

cartpole_swingup_constrained.py

```

1  """
2  Starter code for the problem "Cart-pole swing-up with limited actuation".
3
4  Autonomous Systems Lab (ASL), Stanford University
5  """
6
7  from functools import partial
8
9  from animations import animate_cartpole
10
11 import cvxpy as cvx
12
13 import jax
14 import jax.numpy as jnp
15
16 import matplotlib.pyplot as plt
17
18 import numpy as np
19
20 from tqdm import tqdm
21
22
23 @partial(jax.jit, static_argnums=(0,))
24 @partial(jax.vmap, in_axes=(None, 0, 0))
25 def affinize(f, s, u):
26     """Affinize the function `f(s, u)` around `(s, u)`.
27
28     Arguments
29     -----
30     f : callable
31         A nonlinear function with call signature `f(s, u)`.
32     s : numpy.ndarray
33         The state (1-D).
34     u : numpy.ndarray
35         The control input (1-D).
36
37     Returns
38     -----
39     A : jax.numpy.ndarray
40         The Jacobian of `f` at `(s, u)`, with respect to `s`.
41     B : jax.numpy.ndarray
42         The Jacobian of `f` at `(s, u)`, with respect to `u`.
43     c : jax.numpy.ndarray
44         The offset term in the first-order Taylor expansion of `f` at `(s, u)`
45         that sums all vector terms strictly dependent on the nominal point
46         `(s, u)` alone.
47     """
48     # PART (b) #####

```

```

49 # INSTRUCTIONS: Use JAX to affinize `f` around `(s, u)` in two lines.
50 A,B = jax.jacobian(f,argnums=(0,1))(s,u)
51 c = f(s,u) - A@s - B@u
52 # raise NotImplementedError()
53 # END PART (b) #####
54 return A, B, c
55
56
57 def solve_swingup_scp(f, s0, s_goal, N, P, Q, R, u_max, p, eps, max_iters):
58     """Solve the cart-pole swing-up problem via SCP.
59
60     Arguments
61     -----
62     f : callable
63         A function describing the discrete-time dynamics, such that
64         `s[k+1] = f(s[k], u[k])`.
65     s0 : numpy.ndarray
66         The initial state (1-D).
67     s_goal : numpy.ndarray
68         The goal state (1-D).
69     N : int
70         The time horizon of the LQR cost function.
71     P : numpy.ndarray
72         The terminal state cost matrix (2-D).
73     Q : numpy.ndarray
74         The state stage cost matrix (2-D).
75     R : numpy.ndarray
76         The control stage cost matrix (2-D).
77     u_max : float
78         The bound defining the control set `[-u_max, u_max]`.
79     p : float
80         Trust region radius.
81     eps : float
82         Termination threshold for SCP.
83     max_iters : int
84         Maximum number of SCP iterations.
85
86     Returns
87     -----
88     s : numpy.ndarray
89         A 2-D array where `s[k]` is the open-loop state at time step `k`,
90         for `k = 0, 1, ..., N-1`
91     u : numpy.ndarray
92         A 2-D array where `u[k]` is the open-loop state at time step `k`,
93         for `k = 0, 1, ..., N-1`
94     J : numpy.ndarray
95         A 1-D array where `J[i]` is the SCP sub-problem cost after the i-th
96         iteration, for `i = 0, 1, ..., (iteration when convergence occurred)`
97     """
98     n = Q.shape[0] # state dimension

```

```

99     m = R.shape[0] # control dimension
100
101     # Initialize dynamically feasible nominal trajectories
102     u = np.zeros((N, m))
103     s = np.zeros((N + 1, n))
104     s[0] = s0
105     for k in range(N):
106         s[k + 1] = fd(s[k], u[k])
107
108     # Do SCP until convergence or maximum number of iterations is reached
109     converged = False
110     J = np.zeros(max_iters + 1)
111     J[0] = np.inf
112     for i in (prog_bar := tqdm(range(max_iters))):
113         s, u, J[i + 1] = scp_iteration(f, s0, s_goal, s, u, N, P, Q, R, u_max, p)
114         dJ = np.abs(J[i + 1] - J[i])
115         prog_bar.set_postfix({"objective change": "{:.5f}".format(dJ)})
116         if dJ < eps:
117             converged = True
118             print("SCP converged after {} iterations.".format(i))
119             break
120     if not converged:
121         raise RuntimeError("SCP did not converge!")
122     J = J[1 : i + 1]
123     return s, u, J
124
125
126 def scp_iteration(f, s0, s_goal, s_prev, u_prev, N, P, Q, R, u_max, p):
127     """Solve a single SCP sub-problem for the cart-pole swing-up problem.
128
129     Arguments
130     -----
131     f : callable
132         A function describing the discrete-time dynamics, such that
133         `s[k+1] = f(s[k], u[k])`.
134     s0 : numpy.ndarray
135         The initial state (1-D).
136     s_goal : numpy.ndarray
137         The goal state (1-D).
138     s_prev : numpy.ndarray
139         The state trajectory around which the problem is convexified (2-D).
140     u_prev : numpy.ndarray
141         The control trajectory around which the problem is convexified (2-D).
142     N : int
143         The time horizon of the LQR cost function.
144     P : numpy.ndarray
145         The terminal state cost matrix (2-D).
146     Q : numpy.ndarray
147         The state stage cost matrix (2-D).
148     R : numpy.ndarray

```

```

149     The control stage cost matrix (2-D).
150     u_max : float
151     The bound defining the control set `[-u_max, u_max]`.
152     p : float
153     Trust region radius.
154
155     Returns
156     -----
157     s : numpy.ndarray
158     A 2-D array where `s[k]` is the open-loop state at time step `k`,
159     for `k = 0, 1, ..., N-1`
160     u : numpy.ndarray
161     A 2-D array where `u[k]` is the open-loop state at time step `k`,
162     for `k = 0, 1, ..., N-1`
163     J : float
164     The SCP sub-problem cost.
165     """
166     A, B, c = affinize(f, s_prev[:-1], u_prev)
167     A, B, c = np.array(A), np.array(B), np.array(c)
168     n = Q.shape[0]
169     m = R.shape[0]
170     s_cvx = cvx.Variable((N + 1, n))
171     u_cvx = cvx.Variable((N, m))
172
173     # PART (c) #####
174     # INSTRUCTIONS: Construct the convex SCP sub-problem.
175     objective = 0.0
176     constraints = [s_cvx[0,:] == s0]
177     # constraints = []
178     for k in range(0,N):
179
180         # objective = objective + (s_cvx[k,:].T @ Q @ s_cvx[k,:] + u_cvx[k,:].T @ R @ u_cvx[k,:])
181         objective = objective + cvx.quad_form(s_cvx[k,:]-s_goal,Q) +
cvx.quad_form(u_cvx[k,:],R)
182         constraints.append(s_cvx[k+1,:] == A[k,:,:]@s_cvx[k,:] + B[k,:,:]@u_cvx[k,:] +
c[k,:])
183         constraints.append(u_cvx[k,:] >= -u_max)
184         constraints.append(u_cvx[k,:] <= u_max)
185         constraints.append(cvx.norm(s_cvx[k, :] - s_prev[k, :], p=np.inf) <= p)
186         constraints.append(cvx.norm(u_cvx[k, :] - u_prev[k, :], p=np.inf) <= p)
187
188     # objective = objective + s_cvx[N,:].T @ P @ s_cvx[N,:].T
189     objective = objective + cvx.quad_form(s_cvx[N,:]-s_goal,P)
190     constraints.append(cvx.norm(s_cvx[N, :] - s_prev[N, :], p=np.inf) <= p)
191
192     # raise NotImplementedError()
193     # END PART (c) #####
194
195     prob = cvx.Problem(cvx.Minimize(objective), constraints)
196     prob.solve()

```

```

197     if prob.status != "optimal":
198         raise RuntimeError("SCP solve failed. Problem status: " + prob.status)
199     s = s_cvx.value
200     u = u_cvx.value
201     J = prob.objective.value
202     return s, u, J
203
204
205 def cartpole(s, u):
206     """Compute the cart-pole state derivative."""
207     mp = 1.0 # pendulum mass
208     mc = 4.0 # cart mass
209     L = 1.0 # pendulum length
210     g = 9.81 # gravitational acceleration
211
212     x, theta, dx, dtheta = s
213     sintheta, costheta = jnp.sin(theta), jnp.cos(theta)
214     h = mc + mp * (sintheta**2)
215     ds = jnp.array(
216         [
217             dx,
218             dtheta,
219             (mp * sintheta * (L * (dtheta**2) + g * costheta) + u[0]) / h,
220             -((mc + mp) * g * sintheta + mp * L * (dtheta**2) * sintheta * costheta + u[0] * costheta)
221             / (h * L),
222         ]
223     )
224     return ds
225
226
227 def discretize(f, dt):
228     """Discretize continuous-time dynamics `f` via Runge-Kutta integration."""
229
230     def integrator(s, u, dt=dt):
231         k1 = dt * f(s, u)
232         k2 = dt * f(s + k1 / 2, u)
233         k3 = dt * f(s + k2 / 2, u)
234         k4 = dt * f(s + k3, u)
235         return s + (k1 + 2 * k2 + 2 * k3 + k4) / 6
236
237     return integrator
238
239
240 # Define constants
241 n = 4 # state dimension
242 m = 1 # control dimension
243 s_goal = np.array([0, np.pi, 0, 0]) # desired upright pendulum state
244 s0 = np.array([0, 0, 0, 0]) # initial downright pendulum state
245 dt = 0.1 # discrete time resolution
246 T = 10.0 # total simulation time

```

```

247 P = 1e3 * np.eye(n) # terminal state cost matrix
248 Q = np.diag([1e-2, 1.0, 1e-3, 1e-3]) # state cost matrix
249 R = 1e-3 * np.eye(m) # control cost matrix
250 ρ = 1.0 # trust region parameter
251 u_max = 8.0 # control effort bound
252 eps = 5e-1 # convergence tolerance
253 max_iters = 100 # maximum number of SCP iterations
254 animate = False # flag for animation
255
256 # Initialize the discrete-time dynamics
257 fd = jax.jit(discretize(cartpole, dt))
258
259 # Solve the swing-up problem with SCP
260 t = np.arange(0.0, T + dt, dt)
261 N = t.size - 1
262 s, u, J = solve_swingup_scp(fd, s0, s_goal, N, P, Q, R, u_max, ρ, eps, max_iters)
263
264 # Simulate open-loop control
265 for k in range(N):
266     s[k + 1] = fd(s[k], u[k])
267
268 # Plot state and control trajectories
269 fig, ax = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
270 plt.subplots_adjust(wspace=0.45)
271 labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$", r"$\dot{\theta}(t)$")
272 labels_u = (r"$u(t)$",)
273 for i in range(n):
274     ax[i].plot(t, s[:, i])
275     ax[i].axhline(s_goal[i], linestyle="--", color="tab:orange")
276     ax[i].set_xlabel(r"$t$")
277     ax[i].set_ylabel(labels_s[i])
278 for i in range(m):
279     ax[n + i].plot(t[:-1], u[:, i])
280     ax[n + i].axhline(u_max, linestyle="--", color="tab:orange")
281     ax[n + i].axhline(-u_max, linestyle="--", color="tab:orange")
282     ax[n + i].set_xlabel(r"$t$")
283     ax[n + i].set_ylabel(labels_u[i])
284 plt.savefig("cartpole_swingup_constrained.png", bbox_inches="tight")
285
286 # Plot cost history over SCP iterations
287 fig, ax = plt.subplots(1, 1, dpi=150, figsize=(8, 5))
288 ax.semilogy(J)
289 ax.set_xlabel(r"SCP iteration $i$")
290 ax.set_ylabel(r"SCP cost $\bar{J}(\bar{x}^{(i)}), \bar{u}^{(i)}$")
291 plt.savefig("cartpole_swingup_constrained_cost.png", bbox_inches="tight")
292 plt.show()
293
294 # Animate the solution
295 if animate:
296     fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])

```

```
297 ani.save("cartpole_swingup_constrained.mp4", writer="ffmpeg")
298 plt.show()
299
```