2019학년 창업연계 공학설계 입문 보고서

자율 주차 시스템

국민대학교 소프트웨어융합대학

소프트웨어학부

4분반 4조

2019년 12월 19일

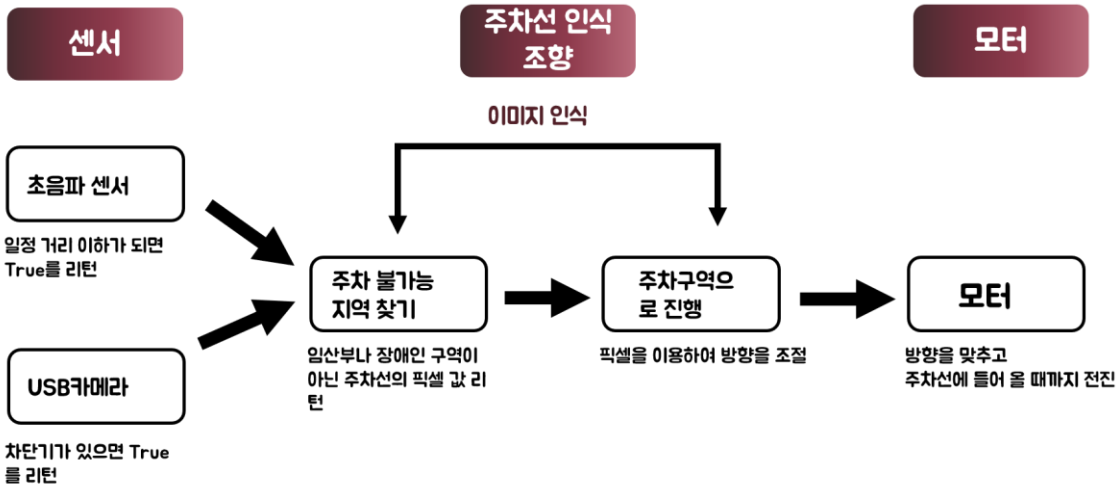
1.주제 선정 이유

이 주제를 선택한 이유는 AD프로젝트를 진행하기 위해서 여러가지 주행에 대해서 찾아보다가 주차사고에 대한 뉴스를 우연히 접했다. 이 뉴스에서 말하고자 하는 바는 주차를 할 때 발생하는 사고가 교통사고에 30%나 된다는 것이었고 이 뉴스를 통해서 주차가 매우 어려운 것을 알게 되었다. 또한 많은 사람들이 자율 주행이라는 말을 들었을 때 떠오르는 이미지는 대부분 도로 위를 달리거나 주변 자동차들과 일정한 거리 유지하면서 안전하게 주행하는 것을 떠올릴 것이다. 하지만 모든 주행의 끝은 주차다라는 말이 있듯이 주차는 매우 중요하다. 그리고 주차가 전체 교통사고의 비율에 30%나 차지했을 뿐만 아니라 주차는 초보운전자들이 가장 어렵게 느끼는 것 중에 끼어들기를 제치고 1위를 할만큼 매우 어려운 문제이다. 그렇기 때문에 안전한 주차 시스템이 구현되어야 자율 주행이 비로소 완성된다고 생각했기에 주제를 선택하게 되었다.

2.첫 구상

처음 구상으로는 센서들을 이용한 상황 판단, 주차선 인식과 조향, 마지막으로 주차선으로의 구동까지 크게 총 3개의 부분으로 나누었다. 그리고 다시 센서들은 이용한 판단은 초음파 센서와, USB 카메라 센서를 이용한 차단기 인식으로 나뉘었고, 주차선 인식과 조향은 주차를 하지 못하는 예를 든다면 임산부 지역이나 장애인 주차구역을 판단하는 알고리즘과 주차구역의 판단이 끝났으면 그곳으로 가게 방향을 정하는 알고리즘으로 나누었다.

처음으로 초음파 센서는 일정 값 이하가 되면 장애물 혹은 주차가 완료가 된 것임으로 일정 값 이하가 되면 True리턴하면 모터가 전진 혹은 후진을 하지 못하게 했다. 그 다음으로는 주차를 하러 들어가는 지역에는 주차를 위한 공간을 알리기 위해서 주차 차단기가 있는데 USB카메라를 이용해서 차단기를 인식하여 주차 지역으로 들어 갔음을 알게 하고 차단기가 있으면 모터의 구동에 제어를 걸었다. 그 다음으로는 주차 지역으로 주차 불가능한 지역을 찾고 가능한 지역의 차선 픽셀 값을 주어 이 픽셀에 따라서 방향을 정하게 했다. 마지막으로 앞에 센서들의 값과 알고리즘을 이용해서 주차선의 들어가도록 모터를 제어했다.



첫 구상의 이미지화

3. 각 부분별 초반 구상

1. 초음파 센서

장애물 인식위해서 초음파 센서를 썼다. 그리고 초음파 센서를 이용한 상황 판단 코드에는 초음파 센서가 불안정하여 튀는 값이 많이 나왔기 때문에 이를 처리하기 위한 필터 코드와 초음파 데이터를 받아서 장애물이 있으면 True를 반환하는 코드 이렇게 두 가지로 나누었다.

```
class MovingAverage:
    def __init__(self, n):
        self.samples = n
        self.data = []
        self.weights = list(range(1, n + 1))

    def add_sample(self, new_sample):
        if len(self.data) < self.samples:
            self.data.append(new_sample)
        else:
            self.data = self.data[1:] + [new_sample]
        # print("samples: %s" % self.data)

    def get_mm(self):
        return float(sum(self.data)) / len(self.data)
```

먼저 필터 모듈은 필터를 만들 때 필터 값을 어떻게 구할 것인지를 생각했는데 대안은 일반 평균과 가중치 평균이 나왔다. 고민 끝에 주행 거리가 길지 않다는 점을 고려해서 일반 평균을 선택했다. 필터를 처음 만들 때 빈 리스트를 만들고 리스트의 길이를 설정하는 인자(n)을 하나 받은 다음에, 리스트의 길이가 n보다 짧으면 계속해서 값을 받고, n을 다 채웠으면 리스트의 인덱스 0번째 값을 버리고 이 후 인덱스에 있는 값들을 한칸씩 전진한 다음에 마지막 인덱스 값에 새로

들어온 값을 계속 추가했다. 그러고 나서 리스트안의 값들을 모두 더해서 평균을 내어 필터 값을 처리했다.

```
    # def drive(angle, speed):
    #     global motor_pub
    #     drive_info = [angle, speed]
    #     pub_data = Int32MultiArray(data=drive_info)
    #     motor_pub.publish(pub_data)

    def callback(self, data):
        global usonic_data
        usonic_data = data.data
```

```
        rate = rospy.sleep(10)
        self.ma = MovingAverage(10)
        while len(self.ma.data) <= 10:
            self.ma.add_sample(usonic_data[1])
            rate.sleep()

    def exit_node(self):
        return
```

그 다음으로 장애물을 인식하는 모듈은 위에 만든 필터에 10의 인자를 주어서 만들었고 그 다음부터 리스트의 값을 채우기 위해 첫 10번에 초음파 센서로부터 데이터를 받아와서 필터 리스트에 값을 추가했다.

```
def obstruction_detect(self):
    rate = rospy.Rate(10)

    while True:
        if self.ma.get_mm() - 5 < usonic_data[1] < self.ma.get_mm() + 5:
            self.ma.add_sample(usonic_data[1])

        else:
            self.ma.add_sample(self.ma.data[-1])

        rate.sleep()
        print(self.ma.data)

        if usonic_data[1] < 30 and (self.ma.getmm - 5 < usonic_data[1] < self.ma.getmm + 5):
            return True
```

그 후 실시간으로 초음파 데이터를 받아오면서 평균값보다 5 초과로 차이가 나는 경우를 튀는 값으로 예상한 다음, 튀는 값이 나올 경우 튀는 값이 아닌 그 전 데이터인 리스트의 맨 마지막 인덱스의 값을 넣어주었습니다. 이렇게 튀는 값을 제거하여 이 후 모터 구동에서 차량이 갑작스럽게 멈추는 일을 없게 해주었다. 그리고 목표인 장애물과의 거리가 30센치 이하인 경우 True를 반환하도록 했다.

2. USB카메라 센서를 이용한 차단기 인식

차단기를 인식하기 위해 생각했던 첫 방법은 이미지 인식하는 3rd party library 인 yolo를 이용하는 것이었다. 하지만 소프트웨어적인 문제, Open CV 버전 충돌, 오픈소스 사용 미숙같은 여러 제약사항 때문에 yolo 사용하는 것을 포기하고, 다른 방법을 사용했다.

다른 대안 방안은 차단기의 색깔을 먼저 인식하여 값을 받아 놓고 xycar가 돌아가면서 찾은 이미지와 비교하는 방법이다. 이 방안은 총 3단계로 나누어 지는데 첫번째로 관심 영역을 설정하는 것이다. 차단기와 30cm 정도로 근접할 때, 관심영역에 차단기가 들어오도록 관심영역을 정했다. 두번째로 차단기의 색 평균값을 설정하는 것이다. 차단기는 동일한 패턴이 반복되기 때문에 차단기 영역의 전체 부분의 색의 평균값을 구했다. 마지막으로 관심영역 내부의 색 평균값을 프레임별로 계산했다. 그 후 미리 정한 색 평균값과 관심영역 내부의 색 평균값을 프레임별로 비교하면서 오차범위 내로 들어오는 경우에 차단기로 인식하게 했다.

```
roi = img[95:98 , 200:553]# 차단기를 인식할 범위
std_color = (129, 121, 214) # 차단기 색의 평균값
isbreaker = False #차단기 인식 여부
blue = 0
green = 0
red = 0
num_pixel = 4 * 353 #roi 픽셀수
```

초기설정에서 관심영역, 차단기의 색 평균값, 차단기 인식여부, for 문을 돌면서 b, g, r 값을 저장할 변수를 선언.

```
#roi의 색 평균값 구하기
for i in range(95,99):
        for j in range(200,554):
                blue +=  img[i][j][0]
                green += img[i][j][1]
                red += img[i][j][2]

blue_avg = blue / num_pixel
green_avg = green / num_pixel
red_avg = red / num_pixel
```

프레임 마다 loop를 돌면서 관심영역의 b,g,r 값을 각각의 변수에 모두 더하고, 전체 픽셀만큼 나눠서 평균 색 값을 구한다..

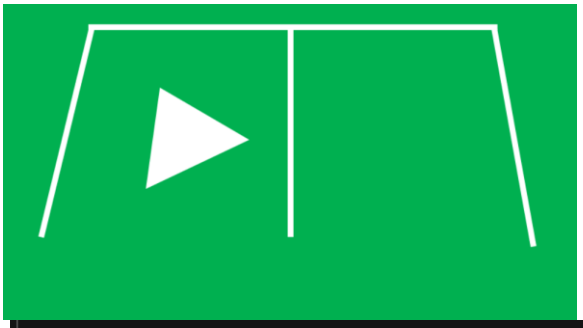
```
#roi의 색 평균값과 기준으로 잡아놓은 색의 값 비교 (오차범위 내에 들어오는지)
if  (std_color[0] - 10 < blue_avg < std_color[0] + 10) and (std_color[1] - 10 < green_avg < std_color
[1] + 210) and (std_color[2] - 10 < red_avg < std_color[2] + 10) :
        isbreaker = True

return isbreaker
```

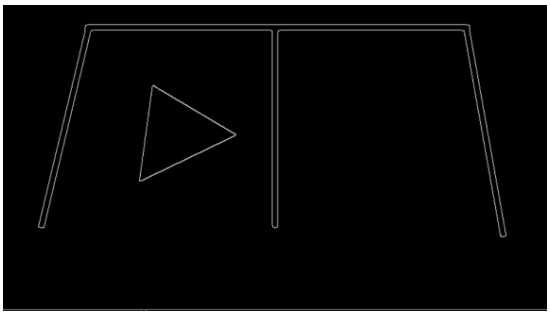
초기에 설정한 색의 값과 loop를 돌면서 구한 값을 비교하여 오차범위 이내로 들어오는 경우 true 반환.

3. 주차 불가능 지역 찾기

주차 불가능한 지역 찾기는 총 6 가지 부분으로 나뉜다.

처음으로는 카메라로 받아드린 이미지에서 윤곽선을 찾아 그 이미지로 변환하는 것이다.

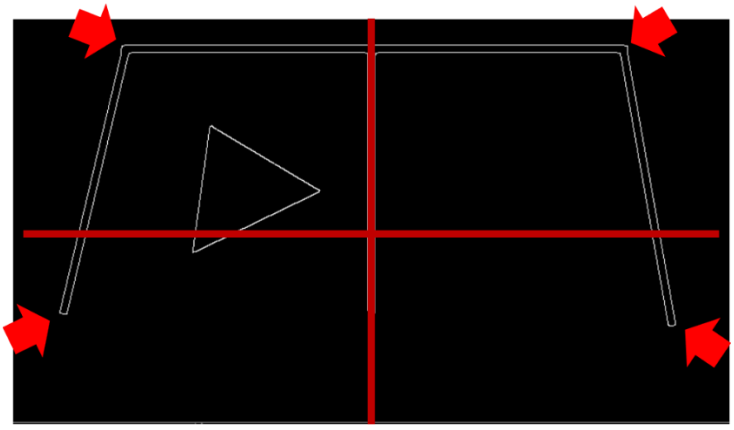




```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (5, 5), 0)

edge = cv2.Canny(blur, 60, 150)
```

2 번째로는 주차 구역이 시작과 끝 부분을 찾기는 것이다.

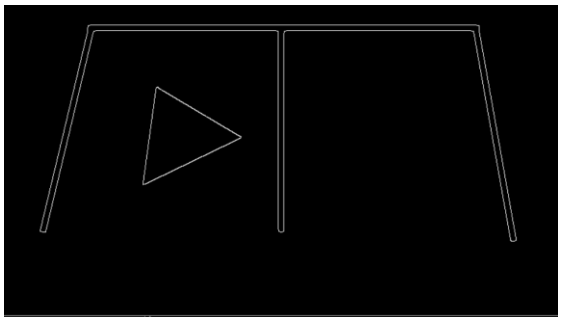
영역을 먼저 4 부분으로 나누고 for 문을 이용하여 주차선이 시작되는 끝점을 찾고 그 점들의 위치를 리스트로 만들어서 나중에 이미지 변환을 할 때 쓰일 리스트에 값으로 저장한다.

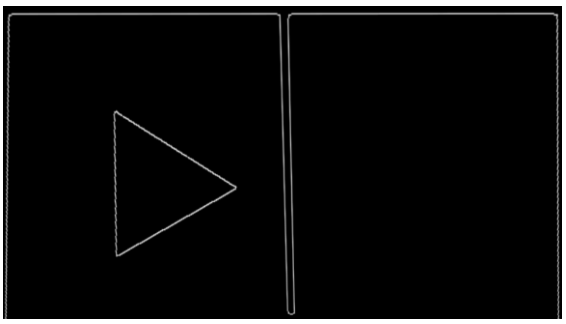


```
for h in range(height // 2):
    for w in range(weight // 2):
        if edge[h, w] == 255:
            point.append([w, h])
            break
    if len(point) != 0:
        break
print(point[0])
```

끝점 찾기 중 왼쪽 위 코드

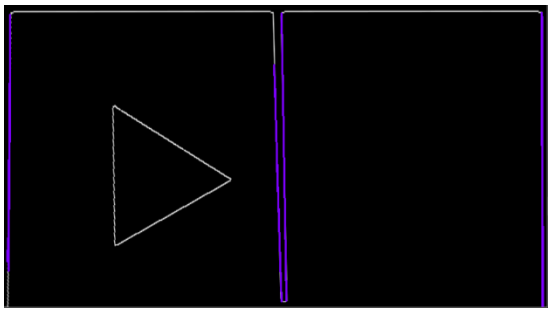
3 번쨰는 앞에 과정에서 찾은 4 개의 점이 들어가 있는 리스트를 이용해서 차선을 수직선에서 본 것처럼 이미지를 전환했다.





```
src = np.float32([point[0], point[1], point[2], point[3]])
dst = np.float32([[0, 0], [800, 0], [0, 450], [800, 450]])

M = cv2.getPerspectiveTransform(src, dst)
warp = cv2.warpPerspective(edge.copy(), M, (800, 450))
```

4 번 째는 확률적 허프 변환으로 주차 차선 찾기이다. 앞에서 변환한 이미지르 open 내장 함수인 확률적 허프 변환을 이용하여 변환한 이미지에서 가로 줄을 찾는다.



```
lines = cv2.HoughLinesP(warp, 1, np.pi/180, 200, 200, 20)

x_list = []
for line in lines:
    for li in line:
        x1, y1, x2, y2 = li
        if abs(y2 - y1) > 10:
            if abs(x2 - x1) < 10:
                x_list.append(x1)
x_list.sort()
```

5 번째와 6 번째는 찾은 차선 안에 이미지가 있는지를 확인하고 이미지가 없는 지역에 픽셀 값을 반환하는 것이다.

차선안에 이미지가 있는 지를 확인하기 위해서 위에서 찾을 찾을 때 x 값을 모두 리스트에 저장해 놓고 리스트를 정렬한다. 이후 정렬한 값들 중에서 필요한 값들만 새로운 리스트에 추가하고 변환한 이미지에서 새로운 리스트에 들어있는 x 값들을 기준으로 흰색 점 계수로 이미지가 있는지 업는지를 확인했다. 그리고 그 값들이 일정 이하면 리스트에 들어 있는 값들을 반환했다.

```
for i in range(len(x_last_list) -1):
    roi = warp[:, x_last_list[i]:x_last_list[i + 1]]
    if cv2.countNonZero(roi) < 5000:
        print(x_last_list[i], x_last_list[i + 1])
```

4. 주차 구역으로 진행

```
#수평까지 이동 ( 원래 코드에서 왼/오 canny값 대신 받아와준 픽셀값 사용 )
    def gotoplace(self,pixel_left,pixel_right, mid):
        self.pixle_left= pixel_left
        self.pixle_right= pixel_right

        mid = (pixel_right + pixel_left) // 2

        if mid < 280:
            angle = -40
        elif mid > 360:
            angle = 40
        else:
            angle = 0
        return angle
```

주차 불가능 지역을 찾는 코드에서 받은 픽셀을 이용해 xycar 를 이동시켰다. 이전에 배웠던 자율주행 코드에 이용한 원리처럼 주차 라인의 픽셀을 받아오게 될 시, 양쪽 라인의 값 대신 받아온 양 쪽 픽셀값을 바탕으로 조향각, 즉 angle 값을 설정해 주어서 주차 공간 안으로 주행하게 해주었다.

```
#받아온 왼/오 canny와 인식한 왼/오 픽셀값이 일치 시 true를 리턴
        if left == pixel_left and right == pixel_right
            return True
```

카메라 영상을 통해 계속해서 받아오는 양 쪽 픽셀값과 이전 단계에서 받아온 올바른 주차 공간의 양 쪽 픽셀값이 일치하게 되면 xycar 가 주차 공간과 수평이 된 것이므로 True 를 반환해준다.

```
#ultrasonic에서 true 리턴해주면 운전 멈추도록
def stopsign(self, obstacle, recognize):
    obstacle = self.obstacle
    recognize = self.recognize
    if obstacle == recognize == True:
        speed = 0
        return speed
```

마지막으로 수평이 된 상태에서 계속 직진해서 주차 구역 안으로 진입하고, 만약 앞에서 이용한 초음파 센서 코드에서 주차막과의 거리가 일정 이하가 되어 True 를 반환하게 된다면 xycar 를 멈추고 주차를 완료했다.

4. 한계점 및 해결 방안

각 모듈별로 통합하는 과정이나 코드를 작성해 나가는 과정에서 발견 혹은 생긴 오류에 대한 한계점이 있었다.

처음으로 센서를 이용한 상환 판단에서 초음파 센서 코드는 통합하는 과정에서 필터가 필요없다고 판단되어서 제대로 사용하지 못했다. 그렇기 때문에 오차값이 5 가 아니라 정확한 값을 찾기 위해 계속적으로 시도해보지 못하여서 제대로 작동하는지에 대해서는 잘 모르게 되었다는 점이 아쉬운 부분으로 남았다. 또한 차단기 인식에서는 세상에는 정말 다양한 형태의 차단기가 존재하기 때문에 고안해낸 방법이 한가지 종류의 차단기만을 인식할 수 있다는 한계점을 가지고 있었다. 이를 해결하기 위해서 두 센서들을 동시에 이용하여 서로 상호 보완되게 했다.

만들어진 이미지로 테스트를 해보았을 때는 이미지에 다른 사물들이 없었기 때문에 차선 인식이 매우 잘 되었다. 하지만 실제 상황에서는 주변에 많은 사물들이 있기에 영역을 잘 자르지 않으면 차선 인식을 정확히 하는 것이 어려웠다. 그렇기 때문에 차선을 인식해서 변환하는 이미지를 얻기 힘들었고 이로 인해 가로로 된 주차선을 찾을 수 없었다. 또한 가로선을 찾을 수 없었기 때문에 정확인 픽셀 값을 넘겨주지 못했다. 이것을 해결하기 위해 직선이 있는 이미지 하나를 받은 다음에 직선의 기울기를 이용해서 고정된 값으로 이미지를 변환했다. 또한 주차 지역이 몇 개 있는지를 찾아서 그 개수에 맞게 영역을 자른 다음에 범위에 맞게 흰색 픽셀 수를 검사했다. 마지막으로 픽셀을 넘겨주는 것 대신해 방향을 넘겨주게 되었다.

트랙 위에서 주행하는 경우가 아닐 때에는 라인을 인식하지 못해 앞에 과정에서 차선과 주차 구역을 비교 후 True 를 반환해주지 못한다. 그렇기 때문에 트랙 위에서 주행하는 경우엔 트랙의 차선을 인식하고, 트랙이 아닌 곳에서 주행하는 경우엔 받아올 라인이 없어 직진하는 경우에

주차 구역의 픽셀을 받아오는 등 지정해준 if 문이 만족되지 않는 상황에선 직진하도록 수정하였다.

앞 과정에서 픽셀 값을 넘겨주는 것이 불가능해 올바른 주차 구역의 방향을 넘겨주는 것으로 코드를 수정했다. 픽셀 값이 아닌 주차 구역의 방향을 받아오기 때문에, 자율적인 xycar 의 판단만으로는 완벽한 주차를 완성하기에 어렵기에 여러 번 시도를 해보면서 방향을 반환했을 때 원하는 차선으로 가게 만들었다.