Trevor Oshiro
263C

MAE C163C / C263C Homework #4

*(Due via Gradescope by **11:59pm PDT on Friday, 5/2**)*

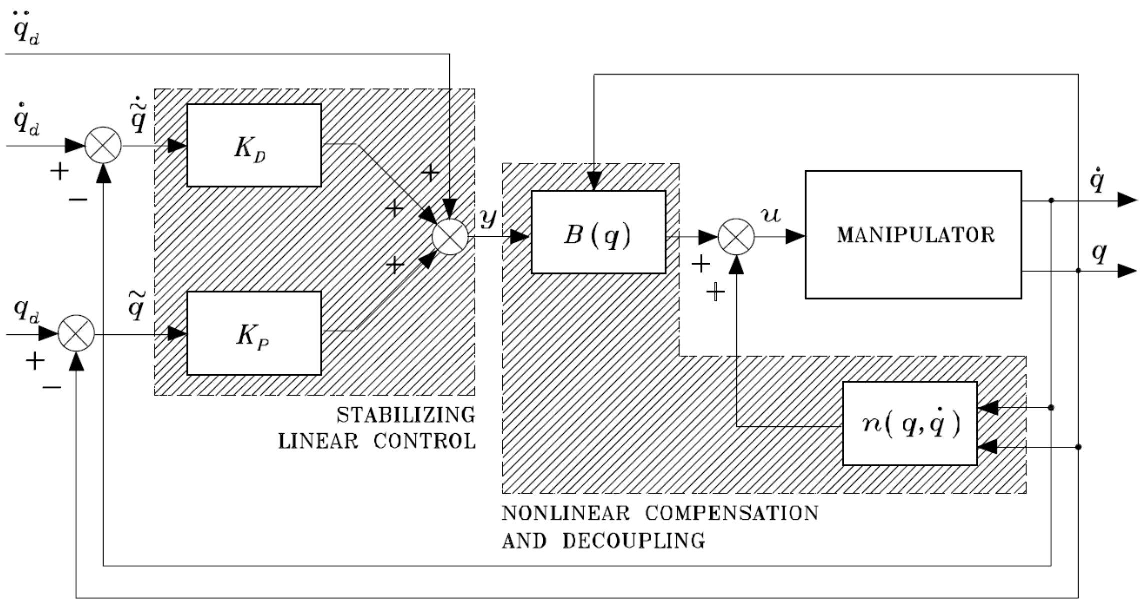
1. Analytical Jacobian

a) Report the 3x3 transformation matrix $T(\phi_e)$ that relates an angular velocity vector $\underline{\omega}_e$ and the time derivative of Euler angles $\dot{\underline{\phi}}_e$ through the equation $\underline{\omega}_e = T(\phi_e)\dot{\underline{\phi}}_e$. Consider XYX Euler angles that describe the (i) rotation of the reference frame by an angle φ about axis x, followed by (ii) the rotation of the current frame by an angle ϑ about axis y', followed by (iii) the rotation of the current frame by an angle ψ about axis x''.

b) Report the 6x6 transformation matrix $T_A(\phi_e)$ that relates the geometric Jacobian J and the analytical Jacobian J_A through the equation $J = T_A(\phi_e)J_A$.

Hint: Reference Siciliano et al., Sec. 2.4 for a review of Euler angles and Sec. 3.6 for a discussion of the analytical Jacobian.

2. Joint Space Inverse Dynamics Control



The block diagram above describes joint space inverse dynamics control.

Consider a two-link planar arm with the following parameters:

$$a_1 = a_2 = 1\ \text{m}, \quad l_1 = l_2 = 0.5\ \text{m}, \quad m_{l_1} = m_{l_2} = 9\ \text{kg}, \quad I_{l_1} = I_{l_2} = 3\ \text{kg}\cdot\text{m}^2$$

The arm is assumed to be driven by two identical actuators with the following parameters:

$$k_{r_1} = k_{r_2} = 50, \quad m_{m1} = m_{m2} = 1\ \text{kg}, \quad I_{m1} = I_{m2} = 0.007\ \text{kg}\cdot\text{m}^2,$$

$$F_{m1} = F_{m2} = 0.01\ \text{N}\cdot\text{m}\cdot\text{s/rad}$$

1. Analytical Jacobian

a) Report the 3x3 transformation matrix $T(\phi_e)$ that relates an angular velocity vector $\underline{\omega}_e$ and the time derivative of Euler angles $\dot{\underline{\phi}}_e$ through the equation $\underline{\omega}_e = T(\phi_e)\dot{\underline{\phi}}_e$. Consider XYX Euler angles that describe the (i) rotation of the reference frame by an angle φ about axis x, followed by (ii) the rotation of the current frame by an angle ϑ about axis y', followed by (iii) the rotation of the current frame by an angle ψ about axis x''.

b) Report the 6x6 transformation matrix $T_A(\phi_e)$ that relates the geometric Jacobian J and the analytical Jacobian J_A through the equation $J = T_A(\phi_e)J_A$.

<u>Hint:</u> Reference Siciliano et al., Sec. 2.4 for a review of Euler angles and Sec. 3.6 for a discussion of the analytical Jacobian.

a)

$$\dot{\varphi} = \dot{\varphi}\begin{bmatrix}1\\0\\0\end{bmatrix}$$

$$\dot{\vartheta} = \dot{\vartheta}\begin{bmatrix}1 & 0 & 0\\0 & \cos\varphi & -\sin\varphi\\0 & \sin\varphi & \cos\varphi\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix} = \dot{\vartheta}\begin{bmatrix}0\\\cos\varphi\\\sin\varphi\end{bmatrix}$$

$$\dot{\psi} = \dot{\psi}\begin{bmatrix}1 & 0 & 0\\0 & \cos\varphi & -\sin\varphi\\0 & \sin\varphi & \cos\varphi\end{bmatrix}\begin{bmatrix}\cos\vartheta & 0 & \sin\vartheta\\0 & 1 & 0\\-\sin\vartheta & 0 & \cos\vartheta\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}$$

$$= \dot{\psi}\begin{bmatrix}1 & 0 & 0\\0 & \cos\varphi & -\sin\varphi\\0 & \sin\varphi & \cos\varphi\end{bmatrix}\begin{bmatrix}\cos\vartheta\\0\\-\sin\vartheta\end{bmatrix}$$

$$= \dot{\psi}\begin{bmatrix}\cos\vartheta\\\sin\varphi\sin\vartheta\\-\cos\varphi\sin\vartheta\end{bmatrix}$$

$$T(\underline{\phi}_e) = \begin{bmatrix}1 & 0 & \cos\vartheta\\0 & \cos\varphi & \sin\varphi\sin\vartheta\\0 & \sin\varphi & -\cos\varphi\sin\vartheta\end{bmatrix}$$

b)

$$T_A(\underline{\Phi}_e) = \left[\begin{array}{c|c}I & 0\\\hline 0 & T(\underline{\Phi}_e)\end{array}\right]$$

$$T_A(\underline{\Phi}_e) = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\\0 & 1 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 0 & 0 & 0\\0 & 0 & 0 & 1 & 0 & \cos\vartheta\\0 & 0 & 0 & 0 & \cos\varphi & \sin\varphi\sin\vartheta\\0 & 0 & 0 & 0 & \sin\varphi & -\cos\varphi\sin\vartheta\end{bmatrix}$$

The generalized inertia matrix for an augmented link model of this planar 2R manipulator can be written as:

$$\mathrm{B}(\boldsymbol{q}) = \begin{bmatrix} B_{11} & m_{l_2} l_2 (a_1 \cos(q_2) + l_2) + k_{r_2} I_{m_2} + I_{l_2} \\ m_{l_2} l_2 (a_1 \cos(q_2) + l_2) + k_{r_2} I_{m_2} + I_{l_2} & B_{22} \end{bmatrix}$$

where

$$B_{11} = I_{l_1} + m_{l_1} l_1^2 + k_{r_1}^2 I_{m_1} + I_{l_2} + m_{l_2}(a_1^2 + l_2^2 + 2a_1 l_2 \cos(q_2))$$

and

$$B_{22} = I_{l_2} + m_{l_2} l_2^2 + k_{r_2}^2 I_{m_2}$$

The motion of this manipulator is subject to the following constraints:

1) The 4th quadrant represents a fixed obstacle such that no part of the manipulator can cross over the positive x-axis or negative y-axis into the 4th quadrant.
2) There exists a "ceiling" at y = 1.5 m such that no part of the manipulator can invade the space y > 1.5 m.
3) The "elbow" joint represented by joint coordinate q_2 is restricted such that link 2 cannot swing through link 1. That is, $-\pi \leq q_2 \leq \pi$.

<u>Implement the Drake Simulation Python function</u>:

Complete the function `run_simulation` in *HW4.py* using the inline instructions within the comments.

<u>Simulate the Passive Dynamics of the Manipulator</u>

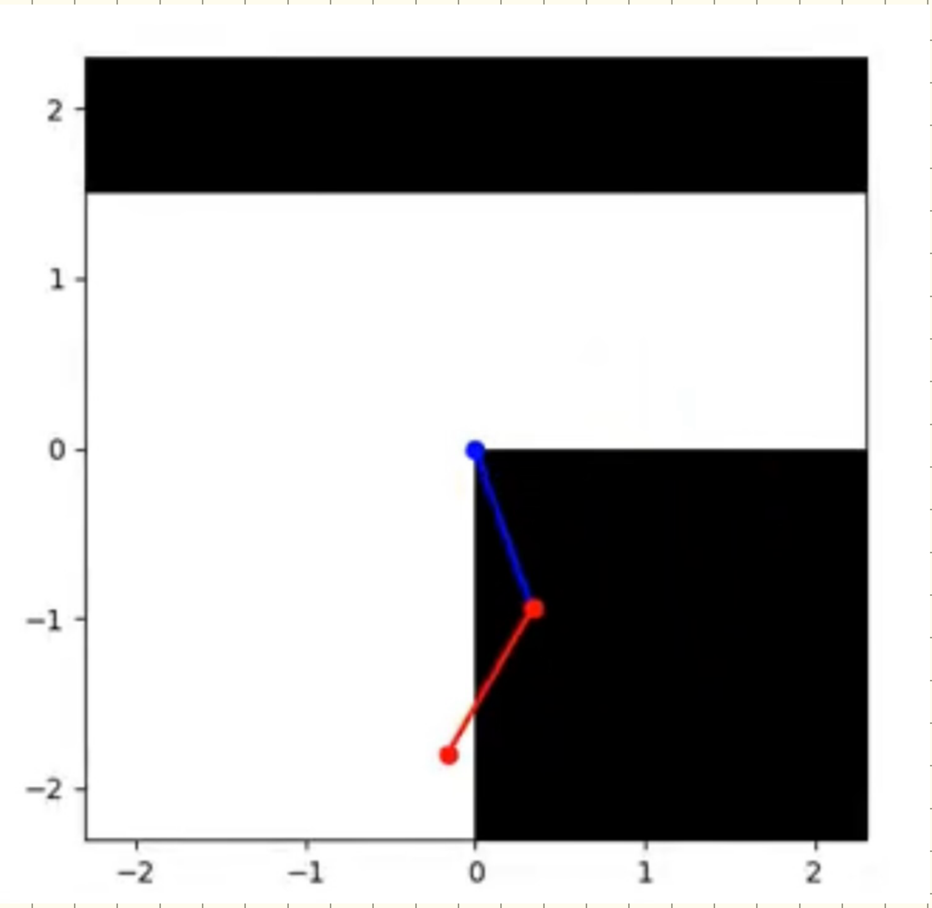
1. Complete the section labeled "Section 1" in *HW4.py* to simulate the <u>complete nonlinear equations of motion</u> of the manipulator with gravity, but without any control torques applied. Assume an initial rest configuration of $\underline{q}_i = [q_{1,i} \;\; q_{2,i}]^T = [30° \; -60°]^T$ and a simulation duration of 2.5 sec.
2. Using the provided Python function `animate_2R_planar_arm_traj`, animate the motion of your planar 2R manipulator to verify that it looks like a double-pendulum falling under its own weight. Note that the manipulator will pass through the 4th quadrant without applied control torques.
3. Using the provided Python function `plot_snapshots`, plot snapshots of the manipulator configurations starting from $t_0 = 0$ and increasing in $\Delta t = 0.1$ sec increments to the final time of $t_f = 2.5$ sec.

<u>Design an Inverse Dynamics Controller and Simulate the Manipulator with Control Torques</u>:

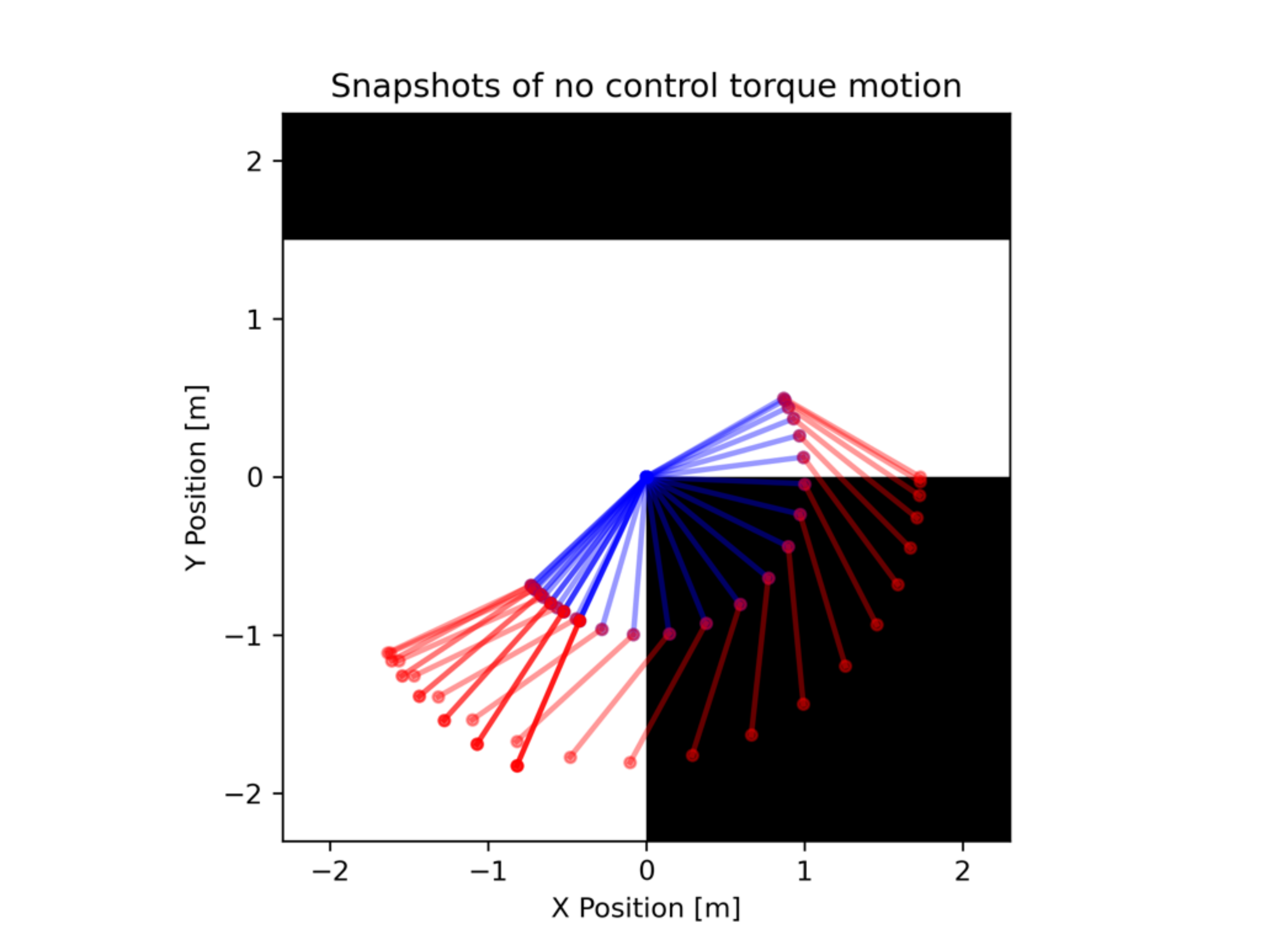
1. Simulate the manipulator under the action of an inverse dynamics controller by completing section labeled "Section 2" in *HW4.py* using tips in this problem statement. Assume an initial rest configuration of $\underline{q}_i = [q_{1,i} \;\; q_{2,i}]^T = [30° \; -60°]^T$, a final desired rest configuration of $\underline{q}_d = [q_{1,d} \;\; q_{2,d}]^T = [240° \; 60°]^T$, and a simulation duration of 2.5 sec.

Passive Dynamics:

1 & 2



3



Report your K_P and K_D matrices. Your controller must satisfy the conditions $|q_{1,d} - q_1| < 2 \text{ deg}$ and $|q_{2,d} - q_2| < 2 \text{ deg}$ over the entire simulated trajectory (the subscript d specifies the desired joint angles).

In this greatly simplified version of inverse dynamics control (also known as computed torque control), you will use an "average" $\hat{B}_{avg}$ generalized mass matrix and neglect centripetal, Coriolis, friction, and gravitational terms. You will further simplify the mass matrix by diagonalizing it in order to treat each joint independently for controller design purposes.

Thus, you will attempt to control the arm by inverting a model of the arm that you know to be inaccurate, but which you hope will be good enough. <u>You will design your controller using an estimated model of the arm dynamics, but simulate the effects of your controller using the full model of the arm dynamics.</u>

$$\boldsymbol{u} = \hat{B}_{\text{avg}}\ddot{\boldsymbol{q}}_d + K_D(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}) + K_P(\boldsymbol{q}_d - \boldsymbol{q})$$

where:

$\hat{B}_{avg} =$ "average" generalized mass matrix (you will create this in *HW4.py*)

$K_D =$ diagonal damping matrix (i.e., diagonal matrix of derivative control gains)

$K_P =$ diagonal stiffness matrix (i.e., diagonal matrix of proportional control gains)

$\underline{q}_d, \underline{\dot{q}}_d, \underline{\ddot{q}}_d =$ desired joint angles, angular vel., and angular accel. from trajectory planner

There is no single correct answer for the gain matrices. There are many viable combinations that will satisfy the design requirements.

2. Using the provided Python function `animate_2R_planar_arm_traj`, animate the motion of your planar 2R manipulator to verify that it follows the desired trajectory. Note that you are applying a controller designed for a simplified system to a manipulator whose motion should abide by its complete nonlinear equations of motion.

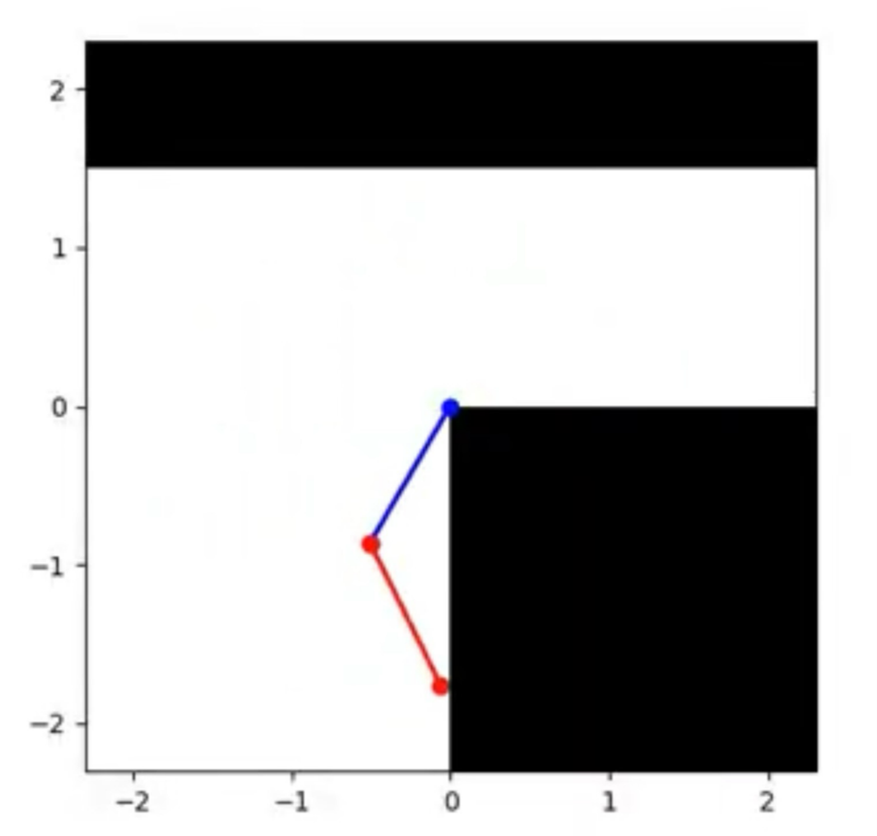
3. Using the provided Python function `plot_snapshots`, plot snapshots of the actual and desired manipulator configurations starting from $t_0 = 0$ and increasing in $\Delta t = 0.1$ sec increments to the final time of $t_f = 2.5$ sec.

4. Plot the time history of both joint position errors in <u>deg</u> in the same figure. Use a solid red line for joint 1 position errors and a solid blue line for joint 2 position errors. Add black horizontal dashed lines at y = -2 deg and y = 2 deg.

5. Plot the joint angles in <u>deg</u> as a function of time. Plot the actual and desired values for both joint coordinates on the same plot. Use a red solid line for q_1, a red dashed line for $q_{1,d}$, a blue solid line for q_2, and a blue dashed line for $q_{2,d}$. Add a vertical dashed line at 1.25 sec (the time at which the manipulator must pass through a via point that has been designed into the pre-planned desired trajectory) and a legend.

Inverse Dynamics Control + Motor Torques

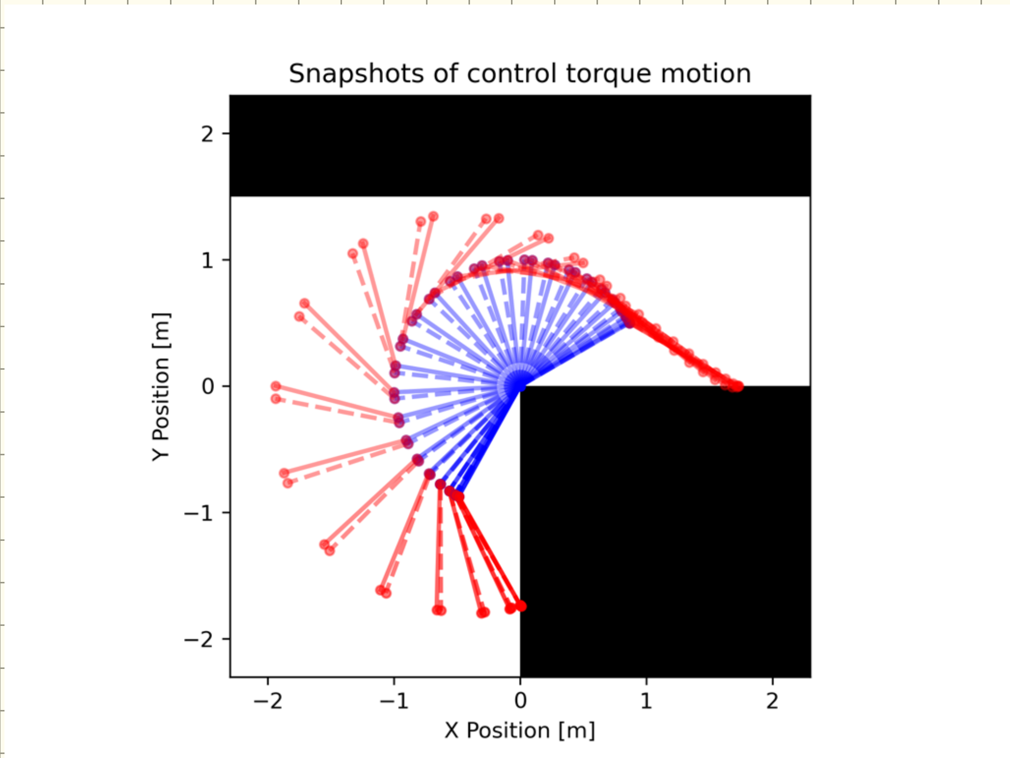
1 & 2



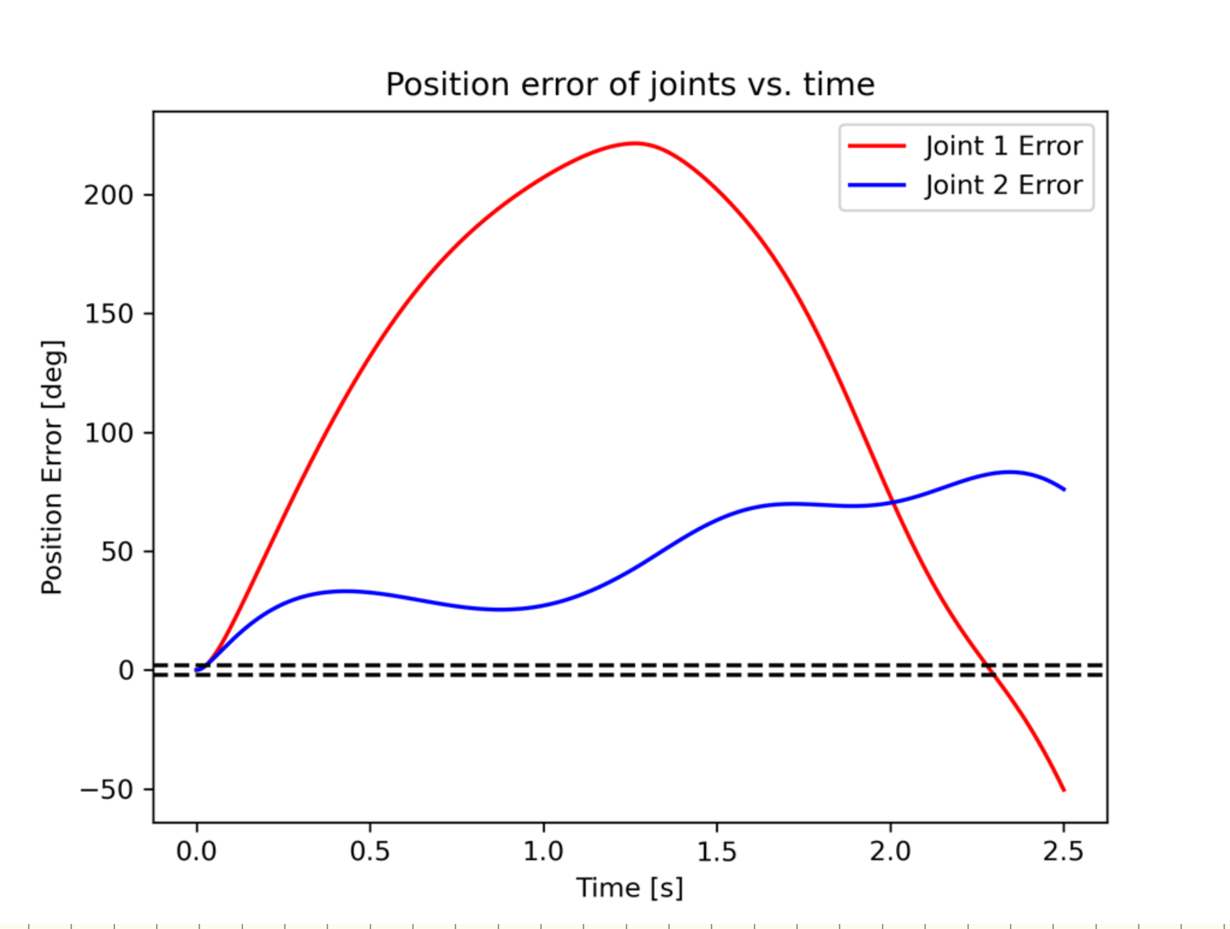
$$K_p = \begin{bmatrix} 1400 & 0 \\ 0 & 1400 \end{bmatrix}$$

$$K_D = \begin{bmatrix} 1200 & 0 \\ 0 & 1200 \end{bmatrix}$$

3

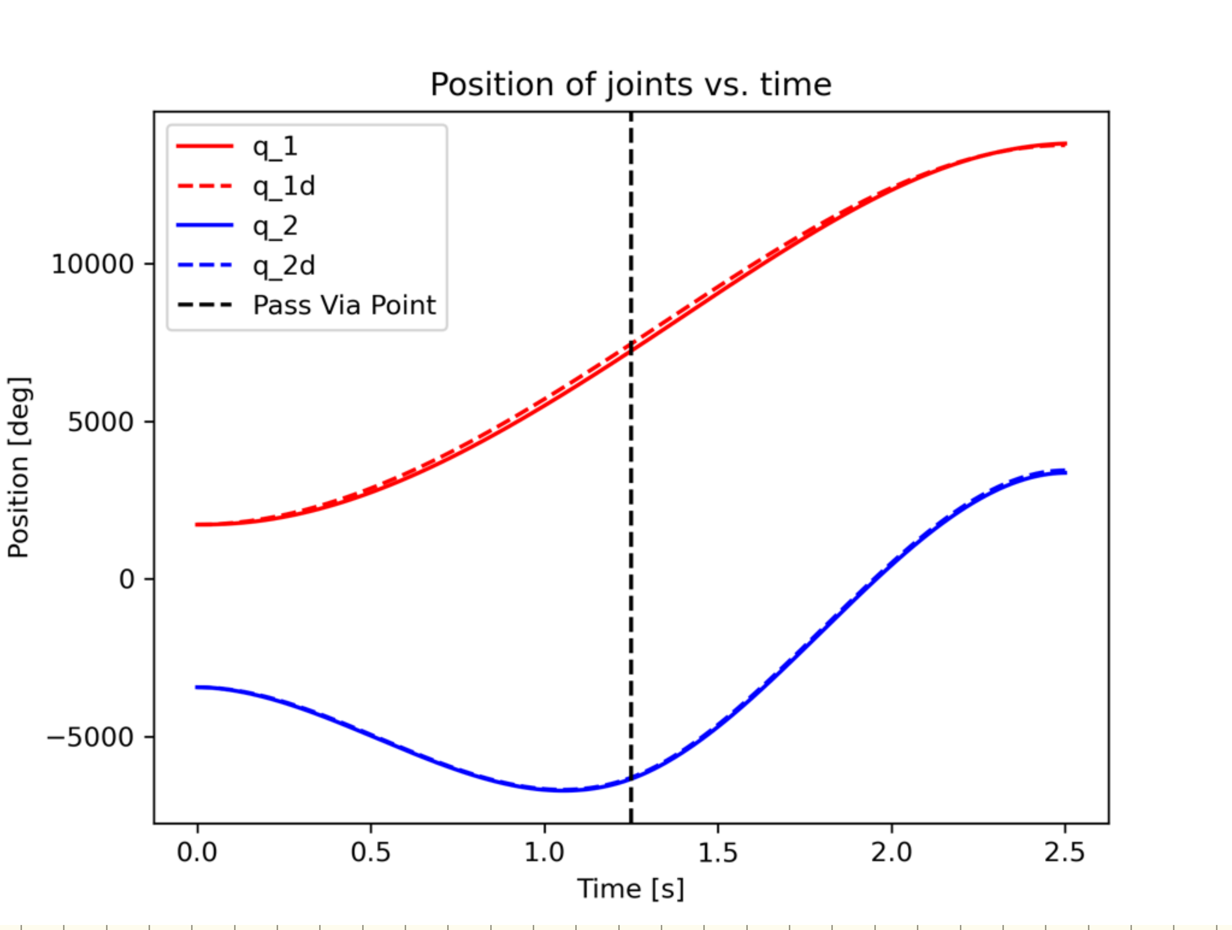


4]



Position error of joints vs. time
Joint 1 Error
Joint 2 Error
Position Error [deg]
200
150
100
50
0
−50
0.0
0.5
1.0
1.5
2.0
2.5
Time [s]

5]



Position of joints vs. time
q_1
q_1d
q_2
q_2d
Pass Via Point
Position [deg]
10000
5000
0
−5000
0.0
0.5
1.0
1.5
2.0
2.5
Time [s]

6. Plot the joint angle velocities in <u>deg/s</u> as a function of time. Plot the actual and desired values for both joint angle velocities on the same plot. Use a red solid line for $\dot{q}_1$, a red dashed line for $\dot{q}_{1,d}$, a blue solid line for $\dot{q}_2$, and a blue dashed line for $\dot{q}_{2,d}$. Add a vertical dashed line at 1.25 seconds and a legend.

7. Plot the control torques in N·m as a function of time. Use a red solid line for τ_1 and a blue solid line for τ_2. Add a vertical dashed line at 1.25 sec and a legend.

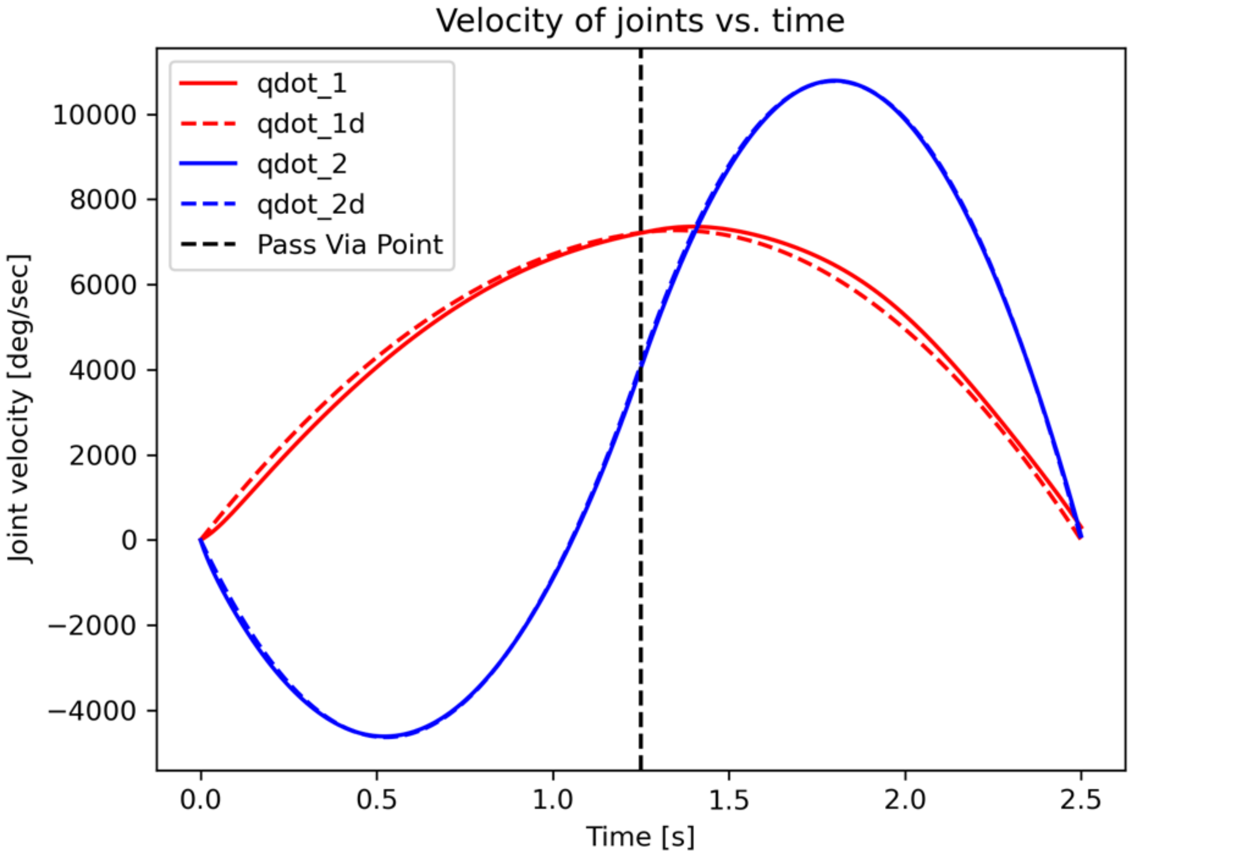
<u>Summary of deliverables:</u>

Your submission should include:

- Plots of animation snapshots
- Your K_P and K_D gain matrices
- Labeled time history plots
- A plot of your final Drake block diagram
- Your **completed** *HW4.py* file converted to a PDF (To facilitate grading, see the relevant PyCharm help page for how to print a *.py* file to a PDF).

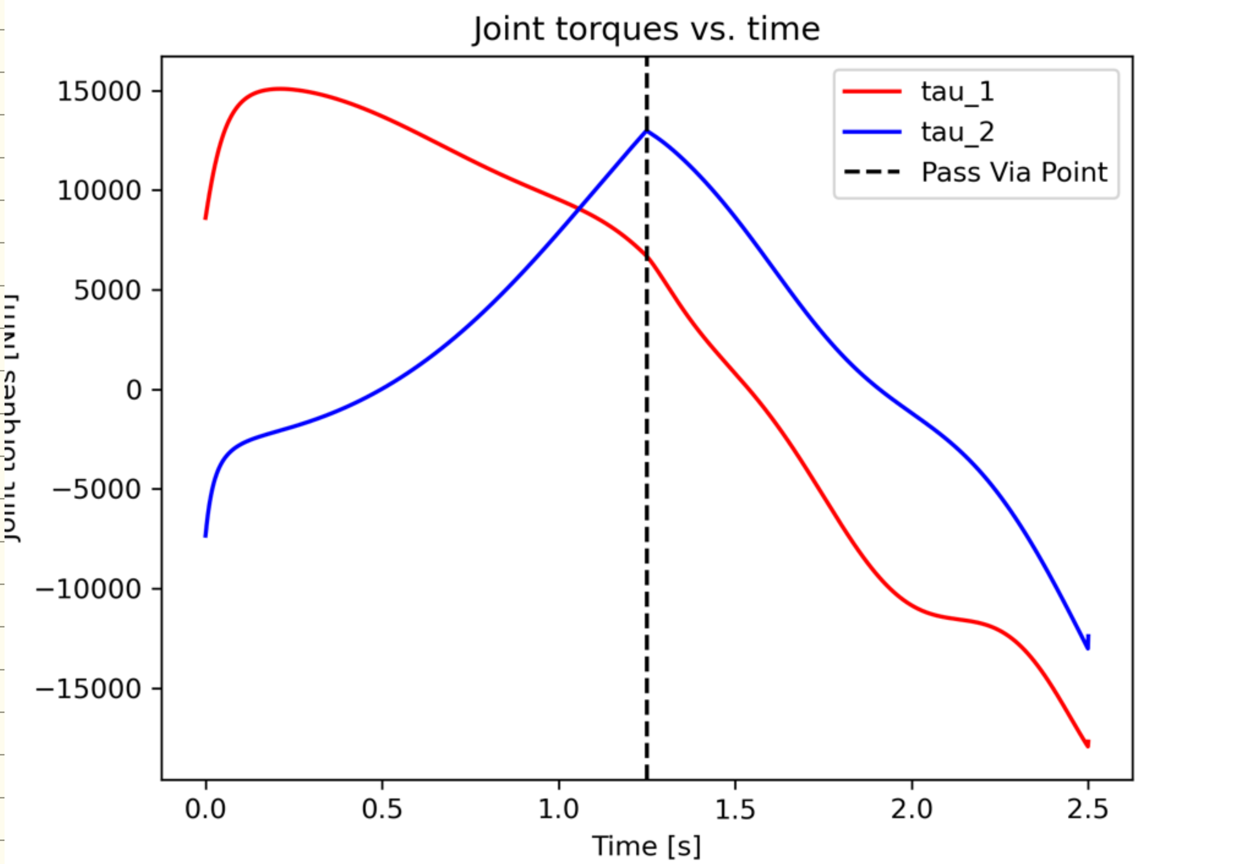
<u>NOTE:</u> Each student must submit their own independent work. **For full credit, you must submit to Gradescope all <u>custom</u> Python code** (e.g. *HW4.py*) **and requested plots with labels.** You may save this content to PDF or take screenshots for electronic submission via Gradescope. Files of the *.py* and *.toml* format cannot be directly uploaded to Gradescope and should <u>not</u> be e-mailed to instructors for grading. The more intermediate results and comments you provide, the greater the opportunity for partial credit.

6



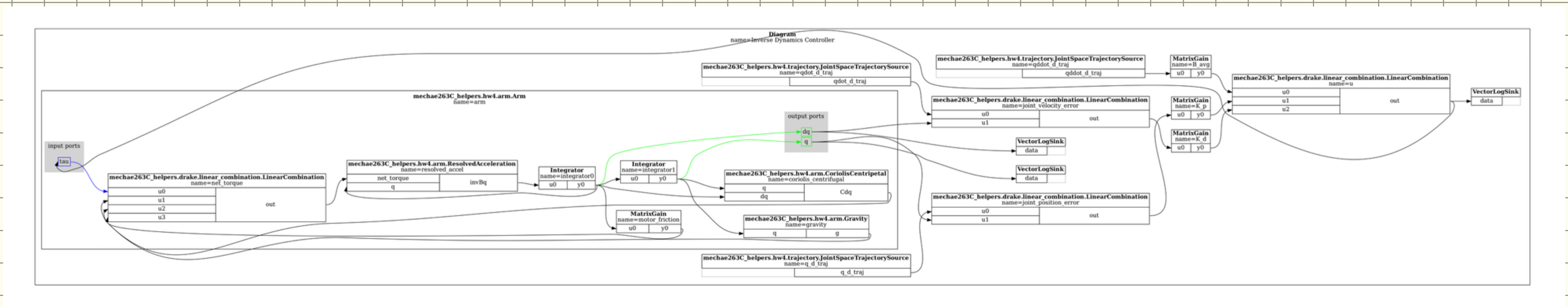
Velocity of joints vs. time
qdot_1
qdot_1d
qdot_2
qdot_2d
Pass Via Point
Joint velocity [deg/sec]
Time [s]
10000
8000
6000
4000
2000
0
−2000
−4000
0.0
0.5
1.0
1.5
2.0
2.5

7



Joint torques vs. time
tau_1
tau_2
Pass Via Point
Time [s]
15000
10000
5000
0
−5000
−10000
−15000
0.0
0.5
1.0
1.5
2.0
2.5

Overall Block Diagram:



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
```

```
    # `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:
```

```python
        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`
```

```
    #         (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')

```