**고급소프트웨어실습I Modular Pipeline 전체보고서**

**20211584 장준영**

**0. Overview**

모듈 공정을 통한 자율 주행의 구현은 크게 1)레인 감지, 2)경로 계획, 3)자동차 조종으로 나뉜다. 가장 먼저 위에서 바라본 도로의 이미지를 입력으로 받아 차선의 경계를 찾는다. 이후 찾은 차선의 경계를 바탕으로, 차선의 중앙부를 따라가도록 경로를 계획한다. 마지막으로, 계획된 경로를 통해 종 및 횡 방향으로 어떻게 이동해야 할지 설계한다. 이 과정을 모두 구현하면 자동차가 차선을 안정적으로 따라가게 된다.

**1. 레인 감지(lane\_detection.py)**

모듈 공정에서 레인 감지는 lane\_detection.py에서 진행된다. 전체적으로, 이 코드에선 다음과 같은 과정을 거쳐 차선을 감지한다.

1) 이미지 전처리 : cut\_gray method에선 위에서 바라본 도로의 모습 이미지(입력)을 그레이스케일 이미지로 바꾸고, 실질적인 도로를 담고 있는 차의 머리 윗 부분까지 잘라 반환한다. 이는 이후 그라디언트 계산을 통해 경계점(edge)을 찾아내는데 사용된다.

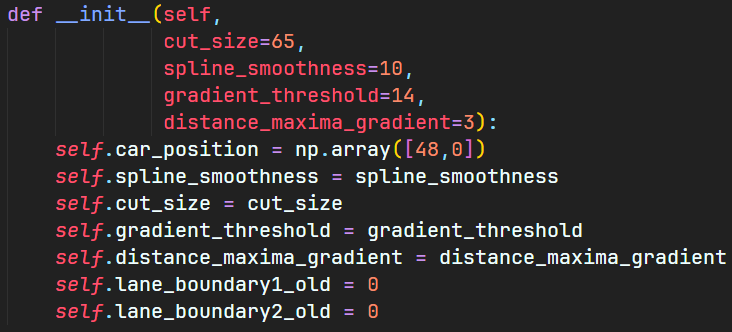
2) 경계점 감지 : edge\_detection method에선 전처리된 이미지에서 그라디언트 절대값을 통해 경계점을 감지한다. 보다 정확한 경계점 감지를 위해 일정 임계값보다 작은 그라디언트 값을 갖는 부분에 대해서는 무시한다.

3) 차선 감지 : find\_maxima\_gradient\_rowwise method에선 edge\_detection을 통해 얻은 그라디언트 이미지에서 각 행의 그라디언트 최대값을 찾는다. 도로와 잔디의 경계, 곧 차선의 경계에서 그라디언트 최대값을 갖기 떄문에, 이 과정에서 이미지의 모든 경계 중 차선의 경계를 찾게 된다.

4) 차선의 시작점 찾기 : find\_first\_lane\_point method에선 edge\_detection을 통해 얻은 그라디언트 이미지에서 차량 머리 위의 첫 번째 차선 경계점을 찾는다.

5) 차선 구성 : lane\_detection method에선 실제 차선 위 점을 탐지하여 spline으로 구성해 반환한다. 앞서 설명한 과정을 거쳐 차선 경계점을 찾으면, 이전까지 구성해온 차선을 따라 가장 가까운 경계점을 계속 추가하여 연속된 차선을 만들 수 있도록 한다. 이 과정은 valid 한 차선을 모두 찾을 수 있을 때까지 반복하며, 추가된 점을 미분 가능하게 잇는 과정은 후술한다.

6) spline fitting : 이전에 찾은 차선 경계점을 통해, 차선을 나타내는 spline을 만드는 과정이다. spline을 만들 만큼 충분한 경계점이 없을 경우 이전에 만들었던 spline을 그대로 사용한다.



parameter 초기화 과정이다. 여기서 중요한 parameter는 gradient\_threshold가 있는데, 이미지의 경계점을 찾는 과정에서 주요한 그라디언트 값으로 보이지 않는 사소한 노이즈를 걸러내는 임계값이다.

왼쪽부터 gradient\_threshold를 3, 8, 14로 설정한 모습이다. 3일 때는 차선의 무늬, 잔디의 무늬 등을 노이즈로서 걸러내지 못해 차선을 제대로 감지하지 못한다. 8일 때는 차선을 얼추 감지하지만, 주행 중 민감한 환경에서 조금씩 노이즈가 발생한다. 14일 때 노이즈를 무시하고 비로소 차선만을 감지하게 된다. 코드 작성 과정에서, 이 값을 주로 조정하면서 최적의 값을 14로 찾아 설정하였다. (이 이상의 값은 타입/사이즈 에러가 발생하거나 차선조차 제대로 감지하지 못했음)

**2. 경로 계획(waypoint\_prediction.py)**

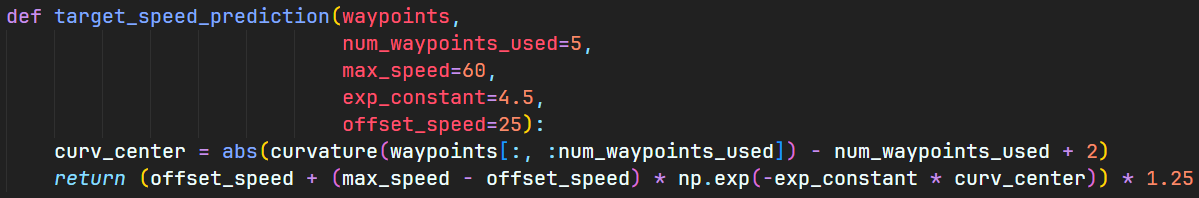
모듈 공정에서 경로 계획은 waypoint\_prediction.py에서 이루어진다. 다음과 같은 과정을 통해, 앞서 lane\_detection.py에서 감지한 차선을 따라 차가 어떻게 움직여야 할지 계획한다.

1) 정규화 및 곡률 계산 : 실질적인 계산에 앞서, normalize 함수에선 입력 벡터의 크기를 1로 정규화하여 계산을 용이하게 만든다. 이후, curvature 함수에서 웨이포인트 간의 곡률을 계산한다. 이 곡률 값은 이후 차가 속도를 얼마나 늘리거나 줄여야 할지 결정하는 요인이 된다.

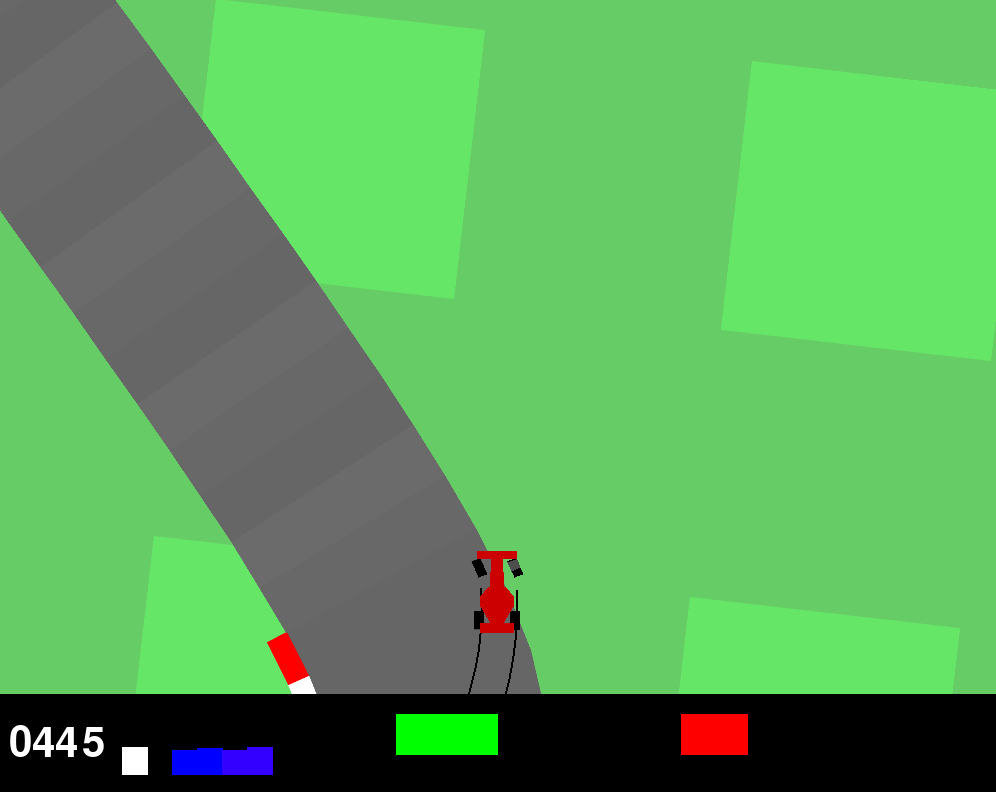
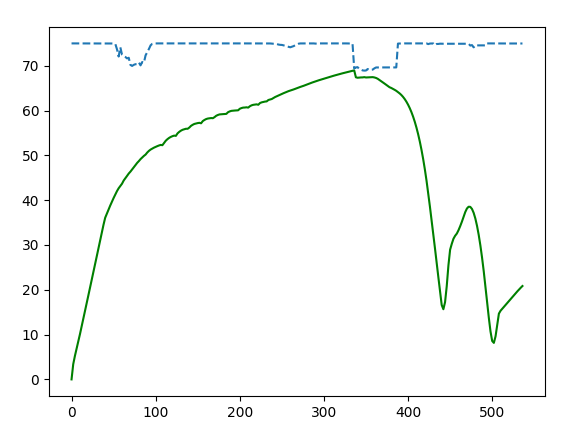
2) smoothing 최적화 : smoothing\_objective 함수에선 웨이포인트와 중심점 사이의 거리를 최소화하고 곡률은 최대화하여, 차량이 도로의 중심선을 최대한 따라가게 함과 동시에 부드러운 곡선으로 이동하게 한다.

3) 웨이포인트 예측 : waypoint\_prediction 함수에선 입력으로 받은 차선 spline을 통해 이후의 웨이포인트를 예측한다. 웨이포인트는 차선의 중심점이 될 것이므로, 두 차선/경계점의 중점을 계산하여 예측한다.

4) 속도 예측 : 웨이포인트를 모두 예측했다면, target\_speed\_prediction 함수에선 차가 움직여야 하는 속도를 예측한다. 곡률이 큰 경우 속도를 낮추고, 작은 경우 속도를 늘려 최적화한다. 이 외에도 다양한 parameter가 존재한다.



parameter가 중요한 함수는 target\_speed\_prediction이 있다. 실제 구현 과정에서도, 이 부분의 구현이 가장 어려웠다. 이 과정은 **3.**에서 설명할 실제 자동차 조종을 구현하는 과정에서 다시 조절하게 되었는데, 속도 예측 과정에 따라 reward(score)가 크게 달라졌기 때문이다. 특히 offset\_speed가 중요하다. 경험에 따르면, offset\_speed가 작아질수록 곡률이 조금이라도 발생했을 때 속도를 확실하게 줄인다.

offset\_speed를 55로 바꾸었을 때의 조종 모습이다. 큰 곡률의 도로가 나타났을 때(t=300 부근), 속도를 충분히 줄이지 않아(파란 점선이 목표 속도, 초록 실선이 실제 속도임) 차가 차선을 이탈하고, 결국 오작동하게 되는 모습을 확인할 수 있다. 따라서, 적절한 offset\_speed를 찾는 것이 가장 중요했다.



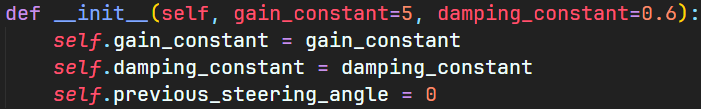
내가 작성한 속도 예측 수식이다. 일반적인 수식에서 수정한 것은, offset\_speed parameter와 마지막에 1.25를 곱하는 점이다. 먼저 1.25를 곱한 이유는, 기본 수식의 예측 속도가 너무 느렸기 때문이다. 곡률이 없는 구간에서도 느린 속도로 가다보니 많이 움직일 수 없어 reward가 낮아졌다. 이를 개선하기 위해 1.25라는 작은 숫자를 곱해 기본적으로 빠른 속도로 달리게 한 것이다. 하지만, 이에 따라 곡률이 큰 구간에서 오작동하게 될 가능성이 커졌다. 이에 따라, offset\_speed를 좀 더 작은 값(25)로 수정하여 곡률이 있을 땐 확실하게 속도를 낮출 수 있도록 하였다. 이 공식을 통해, 최대한 빠른 속도로 달리다가 오작동의 가능성이 존재하는 곡률 구간에선 속도를 낮춰 제대로 완주할 수 있도록 만들었다. 이 과정이 waypoint\_prediction.py를 작성하면서 가장 까다로웠던 부분이다.

**3. 자동차 조종(lateral\_control.py, longitudinal\_control.py)**

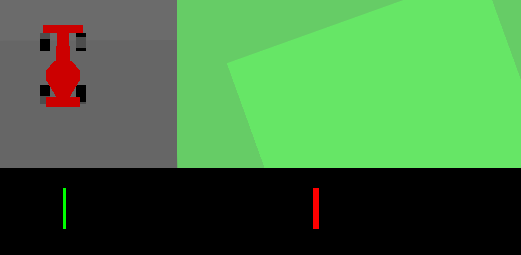
waypoint\_prediction.py에서 경로를 성공적으로 예측하면, 예측한 경로에 따라 차를 조종해야 한다. 자를 조종하는데 필요한 것은 횡 조종 요소와 종 조종 요소인데, 횡 요소(steering\_angle)은 lateral\_control.py에서 stanley 제어 기법을 사용하고, 종 요소(gas/brake)는 longitudinal\_control.py에서 PID 제어 기법을 사용한다. 먼저 lateral\_control.py에선 다음과 같은 과정이 이루어진다.

1) 초기화 : stanley 제어 구현에 필요한 gain, damping 등의 constant를 초기화한다.

2) stanley 제어 : stanley 제어를 단순 공식 차원에서 구현한다. 차량의 현재 위치와 목표 위치 사이의 각도를 계산하고, 이 값에 차량의 현재 위치에서 차선 중앙까지의 거리에 비례하는 각도를 더한다. 그 다음, 이전 제어 입력과 현재 새로 계산된 제어 입력의 차이에 damping constant를 곱해 damping 값을 계산한다. 마지막으로, max(계산된 제어 각도, 임계값=0.4rad)를 반환한다. 이렇게 하면 차량이 웨이포인트를 횡 방향으로 부드럽게 따라갈 수 있다.



stanley 제어의 큰 영향을 주는 parameter는 damping\_constant가 있다. 앞서 설명한 대로, 이전의 제어와 현재의 제어간의 간극을 줄여주는 역할을 한다. 이 값이 커지면 커질수록 제어가 완만해진다.



damping\_constant를 0으로 했을 때의 좌우 제어 게이지이다. 이전 값의 영향을 아예 받지 않기 때문에, 중앙선을 따르기 위해 계속해서 조금씩 진동한다. 이렇게 되면 이전에 설명했던 bang-bang 제어처럼 시스템에 큰 스트레스를 줄 수 있고, 속도가 완만하지 않아 불안정하다.



damping\_constant를 3으로 했을 때의 주행 모습이다. 이전 값의 영향을 너무 크게 받아 이전에 중앙선을 따르기 위해 취했던 방향으로 계속 진행하고, 그러다 보면 차선을 이탈하기 쉽다. 경험적으로, 최적의 값은 0.6이었기 때문에 그렇게 설정했다.

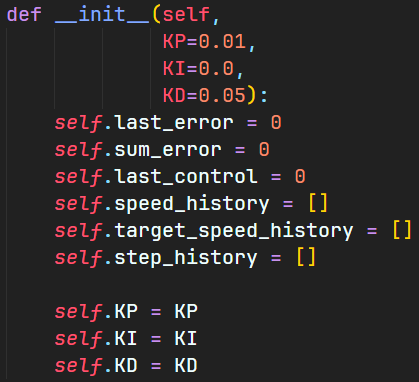
다음은 longitudinal\_control.py의 전체 과정이다.

1) 초기화 : KP, KI, KD라는 세 가지 파라미터를 초기화한다. 이는 PID 제어에서 사용되는 비례 상수, 적분 상수, 미분 상수이다.

2) PID 신호 계산 : PID 제어 신호를 계산한다. 현재 속도와 목표 속도를 입력으로 받아, 차량의 속도 제어를 위한 PID 신호를 계산한다. 먼저, 현재 속도와 목표 속도 사이의 오차를 계산한다. 이후 계산된 오차에 비례 상수를 곱하고, 적분 상수에 오차의 적분 값을 곱하고, 미분 상수에 오차의 미분 값을 곱하여 더한 값을 반환한다.

3) 제어 : PID 제어 신호를 사용하여 차량의 가속 및 제동을 결정한다. PID 신호가 양수이면 차량을 가속하고, 음수이면 차량을 제동하여 속도를 조절한다. 차선 이탈 및 오류를 방지하기 위해, 임계값 이상의 가속 및 제동은 없도록 한다.

4) 그래프 그리기 : PID 신호를 통한 목표 속도와 실제 속도를 보여주는 그래프를 그린다.

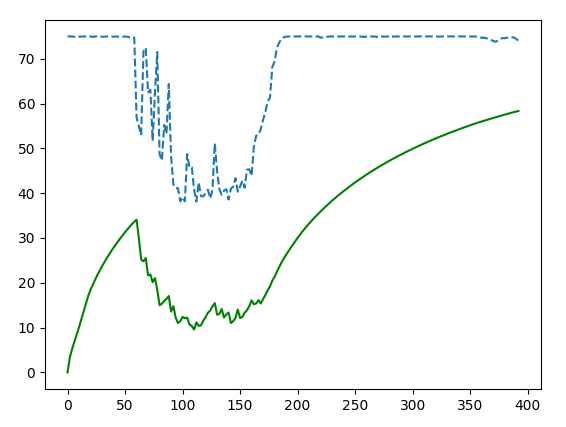
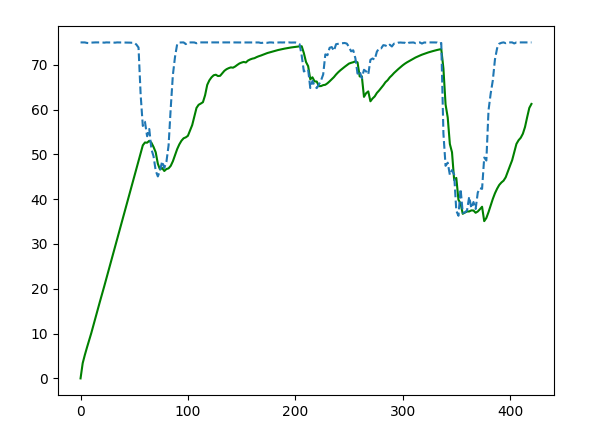


PID 제어에 있어 가장 중요한 parameter는 KP, KI, KD이다.

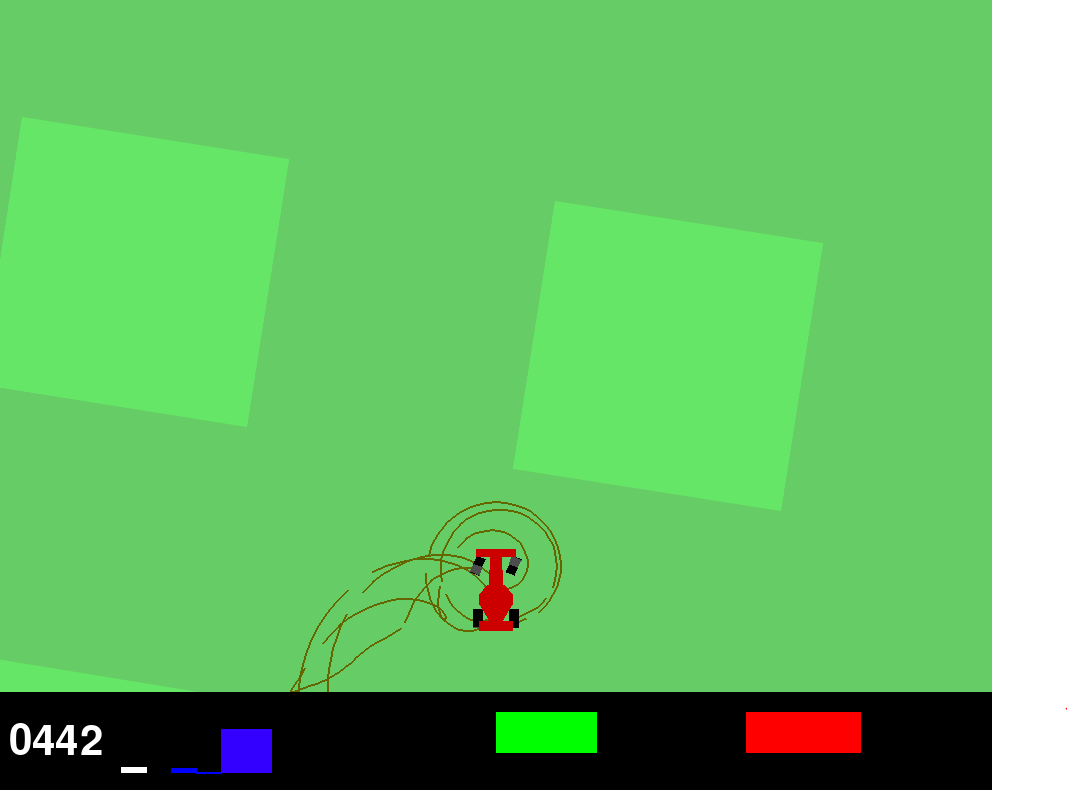
1) KP (비례 상수) : KP는 현재 속도와 목표 속도 사이 오차의 비례 상수이다. KP가 크면, 오차가 클 때 더 큰 제어 신호를 생성하므로 차량의 속도를 변화 시마다 즉각적으로 조정할 수 있다. 그러나 KP가 너무 크면, 차량의 속도 조정이 과도하게 이루어져서 시스템에 스트레스를 주고 차량의 주행이 불안정해질 수 있다. 경험적으로 최적의 값을 0.01로 설정하고 있다.

2) KI (적분 상수) : KI는 오차 적분의 비례 상수이다. 오차가 오랜 시간 지속적으로 발생하는 경우, 적분값이 커져서 제어 신호가 증가한다. 따라서 KI가 크면, 오차가 지속적으로 발생하는 경우에 더 큰 제어 신호를 생성하므로 차량의 속도를 더욱 안정적으로 조정할 수 있다. 그러나 KI가 너무 크면, 즉각적인 변화에 대응하지 못하거나 속도가 과도하게 증가하여 차량의 주행이 불안정해질 수 있다. 우리의 주행 환경에선 오차가 지속적으로 발생할 일이 드물기 때문에, 이는 0으로 설정되어 있다.

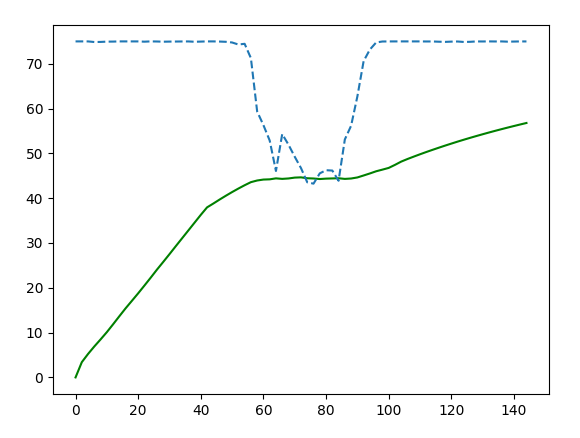
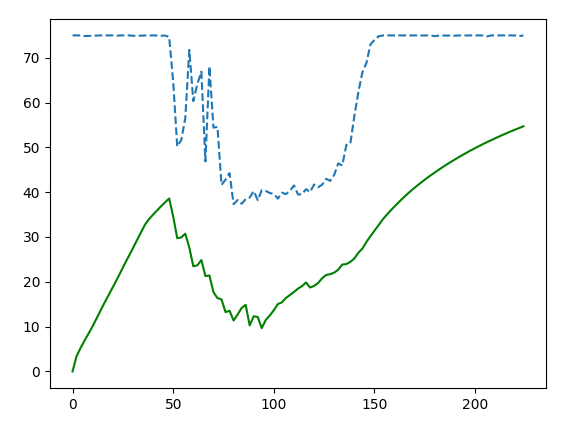
3) KD (미분 상수) : KD는 오차 변화율(오차 미분)의 비례 상수이다. 오차가 빠르게 변하는 경우, 미분값이 커져서 제어 신호가 증가한다. 따라서 KD가 크면, 오차가 빠르게 변하는 경우에 더 큰 제어 신호를 생성하므로 차량의 속도를 더 빠르게 조정할 수 있다. 그러나 KD가 너무 크면, 변화가 일어날 때마다 제어 신호가 과도하게 변동하여 차량의 주행이 불안정해질 수 있다. 경험적으로 최적의 값을 0.05로 설정하고 있다.

왼쪽부터 KP가 0.005인 그래프와 0.05인 그래프이다. 0.005인 경우, 목표 속도에 도달하는 속도가 굉장히 느리고, 변화에 느슨하게 대응하는 것을 볼 수 있다. 반면, 0.05인 경우 목표 속도에 굉장히 빠르게 도달하고, 목표 속도가 변화하여 오차 값이 커졌을 때 급격하게 변화하는 것을 볼 수 있다.



하지만 KP가 0.05인 경우 속도의 급격한 변화가 lateral control과 상응하지 못하여 차선을 이탈하게 되는 경우가 생길 수 있다. 따라서, 적절한 값을 경험적으로 0.01로 설정하였다.

다음으로 KD를 변화시켜보았다. 왼쪽부터 KD가 0.01, 0.1인 경우의 그래프이다. KD가 0.01인 경우 목표 속도의 변화가 있어도 느슨하게 실제 속도가 변화하지만, 0.1인 경우 목표 속도가 변화하는 것에 기민하게 대응하여 실제 속도를 변화시킨다. 이 부분에서 제어가 잘 되어 목표 지점에 제대로 도달하고, 최대한 멀리까지 움직일 수 있는 KD를 경험적으로 0.05로 설정하였다.

**\*\* 한 학기 동안 수고 많으셨습니다. \*\***