

mechae263C_homework5_problem1.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  from __future__ import annotations
8
9  import math
10
11 import matplotlib.pyplot as plt
12 import numpy as np
13 from numpy.typing import NDArray
14 from pydrake.systems.analysis import Simulator
15 from pydrake.systems.framework import (
16     Context,
17     Diagram,
18     DiagramBuilder,
19     InputPort,
20 )
21 from pydrake.systems.primitives import (
22     MatrixGain,
23     PassThrough,
24     ZeroOrderHold,
25     LogVectorOutput,
26     ConstantVectorSource,
27 )
28
29 from mechae263C_helpers.drake import LinearCombination, plot_diagram
30 from mechae263C_helpers.hw5 import validate_np_array
31 from mechae263C_helpers.hw5.arm import Arm, Gravity
32 from mechae263C_helpers.hw5.jacobian_gains import (
33     AnalyticalJacobianTransposeGain,
34     AnalyticalJacobianGain,
35 )
36 from mechae263C_helpers.hw5.kinematics import calc_2R_planar_inverse_kinematics
37 from mechae263C_helpers.hw5.op_space import DirectKinematics
38
39
40 def calc_analytical_jacobian(
41     q1: float, q2: float, a1: float, a2: float
42 ) -> NDArray[np.double]:
43     """
44     Calculates the Analytical Jacobian of a 2R planar manipulator
45
46     Parameters
47     -----
48     q1

```

```

49     A float representing the first joint angle
50     q2
51     A float representing the second joint angle
52     a1
53     A float representing the first link length
54     a2
55     A float representing the second link length
56
57     Returns
58     -----
59     A numpy array of shape (2, 2) representing the Analytical Jacobian of the 2R
60     planar manipulator
61     """
62     J_A = np.zeros(shape=(2, 2), dtype=np.double)
63
64     # =====
65     # TODO: Calculate Analytical Jacobian (J_A)
66     # Fill in the provided numpy array `J_A` with the Analytical Jacobian of the
67     # manipulator
68     # -----
69     J_A[0, 0] = (-a1 * np.sin(q1)) - (a2 * np.sin(q1 + q2))
70     J_A[1, 0] = (a1 * np.cos(q1)) + (a2 * np.cos(q1 + q2))
71     J_A[0, 1] = (-a2 * np.sin(q1 + q2))
72     J_A[1, 1] = (a2 * np.cos(q1 + q2))
73     # =====
74
75     return J_A
76
77
78 def calc_direct_kinematics(
79     q1: float, q2: float, a1: float, a2: float
80 ) -> NDArray[np.double]:
81     """
82     Calculates the direct (a.k.a. forward) kinematics of a 2R planar manipulator
83
84     Parameters
85     -----
86     q1
87         A float representing the first joint angle
88     q2
89         A float representing the second joint angle
90     a1
91         A float representing the first link length
92     a2
93         A float representing the second link length
94
95     Returns
96     -----
97     A numpy array of shape (2,) representing the xy position of the 2R planar
98     manipulator's end effector

```

```

99     """
100     x_e = np.zeros(shape=(2,), dtype=np.double)
101
102     # =====
103     # TODO: Calculate Direct Kinematics
104     #   Fill in the provided numpy array `x_e` with the x and y positions of the
105     #   end-effector using the direct kinematics of a 2R planar manipulator.
106     # -----
107     x_e[0] = (a1 * np.cos(q1)) + (a2 * np.cos(q1 + q2))
108     x_e[1] = (a1 * np.sin(q1)) + (a2 * np.sin(q1 + q2))
109     # =====
110
111     return x_e
112
113
114 class OperationalSpacePDControllerWithGravityCompensation(Diagram):
115     def __init__(
116         self,
117         link_lens: tuple[float, float],
118         K_P: NDArray[np.double],
119         K_D: NDArray[np.double],
120         control_sample_period_s: float,
121         p_desired: NDArray[np.double],
122     ):
123         super().__init__()
124         self.control_sample_period_s = max(1e-10, abs(control_sample_period_s))
125         self.link_lens = tuple(float(a) for a in link_lens)
126         self.num_dofs = len(link_lens)
127         assert self.num_dofs == 2
128
129         validate_np_array(arr=K_P, arr_name="K_P", correct_shape=(2, 2))
130         validate_np_array(arr=K_D, arr_name="K_D", correct_shape=(2, 2))
131         validate_np_array(arr=p_desired, arr_name="p_desired", correct_shape=(2,))
132
133         self.K_P = K_P
134         self.K_D = K_D
135
136         builder = DiagramBuilder()
137
138         proportional_gain: MatrixGain = builder.AddNamedSystem("K_P", MatrixGain(K_P))
139         derivative_gain: MatrixGain = builder.AddNamedSystem("K_D", MatrixGain(K_D))
140         gravity_torques: Gravity = builder.AddNamedSystem(
141             "gravity", Gravity(Arm().dyn_params)
142         )
143         JA_gain: AnalyticalJacobianGain = builder.AddNamedSystem(
144             "J_A",
145             AnalyticalJacobianGain(self.link_lens, calc_analytical_jacobian),
146         )
147         JA_T_gain: AnalyticalJacobianTransposeGain = builder.AddNamedSystem(
148             "J_A.T",

```

```

149     AnalyticalJacobianTransposeGain(self.link_lens, calc_analytical_jacobian),
150 )
151 direct_kinematics: DirectKinematics = builder.AddNamedSystem(
152     "k(q)", DirectKinematics(self.link_lens, calc_direct_kinematics)
153 )
154 control_torques: LinearCombination = builder.AddNamedSystem(
155     "u", LinearCombination(input_coeffs=(1, 1), input_shapes=(2,))
156 )
157 operational_space_position_error: LinearCombination = builder.AddNamedSystem(
158     "x_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
159 )
160 operational_space_control_action: LinearCombination = builder.AddNamedSystem(
161     "f_c", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
162 )
163
164 q = builder.AddNamedSystem("q", PassThrough(vector_size=self.num_dofs))
165 qdot = builder.AddNamedSystem("qdot", PassThrough(vector_size=self.num_dofs))
166 zoh = builder.AddNamedSystem(
167     "sampled_u",
168     ZeroOrderHold(
169         period_sec=self.control_sample_period_s, vector_size=self.num_dofs
170     ),
171 )
172 p_desired_source = builder.AddNamedSystem(
173     "p_desired", ConstantVectorSource(source_value=p_desired)
174 )
175
176 # =====
177 # TODO: Complete Controller Block Diagram
178 # Replace `...` below with the correct output or input port.
179 # Note that following convenience method is available to access the f_c input
180 # port of the `JA_T_gain` system/block
181 #     JA_T_gain.get_f_c_input_port()
182 # -----
183 builder.Connect(
184     operational_space_position_error.get_output_port(),
185     proportional_gain.get_input_port(),
186 )
187 builder.Connect(JA_gain.get_output_port(),
188     derivative_gain.get_input_port())
189
190 # from Kp
191 builder.Connect(
192     proportional_gain.get_output_port(),
193     operational_space_control_action.get_input_port(0),
194 )
195 # from Kd
196 builder.Connect(
197     derivative_gain.get_output_port(),
198     operational_space_control_action.get_input_port(1),

```

```

199     )
200
201     # Sum to Analytical Jacobian transpose block
202     builder.Connect(
203         operational_space_control_action.get_output_port(),
204         JA_T_gain.get_f_c_input_port(),
205     )
206
207     # JAT to sum
208     builder.Connect(JA_T_gain.get_output_port(),
209                     control_torques.get_input_port(0)
210     )
211     # Grav to sum
212     builder.Connect(gravity_torques.get_output_port(),
213                     control_torques.get_input_port(1)
214     )
215
216     # Positon error input
217     builder.Connect(
218         p_desired_source.get_output_port(),
219         operational_space_position_error.get_input_port(0)
220     )
221     builder.Connect(
222         direct_kinematics.get_output_port(),
223         operational_space_position_error.get_input_port(1)
224     )
225     # =====
226
227     builder.Connect(q.get_output_port(), gravity_torques.get_input_port())
228     builder.Connect(q.get_output_port(), direct_kinematics.get_q_input_port())
229     builder.Connect(q.get_output_port(), JA_gain.get_q_input_port())
230     builder.Connect(q.get_output_port(), JA_T_gain.get_q_input_port())
231
232     # This samples the controller at the specified period (to simulate discrete
233     # control)
234     builder.Connect(control_torques.get_output_port(), zoh.get_input_port())
235
236     builder.Connect(qdot.get_output_port(), JA_gain.get_qdot_input_port())
237
238     builder.ExportInput(q.get_input_port(), name="q")
239     builder.ExportInput(qdot.get_input_port(), name="qdot")
240     builder.ExportOutput(zoh.get_output_port(), name="u")
241
242     # -----
243     # Log operational space positions
244     # -----
245     # These systems are special in Drake. They periodically save the output port
246     # value a during a simulation so that it can be accessed later. The value is
247     # saved every
248     # `publish_period` seconds in simulation time.

```

```

249     self.operational_space_position_logger = LogVectorOutput(
250         direct_kinematics.get_output_port(),
251         builder,
252         publish_period=control_sample_period_s,
253     )
254     self.operational_space_position_logger.set_name("Tip Position Logger")
255
256     builder.BuildInto(self)
257     self.set_name("Controller")
258
259     def get_q_input_port(self) -> InputPort:
260         return self.get_input_port(0)
261
262     def get_qdot_input_port(self) -> InputPort:
263         return self.get_input_port(1)
264
265
266     def run_simulation(
267         simulation_duration_s: float,
268         link_lens: tuple[float, ...],
269         load_mass_kg: float,
270         K_P: NDArray[np.double],
271         K_D: NDArray[np.double],
272         p_desired: NDArray[np.double],
273         control_sample_period_s: float,
274     ):
275         """
276         Runs a Drake simulation of operational space PD control with gravity compensation
277         -----
278         simulation_duration_s
279             A float representing the simulation duration in seconds
280
281         link_lens
282             A tuple of two float representing the length of the links (in order)
283
284         load_mass_kg
285             A float representing the load mass in kg
286
287         K_P
288             A numpy array of shape (2, 2) representing the proportional gains of the PD
289             controller, expressed in the base frame
290
291         K_D
292             A numpy array of shape (2, 2) representing the derivative gains of the PD
293             controller, expressed in the base frame
294
295         p_desired
296             A numpy array of shape (2,) representing the desired position of the
297             end-effector
298

```

```

299     control_sample_period_s
300         A float representing the duration of the trajectory in seconds
301
302     Returns
303     -----
304     A tuple of four elements:
305         1) The time-steps of the simulation in seconds
306         2) The simulated end-effector positions in meter corresponding to each time-step
307         4) The controller used during the simulation (this is also a `Diagram` object).
308         4) The high level simulation `Diagram` object
309     """
310     validate_np_array(arr=p_desired, arr_name="p_desired", correct_shape=(2,))
311
312     builder = DiagramBuilder()
313     arm: Arm = builder.AddNamedSystem("arm", Arm(load_mass_kg=load_mass_kg))
314     controller: OperationalSpacePDControllerWithGravityCompensation = (
315         builder.AddNamedSystem(
316             "controller",
317             OperationalSpacePDControllerWithGravityCompensation(
318                 link_lens=link_lens,
319                 K_P=K_P,
320                 K_D=K_D,
321                 control_sample_period_s=control_sample_period_s,
322                 p_desired=p_desired,
323             ),
324         )
325     )
326
327     # =====
328     # TODO: Complete Simulation Block Diagram (Arm + Controller)
329     # Replace `...` below with the correct output or input port.
330     # Note that following convenience methods are available to access the input and
331     # output ports:
332     #     arm:
333     #         - arm.get_q_output_port()
334     #         - arm.get_qdot_output_port()
335     #         - arm.get_input_port()
336     #
337     #     controller:
338     #         - controller.get_q_input_port()
339     #         - controller.get_qdot_input_port()
340     # -----
341     builder.Connect(controller.get_output_port(), arm.get_input_port())
342     builder.Connect(arm.get_q_output_port(), controller.get_q_input_port())
343     builder.Connect(arm.get_qdot_output_port(), controller.get_qdot_input_port())
344     # =====
345
346     # Build a `Diagram` object and use it to make a `Simulator` object for the diagram
347     diagram: Diagram = builder.Build()
348     diagram.set_name("Operational Space PD Control w/ Gravity Compensation")

```

```

349     simulator: Simulator = Simulator(diagram)
350
351     # Get the context (this contains all the information needed to run the simulation)
352     context: Context = simulator.get_mutable_context()
353
354     # Set initial conditions
355     initial_conditions = context.get_mutable_continuous_state_vector()
356     q_initial = calc_2R_planar_inverse_kinematics(
357         link_lens, end_effector_position=p_desired - 0.1, use_elbow_up_soln=True
358     )
359     initial_conditions.SetAtIndex(2, q_initial[0])
360     initial_conditions.SetAtIndex(3, q_initial[1])
361
362     # Advance the simulation by `simulation_duration_s` seconds using the
363     # `simulator.AdvanceTo()` function
364     simulator.AdvanceTo(simulation_duration_s)
365
366     # -----
367     # Extract simulation outputs
368     # -----
369     # The lines below extract the joint position log from the simulator context
370     operational_space_position_log = (
371         controller.operational_space_position_logger.FindLog(simulator.get_context())
372     )
373     t = operational_space_position_log.sample_times()
374     p_actual = operational_space_position_log.data()
375
376     return t, p_actual, controller, diagram
377
378
379 if __name__ == "__main__":
380     # =====
381     # TODO: Problem 1 - Part (b)
382     # Replace `...` with the appropriate value from the problem statement based on
383     # the comment describing each variable (on the line(s) above it).
384     # -----
385     # A tuple with two elements representing the first and second link lengths of the
386     # manipulator, respectively.
387     link_lens = 1, 1
388
389     # A float representing the load mass in kg
390     load_mass_kg = 10
391
392     # A numpy array of shape (2,) representing the desired end-effector position for the
393     # first case
394     p_desired_case1 = np.array([0.6, -0.2])
395
396     # A numpy array of shape (2,) representing the desired end-effector position for the
397     # second case
398     p_desired_case2 = np.array([0.5, 0.5])

```



```

399
400 # A float representing the time horizon of the entire simulation
401 simulation_duration_s = 2.5
402
403 # A float representing the sampling time of discrete-time controller
404 control_sample_period_s = 1e-3
405
406 # A numpy array of shape (2, 2) representing the PD controller proportional gains
407 K_P = np.array([[100000, 0],
408                 [0, 100000]])
409
410 # A numpy array of shape (2, 2) representing the PD controller derivative gains
411 K_D = np.array([[19500, 0],
412                 [0, 19500]])
413
414 # -----
415 # TODO: Run Simulation
416 #   Replace `...` in parameters for `run_simulation` function using the variables
417 #   above
418 # -----
419 # Run case 1
420 t_case1, p_actual_case1, controller_diagram, simulation_diagram = run_simulation(
421     simulation_duration_s=simulation_duration_s,
422     link_lens=link_lens,
423     load_mass_kg=load_mass_kg,
424     K_P=K_P,
425     K_D=K_D,
426     p_desired=p_desired_case1,
427     control_sample_period_s=control_sample_period_s,
428 )
429 print("finish case1 sim")
430
431 # Run case 2
432 t_case2, p_actual_case2, controller_diagram_case2, simulation_diagram_case2 =
run_simulation(
433     simulation_duration_s=simulation_duration_s,
434     link_lens=link_lens,
435     load_mass_kg=load_mass_kg,
436     K_P=K_P,
437     K_D=K_D,
438     p_desired=p_desired_case2,
439     control_sample_period_s=control_sample_period_s,
440 )
441 print("finish case2 sim")
442
443 # -----
444 # TODO: Plot Controller Block Diagram
445 #   Use the `plot_diagram` function to plot the diagram of the controller design
446 #   (which is stored in the `controller_diagram` variable)
447 # -----

```

```

448 controller_diagram_fig, _ = plot_diagram(controller_diagram, fig_width_in=11)
449 controller_diagram_fig.savefig('Problem1/diagram_case1.png', dpi=300)
450 controller_diagram_fig2, _ = plot_diagram(controller_diagram_case2, fig_width_in=11)
451 controller_diagram_fig2.savefig('Problem1/diagram_case2.png', dpi=300)
452 print("plotted control diagram")
453
454 # -----
455 # TODO: Plot Simulation Block Diagram
456 # Use the `plot_diagram` function to plot the high-level diagram of the simulation
457 # (which is stored in the `simulation_diagram` variable)
458 # -----
459 simulation_diagram_fig, _ = plot_diagram(simulation_diagram, fig_width_in=8)
460 simulation_diagram_fig.savefig('Problem1/simulationDiagram_case1.png', dpi=300)
461 simulation_diagram_fig2, _ = plot_diagram(simulation_diagram_case2, fig_width_in=8)
462 simulation_diagram_fig2.savefig('Problem1/simulationDiagram_case2.png', dpi=300)
463 print("plotted simulation diagram")
464 # =====
465
466 # =====
467 # TODO: Problem 2 - Part (c)
468 # Use the `print` function to output your gains
469 # -----
470 print("K_P:")
471 print(K_P)
472 print("\nK_D:")
473 print(K_D)
474
475 # =====
476 # TODO: Problem 2 - Part (d)
477 # -----
478 # TODO: Plot Case 1 Tip X and Y Positions
479 # For Case 1:
480 # 1) Plot the time history of the x and y coordinates of the end effector
481 # position in separate sub-figures for a time horizon of 2.5 seconds. Use a
482 # solid red line for both the x and y positions.
483 # 2) Indicate the desired coordinate value in each sub-figure by drawing a
484 # solid black dashed horizontal line at the desired value
485 # -----
486 # Plot data in `p_actual_case1`
487 # Create figure and axes
488 fig = plt.figure(figsize=(10, 5))
489 case1_x = fig.add_subplot(121)
490 case1_y = fig.add_subplot(122)
491
492 # Label Plots
493 fig.suptitle("Case 1: EE Position")
494 case1_x.set_title("X Position vs Time")
495 case1_x.set_xlabel("Time [s]")
496 case1_x.set_ylabel("X [m]")
497 case1_y.set_title("Y Position vs Time")

```

```

498     case1_y.set_xlabel("Time [s]")
499     case1_y.set_ylabel("Y [m]")
500
501     case1_x.axhline(
502         p_desired_case1[0], ls="--", color="red", label="Desired X"
503     )
504     case1_y.axhline(
505         p_desired_case1[1], ls="--", color="red", label="Desired Y"
506     )
507
508     case1_x.plot(
509         t_case1, p_actual_case1[0], color="black", label="EE X Position"
510     )
511     case1_y.plot(
512         t_case1, p_actual_case1[1], color="black", label="EE Y Position"
513     )
514     case1_x.legend()
515     case1_y.legend()
516     fig.savefig('Problem1/Case1_Positions.png', dpi=300)
517     print("plotted case 1 positions")
518     plt.clf
519
520     # -----
521     # TODO: Plot Case 2 Tip X and Y Positions
522     # For Case 2:
523     #     1) Plot the time history of the x and y coordinates of the end effector
524     #         position in separate sub-figures for a time horizon of 2.5 seconds. Use a
525     #         solid red line for both the x and y positions.
526     #     2) Indicate the desired coordinate value in each sub-figure by drawing a
527     #         solid black dashed horizontal line at the desired value
528     # -----
529     # Plot data in `p_actual_case2`
530     fig2 = plt.figure(figsize=(10, 5))
531     case2_x = fig2.add_subplot(121)
532     case2_y = fig2.add_subplot(122)
533
534     # Label Plots
535     fig2.suptitle("Case 2: EE Position")
536     case2_x.set_title("X Position vs Time")
537     case2_x.set_xlabel("Time [s]")
538     case2_x.set_ylabel("X [m]")
539     case2_y.set_title("Y Position vs Time")
540     case2_y.set_xlabel("Time [s]")
541     case2_y.set_ylabel("Y [m]")
542
543     case2_x.axhline(
544         p_desired_case2[0], ls="--", color="red", label="Desired X"
545     )
546     case2_y.axhline(
547         p_desired_case2[1], ls="--", color="red", label="Desired Y"

```

```
548     )
549     case2_x.plot(
550         t_case2, p_actual_case2[0], color="black", label="EE X Position"
551     )
552     case2_y.plot(
553         t_case2, p_actual_case2[1], color="black", label="EE Y Position"
554     )
555     case2_x.legend()
556     case2_y.legend()
557     fig2.savefig('Problem1/Case2_Positions.png', dpi=300)
558     print("plotted case 2 positions")
559     # =====
560
561     #plt.show()
562
```