[EEC3600-001] 수치해석		
소속: 전기전자공학부	학번: 12191529	이름: 장준영
Term Project		Prob #2

1. Problem

a. 문제

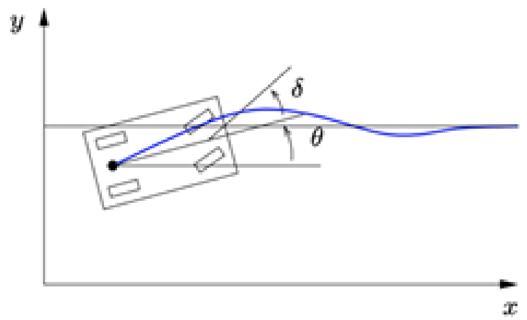
Problem 2. [*(Control)*]

Vehicle steering dynamics The vehicle dynamics are given by a simple bicycle model:

$$\begin{aligned} \dot{x} &= \cos\theta v \\ \dot{y} &= \sin\theta v \qquad \Leftrightarrow \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \\ \dot{\theta} &= \frac{v}{l}\tan\delta \end{aligned}$$

We take the state of the system as $\boldsymbol{x} = (x, y, \theta)$ where (x, y) is the position of the vehicle in the plane and θ is the angle of the vehicle with respect to horizontal. The vehicle input is given by $\boldsymbol{u} = (v, \delta)$ where v is the forward velocity of the vehicle and δ is the angle of the steering wheel. The model does not include saturation of the vehicle steering angle.

Figure: Vehicle steering dynamics.



- Linearize:

$$\dot{\delta \boldsymbol{x}} = \boldsymbol{A}\delta\boldsymbol{x} + \boldsymbol{B}\delta\boldsymbol{u} + \boldsymbol{c}$$

where the Jacobians are given as

$$\boldsymbol{A} = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}, \quad \boldsymbol{B} = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right|_{(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})}, \quad \boldsymbol{c} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$$

with the nominal state-input pair $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$.

- Time-discretization (Euler-forward)[1]:

$$\begin{aligned} \delta\boldsymbol{x}_{t+1} &= (\boldsymbol{I} + \delta t\boldsymbol{A})\delta\boldsymbol{x}_t + \delta t\boldsymbol{B}\delta\boldsymbol{u} + \delta t\boldsymbol{c} \\ &= \boldsymbol{A}_d\delta\boldsymbol{x}_t + \boldsymbol{B}_d\delta\boldsymbol{u}_t + \boldsymbol{c}_d \end{aligned}$$

with which

$$\boldsymbol{x}_t = \bar{\boldsymbol{x}} + \delta\boldsymbol{x}_t \quad \text{and} \quad \boldsymbol{u}_t = \bar{\boldsymbol{u}} + \delta\boldsymbol{u}_t\,.$$

[1] Well, I know this is not good, but let's focus on the main idea, not on the technical details of numerical differentiation for now.

(a) Write a Python code to represent

$$\boldsymbol{x}_T = \boldsymbol{F}\boldsymbol{u}_{0:T-1} + \boldsymbol{G}\boldsymbol{x}_0 + \boldsymbol{H}\boldsymbol{c}_d$$

where

$$\boldsymbol{u}_{0:T-1} = \begin{bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_{T-1} \end{bmatrix}$$

for $T > 0$ and

$$\bar{\boldsymbol{x}} = \begin{bmatrix} 0 \\ -2 \\ 0 \end{bmatrix}, \quad \bar{\boldsymbol{u}} = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

Consider a goal pose

$$\boldsymbol{x}_{\text{goal}} = \begin{bmatrix} 100 \\ 2 \\ 0 \end{bmatrix}$$

with the terminal input

$$\boldsymbol{u}_T = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

where $T = 10$. Consider the cost function

$$J(\boldsymbol{x}_0, \boldsymbol{u}_{0:T-1}) = \sum_{t=1}^{T} \|\boldsymbol{x}_t - \boldsymbol{x}_{\text{goal}}\|^2 + \lambda \sum_{t=0}^{T-1} \|\boldsymbol{u}_t\|^2$$

where $\lambda > 0$ refers to a weight factor.

(b) Write a linear least-squares problem for

$$\underset{\boldsymbol{u}_{0:T-1}}{\text{minimize}} \; J(\boldsymbol{x}_0, \boldsymbol{u}_{0:T-1}).$$

(c) Using Python, solve the least-squares problem you defined in Part (b) with different values of the weight factor $\lambda > 0$.

2. Solution (a)

이 코드는 단순 vehicle 모델 기반 차량의 선형화된 조향 시스템을 시뮬레이션하기 위해, Euler-forward 방식으로 선형 동역학을 이산화하고, 시간 구간 전체에 걸쳐 제어 시퀀스가 상태에 미치는 영향을 행렬 형태로 구성한 것이다. 최종적으로 아래 수식을 만족하는 행렬 F, G, H를 구성한다:

$$x_T = Fu_{0:T-1} + Gx_0 + Hc_d$$

■ 기본 파라미터 및 초기 상태 설정

```
# Parameters
T = 10             # Time horizon
dt = 1.0           # Discrete time step
l = 2.0            # Wheel base (vehicle constant)

# Nominal state and input
x_bar = np.array([[0], [-2], [0]])      # Initial state: [x, y, theta]
u_bar = np.array([[10], [0]])           # Nominal input: [v, delta]

# Goal state
x_goal = np.array([[100], [2], [0]])
```

- 본 실험에서는 차량의 선형화된 조향 모델을 시뮬레이션하기 위해 10 초의 시계열 구간을 설정하였다. 이산화 간격은 1 초로 설정하였으며, 차량의 휠베이스 길이는 2.0m 로 가정하였다. 시스템의 초기 상태는 $x_0 = [0, -2.0]^T$로, 초기 위치는 원점의 하단에 위치하며 차량은 수평방향($\theta = 0$)을 향하고 있다. 제어 입력의 기준값은 $u = [10, 0]^T$로, 차량은 10m/s 의 직진 속도를 유지하며 조향각은 0 이다. 목표 상태는 $x_{goal} = [100, 2.0]^T$로 설정하였으며, 이는 차량이 x 축을 따라 100m 전진하고 y 축으로는 2m 옮겨지도록 유도하는 것이다.

■ 선형화 (Linearization)

```
# Linearization around (x̄, ū)
theta = x_bar[2, 0]
v = u_bar[0, 0]
delta = u_bar[1, 0]

# Jacobians A, B and constant term c
A = np.array([
    [0, 0, -v * np.sin(theta)],
    [0, 0,  v * np.cos(theta)],
    [0, 0,  0]
])
```

- 행렬 A는 상태벡터 $x = [x, y, \theta]^T$에 대한 편미분 Jacobian 이다.
- 조향각이 0 이므로 $\sin(0) = 0$, $\cos(0) = 1$로 단순화된다.

```
B = np.array([
    [np.cos(theta), 0],
    [np.sin(theta), 0],
    [0, v / l]
])
```

- 행렬 B는 입력 벡터 $x = [x, y, \theta]^T$에 대한 편미분 Jacobian 이다.
- $\dot{\theta}$항이 $\frac{v}{l}\tan(\delta)$이므로, delta 에 대한 도함수가 포함된다.

```
c = np.array([
    [v * np.cos(theta)],
    [v * np.sin(theta)],
    [v / l * np.tan(delta)]
])
```

- $f(\bar{x}, \bar{u})$값 자체, 즉 선형화 기준점에서의 시스템 응답이다.
- 주어진 vehicle 모델은 비선형 시스템이므로, 시스템을 선형 형태로 다루기 위해 지정된 평형점 $(\bar{x}, \bar{u})$에서 선형화를 수행하였다. 이를 통해 상태 변수에 대한 Jacobian 행렬 $B = \frac{\partial f}{\partial u}$, 그리고 상수 항 $c = f(\bar{x}, \bar{u})$를 각각 정의하였다. 이러한 선형화는 차량의 주행 경로가 평형점 부근에서 크게 벗어나지 않는다는 가정 하에서 유효하며, 선형 시스템 제어 이론을 적용할 수 있도록 한다.

■ Euler-forward 이산화 (Discretization)

```
# Euler-forward time discretization
Ad = np.eye(3) + dt * A      # Discrete A matrix
Bd = dt * B                  # Discrete B matrix
cd = dt * c                  # Discrete constant term
```

- 오일러 방식에 따라 $A_d = I + \Delta t \cdot A$ 등으로 구성된다.
- 선형화된 연속 시간 시스템을 실제 구현 가능한 이산 시간 시스템으로 변환하기 위해 Euler-forward 방식으로 이산화를 수행하였다. 시간 간격 $\Delta t = 1$ 을 기준으로, 이산화된 시스템은 다음과 같은 행렬로 표현된다:

$$A_d = I + \Delta t \cdot A, \qquad B_d = \Delta t \cdot B, \qquad c_d = \Delta t \cdot c$$

이 과정을 통해 시간 이산 시스템 형태인 $x_{t+1} = A_d x_t + B_d u_t + c_d$ 를 구성할 수 있게 된다.

■ 행렬 F, G, H 구성

```
# Construct F, G, H matrices
F = np.zeros((3 * T, 2 * T))     # For stacking all u_0 to u_{T-1}
G = np.zeros((3 * T, 3))         # For initial state influence
H = np.zeros((3 * T, 1))         # For offset cd influence
```

- 시간 축을 따라 총 T개의 상태벡터(각 3 차원)를 수직으로 쌓은 형태. 총 $3T \times something$ 행렬들이 구성된다.
- 시간 구간 전체(T=10)에 대해, 전체 상태 벡터를 수직으로 쌓은 형태로 표현하면 다음과 같다:

$$x_T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = Fu_{0:T-1} + Gx_0 + Hc_d$$

이를 위해 다음과 같은 3 개의 블록 형태 행렬을 정의하였다:

- $F \in \mathbb{R}^{3T \times 2T}$: 각 제어 입력 $u_0, \ldots, u_{T-1}$이 전체 상태에 미치는 영향.
- $G \in \mathbb{R}^{3T \times 3}$: 초기 상태 x_0의 영향을 누적해서 표현.
- $H \in \mathbb{R}^{3T \times 1}$: 상수 항 c_d의 시간 누적 영향을 표현.

이렇게 정의된 구조는 이후 최소제곱 기반 최적화를 통해 u_i 들을 결정하는 데 필요한 기반을 제공한다.

■ 시간 루프를 통한 누적 행렬 계산

```python
Ad_power = np.eye(3)
for t in range(T):
    row = slice(3*t, 3*(t+1))

    # Fill F: sum of Ad^k * Bd * u_{t-k}
    for j in range(t+1):
        col = slice(2*j, 2*(j+1))
        Ad_k = np.linalg.matrix_power(Ad, t - j)
        F[row, col] += Ad_k @ Bd

    # Fill G: Ad^t * x0
    G[row, :] = np.linalg.matrix_power(Ad, t)

    # Fill H: sum of Ad^k * cd
    H[row, :] = sum(np.linalg.matrix_power(Ad, k) @ cd for k in range(t + 1))
```

- 이 루프는 시간 $t = 0 \sim (T-1)$에 대해 반복하면서:
  - 각 시간의 제어 입력 u_j가 얼마나 영향을 미치는지를 F에 누적.
  - 초기 상태 x_0의 영향을 G에 반영.
  - Offset 상수항 c_d는 누적합으로 H에 반영.

- 시간 구간 t=0부터 T-1까지 반복하면서, 각 time step에서 제어 입력이 전체 상태 벡터에 미치는 누적 효과를 F 행렬에 차곡차곡 더해가는 방식으로 구성하였다. 또한 각 시점에서 초기 상태 x_0가 선형 시스템을 통해 어떻게 전달되는지를 G 행렬로 구성하였고, 상수항 c_d는 행렬 거듭제곱과 누적합을 통해 H에 반영하였다.

■ 예시 실행 및 결과 확인

```python
# Example usage: compute x_T from u and x0
x0 = x_bar
u_seq = np.zeros((2 * T, 1))  # zero input as placeholder
xT = F @ u_seq + G @ x0 + H   # full stacked trajectory

# Print shapes to verify
print("F:", F.shape)
print("G:", G.shape)
print("H:", H.shape)
print("xT:", xT.shape)
```

- x_T는 전체 시간 동안의 상태들을 쌓은 벡터이며, 각 구간마다 $x_t \in \mathbb{R}^3$ 형태로 총 30×1 벡터이다.
- 이 궤적은 다음 (b)와 (c) 단계에서 최적화를 통해 목표 상태에 접근시키는 데 사용된다.
- 작성된 F, G, H 행렬을 바탕으로, 초기 상태 x_0와 제어 입력 $u_t = 0\ (place-holder)$를 이용하여 전체 상태 궤적 $x_T \in \mathbb{R}^{3T}$을 계산하였다.

이 시뮬레이션 결과는 단순한 초기 조건에서의 시스템 응답을 확인하는 용도로 사용되며, 이후 (b), (c) 항목에서 제어 입력을 최적화하여 목표 상태에 도달하도록 조정될 예정이다. 최종적으로 출력된 행렬의 크기를 통해 구성의 정확성을 검증할 수 있다.

3. **Solution (b)**

1) 문제 분석

차량 조향 모델에서, 초기 상태 $x_0 = \bar{x} = [0, -2, 0]^T$ 에서 시작하여 목표 상태 $x_{goal} = [100, 2, 0]^T$ 에 도달하도록 하는 제어 입력 시퀀스 $(u_0, u_1, \dots, u_{T-1})$ 를 설계하고자 한다. 이를 위해, 아래 목적 함수를 최소화하는 최적 제어 문제를 최소제곱 문제 형태로 정식화한다:

$$J(x_0, u_{0:T-1}) = \sum_{t=1}^{T} \left\| x_t - x_{goal} \right\|^2 + \lambda \sum_{t=0}^{T-1} \|u_t\|^2$$

여기서 λ는 제어 입력 크기를 얼마나 패널티 줄 것인지 결정하는 가중치이다.

2) 상태 전개 수식 기반 행렬 표현

앞서 (a)에서 다음과 같은 선형 시스템 전개식을 유도하였다:

$$x_T = Fu + Gx_0 + Hc_d$$

여기서:

- $x_T \in \mathbb{R}^{3T}$: 모든 시점의 상태를 수직으로 쌓은 벡터.
- $u \in \mathbb{R}^{2T}$: 모든 시점의 제어 입력을 수직으로 쌓은 벡터.
- $F \in \mathbb{R}^{3T \times 2T}$, $G \in \mathbb{R}^{3T \times 3}$, $H \in \mathbb{R}^{3T \times 1}$

3) 상태 Least-Square Problem 정식화

목적 함수는 전체 시간 구간 동안의 상태 오차와 제어 입력 크기의 가중합으로 구성된다. 이를 행렬 형태로 다시 정리하면 다음과 같다:

$$J(u) = \left\| Fu - \left(x_{goal}^{stacked} - Gx_0 - Hc_d \right) \right\|^2 + \lambda$$

여기서:

- $x_{goal}^{stacked} \in \mathbb{R}^{3T}$: 목표 상태를 T 회 반복하여 쌓은 벡터.

즉, 최종적으로 우리는 다음과 같은 표준 선형 최소제곱 문제를 얻게 된다:

$$\min_{u \in \mathbb{R}^{2T}} \left\| \begin{bmatrix} F \\ \sqrt{\lambda} I \end{bmatrix} u - \begin{bmatrix} x_{goal}^{stacked} - G x_0 - H c_d \\ 0 \end{bmatrix} \right\|^2$$

이는 $\min_{u} \|Au - b\|^2$ 형태의 표준적인 선형 최소제곱 문제이며, 해는 다음과 같이 계산할 수 있다:

$$u^* = (A^T A)^{-1} A^T b$$

4. Solution (c)

아래는 문제 (c)에서 요구한 내용을 만족하는 Python 전체 코드이다. 이 코드는 다음의 동작을 수행한다:

1) (a)에서 구성한 선형 시스템 행렬 F, G, H를 사용한다.
2) (b)에서 수식화한 목적 함수

$$J(x_0, u_{0:T-1}) = \sum_{t=1}^{T} \|x_t - x_{goal}\|^2 + \lambda \sum_{t=0}^{T-1} \|u_t\|^2$$

를 최소화하는 least-square 문제를 풀고

3) λ 값들을 바꿔가며 해를 시각화한다.

■ 파라미터 및 초기 상태 설정

```
# Parameters
T = 10
dt = 1.0
l = 2.0

# Nominal state and input
x_bar = np.array([[0], [-2], [0]])      # Initial state
u_bar = np.array([[10], [0]])           # Nominal input
x_goal = np.array([[100], [2], [0]])    # Target state
x_goal_stack = np.tile(x_goal, (T, 1))  # Stack target state T ti
```

- T: 제어 시퀀스의 길이 (10 초 동안 제어).
- dt: 시간 간격 (dt).
- l: 차량의 휠베이스 길이 (2m).
- x_bar: 선형화 기준점인 초기 상태.
- u_bar: 기준 제어 입력.
- x_goal: 차량이 도달해야 할 목표 상태.

- x_goal_stack: 각 시간 스텝마다 동일한 목표 상태를 가진다고 가정하여 30x1 형태로 반복.

■ 선형화 (Linearization)

```
# Linearization
theta = x_bar[2, 0]
v = u_bar[0, 0]
delta = u_bar[1, 0]
```

- 초기 상태 및 입력에서 파생된 파라미터들을 추출.

```
# Jacobians
A = np.array([
    [0, 0, -v * np.sin(theta)],
    [0, 0,  v * np.cos(theta)],
    [0, 0,  0]
])

B = np.array([
    [np.cos(theta), 0],
    [np.sin(theta), 0],
    [0, v / l]
])

c = np.array([
    [v * np.cos(theta)],
    [v * np.sin(theta)],
    [v / l * np.tan(delta)]
])
```

- 선형 시스템의 상태 행렬 A, 입력 행렬 B, 상수항 c를 구성.
- Vehicle Model 기반 동역학 행렬을 미분하여 얻은 Jacobian.

■ Euler-forward 이산화

```
# Discretize
Ad = np.eye(3) + dt * A
Bd = dt * B
cd = dt * c
```

- Euler Discretization 을 적용하여 연속 시스템을 이산 시스템으로 변환:

$$x_{t+1} = A_d x_t + B_d u_t + c_d$$

■ 상태 전개 행렬 F, G, H 생성

```python
# Construct F, G, H
F = np.zeros((3 * T, 2 * T))
G = np.zeros((3 * T, 3))
H = np.zeros((3 * T, 1))
```

- 시간 축을 따라 상태들을 쌓은 벡터 x_T를 만들기 위한 전개 행렬:

$$x_T = Fu + Gx_0 + Hc_d$$

```python
for t in range(T):
    row = slice(3*t, 3*(t+1))
    for j in range(t+1):
        col = slice(2*j, 2*(j+1))
        Ad_k = np.linalg.matrix_power(Ad, t - j)
        F[row, col] += Ad_k @ Bd
    G[row, :] = np.linalg.matrix_power(Ad, t)
    H[row, :] = sum(np.linalg.matrix_power(Ad, k) @ cd for k in range(t + 1))
```

- 각 시간 스텝마다:
  - F: 입력 벡터 u가 전체 상태에 미치는 누적 영향.
  - G: 초기 상태 x_0가 누적 영향을 주는 방식.
  - H: Offset c_d의 누적 반영.

■ Least-Square 문제 풀이 (λ 변화에 따라)

```python
# Construct F, G, H
F = np.zeros((3 * T, 2 * T))
G = np.zeros((3 * T, 3))
H = np.zeros((3 * T, 1))
```

- 다양한 λ 값을 실험:
  - 작은 λ: 빠른 수렴 속도
  - 큰 λ: 작은 제어 입력

```python
for lam in lambdas:
    A_ls = np.vstack([F, np.sqrt(lam) * np.eye(2 * T)])  # augmented matrix
    b_ls = np.vstack([x_goal_stack - G @ x_bar - H, np.zeros((2 * T, 1))])  # augmented target
    u_star = np.linalg.lstsq(A_ls, b_ls, rcond=None)[0]
    x_stack = F @ u_star + G @ x_bar + H
    trajectories.append((lam, x_stack, u_star))
```

- 표준 선형 최소제곱 문제 $\min_u \|Au - b\|^2$ 형태로 푼다.
- 각 λ 값에 따른 최적 입력 u^*와 그에 따른 상태 궤적 x를 저장한다.

■ 시각화 (결과 그래프 출력)

```
# Plot results
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
```

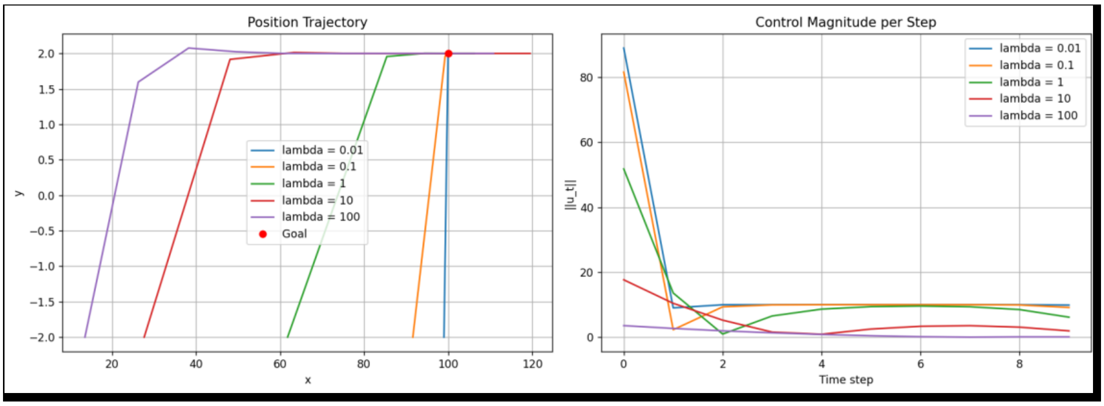
- 2 개의 subplot: 위치 궤적 & 제어 입력 크기

```
# Position trajectory plot
for lam, x_stack, _ in trajectories:
    x = x_stack[0::3].flatten()
    y = x_stack[1::3].flatten()
    axes[0].plot(x, y, label=f"lambda = {lam}")
axes[0].plot(x_goal[0], x_goal[1], 'ro', label='Goal')
axes[0].set_title("Position Trajectory")
axes[0].set_xlabel("x")
axes[0].set_ylabel("y")
axes[0].legend()
axes[0].grid(True)
```

- $x - y$ 평면에서의 이동 궤적 시각화.
- 목표점 표시

```
# Control magnitude plot
for lam, _, u_star in trajectories:
    u_reshaped = u_star.reshape(T, 2)
    u_norm = np.linalg.norm(u_reshaped, axis=1)
    axes[1].plot(range(T), u_norm, label=f"lambda = {lam}")
axes[1].set_title("Control Magnitude per Step")
axes[1].set_xlabel("Time step")
axes[1].set_ylabel("||u_t||")
axes[1].legend()
axes[1].grid(True)
```

- 시간에 따른 제어 입력의 크기 $\|u_t\|$ 시각화.
- λ 값이 커질수록 전체 입력이 줄어듦을 확인할 수 있다.



[결과 분석]

1) 위치 궤적 (Position Trajectory)

■ λ가 작을수록 (예: 0.01):

- 장점: 목표 위치에 더 가까이, 더 빠르게 도달한다.
- 단점: 그 과정에서 매우 큰 제어 입력을 사용한다.

■ λ가 클수록 (예: 100):

- 단점: 제어 입력을 작게 유지하려다 보니 도달 시간이 길어진다.

■ $\lambda = 1 \sim 10$ 정도에서는:

- 장점: 제어 입력과 위치 정확도 간에 좋은 균형을 보인다.

2) 제어 입력 크기 (Control Magnitude per Step)

■ λ가 작을수록:

- 초반에 매우 큰 $\|u_t\|$를 사용해 목표에 강하게 접근한다.

■ λ가 클수록:

- 전체적으로 제어 입력의 크기가 작고, 변화 폭도 적다.
- 이는 목적함수의 두 항,

$$\sum \left\|x_t - x_{goal}\right\|^2 \quad vs. \quad \lambda \sum \|u_t\|^2$$

사이의 trade-off 가 잘 반영된 결과다.

본 실험에서는 목적함수에 포함된 제어 입력 항의 가중치 λ 값을 변화시키며 시스템의 응답 특성을 분석하였다. λ가 작을수록 시스템은 더 정확히 목표 상태에 도달하지만, 그 대가로 매우 큰 제어 입력이 요구되며 이는 실제 시스템에서

과도한 하드웨어 부담이나 안전 문제로 이어질 수 있다. 반면, λ가 클수록 제어 입력은 작아지지만 목표 상태에서의 오차가 커진다. 이 결과는 정확도와 제어 노력 간의 명확한 trade-off 를 잘 보여주며, λ 값의 선택이 제어 시스템 설계에서 중요한 조율 변수임을 나타낸다.