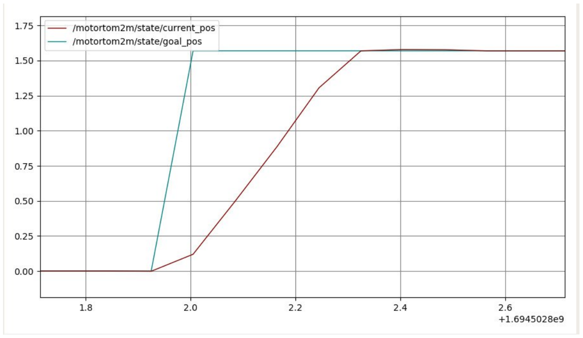


Fig. 3. Step Response for the Base Joint

III. Task 3: Sine-Wave Response.

The objective was to generate sinusoidal motion and publish them to specific ROS topics. We introduced a conditional check on the variable temp2 to ensure that the following code block executes only when temp2 is equal to 0.0. This condition prevents the continuous publication of the same sine wave values.

We obtain the current time in seconds (secs) using the ros::Time::now().toSec() function. Sine waves are generated with an amplitude of 0.2 and a frequency of 0.5 Hz.By changing the frequency we can set how quickly the joints will move back and forth following the sine wave pattern. The sin function is employed for this purpose.

When publishing the sinusoid to a joint, rqt records the current position of that joint in real-time (Fig. 4).

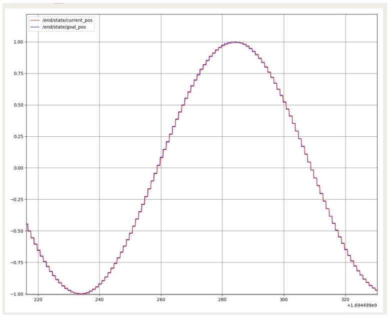
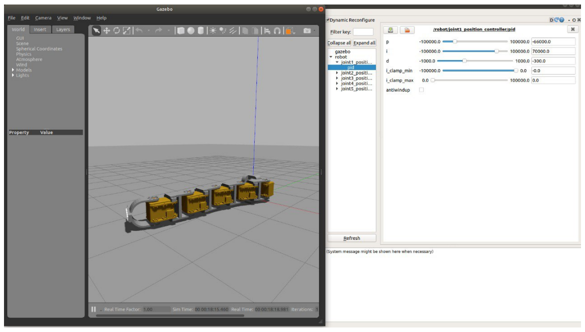


Fig. 4. Sine-Wave Response in rqt

IV. Task 4: PID Control in Gazebo.

GAZEBO is used to simulate the snake robot. It uses new topics /robot/jointX for each joint. We have adapted our publisher.cpp code to publish the sine-wave movements to each joint with a respective phase shift (to create snake-ish movements).

PID control allows to make the movements smoother and can be adjusted via parameters in rqt (Fig. 5). As we changed P value of the PID control, the robot was responding less aggressively to changes in the error signal.

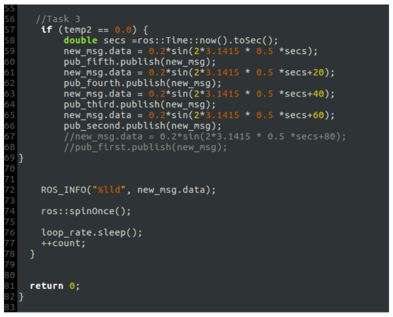


```
/robot/joint1_position_controller/pid
p           -100000.0   100000.0   -66000.0
i           -100000.0   100000.0   70000.0
d           -1000.0     1000.0     -300.0
i_clamp_min -100000.0   0.0        -0.0
i_clamp_max 0.0         100000.0   0.0
antiwindup
```

Fig. 5. Snake Robot Simulation in Gazebo

V. Task 5: Snake Movement.

FINALLY, we have made the robot to move like a snake by publishing sine-wave values to each joint, as we did in the simulator. We changed the rostopics to the corresponding joints with respective phase shifts (Fig. 6).



```
    //Task 3
     if (temp2 == 0.0) {
         double secs =ros::Time::now().toSec();
         new_msg.data = 0.2*sin(2*3.1415 * 0.5 *secs);
         pub_fifth.publish(new_msg);
         new_msg.data = 0.2*sin(2*3.1415 * 0.5 *secs+20);
         pub_fourth.publish(new_msg);
         new_msg.data = 0.2*sin(2*3.1415 * 0.5 *secs+40);
         pub_third.publish(new_msg);
         new_msg.data = 0.2*sin(2*3.1415 * 0.5 *secs+60);
         pub_second.publish(new_msg);
         //new_msg.data = 0.2*sin(2*3.1415 * 0.5 *secs+80);
         //pub_first.publish(new_msg);
}


     ROS_INFO("%lld", new_msg.data);

     ros::spinOnce();

     loop_rate.sleep();
     ++count;
   }


   return 0;
}
```

Fig. 6. Publishing Sine-Wave to All Joints

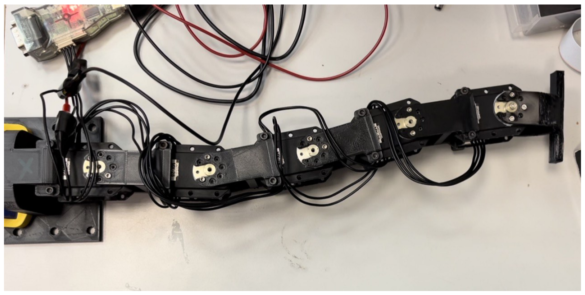


Fig. 7. Snakie Movements

VI. Conclusion

In this lab project, we explored various aspects of robot manipulator control in joint space. We implemented control strategies, visualized responses, and simulated robotic movements. Through these tasks, we gained practical insights into fundamental robotic control concepts and their real-world applications.