Parth Patel, Noopur Koshta, Cameron Schloer

Group Assignment - Part 3

Task 1: Velocity Level Kinematics

As the assignment specified, we created two services for a ROS node, one that would take in joint velocities that would be used to calculate equivalent end-effector velocities, and another that would take in end effector velocities that would be used to calculate equivalent joint velocities.

The topic **/joint_velocity_service** takes in the joint state velocities that will be converted to the equivalent end effector velocities.

The topic **/end_effector_velocity_service** takes in end-effector velocities as received from request to joint state velocities.

```
void set_joint_velocity_from_service(const std::shared_ptr<custom_interfaces::srv::SetJointVelocity::Request> request)
{
  request_joint_velocity = {request->vq1, request->vq2, request->vq3};
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Request (vq1,vq2,vq3): ('%f','%f','%f')", request_joint_velocity[0], request_joint_velocity[1], request_joint_vel
  service_call_for_joint_velocity_control = true;
}

void set_end_effector_velocity_from_service(const std::shared_ptr<custom_interfaces::srv::SetEndEffectorVelocity::Request> request)
{
  request_end_effector_velocity = {request->vx, request->vy, request->vz};
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Request (vx,vy,vz): ('%f','%f','%f')", request_end_effector_velocity[0], request_end_effector_velocity[1], reques
  service_call_for_joint_velocity_control = false;
}
```

Figure 1: Code Snippet for Request Calling the Joint State to End-Effector Velocities and Vice-Versa

Below is part of the code is that handles the conversions of velocity using the Jacobian we derived (the derivation will be shown later in this report).

```
if (service_call_for_joint_velocity_control)
{
  request_end_effector_velocity[0] = -request_joint_velocity[0]*(sin(joint_position[0] + joint_position[1]) + (9.0*sin(joint_position[0]))/10.0) - request_joint_velocity[1]*sin(joint_position[0] + joint_position[1]);
  request_end_effector_velocity[1] = request_joint_velocity[0]*(cos(joint_position[0] + joint_position[1]) + (9.0*cos(joint_position[0]))/10.0) + request_joint_velocity[1]*cos(joint_position[0] + joint_position[1]);
  request_end_effector_velocity[2] = request_joint_velocity[2];

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Requested End Effector Velocity (vx,vy,vz): ('%f','%f','%f')", request_end_effector_velocity[0], request_end_effector_velocity[1], request_end_effector_velocity[2]);
}
else
{
  request_joint_velocity[0] = (10*request_end_effector_velocity[0]*cos(joint_position[0] + joint_position[1]) + 10*request_end_effector_velocity[1]*sin(joint_position[0] +
                               joint_position[1]))/(9*sin(joint_position[1]));
  request_joint_velocity[1] = - (request_end_effector_velocity[0]*(10.0*cos(joint_position[0] + joint_position[1]) + 9.0*cos(joint_position[0])))/(9.0*sin(joint_position[1])) -
                                (request_end_effector_velocity[1]*(10.0*sin(joint_position[0] + joint_position[1]) + 9.0*sin(joint_position[0])))/(9.0*sin(joint_position[1]));
  request_joint_velocity[2] = request_end_effector_velocity[2];

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Actual Joint Velocity (vq1,vq2,vq3): ('%f','%f','%f')", joint_velocity[0], joint_velocity[1], joint_velocity[2]);
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Requested Joint Velocity (vq1,vq2,vq3): ('%f','%f','%f')", request_joint_velocity[0], request_joint_velocity[1], request_joint_velocity[2]);
}
```

Figure 2: Code snippet of the Forward and Inverse Velocity Kinematics Equations for Conversion of Joint Velocities to End-Effector Velocities and Vice-Versa

We derived the Jacobian using a MATLAB script that finds all the homogeneous matrices of the system relative to the base joint, thus giving us the forward kinematics of each joint relative to the base frame. The Jacobian is calculated for the revolute and prismatic joints as follows:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}_{6\times1} = J_{6\times n} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{bmatrix}_{n\times1}$$

Figure 3: Jacobian matrix of all joints

For revolute joint:

Revolute Joints

$$J = \begin{bmatrix} J_v \\ J_w \end{bmatrix} = \begin{bmatrix} R_{i-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times (d_n^0 - d_{i-1}^0) \\ R_{i-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \end{bmatrix}$$

Figure 4: Jacobian matrix for revolute joint

For prismatic joint:

Prismatic Joints

$$J = \begin{bmatrix} J_v \\ J_w \end{bmatrix} = \begin{bmatrix} R_{i-1}^0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix}$$

Figure 5: Jacobian matrix for prismatic joint

We found the rotation matrix and translation vector in the homogenous matrices that were calculated using the DH parameters we found in part 1 of this group assignment. We then took the relevant parts from each of these homogenous matrices, compiled the Jacobian, and then multiplied the Jacobian by all the joint velocities to get the end effector velocities.

To get the joint velocities from the end-effector velocities, we took the inverse of the upper three rows of the Jacobian and multiplied it by the end-effector velocities to yield the joint velocities.

By solving these equations symbolically, we were then able to plug them into our code and solve the reference trajectories for each time step our system is running. This allowed us to start turning our system.

Task 2: Tuning Velocity Controller Gains

The proportional gain (Kp) and derivative gain (Kd) values were obtained by hand-tuning the parameters until we found values that gave the best results.

This tuning was much more difficult than the tuning we did in the second part of this group assignment. This was because of the unruly amount of chattering as the result of the interaction of our controllers, and the fact that velocity can change faster than position.

We tried several things to mitigate this jitter. These attempts at mitigation include the following:

- Increasing the acceptable error band to decrease the frequency of the flipping of the control inputs' respective signs.

- Filtering the error through a low pass filter to try to reduce the intensity of such strong and volatile control inputs. This did decrease the frequency and intensity of the jittering, but the delay it would impose would make our system become unstable anyways.
    - If we had more time we would have tried applying this low pass filter to the control inputs rather than the error to see if this gave a better response, but we figured we would get similar results as filtering the error since the delay would still be present.
- Saturating our control inputs to a parameterizable amount so that massive control inputs couldn't be applied to the robot which would throw the robot violently and sometimes break the robot.

In the end, we had to keep our derivative gains very low or zero to keep the system from exploding. Since these velocities are changing so quickly, the derivative of the error was massive. Using the derivative helped stabilize the system slightly, but it also prevented us from critically damping the system as well as we could have with a system with no jitter.

After much effort, we found the following gains for each joint:

Joint 1 Gains:
- Kp = 50
- Kd = 0.15

Joint 2 Gains:
- Kp = 20
- Kd = 0

Joint 3 Gains:
- Kp = 10
- Kd = 0

Because this is a PD controller and not a PID controller, there was always a little bit of steady-state error in the third joint since gravity was a big force acting on it. The only way we could completely remove the steady state error would be to make the gains very large, but we found that it caused our system to become unstable and go into a violent feedback loop.

Task 3: Apply Constant End Effector Velocity

Our detailed instructions to run this portion of the assignment are in the third section of the README about velocity control:
(https://github.com/parth-20-07/RBE-500-Final-Project/blob/main/README.md)

Our controller is run with the following command:

- `ros2 run rrbot_gazebo end_eff_vel_control`

Normally we would need to run these commands after getting gazebo up, but we run them in the main function of our controller node. So these are run automatically by our system:

- `ros2 run rrbot_gazebo switch`

- `ros2 topic pub --once /forward_effort_controller/commands std_msgs/msg/Float64MultiArray 'data: [0,0,0]'`

We then make a service call to the Set End Effector Velocity service using:

- `ros2 service call /end_effector_velocity_service custom_interfaces/srv/SetEndEffectorVelocity "{vx: 0, vy: 0.2, vz: 0}"`

As specified in the problem, we commanded only a positive movement in the end effector in the y direction relative to the base frame. Any values would work, but it can only be run for a short time before the arm reaches its full length and, in the case of the first two joints, they reach a singularity and instability occurs.

We visualized the reference and actual end effector velocity, as well as the joint velocities with the calculated reference joint velocities calculated at each time stamp using our Jacobian Matrix. These results are shown below in Figures 6 and 7 below. As shown in the command above, the reference value provided was 0.2m/s in the positive ‘y’ direction.

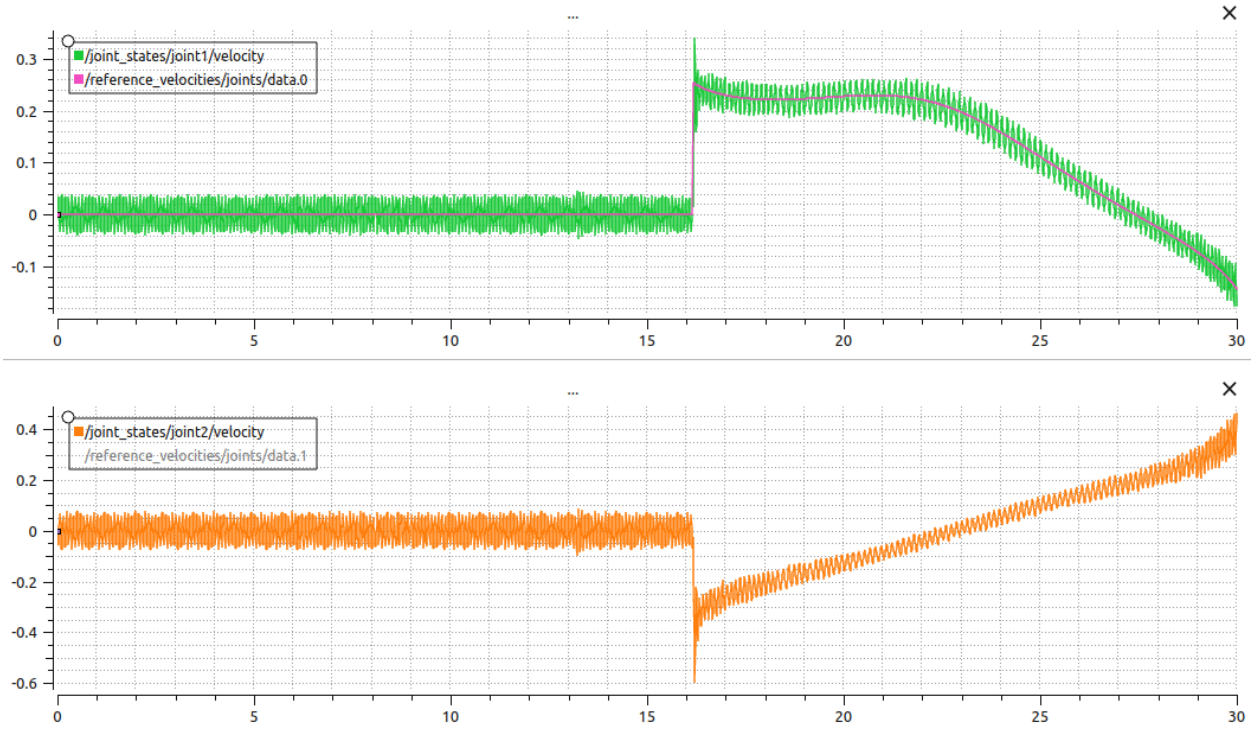

Figure 6: Plotted Data of the Calculated Reference Velocities for Each Joint and the Actual Velocities that correspond to each. Since the Prismatic Joint was not Being Actuated, it was not Plotted

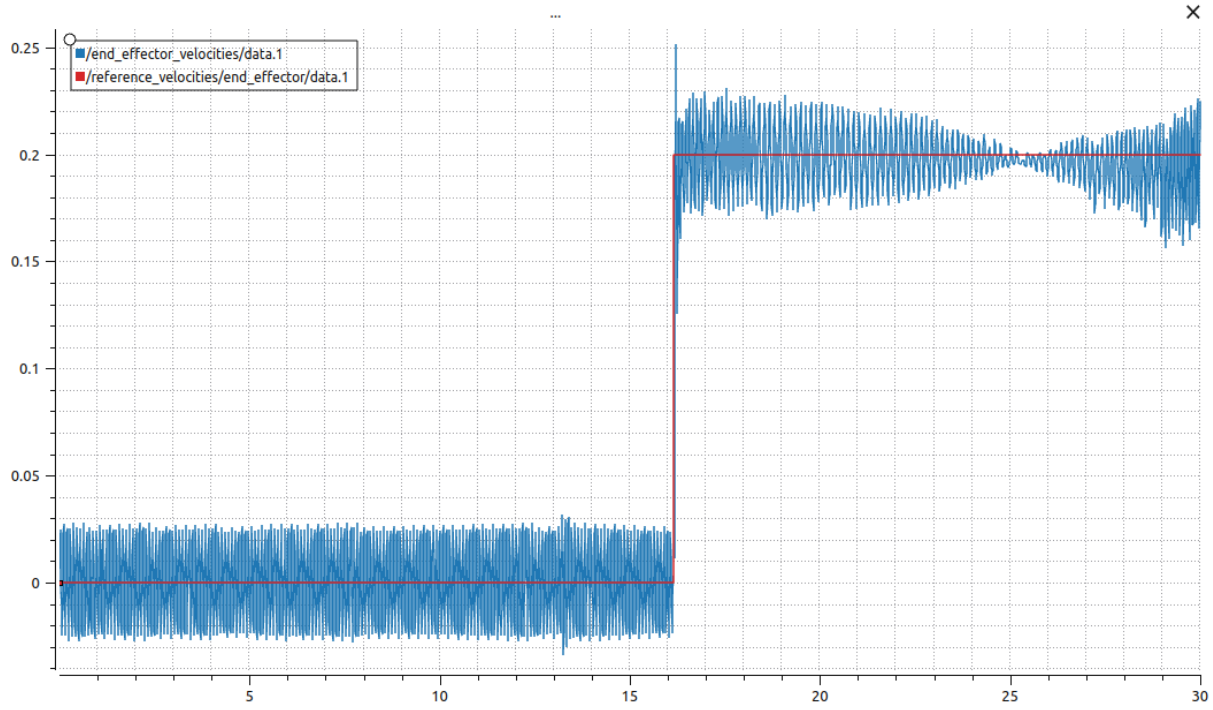

Figure 7: Reference and Actual End Effector Velocities

Although the output is very noisy due to the interactions of the joints with each other, and the volatility of the velocity, the movement is very smooth to the naked eye. This can be seen in the video of the movement of the arm during the time of this recording. The results were not optimal, but the movement was satisfactory.