ECE780_Assignment_2_Q2b_python_code

June 12, 2024

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from progress.bar import Bar

class manipulator_control:
    def __init__(self, L, ML, IL, D, qd, qd_dot, qd_dot_dot, q, q_dot,␣
 ↪q_dot_dot, x, x_dot, xd, xd_dot):
        # RR manipulator situation
        self.L = L # length of each link
        self.ML = ML # mass of each link
        self.D = D # distance between link center of mass and its origin
        self.IL = IL # inertia of each link
        self.qd = qd # desired joint anlge
        self.qd_dot = qd_dot # desired joint velocity
        self.qd_dot_dot = qd_dot_dot # desired joint acceleration
        self.q = q # initial joint anlge
        self.q_dot = q_dot # initial joint velocity
        self.q_dot_dot = q_dot_dot # initial joint acceleration

        self.x = x # end-effector position
        self.x_dot = x_dot # end-effector velocity
        self.xd = xd # desired end-effector position
        self.xd_dot = xd_dot # desired end-effector velocity
        self.g = 9.81 # gravity
        self.Fc11 = 0
        self.Fc22 = 0
        self.Fv11 = 0.1
        self.Fv22 = 0.1
        self.dt = 0.01 # delta time
        self.T = 10.0 # simulation time
        self.max_length = self.L[0][0] + self.L[1][0] # maximum length
        self.text = "assignment_2_Q2b"

    def get_dynamics(self, q, q_dot):
        """
        Calculate the dynamic of the manipulator based on the joint anlge and␣
 ↪joint velocity
```

```python
        :param np.ndarray q: joint angle
        :param np.ndarray q_dot: joint velocity
        :returns:
            - D (np.ndarray) - Inertia matrix
            - C (np.ndarray) - Centrifugal and Coriolis matrix
            - Fc (np.ndarray) - Coulomb friction matrix
            - Fv (np.ndarray) - Viscous friction matrix
            - G (np.ndarray) - Gravity compensation
        """
        ML1, ML2 = self.ML[0][0], self.ML[1][0] # mass of each link
        D1, D2 = self.D[0][0], self.D[1][0] # distance between link center of␣
↪mass and its origin
        L1, L2 = self.L[0][0], self.L[1][0] # length of each link
        IL1, IL2 = self.IL[0][0], self.IL[1][0] # inertia of each link
        q1, q2 = q[0][0], q[1][0] # joint angle
        q1_dot, q2_dot = q_dot[0][0], q_dot[1][0] # joint velocity

        # Jvc1 = np.array([[-D1*np.sin(q1), 0],
        #                  [D1*np.cos(q1), 0],
        #                  [0, 0]])
        # vc1 = Jvc1 @ q_dot
        # Jvc2 = np.array([[-D1*np.sin(q1)-D2*np.sin(q1+q2), -D2*np.sin(q1+q2)],
        #                  [D1*np.cos(q1)+D2*np.cos(q1+q2), D2*np.cos(q1+q2)],
        #                  [0, 0]])
        # vc2 = Jvc2 @ q_dot

        # Jw1 = np.array([[0, 0],
        #                 [0, 0],
        #                 [1, 0]])
        # w1 = Jw1 @ q_dot
        # Jw2 = np.array([[0, 0],
        #                 [0, 0],
        #                 [1, 1]])
        # w2 = Jw2 @ q_dot

        # Inertia Matrix
        # K1 (linear) = 0.5*ML1*vc1.T@vc1 + 0.5*ML2*vc2.T@vc2
        # K1 (linear) = 0.5*ML1*Jvc1.T@q_dot.T@Jvc1@q_dot + 0.5*ML2*Jvc2.
↪T@q_dot.T@Jvc2@q_dot
        # K1 (linear) = 0.5*q_dot.T@(ML1*Jvc1.T@Jvc1 + ML2*Jvc2.T@Jvc2)@q_dot
        # K2 (rotation) = 0.5*IL1*w1.T@w1 + 0.5*IL2*w2.T@w2
        # K2 (rotation) = 0.5*IL1*Jw1.T@q_dot.T@Jw1@q_dot + 0.5*IL2*Jw2.T@q_dot.
↪T@Jw2@q_dot
        # K2 (rotation) = 0.5*q_dot.T@(IL1*Jw1.T@Jw1 + IL1*Jw2.T@Jw2)@q_dot
        # K2 (rotation) = 0.5*q_dot.T@(np.array([[I1, 0],[0, 0]]) + np.
↪array([[I2, I2],[I2, I2]]))@q_dot
```

```python
    # K2 (rotation) = 0.5*q_dot.T@(np.array([[I1+I2, I2], [I2, I2]]))@q_dot
    # K (total) = K1 + K2
    # K (total) = 0.5*q_dot.T@(ML1*Jvc1.T@Jvc1 + ML2*Jvc2.T@Jvc2)@q_dot + 0.5*q_dot.T@(np.array([[I1+I2, I2], [I2, I2]]))@q_dot
    # D = ML1*Jvc1.T@Jvc1 + ML2*Jvc2.T@Jvc2 + np.array([[I1+I2, I2], [I2, I2]])
    # K (total) = 0.5*q_dot.T@D@q_dot
    d11 = ML1*pow(D1,2)+ML2*(pow(L1,2)+pow(D2,2)+2*L1*D2*np.cos(q2))+IL1+IL2
    d12 = ML2*(pow(D2,2)+L1*D2*np.cos(q2))+IL2
    d21 = ML2*(pow(D2,2)+L1*D2*np.cos(q2))+IL2
    d22 = ML2*(pow(D2,2))+IL2
    D = np.array([[d11, d12],[d21, d22]])

    # Centrifugal and Coriolis matrix
    # cijk = 0.5*(partial(dkj/qi)+partial(dki/qj)-partial(dij/qk))
    # c111 = 0.5*(partial(d11/q1)+partial(d11/q1)-partial(d11/q1)) = 0.5*(0+0-0) = 0
    # c112 = 0.5*(partial(d21/q1)+partial(d21/q1)-partial(d11/q2)) = 0.5*(0+0-(-2*ML2*L1*D2*np.sin(q2))) = ML2*L1*D2*np.sin(q2) = -h
    # c121 = 0.5*(partial(d12/q1)+partial(d11/q2)-partial(d12/q1)) = 0.5*(0+(-2*ML2*L1*D2*np.sin(q2))-0) = -ML2*L1*D2*np.sin(q2) = h
    # c122 = 0.5*(partial(d22/q1)+partial(d21/q2)-partial(d12/q2)) = 0.5*(0+(-ML2*L1*D2*np.sin(q2))-(-ML2*L1*D2*np.sin(q2))) = 0
    # c211 = 0.5*(partial(d11/q2)+partial(d12/q1)-partial(d21/q1)) = 0.5*((-2*ML2*L1*D2*np.sin(q2))+0-0) = -ML2*L1*D2*np.sin(q2) = h
    # c212 = 0.5*(partial(d21/q2)+partial(d22/q1)-partial(d21/q2)) = 0.5*((-ML2*L1*D2*np.sin(q2))+0-(-ML2*L1*D2*np.sin(q2))) = 0
    # c221 = 0.5*(partial(d12/q2)+partial(d12/q2)-partial(d22/q1)) = 0.5*((-ML2*L1*D2*np.sin(q2))+(-ML2*L1*D2*np.sin(q2))-0) = -ML2*L1*D2*np.sin(q2) = h
    # c222 = 0.5*(partial(d11/q1)+partial(d11/q1)-partial(d11/q1)) = 0.5*(0+0-0) = 0
    h = -ML2*L1*D2*np.sin(q2)
    # c111 = 0
    # c112 = -h
    # c121 = h
    # c122 = 0
    # c211 = h
    # c212 = 0
    # c221 = h
    # c222 = 0

    # ckj = summation(cijk(q)qi_dot)
    # c11 = c111*q1_dot + c211*q2_dot = 0*q1_dot + h*q2_dot = h*q2_dot
    # c12 = c112*q1_dot + c212*q2_dot = h*q1_dot + h*q2_dot = h*q1_dot + h*q2_dot
```

```
        # c21 = c121*q1_dot + c221*q2_dot = -h*q1_dot + 0*q2_dot = -h*q1_dot
        # c22 = c122*q1_dot + c222*q2_dot = 0*q1_dot + 0*q2_dot = 0
        C = np.array([[h*q2_dot, h*q1_dot + h*q2_dot],
                      [-h*q1_dot, 0]])

        # Gravity component
        # P1 = ML1*g*D1*np.sin(q1)
        # P2 = ML2*g*(L1*np.sin(q1)+D2*np.sin(q1+q2))
        # P = P1 + P2
        g1 = (ML1*D1+ML2*L1)*self.g*np.cos(q1) + ML2*D2*self.g*np.cos(q1+q2) #␣
↪partial derviative of (G/q1)
        g2 = ML2*D2*self.g*np.cos(q1 + q2) # partial derviative of (G/q2)
        G = np.array([[g1], [g2]])

        # Coulomb friction matrix
        # friction exist on object moving relative to each other
        Fc = np.diag(np.array([self.Fc11, self.Fc22]))

        # Viscous friction matrix
        # friction exist on fluid moving relative to each other
        Fv = np.diag(np.array([self.Fv11, self.Fv22]))

        return D, C, Fc, Fv, G

    def forward_kinematic(self, q):
        """
        Calculate the forward kinematic of the manipulator to find end-effector␣
↪position based on the joint anlge

        :param np.ndarray q: joint angle
        :returns:
            - pos (np.ndarray) - end-effector of position
        """
        L1, L2 = self.L[0][0], self.L[1][0] # length of each link
        q1, q2 = q[0][0], q[1][0] # joint angle
        x = L1*np.cos(q1) + L2*np.cos(q1+q2)
        y = L1*np.sin(q1) + L2*np.sin(q1+q2)
        pos = np.array([[x],[y]])
        return pos

    def jacobian(self, q):
        """
        Calculate the jacobian matrix of the manipulator based on the joint␣
↪anlge

        :param np.ndarray q: joint angle
        :returns:
```

```python
            - J (np.ndarray) - jacobian matrix
        """
        L1, L2 = self.L[0][0], self.L[1][0] # length of each link
        q1, q2 = q[0][0], q[1][0] # joint angle
        j11 = -L1*np.sin(q1)-L2*np.sin(q1+q2)
        j12 = -L2*np.sin(q1+q2)
        j21 = L1*np.cos(q1)+L2*np.cos(q1+q2)
        j22 = L2*np.cos(q1+q2)
        J = np.array([[j11, j12],[j21, j22]])
        return J

    def main(self, omega, zeta, torque_limit):
        """
        Implement joint space controller of type inverse dynamics in here and regulate the joint angle to desired angle

        :param float omega: natural frequency
        :param float zeta: damping ratio
        :param float torque_limit: maximum magnitude of torque
        """
        self.Kp = np.diag(np.array([pow(omega,2), pow(omega,2)])) # pow(omega,2)
        self.Ki = np.diag(np.array([0, 0]))
        self.Kd = np.diag(np.array([2*zeta*omega, 2*zeta*omega])) # 2*zeta*omega
        error = np.array([[0], [0]])
        pos_x_list = []
        pos_y_list = []
        q1_list = []
        q2_list = []
        tau1_list = []
        tau2_list = []
        iternation_list = []
        self.xd = self.forward_kinematic(self.qd)
        with Bar('Processing... ', max=int(self.T/self.dt)) as bar:
            for i in range (int(self.T/self.dt)):
                # Get dynamic by current joint angle and joint velocity
                D, C, Fc, Fv, G = self.get_dynamics(self.q, self.q_dot)

                # Acculmated error
                error = error + ((self.qd-self.q)*self.dt)

                # Inverse dynamic (Computed torque) method aims to find non-linear feedback control law u to become zeros
                # u is the joint acceleration we need to achieve
                u = self.qd_dot_dot + self.Kp @ (self.qd-self.q) + self.Kd @ (self.qd_dot-self.q_dot) + self.Ki @ error

                # Calculate the torque
```

```
                tau = D @ u + C @ self.q_dot + Fc @ np.sign(self.q_dot) + Fv @ self.q_dot + G

                # Torque bound
                tau[tau >= torque_limit] = torque_limit
                tau[tau <= -torque_limit] = -torque_limit

                # Calculate the joint acceleration by solving dynamic equation
                self.q_dot_dot = np.linalg.inv(D) @ (tau - C @ self.q_dot - Fc @ np.sign(self.q_dot) - Fv @ self.q_dot - G)

                # Update the joint angle and velocity
                self.q_dot = self.q_dot + self.q_dot_dot * self.dt
                self.q = self.q + self.q_dot * self.dt

                self.x = self.forward_kinematic(self.q)
                x = self.x[0][0]
                y = self.x[1][0]
                q1 = self.q[0][0]
                q2 = self.q[1][0]
                tau1 = tau[0][0]
                tau2 = tau[1][0]
                q1_list.append(q1)
                q2_list.append(q2)
                tau1_list.append(tau1)
                tau2_list.append(tau2)
                iternation_list.append(i)
                pos_x_list.append(x)
                pos_y_list.append(y)

                # Plot the movement
                self.plot_robot()
                bar.next()

        print("Target joint angle: ", self.qd[0][0], self.qd[1][0])
        print("Final joint angle: ", self.q[0][0], self.q[1][0])
        print("Target position: ", self.xd[0][0], self.xd[1][0])
        print("Final position: ", self.x[0][0], self.x[1][0])
        self.plot_angle(q1_list, q2_list, iternation_list)
        self.plot_tau(tau1_list, tau2_list, iternation_list)
        self.plot_position(pos_x_list, pos_y_list, iternation_list)

    def plot_robot(self):
        # Plot robotic arm movement
        x0, y0 = 0, 0
        x1 = self.L[0][0] * np.cos(self.q[0][0])
        y1 = self.L[0][0] * np.sin(self.q[0][0])
```

```
        x2 = x1 + self.L[1][0] * np.cos(self.q[0][0] + self.q[1][0])
        y2 = y1 + self.L[1][0] * np.sin(self.q[0][0] + self.q[1][0])

        plt.plot([x0, x1], [y0, y1], 'b-', label='Link 1')
        plt.plot([x1, x2], [y1, y2], 'g-', label='Link 2')
        plt.plot([x0, x1, x2], [y0, y1, y2], 'ro', label='Joint')

        plt.xlim(-self.max_length, self.max_length)
        plt.ylim(-self.max_length, self.max_length)
        plt.plot(self.xd[0][0], self.xd[1][0], 'yo', label='Desired Position')

        plt.legend()
        plt.savefig('script\\assignment\\assignment_2_picture\\{}_robotic_arm.
↪png'.format(self.text))
        plt.draw()
        plt.pause(0.01)
        plt.clf()

    def plot_angle(self, q1_list, q2_list, iternation_list):
        fig, axs = plt.subplots(1, 1, figsize=(12, 6))
        # plot trajectory
        axs.plot(iternation_list, q1_list, label='q1') # joint angle 1
        axs.plot(iternation_list, q2_list, label='q2') # joint angle 2
        axs.axhline(y=qd[0], color='r', linestyle='--', label='qd1') # joint␣
↪velocity 1
        axs.axhline(y=qd[1], color='g', linestyle='--', label='qd2') # joint␣
↪velocity 2
        axs.set_xlabel('Time [{}s]'.format(self.dt))
        axs.set_ylabel('Joint Angles [rad]')
        axs.legend()
        plt.tight_layout()
        plt.savefig('script\\assignment\\assignment_2_picture\\{}_joint_angle.
↪png'.format(self.text))
        plt.show()

    def plot_tau(self, tau1_list, tau2_list, iternation_list):
        # plot torque
        fig, axs = plt.subplots(1, 1, figsize=(12, 6))
        axs.plot(iternation_list, tau1_list, label='tau1') # joint torque 1
        axs.plot(iternation_list, tau2_list, label='tau2') # joint torque 2
        axs.set_xlabel('Time [{}s]'.format(self.dt))
        axs.set_ylabel('Joint Torques [Nm]')
        axs.legend()
        plt.tight_layout()
        plt.savefig('script\\assignment\\assignment_2_picture\\{}_joint_torque.
↪png'.format(self.text))
        plt.show()
```

```
    def plot_position(self, pos_x_list, pos_y_list, iternation_list):
        # plot position
        fig, axs = plt.subplots(1, 1, figsize=(12, 6))
        axs.plot(iternation_list, pos_x_list, label='current x') # current␣
 ↪position x
        axs.plot(iternation_list, pos_y_list, label='current y') # current␣
 ↪position y
        axs.axhline(y=self.xd[0], color='r', linestyle='--', label='desired x')␣
 ↪# desired position x
        axs.axhline(y=self.xd[1], color='g', linestyle='--', label='desired y')␣
 ↪# desired position y
        axs.set_xlabel('Time [{}s]'.format(self.dt))
        axs.set_ylabel('Position [m]')
        axs.legend()

        plt.tight_layout()
        plt.
 ↪savefig('script\\assignment\\assignment_2_picture\\{}_joint_position.png'.
 ↪format(self.text))
        plt.show()

L = np.array([[1.0], [0.5]], dtype=float) # length of each link
ML = np.array([[0.1], [0.05]], dtype=float) # mass of each link
D = np.array([[0.5], [0.25]], dtype=float) # distance between link center of␣
 ↪mass and its origin
IL = np.array([[0.1], [0.05]], dtype=float) # inertia of each link
q = np.array([[0.5], [1]], dtype=float) # initial joint anlge
qd = np.array([[-2*np.pi/3], [2*np.pi/3]], dtype=float) # desired joint anlge
q_dot = np.array([[0], [0]], dtype=float) # initial joint velocity
qd_dot = np.array([[0], [0]], dtype=float) # desired joint velocity
q_dot_dot = np.array([[0], [0]], dtype=float) # initial joint acceleration
qd_dot_dot = np.array([[0], [0]], dtype=float) # desired joint acceleration
x = np.array([[0], [0]], dtype=float) # initial end-effector position
x_dot = np.array([[0], [0]], dtype=float) # initial end-effector velocity
xd = np.array([[0], [0]], dtype=float) # desired end-effector position
xd_dot = np.array([[0], [0]], dtype=float) # desired end-effector velocity
sim = manipulator_control(L, ML, IL, D, qd, qd_dot, qd_dot_dot, q, q_dot,␣
 ↪q_dot_dot, x, x_dot, xd, xd_dot)
omega = 5 # natural frequency
zeta = 0.5 # damping ratio
torque_limit = 0.1 # 0.5 for Q2a or 0.1 for Q2b
sim.main(omega, zeta, torque_limit)
```