EE368 Lab7: Dynamics

代码及注释

```python
#!/usr/bin/env python3
import math
import numpy as np
import rospy
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Point
from std_msgs.msg import Float64

class Link:
    """
    机器人单个连杆类
    """
    def __init__(self, dh_params):
        """
        初始化 D-H参数 [alpha, a, d, theta_offset]
        """
        self.dh_params_ = dh_params

    def transformation_matrix(self, theta):
        """
        计算连杆的齐次变换矩阵
        """
        alpha = self.dh_params_[0]
        a = self.dh_params_[1]
        d = self.dh_params_[2]
        theta = theta+self.dh_params_[3]
        st = math.sin(theta)
        ct = math.cos(theta)
        sa = math.sin(alpha)
        ca = math.cos(alpha)
        trans = np.array([[ct, -st, 0, a],                    # DH参数的齐次变换矩阵计算
                          [st*ca, ct * ca, - sa, -sa * d],    # 包含旋转和平移变换
                          [st*sa, ct * sa,   ca,  ca * d],    # 用于计算连杆之间的坐标变换
                          [0, 0, 0, 1]])
        return trans

    def set_inertial_parameters(self, mass, center: list, inertia, T_dh_link):
        """
        设置连杆的惯性参数
        Args:
            mass: 连杆质量
            center: 质心位置 [x, y, z]
            inertia: 惯性张量 [Ixx, Ixy, Ixz, Iyy, Iyz, Izz]
            T_dh_link: 从DH坐标系到连杆坐标系的变换矩阵
        """
        self.mass = mass
        ixx = inertia[0]
```

```python
        ixy = inertia[1]
        ixz = inertia[2]
        iyy = inertia[3]
        iyz = inertia[4]
        izz = inertia[5]
        I = np.array(
            [[ixx, ixy, ixz], [ixy, iyy, iyz], [ixz, iyz, izz]])
        R = T_dh_link[0:3, 0:3]
        new_I = R.dot(I).dot(R.T)
        center.append(1.0)
        new_center = T_dh_link.dot(np.array(center).T)

        self.center = new_center[:3]
        self.inertia_tensor = new_I
        print(f"center of mass: {self.center}")
        print(f"inertia tensor: {self.inertia_tensor}")

    @staticmethod
    def basic_jacobian(trans, ee_pos):
        """计算连杆对应的基本雅可比矩阵列
        Args:
            trans: 当前连杆的齐次变换矩阵
            ee_pos: 末端执行器的位置
        Returns:
            6x1的基本雅可比矩阵列，包含线速度和角速度分量
        """
        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]])

        basic_jacobian = np.hstack(
            (np.cross(z_axis, ee_pos - pos), z_axis))  # 计算基本雅可比矩阵，包含线速度和角速度分量
        return basic_jacobian


class NLinkArm:
    """多连杆机器人类，实现运动学和动力学计算"""

    def __init__(self, dh_params_list) -> None:
        """初始化多连杆机器人
        Args:
            dh_params_list: 包含所有连杆D-H参数的列表
        """
        self.link_list = []
        for i in range(len(dh_params_list)):
            self.link_list.append(Link(dh_params_list[i]))

    def transformation_matrix(self, thetas):
        """计算从基坐标系到末端执行器的齐次变换矩阵
        Args:
            thetas: 所有关节角度列表
        Returns:
```

```
        4x4齐次变换矩阵
        """
        trans = np.identity(4)
        for i in range(len(self.link_list)):
            trans = np.dot(
                trans, self.link_list[i].transformation_matrix(thetas[i]))
        return trans

    def forward_kinematics(self, thetas):
        """计算正向运动学
        Args:
            thetas: 所有关节角度列表
        Returns:
            末端执行器位置和姿态 [x, y, z, alpha, beta, gamma]
        """
        trans = self.transformation_matrix(thetas)
        x = trans[0, 3]
        y = trans[1, 3]
        z = trans[2, 3]

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

    def euler_angle(self, thetas):
        """计算末端执行器的欧拉角
        Args:
            thetas: 所有关节角度列表
        Returns:
            ZYZ欧拉角 (alpha, beta, gamma)
        """
        trans = self.transformation_matrix(thetas)

        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

    def inverse_kinematics(self, ref_ee_pose):
        """计算逆运动学（使用雅可比矩阵迭代法）
        Args:
            ref_ee_pose: 目标末端位姿 [x, y, z, alpha, beta, gamma]
        Returns:
            计算得到的关节角度列表
        """
```

```python
        thetas = [0, 0, 0, 0, 0, 0]
        for cnt in range(5000):
            ee_pose = self.forward_kinematics(thetas)
            diff_pose = np.array(ref_ee_pose) - ee_pose

            basic_jacobian_mat = self.basic_jacobian(thetas)
            alpha, beta, gamma = self.euler_angle(thetas)

            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

            theta_dot = np.dot(
                np.dot(np.linalg.pinv(basic_jacobian_mat), K_alpha),
                np.array(diff_pose))
            thetas = thetas + theta_dot / 100.
            if np.linalg.norm(theta_dot) < 0.001:
                break
        # thetas = np.mod(thetas, 2*np.pi)
        return thetas

    def basic_jacobian(self, thetas):
        """计算整个机器人的基本雅可比矩阵
        Args:
            thetas: 关节角度列表
        Returns:
            6xn的基本雅可比矩阵，n为关节数量
        """
        ee_pos = self.forward_kinematics(thetas)[0:3]
        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]))
            basic_jacobian_mat.append(
                self.link_list[i].basic_jacobian(trans, ee_pos))
        return np.array(basic_jacobian_mat).T

    def get_torque(self, thetas, thetas_d, theta_dd, f_ext, n_ext):
        """使用牛顿-欧拉方法计算关节力矩
        Args:
            thetas: 关节角度列表
            thetas_d: 关节角速度列表
            theta_dd: 关节角加速度列表
            f_ext: 末端执行器外力 [fx, fy, fz]
            n_ext: 末端执行器外力矩 [nx, ny, nz]
        Returns:
            计算得到的各关节力矩
        """
        f_ext = np.array(f_ext).T
```

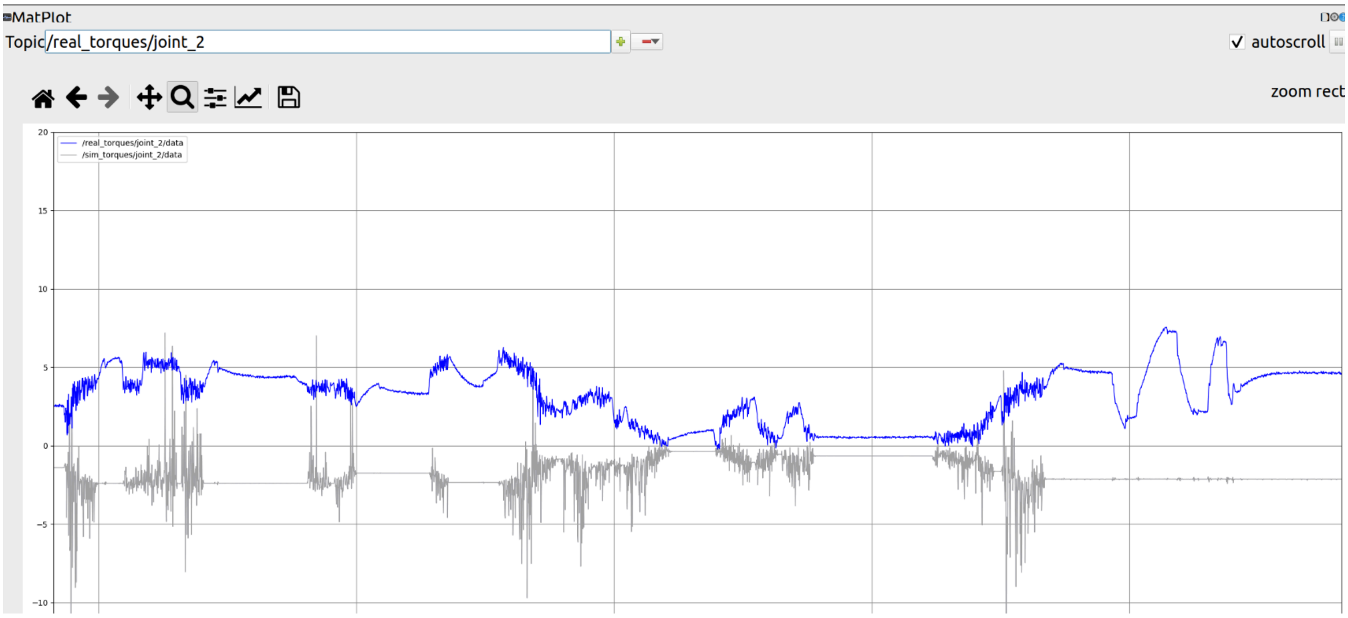
```
        n_ext = np.array(n_ext).T

        link_num = len(self.link_list)
        R_i_iplus1_list = np.zeros((3,3,link_num))
        P_i_iplus1_list = np.zeros((3,link_num))
        P_i_c_list = np.zeros((3,link_num+1))
        for i in range(link_num):
            T_i_iplus1 = self.link_list[i].transformation_matrix(thetas[i])
            R_i_iplus1_list[:,:,i] = T_i_iplus1[:3,:3]
            P_i_iplus1_list[:,i] = T_i_iplus1[:3, 3]
            P_i_c_list[:,i+1] = self.link_list[i].center

        omega = np.zeros((3, link_num+1))
        omega_d = np.zeros((3, link_num+1))
        v_dot_i = np.zeros((3, link_num+1))
        v_dot_c = np.zeros((3, link_num+1))
        v_dot_i[:, 0] = [0, 0, - 9.8] # 此处原代码为9.8，经实验证明应该为-9.8
        F = np.zeros((3, link_num+1))
        N = np.zeros((3, link_num+1))

        for i in range(link_num):
            R = R_i_iplus1_list[:,:,i].T              # 相邻连杆之间的旋转矩阵
            m = self.link_list[i].mass                # 连杆质量
            P_i_iplus1 = P_i_iplus1_list[:,i]        # 相邻关节间的位置向量
            P_iplus1_c = P_i_c_list[:,i+1]           # 从关节到质心的位置向量
            I_iplus1 = self.link_list[i].inertia_tensor  # 连杆惯性张量
            theta_dot_z = thetas_d[i]*np.array([0, 0, 1]).T
            omega[:, i+1] = R.dot(omega[:, i]) + theta_dot_z  # 计算角速度递推公式
            omega_d[:, i+1] = R.dot(omega_d[:, i]) + np.cross(
                R.dot(omega[:, i]), theta_dot_z) + theta_dd[i]*np.array([0, 0, 1]).T  # 计算角加速度递推公式
            v_dot_i[:, i+1] = R.dot(np.cross(omega_d[:, i], P_i_iplus1)+np.cross(
                omega_d[:, i], np.cross(omega_d[:, i], P_i_iplus1))+v_dot_i[:, i])  # 计算线加速度递推公式
            v_dot_c[:, i+1] = np.cross(omega_d[:, i+1], P_iplus1_c) + np.cross(
                omega[:, i+1], np.cross(omega[:, i+1], P_iplus1_c)) + v_dot_i[:, i+1]
            F[:, i+1] = m*v_dot_c[:, i+1]  # 计算连杆的惯性力
            N[:, i+1] = I_iplus1.dot(omega_d[:, i+1]) + \
                np.cross(omega[:, i+1], I_iplus1.dot(omega[:, i+1]))  # 计算连杆的惯性力矩

        f = np.zeros((3, link_num+1))
        n = np.zeros((3, link_num+1))
        tau = np.zeros(link_num+1)

        for i in range(link_num, 0, -1):
            R = T_i_iplus1[:3, :3]
            if i == link_num:
                f[:,i] = f_ext + F[:,i]  # 末端连杆的合力
                n[:,i] = N[:,i] + n_ext + np.cross(P_i_c_list[:,i],F[:,i])  # 末端连杆的合力矩
                tau[i] = n[:,i].T.dot(np.array([0, 0, 1]).T)  # 计算关节力矩
            else:
                R = R_i_iplus1_list[:,:,i]
                f[:,i] = R.dot(f[:,i+1]) + F[:,i]
```

```
                n[:,i] = N[:,i] + R.dot(n[:,i+1]) + np.cross(P_i_c_list[:,i],F[:,i]) +
np.cross(P_i_iplus1_list[:,i],R.dot(f[:,i+1]))
                tau[i] = n[:,i].T.dot(np.array([0, 0, 1]).T)
        return tau[1:]


if __name__ == "__main__":
    """
    初始化ROS节点，设置机器人参数并进行动力学仿真
    - 创建ROS节点和发布者
    - 设置Kinova Gen3 Lite机器人的D-H参数
    - 设置各连杆的惯性参数
    - 实时计算和发布真实力矩和仿真力矩
    """
    rospy.init_node("dynamics_test")
    real_torque_pub_list = []
    sim_torque_pub_list = []

    for i in range(6):
        real_pub = rospy.Publisher(f"/real_torques/joint_{i}",Float64,queue_size=1)
        real_torque_pub_list.append(real_pub)
        sim_pub = rospy.Publisher(f"/sim_torques/joint_{i}",Float64,queue_size=1)
        sim_torque_pub_list.append(sim_pub)

    dh_params_list = np.array([[0, 0, 243.25/1000, 0],
                               [math.pi/2, 0, 30/1000, 0+math.pi/2],
                               [math.pi, 280/1000, 20/1000, 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)
    gen3_lite.link_list[0].set_inertial_parameters(0.95974404, [
                                                   2.477E-05, 0.02213531, 0.09937686],
[0.00165947, 2e-08, 3.6E-07, 0.00140355, 0.00034927, 0.00089493],
                                                   np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0,
0, 1, -115/1000], [0, 0, 0, 1]]))
    gen3_lite.link_list[1].set_inertial_parameters(1.17756164, [0.02998299, 0.21154808,
0.0453031], [
                                                   0.01149277, 1E-06, 1.6E-07, 0.00102851,
0.00140765, 0.01133492],
                                                   np.array([[0, 1, 0, 0], [-1, 0, 0, 0],
[0, 0, 1, 0], [0, 0, 0, 1]]))
    gen3_lite.link_list[2].set_inertial_parameters(0.59767669, [0.0301559, 0.09502206,
0.0073555], [
                                                   0.00163256, 7.11E-06, 1.54E-06,
0.00029798, 9.587E-05, 0.00169091],
                                                   np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0,
0, 1, -20/1000], [0, 0, 0, 1]]))
    gen3_lite.link_list[3].set_inertial_parameters(0.52693412, [0.00575149, 0.01000443,
0.08719207], [
                                                   0.00069098, 2.4E-07, 0.00016483,
0.00078519, 7.4E-07, 0.00034115],
```

```python
                                                    np.array([[0, 1, 0, 0], [-1, 0, 0, 0],
[0, 0, 1, -105/1000], [0, 0, 0, 1]]))
    gen3_lite.link_list[4].set_inertial_parameters(0.58097325, [0.08056517, 0.00980409,
0.01872799], [
                                                    0.00021268, 5.21E-06, 2.91E-06,
0.00106371, 1.1E-07, 0.00108465],
                                                    np.array([[0, 1, 0, 0], [-1, 0, 0, 0],
[0, 0, 1, -28.5/1000], [0, 0, 0, 1]]))
    gen3_lite.link_list[5].set_inertial_parameters(0.2018, [0.00993, 0.00995, 0.06136], [
                                                    0.0003428, 0.00000019, 0.0000001,
0.00028915, 0.00000027, 0.00013076],
                                                    np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0,
0, 1, -130/1000], [0, 0, 0, 1]]))

    # 初始化状态变量
    last_velocities = [0,0,0,0,0,0]
    last_time = rospy.get_time()


    while not rospy.is_shutdown():
        # 获取机器人关节状态信息
        feedback = rospy.wait_for_message("/my_gen3_lite/joint_states", JointState)
        thetas = feedback.position[0:6]      # 当前关节角度
        velocities = feedback.velocity[0:6]    # 当前关节速度
        torques = np.array(feedback.effort[0:6])  # 实际关节力矩
        thetas_d = velocities   # 角速度

        # 计算时间间隔
        dt = rospy.get_time() - last_time
        last_time = rospy.get_time()
        # 计算角加速度
        thetas_dd = np.subtract(velocities,last_velocities)/dt
        last_velocities = velocities

        # 计算仿真力矩（不考虑外力和外力矩）
        sim_torque = gen3_lite.get_torque(thetas,thetas_d,thetas_dd,[0,0,0],[0,0,0])

        # 发布实际力矩和仿真力矩
        for i in range(6):
            real_torque_pub_list[i].publish(torques[i])
            sim_torque_pub_list[i].publish(sim_torque[i])


        print(f"joint torque: {torques}")
        print(f"sim torque: {sim_torque}")
        print(f"diff {np.subtract(torques,sim_torque)}")
```

重力补偿分析

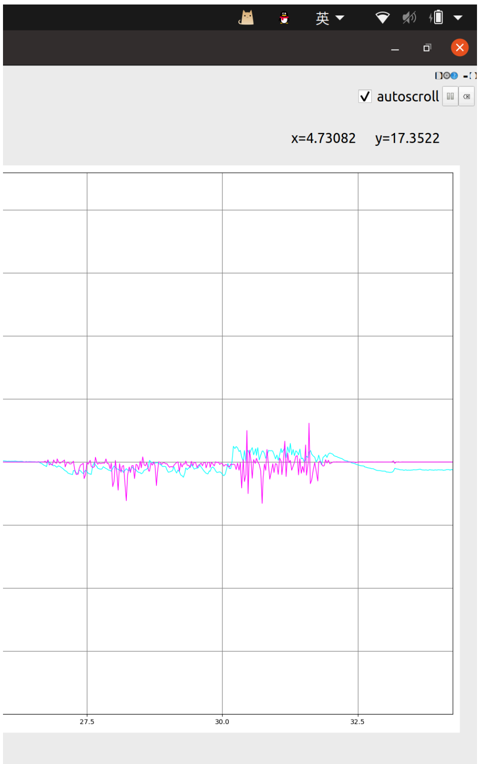
直接运行代码时，出现在遥控器控制的情况下，部分关节（如图中joint 2)的sim torque和real torque差距很大，甚至出现趋势相反的情况。

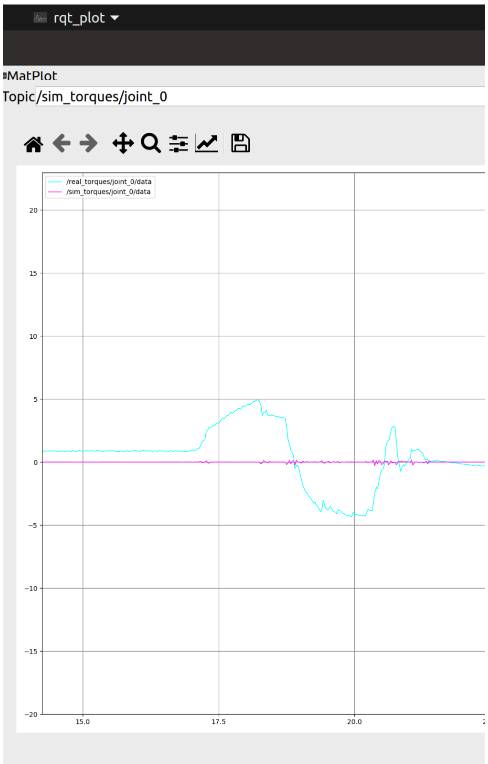


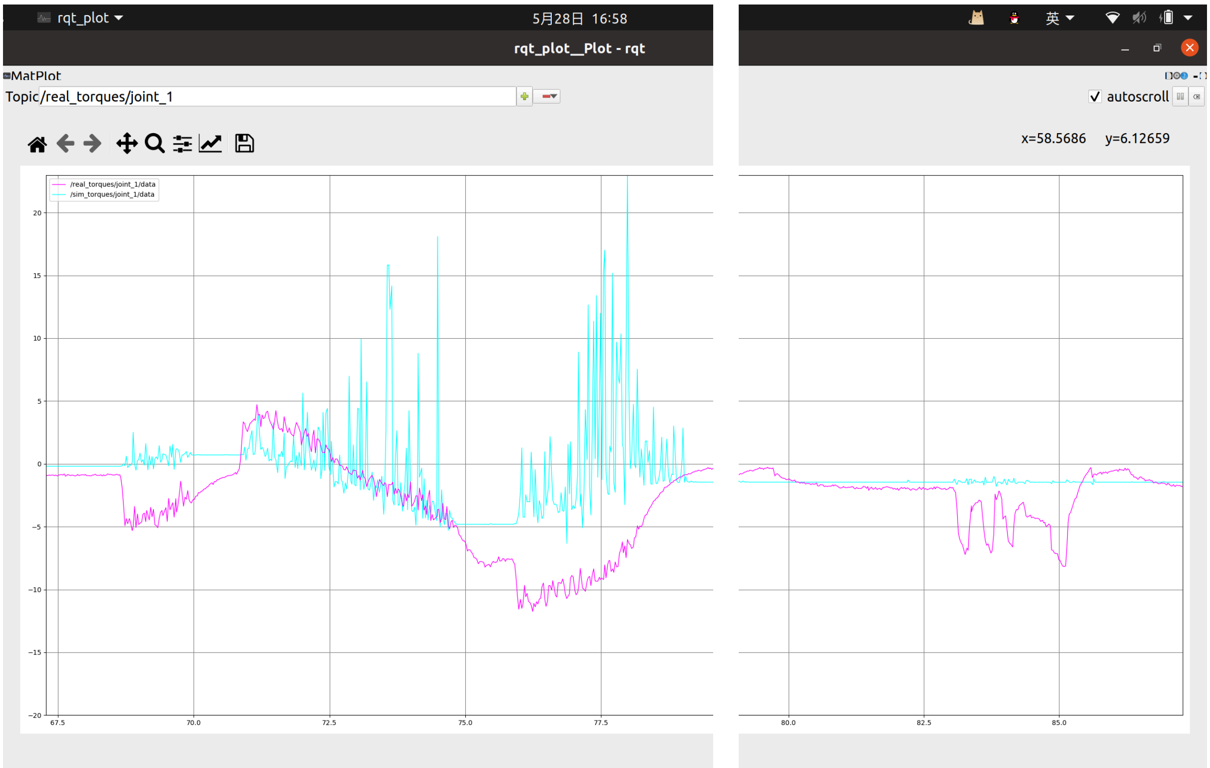
将`v_dot_i[:, 0] = [0, 0, 9.8]`改为`v_dot_i[:, 0] = [0, 0, -9.8]`后问题解决。经分析，原因是坐标系的建立与重力补偿的方向没有保持一致。

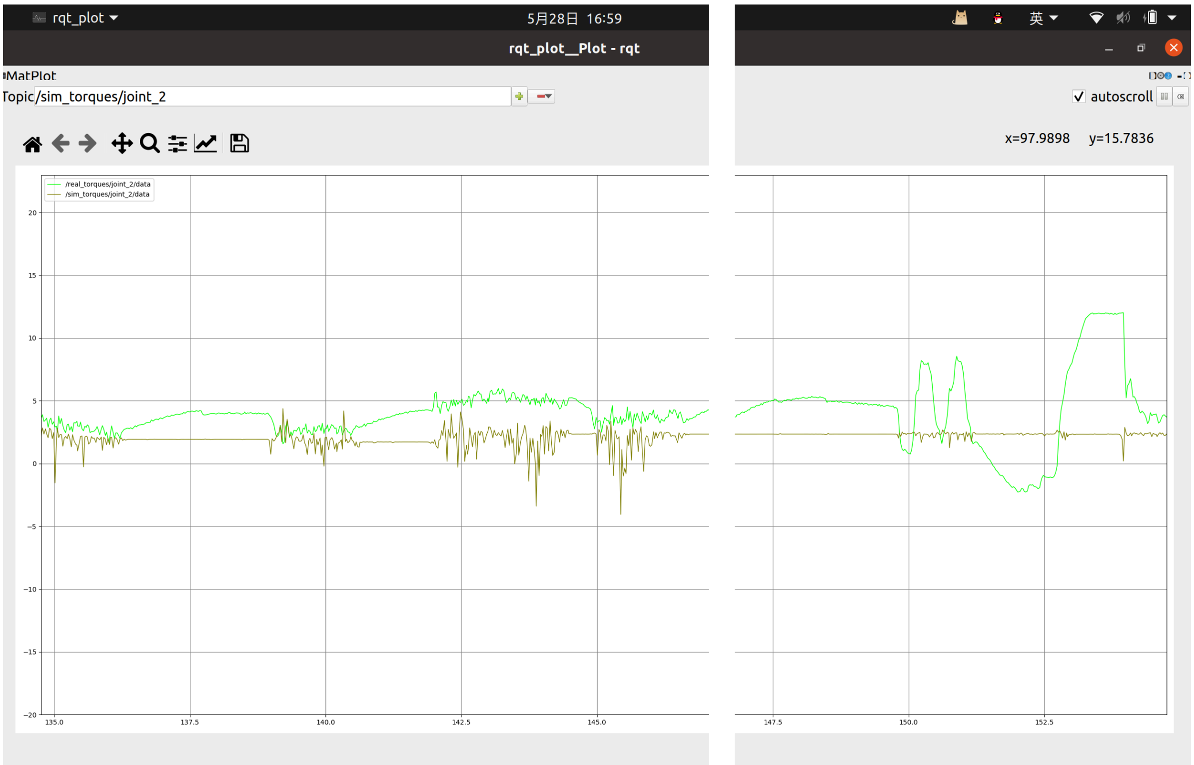
实验内容

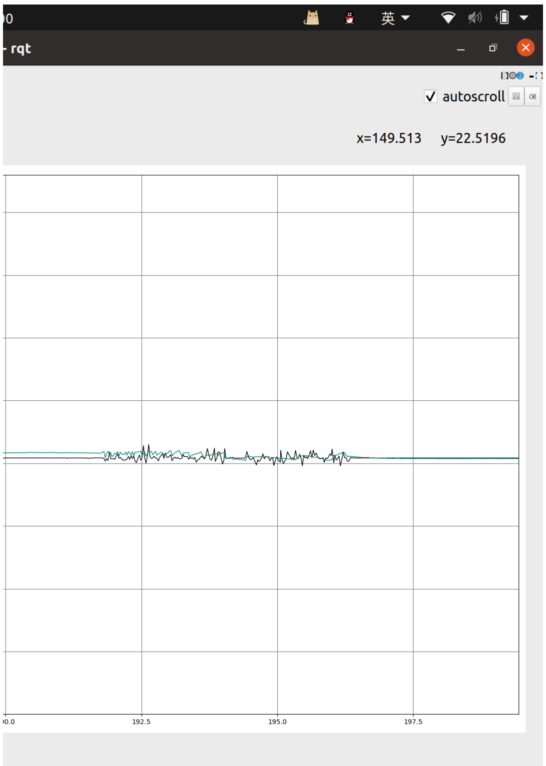
我们对机械臂的每个关节分别进行**遥控器控制**和**施加外力**，并读取各个关节的实际扭矩和仿真扭矩。

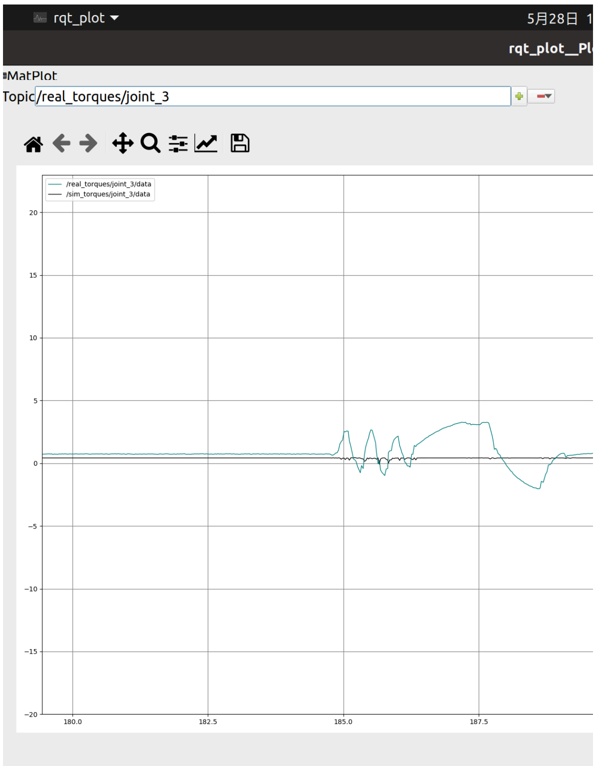


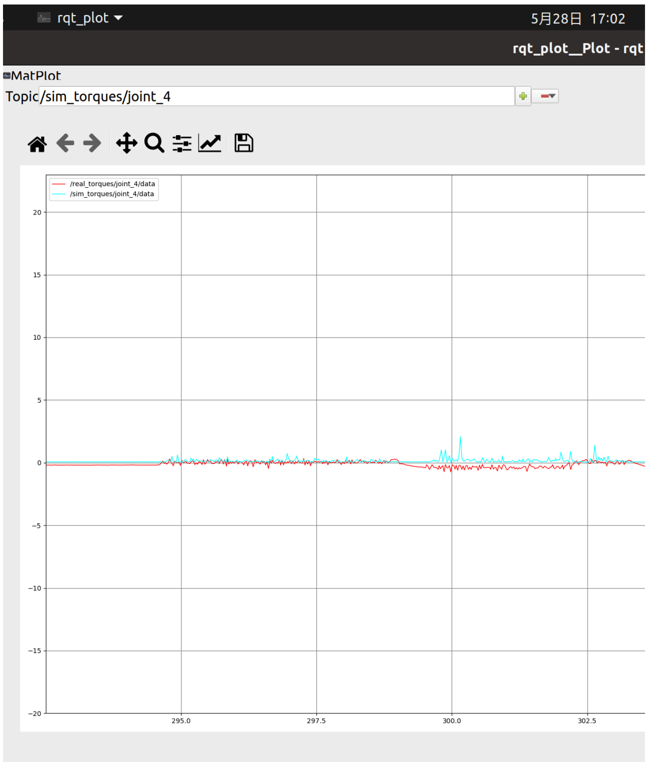


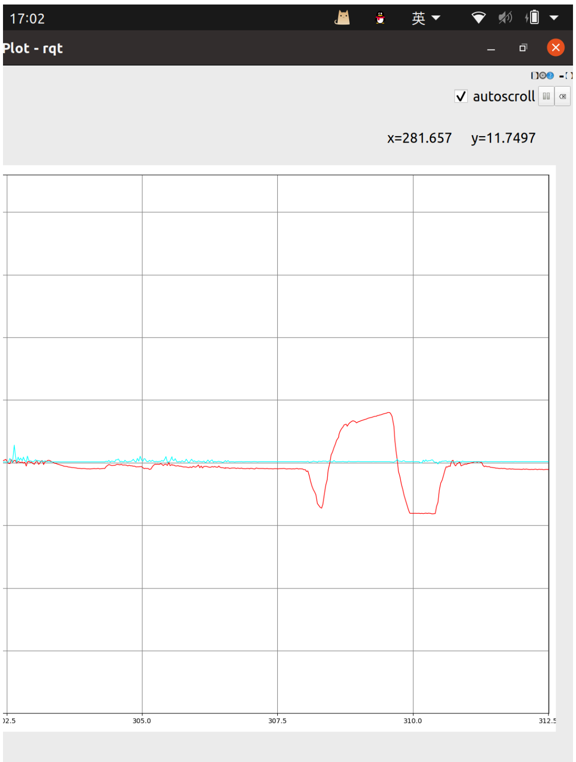

rqt_plot
5月28日 16:58
rqt_plot__Plot - rqt
MatPlot
Topic/real_torques/joint_1
autoscroll
x=58.5686 y=6.12659
/real_torques/joint_1/data
/sim_torques/joint_1/data
20
15
10
5
0
−5
−10
−15
−20
67.5
70.0
72.5
75.0
77.5
80.0
82.5
85.0

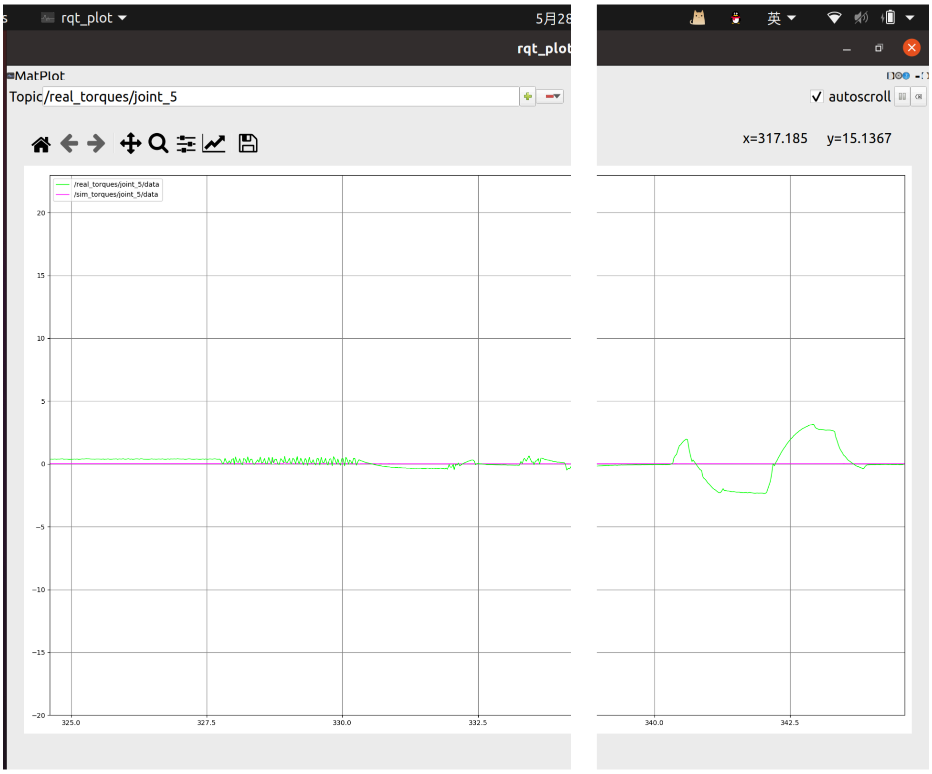

rqt_plot
5月28日 16:59
rqt_plot__Plot - rqt
MatPlot
Topic/sim_torques/joint_2
autoscroll
x=97.9898 y=15.7836
/real_torques/joint_2/data
/sim_torques/joint_2/data
20
15
10
5
0
−5
−10
−15
−20
135.0
137.5
140.0
142.5
145.0
147.5
150.0
152.5


rqt
autoscroll
x=149.513 y=22.5196
192.5
195.0
197.5


rqt_plot
5月28日
MatPlot
Topic /real_torques/joint_3
/real_torques/joint_3/data
/sim_torques/joint_3/data
20
15
10
5
0
−5
−10
−15
−20
180.0
182.5
185.0
187.5


rqt_plot
5月28日 17:02
rqt_plot__Plot - rqt
MatPlot
Topic /sim_torques/joint_4
/real_torques/joint_4/data
/sim_torques/joint_4/data
20
15
10
5
0
−5
−10
−15
−20
295.0
297.5
300.0
302.5


17:02
英
Plot - rqt
autoscroll
x=281.657 y=11.7497
305.0
307.5
310.0



上图分别为关节0到5的真实力矩和仿真力矩的对比图，其中左侧图为遥控器控制，右侧图为施加外力。

可以看到，施加外力时，sim_torques都为0，只有real_torques有变化。这是由于仿真模型在动力学计算时未引入外力参数（代码中f_ext和n_ext始终设为[0,0,0]），而真实机械臂通过关节力矩传感器直接测量到了实际受力。

遥控器控制时，sim_torques和real_torques之间依然有一些误差，这是可能是因为：

1. **动力学模型简化：**仿真中惯性参数（质量、质心位置、惯性张量）的设定存在微小误差（如代码中set_inertial_parameters的实测参数偏差）
2. **未建模因素**：关节摩擦、阻尼特性及电机动力学未包含在牛顿-欧拉计算模型中
3. **数值计算累积**：角加速度通过差分近似（代码中thetas_dd = (velocities - last_velocities)/dt）引入高频噪声
4. **传感器延迟**：真实力矩的采集与仿真计算的异步性导致相位差
5. **传动装置非线性**：谐波减速器等传动部件的回程差未在仿真中体现

等因素。