mechae263C_homework4.py

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

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 DiagramBuilder, Diagram, Context
from pydrake.systems.primitives import MatrixGain, LogVectorOutput

from mechae263C_helpers.drake import LinearCombination, plot_diagram
from mechae263C_helpers.hw4.arm import Arm
from mechae263C_helpers.hw4.kinematics import calc_fk_2D
from mechae263C_helpers.hw4.trajectory import (
    eval_cubic_spline_traj,
    JointSpaceTrajectorySource,
)
from mechae263C_helpers.hw4.plotting import animate_2R_planar_arm_traj, plot_snapshots


def run_simulation(
    q_initial: NDArray[np.double],
    q_final: NDArray[np.double],
    B_avg: NDArray[np.double],
    K_p: NDArray[np.double],
    K_d: NDArray[np.double],
    simulation_duration_s: float,
    should_apply_control_torques: bool,
    control_period_s: float = 1e-3,
) -> tuple[
    NDArray[np.double],
    tuple[NDArray[np.double], NDArray[np.double]],
    tuple[NDArray[np.double], NDArray[np.double]],
    NDArray[np.double],
    Diagram,
]:
    """
    Runs a simulation with a desired joint position

    Parameters
    ----------
    q_initial:
        A numpy array of shape (2,) containing the initial joint positions

    q_final:
```

```
            A numpy array of shape (2,) containing the final desired joint positions

        B_avg:
            A numpy array of shape (2, 2) containing the average linearized inertia matrix

        K_p:
            A numpy array of shape (2, 2) containing the proportional gains of the inverse
            dynamics controller.

        K_d:
            A numpy array of shape (2, 2) containing the derivative gains of the inverse
            dynamics controller.

        control_period_s:
            The period between control commands in units of seconds

        simulation_duration_s:
            The duration of the simulation in units of seconds

        should_apply_control_torques:
            A bool that specifies that control torques should be simulated when set to
            `True`. (If set to `False` then no control torques are simulated).

        Returns
        -------
        A tuple with five elements:
            1. A numpy array with shape (T,) of simulation time steps
            2. A tuple of numpy arrays both with shape (2, T) of desired and actual joint
               positions corresponding to each simulation time step, respectively.
            3. A tuple of numpy arrays both with shape (2, T) of desired and actual joint
               velocities corresponding to each simulation time step, respectively.
            4. A numpy array with shape (2, T) of applied control torques corresponding to
               each simulation time step
            5. A Drake diagram
        """
        # ------------------------------------------------------------------------------
        # Add "systems" to a `DiagramBuilder` object.
        #   - "systems" are the blocks in a block diagram
        #   - Some examples for how to add named systems to a `DiagramBuilder` are given
        #     below
        # ------------------------------------------------------------------------------
        builder = DiagramBuilder()

        # Create the desired joint angle, velocity, and acceleration trajectories
        dt = control_period_s
        times = np.arange(0, simulation_duration_s + dt, dt)
        waypoint_times = np.asarray([0, simulation_duration_s / 2, simulation_duration_s])
        waypoints = np.stack([q_initial, np.deg2rad([130, -110]), q_final], axis=1)

        q_d_traj, qdot_d_traj, qddot_d_traj = eval_cubic_spline_traj(
```

```
        times=times, waypoint_times=waypoint_times, waypoints=waypoints
    )
    q_traj = builder.AddNamedSystem(
        "q_d_traj",
        JointSpaceTrajectorySource(
            name="q_d_traj",
            num_joints=q_d_traj.shape[0],
            times=times,
            joint_coordinates=q_d_traj,
        ),
    )
    qdot_traj = builder.AddNamedSystem(
        "qdot_d_traj",
        JointSpaceTrajectorySource(
            name="qdot_d_traj",
            num_joints=qdot_d_traj.shape[0],
            times=times,
            joint_coordinates=qdot_d_traj,
        ),
    )
    if should_apply_control_torques:
        qddot_traj = builder.AddNamedSystem(
            "qddot_d_traj",
            JointSpaceTrajectorySource(
                name="qddot_d_traj",
                num_joints=qddot_d_traj.shape[0],
                times=times,
                joint_coordinates=qddot_d_traj,
            ),
        )

        K_p_gain = builder.AddNamedSystem(
            "K_p", MatrixGain(np.asarray(K_p, dtype=np.double))
        )
        K_d_gain = builder.AddNamedSystem(
            "K_d", MatrixGain(np.asarray(K_d, dtype=np.double))
        )

    joint_position_error = builder.AddNamedSystem(
        "joint_position_error",
        LinearCombination(input_coeffs=(1, -1), input_shapes=(2,)),
    )
    joint_velocity_error = builder.AddNamedSystem(
        "joint_velocity_error",
        LinearCombination(input_coeffs=(1, -1), input_shapes=(2,)),
    )
    arm = builder.AddNamedSystem("arm", Arm())

    if should_apply_control_torques:
        control_torque = builder.AddNamedSystem(
```

```python
            "u",
            LinearCombination(input_coeffs=(1, 1, 1), input_shapes=(2,))
        )
        inertia_matrix = builder.AddNamedSystem("B_avg", MatrixGain(B_avg))

    # ---------------------------------------------------------------------------------
    # Connect the systems in the `DiagramBuilder` (i.e. add arrows of block diagram)
    # ---------------------------------------------------------------------------------
    # `builder.ExportInput(input_port)` makes the provided "input_port" into an input
    # of the entire diagram
    # The functions system.get_input_port() returns the input port of the given system
    #   - If there is more than one input port, you must specify the index of the
    #     desired input
    # The functions system.get_output_port() returns the output port of the given system
    #   - If there is more than one output port, you must specify the index of the
    #     desired output
    builder.Connect(q_traj.get_output_port(), joint_position_error.get_input_port(0))
    builder.Connect(qdot_traj.get_output_port(), joint_velocity_error.get_input_port(0))

    if should_apply_control_torques:
        builder.Connect(qddot_traj.get_output_port(), inertia_matrix.get_input_port())

    # Delclaring output of the system
    joint_velocity_output = arm.get_output_port(0)
    joint_position_output = arm.get_output_port(1)

    # TODO:
    #   Replace any `...` below with the correct system and values. Please keep the
    #   system names the same
    builder.Connect(joint_position_output, joint_position_error.get_input_port(1))
    builder.Connect(joint_velocity_output, joint_velocity_error.get_input_port(1))
    if should_apply_control_torques:
        builder.Connect(joint_position_error.get_output_port(), K_p_gain.get_input_port())
        builder.Connect(joint_velocity_error.get_output_port(), K_d_gain.get_input_port())

        #
        builder.Connect(
            inertia_matrix.get_output_port(), control_torque.get_input_port(0)
        )
        builder.Connect(K_p_gain.get_output_port(), control_torque.get_input_port(1))
        builder.Connect(K_d_gain.get_output_port(), control_torque.get_input_port(2))
        builder.Connect(control_torque.get_output_port(), arm.get_input_port())
    else:
        builder.ExportInput(arm.get_input_port(), name="control_torque")

    # ---------------------------------------------------------------------------------
    # Log joint 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
```

```python
    # `publish_period` seconds in simulation time.
    joint_position_logger = LogVectorOutput(
        arm.get_output_port(1), builder, publish_period=control_period_s
    )
    joint_velocity_logger = LogVectorOutput(
        arm.get_output_port(0), builder, publish_period=control_period_s
    )
    if should_apply_control_torques:
        control_torque_logger = LogVectorOutput(
            control_torque.get_output_port(), builder, publish_period=control_period_s
        )

    # ------------------------------------------------------------------------------
    # Setup/Run the simulation
    # ------------------------------------------------------------------------------
    # This line builds a `Diagram` object and uses it to make a `Simulator` object for
    # the diagram
    diagram: Diagram = builder.Build()
    diagram.set_name("Inverse Dynamics Controller")
    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()
    initial_conditions.SetAtIndex(2, q_initial[0])
    initial_conditions.SetAtIndex(3, q_initial[1])

    if not should_apply_control_torques:
        diagram.get_input_port().FixValue(context, np.zeros((2,)))

    # 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
    joint_position_log = joint_position_logger.FindLog(simulator.get_context())
    t = joint_position_log.sample_times()
    q_actual = joint_position_log.data()

    joint_velocity_log = joint_velocity_logger.FindLog(simulator.get_context())
    qdot_actual = joint_velocity_log.data()

    control_torques = np.zeros((2, len(t)), dtype=np.double)

    if should_apply_control_torques:
```

```
        control_torque_log = control_torque_logger.FindLog(simulator.get_context())
        control_torques = control_torque_log.data()

    # Return a `tuple` of required results
    return t, (q_d_traj, q_actual), (qdot_d_traj, qdot_actual), control_torques, diagram


if __name__ == "__main__":
    ##################################################################################
    # Section 1
    ##################################################################################
    # --------------------------------------------------------------------------------
    # TODO:
    #   Replace `...` with the correct values for each parameter
    # --------------------------------------------------------------------------------
    # The below functions might be helpful:
    #   np.diag: https://numpy.org/doc/stable/reference/generated/numpy.diag.html
    #   np.eye: https://numpy.org/doc/stable/reference/generated/numpy.eye.html
    a_1 = a_2 = 1
    l_1 = l_2 = 0.5
    m_l1 = m_l2 = 9
    I_l1 = I_l2 = 3
    m_m1 = m_m2 = 1
    I_m1 = I_m2 = 0.007
    k_r1 = k_r2 = 50

    K_p = np.diag([1400, 1400])
    K_d = np.diag([1200, 1200])

    # an_norm = lambda an:an%360.0

    # --------------------------------------------------------------------------------
    # TODO:
    #   Replace `...` with the correct values for the diagonal terms of the "averaged"
    #   generalized inertia matrix. These terms can be found by taking the small angle
    #   approximation of the elements in the full generalized inertia matrix given in
    #   the problem statement.
    # --------------------------------------------------------------------------------
    B_avg = np.zeros((2, 2))
    B_avg[0, 0] = (I_l1) + (m_l1 * l_1**2) + (I_m1 * k_r1**2) + (I_l2) + (m_l2*(a_1**2 +
l_2**2 + 2*a_1*l_2))
    B_avg[1, 1] = (I_l2) + (m_l2 * l_2**2) + (I_m2 * k_r2**2)

    # --------------------------------------------------------------------------------
    # TODO:
    #   Replace `...` with the initial and final joint configurations specified in the
    #   problem statement.
    # --------------------------------------------------------------------------------
    q_initial = np.deg2rad([30, -60])
    q_final = np.deg2rad([240, 60])
```

```python

    # ---------------------------------------------------------------------------------
    # Simulate without control torques
    # TODO:
    #   Replace `...` with the correct values to simulate the un-actuated dynamics of
    #   the planar 2R manipulator.
    # ---------------------------------------------------------------------------------
    t, (q_d, q), (qd_tra, qd_act), ct, diagram = run_simulation(
        q_initial=q_initial,
        q_final=q_final,
        B_avg=B_avg,
        K_p=K_p,
        K_d=K_d,
        simulation_duration_s=2.5,
        should_apply_control_torques=False,  # True or False?
    )
    fig, ax = plot_diagram(diagram)
    fig.savefig("Part1_figs/no_control_torque_diagram.png", dpi=300)
    # Convert `q` and `q_d` to degrees
    q_d = (np.rad2deg(q_d))
    q = (np.rad2deg(q))
    print('Finish part 1 simulation')

    # ---------------------------------------------------------------------------------
    # TODO:
    #   Using the link lengths `[a_1, a_2]`, the simulated joint positions `q`, and the
    #   `calc_fk_2D` function to calculate the xy positions of each joint of the
    #   manipulator for the simulated scenario. (Replace `...` with the correct values)
    #
    #   Hint: Make sure to convert `q` back to radians before using it with `calc_fk_2D`
    #         (using np.deg2rad).
    #
    # ---------------------------------------------------------------------------------
    joint_xs, joint_ys = calc_fk_2D(link_lens=[a_1, a_2], joint_positions=np.deg2rad(q))

    # ---------------------------------------------------------------------------------
    # TODO:
    #   Replace all `...` in the call of the `animate_2R_planar_arm_traj` function with
    #   the correct output of the `calc_fk_2D`.
    # ---------------------------------------------------------------------------------
    print('Saving part 1 plots...')
    _, _, anim_no_control_torques = animate_2R_planar_arm_traj(
        joint_xs=joint_xs,
        joint_ys=joint_ys,
        animation_file_name="no_control_torques_animation"
    )
    # anim_no_control_torques.save('Part1_figs/no_control_torques_animation.mp4')

    # ---------------------------------------------------------------------------------
    # Plot Snapshots
```

```
    # TODO:
    #   Replace all `...` in the call of the `plot_snapshots` function with
    #   the correct output of the `calc_fk_2D` and the dt specified in the problem
    #   statement.
    #   Add code to properly label `ax` and save `fig`
    # ---------------------------------------------------------------------------------
    fig, ax = plot_snapshots(dt=0.1, joint_xs=joint_xs, joint_ys=joint_ys)
    ax.set_xlabel('X Position [m]')
    ax.set_ylabel('Y Position [m]')
    ax.set_title('Snapshots of no control torque motion')
    fig.savefig('Part1_figs/Part1_Snapshots.png', dpi=300)
    print('Saved part 1 plots')

    ##################################################################################
    # Section 2
    ##################################################################################
    # ---------------------------------------------------------------------------------
    # Simulate with control torques
    # TODO:
    #   Replace `...` with the correct values to simulate the dynamics of
    #   the planar 2R manipulator under your inverse dynamics controller.
    # ---------------------------------------------------------------------------------
    t, (q_d, q), (qdot_d, qdot), control_torques, diagram = run_simulation(
        q_initial=q_initial,
        q_final=q_final,
        B_avg=B_avg,
        K_p=K_p,
        K_d=K_d,
        simulation_duration_s=2.5,
        should_apply_control_torques=True,  # True or False?
    )
    fig, ax = plot_diagram(diagram)
    fig.savefig("Part2_figs/control_torque_diagram.png", dpi=300)
    print('Finish part 2 simulation')

    # Convert `q`, `q_d`, `qdot`, and `qdot_d`  to degrees
    q_d = (np.rad2deg(q_d))
    q = (np.rad2deg(q))
    qdot_d = np.rad2deg(qdot_d)
    qdot = np.rad2deg(qdot)

    # ---------------------------------------------------------------------------------
    # Animate Trajectory
    # TODO:
    #   Using the link lengths `[a_1, a_2]`, the actual joint positions `q` and desired
    #   joint positions `q_d`, with the `calc_fk_2D` function to calculate the xy
    #   positions of each joint of the manipulator for the actual and desired
    #   trajectories, respectively. (Replace `...` with the correct values)
    #
    #   Hint: Make sure to convert `q` back to radians before using it with `calc_fk_2D`
```

```python
    #         (using np.deg2rad).
    #
    # ---------------------------------------------------------------------------------
    joint_xs, joint_ys = calc_fk_2D(link_lens=[a_1, a_2], joint_positions=np.deg2rad(q))
    joint_xs_desired, joint_ys_desired = calc_fk_2D(link_lens=[a_1, a_2],
joint_positions=np.deg2rad(q_d))

    # ---------------------------------------------------------------------------------
    # TODO:
    #   Replace all `...` in the call of the `animate_2R_planar_arm_traj` function with
    #   the correct output of the `calc_fk_2D`.
    # ---------------------------------------------------------------------------------
    print('Saving part 2 plots...')
    _, _, anim_control_torques = animate_2R_planar_arm_traj(
        joint_xs=joint_xs,
        joint_ys=joint_ys,
        animation_file_name="control_torques_animation"
    )
    # anim_control_torques.save('Part2_figs/control_torques_animation', 'Pillow', 20)

    # ---------------------------------------------------------------------------------
    # Plot Snapshots
    # TODO:
    #   Replace all `...` in the call of the `plot_snapshots` function with
    #   the correct output of the `calc_fk_2D` and the dt specified in the problem
    #   statement.
    #   Add code to properly label `ax` and save `fig`
    # ---------------------------------------------------------------------------------
    fig, ax = plot_snapshots(
        dt=0.1,
        joint_xs=joint_xs,
        joint_ys=joint_ys,
        joint_xs_desired=joint_xs_desired,
        joint_ys_desired=joint_ys_desired,
    )

    ax.set_xlabel('X Position [m]')
    ax.set_ylabel('Y Position [m]')
    ax.set_title('Snapshots of control torque motion')
    fig.savefig('Part2_figs/Part2_Snapshots.png', dpi=300)
    print('Saved part 2 plots...')
    # ---------------------------------------------------------------------------------
    # Plot Joint Position Error
    # TODO:
    #   Replace `...` with the code to make the specified joint position error plot for
    #   the inverse dynamics controller case.
    #
    # Hints:
    # 1. To plot a black dashed vertical line at `x = x0` use the `ax.axvline` function:
    #       `ax.axvline(x0, ls="--", color="black")
```

```
    # 2. When plotting, use the `label` argument to automatically add a legend item:
    #       `ax.plot(x, y, color="red", label=r"$\theta_1$ Error")`
    # 3. You need to call `ax.legend()` to actually plot the legend.
    # ----------------------------------------------------------------------------------
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    # joint 1 error
    ax.plot(t, np.rad2deg(q_d[0] - q[0]), color='red', label='Joint 1 Error')
    # joint 2 error
    ax.plot(t, np.rad2deg(q_d[1] - q[1]), color='blue', label='Joint 2 Error')
    ax.axhline(-2, ls="--", color="black")
    ax.axhline(2, ls="--", color="black")
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Position Error [deg]')
    ax.set_title('Position error of joints vs. time')
    ax.legend()
    fig.savefig('JointPlots/JointError.png', dpi=300)
    plt.clf()
    print('Saved Joint Error')

    # ----------------------------------------------------------------------------------
    # Plot Joint Positions
    # TODO:
    #   Replace `...` with the code to make the specified joint position plot for the
    #   inverse dynamics controller case.
    # ----------------------------------------------------------------------------------
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    # joint 1 position
    ax.plot(t, np.rad2deg(q[0]), color='red', label='q_1')
    ax.plot(t, np.rad2deg(q_d[0]), color='red', ls="--", label='q_1d')
    # joint 2 position
    ax.plot(t, np.rad2deg(q[1]), color='blue', label='q_2')
    ax.plot(t, np.rad2deg(q_d[1]), color='blue', ls="--", label='q_2d')
    ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Position [deg]')
    ax.set_title('Position of joints vs. time')
    ax.legend()
    fig.savefig('JointPlots/JointPosition.png', dpi=300)
    plt.clf()
    print('Saved Joint Position')

    # ----------------------------------------------------------------------------------
    # Plot Joint Velocities
    # TODO:
    #   Replace `...` with the code to make the specified joint velocity plot for the
    #   inverse dynamics controller case.
    # ----------------------------------------------------------------------------------
    fig = plt.figure()
```

```python
    ax = fig.add_subplot(1, 1, 1)
    # joint 1 velocity
    ax.plot(t, np.rad2deg(qdot[0]), color='red', label='qdot_1')
    ax.plot(t, np.rad2deg(qdot_d[0]), color='red', ls="--", label='qdot_1d')
    # joint 2 velocity
    ax.plot(t, np.rad2deg(qdot[1]), color='blue', label='qdot_2')
    ax.plot(t, np.rad2deg(qdot_d[1]), color='blue', ls="--", label='qdot_2d')
    ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Joint velocity [deg/sec]')
    ax.set_title('Velocity of joints vs. time')
    ax.legend()
    fig.savefig('JointPlots/JointVelocity.png', dpi=300)
    plt.clf()
    print('Saved Joint Velocities')

    # ---------------------------------------------------------------------------------
    # Plot Control Torques
    # TODO:
    #   Replace `...` with the code to make the specified control torque plot for the
    #   inverse dynamics controller case.
    # ---------------------------------------------------------------------------------
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    # joint 1 torques
    ax.plot(t, np.rad2deg(control_torques[0]), color='red', label='tau_1')
    # joint 2 torques
    ax.plot(t, np.rad2deg(control_torques[1]), color='blue', label='tau_2')
    ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
    ax.set_xlabel('Time [s]')
    ax.set_ylabel('Joint torques [Nm]')
    ax.set_title('Joint torques vs. time')
    ax.legend()
    fig.savefig('JointPlots/JointTorques.png', dpi=300)
    plt.clf()
    print('Saved Joint Torques')
```