cartpole_balance.py

```python
"""
Solution code for the problem "Cart-pole balance".

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

import numpy as np
import jax
import jax.numpy as jnp
from scipy.integrate import odeint
import matplotlib.pyplot as plt

from animations import animate_cartpole

# Constants
n = 4  # state dimension
m = 1  # control dimension
mp = 2.0  # pendulum mass
mc = 10.0  # cart mass
L = 1.0  # pendulum length
g = 9.81  # gravitational acceleration
dt = 0.1  # discretization time step
animate = False  # whether or not to animate results


def cartpole(s: np.ndarray, u: np.ndarray) -> np.ndarray:
    """Compute the cart-pole state derivative

    Args:
        s (np.ndarray): The cartpole state: [x, theta, x_dot, theta_dot], shape (n,)
        u (np.ndarray): The cartpole control: [F_x], shape (m,)

    Returns:
        np.ndarray: The state derivative, shape (n,)
    """
    x, θ, dx, dθ = s
    sinθ, cosθ = np.sin(θ), np.cos(θ)
    h = mc + mp * (sinθ**2)
    ds = np.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
```

```python


def reference(t: float) -> np.ndarray:
    """Compute the reference state (s_bar) at time t

    Args:
        t (float): Evaluation time

    Returns:
        np.ndarray: Reference state, shape (n,)
    """
    a = 10.0  # Amplitude
    T = 10.0  # Period

    # PART (d) ###################################################
    # INSTRUCTIONS: Compute the reference state for a given time
    # raise NotImplementedError()
    s_bar = np.array([0,np.pi,0,0])
    s_bar[0] = a*np.sin(2*np.pi*t/T)
    s_bar[2] = a*2*np.pi/T*np.cos(2*np.pi*t/T)
    return s_bar
    # END PART (d) ###############################################


def ricatti_recursion(
    A: np.ndarray, B: np.ndarray, Q: np.ndarray, R: np.ndarray
) -> np.ndarray:
    """Compute the gain matrix K through Ricatti recursion

    Args:
        A (np.ndarray): Dynamics matrix, shape (n, n)
        B (np.ndarray): Controls matrix, shape (n, m)
        Q (np.ndarray): State cost matrix, shape (n, n)
        R (np.ndarray): Control cost matrix, shape (m, m)

    Returns:
        np.ndarray: Gain matrix K, shape (m, n)
    """
    eps = 1e-4  # Riccati recursion convergence tolerance
    max_iters = 1000  # Riccati recursion maximum number of iterations
    P_prev = np.zeros((n, n))  # initialization
    converged = False
    for i in range(max_iters):
        # PART (b) ###################################################
        # INSTRUCTIONS: Apply the Ricatti equation until convergence
        K = -np.linalg.inv(R+B.T@P_prev@B)@B.T@P_prev@A
        if np.max(Q + A.T@P_prev@(A+B@K)-P_prev) <= eps:
            converged = True
            break
        P_prev = Q + A.T@P_prev@(A+B@K)
```

```python
        # raise NotImplementedError()
        # END PART (b) ##############################################
    if not converged:
        raise RuntimeError("Ricatti recursion did not converge!")
    print("K:", K)
    return K


def simulate(
    t: np.ndarray, s_ref: np.ndarray, u_ref: np.ndarray, s0: np.ndarray, K: np.ndarray
) -> tuple[np.ndarray, np.ndarray]:
    """Simulate the cartpole

    Args:
        t (np.ndarray): Evaluation times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t. Shape (num_timesteps, n)
        u_ref (np.ndarray): Reference control u_bar, shape (m,)
        s0 (np.ndarray): Initial state, shape (n,)
        K (np.ndarray): Feedback gain matrix (Ricatti recursion result), shape (m, n)

    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The state history, shape (num_timesteps, n)
            np.ndarray: The control history, shape (num_timesteps, m)
    """

    def cartpole_wrapper(s, tc):
        """Helper function to get cartpole() into a form preferred by odeint, which expects t as the second arg"""
        tind = np.where(t <= tc)[0][np.argmin(np.abs(t[np.where(t <= tc)[0]] - tc))]
        return cartpole(s, K@(s-s_ref[tind,:])+u_ref)

    # PART (c) ##################################################
    # INSTRUCTIONS: Complete the function to simulate the cartpole system
    # Hint: use the cartpole wrapper above with odeint
    # s = NotImplemented
    # u = NotImplemented
    # raise NotImplementedError()
    s = odeint(cartpole_wrapper,s0,t)
    u = np.zeros((len(t),1))
    for k in range(0,len(t)):
        u[k] = K@(s[k,:]-s_ref[k,:])+u_ref
    # END PART (c) ##############################################
    return s, u


def compute_lti_matrices() -> tuple[np.ndarray, np.ndarray]:
    """Compute the linearized dynamics matrices A and B of the LTI system

```

```
    Returns:
        tuple[np.ndarray, np.ndarray]: Tuple of:
            np.ndarray: The A (dynamics) matrix, shape (n, n)
            np.ndarray: The B (controls) matrix, shape (n, m)
    """
    # PART (a) ##################################################
    # INSTRUCTIONS: Construct the A and B matrices
    # dfds,dfdu = jax.jacobian(cartpole,argnums=(0,1))(jnp.array([0.0,np.pi,0.0,0.0],
dtype=jnp.float32),jnp.array([0.0], dtype=jnp.float32))
    A = np.eye(4) + dt*np.array([[0,0,1,0],[0,0,0,1],[0,mp*g/mc,0,0],[0,
(mc+mp)*g/(mc*L),0,0]])
    B = dt*np.array([[0],[0],[1/mc],[1/(mc*L)]])
    # END PART (a) ##############################################
    return A, B


def plot_state_and_control_history(
    s: np.ndarray, u: np.ndarray, t: np.ndarray, s_ref: np.ndarray, name: str
) -> None:
    """Helper function for cartpole visualization

    Args:
        s (np.ndarray): State history, shape (num_timesteps, n)
        u (np.ndarray): Control history, shape (num_timesteps, m)
        t (np.ndarray): Times, shape (num_timesteps,)
        s_ref (np.ndarray): Reference state s_bar, evaluated at each time t. Shape
(num_timesteps, n)
        name (str): Filename prefix for saving figures
    """
    fig, axes = plt.subplots(1, n + m, dpi=150, figsize=(15, 2))
    plt.subplots_adjust(wspace=0.35)
    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].plot(t, s_ref[:, i], "--")
        axes[i].set_xlabel(r"$t$")
        axes[i].set_ylabel(labels_s[i])
    for i in range(m):
        axes[n + i].plot(t, u[:, i])
        axes[n + i].set_xlabel(r"$t$")
        axes[n + i].set_ylabel(labels_u[i])
    plt.savefig(f"{name}.png", bbox_inches="tight")
    plt.show()

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


```

```python
def main():
    # Part A
    A, B = compute_lti_matrices()

    # Part B
    Q = 1*np.eye(n)  # state cost matrix
    # Q= np.diag(np.array([10000,1,10000,1]))
    R = np.eye(m)  # control cost matrix
    K = ricatti_recursion(A, B, Q, R)

    # Part C
    t = np.arange(0.0, 30.0, 1 / 10)
    s_ref = np.array([0.0, np.pi, 0.0, 0.0]) * np.ones((t.size, 1))
    u_ref = np.array([0.0])
    s0 = np.array([0.0, 3 * np.pi / 4, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance")

    # Part D
    # Note: t, u_ref unchanged from part c
    s_ref = np.array([reference(ti) for ti in t])
    s0 = np.array([0.0, np.pi, 0.0, 0.0])
    s, u = simulate(t, s_ref, u_ref, s0, K)
    plot_state_and_control_history(s, u, t, s_ref, "cartpole_balance_tv")


if __name__ == "__main__":
    main()

```