Report: Velocity-Level Kinematics and Robot Linear Path Movement Using ROS 2 on Linux

---

Objective

This assignment focuses on moving a robot in a straight line along the an axis in Cartesian space using velocity-level kinematics. The execution files have been divided in 3 python files and several call functions.

1. Vel_kinematics : This file does the computation of twist, Jacobian and velocities
2. Vel_calc : This file has function which gets joint velocities calculated by vel_kinematics, and calculates the new joint position and passes it to the robotic arm.
3. Vel_pub : This file has function which receives the values of twists provided by the user which is published to the service in vel_kinematics for joint velocity calculations at every instance

1. Velocity-Level Kinematics

Task Overview

Develop a ROS 2 node to handle velocity calculation:

1. **Convert Joint Velocities to End-Effector Velocities**: Compute Cartesian velocity of the robot's end-effector using its Jacobian.
2. **Convert End-Effector Velocities to Joint Velocities**: Use the Jacobian's pseudoinverse for inverse velocity mapping.

Node Implementation: EndeffecVel

This node is implemented in ROS2

1. **Key Functionalities**:
    - **Jacobian Calculation**:
    - Subscribes to the Fwd_op to receive and pose matrices of each joint frames.
    - Constructs the Jacobian matrix:
        - Linear velocities (VV): Computed as the cross product of joint axes and position differences.
        - Angular velocities (WW): Derived directly from joint orientations.

```
def calJacobian(self, msg):
    try:
        Rot = np.array(msg.r)
        Pos = np.array(msg.o)

        R = np.zeros((4, 3, 3))
        O = np.zeros((5, 3, 1))
        idx = 0

        for i in range(4):
            for j in range(3):
                for k in range(3):
                    R[i][j][k] = Rot[idx]
                    idx += 1

        idx_2 = 0
        for i in range(5):
            for j in range(3):
                O[i][j][0] = Pos[idx_2]
                idx_2 += 1

        k = np.array([[0], [0], [1]])
        v = [np.cross((np.dot(R[i], k)).T, ((O[4] - O[i])).T).T for i in range(4)]
        w = [np.dot(R[i], k) for i in range(4)]

        V = np.hstack(v)
        W = np.hstack(w)
        self.J = np.vstack((V, W))

        self.get_logger().info(f'Calculated Jacobian: \n{self.J}')
    except Exception as e:
        self.get_logger().error(f"Error calculating Jacobian: {e}")
```

- **Services**:
    - calculate_twist: Takes joint velocities as input and computes end-effector velocity;

```
def calculate_twist(self, request, response):
    if self.J is None or self.J.size == 0:
        self.get_logger().error("Jacobian is not yet calculated.")
        return response

    try:
        Jointvel = np.array(request.joint_velocity).reshape(-1, 1)
        twist = np.dot(self.J, Jointvel)
        response.twist = twist.flatten().tolist()
        self.get_logger().info(f'Calculated twist: {response.twist}')
    except Exception as e:
        self.get_logger().error(f"Error calculating twist: {e}")
    return response
```

    - calculate_jointvel: Takes Cartesian velocities as input and computes joint velocities

```
def calculate_jointvel(self, request, response):
    if self.J is None or self.J.size == 0:
        self.get_logger().error("Jacobian is not initialized or invalid.")
        return response

    try:
        Tw = request.endeffector_velocity
        twist = np.array([Tw.linear.x, Tw.linear.y, Tw.linear.z, Tw.angular.x, Tw.angular.y, Tw.angular.z]).reshape(-1, 1)
        self.get_logger().info(f'Twist input: {twist}')
        J_inv = np.linalg.pinv(self.J)
        #J_inv = J_inv.T
        self.get_logger().info(f'Jinv = {J_inv}')
        jointvel = np.dot(J_inv, twist)
        self.JV = jointvel.flatten().tolist()
        response.jointvel = self.JV
        self.get_logger().info(f'Calculated joint velocities: {self.JV}')
    except Exception as e:
        self.get_logger().error(f"Error calculating joint velocities: {e}")
    return response
```

2. **Execution**:
    - Start the environment:
    - For giving Joint vel
        - ros2 service call /calculate_twist my_service_package/srv/JointVelservice "{joint_velocity: [1.0, 1.0, 1.0, 1.0]}"
    - For giving Twist
        - ros2 service call /calculate_jointvel my_service_package/srv/EndeffectorVelservice "{endeffector_velocity: {linear: {x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.0}}}"
    - Jacobian: Computed when data is published to Fwd_op.
    - Service calls: Tested with simulated velocity inputs.

---

2. Incremental Position Reference Node

Task Overview (3.5 points)

Node Implementation

The node subscribes to node to receive joint velocities and incrementally computes joint positions.

1. **Execution**:
    - Start the node:
    - for publishing const velocity
        - ros2 topic pub /ConstEndvel geometry_msgs/msg/Twist "{linear: {x: 0.0, y: 10.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
2. **Functional Flow**:
    - Sampling time is set to 1 seconds.
    - Joint reference positions are updated iteratively.

```
while count < 1000 and rclpy.ok():

    if self.current_joint_states is not None and flag:
        old_joint_state = self.current_joint_states
        flag = False

    self.get_logger().info(f"Iteration {count + 1}")

    # Process incoming messages to ensure callbacks are invoked
    rclpy.spin_once(self, timeout_sec=0.1)

    # Ensure joint velocities are received
    if not self.joint_vel_received or self.latest_joint_velocities is None:
        self.get_logger().warn("No joint velocities received yet.")
        continue

    # Ensure joint states are received
    if self.joint_state_received and self.current_joint_states is not None and self.current_joint_states.size > 0:
        self.get_logger().info(f"Current joint states: {self.current_joint_states}")
        self.get_logger().info(f"latest_joint_velocities: {self.latest_joint_velocities}")

        for i in range(len(self.latest_joint_velocities)):
            vel = self.latest_joint_velocities[i]
            self.theta[i] = old_joint_state[i] + vel*self.time #+ e
            old_joint_state[i] = self.theta[i]
            self.theta[i] = round(self.theta[i], 3)
            self.get_logger().info(f"Updated Theta {i}: {self.theta[i]}")

        # differences = [abs(target - current) for target, current in zip(self.theta, old_theta)]
        # self.get_logger().info(f"Difference = {differences}")
        # if any(diff > 0.0001 for diff in differences):
        self.send_request(self.theta)
        self.get_logger().info("Calling wait_until_position_reached()...")
        self.wait_until_position_reached()

        count += 1

    else:
        self.get_logger().warn("Joint states are not received or invalid.")
```

3. Linear Path Motion

Task Overview (1.5 points)

Provided a constant twist for the end effector frame and compute joint velocities. These velocities are fed as input to the incremental position reference calculations to move the robot along a straight line.

Node Implementation:

The Velpub node subscribes to ConstEndvel topic to get a constant twist input and passes the value to calculate_jointvel service.

1. **Execution**:
    - Start the node:
    - ros2 topic pub /ConstEndvel geometry_msgs/msg/Twist "{linear: {x: 0.0, y: 10.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
2. **Integration**:
    - Joint velocities from calculate jointvel are forwarded to the incremental position reference calculations using the topic JV_info topic.

```
def sendconstVel(self, msg):
    # Detailed logging of incoming message
    self.get_logger().info(f'Received Twist message: {msg}')

    # Prepare service request
    linear_velocity = Vector3()
    linear_velocity.x = msg.linear.x
    linear_velocity.y = msg.linear.y
    linear_velocity.z = msg.linear.z

    angular_velocity = Vector3()
    angular_velocity.x = msg.angular.x
    angular_velocity.y = msg.angular.y
    angular_velocity.z = msg.angular.z

    request = EndeffectorVelservice.Request()
    request.endeffector_velocity.linear = linear_velocity
    request.endeffector_velocity.angular = angular_velocity

    # Extensive pre-service call checks
    if not self.client.service_is_ready():
        self.get_logger().error('Service client is not ready')
        return

    # Call the service asynchronously with comprehensive error handling
    future = self.client.call_async(request)

    # More detailed future handling
    def callback(future):
        try:
            response = future.result()
            if response is not None:
                self.get_logger().info(f"Joint Velocities: {response.jointvel}")
            else:
                self.get_logger().error("Service call returned None")
        except Exception as e:
            self.get_logger().error(f"Service call exception: {e}")
```

4. Results

Trajectory Plot

- The Cartesian position was logged and plotted using matplotlib.
- The graph has some error in position as there are error in mathematical calculation by pseudo-inverse of the Jacobian
- This leads to some error in the motion of robot planes.