Fourth Task for SRS

Salam Ali

503263

In this project, we will modify the passive mechanism to become active by adding actuators.

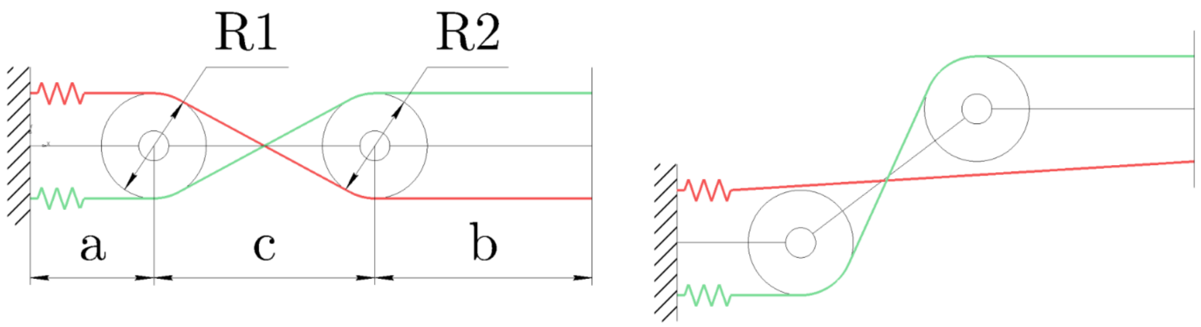


Figure 1: Passive RR mechanism

Given values from table:

503263	14.33	2.78	12.2	33.68	1.67	-39

1.XML modifying:

In this task, two independent actuators were added to the model in order to control the two tendon-driven-joints q_1 and q_2 . The XML file was extended with the required <actuator> and <sensor> elements so that each joint could receive its own control signal and provide feedback for closed-loop regulation.

```
<actuator>
        <motor name="m_elbow" joint="elbow" gear="1" ctrlrange="-15 15"/>
        <motor name="m_wrist" joint="wrist" gear="1" ctrlrange="-15 15"/>
    </actuator>
<sensor>
        <jointpos name="s_elbow_pos" joint="elbow"/>
        <jointvel name="s_elbow_vel" joint="elbow"/>
        <jointpos name="s_wrist_pos" joint="wrist"/>
        <jointvel name="s_wrist_vel" joint="wrist"/>
    </sensor>
```

A separate sinusoidal reference trajectory was assigned to each joint, meaning that the mechanism is driven by two different sine waves, each with its own amplitude, frequency, and bias, taken from the parameter table provided in the assignment.

$$q_1^{des} = 14.33\sin(2.78t) + 12.2$$

$$q_2^{des} = 33.68\sin(1.67t) - 39$$

A PD controller was implemented for both joints, computing control torques based on the tracking error between the desired and actual joint positions and velocities. Safety clipping was applied to ensure that each actuator operated within its allowed control range.

This setup allows the mechanism to follow two distinct periodic motions simultaneously, demonstrating stable and accurate tracking performance throughout the simulation.

2.Python code:

```python
import mujoco
import mujoco.viewer
import numpy as np
import matplotlib.pyplot as plt
import time

model_path = r"C:\Users\user\OneDrive\Desktop\MuJoCo\tendon2.xml"
model = mujoco.MjModel.from_xml_path(model_path)
data  = mujoco.MjData(model)

elbow_id = model.joint('elbow').id
wrist_id = model.joint('wrist').id

elbow_qpos = model.jnt_qposadr[elbow_id]
wrist_qpos = model.jnt_qposadr[wrist_id]

elbow_qvel = model.jnt_dofadr[elbow_id]
wrist_qvel = model.jnt_dofadr[wrist_id]

act1 = model.actuator('m_elbow').id
act2 = model.actuator('m_wrist').id

deg2rad = np.pi / 180

AMP1  = 14.33 * deg2rad
FREQ1 = 2.78
BIAS1 = 12.2 * deg2rad

AMP2  = 33.68 * deg2rad
FREQ2 = 1.67
BIAS2 = -39 * deg2rad

Kp = np.array([50.0, 40.0])
Kd = np.array([0.2, 0.2])

ctrl_min = model.actuator_ctrlrange[:, 0]
ctrl_max = model.actuator_ctrlrange[:, 1]

dt = model.opt.timestep
sim_time = 100.0
steps = int(sim_time / dt)
t0 = 0.0

q1_start = AMP1 * np.sin(2*np.pi*FREQ1 * t0) + BIAS1
q2_start = AMP2 * np.sin(2*np.pi*FREQ2 * t0) + BIAS2

data.qpos[elbow_qpos]  = -q1_start
data.qpos[wrist_qpos]  = -q2_start
data.qvel[elbow_qvel]  = 0
data.qvel[wrist_qvel]  = 0

mujoco.mj_forward(model, data)

t_log  = []
q1_log = []
q2_log = []
q1d_log = []
q2d_log = []
```

```python
with mujoco.viewer.launch_passive(model, data) as viewer:

    for step in range(steps):

        if not viewer.is_running():
            break

        t = step * dt

        # desired sine motion
        q1_des = AMP1 * np.sin(2*np.pi*FREQ1 * t) - BIAS1
        q2_des = AMP2 * np.sin(2*np.pi*FREQ2 * t) - BIAS2

        # desired velocities
        dq1_des = AMP1 * 2*np.pi*FREQ1 * np.cos(2*np.pi*FREQ1 * t)
        dq2_des = AMP2 * 2*np.pi*FREQ2 * np.cos(2*np.pi*FREQ2 * t)

        # actual states
        q1  = data.qpos[elbow_qpos]
        q2  = data.qpos[wrist_qpos]
        dq1 = data.qvel[elbow_qvel]
        dq2 = data.qvel[wrist_qvel]

        # PD control
        u1 = Kp[0] * (q1_des - q1) + Kd[0] * (dq1_des - dq1)
        u2 = Kp[1] * (q2_des - q2) + Kd[1] * (dq2_des - dq2)

        # safety clipping
        u1 = float(np.clip(u1, ctrl_min[act1], ctrl_max[act1]))
        u2 = float(np.clip(u2, ctrl_min[act2], ctrl_max[act2]))

        # apply control
        data.ctrl[act1] = u1
        data.ctrl[act2] = u2

        # step simulation
        mujoco.mj_step(model, data)
        viewer.sync()

        # logging
        t_log.append(t)
        q1_log.append(q1)
        q2_log.append(q2)
        q1d_log.append(q1_des)
        q2d_log.append(q2_des)
    print("\nSimulation finished — close the viewer window to exit.")
    while viewer.is_running():
        viewer.sync()
        time.sleep(0.01)

# PLOTS
plt.figure()
plt.plot(t_log, q1_log,  color="green", label="q1 actual")
plt.plot(t_log, q1d_log, color="purple", linestyle="--", label="q1 desired")
plt.legend(); plt.grid()
plt.xlabel("time (s)")
plt.ylabel("q1 (rad)")
plt.figure()
plt.plot(t_log, q2_log,  color="green", label="q2 actual")
plt.plot(t_log, q2d_log, color="purple", linestyle="--", label="q2 desired")
plt.legend(); plt.grid()
plt.xlabel("time (s)")
plt.ylabel("q2 (rad)")
plt.show()
```

3.Simulation Result:

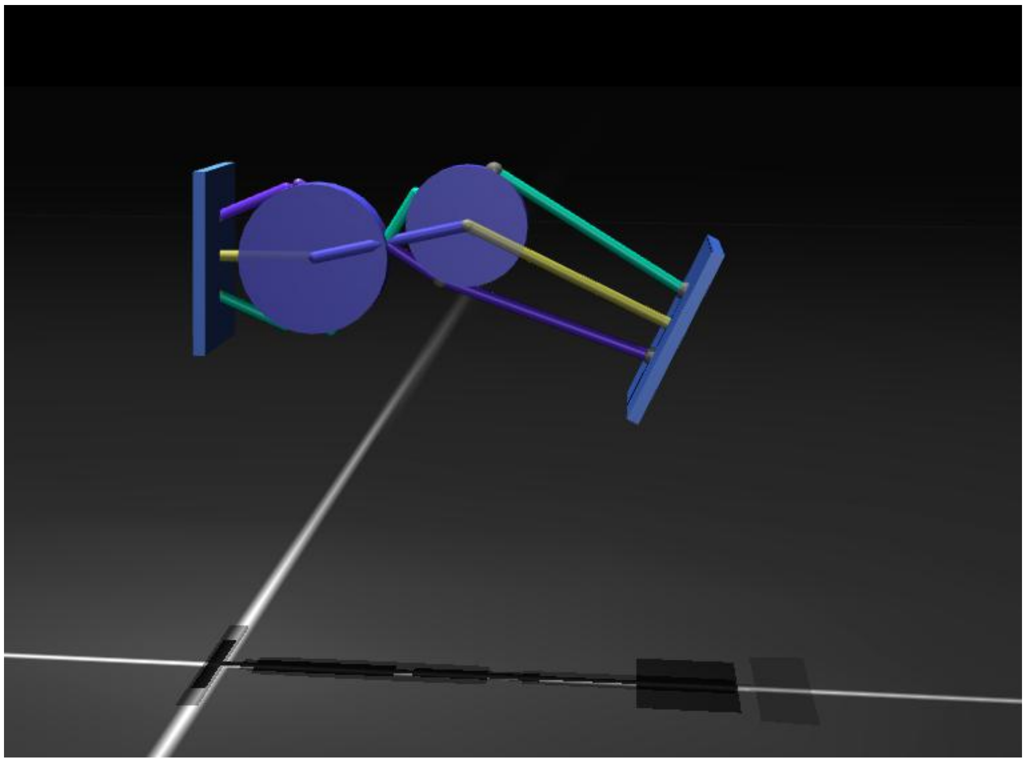
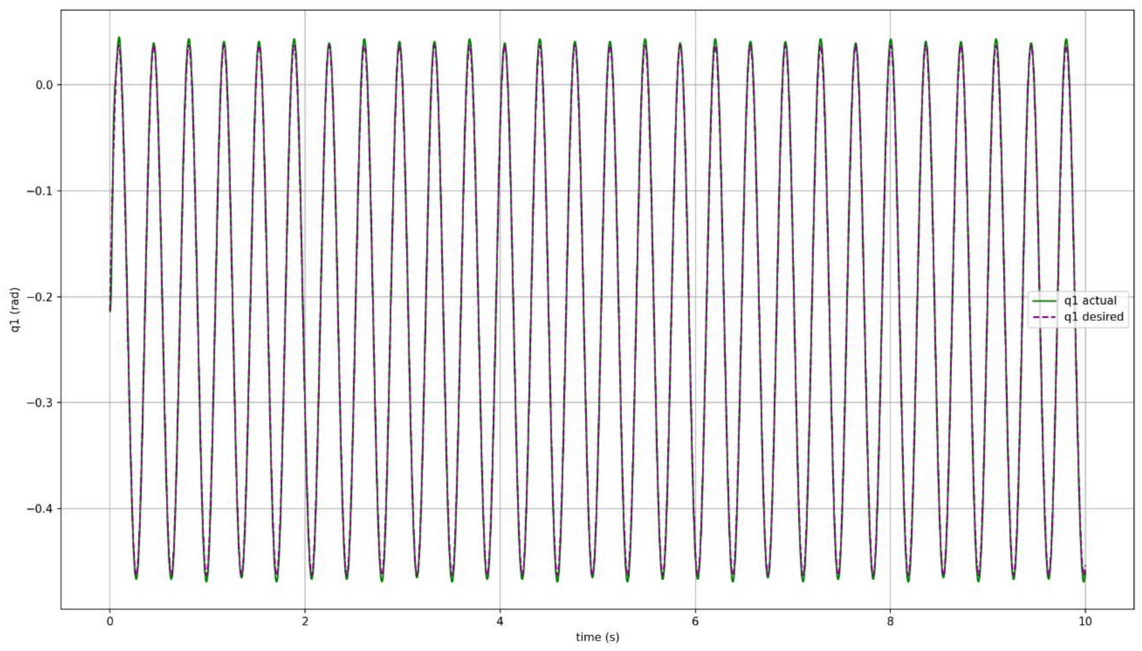
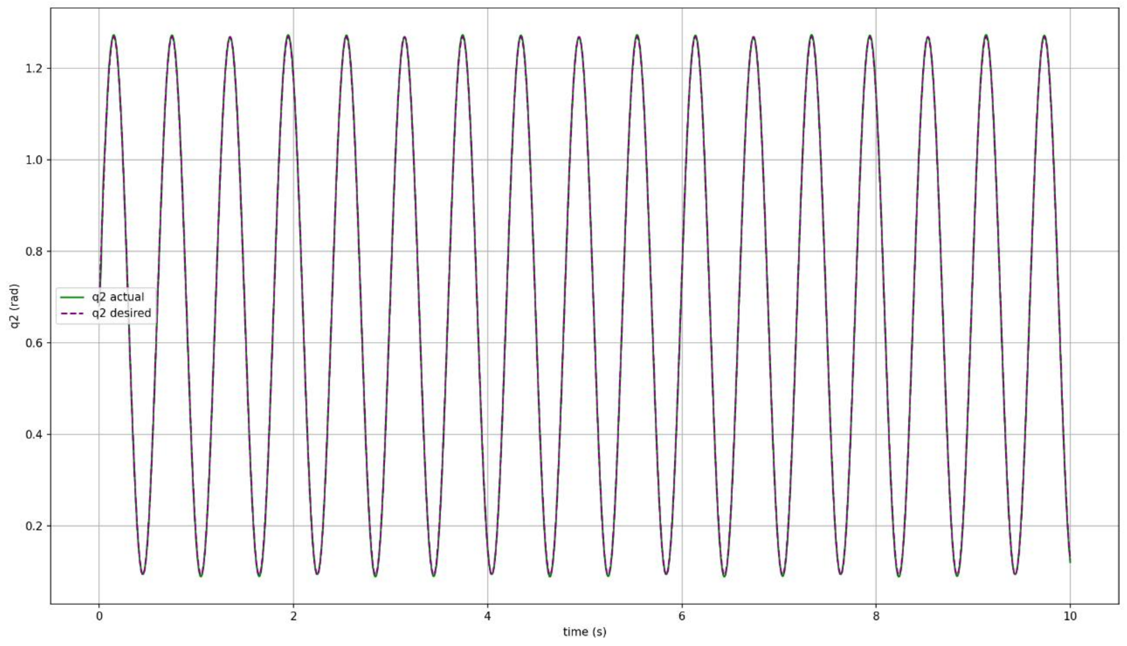


Figure 2: Simulation Result

To verify that the implemented controller drives the mechanism correctly, we visualized both the desired and actual joint trajectories using time-based plots. By comparing the sine-wave reference signals with the measured joint angles, we can evaluate how accurately the system follows the commanded motion. These plots allow us to inspect tracking performance, identify any phase lag or steady-state offsets, and ensure that the generated movement matches the expected behavior of the mechanism.





The controller was successfully implemented by adding two actuators and defining sine-wave reference trajectories for both joints. Using the PD regulation, the mechanism was able to track the desired motion without exceeding its workspace. The comparison between the actual and desired joint angles shows a very close match, confirming that the controller parameters and sine inputs were correctly tuned. Overall, the system behaves as expected and the control task is completed successfully.