THE MINISTRY OF SCIENCE AND HIGHER EDUCATION OF
THE RUSSIAN FEDERATION

ITMO University
(ITMO)

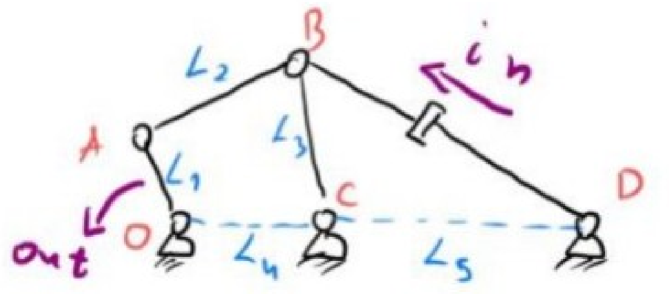
Faculty of Control Systems and Robotics

Simulation of Robotic Systems
task 4

Ali Ahmad
ISU 475789

Mechanism Modeling and Simulation Environment

Optimus' knee closed-chain mechanism:

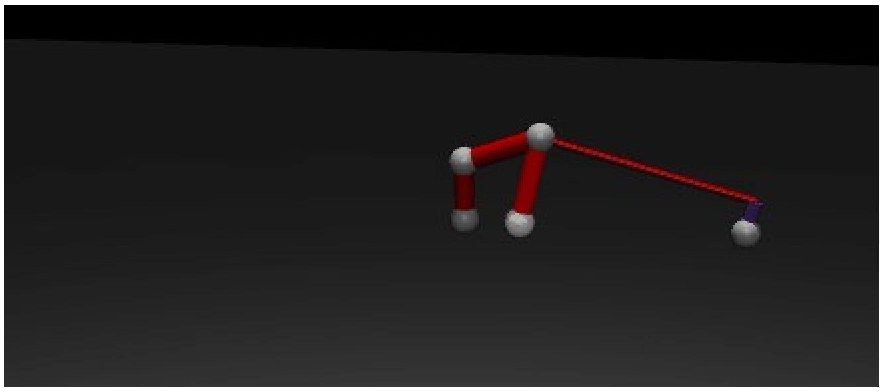


The parameters of the figure are shown in the following table:

Parameter	L_1	L_2	L_3	L_4	L_5
Value (m)	0.057	0.0741	0.0855	0.057	0.285

First, the closed-loop mechanism was mathematically defined using the MuJoCo modeling language, specified within an XML file **.** This file served as the blueprint, defining all rigid bodies (links), joints (allowing rotation), and the required equality constraints necessary to form the complex closed-loop structure.

The simulation execution began with a dedicated Python script designed to interface with the MuJoCo physics engine. The script first loads the mechanism model from the XML file and initializes the simulation data structure, which holds the current state, controls, and sensor outputs.



2. Trajectory Generation and Control Implementation

Reference Trajectory

A precise reference signal was mathematically generated to define the desired motion for the main actuated joint, Joint 'O'. This signal is a simple sinusoidal function:

$$q_{des} = AMP \, . \sin(FREQ \, . \, t) + BIAS$$

Using the prescribed parameters (Amplitude: , Frequency: , Bias:), this signal represents the target angular position the system is intended to track across the entire simulation period. The derivative of this signal, , provides the target velocity necessary for effective derivative control.

Using the prescribed parameters :

Amplitude	Frequency	Bias
$52.67\, deg$	$4\, HZ$	$3.6\, deg$

1. Convert Parameters to Radians

$$Amplitude = 52.67 * 180\pi \approx 0.919\, rad$$

$$Frequency = 3.6 * 180\pi \approx 0.063\, rad$$

2. Calculate Angular Frequency

The angular frequency (ω) is determined from the frequency in Hertz (FREQ)

$$\omega = 2\pi * FREQ$$
$$\omega = 2\pi * (4\, Hz) = 8\pi\, rad/s \approx 25.13\, rad/s$$

Proportional-Derivative (PD) Controller

A simple Proportional-Derivative (PD) feedback controller was implemented to generate the required torque commands () for the actuator (`act_O`) on Joint 'O'. The controller determines the control effort by calculating the weighted sum of two error term

$$\tau_O = K_P . (q_{des} - q_{curr}) + K_D . (\dot{q_{des}} - \dot{q_{curr}})$$

where:

K_P:Proportional Gain

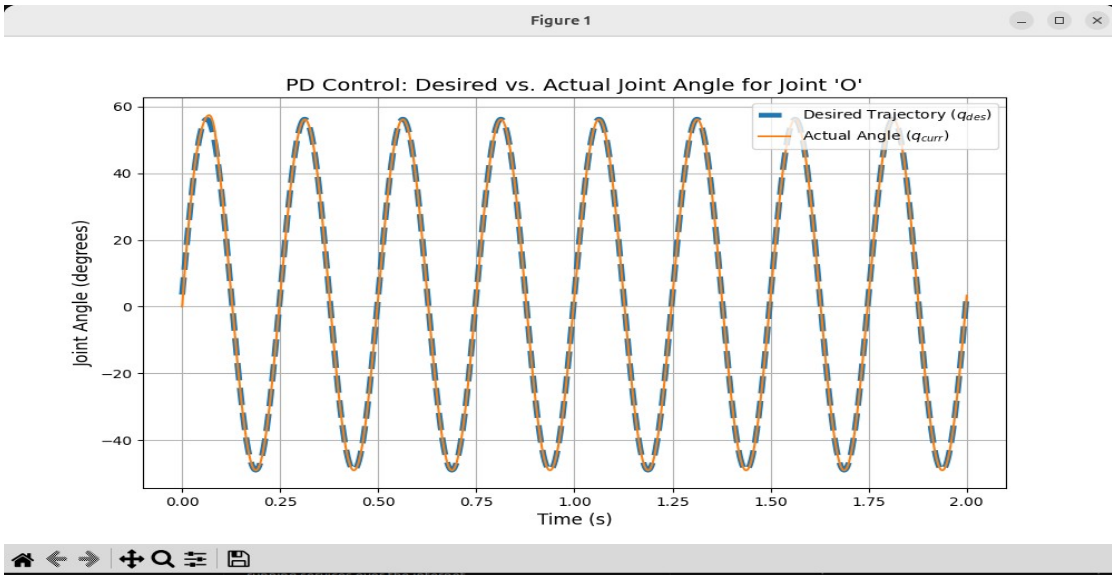
K_D:Derivative Gain

3. Data Collection and Visualization

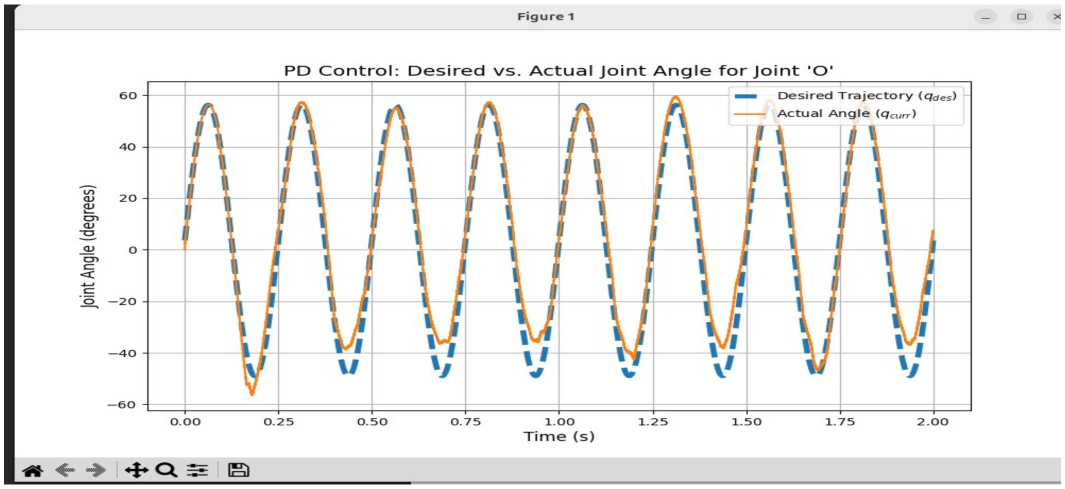
The simulation loop continuously steps the physics engine forward. In each step, the script reads the current joint position (q_{curr}) and velocity ($\dot{q}_{curr}$) from MuJoCo's sensor. These actual values, along with the desired trajectory (q_{des}) and the simulation time, are collected into separate history lists.

Upon completion of the simulation, a time-series plot is generated using the `matplotlib` library. This plot visually compares the Desired Trajectory (dashed line) against the Actual Angle (solid line), allowing for immediate assessment of the PD controller's tracking accuracy, response time, and steady-state error

for $K_P = 200 \quad K_D = 1$



for $K_P = 200 \quad K_D = 10$



The periodic motion observed throughout the simulation demonstrates the effective transmission of the linear actuator's control input through the closed-chain mechanical structure, successfully generating the desired output at point A.

In conclusion, the results firmly confirm that the actuation strategy and the linkage geometry are highly suitable for converting a linear actuation input into a reliable periodic rotational output.

XML_file

```xml
<mujoco model="Optimus">
<option gravity="0 0 -9.81" timestep="0.0001" solver="Newton" iterations="200" tolerance="1e-8"/>
<default>
<site rgba="0 0 1 1"/>
</default>
<worldbody>
<light pos="0 0 3"/>
<geom type="plane" size="5 5 0.1" rgba=".5 .5 .5 .7"/>
<body pos="0 0 0.5">
<joint name="O" axis="0 1 0" range="-90 90" limited="true" />
<geom name="O" type="sphere" size="0.015" rgba="1 1 1 1"/>
<geom name="L1" type="cylinder" pos="0 0 0.0285" size="0.01 0.0285" rgba="1 0 0 1"/>
<body pos="0 0 0.057" euler="0 0 0">
<joint name="A" axis="0 1 0" range="-90 90" limited="true"/>
<geom name="A" type="sphere" pos="0 0 0" size="0.015" rgba="1 1 1 1"/>
<geom name="L2" type="cylinder" pos="0 0 0.03705" size="0.01 0.03705" rgba="1 0 0 1"/>
<site name="s1" pos="0 0 0.0741" size="0.005"/> <!-- Connection site 1 -->
</body>
</body>
<body name="H" pos="0.057 0 0.5">
<joint name="C" axis="0 1 0" range="-90 90" limited="true"/>
<geom name="C" type="sphere" size="0.015" rgba="1 1 1 1"/>
<geom name="L3" type="cylinder" size="0.01" fromto="0 0 0 0 0 0.0855" rgba="1 0 0 1"/>
<geom type="sphere" size="0.005" mass="0.002" rgba="1 0 0 1"/>
<site name="s2" pos="0 0 0.0855" size="0.005"/> <
<body pos="0 0 0.0855">
<geom name="B" type="sphere" pos="0 0 0" size="0.015" rgba="1 1 1 1"/>
</body>
</body>
```

```
<body pos="0.285 0 0.5" euler="0 -65 0">
<joint name="D" axis="0 1 0" range="-90 180" limited="true"/>
<geom name="D" type="sphere" size="0.015" rgba="1 1 1 1"/>
<site name="sD" pos="0 0 0"/> <!-- Target site for the actuator -->
</body>
</worldbody>
<equality>
<connect site1="s1" site2="s2"/>
</equality>
<sensor>
    <jointpos name="pos_O" joint="O"/>
    <jointvel name="vel_O" joint="O"/>
</sensor>
<actuator>
    <motor name="act_O" joint="O" ctrlrange="-100 100" ctrllimited="true"/>
    <general name="act_B_to_D"
             cranksite="s2"
             slidersite="sD"
             ctrlrange="0 0.2"
             cranklength="0.1"
             />
</actuator>
</mujoco>
```

python_ code:

```
import mujoco

import mujoco.viewer

import numpy as np

import os

import matplotlib.pyplot as plt

# --- Setup and Initialization ---

SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))

XML_MODEL_PATH = os.path.join(SCRIPT_DIR, 'n.xml')

# Load model and initialize data

model = mujoco.MjModel.from_xml_path(XML_MODEL_PATH)

data = mujoco.MjData(model)

# Get element IDs

act_O_cid = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_ACTUATOR, 'act_O')

act_B_to_D_cid = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_ACTUATOR, 'act_B_to_D')

pos_O_sid = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_SENSOR, 'pos_O')

vel_O_sid = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_SENSOR, 'vel_O')

# --- Control Parameters ---

DEG2RAD = np.pi / 180.0

RAD2DEG = 180.0 / np.pi

# Trajectory parameters

AMP_DEG = 52.67

BIAS_DEG = 3.6

FREQ_HZ = 4.0

AMP_RAD = AMP_DEG * DEG2RAD

BIAS_RAD = BIAS_DEG * DEG2RAD

OMEGA = 2 * np.pi * FREQ_HZ

# PD Gains

KP = 200.0
```

```
KD = 10.0

# Simulation settings

sim_time = 2.0

FIXED_CTRL_D = 0.1

# --- Data Collection Setup ---

time_history = []

qdes_history = []

qact_history = []

# --- Simulation Loop ---

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

    # Set camera for a good view

    viewer.cam.azimuth = 135

    viewer.cam.elevation = -10

    viewer.cam.distance = 2.0

    print("Simulation started.")

    while viewer.is_running() and data.time < sim_time:

        q_O = data.sensordata[pos_O_sid] if pos_O_sid != -1 else 0.0

        q_O_dot = data.sensordata[vel_O_sid] if vel_O_sid != -1 else 0.0

        qdes_O = AMP_RAD * np.sin(OMEGA * data.time) + BIAS_RAD

        qdes_O_dot = AMP_RAD * OMEGA * np.cos(OMEGA * data.time)

        pos_error = qdes_O - q_O

        vel_error = qdes_O_dot - q_O_dot

        tau_O = (KP * pos_error) + (KD * vel_error)

        if act_O_cid != -1:

            data.ctrl[act_O_cid] = tau_O

        if act_B_to_D_cid != -1:

            data.ctrl[act_B_to_D_cid] = FIXED_CTRL_D

        mujoco.mj_step(model, data)

        viewer.sync()
        # Data logging
        time_history.append(data.time)
        qdes_history.append(qdes_O * RAD2DEG)
        qact_history.append(q_O * RAD2DEG)
print("Simulation finished. Generating plot...")
# --- Plotting Results ---
try:
    plt.figure(figsize=(10, 6))
    plt.plot(time_history, qdes_history, label='Desired Trajectory ($q_{des}$)', color='tab:blue',
linestyle='--', linewidth=4)
    plt.plot(time_history, qact_history, label='Actual Angle ($q_{curr}$)', color='tab:orange')
    plt.title("PD Control: Desired vs. Actual Joint Angle for Joint 'O'", fontsize=14)
    plt.xlabel("Time (s)", fontsize=12)
    plt.ylabel("Joint Angle (degrees)", fontsize=12)
    plt.grid(True)
    plt.legend(loc='upper right')
    plt.show()
except Exception as e:
    print(f"\nError generating plot. Error: {e}")
print("Script finished.")
```