1)
자율주행 로봇이 직선 경로를 따라 목표 위치로 이동하려고 합니다. 현재 위치는 `0.0`이고, 목표 위치는 `10.0`입니다. 단순한 P 제어기를 사용하여 로봇을 제어하려고 합니다. 비례 이득 `Kp = 0.5`일 때, 로봇이 목표 위치에 도달할 때까지의 위치 변화를 시뮬레이션하세요. 위치 업데이트는 다음 식을 따릅니다.

control = Kp * (target_position - current_position)
current_position += control

```python
# 파라미터 설정
Kp = 0.5
target_position = 10.0
current_position = 0.0

positions = [current_position]

# 시뮬레이션
for _ in range(20):
    control = Kp * (target_position - current_position)
    current_position += control
    positions.append(current_position)

# 결과 출력
for i, pos in enumerate(positions):
    print(f"Step {i}: Position = {pos}")
```

```
Step 0: Position = 0.0
Step 1: Position = 5.0
Step 2: Position = 7.5
Step 3: Position = 8.75
Step 4: Position = 9.375
Step 5: Position = 9.6875
```

- 비례 제어는 현재 위치와 목표 위치 간의 오차에 비례하여 제어 입력을 생성합니다.
- 이 코드는 단순히 제어 입력을 계산하고 현재 위치를 업데이트하는 과정을 반복합니다.
- 반복 횟수를 20으로 설정하여 로봇이 목표 위치에 근접하는 과정을 관찰합니다.

2)
자율주행 로봇이 속도를 제어하기 위해 PD 제어기를 사용합니다. 목표 속도는 `5.0` m/s이고, 현재 속도는 `0.0` m/s입니다. 비례 이득 `Kp = 0.6`, 미분 이득 `Kd = 0.1`입니다. 로봇의 속도가

목표 속도에 도달할 때까지의 속도 변화를 시뮬레이션하세요. 속도 업데이트는 다음 식을 따릅니다.

control = Kp * (target_speed - current_speed) + Kd * (previous_error - current_error)
current_speed += control

```
Step 0: Speed = 0.0
Step 1: Speed = 3.0
Step 2: Speed = 3.9
Step 3: Speed = 4.47
Step 4: Speed = 4.731
Step 5: Speed = 4.8663
```

PD 제어기는 오차와 오차의 변화량(미분)에 기반하여 제어 입력을 생성합니다.
`previous_error`를 사용하여 오차의 변화를 계산합니다.
반복적으로 제어 입력을 계산하고 속도를 업데이트하여 목표 속도에 도달하는 과정을 시뮬레이션합니다.

3)
자율주행 로봇이 궤적을 따라 이동하기 위해 PID 제어기를 사용합니다. 로봇의 현재 위치는 `0.0`, 목표 위치는 `10.0`입니다. 비례 이득 `Kp = 1.0`, 적분 이득 `Ki = 0.2`, 미분 이득 `Kd = 0.05`입니다. 시뮬레이션을 통해 로봇이 목표 위치에 도달할 때까지의 위치 변화를 출력하세요. 위치 업데이트는 다음 식을 따릅니다.

error = target_position - current_position
integral += error * dt
derivative = (error - previous_error) / dt
control = Kp * error + Ki * integral + Kd * derivative
current_position += control * dt
previous_error = error

```
# 파라미터 설정
Kp = 1.0
Ki = 0.2
Kd = 0.05
dt = 1.0

target_position = 10.0
current_position = 0.0
previous_error = target_position - current_position
```

```
integral = 0.0

positions = [current_position]

# 시뮬레이션
for _ in range(20):
    error = target_position - current_position
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_position += control * dt
    positions.append(current_position)
    previous_error = error

# 결과 출력
for i, pos in enumerate(positions):
    print(f"Step {i}: Position = {pos}")
```

PID 제어는 비례, 적분, 미분 요소를 모두 포함하여 제어 입력을 계산합니다.
`integral`은 오차의 누적값이며, 시스템의 지속적인 오차를 수정하는 데 도움을 줍니다.
`derivative`는 오차의 변화율로, 시스템의 응답 속도를 조절합니다.
`dt`는 시간 간격으로, 단순화를 위해 `1.0`으로 설정했습니다.

4)
자율주행 차량이 곡선 도로를 주행하고 있습니다. 차량의 조향 각도를 제어하기 위해 PID 제어기를 사용합니다. 현재 조향 각도는 `0.0`도이며, 목표 조향 각도는 `15.0`도입니다. 비례 이득 `Kp = 0.8`, 적분 이득 `Ki = 0.1`, 미분 이득 `Kd = 0.05`입니다. 시뮬레이션을 통해 차량이 목표 조향 각도에 도달하는 과정을 출력하세요.

```
# 파라미터 설정
Kp = 0.8
Ki = 0.1
Kd = 0.05
dt = 1.0

target_angle = 15.0
current_angle = 0.0
previous_error = target_angle - current_angle
integral = 0.0
```

```
angles = [current_angle]

# 시뮬레이션
for _ in range(20):
    error = target_angle - current_angle
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_angle += control * dt
    angles.append(current_angle)
    previous_error = error

# 결과 출력
for i, angle in enumerate(angles):
    print(f"Step {i}: Steering Angle = {angle}")
```

조향 각도 제어에도 PID 제어기를 적용할 수 있습니다.
조향 각도의 오차를 계산하고, PID 제어기를 통해 제어 입력을 생성합니다.
시뮬레이션을 통해 차량이 목표 조향 각도에 도달하고 안정화되는 과정을 확인할 수 있습니다.

5)
PID 제어기를 사용하는 자율주행 로봇에서 적분 포화 현상을 방지하기 위해 적분 항에 제한을 걸어야 합니다. 위의 문제 3의 코드를 수정하여 적분 항 `integral`이 `-10`과 `10` 사이로 제한되도록 하세요.

```
# 파라미터 설정
Kp = 1.0
Ki = 0.2
Kd = 0.05
dt = 1.0

target_position = 10.0
current_position = 0.0
previous_error = target_position - current_position
integral = 0.0
```

```python
integral_min = -10.0
integral_max = 10.0

positions = [current_position]

# 시뮬레이션
for _ in range(20):
    error = target_position - current_position
    integral += error * dt
    # 적분 항 제한
    if integral > integral_max:
        integral = integral_max
    elif integral < integral_min:
        integral = integral_min
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_position += control * dt
    positions.append(current_position)
    previous_error = error

# 결과 출력
for i, pos in enumerate(positions):
    print(f"Step {i}: Position = {pos}")
```

```
Step 0: Position = 0.0
Step 1: Position = 12.0
Step 2: Position = 11.0
Step 3: Position = 11.45
Step 4: Position = 11.0875
Step 5: Position = 10.910625
```

적분 포화 현상은 적분 항이 너무 커져 시스템의 안정성을 해치는 현상입니다.
이를 방지하기 위해 적분 항 `integral`의 값을 일정 범위로 제한합니다.
코드를 수정하여 적분 항이 `-10`과 `10` 사이로 유지되도록 합니다.

6)

자율주행 로봇의 속도 제어에서 미분 신호의 노이즈 영향을 줄이기 위해 저역 통과 필터를 적용하려고 합니다. 위의 문제 2의 코드에서 미분 항 계산 시 저역 통과 필터를 적용하여 노이즈를 줄이세요. 필터의 시간 상수는 `Tf = 0.1`입니다.

```
# 파라미터 설정
Kp = 0.6
Kd = 0.1
Tf = 0.1
dt = 1.0
target_speed = 5.0
current_speed = 0.0
previous_error = target_speed - current_speed
derivative = 0.0

speeds = [current_speed]

# 시뮬레이션
for _ in range(20):
    current_error = target_speed - current_speed
    # 미분 항 계산 시 저역 통과 필터 적용
    derivative = ( (2 * Tf - dt) * derivative + 2 * (current_error - previous_error) ) / (2 * Tf + dt)
    control = Kp * current_error + Kd * derivative
    current_speed += control
    speeds.append(current_speed)
    previous_error = current_error

# 결과 출력
for i, speed in enumerate(speeds):
    print(f"Step {i}: Speed = {speed}")
```

```
Step 0: Speed = 0.0
Step 1: Speed = 3.0
Step 2: Speed = 3.7
Step 3: Speed = 4.696666666666667
Step 4: Speed = 4.5681111111111115
Step 5: Speed = 5.055707407407407
Step 6: Speed = 4.788708271604938
```

미분 항은 노이즈에 민감하므로, 저역 통과 필터를 적용하여 노이즈 영향을 줄입니다.
저역 통과 필터의 이산화된 형태를 사용하여 미분 신호를 필터링합니다.
필터의 시간 상수 `Tf`를 조절하여 필터의 특성을 변경할 수 있습니다.

7)
자율주행 차량의 속도 제어에서 PID 제어기를 사용하고 있습니다. 차량의 질량이 변경되어 시스템의 동특성이 변했습니다. 이를 자동으로 보상하기 위해 적응형 PID 제어기를 구현하려고 합니다. 간단히 비례 이득 Kp를 오차의 크기에 따라 조절하도록 코드를 수정하세요.

```
# 초기 파라미터 설정
Kp_base = 0.6
Ki = 0.1
Kd = 0.05
dt = 1.0

target_speed = 10.0
current_speed = 0.0
previous_error = target_speed - current_speed
integral = 0.0

speeds = [current_speed]

# 시뮬레이션
for _ in range(20):
    error = target_speed - current_speed
    # 비례 이득을 오차에 따라 조절 (예: 오차가 클수록 Kp 증가)
    Kp = Kp_base * (1 + abs(error)/10)
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_speed += control * dt
    speeds.append(current_speed)
    previous_error = error

# 결과 출력
for i, speed in enumerate(speeds):
    print(f"Step {i}: Speed = {speed}")
```

```
Step 0: Speed = 0.0
Step 1: Speed = 13.0
Step 2: Speed = 10.71
Step 3: Speed = 10.997254
Step 4: Speed = 10.85414256756904
Step 5: Speed = 10.748899370348255
Step 6: Speed = 10.64014129819413
```

적응형 PID 제어기는 시스템의 변화에 따라 제어기 이득을 조절합니다.
여기서는 오차의 크기에 따라 비례 이득 `Kp`를 조절하여 큰 오차에서는 빠르게 반응하고, 작은 오차에서는 안정적으로 동작하도록 합니다.
간단한 형태의 적응형 제어를 구현하였습니다.

8)
자율주행 로봇의 위치 제어에서 샘플링 주기가 불규칙하게 변한다고 가정합니다. PID 제어기를 구현할 때, 가변적인 `dt`를 고려하여 코드를 수정하세요. `dt`는 임의로 `0.8`에서 `1.2` 사이의 값을 갖습니다.

```python
import random

# 파라미터 설정
Kp = 1.0
Ki = 0.2
Kd = 0.05

target_position = 10.0
current_position = 0.0
previous_error = target_position - current_position
integral = 0.0

positions = [current_position]

# 시뮬레이션
for _ in range(20):
    dt = random.uniform(0.8, 1.2)  # dt를 0.8 ~ 1.2 사이에서 임의로 선택
    error = target_position - current_position
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_position += control * dt
    positions.append(current_position)
    previous_error = error
```

```
# 결과 출력
for i, pos in enumerate(positions):
    print(f"Step {i}: Position = {pos}")
```

```
Step 0: Position = 0.0
Step 1: Position = 11.655654804194034
Step 2: Position = 11.063007217350144
Step 3: Position = 11.474432624123656
Step 4: Position = 10.968050205436398
Step 5: Position = 10.889775396185117
Step 6: Position = 10.697797204432758
```

`dt`가 가변적일 때, 적분 항과 미분 항 계산 시 `dt`를 고려해야 합니다.
`random.uniform` 함수를 사용하여 `dt`를 임의로 설정합니다.
이를 통해 샘플링 주기가 불규칙한 시스템에서도 PID 제어기가 올바르게 동작하도록 합니다.

9)
자율주행 로봇의 속도 제어에서 PID 제어기의 튜닝을 자동화하기 위해 지글러-니콜스 방법을 사용하려고 합니다. 아래의 과정을 따라 임계 진동 주기 `Tu = 5`와 임계 이득 `Ku = 2.0`이 측정되었다고 가정합니다. 이에 따라 PID 제어기의 이득을 계산하고, 이를 코드에 적용하여 시뮬레이션하세요.

```
# 지글러-니콜스 방법에 따른 이득 계산
Ku = 2.0  # 임계 이득
Tu = 5.0  # 임계 진동 주기

Kp = 0.6 * Ku
Ki = 1.2 * Ku / Tu
Kd = 3 * Ku * Tu / 40

print(f"Calculated Gains - Kp: {Kp}, Ki: {Ki}, Kd: {Kd}")

# 시뮬레이션 파라미터 설정
dt = 1.0
target_speed = 10.0
current_speed = 0.0
previous_error = target_speed - current_speed
integral = 0.0
```

```
speeds = [current_speed]

# 시뮬레이션
for _ in range(20):
    error = target_speed - current_speed
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_speed += control * dt
    speeds.append(current_speed)
    previous_error = error

# 결과 출력
for i, speed in enumerate(speeds):
    print(f"Step {i}: Speed = {speed}")
```

```
Step 0: Speed = 0.0
Step 1: Speed = 16.8
Step 2: Speed = -2.424000000000003
Step 3: Speed = 34.40232
Step 4: Speed = -26.713797600000007
Step 5: Speed = 76.58887696800001
Step 6: Speed = -99.34841301624
```

지글러-니콜스 방법은 PID 제어기의 초기 이득을 결정하는 경험적인 방법입니다.
임계 진동 주기 `Tu`와 임계 이득 `Ku`를 기반으로 `Kp`, `Ki`, `Kd`를 계산합니다.
계산된 이득을 코드에 적용하여 시뮬레이션을 수행합니다.

10)
자율주행 차량의 조향 제어에서 PID 제어기를 사용하고 있습니다. 하지만 차량이 곡선 주행 시 오버슈트가 발생합니다. 미분 이득 `Kd`를 조절하여 오버슈트를 줄이는 방향으로 코드를 수정하세요. `Kd`를 기존 값의 두 배로 설정하고, 시뮬레이션 결과를 비교하세요.

```
# 기존 파라미터 설정
Kp = 0.8
Ki = 0.1
Kd = 0.05  # 기존 미분 이득
dt = 1.0

target_angle = 15.0
current_angle = 0.0
```

```
previous_error = target_angle - current_angle
integral = 0.0

angles_original = [current_angle]

# 기존 미분 이득으로 시뮬레이션
for _ in range(20):
    error = target_angle - current_angle
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd * derivative
    current_angle += control * dt
    angles_original.append(current_angle)
    previous_error = error

# 미분 이득을 두 배로 설정
Kd_new = Kd * 2
current_angle = 0.0
previous_error = target_angle - current_angle
integral = 0.0

angles_new = [current_angle]

# 새로운 미분 이득으로 시뮬레이션
for _ in range(20):
    error = target_angle - current_angle
    integral += error * dt
    derivative = (error - previous_error) / dt
    control = Kp * error + Ki * integral + Kd_new * derivative
    current_angle += control * dt
    angles_new.append(current_angle)
    previous_error = error

# 결과 비교 출력
print("Original Kd Results:")
for i, angle in enumerate(angles_original):
    print(f"Step {i}: Steering Angle = {angle}")

print("\nNew Kd Results (Kd doubled):")
for i, angle in enumerate(angles_new):
```

```
    print(f"Step {i}: Steering Angle = {angle}")
```

```
New Kd Results (Kd doubled):
Step 0: Steering Angle = 0.0
Step 1: Steering Angle = 13.5
Step 2: Steering Angle = 15.0
Step 3: Steering Angle = 16.5
Step 4: Steering Angle = 16.65
```

미분 이득 Kd는 시스템의 응답 속도와 안정성에 영향을 줍니다.
Kd를 증가시키면 오버슈트를 줄이고 시스템의 감쇠를 증가시킬 수 있습니다.
코드를 수정하여 Kd를 두 배로 설정하고, 시뮬레이션 결과를 비교하여 오버슈트의 변화를 관찰합니다.

11)
자율주행 차량이 일정한 속도로 직선 도로를 주행하고 있습니다. 그러나 도로의 경사가 변화하여 차량의 속도에 영향을 미칩니다. 차량은 속도를 일정하게 유지하기 위해 PID 제어기를 사용합니다.

목표:

- 차량의 속도를 PID 제어기를 사용하여 목표 속도에 유지합니다.
- 도로의 경사로 인한 외란을 극복합니다.
- 시뮬레이션을 통해 시간에 따른 속도 변화를 그래프로 나타냅니다.

조건:

- 목표 속도는 vtarget=20 m/sv_{\text{target}} = 20 \, \text{m/s}vtarget=20m/s입니다.
- 도로의 경사는 시간에 따라 θ(t)=5∘sin(0.1t)\theta(t) = 5^\circ \sin(0.1 t)θ(t)=5∘sin(0.1t)로 변합니다.
- 차량의 질량은 m=1500 kgm = 1500 \, \text{kg}m=1500kg입니다.
- 공기 저항과 마찰은 무시합니다.
- PID 이득은 Kp=800K_p = 800Kp=800, Ki=40K_i = 40Ki=40, Kd=100K_d = 100Kd=100로 설정합니다.
- 시뮬레이션 시간은 t=0~60 st = 0 \sim 60 \, \text{s}t=0~60s이며, 시간 간격은 Δt=0.1 s\Delta t = 0.1 \, \text{s}Δt=0.1s입니다.

문제 이해 및 접근 방법:

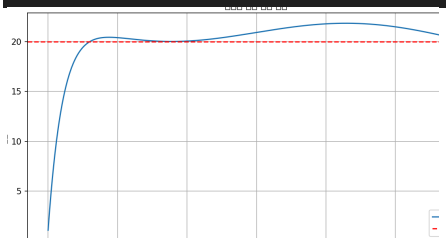
- 도로의 경사가 변하므로 차량에 작용하는 중력 성분이 달라져 차량의 속도가 변합니다.
- PID 제어기를 사용하여 엔진의 힘을 조절하여 속도를 일정하게 유지합니다.

- 차량의 운동 방정식을 사용하여 속도를 업데이트합니다.

```
import numpy as np
import matplotlib.pyplot as plt

# 시뮬레이션 파라미터
dt = 0.1  # 시간 간격
t_end = 60  # 시뮬레이션 종료 시간
t = np.arange(0, t_end, dt)

# 차량 파라미터
m = 1500  # 차량 질량 (kg)
v_target = 20.0  # 목표 속도 (m/s)

# PID 제어기 이득
Kp = 800
Ki = 40
Kd = 100

# 초기화
v = 0.0  # 초기 속도
F_engine = 0.0  # 초기 엔진 힘
integral = 0.0
prev_error = v_target - v

# 로그 저장용 리스트
v_history = []
F_history = []
theta_history = []

for ti in t:
    # 도로 경사 계산 (rad 단위)
    theta = np.deg2rad(5 * np.sin(0.1 * ti))
    theta_history.append(np.rad2deg(theta))

    # 중력에 의한 힘 계산
    F_gravity = m * 9.81 * np.sin(theta)

    # 속도 오차 계산
    error = v_target - v
```

```
    integral += error * dt
    derivative = (error - prev_error) / dt
    prev_error = error

    # PID 제어기를 사용하여 엔진 힘 계산
    F_engine = Kp * error + Ki * integral + Kd * derivative

    # 차량의 가속도 계산
    a = (F_engine - F_gravity) / m

    # 속도 업데이트
    v += a * dt

    # 로그 저장
    v_history.append(v)
    F_history.append(F_engine)

# 결과 시각화
# 1. 시간에 따른 속도 변화
plt.figure(figsize=(10, 5))
plt.plot(t, v_history, label='실제 속도')
plt.axhline(v_target, color='r', linestyle='--', label='목표 속도')
plt.xlabel('시간 (s)')
plt.ylabel('속도 (m/s)')
plt.title('시간에 따른 차량 속도')
plt.legend()
plt.grid(True)
plt.show()

# 2. 시간에 따른 엔진 힘 변화
plt.figure(figsize=(10, 5))
plt.plot(t, F_history, label='엔진 힘')
plt.xlabel('시간 (s)')
plt.ylabel('엔진 힘 (N)')
plt.title('시간에 따른 엔진 힘')
plt.legend()
plt.grid(True)
plt.show()

# 3. 시간에 따른 도로 경사 변화
```

```
plt.figure(figsize=(10, 5))
plt.plot(t, theta_history, label='도로 경사 각도')
plt.xlabel('시간 (s)')
plt.ylabel('경사 각도 (deg)')
plt.title('시간에 따른 도로 경사 변화')
plt.legend()
plt.grid(True)
plt.show()
```



코드 설명:

- 도로 경사 계산:
  - θ(t)=5∘sin(0.1t)\theta(t) = 5^\circ \sin(0.1 t)θ(t)=5∘sin(0.1t)를 사용하여 시간에 따른 경사 각도를 계산합니다.
  - 경사 각도는 라디안으로 변환하여 사용합니다.
- 중력에 의한 힘 계산:
  - 경사로에서 차량에 작용하는 중력 성분은 Fgravity=mgsin(θ)F_{\text{gravity}} = m g \sin(\theta)Fgravity=mgsin(θ)입니다.
- **PID** 제어기 구현:
  - 속도 오차 `error`를 계산하고, 적분 및 미분 항을 구합니다.
  - 엔진 힘 `F_engine`을 PID 제어기를 사용하여 계산합니다.
- 차량의 운동 방정식:
  - 가속도 a=Fengine−Fgravityma = \frac{F_{\text{engine}} - F_{\text{gravity}}}{m}a=mFengine−Fgravity를 계산합니다.
  - 속도를 업데이트합니다.
- 결과 시각화:
  - 시간에 따른 속도 변화, 엔진 힘, 도로 경사 각도를 그래프로 표시합니다.

4. 결과 분석:

- 속도 그래프:
  - 차량의 속도가 목표 속도인 20 m/s에 근접하여 유지되는 것을 확인할 수 있습니다.
  - 도로 경사의 변화로 인해 약간의 속도 변동이 있지만, PID 제어기가 이를 보상합니다.
- 엔진 힘 그래프:

    - 도로 경사의 변화에 따라 엔진 힘이 조절되는 것을 볼 수 있습니다.
    - 경사가 올라갈 때는 엔진 힘이 증가하고, 내려갈 때는 감소합니다.
- 도로 경사 그래프:
    - 경사 각도가 시간에 따라 주기적으로 변하는 것을 확인할 수 있습니다.

12)

자율주행 차량이 바람으로 인한 측면 외란이 존재하는 도로에서 차선을 유지하며 주행해야 합니다. 차량은 PID 제어기를 사용하여 조향 각도를 조절합니다.

목표:

- 차량이 차선 중앙을 따라 주행하도록 횡방향 위치를 제어합니다.
- 바람으로 인한 측면 외란을 극복합니다.
- 시뮬레이션 결과를 통해 제어 성능을 평가합니다.

조건:

- 차량의 종방향 속도는 v=20 m/sv = 20 \, \text{m/s}v=20m/s로 일정합니다.
- 측면 외란은 Fwind=500sin(0.5t) NF_{\text{wind}} = 500 \sin(0.5 t) \, \text{N}Fwind=500sin(0.5t)N로 모델링됩니다.
- 차량의 질량은 m=1200 kgm = 1200 \, \text{kg}m=1200kg입니다.
- 차량의 회전 관성은 무시합니다.
- PID 이득은 Kp=0.2K_p = 0.2Kp=0.2, Ki=0.05K_i = 0.05Ki=0.05, Kd=0.1K_d = 0.1Kd=0.1로 설정합니다.
- 시뮬레이션 시간은 t=0~30 st = 0 \sim 30 \, \text{s}t=0~30s이며, 시간 간격은 Δt=0.1 s\Delta t = 0.1 \, \text{s}Δt=0.1s입니다.

문제 이해 및 접근 방법:

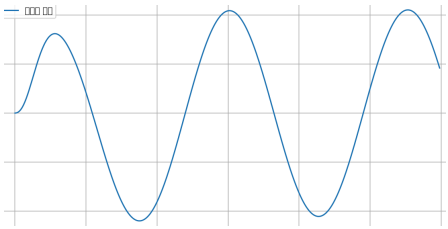
- 측면 외란으로 인해 차량이 차선에서 벗어날 수 있으므로, 이를 PID 제어기로 보상합니다.
- 횡방향 위치를 제어하여 차선 중앙을 유지합니다.
- 차량의 횡방향 운동을 모델링합니다.

```
import numpy as np
import matplotlib.pyplot as plt

# 시뮬레이션 파라미터
```

```
dt = 0.1
t_end = 30
t = np.arange(0, t_end, dt)

# 차량 파라미터
m = 1200  # 질량 (kg)
v = 20.0  # 종방향 속도 (m/s)

# PID 제어기 이득
Kp = 0.2
Ki = 0.05
Kd = 0.1

# 초기화
y = 0.0  # 횡방향 위치 (차선 중앙이 0)
vy = 0.0  # 횡방향 속도
integral = 0.0
prev_error = 0.0

# 로그 저장용 리스트
y_history = []
delta_history = []
wind_history = []

for ti in t:
    # 측면 외란 계산
    F_wind = 500 * np.sin(0.5 * ti)
    wind_history.append(F_wind)

    # 횡방향 위치 오차 (차선 중앙에서의 거리)
    error = -y  # 목표 위치가 y=0이므로

    # PID 제어기 계산
    integral += error * dt
    derivative = (error - prev_error) / dt
    prev_error = error

    # 조향 각도 계산 (단순히 PID 출력으로 모델링)
    delta = Kp * error + Ki * integral + Kd * derivative
```

```
    # 횡방향 가속도 계산
    ay = (F_wind + m * v * delta) / m

    # 횡방향 속도 및 위치 업데이트
    vy += ay * dt
    y += vy * dt

    # 로그 저장
    y_history.append(y)
    delta_history.append(delta)

# 결과 시각화
# 1. 횡방향 위치 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, y_history, label='횡방향 위치')
plt.xlabel('시간 (s)')
plt.ylabel('횡방향 위치 (m)')
plt.title('시간에 따른 횡방향 위치')
plt.legend()
plt.grid(True)
plt.show()

# 2. 조향 각도 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, delta_history, label='조향 각도')
plt.xlabel('시간 (s)')
plt.ylabel('조향 각도 (rad)')
plt.title('시간에 따른 조향 각도')
plt.legend()
plt.grid(True)
plt.show()

# 3. 측면 외란 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, wind_history, label='측면 외란 힘')
plt.xlabel('시간 (s)')
plt.ylabel('외란 힘 (N)')
plt.title('시간에 따른 측면 외란')
plt.legend()
plt.grid(True)
```

```
plt.show()
```



코드 설명:

- 측면 외란 계산:
  - Fwind=500sin(0.5t)F_{\text{wind}} = 500 \sin(0.5 t)Fwind=500sin(0.5t)를 사용하여 시간에 따른 외란 힘을 계산합니다.
- **PID** 제어기 구현:
  - 횡방향 위치 오차 `error`를 계산하고, 적분 및 미분 항을 구합니다.
  - 조향 각도 `delta`를 PID 제어기로 계산합니다.
- 횡방향 운동 방정식:
  - 횡방향 가속도 ay=Fwind+mvδma_y = \frac{F_{\text{wind}} + m v \delta}{m}ay=mFwind+mvδ를 계산합니다.
  - 횡방향 속도 `vy`와 위치 `y`를 업데이트합니다.
- 결과 시각화:
  - 횡방향 위치, 조향 각도, 측면 외란 힘을 그래프로 표시합니다.

결과 분석:

- 횡방향 위치 그래프:
  - PID 제어기를 통해 차량이 차선 중앙을 유지하려고 노력하는 모습을 볼 수 있습니다.
  - 외란에 의해 위치가 변하지만 제어기를 통해 다시 중앙으로 복귀합니다.
- 조향 각도 그래프:
  - 외란에 대응하여 조향 각도가 조절되는 것을 확인할 수 있습니다.
- 측면 외란 그래프:
  - 시간에 따라 외란 힘이 어떻게 변하는지 보여줍니다.

13)

자율주행 차량이 앞차와의 거리를 일정하게 유지하며 주행해야 합니다. 앞차의 속도는 시간에 따라 변합니다. 차량은 PID 제어기를 사용하여 속도를 조절합니다.

목표:

- 앞차와의 거리를 일정한 목표 거리로 유지합니다.
- 앞차의 속도 변화에 대응하여 적절히 속도를 조절합니다.
- 시뮬레이션 결과를 통해 제어 성능을 평가합니다.

조건:

- 목표 거리 dtarget=10 md_{\text{target}} = 10 \, \text{m}dtarget=10m입니다.
- 앞차의 속도는 vlead=15+5sin(0.2t) m/sv_{\text{lead}} = 15 + 5 \sin(0.2 t) \, \text{m/s}vlead=15+5sin(0.2t)m/s로 변합니다.
- 차량의 초기 속도는 v=15 m/sv = 15 \, \text{m/s}v=15m/s입니다.
- PID 이득은 Kp=1.0K_p = 1.0Kp=1.0, Ki=0.1K_i = 0.1Ki=0.1, Kd=0.5K_d = 0.5Kd=0.5로 설정합니다.
- 시뮬레이션 시간은 t=0~60 st = 0 \sim 60 \, \text{s}t=0~60s이며, 시간 간격은 Δt=0.1 s\Delta t = 0.1 \, \text{s}Δt=0.1s입니다.

문제 이해 및 접근 방법:

- 차량은 앞차와의 거리를 일정하게 유지해야 합니다.
- 앞차의 속도가 변하므로, 차량의 속도를 PID 제어기로 조절합니다.
- 상대 거리를 계산하여 오차로 사용합니다.

```
import numpy as np
import matplotlib.pyplot as plt

# 시뮬레이션 파라미터
dt = 0.1
t_end = 60
t = np.arange(0, t_end, dt)

# 목표 거리
d_target = 10.0  # m

# PID 제어기 이득
Kp = 1.0
Ki = 0.1
```

```
Kd = 0.5

# 초기화
v = 15.0  # 차량의 초기 속도
v_lead = 15.0  # 앞차의 초기 속도
d = 10.0  # 초기 거리
integral = 0.0
prev_error = d_target - d

# 로그 저장용 리스트
v_history = []
v_lead_history = []
d_history = []
a_history = []

for ti in t:
    # 앞차의 속도 업데이트
    v_lead = 15 + 5 * np.sin(0.2 * ti)
    v_lead_history.append(v_lead)

    # 거리 업데이트
    d += (v_lead - v) * dt
    d_history.append(d)

    # 거리 오차 계산
    error = d - d_target
    integral += error * dt
    derivative = (error - prev_error) / dt
    prev_error = error

    # 가속도 계산 (간단히 가속도를 제어 입력으로 가정)
    a = - (Kp * error + Ki * integral + Kd * derivative)
    a_history.append(a)

    # 속도 업데이트
    v += a * dt
    v_history.append(v)

# 결과 시각화
# 1. 속도 그래프
```

```
plt.figure(figsize=(10, 5))
plt.plot(t, v_lead_history, label='앞차 속도')
plt.plot(t, v_history, label='자차 속도')
plt.xlabel('시간 (s)')
plt.ylabel('속도 (m/s)')
plt.title('시간에 따른 차량 속도')
plt.legend()
plt.grid(True)
plt.show()

# 2. 거리 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, d_history, label='앞차와의 거리')
plt.axhline(d_target, color='r', linestyle='--', label='목표 거리')
plt.xlabel('시간 (s)')
plt.ylabel('거리 (m)')
plt.title('시간에 따른 앞차와의 거리')
plt.legend()
plt.grid(True)
plt.show()

# 3. 가속도 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, a_history, label='가속도')
plt.xlabel('시간 (s)')
plt.ylabel('가속도 (m/s^2)')
plt.title('시간에 따른 가속도')
plt.legend()
plt.grid(True)
plt.show()
```

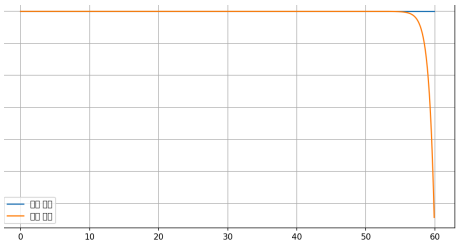
코드 설명:

- 앞차의 속도 업데이트:
  - vlead=15+5sin(0.2t)v_{\text{lead}} = 15 + 5 \sin(0.2 t)vlead=15+5sin(0.2t)를 사용하여 시간에 따른 앞차의 속도를 계산합니다.
- 거리 계산:
  - 앞차와의 거리는 d+=(vlead−v)×dtd += (v_{\text{lead}} - v) \times dtd+=(vlead−v)×dt로 업데이트됩니다.

- **PID** 제어기 구현:
    - 거리 오차 `error`를 계산하고, 적분 및 미분 항을 구합니다.
    - 가속도 `a`를 PID 제어기로 계산합니다.
- 속도 업데이트:
    - 가속도를 사용하여 차량의 속도를 업데이트합니다.
- 결과 시각화:
    - 차량과 앞차의 속도, 앞차와의 거리, 가속도를 그래프로 표시합니다.

결과 분석:

- 속도 그래프:
    - 차량의 속도가 앞차의 속도 변화에 따라 조절되는 것을 확인할 수 있습니다.
- 거리 그래프:
    - 앞차와의 거리가 목표 거리인 10 m 근처에서 유지되는 것을 볼 수 있습니다.
- 가속도 그래프:
    - 앞차의 속도 변화에 대응하여 가속도가 조절됩니다.



14)

자율주행 차량이 오르막과 내리막이 반복되는 도로에서 일정한 속도를 유지하며 주행하려고 합니다. 도로의 경사로 인해 차량의 속도가 영향을 받습니다. PID 제어기를 사용하여 속도를 제어하세요.

목표:

- 차량의 속도를 일정한 목표 속도로 유지합니다.
- 도로의 경사로 인한 영향을 극복합니다.
- 시뮬레이션 결과를 통해 제어 성능을 평가합니다.

조건:

- 목표 속도 vtarget=25 m/sv_{\text{target}} = 25 \, \text{m/s}vtarget=25m/s입니다.
- 도로의 경사 각도는 θ(t)=3∘sin(0.05t)\theta(t) = 3^\circ \sin(0.05 t)θ(t)=3∘sin(0.05t)로 변합니다.
- 차량의 질량은 m=1300 kgm = 1300 \, \text{kg}m=1300kg입니다.

- PID 이득은 Kp=500K_p = 500Kp=500, Ki=30K_i = 30Ki=30, Kd=80K_d = 80Kd=80로 설정합니다.
- 시뮬레이션 시간은 t=0~100 st = 0 \sim 100 \, \text{s}t=0~100s이며, 시간 간격은 Δt=0.1 s\Delta t = 0.1 \, \text{s}Δt=0.1s입니다.

문제 이해 및 접근 방법:

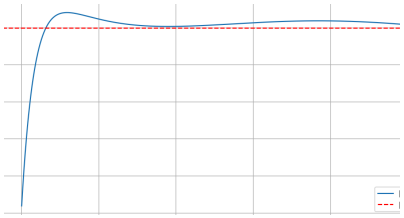
- 도로 경사의 변화로 인해 차량에 작용하는 중력 성분이 달라집니다.
- PID 제어기를 사용하여 엔진 출력을 조절하여 속도를 일정하게 유지합니다.
- 차량의 운동 방정식을 사용하여 속도를 업데이트합니다.

```
import numpy as np

import matplotlib.pyplot as plt



# 시뮬레이션 파라미터

dt = 0.1

t_end = 100

t = np.arange(0, t_end, dt)



# 차량 파라미터

m = 1300  # kg

v_target = 25.0  # m/s



# PID 제어기 이득

Kp = 500

Ki = 30

Kd = 80
```

```python
# 초기화
v = 0.0
F_engine = 0.0
integral = 0.0
prev_error = v_target - v

# 로그 저장용 리스트
v_history = []
F_history = []
theta_history = []

for ti in t:
    # 도로 경사 계산 (rad 단위)
    theta = np.deg2rad(3 * np.sin(0.05 * ti))
    theta_history.append(np.rad2deg(theta))

    # 중력에 의한 힘 계산
    F_gravity = m * 9.81 * np.sin(theta)

    # 속도 오차 계산
    error = v_target - v
    integral += error * dt
    derivative = (error - prev_error) / dt
```

```
    prev_error = error

    # PID 제어기로 엔진 힘 계산
    F_engine = Kp * error + Ki * integral + Kd * derivative

    # 차량의 가속도 계산
    a = (F_engine - F_gravity) / m

    # 속도 업데이트
    v += a * dt

    # 로그 저장
    v_history.append(v)
    F_history.append(F_engine)

# 결과 시각화
# 1. 속도 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, v_history, label='실제 속도')
plt.axhline(v_target, color='r', linestyle='--', label='목표 속도')
plt.xlabel('시간 (s)')
plt.ylabel('속도 (m/s)')
plt.title('시간에 따른 차량 속도')
```

```
plt.legend()

plt.grid(True)

plt.show()



# 2. 엔진 힘 그래프

plt.figure(figsize=(10, 5))

plt.plot(t, F_history, label='엔진 힘')

plt.xlabel('시간 (s)')

plt.ylabel('엔진 힘 (N)')

plt.title('시간에 따른 엔진 힘')

plt.legend()

plt.grid(True)

plt.show()



# 3. 도로 경사 그래프

plt.figure(figsize=(10, 5))

plt.plot(t, theta_history, label='도로 경사 각도')

plt.xlabel('시간 (s)')

plt.ylabel('경사 각도 (deg)')

plt.title('시간에 따른 도로 경사 변화')

plt.legend()

plt.grid(True)

plt.show()
```



코드 설명:

- 도로 경사 계산:
  - θ(t)=3∘sin(0.05t)\theta(t) = 3^\circ \sin(0.05 t)θ(t)=3∘sin(0.05t)를 사용하여 경사 각도를 계산합니다.
- 중력에 의한 힘 계산:
  - Fgravity=mgsin(θ)F_{\text{gravity}} = m g \sin(\theta)Fgravity=mgsin(θ)
- **PID** 제어기 구현:
  - 속도 오차 `error`를 계산하고, 적분 및 미분 항을 구합니다.
  - 엔진 힘 `F_engine`을 PID 제어기로 계산합니다.
- 차량의 운동 방정식:
  - 가속도 a=Fengine−Fgravityma = \frac{F_{\text{engine}} - F_{\text{gravity}}}{m}a=mFengine−Fgravity를 계산합니다.
  - 속도를 업데이트합니다.
- 결과 시각화:
  - 속도, 엔진 힘, 도로 경사 각도를 그래프로 표시합니다.

결과 분석:

- 속도 그래프:
  - 차량의 속도가 목표 속도인 25 m/s 근처에서 유지되는 것을 확인할 수 있습니다.
- 엔진 힘 그래프:
  - 도로 경사의 변화에 따라 엔진 힘이 조절됩니다.
- 도로 경사 그래프:
  - 경사 각도가 주기적으로 변하는 것을 확인할 수 있습니다.

15)

자율주행 차량이 곡선 도로를 주행하고 있습니다. 차량은 도로의 곡률에 따라 조향 각도를 조절해야 합니다. PID 제어기를 사용하여 차량의 방향을 제어하세요.

목표:

- 차량이 도로의 곡선을 따라 주행하도록 방향을 제어합니다.
- 차량의 횡방향 오차를 최소화합니다.
- 시뮬레이션 결과를 통해 제어 성능을 평가합니다.

조건:

- 도로의 곡률은 $\kappa(x) = 0.01 \sin(0.1 x)$로 주어집니다.
- 차량의 속도는 $v = 15 \, \text{m/s}$로 일정합니다.
- PID 이득은 $K_p = 0.3$, $K_i = 0.05$, $K_d = 0.2$로 설정합니다.
- 시뮬레이션 시간은 $t = 0 \sim 50 \, \text{s}$이며, 시간 간격은 $\Delta t = 0.1 \, \text{s}$입니다.

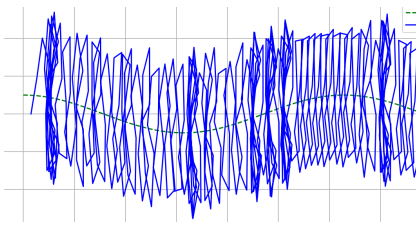
문제 이해 및 접근 방법:

- 도로의 곡률에 따라 차량의 목표 경로가 결정됩니다.
- 차량의 횡방향 오차를 계산하여 PID 제어기로 조향 각도를 조절합니다.
- 자전거 모델을 사용하여 차량의 운동을 시뮬레이션합니다.

```python
import numpy as np
import matplotlib.pyplot as plt

# 시뮬레이션 파라미터
dt = 0.1
t_end = 50
t = np.arange(0, t_end, dt)

# 차량 파라미터
v = 15.0  # m/s

# PID 제어기 이득
Kp = 0.3
Ki = 0.05
Kd = 0.2

# 초기화
x = 0.0
y = 0.0
theta = 0.0
delta = 0.0
```

```
integral = 0.0
prev_cte = 0.0

# 로그 저장용 리스트
x_history = []
y_history = []
delta_history = []
cte_history = []

for ti in t:
    # 도로의 곡률 계산
    kappa = 0.01 * np.sin(0.1 * x)

    # 도로의 목표 방향 계산
    theta_desired = kappa * x

    # 횡방향 오차 계산
    cte = y - (0.5 * np.cos(0.1 * x))

    # PID 제어기 계산
    integral += cte * dt
    derivative = (cte - prev_cte) / dt
    prev_cte = cte

    delta = - (Kp * cte + Ki * integral + Kd * derivative)

    # 차량의 위치 및 방향 업데이트 (자전거 모델)
    x += v * np.cos(theta) * dt
    y += v * np.sin(theta) * dt
    theta += (v / 2.5) * delta * dt

    # 로그 저장
    x_history.append(x)
    y_history.append(y)
    delta_history.append(delta)
    cte_history.append(cte)

# 결과 시각화
# 1. 차량 궤적과 도로 곡선
road_x = np.linspace(0, max(x_history), len(x_history))
```

```
road_y = 0.5 * np.cos(0.1 * road_x)

plt.figure(figsize=(10, 5))
plt.plot(road_x, road_y, 'g--', label='도로 경로')
plt.plot(x_history, y_history, 'b-', label='차량 궤적')
plt.xlabel('x 위치 (m)')
plt.ylabel('y 위치 (m)')
plt.title('차량 궤적과 도로 경로')
plt.legend()
plt.grid(True)
plt.show()

# 2. 횡방향 오차 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, cte_history, label='횡방향 오차')
plt.xlabel('시간 (s)')
plt.ylabel('오차 (m)')
plt.title('시간에 따른 횡방향 오차')
plt.legend()
plt.grid(True)
plt.show()

# 3. 조향 각도 그래프
plt.figure(figsize=(10, 5))
plt.plot(t, delta_history, label='조향 각도')
plt.xlabel('시간 (s)')
plt.ylabel('조향 각도 (rad)')
plt.title('시간에 따른 조향 각도')
plt.legend()
plt.grid(True)
plt.show()
```



코드 설명:

- 도로 곡률 및 목표 방향 계산:
  - 곡률 $\kappa(x) = 0.01 \sin(0.1 x)$를 사용하여 도로의 형태를 정의합니다.
  - 목표 방향은 단순히 곡률과 x 위치를 곱하여 근사합니다.
- 횡방향 오차 계산:
  - 차량의 y 위치와 도로의 y 위치 간의 차이를 계산합니다.
- **PID** 제어기 구현:
  - 횡방향 오차를 기반으로 조향 각도를 계산합니다.
- 자전거 모델 사용:
  - 차량의 위치와 방향을 업데이트합니다.
- 결과 시각화:
  - 차량의 궤적과 도로 경로, 횡방향 오차, 조향 각도를 그래프로 표시합니다.

결과 분석:

- 차량 궤적 그래프:
  - 차량이 도로의 곡선을 따라 주행하는 것을 확인할 수 있습니다.
- 횡방향 오차 그래프:
  - 오차가 작은 범위 내에서 유지되는 것을 볼 수 있습니다.
- 조향 각도 그래프:
  - 도로의 곡률에 따라 조향 각도가 조절됩니다.