cartpole_swingup_constrained.py

```python
"""
Starter code for the problem "Cart-pole swing-up with limited actuation".

Autonomous Systems Lab (ASL), Stanford University
"""

from functools import partial

from animations import animate_cartpole

import cvxpy as cvx

import jax
import jax.numpy as jnp

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm


@partial(jax.jit, static_argnums=(0,))
@partial(jax.vmap, in_axes=(None, 0, 0))
def affinize(f, s, u):
    """Affinize the function `f(s, u)` around `(s, u)`.

    Arguments
    ---------
    f : callable
        A nonlinear function with call signature `f(s, u)`.
    s : numpy.ndarray
        The state (1-D).
    u : numpy.ndarray
        The control input (1-D).

    Returns
    -------
    A : jax.numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `s`.
    B : jax.numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `u`.
    c : jax.numpy.ndarray
        The offset term in the first-order Taylor expansion of `f` at `(s, u)`
        that sums all vector terms strictly dependent on the nominal point
        `(s, u)` alone.
    """
    # PART (b) ############################################################
```

```python
    # INSTRUCTIONS: Use JAX to affinize `f` around `(s, u)` in two lines.
    A,B = jax.jacobian(f,argnums=(0,1))(s,u)
    c = f(s,u) - A@s - B@u
    # raise NotImplementedError()
    # END PART (b) ############################################################
    return A, B, c


def solve_swingup_scp(f, s0, s_goal, N, P, Q, R, u_max, ρ, eps, max_iters):
    """Solve the cart-pole swing-up problem via SCP.

    Arguments
    ---------
    f : callable
        A function describing the discrete-time dynamics, such that
        `s[k+1] = f(s[k], u[k])`.
    s0 : numpy.ndarray
        The initial state (1-D).
    s_goal : numpy.ndarray
        The goal state (1-D).
    N : int
        The time horizon of the LQR cost function.
    P : numpy.ndarray
        The terminal state cost matrix (2-D).
    Q : numpy.ndarray
        The state stage cost matrix (2-D).
    R : numpy.ndarray
        The control stage cost matrix (2-D).
    u_max : float
        The bound defining the control set `[-u_max, u_max]`.
    ρ : float
        Trust region radius.
    eps : float
        Termination threshold for SCP.
    max_iters : int
        Maximum number of SCP iterations.

    Returns
    -------
    s : numpy.ndarray
        A 2-D array where `s[k]` is the open-loop state at time step `k`,
        for `k = 0, 1, ..., N-1`
    u : numpy.ndarray
        A 2-D array where `u[k]` is the open-loop state at time step `k`,
        for `k = 0, 1, ..., N-1`
    J : numpy.ndarray
        A 1-D array where `J[i]` is the SCP sub-problem cost after the i-th
        iteration, for `i = 0, 1, ..., (iteration when convergence occured)`
    """
    n = Q.shape[0]  # state dimension
```

```python
    m = R.shape[0]  # control dimension

    # Initialize dynamically feasible nominal trajectories
    u = np.zeros((N, m))
    s = np.zeros((N + 1, n))
    s[0] = s0
    for k in range(N):
        s[k + 1] = fd(s[k], u[k])

    # Do SCP until convergence or maximum number of iterations is reached
    converged = False
    J = np.zeros(max_iters + 1)
    J[0] = np.inf
    for i in (prog_bar := tqdm(range(max_iters))):
        s, u, J[i + 1] = scp_iteration(f, s0, s_goal, s, u, N, P, Q, R, u_max, ρ)
        dJ = np.abs(J[i + 1] - J[i])
        prog_bar.set_postfix({"objective change": "{:.5f}".format(dJ)})
        if dJ < eps:
            converged = True
            print("SCP converged after {} iterations.".format(i))
            break
    if not converged:
        raise RuntimeError("SCP did not converge!")
    J = J[1 : i + 1]
    return s, u, J


def scp_iteration(f, s0, s_goal, s_prev, u_prev, N, P, Q, R, u_max, ρ):
    """Solve a single SCP sub-problem for the cart-pole swing-up problem.

    Arguments
    ---------
    f : callable
        A function describing the discrete-time dynamics, such that
        `s[k+1] = f(s[k], u[k])`.
    s0 : numpy.ndarray
        The initial state (1-D).
    s_goal : numpy.ndarray
        The goal state (1-D).
    s_prev : numpy.ndarray
        The state trajectory around which the problem is convexified (2-D).
    u_prev : numpy.ndarray
        The control trajectory around which the problem is convexified (2-D).
    N : int
        The time horizon of the LQR cost function.
    P : numpy.ndarray
        The terminal state cost matrix (2-D).
    Q : numpy.ndarray
        The state stage cost matrix (2-D).
    R : numpy.ndarray
```

```
        The control stage cost matrix (2-D).
    u_max : float
        The bound defining the control set `[-u_max, u_max]`.
    ρ : float
        Trust region radius.

    Returns
    -------
    s : numpy.ndarray
        A 2-D array where `s[k]` is the open-loop state at time step `k`,
        for `k = 0, 1, ..., N-1`
    u : numpy.ndarray
        A 2-D array where `u[k]` is the open-loop state at time step `k`,
        for `k = 0, 1, ..., N-1`
    J : float
        The SCP sub-problem cost.
    """
    A, B, c = affinize(f, s_prev[:-1], u_prev)
    A, B, c = np.array(A), np.array(B), np.array(c)
    n = Q.shape[0]
    m = R.shape[0]
    s_cvx = cvx.Variable((N + 1, n))
    u_cvx = cvx.Variable((N, m))

    # PART (c) ##################################################################
    # INSTRUCTIONS: Construct the convex SCP sub-problem.
    objective = 0.0
    constraints = [s_cvx[0,:] == s0]
    # constraints = []
    for k in range(0,N):

        # objective = objective + (s_cvx[k,:]@Q@s_cvx[k,:].T + u_cvx[k,:]@R@u_cvx[k,:].T)
        objective = objective + cvx.quad_form(s_cvx[k,:]-s_goal,Q) + cvx.quad_form(u_cvx[k,:],R)
        constraints.append(s_cvx[k+1,:] == A[k,:,:]@s_cvx[k,:] + B[k,:,:]@u_cvx[k,:] + c[k,:])
        constraints.append(u_cvx[k,:] >= -u_max)
        constraints.append(u_cvx[k,:] <= u_max)
        constraints.append(cvx.norm(s_cvx[k, :] - s_prev[k, :], p=np.inf) <= ρ)
        constraints.append(cvx.norm(u_cvx[k, :] - u_prev[k, :], p=np.inf) <= ρ)

    # objective = objective + s_cvx[N,:]@P@s_cvx[N,:].T
    objective = objective + cvx.quad_form(s_cvx[N,:]-s_goal,P)
    constraints.append(cvx.norm(s_cvx[N, :] - s_prev[N, :], p=np.inf) <= ρ)

    # raise NotImplementedError()
    # END PART (c) ##############################################################

    prob = cvx.Problem(cvx.Minimize(objective), constraints)
    prob.solve()
```

```python
    if prob.status != "optimal":
        raise RuntimeError("SCP solve failed. Problem status: " + prob.status)
    s = s_cvx.value
    u = u_cvx.value
    J = prob.objective.value
    return s, u, J


def cartpole(s, u):
    """Compute the cart-pole state derivative."""
    mp = 1.0  # pendulum mass
    mc = 4.0  # cart mass
    L = 1.0  # pendulum length
    g = 9.81  # gravitational acceleration

    x, θ, dx, dθ = s
    sinθ, cosθ = jnp.sin(θ), jnp.cos(θ)
    h = mc + mp * (sinθ**2)
    ds = jnp.array(
        [
            dx,
            dθ,
            (mp * sinθ * (L * (dθ**2) + g * cosθ) + u[0]) / h,
            -((mc + mp) * g * sinθ + mp * L * (dθ**2) * sinθ * cosθ + u[0] * cosθ)
            / (h * L),
        ]
    )
    return ds


def discretize(f, dt):
    """Discretize continuous-time dynamics `f` via Runge-Kutta integration."""

    def integrator(s, u, dt=dt):
        k1 = dt * f(s, u)
        k2 = dt * f(s + k1 / 2, u)
        k3 = dt * f(s + k2 / 2, u)
        k4 = dt * f(s + k3, u)
        return s + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    return integrator


# Define constants
n = 4  # state dimension
m = 1  # control dimension
s_goal = np.array([0, np.pi, 0, 0])  # desired upright pendulum state
s0 = np.array([0, 0, 0, 0])  # initial downright pendulum state
dt = 0.1  # discrete time resolution
T = 10.0  # total simulation time
```

```python
P = 1e3 * np.eye(n)  # terminal state cost matrix
Q = np.diag([1e-2, 1.0, 1e-3, 1e-3])  # state cost matrix
R = 1e-3 * np.eye(m)  # control cost matrix
ρ = 1.0  # trust region parameter
u_max = 8.0  # control effort bound
eps = 5e-1  # convergence tolerance
max_iters = 100  # maximum number of SCP iterations
animate = False  # flag for animation

# Initialize the discrete-time dynamics
fd = jax.jit(discretize(cartpole, dt))

# Solve the swing-up problem with SCP
t = np.arange(0.0, T + dt, dt)
N = t.size - 1
s, u, J = solve_swingup_scp(fd, s0, s_goal, N, P, Q, R, u_max, ρ, eps, max_iters)

# Simulate open-loop control
for k in range(N):
    s[k + 1] = fd(s[k], u[k])

# Plot state and control trajectories
fig, ax = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
plt.subplots_adjust(wspace=0.45)
labels_s = (r"$x(t)$", r"$\theta(t)$", r"$\dot{x}(t)$", r"$\dot{\theta}(t)$")
labels_u = (r"$u(t)$",)
for i in range(n):
    ax[i].plot(t, s[:, i])
    ax[i].axhline(s_goal[i], linestyle="--", color="tab:orange")
    ax[i].set_xlabel(r"$t$")
    ax[i].set_ylabel(labels_s[i])
for i in range(m):
    ax[n + i].plot(t[:-1], u[:, i])
    ax[n + i].axhline(u_max, linestyle="--", color="tab:orange")
    ax[n + i].axhline(-u_max, linestyle="--", color="tab:orange")
    ax[n + i].set_xlabel(r"$t$")
    ax[n + i].set_ylabel(labels_u[i])
plt.savefig("cartpole_swingup_constrained.png", bbox_inches="tight")

# Plot cost history over SCP iterations
fig, ax = plt.subplots(1, 1, dpi=150, figsize=(8, 5))
ax.semilogy(J)
ax.set_xlabel(r"SCP iteration $i$")
ax.set_ylabel(r"SCP cost $J(\bar{x}^{(i)}, \bar{u}^{(i)})$")
plt.savefig("cartpole_swingup_constrained_cost.png", bbox_inches="tight")
plt.show()

# Animate the solution
if animate:
    fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
```

```python
    ani.save("cartpole_swingup_constrained.mp4", writer="ffmpeg")
    plt.show()

```