mechae263C_homework5_problem1.py

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

from __future__ import annotations

import math

import matplotlib.pyplot as plt
import numpy as np
from numpy.typing import NDArray
from pydrake.systems.analysis import Simulator
from pydrake.systems.framework import (
    Context,
    Diagram,
    DiagramBuilder,
    InputPort,
)
from pydrake.systems.primitives import (
    MatrixGain,
    PassThrough,
    ZeroOrderHold,
    LogVectorOutput,
    ConstantVectorSource,
)

from mechae263C_helpers.drake import LinearCombination, plot_diagram
from mechae263C_helpers.hw5 import validate_np_array
from mechae263C_helpers.hw5.arm import Arm, Gravity
from mechae263C_helpers.hw5.jacobian_gains import (
    AnalyticalJacobianTransposeGain,
    AnalyticalJacobianGain,
)
from mechae263C_helpers.hw5.kinematics import calc_2R_planar_inverse_kinematics
from mechae263C_helpers.hw5.op_space import DirectKinematics


def calc_analytical_jacobian(
    q1: float, q2: float, a1: float, a2: float
) -> NDArray[np.double]:
    """
    Calculates the Analytical Jacobian of a 2R planar manipulator

    Parameters
    ----------
    q1
```

```
        A float representing the first joint angle
    q2
        A float representing the second joint angle
    a1
         A float representing the first link length
    a2
        A float representing the second link length

    Returns
    -------
    A numpy array of shape (2, 2) representing the Analytical Jacobian of the 2R
    planar manipulator
    """
    J_A = np.zeros(shape=(2, 2), dtype=np.double)

    # ================================================================================
    # TODO: Calculate Analytical Jacobian (J_A)
    #   Fill in the provided numpy array `J_A` with the Analytical Jacobian of the
    #   manipulator
    # --------------------------------------------------------------------------------
    J_A[0, 0] = (-a1 * np.sin(q1)) - (a2 * np.sin(q1 + q2))
    J_A[1, 0] = (a1 * np.cos(q1)) + (a2 * np.cos(q1 + q2))
    J_A[0, 1] = (-a2 * np.sin(q1 + q2))
    J_A[1, 1] = (a2 * np.cos(q1 + q2))
    # ================================================================================

    return J_A


def calc_direct_kinematics(
    q1: float, q2: float, a1: float, a2: float
) -> NDArray[np.double]:
    """
    Calculates the direct (a.k.a. forward) kinematics of a 2R planar manipulator

    Parameters
    ----------
    q1
        A float representing the first joint angle
    q2
        A float representing the second joint angle
    a1
         A float representing the first link length
    a2
        A float representing the second link length

    Returns
    -------
    A numpy array of shape (2,) representing the xy position of the 2R planar
    manipulator's end effector
```

```python
    """
    x_e = np.zeros(shape=(2,), dtype=np.double)

    # ==========================================================================
    # TODO: Calculate Direct Kinematics
    #   Fill in the provided numpy array `x_e` with the x and y positions of the
    #   end-effector using the direct kinematics of a 2R planar manipulator.
    # --------------------------------------------------------------------------
    x_e[0] = (a1 * np.cos(q1)) + (a2 * np.cos(q1 + q2))
    x_e[1] = (a1 * np.sin(q1)) + (a2 * np.sin(q1 + q2))
    # ==========================================================================

    return x_e


class OperationalSpacePDControllerWithGravityCompensation(Diagram):
    def __init__(
        self,
        link_lens: tuple[float, float],
        K_P: NDArray[np.double],
        K_D: NDArray[np.double],
        control_sample_period_s: float,
        p_desired: NDArray[np.double],
    ):
        super().__init__()
        self.control_sample_period_s = max(1e-10, abs(control_sample_period_s))
        self.link_lens = tuple(float(a) for a in link_lens)
        self.num_dofs = len(link_lens)
        assert self.num_dofs == 2

        validate_np_array(arr=K_P, arr_name="K_P", correct_shape=(2, 2))
        validate_np_array(arr=K_D, arr_name="K_D", correct_shape=(2, 2))
        validate_np_array(arr=p_desired, arr_name="p_desired", correct_shape=(2,))

        self.K_P = K_P
        self.K_D = K_D

        builder = DiagramBuilder()

        proportional_gain: MatrixGain = builder.AddNamedSystem("K_P", MatrixGain(K_P))
        derivative_gain: MatrixGain = builder.AddNamedSystem("K_D", MatrixGain(K_D))
        gravity_torques: Gravity = builder.AddNamedSystem(
            "gravity", Gravity(Arm().dyn_params)
        )
        JA_gain: AnalyticalJacobianGain = builder.AddNamedSystem(
            "J_A",
            AnalyticalJacobianGain(self.link_lens, calc_analytical_jacobian),
        )
        JA_T_gain: AnalyticalJacobianTransposeGain = builder.AddNamedSystem(
            "J_A.T",
```

```
            AnalyticalJacobianTransposeGain(self.link_lens, calc_analytical_jacobian),
        )
        direct_kinematics: DirectKinematics = builder.AddNamedSystem(
            "k(q)", DirectKinematics(self.link_lens, calc_direct_kinematics)
        )
        control_torques: LinearCombination = builder.AddNamedSystem(
            "u", LinearCombination(input_coeffs=(1, 1), input_shapes=(2,))
        )
        operational_space_position_error: LinearCombination = builder.AddNamedSystem(
            "x_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
        )
        operational_space_control_action: LinearCombination = builder.AddNamedSystem(
            "f_c", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
        )

        q = builder.AddNamedSystem("q", PassThrough(vector_size=self.num_dofs))
        qdot = builder.AddNamedSystem("qdot", PassThrough(vector_size=self.num_dofs))
        zoh = builder.AddNamedSystem(
            "sampled_u",
            ZeroOrderHold(
                period_sec=self.control_sample_period_s, vector_size=self.num_dofs
            ),
        )
        p_desired_source = builder.AddNamedSystem(
            "p_desired", ConstantVectorSource(source_value=p_desired)
        )

        # ================================================================================
        # TODO: Complete Controller Block Diagram
        #   Replace `...` below with the correct output or input port.
        #   Note that following convenience method is available to access the f_c input
        #   port of the `JA_T_gain` system/block
        #       JA_T_gain.get_f_c_input_port()
        # --------------------------------------------------------------------------------
        builder.Connect(
            operational_space_position_error.get_output_port(),
            proportional_gain.get_input_port(),
        )
        builder.Connect(JA_gain.get_output_port(),
                        derivative_gain.get_input_port())

        # from Kp
        builder.Connect(
            proportional_gain.get_output_port(),
            operational_space_control_action.get_input_port(0),
        )
        # from Kd
        builder.Connect(
            derivative_gain.get_output_port(),
            operational_space_control_action.get_input_port(1),
```

```
        )

        # Sum to Analytical Jacobian transpose block
        builder.Connect(
            operational_space_control_action.get_output_port(),
            JA_T_gain.get_f_c_input_port(),
        )

        # JAT to sum
        builder.Connect(JA_T_gain.get_output_port(),
                        control_torques.get_input_port(0)
        )
        # Grav to sum
        builder.Connect(gravity_torques.get_output_port(),
                        control_torques.get_input_port(1)
        )

        # Positon error input
        builder.Connect(
            p_desired_source.get_output_port(),
            operational_space_position_error.get_input_port(0)
        )
        builder.Connect(
            direct_kinematics.get_output_port(),
            operational_space_position_error.get_input_port(1)
        )
        # ==============================================================================

        builder.Connect(q.get_output_port(), gravity_torques.get_input_port())
        builder.Connect(q.get_output_port(), direct_kinematics.get_q_input_port())
        builder.Connect(q.get_output_port(), JA_gain.get_q_input_port())
        builder.Connect(q.get_output_port(), JA_T_gain.get_q_input_port())

        # This samples the controller at the specified period (to simulate discrete
        # control)
        builder.Connect(control_torques.get_output_port(), zoh.get_input_port())

        builder.Connect(qdot.get_output_port(), JA_gain.get_qdot_input_port())

        builder.ExportInput(q.get_input_port(), name="q")
        builder.ExportInput(qdot.get_input_port(), name="qdot")
        builder.ExportOutput(zoh.get_output_port(), name="u")

        # ------------------------------------------------------------------------------
        # Log operational space positions
        # ------------------------------------------------------------------------------
        # These systems are special in Drake. They periodically save the output port
        # value a during a simulation so that it can be accessed later. The value is
        # saved every
        # `publish_period` seconds in simulation time.
```

```
        self.operational_space_position_logger = LogVectorOutput(
            direct_kinematics.get_output_port(),
            builder,
            publish_period=control_sample_period_s,
        )
        self.operational_space_position_logger.set_name("Tip Position Logger")

        builder.BuildInto(self)
        self.set_name("Controller")

    def get_q_input_port(self) -> InputPort:
        return self.get_input_port(0)

    def get_qdot_input_port(self) -> InputPort:
        return self.get_input_port(1)


def run_simulation(
    simulation_duration_s: float,
    link_lens: tuple[float, ...],
    load_mass_kg: float,
    K_P: NDArray[np.double],
    K_D: NDArray[np.double],
    p_desired: NDArray[np.double],
    control_sample_period_s: float,
):
    """
    Runs a Drake simulation of operational space PD control with gravity compensation
    ----------
    simulation_duration_s
        A float representing the simulation duration in seconds

    link_lens
        A tuple of two float representing the length of the links (in order)

    load_mass_kg
        A float representing the load mass in kg

    K_P
        A numpy array of shape (2, 2) representing the proportional gains of the PD
        controller, expressed in the base frame

    K_D
        A numpy array of shape (2, 2) representing the derivative gains of the PD
        controller, expressed in the base frame

    p_desired
        A numpy array of shape (2,) representing the desired position of the
        end-effector

```

```python
    control_sample_period_s
        A float representing the duration of the trajectory in seconds

    Returns
    -------
    A tuple of four elements:
        1) The time-steps of the simulation in seconds
        2) The simulated end-effector positions in meter corresponding to each time-step
        4) The controller used during the simulation (this is also a `Diagram` object).
        4) The high level simulation `Diagram` object
    """
    validate_np_array(arr=p_desired, arr_name="p_desired", correct_shape=(2,))

    builder = DiagramBuilder()
    arm: Arm = builder.AddNamedSystem("arm", Arm(load_mass_kg=load_mass_kg))
    controller: OperationalSpacePDControllerWithGravityCompensation = (
        builder.AddNamedSystem(
            "controller",
            OperationalSpacePDControllerWithGravityCompensation(
                link_lens=link_lens,
                K_P=K_P,
                K_D=K_D,
                control_sample_period_s=control_sample_period_s,
                p_desired=p_desired,
            ),
        )
    )

    # ==================================================================================
    # TODO: Complete Simulation Block Diagram (Arm + Controller)
    #   Replace `...` below with the correct output or input port.
    #   Note that following convenience methods are available to access the input and
    #   output ports:
    #       arm:
    #       - arm.get_q_output_port()
    #       - arm.get_qdot_output_port()
    #       - arm.get_input_port()
    #
    #       controller:
    #       - controller.get_q_input_port()
    #       - controller.get_qdot_input_port()
    # ----------------------------------------------------------------------------------
    builder.Connect(controller.get_output_port(), arm.get_input_port())
    builder.Connect(arm.get_q_output_port(), controller.get_q_input_port())
    builder.Connect(arm.get_qdot_output_port(), controller.get_qdot_input_port())
    # ==================================================================================

    # Build a `Diagram` object and use it to make a `Simulator` object for the diagram
    diagram: Diagram = builder.Build()
    diagram.set_name("Operational Space PD Control w/ Gravity Compensation")
```

```python
    simulator: Simulator = Simulator(diagram)

    # Get the context (this contains all the information needed to run the simulation)
    context: Context = simulator.get_mutable_context()

    # Set initial conditions
    initial_conditions = context.get_mutable_continuous_state_vector()
    q_initial = calc_2R_planar_inverse_kinematics(
        link_lens, end_effector_position=p_desired - 0.1, use_elbow_up_soln=True
    )
    initial_conditions.SetAtIndex(2, q_initial[0])
    initial_conditions.SetAtIndex(3, q_initial[1])

    # Advance the simulation by `simulation_duration_s` seconds using the
    # `simulator.AdvanceTo()` function
    simulator.AdvanceTo(simulation_duration_s)

    # ------------------------------------------------------------------------------
    # Extract simulation outputs
    # ------------------------------------------------------------------------------
    # The lines below extract the joint position log from the simulator context
    operational_space_position_log = (
        controller.operational_space_position_logger.FindLog(simulator.get_context())
    )
    t = operational_space_position_log.sample_times()
    p_actual = operational_space_position_log.data()

    return t, p_actual, controller, diagram


if __name__ == "__main__":
    # ==============================================================================
    # TODO: Problem 1 - Part (b)
    #   Replace `...` with the appropriate value from the problem statement based on
    #   the comment describing each variable (on the line(s) above it).
    # ------------------------------------------------------------------------------
    # A tuple with two elements representing the first and second link lengths of the
    # manipulator, respectively.
    link_lens = 1, 1

    # A float representing the load mass in kg
    load_mass_kg = 10

    # A numpy array of shape (2,) representing the desired end-effector position for the
    # first case
    p_desired_case1 = np.array([0.6, -0.2])

    # A numpy array of shape (2,) representing the desired end-effector position for the
    # second case
    p_desired_case2 = np.array([0.5, 0.5])
```

```python

    # A float representing the time horizon of the entire simulation
    simulation_duration_s = 2.5

    # A float representing the sampling time of discrete-time controller
    control_sample_period_s = 1e-3

    # A numpy array of shape (2, 2) representing the PD controller proportional gains
    K_P = np.array([[100000, 0],
                    [0, 100000]])

    # A numpy array of shape (2, 2) representing the PD controller derivative gains
    K_D = np.array([[19500, 0],
                    [0, 19500]])

    # ------------------------------------------------------------------------------
    # TODO: Run Simulation
    #   Replace `...` in parameters for `run_simulation` function using the variables
    #   above
    # ------------------------------------------------------------------------------
    # Run case 1
    t_case1, p_actual_case1, controller_diagram, simulation_diagram = run_simulation(
        simulation_duration_s=simulation_duration_s,
        link_lens=link_lens,
        load_mass_kg=load_mass_kg,
        K_P=K_P,
        K_D=K_D,
        p_desired=p_desired_case1,
        control_sample_period_s=control_sample_period_s,
    )
    print("finish case1 sim")

    # Run case 2
    t_case2, p_actual_case2, controller_diagram_case2, simulation_diagram_case2 = run_simulation(
        simulation_duration_s=simulation_duration_s,
        link_lens=link_lens,
        load_mass_kg=load_mass_kg,
        K_P=K_P,
        K_D=K_D,
        p_desired=p_desired_case2,
        control_sample_period_s=control_sample_period_s,
    )
    print("finish case2 sim")

    # ------------------------------------------------------------------------------
    # TODO: Plot Controller Block Diagram
    #   Use the `plot_diagram` function to plot the diagram of the controller design
    #   (which is stored in the `controller_diagram` variable)
    # ------------------------------------------------------------------------------
```

```
    controller_diagram_fig, _ = plot_diagram(controller_diagram, fig_width_in=11)
    controller_diagram_fig.savefig('Problem1/diagram_case1.png', dpi=300)
    controller_diagram_fig2, _ = plot_diagram(controller_diagram_case2, fig_width_in=11)
    controller_diagram_fig2.savefig('Problem1/diagram_case2.png', dpi=300)
    print("plotted control diagram")

    # ----------------------------------------------------------------------------------
    # TODO: Plot Simulation Block Diagram
    #   Use the `plot_diagram` function to plot the high-level diagram of the simulation
    #   (which is stored in the `simulation_diagram` variable)
    # ----------------------------------------------------------------------------------
    simulation_diagram_fig, _ = plot_diagram(simulation_diagram, fig_width_in=8)
    simulation_diagram_fig.savefig('Problem1/simulationDiagram_case1.png', dpi=300)
    simulation_diagram_fig2, _ = plot_diagram(simulation_diagram_case2, fig_width_in=8)
    simulation_diagram_fig2.savefig('Problem1/simulationDiagram_case2.png', dpi=300)
    print("plotted simulation diagram")
    # ==================================================================================

    # ==================================================================================
    # TODO: Problem 2 - Part (c)
    #   Use the `print` function to output your gains
    # ----------------------------------------------------------------------------------
    print("K_P:")
    print(K_P)
    print("\nK_D:")
    print(K_D)

    # ==================================================================================
    # TODO: Problem 2 - Part (d)
    # ----------------------------------------------------------------------------------
    # TODO: Plot Case 1 Tip X and Y Positions
    #   For Case 1:
    #       1) Plot the time history of the x and y coordinates of the end effector
    #          position in separate sub-figures for a time horizon of 2.5 seconds. Use a
    #          solid red line for both the x and y positions.
    #       2) Indicate the desired coordinate value in each sub-figure by drawing a
    #          solid black dashed horizontal line at the desired value
    # ----------------------------------------------------------------------------------
    # Plot data in `p_actual_case1`
        # Create figure and axes
    fig = plt.figure(figsize=(10, 5))
    case1_x = fig.add_subplot(121)
    case1_y = fig.add_subplot(122)

    # Label Plots
    fig.suptitle("Case 1: EE Position")
    case1_x.set_title("X Position vs Time")
    case1_x.set_xlabel("Time [s]")
    case1_x.set_ylabel("X [m]")
    case1_y.set_title("Y Position vs Time")
```

```python
    case1_y.set_xlabel("Time [s]")
    case1_y.set_ylabel("Y [m]")

    case1_x.axhline(
        p_desired_case1[0], ls="--", color="red", label="Desired X"
    )
    case1_y.axhline(
        p_desired_case1[1], ls="--", color="red", label="Desired Y"
    )

    case1_x.plot(
        t_case1, p_actual_case1[0], color="black", label="EE X Position"
    )
    case1_y.plot(
        t_case1, p_actual_case1[1], color="black", label="EE Y Position"
    )
    case1_x.legend()
    case1_y.legend()
    fig.savefig('Problem1/Case1_Positions.png', dpi=300)
    print("plotted case 1 positions")
    plt.clf

    # ----------------------------------------------------------------------------------
    # TODO: Plot Case 2 Tip X and Y Positions
    #   For Case 2:
    #       1) Plot the time history of the x and y coordinates of the end effector
    #          position in separate sub-figures for a time horizon of 2.5 seconds. Use a
    #          solid red line for both the x and y positions.
    #       2) Indicate the desired coordinate value in each sub-figure by drawing a
    #          solid black dashed horizontal line at the desired value
    # ----------------------------------------------------------------------------------
    # Plot data in `p_actual_case2`
    fig2 = plt.figure(figsize=(10, 5))
    case2_x = fig2.add_subplot(121)
    case2_y = fig2.add_subplot(122)

    # Label Plots
    fig2.suptitle("Case 2: EE Position")
    case2_x.set_title("X Position vs Time")
    case2_x.set_xlabel("Time [s]")
    case2_x.set_ylabel("X [m]")
    case2_y.set_title("Y Position vs Time")
    case2_y.set_xlabel("Time [s]")
    case2_y.set_ylabel("Y [m]")

    case2_x.axhline(
        p_desired_case2[0], ls="--", color="red", label="Desired X"
    )
    case2_y.axhline(
        p_desired_case2[1], ls="--", color="red", label="Desired Y"
```

```
    )
    case2_x.plot(
        t_case2, p_actual_case2[0], color="black", label="EE X Position"
    )
    case2_y.plot(
        t_case2, p_actual_case2[1], color="black", label="EE Y Position"
    )
    case2_x.legend()
    case2_y.legend()
    fig2.savefig('Problem1/Case2_Positions.png', dpi=300)
    print("plotted case 2 positions")
    # ==================================================================================

    #plt.show()

```