

# 국민대학교 자율주행 경진대회

## < 예선 1 - 1 보고서 >

| 참가대학 |        | 국민대학교 소프트웨어학부 |     |
|------|--------|---------------|-----|
| 참가팀명 | Focar칩 | 팀장            | 송경민 |
| 팀원   | 김지민    | 팀원            | 강기범 |
| 팀원   | 박준석    | 팀원            | 박재훈 |



# 목차

## 1. 개요

### 1.1) 흐름도

## 2. 개발과정

### 2.1) 초음파

#### 2.1.1) ROS

#### 2.1.2) fr만 쓰게 된 이유

### 2.2) PID

#### 2.2.1) P

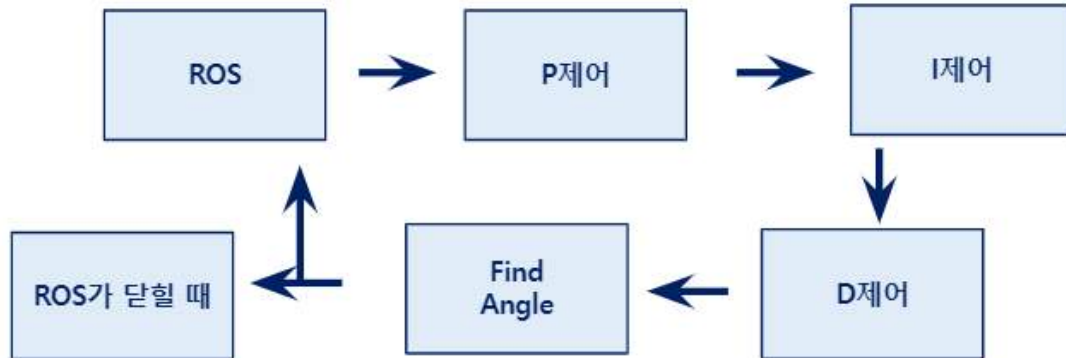
#### 2.2.2) PI

#### 2.2.3) PID

## 3. 마무리

## 1. 개요

### 1-1. 흐름도



요구사항 : 시뮬레이터의 차량이 벽과 충돌하지 않고 시작 지점에서 시계 방향으로 돌아서 다시 원점에 도착하라.

## 2. 개발과정

## 2-1. 초음파

### 2.1.1) ROS

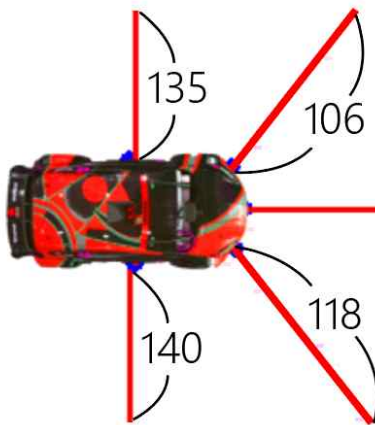















```
import rospy, math, time
from std_msgs.msg import Int32MultiArray
fr = 0.0
def callback(msg):
    global fr
    fr = msg.data[2]
    print(msg)
rospy.init_node('guide')
motor_pub = rospy.Publisher('xycar_motor_msg', Int32MultiArray, queue_size=1)
ultra_sub = rospy.Subscriber('ultrasonic', Int32MultiArray, callback)
xycar_msg = Int32MultiArray()
```

ROS MASTER에게 guide라는 이름으로 노드를 선언하고 xycar\_motor\_msg 토픽에 발행한다.

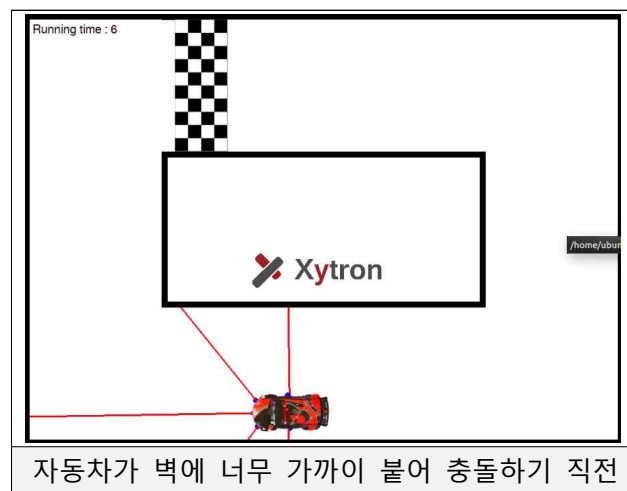
또한 ultrasonic 토픽의 데이터를 구독한다. 데이터를 해당 토픽에서 수신할 때마다 callback을 호출해 fr데이터를 받아온다.

Xycar\_msg를 Int32MultiArray()객체로 초기화한다.

### 2.1.2) fr만 쓰게 된 이유

|  | <table><tr><th>fl</th><th>fm</th><th>fr</th><th>r</th><th>l</th></tr><tr><td></td><td></td><td></td><td></td><td></td></tr></table> <pre>data: [106, 477, 118, 0, 0, 0, 140, 135] layout:   dim: []   data_offset: 0 data: [106, 475, 118, 0, 0, 0, 140, 135] layout:   dim: []   data_offset: 0 data: [106, 473, 118, 0, 0, 0, 140, 135] layout:   dim: []   data_offset: 0</pre> | fl  | fm  | fr  | r | l |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
| fl  | fm  | fr  | r   | l   |   |   |   |   |   |   |   |
|  |    |  |  |  |   |   |   |   |   |   |   |
| 시뮬레이터 ultrasonic값   | ultrasonic  |   |   |   |   |   |   |   |   |   |   |

시뮬레이터를 처음 실행했을 때 자동차는 수평 상태이며 이 때 fr, fl, r, l의 합은 499또는 498로 일정하다. 두 경우의 평균을 계산하면 124.625이다. 이 값을 이용하면 자동차는 수평 상태를 유지할 수 있다.



하지만 이 값을 기준으로 유지하면 사진과 같이 자동차가 벽에 너무 가까이 붙어있을 때에도 직진하여 급커브 구간에서 충돌할 확률이 높기 때문에 자동차를 최대한 트랙의 중앙에서 주행시킨다.

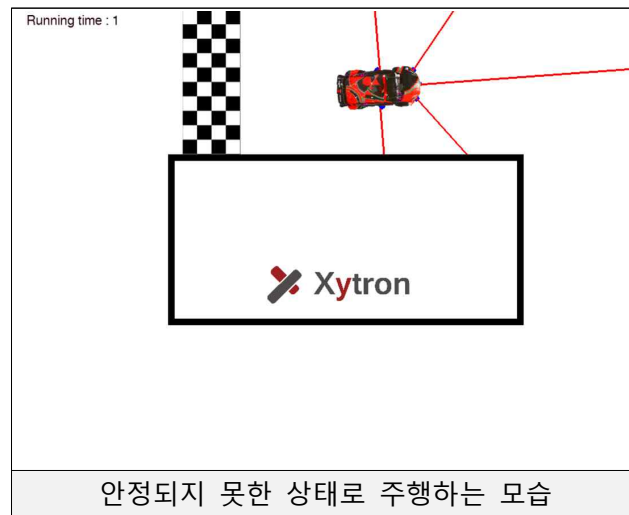
자동차를 트랙의 중앙에서 주행시키려면 센서 하나를 정해야 한다. 급커브 구간까지 확인할 수 있는 fr센서를 사용한다. fr센서와 fl센서의 합은 224로 일정하다. fr센서를 112로 유지하면 자동차는 트랙의 중앙에서 주행할 수 있다. setPoint를 112로 설정한다.



## 2. 개발과정

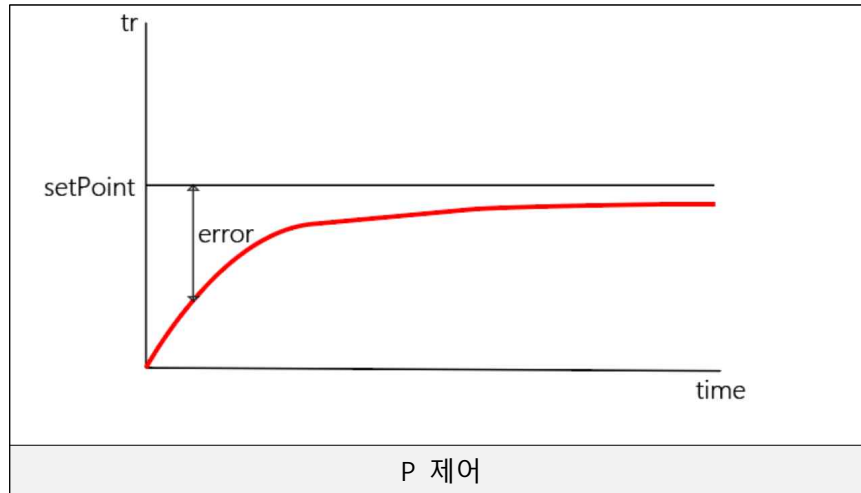
### 2-1. 단순한 방법

- fr센서를 setPoint로 유지하려면 fr센서 길이가 setPoint보다 커질 때 angle값을 양수, 작아질 때 angle값을 음수로 설정하면 유지할 수 있다. 그러나 setPoint에 가까워져도 제어하는 양은 그대로이기 때문에 완벽하게 유지하지 못하고 진동하게 된다. 더 정확하게 유지할 수 있는 다른 방법을 찾아야한다.



## 2-2. P(비례) 제어

- 목표값에 가까워질 수록 제어하는 양을 줄이면 더 안정적으로 fr값을 setPoint로 유지할 수 있을것이다. 목표값과 현재 데이터의 차이 (setPoint - fr)인 error값을 이용해 차를 제어한다.

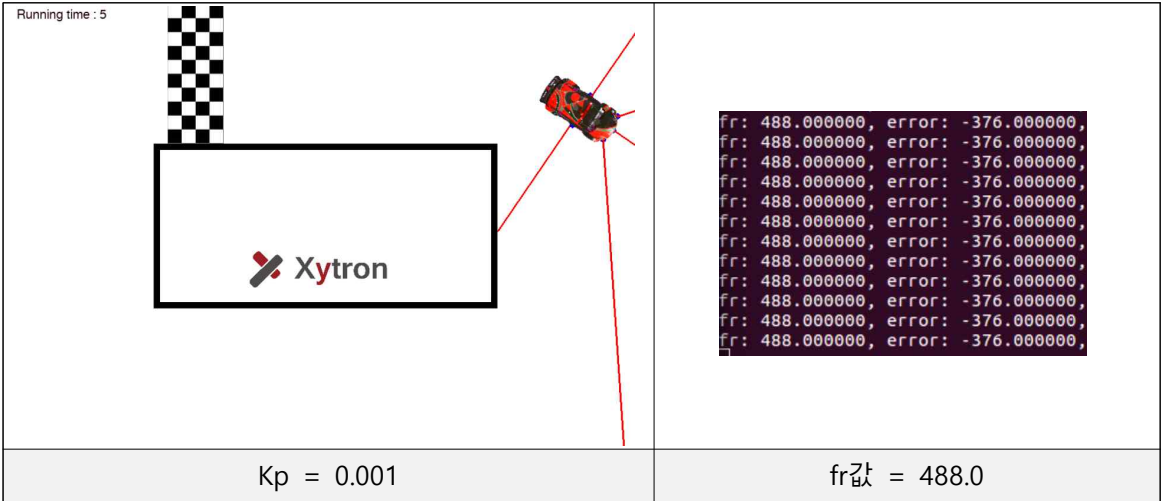


다음은 비례 제어를 통한 결과를 도출하는 식이다.

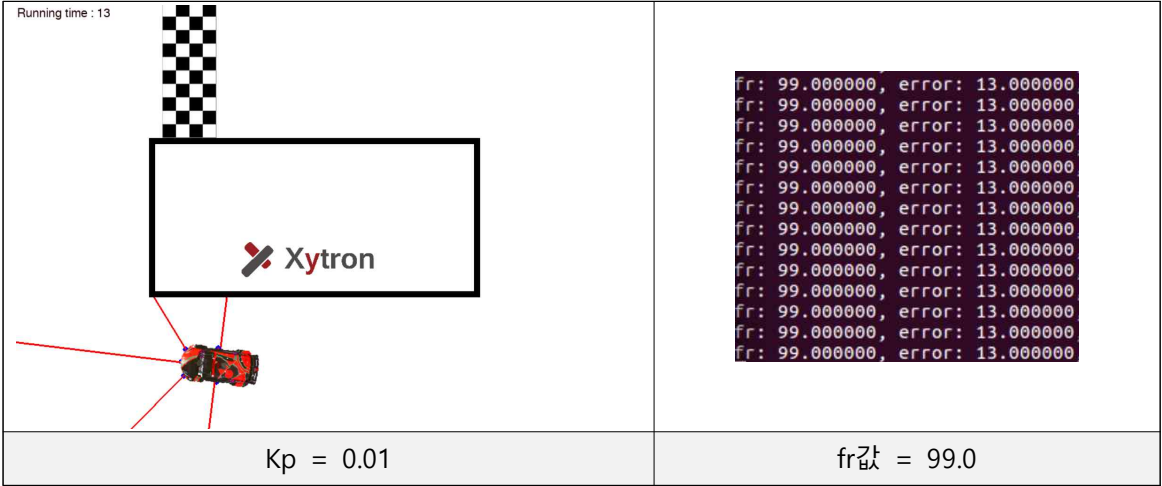
$$p(t) = k_p e(t)$$

Kp값이 커질수록 목표값에 빠르게 접근할 수 있지만 진동이 심해질 수 있다. 작아질수록 안정적으로 접근할 수 있지만 너무 작은 값을 설정할 경우 목표값에 도달하

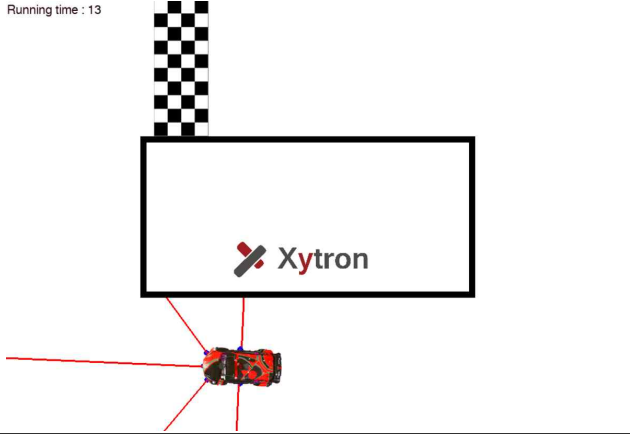
지 못할 수 있다. 그러므로 error값에 곱할 Kp값을 하나씩 대입해서 적절한 값을 찾는다.



Kp = 0.001일 때 급커브 구간에서 setPoint에 도달하지 못하고 충돌한다. Kp값을 일정량 크게 설정해 실험해본다.



Kp = 0.01 일 때 충돌하지는 않지만 setPoint에 도달하지 못하고 99에 머물러있는 것을 볼 수 있다. Kp값을 일정량 증가해 실험해본다.

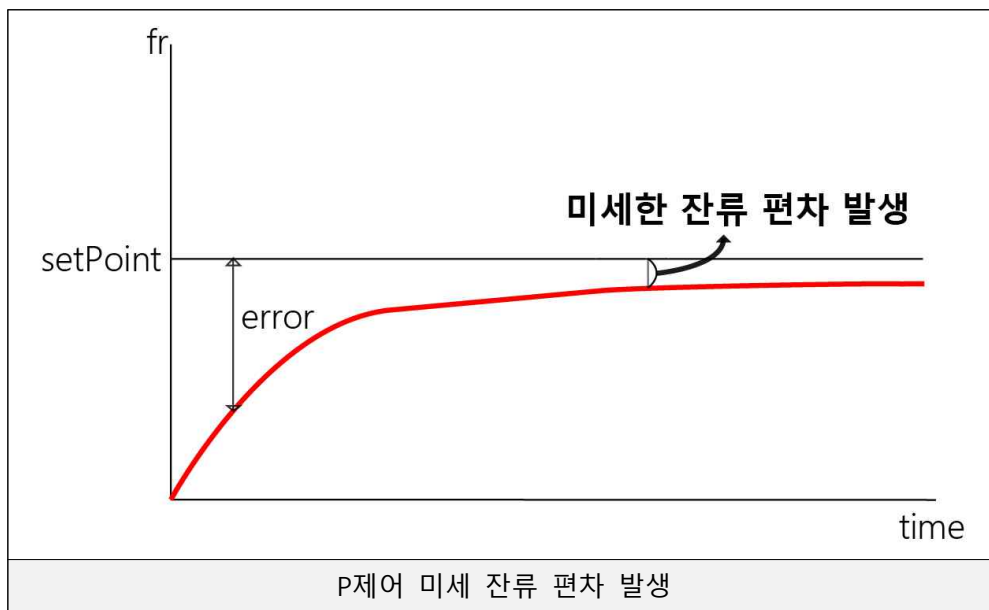
|  |   |
|--|---|
| <div>Running time : 13</div>  | <pre>fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000, fr: 110.000000, error: 2.000000,</pre> |
| Kp = 0.05  | fr값 = 110.0   |

Kp = 0.05일 때 setPoint에 가장 가깝게 도달할 수 있는 것을 볼 수 있다.

1. error = setPoint - data
2. kp = 0.05
3. pTerm = kp \* error

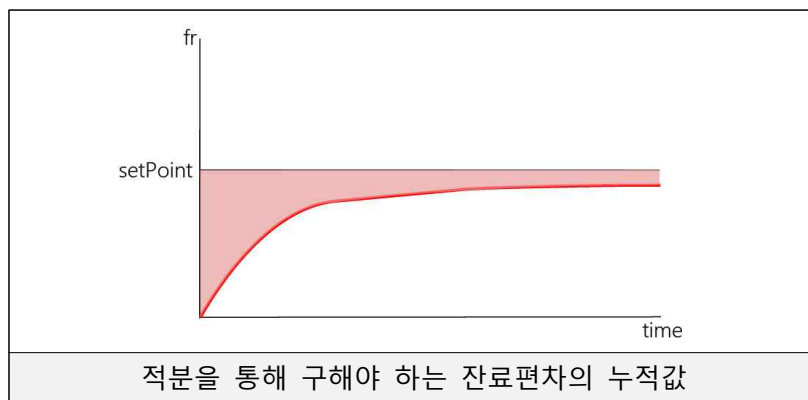
error값은 1행에서 setPoint - 현재 fr값인 편차값이다. 여기서 setPoint는 자동차가 트랙의 중앙을 주행할 수 있도록 112로 설정하였다. P제어 결과값인 pTerm은 Kp에 error값을 곱해 구할 수 있다.

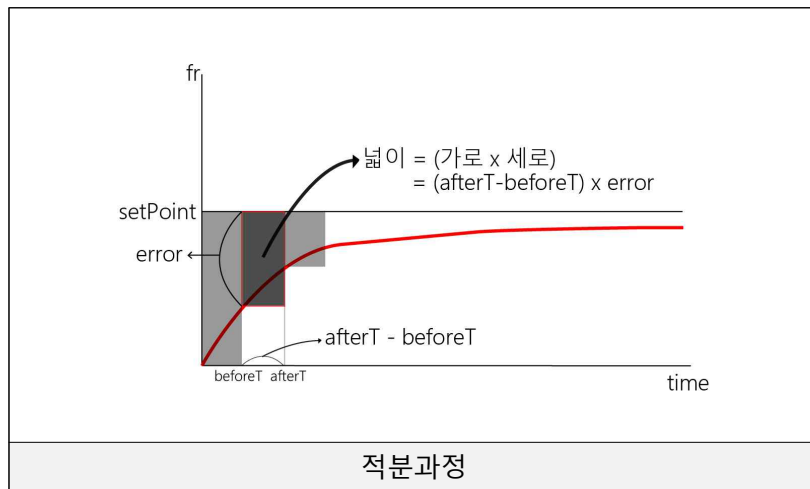
하지만 이렇게 P(비례) 제어만 사용할 경우, setPoint상태를 유지할 수는 없고 setPoint에 아주 가까운 값을 유지하기 때문에 미세한 차이가 발생한다. 이 미세한 차이를 잔류편차라고 한다.



### 2-3. PI(비례+적분) 제어

- 이 잔류편차를 해결하기 위해 지금까지의 편차를 누적하여 결과값에 더해 준다.



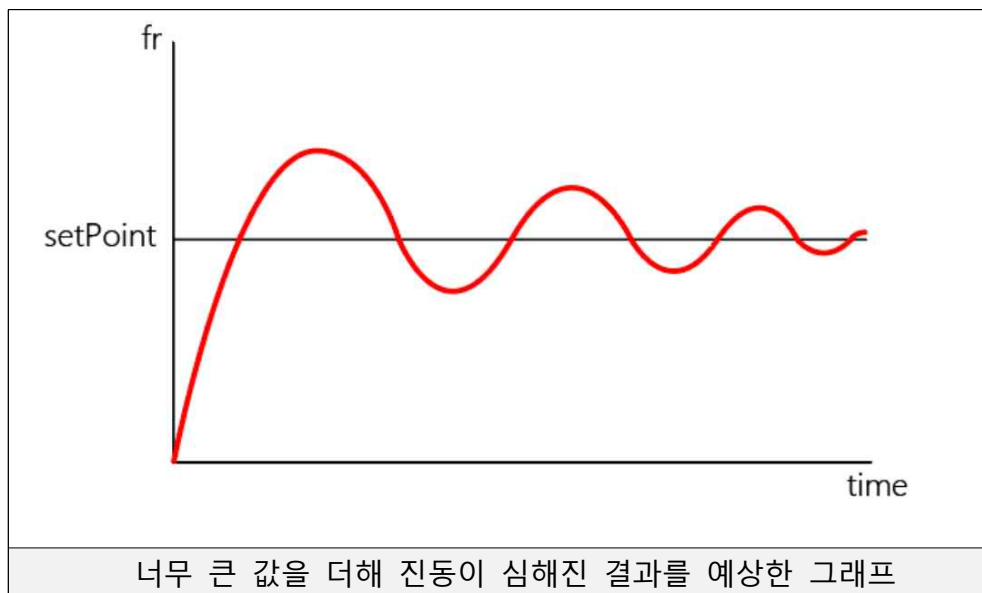


다음은 적분 제어를 통한 결과를 도출하는 식이다.

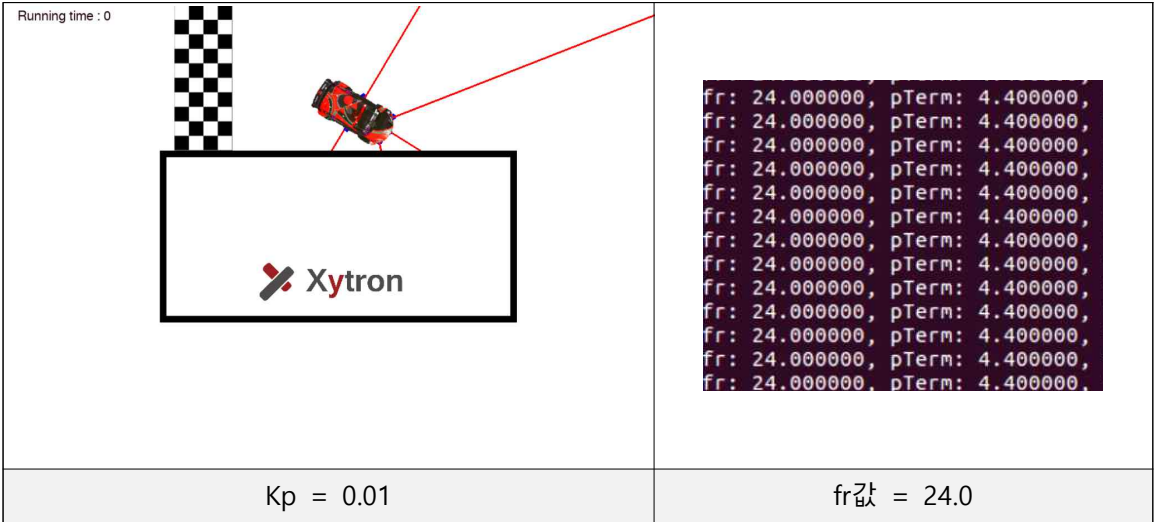
$$i(t) = K_i \int_0^t e(t) dt$$

색칠한 부분을 적분하려면 time축을 무수히 나누어 가로길이를 구해야한다.

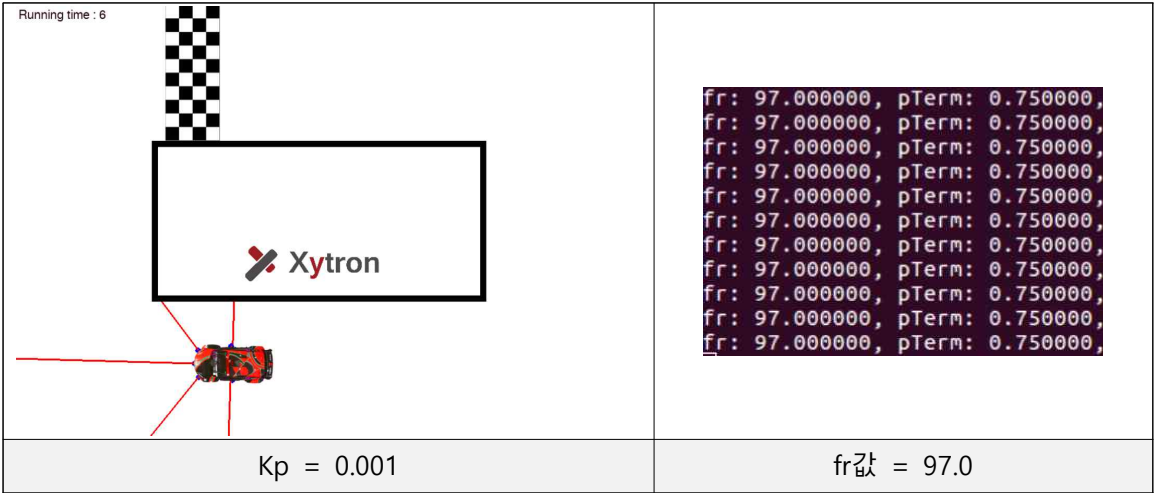
$\Delta\text{time} = \text{afterT} - \text{beforeT}$  과 같이 구한다. 이 때 적분값은  $\int_0^t e(t) dt$  로 한다. 그러나 이 값을 그대로 더하게 될 경우 잔류편차보다 더 큰 값을 더하게 되므로 다음과 같이 될 것이다.



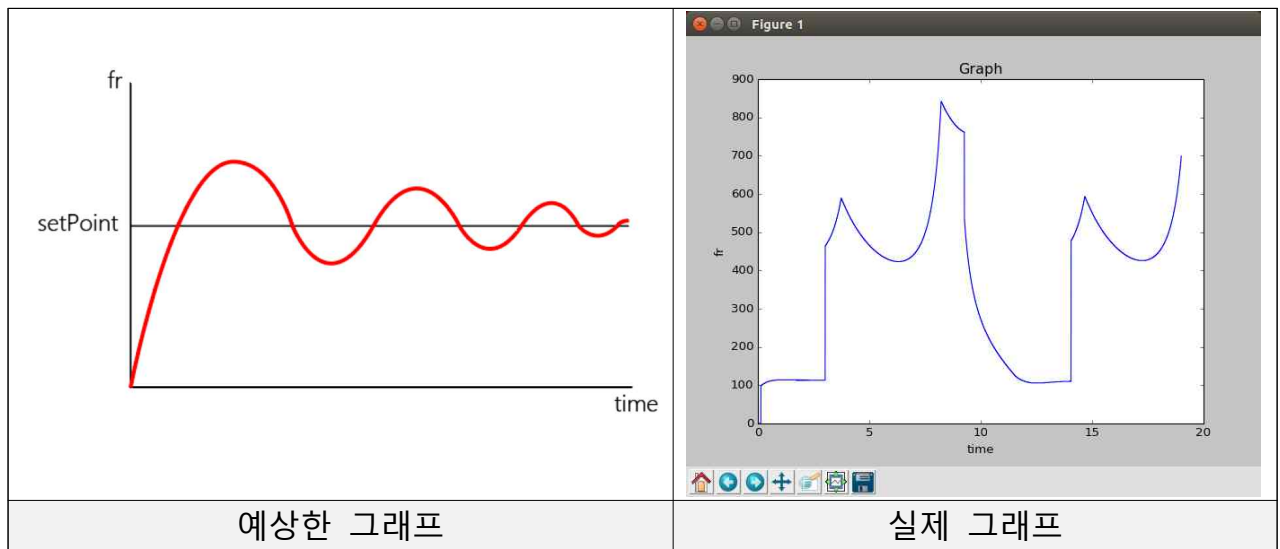
그러므로  $\int_0^t e(t)dt$  값에 곱할 Ki값을 하나 씩 대입해서 적절한 값을 찾는다.



Ki = 0.01일 때 시간이 지나자 누적되는 값이 너무 커서 충돌한다.



$K_i = 0.001$ 일 때 시간이 지나자 수평 상태를 유지하는 것을 볼 수 있다. 하지만 적분 제어를 추가한다면 잔류 편차가 제거돼서 setPoint가 유지되어야 하는데 더 작은 값이 유지되고 있다.



시뮬레이터 자동차가 주행할 때의 실제  $fr$ 값을 Matplotlib을 이용해 그래프로 나타내어 보았다.

예상한 그래프와는 전혀 다른 모양이었다. angle값이 확 줄어들지 않는 한계가 있어서 예상하던 그래프와 다르게 나온 것이다. 트랙의 급커브 구간을 4번 지나가  $fr$ 값이 setPoint보다 훨씬 높은 값이 4번 나타나는 것을 볼 수 있다. 이렇게 되면  $iTerm$ 값은 이 편차가 계속 누적되어 결과에 영향을 미치게 될 것이다.

1.  $k_i = 0.001$
2.  $afterT = time.time()$
3.  $dt = afterT - beforeT$
4.  $beforeT = afterT$
5.  $iTerm += k_i * error * dt$

시작부터 지금까지 발생한 오차를 적분을 한다. 이때  $e(t)$ 는  $t$ 초에서 발생한

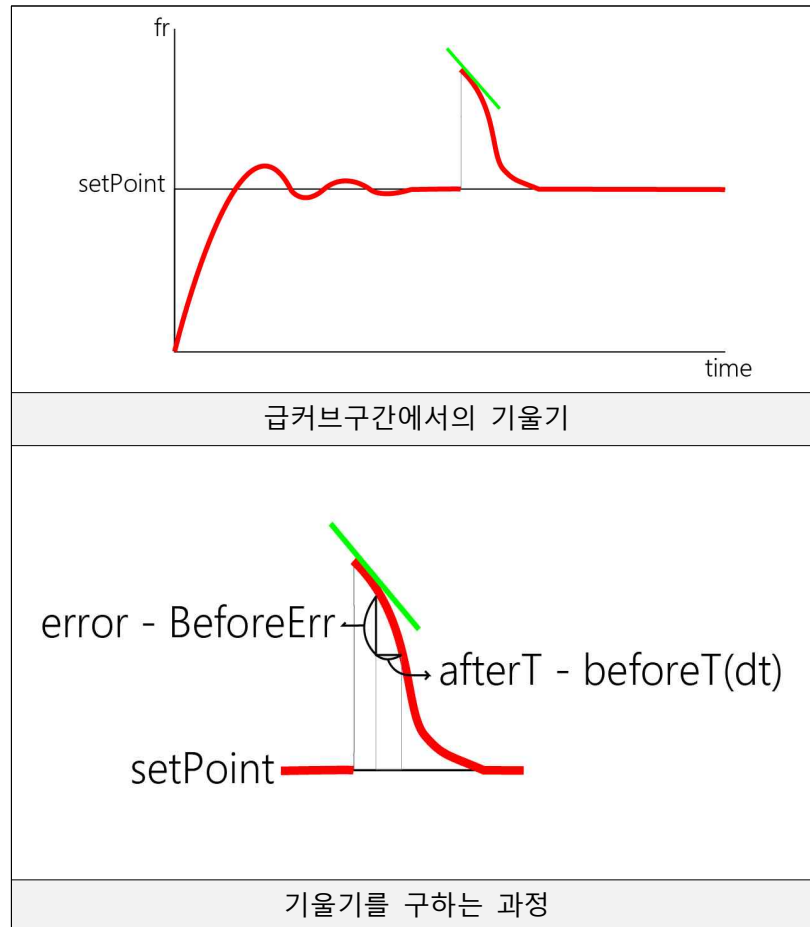


오차 값,  $dt$ 는 다음 에러가 발생하기까지의 시간차이다. 시뮬레이터를 처음 실행하면  $beforeT$ 는 0, 2행에서  $afterT$ 값은 갱신이 된다.  $afterT$ 에서  $beforeT$  값을 빼 입력주기인  $dt$ 를 구한다. 현재 까지의 오차를 누적하는 것이므로  $it$   $erm$ 에 오차를 더한다.  $beforeT$ 는 4행에서  $afterT$ 로 갱신한다.

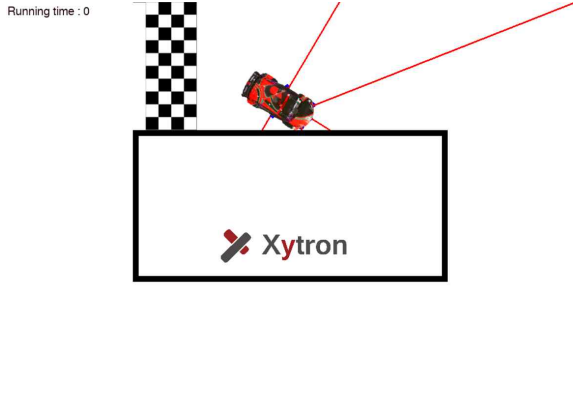
## 2-4. PID(비례+적분+미분) 제어

자동차가 급커브 구간에서 빠르게 setPoint로 돌아올 수 있도록 순간변화율이 높은 구간에서 기울기를 구해 기울기만큼 결과값에서 뺀다.

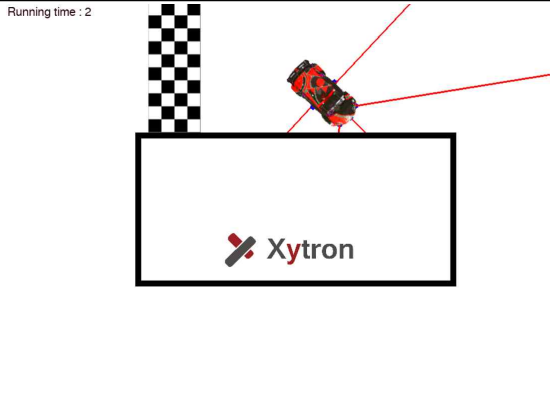
$$d(t) = K_d \frac{d}{dt} e(t)$$



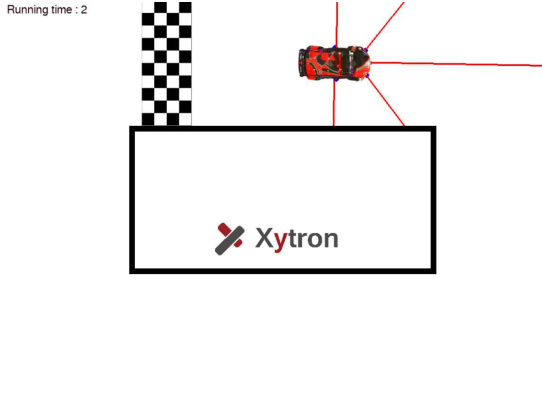
위 그래프와 같이 fr이 급격히 커지는 부분에서 빠르게 setPoint로 돌아오기 위해 다음과 같이 기울기를 구한다. 기울기 값을 확인하기 위해 Kd값은 1로 설정하고 테스트를 해본다. 다음은 테스트 결과다.

|   |  |
|---|--|
|  | <pre>fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt: fr: 9.000000, dTerm: 1315018.954418, dError / dt:</pre> |
| Kd = 1  | fr값 = 9.0  |

위 결과를 보면 dTerm값이 매우 커 벽과 충돌하는 것을 볼 수 있다. dTerm값이 매우 큰 이유는 dt가 매우 작기 때문에 fr값이 조금만 변해도 dError/dt값이 매우 크게 나오는 것이다. Kd값을 일정량 작게 설정해서 실험해본다.

|   |  |
|---|--|
|  | <pre>dTerm: -48.395815, dt: 0.000031, dError / dt: 483958.153846 dTerm: -38.362537, dt: 0.000039, dError / dt: 383625.365854 dTerm: -45.590261, dt: 0.000033, dError / dt: 455902.608696 dTerm: -51.569311, dt: 0.000029, dError / dt: 515693.114754 dTerm: -51.569311, dt: 0.000029, dError / dt: 515693.114754 dTerm: -50.331648, dt: 0.000030, dError / dt: 503316.480000 dTerm: -30.541049, dt: 0.000049, dError / dt: 305410.485437 dTerm: -42.799020, dt: 0.000035, dError / dt: 427990.204082 dTerm: -48.395815, dt: 0.000031, dError / dt: 483958.153846 dTerm: -22.712838, dt: 0.000066, dError / dt: 227128.375451 dTerm: -48.395815, dt: 0.000031, dError / dt: 483958.153846 dTerm: -37.673389, dt: 0.000040, dError / dt: 376733.892216 dTerm: -40.329846, dt: 0.000037, dError / dt: 403298.461538 dTerm: -44.306028, dt: 0.000034, dError / dt: 443060.281690 dTerm: -31.145822, dt: 0.000048, dError / dt: 311458.217822</pre> |
| Kd = 0.0001   | dTerm값 = -50 ~ -31   |

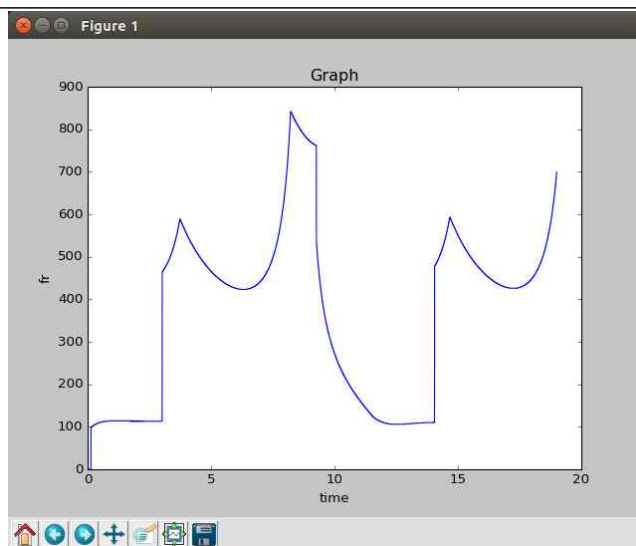
Kd값을 0.0001로 설정해서 dTerm값이 작아졌으나 여전히 벽과 충돌한다. 일정량 작게 설정해본다.

|   |  |
|---|--|
|  | <pre> dTerm: -0.002260, dt: 0.000049, dError / dt: 2260037.592233 dTerm: -0.001589, dt: 0.000070, dError / dt: 1588968.409556 dTerm: -0.002305, dt: 0.000048, dError / dt: 2304790.811881 dTerm: -0.002176, dt: 0.000051, dError / dt: 2175550.205607 dTerm: -0.001948, dt: 0.000057, dError / dt: 1947982.192469 dTerm: -0.001407, dt: 0.000079, dError / dt: 1406549.075529 dTerm: -0.001791, dt: 0.000062, dError / dt: 1790645.169231 dTerm: -0.001681, dt: 0.000066, dError / dt: 1680749.978339 dTerm: -0.002271, dt: 0.000049, dError / dt: 2271062.165854 dTerm: -0.001245, dt: 0.000089, dError / dt: 1244833.540107 dTerm: -0.001262, dt: 0.000088, dError / dt: 1261701.203252 dTerm: -0.001390, dt: 0.000080, dError / dt: 1389754.459701 dTerm: -0.001478, dt: 0.000075, dError / dt: 1477992.838095 dTerm: -0.001478, dt: 0.000075, dError / dt: 1477992.838095 dTerm: -0.001611, dt: 0.000069, dError / dt: 1610961.051903 </pre> |
| Kd = 0.000000001  | dTerm값 = -0.002305 ~ -0.001245   |

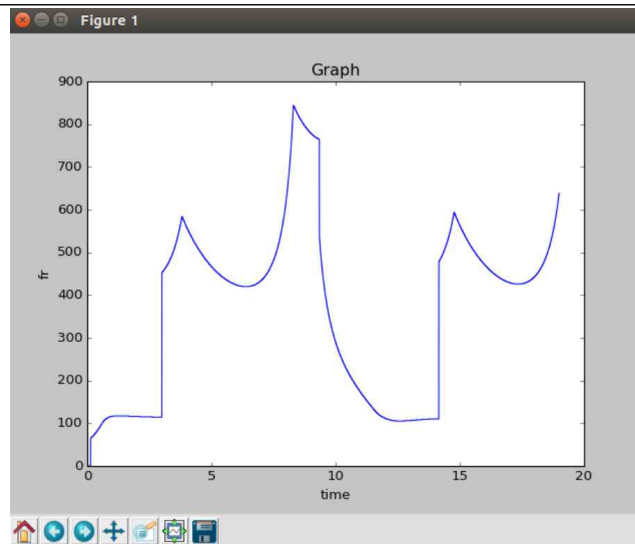
Kd값을 0.000000001로 설정하여 dTerm값이 매우 적게 나온 것을 알 수 있다. 그러나 이 값은 자동차 주행에 영향을 거의 미치지 않는 수준의 값이다. 다음은 소스코드와 P, PI, PID제어를 이용해 주행한 결과를 그래프로 나타낸 것이다.

1.  $kd = 0.000000001$
2.  $dError = setPoint - error$
3. **if**  $dt \neq 0$ :
4.  $dTerm = -kd * (dError / dt)$
5.  $pid = pTerm + iTerm + dTerm$
6.  $angle = -pid * (180.0 / \text{math.pi})$

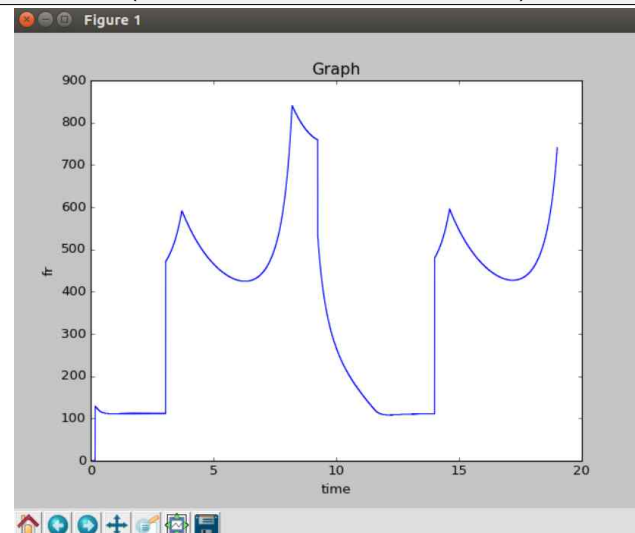
현재 에러값과 그 전의 에러값을 뺀 dError과 입력 주기인 dt를 이용해 기울기를 구할 수 있다. 하지만 입력주기 dt값이 0.00006 ~ 0.00008로 매우 작다. 이 때문에 dError가 크지 않아도 기울기 값은 매우 크게 나온다. Kd에 적절한 값을 설정하여 정상적인 기울기가 나오도록 한다. 시뮬레이터를 처음 실행하면 dt값이 갱신되지 않았기 때문에 dError을 0으로 나누어 버리는 경우가 발생한다. 이를 막기 위해 3행에서 dt값이 0이 아닐때만 dTerm을 계산하도록 한다. 기울기를 빼야하기 때문에 4행에서 kd에 -1을 곱하여 dTerm에 저장한다. 5행에서 지금까지 구한 비례, 적분, 미분 제어 결과값을 모두 더해pid값을 산출한다. 이 때 발생하는 pid값은 라디안 값이기 때문에  $(180 / \pi)$ 을 곱하여 angle에 저장한다. 여기서 pid값은 setPoint에서 현재 데이터를 뺀 값이기 때문에 setPoint를 초과하면 음수가 나온다. fr센서 길이를 줄이려면 angle값은 양수가 되어야 하므로 6행에서 최종 결과에 -1을 곱한 값을 저장한다.



(P제어를 이용해 주행한 그래프)



(PI제어를 이용해 주행한 그래프)



(PID제어를 이용해 주행한 그래프)

이 3개의 그래프를 보면 P제어만 사용했을 때와 I, D 제어를 추가했을 때, 모두 완주 했으나 주행은 거의 차이가 없다. 이 시뮬레이션은 최대 회전 각도가 제한되어 있어서 급격한 변화에 빠른 회전을 하지 못한다. 여기서 실제 차량 주행과 차이점을 발견할 수 있었다. 시뮬레이션은 입력값을 원하는 만큼 조절이 가능하고 목표값에 도달했을 때 바로 멈출 수 있다. 이 PID제어를 제대로 활용하려면 실제 차량을 가지고 테스트해봐야 할 것이다.