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

Source Filename: /main.py

Rico A. R. Picone

First, import the necessary libraries:

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

Next, define the system parameters:

```
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': 0.1,  # N*m*s/rad
    'b2': 0.1,  # 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 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 τ_1 and τ_2, but we will only consider τ_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.

```
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
    else:
        sign = -1
    if tau1only:  # Single input
        A = np.array([
```

```
            [0, 0, 1, 0],
            [0, 0, 0, 1],
            [A31, A32, A33, sign*A34],
            [A41, sign*A42, sign*A43, A44]
        ])
        B = np.array([
            [0],
            [0],
            [B31],
            [sign*B41]
        ])
    else:
        A = np.array([
            [0, 0, 1, 0],
            [0, 0, 0, 1],
            [A31, A32, A33, sign*A34],
            [A41, sign*A42, sign*A43, A44]
        ])
        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.

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

# [markdown]
# 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])))
K, S, E = control.lqr(sys_lin, Q, R)

#% [markdown]
# Define the control law function:

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

    Incorporate the command state x_command to compute the control input.
    """
    u = -K @ (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, x_command, 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([-40, 199, 0, 0]) * np.pi/180  # Initial state
u0 = np.array([0, 0])  # Initial control input
x_command = np.array([np.pi/3, np.pi, 0, 0])  # Command state (static)
def ufun(t, x, tau1only=True):  # Control input function
    u =  lqr_control(x, x_command, K)
    if tau1only:
        tau2 = np.zeros_like(u)  # Zero out tau2
        return np.hstack((u, tau2))
    else:
        return u
```

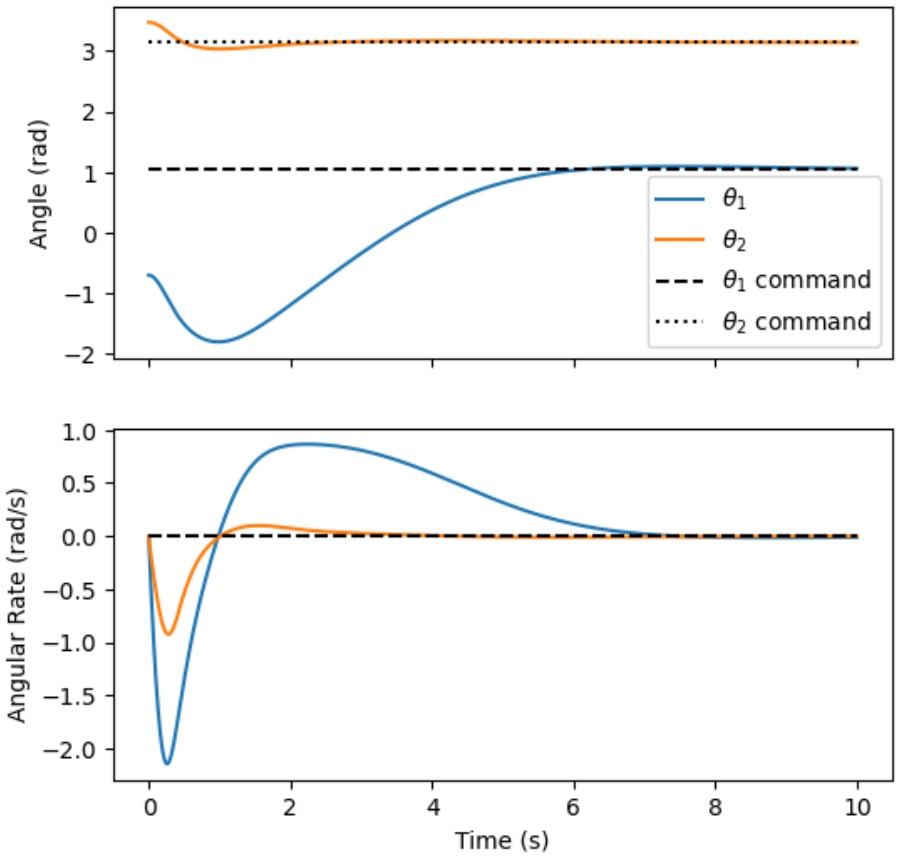
Simulate the response:

```
x_sim = simulate_nonlinear_system(x0, t_sim, x_command, 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, x_command[0] * np.ones_like(t_sim), 'k--', label=r'$\theta_1$ command')
ax[0].plot(t_sim, 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, x_command[2] * np.ones_like(t_sim), 'k--', label=r'$\dot{\theta}_1$ commar
ax[1].plot(t_sim, x_command[3] * np.ones_like(t_sim), 'k:', label=r'$\dot{\theta}_2$ commanc
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:

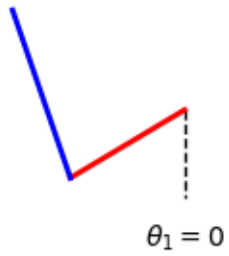
```
from matplotlib.animation import FuncAnimation
from matplotlib.patches import Rectangle

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
rod0, = 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'
)

# 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
        rod0.set_data([x0, -x1], [y0, y1])
        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)))
        return rod0, rod1, rod2, text_theta1
    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)
        rod0.set_data([x0, x0], [y0, y_theta1])
        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))
        return rod0, rod1, rod2, text_theta1
```

```
    # 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()
```

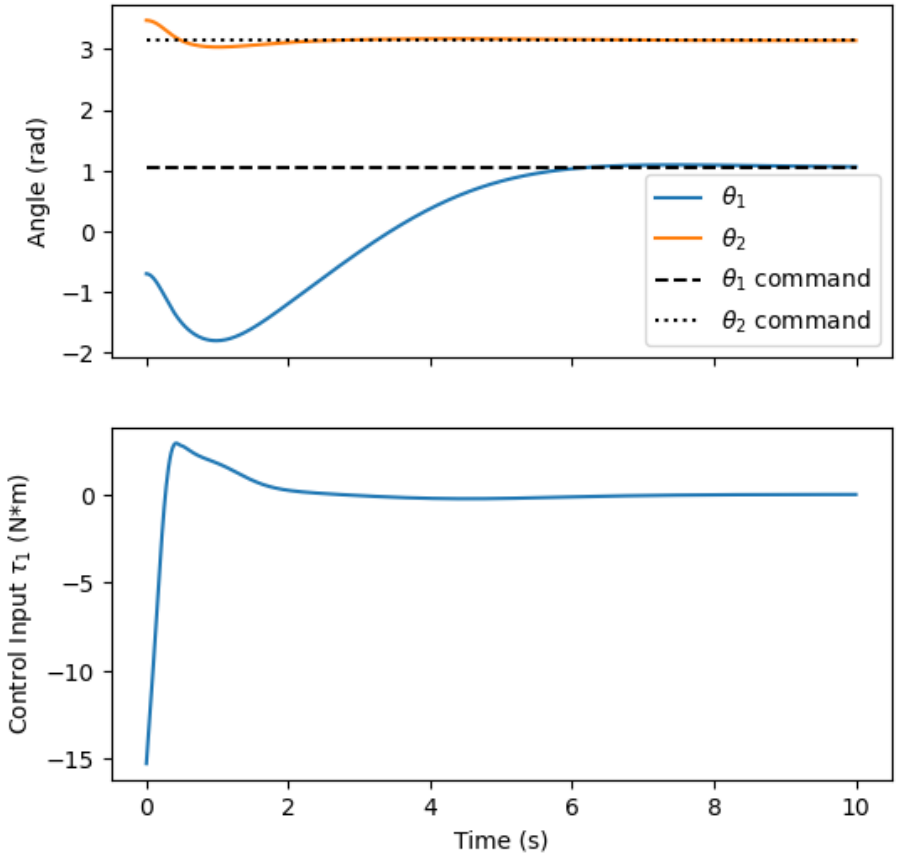


Now plot the control inputs over time along with the angular position states:

```
u_sim = np.array([ufun(t, x) for t, x in zip(t_sim, x_sim.y.T)])
print(u_sim.shape)
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, x_command[0] * np.ones_like(t_sim), 'k--', label=r'$\theta_1$ command')
ax[0].plot(t_sim, 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.show()
```

```
(1000, 2)
```



Observability

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

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

Define the output equation C matrix for each sensor configuration:

```
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],
            [0, 1, 0, 0]
        ])
    elif sensor_config == 2:
        C = np.array([
            [0, 0, 1, 0],
            [0, 0, 0, 1]
        ])
    elif sensor_config == 3:
        C = np.array([
            [1, 0, 0, 0],
            [0, 0, 1, 0]
        ])
    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: 3
Full rank: False
Sensor Configuration 3
Observability matrix rank: 4
Full rank: True
```

So the system is observable for sensor configurations 1 and 3 only, but not for 2. Therefore, configuration 2 is not as observable as the other two configurations.

Now explore which of sensor configurations 1 and 3 is more observable. We can compute the observability Gramian for each configuration and compare the eigenvalues. The control library function `control.gram()` can compute the observability Gramian. However, it can only do so for stable systems. The

stable 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 A matrix to the left, slightly, to make it stable.

```
sensor_configs = [1, 2, 3]
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         -0.08330781+2.55533228j -0.08330781-2.5553
 -0.10005105+0.j        ]
Eigenvalues of nudged A matrix: [-1.00000000e-04+0.j         -8.34078101e-02+2.55533228j
 -8.34078101e-02-2.55533228j -1.00151046e-01+0.j        ]
Sensor Configuration 1 (Downward Equilibrium)
Observability Gramian Eigenvalues: [6.28125222e+05 7.45701348e+00 2.50665469e+00 4.54469532e
Observability Gramian Determinant: 5.336e+06
Sensor Configuration 2 (Downward Equilibrium)
Observability Gramian Eigenvalues: [ 2.49713154  7.49210409 24.49549161  0.        ]
Observability Gramian Determinant: 0.000e+00
Sensor Configuration 3 (Downward Equilibrium)
Observability Gramian Eigenvalues: [6.28131466e+05 8.56847847e+00 3.31281002e+00 6.88638688e
Observability Gramian Determinant: 1.228e+07
```

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

As a practical matter, it is easier to measure angular position than angular velocity. Therefore, we will go forward with sensor configuration 1 (observe θ_1 and θ_2) for the observer design.

Full-State Observer Design

Define the observer gain matrix using the control library:

```
C = get_C(params, sensor_config=1)
Vd = np.array([
    [1, 0],  # theta1 disturbance covariance
```

```
    [0, 1]  # theta2 disturbance covariance
])  # Disturbance covariance matrix
Vn = np.array([
    [1, 0],  # theta1 noise covariance
    [0, 1]  # theta2 noise covariance
])  # Noise covariance matrix
```