

PSET 4 — Model-base Control

Due date: November 30th, 11:59 pm

This is the last problem set and we will cover topics from LQR control to trajectory optimization and model predictive control. You will learn how to write python code to solve a trajectory when given dynamics using numerical solver.

Problem 1 - Linear system with bounded control (25pt + 10pt bonus)

- 1.1 Prove convexity (5pt)
- 1.2 Discretization of dynamics (5pt)
- 1.3 Numerical solve trajectory optimization (10pt)
- 1.4 Resolution study (5pt)
- 1.5 KKT system (Bonus) (10pt)

Problem 2 - Use linearized dynamics to design a LQR controller (25pt + 5pt bonus)

- 2.1 Lagrangian dynamics (Bonus) (5pt)
- 2.2 State space dynamics (5pt)
- 2.3 Linearize dynamics (5pt)
- 2.4 Simulate error (5pt)
- 2.5 LQR controller (10pt)

Problem 3 - Solve obstacle avoidance problem using CRISP solver (20pt)

- 3.1 Write down the problem (10pt)
- 3.2 Solve the problem using CRISP (10pt)

Problem 4 - Model predictive control on Inverted pendulum (30pt)

- 4.1 Trajectory optimization (10pt)
- 4.2 Fail of open loop control (10pt)
- 4.3 MPC control (10pt)

Problem 1— LQR with bounded control

In class, we studied the unconstrained LQR problem solved via Bellman's equation. In practice, inputs and states are rarely unbounded—actuators saturate and safety limits constrain the state.

For linear systems with convex constraints on inputs and states, the seminal paper [Bemporad et al., 2002](#) analyzes the structure of the optimal cost-to-go (value function) and controller. We consider a finite-horizon, discrete-time linear system with fixed matrices A, B , deterministic dynamics, and a quadratic cost. Inputs are **bounded**.

Given x_0 , horizon N , weights $Q_k \succeq 0$ for $k = 0, \dots, N$ and $R_k \succ 0$ for $k = 0, \dots, N-1$, and a bound $u_{\max} > 0$, consider

$$\begin{aligned} J(x_0) = \min_{u_0, \dots, u_{N-1}} & \sum_{k=0}^N x_k^\top Q_k x_k + \sum_{k=0}^{N-1} u_k^\top R_k u_k \\ \text{s.t. } & x_{k+1} = Ax_k + Bu_k, \quad k = 0, \dots, N-1, \\ & u_k \in [-u_{\max}, u_{\max}] \text{ (componentwise),} \quad k = 0, \dots, N-1. \end{aligned}$$

1.1. Convexity.

Show the problem is a convex optimization problem when x_0 is fixed.

Answer:

Let the decision variables be $z = (x_1, \dots, x_N, u_0, \dots, u_{N-1})$, with x_0 fixed.

- Each term $x_k^\top Q_k x_k$ is convex in x_k because $Q_k \succeq 0$.
- Each term $u_k^\top R_k u_k$ is **strictly** convex in u_k because $R_k \succ 0$.
- The objective

$$f(z) = \sum_{k=0}^N x_k^\top Q_k x_k + \sum_{k=0}^{N-1} u_k^\top R_k u_k$$

is a sum of convex functions, hence convex in z .

Constraints:

- Dynamics $x_{k+1} = Ax_k + Bu_k$ are affine equalities in (x_k, u_k) .
- Bounds $u_k \in [-u_{\max}, u_{\max}]$ are componentwise linear inequalities (a box), hence convex.

Thus the feasible set (intersection of an affine subspace and a box) is convex, and the objective is convex on this set. Therefore, for fixed x_0 , the problem is a convex optimization problem.

1.2. Double integrator discretization

Start from the continuous-time model $\ddot{q} = u$, $u \in [-1, 1]$. Using $x = [q, \dot{q}]^\top$ and a fixed time step dt , derive the discrete system $x_{k+1} = Ax_k + Bu_k$.

Answer:

We have the continuous-time double integrator

$$\ddot{q}(t) = u(t), \quad x(t) = \begin{bmatrix} q(t) \\ \dot{q}(t) \end{bmatrix}$$

Then

$$\dot{x}(t) = \begin{bmatrix} \dot{q}(t) \\ \ddot{q}(t) \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}}_{A_c} x(t) + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{B_c} u(t)$$

Assuming zero-order hold on u over each interval $[k dt, (k + 1) dt]$, the exact-discrete dynamics are

$$A = e^{A_c dt} = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}, \quad B = \int_0^{dt} e^{A_c \tau} B_c d\tau = \begin{bmatrix} \frac{1}{2} dt^2 \\ dt \end{bmatrix}$$

Thus

$$x_{k+1} = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} \frac{1}{2} dt^2 \\ dt \end{bmatrix} u_k$$

1.3 Numerical exploration

With $N = 50$, $dt = 0.1$, and your choice of Q_k , R_k , solve the problem for a grid of initial states x_0 (using cvxpy). Plot $J(x_0)$ and a representative control (e.g., $u_0(x_0)$).

```
In [43]: import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

# =====#
# Problem setup
# =====#

#####
# TODO: Define the hyperparameters
#####
N = 50
dt = 0.1
umax = 1.0

# Euler discretization for the double integrator: x = [q, qdot]
A = np.array([[1.0, dt],
              [0.0, 1.0]])
B = np.array([[0.5 * dt**2],
              [dt]])

# Cost weights (time-invariant)
Q = np.diag([1.0, 0.1])
```

```

R = np.array([[0.1]])

n, m = A.shape[0], B.shape[1]

# =====
# Build one QP (with Parameter x0) that we can reuse for all grid points
# =====

#####
# TODO: Define problem using cvxpy
#####

x = cp.Variable((n, N+1))
u = cp.Variable((m, N))
x0_param = cp.Parameter(n) # initial condition parameter

cost = 0
constraints = [x[:, 0] == x0_param]

for k in range(N):
    cost += cp.quad_form(x[:, k], Q) + cp.quad_form(u[:, k], R)
    constraints += [
        x[:, k+1] == A @ x[:, k] + B @ u[:, k],
        u[:, k] <= umax,
        u[:, k] >= -umax,
    ]

# terminal cost
cost += cp.quad_form(x[:, N], Q)

prob = cp.Problem(cp.Minimize(cost), constraints)

# =====
# Grid of initial states
# =====

#####
# TODO: Plot here
#####

q_vals = np.linspace(-2.0, 2.0, 41)
qd_vals = np.linspace(-2.0, 2.0, 41)
J_vals = np.zeros((len(q_vals), len(qd_vals)))
u0_vals = np.zeros_like(J_vals)

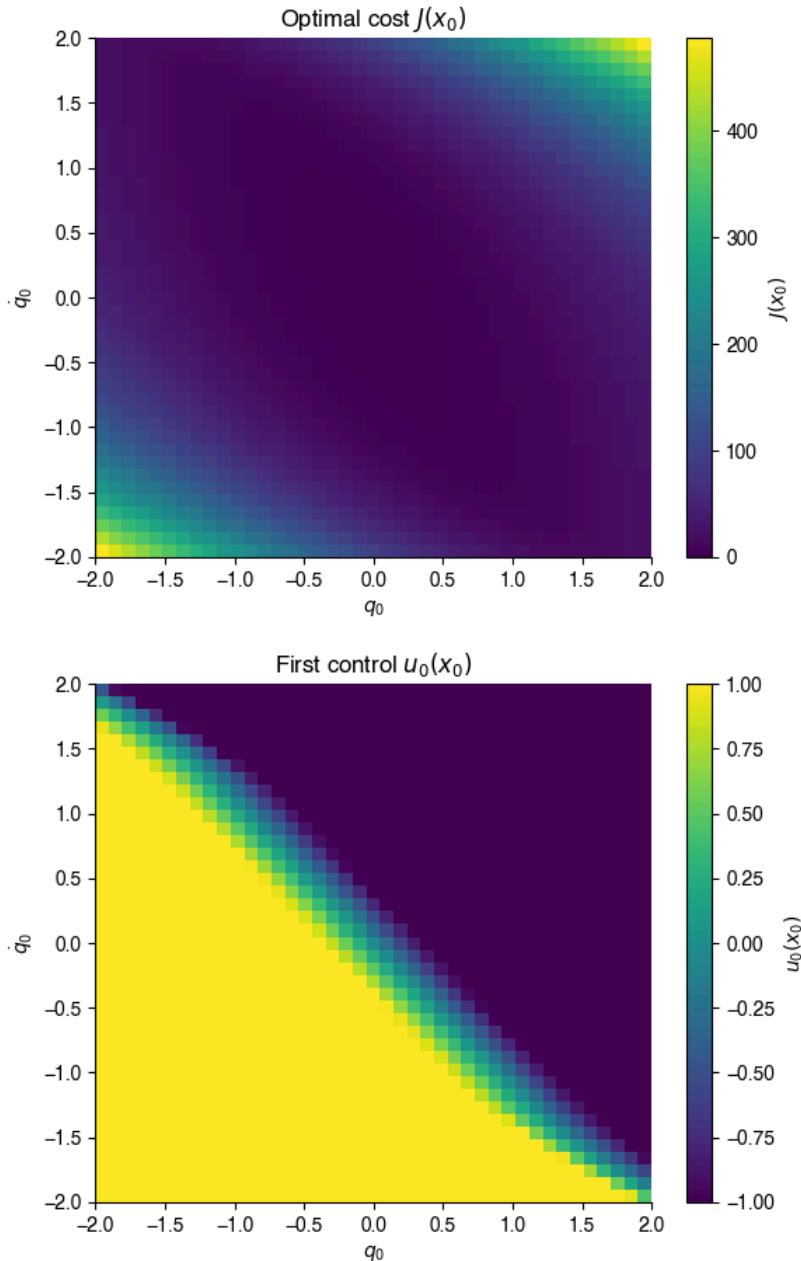
for i, q0 in enumerate(q_vals):
    for j, qd0 in enumerate(qd_vals):
        x0_param.value = np.array([q0, qd0])
        prob.solve(solver=cp.OSQP, warm_start=True)
        J_vals[i, j] = prob.value
        u0_vals[i, j] = u.value[0, 0]

# Plot J(x0)
plt.figure()
plt.imshow(
    J_vals.T,
    extent=[q_vals[0], q_vals[-1], qd_vals[0], qd_vals[-1]],
    origin="lower",
    aspect="auto"
)
plt.colorbar(label="$J(x_0)$")
plt.xlabel("$q_0$")
plt.ylabel("$\dot{q}_0$")
plt.title("Optimal cost $J(x_0)$")

# Plot u0(x0)
plt.figure()
plt.imshow(
    u0_vals.T,
    extent=[q_vals[0], q_vals[-1], qd_vals[0], qd_vals[-1]],
    origin="lower",
    aspect="auto"
)
plt.colorbar(label="$u_0(x_0)$")
plt.xlabel("$q_0$")
plt.ylabel("$\dot{q}_0$")
plt.title("First control $u_0(x_0)$")

plt.show()

```



1.4 Resolution study

Increase N , decrease dt , and plot J, u on a denser grid of initial states. What can you observe from the plots? Comment on solver runtime.

Answer:

-
- Increasing N and decreasing dt (keeping $T = N dt$ fixed) makes the discrete model a better approximation of the continuous double integrator. As a result:
 - The cost surface $J(x_0)$ becomes smoother and more curved near the origin.
 - The control surface $u_0(x_0)$ shows clearer saturated regions ($u = \pm u_{\max}$) and a sharper linear region in the middle.
 - A denser state grid reveals the piecewise-quadratic structure of $J(x_0)$ and the piecewise-affine structure of $u_0(x_0)$ expected from an MPC/QP with box constraints.
 - Solver runtime:
 - Each QP has $O(N)$ variables, so the solve time per QP increases roughly linearly with N .
 - If the initial-state grid has G points, total runtime scales approximately like $N \cdot G$.
 - Thus refining both the horizon and the grid significantly increases overall computation time.
-

1.5 KKT conditions (Bonus)

Write the KKT optimality conditions for the problem.

Answer:

Introduce Lagrange multipliers

- $\lambda_k \in \mathbb{R}^n$ for the dynamics $x_{k+1} = Ax_k + Bu_k$,
- $\alpha_k, \beta_k \in \mathbb{R}^m$ for $u_k - u_{\max} \leq 0$ and $-u_k - u_{\max} \leq 0$ (componentwise).

Primal feasibility

$$x_{k+1} = Ax_k + Bu_k, \quad k = 0, \dots, N-1,$$

$$u_k - u_{\max} \leq 0, \quad -u_k - u_{\max} \leq 0, \quad k = 0, \dots, N-1,$$

with x_0 fixed.

Dual feasibility

$$\alpha_k \geq 0, \quad \beta_k \geq 0, \quad k = 0, \dots, N-1.$$

Stationarity

For $k = 1, \dots, N-1$:

$$\frac{\partial \mathcal{L}}{\partial x_k} : \quad 2Q_k x_k + A^\top \lambda_{k+1} - \lambda_k = 0,$$

terminal state:

$$\frac{\partial \mathcal{L}}{\partial x_N} : \quad 2Q_N x_N - \lambda_N = 0,$$

and for $k = 0, \dots, N-1$:

$$\frac{\partial \mathcal{L}}{\partial u_k} : \quad 2R_k u_k + B^\top \lambda_{k+1} + \alpha_k - \beta_k = 0.$$

Complementary slackness (componentwise)

$$\alpha_{k,i}(u_{k,i} - u_{\max}) = 0, \quad \beta_{k,i}(-u_{k,i} - u_{\max}) = 0,$$

for all $k = 0, \dots, N-1$ and all components i .

Problem 2 – Cart–Pole system (linearization + LQR)

We consider a cart of mass m_c moving on a horizontal track with a pendulum (mass m_p , length l) hinged to the cart. The input f is a horizontal force applied to the cart. We assume no friction and model the pendulum as a point mass at its tip. See the [Video](#).

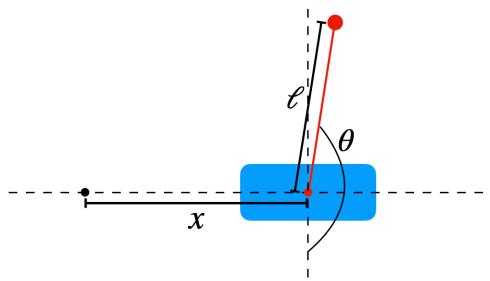


Figure 1. Cartpole.

With this setup, we parameterize the system by two scalars: x denotes the cart position, and θ is the angle between the pole and the stable equilibrium. Our goal is to study the cart–pole dynamics under the horizontal control force f .

2.1 Lagrangian derivation (Bonus)

For those of you who have background in rigid-body dynamics, this is an opportunity for you to apply your knowledge. However, feel free to skip this subproblem and it won't affect the rest of the exercise. Derive the equations of motion:

$$(m_c + m_p) \ddot{x} + m_p l \ddot{\theta} \cos \theta - m_p l \dot{\theta}^2 \sin \theta = f,$$

$$m_p l \ddot{x} \cos \theta + m_p l^2 \ddot{\theta} + m_p g l \sin \theta = 0.$$

(Hints: compute the Lagrangian of the system and the corresponding Lagrangian equations. Analyzing the two objects separately also works.)

Answer:

Let x be the cart position and θ the angle from the downward (stable) vertical.

Kinematics

Cart:

- Position: $(x, 0)$
- Velocity: \dot{x}

Pendulum mass (point mass at tip):

- Position:

$$X_p = x + l \sin \theta, \quad Y_p = l \cos \theta$$

- Velocities:

$$\dot{X}_p = \dot{x} + l \dot{\theta} \cos \theta, \quad \dot{Y}_p = -l \dot{\theta} \sin \theta$$

- Speed squared:

$$v_p^2 = \dot{X}_p^2 + \dot{Y}_p^2 = \dot{x}^2 + 2l\dot{x}\dot{\theta} \cos \theta + l^2 \dot{\theta}^2$$

Energies

Cart kinetic energy:

$$T_c = \frac{1}{2} m_c \dot{x}^2$$

Pendulum kinetic energy:

$$T_p = \frac{1}{2} m_p v_p^2 = \frac{1}{2} m_p (\dot{x}^2 + 2l\dot{x}\dot{\theta} \cos \theta + l^2 \dot{\theta}^2)$$

Total kinetic:

$$T = T_c + T_p = \frac{1}{2} (m_c + m_p) \dot{x}^2 + m_p l \dot{x} \dot{\theta} \cos \theta + \frac{1}{2} m_p l^2 \dot{\theta}^2$$

Choose zero potential at the cart level; the pendulum mass height is $Y_p = l \cos \theta$:

$$V = m_p g l \cos \theta.$$

Lagrangian:

$$\mathcal{L}(x, \theta, \dot{x}, \dot{\theta}) = T - V = \frac{1}{2} (m_c + m_p) \dot{x}^2 + m_p l \dot{x} \dot{\theta} \cos \theta + \frac{1}{2} m_p l^2 \dot{\theta}^2 - m_p g l \cos \theta$$

Generalized forces:

- For x : $Q_x = f$ (horizontal force on cart),
- For θ : $Q_\theta = 0$ (no direct torque at the joint).

Lagrange's equations

1. For x : $\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{x}} \right) = Q_x = f$.

- $\frac{\partial \mathcal{L}}{\partial \dot{x}} = (m_c + m_p) \dot{x} + m_p l \dot{\theta} \cos \theta$

Compute:

$$\frac{\partial \mathcal{L}}{\partial \dot{x}} = (m_c + m_p) \dot{x} + m_p l \dot{\theta} \cos \theta,$$

so

$$\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \ddot{x}} \right) = (m_c + m_p)\ddot{x} + m_p l \ddot{\theta} \cos \theta - m_p l \dot{\theta}^2 \sin \theta$$

Since $\partial \mathcal{L} / \partial x = 0$, the x -equation is

$$(m_c + m_p)\ddot{x} + m_p l \ddot{\theta} \cos \theta - m_p l \dot{\theta}^2 \sin \theta = f$$

2. For θ : $\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}} \right) = (m_c + m_p)\ddot{x} + m_p l \ddot{\theta} \cos \theta - m_p l \dot{\theta}^2 \sin \theta$

- $\frac{\partial \mathcal{L}}{\partial \dot{\theta}} = Q_\theta = 0$

Compute:

$$\frac{\partial \mathcal{L}}{\partial \dot{\theta}} = m_p l \dot{x} \cos \theta + m_p l^2 \dot{\theta},$$

so

$$\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}} \right) = m_p l \ddot{x} \cos \theta - m_p l \dot{x} \dot{\theta} \sin \theta + m_p l^2 \ddot{\theta}$$

Also,

$$\frac{\partial \mathcal{L}}{\partial \theta} = -m_p l \dot{x} \dot{\theta} \sin \theta + m_p g l \sin \theta$$

Thus

$$m_p l \ddot{x} \cos \theta - m_p l \dot{x} \dot{\theta} \sin \theta + m_p l^2 \ddot{\theta} - (-m_p l \dot{x} \dot{\theta} \sin \theta + m_p g l \sin \theta) = 0,$$

which simplifies to

$$m_p l \ddot{x} \cos \theta + m_p l^2 \ddot{\theta} + m_p g l \sin \theta = 0$$

These are exactly the given equations of motion:

$$\begin{aligned} (m_c + m_p)\ddot{x} + m_p l \ddot{\theta} \cos \theta - m_p l \dot{\theta}^2 \sin \theta &= f, \\ m_p l \ddot{x} \cos \theta + m_p l^2 \ddot{\theta} + m_p g l \sin \theta &= 0 \end{aligned}$$

2.2 State-space form.

Let $x_1 = x$, $x_2 = \theta$, $x_3 = \dot{x}$, $x_4 = \dot{\theta}$, and input $u = f$. Translate the equations in 2.1 into the basic state-space dynamics form $\dot{\mathbf{x}} = F(\mathbf{x}, u)$.

Answer:

Let $\mathbf{x} = [x_1, x_2, x_3, x_4]^\top = [x, \theta, \dot{x}, \dot{\theta}]^\top$, input $u = f$.

Then

$$\dot{x}_1 = x_3, \quad \dot{x}_2 = x_4$$

From the equations of motion, solving for \ddot{x} and $\ddot{\theta}$ gives

$$\begin{aligned} \dot{x}_3 &= \ddot{x} = \frac{u + m_p l x_4^2 \sin x_2 + m_p g \sin x_2 \cos x_2}{m_c + m_p \sin^2 x_2}, \\ \dot{x}_4 &= \ddot{\theta} = -\frac{(m_c + m_p)g \sin x_2 + m_p l x_4^2 \sin x_2 \cos x_2 + u \cos x_2}{l(m_c + m_p \sin^2 x_2)} \end{aligned}$$

Thus

$$\dot{\mathbf{x}} = F(\mathbf{x}, u) = \begin{bmatrix} x_3 \\ x_4 \\ \frac{u + m_p l x_4^2 \sin x_2 + m_p g \sin x_2 \cos x_2}{m_c + m_p \sin^2 x_2} \\ -\frac{(m_c + m_p)g \sin x_2 + m_p l x_4^2 \sin x_2 \cos x_2 + u \cos x_2}{l(m_c + m_p \sin^2 x_2)} \end{bmatrix}$$

2.3 Linearize about the upright equilibrium.

Linearize around the unstable equilibrium point $x^* = 0, \dot{x}^* = 0, \theta^* = \pi, \dot{\theta}^* = 0$ (i.e., the pole is in the upright position and the cart stay at zero.) to obtain $\dot{\Delta x} = A \Delta x + B \Delta u$ with $\Delta x = x - x^*, \Delta u = u - u^*$. Write down what is A and B .

Answer:

Define deviations $\Delta x = x - x^*, \Delta u = u - u^*$ with

$$x^* = \begin{bmatrix} 0 \\ \pi \\ 0 \\ 0 \end{bmatrix}, \quad u^* = 0, \quad \Delta x = \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \Delta x_3 \\ \Delta x_4 \end{bmatrix} = \begin{bmatrix} x \\ \theta - \pi \\ \dot{x} \\ \dot{\theta} \end{bmatrix}.$$

Linearizing $\dot{x} = F(x, u)$ about (x^*, u^*) gives

$$\dot{\Delta x} = A \Delta x + B \Delta u,$$

with

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m_p g}{m_c} & 0 & 0 \\ 0 & \frac{(m_c + m_p)g}{lm_c} & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{lm_c} \end{bmatrix}.$$

2.4 Linearization Error

Define the linearization error $e(\mathbf{x}, \mathbf{u}) = \|F(\mathbf{x}, \mathbf{u}) - (A \Delta \mathbf{x} + B \Delta \mathbf{u})\|^2$ (i.e., the error between the true nonlinear system and the linearized system). Simulate both the nonlinear system and its linearization from several initial conditions, and plot how this error evolves over time.

Simulation

A minimal forward-Euler loop is:

```
for i in range(100):
    x = x + dt * F(x,u)
```

But here we use the [Runge-Kutta](#) format.

```
for i in range(100):
    x = RK4(F(as a handle), x, u, dt)
```

See the provided `rk4_step(f, x, u, dt)` function for details.

TODO: Finish the code block.

```
In [44]: # Cart-pole: simulate nonlinear vs. linearized model and plot linearization error.
# Requirements: numpy, matplotlib

import numpy as np
import matplotlib.pyplot as plt

# -----
# Physical parameters (edit if needed)
# -----
```

```

m_c = 1.0    # cart mass [kg]
m_p = 0.1    # pole mass [kg]
l   = 0.5    # pole length [m]
g   = 9.81   # gravity [m/s^2]

# Upright equilibrium (x*, theta*, xdot*, thetadot*, u*)
x_star = np.array([0.0, np.pi, 0.0, 0.0])
u_star = np.array([0.0])

# -----
# Dynamics
# -----
def F(x, u):
    """
    Nonlinear cart-pole dynamics: x=[x, theta, xdot, thetadot]
    returns [xdot, thetadot, xddot, thetaddot]
    """
    x1, th, x3, thd = x
    u = u[0]

    denom = m_c + m_p * np.sin(th)**2

    xdd = (u
            + m_p * l * thd**2 * np.sin(th)
            + m_p * g * np.sin(th) * np.cos(th)) / denom

    thdd = -( (m_c + m_p) * g * np.sin(th)
               + m_p * l * thd**2 * np.sin(th) * np.cos(th)
               + u * np.cos(th) ) / (l * denom)

    return np.array([x3, thd, xdd, thdd])

A = np.array([
    [0, 0, 1, 0],
    [0, 0, 0, 1],
    [0, m_p*g/m_c, 0, 0],
    [0, (m_c+m_p)*g/(l*m_c), 0, 0]
])

B = np.array([
    [0],
    [0],
    [1/m_c],
    [1/(l*m_c)]
])

def F_lin_state(x, u):
    """
    Linearized dynamics about (x_star, u_star):
        xdot ≈ A (x - x_star) + B (u - u_star)
    using the provided A and B matrices.
    """
    #####
    # TODO: Define the linear Dynamics here
    #####
    dx = x - x_star
    du = u - u_star
    return (A @ dx + B @ du).reshape(4,)

# -----
# Integrator (RK4)
# -----
def rk4_step(f, x, u, dt):
    k1 = f(x, u)
    k2 = f(x + 0.5*dt*k1, u)
    k3 = f(x + 0.5*dt*k2, u)
    k4 = f(x + dt*k3, u)
    return x + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

# -----
# Input profiles (edit or add your own)
# -----
def make_u_fun(kind="zero", amp=5.0, t0=0.5, width=0.2, freq=1.0):
    """
    kind: "zero" | "step" | "pulse" | "sine"
    amp:  amplitude (N)
    t0:   step/pulse start (s)

```

```

width:pulse width (s)
freq: sine frequency (Hz)
"""
if kind == "zero":
    return lambda t, x: 0.0
if kind == "step":
    return lambda t, x: (amp if t >= t0 else 0.0)
if kind == "pulse":
    return lambda t, x: (amp if (t0 <= t <= t0+width) else 0.0)
if kind == "sine":
    return lambda t, x: amp*np.sin(2*np.pi*freq*t)
raise ValueError("unknown kind")

# -----
# Simulation
# -----
def simulate_cartpole_error(x0, T=5.0, dt=0.01, u_kind="zero", **u_kwargs):
"""
Simulate nonlinear and linearized models from initial state x0.
Returns dict with time, states, and error e(t).
"""
u_fun = make_u_fun(u_kind, **u_kwargs)

ts = np.arange(0.0, T + dt, dt)
X_nl = np.zeros((len(ts), 4))
X_li = np.zeros((len(ts), 4))
err = np.zeros(len(ts))

x_nl = x0.copy()
x_li = x0.copy()

for k, t in enumerate(ts):
    u = np.array([u_fun(t, x_nl)])

    # store
    X_nl[k] = x_nl
    X_li[k] = x_li

    # linearization error at the *nonlinear* state (definition in the exercise)
    e_vec = F(x_nl, u) - F_lin_state(x_nl, u)
    err[k] = float(np.dot(e_vec, e_vec)) # squared 2-norm

    # advance one step
    x_nl = rk4_step(F, x_nl, u, dt)
    x_li = rk4_step(F_lin_state, x_li, u, dt)

return {"t": ts, "X_nl": X_nl, "X_li": X_li, "e": err}

# -----
# Plotting
# -----
def plot_results(sim):
#####
# TODO: plot result here
#####
t = sim["t"]
e = sim["e"]

plt.figure(figsize=(7,4))
plt.plot(t, e, linewidth=2)
plt.xlabel("Time [s]")
plt.ylabel("Linearization Error $e(t)$")
plt.title("Linearization Error vs Time")
plt.grid(True)
plt.tight_layout()
plt.show()

x0 = np.array([0.0, np.pi + np.deg2rad(2.0), 0.0, 0.0]) # 10° from upright

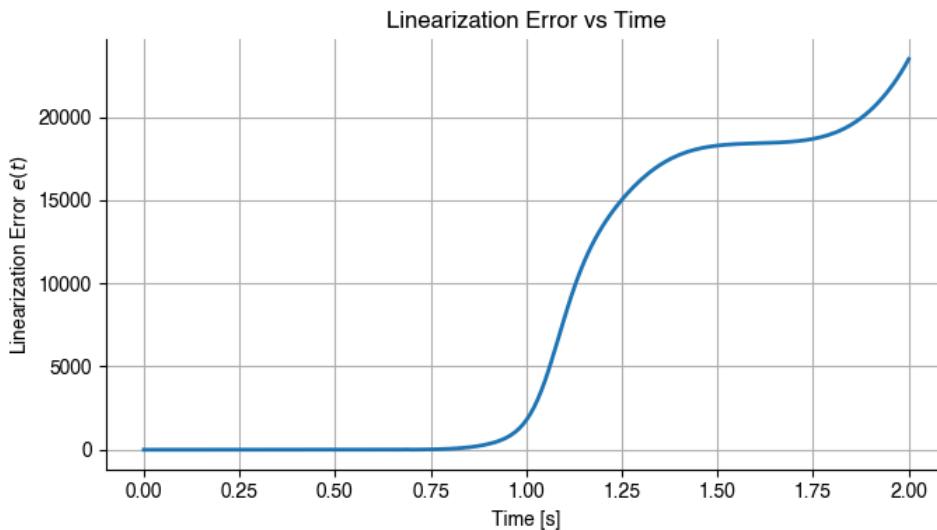
sim = simulate_cartpole_error(
    x0,
    T=2.0,
    dt=0.01,
    u_kind="pulse", # "zero", "step", "pulse", or "sine"
    amp=5.0,
    t0=0.5,
    width=0.2,
)

```

```

        freq=1.0,
    )
plot_results(sim)

```



2.5 Discrete-time model and LQR design.

Convert the linear model in 2.3 to discrete time with step dt , then design a discrete LQR to stabilize the upright equilibrium. Does it work for all initial conditions?

TODO: Get the linear feedback control gain K and simulate. Finish the code block.

In [45]:

```

import numpy as np
import matplotlib.pyplot as plt

from scipy.linalg import solve_discrete_are

# -----
# Parameters
# -----
m_c = 1.0    # cart mass [kg]
m_p = 0.1    # pole mass [kg]
l   = 0.5    # pole length [m]
g   = 9.81   # gravity [m/s^2]

dt = 0.02    # discretization step for LQR and simulation [s]
T  = 6.0     # total simulation time [s]
N  = int(T / dt)

# LQR weights (tune as you like)
Q = np.diag([1.0, 50.0, 0.2, 1.0])  # penalize angle deviation strongly
R = np.array([[0.1]])                 # control effort penalty

u_max = 10.0   # actuator saturation (set to None to disable)

# -----
# Nonlinear dynamics + integrator
# -----
def F_nonlinear(x, u, m_c=1.0, m_p=0.1, l=0.5, g=9.81):
    ##### TODO: Finish the nonlinear dynamics here #####
    # x = [x, theta, xdot, thetadot]
    # ...
    x = [x, theta, xdot, thetadot]
    u = scalar force (N)
    ...
    x_pos, th, xdot, thdot = x
    u = float(u) # ensure scalar

    denom = m_c + m_p * np.sin(th)**2

    xdd = (u
            + m_p * l * thdot**2 * np.sin(th)
            + m_p * g * np.sin(th) * np.cos(th)) / denom

```

```

thdd = -((m_c + m_p) * g * np.sin(th)
         + m_p * l * thdot**2 * np.sin(th) * np.cos(th)
         + u * np.cos(th)) / (l * denom)

return np.array([xdot, thdot, xdd, thdd])

def rk4_step(f, x, u, dt):
    k1 = f(x, u)
    k2 = f(x + 0.5*dt*k1, u)
    k3 = f(x + 0.5*dt*k2, u)
    k4 = f(x + dt*k3, u)
    return x + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)

# -----
# Build controller K, then simulate NONLINEAR plant
# -----
##### TODO: Build the K here
#####

# Continuous-time linearization (from 2.3) about upright
A_c = np.array([
    [0.0, 0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0, 1.0],
    [0.0, m_p*g/m_c, 0.0, 0.0],
    [0.0, (m_c+m_p)*g/(l*m_c), 0.0, 0.0]
])
B_c = np.array([
    [0.0],
    [0.0],
    [1.0/m_c],
    [1.0/(l*m_c)]
])

# Simple forward-Euler discretization
A_d = np.eye(4) + dt * A_c
B_d = dt * B_c

# Solve discrete-time Riccati equation
P = solve_discrete_are(A_d, B_d, Q, R)

# LQR gain: K (1x4)
K = np.linalg.inv(R + B_d.T @ P @ B_d) @ (B_d.T @ P @ A_d)

def simulate_nonlinear_from(x0, K, dt, N, u_max=None):
    """
    x0 is full state: [x, theta, xdot, thetadot].
    Control law: u = -K @ [Delta x, Delta theta, Delta xdot, Delta thetadot], where Delta theta = theta - pi.
    """
    X = np.zeros((N+1, 4))
    U = np.zeros((N, 1))
    X[0] = x0
    for k in range(N):
        dx = np.array([
            X[k, 0],           # Delta x (cart position about 0)
            X[k, 1] - np.pi,   # Delta theta about upright
            X[k, 2],           # Delta xdot
            X[k, 3],           # Delta thetadot
        ])
        u = -K @ dx
        if u_max is not None:
            u = np.clip(u, -u_max, u_max)
        U[k, 0] = u
        X[k+1] = rk4_step(lambda x, uu: F_nonlinear(x, uu, m_c, m_p, l, g), X[k], u, dt)
    return X, U

x0 = np.array([0.0, np.pi + np.deg2rad(30.0), 0.0, 0.0])

X, U = simulate_nonlinear_from(x0, K, dt, N, u_max=u_max)

# -----
# Plot
# -----
t = np.linspace(0.0, T, N+1)
tk = np.linspace(0.0, T, N)

plt.figure(figsize=(9, 4))
plt.plot(t, X[:,1] - np.pi)

```

```

plt.xlabel("time (s)")
plt.ylabel("Δθ = θ - π (rad)")
plt.title("Nonlinear cart-pole under LQR (angle deviation)")
plt.grid(True, alpha=0.3)
plt.show()

plt.figure(figsize=(9, 4))
plt.plot(t, X[:,0])
plt.xlabel("time (s)")
plt.ylabel("x (m)")
plt.title("Cart position (nonlinear plant)")
plt.grid(True, alpha=0.3)
plt.show()

plt.figure(figsize=(9, 4))
plt.step(tk, Ū[:,0], where='post')
plt.xlabel("time (s)")
plt.ylabel("u (N)")
plt.title("Control input (with saturation)" if u_max is not None else "Control input")
plt.grid(True, alpha=0.3)
plt.show()

print("Final angle deviation (deg):", np.rad2deg(X[-1,1] - np.pi))
print("Did it stabilize? (|Δθ| < 2° at end):", abs(np.rad2deg(X[-1,1] - np.pi)) < 2.0)

```

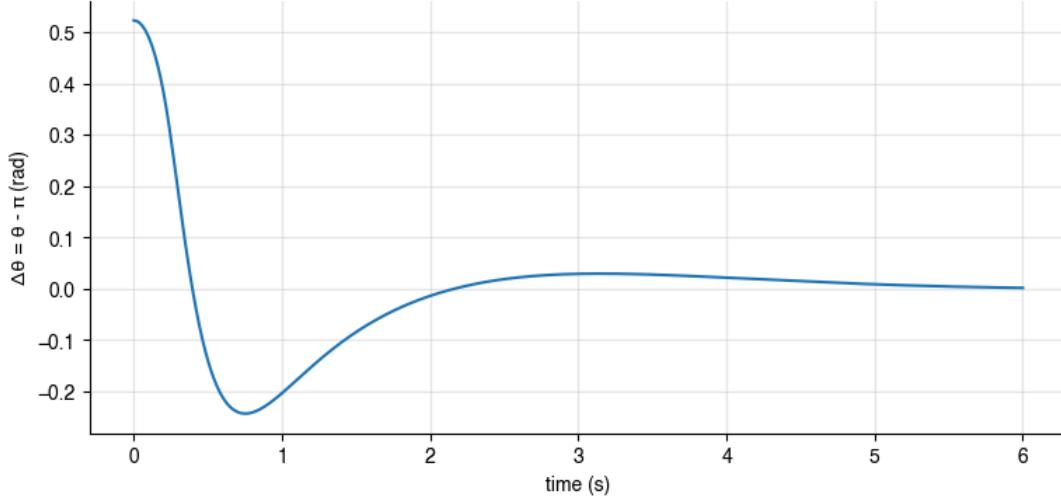
/var/folders/4l/6j6yzwln19z174sxt8p6prxr0000gp/T/ipykernel_339/2472957322.py:109: DeprecationWarning: Conversion of an array with ndim $m > 0$ to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)

U[k, 0] = u

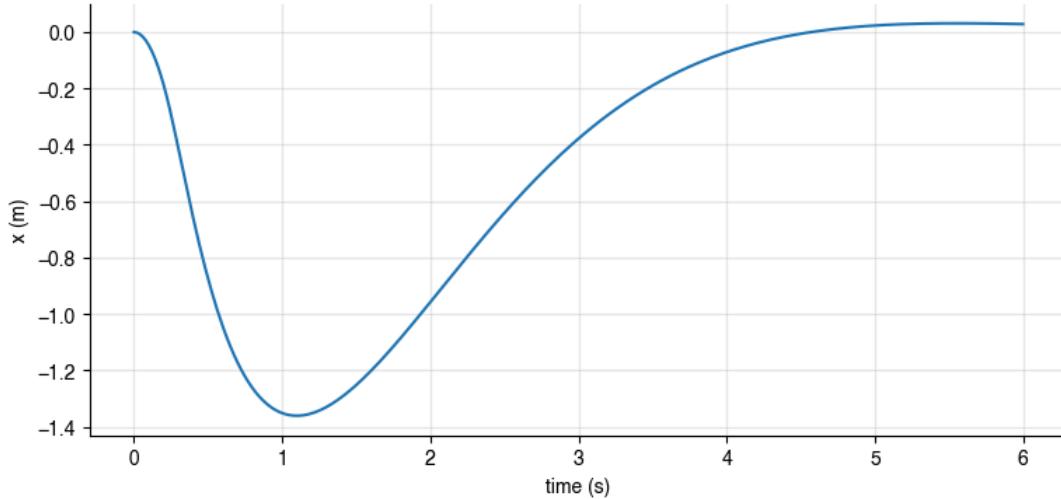
/var/folders/4l/6j6yzwln19z174sxt8p6prxr0000gp/T/ipykernel_339/2472957322.py:38: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)

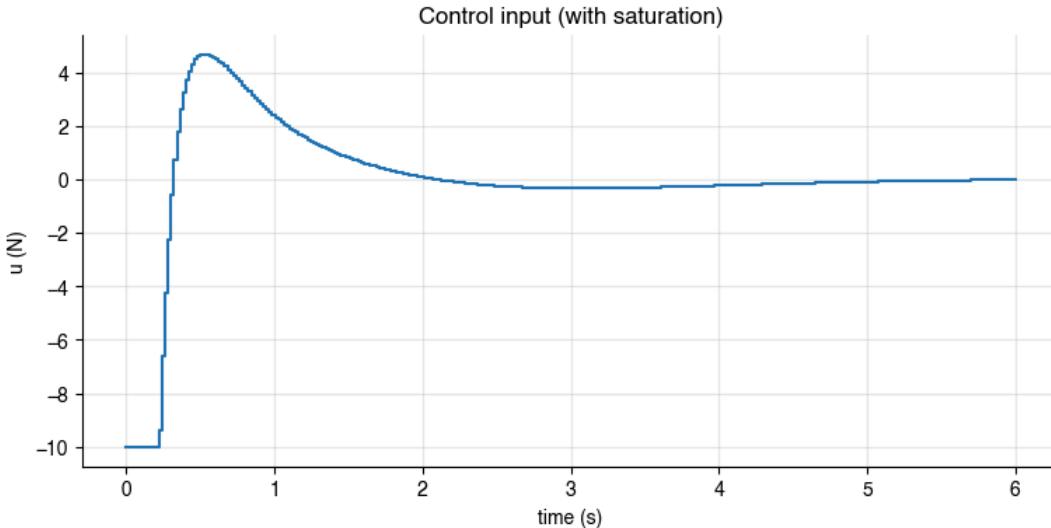
u = float(u) # ensure scalar

Nonlinear cart-pole under LQR (angle deviation)



Cart position (nonlinear plant)





Final angle deviation (deg): 0.10317817538368841
 Did it stabilize? ($|\Delta\theta| < 2^\circ$ at end): True

Problem 3 — Obstacle Avoidance with the CRISP Solver

Adapt Example 4.4 from the lecture notes to implement a trajectory optimization in Python using the CRISP solver. See the paper for details: [CRISP](#). Run the code locally, or copy `crisp.py` into your Google Colab folder.

CRISP need QP solver, for example I use PIQP here. Install the solver using `pip install 'qpsolvers[piqp]'`. You may change to other solvers by looking into CRISP's source code.

We consider a unicycle robot that must travel from a start pose **A** to a goal pose **B** while avoiding circular (disk) obstacles.

System Model

We use standard unicycle (Dubins-like) kinematics in continuous time:

$$\dot{x}(t) = \begin{bmatrix} \dot{p}_x \\ \dot{p}_y \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v(t) \cos \theta(t) \\ v(t) \sin \theta(t) \\ \omega(t) \end{bmatrix}, \quad x = [p_x \ p_y \ \theta]^\top, \ u = [v \ \omega]^\top.$$

Discretize on a uniform grid $t_k = kh$ for $k = 0, \dots, N$ with step $h > 0$ using forward Euler:

$$x_{k+1} = f_h(x_k, u_k) := \begin{bmatrix} p_{x,k} + h v_k \cos \theta_k \\ p_{y,k} + h v_k \sin \theta_k \\ \theta_k + h \omega_k \end{bmatrix}.$$

Decision Variables

Optimize over $\{x_k\}_{k=0}^N$ and $\{u_k\}_{k=0}^{N-1}$, collected as

$$z = [x_0^\top, \ u_0^\top, \ x_1^\top, \ u_1^\top, \ \dots, \ x_{N-1}^\top, \ u_{N-1}^\top, \ x_N^\top]^\top \in \mathbb{R}^{(3+2)N+3}.$$

Constraints

(i) Initial condition.

$$x_0 = A \in \mathbb{R}^3.$$

(ii) Dynamics (equalities), for $k = 0, \dots, N-1$.

$$x_{k+1} - f_h(x_k, u_k) = 0.$$

(iii) Obstacle avoidance (inequalities).

Let $\mathcal{O} = \{(c_j, r_j)\}_{j=1}^{n_{\text{obs}}}$ with centers $c_j = [c_{x,j} \ c_{y,j}]^\top$ and radii $r_j > 0$. Enforce a safety margin $\delta \geq 0$ at each knot:

$$(p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2 \geq (r_j + \delta)^2, \quad \forall k = 0, \dots, N, \ \forall j.$$

(iv) Box limits (controls, optionally states).

$$v_{\min} \leq v_k \leq v_{\max}, \quad \omega_{\min} \leq \omega_k \leq \omega_{\max}, \quad k = 0, \dots, N-1.$$

4.1 Write down your optimization problem

According to the dynamics and the constraints, design the loss function yourself and write down the problem formulation as a mathematical optimization problem.

Answer:

Let the decision variables be $\{x_k\}_{k=0}^N, \{u_k\}_{k=0}^{N-1}$ with

$$x_k = \begin{bmatrix} p_{x,k} \\ p_{y,k} \\ \theta_k \end{bmatrix}, \quad u_k = \begin{bmatrix} v_k \\ \omega_k \end{bmatrix}$$

Given start $A \in \mathbb{R}^3$, goal $B \in \mathbb{R}^3$, step h , obstacles $\mathcal{O} = \{(c_j, r_j)\}_{j=1}^{n_{\text{obs}}}$ and safety margin δ , consider for example the quadratic cost

$$J = \sum_{k=0}^{N-1} \left(\|x_k - B\|_Q^2 + \|u_k\|_R^2 \right) + \|x_N - B\|_{Q_f}^2,$$

where $Q, Q_f \succeq 0, R \succ 0$.

The trajectory optimization problem is

$$\begin{aligned} \min_{\{x_k, u_k\}} \quad & \sum_{k=0}^{N-1} \left((x_k - B)^\top Q (x_k - B) + u_k^\top R u_k \right) + (x_N - B)^\top Q_f (x_N - B) \\ \text{s.t.} \quad & x_0 = A, \\ & x_{k+1} = \begin{bmatrix} p_{x,k} + h v_k \cos \theta_k \\ p_{y,k} + h v_k \sin \theta_k \\ \theta_k + h \omega_k \end{bmatrix}, \quad k = 0, \dots, N-1, \\ & (p_{x,k} - c_{x,j})^2 + (p_{y,k} - c_{y,j})^2 \geq (r_j + \delta)^2, \\ & \quad k = 0, \dots, N, \quad j = 1, \dots, n_{\text{obs}}, \\ & v_{\min} \leq v_k \leq v_{\max}, \quad \omega_{\min} \leq \omega_k \leq \omega_{\max}, \quad k = 0, \dots, N-1 \end{aligned}$$

4.2 Program your problem and solve it using CRISP

Crisp's interface allow you to solve the problem

$$\min_x f(x) \tag{1}$$

$$s.t. \quad h(x) = 0 \tag{2}$$

$$g(x) \geq 0 \tag{3}$$

When given the cost function f , its gradient and hessian $\nabla f, \nabla^2 f$, the equality and inequality constraint $h(x), g(x)$, and the jacobian of the constraint $\frac{\partial h}{\partial x}$ and $\frac{\partial g}{\partial x}$.

The code below provides a simple example:

```
In [46]: from crisp import SQPSolver
import numpy as np

def cost(x):
    return x.dot(x)

def grad_cost(x):
    return 2*x

def hess_cost(x):
    return 2*np.eye(x.size)

def eq_cons(x):
    return np.array([x[0] + x[1] - 1,
                    x[2] - x[3] + 1])

def jac_eq(x):
    return np.vstack([
```

```

        np.array([1,1,0,0,0]),
        np.array([0,0,1,-1,0])
    ])

def ineq_cons(x):
    return np.array([x[2], x[3]])

def jac_in(x):
    return np.vstack([
        np.array([0,0,1,0,0]),
        np.array([0,0,0,1,0])
    ])

x0 = np.zeros(5)

solver = SQPSolver(cost, grad_cost, hess_cost,
                    eq_cons, jac_eq,
                    ineq_cons, jac_in,
                    x0,
                    mu_eq_init=np.ones(2),
                    mu_in_init=np.ones(2),
                    max_iter=100)
x_opt, history = solver.solve()
print("Optimal x:", x_opt)

TR      predicted = 0.750000 , actual = 0.750000 , cost = 0.000000 , merit = 2.000000 , eq_vio = 1.000000 , in_vio = 0.000000 , QP_time = 0.000145s
REJ     predicted = 0.000000 , actual = 0.000000 , cost = 0.749971 , merit = 1.250000 , eq_vio = 0.499978 , in_vio = 0.000022 , QP_time = 0.000093s
Increase lower than tolerance at iter 1
TR+    predicted = 2.187760 , actual = 2.187760 , cost = 0.749971 , merit = 5.750260 , eq_vio = 0.499978 , in_vio = 0.000022 , QP_time = 0.000095s
TR      predicted = 2.062500 , actual = 2.062500 , cost = 1.062500 , merit = 3.562500 , eq_vio = 0.249949 , in_vio = 0.000051 , QP_time = 0.000084s
REJ     predicted = 0.000000 , actual = 0.000000 , cost = 1.500000 , merit = 1.500000 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_time = 0.000081s
Solved at iter 4
Optimal x: [ 5.0000000e-01  5.0000000e-01  1.49686849e-10  1.00000000e+00
 -1.67234840e-24]

```

Then your task is to implement the obstacle avoidance problem using CRISP. The code structure is given, but you need to fill in objective, constraints and its derivative.

TODO: Finish `TODO` functions.

```
In [47]: from __future__ import annotations

import math
from dataclasses import dataclass
from typing import Tuple

import numpy as np
from scipy import sparse as sp
import matplotlib.pyplot as plt

from crisp import SQPSolver

# ----- Problem parameters -----
@dataclass
class Params:
    N: int = 60
    h: float = 0.1
    A: np.ndarray = np.array([0.0, 0.0, 0.0])
    B: np.ndarray = np.array([6.0, 5.0, 0.0])
    obstacles: np.ndarray = np.array([
        [2.0, 1.5, 0.8],
        [3.5, 3.5, 0.9],
        [5.2, 3.8, 0.7],
        [2.2, 3.2, 0.3],
        [4.5, 1.0, 1.0],
        [5.0, 2.5, 0.05],
    ])
    safety_margin: float = 0.25

    v_min: float = -2.0
    v_max: float = 2.0
    w_min: float = -2.5
```

```

w_max: float = 2.5
w_goal_pos: float = 20.0
w_goal_theta: float = 20.0
w_u: float = 0.05
w_du: float = 0.2

n_x: int = 3
n_u: int = 2

def __post_init__(self):
    self.Z_DIM = (self.n_x + self.n_u) * self.N + self.n_x

# ----- Index helpers -----
def idx_x(k: int, p: Params) -> slice:
    base = (p.n_x + p.n_u) * k
    return slice(base, base + p.n_x)

def idx_u(k: int, p: Params) -> slice:
    base = (p.n_x + p.n_u) * k + p.n_x
    return slice(base, base + p.n_u)

# ----- Pack / Unpack -----
def pack(X: np.ndarray, U: np.ndarray, p: Params) -> np.ndarray:
    z = np.zeros(p.Z_DIM, dtype=float)
    for kk in range(p.N):
        z[idx_x(kk, p)] = X[kk, :]
        z[idx_u(kk, p)] = U[kk, :]
    z[idx_x(p.N, p)] = X[p.N, :]
    return z

def unpack(z: np.ndarray, p: Params) -> Tuple[np.ndarray, np.ndarray]:
    X = np.zeros((p.N + 1, p.n_x), dtype=float)
    U = np.zeros((p.N, p.n_u), dtype=float)
    for kk in range(p.N):
        X[kk, :] = z[idx_x(kk, p)]
        U[kk, :] = z[idx_u(kk, p)]
    X[p.N, :] = z[idx_x(p.N, p)]
    return X, U

# ----- Dynamics -----
def f_disc(xk: np.ndarray, uk: np.ndarray, h: float) -> np.ndarray:
    px, py, th = xk
    v, w = uk
    return np.array([
        px + h * v * math.cos(th),
        py + h * v * math.sin(th),
        th + h * w,
    ])

# ----- Objective & derivatives -----
def objective(z: np.ndarray, p: Params) -> float:
    """
    Cost:
    - terminal goal cost on position and heading
    - quadratic control effort over all stages
    - quadratic control variation ( $u_{k+1} - u_k$ )
    """
    X, U = unpack(z, p)

    # terminal goal cost
    pos_err = X[-1, 0:2] - p.B[0:2]
    th_err = X[-1, 2] - p.B[2]
    J = p.w_goal_pos * pos_err @ pos_err + p.w_goal_theta * (th_err**2)

    # control effort
    for k in range(p.N):
        J += p.w_u * (U[k] @ U[k])

    # control smoothness
    for k in range(p.N - 1):
        du = U[k+1] - U[k]
        J += p.w_du * (du @ du)

    return float(J)

```

```

def grad_cost(z: np.ndarray, p: Params) -> np.ndarray:
    g = np.zeros_like(z)
    X, U = unpack(z, p)

    # terminal state gradient
    idxN = idx_x(p.N, p)
    baseN = idxN.start

    pos_err = X[-1, 0:2] - p.B[0:2]
    th_err = X[-1, 2] - p.B[2]

    g[baseN + 0] += 2.0 * p.w_goal_pos * pos_err[0]
    g[baseN + 1] += 2.0 * p.w_goal_pos * pos_err[1]
    g[baseN + 2] += 2.0 * p.w_goal_theta * th_err

    # control effort gradient
    for k in range(p.N):
        s_u = idx_u(k, p)
        g[s_u] += 2.0 * p.w_u * U[k]

    # control smoothness gradient
    for k in range(p.N - 1):
        du = U[k+1] - U[k]
        s_uk = idx_u(k, p)
        s_uk1 = idx_u(k+1, p)
        g[s_uk] += -2.0 * p.w_du * du
        g[s_uk1] += 2.0 * p.w_du * du

    return g

def hess_cost(z: np.ndarray, p: Params):
    """
    Constant Hessian (quadratic cost) - return sparse CSR.
    """
    H = sp.lil_matrix((p.Z_DIM, p.Z_DIM))

    # terminal state Hessian
    idxN = idx_x(p.N, p)
    baseN = idxN.start
    H[baseN + 0, baseN + 0] += 2.0 * p.w_goal_pos
    H[baseN + 1, baseN + 1] += 2.0 * p.w_goal_pos
    H[baseN + 2, baseN + 2] += 2.0 * p.w_goal_theta

    # control effort Hessian
    for k in range(p.N):
        base = idx_u(k, p).start
        for j in range(p.n_u):
            H[base + j, base + j] += 2.0 * p.w_u

    # control smoothness Hessian:
    # w_du * ||u_{k+1}-u_k||^2
    for k in range(p.N - 1):
        base_k = idx_u(k, p).start
        base_k1 = idx_u(k+1, p).start
        for j in range(p.n_u):
            a = base_k + j
            b = base_k1 + j
            H[a, a] += 2.0 * p.w_du
            H[b, b] += 2.0 * p.w_du
            H[a, b] += -2.0 * p.w_du
            H[b, a] += -2.0 * p.w_du

    return H.tocsr()

# ----- Constraints -----
def eq_cons(z: np.ndarray, p: Params) -> np.ndarray:
    """
    Equality constraints:
    - x_0 = A (3)
    - x_{k+1} - f_disc(x_k, u_k) = 0, k = 0..N-1 (3N)
    Total: 3 + 3N entries.
    """
    X, U = unpack(z, p)
    cons = []

```

```

# initial condition
cons.append(X[0] - p.A)

# dynamics
for k in range(p.N):
    cons.append(X[k+1] - f_disc(X[k], U[k], p.h))

return np.concatenate(cons)

def jac_eq(z: np.ndarray, p: Params):
    """
    Sparse Jacobian of equality constraints.
    """
    X, U = unpack(z, p)
    m_eq = 3 + 3 * p.N
    J = sp.lil_matrix((m_eq, p.Z_DIM))

    row = 0

    # initial condition: x0 - A = 0 -> d/dx0 = I
    s_x0 = idx_x(0, p)
    for i in range(p.n_x):
        J[row + i, s_x0.start + i] = 1.0
    row += 3

    # dynamics constraints
    for k in range(p.N):
        xk = X[k]
        uk = U[k]
        px, py, th = xk
        v, w = uk

        cth = math.cos(th)
        sth = math.sin(th)

        # Fx = df/dxk
        Fx = np.array([
            [1.0, 0.0, -p.h * v * sth],
            [0.0, 1.0, p.h * v * cth],
            [0.0, 0.0, 1.0]
        ])
        # Fu = df/duk
        Fu = np.array([
            [p.h * cth, 0.0],
            [p.h * sth, 0.0],
            [0.0, p.h]
        ])

        s_xk = idx_x(k, p)
        s_xkp = idx_x(k+1, p)
        s_uk = idx_u(k, p)

        # derivative wrt x_{k+1}: I
        for i in range(3):
            J[row + i, s_xkp.start + i] = 1.0

        # derivative wrt x_k: -Fx
        for i in range(3):
            for j in range(3):
                if Fx[i, j] != 0.0:
                    J[row + i, s_xk.start + j] += -Fx[i, j]

        # derivative wrt u_k: -Fu
        for i in range(3):
            for j in range(2):
                if Fu[i, j] != 0.0:
                    J[row + i, s_uk.start + j] += -Fu[i, j]

        row += 3

    return J.tocsr()

def ineq_cons(z: np.ndarray, p: Params) -> np.ndarray:
    """
    Obstacle clearance + control box constraints, all as g(z) >= 0:
    - For each k,j: (px_k - cx_j)^2 + (py_k - cy_j)^2 - (r_j + delta)^2 >= 0
    - For each k: v_max - v_k >= 0, v_k - v_min >= 0,

```

```

        w_max - w_k >= 0,  w_k - w_min >= 0
"""
X, U = unpack(z, p)
n_obs = p.obstacles.shape[0]

vals = []

# obstacle constraints
for k in range(p.N + 1):
    px, py, th = X[k]
    for j in range(n_obs):
        cx, cy, r = p.obstacles[j]
        r_eff = r + p.safety_margin
        vals.append((px - cx)**2 + (py - cy)**2 - r_eff**2)

# control bounds
for k in range(p.N):
    v, w = U[k]
    vals.append(p.v_max - v)    # v <= v_max
    vals.append(v - p.v_min)    # v >= v_min
    vals.append(p.w_max - w)    # w <= w_max
    vals.append(w - p.w_min)    # w >= w_min

return np.array(vals, dtype=float)

def jac_in(z: np.ndarray, p: Params):
    """
    Sparse Jacobian of ineq_cons matching the order in ineq_cons.
    """
    X, U = unpack(z, p)
    n_obs = p.obstacles.shape[0]
    m_in = (p.N + 1) * n_obs + 4 * p.N

    J = sp.lil_matrix((m_in, p.Z_DIM))

    row = 0

    # obstacle gradients
    for k in range(p.N + 1):
        px, py, th = X[k]
        s_xk = idx_x(k, p)
        for j in range(n_obs):
            cx, cy, r = p.obstacles[j]
            dpx = 2.0 * (px - cx)
            dpy = 2.0 * (py - cy)
            J[row, s_xk.start + 0] = dpx
            J[row, s_xk.start + 1] = dpy
            # no dependence on theta or controls
        row += 1

    # control bounds gradients
    for k in range(p.N):
        s_uk = idx_u(k, p)
        # v_max - v_k >= 0 -> grad wrt v_k is -1
        J[row, s_uk.start + 0] = -1.0
        row += 1
        # v_k - v_min >= 0 -> grad wrt v_k is +1
        J[row, s_uk.start + 0] = 1.0
        row += 1
        # w_max - w_k >= 0 -> grad wrt w_k is -1
        J[row, s_uk.start + 1] = -1.0
        row += 1
        # w_k - w_min >= 0 -> grad wrt w_k is +1
        J[row, s_uk.start + 1] = 1.0
        row += 1

    return J.tocsr()

# ----- Plotting -----
def plot_results(p: Params, X: np.ndarray, U: np.ndarray) -> None:
    blue = (0.0, 0.4470, 0.7410)

    fig = plt.figure(figsize=(8, 6.5), dpi=120)
    ax = fig.add_subplot(111)
    ax.set_title("Unicycle T0 (Custom SQP + Control Bounds)")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.grid(True)
    ax.set_aspect("equal", adjustable="box")

```

```

ang = np.linspace(0, 2 * math.pi, 200)
for j in range(p.obstacles.shape[0]):
    cx, cy, r = p.obstacles[j]
    r_eff = r + p.safety_margin
    ox = cx + r_eff * np.cos(ang)
    oy = cy + r_eff * np.sin(ang)
    ax.plot(ox, oy, "r-", linewidth=1.2)
    ax.fill(ox, oy, color=(1, 0, 0), alpha=0.12, edgecolor="none")

ax.plot(p.A[0], p.A[1], "go", label="start")
ax.plot(p.B[0], p.B[1], "b*", markersize=10, label="goal")

ax.plot(X[:, 0], X[:, 1], color=blue, linewidth=2.0, label="optimized")

skip = max(1, p.N // 20)
for k in range(0, p.N + 1, skip):
    px, py, th = X[k]
    ax.quiver(px, py, 0.4 * math.cos(th), 0.4 * math.sin(th),
              angles='xy', scale_units='xy', scale=1, color=blue, width=0.004)

ax.legend(loc="best")

t = np.arange(p.N) * p.h
fig2 = plt.figure(figsize=(8.5, 5.0), dpi=120)

ax1 = fig2.add_subplot(211)
ax1.plot(t, U[:, 0], color=blue, label="v*")
ax1.axhline(p.v_min, linestyle=":", color="k")
ax1.axhline(p.v_max, linestyle=":", color="k")
ax1.set_ylabel("v [m/s]")
ax1.grid(True)
ax1.legend()

ax2 = fig2.add_subplot(212)
ax2.plot(t, U[:, 1], color=blue, label="omega*")
ax2.axhline(p.w_min, linestyle=":", color="k")
ax2.axhline(p.w_max, linestyle=":", color="k")
ax2.set_xlabel("time [s]")
ax2.set_ylabel("omega [rad/s]")
ax2.grid(True)
ax2.legend()

plt.tight_layout()
plt.show()

p = Params()
X0, U0 = np.zeros((p.N + 1, p.n_x)), np.zeros((p.N, p.n_u))
z0 = pack(X0, U0, p)

cost_fn = lambda z: objective(z, p)
grad_fn = lambda z: grad_cost(z, p)
hess_fn = lambda z: hess_cost(z, p)
eq_fn = lambda z: eq_cons(z, p)
jeq_fn = lambda z: jac_eq(z, p)
in_fn = lambda z: ineq_cons(z, p)
jin_fn = lambda z: jac_in(z, p)

m_eq = 3 + 3 * p.N
m_in = (p.N + 1) * p.obstacles.shape[0] + 4 * p.N # add 4 bounds per stage
mu_eq_init = 1.0 * np.ones(m_eq)
mu_in_init = 1.0 * np.ones(m_in)

solver = SQPSolver(cost_fn, grad_fn, hess_fn,
                    eq_fn, jeq_fn,
                    in_fn, jin_fn,
                    x0=z0,
                    mu_eq_init=mu_eq_init,
                    mu_in_init=mu_in_init,
                    delta_init=0.5,
                    delta_max=2.0,
                    tol=1e-6,
                    max_iter=2000)

z_star, hist = solver.solve()

X_star, U_star = unpack(z_star, p)
print("\nTerminal state [px py theta]^T = [" +
      f"{X_star[-1,0]:.3f} {X_star[-1,1]:.3f} {X_star[-1,2]:.3f}]")

```

```
plot_results(p, X_star, U_star)
```

TR+ predicted = 209.479167 , actual = 209.479167 , cost = 1220.000000 , merit = 1220.000000 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_time = 0.001279s
 TR+ predicted = 356.043274 , actual = 357.749725 , cost = 1010.020833 , merit = 1010.520833 , eq_vio = 0.053831 , in_vio = 0.000000 , QP_time = 0.001535s
 TR+ predicted = 446.189809 , actual = 475.811657 , cost = 650.248311 , merit = 652.771108 , eq_vio = 1.376244 , in_vio = 0.852500 , QP_time = 0.001648s
 TR+ predicted = 157.563776 , actual = 163.817354 , cost = 170.749444 , merit = 176.959451 , eq_vio = 3.535106 , in_vio = 1.322500 , QP_time = 0.001502s
 TR predicted = 5.210720 , actual = 5.173338 , cost = 5.832165 , merit = 13.142097 , eq_vio = 4.311064 , in_vio = 0.000000 , QP_time = 0.001786s
 TR predicted = 0.055571 , actual = 0.040526 , cost = 1.068005 , merit = 7.968760 , eq_vio = 4.216504 , in_vio = 0.000000 , QP_time = 0.001870s
 TR predicted = 0.012927 , actual = 0.011919 , cost = 1.142164 , merit = 7.928234 , eq_vio = 4.116094 , in_vio = 0.000000 , QP_time = 0.001707s
 TR predicted = 0.000805 , actual = 0.000719 , cost = 1.133511 , merit = 7.916315 , eq_vio = 4.131968 , in_vio = 0.000000 , QP_time = 0.001560s
 TR predicted = 0.000069 , actual = 0.000063 , cost = 1.138810 , merit = 7.915595 , eq_vio = 4.126345 , in_vio = 0.000000 , QP_time = 0.001521s
 TR predicted = 0.000005 , actual = 0.000004 , cost = 1.137694 , merit = 7.915533 , eq_vio = 4.127539 , in_vio = 0.000000 , QP_time = 0.001498s
 TR predicted = 0.000000 , actual = 0.000000 , cost = 1.138007 , merit = 7.915528 , eq_vio = 4.127204 , in_vio = 0.000000 , QP_time = 0.001590s
 Increase lower than tolerance at iter 10
 TR predicted = 6.557732 , actual = 2.984760 , cost = 1.137927 , merit = 68.913935 , eq_vio = 4.127285 , in_vio = 0.000000 , QP_time = 0.001855s
 TR predicted = 13.344734 , actual = 10.375904 , cost = 5.516250 , merit = 65.929175 , eq_vio = 3.301220 , in_vio = 0.000000 , QP_time = 0.002011s
 TR predicted = 11.933477 , actual = 8.008191 , cost = 6.416066 , merit = 55.553271 , eq_vio = 2.588833 , in_vio = 0.000000 , QP_time = 0.002222s
 TR predicted = 9.786803 , actual = 8.418291 , cost = 5.454355 , merit = 47.545080 , eq_vio = 2.372618 , in_vio = 0.000000 , QP_time = 0.002055s
 TR predicted = 6.240873 , actual = 5.956750 , cost = 6.421464 , merit = 39.126790 , eq_vio = 2.039198 , in_vio = 0.000000 , QP_time = 0.001843s
 TR predicted = 2.112967 , actual = 1.605584 , cost = 6.654079 , merit = 33.170040 , eq_vio = 1.802272 , in_vio = 0.000000 , QP_time = 0.001965s
 TR predicted = 1.693644 , actual = 1.703318 , cost = 6.446620 , merit = 31.564456 , eq_vio = 1.852389 , in_vio = 0.000000 , QP_time = 0.002289s
 TR predicted = 1.710562 , actual = 1.839779 , cost = 6.527128 , merit = 29.861138 , eq_vio = 1.896400 , in_vio = 0.000957 , QP_time = 0.001927s
 TR predicted = 2.025355 , actual = 2.234169 , cost = 6.853720 , merit = 28.021359 , eq_vio = 1.902882 , in_vio = 0.000000 , QP_time = 0.002076s
 TR predicted = 1.340793 , actual = 1.351907 , cost = 6.929372 , merit = 25.787190 , eq_vio = 1.849655 , in_vio = 0.000000 , QP_time = 0.001905s
 TR predicted = 0.391986 , actual = 0.390816 , cost = 6.329615 , merit = 24.435283 , eq_vio = 1.757921 , in_vio = 0.015098 , QP_time = 0.001961s
 TR predicted = 0.082434 , actual = 0.076127 , cost = 6.406107 , merit = 24.044467 , eq_vio = 1.716600 , in_vio = 0.017974 , QP_time = 0.001894s
 TR predicted = 0.091206 , actual = 0.085517 , cost = 6.423412 , merit = 23.968340 , eq_vio = 1.703355 , in_vio = 0.004083 , QP_time = 0.001992s
 TR predicted = 0.046937 , actual = 0.041629 , cost = 6.485674 , merit = 23.882823 , eq_vio = 1.687037 , in_vio = 0.000000 , QP_time = 0.001910s
 TR predicted = 0.079075 , actual = 0.100113 , cost = 6.447998 , merit = 23.841195 , eq_vio = 1.671240 , in_vio = 0.020598 , QP_time = 0.001902s
 TR predicted = 0.096119 , actual = 0.098072 , cost = 6.440668 , merit = 23.741082 , eq_vio = 1.648198 , in_vio = 0.011005 , QP_time = 0.002339s
 TR predicted = 0.052196 , actual = 0.042529 , cost = 6.507208 , merit = 23.643010 , eq_vio = 1.638327 , in_vio = 0.001813 , QP_time = 0.001910s
 TR predicted = 0.029615 , actual = 0.027561 , cost = 6.560981 , merit = 23.600481 , eq_vio = 1.628930 , in_vio = 0.000000 , QP_time = 0.001939s
 TR predicted = 0.060353 , actual = 0.051742 , cost = 6.524333 , merit = 23.572920 , eq_vio = 1.610479 , in_vio = 0.016834 , QP_time = 0.001965s
 TR predicted = 0.011615 , actual = 0.011464 , cost = 6.534695 , merit = 23.521178 , eq_vio = 1.597874 , in_vio = 0.021998 , QP_time = 0.002190s
 TR predicted = 0.000453 , actual = 0.000400 , cost = 6.550821 , merit = 23.509714 , eq_vio = 1.596287 , in_vio = 0.022461 , QP_time = 0.002044s
 TR predicted = 0.000106 , actual = 0.000101 , cost = 6.559511 , merit = 23.509314 , eq_vio = 1.591469 , in_vio = 0.022468 , QP_time = 0.001747s
 TR predicted = 0.000029 , actual = 0.000025 , cost = 6.563090 , merit = 23.509212 , eq_vio = 1.589022 , in_vio = 0.022313 , QP_time = 0.001875s
 TR predicted = 0.000028 , actual = 0.000017 , cost = 6.565625 , merit = 23.509187 , eq_vio = 1.589768 , in_vio = 0.022117 , QP_time = 0.002046s
 TR predicted = 0.000038 , actual = 0.000022 , cost = 6.567608 , merit = 23.509170 , eq_vio = 1.599741 , in_vio = 0.021891 , QP_time = 0.002022s
 TR predicted = 0.000049 , actual = 0.000028 , cost = 6.569613 , merit = 23.509149 , eq_vio = 1.600481 , in_vio = 0.021642 , QP_time = 0.002024s
 TR predicted = 0.000062 , actual = 0.000035 , cost = 6.571741 , merit = 23.509121 , eq_vio = 1.592949 , in_vio = 0.021366 , QP_time = 0.001960s
 TR predicted = 0.000077 , actual = 0.000044 , cost = 6.574113 , merit = 23.509086 , eq_vio = 1.590853 , in_vio = 0.021058 , QP_time = 0.001944s
 TR predicted = 0.000097 , actual = 0.000056 , cost = 6.576804 , merit = 23.509043 , eq_vio = 1.597566 , in_vio = 0.020710 , QP_time =

```

e = 0.001976s
TR      predicted = 0.000126 , actual = 0.000073 , cost = 6.579917 , merit = 23.508987 , eq_vio = 1.592774 , in_vio = 0.020312 , QP_time
e = 0.002014s
TR      predicted = 0.000168 , actual = 0.000099 , cost = 6.583582 , merit = 23.508914 , eq_vio = 1.592438 , in_vio = 0.019848 , QP_time
e = 0.002040s
TR      predicted = 0.000234 , actual = 0.000139 , cost = 6.587989 , merit = 23.508816 , eq_vio = 1.593673 , in_vio = 0.019298 , QP_time
e = 0.001969s
TR      predicted = 0.000341 , actual = 0.000207 , cost = 6.593426 , merit = 23.508677 , eq_vio = 1.594340 , in_vio = 0.018630 , QP_time
e = 0.002082s
TR      predicted = 0.000531 , actual = 0.000328 , cost = 6.600348 , merit = 23.508470 , eq_vio = 1.594715 , in_vio = 0.017791 , QP_time
e = 0.002052s
TR      predicted = 0.000899 , actual = 0.000530 , cost = 6.609528 , merit = 23.508142 , eq_vio = 1.595301 , in_vio = 0.016697 , QP_time
e = 0.002049s
TR-     predicted = 0.001943 , actual = 0.000024 , cost = 6.622554 , merit = 23.507612 , eq_vio = 1.596232 , in_vio = 0.015167 , QP_time
e = 0.001969s
TR      predicted = 0.006610 , actual = 0.004892 , cost = 6.644182 , merit = 23.507588 , eq_vio = 1.596902 , in_vio = 0.012622 , QP_time
e = 0.001744s
TR-     predicted = 0.016786 , actual = 0.003876 , cost = 6.677837 , merit = 23.502696 , eq_vio = 1.610062 , in_vio = 0.008632 , QP_time
e = 0.001682s
TR      predicted = 0.053961 , actual = 0.037585 , cost = 6.727590 , merit = 23.498820 , eq_vio = 1.609287 , in_vio = 0.000000 , QP_time
e = 0.001619s
TR      predicted = 0.072383 , actual = 0.054108 , cost = 6.719039 , merit = 23.461235 , eq_vio = 1.593042 , in_vio = 0.010766 , QP_time
e = 0.001624s
TR+     predicted = 0.053407 , actual = 0.062997 , cost = 6.693819 , merit = 23.407128 , eq_vio = 1.573960 , in_vio = 0.019310 , QP_time
e = 0.001795s
TR      predicted = 0.042004 , actual = 0.044839 , cost = 6.721769 , merit = 23.344130 , eq_vio = 1.561979 , in_vio = 0.011644 , QP_time
e = 0.001610s
TR      predicted = 0.050792 , actual = 0.044318 , cost = 6.840465 , merit = 23.299291 , eq_vio = 1.555395 , in_vio = 0.001306 , QP_time
e = 0.001702s
TR      predicted = 0.020230 , actual = 0.018565 , cost = 6.934746 , merit = 23.254973 , eq_vio = 1.545001 , in_vio = 0.000000 , QP_time
e = 0.001826s
TR      predicted = 0.048519 , actual = 0.040031 , cost = 6.951696 , merit = 23.236407 , eq_vio = 1.528440 , in_vio = 0.011246 , QP_time
e = 0.001630s
TR      predicted = 0.019634 , actual = 0.019668 , cost = 6.969844 , merit = 23.196376 , eq_vio = 1.509973 , in_vio = 0.019102 , QP_time
e = 0.001732s
TR      predicted = 0.005609 , actual = 0.007221 , cost = 7.020030 , merit = 23.176708 , eq_vio = 1.500451 , in_vio = 0.017801 , QP_time
e = 0.001737s
TR      predicted = 0.029802 , actual = 0.010889 , cost = 7.093695 , merit = 23.169486 , eq_vio = 1.498497 , in_vio = 0.013098 , QP_time
e = 0.001826s
TR      predicted = 0.096748 , actual = 0.061730 , cost = 7.196429 , merit = 23.158598 , eq_vio = 1.490741 , in_vio = 0.000000 , QP_time
e = 0.001803s
TR      predicted = 0.053121 , actual = 0.052377 , cost = 7.292111 , merit = 23.096867 , eq_vio = 1.468043 , in_vio = 0.000557 , QP_time
e = 0.001733s
TR      predicted = 0.047918 , actual = 0.036332 , cost = 7.303885 , merit = 23.044491 , eq_vio = 1.443314 , in_vio = 0.010272 , QP_time
e = 0.001802s
TR      predicted = 0.045880 , actual = 0.051498 , cost = 7.365000 , merit = 23.008159 , eq_vio = 1.416979 , in_vio = 0.019384 , QP_time
e = 0.001686s
TR      predicted = 0.067947 , actual = 0.041525 , cost = 7.502035 , merit = 22.956661 , eq_vio = 1.398187 , in_vio = 0.010572 , QP_time
e = 0.001809s
TR      predicted = 0.088172 , actual = 0.057179 , cost = 7.757111 , merit = 22.915135 , eq_vio = 1.385093 , in_vio = 0.000000 , QP_time
e = 0.001602s
TR      predicted = 0.095423 , actual = 0.103777 , cost = 7.919304 , merit = 22.857957 , eq_vio = 1.341853 , in_vio = 0.010737 , QP_time
e = 0.001753s
TR      predicted = 0.118380 , actual = 0.086994 , cost = 8.117147 , merit = 22.754180 , eq_vio = 1.287982 , in_vio = 0.014275 , QP_time
e = 0.001820s
TR-     predicted = 0.188154 , actual = 0.017719 , cost = 8.451854 , merit = 22.667185 , eq_vio = 1.231159 , in_vio = 0.000000 , QP_time
e = 0.001807s
TR+     predicted = 0.355580 , actual = 0.358871 , cost = 8.917805 , merit = 22.649466 , eq_vio = 1.127490 , in_vio = 0.018096 , QP_time
e = 0.001876s
TR+     predicted = 0.130238 , actual = 0.114205 , cost = 9.115102 , merit = 22.290596 , eq_vio = 1.109017 , in_vio = 0.013882 , QP_time
e = 0.001776s
TR      predicted = 0.266835 , actual = 0.144508 , cost = 9.553101 , merit = 22.176391 , eq_vio = 1.048367 , in_vio = 0.006477 , QP_time
e = 0.001624s
TR      predicted = 0.600921 , actual = 0.314182 , cost = 10.449473 , merit = 22.031882 , eq_vio = 0.888749 , in_vio = 0.000000 , QP_time
me = 0.001650s
TR+     predicted = 0.947622 , actual = 0.834440 , cost = 11.398127 , merit = 21.717700 , eq_vio = 0.726845 , in_vio = 0.025194 , QP_time
me = 0.001779s
TR+     predicted = 0.845655 , actual = 0.869520 , cost = 12.352050 , merit = 20.883260 , eq_vio = 0.559616 , in_vio = 0.020946 , QP_time
me = 0.001722s
TR      predicted = 1.151005 , actual = 1.235847 , cost = 13.329342 , merit = 20.013740 , eq_vio = 0.204165 , in_vio = 0.018555 , QP_time
me = 0.001910s
TR      predicted = 1.405580 , actual = 1.300428 , cost = 14.027919 , merit = 18.777893 , eq_vio = 0.115600 , in_vio = 0.007674 , QP_time
me = 0.001868s
TR      predicted = 1.557272 , actual = 1.478397 , cost = 14.647205 , merit = 17.477465 , eq_vio = 0.055744 , in_vio = 0.000000 , QP_time
me = 0.001795s
TR      predicted = 1.644853 , actual = 1.589171 , cost = 15.130457 , merit = 15.999068 , eq_vio = 0.007689 , in_vio = 0.000974 , QP_time
me = 0.001756s
TR      predicted = 0.902309 , actual = 0.798529 , cost = 13.957150 , merit = 14.409897 , eq_vio = 0.002371 , in_vio = 0.016275 , QP_time
me = 0.001784s
TR      predicted = 0.579521 , actual = 0.524830 , cost = 13.481063 , merit = 13.611368 , eq_vio = 0.001040 , in_vio = 0.000000 , QP_time

```

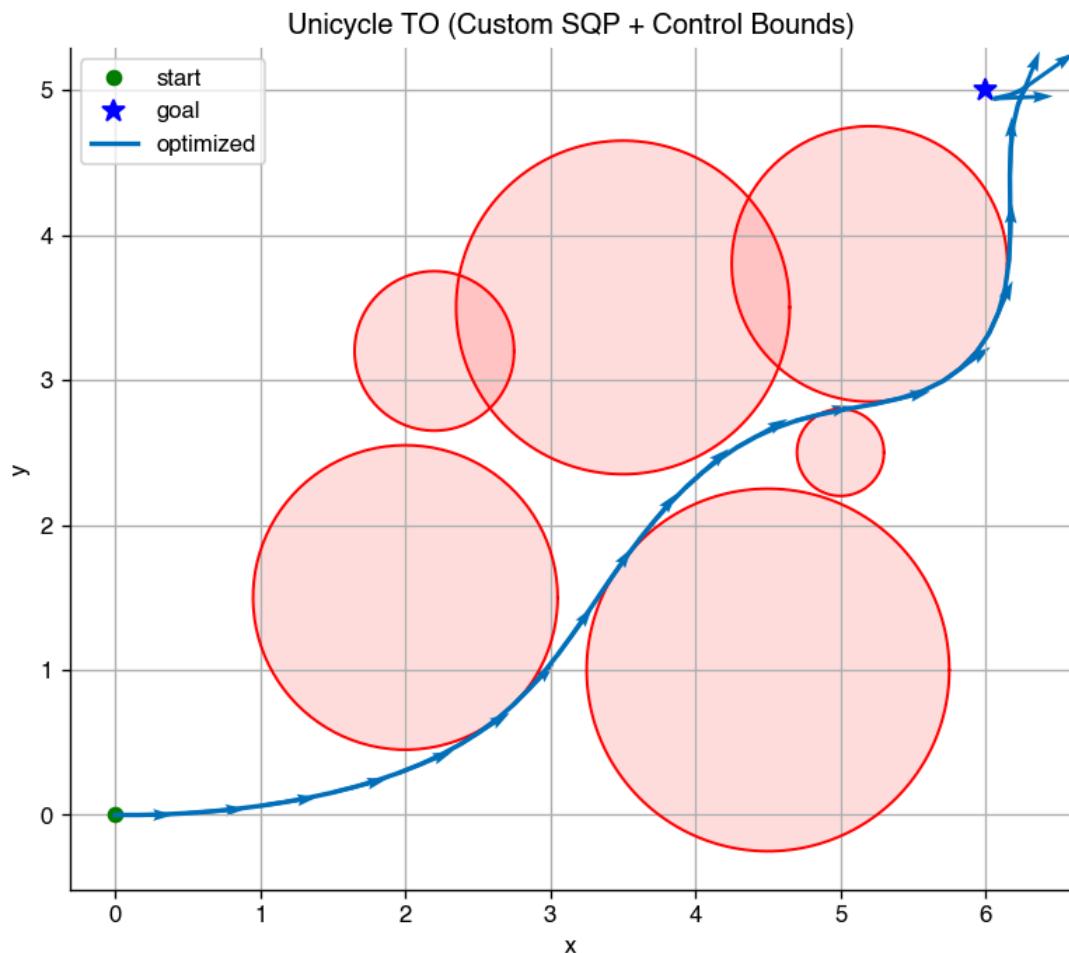
```
me = 0.001711s
TR    predicted = 0.395350 , actual = 0.354285 , cost = 12.743497 , merit = 13.086537 , eq_vio = 0.001011 , in_vio = 0.017985 , QP_t
me = 0.001828s
TR    predicted = 0.271746 , actual = 0.225455 , cost = 12.637911 , merit = 12.732252 , eq_vio = 0.000640 , in_vio = 0.000000 , QP_t
me = 0.001812s
TR    predicted = 0.220124 , actual = 0.218862 , cost = 12.133981 , merit = 12.506797 , eq_vio = 0.000899 , in_vio = 0.025158 , QP_t
me = 0.001770s
TR    predicted = 0.108210 , actual = 0.113994 , cost = 12.066444 , merit = 12.287935 , eq_vio = 0.000403 , in_vio = 0.019189 , QP_t
me = 0.001747s
TR    predicted = 0.105370 , actual = 0.075878 , cost = 12.144063 , merit = 12.173941 , eq_vio = 0.000320 , in_vio = 0.000000 , QP_t
me = 0.001738s
TR    predicted = 0.114078 , actual = 0.109765 , cost = 11.876034 , merit = 12.098063 , eq_vio = 0.000490 , in_vio = 0.016386 , QP_t
me = 0.001663s
TR    predicted = 0.073513 , actual = 0.068039 , cost = 11.707193 , merit = 11.988297 , eq_vio = 0.000231 , in_vio = 0.024958 , QP_t
me = 0.001625s
TR    predicted = 0.088727 , actual = 0.056921 , cost = 11.738710 , merit = 11.920258 , eq_vio = 0.000307 , in_vio = 0.015705 , QP_t
me = 0.001813s
TR    predicted = 0.109369 , actual = 0.055651 , cost = 11.831531 , merit = 11.863338 , eq_vio = 0.000261 , in_vio = 0.000000 , QP_t
me = 0.001746s
TR    predicted = 0.094819 , actual = 0.070055 , cost = 11.637216 , merit = 11.807686 , eq_vio = 0.000547 , in_vio = 0.010801 , QP_t
me = 0.001673s
TR    predicted = 0.078591 , actual = 0.082831 , cost = 11.397990 , merit = 11.737631 , eq_vio = 0.000300 , in_vio = 0.028966 , QP_t
me = 0.001659s
TR    predicted = 0.039031 , actual = 0.041116 , cost = 11.422515 , merit = 11.654800 , eq_vio = 0.000210 , in_vio = 0.021964 , QP_t
me = 0.001660s
TR    predicted = 0.082108 , actual = 0.064628 , cost = 11.517069 , merit = 11.613683 , eq_vio = 0.000362 , in_vio = 0.007169 , QP_t
me = 0.001584s
TR    predicted = 0.044356 , actual = 0.034547 , cost = 11.531575 , merit = 11.549056 , eq_vio = 0.000203 , in_vio = 0.000000 , QP_t
me = 0.001974s
TR    predicted = 0.011122 , actual = 0.009325 , cost = 11.504699 , merit = 11.514509 , eq_vio = 0.000112 , in_vio = 0.000000 , QP_t
me = 0.001761s
TR    predicted = 0.002728 , actual = 0.001384 , cost = 11.503387 , merit = 11.505184 , eq_vio = 0.000016 , in_vio = 0.000000 , QP_t
me = 0.001776s
TR    predicted = 0.001983 , actual = 0.001171 , cost = 11.502456 , merit = 11.503799 , eq_vio = 0.000011 , in_vio = 0.000000 , QP_t
me = 0.001868s
TR    predicted = 0.001164 , actual = 0.000697 , cost = 11.501816 , merit = 11.502628 , eq_vio = 0.000006 , in_vio = 0.000000 , QP_t
me = 0.001777s
TR    predicted = 0.000674 , actual = 0.000398 , cost = 11.501464 , merit = 11.501931 , eq_vio = 0.000004 , in_vio = 0.000000 , QP_t
me = 0.001676s
TR    predicted = 0.000402 , actual = 0.000235 , cost = 11.501257 , merit = 11.501534 , eq_vio = 0.000002 , in_vio = 0.000000 , QP_t
me = 0.001708s
TR    predicted = 0.000244 , actual = 0.000141 , cost = 11.501132 , merit = 11.501298 , eq_vio = 0.000002 , in_vio = 0.000000 , QP_t
me = 0.001684s
TR    predicted = 0.000150 , actual = 0.000084 , cost = 11.501054 , merit = 11.501157 , eq_vio = 0.000001 , in_vio = 0.000000 , QP_t
me = 0.001853s
TR    predicted = 0.000104 , actual = 0.000053 , cost = 11.500168 , merit = 11.501073 , eq_vio = 0.000001 , in_vio = 0.000084 , QP_t
me = 0.001740s
TR    predicted = 0.000170 , actual = 0.000078 , cost = 11.495336 , merit = 11.501020 , eq_vio = 0.000001 , in_vio = 0.000562 , QP_t
me = 0.001642s
TR    predicted = 0.000935 , actual = 0.000481 , cost = 11.479769 , merit = 11.500942 , eq_vio = 0.000001 , in_vio = 0.002094 , QP_t
me = 0.001820s
TR    predicted = 0.006404 , actual = 0.003981 , cost = 11.435175 , merit = 11.500462 , eq_vio = 0.000010 , in_vio = 0.006366 , QP_t
me = 0.001875s
TR    predicted = 0.028567 , actual = 0.024238 , cost = 11.320836 , merit = 11.496480 , eq_vio = 0.000065 , in_vio = 0.016527 , QP_t
me = 0.001768s
TR    predicted = 0.042069 , actual = 0.047155 , cost = 11.177827 , merit = 11.472242 , eq_vio = 0.000125 , in_vio = 0.027429 , QP_t
me = 0.001765s
TR    predicted = 0.032454 , actual = 0.035134 , cost = 11.227466 , merit = 11.425087 , eq_vio = 0.000176 , in_vio = 0.018549 , QP_t
me = 0.001711s
TR    predicted = 0.061905 , actual = 0.042847 , cost = 11.308844 , merit = 11.389953 , eq_vio = 0.000283 , in_vio = 0.006319 , QP_t
me = 0.001691s
TR    predicted = 0.043144 , actual = 0.032990 , cost = 11.328048 , merit = 11.347106 , eq_vio = 0.000170 , in_vio = 0.000000 , QP_t
me = 0.001669s
TR    predicted = 0.011499 , actual = 0.010363 , cost = 11.303962 , merit = 11.314116 , eq_vio = 0.000082 , in_vio = 0.000000 , QP_t
me = 0.001765s
TR    predicted = 0.001502 , actual = 0.001184 , cost = 11.302617 , merit = 11.303753 , eq_vio = 0.000009 , in_vio = 0.000000 , QP_t
me = 0.001729s
TR    predicted = 0.000429 , actual = 0.000327 , cost = 11.302250 , merit = 11.302569 , eq_vio = 0.000003 , in_vio = 0.000000 , QP_t
me = 0.001898s
TR    predicted = 0.000137 , actual = 0.000104 , cost = 11.302140 , merit = 11.302242 , eq_vio = 0.000001 , in_vio = 0.000000 , QP_t
me = 0.001715s
TR    predicted = 0.000044 , actual = 0.000033 , cost = 11.302105 , merit = 11.302137 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_t
me = 0.001670s
TR    predicted = 0.000015 , actual = 0.000011 , cost = 11.302094 , merit = 11.302105 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_t
me = 0.001643s
TR    predicted = 0.000005 , actual = 0.000004 , cost = 11.302090 , merit = 11.302093 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_t
me = 0.001705s
TR    predicted = 0.000002 , actual = 0.000001 , cost = 11.302089 , merit = 11.302090 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_t
me = 0.001655s
TR    predicted = 0.000001 , actual = 0.000000 , cost = 11.302088 , merit = 11.302089 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_t
```

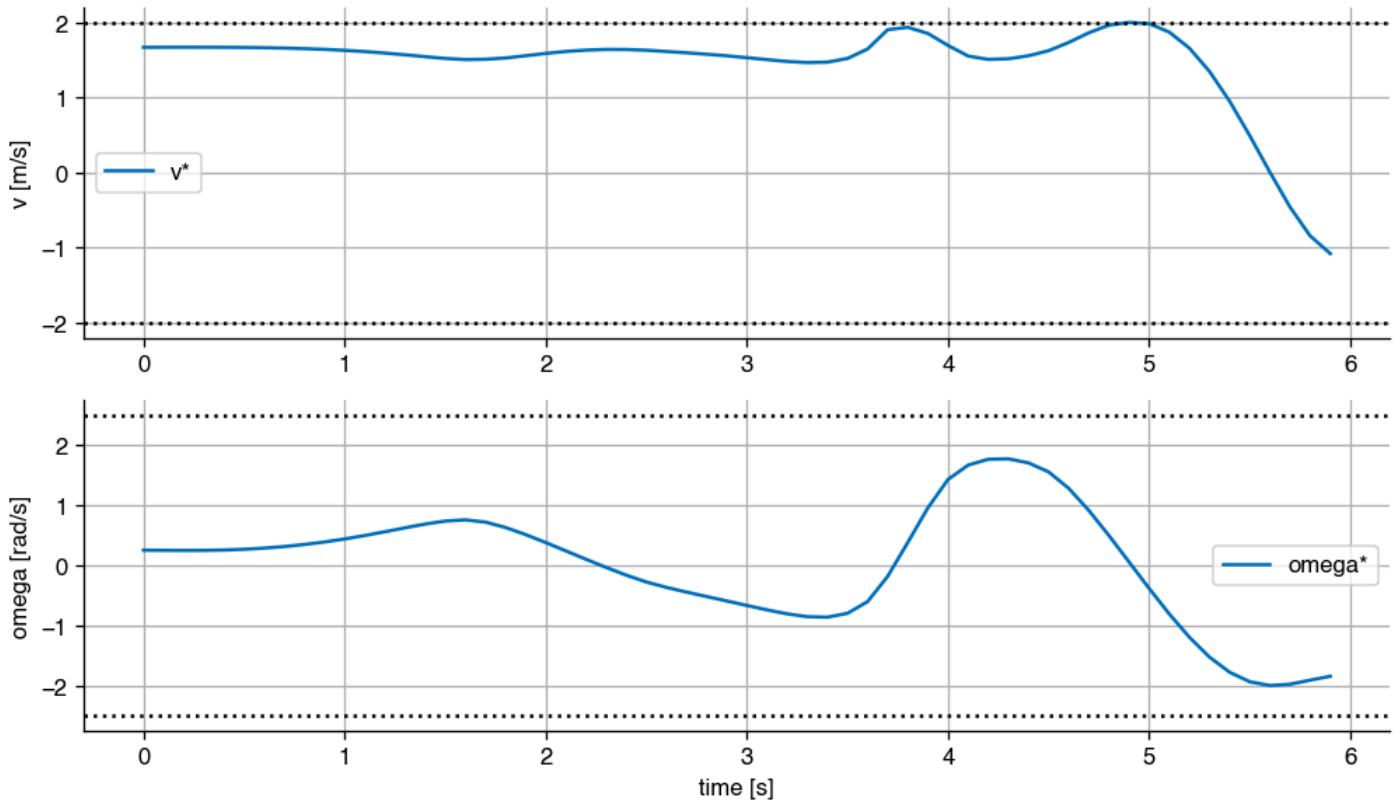
```

me = 0.001715s
Increase lower than tolerance at iter 119
TR    predicted = 0.000001 , actual = 0.000001 , cost = 11.302088 , merit = 11.302089 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_ti
me = 0.001995s
Increase lower than tolerance at iter 120
TR    predicted = 0.000004 , actual = 0.000003 , cost = 11.302088 , merit = 11.302092 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_ti
me = 0.002104s
TR    predicted = 0.000001 , actual = 0.000001 , cost = 11.302088 , merit = 11.302089 , eq_vio = 0.000000 , in_vio = 0.000000 , QP_ti
me = 0.002043s
Increase lower than tolerance at iter 122
Penalty too large at iter 122

```

Terminal state [px py theta]^T = [6.065 4.941 0.040]





Problem 4: Model Predictive Control for Pendulum Swing-Up

A simple pendulum consists of a point mass m attached to a massless rigid rod of length ℓ that pivots freely under gravity g . We measure the angle θ from the **upright** position ($\theta = 0$), with angular velocity $\omega = \dot{\theta}$, so the state is $x = [\theta, \omega]^\top$ and the control input is a torque τ applied at the pivot. The classic **swing-up** problem starts near the downward equilibrium ($\theta \approx \pi$, $\omega \approx 0$) and seeks a torque sequence that drives the pendulum to the upright ($\theta = 0$, $\omega = 0$) efficiently while respecting actuator limits.

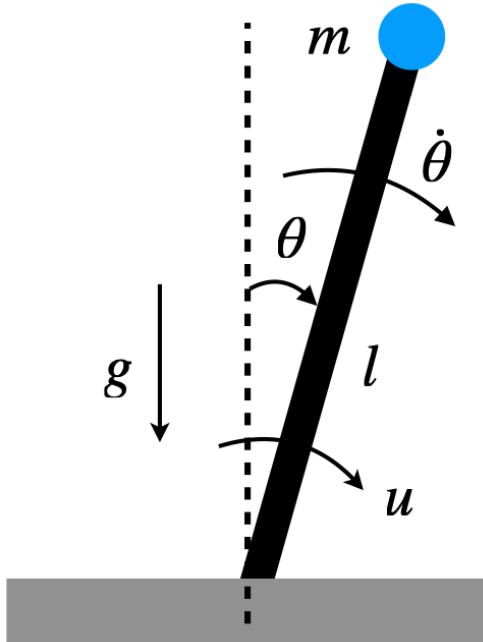


Figure 2. Pendulum.

$$\text{State / input: } x = \begin{bmatrix} \theta \\ \omega \end{bmatrix}, u = \tau \text{ (torque)}$$

Parameters: mass m , length ℓ , gravity g .

Bounds: $\tau(t) \in [\tau_{\min}, \tau_{\max}]$.

Dynamics

$$\dot{\theta} = \omega, \quad \dot{\omega} = -\frac{g}{\ell} \sin \theta + \frac{1}{m\ell^2} \tau.$$

Initial condition

$$x(0) = \begin{bmatrix} \pi \\ 0 \end{bmatrix} \quad (\text{downward at rest}).$$

Objective

$$\min_{\tau(\cdot)} \int_0^T \left[q_a (\sin^2 \theta + (\cos \theta - 1)^2) + q_\omega \omega^2 + q_r \tau^2 \right] dt$$

subject to the dynamics, initial condition, and torque bounds.

Below is a simulation environment to test your controller.

```
In [48]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def simulate_pendulum(u_fun,
                      x0=(0.01, 0.0), T=10.0, dt=0.01,
                      m=1.0, l=1.0, g=9.81,
                      tau_limits=None,
                      save_gif=False, gif_path="pendulum.gif", fps=None):
    """
    Simulate θ (upright=0) with torque policy u_fun(t, x) → scalar.
    Returns (t, X, U). If save_gif, writes an animated GIF.
    """
    N = int(T/dt); t = np.linspace(0, T, N+1)
    X = np.zeros((N+1, 2)); X[0] = np.array(x0, float)
    U = np.zeros(N)

    def f(x, tt):
        th, w = x
        u = float(u_fun(tt, x))
        if tau_limits is not None: u = float(np.clip(u, tau_limits[0], tau_limits[1]))
        return np.array([w, (g/l)*np.sin(th) + u/(m*l*l)])

    def rk4(x, tt, h):
        k1 = f(x, tt)
        k2 = f(x + 0.5*h*k1, tt + 0.5*h)
        k3 = f(x + 0.5*h*k2, tt + 0.5*h)
        k4 = f(x + h*k3, tt + h)
        return x + (h/6.0)*(k1 + 2*k2 + 2*k3 + k4)

    for k in range(N):
        th, w = X[k]
        u = float(u_fun(t[k], X[k]))
        if tau_limits is not None: u = float(np.clip(u, tau_limits[0], tau_limits[1]))
        U[k] = u
        X[k+1] = rk4(X[k], t[k], dt)

    if save_gif:
        xx_full = l*np.sin(X[:,0])
        yy_full = l*np.cos(X[:,0])

        stride = 10
        idx = np.arange(0, N+1, stride)
        xx, yy = xx_full[idx], yy_full[idx]
        fig, ax = plt.subplots(figsize=(5,5))
        ax.set_aspect("equal", adjustable="box")
        ax.set_xlim(-1.15*l, 1.15*l); ax.set_ylim(-1.15*l, 1.15*l); ax.axis("off")
        ax.plot(0, 0, "o", ms=6, color="black")
        rod, = ax.plot([0, xx[0]], [0, yy[0]], lw=3, alpha=0.9)
        bob = plt.Circle((xx[0], yy[0]), 0.06*l, ec="black", lw=1.2, fc="#87CEFA")
        ax.add_patch(bob)

        def update(i):
            rod.set_data([0, xx[i]], [0, yy[i]])
            bob.center = (xx[i], yy[i])
            return rod, bob

        out_fps = max(1, int(1/(dt*stride))) if fps is None else fps
        ani = FuncAnimation(fig, update, frames=len(idx), interval=1000*dt*stride, blit=True)
```

```

        ani.save(gif_path, writer="pillow", fps=out_fps)
        plt.close(fig)

    return t, X, U

t, X, U = simulate_pendulum(lambda tt, x: 0.0, save_gif=True)

```

4.1 Trajectory optimization

With $N = 50$ and $dt = 0.1$, solve the trajectory optimization problem of swinging up the pendulum from the bottom right position: parameterize controls, discretize dynamics on each interval, and enforce control bound constraints. Plot your optimal trajectory and control signal.

This problem is similar to Problem 3.

```
In [49]: import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize, Bounds

# -----
# Problem setup (pendulum)
# -----
N = 50
dt = 0.1
T = N * dt
m, l, g = 1.0, 1.0, 9.81
x0 = np.array([np.pi, 0.0]) # [theta (upright=0), omega]

# cost weights
q_a, q_omega, q_r = 20.0, 2.0, 1e-2
u_min, u_max = -2.0, 2.0

def f_pend(x, u):
    th, w = x
    return np.array([w, (g/l)*np.sin(th) + u/(m*l)], dtype=float)

# -----
# Decision variables: z = [th_0..th_N, w_0..w_N, u_0..u_{N-1}]
# -----
def unpack(z):
    th = z[0:N+1]
    w = z[N+1:2*(N+1)]
    u = z[2*(N+1):]
    return th, w, u

def pack(th, w, u):
    return np.concatenate([th, w, u], axis=0)

# -----
# Objective (pendulum swing-up)
# J ≈ ∑ dt * [ q_a( sin^2 th + (cos th - 1)^2 ) + q_omega w^2 + q_r u^2 ]
#     + strong terminal state term
# -----
def objective(z):
    #####
    # TODO: Build objective here
    #####
    th, w, u = unpack(z)
    J = 0.0

    # running cost
    for k in range(N):
        angle_term = np.sin(th[k])**2 + (np.cos(th[k]) - 1.0)**2
        J += dt * (q_a * angle_term + q_omega * w[k]**2 + q_r * u[k]**2)

    # strong terminal cost
    q_a_T, q_omega_T = 2000.0, 50.0
    angle_term_N = np.sin(th[-1])**2 + (np.cos(th[-1]) - 1.0)**2
    J += q_a_T * angle_term_N + q_omega_T * w[-1]**2

    return J

# -----
# Multiple-shooting equality constraints (Euler)
# -----
def dynamics_constraints(z):
    #####
    # TODO: Build the dynamic constraints here
    #####
```

```

#####
th, w, u = unpack(z)
cons = []

# initial condition
cons.append(th[0] - x0[0])
cons.append(w[0] - x0[1])

# dynamics for k = 0..N-1 using forward Euler
for k in range(N):
    xk = np.array([th[k], w[k]])
    fk = f_pend(xk, u[k])
    th_next = th[k] + dt * fk[0]
    w_next = w[k] + dt * fk[1]

    cons.append(th[k+1] - th_next)
    cons.append(w[k+1] - w_next)

return np.array(cons)

# -----
# Bounds
# -----
#####
# TODO: Finish Bounds
#####
# theta and omega unbounded; u in [u_min, u_max]
lb_th = -np.inf * np.ones(N+1)
ub_th = np.inf * np.ones(N+1)
lb_w = -np.inf * np.ones(N+1)
ub_w = np.inf * np.ones(N+1)
lb_u = u_min * np.ones(N)
ub_u = u_max * np.ones(N)

lb = np.concatenate([lb_th, lb_w, lb_u])
ub = np.concatenate([ub_th, ub_w, ub_u])
bounds = Bounds(lb, ub)

# -----
# Feasible initial guess
# -----
#####
# TODO: Finish Initial guess
#####
# Non-trivial u0: sinusoid inside [-2, 2], then roll out with Euler
th0 = np.zeros(N+1)
w0 = np.zeros(N+1)
th0[0] = x0[0]
w0[0] = x0[1]

t_ctrl = np.linspace(0.0, T - dt, N)
u0 = 1.8 * np.sin(2 * np.pi * t_ctrl / T) # stays within torque bounds

for k in range(N):
    xk = np.array([th0[k], w0[k]])
    fk = f_pend(xk, u0[k])
    th0[k+1] = th0[k] + dt * fk[0]
    w0[k+1] = w0[k] + dt * fk[1]

z0 = pack(th0, w0, u0)

# -----
# Solve
# -----
eq_con = {'type': 'eq', 'fun': dynamics_constraints}

res = minimize(
    objective,
    z0,
    method='SLSQP',
    bounds=bounds,
    constraints=[eq_con],
    options={'maxiter': 500, 'ftol': 1e-4, 'disp': True}
)

print("\nSuccess:", res.success, "| Status:", res.status)
print("Message:", res.message)
print("Final objective:", res.fun)

# -----

```

```

# Extract & verify by RK4 sim with u_k (piecewise-constant)
# -----
th_sol, w_sol, u_sol = unpack(res.x)
t_nodes = np.linspace(0.0, T, N+1)
t_ctrl = np.linspace(0.0, T - dt, N)

# -----
# Plots
# -----
fig, axes = plt.subplots(3, 1, figsize=(10, 9), sharex=True)

axes[0].plot(t_nodes, th_sol, label="theta (OCP, Euler nodes)")
axes[0].set_ylabel("theta [rad]"); axes[0].legend(); axes[0].grid(alpha=0.3)

axes[1].plot(t_nodes, w_sol, label="omega (OCP, Euler nodes)")
axes[1].set_ylabel("omega [rad/s]"); axes[1].legend(); axes[1].grid(alpha=0.3)

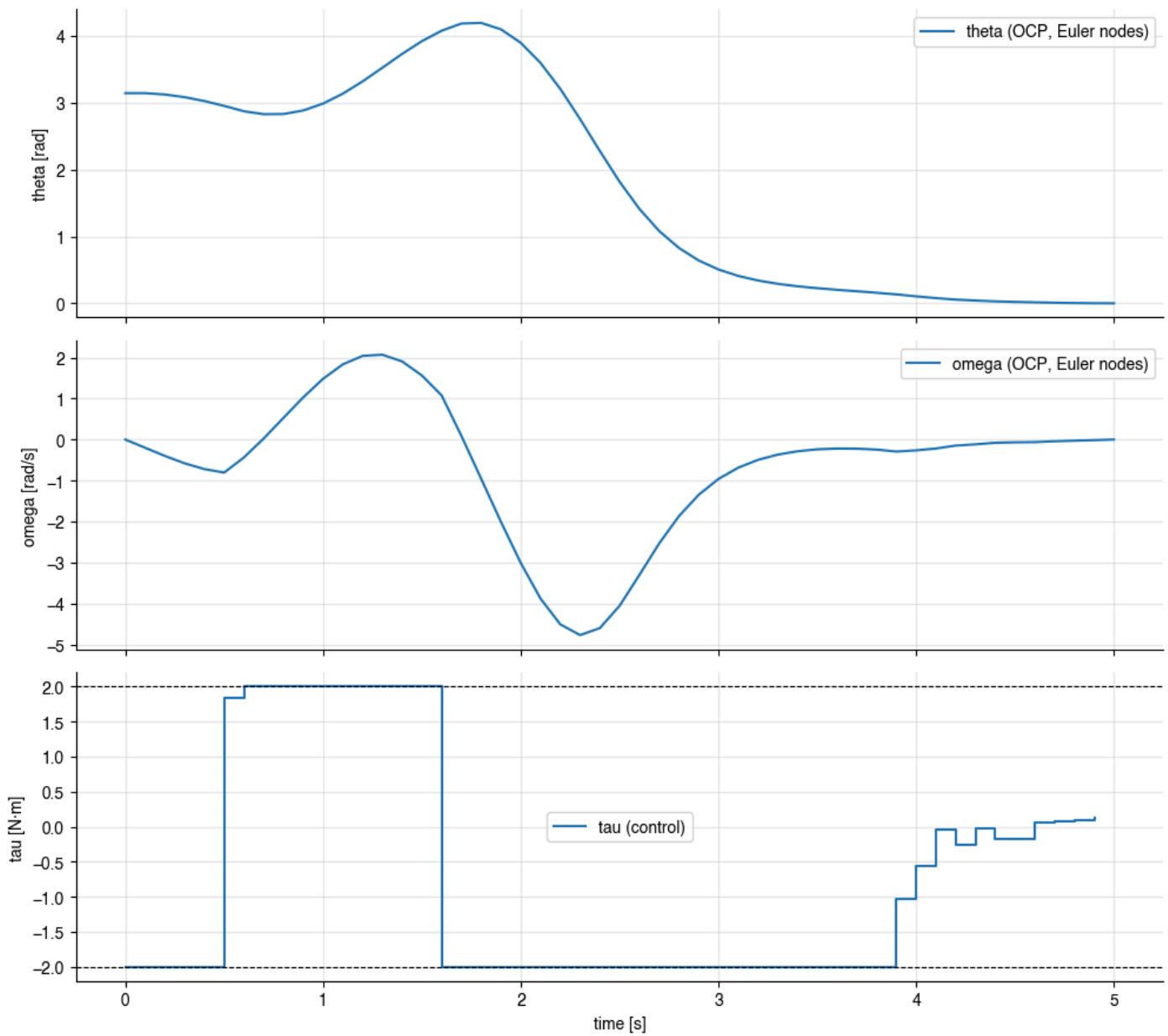
axes[2].step(t_ctrl, u_sol, where='post', label="tau (control)")
axes[2].axhline(u_min, ls='--', c='k', lw=0.8); axes[2].axhline(u_max, ls='--', c='k', lw=0.8)
axes[2].set_xlabel("time [s]"); axes[2].set_ylabel("tau [N·m]"); axes[2].legend(); axes[2].grid(alpha=0.3)

plt.tight_layout(); plt.show()

```

Optimization terminated successfully (Exit mode 0)
 Current function value: 232.82303291522223
 Iterations: 53
 Function evaluations: 8113
 Gradient evaluations: 53

Success: True | Status: 0
 Message: Optimization terminated successfully
 Final objective: 232.8230329152223



4.2 Failure of open loop control

Now we solved out a trajectory with a given initial state. However, does this imply that if we simply run the planned controls on a real system, i.e., doing open-loop control, the trajectory will behave as planned? Unfortunately the answer is No, trajectory optimization only produces an approximate solution to the optimal control problem (OCP), regardless of what transcription method is used. So the error may cumulate along time horizon.

Rollout the control to see the fail of swing up.

TODO: Finish the code and comment on what you see.

```
In [50]: def u_fun(t, x):
    k = int(t // dt)
    if k < 0: k = 0
    if k >= len(u_sol): k = len(u_sol) - 1
    return float(u_sol[k]) # open-loop (ignores x)

# --- "real-world" simulation (can be finer dt than planning; choose dt_sim) ---
dt_sim = 0.01 # finer integration for verification
t_sim, X_sim, U_sim = simulate_pendulum(
    u_fun,
    x0=x0, T=T,
    tau_limits=(u_min, u_max),
    save_gif=True # set True to export an animation
)
```

```

#####
# TODO: Plot here
#####
fig, axes = plt.subplots(3, 1, figsize=(10, 9), sharex=True)

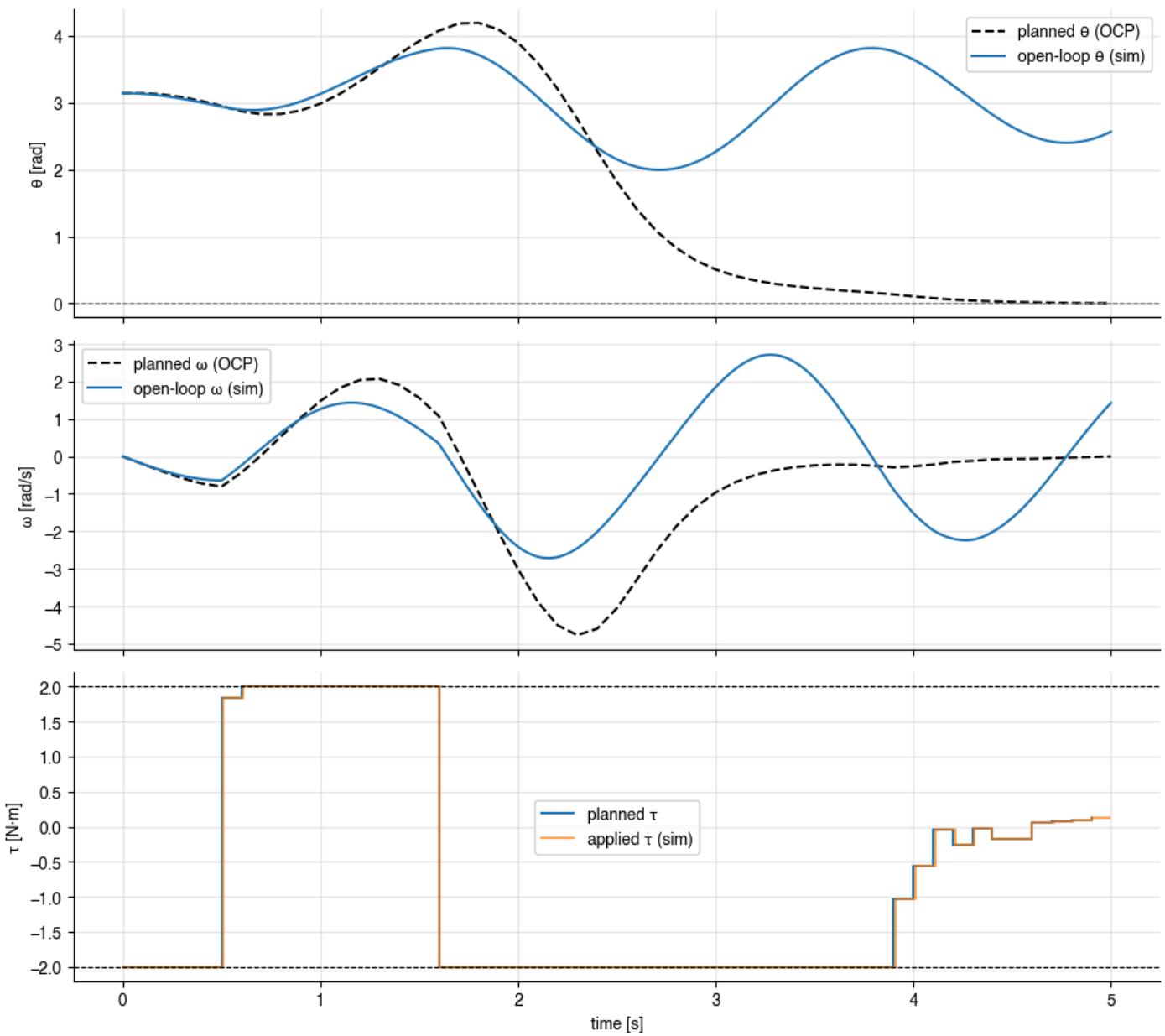
# θ comparison
axes[0].plot(t_nodes, th_sol, "k--", label="planned θ (OCP)")
axes[0].plot(t_sim, X_sim[:, 0], label="open-loop θ (sim)")
axes[0].axhline(0.0, ls="--", c="gray", lw=0.8)
axes[0].set_ylabel("θ [rad]")
axes[0].legend()
axes[0].grid(alpha=0.3)

# ω comparison
axes[1].plot(t_nodes, w_sol, "k--", label="planned ω (OCP)")
axes[1].plot(t_sim, X_sim[:, 1], label="open-loop ω (sim)")
axes[1].set_ylabel("ω [rad/s]")
axes[1].legend()
axes[1].grid(alpha=0.3)

# τ comparison
axes[2].step(t_ctrl, u_sol, where="post", label="planned τ")
axes[2].step(t_sim[:-1], U_sim, where="post", alpha=0.7, label="applied τ (sim)")
axes[2].axhline(u_min, ls="--", c="k", lw=0.8)
axes[2].axhline(u_max, ls="--", c="k", lw=0.8)
axes[2].set_xlabel("time [s]")
axes[2].set_ylabel("τ [N·m]")
axes[2].legend()
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

```



Comment:

When we apply the optimized control sequence in open-loop, the actual trajectory initially matches the planned swing-up, but then gradually diverges. Small integration and modeling errors accumulate because there is no feedback to correct the state. As a result, the pendulum fails to reach the upright position in simulation despite using the same control inputs. This demonstrates that open-loop solutions from trajectory optimization are inherently fragile and not robust to small perturbations.

4.3 Model Predictive Control

Now we will use model predictive control to get a feedback controller. Basically at every time step, we solve a trajectory optimization problem to obtain an open-loop state-control trajectory, but we only execute the first control input. Executing the first control input will bring the system to a new state, and we resolve the trajectory optimization problem with this new initial state. When solving the new TO problem, you can reuse the solution from previous solve as the initial guess.

TODO: Finish the code and plot the final trajectory

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize, Bounds
```

```

# Problem parameters
N = 50           # horizon length (number of steps)
dt = 0.1         # time step
T_horizon = N * dt

T_total = 10.0   # total MPC simulation time
N_mpc_steps = int(T_total / dt)

m, l, g = 1.0, 1.0, 9.81
x0_initial = np.array([np.pi, 0.0]) # [theta, omega] (downward start)

# Cost weights
q_a, q_omega, q_r = 20.0, 2.0, 1e-2
q_a_T, q_omega_T = 5000.0, 500.0

# Control bounds
u_min, u_max = -2.0, 2.0

# Dynamics and integrator
def f_pend(x, u):
    th, w = x
    return np.array([
        w,
        (g / l) * np.sin(th) + u / (m * l * l)
    ], dtype=float)

def rk4_step(x, u, h, u_limits):
    u = float(np.clip(u, u_limits[0], u_limits[1]))
    k1 = f_pend(x, u)
    k2 = f_pend(x + 0.5 * h * k1, u)
    k3 = f_pend(x + 0.5 * h * k2, u)
    k4 = f_pend(x + h * k3, u)
    return x + (h / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)

# Decision variable packing: z = [th_0..th_N, w_0..w_N, u_0..u_{N-1}]
def unpack(z):
    th = z[0:N + 1]
    w = z[N + 1:2 * (N + 1)]
    u = z[2 * (N + 1):]
    return th, w, u

def pack(th, w, u):
    return np.concatenate([th, w, u], axis=0)

def objective(z):
    th, w, u = unpack(z)

    # running cost
    angle_term = np.sin(th[:-1])**2 + (np.cos(th[:-1]) - 1.0)**2
    stage_cost = q_a * angle_term + q_omega * w[:-1]**2 + q_r * u**2
    J = dt * np.sum(stage_cost)

    # terminal cost
    angle_term_N = np.sin(th[-1])**2 + (np.cos(th[-1]) - 1.0)**2
    J += q_a_T * angle_term_N + q_omega_T * w[-1]**2

    return J

# Current initial state for the OCP (updated each MPC step)
x_current = x0_initial.copy()

def dynamics_constraints(z):
    th, w, u = unpack(z)
    cons = []

    # initial condition
    cons.append(th[0] - x_current[0])
    cons.append(w[0] - x_current[1])

    # dynamics over the horizon
    for k in range(N):
        xk = np.array([th[k], w[k]])
        uk = u[k]

```

```

k1 = f_pend(xk, uk)
k2 = f_pend(xk + 0.5 * dt * k1, uk)
k3 = f_pend(xk + 0.5 * dt * k2, uk)
k4 = f_pend(xk + dt * k3, uk)
x_next = xk + (dt / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4)

cons.append(th[k + 1] - x_next[0])
cons.append(w[k + 1] - x_next[1])

return np.array(cons)

# Bounds
lb_th = -np.inf * np.ones(N + 1)
ub_th = np.inf * np.ones(N + 1)
lb_w = -np.inf * np.ones(N + 1)
ub_w = np.inf * np.ones(N + 1)
lb_u = u_min * np.ones(N)
ub_u = u_max * np.ones(N)

lb = np.concatenate([lb_th, lb_w, lb_u])
ub = np.concatenate([ub_th, ub_w, ub_u])
bounds = Bounds(lb, ub)

eq_con = {'type': 'eq', 'fun': dynamics_constraints}

# MPC setup
print(f'MPC: {N_mpc_steps} steps, dt={dt}, total time={T_total}s')

t_mpc = np.linspace(0.0, T_total, N_mpc_steps + 1)
X_mpc = np.zeros((N_mpc_steps + 1, 2))
U_mpc = np.zeros(N_mpc_steps)
X_mpc[0] = x0_initial.copy()

# initial guess (simple forward simulation)
th0 = np.zeros(N + 1)
w0 = np.zeros(N + 1)
th0[0] = x0_initial[0]
w0[0] = x0_initial[1]

t_ctrl = np.linspace(0.0, T_horizon - dt, N)
u0 = 1.8 * np.sin(2 * np.pi * t_ctrl / T_horizon)

for k in range(N):
    xk = np.array([th0[k], w0[k]])
    fk = f_pend(xk, u0[k])
    th0[k + 1] = th0[k] + dt * fk[0]
    w0[k + 1] = w0[k] + dt * fk[1]

z_guess = pack(th0, w0, u0)

# MPC loop
for k in range(N_mpc_steps):
    # update OCP initial condition
    x_current = X_mpc[k]

    res = minimize(
        objective,
        z_guess,
        method='SLSQP',
        bounds=bounds,
        constraints=[eq_con],
        options={'maxiter': 150, 'ftol': 1e-4, 'disp': False},
    )

    th_opt, w_opt, u_opt = unpack(res.x)
    u_k = u_opt[0]

    # shift warm start
    th_shift = np.concatenate([th_opt[1:], th_opt[-1:]])
    w_shift = np.concatenate([w_opt[1:], w_opt[-1:]])
    u_shift = np.concatenate([u_opt[1:], u_opt[-1:]])
    z_guess = pack(th_shift, w_shift, u_shift)

    # apply control and step "true" system
    U_mpc[k] = u_k
    X_mpc[k + 1] = rk4_step(X_mpc[k], u_k, dt, (u_min, u_max))

```

```

if (k % 10 == 0) or (k == N_mpc_steps - 1):
    th, w = X_mpc[k + 1]
    print(f"Step {k}/{N_mpc_steps - 1} | theta={th:6.3f}, omega={w:6.3f}")

print("MPC simulation complete.")

# Plots
t_ctrl = t_mpc[:-1]

fig, axes = plt.subplots(3, 1, figsize=(10, 9), sharex=True)

axes[0].plot(t_mpc, X_mpc[:, 0], label="θ (MPC)")
axes[0].axhline(0.0, ls=':', c='gray', lw=1.0)
axes[0].axhline(np.pi, ls=':', c='gray', lw=1.0)
axes[0].set_ylabel("θ [rad]")
axes[0].legend(loc="upper right")
axes[0].grid(alpha=0.3)

axes[1].plot(t_mpc, X_mpc[:, 1], label="ω (MPC)")
axes[1].axhline(0.0, ls=':', c='gray', lw=1.0)
axes[1].set_ylabel("ω [rad/s]")
axes[1].legend(loc="upper right")
axes[1].grid(alpha=0.3)

axes[2].step(t_ctrl, U_mpc, where='post', label="τ (MPC)")
axes[2].axhline(u_min, ls='--', c='black', lw=0.8)
axes[2].axhline(u_max, ls='--', c='black', lw=0.8)
axes[2].set_ylabel("τ [N·m]")
axes[2].set_xlabel("Time [s]")
axes[2].legend(loc="upper right")
axes[2].grid(alpha=0.3)
axes[2].set_xlim(0, T_total)

plt.tight_layout()
plt.show()

```

MPC: 100 steps, dt=0.1, total time=10.0s
Step 0/99 | theta= 3.132, omega=-0.197
Step 10/99 | theta= 3.272, omega= 1.407
Step 20/99 | theta= 2.851, omega=-2.517
Step 30/99 | theta= 3.323, omega= 3.823
Step 40/99 | theta= 3.935, omega=-4.261
Step 50/99 | theta= 0.506, omega=-0.971
Step 60/99 | theta= 0.073, omega=-0.217
Step 70/99 | theta= 0.003, omega=-0.013
Step 80/99 | theta= 0.000, omega=-0.000
Step 90/99 | theta=-0.000, omega=-0.000
Step 99/99 | theta=-0.000, omega= 0.000
MPC simulation complete.

