Brunton and Kutz Problem 8.2: Optimal Control and Estimation of a Rotary Inverted Pendulum

Source Filename: /main.py

Rico A. R. Picone

First, import the necessary libraries:

```python
import numpy as np
import matplotlib.pyplot as plt
import control
from scipy import integrate
```

Next, define the system parameters:

```python
params = {
    'm1': 1.0,  # kg
    'm2': 1.0,  # kg
    'l1': 0.5,  # m
    'l2': 0.5,  # m
    'L1': 1.0,  # m
    'L2': 1.0,  # m
    'J0h': 1.0,  # kg*m^2
    'J2h': 1.0,  # kg*m^2
    'b1': 0.01,  # N*m*s/rad
    'b2': 0.01,  # N*m*s/rad
    'g': 9.81  # m/s^2
}

def unpack_params(params):
    """Unpack parameters dictionary into individual variables"""

    m1 = params['m1']
    m2 = params['m2']
    l1 = params['l1']
    l2 = params['l2']
    L1 = params['L1']
    L2 = params['L2']
    J0h = params['J0h']
    J2h = params['J2h']
    b1 = params['b1']
    b2 = params['b2']
    g = params['g']
    return m1, m2, l1, l2, L1, L2, J0h, J2h, b1, b2, g
```

Define the System Dynamics

Deriving the system dynamics of this system is rather involved and would distract from the main purpose of this exercise. We will use the system dynamics provided in the paper *On the Dynamics of the Furuta Pendulum* by Cazzolato and Prime.

The nonlinear system dynamics are given by equations (33) and (34). See figure 1 in the paper for the system diagram. We will assume that the center of mass of each arm is at the geometric center of the arm. Let the state vector be

$$x = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix},$$

Define the nonlinear system dynamics:

```
def rotary_inverted_pendulum(t, x, ufun, params):
    """Rotary inverted pendulum system dynamics

    From https://doi.org/10.1155/2011/528341, equations (33) and (34).
    State vector x = [theta1, theta2, theta1_dot, theta2_dot].
    """

    # Unpack parameters
    m1, m2, l1, l2, L1, L2, J0h, J2h, b1, b2, g = unpack_params(params)

    # Unpack states
    theta1, theta2, theta1_dot, theta2_dot = x

    # Unpack inputs
    tau1, tau2 = ufun(t, x)

    # Compute state derivatives and return
    theta_terms = np.array([[
            theta1_dot,
            theta2_dot,
            theta1_dot * theta2_dot,
            theta1_dot**2,
            theta2_dot**2
        ]]).T  # 5x1 matrix
```

```
forcing_terms = np.array([[
        tau1,
        tau2,
        g
    ]]).T  # 3x1 matrix
constant_factor = 1 / (
    J0h*J2h
    + J2h**2 * np.sin(theta2)**2
    - m2**2 * L1**2 * l2**2 * np.cos(theta2)**2
)
dtheta1_dt = theta1_dot
dtheta1_dot_dt = constant_factor  * (
    np.array([[
        -J2h*b1,
        m2*L1*l2 * np.cos(theta2) * b2,
        -J2h**2 * np.sin(2*theta2),
        -1/2 * J2h*m2*L1*l2 * np.cos(theta2) * np.sin(2*theta2),
        J2h*m2*L1*l2 * np.sin(theta2)
    ]]) @ theta_terms +
    np.array([[
        J2h,
        -m2*L1*l2*np.cos(theta2),
        1/2 * m2**2 * l2**2 * L1 * np.sin(2*theta2)
    ]]) @ forcing_terms
)
dtheta2_dt = theta2_dot
dtheta2_dot_dt = constant_factor * (
    np.array([[
        m2*L1*l2 * np.cos(theta2) * b1,
        -b2 * (J0h + J2h * np.sin(theta2)**2),
        m2*L1*l2*J2h * np.cos(theta2) * np.sin(2*theta2),
        -1/2 * np.sin(2*theta2) * (J0h*J2h + J2h**2 * np.sin(theta2)**2),
        -1/2 * m2*2*L1**2*l2 * np.sin(2*theta2)
    ]]) @ theta_terms +
    np.array([[
        -m2*L1*l2 * np.cos(theta2),
        J0h + J2h * np.sin(theta2)**2,
        -m2*l2*np.sin(theta2) * (J0h + J2h * np.sin(theta2)**2)
    ]]) @ forcing_terms
)
dx_dt = np.array([
    dtheta1_dt,
    dtheta2_dt,
    dtheta1_dot_dt.item(),
    dtheta2_dot_dt.item()
])
```

```
    return dx_dt
```

Define lists to label the states and inputs:

```
state_names = ["theta1", "theta2", "theta1_dot", "theta2_dot"]
state_labels = [r"$\theta_1$", r"$\theta_2$", r"$\dot{\theta}_1$", r"$\dot{\theta}_2$"]
input_names = ["tau1", "tau2"]
input_labels = [r"$\tau_1$", r"$\tau_2$"]
```

Define the linearized system dynamics. The paper provides the linearized system dynamics in equations (35) and (36) for the unstable upright equilibrium point

$$x_e = \begin{bmatrix} 0 \\ \pi \\ 0 \\ 0 \end{bmatrix}.$$

It includes torque inputs $\tau_1$ and $\tau_2$, but we will only consider $\tau_1$ in accordance with the problem statement. This equilibrium point can be derived by setting the nonlinear system dynamics to zero and solving for the state variables. The state vector for the linearized system is $x' = x - x_e$. Consider the following function:

```
def get_AB(params, tau1only=True, upright=True):
    """Linearized rotary inverted pendulum system dynamics

    From https://doi.org/10.1155/2011/528341, equations (35) and (36).
    State vector x = [theta1, theta2, theta1_dot, theta2_dot].
    Operating point is at the upright equilibrium (default) x = [0, pi, 0, 0]
    or the unstable equilibrium x = [0, 0, 0, 0] if upright=False.
    Input vector u = [tau1, tau2] or [tau1] if tau1only=True.
    """
    # Unpack parameters
    m1, m2, l1, l2, L1, L2, J0h, J2h, b1, b2, g = unpack_params(params)

    # Linearized system dynamics
    den = J0h*J2h - m2**2*L1**2*l2**2
    A31 = 0
    A32 = g * m2**2 * l2**2 * L1 / den
    A33 = -b1 * J2h / den
    A34 = -b2 * m2 * l2 * L1 / den
    A41 = 0
    A42 = g * m2 * l2 * J0h / den
    A43 = -b1 * m2 * l2 * L1 / den
    A44 = -b2 * J0h / den
    B31 = J2h / den
    B41 = m2 * L1 * l2 / den
    B32 = m2 * L1 * l2 / den
    B42 = J0h / den
    if upright:
        sign = 1
        get_AB.equilibrium = np.array([0, np.pi, 0, 0])
    else:
        sign = -1
        get_AB.equilibrium = np.array([0, 0, 0, 0])
    A = np.array([
        [0, 0, 1, 0],
        [0, 0, 0, 1],
        [A31, A32, A33, sign*A34],
        [A41, sign*A42, sign*A43, A44]
    ])
    if tau1only:  # Single input
        B = np.array([
            [0],
            [0],
            [B31],
            [sign*B41]
        ])
    else:
        B = np.array([
            [0, 0],
            [0, 0],
            [B31, sign*B32],
            [sign*B41, B42]
        ])
    return A, B
```

Controllability

Check the controllability of the linearized system. The system is controllable if the controllability matrix has full rank. The controllability matrix is given by

```
A, B = get_AB(params, tau1only=True)
Ctrb = control.ctrb(A, B)
rank_Ctrb = np.linalg.matrix_rank(Ctrb)
full_rank = rank_Ctrb == A.shape[0]
print(f'Controllability matrix rank: {rank_Ctrb}')
print(f'Full rank: {full_rank}')
```

```
Controllability matrix rank: 4
Full rank: True
```

Since the controllability matrix has full rank, the system is controllable. It would be more controllable if both torque inputs τ_1 and τ_2 were used, but it is remarkable that the system is controllable with only one input. The controllability gramian could be used to quantify the controllability of the system.

Full-State Feedback LQR Control

Define the LQR controller using the control library. Begin by defining the Q and R cost function matrices:

```
Q = np.diag([
    1,  # theta1 error cost
    10,  # theta2 error cost
    1,  # theta1 rate error cost
    1,  # theta2 rate error cost
])  # State error cost matrix
R = np.diag([
    1,  # tau1 cost
])  # Control effort cost matrix
```

Next, compute the LQR gain matrix:

```
A, B = get_AB(params=params, tau1only=True)
sys_lin = control.ss(A, B, np.eye(4), np.zeros((A.shape[0], B.shape[1])))
Kr, S, E = control.lqr(sys_lin, Q, R)
```

Define the control law function:

```
def lqr_control(x, x_command, Kr):
    """LQR full-state feedback control law

    Incorporate the command state x_command to compute the control input.
    """
    u = -Kr @ (x - x_command)
    return u
```

Simulation

Now simulate the nonlinear system with the LQR controller. Define the simulation function using scipy for integration:

```
def simulate_nonlinear_system(x0, t_sim, params, ufun):
    """Simulate the nonlinear rotary inverted pendulum system

    Use the scipy `solve_ivp` function to simulate the system dynamics.
    """
    x_sim = integrate.solve_ivp(
        rotary_inverted_pendulum,
        t_span=(t_sim[0], t_sim[-1]),
        y0=x0,
        t_eval=t_sim,
        args=(ufun, params),
    )
    return x_sim
```

Define the simulation parameters:

```
t_sim = np.linspace(0, 10, 1000)  # Simulation time
x0 = np.array([-4, 184, 0, 0]) * np.pi/180  # Initial state
def ufun(t, x, tau1only=True):  # Control input function
    ufun.x_command = np.array([np.pi/3, np.pi, 0, 0])  # Command state (static)
    u =  lqr_control(x, ufun.x_command, Kr)
    if tau1only:
        tau2 = np.zeros_like(u)  # Zero out tau2
        return np.hstack((u, tau2))
    else:
        return u
```

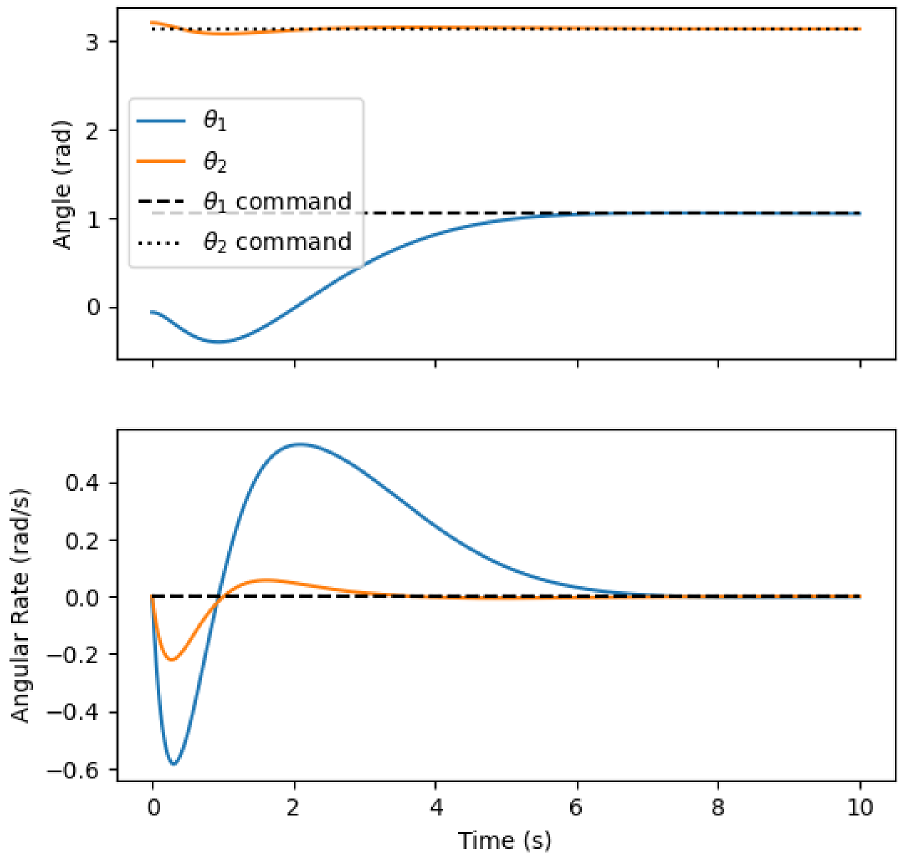
Simulate the response:

```
x_sim = simulate_nonlinear_system(x0, t_sim, params, ufun)
```

Plot the Closed-Loop Response

Plot the response of the system states and control inputs:

```
fig, ax = plt.subplots(2, 1, figsize=(6, 6), sharex=True)
ax[0].plot(t_sim, x_sim.y[0], label=r'$\theta_1$')
ax[0].plot(t_sim, x_sim.y[1], label=r'$\theta_2$')
ax[0].plot(t_sim, ufun.x_command[0] * np.ones_like(t_sim), 'k--', label=r'$\theta_1$ command
ax[0].plot(t_sim, ufun.x_command[1] * np.ones_like(t_sim), 'k:', label=r'$\theta_2$ command'
ax[0].set_ylabel('Angle (rad)')
ax[0].legend()
ax[1].plot(t_sim, x_sim.y[2], label=r'$\dot{\theta}_1$')
ax[1].plot(t_sim, x_sim.y[3], label=r'$\dot{\theta}_2$')
ax[1].plot(t_sim, ufun.x_command[2] * np.ones_like(t_sim), 'k--', label=r'$\dot{\theta}_1$ c
ax[1].plot(t_sim, ufun.x_command[3] * np.ones_like(t_sim), 'k:', label=r'$\dot{\theta}_2$ co
ax[1].set_ylabel('Angular Rate (rad/s)')
ax[1].set_xlabel('Time (s)')
plt.draw()
```



The response shows that the system is able to stabilize around the commanded state.

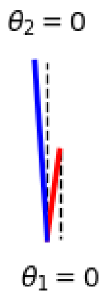
Visualize the rotary inverted pendulum response as an animation:

```python
from matplotlib.animation import FuncAnimation
from matplotlib.patches import Rectangle

def animate_rotary_pendulum(t, x, params, track_theta2=False):
    """Animate the rotary inverted pendulum response"""

    # Unpack parameters
    m1, m2, l1, l2, L1, L2, J0h, J2h, b1, b2, g = unpack_params(params)

    # Create the figure and axis
    fig, ax = plt.subplots()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_aspect('equal')
    ax.axis('off')

    # Initialize the plot elements
    rod01, = ax.plot([], [], 'k--', lw=1)
    rod02, = ax.plot([], [], 'k--', lw=1)
    rod1, = ax.plot([], [], 'r', lw=2)
    rod2, = ax.plot([], [], 'b', lw=2)
    text_theta1 = ax.annotate(
        text=r'$\theta_1 = 0$',
        xy=(0, -L1/2), xycoords='data',
        xytext=(20, -20), textcoords='offset pixels',
        ha='center', va='center'
    )
    text_theta2 = ax.annotate(
        text=r'$\theta_2 = 0$',
        xy=(0, -L2/2), xycoords='data',
        xytext=(20, -20), textcoords='offset pixels',
        ha='center', va='center'
    )

    # Update function for the animation
    def update(i):
        theta1 = x[0, i]
        theta2 = x[1, i]
        x0 = 0
        y0 = 0
        if track_theta2:
            x_theta1 = -L1 * np.sin(theta1)
            y_theta1 = -L1/2 * np.cos(theta1)
            y0d = -L1/2
            x1 = L1 * np.sin(theta1)
            y1 = -L1/2 * np.cos(theta1)
            x2 = x1 + L2 * np.sin(theta2) - x1
            y2 = -L2 * np.cos(theta2) + y0d
            rod01.set_data([x0, -x1], [y0, y1])
            rod02.set_data([x0, x0], [y0d, y0d+L2])
            rod1.set_data([x0, x0], [y0, y0d])
            rod2.set_data([x0, x2], [y0d, y2])
            text_theta1.xy = (x_theta1, y_theta1)
            text_theta1.set_position((-45*np.sin(theta1), -45*np.cos(theta1)))
            text_theta2.xy = (x0, y0d+L2)
            text_theta2.set_position((x0, y0+L2+20))
            return rod01, rod02, rod1, rod2, text_theta1, text_theta2
        else:
            y_theta1 = -L1/2
```

```
            x1 = L1 * np.sin(theta1)
            y1 = -L1/2 * np.cos(theta1)
            x2 = x1 + L2 * np.sin(theta2)
            y2 = y1 - L2 * np.cos(theta2)
            rod01.set_data([x0, x0], [y0, y_theta1])
            rod02.set_data([x1, x1], [y1, y2])
            rod1.set_data([x0, x1], [y0, y1])
            rod2.set_data([x1, x2], [y1, y2])
            text_theta1.xy = (x0, y_theta1)
            text_theta1.set_position((0, -20))
            text_theta2.xy = (x1, y2)
            text_theta2.set_position((x1, y2+20))
            return rod01, rod02, rod1, rod2, text_theta1, text_theta2

    # Create the animation
    anim = FuncAnimation(fig, update, frames=range(len(t)), blit=True, interval=6)
    return anim
```

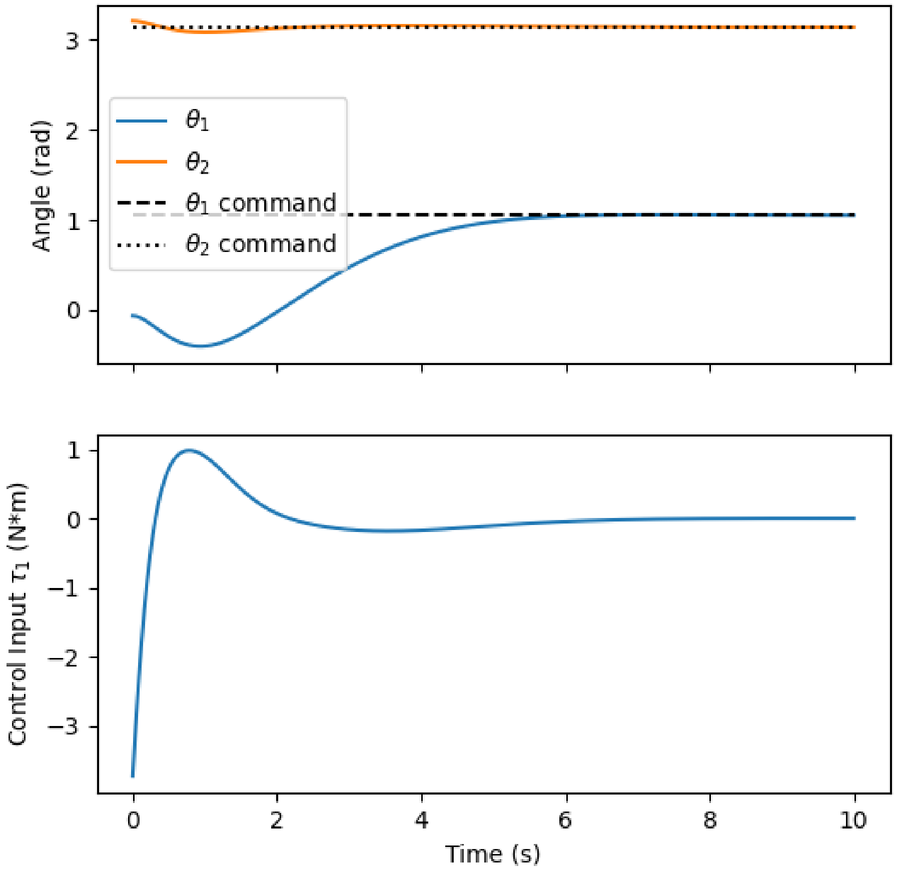
Animate the response:

```
anim = animate_rotary_pendulum(t_sim, x_sim.y, params, track_theta2=False)
plt.draw()
```



Now plot the control inputs over time along with the angular position states:

```
u_sim = np.array([ufun(t, x) for t, x in zip(t_sim, x_sim.y.T)])
fig, ax = plt.subplots(2, 1, figsize=(6, 6), sharex=True)
ax[0].plot(t_sim, x_sim.y[0], label=r'$\theta_1$')
ax[0].plot(t_sim, x_sim.y[1], label=r'$\theta_2$')
ax[0].plot(t_sim, ufun.x_command[0] * np.ones_like(t_sim), 'k--', label=r'$\theta_1$ comman
ax[0].plot(t_sim, ufun.x_command[1] * np.ones_like(t_sim), 'k:', label=r'$\theta_2$ command'
ax[0].set_ylabel('Angle (rad)')
ax[0].legend()
ax[1].plot(t_sim, u_sim[:,0], label=r'$\tau_1$')
ax[1].set_ylabel(r'Control Input $\tau_1$ (N*m)')
ax[1].set_xlabel('Time (s)')
plt.draw()
```



Observability

Explore the observability of the system. We carefully consider the observability of three different sensor configurations:

1. Observe θ_1 and θ_2
2. Observe θ_1 and $\dot{\theta}_2$
3. Observe $\dot{\theta}_1$ and $\dot{\theta}_2$

Define the output equation C matrix for each sensor configuration (and a couple of others):

```python
def get_C(params, sensor_config):
    """C matrix for the rotary inverted pendulum system

    Define the C matrix for different sensor configurations.
    """
    if sensor_config == 1:
        C = np.array([
            [1, 0, 0, 0],  # theta1
            [0, 1, 0, 0]  # theta2
        ])
    elif sensor_config == 2:
        C = np.array([
            [1, 0, 0, 0],  # theta1
            [0, 0, 0, 1]  # theta2_dot
        ])
    elif sensor_config == 3:
        C = np.array([
            [0, 0, 1, 0],  # theta1_dot
            [0, 0, 0, 1]  # theta2_dot
        ])
    elif sensor_config == 4:
        C = np.array([
            [1, 0, 0, 0],  # theta1
            [0, 1, 0, 0],  # theta2
            [0, 0, 0, 1]  # theta2_dot
        ])
    elif sensor_config == 5:
        C = np.array([
            [1, 0, 0, 0],  # theta1
            [0, 1, 0, 0],  # theta2
            [0, 0, 1, 0],  # theta1_dot
            [0, 0, 0, 1]  # theta2_dot
        ])
    return C
```

Define the observability matrix for each sensor configuration:

```
sensor_configs = [1, 2, 3]
for sensor_config in sensor_configs:
    C = get_C(params, sensor_config)
    Obsv = control.obsv(A, C)
    rank_Obsv = np.linalg.matrix_rank(Obsv)
    full_rank = rank_Obsv == A.shape[0]
    print(f'Sensor Configuration {sensor_config}')
    print(f'\tObservability matrix rank: {rank_Obsv}')
    print(f'\tFull rank: {full_rank}')
```

```
Sensor Configuration 1
Observability matrix rank: 4
Full rank: True
Sensor Configuration 2
Observability matrix rank: 4
Full rank: True
Sensor Configuration 3
Observability matrix rank: 3
Full rank: False
```

Of the three considered, the system is observable for sensor configurations 1 and 2 only, but not for 3. Now explore which of sensor configurations 1 and 2 is more observable. We can compute the observability Gramian for each configuration and compare its eigenvalues. The control library function `control.gram()` can compute the observability Gramian. However, it can only do so for stable systems. This is clearly out for the unstable equilibrium. The downward equilibrium point $x_e = [0, 0, 0, 0]$ is only marginally stable, so we can't use it, either. Let's try nudging the eigenvalues of the downward A matrix to the left, slightly, to make it stable.

```
As, Bs = get_AB(params, tau1only=True, upright=False)
print(f'Eigenvalues of A matrix: {np.linalg.eigvals(As)}')
As = As - 0.0001 * np.eye(4)  # Nudge the eigenvalues to the left
print(f'Eigenvalues of nudged A matrix: {np.linalg.eigvals(As)}')
for sensor_config in sensor_configs:
    C = get_C(params, sensor_config)
    sys_lin_down = control.ss(As, Bs, C, np.zeros((C.shape[0], B.shape[1])))
    W = control.gram(sys_lin_down, 'o')
    print(f'Sensor Configuration {sensor_config} (Downward Equilibrium)')
    print(f'\tObservability Gramian Eigenvalues: {np.linalg.eigvals(W)}')
    print(f'\tObservability Gramian Determinant: {np.linalg.det(W):.3e}')
```

```
Eigenvalues of A matrix: [ 0.         +0.j          -0.00833331+2.55732228j -0.00833331-2.5573
 -0.01000005+0.j         ]
Eigenvalues of nudged A matrix: [-0.0001    +0.j          -0.00843331+2.55732228j -0.0084333
 -0.01010005+0.j         ]
Sensor Configuration 1 (Downward Equilibrium)
Observability Gramian Eigenvalues: [6.06727904e+07 7.36721282e+01 2.46557254e+01 4.53238470e
Observability Gramian Determinant: 4.995e+11
Sensor Configuration 2 (Downward Equilibrium)
Observability Gramian Eigenvalues: [6.06727954e+07 2.17114083e+02 4.54439315e+01 2.46210052e
Observability Gramian Determinant: 1.474e+13
Sensor Configuration 3 (Downward Equilibrium)
Observability Gramian Eigenvalues: [ 24.71576285  74.22073516 242.34178445   0.        ]
Observability Gramian Determinant: 0.000e+00
```

We see that the observability Gramian for sensor configuration 2 has a higher determinant than for configuration 1. Therefore, sensor configuration 2 is more observable than configuration 1. However, both configurations are observable. We also observe that the observability Gramian for configuration 3 is singular, which is another way of determining that it is not observable for that configuration.

As a practical matter, it is easier to measure angular position than angular velocity. Below, we will consider sensor configuration 5, which is the most observable configuration because it measures all four states.

Full-State Observer Design

Define the observer gain matrix using the control library. Larger values in the observer weighting matrices V_d and V_n will make the observer more sensitive to disturbances (i.e., process noise) and measurement noise, respectively. The linear state equation assumed for the observer design is

$$\begin{aligned}\dot{x}' &= Ax' + Bu + Gw_d \\ y &= Cx' + Du + w_n\end{aligned}$$

where w_d is the disturbance input, G is the disturbance input matrix, and w_n is the measurement noise. The observer dynamics are given by

$$\dot{\hat{x}}' = \left(A - K_f C - (B - K_f D)K_r\right)\hat{x}' + K_f y,$$

where $\hat{x}'$ is the state estimate (in the coordinates of the linearized system), K_f is the observer gain matrix, and K_r is the control gain matrix.

The observer gain matrix K_f can be computed using the control library function `control.lqe()` as follows:

```python
A, B = get_AB(params, tau1only=True)
C = get_C(params, sensor_config=5)
n_states = A.shape[0]  # Number of states
n_inputs = B.shape[1]  # Number of inputs
n_outputs = C.shape[0]  # Number of outputs
n_disturbances = n_states  # Number of disturbance inputs
n_noises = n_outputs  # Number of measurement noise inputs
D = np.zeros((n_outputs, n_inputs))
```

```
stdd = 1e1  # Standard deviation of disturbances
stdn = 1e-3  # Standard deviation of measurement noise
Vd = stdd**2 * np.diag(np.ones((n_disturbances,)))  # Disturbance covar
Vn = stdn**2 * np.diag(np.ones((n_noises,)))  # Measurement noise covar
G = np.eye(n_states)  # Disturbance input matrix
Kf, P, E = control.lqe(A, G, C, Vd, Vn)
```

LQG Control

Define the LQG controller function. The LQG controller combines the LQR controller with the LQE observer. The control input is computed as

$$u = -K_f \hat{x}.$$

The observer dynamics have already been defined, above. The LQG controller, then, is a dynamic system with input y, internal state $\hat{x}$, and output u. However, the dynamical equation is in terms of the estimated state of the linearized system $\hat{x}'$, not $\hat{x} = \hat{x}' + x_e$. To deploy the LQG controller, we will need to augment the dynamical equation to account for the equilibrium offset.

The approach we use is to add the equilibrium offset to the observer state, resulting in twice the number of states. The equilibrium-augmented observer dynamics are given by

$$\frac{d}{dt}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix} = \begin{bmatrix}A_c & A_c \\ 0 & 0\end{bmatrix}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix} + \begin{bmatrix}K_f \\ 0\end{bmatrix} y,$$

where $A_c = A - K_f C - (B - K_f D)K_r$, the original observer dynamics matrix, and the zeros are appropriately sized 0-matrices. The corresponding output equation is

$$u = \begin{bmatrix}-K_f & 0\end{bmatrix}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix}.$$

Note that the equilibrium offset x_e is not affected by the observer dynamics (i.e., it doesn't change). Now the dynamics have been corrected for the equilibrium offset. Note that this would unnecessary if the equilibrium offset were zero.

A second augmentation is needed to introduce the command state x_r. We cannot use the same approach as we did for the equilibrium offset because we would like to be able to vary the command state. The approach we use is similar to the one used for the LQR controller: we augment the input vector with the command state. The command-and-equilibrium-augmented observer dynamics are given by

$$\frac{d}{dt}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix} = \begin{bmatrix}A_c & A_c \\ 0 & 0\end{bmatrix}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix} + \begin{bmatrix}K_f & 0 \\ 0 & 0\end{bmatrix}\begin{bmatrix}y \\ x_r\end{bmatrix}.$$

So the dynamics themselves are not affected by the command state. Its effects are only seen in the output equation, given by

$$u = \begin{bmatrix}-K_f & 0\end{bmatrix}\begin{bmatrix}\hat{x}' \\ x_e\end{bmatrix} + \begin{bmatrix}0 & K_f\end{bmatrix}\begin{bmatrix}y \\ x_r\end{bmatrix}.$$

When the plant is actually linear, the closed-loop system can be represented as a single state-space system. In our case, the plant is nonlinear, so we will need to model the LQG controller as a separate system, which will be used to compute the control input u at each time step. Perhaps the easiest way to do this is to use the control library to create a state-space system for the plant and the LQG controller, then connect them with the `control.interconnect()` function. Begin by defining the LQR controller as follows:

```
Q = np.diag([
    1e4,  # theta1 error cost
    1e4,  # theta2 error cost
    1,  # theta1 rate error cost
    1,  # theta2 rate error cost
])  # State error cost matrix
R = np.diag([
    1e0,  # tau1 cost
])  # Control effort cost matrix
sys_lin = control.ss(A, B, C, D)  # Ignore augmented states for now
Kr, S, E = control.lqr(sys_lin, Q, R)
```

Define the LQG controller system using the control library. In addition to the equilibrium and command augmentations, there is a third augmentation in the following to account for the fact that our model of the plant as having two inputs τ_1 and τ_2, but we are only using τ_1 (i.e., we must zero out τ_2):

```
Ac = A - Kf @ C - (B - Kf @ D) @ Kr  # LQG controller A matrix
Ac = np.vstack([
    np.hstack([Ac, Ac]),
    np.zeros((n_states, 2*n_states))  # Augment for equilibrium offset
])  # LQG controller A matrix augmented
Bc = np.hstack([
    Kf,  # Actual LQG controller B matrix
    np.zeros((n_states, n_states)),  # Augment for the command "input"
])  # LQG controller B matrix augmented
Bc = np.vstack([
    Bc,
    np.zeros((n_states, Bc.shape[1]))  # Augment for equilibrium offset
])  # LQG controller B matrix augmented
Cc1 = np.vstack([
    -Kr,  # Actual LQG controller C matrix
    np.zeros((1, n_states)),  # Zero out tau2
])  # LQG controller C matrix partially augmented
Cc = np.hstack([
    Cc1,
    np.zeros((2, n_states)),  # Augmented for the equilibrium offset
])  # LQG controller C matrix augmented
Dc = np.hstack([
    np.zeros((2, n_outputs)),  # LQG controller D matrix, zero out tau2
    -Cc1,  # Augmented for the command "input"
])  # LQG controller D matrix augmented
sysc = control.ss(Ac, Bc, Cc, Dc, name='sysc')  # LQG controller system
sysc.set_inputs(n_outputs+n_states, prefix='yre')
sysc.set_outputs(2, prefix='u')  # Control output (i.e., tau1, tau2)
sysc.set_states(2*n_states, prefix='x_hat')  # Observer state
print(sysc)
```

```
<StateSpace>: sysc
Inputs (8): ['yre[0]', 'yre[1]', 'yre[2]', 'yre[3]', 'yre[4]', 'yre[5]', 'yre[6]', 'yre[7]'
Outputs (2): ['u[0]', 'u[1]']
States (8): ['x_hat[0]', 'x_hat[1]', 'x_hat[2]', 'x_hat[3]', 'x_hat[4]', 'x_hat[5]', 'x_hat

A = [[-1.00000000e+04 -4.08750488e-05  5.00000655e-01  3.06973731e-07
      -1.00000000e+04 -4.08750488e-05  5.00000655e-01  3.06973731e-07]
     [-4.08750488e-05 -9.99999953e+03 -1.63499948e+00 -2.76999896e+00
      -4.08750488e-05 -9.99999953e+03 -1.63499948e+00 -2.76999896e+00]
     [ 1.32833334e+02 -7.30248876e+02 -9.86019747e+03 -3.22664562e+02
       1.32833334e+02 -7.30248876e+02 -9.86019747e+03 -3.22664562e+02]
     [ 6.66666669e+01 -3.63171937e+02  6.99006179e+01 -1.01613336e+04
       6.66666669e+01 -3.63171937e+02  6.99006179e+01 -1.01613336e+04]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

B = [[ 1.00000000e+04  4.08750488e-05  4.99999345e-01 -3.06973731e-07
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 4.08750488e-05  9.99999953e+03  1.63499948e+00  3.76999896e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 4.99999345e-01  1.63499948e+00  9.99998706e+03 -5.82381556e-03
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [-3.06973731e-07  3.76999896e+00 -5.82381556e-03  9.99998842e+03
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
       0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
     [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
```

```
         0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]

C = [[  99.99999989 -548.91290742  104.85219119 -241.99778913    0.
          0.            0.            0.        ]
     [   0.            0.            0.            0.            0.
          0.            0.            0.        ]]

D = [[   0.            0.            0.            0.          -99.99999989
        548.91290742 -104.85219119  241.99778913]
     [   0.            0.            0.            0.           -0.
         -0.           -0.           -0.        ]]
```

Now create a `control.InputOutputSystem` object for the nonlinear plant. Before we can do that, we need to define a plant function that can incorporate disturbances. Consider the following:

```python
def rotary_inverted_pendulum_disturbed(t, x, v, params):
    """Rotary inverted pendulum system dynamics with disturbances

    Input vector v = [u, w_d, w_n].
    State vector x = [theta1, theta2, theta1_dot, theta2_dot].
    """

    n_inputs = 2  # Number of control inputs (tau1, tau2)
    n_disturbances = 4  # Number of disturbance inputs
    # Get the undisturbed state derivatives
    def ufun(t, x):
        return v[:n_inputs]
    dx_dt = rotary_inverted_pendulum(t, x, ufun, params)

    # Add the disturbance terms
    G = np.eye(x.shape[0])  # Disturbance input matrix
        # Each disturbance input affects the corresponding state
    dx_dt += G @ v[n_inputs:n_inputs+n_disturbances]

    return dx_dt
```

Additionally, we need a corresponding output function to incorporate measurement noise. Consider the following:

```python
def output_function_noised(t, x, v, params):
    """Output function for the rotary inverted pendulum system

    Output function for the rotary inverted pendulum system with
    measurement noise.
    """
    n_noises = n_outputs  # Number of measurement noise inputs
    # TODO - Should be passing C and D as arguments
    y = C @ x + D @ v[0:n_inputs] + v[-n_noises:]
    return y
```

Now create the `control.InputOutputSystem` object for the nonlinear plant:

```
sys_plant = control.NonlinearIOSystem(
    rotary_inverted_pendulum_disturbed,
    outfcn=output_function_noised,
    name='sys_plant',
    params=params
)
sys_plant.set_inputs(2 + n_disturbances + n_noises, prefix='v')  # Augmented input
sys_plant.set_states(n_states, prefix='x')  # State
sys_plant.set_outputs(n_outputs, prefix='y')  # Output
print(sys_plant)
```

```
<NonlinearIOSystem>: sys_plant
Inputs (10): ['v[0]', 'v[1]', 'v[2]', 'v[3]', 'v[4]', 'v[5]', 'v[6]', 'v[7]', 'v[8]', 'v[9]
Outputs (4): ['y[0]', 'y[1]', 'y[2]', 'y[3]']
States (4): ['x[0]', 'x[1]', 'x[2]', 'x[3]']

Update: <function rotary_inverted_pendulum_disturbed at 0x15f2ca020>
Output: <function output_function_noised at 0x15f2c84a0>
```

Now we can connect the LQG controller to the nonlinear plant using the `control.interconnect()` function as follows:

```
cl_outputs_y = [f'y[{i}]' for i in range(n_outputs)]
cl_outputs_u = [f'u[{i}]' for i in range(2)]
cl_outputs = cl_outputs_y + cl_outputs_u
sys_cl = control.interconnect(
    syslist=[sys_plant, sysc],
    connections=[
        [f'sys_plant.v[0:2]', 'sysc.u'],  # Connect plant input
        [f'sysc.yre[0:{n_outputs}]', 'sys_plant.y'],  # Connect control input
    ],  # Other internal connections are connected by name
    inplist=[
        f'sysc.yre[{n_outputs}:]', # xc
        f'sys_plant.v[2:]',  # [wd, wn]
    ],  # External inputs
    inputs=n_states + n_disturbances + n_noises,
    outputs=cl_outputs,  # External outputs
    name='sys_cl'
)
print(sys_cl)
```

```
<InterconnectedSystem>: sys_cl
```

```
Inputs (12): ['u[0]', 'u[1]', 'u[2]', 'u[3]', 'u[4]', 'u[5]', 'u[6]', 'u[7]', 'u[8]', 'u[9]'
Outputs (6): ['y[0]', 'y[1]', 'y[2]', 'y[3]', 'u[0]', 'u[1]']
States (12): ['sys_plant_x[0]', 'sys_plant_x[1]', 'sys_plant_x[2]', 'sys_plant_x[3]', 'sysc_

Update: <function InterconnectedSystem.__init__.<locals>.updfcn at 0x15f2c8cc0>
Output: <function InterconnectedSystem.__init__.<locals>.outfcn at 0x15f2ca0c0>
```

Simulation with LQG Control

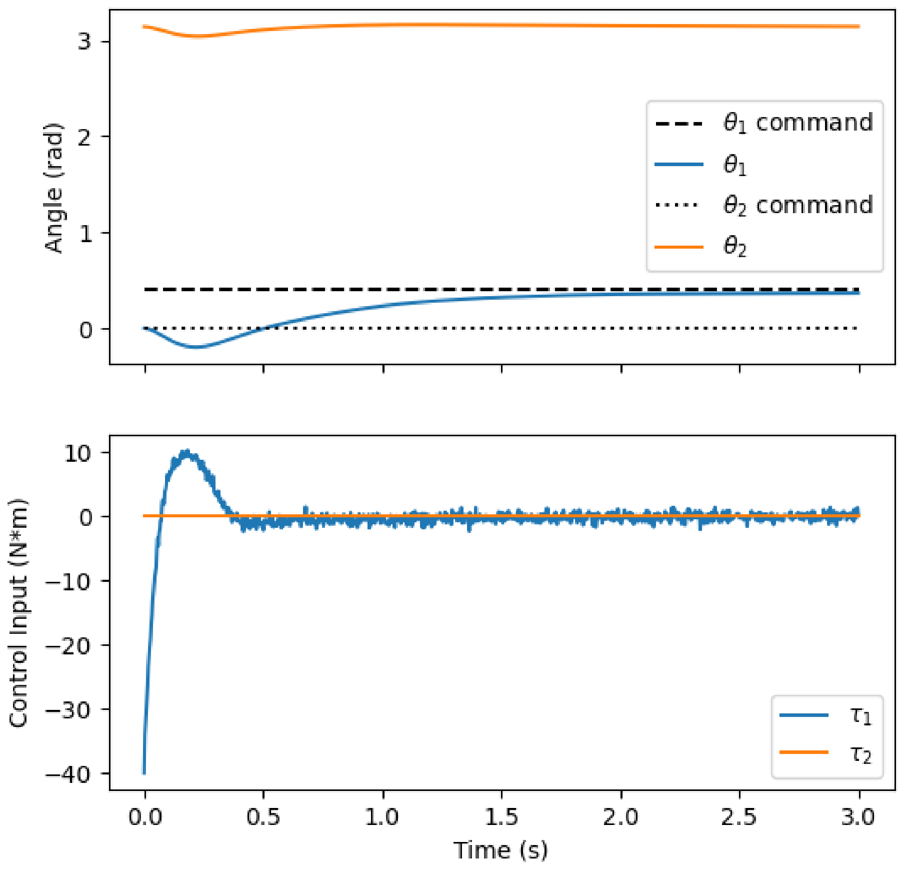
Simulate the closed-loop, LQG-controlled system using the `control.forced_response()` function:

```
t_sim = np.linspace(0, 3, 1000)  # Simulation time
xe = np.array([0, np.pi, 0, 0])  # Equilibrium state
x0 = xe  # Initial state
x0_hat_prime = np.array([0, 0, 0, 0])  # Initial observer state
command_inputs = np.vstack([
    0.4*np.ones_like(t_sim),  # theta1 command
    np.pi*np.ones_like(t_sim),  # theta2 command
    np.zeros_like(t_sim),  # theta1_dot command
    np.zeros_like(t_sim)  # theta2_dot command
])  # Command inputs in original coordinates
command_inputs = command_inputs - np.atleast_2d(xe).T  # Lin. coord's
disturbance_inputs = 5e-3*np.random.randn(n_states, len(t_sim))
noise_inputs = stdn*np.random.randn(n_noises, len(t_sim))
all_inputs = np.vstack([
    command_inputs,
    disturbance_inputs,
    noise_inputs
])  # All inputs
lqr_response = control.forced_response(
    sys_cl,
    T=t_sim,
    U=all_inputs,
    X0=np.concatenate([x0, x0_hat_prime, xe]),
)
y = lqr_response.outputs
x = lqr_response.states
x_hat_prime = x[n_states:2*n_states]  # Observer states
x_hat = x_hat_prime + np.atleast_2d(xe).T  # In original coordinates
```

Plot the response of the system states and control inputs:

```
fig, ax = plt.subplots(2, 1, figsize=(6, 6), sharex=True)
for i, label in enumerate(state_labels[0:2]):
    ax[0].plot(t_sim, command_inputs[i], f"k{['--',':'][i]}", label=f'{label} command')
    ax[0].plot(t_sim, x[i], label=label)  # theta1, theta2
```

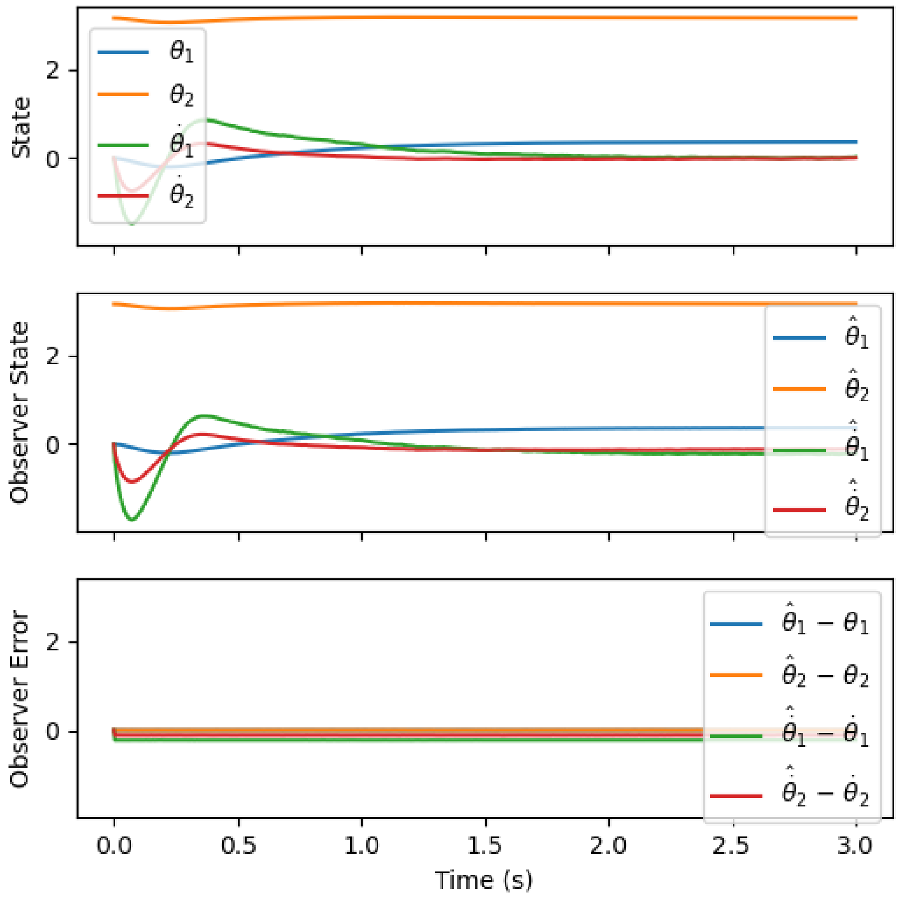
```
ax[0].set_ylabel('Angle (rad)')
ax[0].legend()
for i, label in enumerate(input_labels):
    ax[1].plot(t_sim, y[n_outputs+i], label=label)
ax[1].set_ylabel('Control Input (N*m)')
ax[1].set_xlabel('Time (s)')
ax[1].legend()
plt.draw()
```



Plot the plant states and observer states:

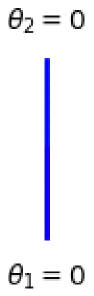
```
state_hat_labels = [
    r'$\hat{\theta}_1$', r'$\hat{\theta}_2$',
    r'$\hat{\dot{\theta}}_1$', r'$\hat{\dot{\theta}}_2$'
]
fig, ax = plt.subplots(3, 1, figsize=(6, 6), sharex=True, sharey=True)
for i, label in enumerate(state_labels):
    ax[0].plot(t_sim, x[i], label=label)
```

```
ax[0].set_ylabel('State')
ax[0].legend()
for i, label in enumerate(state_hat_labels):
    ax[1].plot(t_sim, x_hat[i], label=label)
ax[1].set_ylabel('Observer State')
ax[1].legend()
for i, label in enumerate(state_labels):
    ax[2].plot(t_sim, x_hat[i] - x[i], label=f'{state_hat_labels[i]} $-$ {label}')
ax[2].set_ylabel('Observer Error')
ax[2].set_xlabel('Time (s)')
ax[2].legend()
plt.draw()
```



Animate the response:

```
anim = animate_rotary_pendulum(t_sim, x, params, track_theta2=False)
plt.show()
```


$\theta_2 = 0$
$\theta_1 = 0$