mechae263C_homework5_problem2.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,
    OutputPort,
)
from pydrake.systems.primitives import (
    MatrixGain,
    PassThrough,
    ZeroOrderHold,
    LogVectorOutput,
)

from mechae263C_helpers.drake import LinearCombination, plot_diagram
from mechae263C_helpers.hw5 import validate_np_array
from mechae263C_helpers.hw5.kinematics import calc_2R_planar_inverse_kinematics
from mechae263C_helpers.hw5.op_space import Environment, OperationalSpaceDecoupledArm
from mechae263C_helpers.hw5.trajectory import (
    PrePlannedTrajectorySource,
    eval_trapz_traj,
)


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
```

```python
    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 OperationalSpaceImpedanceController(Diagram):
    def __init__(
        self,
        link_lens: tuple[float, float],
        M_d: NDArray[np.double],
        K_P: NDArray[np.double],
        K_D: NDArray[np.double],
        control_sample_period_s: float,
        trajectory_duration_s: float,
        p_initial: NDArray[np.double],
        p_final: 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=M_d, arr_name="M_d", correct_shape=(2, 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_initial, arr_name="p_initial", correct_shape=(2,))
        validate_np_array(arr=p_final, arr_name="p_desired", correct_shape=(2,))

        self.M_d = M_d
        self.K_P = K_P
        self.K_D = K_D

        builder = DiagramBuilder()

        invM_d: MatrixGain = builder.AddNamedSystem(
            "invM_d", MatrixGain(np.linalg.inv(M_d))
        )
        K_P: MatrixGain = builder.AddNamedSystem("K_P", MatrixGain(K_P))
        K_D: MatrixGain = builder.AddNamedSystem("K_D", MatrixGain(K_D))

        operational_space_position_error: LinearCombination = builder.AddNamedSystem(
```

```
            "o_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
        )
        operational_space_velocity_error: LinearCombination = builder.AddNamedSystem(
            "odot_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
        )
        operational_space_control_action: LinearCombination = builder.AddNamedSystem(
            "f_c", LinearCombination(input_coeffs=(1, 1, -1), input_shapes=(2,))
        )
        control_torques: LinearCombination = builder.AddNamedSystem(
            "u", LinearCombination(input_coeffs=(1, 1), input_shapes=(2,))
        )

        traj_times = np.arange(
            0, simulation_duration_s + control_sample_period_s, control_sample_period_s
        )
        o_d, odot_d, oddot_d = eval_trapz_traj(
            times=traj_times,
            max_velocity=0.5,
            final_time=trajectory_duration_s,
            initial_position=p_initial,
            final_position=p_final,
        )

        o_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
            "o_d",
            PrePlannedTrajectorySource(
                name="o_d",
                num_joints=self.num_dofs,
                times=traj_times,
                values=o_d,
            ),
        )
        odot_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
            "odot_d",
            PrePlannedTrajectorySource(
                name="odot_d",
                num_joints=self.num_dofs,
                times=traj_times,
                values=odot_d,
            ),
        )
        oddot_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
            "oddot_d",
            PrePlannedTrajectorySource(
                name="oddot_d",
                num_joints=self.num_dofs,
                times=traj_times,
                values=oddot_d,
            ),
        )
```

```python

        zoh: ZeroOrderHold = builder.AddNamedSystem(
            "sampled_oddot_e",
            ZeroOrderHold(
                period_sec=self.control_sample_period_s, vector_size=self.num_dofs
            ),
        )

        o_e: PassThrough = builder.AddNamedSystem("o_e", PassThrough(vector_size=2))
        odot_e: PassThrough = builder.AddNamedSystem(
            "odot_e", PassThrough(vector_size=2)
        )
        u: PassThrough = builder.AddNamedSystem("command", PassThrough(vector_size=2))
        f_e: PassThrough = builder.AddNamedSystem("f_e", PassThrough(vector_size=2))

        # ===============================================================================
        # TODO: Complete Controller Block Diagram
        #   Replace `...` below with the correct output or input port.
        # -------------------------------------------------------------------------------
        builder.Connect(
            o_d.get_output_port(),
            operational_space_position_error.get_input_port(0)
        )
        builder.Connect(
            o_e.get_output_port(),
            operational_space_position_error.get_input_port(1)
        )
        builder.Connect(
            odot_d.get_output_port(),
            operational_space_velocity_error.get_input_port(0)
        )
        builder.Connect(
            odot_e.get_output_port(),
            operational_space_velocity_error.get_input_port(1)
        )

        builder.Connect(
            oddot_d.get_output_port(),
            control_torques.get_input_port(0)
        )
        # Find the difference then apply the gain matrix
        builder.Connect(
            operational_space_position_error.get_output_port(),
            K_P.get_input_port()
        )

        # Operational control action
        builder.Connect(
            operational_space_velocity_error.get_output_port(),
            K_D.get_input_port()
```

```
        )
        builder.Connect(
            K_D.get_output_port(),
            operational_space_control_action.get_input_port(0)
        )
        # sum before inverse mass
        builder.Connect(
            K_P.get_output_port(),
            operational_space_control_action.get_input_port(1)
        )
        builder.Connect(
            f_e.get_output_port(),
            operational_space_control_action.get_input_port(2)
        )

        builder.Connect(
            operational_space_control_action.get_output_port(),
            invM_d.get_input_port()
        )
        builder.Connect(
            invM_d.get_output_port(),
            control_torques.get_input_port(1))
        # ================================================================================

        # 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(zoh.get_output_port(), u.get_input_port())

        builder.ExportInput(o_e.get_input_port(), "o_e")
        builder.ExportInput(odot_e.get_input_port(), "odot_e")
        builder.ExportInput(f_e.get_input_port(), "f_e")
        builder.ExportOutput(u.get_output_port(), "u")

        # --------------------------------------------------------------------------------
        # Log Position Error and Force
        # --------------------------------------------------------------------------------
        # 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.force_logger = LogVectorOutput(
            f_e.get_output_port(),
            builder,
            publish_period=control_sample_period_s,
        )
        self.force_logger.set_name("Force Logger")
        self.position_error_logger = LogVectorOutput(
            operational_space_position_error.get_output_port(),
            builder,
            publish_period=control_sample_period_s,
```

```
        )
        self.position_error_logger.set_name("Tip Position Error Logger")

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

    def get_o_e_input_port(self) -> InputPort:
        """Returns the input port for operational space end-effector position"""
        return self.get_input_port(0)

    def get_odot_e_input_port(self) -> InputPort:
        """Returns the input port for operational space end-effector velocity"""

        return self.get_input_port(1)

    def get_f_e_input_port(self) -> InputPort:
        """Returns the input port for operational space end-effector force"""
        return self.get_input_port(2)

    def get_u_output_port(self) -> OutputPort:
        """Returns the output port for the control command"""
        return self.get_output_port()


def run_simulation(
    simulation_duration_s: float,
    link_lens: tuple[float, float],
    M_d: NDArray,
    K_P: NDArray,
    K_D: NDArray,
    p_initial: NDArray[np.double],
    p_final: NDArray[np.double],
    trajectory_duration_s: float,
    o_r: NDArray[np.double],
    K: NDArray,
    control_sample_period_s: float,
):
    """
    Runs a Drake simulation of an operational space impedance controller
    Parameters
    ----------
    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)

    M_d
        A numpy array of shape (2, 2) representing the inertia gains of the impedance
        controller, expressed in the base frame
```

```

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

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

    p_initial
        A numpy array of shape (2,) representing the initial position of the
        end-effector, expressed in the base frame

    p_final
        A numpy array of shape (2,) representing the final position of the end-effector,
        expressed in the base frame

    trajectory_duration_s
        A float representing the duration of the trajectory in seconds

    o_r
        A numpy array of shape (2,) representing the base frame coordinates for the
        point where the undeformed elastically compliant plane intersects the base frame
        x-axis

    K
        A numpy array of shape (2, 2) representing the environment's stiffness matrix in
        the base frame (see hint in problem statement)

    control_sample_period_s
        A float representing the duration of the trajectory in seconds

    Returns
    -------
    A tuple of five elements:
        1) The time-steps of the simulation in seconds
        2) The simulated end-effector forces in Newtons corresponding to each time-step
        3) The simulated position errors in meters 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_initial, arr_name="p_initial", correct_shape=(2,))
    validate_np_array(arr=p_final, arr_name="p_final", correct_shape=(2,))

    builder = DiagramBuilder()
    arm: OperationalSpaceDecoupledArm = builder.AddNamedSystem(
        "arm",
        OperationalSpaceDecoupledArm(
            link_lens, calc_direct_kinematics, calc_analytical_jacobian
        ),
```

```
    )
    controller: OperationalSpaceImpedanceController = builder.AddNamedSystem(
        "controller",
        OperationalSpaceImpedanceController(
            link_lens=link_lens,
            M_d=M_d,
            K_P=K_P,
            K_D=K_D,
            control_sample_period_s=control_sample_period_s,
            p_initial=p_initial,
            p_final=p_final,
            trajectory_duration_s=trajectory_duration_s,
        ),
    )
    environment: Environment = builder.AddNamedSystem(
        "environment", Environment(equilibrium_position=o_r, stiffness=K)
    )

    # ================================================================================
    # TODO: Complete High-Level Simulation Block Diagram: Arm + Controller + Environment
    #   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_f_e_input_port()
    #       - arm.get_u_input_port()
    #       - arm.get_o_e_output_port()
    #       - arm.get_odot_e_output_port()
    #
    #       controller:
    #       - controller.get_f_e_input_port()
    #       - controller.get_o_e_input_port()
    #       - controller.get_odot_e_input_port()
    #       - controller.get_u_output_port()
    #
    #       environment
    #       - environment.get_o_e_input_port()
    #       - environment.get_f_e_output_port()
    # --------------------------------------------------------------------------------
    builder.Connect(controller.get_u_output_port(),
                    arm.get_u_input_port())

    builder.Connect(arm.get_o_e_output_port(),
                    controller.get_o_e_input_port())
    builder.Connect(arm.get_o_e_output_port(),
                    environment.get_o_e_input_port())
    builder.Connect(arm.get_odot_e_output_port(),
                    controller.get_odot_e_input_port())

    builder.Connect(environment.get_f_e_output_port(),
```

```python
                    arm.get_f_e_input_port())
    builder.Connect(environment.get_f_e_output_port(),
                    controller.get_f_e_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 Impedance Control")
    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_initial, 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.set_target_realtime_rate(1.0)
    simulator.AdvanceTo(simulation_duration_s)

    # --------------------------------------------------------------------------------
    # Extract simulation outputs
    # --------------------------------------------------------------------------------
    # The lines below extract the joint position log from the simulator context
    force_log = controller.force_logger.FindLog(simulator.get_context())
    t = force_log.sample_times()
    force = force_log.data()
    position_error_log = controller.position_error_logger.FindLog(
        simulator.get_context()
    )
    position_error = position_error_log.data()

    return t, force, position_error, controller, diagram


if __name__ == "__main__":
    # ================================================================================
    # TODO: Problem 2 - Part (a)
    #   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
```

```python

    # A numpy array of shape (2, 2) representing the rotation matrix that rotates from
    # the base frame to the constraint frame
    R_c = np.array([[np.cos(np.pi/4), -np.sin(np.pi/4)],
                    [np.sin(np.pi/4), np.cos(np.pi/4)]])

    # A numpy array of shape (2, 2) representing the environment's stiffness matrix in
    # the constraint frame
    K_c = np.array([[0, 0],
                    [0, 5000]])

    # A numpy array of shape (2, 2) representing the environment's stiffness matrix in
    # the base frame (see hint in problem statement)
    K = R_c @ K_c @ np.transpose(R_c)

    # A numpy array of shape (2,) representing the base frame coordinates for the point
    # where the undeformed elastically compliant plane intersects the base frame x-axis
    o_r = np.array([1, 0])

    # A numpy array of shape (2, 2) representing the impedance controller proportional
    # gains in the constraint frame
    K_P_c = np.array([[1.2e7, 0],
                      [0, 1.2e7]])

    # A numpy array of shape (2, 2) representing the impedance controller derivative
    # gains in the constraint frame
    K_D_c = np.array([[1.15e5, 0],
                      [0, 1.15e5]])

    # A numpy array of shape (2, 2) representing the impedance controller inertia gains
    # in the constraint frame
    M_d_c = np.array([[5e3, 0],
                      [0, 5e3]])

    # A numpy array of shape (2, 2) representing the impedance controller proportional
    # gains in the base frame (see hint in problem statement)
    K_P = R_c @ K_P_c @ np.transpose(R_c)

    # A numpy array of shape (2, 2) representing the impedance controller derivative
    # gains in the base frame (see hint in problem statement)
    K_D = R_c @ K_D_c @ np.transpose(R_c)

    # A numpy array of shape (2, 2) representing the impedance controller inertia gains
    # in the base frame (see hint in problem statement)
    M_d = R_c @ M_d_c @ np.transpose(R_c)

    # Print out your impedance controller gains
    print("M_d_c:")
    print(M_d_c)
    print("\nK_P_c:")
```

```python
    print(K_P_c)
    print("\nK_D_c:")
    print(K_D_c)

    # A numpy array of shape (2,) representing the initial end-effector position in base
    # frame
    p_initial = np.array([1 + 0.1*np.sqrt(2), 0])

    # A numpy array of shape (2,) representing the final end-effector position in base
    # frame
    p_final = np.array([1.2 + 0.1*np.sqrt(2), 0.2])

    # A float representing the duration of the trapezoidal velocity trajectory in units
    # of seconds
    trajectory_duration_s = 2

    # 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

    # ---------------------------------------------------------------------------------
    # TODO: Run Simulation
    #   Replace `...` in parameters for `run_simulation` function using the variables
    #   above
    # ---------------------------------------------------------------------------------
    t, forces, position_error, controller_diagram, simulation_diagram = run_simulation(
        simulation_duration_s=simulation_duration_s,
        link_lens=link_lens,
        M_d=M_d,
        K_P=K_P,
        K_D=K_D,
        p_initial=p_initial,
        p_final=p_final,
        trajectory_duration_s=trajectory_duration_s,
        control_sample_period_s=control_sample_period_s,
        o_r=o_r,
        K=K,
    )

    print('sim finished')

    # ---------------------------------------------------------------------------------
    # TODO: Plot Control 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, max_depth=1
```

```python
    # )
    # controller_diagram_fig.savefig('Problem2/controlDia.png', dpi=300)
    # print('saved controller 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, max_depth=1
    # )
    # simulation_diagram_fig.savefig('Problem2/simulationDia.png', dpi=300)
    # print('saved simulation diagram')
    # ==================================================================================

    # ==================================================================================
    # TODO: Problem 2 - Part (b)
    #   Report/output the damping ratio and natural frequency in both the x_c and y_c
    #   directions.
    #   Hint: See Example 9.2 in Siciliano et al.
    # ----------------------------------------------------------------------------------
    kpx = K_P_c[0][0]
    kpy = K_P_c[1][1]
    kdx = K_D_c[0][0]
    kdy = K_D_c[1][1]
    mdx = M_d_c[0][0]
    mdy = M_d_c[1][1]
    kx = K_c[0][0]
    ky = K_c[1][1]

    natural_frequency_x = np.sqrt(kpx / mdx)
    natural_frequency_y =  np.sqrt((kpy + ky) / mdy)
    damping_ratio_x = (kdx) / (2*np.sqrt(mdx*kpx))
    damping_ratio_y = (kdy) / (2*np.sqrt(mdy*(kpy + ky)))

    print("\nnatural_frequency_x:", natural_frequency_x)
    print("\nnatural_frequency_y:", natural_frequency_y)
    print("\ndamping_ratio_x:", damping_ratio_x)
    print("\ndamping_ratio_y:", damping_ratio_y)
    # ==================================================================================

    # ==================================================================================
    # TODO: Problem 2 - Part (c)
    #   1) Plot x and y-coordinate of end effector position errors in meters as a
    #      function of time, as expressed in the **base** frame (on the same figure).
    #      Set the x limits to [0, `simulation_duration_s`] and the y limits to
    #      [-0.06, 0.06].
    #   2) Plot the x-and y-coordinate end-effector contact forces in N as a
    #      function of time, as expressed in the **base** frame (on the same figure).
```

```python
    #      Set the x limits to [0, `simulation_duration_s`] and the y limits to
    #      [-550, 550].
    # Hints:
    #   1) When plotting, use the `label` argument to automatically add a legend item:
    #      `ax.plot(x, y, label=r"$x_0$")`
    #   2) You need to call `ax.legend()` to actually plot the legend.
    # ----------------------------------------------------------------------------------
    # Plot data in `position_error` variable
    fig = plt.figure(figsize=(10, 5))
    base_error = fig.add_subplot(111)

    # Label Plots
    base_error.set_title("Base Frame: Position Error vs Time")
    base_error.set_xlabel("Time [s]")
    base_error.set_ylabel("Error [m]")
    base_error.set_xlim([0, simulation_duration_s])
    base_error.set_ylim([-0.06, 0.06])

    base_error.plot(
        t, position_error[0], color="black", label="X Error"
    )
    base_error.plot(
        t, position_error[1], color="blue", label="Y Error"
    )
    base_error.legend()
    fig.savefig('Problem2/Base_PositionError.png', dpi=300)
    print("plotted base position errors")
    plt.clf


    # Plot data in `forces` variable
    fig = plt.figure(figsize=(10, 5))
    base_force = fig.add_subplot(111)

    # Label Plots
    base_force.set_title("Base Frame: Contact Force vs Time")
    base_force.set_xlabel("Time [s]")
    base_force.set_ylabel("Force [N]")
    base_force.set_xlim([0, simulation_duration_s])
    base_force.set_ylim([-550, 550])

    base_force.plot(
        t, forces[0], color="black", label="X Contact Force"
    )
    base_force.plot(
        t, forces[1], color="blue", label="Y Contact Force"
    )
    base_force.legend()
    fig.savefig('Problem2/Base_Force.png', dpi=300)
    print("plotted base contact forces")
```

```python
    plt.clf
    # ==============================================================================

    # ==============================================================================
    # TODO: Problem 2 - Part (d)
    #   1) Plot x and y-coordinate of end effector position errors in meters as a
    #      function of time, as expressed in the **constraint** frame (on the same
    #      figure).
    #      Set the x limits to [0, `simulation_duration_s`] and the y limits to
    #      [-0.06, 0.06].
    #   2) Plot the x-and y-coordinate end-effector contact forces in N as a
    #      function of time, as expressed in the **constraint** frame (on the same
    #      figure).
    #      Set the x limits to [0, `simulation_duration_s`] and the y limits to
    #      [-550, 550].
    # ------------------------------------------------------------------------------
    position_error_in_constraint_frame = R_c @ position_error
    forces_in_constraint_frame = R_c @ forces

    # Plot data in `position_error` variable
    fig = plt.figure(figsize=(10, 5))
    con_error = fig.add_subplot(111)

    # Label Plots
    con_error.set_title("Constraint Frame: Position Error vs Time")
    con_error.set_xlabel("Time [s]")
    con_error.set_ylabel("Error [m]")
    con_error.set_xlim([0, simulation_duration_s])
    con_error.set_ylim([-0.06, 0.06])

    con_error.plot(
        t, position_error_in_constraint_frame[0], color="black", label="X Error"
    )
    con_error.plot(
        t, position_error_in_constraint_frame[1], color="blue", label="Y Error"
    )
    con_error.legend()
    fig.savefig('Problem2/Cosntraint_PositionError.png', dpi=300)
    print("plotted cosntraint position errors")
    plt.clf


    # Plot data in `forces` variable
    fig = plt.figure(figsize=(10, 5))
    con_force = fig.add_subplot(111)

    # Label Plots
    con_force.set_title("Constraint Frame: Contact Force vs Time")
    con_force.set_xlabel("Time [s]")
    con_force.set_ylabel("Force [N]")
```

```python
    con_force.set_xlim([0, simulation_duration_s])
    con_force.set_ylim([-550, 550])

    con_force.plot(
        t, forces_in_constraint_frame[0], color="black", label="X Contact Force"
    )
    con_force.plot(
        t, forces_in_constraint_frame[1], color="blue", label="Y Contact Force"
    )
    con_force.legend()
    fig.savefig('Problem2/Constraint_Force.png', dpi=300)
    print("plotted cosntraint contact forces")
    plt.clf
    # ==================================================================================

```