

mechae263C_homework4.py

```

1  """
2  IMPORTANT NOTE:
3      The instructions for completing this template are inline with the code. You can
4      find them by searching for: "TODO:"
5  """
6
7  import matplotlib.pyplot as plt
8  import numpy as np
9  from numpy.typing import NDArray
10 from pydrake.systems.analysis import Simulator
11 from pydrake.systems.framework import DiagramBuilder, Diagram, Context
12 from pydrake.systems.primitives import MatrixGain, LogVectorOutput
13
14 from mechae263C_helpers.drake import LinearCombination, plot_diagram
15 from mechae263C_helpers.hw4.arm import Arm
16 from mechae263C_helpers.hw4.kinematics import calc_fk_2D
17 from mechae263C_helpers.hw4.trajectory import (
18     eval_cubic_spline_traj,
19     JointSpaceTrajectorySource,
20 )
21 from mechae263C_helpers.hw4.plotting import animate_2R_planar_arm_traj, plot_snapshots
22
23
24 def run_simulation(
25     q_initial: NDArray[np.double],
26     q_final: NDArray[np.double],
27     B_avg: NDArray[np.double],
28     K_p: NDArray[np.double],
29     K_d: NDArray[np.double],
30     simulation_duration_s: float,
31     should_apply_control_torques: bool,
32     control_period_s: float = 1e-3,
33 ) -> tuple[
34     NDArray[np.double],
35     tuple[NDArray[np.double], NDArray[np.double]],
36     tuple[NDArray[np.double], NDArray[np.double]],
37     NDArray[np.double],
38     Diagram,
39 ]:
40     """
41     Runs a simulation with a desired joint position
42
43     Parameters
44     -----
45     q_initial:
46         A numpy array of shape (2,) containing the initial joint positions
47
48     q_final:

```

```

49     A numpy array of shape (2,) containing the final desired joint positions
50
51     B_avg:
52     A numpy array of shape (2, 2) containing the average linearized inertia matrix
53
54     K_p:
55     A numpy array of shape (2, 2) containing the proportional gains of the inverse
56     dynamics controller.
57
58     K_d:
59     A numpy array of shape (2, 2) containing the derivative gains of the inverse
60     dynamics controller.
61
62     control_period_s:
63     The period between control commands in units of seconds
64
65     simulation_duration_s:
66     The duration of the simulation in units of seconds
67
68     should_apply_control_torques:
69     A bool that specifies that control torques should be simulated when set to
70     `True`. (If set to `False` then no control torques are simulated).
71
72     Returns
73     -----
74     A tuple with five elements:
75     1. A numpy array with shape (T,) of simulation time steps
76     2. A tuple of numpy arrays both with shape (2, T) of desired and actual joint
77         positions corresponding to each simulation time step, respectively.
78     3. A tuple of numpy arrays both with shape (2, T) of desired and actual joint
79         velocities corresponding to each simulation time step, respectively.
80     4. A numpy array with shape (2, T) of applied control torques corresponding to
81         each simulation time step
82     5. A Drake diagram
83     """
84     # -----
85     # Add "systems" to a `DiagramBuilder` object.
86     # - "systems" are the blocks in a block diagram
87     # - Some examples for how to add named systems to a `DiagramBuilder` are given
88     #   below
89     # -----
90     builder = DiagramBuilder()
91
92     # Create the desired joint angle, velocity, and acceleration trajectories
93     dt = control_period_s
94     times = np.arange(0, simulation_duration_s + dt, dt)
95     waypoint_times = np.asarray([0, simulation_duration_s / 2, simulation_duration_s])
96     waypoints = np.stack([q_initial, np.deg2rad([130, -110]), q_final], axis=1)
97
98     q_d_traj, qdot_d_traj, qddot_d_traj = eval_cubic_spline_traj(

```

```
99         times=times, waypoint_times=waypoint_times, waypoints=waypoints
100     )
101     q_traj = builder.AddNamedSystem(
102         "q_d_traj",
103         JointSpaceTrajectorySource(
104             name="q_d_traj",
105             num_joints=q_d_traj.shape[0],
106             times=times,
107             joint_coordinates=q_d_traj,
108         ),
109     )
110     qdot_traj = builder.AddNamedSystem(
111         "qdot_d_traj",
112         JointSpaceTrajectorySource(
113             name="qdot_d_traj",
114             num_joints=qdot_d_traj.shape[0],
115             times=times,
116             joint_coordinates=qdot_d_traj,
117         ),
118     )
119     if should_apply_control_torques:
120         qddot_traj = builder.AddNamedSystem(
121             "qddot_d_traj",
122             JointSpaceTrajectorySource(
123                 name="qddot_d_traj",
124                 num_joints=qddot_d_traj.shape[0],
125                 times=times,
126                 joint_coordinates=qddot_d_traj,
127             ),
128         )
129
130         K_p_gain = builder.AddNamedSystem(
131             "K_p", MatrixGain(np.asarray(K_p, dtype=np.double))
132         )
133         K_d_gain = builder.AddNamedSystem(
134             "K_d", MatrixGain(np.asarray(K_d, dtype=np.double))
135         )
136
137     joint_position_error = builder.AddNamedSystem(
138         "joint_position_error",
139         LinearCombination(input_coeffs=(1, -1), input_shapes=(2,)),
140     )
141     joint_velocity_error = builder.AddNamedSystem(
142         "joint_velocity_error",
143         LinearCombination(input_coeffs=(1, -1), input_shapes=(2,)),
144     )
145     arm = builder.AddNamedSystem("arm", Arm())
146
147     if should_apply_control_torques:
148         control_torque = builder.AddNamedSystem(
```

```

149         "u",
150         LinearCombination(input_coeffs=(1, 1, 1), input_shapes=(2,))
151     )
152     inertia_matrix = builder.AddNamedSystem("B_avg", MatrixGain(B_avg))
153
154     # -----
155     # Connect the systems in the `DiagramBuilder` (i.e. add arrows of block diagram)
156     # -----
157     # `builder.ExportInput(input_port)` makes the provided "input_port" into an input
158     # of the entire diagram
159     # The functions system.get_input_port() returns the input port of the given system
160     # - If there is more than one input port, you must specify the index of the
161     #   desired input
162     # The functions system.get_output_port() returns the output port of the given system
163     # - If there is more than one output port, you must specify the index of the
164     #   desired output
165     builder.Connect(q_traj.get_output_port(), joint_position_error.get_input_port(0))
166     builder.Connect(qdot_traj.get_output_port(), joint_velocity_error.get_input_port(0))
167
168     if should_apply_control_torques:
169         builder.Connect(qddot_traj.get_output_port(), inertia_matrix.get_input_port())
170
171     # Declaring output of the system
172     joint_velocity_output = arm.get_output_port(0)
173     joint_position_output = arm.get_output_port(1)
174
175     # TODO:
176     # Replace any `...` below with the correct system and values. Please keep the
177     # system names the same
178     builder.Connect(joint_position_output, joint_position_error.get_input_port(1))
179     builder.Connect(joint_velocity_output, joint_velocity_error.get_input_port(1))
180     if should_apply_control_torques:
181         builder.Connect(joint_position_error.get_output_port(), K_p_gain.get_input_port())
182         builder.Connect(joint_velocity_error.get_output_port(), K_d_gain.get_input_port())
183
184         #
185         builder.Connect(
186             inertia_matrix.get_output_port(), control_torque.get_input_port(0)
187         )
188         builder.Connect(K_p_gain.get_output_port(), control_torque.get_input_port(1))
189         builder.Connect(K_d_gain.get_output_port(), control_torque.get_input_port(2))
190         builder.Connect(control_torque.get_output_port(), arm.get_input_port())
191     else:
192         builder.ExportInput(arm.get_input_port(), name="control_torque")
193
194     # -----
195     # Log joint positions
196     # -----
197     # These systems are special in Drake. They periodically save the output port value
198     # a during a simulation so that it can be accessed later. The value is saved every

```

```

199     # `publish_period` seconds in simulation time.
200     joint_position_logger = LogVectorOutput(
201         arm.get_output_port(1), builder, publish_period=control_period_s
202     )
203     joint_velocity_logger = LogVectorOutput(
204         arm.get_output_port(0), builder, publish_period=control_period_s
205     )
206     if should_apply_control_torques:
207         control_torque_logger = LogVectorOutput(
208             control_torque.get_output_port(), builder, publish_period=control_period_s
209         )
210
211     # -----
212     # Setup/Run the simulation
213     # -----
214     # This line builds a `Diagram` object and uses it to make a `Simulator` object for
215     # the diagram
216     diagram: Diagram = builder.Build()
217     diagram.set_name("Inverse Dynamics Controller")
218     simulator: Simulator = Simulator(diagram)
219
220     # Get the context (this contains all the information needed to run the simulation)
221     context: Context = simulator.get_mutable_context()
222
223     # Set initial conditions
224     initial_conditions = context.get_mutable_continuous_state_vector()
225     initial_conditions.SetAtIndex(2, q_initial[0])
226     initial_conditions.SetAtIndex(3, q_initial[1])
227
228     if not should_apply_control_torques:
229         diagram.get_input_port().FixValue(context, np.zeros((2,)))
230
231     # Advance the simulation by `simulation_duration_s` seconds using the
232     # `simulator.AdvanceTo()` function
233     simulator.AdvanceTo(simulation_duration_s)
234
235     # -----
236     # Extract simulation outputs
237     # -----
238     # The lines below extract the joint position log from the simulator context
239     joint_position_log = joint_position_logger.FindLog(simulator.get_context())
240     t = joint_position_log.sample_times()
241     q_actual = joint_position_log.data()
242
243     joint_velocity_log = joint_velocity_logger.FindLog(simulator.get_context())
244     qdot_actual = joint_velocity_log.data()
245
246     control_torques = np.zeros((2, len(t)), dtype=np.double)
247
248     if should_apply_control_torques:

```

```

249     control_torque_log = control_torque_logger.FindLog(simulator.get_context())
250     control_torques = control_torque_log.data()
251
252     # Return a `tuple` of required results
253     return t, (q_d_traj, q_actual), (qdot_d_traj, qdot_actual), control_torques, diagram
254
255
256 if __name__ == "__main__":
257     #####
258     # Section 1
259     #####
260     # -----
261     # TODO:
262     #   Replace `...` with the correct values for each parameter
263     # -----
264     # The below functions might be helpful:
265     #   np.diag: https://numpy.org/doc/stable/reference/generated/numpy.diag.html
266     #   np.eye: https://numpy.org/doc/stable/reference/generated/numpy.eye.html
267     a_1 = a_2 = 1
268     l_1 = l_2 = 0.5
269     m_l1 = m_l2 = 9
270     I_l1 = I_l2 = 3
271     m_m1 = m_m2 = 1
272     I_m1 = I_m2 = 0.007
273     k_r1 = k_r2 = 50
274
275     K_p = np.diag([1400, 1400])
276     K_d = np.diag([1200, 1200])
277
278     # an_norm = lambda an:an%360.0
279
280     # -----
281     # TODO:
282     #   Replace `...` with the correct values for the diagonal terms of the "averaged"
283     #   generalized inertia matrix. These terms can be found by taking the small angle
284     #   approximation of the elements in the full generalized inertia matrix given in
285     #   the problem statement.
286     # -----
287     B_avg = np.zeros((2, 2))
288     B_avg[0, 0] = (I_l1) + (m_l1 * l_1**2) + (I_m1 * k_r1**2) + (I_l2) + (m_l2*(a_1**2 +
289 l_2**2 + 2*a_1*l_2))
290     B_avg[1, 1] = (I_l2) + (m_l2 * l_2**2) + (I_m2 * k_r2**2)
291
292     # -----
293     # TODO:
294     #   Replace `...` with the initial and final joint configurations specified in the
295     #   problem statement.
296     # -----
297     q_initial = np.deg2rad([30, -60])
298     q_final = np.deg2rad([240, 60])

```

```

298
299 # -----
300 # Simulate without control torques
301 # TODO:
302 #   Replace `...` with the correct values to simulate the un-actuated dynamics of
303 #   the planar 2R manipulator.
304 # -----
305 t, (q_d, q), (qd_tra, qd_act), ct, diagram = run_simulation(
306     q_initial=q_initial,
307     q_final=q_final,
308     B_avg=B_avg,
309     K_p=K_p,
310     K_d=K_d,
311     simulation_duration_s=2.5,
312     should_apply_control_torques=False, # True or False?
313 )
314 fig, ax = plot_diagram(diagram)
315 fig.savefig("Part1_figs/no_control_torque_diagram.png", dpi=300)
316 # Convert `q` and `q_d` to degrees
317 q_d = (np.rad2deg(q_d))
318 q = (np.rad2deg(q))
319 print('Finish part 1 simulation')
320
321 # -----
322 # TODO:
323 #   Using the link lengths `[a_1, a_2]`, the simulated joint positions `q`, and the
324 #   `calc_fk_2D` function to calculate the xy positions of each joint of the
325 #   manipulator for the simulated scenario. (Replace `...` with the correct values)
326 #
327 #   Hint: Make sure to convert `q` back to radians before using it with `calc_fk_2D`
328 #         (using np.deg2rad).
329 #
330 # -----
331 joint_xs, joint_ys = calc_fk_2D(link_lens=[a_1, a_2], joint_positions=np.deg2rad(q))
332
333 # -----
334 # TODO:
335 #   Replace all `...` in the call of the `animate_2R_planar_arm_traj` function with
336 #   the correct output of the `calc_fk_2D`.
337 # -----
338 print('Saving part 1 plots...')
339 _, _, anim_no_control_torques = animate_2R_planar_arm_traj(
340     joint_xs=joint_xs,
341     joint_ys=joint_ys,
342     animation_file_name="no_control_torques_animation"
343 )
344 # anim_no_control_torques.save('Part1_figs/no_control_torques_animation.mp4')
345
346 # -----
347 # Plot Snapshots

```

```

348 # TODO:
349 # Replace all `...` in the call of the `plot_snapshots` function with
350 # the correct output of the `calc_fk_2D` and the dt specified in the problem
351 # statement.
352 # Add code to properly label `ax` and save `fig`
353 # -----
354 fig, ax = plot_snapshots(dt=0.1, joint_xs=joint_xs, joint_ys=joint_ys)
355 ax.set_xlabel('X Position [m]')
356 ax.set_ylabel('Y Position [m]')
357 ax.set_title('Snapshots of no control torque motion')
358 fig.savefig('Part1_figs/Part1_Snapshots.png', dpi=300)
359 print('Saved part 1 plots')
360
361 #####
362 # Section 2
363 #####
364 # -----
365 # Simulate with control torques
366 # TODO:
367 # Replace `...` with the correct values to simulate the dynamics of
368 # the planar 2R manipulator under your inverse dynamics controller.
369 # -----
370 t, (q_d, q), (qdot_d, qdot), control_torques, diagram = run_simulation(
371     q_initial=q_initial,
372     q_final=q_final,
373     B_avg=B_avg,
374     K_p=K_p,
375     K_d=K_d,
376     simulation_duration_s=2.5,
377     should_apply_control_torques=True, # True or False?
378 )
379 fig, ax = plot_diagram(diagram)
380 fig.savefig("Part2_figs/control_torque_diagram.png", dpi=300)
381 print('Finish part 2 simulation')
382
383 # Convert `q`, `q_d`, `qdot`, and `qdot_d` to degrees
384 q_d = (np.rad2deg(q_d))
385 q = (np.rad2deg(q))
386 qdot_d = np.rad2deg(qdot_d)
387 qdot = np.rad2deg(qdot)
388
389 # -----
390 # Animate Trajectory
391 # TODO:
392 # Using the link lengths `[a_1, a_2]`, the actual joint positions `q` and desired
393 # joint positions `q_d`, with the `calc_fk_2D` function to calculate the xy
394 # positions of each joint of the manipulator for the actual and desired
395 # trajectories, respectively. (Replace `...` with the correct values)
396 #
397 # Hint: Make sure to convert `q` back to radians before using it with `calc_fk_2D`

```



```

398     #         (using np.deg2rad).
399     #
400     # -----
401     joint_xs, joint_ys = calc_fk_2D(link_lens=[a_1, a_2], joint_positions=np.deg2rad(q))
402     joint_xs_desired, joint_ys_desired = calc_fk_2D(link_lens=[a_1, a_2],
joint_positions=np.deg2rad(q_d))
403
404     # -----
405     # TODO:
406     # Replace all `...` in the call of the `animate_2R_planar_arm_traj` function with
407     # the correct output of the `calc_fk_2D`.
408     # -----
409     print('Saving part 2 plots...')
410     _, _, anim_control_torques = animate_2R_planar_arm_traj(
411         joint_xs=joint_xs,
412         joint_ys=joint_ys,
413         animation_file_name="control_torques_animation"
414     )
415     # anim_control_torques.save('Part2_figs/control_torques_animation', 'Pillow', 20)
416
417     # -----
418     # Plot Snapshots
419     # TODO:
420     # Replace all `...` in the call of the `plot_snapshots` function with
421     # the correct output of the `calc_fk_2D` and the dt specified in the problem
422     # statement.
423     # Add code to properly label `ax` and save `fig`
424     # -----
425     fig, ax = plot_snapshots(
426         dt=0.1,
427         joint_xs=joint_xs,
428         joint_ys=joint_ys,
429         joint_xs_desired=joint_xs_desired,
430         joint_ys_desired=joint_ys_desired,
431     )
432
433     ax.set_xlabel('X Position [m]')
434     ax.set_ylabel('Y Position [m]')
435     ax.set_title('Snapshots of control torque motion')
436     fig.savefig('Part2_figs/Part2_Snapshots.png', dpi=300)
437     print('Saved part 2 plots...')
438     # -----
439     # Plot Joint Position Error
440     # TODO:
441     # Replace `...` with the code to make the specified joint position error plot for
442     # the inverse dynamics controller case.
443     #
444     # Hints:
445     # 1. To plot a black dashed vertical line at `x = x0` use the `ax.axvline` function:
446     #     `ax.axvline(x0, ls="--", color="black")

```

```

447 # 2. When plotting, use the `label` argument to automatically add a legend item:
448 #     `ax.plot(x, y, color="red", label=r"$\theta_1$ Error")`
449 # 3. You need to call `ax.legend()` to actually plot the legend.
450 # -----
451 fig = plt.figure()
452 ax = fig.add_subplot(1, 1, 1)
453 # joint 1 error
454 ax.plot(t, np.rad2deg(q_d[0] - q[0]), color='red', label='Joint 1 Error')
455 # joint 2 error
456 ax.plot(t, np.rad2deg(q_d[1] - q[1]), color='blue', label='Joint 2 Error')
457 ax.axhline(-2, ls="--", color="black")
458 ax.axhline(2, ls="--", color="black")
459 ax.set_xlabel('Time [s]')
460 ax.set_ylabel('Position Error [deg]')
461 ax.set_title('Position error of joints vs. time')
462 ax.legend()
463 fig.savefig('JointPlots/JointError.png', dpi=300)
464 plt.clf()
465 print('Saved Joint Error')
466
467 # -----
468 # Plot Joint Positions
469 # TODO:
470 #   Replace `...` with the code to make the specified joint position plot for the
471 #   inverse dynamics controller case.
472 # -----
473 fig = plt.figure()
474 ax = fig.add_subplot(1, 1, 1)
475 # joint 1 position
476 ax.plot(t, np.rad2deg(q[0]), color='red', label='q_1')
477 ax.plot(t, np.rad2deg(q_d[0]), color='red', ls="--", label='q_1d')
478 # joint 2 position
479 ax.plot(t, np.rad2deg(q[1]), color='blue', label='q_2')
480 ax.plot(t, np.rad2deg(q_d[1]), color='blue', ls="--", label='q_2d')
481 ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
482 ax.set_xlabel('Time [s]')
483 ax.set_ylabel('Position [deg]')
484 ax.set_title('Position of joints vs. time')
485 ax.legend()
486 fig.savefig('JointPlots/JointPosition.png', dpi=300)
487 plt.clf()
488 print('Saved Joint Position')
489
490 # -----
491 # Plot Joint Velocities
492 # TODO:
493 #   Replace `...` with the code to make the specified joint velocity plot for the
494 #   inverse dynamics controller case.
495 # -----
496 fig = plt.figure()

```

```

497     ax = fig.add_subplot(1, 1, 1)
498     # joint 1 velocity
499     ax.plot(t, np.rad2deg(qdot[0]), color='red', label='qdot_1')
500     ax.plot(t, np.rad2deg(qdot_d[0]), color='red', ls="--", label='qdot_1d')
501     # joint 2 velocity
502     ax.plot(t, np.rad2deg(qdot[1]), color='blue', label='qdot_2')
503     ax.plot(t, np.rad2deg(qdot_d[1]), color='blue', ls="--", label='qdot_2d')
504     ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
505     ax.set_xlabel('Time [s]')
506     ax.set_ylabel('Joint velocity [deg/sec]')
507     ax.set_title('Velocity of joints vs. time')
508     ax.legend()
509     fig.savefig('JointPlots/JointVelocity.png', dpi=300)
510     plt.clf()
511     print('Saved Joint Velocities')
512
513     # -----
514     # Plot Control Torques
515     # TODO:
516     #   Replace `...` with the code to make the specified control torque plot for the
517     #   inverse dynamics controller case.
518     # -----
519     fig = plt.figure()
520     ax = fig.add_subplot(1, 1, 1)
521     # joint 1 torques
522     ax.plot(t, np.rad2deg(control_torques[0]), color='red', label='tau_1')
523     # joint 2 torques
524     ax.plot(t, np.rad2deg(control_torques[1]), color='blue', label='tau_2')
525     ax.axvline(1.25, ls="--", color="black", label='Pass Via Point')
526     ax.set_xlabel('Time [s]')
527     ax.set_ylabel('Joint torques [Nm]')
528     ax.set_title('Joint torques vs. time')
529     ax.legend()
530     fig.savefig('JointPlots/JointTorques.png', dpi=300)
531     plt.clf()
532     print('Saved Joint Torques')
533

```