МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

(Университет ИТМО)

Международный научно-образовательный центр

Физики наноструктур

Отчет лаб 4

по дисциплине:

Simulation of Robotic Systems

Студент:

Группа № R4134c *Буй Динь Кхай Нгуен*

Преподаватель: *Ракшин Егор Александрович*

Санкт-Петербург 2025

TABLE OF CONTENTS

1. PROJECT OVERVIEW 3
1.1 Parameters 3
1.2 Desired Trajectory (Sinusoidal Input) 3
2. PD CONTROLLER 4
2.1 Pd controller 4
2.2 Sensors and motors 5
3. APPENDIX 6
4. RESULTS 14
5. CONCLUSION 16

1. PROJECT OVERVIEW

The objective of Task 4 was to design and implement a PD controller directly acting on the rotary joints of the 2R mechanism to track a desired sinusoidal angular trajectory.

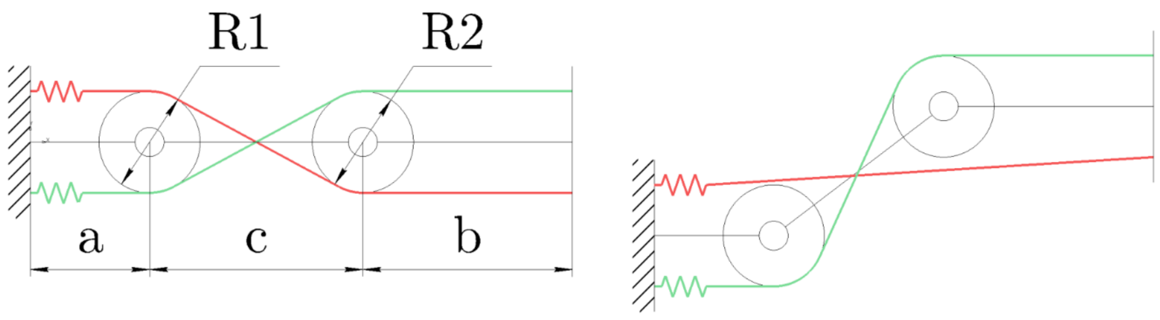

Figure 1: Variant 1 - Tendon connected 2R planar mechanism

1.1 Parameters

Parameter	Symbol	Value	Unit
Diameter R1	$R1$	0.0019	m
Diameter R2	$R2$	0.0012	m
Linkage 1 length	a	0.052	m
Linkage 3 Length	b	0.052	m
Linkage 2 length	c	0.044	m

Table 1: System Parameters

1.2 Desired Trajectory (Sinusoidal Input)

Joint	Equation	AMP	FREQ	BIAS
rotate_1	$qdes(t) = A \cdot sin(\omega t) + \phi$	50	3.53	41.4
rotate_2	$qdes(t) = A \cdot sin(\omega t) + \phi$	56.39	2.85	43.1

Table 2: Desired Trajectory

2. PD CONTROLLER

2.1 Pd controller

A standard PD controller is employed to compute the required torque τat each joint to track the desired trajectory.

- **Proportional Gain:** $K_P = 11.0$
- **Derivative Gain:** $K_D = 0.01$

The PD control law is given by:

$$\tau = K_P(q_{\text{des}} - q) + K_D(\dot{q}_{\text{des}} - \dot{q})$$

Where:

- q: Actual joint position
- $\dot{q}$: Actual joint velocity
- q_{des}: Desired joint position
- $\dot{q}_{\text{des}}$: Desired joint velocity

The formula of PD controller was implemented below:

```python
def qdes(t, AMP, FREQ, BIAS):
    """Desired position for Joint """
    return AMP * np.sin(FREQ * t) + BIAS

def qd_des(t, AMP, FREQ):
    """Desired velocity for Joint """
    return AMP * FREQ * np.cos(FREQ * t)

def pd_control(q, qd, q_des, qd_des, kp, kd):
    """
    Standard PD controller to calculate required torque (tau).
    tau = Kp * (q_des - q) + Kd * (qd_des - qd)
    """
    position_error = q_des - q
    velocity_error = qd_des - qd
    tau = kp * position_error + kd * velocity_error
    return tau
```

2.2 Sensors and motors

Sensors and motors are included in the XML of the model

```
      <actuator>
        <motor name="motor_1" joint="rotate_1" gear="1" ctrlrange="-100 100"/>
        <motor name="motor_2" joint="rotate_2" gear="1" ctrlrange="-100 100"/>
      </actuator>

        <sensor>
            <framepos objtype="site" objname="end"/>
            <jointpos joint = "rotate_1"/>
            <jointvel joint = "rotate_1"/>
            <jointpos joint = "rotate_2"/>
            <jointvel joint = "rotate_2"/>
        </sensor>
```

3. APPENDIX

Below is the code implemented in python to model the mechanism:

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


# Define geometric constants for the 2R mechanism and pulleys
# R1, R2: Diameter of the two pulleys
# a, b, c: Lengths of the linkage segments
R1, R2, a, b, c = 0.019, 0.012, 0.052, 0.044, 0.052


MAX_SIM_TIME = 5.0  # Maximum simulation time in seconds
AMP_1, FREQ_1, BIAS_1 = math.radians(50), 3.53, math.radians(41.4)
AMP_2, FREQ_2, BIAS_2 = math.radians(56.39), 2.85, math.radians(43.7)

# PD Control Gains (Kp and Kd)
KP = 11.0  # Proportional Gain
KD = 0.01   # Derivative Gain

sensor_pos_x = []
sensor_pos_z = []

# New logs for joint control analysis
time_history = []
q1_actual = []
q1_desired = []
q2_actual = []
q2_desired = []

def qdes(t, AMP, FREQ, BIAS):
    """Desired position for Joint """
    return AMP * np.sin(FREQ * t) + BIAS

def qd_des(t, AMP, FREQ):
    """Desired velocity for Joint """
    return AMP * FREQ * np.cos(FREQ * t)

def pd_control(q, qd, q_des, qd_des, kp, kd):
    """
    Standard PD controller to calculate required torque (tau).
    tau = Kp * (q_des - q) + Kd * (qd_des - qd)
    """
    position_error = q_des - q
    velocity_error = qd_des - qd
    tau = kp * position_error + kd * velocity_error
    return tau


def generate_model_xml(R1: float, R2: float, a: float, b: float, c:
float):
    return f"""
        <mujoco model="2R_tendon_planar">

        <asset>
            <texture type="skybox" builtin="gradient" rgb1="1 1 1" rgb2="0.5
0.5 0.5" width="265" height="256"/>
        </asset>

        <option timestep="1e-4" integrator="RK4" gravity="0 0 -9.81"/>

        <worldbody>

            <body name="wall" pos="0 0 0" euler="0 90 0">
                <geom type="box" size="0.05 0.05 0.005" pos="0 0 0" rgba="0.5
0.5 0.5 1"/>
                <site name="start_green" pos="{R1 / 2} 0 0" type="sphere"
size="0.001" rgba="1 0 0 0.5" />
                <site name="start_red" pos="{-R1 / 2} 0 0" type="sphere"
size="0.001" rgba="0 1 0 0.5" />
            </body>

            <body name="green_mid" pos="{a + c / 2} 0 0">
                <site name="mid_green" pos="0 0 0" type="sphere"
size="0.001"/>
                <joint name="mid_joint_x_green" type="slide" axis="1 0 0"/>
                <joint name="mid_joint_y_green" type="slide" axis="0 0 1"/>
                <geom type="sphere" size="0.001" mass="0.0001" rgba="0 1 0
0.5" contype="0"/>
            </body>

            <body name="red_mid" pos="{a + c / 2} 0 0">
                <site name="mid_red" pos="0 0 0" type="sphere" size="0.001"/>
                <joint name="mid_joint_x_red" type="slide" axis="1 0 0"/>
                <joint name="mid_joint_y_red" type="slide" axis="0 0 1"/>
                <geom type="sphere" size="0.001" mass="0.0001" rgba="1 0 0
0.5" contype="0"/>
            </body>

            <body name="link_end" pos="{a + b + c} 0 0">
                <site name="end_link" pos="0 0 0" type="sphere" size="0.001"/>
                <joint name="end_x" type="slide" axis="1 0 0"/>
                <joint name="end_y" type="slide" axis="0 0 1"/>
                <geom type="box" size="0.002 0.002 {R2 / 2}" rgba="1 1 0 0.5"
mass="0.001" contype="0"/>
            </body>

            <body name="link_1" pos="0 0 0">
                <geom type="cylinder" pos="{a / 2} 0 0" size="0.0002 {a / 2}"
euler="0 90 0" rgba="0 1 1 0.5" contype="0"/>

                <body name="link_2" pos="{a} 0 0">
                    <joint name="rotate_1" type="hinge" axis="0 1 0"
stiffness="0" springref="0" damping="0.1"/>
                    <geom type="cylinder" pos="{c / 2} 0 0" size="0.0002 {c /
2}" euler="0 90 0" rgba="0 1 1 0.5" contype="0"/>

                    <geom name="pulley_r1" type="cylinder" size="{R1 / 2}
0.01" pos="0 0 0" euler="90 0 0" rgba="1 0 0 0.5" contype="0"/>
                    <site name="side_r1_green" pos="0 0 {-R1 / 2 - 0.0001}"
type="sphere" size="0.001"/>
                    <site name="side_r1_red" pos="0 0 {R1 / 2 + 0.0001}"
type="sphere" size="0.001"/>
                    <site name="pulley_r1_center" pos="0 0 0" type="sphere"
size="0.001"/>

                    <body name="link_3" pos="{c} 0 0">
                        <joint name="rotate_2" type="hinge" axis="0 1 0"
stiffness="0" springref="0" damping="0.1"/>
                        <geom type="cylinder" pos="{b / 2} 0 0" size="0.0002
{b / 2}" euler="0 90 0" rgba="0 1 1 0.5" contype="0"/>

                        <site name="end_green" pos="{b} 0 {R2 / 2}"
type="sphere" size="0.001"/>
                        <site name="end_red" pos="{b} 0 {-R2 / 2}"
type="sphere" size="0.001"/>

                        <geom name="pulley_r2" type="cylinder" size="{R2 / 2}
0.01" pos="0 0 0" euler="90 0 0" rgba="0 1 0 0.5" contype="0"/>
                        <site name="side_r2_green" pos="0 0 {R2 / 2 + 0.0001}"
type="sphere" size="0.001"/>
                        <site name="side_r2_red" pos="0 0 {-R2 / 2 - 0.0001}"
type="sphere" size="0.001"/>
                        <site name="pulley_r2_center" pos="0 0 0"
type="sphere" size="0.001"/>

                        <site name="end" pos="{b} 0 0" type="sphere"
size="0.001"/>
                    </body>
```

```xml
                </body>
            </body>
        </worldbody>

        <tendon>
            <spatial name="tendon_green" width="0.001" rgba="1 0 0 0.5"
stiffness="0.1" damping="0.1" springlength="0.005">
                <site site="start_green"/>
                <geom geom="pulley_r1" sidesite="side_r1_green"/>
                <site site="mid_green"/>
                <geom geom="pulley_r2" sidesite="side_r2_green"/>
                <site site="end_green"/>
            </spatial>

            <spatial name="tendon_red" width="0.001" rgba="0 1 0 0.5"
stiffness="0.1" damping="0.1" springlength="0.005">
                <site site="start_red"/>
                <geom geom="pulley_r1" sidesite="side_r1_red"/>
                <site site="mid_red"/>
                <geom geom="pulley_r2" sidesite="side_r2_red"/>
                <site site="end_red"/>
            </spatial>
        </tendon>

        <equality>
            <weld site1="end" site2="end_link" torquescale="100"/>

            <connect site1="mid_green" site2="pulley_r1_center"/>
            <connect site1="mid_green" site2="pulley_r2_center"/>
            <connect site1="mid_red" site2="pulley_r1_center"/>
            <connect site1="mid_red" site2="pulley_r2_center"/>
        </equality>

        <actuator>
            <motor name="motor_1" joint="rotate_1" gear="1" ctrlrange="-100
100"/>
            <motor name="motor_2" joint="rotate_2" gear="1" ctrlrange="-100
100"/>
        </actuator>

        <sensor>
            <framepos objtype="site" objname="end"/>
            <jointpos joint = "rotate_1"/>
            <jointvel joint = "rotate_1"/>
            <jointpos joint = "rotate_2"/>
            <jointvel joint = "rotate_2"/>
        </sensor>
    </mujoco>
```

```python
    """

def main():
    # Generate the XML model
    xml = generate_model_xml(R1, R2, a, b, c)

    # Load the model and create the data structure
    try:
        model = mujoco.MjModel.from_xml_string(xml.encode("utf-8"))
    except Exception as e:
        print(f"Error loading MuJoCo model: {e}")
        sys.exit(1)

    data = mujoco.MjData(model)

    data.qpos[6] = BIAS_1 # rotate_1
    data.qpos[7] = BIAS_2 # rotate_2

    mujoco.mj_forward(model, data)

    TARGET_FRAMERATE = 60.0
    FRAME_TIME = 1.0 / TARGET_FRAMERATE


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

            viewer.cam.azimuth = 90
            viewer.cam.elevation = -10
            viewer.cam.distance = 0.5
            viewer.cam.lookat[:] = [0.05, 0, 0.05]

            while viewer.is_running():
                t = data.time

                # Stop simulation after MAX_SIM_TIME for clean plotting
                if t > MAX_SIM_TIME:
                    viewer.close()
                    break

                # READ SENSOR DATA
                # Joint 1 Position (q1) and Velocity (q1d)
                q1 = data.sensordata[3]
                q1d = data.sensordata[4]
                # Joint 2 Position (q2) and Velocity (q2d)
                q2 = data.sensordata[5]
                q2d = data.sensordata[6]
                # CALCULATE DESIRED TRAJECTORY
                q1_des = qdes(t, AMP_1, FREQ_1, BIAS_1)
                q1d_des = qd_des(t, AMP_1, FREQ_1)

                q2_des = qdes(t, AMP_2, FREQ_2, BIAS_2)
                q2d_des = qd_des(t, AMP_2, FREQ_2)

                # CALCULATE CONTROL TORQUE using PD controller
                tau1 = pd_control(q1, q1d, q1_des, q1d_des, KP, KD)
                tau2 = pd_control(q2, q2d, q2_des, q2d_des, KP, KD)

                # APPLY CONTROL INPUT
                data.ctrl[0] = tau1 # motor_1
                data.ctrl[1] = tau2 # motor_2

                # LOG END-EFFECTOR POSITION
                sensor_pos = data.sensordata[:3]
                sensor_pos_x.append(sensor_pos[0])
                sensor_pos_z.append(sensor_pos[2])

                # Log Joint Data for plotting
                time_history.append(t)
                q1_actual.append(q1)
                q1_desired.append(q1_des)
                q2_actual.append(q2)
                q2_desired.append(q2_des)

                # Simulation step loop
                simstart = data.time
                while data.time - simstart < 1.0/60.0:
                    mujoco.mj_step(model, data)

                # Update the viewer's scene
                viewer.sync()

                # Sleep to maintain the real-time simulation rate
                time.sleep(max(0, FRAME_TIME - (data.time - simstart)))

    except Exception as e:
        # Handle potential errors in the simulation process
        print(f"Error during simulation: {e}")
        sys.exit(1)

    # PLOTTING SECTION: Joint Positions and Errors
    if len(time_history) > 50:

        q1_error = np.array(q1_desired) - np.array(q1_actual)
        q2_error = np.array(q2_desired) - np.array(q2_actual)
```

```
        # Create a figure with three subplots
        plt.figure(figsize=(18, 5))

        # PLOT 1: Joint 1 Position (Actual vs Desired)
        plt.subplot(1, 3, 1)
        plt.plot(time_history, q1_actual, '-', linewidth=2, label='q1
(actual)')
        plt.plot(time_history, q1_desired, '--', linewidth=2, label='q1
(desired)')
        plt.title('Joint 1 Position', fontsize=12, fontweight='bold')
        plt.legend(loc='upper right')
        plt.xlabel('Time, s')
        plt.ylabel('Position, rad')
        plt.grid(True)

        # PLOT 2: Joint 2 Position (Actual vs Desired)
        plt.subplot(1, 3, 2)
        plt.plot(time_history, q2_actual, '-', linewidth=2, label='q2
(actual)')
        plt.plot(time_history, q2_desired, '--', linewidth=2, label='q2
(desired)')
        plt.title('Joint 2 Position', fontsize=12, fontweight='bold')
        plt.legend(loc='upper right')
        plt.xlabel('Time, s')
        plt.ylabel('Position, rad')
        plt.grid(True)

        # PLOT 3: Joint Position Errors
        plt.subplot(1, 3, 3)
        plt.plot(time_history, q1_error, '-', linewidth=2, label='q1 error')
        plt.plot(time_history, q2_error, '-', linewidth=2, label='q2 error')
        plt.title('Joint Position Errors', fontsize=12, fontweight='bold')
        plt.legend(loc='upper right')
        plt.xlabel('Time, s')
        plt.ylabel('Error, rad')
        plt.grid(True)


        # Trajectory plot
        plt.figure(figsize=(8, 8))
        plt.clf()
        plt.plot(sensor_pos_x[50:], sensor_pos_z[50:], '-', linewidth=2,
label='End-EffectorTrajectory')
        plt.title('End-effector trajectory', fontsize=12, fontweight='bold')
        plt.legend(loc='upper left')
        plt.xlabel('X-Axis [m]')
        plt.ylabel('Z-Axis [m]')
        plt.axis('equal')
```

```python
        plt.grid(True)

        # Show all plots with tight layout
        plt.tight_layout()
        plt.show()

    else:
        print("Not enough data points collected to plot the joint analysis.")


if __name__ == "__main__":
    main()
```

4. RESULTS

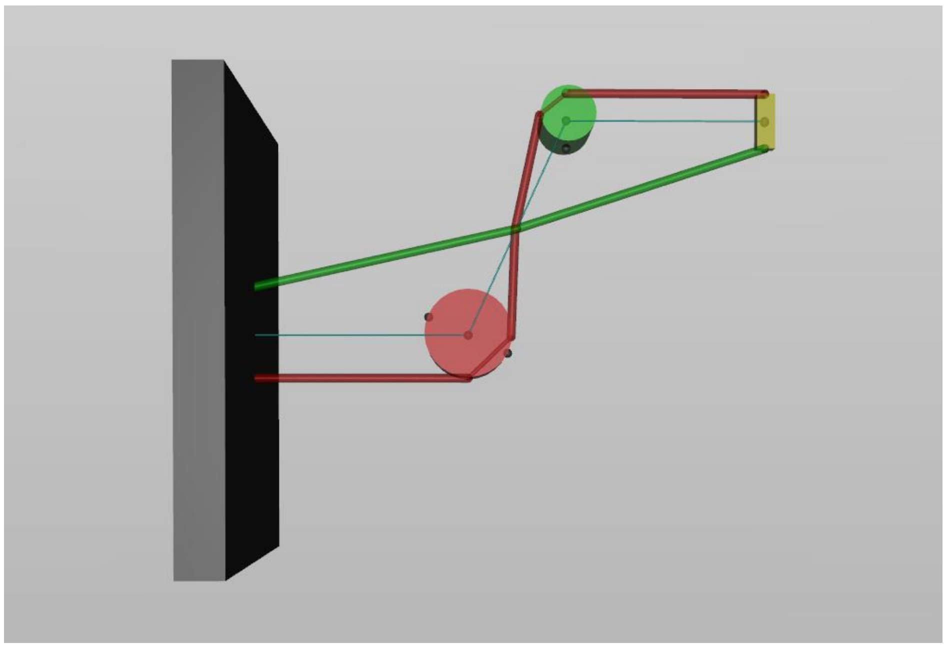

Figure 2: Tendon connected 2R planar mechanism in simulation

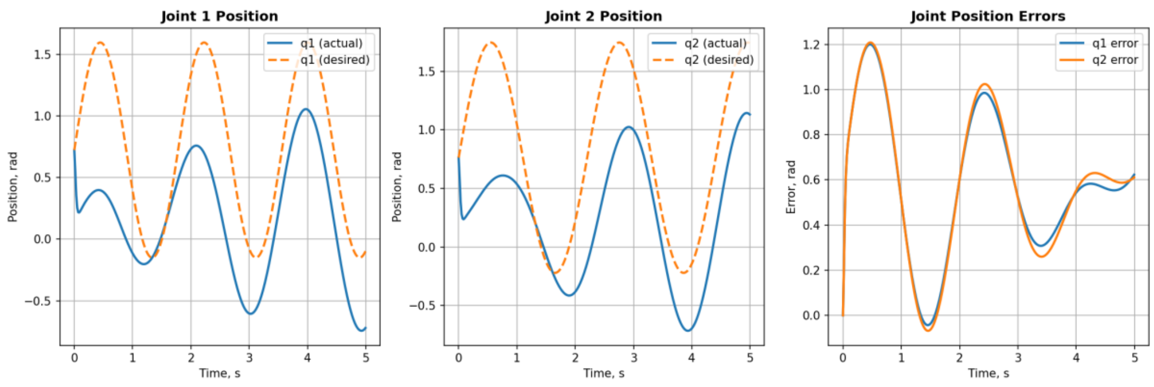

Figure 3: Joint_1 position, Joint_2 position and Joint position errors

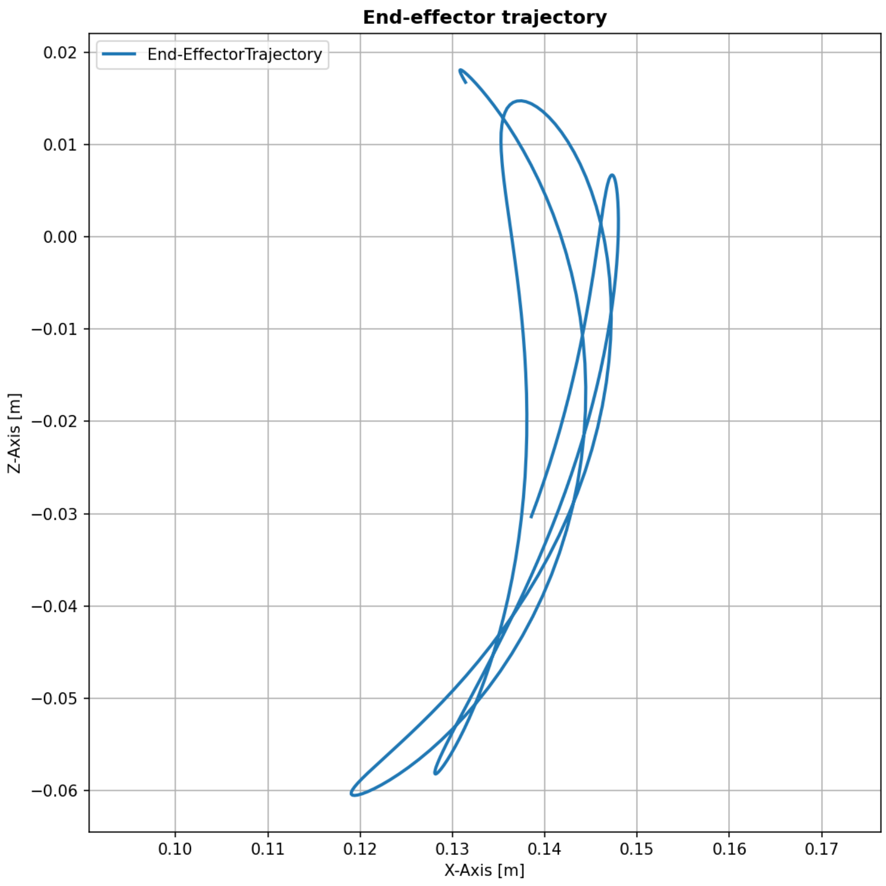

Figure 4: End-effector trajectory

5. CONCLUSION

During the work, the mechanism model was modified: actuators were added for both joints, along with sensors to obtain their positions and velocities. A PD controller was implemented to track the desired generalized coordinate values. However, the experimental results showed that the model does not follow the target trajectories with sufficient accuracy, exhibiting noticeable tracking errors. Therefore, future work will focus on improving the controller to enhance tracking performance and system stability.