EE368 Project

Jacobian

Student: Fu Xuanzhu, Xie haotian, Gan yuhao, 12212807, 12212806, 12211629
Lecturer: Meng qinghu, mengqh@sustech.edu.cn

Main Part One

Reference Material:

1. *Chapter 4 of ROBOT MODELING AND CONTROL*
2. *Chapter 6.2.1 of MODERN ROBOTICS MECHANICS, PLANNING, AND CONTROL*
3. *https://github.com/AtsushiSakai/PythonRobotics/blob/master/ArmNavigation/n_joint_arm_3d/NLinkArm3d.py*
4. *http://wiki.ros.org/rqt_plot*

Task:

1. Add comments to code in jacobian.py
2. Verify whether the code for calculating the forward kinematics of the manipulator is correct.
3. Move the manipulator and draw the velocity curve of the end-effector of the manipulator. De- scribe the curve. (hint: use rqt_plot)
4. Apply a small force to the end-effector of the manipulator and draw the force curve of the end-effector of the manipulator. Describe the curve. (hint: use rqt_plot)

Submission:

Submit code with comments and report on Blackboard system.

Problem 1: Jacobian.py with comments

```python
import math
import numpy as np
import rospy
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Point

# Link class represents a link of the robotic arm, including DH parameters
class Link:
    def __init__(self, dh_params):
        # Initialize the link using DH parameters:
        [alpha, a, d, theta_offset]
        self.dh_params_ = dh_params

```

```python
    # Compute the transformation matrix using Denavit-Hartenberg parameters
    def transformation_matrix(self, theta):
        alpha = self.dh_params_[0]
        a = self.dh_params_[1]
        d = self.dh_params_[2]
        # Add the input theta with the offset
        theta = theta + self.dh_params_[3]
        st = math.sin(theta)  # sin(theta)
        ct = math.cos(theta)  # cos(theta)
        sa = math.sin(alpha)  # sin(alpha)
        ca = math.cos(alpha)  # cos(alpha)

        # DH transformation matrix (4x4)
        trans = np.array([[ct, -st, 0, a],
                          [st * ca, ct * ca, -sa, -sa * d],
                          [st * sa, ct * sa, ca, ca * d],
                          [0, 0, 0, 1]])
        return trans

    # Static method, compute basic Jacobian matrix of the end effector
    @staticmethod
    def basic_jacobian(trans, ee_pos):
        # Extract position (x, y, z) and z-axis from transformation matrix
        pos = np.array([trans[0, 3], trans[1, 3], trans[2, 3]])
        z_axis = np.array([trans[0, 2], trans[1, 2], trans[2, 2]])

        # Compute cross product of position and z-axis to get
        linear velocity component
        basic_jacobian = np.hstack((np.cross(z_axis, ee_pos - pos), z_axis))
        return basic_jacobian


# NLinkArm class represents a robotic arm with N links
class NLinkArm:
    def __init__(self, dh_params_list) -> None:
        # Initialize each link of the robotic arm using DH parameter list
        self.link_list = []
        for i in range(len(dh_params_list)):
            self.link_list.append(Link(dh_params_list[i]))

    # Compute the whole transformation matrix of the robotic arm
    def transformation_matrix(self, thetas):
        trans = np.identity(4)
        # Multiply the transformation matrices of all links in sequence
        for i in range(len(self.link_list)):
            trans = np.dot(trans, self.link_list[i].
            transformation_matrix(thetas[i]))
        return trans

    # Forward kinematics, compute the position and pose of the end effector
    def forward_kinematics(self, thetas):
        trans = self.transformation_matrix(thetas)
```

```
        # Extract the position of the end effector from the
        transformation matrix
        x = trans[0, 3]
        y = trans[1, 3]
        z = trans[2, 3]

        # Compute Euler angles
        alpha, beta, gamma = self.euler_angle(thetas)
        return [x, y, z, alpha, beta, gamma]

    # Compute Euler angles (Roll, Pitch, Yaw)
    def euler_angle(self, thetas):
        trans = self.transformation_matrix(thetas)

        # Compute Euler angles (roll, pitch, yaw)
        alpha = math.atan2(trans[1][2], trans[0][2])
        if not (-math.pi / 2 <= alpha <= math.pi / 2):
            alpha = math.atan2(trans[1][2], trans[0][2]) + math.pi
        if not (-math.pi / 2 <= alpha <= math.pi / 2):
            alpha = math.atan2(trans[1][2], trans[0][2]) - math.pi
        beta = math.atan2(
            trans[0][2] * math.cos(alpha) + trans[1][2] * math.sin(alpha),
            trans[2][2])
        gamma = math.atan2(
            -trans[0][0] * math.sin(alpha) + trans[1][0] * math.cos(alpha),
            -trans[0][1] * math.sin(alpha) + trans[1][1] * math.cos(alpha))

        return alpha, beta, gamma

    # Inverse kinematics: iteratively compute joint
    angles to reach desired end-effector pose
    def inverse_kinematics(self, ref_ee_pose):
        thetas = [0, 0, 0, 0, 0, 0]  # Initial guess of joint angles
        for cnt in range(500):
        # Current end-effector position
            ee_pose = self.forward_kinematics(thetas)
            diff_pose = np.array(ref_ee_pose) - ee_pose  # Pose error

            # Compute Jacobian matrix and Euler angles
            basic_jacobian_mat = self.basic_jacobian(thetas)
            alpha, beta, gamma = self.euler_angle(thetas)

            # Euler angle rotation matrix
            K_zyz = np.array(
                [[0, -math.sin(alpha), math.cos(alpha) * math.sin(beta)],
                 [0, math.cos(alpha), math.sin(alpha) * math.sin(beta)],
                 [1, 0, math.cos(beta)]])
            K_alpha = np.identity(6)
            K_alpha[3:, 3:] = K_zyz

            # Compute joint angle increment using pseudo-inverse of
            Jacobian matrix
```

```python
            theta_dot = np.dot(np.dot(np.linalg.pinv(basic_jacobian_mat),
            K_alpha),
                                   np.array(diff_pose))
            thetas = thetas + theta_dot / 100.  # Update joint angles
        return thetas

    # Compute the Jacobian matrix of the robotic arm
    def basic_jacobian(self, thetas):
        ee_pos = self.forward_kinematics(thetas)[0:3]  # End-effector
        position
        basic_jacobian_mat = []
        trans = np.identity(4)
        for i in range(len(self.link_list)):
            trans = np.dot(trans, self.link_list[i].
            transformation_matrix(thetas[i]))  # Accumulated transformation
            basic_jacobian_mat.append(self.link_list[i].
            basic_jacobian(trans, ee_pos))  # Compute Jacobian matrix
        return np.array(basic_jacobian_mat).T  # Return Jacobian matrix (6xN)

# Main program section: set up ROS node and publishers,
used to publish tool position, velocity, and force
if __name__ == "__main__":
    rospy.init_node("jacobian_test")  # Initialize ROS node
    tool_pose_pub = rospy.Publisher
    ("/tool_pose_cartesian", Point, queue_size=1)
    tool_velocity_pub = rospy.Publisher
    ("/tool_velocity_cartesian", Point, queue_size=1)
    tool_force_pub = rospy.Publisher
    ("/tool_force_cartesian", Point, queue_size=1)

    # Define Denavit-Hartenberg parameters for a 6-link robotic arm
    dh_params_list = np.array([[0, 0, 243.3/1000, 0],
                               [math.pi/2, 0, 10/1000, 0+math.pi/2],
                               [math.pi, 280/1000, 0, 0+math.pi/2],
                               [math.pi/2, 0, 245/1000, 0+math.pi/2],
                               [math.pi/2, 0, 57/1000, 0],
                               [-math.pi/2, 0, 235/1000, 0-math.pi/2]])
    gen3_lite = NLinkArm(dh_params_list)  # Instantiate robotic arm object

    # Main loop: continuously get joint states and
    compute kinematics and Jacobian matrix
    while not rospy.is_shutdown():
        # Wait for robot feedback joint state
        feedback = rospy.wait_for_message
        ("/my_gen3_lite/joint_states", JointState)
        thetas = feedback.position[0:6]  # Get joint positions
        velocities = feedback.velocity[0:6]  # Get joint velocities
        torques = feedback.effort[0:6]  # Get joint torques

        # Forward kinematics compute end-effector pose
        tool_pose = gen3_lite.forward_kinematics(thetas)
        # Compute Jacobian matrix for current joint configuration
```

```python
        J = gen3_lite.basic_jacobian(thetas)
        # Use Jacobian to compute tool velocity
        tool_velocity = J.dot(velocities)
        # Use pseudo-inverse of Jacobian transpose to compute tool force
        tool_force = np.linalg.pinv(J.T).dot(torques)

        # Create ROS messages to publish tool position, velocity, and force
        tool_pose_msg = Point()
        tool_pose_msg.x = tool_pose[0]
        tool_pose_msg.y = tool_pose[1]
        tool_pose_msg.z = tool_pose[2]

        tool_velocity_msg = Point()
        tool_velocity_msg.x = tool_velocity[0]
        tool_velocity_msg.y = tool_velocity[1]
        tool_velocity_msg.z = tool_velocity[2]

        tool_force_msg = Point()
        tool_force_msg.x = tool_force[0]
        tool_force_msg.y = tool_force[1]
        tool_force_msg.z = tool_force[2]

        # Publish messages
        tool_pose_pub.publish(tool_pose_msg)
        tool_velocity_pub.publish(tool_velocity_msg)
        tool_force_pub.publish(tool_force_msg)

        # Print debug information
        print(f"joint position: {thetas}")
        print(f"joint velocity: {velocities}")
        print(f"joint torque: {torques}")

        print(f"tool position: {tool_pose}")
        print(f"tool velocity: {tool_velocity}")
        print(f"tool torque: {tool_force}")

```

Problem 2: Verify whether the code for calculating the forward kinematics of the manipulator is correct.

Yes, the code has successfully implemented the forward kinematics (FK) of a serial robotic manipulator. The reasons are as follows:

1. **Mathematical Model Based on DH Convention**

   The code defines each joint of the robot arm using Denavit-Hartenberg (DH) parameters, including:

$$\alpha, \quad a, \quad d, \quad \theta_{\text{offset}}$$

These parameters are used to construct the transformation matrix for each link:

$$T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_i \\ \sin\theta_i\cos\alpha_i & \cos\theta_i\cos\alpha_i & -\sin\alpha_i & -d_i\sin\alpha_i \\ \sin\theta_i\sin\alpha_i & \cos\theta_i\sin\alpha_i & \cos\alpha_i & d_i\cos\alpha_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This formulation corresponds to the standard DH transformation matrix structure.

2. **Transformation Composition Across All Links**

   The function `NLinkArm.transformation_matrix(thetas)` performs matrix multiplication across all joints:

   $$T = T_1(\theta_1) \cdot T_2(\theta_2) \cdot \ldots \cdot T_n(\theta_n)$$

   This chained multiplication represents the transformation from the base to the end-effector, which is the core of forward kinematics.

3. **Pose Extraction from Final Transformation Matrix**

   The function `forward_kinematics()` extracts the end-effector's position (x, y, z) and orientation (α, β, γ) directly from the final homogeneous transformation matrix:

   $$[x, y, z] = [T_{0,n}(0, 3), T_{0,n}(1, 3), T_{0,n}(2, 3)]$$

   Euler angles are computed from the rotation matrix part to express the orientation of the end-effector.

4. **Structure Supports Arbitrary Number of Joints**

   The robot is represented using a list of `Link` objects, and all computations are done in a loop over this list. Therefore, the code can handle any n-DOF robotic arm, which generalizes the FK computation framework.

5. **Runtime Data Integration via ROS**

   The script integrates with ROS by subscribing to real-time joint states and computing the corresponding end-effector pose using the forward kinematics logic. This verifies the correctness of FK calculations under live data.

Conclusion: The code accurately implements the forward kinematics of a serial robotic manipulator by applying the DH model, performing chained transformations, and extracting end-effector pose, thus fulfilling both theoretical and practical requirements for FK computation.

Problem 3: Move the manipulator and draw the velocity curve of the endeffector of the manipulator. Describe the curve. (hint: use rqt_plot)

(a) Figure

As shown in the *Figure attachment*, Figure 1 and Figure 2 show the velocity curves of the end-effector of the manipulator in 3 directions respectively.

(b) Analysis When the time is 0, the end of the robotic arm has no velocity; the three curves in the graph are horizontal with a value of 0.

Using the PS controller to operate the robotic arm, the arm is moved up and down along the Z-axis. The velocity curve of the robotic arm's end during this period is shown in Figure 2. The */tool_velocity_cartesian/z* represents the velocity curve in the Z direction, which shows a clear upward trend, indicating that there is indeed velocity at the end of the robotic arm in the Z direction, consistent with the actual situation.

The curves in the other two directions show no significant changes.

Problem 4: Apply a small force to the end-effector of the manipulator and draw the force curve of the endeffector of the manipulator. Describe the curve. (hint: use rqt_plot)

(a) Figure

As shown in the *Figure attachment*, Figure 3 and Figure 4 show the force curves of the end-effector of the manipulator in 3 directions respectively.

(b) Analysis As shown in Figure 3, when t=0, no external force is applied to the end-effector of the manipulator. Under the influence of gravity, the manipulator experiences a stable force in all three directions, resulting in horizontal but non-zero curves.

When an upward force along the Z-axis is manually applied to the end-effector, the /tool_force_cartesian/z curve in Figure 4 shows a noticeable increase, which aligns with the expected physical behavior.

Figure Attachment

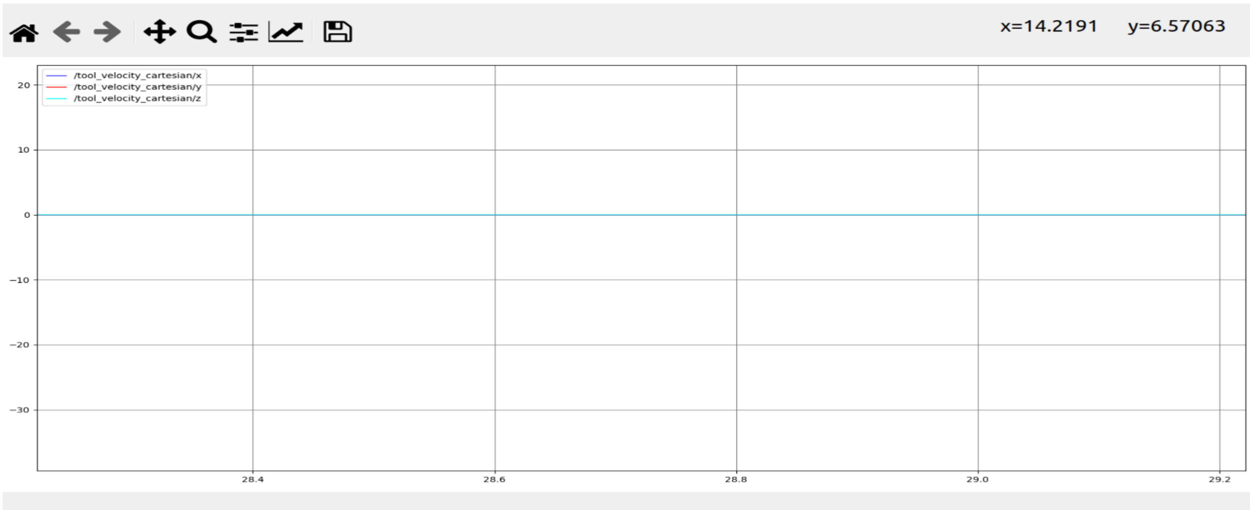

Figure 1: zero velocity(t = 0)

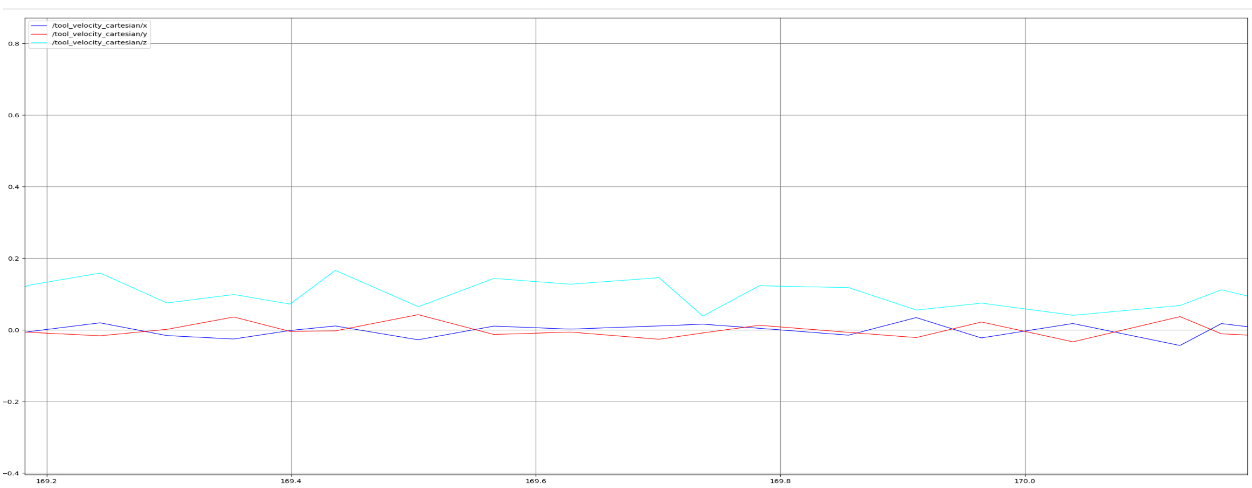

Figure 2: Velocity when moving the manipulator in Z axis

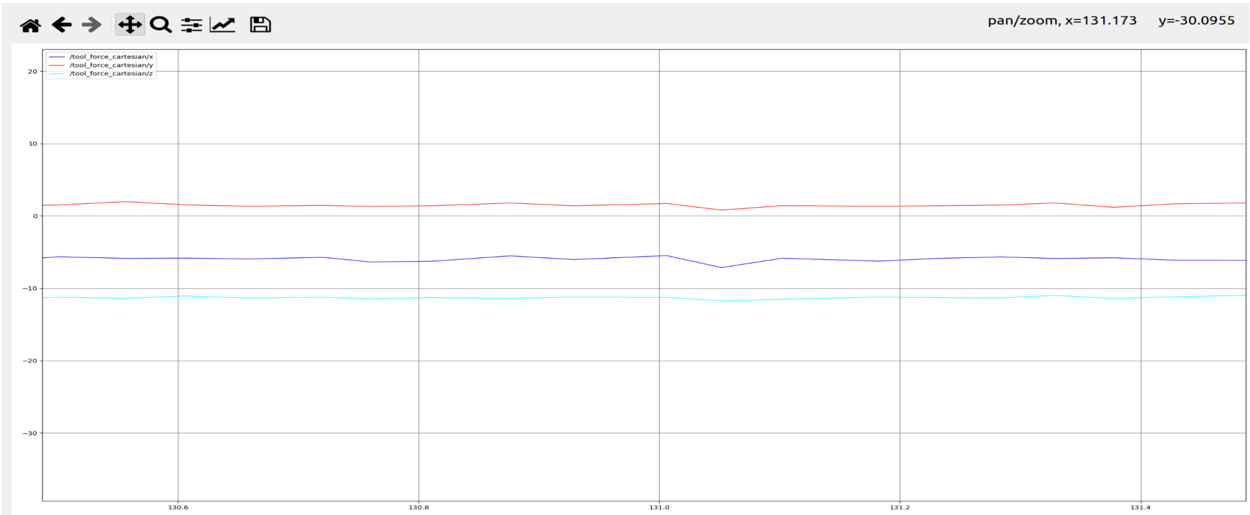

Figure 3: zero force(t = 0)

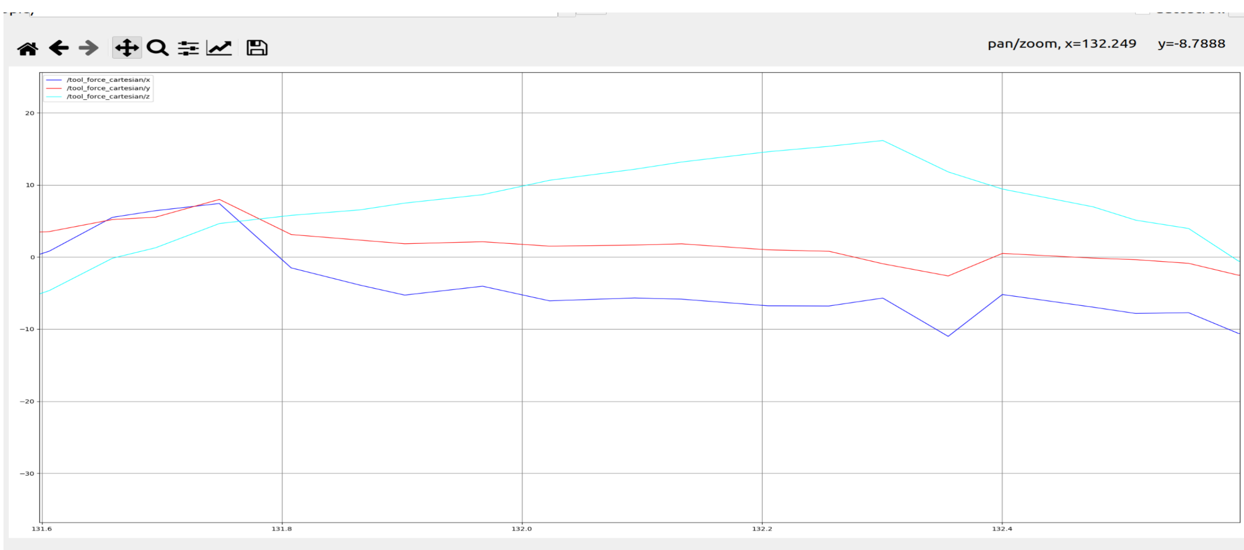

Figure 4: Force when applying a small force to the manipulator in Z axis