cartpole_swingup.py

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

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

import time

from animations import animate_cartpole

import jax
import jax.numpy as jnp

import matplotlib.pyplot as plt

import numpy as np

from scipy.integrate import odeint


def linearize(f, s, u):
    """Linearize 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 : numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `s`.
    B : numpy.ndarray
        The Jacobian of `f` at `(s, u)`, with respect to `u`.
    """
    # WRITE YOUR CODE BELOW ###################################################
    # INSTRUCTIONS: Use JAX to compute `A` and `B` in one line.
    A,B = jax.jacobian(f,argnums=(0,1))(s,u)
    ###########################################################################
    return A, B


def ilqr(f, s0, s_goal, N, Q, R, QN, eps=1e-3, max_iters=1000):
    """Compute the iLQR set-point tracking solution.
```

```python

    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.
    Q : numpy.ndarray
        The state cost matrix (2-D).
    R : numpy.ndarray
        The control cost matrix (2-D).
    QN : numpy.ndarray
        The terminal state cost matrix (2-D).
    eps : float, optional
        Termination threshold for iLQR.
    max_iters : int, optional
        Maximum number of iLQR iterations.

    Returns
    -------
    s_bar : numpy.ndarray
        A 2-D array where `s_bar[k]` is the nominal state at time step `k`,
        for `k = 0, 1, ..., N-1`
    u_bar : numpy.ndarray
        A 2-D array where `u_bar[k]` is the nominal control at time step `k`,
        for `k = 0, 1, ..., N-1`
    Y : numpy.ndarray
        A 3-D array where `Y[k]` is the matrix gain term of the iLQR control
        law at time step `k`, for `k = 0, 1, ..., N-1`
    y : numpy.ndarray
        A 2-D array where `y[k]` is the offset term of the iLQR control law
        at time step `k`, for `k = 0, 1, ..., N-1`
    """
    if max_iters <= 1:
        raise ValueError("Argument `max_iters` must be at least 1.")
    n = Q.shape[0]  # state dimension
    m = R.shape[0]  # control dimension

    # Initialize gains `Y` and offsets `y` for the policy
    Y = np.zeros((N, m, n))
    y = np.zeros((N, m))

    # Initialize the nominal trajectory `(s_bar, u_bar`), and the
    # deviations `(ds, du)`
    u_bar = np.zeros((N, m))
```

```python
    s_bar = np.zeros((N + 1, n))
    s_bar[0] = s0
    for k in range(N):
        s_bar[k + 1] = f(s_bar[k], u_bar[k])
    ds = np.zeros((N + 1, n))
    du = np.zeros((N, m))

    # iLQR loop
    converged = False
    for _ in range(max_iters):
        # Linearize the dynamics at each step `k` of `(s_bar, u_bar)`
        A, B = jax.vmap(linearize, in_axes=(None, 0, 0))(f, s_bar[:-1], u_bar)
        A, B = np.array(A), np.array(B)

        # PART (c) ############################################################
        # INSTRUCTIONS: Update `Y`, `y`, `ds`, `du`, `s_bar`, and `u_bar`.
        Vxx = QN
        Vx = QN@(s_bar[N,:]-s_goal)
        V = (s_bar[N,:]-s_goal).T@QN@(s_bar[N,:]-s_goal)
        for k in reversed(range(N)):
            Qk = (s_bar[k,:]-s_goal).T@Q@(s_bar[k,:]-s_goal) + (u_bar[k,:]).T@R@(u_bar[k,:]) + V
            Qx = Q@(s_bar[k,:]-s_goal) + A[k,:,:].T@Vx
            Qu = R@(u_bar[k,:]) +B[k,:,:].T@Vx
            Qxx = Q + A[k,:,:].T@Vxx@A[k,:,:]
            Quu = R + B[k,:,:].T@Vxx@B[k,:,:]
            Qux = B[k,:,:].T@Vxx@A[k,:,:]
            Y[k,:,:] = -np.linalg.inv(Quu)@Qux
            y[k,:] = -np.linalg.inv(Quu)@Qu
            # u_bar[k,:] = u_bar[k,:] + du[k,:]
            V = Qk - 1/2*y[k,:].T@Quu@y[k,:]
            Vx = Qx - Y[k,:,:].T@Quu@y[k,:]
            Vxx = Qxx - Y[k,:,:].T@Quu@Y[k,:,:]

        for k in range(N):
            du[k,:] = Y[k,:,:]@(ds[k,:])+y[k,:]
            ds[k+1,:] = f(s_bar[k,:], u_bar[k,:]+du[k,:]) - s_bar[k+1,:]
            s_bar[k+1,:] = s_bar[k+1,:] + ds[k+1,:]
            u_bar[k,:] = u_bar[k,:] + du[k,:]
        ######################################################################

        if np.max(np.abs(du)) < eps:
            converged = True
            break
    if not converged:
        raise RuntimeError("iLQR did not converge!")
    return s_bar, u_bar, Y, y


def cartpole(s, u):
```

```python
    """Compute the cart-pole state derivative."""
    mp = 2.0  # pendulum mass
    mc = 10.0  # 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


# Define constants
n = 4  # state dimension
m = 1  # control dimension
Q = np.diag(np.array([10.0, 10.0, 2.0, 2.0]))  # state cost matrix
R = 1e-2 * np.eye(m)  # control cost matrix
QN = 1e2 * np.eye(n)  # terminal state cost matrix
s0 = np.array([0.0, 0.0, 0.0, 0.0])  # initial state
s_goal = np.array([0.0, np.pi, 0.0, 0.0])  # goal state
T = 10.0  # simulation time
dt = 0.1  # sampling time
animate = False  # flag for animation
closed_loop = True  # flag for closed-loop control

# Initialize continuous-time and discretized dynamics
f = jax.jit(cartpole)
fd = jax.jit(lambda s, u, dt=dt: s + dt * f(s, u))

# Compute the iLQR solution with the discretized dynamics
print("Computing iLQR solution ... ", end="", flush=True)
start = time.time()
t = np.arange(0.0, T, dt)
N = t.size - 1
s_bar, u_bar, Y, y = ilqr(fd, s0, s_goal, N, Q, R, QN)
print("done! ({:.2f} s)".format(time.time() - start), flush=True)

# Simulate on the true continuous-time system
print("Simulating ... ", end="", flush=True)
start = time.time()
s = np.zeros((N + 1, n))
```

```
u = np.zeros((N, m))
s[0] = s0
for k in range(N):
    # PART (d) ################################################################
    # INSTRUCTIONS: Compute either the closed-loop or open-loop value of
    # `u[k]`, depending on the Boolean flag `closed_loop`.
    if closed_loop:
        u[k] = Y[k,:,:]@(s[k,:]-s_bar[k,:])+y[k,:]
        # raise NotImplementedError()
    else:  # do open-loop control
        u[k] = u_bar[k]
        # raise NotImplementedError()
    ###########################################################################
    s[k + 1] = odeint(lambda s, t: f(s, u[k]), s[k], t[k : k + 2])[1]
print("done! ({:.2f} s)".format(time.time() - start), flush=True)

# Plot
fig, axes = 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):
    axes[i].plot(t, s[:, i])
    axes[i].set_xlabel(r"$t$")
    axes[i].set_ylabel(labels_s[i])
for i in range(m):
    axes[n + i].plot(t[:-1], u[:, i])
    axes[n + i].set_xlabel(r"$t$")
    axes[n + i].set_ylabel(labels_u[i])
if closed_loop:
    plt.savefig("cartpole_swingup_cl.png", bbox_inches="tight")
else:
    plt.savefig("cartpole_swingup_ol.png", bbox_inches="tight")
plt.show()

if animate:
    fig, ani = animate_cartpole(t, s[:, 0], s[:, 1])
    ani.save("cartpole_swingup.mp4", writer="ffmpeg")
    plt.show()
```