RBE 500 Group Assignment #3

Joshua Gross, Arjan Gupta, Melissa Kelly

Velocity Kinematics

Calculations

Solving for both the forward and inverse velocity kinematics are centered around solving the Jacobian matrix. In order to solve for the Jacobian, we first need to establish the linear and angular velocity components in all directions for each of the joints. These expressions will vary dependent upon whether the joint is revolute or prismatic. Once these determinations are made, a matrix of the form seem in the figure below is created, where, each of the elements is a 3×1 matrix. This matrix comes from the lecture slides.

$$J = \begin{bmatrix} z_0 \times (o_3 - o_0) & z_1 \times (o_3 - o_1) & z_2 \\ z_0 & z_1 & 0 \end{bmatrix}$$

As clearly seen in the above figure, the following elements are needed in order to solve the Jacobian in its entirety: $z_0, z_1, z_2, o_0, o_1, o_3$. These values are found by using elements of the individual transformation matrices found when solving the forward position kinematics. The image below shows what elements of the transformation matrix are extracted in order to solve for Jacobian.

ROS Code

Programmatically, the elements $z_0, z_1, z_2, o_0, o_1, o_3$ mentioned in the Calculations subsection above are extracted and plugged into the Jacobian framework as seen below from the `velocity_kinematics.py` file (which is submitted as part of the `velocity_kin` package).

```
1          # Get z's and o's for Jacobian
2          z0 = np.array([[0], [0], [1]])
3          z1 = A1[:3, 2:3]
4          z2 = A2[:3, 2:3]
5          O0 = np.array([[0], [0], [0]])
6          O1 = A1[:3, 3:4]
7          O3 = A3[:3, 3:4]
8          # Write columns of Jacobian
9          E11 = np.cross(z0, (O3-O0), axis=0)
10         E12 = np.cross(z1, (O3-O1), axis=0)
11         E13 = z2
12         E21 = z0
13         E22 = z1
14         E23 = np.array([[0], [0], [0]])
15         # Build up the Jacobian
16         Jacobian = np.zeros((6,3), dtype=float)
17         Jacobian[:3, :1] = E11
18         Jacobian[:3, 1:2] = E12
19         Jacobian[:3, 2:3] = E13
20         Jacobian[3:6, :1] = E21
21         Jacobian[3:6, 1:2] = E22
22         Jacobian[3:6, 2:3] = E23
```

At this point, we have the ability to solve for the forward or inverse velocity kinematics. By multiplying the Jacobian by the joint's velocities matrix, we can solve for the velocity of the end effector. Likewise, if the inverse of the Jacobian is multiplied by the end effector velocity matrix, the velocity of each of the joints can be found. It should be noted that since this particular Jacobian is not square, the inverse cannot simply be taken and therefore the pseudo-inverse numpy function is used in order to take the inverse. Having the Jacobian solved and regularly available allows us to calculate the forward and inverse velocity kinematics and therefore create a controller for the robot so achieve the desired movement and speed.

Velocity Controller

Mathematical Modeling

The process of creating a controller for this application stems from the previous position controller as well as other controllers created throughout the course. The goal velocity of each joint, V_r is is received through a service response. The name of the service is `velocity_inv_kin_service`. Suppose our current velocity is given as V. The error for each joint is then calculated as follows:

$$E = V_r - V$$

When paired with the proportional gain value, we now have one of the terms of the controller. In order to get the other term, the derivative of the error is also needed, and is calculated as shown:

$$\dot{E} = \frac{V - V_{prev}}{\Delta time}$$

This derivative of the error is paired with the derivate gain and now completes the controller. Our controller takers on the form:

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

Where F is the effort that will be applied in Gazebo and will cause the joints to move.

Creating the Controller in ROS

Creating the velocity controller also followed a similar process to the position controller.

In the `scara_velocity_controller.py` node, we first initialize many different variables that will be used throughout the various functions. This is where the reference velocities are set based on the Jacobian and input end effector velocity. Additionally, this is where we set the proportional and derivative gain values to be used by the velocity controller later. Once these variables are set, we initialize member variables for the clients. Next, the empty arrays to be filled with velocity data are created to be later be filled in and then plotted in the graphing section of the code. At this point, we create the client that will activate the required controller. Another client is created that will get the joint velocity references from the other ROS node and the service that receives the reference velocity is also created. Then subscribers are made that call the joint state callbacks.

Once the `run_controller` function is called, the PD controller us activated therefore controlling the speed of each joint and ideally achieving the desired behavior of the robot. This function takes all of the velocities

of the individual joints, applies the proportional and derivate gain values, and computes the output efforts to be sent back to Gazebo therefore causing movement. In order to implement the controller, we begin by finding the errors, or the difference between the reference velocity and the true velocity value from the simulation. From here, the acceleration is approximated by using the derivative approximation technique as shown in lectures. These steps can be seen in the code below.

```
1          # ----- Implement controller ------
2          # Find errors
3          err_v1 = self.ref_v1 - v1
4          err_v2 = (self.ref_v2 * -1) - v2
5          err_v3 = self.ref_v3 - v3
6          # Approximate the acceleration by using the derivative
7          # approximation technique shown in Week 8 lecture
8          accel1 = (v1 - self.last_v1)/(time.time() - self.accel_approx_last_time)
9          accel2 = (v2 - self.last_v2)/(time.time() - self.accel_approx_last_time)
10         accel3 = (v3 - self.last_v3)/(time.time() - self.accel_approx_last_time)
```

The next step is to tune our controller.

Tuning the Controller

In order to tune the velocity controller, we followed a similar process as the previous assignment. We began with arbitrary proportional and derivative gain values and then ran the simulation to observe the robot's behavior. Once we observed how the robot was behaving, we would adjust the gain values based on the table below found in the lecture slides.

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
$\mathbf{K_p}$	Decrease	Increase	Small change	Decrease	Degrade
$\mathbf{K_d}$	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Table 1: This table helps tuning the gain values. Behavior described herein is for increasing the parameters.

After tuning, the robot achieved the desired behavior and the gain values were finalized. Using the gain values in the table below, the robot moved in a straight line along the +Y axis and achieved the reference velocities.

Joint	$\mathbf{K_p}$	$\mathbf{K_d}$
Joint #1	0.35	0.3
Joint #2	-0.35	-0.15

Table 2: Our resultant gain values after tuning.

Graphing

Similar to the previous assignment, we graph the velocity of joints 1 and 2 against time. Every time that the controller function is called, the `dump_graph_data` function is also called. Here, an empty array is appended with the latest velocity value being spit out to the robot. Additionally, an iterator, timer, and graph flag is created. These allow us to keep track of the time between each data point as well as making sure that

we only are sampling the controller for a designated 10 second period. The graph flag lets us know that the plots have not yet been generated so that we do not run into problems with already having a plot open or creating a new plot at an inappropriate time. Once the 10 second sampling time is complete, we create 2 subplots and plot each joint's collected velocities against time. After the code runs, it produces the graph shown below.