

# PID Control



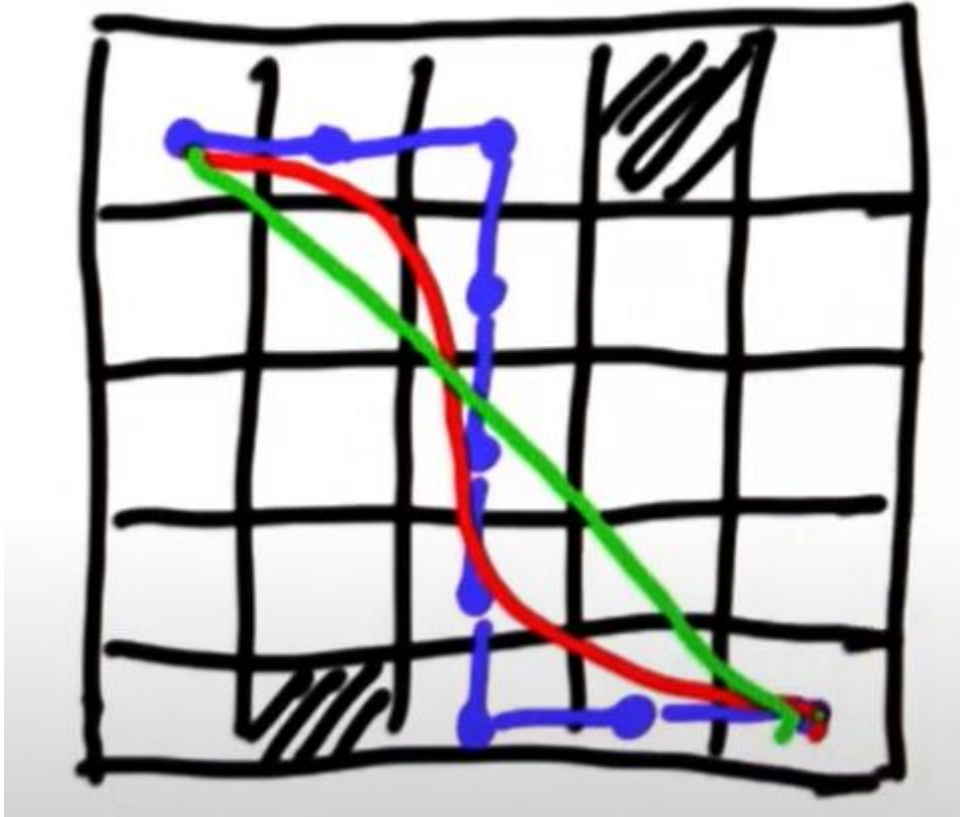
- <https://www.youtube.com/watch?v=fv6dLTEvI74>

# Robot Motion

Search 를 통해 경로를 찾는 방법을 확인 하였다면,  
이제 찾은 경로를 실제 모션 명령으로 변환을 해야 한다.

특히 부드러운 경로 생성을 위해 필요한 것을 찾아보자.

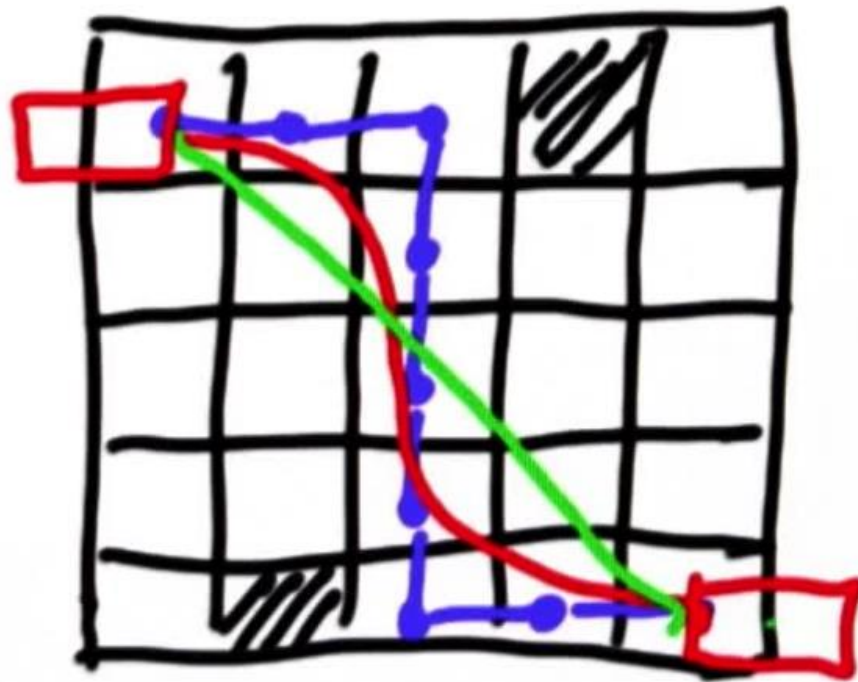
# Generate smooth paths



- 90도 기준으로 이동하는 것은 속도를 감소시키고, 불편한 이동이 됩니다.
- 기존 파란 라인에 비해 부드러운 경로입니다.
- 극단적인 연결 구조입니다.

\* 파란 라인이 빨간색이나, 초록 경로와 비슷하게 전환을 시킬수 있는가 가 관건!

어떤 이동 라인을 선호 하나요?



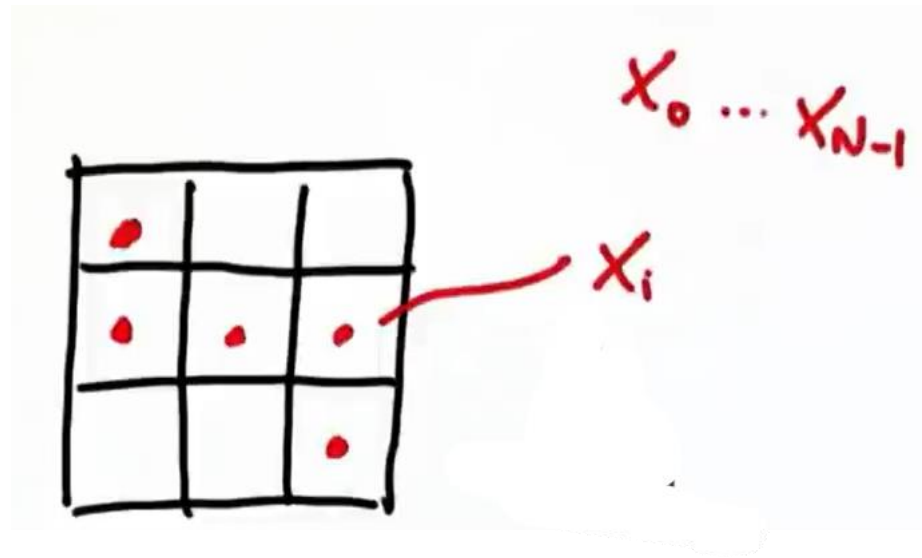
BLUE ○

RED ○

GREEN ○

# Smoothing Algorithm

- 서치를 이용 각 점들을 통해 경로를 찾을 수 있습니다.
- $X_0 \rightarrow X_{n-1}$





# Smoothing Algorithm

- 모든  $x_i$  와 동일한 변수  $y_i$  구성
  - 매끄럽지 않은 위치이다

→ ①  $y_i = x_i$

② OPTIMIZE

- 1번째 원점과 1 번째 평활점의 오차  
최소화

→  $(x_i - y_i)^2 \rightarrow \min$

- 연속된 평활점 사이의 거리를 모두  
정사각형으로 최소화

→  $(y_i - y_{i+1})^2 \rightarrow \min$

# Smoothing Algorithm

- 다음 공식을 통해 가능한 것은?

$$(x_i - y_i)^2 \rightarrow \min$$

- o ORIGINAL PATH
- o SMOOTH PATH
- o NO PATH



# Smoothing Algorithm

- 다음중 맞는것은?

$$(x_i - y_i)^2 \rightarrow \min$$

- ✓ ORIGINAL PATH
- SMOOTH PATH
- NO PATH

더 이상 최소화 시키는 것은 작아지지 않기에 원래 경로를 얻을수 있다

# Smoothing Algorithm

- 다음중 맞는것은?

$$(y_i - y_{i+1})^2 \rightarrow \min$$

- o ORIGINAL PATH
- o SMOOTH PATH
- o NO PATH

# Smoothing Algorithm

- 다음중 맞는것은?

$$(y_i - y_{i+1})^2 \rightarrow \min$$

- ORIGINAL PATH
- SMOOTH PATH
- ✓ ○ NO PATH

→ 길을 찾을수 없다.

→ 위 기준은 y 가 가능한 유사해야 한다. 최소화 되면 모든 y 가 동일하다 단일점만 얻고, 경로가 없다.

# Smoothing Algorithm

- 알파를 최적화 시켜 두가지를 통해 얻을수 있는것은?

② OPTIMIZE

$$(x_i - y_i)^2 \rightarrow \min$$

$$+ \alpha (y_i - y_{i+1})^2 \rightarrow \min$$

- ORIGINAL PATH
- SMOOTH PATH
- NO PATH

# Smoothing Algorithm

- 알파를 최적화 시켜 두가지를 통해 얻을수 있는것은?

② OPTIMIZE

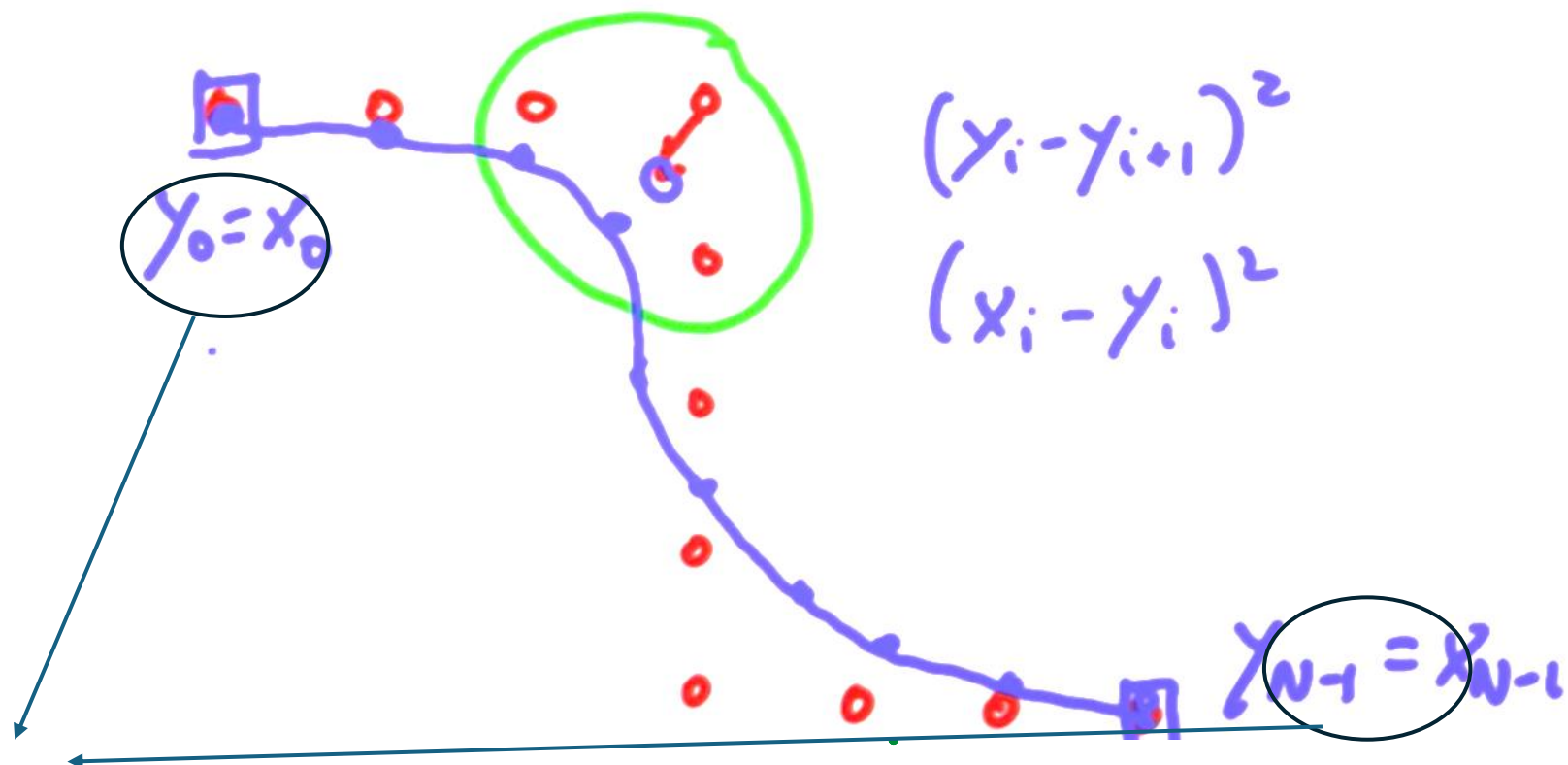
$$(x_i - y_i)^2 \rightarrow \min$$

$$+ \alpha (y_i - y_{i+1})^2 \rightarrow \min$$

○ ORIGINAL PATH

✓ ○ SMOOTH PATH

○ NO PATH



- 원래점(시작점, 끝점) 이 고정이라면, 최적화에선 해당점을 제외할 수 있다.
- 위 값들이 항상 같다고 고정하지 않는다.
- 최적화는 중간 지점에만 적용



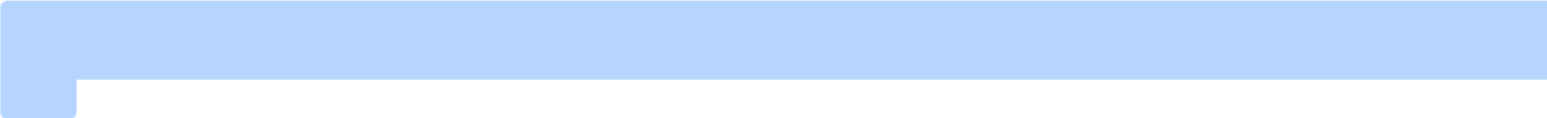
# Path Smoothing

```
# thank you to EnTerr for posting this on our discussion forum
def printpaths(path,newpath):
    for old,new in zip(path,newpath):
        print '['+ ', '.join('%3f'%x for x in old) + \
            ']' -> '['+ ', '.join('%3f'%x for x in new) +']'

# Don't modify path inside your function.
path = [[0, 0],
        [0, 1],
        [0, 2],
        [1, 2],
        [2, 2],
        [3, 2],
        [4, 2],
        [4, 3],
        [4, 4]]

def smooth(path, weight_data = 0.5, weight_smooth = 0.1, tolerance = 0.000001):

    # Make a deep copy of path into newpath
    newpath = deepcopy(path)

    return newpath # Leave this line for the grader!
```

```
# 경로를 입력으로 사용하는 함수 Smooth를 정의합니다.  
# (weight_data, Weight_smooth에 대한 선택적 매개변수 포함,  
# 및 허용 오차) 및 부드러운 경로를 반환합니다. 첫 번째와  
# 마지막 포인트는 변경되지 않고 그대로 유지되어야 합니다.  
#  
# 스무딩은 반복적인 업데이트를 통해 구현되어야 합니다.  
# 원하는 정확도 수준까지 newpath의 각 항목  
#에 도달했습니다. 업데이트는 다음 지침에 따라 수행되어야 합니다.  
# 경사 하강 방정식  
# -----
```

```
from copy import deepcopy
```

```
def printpaths(path,newpath):  
    for old,new in zip(path,newpath):  
        print( '['+ ' ', '.join('%.3f'%x for x in old) + ']' -> '['+ ' ', '.join('%.3f'%x for x in new) +']')
```

```
path = [[0, 0],  
        [0, 1],  
        [0, 2],  
        [1, 2],  
        [2, 2],  
        [3, 2],  
        [4, 2],  
        [4, 3],  
        [4, 4]]
```

```

def smooth(path, weight_data = 0.5, weight_smooth = 0.1, tolerance = 0.000001):

    # Make a deep copy of path into newpath
    newpath = deepcopy(path)

    change = tolerance

    while change >= tolerance:
        change = 0
        for i in range(1, len(path) - 1):
            for j in range(len(path[0])):
                d1 = weight_data*(path[i][j] - newpath[i][j])
                d2 = weight_smooth*(newpath[i-1][j] + newpath[i+1][j] - 2*newpath[i][j])
                change += abs(d1 + d2)
                newpath[i][j] += d1 + d2

    return newpath

printpaths(path,smooth(path))

```

```

[0.000, 2.000] -> [0.149, 1.851]
[1.000, 2.000] -> [1.021, 1.979]
[2.000, 2.000] -> [2.000, 2.000]
[3.000, 2.000] -> [2.979, 2.021]
[4.000, 2.000] -> [3.851, 2.149]
[4.000, 3.000] -> [3.979, 3.021]
[4.000, 4.000] -> [4.000, 4.000]

```

# Zero Data Weight

$$\alpha = 0$$
$$\beta = 0.1$$

- original path
- straight line
- a single point

알파(weight data) = 0  
베타(평활가중치) = 0.1  
진행시 영향을 주는것은?

두개를 완료될때까지 실행

이때 원래 경로, 최종위치 연결 직선

모든점의 붕괴

# Zero Data Weight

$$\alpha = 0$$
$$\beta = 0.1$$

- ☐ original path
- ☒ straight line
- ☐ a single point

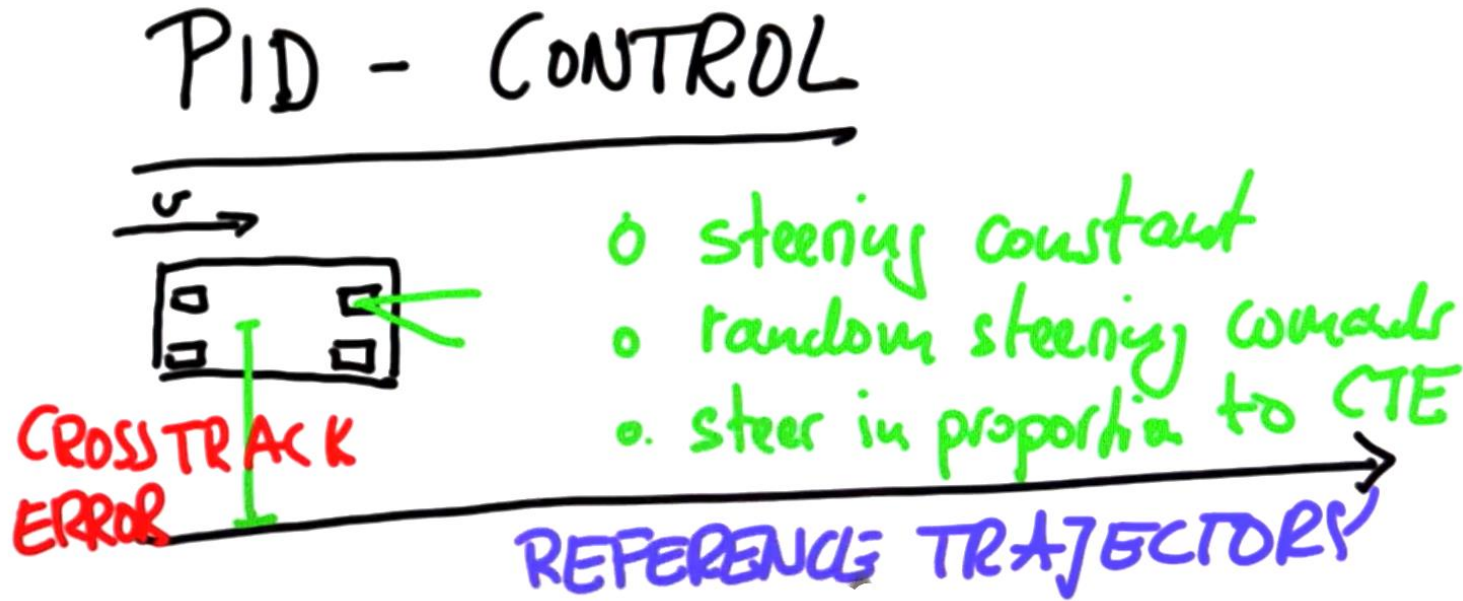
시작점, 끝점이 변화가 없다.

.

- <https://www.youtube.com/watch?v=wkfEZmsQqiA&list=PLn8PRpmsu08pQBgjxYFXSsODEF3Jqmm-y>



# PID



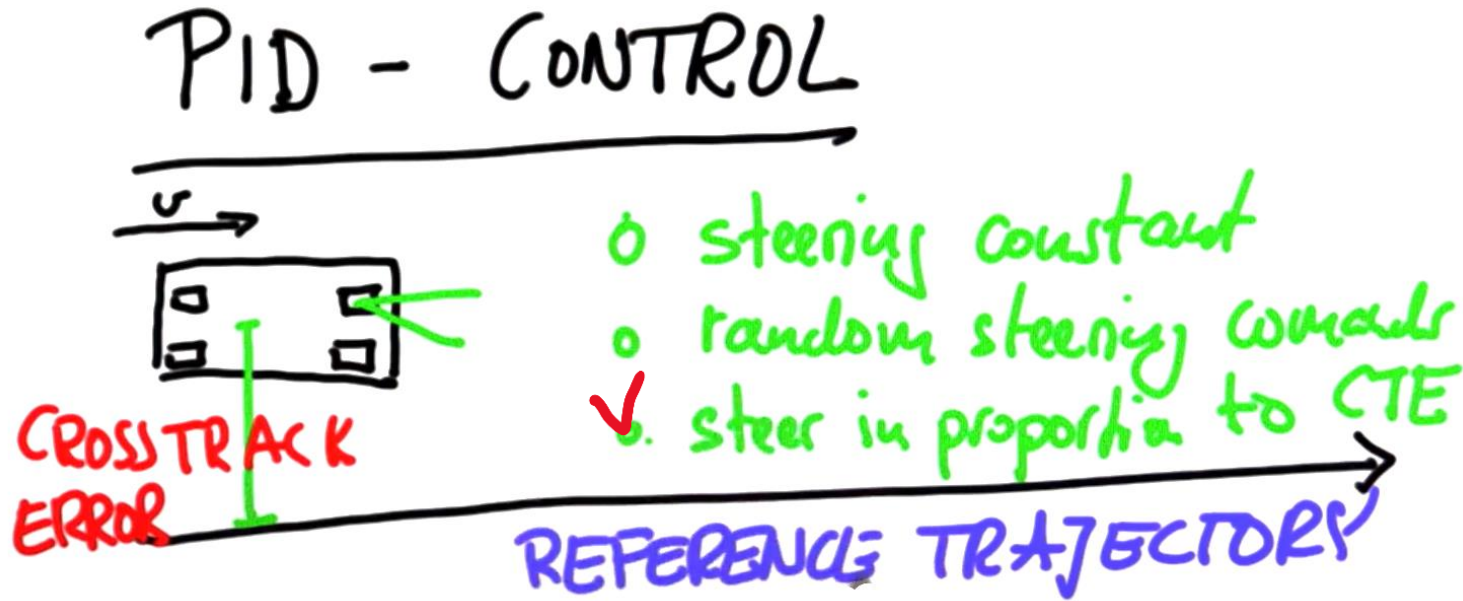
운전대의 일정한 유지

랜덤 스티어링 명령

크로스트랙 오차에 의한 조향 각도 설정

- 먼저 구해본 평활화 기술로 차량을 운행을 합니다.
- 어떤 방식으로 차량의 조향 각도를 설정할까요?

# PID



운전대의 일정한 유지

랜덤 스티어링 명령

크로스트랙 오차에 의한 조향 각도 설정

- CTE Error 기준으로 방향을 잡아야 합니다.
- 오차(간격) 이 클수록 더 많은 방향을 바꿀수 있다.
- 궤도에 들어갈수록 조향 속도는 느려지게 된다.

# PID

## P - CONTROLLER

↖ P = "PROPORTIONAL"



$$\alpha = \gamma \cdot \text{CTE}$$

조향각도는  $\text{cte} \cdot \text{타우}(\text{일부요인})$  비례

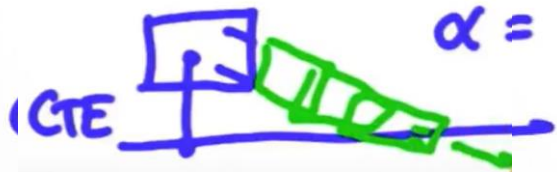
- never quite reaches reference
- overshoots
- either can happen.

3개중 어떤 차에 어떤 영향이 있을까?

# P-Controller

## P - CONTROLLER

↖ P = "PROPORTIONAL"



$$\alpha = \gamma \cdot \text{CTE}$$

조향각도는 cte\*타우(일부요인) 비례

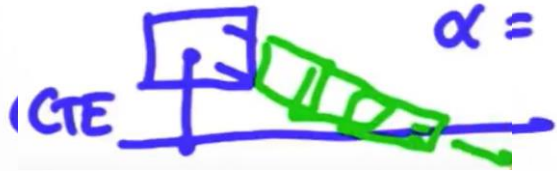
- never quite reaches reference
- ✓ overshoots
- either can happen.

타우 상수가 아무리 작아도, 궤도를 향해 바퀴를 많이 돌게 한다.  
달고 정지..

# P-Controller

## P - CONTROLLER

↖ P = "PROPORTIONAL"



$$\alpha = \gamma \cdot \text{CTE}$$

조향각도는 cte\*타우(일부요인) 비례

- never quite reaches reference
- ✓ overshoots
- rither can happen.

타우 상수가 아무리 작아도, 궤도를 향해 바퀴를 많이 돌게 한다.  
달고 정지..

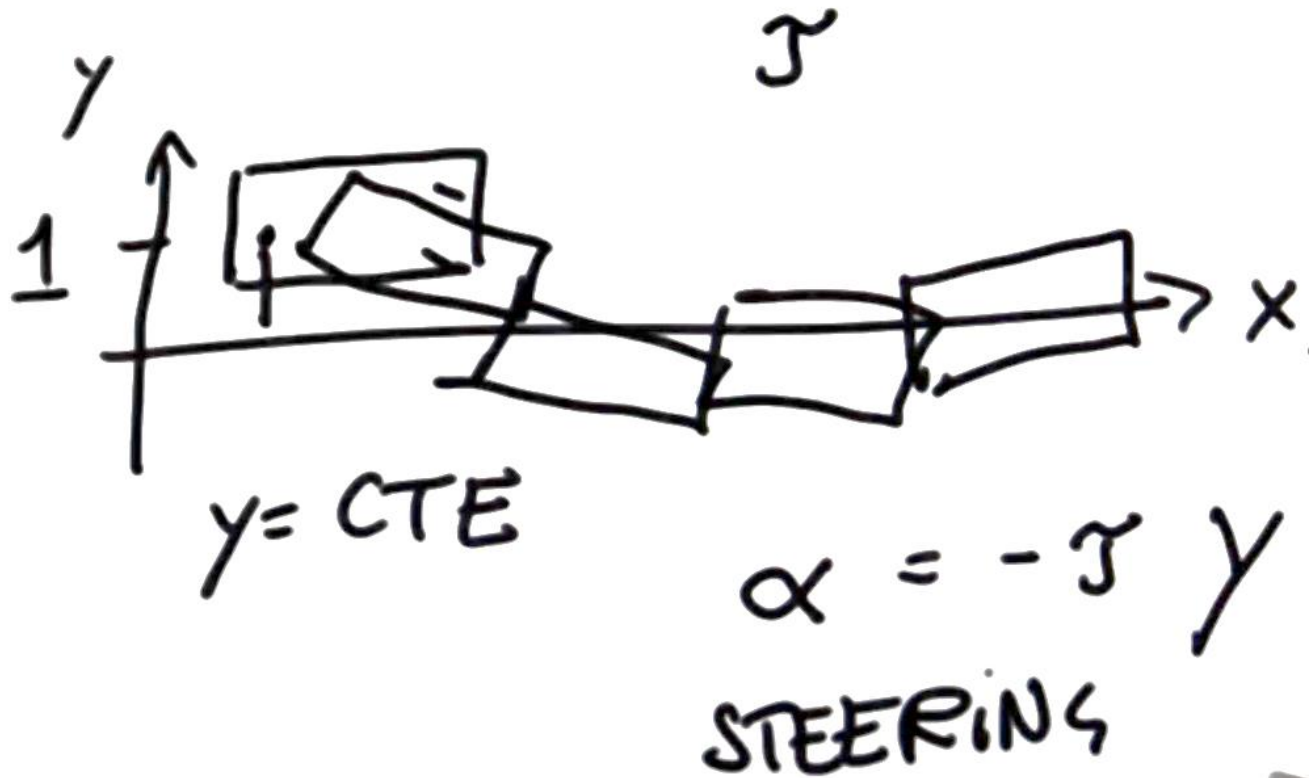
# P-CONTROLLER



MARGINALLY STABLE

- P-Controller 는 항상 약간의 오버슈트(매우작게) 이거나 관촬을수 있다.
- 절대 수렴은 안된다.(그 사이 어딘가 계속..)
- 이것을 약간 안정적 또는 안정적이라 표현할수 있다.





- 로봇의 초기 위치 (0,1,0) 속도 = 1 기준일때 , 100 단계 시뮬레이션 진행
- $Y = cte$  동일
- $Y$  값에 반비례하여 비례제어기의 반응 강도를 설정하는 모수 타우 활용
- 로봇이  $x$  축방향 회전 및 이동, 오버슈트 , 후 다시 돌아서는지에 대해 테스트해보자 (800step)
- Cte 오차에 비례 조향각도 각도 알파를 사용하자

```
# 100번의 반복을 실행하여 P 컨트롤러를 구현합니다.  
# 로봇 모션. 원하는 궤적  
# 로봇은 x축입니다. 조향 각도를 설정해야 합니다.  
# 매개변수 tau를 사용하여 다음을 수행합니다.  
#  
# steering = -tau * crosstrack_error
```

```
from math import *  
import random
```

```
# -----  
#  
# this is the robot class  
#
```

```
class robot:
```

```
    # -----  
    # init:  
    # creates robot and initializes location/orientation to 0, 0, 0  
    #
```

```
    def __init__(self, length = 20.0):  
        self.x = 0.0  
        self.y = 0.0  
        self.orientation = 0.0  
        self.length = length
```

```
self.steering_noise = 0.0
self.distance_noise = 0.0
self.steering_drift = 0.0

# -----
# set:
#     sets a robot coordinate
#

def set(self, new_x, new_y, new_orientation):

    self.x = float(new_x)
    self.y = float(new_y)
    self.orientation = float(new_orientation) % (2.0 * pi)

# -----
# set_noise:
#     sets the noise parameters
#

def set_noise(self, new_s_noise, new_d_noise):
    # makes it possible to change the noise parameters
    # this is often useful in particle filters
    self.steering_noise = float(new_s_noise)
    self.distance_noise = float(new_d_noise)

# -----
# set_steering_drift:
#     sets the systematical steering drift parameter
#
```

```
def set_steering_drift(self, drift):
    self.steering_drift = drift

# -----
# move:
#   steering = front wheel steering angle, limited by max_steering_angle
#   distance = total distance driven, must be non-negative

def move(self, steering, distance,
        tolerance = 0.001, max_steering_angle = pi / 4.0):

    if steering > max_steering_angle:
        steering = max_steering_angle
    if steering < -max_steering_angle:
        steering = -max_steering_angle
    if distance < 0.0:
        distance = 0.0

    # make a new copy
    res = robot()
    res.length      = self.length
    res.steering_noise = self.steering_noise
    res.distance_noise = self.distance_noise
    res.steering_drift = self.steering_drift
```

```
# apply noise
steering2 = random.gauss(steering, self.steering_noise)
distance2 = random.gauss(distance, self.distance_noise)

# apply steering drift
steering2 += self.steering_drift

# Execute motion
turn = tan(steering2) * distance2 / res.length

if abs(turn) < tolerance:

    # approximate by straight line motion

    res.x = self.x + (distance2 * cos(self.orientation))
    res.y = self.y + (distance2 * sin(self.orientation))
    res.orientation = (self.orientation + turn) % (2.0 * pi)

else:

    # approximate bicycle model for motion

    radius = distance2 / turn
    cx = self.x - (sin(self.orientation) * radius)
    cy = self.y + (cos(self.orientation) * radius)
    res.orientation = (self.orientation + turn) % (2.0 * pi)
    res.x = cx + (sin(res.orientation) * radius)
    res.y = cy - (cos(res.orientation) * radius)

return res
```

```
def __repr__(self):
    return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

# -----
#
# run - does a single control run

def run(param):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    for _ in range(N):
        # steering angle = - tau * crosstrack_error
        steering_angle = -param*myrobot.y
        myrobot = myrobot.move(steering_angle, speed)
        print(myrobot, steering_angle)

run(0.1) # call function with parameter tau of 0.1 and print results
```



```
[x=1.00000 y=0.99749 orient=6.27817] -0.1
[x=1.99997 y=0.98997 orient=6.27316] -0.0997491638458655
[x=2.99989 y=0.97747 orient=6.26820] -0.09899729506124687
[x=3.99973 y=0.96003 orient=6.26330] -0.0977469440459231
[x=4.99948 y=0.93774 orient=6.25848] -0.0960031957433074
[x=5.99912 y=0.91068 orient=6.25378] -0.09377364753807171
[x=6.99861 y=0.87900 orient=6.24921] -0.09106837322063087
[x=7.99796 y=0.84283 orient=6.24481] -0.08789987335156867
```

$\tau = 0.1 \rightarrow 0.3$

- OSCILLATES FASTER
- OSCILLATES SLOWER
- NONE OF ABOVE

- 타우 값을 0.1  $\rightarrow$  0.3 으로 변경 한다면 어떤 부분에 영향이 있을까?

$T = 0.1 \rightarrow 0.3$

✓ OSCILLATES FASTER

• OSCILLATES SLOWER

• NONE OF ABOVE

- 진동속도가 빨라진다

```
[x=0.99996 y=0.99227 orient=6.26772] -0.3
[x=1.99968 y=0.96913 orient=6.25238] -0.2976800243776367
[x=2.99894 y=0.93085 orient=6.23742] -0.2907396972040246
[x=3.99753 y=0.87794 orient=6.22308] -0.2792564861664303
[x=4.99529 y=0.81115 orient=6.20960] -0.2633831536658889
[x=5.99210 y=0.73144 orient=6.19718] -0.24334437577115153
[x=6.98791 y=0.63999 orient=6.18603] -0.21943172529507535
[x=7.98270 y=0.53816 orient=6.17631] -0.19199739833890134
```

# PD-Controller



- 오버슈트는 필연적으로 동반해야 하는가?
- 자율 주행의 안정성은?

# PD-Controller

$$\alpha = -\gamma_p \text{CTE} - \gamma \frac{d}{dt} \text{CTE}$$

조향 알파는 게인 매개변수에 의한 CTE 오류와 관련이 있을뿐 아니라 CTE의 시간적 파생과도 연결된다. 즉 차가 CTE를 줄일 수 있을 만큼 충분히 회전 되었을때, x축에 대해서만 슈팅한것이 아니라, 오차를 줄일수 있다는 뜻  
미분은 시간 경과에따라 오차를 줄이고, 다시 위로 조향 가능케 함, 차등과 비례의 적절한 설정을 통해 목표 궤적 안착에 도움



$$\gamma \frac{d}{dt} \text{CTE} = \frac{\text{CTE}_t - \text{CTE}_{t-1}}{\Delta t}$$

Cte의 시간 t와 t-1 사이의 시간 범위로 나눈것과 동일하다. ( $\Delta T = 1$ )

```
# 100번의 반복을 실행하여 PD 컨트롤러를 구현합니다.  
# 로봇 모션. 조향 각도를 설정해야 합니다.  
# 매개변수 tau를 사용하여 다음을 수행  
#  
# steering = -tau_p * CTE - tau_d * diff_CTE  
# 차동 교차 추적 오류(diff_CTE)가 있다면 CTE(t) - CTE(t-1)로 제공
```

```
from math import *  
import random
```

```
# -----  
#  
# this is the robot class  
#
```

```
class robot:
```

```
    # -----  
    # init:  
    # creates robot and initializes location/orientation to 0, 0, 0  
    #
```

```
    def __init__(self, length = 20.0):  
        self.x = 0.0  
        self.y = 0.0  
        self.orientation = 0.0
```

```
self.length = length
self.steering_noise = 0.0
self.distance_noise = 0.0
self.steering_drift = 0.0
```

```
# -----
```

```
# set:
```

```
# sets a robot coordinate
```

```
#
```

```
def set(self, new_x, new_y, new_orientation):
```

```
    self.x = float(new_x)
```

```
    self.y = float(new_y)
```

```
    self.orientation = float(new_orientation) % (2.0 * pi)
```

```
# -----
```

```
# set_noise:
```

```
# sets the noise parameters
```

```
#
```

```
def set_noise(self, new_s_noise, new_d_noise):
```

```
    # makes it possible to change the noise parameters
```

```
    # this is often useful in particle filters
```

```
    self.steering_noise = float(new_s_noise)
```

```
    self.distance_noise = float(new_d_noise)
```

```
# -----
# set_steering_drift:
#   sets the systematical steering drift parameter
#

def set_steering_drift(self, drift):
    self.steering_drift = drift

# -----
# move:
#   steering = front wheel steering angle, limited by max_steering_angle
#   distance = total distance driven, must be non-negative

def move(self, steering, distance,
        tolerance = 0.001, max_steering_angle = pi / 4.0):

    if steering > max_steering_angle:
        steering = max_steering_angle
    if steering < -max_steering_angle:
        steering = -max_steering_angle
    if distance < 0.0:
        distance = 0.0

    # make a new copy
    res = robot()
    res.length      = self.length
    res.steering_noise = self.steering_noise
    res.distance_noise = self.distance_noise
    res.steering_drift = self.steering_drift
```



```
# apply noise
steering2 = random.gauss(steering, self.steering_noise)
distance2 = random.gauss(distance, self.distance_noise)

# apply steering drift
steering2 += self.steering_drift

# Execute motion
turn = tan(steering2) * distance2 / res.length

if abs(turn) < tolerance:

    # approximate by straight line motion

    res.x = self.x + (distance2 * cos(self.orientation))
    res.y = self.y + (distance2 * sin(self.orientation))
    res.orientation = (self.orientation + turn) % (2.0 * pi)

else:

    # approximate bicycle model for motion

    radius = distance2 / turn
    cx = self.x - (sin(self.orientation) * radius)
    cy = self.y + (cos(self.orientation) * radius)
    res.orientation = (self.orientation + turn) % (2.0 * pi)
    res.x = cx + (sin(res.orientation) * radius)
    res.y = cy - (cos(res.orientation) * radius)

return res
```

```
def __repr__(self):
    return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)

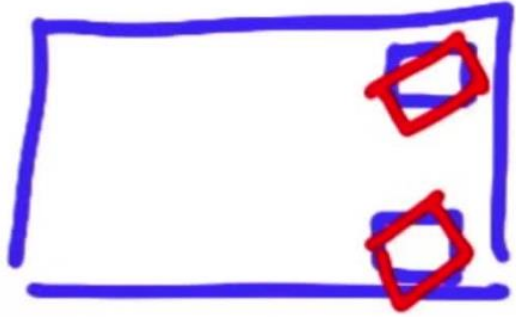
# -----
#
# run - does a single control run.

def run(param1, param2):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    #myrobot.set_steering_drift(10.0/180*pi)
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
    N = 100
    previous_CTE = myrobot.y
    for _ in range(N):
        current_CTE = myrobot.y
        diff_CTE = current_CTE - previous_CTE
        steer = -param1*myrobot.y - param2*diff_CTE
        myrobot = myrobot.move(steer, speed)
        previous_CTE = current_CTE
        print(myrobot, steer)

# Call your function with parameters of 0.2 and 3.0 and print results
run(0.2, 3.0)
```

# SYSTEMATIC BIAS

WHAT HAPPENS?

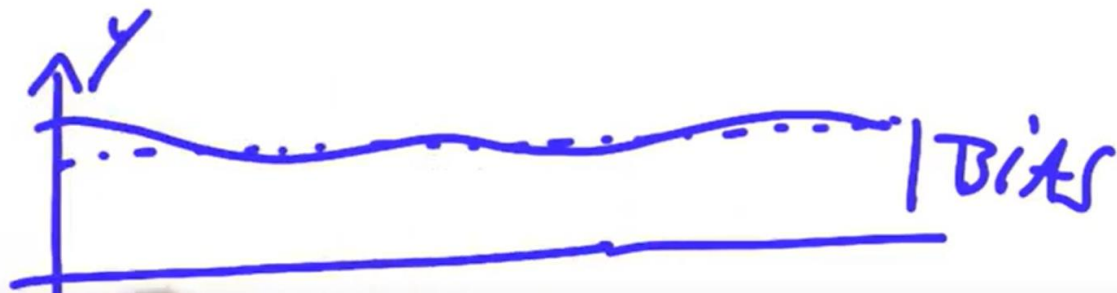


- JUST AS BEFORE
- CAUSES BIG CTE

차량 정비 실수로 바퀴를 비스듬히 정렬 되었다면? (약 10도?)

```
def run(param1, param2):
    myrobot = robot()
    myrobot.set(0.0, 1.0, 0.0)
    myrobot.set_steering_drift(10.0/180*pi)
    speed = 1.0 # motion distance is equal to speed (
    N = 100
```

```
# Call your function
run(0.2, 0.0)
```



x=38.99847	y=0.77061	orient=0.00974]	-0.15228518883595826
x=39.99842	y=0.78086	orient=0.01076]	-0.15412143636340261
x=40.99836	y=0.79161	orient=0.01168]	-0.1561710024784361
x=41.99829	y=0.80329	orient=0.01249]	-0.1583226346711613
x=42.99821	y=0.81578	orient=0.01318]	-0.1606578951700676
x=43.99813	y=0.82896	orient=0.01375]	-0.16315526093801197

Y 방향으로 증가된 편향이 발생, 진동을 유발

# PID Implementation

$$\alpha = \underbrace{-T_p CTE}_P - \underbrace{T_D \frac{d}{dt} CTE}_D - \underbrace{T_I \sum CTE}_I$$

Proportional      differentials      Integral

```
# 100번의 반복을 실행하여 PID 컨트롤러를 구현합니다.  
# 로봇 모션. 조향 각도를 설정해야 합니다.  
# 매개변수 tau를 사용하여 다음을 수행합니다.  
#  
# 스티어링 = -tau_p * CTE - tau_d * diff_CTE - tau_i * int_CTE  
#  
# 통합 교차 추적 오류(int_CTE)는 다음과 같습니다.  
# 이전의 모든 크로스 트랙 오류의 합계입니다.  
# 이 용어는 스티어링 드리프트를 상쇄하는 데 사용됩니다.
```

```
from math import *  
import random
```

```
# -----  
#  
# this is the robot class  
#
```

```
class robot:
```

```
    # -----  
    # init:  
    # creates robot and initializes location/orientation to 0, 0, 0  
    #
```

```
    def __init__(self, length = 20.0):  
        self.x = 0.0  
        self.y = 0.0  
        self.orientation = 0.0
```

```
self.length = length
    self.steering_noise = 0.0
    self.distance_noise = 0.0
    self.steering_drift = 0.0

# -----
# set:
#     sets a robot coordinate
#

def set(self, new_x, new_y, new_orientation):

    self.x = float(new_x)
    self.y = float(new_y)
    self.orientation = float(new_orientation) % (2.0 * pi)

# -----
# set_noise:
#     sets the noise parameters
#

def set_noise(self, new_s_noise, new_d_noise):
    # makes it possible to change the noise parameters
    # this is often useful in particle filters
    self.steering_noise = float(new_s_noise)
    self.distance_noise = float(new_d_noise)
```

```
# -----
# set_steering_drift:
#     sets the systematical steering drift parameter
#

def set_steering_drift(self, drift):
    self.steering_drift = drift

# -----
# move:
#     steering = front wheel steering angle, limited by max_steering_angle
#     distance = total distance driven, must be non-negative

def move(self, steering, distance,
        tolerance = 0.001, max_steering_angle = pi / 4.0):

    if steering > max_steering_angle:
        steering = max_steering_angle
    if steering < -max_steering_angle:
        steering = -max_steering_angle
    if distance < 0.0:
        distance = 0.0

    # make a new copy
    res = robot()
    res.length = self.length
    res.steering_noise = self.steering_noise
    res.distance_noise = self.distance_noise
    res.steering_drift = self.steering_drift
```



```
# apply noise
steering2 = random.gauss(steering, self.steering_noise)
distance2 = random.gauss(distance, self.distance_noise)

# apply steering drift
steering2 += self.steering_drift

# Execute motion
turn = tan(steering2) * distance2 / res.length

if abs(turn) < tolerance:

    # approximate by straight line motion

    res.x = self.x + (distance2 * cos(self.orientation))
    res.y = self.y + (distance2 * sin(self.orientation))
    res.orientation = (self.orientation + turn) % (2.0 * pi)

else:

    # approximate bicycle model for motion

    radius = distance2 / turn
    cx = self.x - (sin(self.orientation) * radius)
    cy = self.y + (cos(self.orientation) * radius)
    res.orientation = (self.orientation + turn) % (2.0 * pi)
    res.x = cx + (sin(res.orientation) * radius)
    res.y = cy - (cos(res.orientation) * radius)

return res
```

```
def __repr__(self):  
    return '[x=%.5f y=%.5f orient=%.5f]' % (self.x, self.y, self.orientation)
```

```
# -----
```

```
#
```

```
# run - does a single control run.
```

```
def run(param1, param2, param3):
```

```
    myrobot = robot()
```

```
    myrobot.set(0.0, 1.0, 0.0)
```

```
    speed = 1.0 # motion distance is equal to speed (we assume time = 1)
```

```
    N = 100
```

```
    myrobot.set_steering_drift(10.0 / 180.0 * pi) # 10 degree bias, this will be added in by the move function, you do not  
    need to add it below!
```

```
    previous_CTE = myrobot.y
```

```
    int_CTE = 0.0
```

```
    for i in range(N):
```

```
        current_CTE = myrobot.y
```

```
        diff_CTE = current_CTE - previous_CTE
```

```
        int_CTE += current_CTE
```

```
        steer = -param1*current_CTE - param2*diff_CTE - param3*int_CTE
```

```
        myrobot = myrobot.move(steer, speed)
```

```
        previous_CTE = current_CTE
```

```
    print (myrobot, steer)
```

```
# Call your function with parameters of (0.2, 3.0, and 0.004)
```

```
run(0.2, 3.0, 0.004)
```