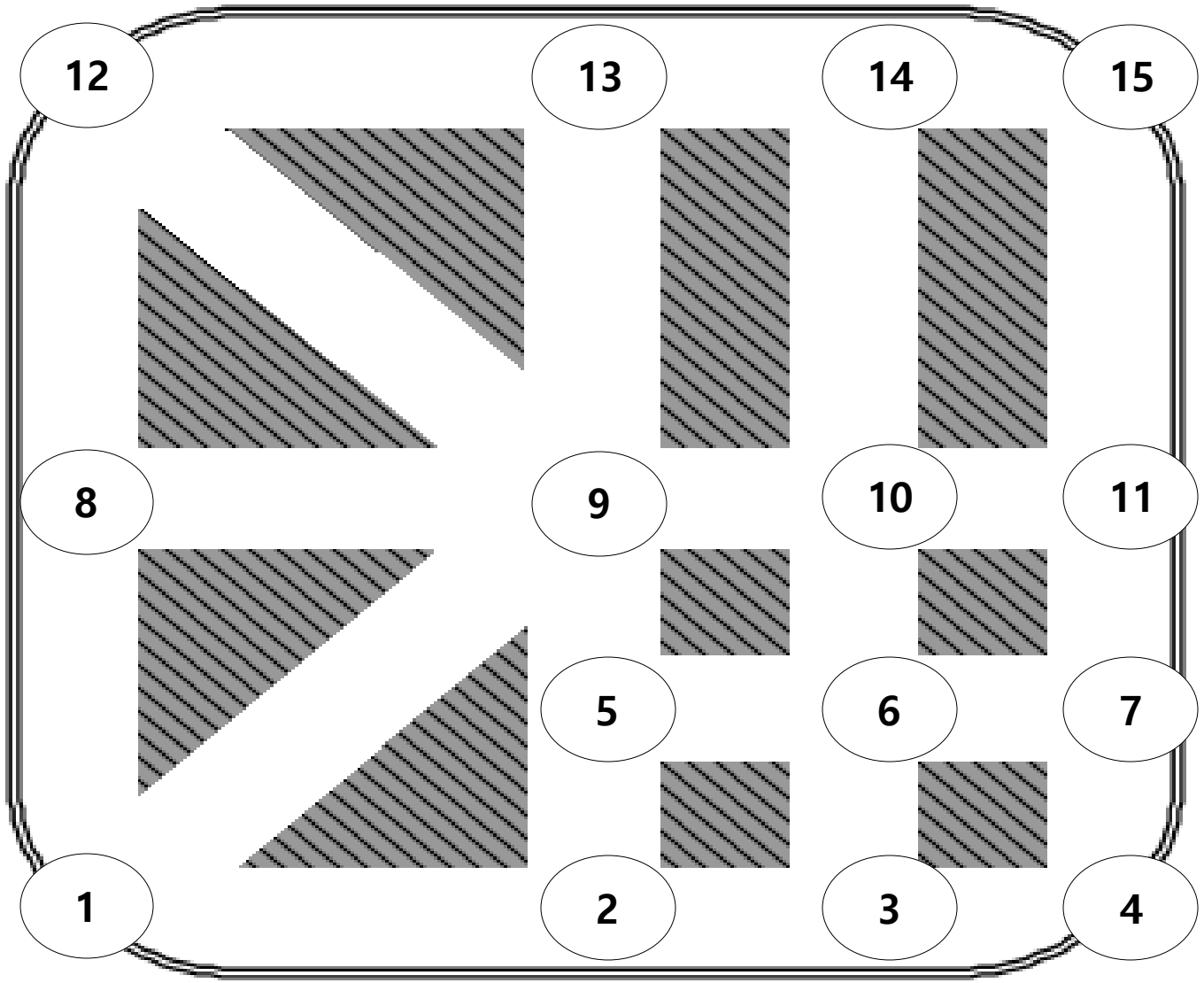
# 최단 경로 추종 차량

캡스톤디자인1 7조

# 최단 경로 추종 차량

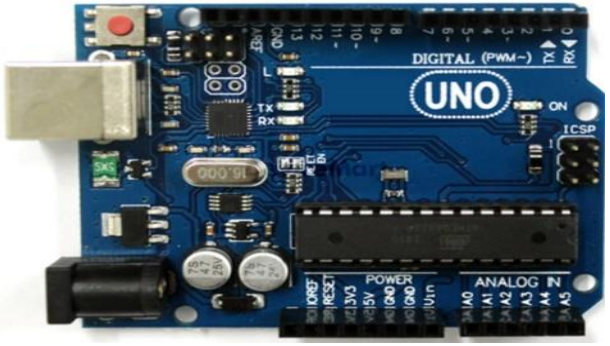



: 장애물

: 외벽

# 1. 트랙 스캔

연결




아두이노 우노

차량 각도/
장애물 거리
데이터

블루투스 모듈

블루투스 연결

데이터송신

라이다 센서를 이용한
자율 주행

# 2. 최단 경로 선택

출발 좌표/도착 좌표
최단 좌표 선택
(x, y)

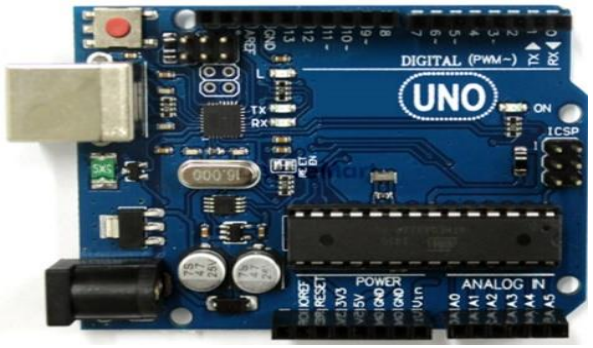
SLAM 2D 지도 생성

지도 위에 좌표 생성
(x, y)

작품 시연 시
여기까지 미리 준비

# 3. 경로 추종

연결




아두이노 우노

(x, y)
좌표

블루투스 모듈

(x, y)
좌표송신

출발 좌표에서 도착 좌표까지의
경로 추종

# 사진 출처

[SMG] 아두이노 4WD 주행로봇 프레임 세트 [SZH-EK098],
https://www.devicemart.co.kr/goods/view?no=1327455

[SMG] 아두이노 우노 R3 호환보드 [SZH-EK002],
https://www.devicemart.co.kr/goods/view?no=1245596

[OEM] 블루투스 모듈 HC-06 (DIP) 펌웨어 v1.8 [SZH-EK105],
https://www.devicemart.co.kr/goods/view?no=1376882

[스마트폰 이미지]
https://kr.freepik.com/free-vector/realistic-blue-smartphone-front-and-back_6208028.htm

노트북 이미지
https://prod.danawa.com/info/?pcode=13942415

# 카메라와 OPENCV(영상화 처리)를 통한 트랙 스캔과 차량 제어

# OPENCV(영상화 처리)

이번 캡스톤 디자인에서 OPENCV는 트랙 전체를 촬영하고 트랙 내에 있는 색깔을 통해 제어하는

즉 카메라를 센서로 사용하여 트랙에 좌표를 생성하여 좌표와 색깔을 통해 차량을 제어하고 차량에게 명령을 주는 역할을 수행 할 수 있도록 하였습니다.

# OPENCV(영상화 처리)

처음에는 SLAM을 맵을 스캔하여 작성을 하려 하였으나 ROS, SLAM, Linux등 기존에 배우지 않은 방식이 너무 많으며

버거봇 같은 어느정도 코드 정의가 되어있는 상태에서 하는 것이 아닌 아두이노와 라즈베리파이에 처음 설정부터 코딩까지 다 해야하며

라이다 센서와 ROS SLAM 등의 소프트웨어와 이외의 프레임워크를 이용해야 하므로 OPENCV를 통해 맵을 스캔하는 방식을 택하였습니다.

# OPENCV(영상화 처리)

PYTHON은 포인터가 없는 프로그래밍 언어이므로 좌표계를 생성하려면 numpy를 이용하여 눈에 보이지 않는 행렬을 카메라 화면에 깔고 그를 기반으로 x좌표와 y좌표를 생성해 낸다.

트랙 내에 파란색이 검출 될 경우 해당 위치에 차량이 있다고 코드에서 판단하고 해당 위치의 좌표값을 출력하며,

차량 위에 있는 파란색과 초록색의 사이값을 통하여 차량이 현재 얼마나 기울어져 있는지에 대해 알아 볼 수 있으며 해당 값은 차량 기울어짐에 대한 제어를 하는 Pure Pursuit 알고리즘에서 이용되었습니다.

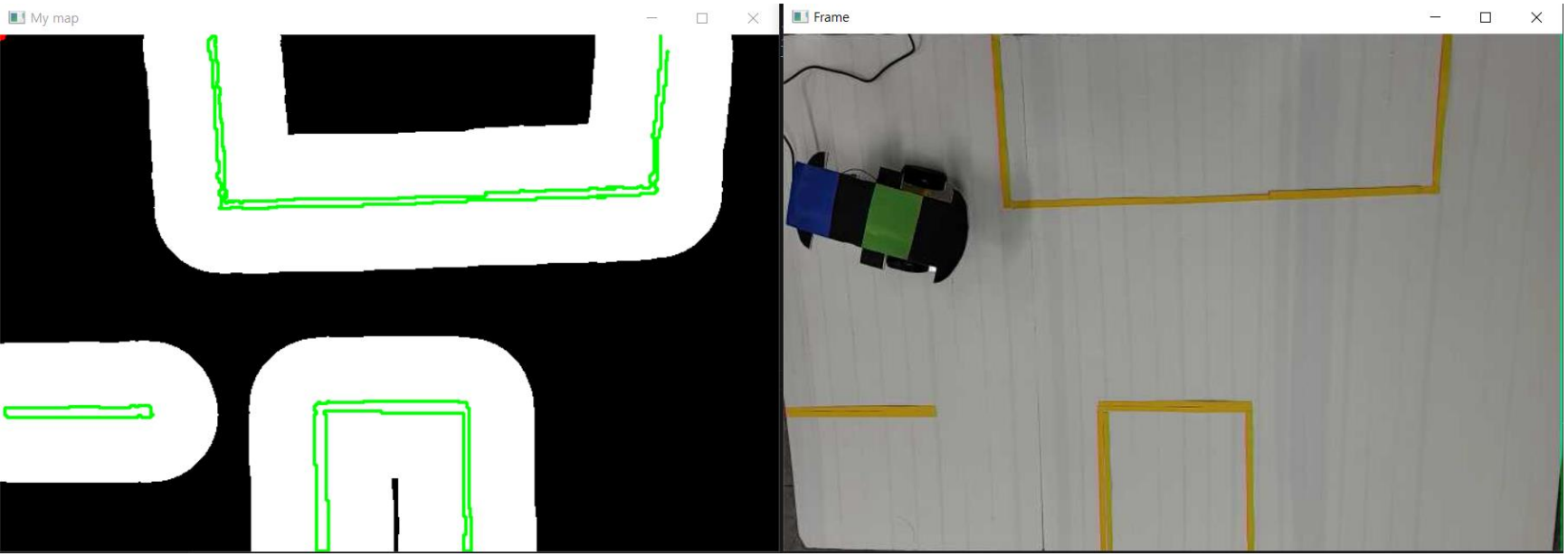
트랙 내에 노란색이 검출 될 경우 해당 위치에 장애물이 있다 판단하며, 앞에 나온 개념과 달리 특정 지점이 아닌 넓이의 형태이기 때문에, 노란색을 인식하면 노란색의 무게중심을 찾고 그 중심으로 좌표를 리스트에 담아 장애물의 범위를 최대한 정확하게 작성합니다.


# OPENCV(영상 화 처리)

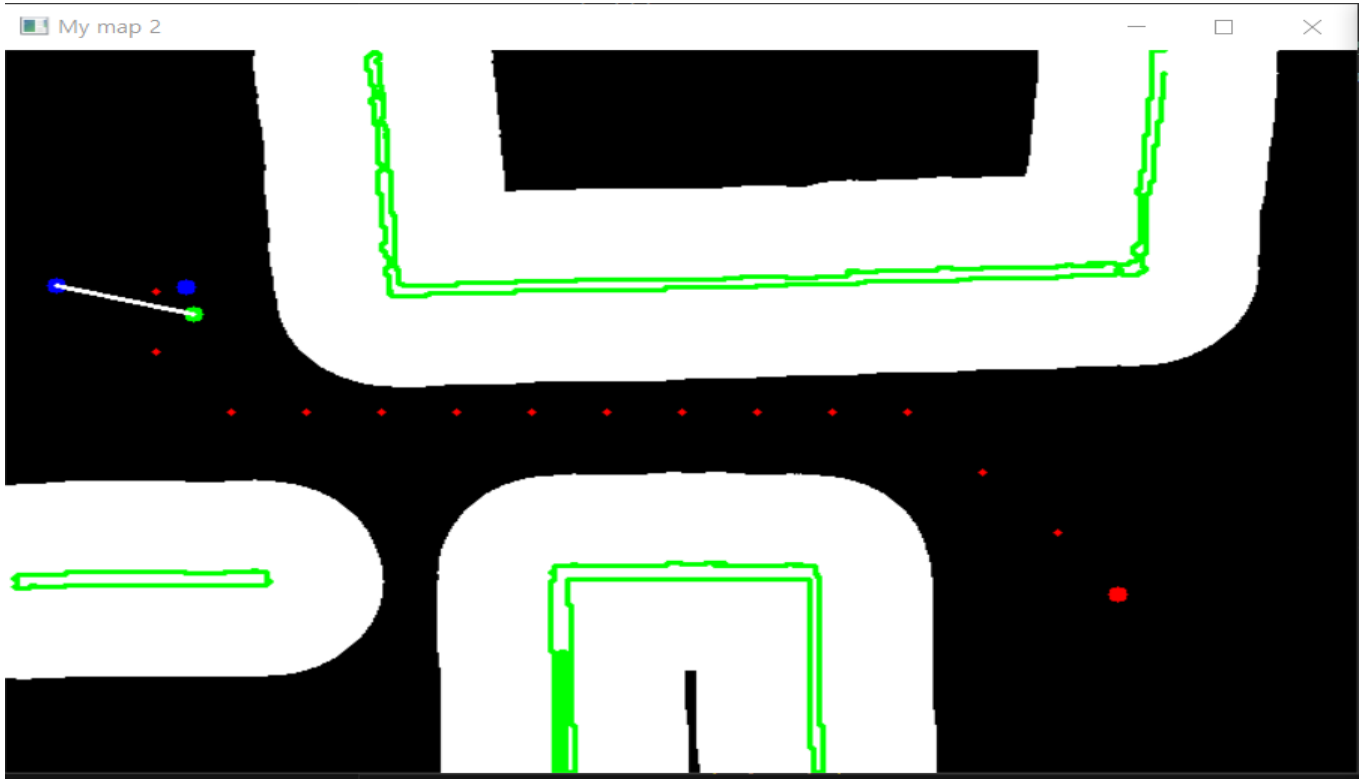
- 사진과 같이 천장 부분에 휴대폰 거치대를 부착한 다음 휴대폰을 거치대에 연결

- IP WEBCAM을 통해 같은 와이파이에 접속되어 있는 카메라 화면을 노트북과 공유하게 되며, 트랙 전체를 촬영한다.

# OPENCV(영상화 처리)



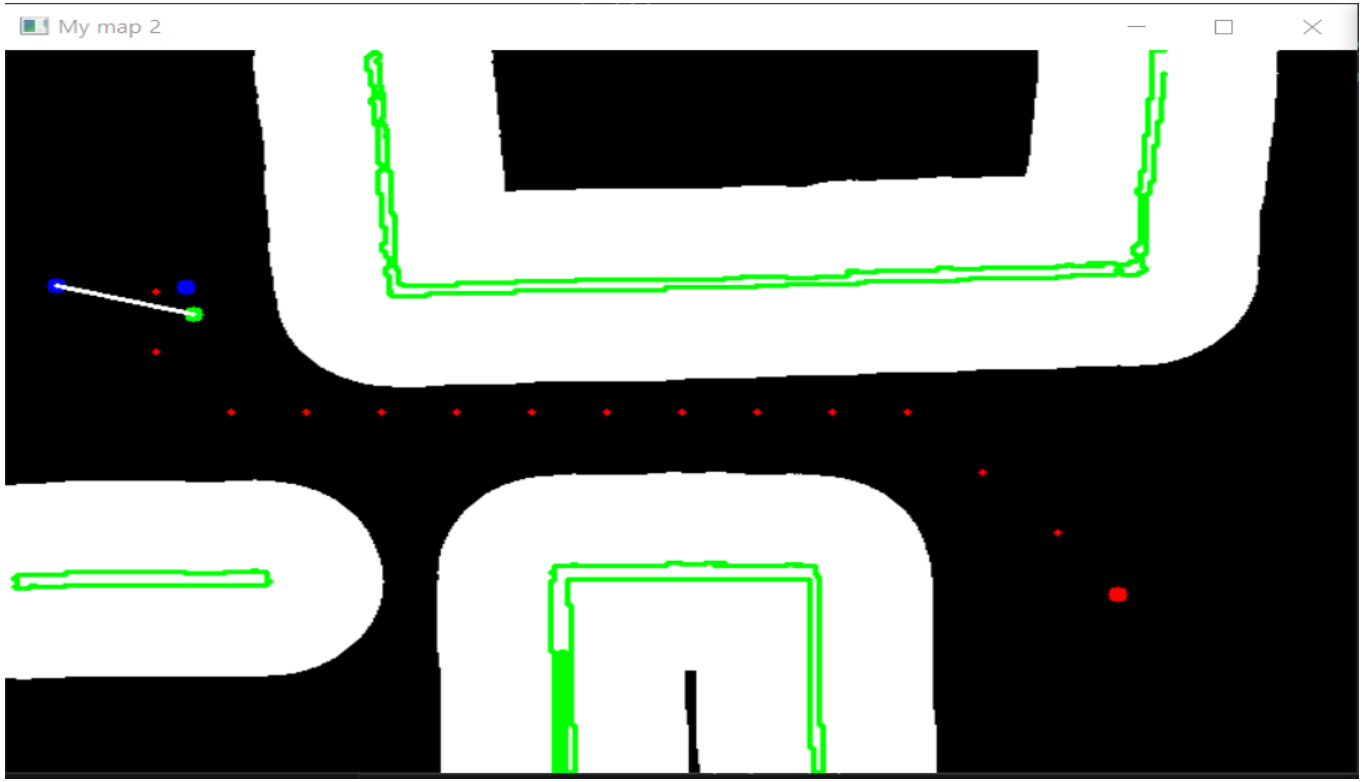

- 코드를 실행시키면 카메라가 트랙을 현재 촬영화면 그대로 가져오며, 노란색은 장애물로서 인식하며,
- 하얀부분은 차량이 최단거리를 찾아보니 최대한 붙어서 가는데 그것을 방지하기 위해 보정을 통해 장애물의 크기를 실제보다 더 크게 화면에 설치한 것이다.




# OPENCV(영상화 처리)

화면을 클릭하게 될 경우 사진과 같이 파란색과 빨간색이 화면에 찍히며 , 첫 번째 클릭을 파란색 즉 출발점이며

두 번째 클릭을 빨간색 점으로 도착점이 되며 이 두 점사이의 좌표를 받아 Astar 알고리즘이 경로를 생성하게 된다.




# OPENCV(영상화 처리)

하얀색 선으로 연결된 파란색과 초록색은 현재 차량의 기울어짐을 나타내며,

차량이 기울지 않고 올바르게 갈 수 있게 해주는

Pure Pursuit 알고리즘을 정상적으로 동작해주기 위함이다.

# A*알고리즘을 이용한 경로생성

# A*알고리즘이란?

- 컴퓨터과학 분야에 많이 사용되는 알고리즘으로 출발노드와 도착노드간의 최단경로를 찾아내는 그래프 탐색 알고리즘

- 평가함수 f(n)을 사용하여 동작하며 f(n)은 출발지와 인접노드간의 비용 g(n)과 휴리스틱함수 h(n)의 합으로 표현됨

$$f(n) = g(n)+h(n)$$

- f(n)의 가능한 모든 경우의수 중 가장 작은 값이 나오는 경우를 선택하여 최단경로생성

# 휴리스틱 함수

- 주어진 정보를 통해 동작의 분기단계에서 어느 한 분기를 선택하기 위해 사용하는 함수
- 종류로는 맨하탄거리방식, 유클리드거리방식, 체비셰프거리방식등이 존재함
- 프로젝트에서는 pure pursuit 알고리즘간의 연계를 고려해 각 노드간의 직선거리와 대각선 거리 모두를 고려하는 유클리드거리 방식을 사용함

# A*알고리즘의 동작 과정

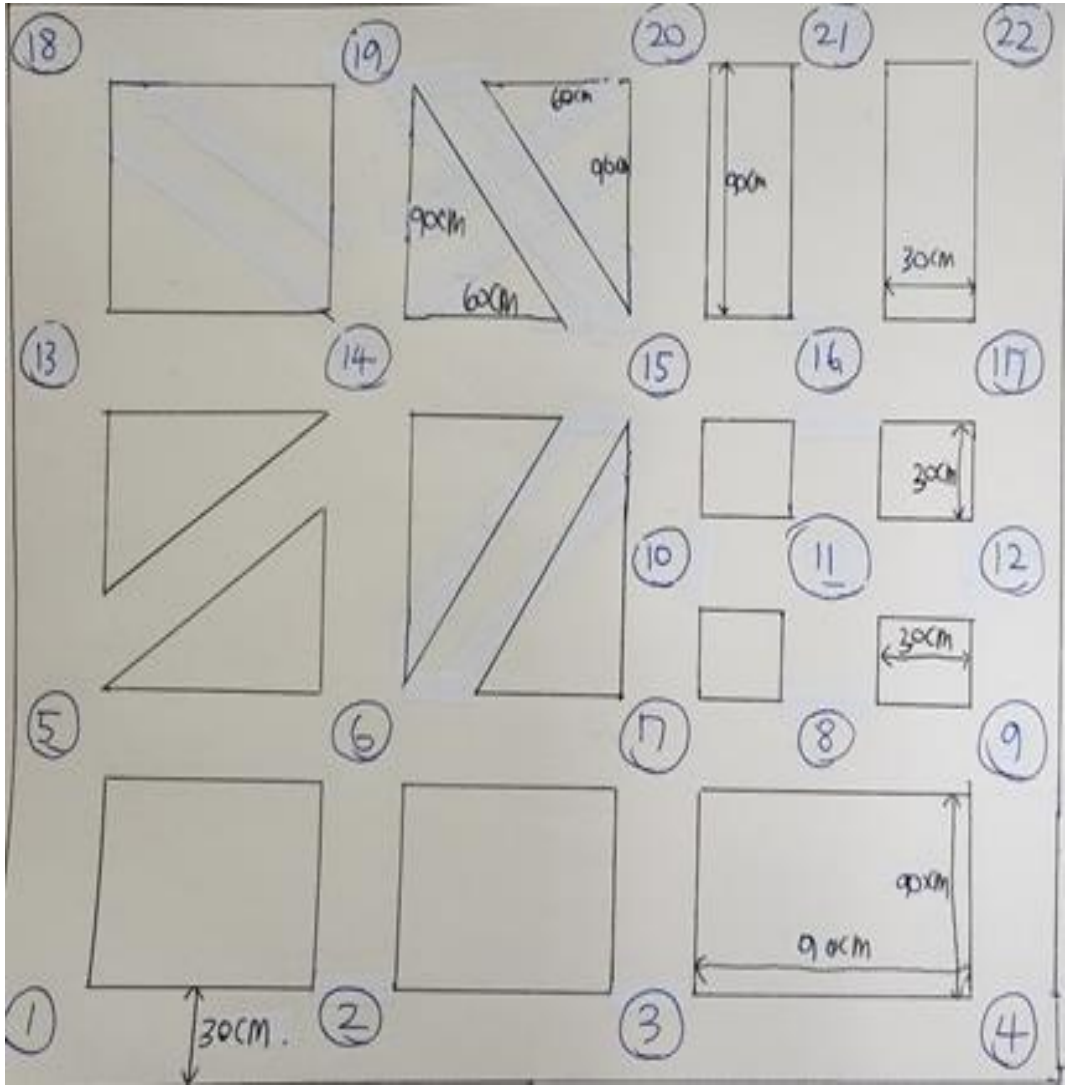
1.그림과 같은 랜덤한 트랙에서 노드의 위치와 개수를 결정

2.각 노드간의 비용을 결정

목적은 최단경로 탐색임으로 비용을 각 노드간의 거리로 결정함

3.시작노드와 도착노드를 설정

4.A*알고리즘을 통해 최단경로 도출



# A*알고리즘 모의실험(1) (OCTAVE)

임의의 트랙을 생성후 장애물의 비용을
무한으로 하여 임의로 배치

시작점은 상단 맨 왼쪽
(트랙 노드 행렬의 1x1 성분)
도착점은 하단 맨 오른쪽
(트랙 노드 행렬의 7x6 성분)

이동가능한 경로는 0이 아닌
상수로 표현

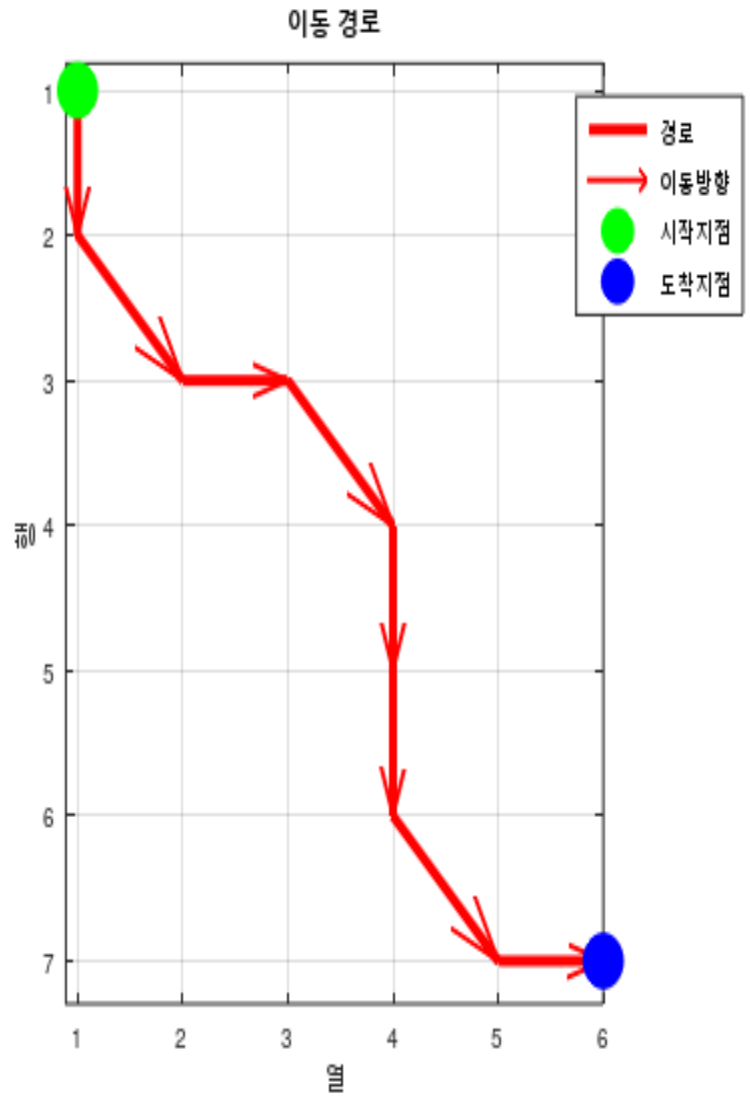
장애물을 제외한 각 노드간의 비용은
시작점을 제외하고 1로 설정
(각 노드간의 거리가 균일하다 가정함)

경로와 비용이 출력되고 이동방향이
화살표로 표시됨

```
start point:
   1   1
Goal point:
   7   6
Map Size:
   7   6
Map:
    0     1     1     1     1     1
    1   Inf   Inf     1   Inf     1
    1     1     1     1     1     1
    1   Inf   Inf     1   Inf     1
    1     1     1     1     1     1
    1   Inf     1     1   Inf   Inf
    1   Inf   Inf     1     1     1
Path:
   1   1
   2   1
   3   2
   3   3
   4   4
   5   4
   6   4
   7   5
   7   6
Cost (excluding start):
9.2426
```

시작점

도착점



# A*알고리즘 모의실험(2) (OCTAVE)

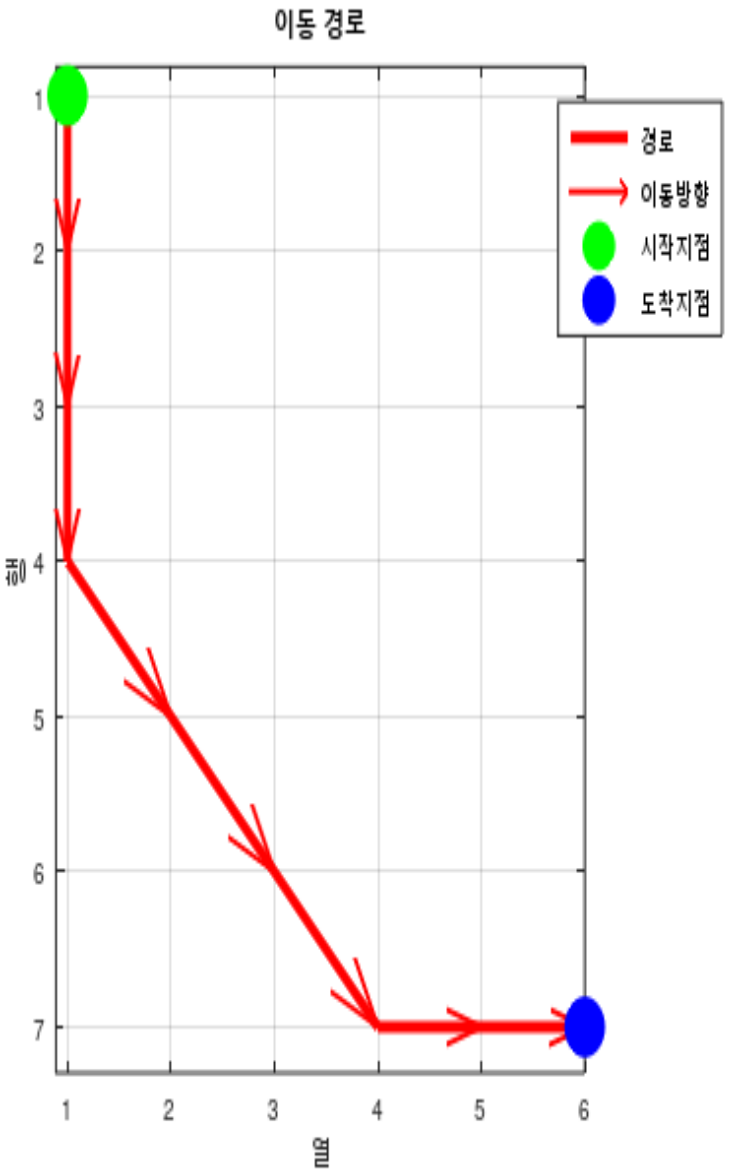
이전 페이지의 이동경로에 장애물을 추가 배치하여 최단경로를 재탐색

새로운 경로를 생성하는것을 확인할 수 있음

```
start point:
   1   1
Goal point:
   7   6
Map Size:
   7   6
Map:
     0     1     1     1     1     1
     1   Inf   Inf     1   Inf     1
     1     1     1     1     1     1
     1   Inf   Inf   Inf   Inf     1
     1     1     1     1     1     1
     1   Inf     1     1   Inf   Inf
     1   Inf   Inf     1     1     1
Path:
   1   1
   2   1
   3   1
   4   1
   5   2
   6   3
   7   4
   7   5
   7   6
Cost (excluding start):
9.2426
```

장애물 추가

변경된 경로



# A*알고리즘 모의실험(3) (OCTAVE)

추가 장애물을 제거하고 원래의 트랙에서 시작점과 도착점을 변경

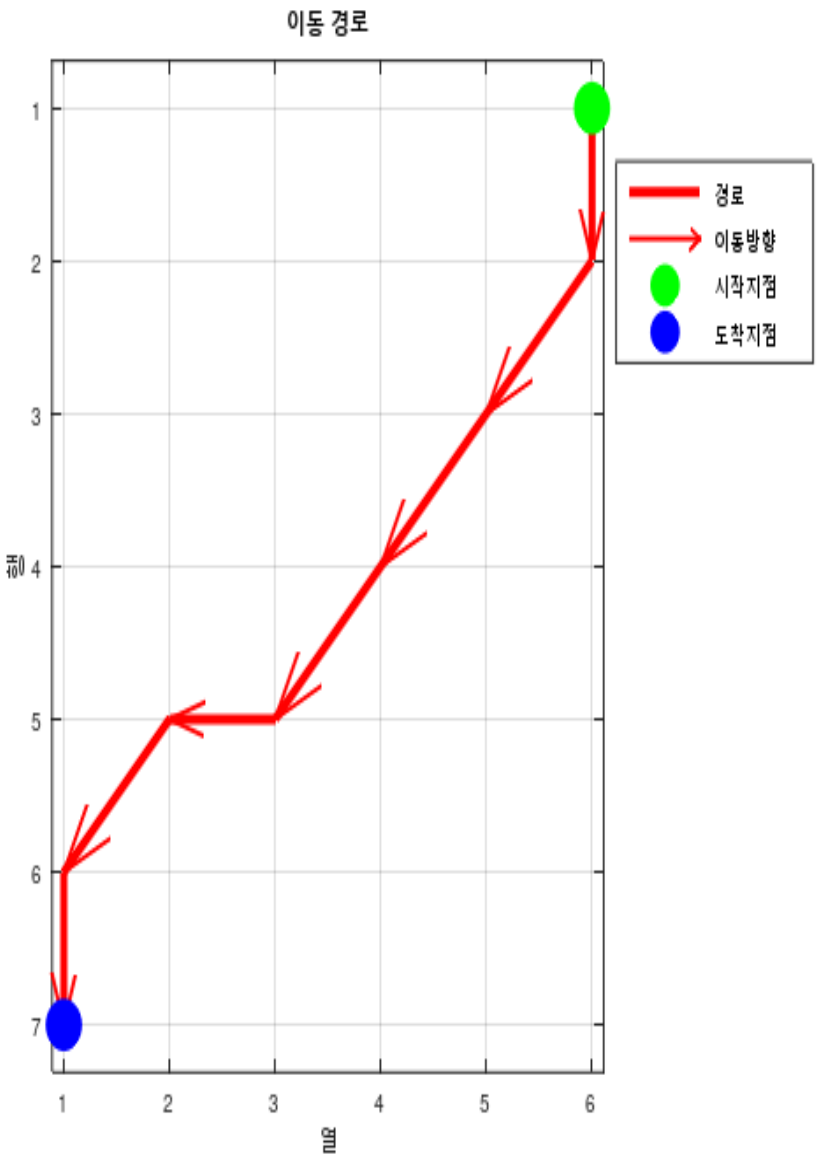
시작점은 상단 맨 오른쪽
(트랙 노드 행렬의 1x6 성분)
도착점은 하단 맨 쪽
(트랙 노드 행렬의 7x1 성분)

새로운 이동 경로가 생성됨

```
start point:
   1   6
Goal point:
   7   1
Map Size:
   7   6
Map:
     1     1     1     1     1     0
     1   Inf   Inf     1   Inf     1
     1     1     1     1     1     1
     1   Inf   Inf     1   Inf     1
     1     1     1     1     1     1
     1   Inf     1     1   Inf   Inf
     1   Inf   Inf     1     1     1
Path:
   1   6
   2   6
   3   5
   4   4
   5   3
   5   2
   6   1
   7   1
Cost (excluding start):
8.6569
```

시작점

도착점

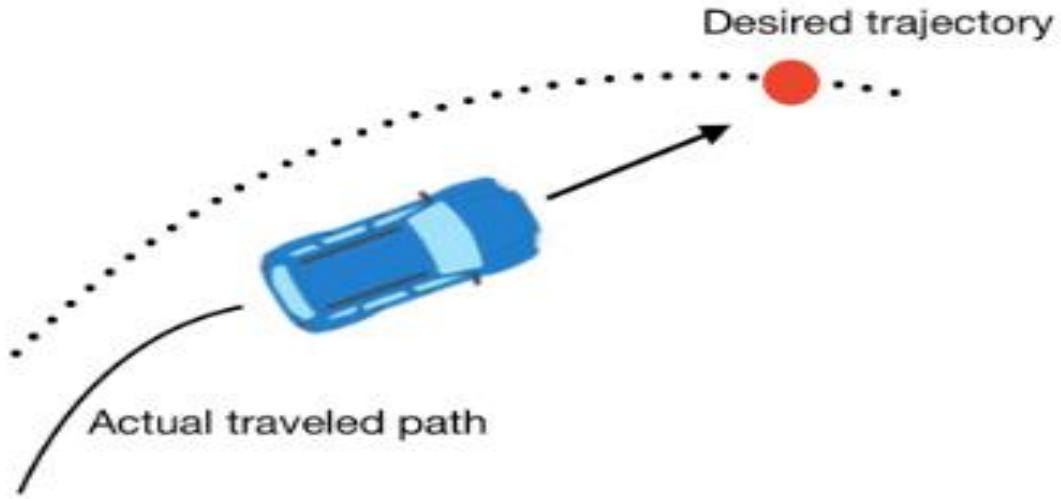


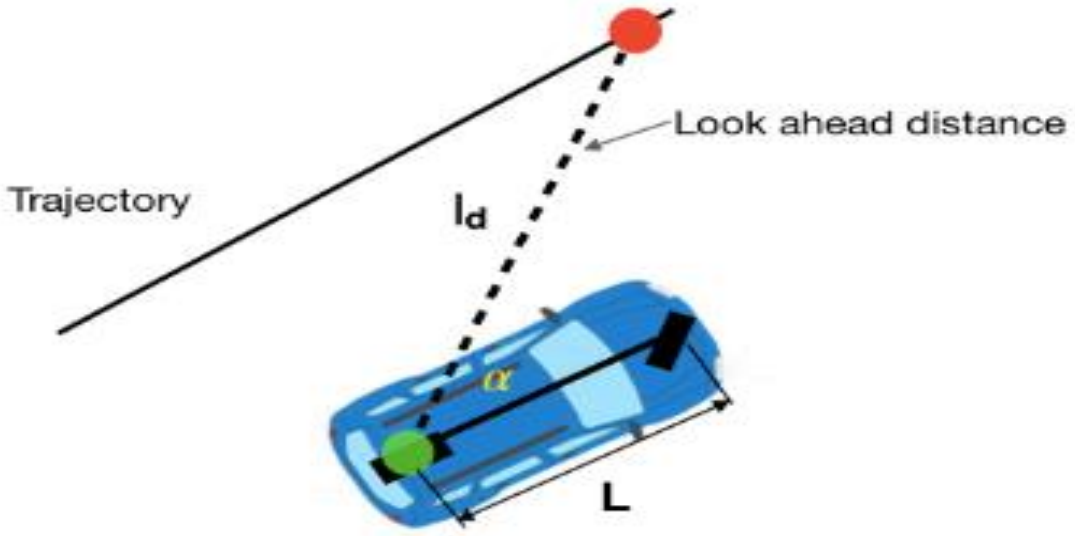
# 프로젝트에서의 A*알고리즘

- 카메라로 스캔된 맵에서 numpy와 opencv로 생성된 좌표를 사용해 A*알고리즘을 실행
- 호완성을 위해 파이썬에서 작성된 코드를 사용하여 각 요소들과 연계
- 생성된 경로를 이루는 좌표는 pure pursuit 알고리즘의 waypoint가 됨

# Pure pursuit 알고리즘을 통한 경로 추종

# pure pursuit 알고리즘의 경로 추종 방식



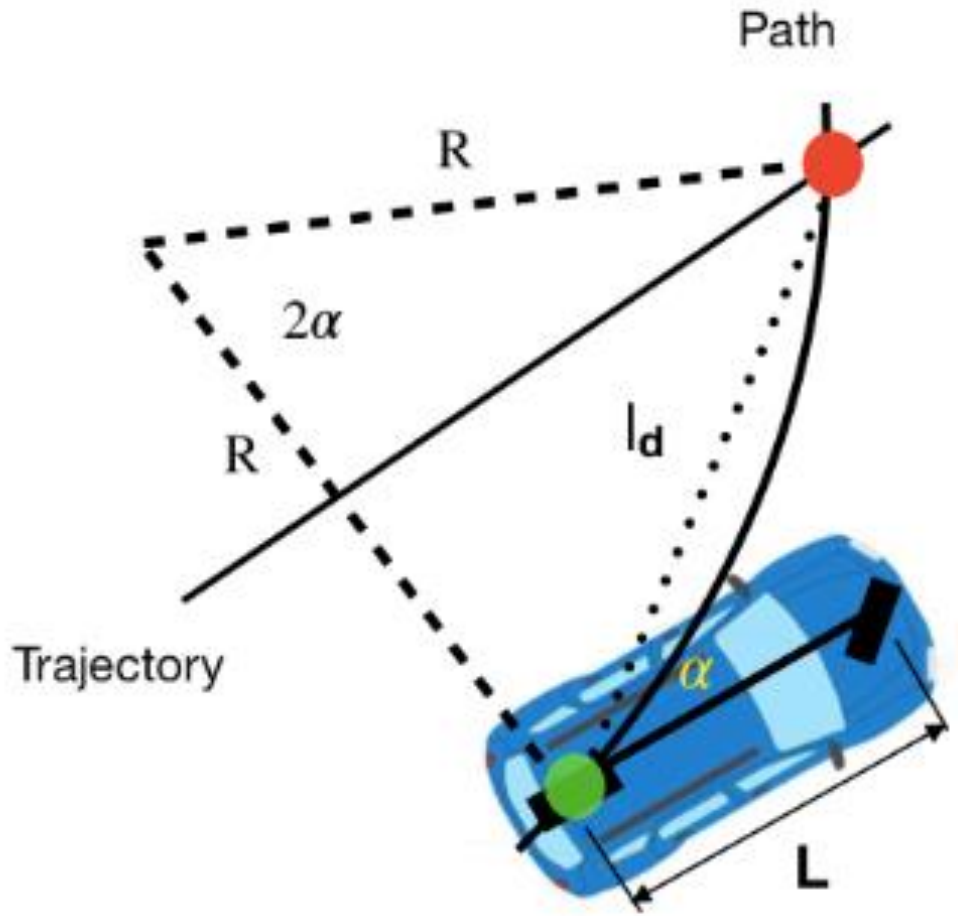




이미지 출처 : https://www.shuffleai.blog/blog/Three_Methods_of_Vehicle_Lateral_Control.html

- 일련의 Waypoint로 이루어진 경로를 추종할 때, 일정한 거리(Lookahead Distance) 만큼 앞에 위치한 Waypoint에 초점을 맞춰 해당 지점까지 원의 궤적을 그리며 경로를 추종하는 알고리즘.

- 차량의 후륜축의 중심과 목표지점 사이의 거리를 Ld(Lookahead Distance)라고 하고 차량의 방향(헤딩 방향)과 Ld가 이루는 각도를 α라고 한다. 또한 전륜과 후륜 사이의 길이를 L이라고 한다.

# $l_d$값에 따른 궤적의 곡률




이미지 출처 : https://www.shuffleai.blog/blog/Three_Methods_of_Vehicle_Lateral_Control.html

- 원의 순간회전중심(ICR)과 반지름R이 다음과 같이 나타나며 싸인법칙을 통해서 $l_d$와 α값이 주어졌을 때, 어떠한 원의 궤적을 그리며 경로에 복귀하는 지 계산할 수 있음.

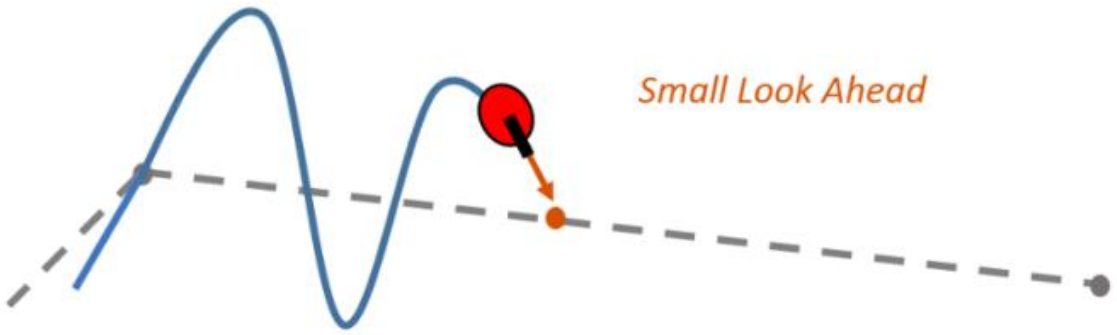
- 계산과정

$$\frac{l_d}{sin2\alpha} = \frac{R}{sin(\frac{\pi}{2} - \alpha)} \Rightarrow \frac{l_d}{2sin\alpha cos\alpha} = \frac{R}{cos\alpha}$$

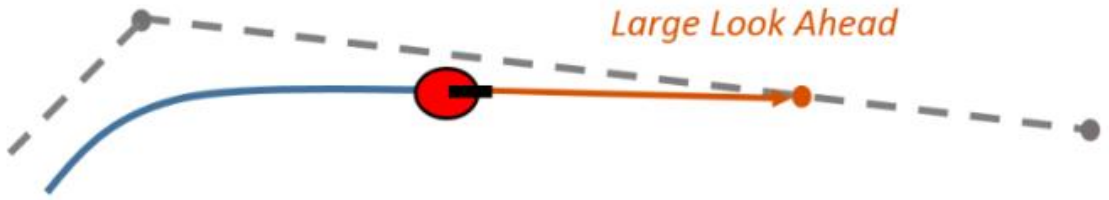
$$\frac{l_d}{sin\alpha} = 2R \Rightarrow k = \frac{1}{R} = \frac{2sin\alpha}{l_d}$$

위의 계산을 통해서 $l_d$값에 따른 궤적의 곡률을 구할 수 있게됨.

# $l_d$값을 정하는 기준



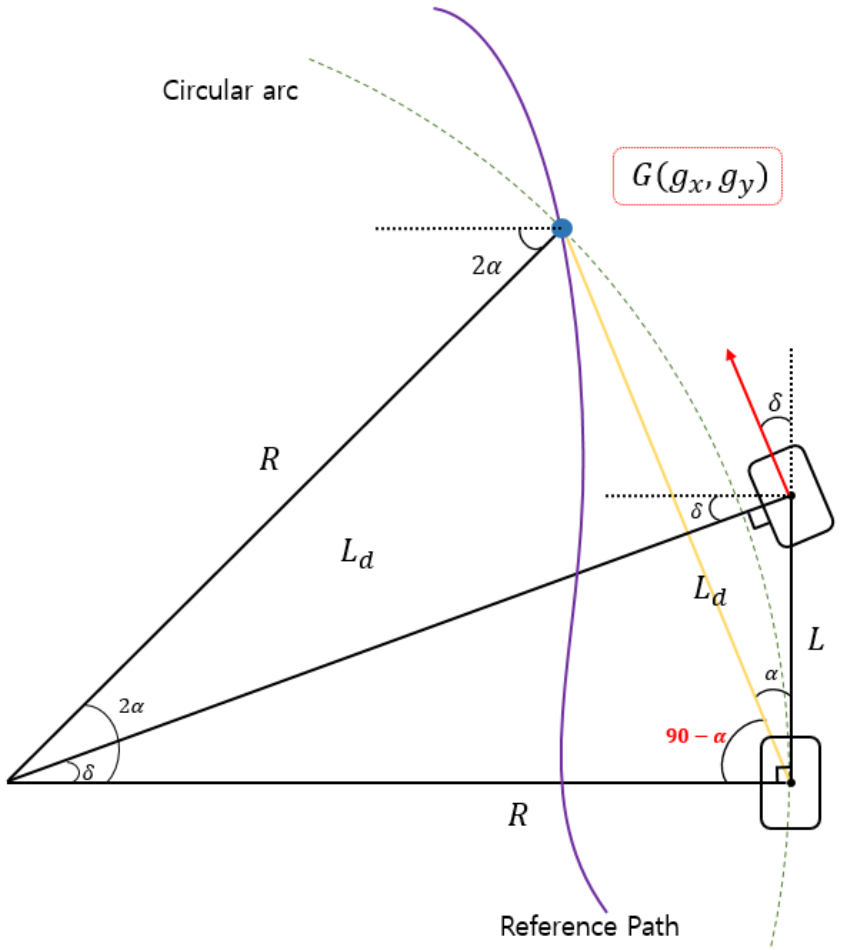

- lookahead distance를 너무 작게 설정한 경우 짧은 거리의 경유지를 목표지점으로 설정하여  낮은 곡률로 빠르게 경로로 되돌아옴
- 그러나 차량이 경로를 초과하여 경로를 따라 진동할 수 있음.




- lookahead distance를 너무 먼 거리로 설정하면 경로를 따라 부드럽게 주행하지만 방향 전환을 크게해야 하는 경로가 존재할 경우엔 경로를 정확히 따라가지 않아 경로 추종 성능이 저하될 수 있다.

이미지 출처 : https://kr.mathworks.com/help/robotics/ug/pure-pursuit-controller.html

# 조향각(δ) 결정




이미지 출처 : https://blog.naver.com/thoon1113/222887453151

- 차량의 전륜과 후륜 사이의 거리를 알고 있다면 닮음을 이용하여 조향각(δ)를 구할 수 있음.

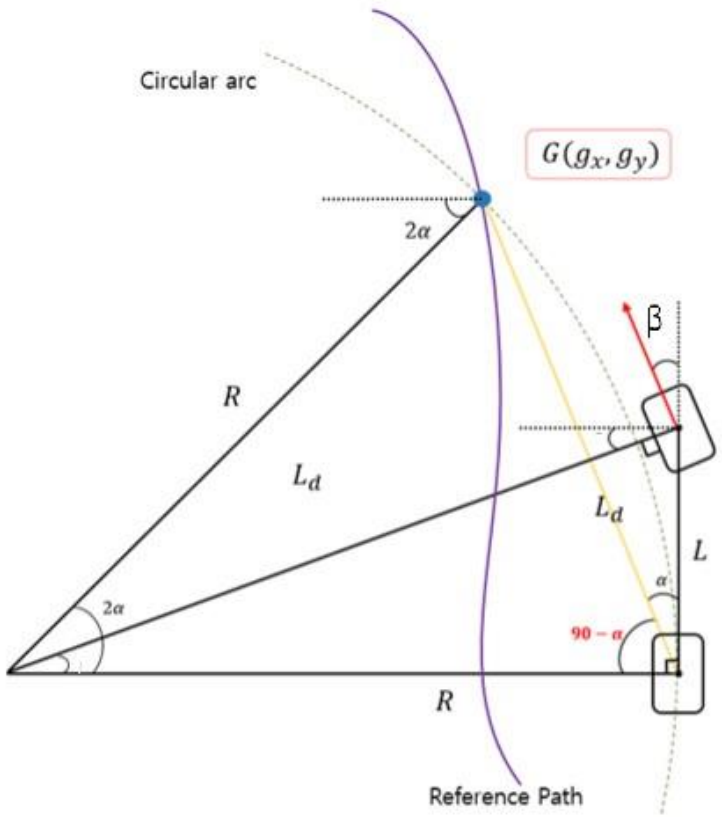
$$\delta = \tan^{-1}\frac{L}{R}$$

- 이때 위에서 구했던 곡률 k의 식을 이용하면

$$\delta = \tan^{-1}(\frac{L}{R}) = \tan^{-1}(L \times k) = \tan^{-1}(\frac{2 \cdot L \cdot sin\alpha}{L_d})$$

- 앞으로 ω = β 로 표현함.

# 각속도(ω) 결정




- 반지름과 조향각(β)를 통해서 차량이 얼마의 속도로 돌아야 하는지에 대한 각속도를 구할 수 있음.

$$R = \frac{L}{tan(\beta)}, \qquad \omega = \frac{v}{R}$$

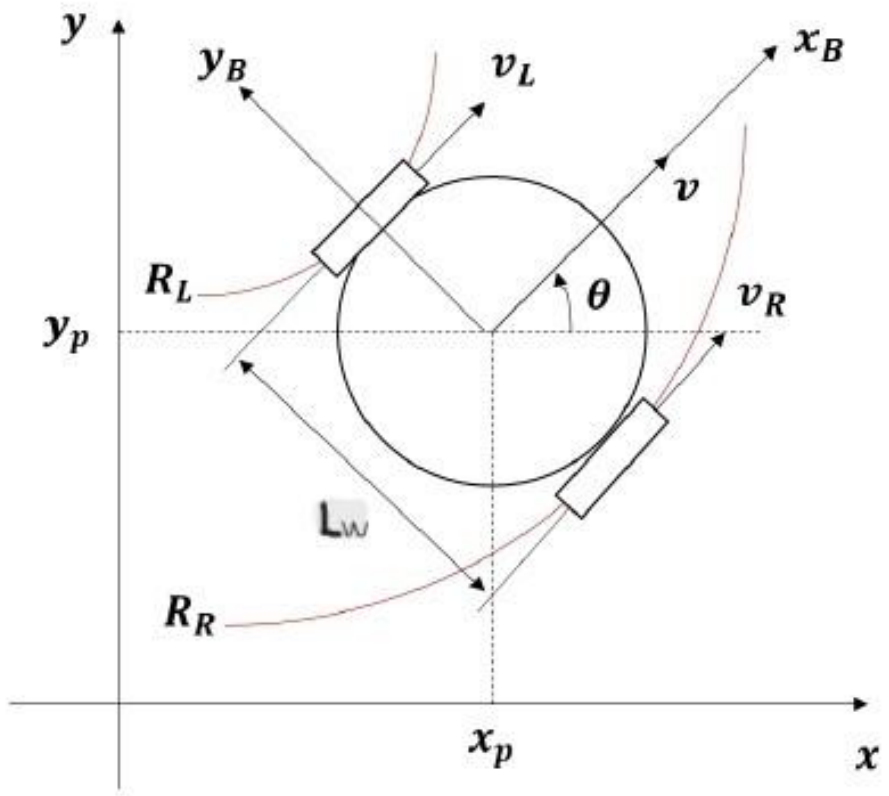
- 위의 두 식에 의하여 각속도(ω)는 다음과 같이 구해진다.

$$\omega = v \times \frac{tan(\beta)}{L}$$

- 위의 식은 차량의 전진 속도와 원의 반지름을 이용하여 차량이 얼마나 빠르게 회전해야 하는지를 알려준다.

이미지 출처 : https://blog.naver.com/thoon1113/222887453151

# 차동 구동 차량의 모터 입력 결정



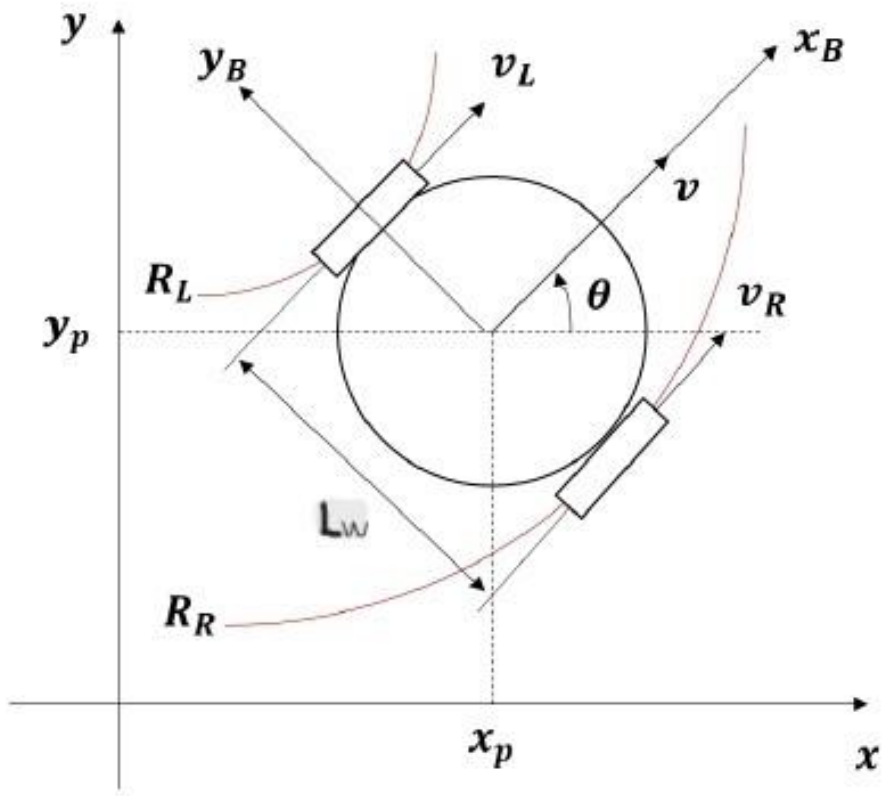

- 차량이 회전할 때, 어떤 가상의 중심점을 가진 원의 형태로 회전함. 이때 한쪽 바퀴는 더 작은 반지름을 갖는 원 주위로 움직이고, 다른쪽 바퀴는 더 큰 반지름을 가지는 원 주위로 움직임. 참고로 각속도와 선속도의 관계는 $v = \omega \times r$ 임.
- 그림과 같이 회전중이라고 가정하고, 양 바퀴 사이의 간격을 Lw라고 할 때, 왼쪽 바퀴와 오른쪽 바퀴의 회전 반경과 중심 선속도와 각각 $\frac{L_w}{2}$ 씩 떨어져 있으므로 바퀴의 선속도는 다음과 같음.

$$R_L = R - \frac{L_w}{2}, v_L = \omega \times (R - \frac{L_w}{2}),$$

$$R_R = R + \frac{L_w}{2}, v_R = \omega \times (R + \frac{L_w}{2})$$

이미지 출처 : https://jkros.org/_common/do.php?a=full&b=12&bidx=2192&aidx=26078

# 차동 구동 차량의 모터 입력 결정




- 위의 식을 전개하면 다음과 같다.

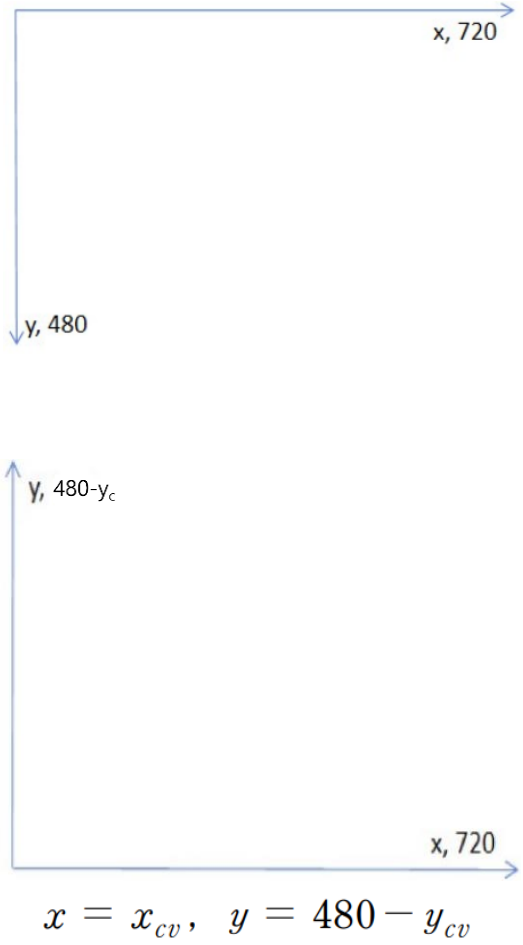
$$leftSpeed = v_{L=}\nu - \omega \times \frac{L_w}{2}$$

$$rightSpeed = v_{R=}\nu + \omega \times \frac{L_w}{2}$$

- 위와 같이 전개하면 조향각(β)에 따라 ω값의 크기와 부호가 변하며 회전 방향을 조절함을 더 직관적으로 알 수 있다.

이미지 출처 : https://jkros.org/_common/do.php?a=full&b=12&bidx=2192&aidx=26078

# opencv 좌표계 -> 직교 좌표계




- Pure pursuit 알고리즘과 양쪽 모터의  선속도는 직교 좌표계를 기준으로 계산되었다. 그러나 필요한 데이터를 전송해주는 opencv에서는 직교 좌표계가 아닌 opencv 좌표계를 사용하며 이는 직교좌표계와 y축의 방향이 다르며 왼쪽의 가장 상단부가 원점이 된다.

- 따라서 opencv에서 전송해주는 차량의 헤딩방향, 차량의 현재 위치, 경로의 Waypoints 좌표들의 데이터들은 직교 좌표계에 맞게 다시 계산되어야 하며 그 값은 아래와 같다.

- Waypoints 및 차량의 현재 위치 : $(x_{cv}, 480 - y_{cv})$

- 차량의 헤딩 방향 : $\theta = \tan^{-1}\frac{480 - y_{cv}}{x_{cv}}$

# 코드 병합 및 데이터 송신

캡스톤디자인2 7조 60191782 임홍균

# 컴퓨터 비전 (OPENCV)

- 장애물을 인식하고 차량의 현재 좌표와 현재 각도를 계산

코드 보수/병합

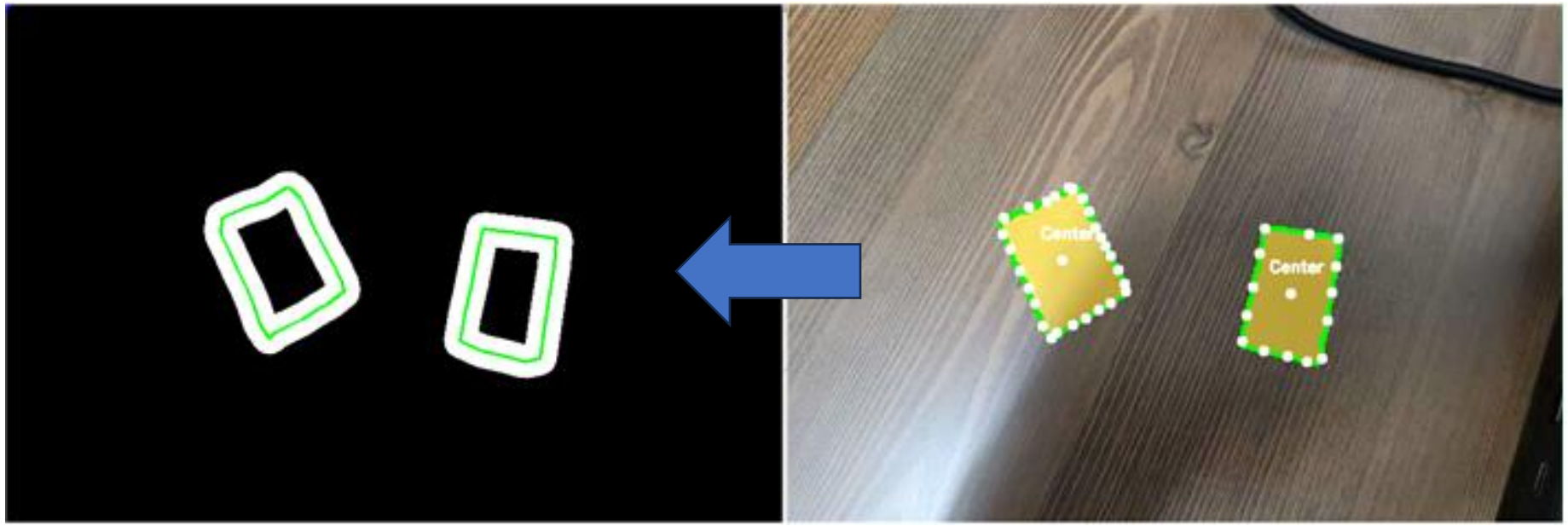
# A* 알고리즘

- 시작 좌표로부터 도착 좌표까지의 최단 경로 좌표 계산

데이터 송신

# Pure Pursuit 알고리즘

- 차량의 현재 좌표와 현재 각도를 이용한 경로 추종




장애물 좌표를
my_map이라는 배열에 담는다

$$mymap = \begin{bmatrix} (0,0,0) & (0,0,0) & (0,0,0) & \cdots \\ (0,0,0) & (254,255,255) & (254,255,255) & \\ (0,0,0) & (254,255,255) & (0,255,0) & \\ \vdots & & & \ddots \end{bmatrix} \quad costmap = \begin{bmatrix} (1) & (1) & (1) & \cdots \\ (1) & (255) & (255) & \\ (1) & (255) & (1) & \\ \vdots & & & \ddots \end{bmatrix}$$

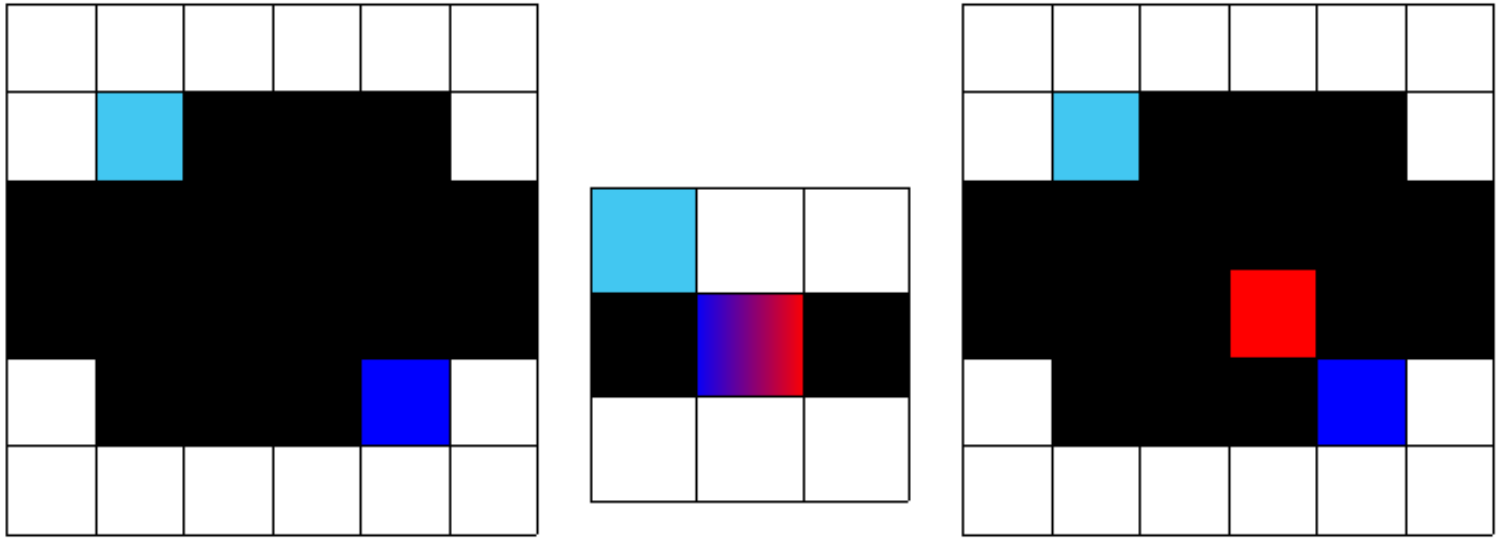
A* 알고리즘에 필요한 비용 좌표가 담긴 배열을 cost_map이라고 하자.

my_map의 흰색 픽셀 좌표 (장애물 좌표)를 (254, 255, 255) 원소로 만든다. 그리고 cost_map을 my_map의 첫 번째 채널로 만든 뒤에 모든 원소에 1을 더하면 my_map의 장애물 좌표를 cost_map에 옮길 수 있다. 현재 좌표는 비용이 0 이기 때문에 cost_map에서 (0) 원소로 만든다.

my_map의 크기는 카메라 해상도와 일치하므로 카메라 해상도가 1280 X 720이라면 my_map의 크기도 1280 X 720이다.

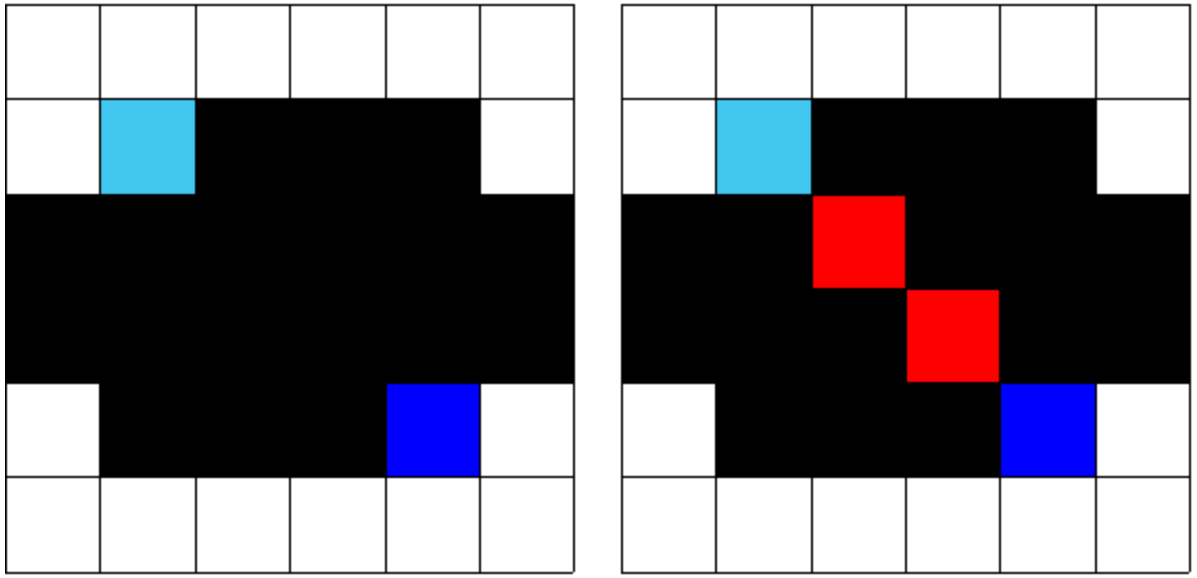
결국 cost_map의 크기도 1280 X 720이 되며 원소 개수 (좌표 개수)는 1280 X 720 = 921,600개가 된다.

=> cost_map의 좌표 개수가 많아 최단 경로 좌표를 계산하는 시간이 오래 걸려서 이를 수정



이번에도 왼쪽 위의 그림이 my_map이라고 하자. 하지만 이번에는 my_map에서 폭과 높이를 3등분한다. 따라서 my_map은 9개의 원소를 가진 새로운 배열이 된다.

새로운 배열은 중앙 위의 그림이며 이를 cost_map이라고 하자. 현재 좌표로부터 도착 좌표까지의 최단 경로 좌표는 중앙 위 그림에서 보이는 것과 같다.



흰색 픽셀 좌표는 장애물 좌표, 하늘색 픽셀 좌표는 현재 좌표, 파란색 픽셀 좌표는 도착 좌표, 빨간색 픽셀 좌표는 최단 경로 좌표이다.

my_map과 cost_map의 크기가 같으면 현재 좌표로부터 도착 좌표까지의 최단 경로 좌표는 오른쪽 위 그림과 같이 계산된다.

cost_map의 좌표 개수가 6 X 6 = 36개에서 3 X 3 = 9개로 감소한 것을 알 수 있다.

=> cost_map의 좌표 개수가 감소해 최단 경로 좌표를 계산하는 시간이 감소

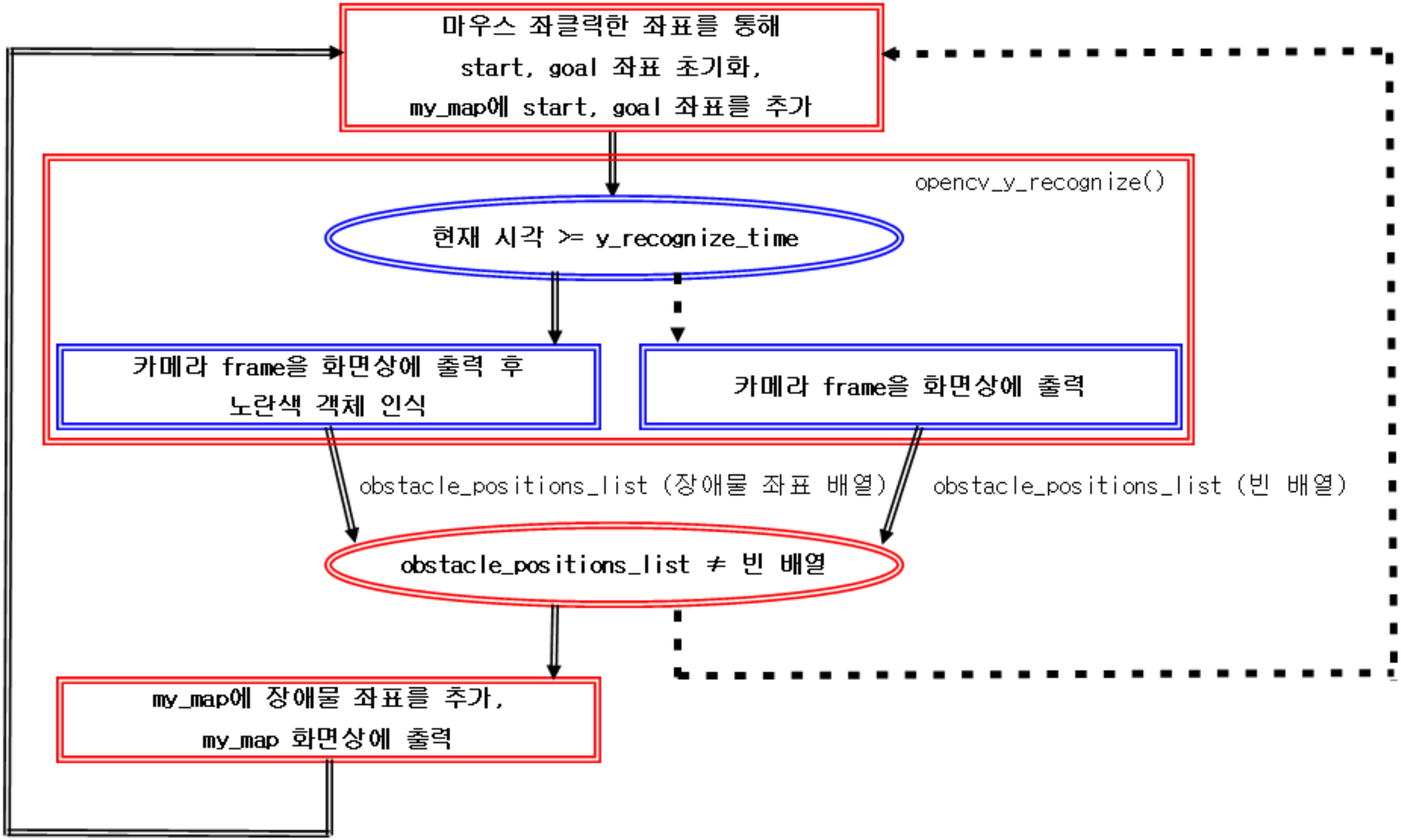
아두이노가 시작 좌표로부터 도착 좌표까지의 최단 경로 좌표를 받아들일 수 있는가?

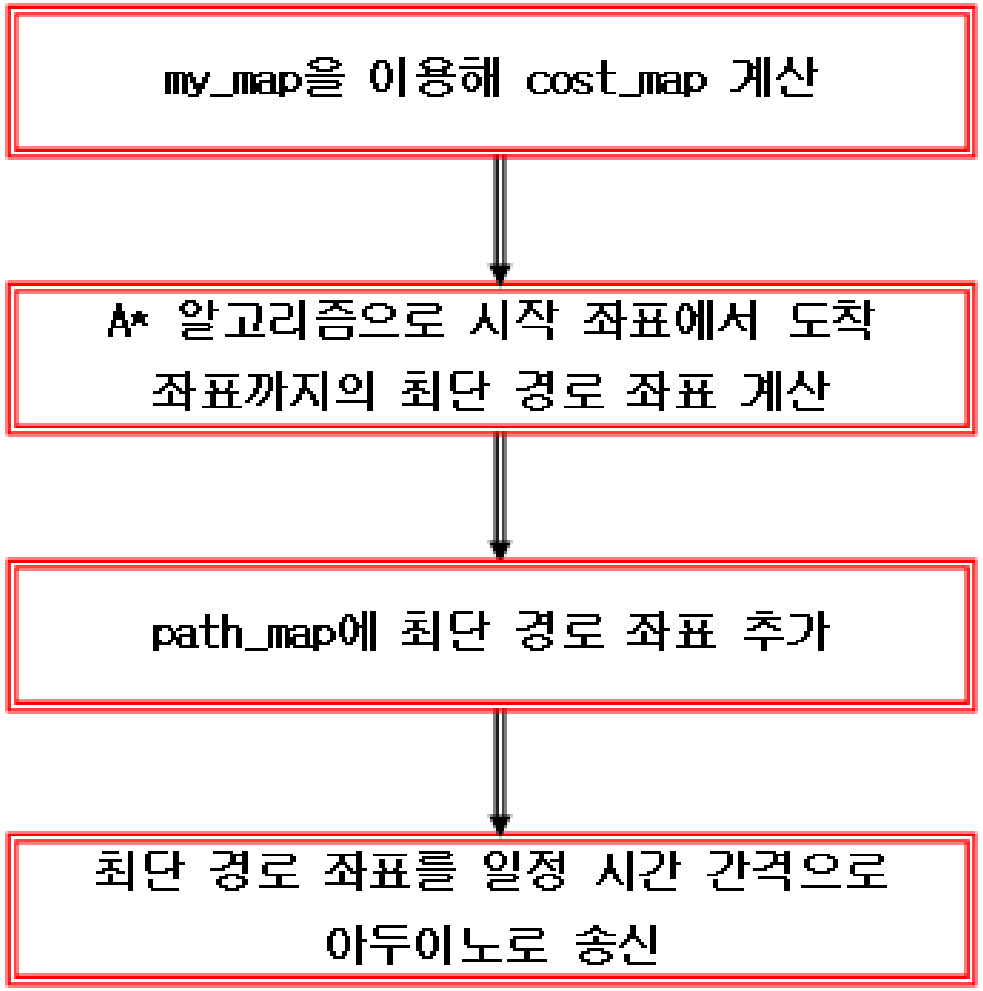
=> 노트북이 최단 경로 좌표를 문자열의 형태로 보내되 일정 시간 간격으로 아두이노로 보내서 아두이노에 최단 경로 좌표 배열을 생성하게끔 하였다.

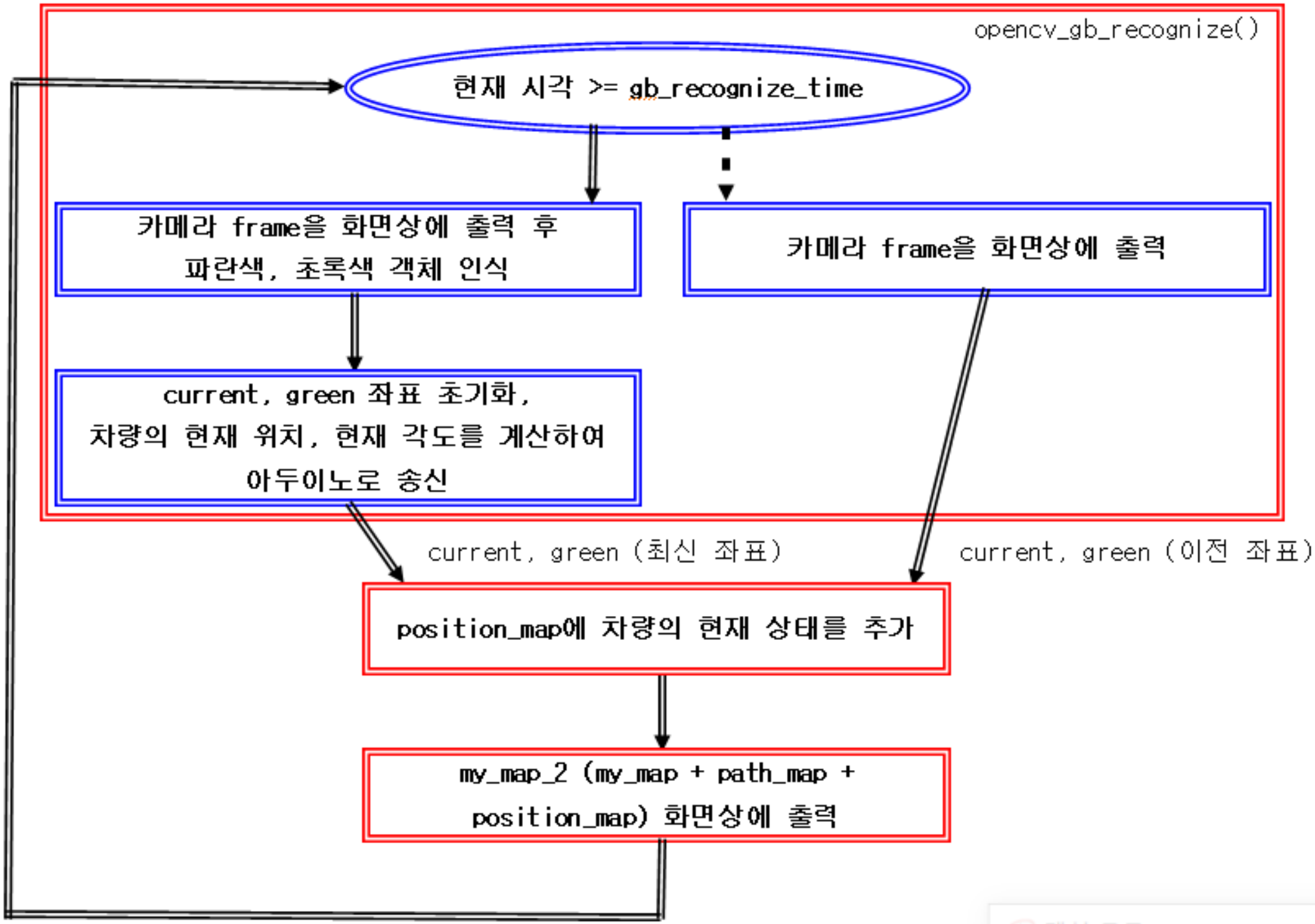
따라서 노트북이 차량이 이동할 때 차량의 현재 좌표로부터 도착 좌표까지의 최단 경로 좌표를 실시간으로 송신할 수 없으므로 차량이 이동하기 전에 시작 좌표에서 도착 좌표까지의 최단 경로 하나만을 아두이노로 송신한다.

또한, 차량이 이동할 때 장애물의 현재 좌표가 실시간으로 변화할 수도 없게 된다.

코드의 동작 순서는 다음과 같다.


마우스 좌클릭한 좌표를 통해
start, goal 좌표 초기화,
my_map에 start, goal 좌표를 추가
opencv_y_recognize()
현재 시각 >= y_recognize_time
카메라 frame을 화면상에 출력 후
노란색 객체 인식
카메라 frame을 화면상에 출력
obstacle_positions_list (장애물 좌표 배열)
obstacle_positions_list (빈 배열)
obstacle_positions_list ≠ 빈 배열
my_map에 장애물 좌표를 추가,
my_map 화면상에 출력


my_map을 이용해 cost_map 계산
A* 알고리즘으로 시작 좌표에서 도착 좌표까지의 최단 경로 좌표 계산
path_map에 최단 경로 좌표 추가
최단 경로 좌표를 일정 시간 간격으로 아두이노로 송신


opencv_gb_recognize()
현재 시각 >= gb_recognize_time
카메라 frame을 화면상에 출력 후
파란색, 초록색 객체 인식
카메라 frame을 화면상에 출력
current, green 좌표 초기화,
차량의 현재 위치, 현재 각도를 계산하여
아두이노로 송신
current, green (최신 좌표)
current, green (이전 좌표)
position_map에 차량의 현재 상태를 추가
my_map_2 (my_map + path_map +
position_map) 화면상에 출력

# 최단 경로 추종 차량