

# Mini Project 3

차량 운전자를 위한 횡단보도 무단횡단자 추적 알고리즘

- 보행자 검출, 신호등 검출, 객체 추적

## Contents.

---

- 1 보행자 검출
- 2 신호등 검출
- 3 객체 추적
- 4 Q&A



**Part**

**1**

박두레

## 1. 보행자 검출

### 1. HOG - SVM 보행자 검출

1. SVM과 HOG
2. HOG와 SVM을 이용한 보행자 인식
3. 디폴트 보행자 검출기와 다임러 보행자 검출기

### 2. MOG 배경 제거를 이용한 보행자 검출

1. 객체 추적(Object Tracking)
2. 동영상 배경 제거
3. MOG2-1, MOG2-2 검출 결과 영상

### 3. 영상 종류에 따른 네 가지 검출기 성능 비교

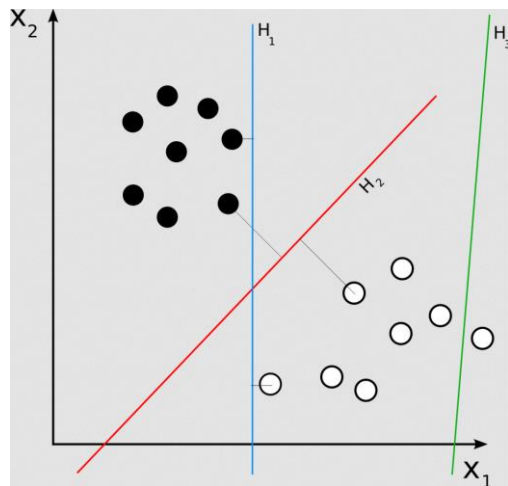
1. 고정된 영상에서 사람만 움직일 때
2. 고정된 영상에서 사람과 차량(사물)이 동시에 움직일 때
3. 움직이는 영상에서 사람과 사물이 움직일 때

### 4. 보행자 인식과 검출: 결론

# 1.1 HOG - SVM 보행자 검출

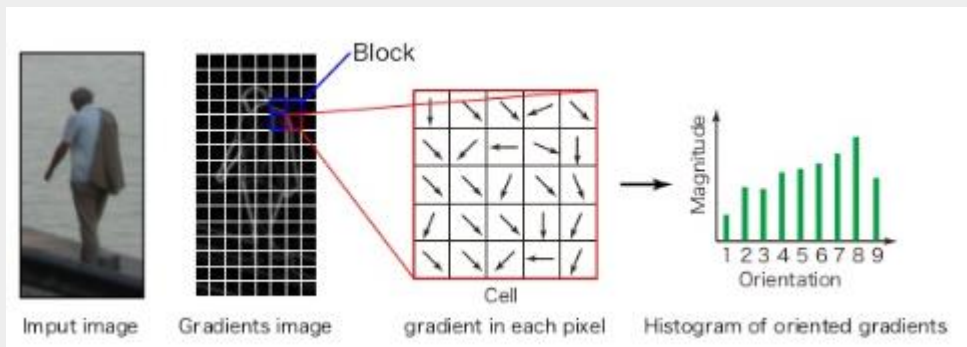
## 1.1.1 SVM과 HOG

- SVM(Support Vector Machine): 기계 학습 분야 중 하나로 패턴 인식, 자료 분석을 위한 지도 학습 모델이며, 주로 분류와 회귀 분석을 위해 사용. 훈련 데이터를 기반으로 데이터 집합을 두 그룹으로 분류하는 대표적인 분류 알고리즘.



- 픽셀 데이터 그 자체를 훈련 데이터로 사용하기도 하지만 픽셀 데이터에서 여러 가지 특징 디스크립터를 추출해서 훈련 데이터로 사용할 수도 있는데, 객체의 지역적 특성이 아닌 전체적인 특성을 표현하는 HOG 디스크립터를 사용하면 객체의 형태를 일반화할 수 있어서 상태나 자세가 조금씩 다른 객체를 인식하는 데 SVM과 HOG를 함께 사용하는 경우가 많다(이세우 2019, 398).

- HOG(Histogram of Oriented Gradient) 디스크립터: 객체 검출을 목적으로 컴퓨터 비전 및 이미지 처리에 사용되는 특징 디스크립터로, 엣지의 기울기 방향과 규모를 히스토그램으로 나타낸 것. HOG는 지역적인 특징보다는 전체적인 모양 표현에 적합. 대상 객체의 상태나 자세가 다르더라도 그 특징을 일반화해서 같은 객체로 인식하는 것이 특징(이세우 2019, 403).



계산 과정: 일정 크기의 셀로 분할한 뒤 기울기(gradient)의 크기(magnitude)가 일정 값 이상인 픽셀들을 방향에 따라 히스토그램으로 표현  
출처: 166.104.231.121/ysmoon/mip2017/lecture\_note/제6장-추가.pdf

## 1.1.2 HOG와 SVM을 이용한 보행자 인식

- 객체를 인식할 때 학습한 객체와 인식하려는 객체의 크기 변화와 회전 발생 여부를 항상 고려해야 함. 이에 윈도우(window) 크기를 달리하면서 검출 작업을 반복해야 함. 학습을 위해 긴 시간과 컴퓨팅 파워 필요(이세우 2019, 409).
- OpenCV는 보행자 인식을 위해 미리 훈련된 API를 제공
- cv2.HOGDescriptor 클래스: HOG 디스크립터 계산, 훈련된 SVM 모델을 전달받아 객체 인식, 이미 훈련된 SVM 모델 제공
- 디폴트 보행자 검출기와 다임러 보행자 검출기

## 1.1.3 디폴트 보행자 검출기와 다임러 보행자 검출기

- 기본 검출기: 불필요한 검출이 적은 대신 멀리 있는 작은 보행자는 검출하지 못함 (사진 왼쪽)
- 다임러 검출기: 작은 보행자뿐만 아니라 다른 사물이나 그림자도 보행자로 인식함 (사진 오른쪽)





## 1.2 MOG 배경 제거를 이용한 보행자 검출

### 1.2.1 객체 추적(Object Tracking)

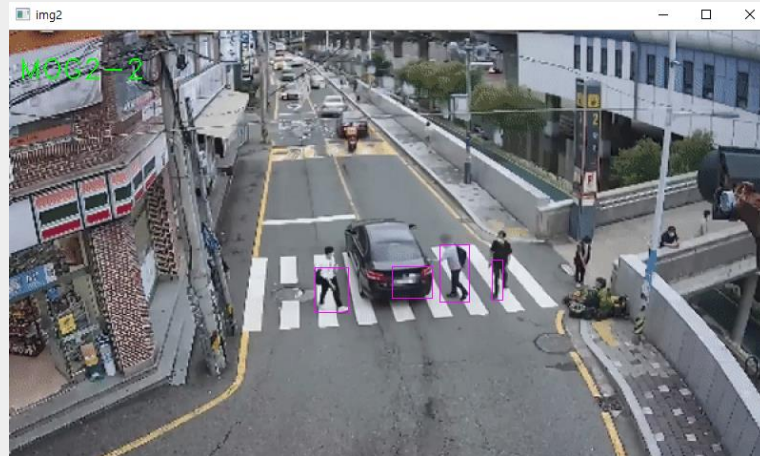
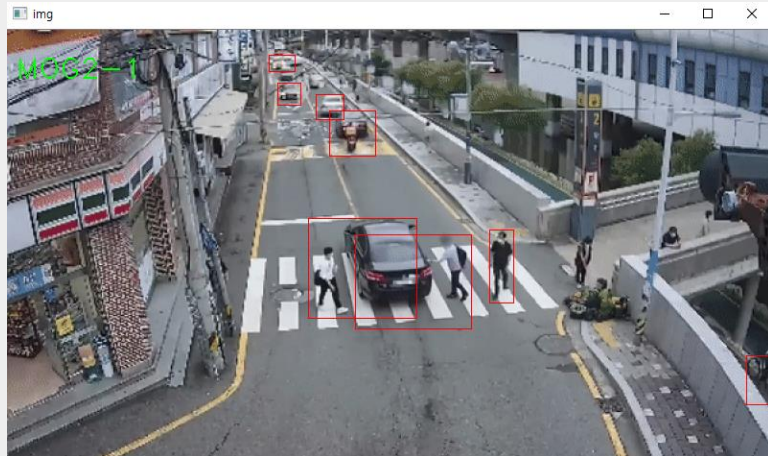
- 객체 추적이 필요한 이유: 매 장면에서 원하는 객체를 찾아낼 수 있음. 그러나 객체 검출이나 인식은 많은 자원을 필요로 하고 속도가 느리기 때문에 빠르고 단순한 객체 추적 방법이 필요. 또한 객체 추적은 객체 검출이나 인식에 실패했을 때 좋은 대안(이세우 2019, 353).
- 객체 추적 알고리즘에 따른 OpenCV 구현 함수 이용

## 1.2.2 동영상 배경 제거

- 입력 영상에서 객체를 추출하기 위해 배경을 모델링하여 전경만을 추출하는 GMM(Gaussian Mixture Models) 기법을 이용하는 MOG(Model of Gaussian) 알고리즘 이용. GMM을 이용하여 모델링 된 배경을 제거하고 움직임 영역만을 추출.
- OpenCV는 동영상에서 배경을 제거하는 알고리즘을 하나의 인터페이스로 통일하기 위해 `cv2.BackgroundSubtractor` 추상 클래스를 만들고 이것을 상속받은 10가지 알고리즘 구현 클래스를 제공(이세우 2019, 353).
- 지브코비치(Zivkovic) 알고리즘을 구현한 `BackgroundSubtractorMOG2` 함수 사용: 각 픽셀의 적절한 가우시안 분포 값을 선택하므로 빛에 대한 변화가 많은 장면에 적합(이세우 2019, 356).

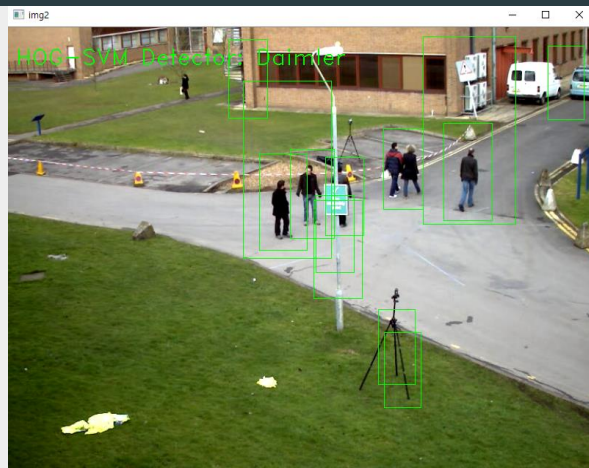


## 1.2.3 MOG2-1, MOG2-2 검출 결과 영상



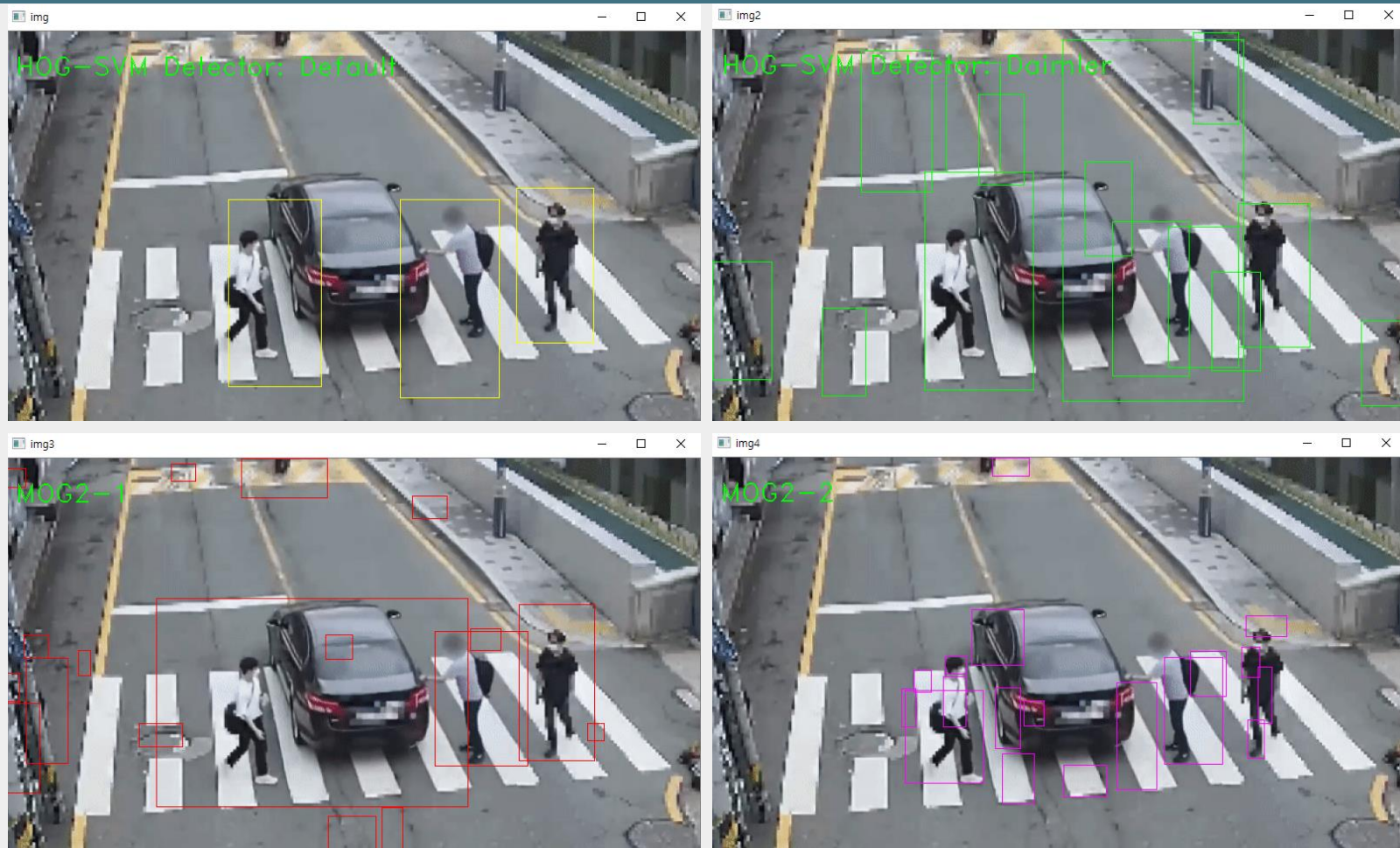
## 1.3 영상 종류에 따른 네 가지 검출기 성능 비교

### 1.3.1 고정된 영상에서 사람만 움직일 때



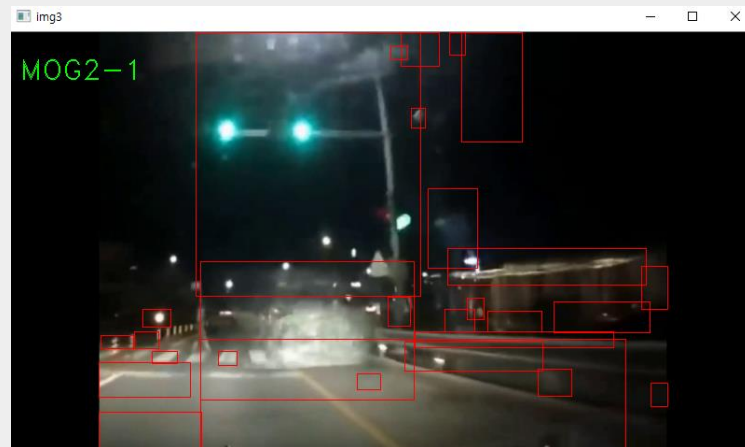
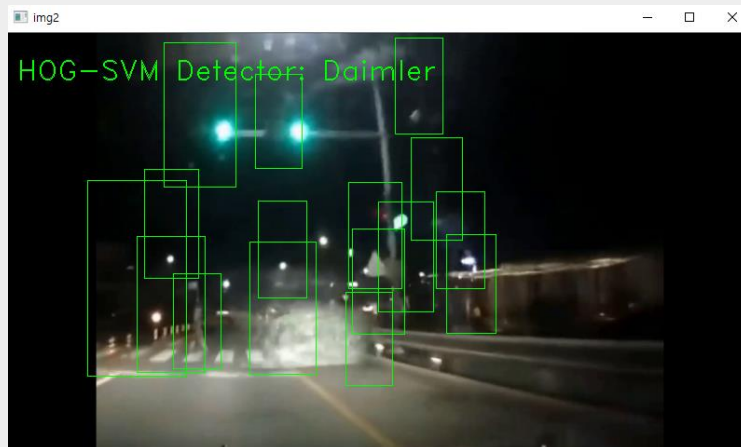
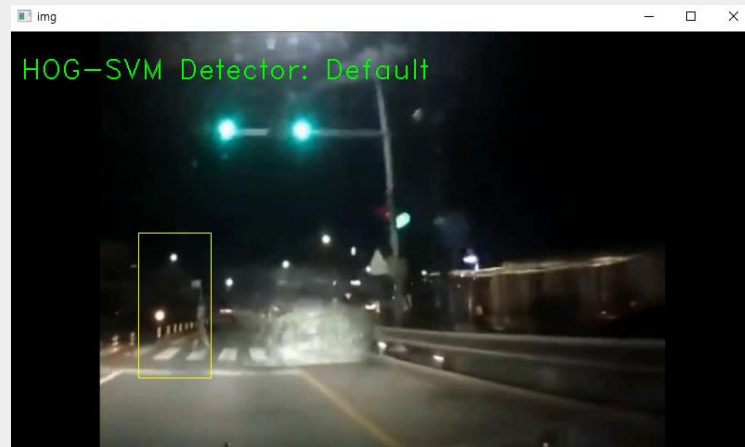


### 1.3.2 고정된 영상에서 사람과 차량(사물)이 동시에 움직일 때





### 1.3.3 움직이는 영상에서 사람과 사물이 움직일 때



## 1.4 보행자 인식과 검출: 결론

- HOG 디스크립터와 MOG 배경 제거를 이용하여 네 가지 보행자 검출기의 성능을 비교
- 고정된 영상에서 사람만 움직일 때, 고정된 영상에서 사람과 차량(사물)이 동시에 움직일 때, 움직이는 영상에서 사람과 사물이 움직일 때로 경우를 나누어 실험
- 본 연구의 수행을 위해 움직이는 영상에서 보행자를 검출해야 하므로 보행자 인식을 위해 미리 훈련되고, 불필요한 검출이 적은 HOG-SVM 기본 검출기가 가장 적합한 검출기로 판단됨
- 이후 이어지는 작업에서 HOG-SVM 디폴트 검출기 이용
- 보행자 검출 정확도가 낮다는 한계
- 연산 처리 속도 향상과 객체 추적률 향상, 불필요한 영역에서의 검출 감소를 통해 정확도를 향상시키는 추가적인 연구 필요

**Part**

**2**

오재동

## 2. 신호등 검출

### 2\_0. 신호등 판별 알고리즘

#### 2\_1. 교통 신호등 후보 영역 검출

2\_1\_1. BGR 색상 모델 -> HSV 색상 모델

2\_1\_2. ROI 영역 설정

2\_1\_3. 녹색 영역 추출

#### 2\_2. 신호등 배면판 후보군 검출

2\_2\_1. BGR 색상 모델 -> GRAY 색상 모델

2\_2\_2. 모폴로지 연산

2\_2\_3. SimpleBlobDetector를 이용한 배면판 후보 검출

#### 2\_3. 신호등 후보군 검출

2\_3\_1. 교통 신호등 & 신호등 배면판 후보군 검출 결과

2\_3\_2. 검출 영역 위치 확인, 교통 신호등 알림

2\_3\_3. SimpleBlobDetector 파라미터 설정 값

## 2.0 신호등 판별 알고리즘

### 1. 교통 신호등 후보 영역 검출

-> 교통신호등의 경우 스스로 빛을 발하기 때문에, 신호등의 적색, 황색, 녹색 점등 영역을 추출한다.

- 1) BGR 색상 모델 -> hsv 색상 모델
- 2) ROI 영역 설정 (처리속도 향상)
- 3) 적색, 녹색 영역 추출

### 2. 신호등 배면판 후보군 검출

-> 신호등 배면판의 경우 직사각형 형태의 무광 혹은 흑색 판으로 제작되어 있다.

- 1) 입력 영상으로부터 흑색 영역 추출
- 2) 모폴로지 연산을 통해 배면판 내부 영역 채움 -> 적용 x
- 3) 직사각형 특징을 이용해 배면판 후보 검출

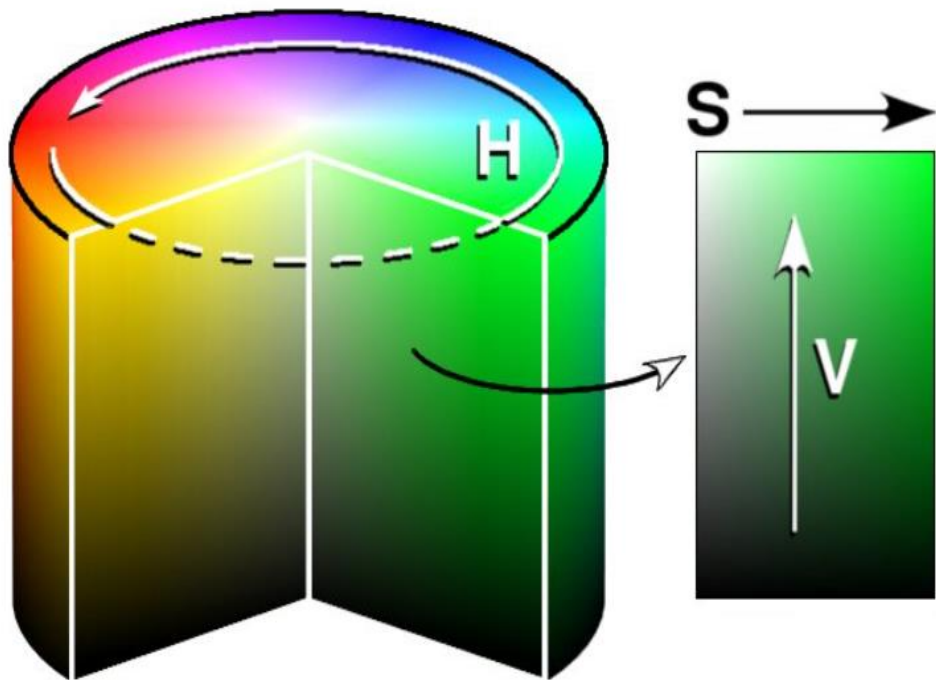
### 3. 신호등 후보군 검출

-> 신호 점등영역 추출 과정을 통해 얻은 영역이 신호등 배면판 후보군 내에 존재하는지 여부 판단



## 2.1 교통 신호등 후보 영역 검출

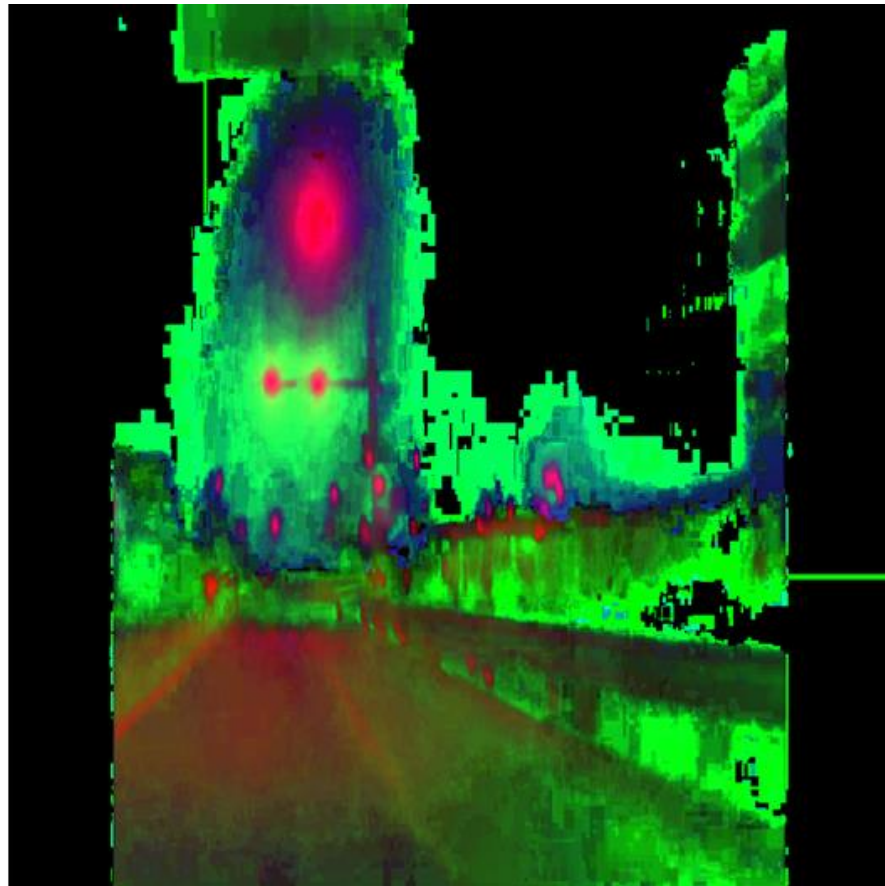
### 2\_1\_1. BGR 색상 모델 -> HSV 색상 모델



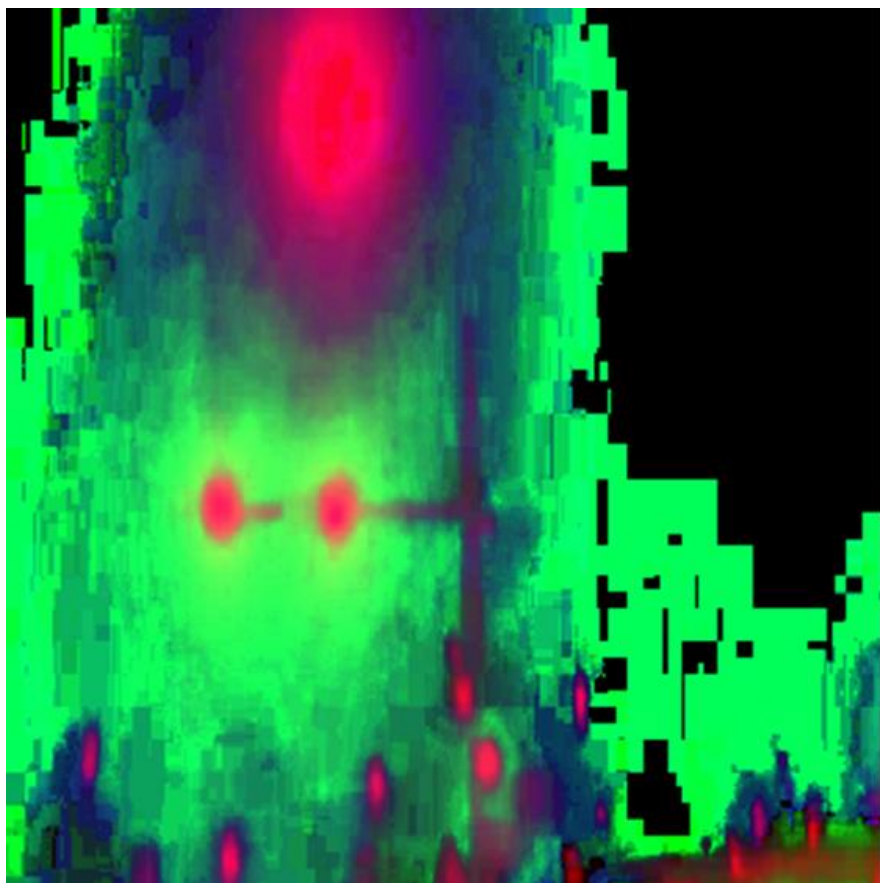
신호등의 적색, 황색, 녹색 점등 영역을 추출하기 위한 과정에서 RGB 색상 모델은 조명에 영향을 많이 받기 때문에, 신호등의 점등 영역을 찾기에 부적합하다.

이에, BGR 색상 모델을 HSV 색상 모델로 변환하여 신호등의 점등 영역을 추출한다.

## 2\_1\_1. BGR 색상 모델 -> HSV 색상 모델 변환



## 2\_1\_2. ROI 영역 설정



```
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)  
cv2.imwrite(data_output_path + 'img_hsv.png', img_hsv)
```

```
roi_c = img_hsv[int((1/5)*w):int((3/5) * w), int((1/5)*h): int((3/5)*h)]
```

[관심 영역]

너비: 전체 이미지의 1/5 ~ 3/5 지점

높이: 전체 이미지의 1/5 ~ 3/5 지점

## 2\_1\_3. 녹색 영역 추출



```
lower_green = np.array([50, 50, 80]); upper_green = np.array([90, 255, 255])  
lower_red = np.array([-10, 30, 50]); upper_red = np.array([10, 255, 255])  
lower_yellow = np.array([11, 50, 50]); upper_yellow = np.array([30, 200, 200])
```

```
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)  
cv2.imwrite(data_output_path + 'img_hsv.png', img_hsv)
```

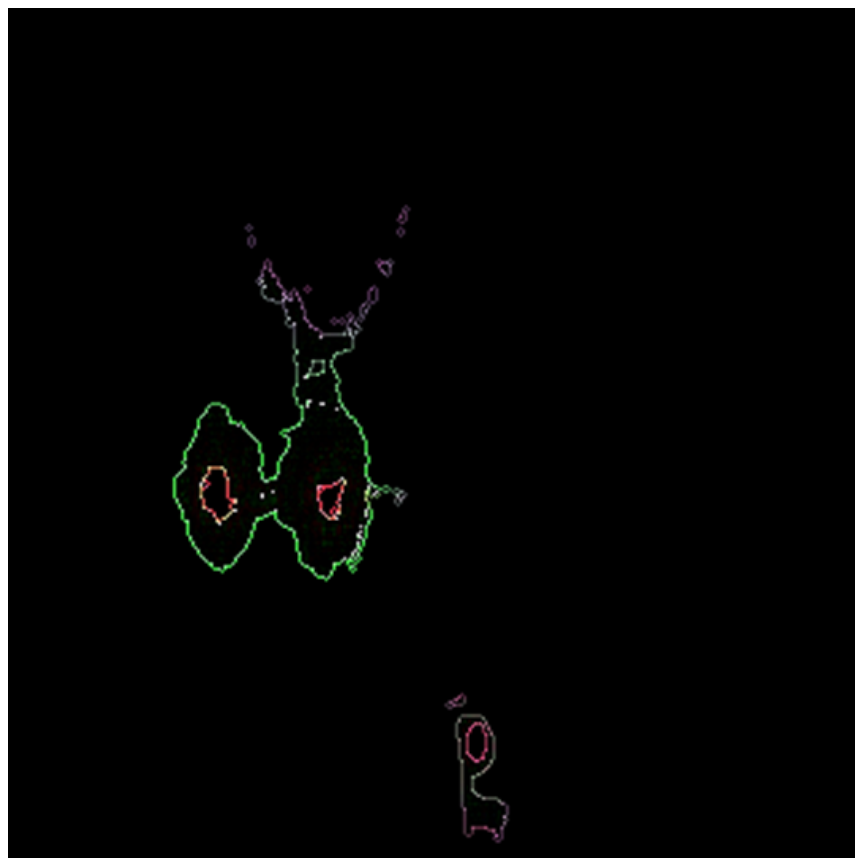
```
roi_c = img_hsv[int((1/5)*w):int((3/5) * w), int((1/5)*h): int((3/5)*h)]
```

```
if color == 'red':  
    mask = cv2.inRange(roi_c, lower_red, upper_red)  
elif color == 'green':  
    mask = cv2.inRange(roi_c, lower_green, upper_green)  
elif color == 'yellow':  
    mask = cv2.inRange(roi_c, lower_yellow, upper_yellow)
```

\* 참고 자료

	H	S	V
적색	0~10 or 150~180	80~255	80~255
황색	11~30	100~255	80~255
녹색	60~100	75~255	80~255

## 2\_1\_3. 녹색 영역 추출

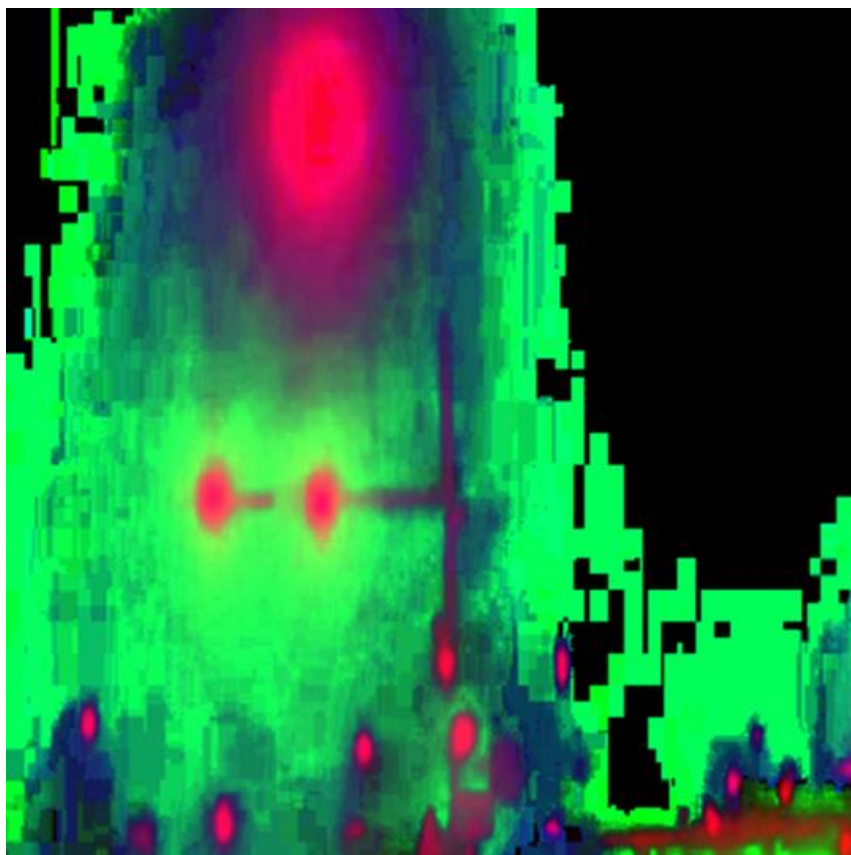


```
img_mask = cv2.bitwise_and(roi_c, roi_c, mask=mask)
cv2.imwrite(data_output_path + 'img_mask.png', img_mask)
edge_c = cv2.Laplacian(img_mask, -2)
```

라플라시안 필터를 적용하여, 경계 검출 수행



### 3\_1\_3. 녹색 영역 추출, SimpleBlobDetector 적용



```
detector = cv2.SimpleBlobDetector_create(param_c)
c_keypoints = detector.detect(edge_c)
img_draw_c = cv2.drawKeypoints(roi_c, c_keypoints, None, None, cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imwrite(data_output_path + 'roi_c.png', roi_c)
```

\* Blob 파라미터 설정 값

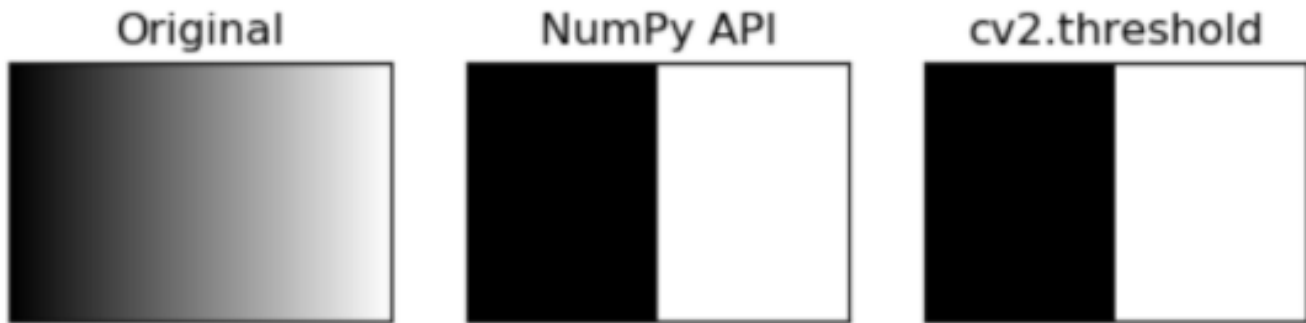
```
def set_blob_param(r_type: str):
    # BLOB 필터 생성하기
    params = cv2.SimpleBlobDetector_Params()

    params.minThreshold = 10
    params.maxThreshold = 255
    params.thresholdStep = 3
    params.filterByArea = False
    params.filterByColor = False
    params.filterByCircularity = True
    params.filterByConvexity = False
    params.filterByInertia = False

    if r_type == 'color':
        # 경계 값 조정
        params.minCircularity = 0.8
        params.maxCircularity = 0.85
```

## 2.2 신호등 배면판 후보군 검출

### 2.2.1 입력 영상으로부터 흑색 영역 추출



스레시홀딩을 적용하여, 입력받은 영상을 바이너리 이미지로 변환

## 2\_2\_1. 입력 영상으로부터 흑색 영역 추출



## 2\_2\_1. 입력 영상으로부터 흑색 영역 추출



가우시안 필터 적용하여 노이즈 제거



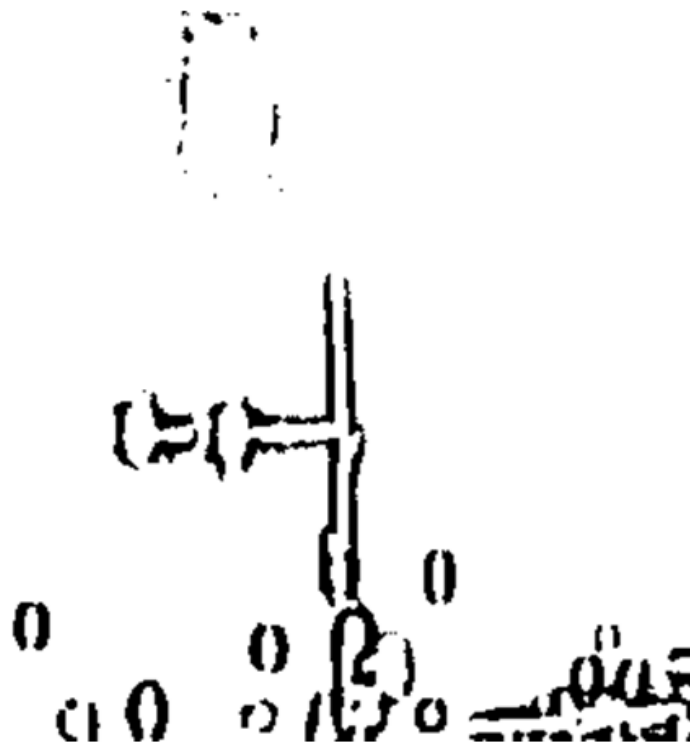
오츠 알고리즘을 사용하여 바이너리 이미지 생성

## 2\_2\_1. 입력 영상으로부터 흑색 영역 추출

\* 다른 스레시홀드 적용한 결과 이미지



직접 범위 지정한 스레시홀드 (55, 255)



적응형 스레시홀드 (Adaptive Thresh Mean)



## 2\_2\_2. 모폴로지 연산을 이용한 배면판 내부 채움 (적용 x)



(a) 흑색 영역에 대한 색 분할

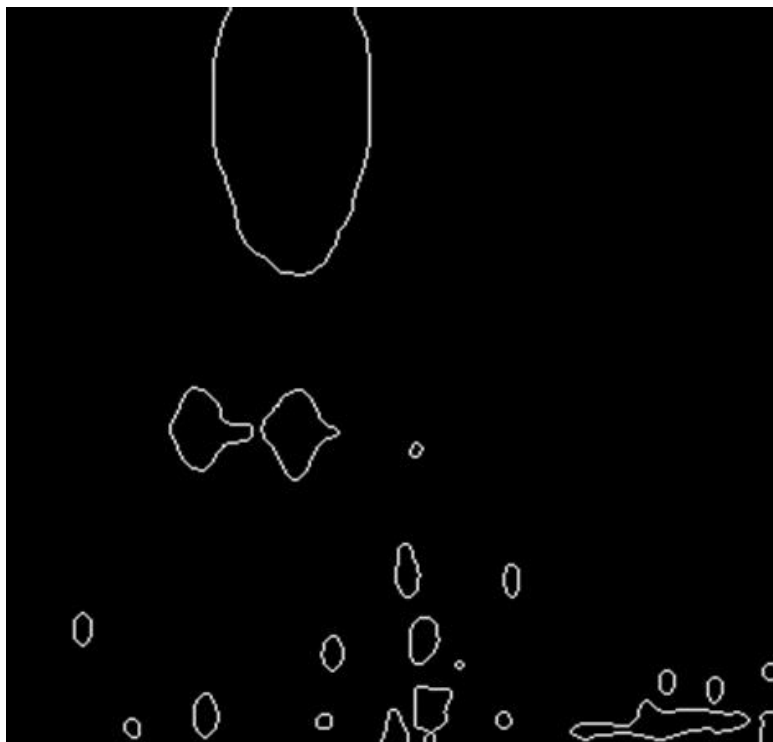


(b) 모폴로지 닫기 연산



\* 참고 자료에서는 모폴로지 닫기 연산을 통해 배면판을 채웠지만, 사용 영상에서는 채워야 할 배면판 영역이 검출되지 않아 적용 x

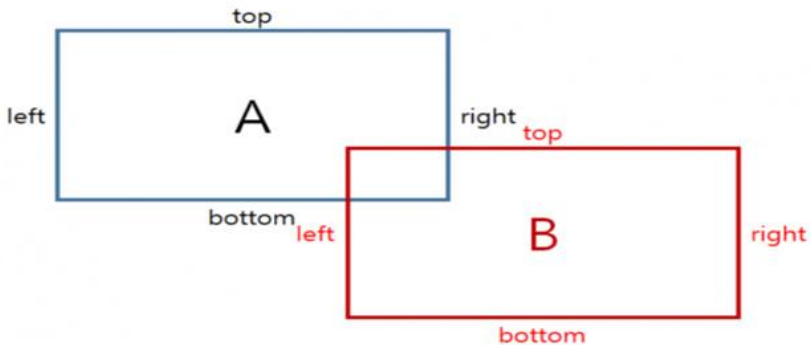
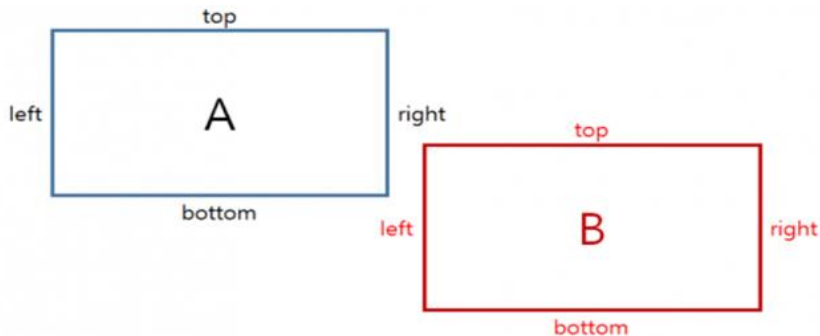
## 2\_2\_3. 직사각형 특징을 이용한 배면판 후보 검출 (Laplacian 필터 -> Blob 필터)



Blob 필터를 적용하여 조건에 맞는 후보군들만 추출

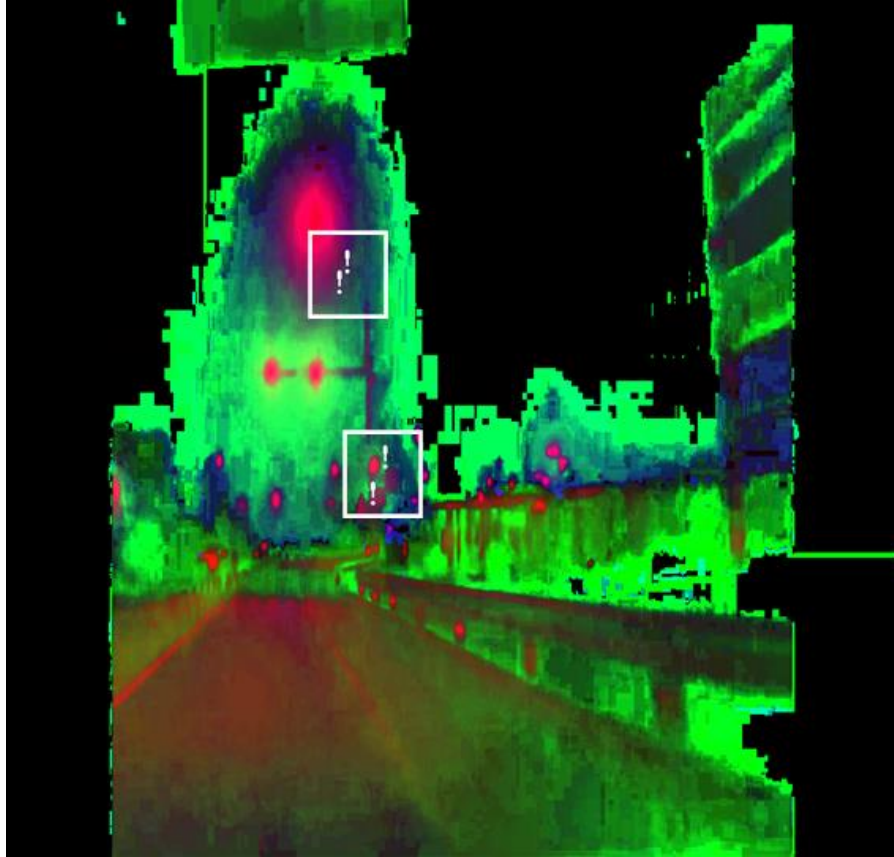
## 2.3 신호등 후보군 검출

### 2\_3\_1. 교통 신호등 & 신호등 배면판 후보군 검출 결과



두 영역의  $x$ ,  $y$ ,  $x_2$ ,  $y_2$  좌표값을 이용해  
겹치는 영역 판단

## 2\_3\_1. 교통 신호등 후보군 & 신호등 배면판 후보군 검출 결과



## 2\_3\_2. 검출 영역 위치 확인, 교통 신호등 알림



```
def check_location(img, shape_list, color_list): # 직사각형 겹치는 좌표 개수 구하기
    global tl_flag, color_flag
    for a1, a2, b1, b2 in color_list:
        for x1, x2, y1, y2 in shape_list:
            if ((a1 >= x1 and a1 <= x2) and (b1 >= y1 and b1 <= y2)) and tl_flag == False:
                cv2.putText(img, f'{color_flag} light Detected !', (a1, b1), cv2.FONT_HERSHEY_TRIPLEX, 1, (255, 255, 255))
                cv2.imwrite(data_output_path + 'light_detected.png', img)
                tl_flag = True
```

- \* 검출한 색상 blob의 x, y 좌표 값이  
형태 blob의 x, y, x2, y2 좌표 값 내부에 있는지를  
판단, 조건을 만족하면 신호등 검출 알림

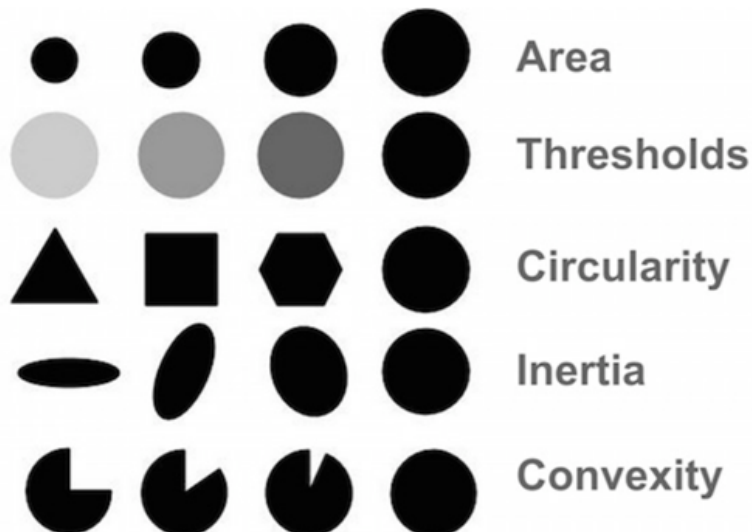


## 2\_3\_3. SimpleBlobDetector 파라미터 설정 값

```
def set_blob_param(r_type: str):  
    # BLOB 필터 생성하기  
    params = cv2.SimpleBlobDetector_Params()  
  
    params.minThreshold = 10  
    params.maxThreshold = 255  
    params.thresholdStep = 3  
    params.filterByArea = False  
    params.filterByColor = False  
    params.filterByCircularity = True  
    params.filterByConvexity = False  
    params.filterByInertia = False  
  
    if r_type == 'color':  
        # 경계 값 조정  
        params.minCircularity = 0.8  
        params.maxCircularity = 0.85  
    elif r_type == 'shape':  
        params.filterByArea = True  
        params.minArea = 100  
        params.maxArea = 250  
  
        params.filterByColor = False  
        params.minCircularity = 0.4  
        params.maxCircularity = 0.8  
  
    return params
```

filterByCircularity = True를 설정하여 Blob 검출

\* 참고 자료



Part

3

신나정

### 3. 객체 추적

#### 3.1. 객체 추적을 위한 전처리 영상 생성

##### 3.1.1. 코드에 사용할 유튜브 전처리

##### 3.1.2. 횡단보도 객체 인식

#### 3.2. 횡단보도 객체인식

##### 3.2.1. 특징 검출을 통한 객체 인식 시행착오

##### 3.2.2. ROIs를 통한 2개 이상의 ROI 객체 검출 및 추적

#### 3.3. 영상 내 횡단보도 보행자 객체 인식

##### 3.3.1. 꼭지점 기준으로 도형이 겹치는 영역 찾기

##### 3.3.2. 예외처리

#### 3.4. 알고리즘 시연

##### 3.4.1. 알고리즘 시연

#### 3.5. 제한점

##### 3.5.1. 제한점

## 3.1 객체 추적을 위한 전처리 영상 생성

### 3.1.1 코드에 사용할 유튜브 영상 전처리

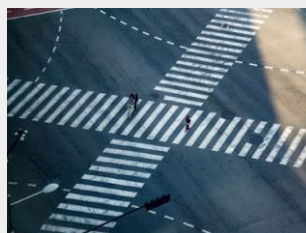
- 관심 영역을 지정하기 위해 영상 내에서 이미지 추출
  - `cv2.imwrite()` 함수를 통해 특정 이미지 추출
- 이후 영상내에서 사용할 관심 영역 지정
  - `cv2.selectROI()` 함수를 사용해 이미지내 특정 범위 좌표 추출
- 영상 ROI 기준으로 크롭된 영상 재생성
  - `cv2.VideoWriter()` 함수로 영상을 만들고 roi 좌표 기준 범위를 특정하고 `out(cv2.VideoWriter 객체).write()` 함수로 영상 저장



## 3.1 횡단보도 객체 인식

### 3.1.2 특징 검출을 통한 객체인식 시행착오

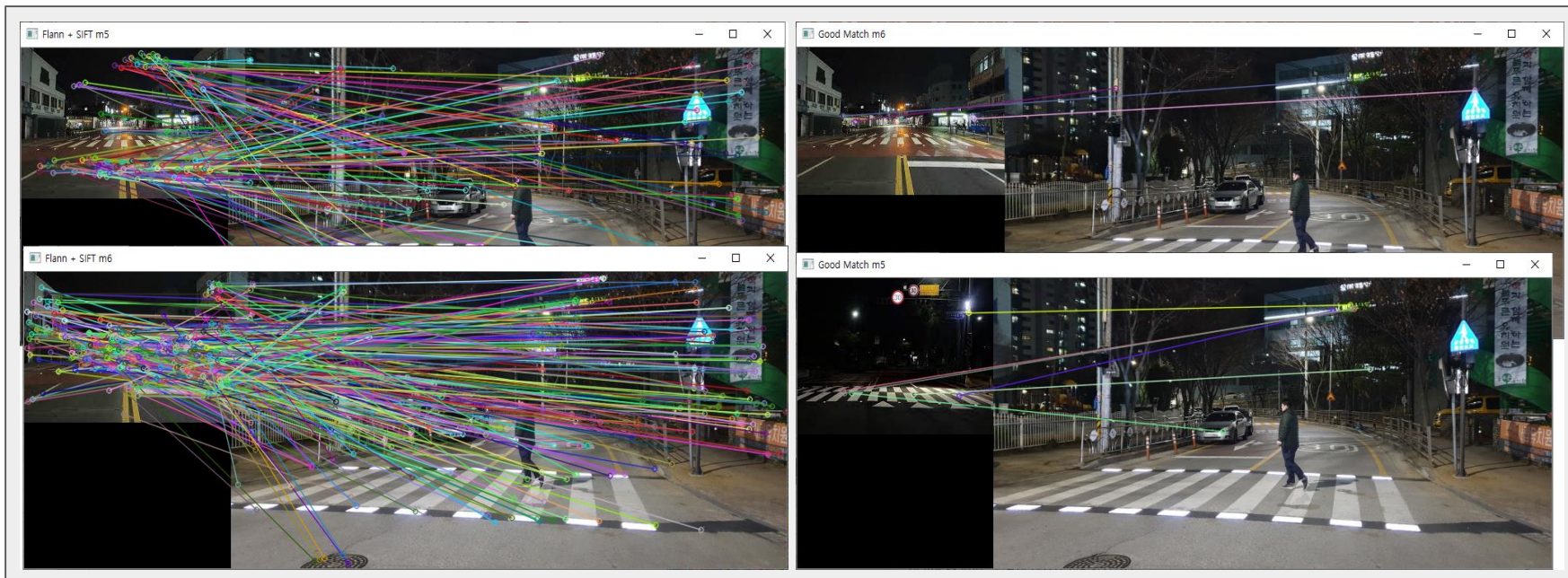
- opencv 3.4.2.16이상에서 SIFT와 SURF는 더 이상 지원안함.
- 시도해본 특징 검출 알고리즘 (구글 이미지 6개 등 특징검출에 사용)
  - BFMatcher와 SIFT로 매칭
  - BFMatcher와 SUFT로 매칭
  - BFMatcher와 ORB로 매칭
  - FLANNMatcher와 SIFT로 매칭
  - FLANNMatcher와 ORB로 매칭
  - match 함수로부터 좋은 매칭점 찾기
  - knnmatch 함수로부터 좋은 매칭점 찾기
  - 매칭점 원근 변환으로 영역 찾기
  - RANSAC 원근 변환 근사 계산으로 나쁜 매칭 제거
- > 특징검출을 통한 매치 알고리즘을 전부 돌려본 결과 제대로 검출되는 것이 없었음





## 3.2 횡단보도 객체 인식

### 3.2.1 특징 검출을 통한 객체인식 시행착오



- 횡단보도를 제대로 검출하지 못함

## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.1 ROIs 를 통한 2개 이상의 ROI 객체 검출 및 추적

- 순서
  1. cv2.ROIs를 이용해 ROI두개 지정(횡단보도, 사람)
  2. cv2.TrackerAPI를 활용해 객체 추적
  3. 횡단보도 ROI와 사람 ROI 기준 한쪽 ROI 꼭지점이 다른 ROI 내에 포함되면 검출
- 좌표를 산출하기 전 사람 roi와 횡단보도 roi를 구분
  - 사람 ROI는 가로보다 세로가 길음
  - 횡단보도 ROI는 세로보다 가로가 길음
  - 이 점을 이용해 횡단보도와 사람을 구분
- 세로가 더 긴 ROI(사람 ROI) 좌표를 기준으로 나머지 횡단보도 ROI 좌표 범위내에 존재하는지 비교



## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.1 ROIs 를 통한 2개 이상의 ROI 객체 검출 및 추적

- opencv 다운 그레이드

```
In [19]: 1 #!pip install opencv-contrib-python==3.4.11.45
          2 import cv2
          3 print(cv2.__version__)
          4
          5 #cv2.Tracker()함수를 사용하기 위해 opencv 다운그레이드
          3.4.11
```

- 여러가지 머신러닝 알고리즘으로 구현한 Tracking API 사용

- 머신러닝 알고리즘 목록
  - AdaBoost 알고리즘
  - MIL(Multiple Instance Learning) 알고리즘
  - KCF(Kernelized Correlation Filters) 알고리즘
  - TLD(Tracking, Learning and Detection) 알고리즘
  - MedianFlow 알고리즘
  - CSRT(Channel and Spatial Reliability) 알고리즘
  - MOSSE 알고리즘

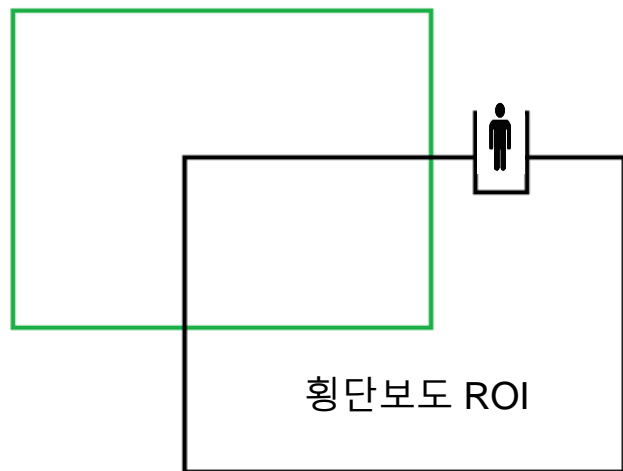
## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.1 꼭지점 기준으로 도형이 겹치는 영역 찾기

- 보행자가 횡단보도를 보행한다는 것은 각각의 ROI 사각형이 겹친다는 것으로 꼭지점 좌표가 어느 구간에 들어간다면 사각형에 겹치는 구간이 있다는 것으로 인식
- 따라서 비교대상이 되는 rect2(횡단보도 ROI)의 x값과 y값의 최댓값과 최소값으로 범위내에 있는지 검출

```
24 xrange = range(212, 212+480+1)
25 yrange = range(200, 200+363+1)
26 xrange = range(212, 693)
27 yrange = range(200, 584)
28
29 #비교대상(하단의 작은 검은 직사각형) 겹침
30 check_x, check_y = 527, 145      #x의 최소값: check_x, x의 최댓값: check_x+check_w
31 check_w, check_h = 60, 90        #y의 최소값: check_y, y의 최댓값: check_y+check_h
32
33 total = 0
34 #좌측상단 꼭지점 확인
35 if check_x in xrange and check_y in yrange:
36     total += 1
37
38 #우측상단 꼭지점 확인
39 if check_x+check_w in xrange and check_y in yrange:
40     total += 1
41
42 #좌측하단 꼭지점 확인
43 if check_y+check_h in yrange and check_x in xrange:
44     total += 1
45
46 #우측하단 꼭지점 확인
47 if check_x+check_w in xrange and check_y+check_h in yrange:
48     total += 1
49
50 print(total)      #꼭지점이 두개 겹친다는 의미
```

2



- 검은 사각형 2개를 대상으로 특정 사각형 내 꼭지점 포함 유무에 따라 겹침을 파악

## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.2 예외처리

- 꼭지점을 기준으로 겹침을 검출하는데 예외사항을 확인

Rect1 (흰 직사각형)    Rect2 (빨간 직사각형)

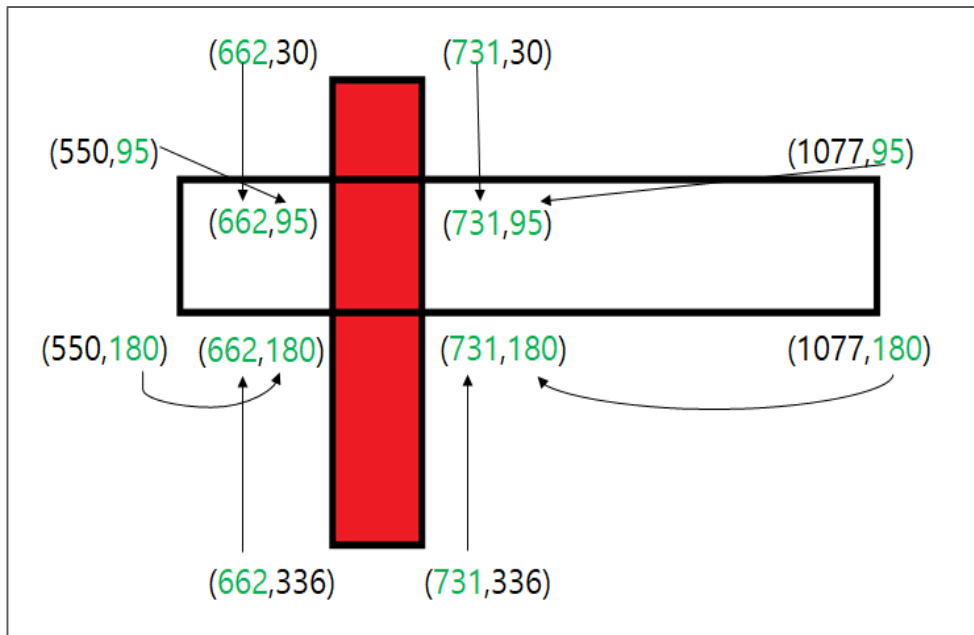
(550,95,527,85)    (662,30,69,306)

겹치는 구간

(662, 95)    (731, 95)

(662,180)    (731,180)

- rect1좌표와 rect2 좌표를 이용해 고정된 4개의 좌표를 얻을 수 있음
- 고정된 4개의 좌표가 rect1, rect2에 모두 포함되어 있을 때 다음과 같은 겹침 상태임을 알 수 있음



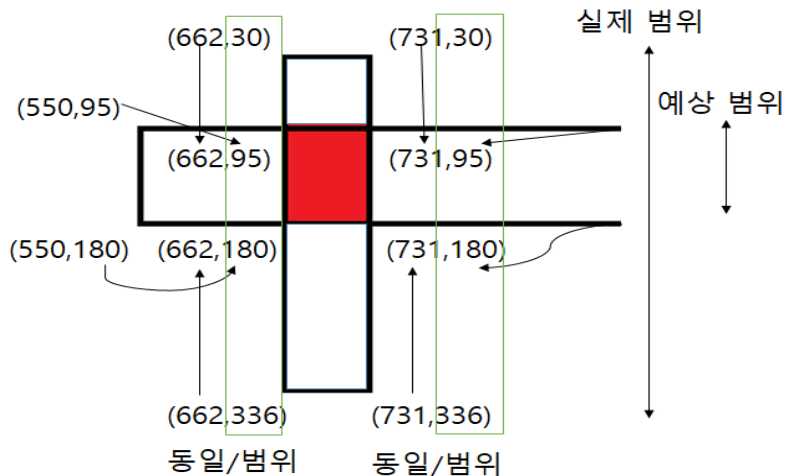


## 3.3 영상 내 횡단보도 보행자 객체 인식

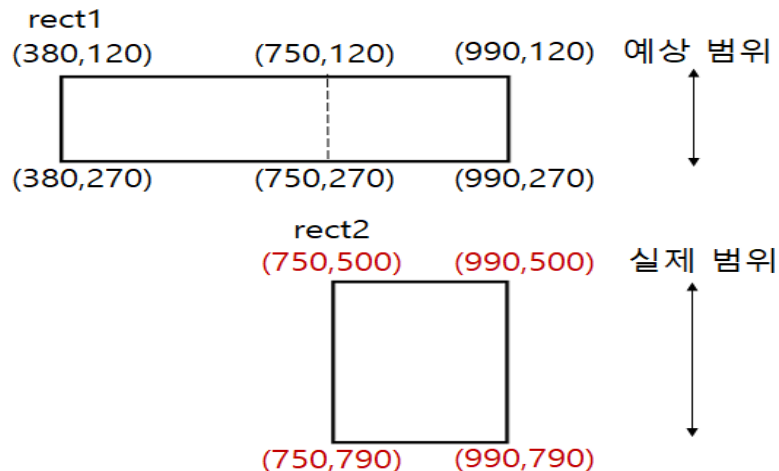
### 3.3.2 예외처리

- 고정된 4개의 좌표를 기준으로 예상범위와 실제범위 일체 유무에 따른 예시

- 예상범위에 실제범위 값이 포함되는 예시



- 예상범위에 실제범위 값이 포함되지 않는 예시



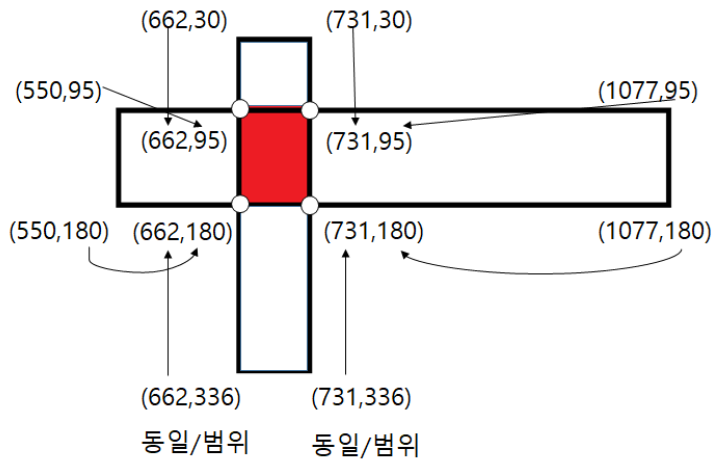
- 가로또한 세로와 동일한 방식으로 포함되는지 검출

## 3.3 영상 내 횡단보도 보행자 객체 인식

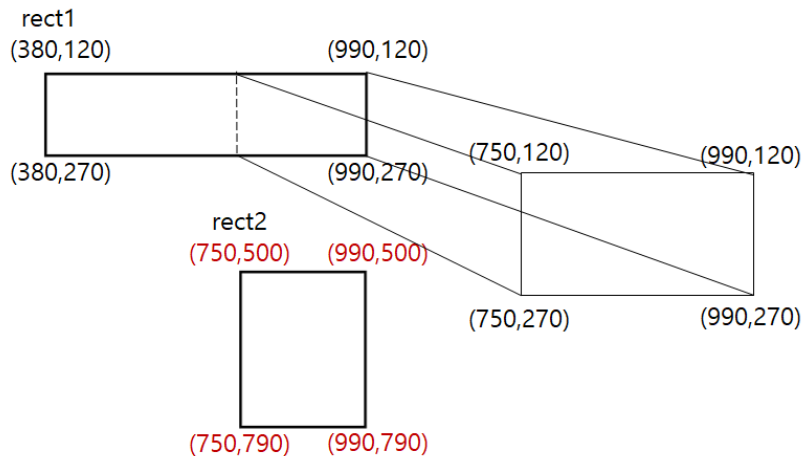
### 3.3.2 예외처리

- 즉, 고정된 4개의 좌표가 두 직사각형에 1개라도 포함되지 않을 경우 겹치지 않는다고 인식

- 예상범위에 실제범위 값이 포함되는 예시



- 예상범위에 실제범위 값이 포함되지 않는 예시



- 사람의 ROI에 횡단보도의 ROI가 일부 포함되는지 확인하기 위해, 겹친다고 생각되는 좌표 4개중 하나라도 두 직사각형에 포함되는지 확인

## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.2 예외처리

- 이전에 설명한 개념으로 코드를 작성했을 때 결과

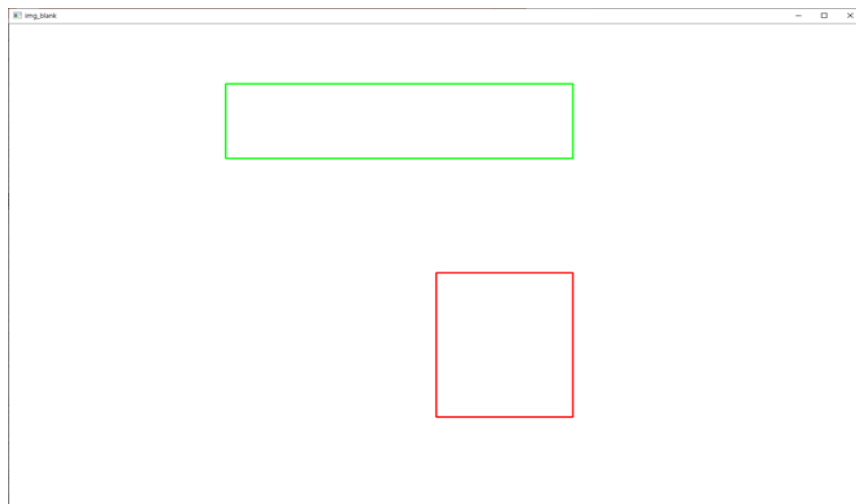
#### - 코드 일부

```
In [11]: 1 #사각형 결집 상태 확인3
2 img_test = cv2.imread('../img/blank_500.jpg')
3 #배율 지정으로 도화지 늘리기
4 img_test = cv2.resize(img_test, None, None, 3, 2, cv2.INTER_CUBIC)
5
6 rect1 = (380, 120, 610, 150)
7 rect2 = (750, 500, 240, 290)
8
9 #언두색 큰 사각형
10 cv2.rectangle(img_test, (rect1[0], rect1[1]), (rect1[0]+rect1[2], rect1[1]+rect1[3]), (0, 255, 0), 2)
11 #빨간 작은 사각형
12 cv2.rectangle(img_test, (rect2[0], rect2[1]), (rect2[0]+rect2[2], rect2[1]+rect2[3]), (0, 0, 255), 2)
13
14 cv2.imshow('img_blank', img_test)
15 cv2.waitKey()
16 cv2.destroyAllWindows()
17
18 #####
19 #결집은 영역이라고 예상된 곳에서의 꼭지점 수 구하기 (spot_check 함수사용)
20 rect1 = (380, 120, 610, 150)
21 rect2 = (750, 500, 240, 290)
22
23 print(rect1, rect2)
24 spot_check(rect1, rect2)
25
26 #성공
```

(380, 120, 610, 150) (750, 500, 240, 290)  
points (750, 120) (990, 120) (750, 270) (990, 270)

Out [11]: 0

#### - 예시 도형1



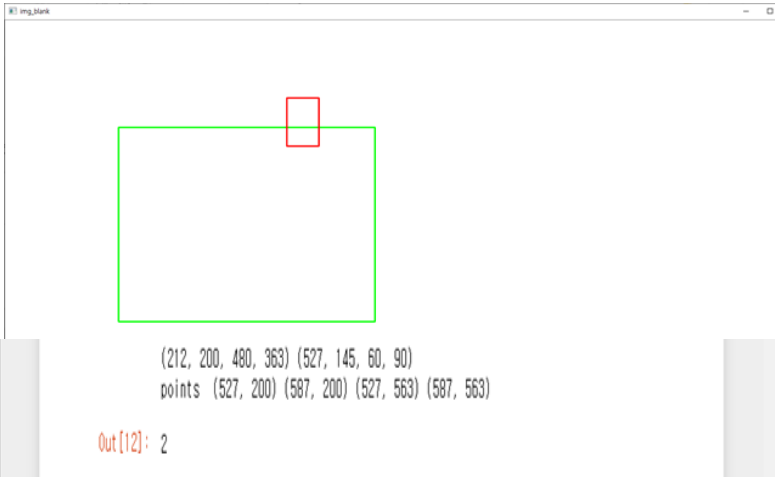
- 제대로 예외사항까지 처리함을 확인

## 3.3 영상 내 횡단보도 보행자 객체 인식

### 3.3.2 예외처리

- 이전에 설명한 개념으로 코드를 작성했을 때 결과

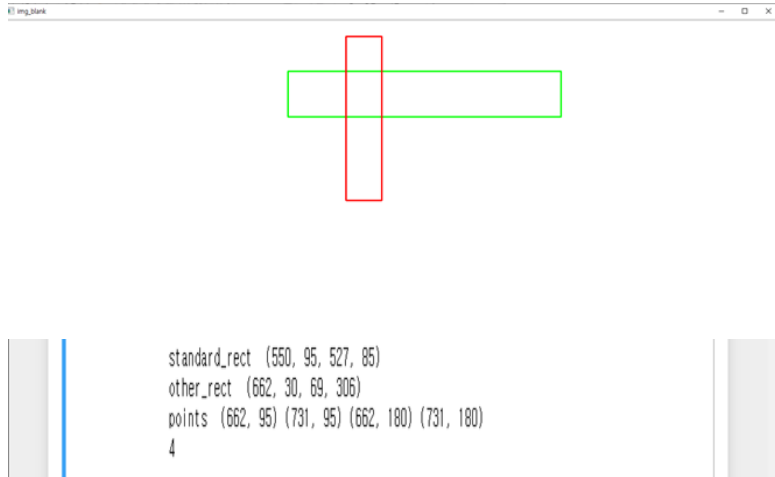
- 예시 도형2 및 결과



```
(212, 200, 480, 363) (527, 145, 60, 90)
points (527, 200) (587, 200) (527, 563) (587, 563)

Out[12]: 2
```

- 예시 도형3 및 결과



```
standard_rect (550, 95, 527, 85)
other_rect (662, 30, 69, 306)
points (662, 95) (731, 95) (662, 180) (731, 180)

4
```

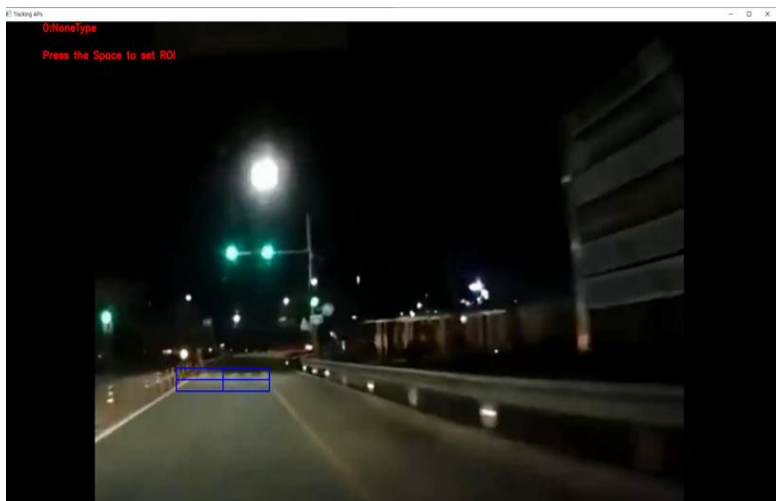
- 제대로 예외사항까지 처리함을 확인

## 3.3 영상 내 횡단보도 보행자 객체 인식

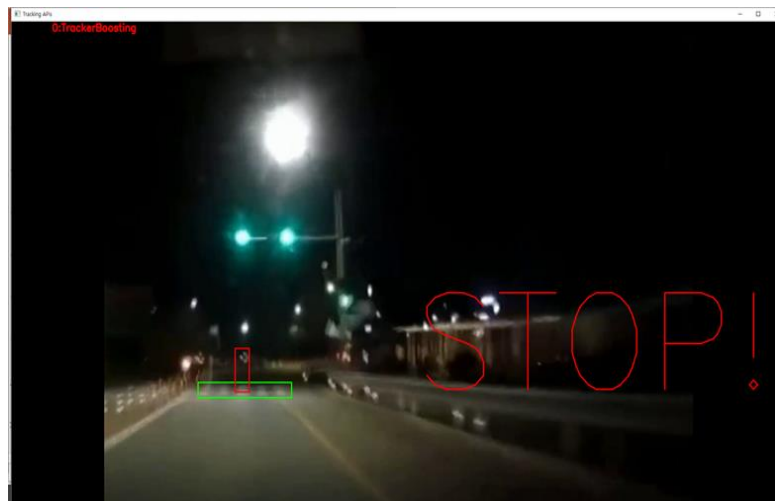
### 3.3.2 예외처리

- 이전에 설명한 개념으로 코드를 작성하고 실제 영상으로 검출했을 때 결과

- ROI 지정



- ROI 기준 보행자 객체검출 중간 확인





## 3.4 알고리즘 시연

### 3.4.1 알고리즘 시연

- 코드 사용방법

노란박스로 hog 알고리즘을 통해 사람 객체 예상  
원하는 시점에서 spacebar로 화면 1차 정지

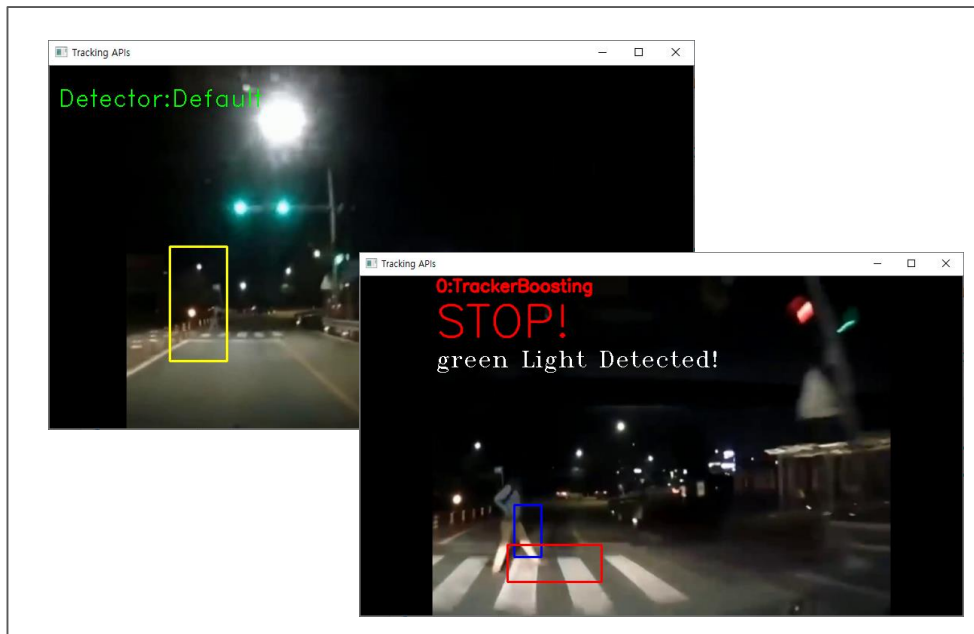
드래그 1번째 ROI 등록 + spacebar  
드래그 2번째 ROI 등록 + spacebar

#반드시 ROI는 두개만 등록하도록 하며  
#횡단보도 먼저 등록

esc로 사람 객체 인식 알고리즘 검출화면 나오기  
#객체추적 알고리즘은 숫자로 변환 가능(1~6번까지)

esc로 종료

- 코드 실행 화면



## 3.5 제한점

### 3.5.1 제한점

- 영상의 길이가 길어질수록 객체추적의 정확도가 떨어짐
- 영상의 정면에 있는 대상보다 좌측에 있는 대상을 추적할 때 정확도가 훨씬 떨어짐
- 여러명의 사람들을 대상으로 객체추적이 가능하나 일일이 tracker를 지정해줘야 하기 때문에 원하는 때에 원하는 대상을 추가해 tracking하기 어려움
- ROI를 여러 개 지정해 추적할 수는 있으나 원하는 객체가 생길 때 마다 ROI를 계속 추가해서 추적할 수는 없음
- 7가지 Tracking 알고리즘이 존재하는데 영상에 따라 알고리즘이 안되거나 되는것이 있음.
- 차량 신호등 초록불 검출이 코드 실행에 따라 결과가 다르게 나옴 하지만 이에 대해 규명 불가

# 참고문헌

## 단행본

이세우. 파이썬으로 만드는 OpenCV 프로젝트. 서울: 인사이트, 2019.

## 웹페이지

[https://ko.wikipedia.org/wiki/%EC%84%9C%ED%8F%AC%ED%8A%B8\\_%EB%B2%A1%ED%84%B0\\_%EB%A8%B8%EC%8B%A0](https://ko.wikipedia.org/wiki/%EC%84%9C%ED%8F%AC%ED%8A%B8_%EB%B2%A1%ED%84%B0_%EB%A8%B8%EC%8B%A0)

166.104.231.121/ysmoon/mip2017/lecture\_note/제6장-추가.pdf

「

Q&A

」