ECE780_Assignment_2_Q3i_python_code

June 12, 2024

[ ]:
```python
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, q, q_dot, q_dot_dot, x, x_dot,␣
↪x_dot_dot, xd, xd_dot, xd_dot_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.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.x_dot_dot = x_dot_dot # end-effector acceleration
        self.xd = xd # desired end-effector position
        self.xd_dot = xd_dot # desired end-effector velocity
        self.xd_dot_dot = xd_dot_dot # desired end-effector acceleration
        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 = 5.0 # simulation time
        self.Kwall = 50 # N/m
        self.max_length = self.L[0][0] + self.L[1][0] # maximum length
        self.text = "assignment_2_Q3ii"

    def get_dynamics(self, q, q_dot):
        """
```

```
        Calculate the dynamic of the manipulator based on the joint anlge and␣
↪joint velocity

        :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
    # 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␣
↪angle
```

```
        :param np.ndarray q: joint angle
        :returns:
            - 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 derivative_jacobian(self, q, q_dot):
        """
        Calculate the derivative jacobian matrix of the manipulator based on␣
↪the joint angle and velocity

        :param np.ndarray q: joint angle
        :param np.ndarray q_dot: joint velocity
        :returns:
            - 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
        q_dot_1, q_dot_2 = q_dot[0][0], q_dot[1][0] # joint velocity
        j_dot_11 = -L1*np.cos(q1)*q_dot_1-L2*np.cos(q1+q2)*(q_dot_1+q_dot_2)
        j_dot_12 = -L2*np.cos(q1+q2)*(q_dot_1+q_dot_2)
        j_dot_21 = -L1*np.sin(q1)*q_dot_1-L2*np.sin(q1+q2)*(q_dot_1+q_dot_2)
        j_dot_22 = -L2*np.sin(q1+q2)*(q_dot_1+q_dot_2)
        J_dot = np.array([[j_dot_11, j_dot_12],[j_dot_21, j_dot_22]])
        return J_dot

    def main(self, M_d, Kd, Kp, elastic):
        """
        Implement impedance controller in here and regulate the end-effector␣
↪position to desired position

        :param float Kp: proportional end-effector position error gain
        :param float Ki: integral end-effector position error gain
        :param float Kd: derivative end-effector velocity error gain
        :param float M: mass matrix parameter
        :param bool elastic: elastic wall exist or not
        """
        self.M_d = np.diag(np.array([M_d, M_d]))
        self.Kd = np.diag(np.array([Kd, Kd]))
        self.Kp = np.diag(np.array([Kp, Kp]))
```

```python
        pos_x_list = []
        pos_y_list = []
        q1_list = []
        q2_list = []
        tau1_list = []
        tau2_list = []
        iternation_list = []
        force_list = []
        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)

                J = self.jacobian(self.q)
                J_dot = self.derivative_jacobian(self.q, self.q_dot)

                # Calculate the end-effector position based on the joint angle
                self.x = self.forward_kinematic(self.q)

                # Calculate the end-effector velocity based on the joint velocity
                self.x_dot = J @ self.q_dot

                if elastic == True:
                    if self.x[1][0] <= 0:
                        # elastic wall situation
                        # Calculate the external force matrix
                        K = np.diag(np.array([0, self.Kwall]))

                        # Calculate the external force vector
                        Fe = K @ -self.x
                    else:
                        Fe = np.array([[0], [0]])
                else:
                    # free motion situation
                    # Calculate the external force vector
                    Fe = np.array([[0], [0]])

                # Impedance control
                # u is the joint acceleration we need to achieve
                u = np.linalg.inv(J) @ np.linalg.inv(self.M_d) @ (self.M_d @ self.xd_dot_dot + self.Kd @ (self.xd_dot-self.x_dot) + self.Kp @ (self.xd-self.x) - self.M_d @ J_dot @ self.q_dot - Fe)

                # Calculate the torque
                tau = D @ u + C @ self.q_dot + G + J.T @ Fe
```

```python
            # Calculate the joint acceleration by solving dynamic equation
            self.q_dot_dot = np.linalg.inv(D) @ (tau - C @ 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]
            F_wall = Fe[1][0]
            q1_list.append(q1)
            q2_list.append(q2)
            tau1_list.append(tau1)
            tau2_list.append(tau2)
            iternation_list.append(i)
            force_list.append(F_wall)
            pos_x_list.append(x)
            pos_y_list.append(y)

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

    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)
    self.plot_force(force_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\\assignment_2_Q3i_robotic_arm.
↪png')
        plt.draw()
        plt.pause(0.01)
        plt.clf()

    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()

    def plot_force(self, force_list, iternation_list):
        # plot magnitude of the force exerted by the wall on the robot end␣
↪effector
        fig, axs = plt.subplots(1, 1, figsize=(12, 6))
        axs.plot(iternation_list, force_list, label="wall force") # joint␣
↪torque 1
        axs.set_xlabel('Time [{}s]'.format(self.dt))
```

```
        axs.set_ylabel('Force [N]')
        axs.legend()

        plt.tight_layout()
        plt.savefig('script\\assignment\\assignment_2_picture\\{}_joint_force.
 ↪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([[np.pi/2], [-np.pi/2]], dtype=float) # initial joint anlge
qd = np.array([[0], [0]], 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
x = np.array([[0], [0]], dtype=float) # initial end-effector position
x_dot = np.array([[0], [0]], dtype=float) # initial end-effector velocity
x_dot_dot = np.array([[0], [0]], dtype=float) # initial end-effector␣
 ↪acceleration
xd = np.array([[1.2], [-0.5]], dtype=float) # desired end-effector position
xd_dot = np.array([[0], [0]], dtype=float) # desired end-effector velocity
xd_dot_dot = np.array([[0], [0]], dtype=float) # desired end-effector␣
 ↪acceleration
sim = manipulator_control(L, ML, IL, D, qd, qd_dot, q, q_dot, q_dot_dot, x,␣
 ↪x_dot, x_dot_dot, xd, xd_dot, xd_dot_dot)
M_d = 1.0
Kd = 10 # Kd
Kp = 25 # Kp
elastic = False # False for Q3i, True for Q3ii
sim.main(M_d, Kd, Kp, elastic)
```