

## mechae263C\_homework5\_problem2.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     OutputPort,
21 )
22 from pydrake.systems.primitives import (
23     MatrixGain,
24     PassThrough,
25     ZeroOrderHold,
26     LogVectorOutput,
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.kinematics import calc_2R_planar_inverse_kinematics
32 from mechae263C_helpers.hw5.op_space import Environment, OperationalSpaceDecoupledArm
33 from mechae263C_helpers.hw5.trajectory import (
34     PrePlannedTrajectorySource,
35     eval_trapz_traj,
36 )
37
38
39 def calc_analytical_jacobian(
40     q1: float, q2: float, a1: float, a2: float
41 ) -> NDArray[np.double]:
42     """
43     Calculates the Analytical Jacobian of a 2R planar manipulator
44
45     Parameters
46     -----
47     q1
48         A float representing the first joint angle

```

```

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

```

```

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

```

```

149         "o_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
150     )
151     operational_space_velocity_error: LinearCombination = builder.AddNamedSystem(
152         "odot_tilde", LinearCombination(input_coeffs=(1, -1), input_shapes=(2,))
153     )
154     operational_space_control_action: LinearCombination = builder.AddNamedSystem(
155         "f_c", LinearCombination(input_coeffs=(1, 1, -1), input_shapes=(2,))
156     )
157     control_torques: LinearCombination = builder.AddNamedSystem(
158         "u", LinearCombination(input_coeffs=(1, 1), input_shapes=(2,))
159     )
160
161     traj_times = np.arange(
162         0, simulation_duration_s + control_sample_period_s, control_sample_period_s
163     )
164     o_d, odot_d, oddot_d = eval_trapz_traj(
165         times=traj_times,
166         max_velocity=0.5,
167         final_time=trajectory_duration_s,
168         initial_position=p_initial,
169         final_position=p_final,
170     )
171
172     o_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
173         "o_d",
174         PrePlannedTrajectorySource(
175             name="o_d",
176             num_joints=self.num_dofs,
177             times=traj_times,
178             values=o_d,
179         ),
180     )
181     odot_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
182         "odot_d",
183         PrePlannedTrajectorySource(
184             name="odot_d",
185             num_joints=self.num_dofs,
186             times=traj_times,
187             values=odot_d,
188         ),
189     )
190     oddot_d: PrePlannedTrajectorySource = builder.AddNamedSystem(
191         "oddot_d",
192         PrePlannedTrajectorySource(
193             name="oddot_d",
194             num_joints=self.num_dofs,
195             times=traj_times,
196             values=oddot_d,
197         ),
198     )

```

```

199
200     zoh: ZeroOrderHold = builder.AddNamedSystem(
201         "sampled_oddodt_e",
202         ZeroOrderHold(
203             period_sec=self.control_sample_period_s, vector_size=self.num_dofs
204         ),
205     )
206
207     o_e: PassThrough = builder.AddNamedSystem("o_e", PassThrough(vector_size=2))
208     odot_e: PassThrough = builder.AddNamedSystem(
209         "odot_e", PassThrough(vector_size=2)
210     )
211     u: PassThrough = builder.AddNamedSystem("command", PassThrough(vector_size=2))
212     f_e: PassThrough = builder.AddNamedSystem("f_e", PassThrough(vector_size=2))
213
214     # =====
215     # TODO: Complete Controller Block Diagram
216     #   Replace `...` below with the correct output or input port.
217     # -----
218     builder.Connect(
219         o_d.get_output_port(),
220         operational_space_position_error.get_input_port(0)
221     )
222     builder.Connect(
223         o_e.get_output_port(),
224         operational_space_position_error.get_input_port(1)
225     )
226     builder.Connect(
227         odot_d.get_output_port(),
228         operational_space_velocity_error.get_input_port(0)
229     )
230     builder.Connect(
231         odot_e.get_output_port(),
232         operational_space_velocity_error.get_input_port(1)
233     )
234
235     builder.Connect(
236         oddot_d.get_output_port(),
237         control_torques.get_input_port(0)
238     )
239     # Find the difference then apply the gain matrix
240     builder.Connect(
241         operational_space_position_error.get_output_port(),
242         K_P.get_input_port()
243     )
244
245     # Operational control action
246     builder.Connect(
247         operational_space_velocity_error.get_output_port(),
248         K_D.get_input_port()

```

```

249     )
250     builder.Connect(
251         K_D.get_output_port(),
252         operational_space_control_action.get_input_port(0)
253     )
254     # sum before inverse mass
255     builder.Connect(
256         K_P.get_output_port(),
257         operational_space_control_action.get_input_port(1)
258     )
259     builder.Connect(
260         f_e.get_output_port(),
261         operational_space_control_action.get_input_port(2)
262     )
263
264     builder.Connect(
265         operational_space_control_action.get_output_port(),
266         invM_d.get_input_port()
267     )
268     builder.Connect(
269         invM_d.get_output_port(),
270         control_torques.get_input_port(1))
271     # =====
272
273     # This samples the controller at the specified period (to simulate discrete
274     # control)
275     builder.Connect(control_torques.get_output_port(), zoh.get_input_port())
276     builder.Connect(zoh.get_output_port(), u.get_input_port())
277
278     builder.ExportInput(o_e.get_input_port(), "o_e")
279     builder.ExportInput(odot_e.get_input_port(), "odot_e")
280     builder.ExportInput(f_e.get_input_port(), "f_e")
281     builder.ExportOutput(u.get_output_port(), "u")
282
283     # -----
284     # Log Position Error and Force
285     # -----
286     # These systems are special in Drake. They periodically save the output port
287     # value a during a simulation so that it can be accessed later. The value is
288     # saved every `publish_period` seconds in simulation time.
289     self.force_logger = LogVectorOutput(
290         f_e.get_output_port(),
291         builder,
292         publish_period=control_sample_period_s,
293     )
294     self.force_logger.set_name("Force Logger")
295     self.position_error_logger = LogVectorOutput(
296         operational_space_position_error.get_output_port(),
297         builder,
298         publish_period=control_sample_period_s,

```

```

299         )
300         self.position_error_logger.set_name("Tip Position Error Logger")
301
302         builder.BuildInto(self)
303         self.set_name("Controller")
304
305     def get_o_e_input_port(self) -> InputPort:
306         """Returns the input port for operational space end-effector position"""
307         return self.get_input_port(0)
308
309     def get_odot_e_input_port(self) -> InputPort:
310         """Returns the input port for operational space end-effector velocity"""
311
312         return self.get_input_port(1)
313
314     def get_f_e_input_port(self) -> InputPort:
315         """Returns the input port for operational space end-effector force"""
316         return self.get_input_port(2)
317
318     def get_u_output_port(self) -> OutputPort:
319         """Returns the output port for the control command"""
320         return self.get_output_port()
321
322
323     def run_simulation(
324         simulation_duration_s: float,
325         link_lens: tuple[float, float],
326         M_d: NDArray,
327         K_P: NDArray,
328         K_D: NDArray,
329         p_initial: NDArray[np.double],
330         p_final: NDArray[np.double],
331         trajectory_duration_s: float,
332         o_r: NDArray[np.double],
333         K: NDArray,
334         control_sample_period_s: float,
335     ):
336         """
337         Runs a Drake simulation of an operational space impedance controller
338         Parameters
339         -----
340         simulation_duration_s
341             A float representing the simulation duration in seconds
342
343         link_lens
344             A tuple of two float representing the length of the links (in order)
345
346         M_d
347             A numpy array of shape (2, 2) representing the inertia gains of the impedance
348             controller, expressed in the base frame

```

```

349
350     K_P
351         A numpy array of shape (2, 2) representing the proportional gains of the
352         impedance controller, expressed in the base frame
353
354     K_D
355         A numpy array of shape (2, 2) representing the derivative gains of the impedance
356         controller, expressed in the base frame
357
358     p_initial
359         A numpy array of shape (2,) representing the initial position of the
360         end-effector, expressed in the base frame
361
362     p_final
363         A numpy array of shape (2,) representing the final position of the end-effector,
364         expressed in the base frame
365
366     trajectory_duration_s
367         A float representing the duration of the trajectory in seconds
368
369     o_r
370         A numpy array of shape (2,) representing the base frame coordinates for the
371         point where the undeformed elastically compliant plane intersects the base frame
372         x-axis
373
374     K
375         A numpy array of shape (2, 2) representing the environment's stiffness matrix in
376         the base frame (see hint in problem statement)
377
378     control_sample_period_s
379         A float representing the duration of the trajectory in seconds
380
381     Returns
382     -----
383     A tuple of five elements:
384         1) The time-steps of the simulation in seconds
385         2) The simulated end-effector forces in Newtons corresponding to each time-step
386         3) The simulated position errors in meters corresponding to each time-step
387         4) The controller used during the simulation (this is also a `Diagram` object).
388         4) The high level simulation `Diagram` object
389     """
390     validate_np_array(arr=p_initial, arr_name="p_initial", correct_shape=(2,))
391     validate_np_array(arr=p_final, arr_name="p_final", correct_shape=(2,))
392
393     builder = DiagramBuilder()
394     arm: OperationalSpaceDecoupledArm = builder.AddNamedSystem(
395         "arm",
396         OperationalSpaceDecoupledArm(
397             link_lens, calc_direct_kinematics, calc_analytical_jacobian
398         ),

```



```

399     )
400     controller: OperationalSpaceImpedanceController = builder.AddNamedSystem(
401         "controller",
402         OperationalSpaceImpedanceController(
403             link_lens=link_lens,
404             M_d=M_d,
405             K_P=K_P,
406             K_D=K_D,
407             control_sample_period_s=control_sample_period_s,
408             p_initial=p_initial,
409             p_final=p_final,
410             trajectory_duration_s=trajectory_duration_s,
411         ),
412     )
413     environment: Environment = builder.AddNamedSystem(
414         "environment", Environment(equilibrium_position=o_r, stiffness=K)
415     )
416
417     # =====
418     # TODO: Complete High-Level Simulation Block Diagram: Arm + Controller + Environment
419     # Replace `...` below with the correct output or input port.
420     # Note that following convenience methods are available to access the input and
421     # output ports:
422     #     arm:
423     #         - arm.get_f_e_input_port()
424     #         - arm.get_u_input_port()
425     #         - arm.get_o_e_output_port()
426     #         - arm.get_odot_e_output_port()
427     #
428     #     controller:
429     #         - controller.get_f_e_input_port()
430     #         - controller.get_o_e_input_port()
431     #         - controller.get_odot_e_input_port()
432     #         - controller.get_u_output_port()
433     #
434     #     environment
435     #         - environment.get_o_e_input_port()
436     #         - environment.get_f_e_output_port()
437     # -----
438     builder.Connect(controller.get_u_output_port(),
439                     arm.get_u_input_port())
440
441     builder.Connect(arm.get_o_e_output_port(),
442                     controller.get_o_e_input_port())
443     builder.Connect(arm.get_o_e_output_port(),
444                     environment.get_o_e_input_port())
445     builder.Connect(arm.get_odot_e_output_port(),
446                     controller.get_odot_e_input_port())
447
448     builder.Connect(environment.get_f_e_output_port(),

```

```

449         arm.get_f_e_input_port())
450     builder.Connect(environment.get_f_e_output_port(),
451                     controller.get_f_e_input_port())
452     # =====
453
454     # Build a `Diagram` object and use it to make a `Simulator` object for the diagram
455     diagram: Diagram = builder.Build()
456     diagram.set_name("Operational Impedance Control")
457     simulator: Simulator = Simulator(diagram)
458
459     # Get the context (this contains all the information needed to run the simulation)
460     context: Context = simulator.get_mutable_context()
461
462     # Set initial conditions
463     initial_conditions = context.get_mutable_continuous_state_vector()
464     q_initial = calc_2R_planar_inverse_kinematics(
465         link_lens, end_effector_position=p_initial, use_elbow_up_soln=True
466     )
467     initial_conditions.SetAtIndex(2, q_initial[0])
468     initial_conditions.SetAtIndex(3, q_initial[1])
469
470     # Advance the simulation by `simulation_duration_s` seconds using the
471     # `simulator.AdvanceTo()` function
472     simulator.set_target_realtime_rate(1.0)
473     simulator.AdvanceTo(simulation_duration_s)
474
475     # -----
476     # Extract simulation outputs
477     # -----
478     # The lines below extract the joint position log from the simulator context
479     force_log = controller.force_logger.FindLog(simulator.get_context())
480     t = force_log.sample_times()
481     force = force_log.data()
482     position_error_log = controller.position_error_logger.FindLog(
483         simulator.get_context()
484     )
485     position_error = position_error_log.data()
486
487     return t, force, position_error, controller, diagram
488
489 if __name__ == "__main__":
490     # =====
491     # TODO: Problem 2 - Part (a)
492     # Replace `...` with the appropriate value from the problem statement based on
493     # the comment describing each variable (on the line(s) above it).
494     # -----
495     # A tuple with two elements representing the first and second link lengths of the
496     # manipulator, respectively.
497     link_lens = 1, 1

```

```
499
500 # A numpy array of shape (2, 2) representing the rotation matrix that rotates from
501 # the base frame to the constraint frame
502 R_c = np.array([[np.cos(np.pi/4), -np.sin(np.pi/4)],
503                 [np.sin(np.pi/4), np.cos(np.pi/4)]])
504
505 # A numpy array of shape (2, 2) representing the environment's stiffness matrix in
506 # the constraint frame
507 K_c = np.array([[0, 0],
508                 [0, 5000]])
509
510 # A numpy array of shape (2, 2) representing the environment's stiffness matrix in
511 # the base frame (see hint in problem statement)
512 K = R_c @ K_c @ np.transpose(R_c)
513
514 # A numpy array of shape (2,) representing the base frame coordinates for the point
515 # where the undeformed elastically compliant plane intersects the base frame x-axis
516 o_r = np.array([1, 0])
517
518 # A numpy array of shape (2, 2) representing the impedance controller proportional
519 # gains in the constraint frame
520 K_P_c = np.array([[1.2e7, 0],
521                   [0, 1.2e7]])
522
523 # A numpy array of shape (2, 2) representing the impedance controller derivative
524 # gains in the constraint frame
525 K_D_c = np.array([[1.15e5, 0],
526                   [0, 1.15e5]])
527
528 # A numpy array of shape (2, 2) representing the impedance controller inertia gains
529 # in the constraint frame
530 M_d_c = np.array([[5e3, 0],
531                   [0, 5e3]])
532
533 # A numpy array of shape (2, 2) representing the impedance controller proportional
534 # gains in the base frame (see hint in problem statement)
535 K_P = R_c @ K_P_c @ np.transpose(R_c)
536
537 # A numpy array of shape (2, 2) representing the impedance controller derivative
538 # gains in the base frame (see hint in problem statement)
539 K_D = R_c @ K_D_c @ np.transpose(R_c)
540
541 # A numpy array of shape (2, 2) representing the impedance controller inertia gains
542 # in the base frame (see hint in problem statement)
543 M_d = R_c @ M_d_c @ np.transpose(R_c)
544
545 # Print out your impedance controller gains
546 print("M_d_c:")
547 print(M_d_c)
548 print("\nK_P_c:")
```

```

549     print(K_P_c)
550     print("\nK_D_c:")
551     print(K_D_c)
552
553     # A numpy array of shape (2,) representing the initial end-effector position in base
554     # frame
555     p_initial = np.array([1 + 0.1*np.sqrt(2), 0])
556
557     # A numpy array of shape (2,) representing the final end-effector position in base
558     # frame
559     p_final = np.array([1.2 + 0.1*np.sqrt(2), 0.2])
560
561     # A float representing the duration of the trapezoidal velocity trajectory in units
562     # of seconds
563     trajectory_duration_s = 2
564
565     # A float representing the time horizon of the entire simulation
566     simulation_duration_s = 2.5
567
568     # A float representing the sampling time of discrete-time controller
569     control_sample_period_s = 1e-3
570
571     # -----
572     # TODO: Run Simulation
573     # Replace `...` in parameters for `run_simulation` function using the variables
574     # above
575     # -----
576     t, forces, position_error, controller_diagram, simulation_diagram = run_simulation(
577         simulation_duration_s=simulation_duration_s,
578         link_lens=link_lens,
579         M_d=M_d,
580         K_P=K_P,
581         K_D=K_D,
582         p_initial=p_initial,
583         p_final=p_final,
584         trajectory_duration_s=trajectory_duration_s,
585         control_sample_period_s=control_sample_period_s,
586         o_r=o_r,
587         K=K,
588     )
589
590     print('sim finished')
591
592     # -----
593     # TODO: Plot Control Block Diagram
594     # Use the `plot_diagram` function to plot the diagram of the controller design
595     # (which is stored in the `controller_diagram` variable)
596     # -----
597     # controller_diagram_fig, _ = plot_diagram(
598     #     controller_diagram, fig_width_in=11, max_depth=1

```

```

599 # )
600 # controller_diagram_fig.savefig('Problem2/controlDia.png', dpi=300)
601 # print('saved controller diagram')
602
603 # -----
604 # TODO: Plot Simulation Block Diagram
605 # Use the `plot_diagram` function to plot the high-level diagram of the simulation
606 # (which is stored in the `simulation_diagram` variable)
607 # -----
608 # simulation_diagram_fig, _ = plot_diagram(
609 #     simulation_diagram, fig_width_in=8, max_depth=1
610 # )
611 # simulation_diagram_fig.savefig('Problem2/simulationDia.png', dpi=300)
612 # print('saved simulation diagram')
613 # =====
614
615 # =====
616 # TODO: Problem 2 - Part (b)
617 # Report/output the damping ratio and natural frequency in both the x_c and y_c
618 # directions.
619 # Hint: See Example 9.2 in Siciliano et al.
620 # -----
621 kpx = K_P_c[0][0]
622 kpy = K_P_c[1][1]
623 kdx = K_D_c[0][0]
624 kdy = K_D_c[1][1]
625 mdx = M_d_c[0][0]
626 mdy = M_d_c[1][1]
627 kx = K_c[0][0]
628 ky = K_c[1][1]
629
630 natural_frequency_x = np.sqrt(kpx / mdx)
631 natural_frequency_y = np.sqrt((kpy + ky) / mdy)
632 damping_ratio_x = (kdx) / (2*np.sqrt(mdx*kpx))
633 damping_ratio_y = (kdy) / (2*np.sqrt(mdy*(kpy + ky)))
634
635 print("\nnatural_frequency_x:", natural_frequency_x)
636 print("\nnatural_frequency_y:", natural_frequency_y)
637 print("\ndamping_ratio_x:", damping_ratio_x)
638 print("\ndamping_ratio_y:", damping_ratio_y)
639 # =====
640
641 # =====
642 # TODO: Problem 2 - Part (c)
643 # 1) Plot x and y-coordinate of end effector position errors in meters as a
644 # function of time, as expressed in the **base** frame (on the same figure).
645 # Set the x limits to [0, `simulation_duration_s`] and the y limits to
646 # [-0.06, 0.06].
647 # 2) Plot the x-and y-coordinate end-effector contact forces in N as a
648 # function of time, as expressed in the **base** frame (on the same figure).

```

```
649     # Set the x limits to [0, `simulation_duration_s`] and the y limits to
650     # [-550, 550].
651     # Hints:
652     # 1) When plotting, use the `label` argument to automatically add a legend item:
653     # `ax.plot(x, y, label=r"$x_0$")`
654     # 2) You need to call `ax.legend()` to actually plot the legend.
655     # -----
656     # Plot data in `position_error` variable
657     fig = plt.figure(figsize=(10, 5))
658     base_error = fig.add_subplot(111)
659
660     # Label Plots
661     base_error.set_title("Base Frame: Position Error vs Time")
662     base_error.set_xlabel("Time [s]")
663     base_error.set_ylabel("Error [m]")
664     base_error.set_xlim([0, simulation_duration_s])
665     base_error.set_ylim([-0.06, 0.06])
666
667     base_error.plot(
668         t, position_error[0], color="black", label="X Error"
669     )
670     base_error.plot(
671         t, position_error[1], color="blue", label="Y Error"
672     )
673     base_error.legend()
674     fig.savefig('Problem2/Base_PositionError.png', dpi=300)
675     print("plotted base position errors")
676     plt.clf
677
678
679     # Plot data in `forces` variable
680     fig = plt.figure(figsize=(10, 5))
681     base_force = fig.add_subplot(111)
682
683     # Label Plots
684     base_force.set_title("Base Frame: Contact Force vs Time")
685     base_force.set_xlabel("Time [s]")
686     base_force.set_ylabel("Force [N]")
687     base_force.set_xlim([0, simulation_duration_s])
688     base_force.set_ylim([-550, 550])
689
690     base_force.plot(
691         t, forces[0], color="black", label="X Contact Force"
692     )
693     base_force.plot(
694         t, forces[1], color="blue", label="Y Contact Force"
695     )
696     base_force.legend()
697     fig.savefig('Problem2/Base_Force.png', dpi=300)
698     print("plotted base contact forces")
```

```

699 plt.clf
700 # =====
701
702 # =====
703 # TODO: Problem 2 - Part (d)
704 # 1) Plot x and y-coordinate of end effector position errors in meters as a
705 #     function of time, as expressed in the **constraint** frame (on the same
706 #     figure).
707 #     Set the x limits to [0, `simulation_duration_s`] and the y limits to
708 #     [-0.06, 0.06].
709 # 2) Plot the x-and y-coordinate end-effector contact forces in N as a
710 #     function of time, as expressed in the **constraint** frame (on the same
711 #     figure).
712 #     Set the x limits to [0, `simulation_duration_s`] and the y limits to
713 #     [-550, 550].
714 # -----
715 position_error_in_constraint_frame = R_c @ position_error
716 forces_in_constraint_frame = R_c @ forces
717
718 # Plot data in `position_error` variable
719 fig = plt.figure(figsize=(10, 5))
720 con_error = fig.add_subplot(111)
721
722 # Label Plots
723 con_error.set_title("Constraint Frame: Position Error vs Time")
724 con_error.set_xlabel("Time [s]")
725 con_error.set_ylabel("Error [m]")
726 con_error.set_xlim([0, simulation_duration_s])
727 con_error.set_ylim([-0.06, 0.06])
728
729 con_error.plot(
730     t, position_error_in_constraint_frame[0], color="black", label="X Error"
731 )
732 con_error.plot(
733     t, position_error_in_constraint_frame[1], color="blue", label="Y Error"
734 )
735 con_error.legend()
736 fig.savefig('Problem2/Cosntraint_PositionError.png', dpi=300)
737 print("plotted cosntraint position errors")
738 plt.clf
739
740
741 # Plot data in `forces` variable
742 fig = plt.figure(figsize=(10, 5))
743 con_force = fig.add_subplot(111)
744
745 # Label Plots
746 con_force.set_title("Constraint Frame: Contact Force vs Time")
747 con_force.set_xlabel("Time [s]")
748 con_force.set_ylabel("Force [N]")

```

```
749     con_force.set_xlim([0, simulation_duration_s])
750     con_force.set_ylim([-550, 550])
751
752     con_force.plot(
753         t, forces_in_constraint_frame[0], color="black", label="X Contact Force"
754     )
755     con_force.plot(
756         t, forces_in_constraint_frame[1], color="blue", label="Y Contact Force"
757     )
758     con_force.legend()
759     fig.savefig('Problem2/Constraint_Force.png', dpi=300)
760     print("plotted cosntraint contact forces")
761     plt.clf
762     # =====
763
```