minilab2_simulation.py

```python
"""
IMPORTANT NOTE:
    The instructions for completing this template are inline with the code. You can
    find them by searching for: "TODO:"
"""

import math
from datetime import datetime

from pathlib import Path

import matplotlib.pyplot as plt
import tqdm
import numpy as np
from numpy.typing import NDArray

from pydrake.geometry import Meshcat, MeshcatVisualizer
from pydrake.multibody.parsing import Parser
from pydrake.multibody.plant import AddMultibodyPlantSceneGraph, MultibodyPlant
from pydrake.systems.analysis import Simulator
from pydrake.systems.framework import (
    DiagramBuilder, LeafSystem, Context, BasicVector, DiscreteValues
)
from pydrake.systems.primitives import LogVectorOutput


class PDwGravityCompensationController(LeafSystem):
    """
    This class manages a PD with gravity compensation controller
    """

    def __init__(
        self,
        q_desired_deg: list[float],
        K_P: NDArray[np.double],
        K_D: NDArray[np.double],
        abs_position_error_tol_deg: float = 1e-3
    ):
        super().__init__()

        self.m1, self.m2 = 0.193537, 0.0156075
        self.lc1, self.lc2 = 0.0533903, 0.0281188
        self.l1 = 0.0675

        # ---------------------------------------------------------------------------------
        # Controller Related Variables
        # ---------------------------------------------------------------------------------
        self.q_desired_rad = np.deg2rad(q_desired_deg)
```

```python
        self.abs_position_error_tol_deg = abs(float(abs_position_error_tol_deg))

        # Set PID gains
        self.K_P = np.asarray(K_P, dtype=np.double)
        self.K_D = np.asarray(K_D, dtype=np.double)

        self.dt = 1e-3

        self.input_port = self.DeclareVectorInputPort("state", BasicVector(4))
        self.state_ix = self.DeclareDiscreteState(2)
        self.DeclarePeriodicDiscreteUpdateEvent(
            self.dt,
            0.0,
            self.update_output_torque
        )
        self.DeclareVectorOutputPort(
            "motor_torque", BasicVector(2), self.extract_output_torque
        )

        num_steps = math.ceil(simulation_duration_s / plant.time_step())
        self.pbar = tqdm.tqdm(
            range(num_steps),
            total=num_steps,
            leave=True,
            desc="Simulation Progress",
            ncols=100
        )

    def __del__(self):
        self.pbar.close()

    def extract_output_torque(self, context: Context, output: BasicVector):
        torque = context.get_discrete_state_vector()
        output.SetFromVector(torque.get_value())

    def update_output_torque(self, context: Context, discrete_values: DiscreteValues):
        # ------------------------------------------------------------------------------
        # Step 1 - Get position feedback
        # ------------------------------------------------------------------------------
        state = self.get_input_port().Eval(context)

        q_rad = np.asarray([state[0], state[1]])
        qdot_rad_per_s = np.asarray([state[2], state[3]])

        # ------------------------------------------------------------------------------
        # TODO: Step 2 - Compute error term (Question 2)
        # ------------------------------------------------------------------------------
        # Use the `self.q_desired_rad` variable and the `q_rad` variable to compute
        # the joint position error for the current time step.
        # ------------------------------------------------------------------------------
```

```python
        q_error = self.q_desired_rad - q_rad

        # ----------------------------------------------------------------------------
        # Step 3 - Calculate gravity compensation term
        # ----------------------------------------------------------------------------
        gravity_comp_torques = self.calc_gravity_compensation_torque(q_rad)

        # ----------------------------------------------------------------------------
        # TODO: Step 4 - Calculate and send control action (Question 2)
        # ----------------------------------------------------------------------------
        # Use the `self.K_P`, `q_error`, `self.K_D`, `qdot_rad_per_s`, and
        # `gravity_comp_torques` variables to compute the control action for joint
        # space PD control with gravity compensation.
        #
        # Tip: A NumPy array `A` of shape (2, 2) and a NumPy array `b` of shape (2,)
        #      can be matrix-vector multiplied via the Python syntax `A @ b`.
        # ----------------------------------------------------------------------------
        u = gravity_comp_torques + self.K_P @ q_error - self.K_D @ qdot_rad_per_s

        # Saturate joint torque output to motor limits
        u = np.minimum(np.maximum(u, -2.5), 2.5)

        # "Send" control action
        discrete_values.get_mutable_vector().SetFromVector(u)
        # ----------------------------------------------------------------------------

        # Update progress bar
        self.pbar.update(1)
        self.pbar.set_postfix_str(
            f"q_deg: [{math.degrees(q_rad[0]):.4f}, {math.degrees(q_rad[1]):.4f}]"
        )

    def calc_gravity_compensation_torque(
        self, joint_positions_rad: NDArray[np.double]
    ) -> NDArray[np.double]:
        q1, q2 = joint_positions_rad

        from math import sin, cos
        g = 9.81

        m1, m2 = self.m1, self.m2
        l1 = self.l1
        lc1, lc2 = self.lc1, self.lc2

        return -np.array(
            [
                m1 * g * lc1 * cos(q1) + m2 * g * (l1 * cos(q1) + lc2 * cos(q1 + q2)),
                m2 * g * lc2 * cos(q1 + q2)
            ]
        )
```

```python


if __name__ == "__main__":
    np.set_printoptions(suppress=True, precision=5, floatmode="fixed")
    simulation_duration_s = 2.0

    # Create a `DiagramBuilder`` to which systems and connections will be added for the
    # simulation
    builder = DiagramBuilder()

    # Create `MultibodyPlant` and `SceneGraph`
    # `MultibodyPlant` provides an API for kinematics and dynamics of multiple bodies
    # `SceneGraph` provides an API to visualize the results of a physics engine
    plant, scene_graph = AddMultibodyPlantSceneGraph(builder, 1e-3)

    # Add models using a `Parser` object (parses ".sdf" and ".urdf" files)
    parser = Parser(plant)
    model_instance_ix = parser.AddModels(
        file_name=str((Path(__file__).parent / "urdf" / "robot.urdf"))
    )[0]
    plant.set_gravity_enabled(model_instance_ix, True)

    # Fix the base frame of the "robot" in the world frame
    plant.WeldFrames(
        plant.world_frame(),
        plant.GetFrameByName("link0")
    )

    # Create a `MeshcatVisualizer` to view our scene graph using Meshcat and add it
    # to our diagram builder
    meshcat = Meshcat(port=8888)
    visualizer: MeshcatVisualizer = MeshcatVisualizer.AddToBuilder(
        builder, scene_graph, meshcat
    )

    # Finalize `MultibodyPlant` to tell Drake we are finished adding models
    # (You can't add anymore models after calling `MultibodyPlant::Finalize()`).
    plant.Finalize()

    # ================================================================================
    # TODO: Set Initial Conditions and Setpoint / Tune Controller Gains (Question 3)
    # --------------------------------------------------------------------------------
    # 1) Replace the corresponding `...` values with the initial and desired joint
    #    configurations.
    # 2) Replace the corresponding `...` values with your K_P and K_D gain matrices
    # --------------------------------------------------------------------------------
    # A Python list with two elements representing the initial joint configuration
    q_initial = [65.0, 25.0]

    # A Python list with two elements representing the desired joint configuration
```

```python
    q_desired = [45.0, 45.0]

    # A numpy array of shape (2, 2) representing the proportional gains of your
    # controller
    K_P = np.array([[10, 0],
                   [0, 10]])

    # A numpy array of shape (2, 2) representing the derivative gains of your controller
    K_D = np.array([[0.155, 0],
                   [0, 0.0206]])

    # Add controller to diagram builder
    controller: PDwGravityCompensationController = builder.AddNamedSystem(
        "pd_w_gravity_compensation_controller",
        PDwGravityCompensationController(
            q_desired_deg=q_desired,
            K_P=K_P,
            K_D=K_D
        )
    )
    # ==================================================================================

    joint_state_logger = LogVectorOutput(
        plant.get_state_output_port(), builder, publish_period=1e-3
    )

    # Connect systems in diagram builder
    builder.Connect(plant.get_state_output_port(), controller.get_input_port())
    builder.Connect(controller.get_output_port(), plant.get_actuation_input_port())

    # Build the diagram
    diagram = builder.Build()

    # Get root context of diagram (everything needed for the simulation to run)
    root_context = diagram.CreateDefaultContext()

    # Set initial joint position and velocity of motors
    plant_context = plant.GetMyMutableContextFromRoot(root_context)

    # Set initial motor state
    plant.SetPositions(plant_context, model_instance_ix, np.deg2rad(q_initial))
    plant.SetVelocities(plant_context, model_instance_ix, [0.0, 0.0])

    # Create simulator from diagram and root context
    simulator = Simulator(diagram, root_context)

    # Set realtime target rate to 1x speed
    simulator.set_target_realtime_rate(1.0)

    # Set camera view
```

```
    meshcat.SetCameraPose(
        [0.0, 0.15, 0.17],
        [0.0, 0.0, 0.0]
    )

    # Start listening for events in Meshcat
    visualizer.StartRecording()

    # Run simulation for preconfigured duration
    simulator.AdvanceTo(2.0)

    # Publish events in Meshcat
    visualizer.StopRecording()
    visualizer.PublishRecording()

    # ----------------------------------------------------------------------------------
    # Plot Results
    # ----------------------------------------------------------------------------------
    # Extract Data
    joint_state_log = joint_state_logger.FindLog(simulator.get_context())
    timestamps = joint_state_log.sample_times()
    position_history = np.rad2deg(joint_state_log.data()[:2, :])

    # date_str = datetime.now().strftime("%d-%m_%H-%M-%S")
    # fig_file_name = f"joint_positions_vs_time_{date_str}.png"
    fig_file_name = "joint_positions_vs_time.png"

    # Create figure and axes
    fig = plt.figure(figsize=(10, 5))
    ax_motor0 = fig.add_subplot(121)
    ax_motor1 = fig.add_subplot(122)

    # Label Plots
    fig.suptitle(f"Motor Angles vs Time")
    ax_motor0.set_title("Motor Joint 0")
    ax_motor1.set_title("Motor Joint 1")
    ax_motor0.set_xlabel("Time [s]")
    ax_motor1.set_xlabel("Time [s]")
    ax_motor0.set_ylabel("Motor Angle [deg]")
    ax_motor1.set_ylabel("Motor Angle [deg]")

    ax_motor0.axhline(
        math.degrees(controller.q_desired_rad[0]),
        ls="--",
        color="red",
        label="Setpoint"
    )
    ax_motor1.axhline(
        math.degrees(controller.q_desired_rad[1]),
        ls="--",
```

```
            color="red",
            label="Setpoint"
        )
        ax_motor0.axhline(
            math.degrees(controller.q_desired_rad[0]) - 1, ls=":", color="blue"
        )
        ax_motor0.axhline(
            math.degrees(controller.q_desired_rad[0]) + 1,
            ls=":",
            color="blue",
            label="Convergence Bound"
        )
        ax_motor0.axvline(1.5, ls=":", color="purple")
        ax_motor1.axhline(
            math.degrees(controller.q_desired_rad[1]) - 1,
            ls=":",
            color="blue",
            label="Convergence Bound"
        )
        ax_motor1.axhline(
            math.degrees(controller.q_desired_rad[1]) + 1, ls=":", color="blue"
        )
        ax_motor1.axvline(1.5, ls=":", color="purple")

        # Plot motor angle trajectories
        ax_motor0.plot(
            timestamps,
            position_history[0],
            color="black",
            label="Motor Angle Trajectory",
        )
        ax_motor1.plot(
            timestamps,
            position_history[1],
            color="black",
            label="Motor Angle Trajectory",
        )
        ax_motor0.legend()
        ax_motor1.legend()
        fig.savefig(fig_file_name)

        print()
        # plt.show()


```