THE MINISTRY OF SCIENCE AND HIGHER EDUCATION

OF THE RUSSIAN FEDERATION

ITMO University

(ITMO)

Faculty of Control Systems and Robotics

Project Report

For the Subject:

Robot Programming

Project Title

Ball Balancing Using a Robotic Arm in Simulation

Student:

ISU No. 476937– Nagham Sleman

ISU No. 503259 – Hazem Afif

Date: 03.01.2026

1. Introduction:

Ball balancing is a well-known problem in robotics because it represents an unstable system that is difficult to control. Even small changes in the plate orientation can cause the ball to move, which makes this task a good example for studying robot dynamics and control behavior.

In this project, we simulate a ball balancing system using a robotic arm. The arm holds a flat plate at its end, and a ball is placed on the plate. By tilting the plate through the arm's joints, the motion of the ball can be influenced and observed.

The main goal of the project is not to build a perfect controller, but to create a realistic and stable simulation model that shows how the ball reacts to the plate motion. The system is implemented in the MuJoCo simulation environment, which allows accurate modeling of rigid bodies, gravity, and contact forces.

This simulation-based approach helps us understand the behavior of the system and provides a solid basis for further control experiments, without the need for a real robotic platform.

Project Structure and Team Contribution

This report is structured into two main parts.
The first part focuses on the simulation modeling and physical validation of the robotic arm and ball–plate system.
The second part presents the control design and performance evaluation.

Each part is developed independently by a team member, while maintaining a consistent system description and experimental setup.

2. System Kinematics and Modeling

The robotic arm used in this project is modeled as a serial manipulator with two rotational degrees of freedom.
Each joint allows rotation around a single axis, enabling the arm to orient the plate attached at its end-effector.

The first joint provides rotation around the vertical axis, while the second joint controls the tilt of the plate.

By adjusting the angles of these joints, the orientation of the plate changes, which directly affects the motion of the ball placed on top of it.

The kinematic structure of the arm is kept intentionally simple. This allows clear observation of how joint motions influence the plate orientation without introducing unnecessary complexity.

3. Kinematic Equations

The configuration of the robotic arm is described by the joint angle vector:
$\theta = [\theta_1, \theta_2]$
where θ_1 and θ_2 represent the angles of the first and second joints, respectively.
The orientation of the plate is directly determined by the second joint angle.
When θ_2 changes, the plate tilts accordingly, creating a gravitational component that causes the ball to roll on the surface.
In this project, inverse kinematics is not required, since the plate orientation is controlled directly through joint commands.
This simplifies the control process and allows direct mapping between joint motion and plate tilt.

Plate Tilt and Ball Motion Relationship

When the plate is tilted by an angle θ_2, a component of the gravitational force acts along the surface of the plate.
This component is responsible for accelerating the ball in the direction of the tilt.

Assuming small tilt angles, the acceleration of the ball along the plate can be approximated as:

$$a \approx g \cdot sin(\theta_2)$$

where g represents the gravitational acceleration.

For small angles, the approximation $sin(\theta_2) \approx \theta_2$ holds, which indicates that the ball acceleration is approximately proportional to the plate tilt angle.
As a result, increasing the joint angle θ_2 leads to a stronger acceleration of the ball along the plate surface.

This simplified relationship is sufficient for analyzing the qualitative behavior of the system and validating the physical consistency of the simulation model.

4. Simulation Model Description (XML Implementation)

The simulation model is defined using an XML file compatible with the MuJoCo physics engine.
This file describes the physical structure of the system, including the robotic arm, the plate, and the ball, as well as their interactions.

The robotic arm is modeled as a hierarchy of rigid bodies connected by hinge joints.
Each joint has a limited range of motion to prevent unrealistic rotations and to maintain numerical stability.
An example of the joint definition used in the model is shown below.

```
<joint name="joint_x"
       type="hinge"
       axis="1 0 0"
       limited="true"
       range="-45 45"/>
```

The plate is rigidly attached to the second link of the arm.
This design choice ensures that the plate orientation follows the joint motion directly, without introducing additional degrees of freedom.

To keep the ball within the plate boundaries during experiments, small vertical walls are added around the plate edges.
These walls prevent the ball from falling while preserving realistic contact dynamics.

The ball itself is modeled as a free rigid body using a free joint, allowing it to move naturally under the influence of gravity and contact forces.
A simplified representation of the ball definition is shown below.

```
<body name="ball">
    <joint type="free"/>
    <geom type="sphere" size="0.02"/>
</body>
```

Physical parameters such as friction, gravity, and joint damping are carefully selected to ensure stable and realistic simulation behavior.

5. Simulation Visualization

Before performing any experiments, a Python script was used to load and visualize the simulation model.
This step was intended to verify the correctness of the XML implementation and to ensure that the robotic arm, plate, and ball behave as expected under gravity.

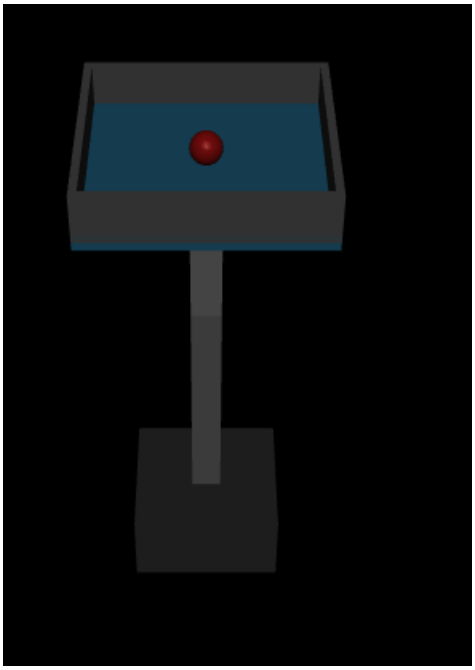
The script initializes the MuJoCo model, advances the simulation step by step, and displays the system in real time using the MuJoCo viewer.
This visualization stage helped confirm that the joints move correctly, the plate orientation follows the arm motion, and the ball interacts properly with the plate surface.

The Python Code:

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

# ---------------- Model ----------------
MODEL_PATH = r"C:\Users\Nagham\Documents\mujoco_models\ball.xml"

model = mujoco.MjModel.from_xml_path(MODEL_PATH)
data = mujoco.MjData(model)

# ---------------- IDs ----------------
ball_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "ball")
plate_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "plate")
joint_x_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_JOINT, "joint_x")

# ---------------- Simulation settings ----------------
sim_time = 5.0
dt = model.opt.timestep
steps = int(sim_time / dt)

# Logs
time_log = []
ball_rel_x_log = []
joint_angle_log = []

# Reset simulation
mujoco.mj_resetData(model, data)

# ---------------- Simulation loop ----------------
```

```
with mujoco.viewer.launch_passive(model, data) as viewer:
    for i in range(steps):
        t = i * dt

        # Open-loop smooth tilt
        if 1.0 < t < 2.0:
            data.ctrl[1] = 0.6 * (t - 1.0)
        elif 2.0 <= t < 3.0:
            data.ctrl[1] = 0.6 * (3.0 - t)
        else:
            data.ctrl[1] = 0.0

        mujoco.mj_step(model, data)
        viewer.sync()

        # Logging
        time_log.append(t)
        joint_angle_log.append(data.qpos[joint_x_id])
        ball_rel_x_log.append(
            data.xpos[ball_id][0] - data.xpos[plate_id][0]
        )

        time.sleep(dt)

# ---------------- Plot 1: Arm response ----------------
plt.figure()
plt.plot(time_log, joint_angle_log)
plt.xlabel("Time [s]")
plt.ylabel("Joint X Angle [rad]")
plt.title("Arm Joint Angle Response")
plt.grid()
plt.show()

# ---------------- Plot 2: Ball response ----------------
plt.figure()
plt.plot(time_log, ball_rel_x_log)
plt.xlabel("Time [s]")
plt.ylabel("Ball X Position Relative to Plate [m]")
plt.title("Ball Position Relative to Plate")
plt.grid()
plt.show()
```

The model:



6. Experimental Motion and Data Collection (OPEN LOOP)

After validating the simulation through visualization, an experimental script was implemented to analyze the system behavior quantitatively.
In this experiment, open-loop control signals were applied to the robotic arm joints in order to tilt the plate smoothly over time.
During the simulation, the joint angles and the ball position relative to the plate were recorded.
These measurements were later used to generate plots illustrating the arm response and the resulting ball motion.
This experimental setup allows evaluation of the physical interaction between the arm, the plate, and the ball, and serves as a baseline for further control design.

The Code:

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

```python
# ---------------- Model ----------------
MODEL_PATH = r"C:\Users\Nagham\Documents\mujoco_models\ball.xml"

model = mujoco.MjModel.from_xml_path(MODEL_PATH)
data = mujoco.MjData(model)

# ---------------- IDs ----------------
ball_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "ball")
plate_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "plate")
joint_x_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_JOINT, "joint_x")

# ---------------- Simulation settings ----------------
sim_time = 5.0
dt = model.opt.timestep
steps = int(sim_time / dt)

# Logs
time_log = []
joint_angle_log = []
ball_rel_x_log = []

# Reset simulation
mujoco.mj_resetData(model, data)

# ---------------- Simulation loop ----------------
with mujoco.viewer.launch_passive(model, data) as viewer:
    for i in range(steps):
        t = i * dt

        # Bidirectional smooth motion
        if 1.0 < t < 2.0:
            data.ctrl[1] = 0.8 * (t - 1.0)
        elif 2.0 <= t < 3.0:
            data.ctrl[1] = 0.8 * (3.0 - t)
        elif 3.0 < t < 4.0:
            data.ctrl[1] = -0.8 * (t - 3.0)
        elif 4.0 <= t < 5.0:
            data.ctrl[1] = -0.8 * (5.0 - t)
        else:
            data.ctrl[1] = 0.0

        mujoco.mj_step(model, data)
        viewer.sync()
```

```
        # Logging
        time_log.append(t)
        joint_angle_log.append(data.qpos[joint_x_id])
        ball_rel_x_log.append(
            data.xpos[ball_id][0] - data.xpos[plate_id][0]
        )

        time.sleep(dt)

# ---------------- Plot 1: Arm joint response ----------------
plt.figure()
plt.plot(time_log, joint_angle_log)
plt.xlabel("Time [s]")
plt.ylabel("Joint X Angle [rad]")
plt.title("Arm Joint Angle – Bidirectional Motion")
plt.grid()
plt.show()

# ---------------- Plot 2: Ball response ----------------
plt.figure()
plt.plot(time_log, ball_rel_x_log)
plt.xlabel("Time [s]")
plt.ylabel("Ball X Position Relative to Plate [m]")
plt.title("Ball Response to Bidirectional Plate Tilt")
plt.grid()
plt.show()
```

Experimental Results and Discussion (OPEN LOOP)

Arm Joint Angle Response

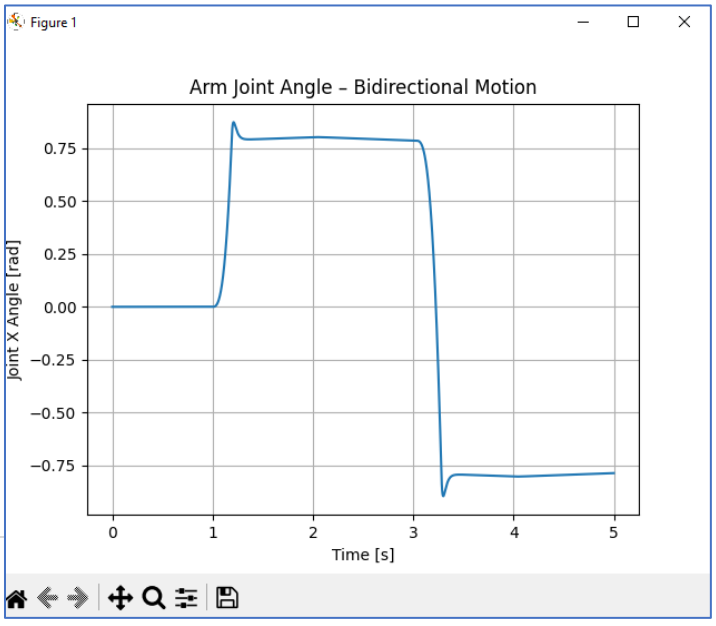


Figure 1 shows the angular response of the robotic arm joint responsible for tilting the plate during the bidirectional motion experiment.
Initially, the joint angle remains at zero, indicating that the plate is in a horizontal position and no motion is applied.

Between 1 s and 3 s, a smooth positive command is applied to the joint, causing the

plate to tilt in one direction.
The joint angle increases gradually and reaches a steady value, demonstrating that the actuator and joint dynamics respond smoothly without excessive oscillations.

After 3 s, the control input is reversed, causing the joint angle to move in the opposite direction. This results in a negative joint angle, corresponding to a tilt of the plate toward the opposite side. The smooth transition between positive and negative angles confirms stable joint behavior and proper actuator configuration.

Overall, this plot verifies that the robotic arm can generate controlled and repeatable plate tilting motions in both directions.

Ball Position Response to Plate Tilt

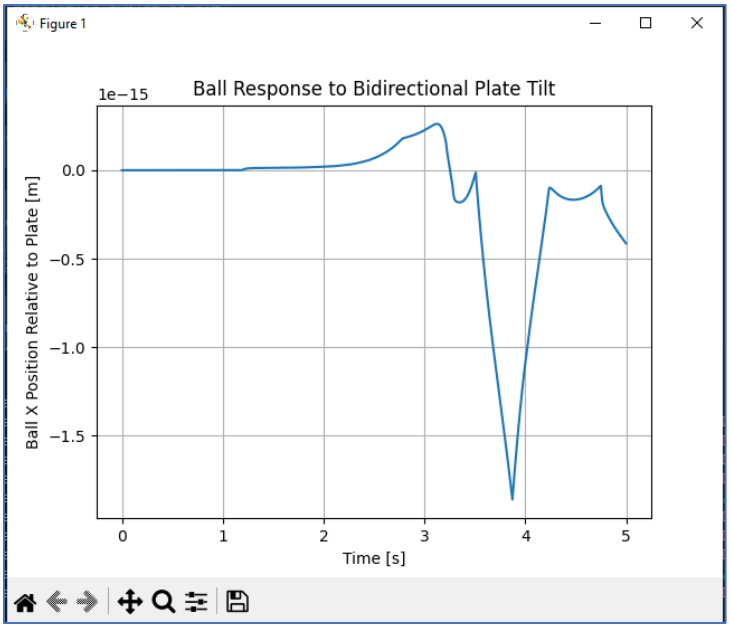


Figure 2 illustrates the ball's position relative to the plate along the x-axis during the same bidirectional tilting experiment.
At the beginning, the ball remains nearly stationary, since the plate is horizontal and no gravitational component acts along the surface.

As the plate tilts in the positive direction, the ball begins to roll smoothly along the plate due to gravity.
The change in slope introduces a force component that accelerates the ball away from its initial position.

When the plate tilt direction is reversed, the ball responds accordingly and moves back toward the opposite side.
The noticeable dip in the curve around 3.5 – 4 s corresponds to the moment when the plate angle changes rapidly, producing a stronger acceleration.

Small oscillations in the ball position are observed after the direction change.
These oscillations are caused by inertia, contact dynamics, and friction between the ball and the plate, and they reflect realistic physical behavior of the system.

This response confirms the strong coupling between plate orientation and ball motion and highlights the unstable nature of the ball–plate system in open-loop control.

7. Control Design and Implementation

7.1 Control Objectives

Ball balancing on a tilting plate is a classic robotics control problem because the ball–plate system is inherently unstable in open loop: any small tilt generates a gravity component along the surface, which accelerates the ball away from its current position. In addition, the dynamics are nonlinear and strongly coupled, since the ball motion depends directly on the plate orientation, while contact forces and friction introduce disturbances and small oscillations. For this reason, a feedback controller is required to continuously correct the plate angles based on the measured ball position.

The objective of control in this project is to regulate the ball position around the plate center (setpoint at the origin of the plate frame), while minimizing overshoot, steady-state error, and unwanted vibrations. Practically, this means the controller must respond quickly when the ball deviates from the center, apply smooth corrective tilts through the arm joints, and remain robust to modeling imperfections (e.g., friction variations and contact effects). A successful controller stabilizes the ball near the center and maintains stable behavior over time despite the system's natural tendency to drift and oscillate.

7.2 Control Strategy: PID Regulation for Ball Balancing

To stabilize the ball at the center of the plate, we adopt a PID (Proportional–Integral–Derivative) controller as a feedback strategy. The key idea is simple: the controller continuously measures how far the ball is from the desired location (the center) and then adjusts the plate tilt angle so gravity generates a restoring acceleration that brings the ball back.

Error Definition (Tracking Objective)

Let the desired ball position on the plate be x_d (and similarly y_d because we control both axes). The measured ball position is $x(t)$. The tracking error is defined as:

$$e_x(t) = x_d - x(t)$$

$$e_y(t) = y_d - y(t)$$

In this project the setpoint is the plate center, so typically $x_d = 0$ and $y_d = 0$.

PID Control Law

The PID controller computes the commanded plate tilt (output) as a weighted combination of:

- the current error (P term),
- the accumulated error over time (I term),
- and the rate of change of error (D term).

A continuous-time PID law for the plate tilt around the x-axis can be written as:

$$\theta_x(t) = K_p e_x(t) + K_i \int_0^t e_x(\tau)d\tau + K_d \frac{de_x(t)}{dt}$$

where:

- $\theta_x(t)$ is the commanded tilt angle (or an equivalent command that produces this tilt through the actuators)
- K_p , K_i , K_d are the proportional, integral, and derivative gains.

The system is controlled in two directions, so we typically use two independent PID loops. Hence, we get the $\theta_y(t)$ as follows:

$$\theta_y(t) = K_p^y e_y(t) + K_i^y \int_0^t e_y(\tau)d\tau + K_d^y \frac{de_y(t)}{dt}$$

This matches the physical structure of the model, where plate orientation is produced by two hinge joints (one mainly affecting tilt in one direction and the other affecting the orthogonal direction).

Discrete-Time Implementation (Practical Form)

Since simulation runs with a fixed timestep, the PID is implemented in discrete time:

$$e[k] = x_d - x[k]$$

$$I[k] = I[k-1] + e[k]\,\Delta t$$

$$D[k] = \frac{e[k] - e[k-1]}{\Delta t}$$

$$\theta\,[k] = K_p\, e[k] + K_i\, I[k] + K_d\, D[k]$$

In practice, the integral term is often limited (anti-windup) to avoid excessive buildup when the actuator saturates:

$$I\,[k]\ \leftarrow clip\ (I[k], -I_{max}, -I_{max})$$

Why PID is Suitable for This System

PID control is a good fit for ball balancing on a plate for a few practical reasons:

- **Direct physical meaning:**

  When the ball drifts away from the center, tilting the plate creates a gravitational component along the surface that naturally accelerates the ball back. PID translates the position error into exactly this corrective tilt.

- **Handles instability in open-loop:**

  The ball–plate system tends to drift because small tilts produce motion that can grow over time. A feedback loop with PID provides continuous correction and prevents the ball from "running away."

- **Good performance without a complex model:**

  The dynamics are nonlinear and include friction and contact effects. PID does not require an exact mathematical model; it relies mainly on measured error, which makes it robust and easier to tune in simulation.

- **Clear role for each term:**

  K_p: provides the main restoring action (stronger correction when the ball is far from the center).

  K_i: removes steady-state bias (for example, if friction or slight plate asymmetry causes a persistent offset).

  K_d: improves damping and reduces overshoot by reacting to fast changes, which helps reduce oscillations.

Overall, using PID allows us to achieve the project goal: keep the ball near the center while reducing oscillations and improving stability, using a controller that is simple, interpretable, and effective for real-time implementation in MuJoCo.

7.3 Control Implementation

PID Code (Part 1/3)

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

# ---------- XML path ----------
XML_MODEL = r"""
<mujoco model="ball_balancing_arm">
    <compiler angle="degree" coordinate="local"/>
    <option timestep="0.002" gravity="0 0 -9.81"/>

    <default>
        <joint damping="1"/>
        <geom friction="0.6 0.2 0.1" density="1000"/>
    </default>

    <worldbody>

        <!-- Base -->
        <body name="base" pos="0 0 0">
            <geom type="box" size="0.1 0.1 0.05" rgba="0.3 0.3 0.3 1"/>

            <!-- Link 1 -->
            <body name="link1" pos="0 0 0.05">
                <joint name="joint_y"
                       type="hinge"
                       axis="0 1 0"
                       limited="true"
                       range="-25 25"/>

                <geom type="box"
                      size="0.02 0.02 0.15"
                      pos="0 0 0.15"
                      rgba="0.6 0.6 0.6 1"/>

                <!-- Link 2 -->
                <body name="link2" pos="0 0 0.3">
                    <joint name="joint_x"
                           type="hinge"
                           axis="1 0 0"
                           limited="true"
                           range="-45 45"/>

                    <geom type="box"
                          size="0.02 0.02 0.1"
                          pos="0 0 0.1"
                          rgba="0.7 0.7 0.7 1"/>

                    <!-- Plate with walls -->
                    <body name="plate" pos="0 0 0.2">
                        <!-- Plate surface -->
                        <geom type="box"
                              size="0.15 0.15 0.01"
                              rgba="0.2 0.6 0.8 1"/>

                        <!-- Walls -->
                        <geom type="box" size="0.15 0.005 0.03" pos="0 0.145 0.03"/>
                        <geom type="box" size="0.15 0.005 0.03" pos="0 -0.145 0.03"/>
                        <geom type="box" size="0.005 0.15 0.03" pos="0.145 0 0.03"/>
                        <geom type="box" size="0.005 0.15 0.03" pos="-0.145 0 0.03"/>
                    </body>

                </body>
            </body>
        </body>

        <!-- Ball -->
        <body name="ball" pos="0 0 0.7">
            <joint type="free"/>
            <geom type="sphere"
                  size="0.02"
                  rgba="0.9 0.1 0.1 1"/>
        </body>

    </worldbody>

    <!-- Actuators -->
    <actuator>
        <motor joint="joint_y" ctrlrange="-3 3" gear="150"/>
        <motor joint="joint_x" ctrlrange="-3 3" gear="150"/>
    </actuator>

</mujoco>
"""

# ============================================================
# 2) Build model + IDs (robust mapping to match XML)
# ============================================================
model = mujoco.MjModel.from_xml_string(XML_MODEL)
data = mujoco.MjData(model)

dt = model.opt.timestep
SIM_TIME = 10.0
steps = int(SIM_TIME / dt)

# ---------- IDs ----------
```

PID Code (Part 2/3)

```
ball_id  = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "ball")
plate_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_BODY, "plate")

joint_x_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_JOINT, "joint_x")
joint_y_id = mujoco.mj_name2id(model, mujoco.mjtObj.mjOBJ_JOINT, "joint_y")

joint_x_adr = model.jnt_qposadr[joint_x_id]
joint_y_adr = model.jnt_qposadr[joint_y_id]

# ---------- Actuator indices (بدون افتراض ترتيب ctrl) ----------
act_x = None
act_y = None
for a in range(model.nu):
    if model.actuator_trnid[a, 0] == joint_x_id:
        act_x = a
    if model.actuator_trnid[a, 0] == joint_y_id:
        act_y = a

if act_x is None and act_y is None:
    raise RuntimeError("No actuators found for joint_x / joint_y. Check <actuator> in XML.")

# ---------- Helper: position of ball along PLATE X axis ----------
def plate_x_coord():
    # ball and plate positions in world
    p_ball  = data.xpos[ball_id].copy()
    p_plate = data.xpos[plate_id].copy()

    # plate rotation matrix in world: reshape 9 -> 3x3
    R = data.xmat[plate_id].reshape(3, 3)  # world_R_plate

# plate X axis in world is first column of R
    x_axis_world = R[:, 0]

    # coordinate of relative vector projected on plate X axis
    rel = p_ball - p_plate
    x_on_plate = float(np.dot(rel, x_axis_world))
    return x_on_plate

# ---------- Simple metrics ----------
def compute_metrics(t, err, band=0.01):
    t = np.asarray(t, float)
    err = np.asarray(err, float)
    peak = float(np.max(np.abs(err)))
    rms  = float(np.sqrt(np.mean(err**2)))
    tail = max(1, int(0.10 * len(err)))
    ss   = float(np.mean(np.abs(err[-tail:])))
    # settling time: first time stays within band until end
    inside = np.abs(err) <= band
    st = None
    for k in range(len(err)):
        if inside[k] and np.all(inside[k:]):
            st = float(t[k])
            break
    return peak, rms, ss, st

# ---------- Reset + initial disturbance ----------
mujoco.mj_resetData(model, data)

# find ball free joint qpos address (7 values)
ball_free_adr = None
for j in range(model.njnt):
    if (model.jnt_bodyid[j] == ball_id) and (model.jnt_type[j] == mujoco.mjtJoint.mjJNT_FREE):
        ball_free_adr = model.jnt_qposadr[j]
        break
if ball_free_adr is None:
    raise RuntimeError("Ball must have a free joint (<joint type='free'/>).")

# put ball slightly off center (so control has something to do)
data.qpos[ball_free_adr + 0] += 0.06  # 6 cm
mujoco.mj_forward(model, data)

# ---------- Tiny calibration: choose which actuator controls plate-X, and sign ----------
# Idea: apply a small positive command and see how plate-X coordinate changes tendency.
def test_actuator(act_idx, u_test=0.3, test_time=0.2):
    # save state
    qpos0 = data.qpos.copy()
    qvel0 = data.qvel.copy()

    x0 = plate_x_coord()

    data.ctrl[:] = 0.0
    data.ctrl[act_idx] = u_test
    n = int(test_time / dt)
    for _ in range(n):
        mujoco.mj_step(model, data)
    mujoco.mj_forward(model, data)

    x1 = plate_x_coord()
    dx = x1 - x0

    # restore state
    data.qpos[:] = qpos0
    data.qvel[:] = qvel0
    mujoco.mj_forward(model, data)

    return dx
```

PID Code (Part 3/3)

```
candidates = []
if act_x is not None:
    candidates.append(act_x)
if act_y is not None:
    candidates.append(act_y)

# pick actuator with stronger effect on plate-X coordinate
effects = [(a, abs(test_actuator(a))) for a in candidates]
act_use = max(effects, key=lambda z: z[1])[0]

# sign: if +u makes x increase, then to reduce positive x we need negative u
dx_sign = test_actuator(act_use)
SIGN = -1.0 if dx_sign > 0 else 1.0

print(f"Using actuator index: {act_use}  |  SIGN={SIGN:+.1f}")

# ---------- PID gains
Kp = 1.0
Ki = 0.3
Kd = 2.0
I_LIM = 0.1

# ---------- PID states ----------
x_d = 0.0
e_prev = 0.0
I = 0.0

# ---------- Logs for plots ----------
t_log, x_log, jx_log, jy_log, u_log = [], [], [], [], []

# ---------- Run simulation ----------
with mujoco.viewer.launch_passive(model, data) as viewer:
    for i in range(steps):
        t = i * dt

        # 1) measure ball coordinate along plate X axis
        x = plate_x_coord()

        # 2) error
        e = x_d - x

        # 3) PID
        I += e * dt
        I = float(np.clip(I, -I_LIM, I_LIM))
        D = (e - e_prev) / dt
        e_prev = e

        u = SIGN * (Kp * e + Ki * I + Kd * D)
        u = float(np.clip(u, -1.0, 1.0))

        # 4) apply ONLY one actuator (X-axis control), set others to zero
        data.ctrl[:] = 0.0
        data.ctrl[act_use] = u

        mujoco.mj_step(model, data)
        viewer.sync()
        time.sleep(dt)

        # logs
        t_log.append(t)
        x_log.append(x)
        jx_log.append(float(data.qpos[joint_x_adr]))
        jy_log.append(float(data.qpos[joint_y_adr]))
        u_log.append(u)

print("Done.")

# ---------- Metrics + Plots ----------
err = np.array(x_log) - x_d
band = 0.025
peak, rms, ss, st = compute_metrics(t_log, err, band=band)

print("\nMetrics (X regulation):")
print(f"Peak deviation [m]     : {peak:.6f}")
print(f"RMS error [m]          : {rms:.6f}")
print(f"Steady-state error [m] : {ss:.6f}")
print(f"Settling time [s]      : {st if st is not None else 'Not settled'}")

plt.figure()
plt.plot(t_log, x_log, label="Ball coord along plate X [m]")
plt.axhline(+band, linestyle="--", label="band")
plt.axhline(-band, linestyle="--")
plt.xlabel("Time [s]")
plt.ylabel("Position [m]")
plt.title("Ball Position (Plate X axis)")
plt.grid(True)
plt.legend()

plt.figure()
plt.plot(t_log, jy_log, label="joint_y [rad]")
plt.xlabel("Time [s]")
plt.ylabel("Angle [rad]")
plt.title("Joint Angle")
plt.grid(True)
plt.legend()

plt.show()
```

The controller implementation was developed in Python using the MuJoCo API and is organized as a clear sequence of steps from model construction to performance evaluation. First, the MuJoCo model is created directly from the XML string, and the main simulation parameters are defined (time step, total simulation time, and number of steps). The script then retrieves the required identifiers for the ball, the plate, the two joints, and the available actuators. To avoid any dependence on actuator ordering, the code searches for the actuator indices programmatically based on their joint association.

Next, the ball position is measured in a physically meaningful way relative to the plate. Instead of using global coordinates, the script computes the ball displacement along the plate local X-axis by projecting the relative position vector (ball position minus plate position) onto the plate's X-direction extracted from the plate rotation matrix. This provides a consistent measurement of the controlled variable even when the plate rotates. The regulation objective is defined as keeping this coordinate at the setpoint $x_d = 0$, which corresponds to the center of the plate along the X direction.

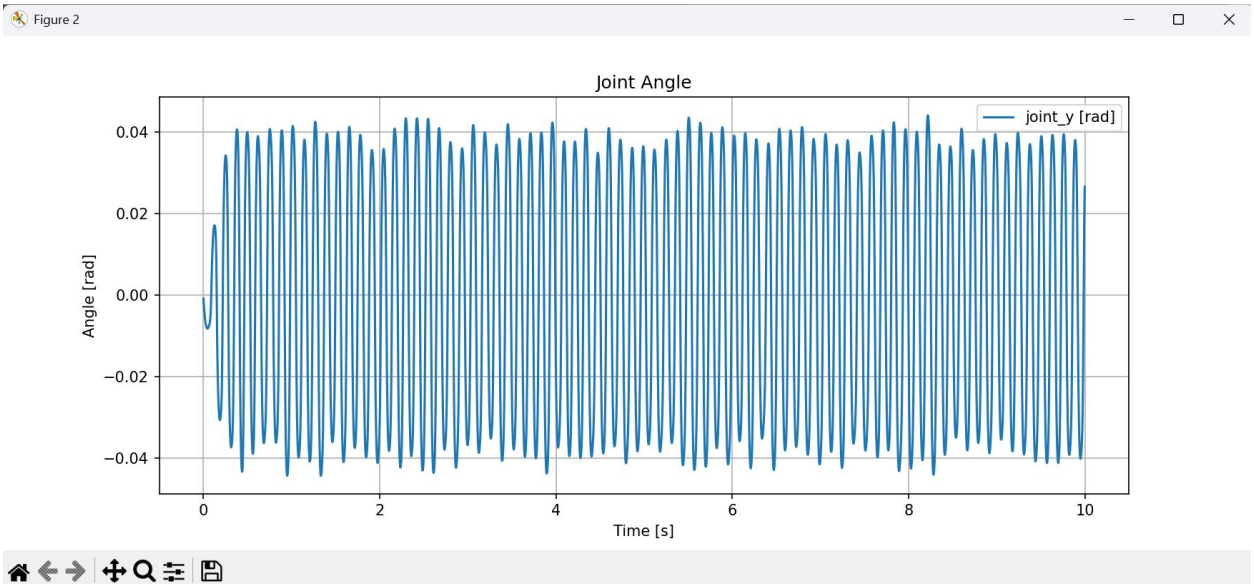
Before starting the control loop, the simulation is reset and a small initial disturbance is applied by offsetting the ball slightly from the center. This ensures that the controller response can be observed. In addition, a short calibration routine is executed to select the actuator that most strongly influences the plate X-direction and to determine the correct control sign. This is done by applying a small test command, observing the resulting change in the measured plate-axis coordinate, and then choosing the appropriate sign so that the feedback action drives the error toward zero.

The main control loop implements a discrete-time PID regulator. At each timestep, the ball's plate-axis coordinate is measured, the tracking error is computed as $e(t) = x_d - x(t)$, and the PID terms are updated: the integral term accumulates error over time with anti-windup clamping, while the derivative term estimates the error rate of change. The resulting control signal is saturated to remain within a safe normalized range and is applied to the selected actuator through data.ctrl, while all other actuators are set to zero to maintain single-axis control.

Finally, the script logs the key signals (time, ball position along the plate X-axis, joint angles, and control input) and produces standard evaluation metrics and plots. Performance is quantified using the peak deviation, RMS error, steady-state error (computed over the last portion of the simulation), and settling time based on a specified tolerance band. The results are visualized using time histories of the ball displacement and the relevant joint behavior, which together provide a clear validation of stability and regulation quality for the chosen PID strategy.

8. Results and Performance Evaluation

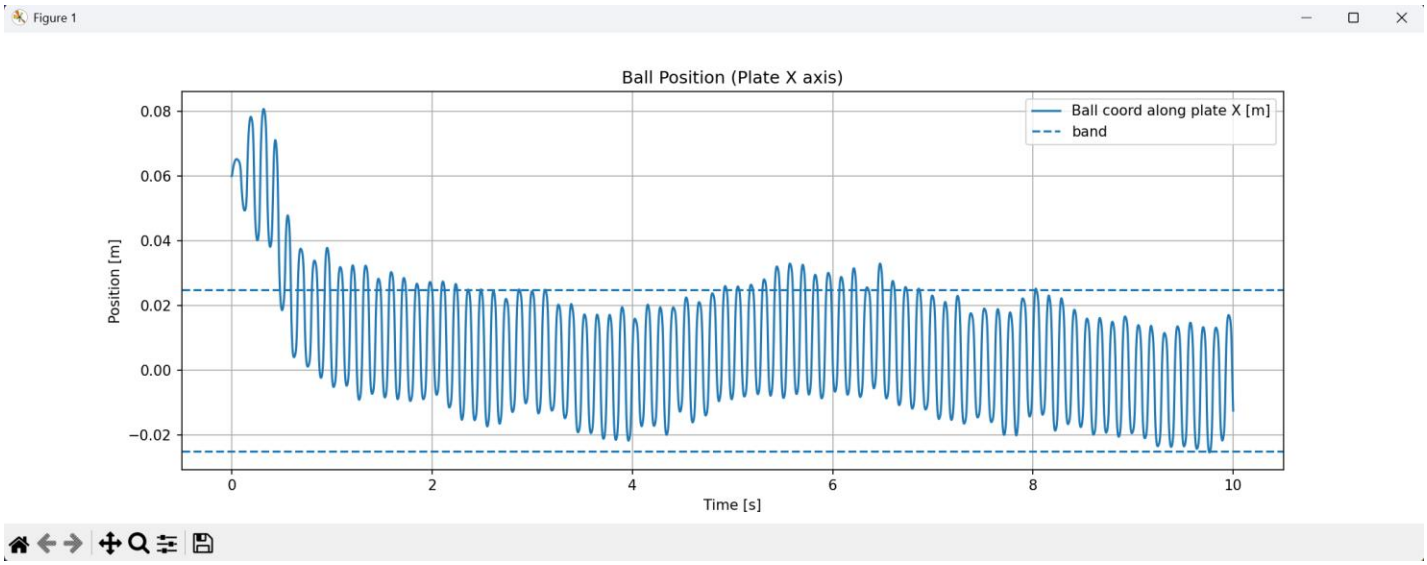
In the **open-loop case**, the plate tilt is commanded without feedback. As a result, the ball motion mainly reflects the applied tilt: when the plate angle changes, gravity creates a component along the surface and the ball accelerates in that direction. Practically, this means the ball does not regulate to the center; it drifts and keeps moving depending on the excitation, and any oscillations are not actively damped. The joint angle plot in open-loop therefore represents the input motion, not a corrective action.



The joint angle vs. Time using PID Controller

With PID feedback, the system continuously measures the ball displacement along the plate X-axis and generates corrective action through the actuator/joint. The main improvement is that the ball position becomes bounded around the center instead of drifting away. In your plot, the trajectory starts with a relatively large initial offset, but then it is pulled back toward zero and remains in a small oscillatory region around the center.

The joint-angle plot also changes in meaning: instead of following a pre-defined input, the joint motion becomes a regulation effort. The joint oscillations stay relatively small (around a few hundredths of a radian), which indicates the controller is stabilizing the ball using moderate plate tilts, not extreme motions.



The ball position vs. Time using PID Controller

The Ball Position plot shows the ball displacement along the plate's local X direction versus time. The response starts with a positive value because the ball is initialized with an offset from the center. After enabling the PID controller, the position moves toward zero, indicating that the controller generates corrective plate tilts to bring the ball back to the center. The signal then oscillates around zero, which is expected due to rolling/contact dynamics and limited damping, but it remains bounded rather than drifting as in open loop. The dashed lines represent the ±0.025 m tolerance band used to compute settling time, which is reached when the position stays inside this band until the end of the run.

Evaluation metrics used

To quantify performance, we relied on standard time-domain metrics commonly used in control:

- **Peak deviation (overshoot):** Because this is a regulation problem around zero (center), the most meaningful "overshoot" is the maximum absolute deviation from the setpoint

$$Peak\ Deviation = \max|x(t) - x_d|$$

$$RMS = \sqrt{\frac{1}{T}\int_0^T e(t)^2\,dt}$$

- **Settling time:** the first time after which the response stays within a tolerance band (here the band was $\pm\ 0.025\ m$) until the end.

These metrics are useful because they jointly capture: maximum excursion, overall smoothness, final accuracy, and how fast the system becomes reliably “close enough” to the center.

```
C:\Users\user\OneDrive\Desktop\MuJoCo>python test.py
Using actuator index: 0  |  SIGN=+1.0
Done.

Metrics (X regulation):
Peak deviation [m]     : 0.080732
RMS error [m]          : 0.021522
Steady-state error [m] : 0.013868
Settling time [s]      : 9.768
```

The peak deviation is mainly influenced by the initial disturbance and the early transient. The RMS and steady-state values show that, after the transient, the controller maintains the ball close to the center, with typical residual deviations on the order of centimeters. The settling time is relatively late because your trajectory stays oscillatory and only satisfies the “stay inside the band until the end” condition near the end of the run. Still, compared with open-loop (which does not truly “settle” around the center), this is a clear improvement in regulation.

Overall, the closed-loop PID controller provides a major qualitative improvement over the open-loop response. In open-loop, the ball motion is dominated by the plate excitation and naturally tends to drift rather than stabilize. With PID feedback, the ball position becomes bounded and centered, and the joint motion represents purposeful corrective action rather than a pre-defined input. Even though some oscillations remain (which is expected due to contact dynamics, friction, and the nonlinear nature of rolling motion), the controller significantly improves stability, repeatability, and center regulation compared to the open-loop behavior.

Metric	Open-loop	Closed-loop PID
Control mode	Open-loop	Closed-loop PID
Goal (center regulation)	No (no feedback)	Yes (x_d = 0)
Ball behavior	Drifts with applied tilt; not regulated	Bounded near center with damped oscillations
Peak deviation |x| [m]	N/A	0.080732
RMS error [m]	N/A	0.021522
Steady-state error [m]	N/A	0.013868
Settling time (±0.025 m) [s]	Not settled	9.768
Joint behavior	Follows commanded motion	Corrective small-angle actuation
Overall stability	Naturally unstable / not self-correcting	Stable regulation around center

9. Conclusion and Future Work

In conclusion, this project demonstrated a complete workflow for stabilizing a ball on a tilting plate using a MuJoCo-based robotic arm model. Starting from the XML system design and open-loop testing, we implemented a closed-loop PID controller that measures the ball displacement relative to the plate and generates corrective joint actuation to keep the ball near the center. Through simulation, plotting, and performance metrics, the work highlighted the practical difference between open-loop behavior (drift and weak damping) and feedback control (bounded motion and improved stability).

For future work, several extensions could significantly enhance performance and realism. A natural next step is to replace or complement PID with an optimal state-feedback method such as LQR, using a linearized model around the equilibrium to achieve faster settling with better damping and less oscillation. Another improvement is to introduce vision-based control, where the ball position is estimated from a simulated camera (or real camera in a hardware version), allowing the controller to operate without direct access to simulator states and making the pipeline closer to real-world deployment. Additional directions include adding integral anti-windup refinement, filtering the derivative term to reduce noise sensitivity, and exploring more advanced approaches such as MPC or reinforcement learning for robust control under larger disturbances and model uncertainty.