

2주차 요약본

작성자



이의진(엘텍공과대학 휴먼기계바이오공학부)

4.1 에지 검출(Edge Detection)

edge : 물체 경계에서 pixel value가 급변하는 지점 → **pixel value의 변화량이 큰 픽셀을 선택!**

영상의 미분

미분: x값이 미세하게 증가할 때 함수의 변화량 측정

$$(식) \lim_{\delta x \rightarrow 0} \frac{f(x+\delta x)-f(x)}{\delta x}$$

◆ 디지털 영상의 미분 : 연속 구간X, 정수 좌표 → $\delta x = 1$

$$\text{식: } f'(x) = \frac{f(x+\delta x)-f(x)}{\delta x} = f(x+1) - f(x)$$

- 영상에 적용하려면? → Edge Operator(u)로 convolution(u의 중심점은 왼쪽 pixel)

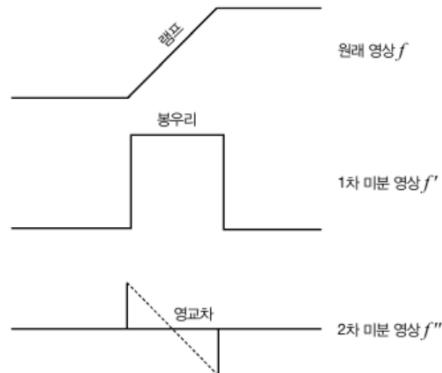


(b) 디지털 영상의 미분(필터 u 로 컨볼루션)

- pixel value 변화가 없는 곳은 0, 급변하는 부분은 3
- 위 영상은 계단 에지이므로 에지 찾기가 쉬움(두께가 1)

에지 연산자

◆ 램프 에지 : pixel value 가 서서히 변함. 급변X



• 1차 미분 연산

- (장) 에지 발생 여부, 에지의 방향(pixel value가 커지면 양수, 작아지면 음수)

- (단) 위치 찾기(localization)문제

- 봉우리 발생

• 2차 미분 연산-영교차(zero crossing)

- (-1 1)로 두 번의 컨볼루션을 적용 or (1 -2 1)로 한 번의 컨볼루션

- 영교차(zero crossing): 에지 부분에서 부호가 바뀜

$$\begin{aligned}
 f''(x) &= \frac{f'(x) - f'(x-\delta)}{\delta} = f'(x) - f'(x-1) \\
 &= (f(x+1) - f(x)) - (f(x) - f(x-1)) \\
 &= f(x+1) - 2f(x) + f(x-1)
 \end{aligned}$$

이 식을 구현하는 필터는 $\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$

⇒ 에지 검출: 1차 미분에서 봉우리, 2차 미분에서 영교차를 찾는다.

◆ 1차 미분에 기반한 Edge Operator

- 2차원 확장: x 방향과 y 방향의 두 연산자 사용

$$f'_x(y, x) = f(y, x+1) - f(y, x-1)$$

$$f'_y(y, x) = f(y+1, x) - f(y-1, x)$$

이 식을 구현하는 필터는 $u_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ 와 $u_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$

- 가장 널리 쓰이는 에지 연산자(Prewitt, Sobel)

- 노이즈 흡수, smoothing 효과(윗 행과 아래 행을 같이 고려하므로)
- Sobel은 가까운 pixel에 가중치 2를 줌

$$u_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad u_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

(a) 프레윗(Prewitt) 연산자

$$u_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad u_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(b) 소벨(Sobel) 연산자

- 에지 강도와 에지 방향

- Gradient = (f'_y, f'_x)

$$\text{에지 강도: } s(y, x) = \sqrt{f'_x(y, x)^2 + f'_y(y, x)^2}$$

$$\text{그레이디언트 방향: } d(y, x) = \arctan\left(\frac{f'_y(y, x)}{f'_x(y, x)}\right)$$

에지 방향은 그레이디언트 방향에 수직한 방향(90도 회전)

⇒ 모두 실수이므로 OpenCV에서 cv.CV_32F로 지정하자.

- 소벨 연산자 적용하면?

[예시 4-1] 소벨 연산자 적용 과정

[그림 4-7]은 대각선을 기준으로 위쪽은 3, 아래쪽은 1인 가상의 영상에 소벨 에지 연산자를 적용하는 과정을 예시한다. 회색으로 표시한 (3,4) 화소에 대한 자세한 계산 과정을 설명한다.



그림 4-7 소벨 연산자 적용 사례

◆ 소벨 연산자의 적용

```
#4-1 소벨 에지 검출

import cv2 as cv

img=cv.imread('soccer.jpg')
resize_img = cv.resize(img, (300, 300))

gray=cv.cvtColor(resize_img, cv.COLOR_BGR2GRAY)

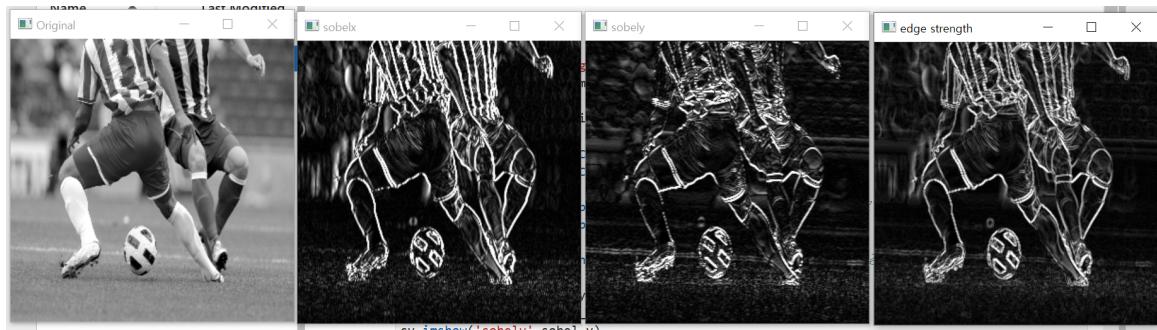
grad_x=cv.Sobel(gray, cv.CV_32F, 1, 0, ksize=3) # 소벨 연산자 적용, x방향 연산자 사용 지시
grad_y=cv.Sobel(gray, cv.CV_32F, 0, 1, ksize=3)

sobel_x=cv.convertScaleAbs(grad_x) # 절대값을 취해 양수 영상으로 변환
sobel_y=cv.convertScaleAbs(grad_y)
# convertScaleAbs: 부호 없는 8비트형 맵을 만들
# 0이하 값은 0, 255이상 값은 255로 범위 변경

edge_strength=cv.addWeighted(sobel_x, 0.5, sobel_y, 0.5, 0)
# 에지 강도 계산. 이미지 결합 ->에지 검출 완료!
# 255이상 값은 255로 범위 변경

cv.imshow('Original',gray)
cv.imshow('sobelx',sobel_x)
cv.imshow('sobely',sobel_y)
cv.imshow('edge strength',edge_strength)

cv.waitKey()
cv.destroyAllWindows()
```



- Sobel함수

```
cv2.Sobel(입력 영상, ddepth, dx, dy, dst=None, ksize=None, scale=None, delta=None, borderType=None) -> dst
```

▼ 인수 설명

- ddepth: 출력 영상 데이터 타입. -1이면 입력 영상과 같은 데이터 타입을 사용.
- dx: x 방향 미분 차수. 1차미분할지 2차미분 할지 결정
- dy: y 방향 미분 차수.
- dst: 출력 영상(행렬)
- ksize: 커널 크기. 기본값은 3.
- scale: 연산 결과에 추가적으로 곱할 값. 기본값은 1.
- delta: 연산 결과에 추가적으로 더할 값. 기본값은 0.
- borderType: 가장자리 픽셀 확장 방식. 기본값은 cv2.BORDER_DEFAULT.

sobel edge detection 두꺼운 형태로 에지 검출, 정확하지 X

4.2 캐니 에지

◆**최소 오류율(Good Detection):** 모든 에지 검출, 검출된 에지는 모두 참

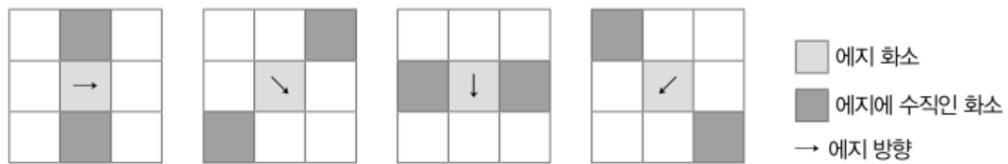
위치 정확도(Good Localization): 실제 에지 중심을 검출

한 두께(Single Edge): 하나의 에지에 대해 한 두께만 반환

⇒ 위 기준에 따라 목적함수를 정의하고 최적화 방식으로 에지 검출

◆**알고리즘**

1. 노이즈 제거(가우시안 필터 이용)
2. 그래디언트 찾기(1차 미분) → 에지 찾기
3. 비최대 억제(Non-Maximum Suppression): 에지 방향에 수직인 두 이웃 화소와 비교판단. 현재 pixel edge가 가장 크면 보존. 아니면 삭제



4. 이중 임계값(Double Thresholding)을 사용한 히스테리시스 에지 트레킹



- 파란색 영역(non-relevant)은 삭제
- 빨간색 영역은 강한 에지 → 최종 에지 영상에 추가
- 주황색 영역은 약한 에지 → 강한 에지와 연결될 경우에만 추가

⇒ 약한 에지에 대해서만 연결 전체를 따라다니며 8방향으로 검색하여 강한 에지와의 연결성 여부 판단

<https://com24everyday.tistory.com/370>

◆ 4-2 캐니에지 실험하기

```
import cv2 as cv

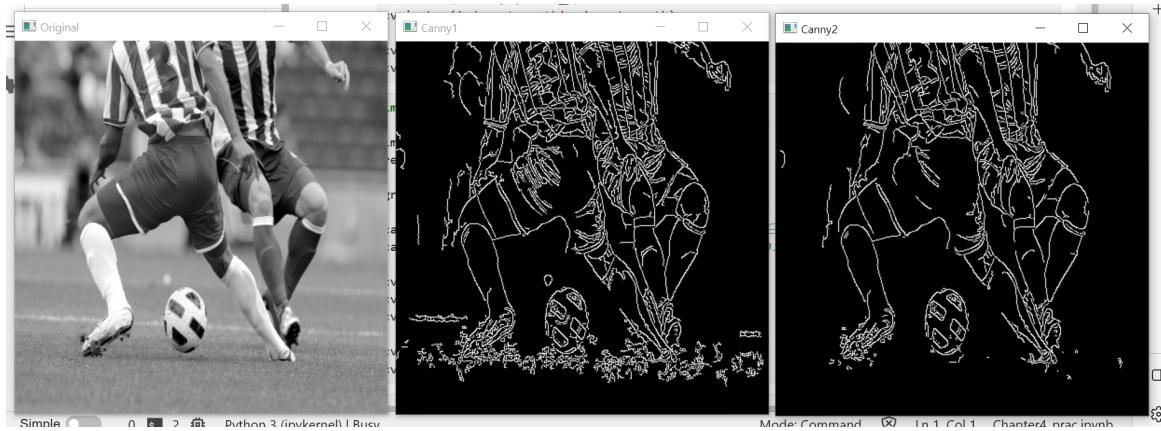
img=cv.imread('soccer.jpg') # 영상 읽기
resize_img = cv.resize(img, (400, 400))

gray=cv.cvtColor(resize_img, cv.COLOR_BGR2GRAY)

canny1=cv.Canny(gray,50,150) # Tlow=50, Thigh=150으로 설정
canny2=cv.Canny(gray,100,200) # Tlow=100, Thigh=200으로 설정

cv.imshow('Original',gray)
cv.imshow('Canny1',canny1)
cv.imshow('Canny2',canny2)

cv.waitKey()
cv.destroyAllWindows()
```



- 임계값이 높을 때: 더 적은 에지 발생. 획의 일부 손실
 - 임계값이 낮을 때: 획이 온전하게 검출되나 노이즈 에지 발생.
- ◆에지 검출의 한계: pixel value의 변화에만 의존함. 캐니 이후 개선 없음.

4.3 직선 검출

경계선 찾기

- 8-연결 에지 화소(이웃한 에지)를 연결해 경계선(contour) 구성

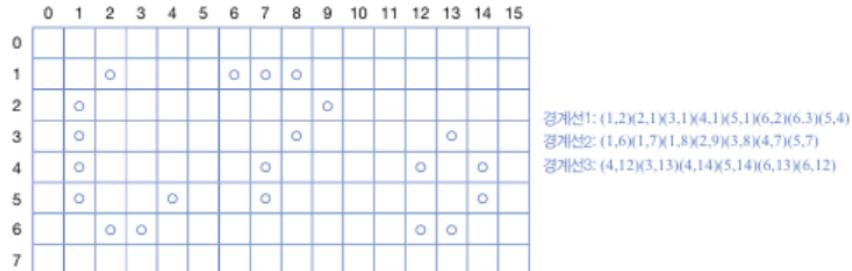


그림 4-9 에지 맵에서 경계선 찾기

- ◆에지 맵에서 경계선 찾기
(에지 맵에서 경계선 검출 → 길이가 임곗값 이상인 경계선만 추한다.)

```

import cv2 as cv
import numpy as np

img=cv.imread('soccer.jpg') # 영상 읽기
resize_img = cv.resize(img, (717,474))
resize_img_ = cv.resize(img, (717,474))

gray=cv.cvtColor(resize_img, cv.COLOR_BGR2GRAY)
canny=cv.Canny(gray,100,200) #에지맵 구하기

contour,hierarchy=cv.findContours(canny, cv.RETR_LIST, cv.CHAIN_APPROX_NONE)
# findContours함수로 경계선을 찾아 contour 객체에 저장.
# cv.RETR_LIST: 맨 바깥쪽 경계선만 찾도록 지시
# cv.CHAIN_APPROX_NONE: 모든 점을 기록.
lcontour=[]

```

```

for i in range(len(contour)):
    if contour[i].shape[0]>100:
        lcontour.append(contour[i])
#시작점부터 끝점까지 추적한 다음 역추적하여 시작점으로 돌아옴
#->길이가 100이상인 경계선만 골라 lcontour객체에 저장
cv.drawContours(resize_img,lcontour,-1,(0,255,0),3)

contour_,hierarchy_=cv.findContours(canny, cv.RETR_LIST, cv.CHAIN_APPROX_SIMPLE)
# cv.CHAIN_APPROX_SIMPLE: 직선에 대해서는 양 끝점만 기록

lcontour_=[]
for i in range(len(contour_)):
    if contour_[i].shape[0]>100:
        lcontour_.append(contour_[i])
#길이가 100이상인 경계선만 골라 lcontour객체에 저장
cv.drawContours(resize_img_,lcontour_,-1,(0,255,0),3)

cv.imshow('Original with contours',resize_img)
cv.imshow('Original with contours_',resize_img_)

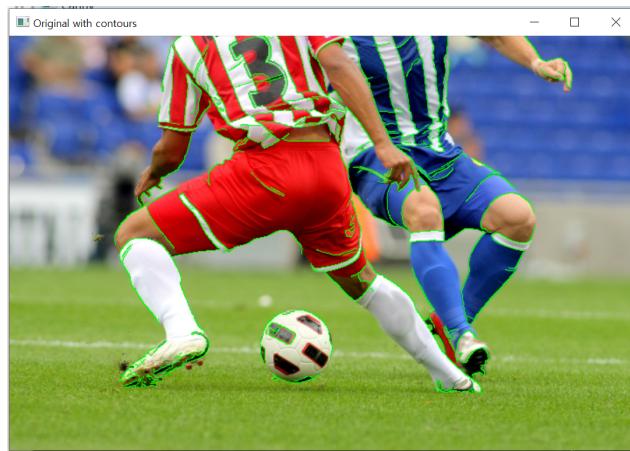
cv.imshow('Canny',canny)

cv.waitKey()
cv.destroyAllWindows()

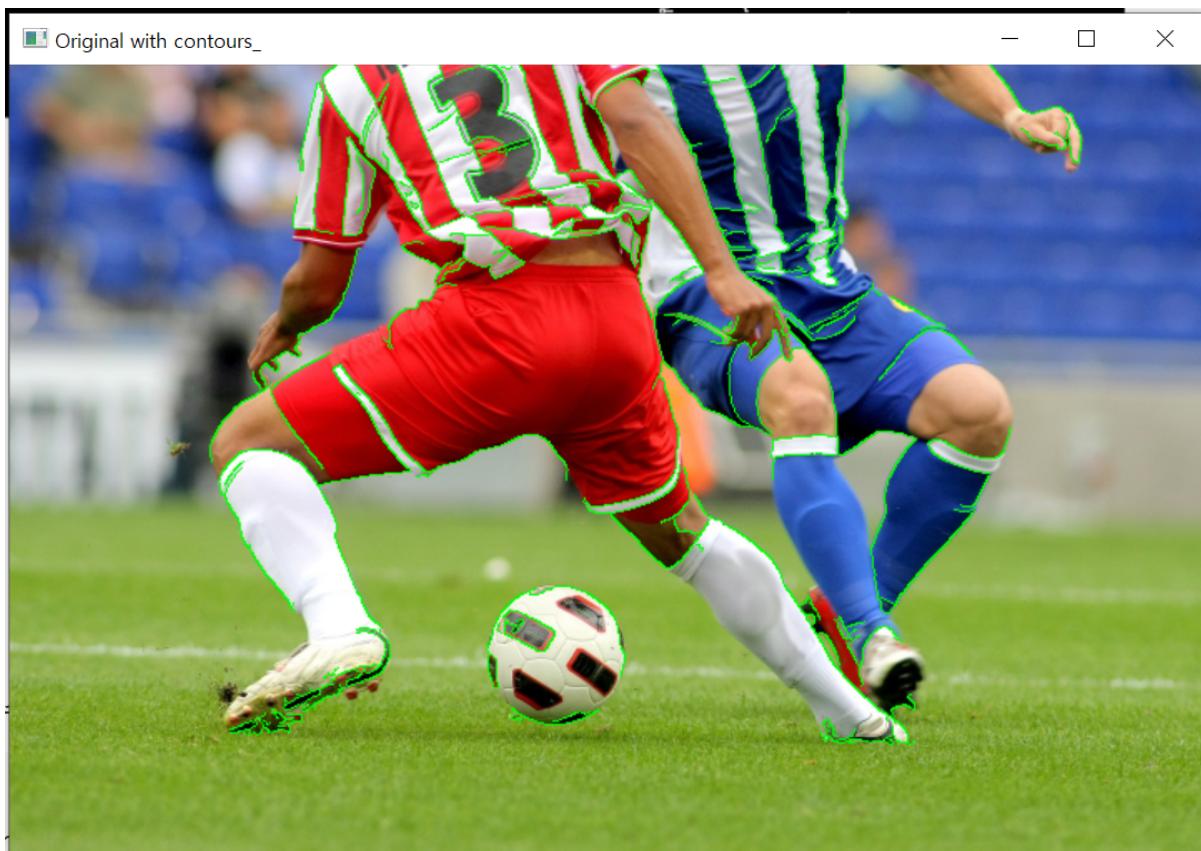
```



에지 맵까지만 추출



경계선 검출-모든 점을 기록(엉덩이 부근 직선??)



직선의 경우 양 끝점만 기록

- **findContours함수 - 외곽선 검출**

```
cv2.findContours(image, mode, method) -> contour, hierarchy
```

▼ 인수 설명

- **image:** 경계선을 찾을 에지 영상.
- **mode:** 외곽선 검출 모드.
- **method:** 경계선을 표현하는 방식 지정

- `drawContours` 함수- 외곽선 그리기

```
cv2.drawContours(img, lcontour_, contourIdx, (0,255,0), 3)
```

▼ 인수 설명

- img: 입출력 영상
- lcontour_: (`cv2.findContours()` 함수로 구한) 외곽선 좌표 정보
- contourIdx: 외곽선 인덱스. 음수(-1)를 지정하면 모든 외곽선을 그린다.
- color: 외곽선 색상
- thick: 외곽선 두께

히프 변환

이웃한 에지를 연결하는 경계선 검출 방식 → 에지가 끊겨 나타남

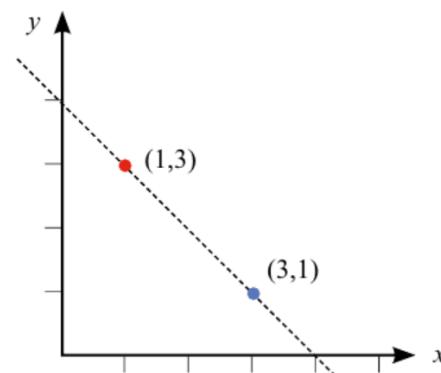
⇒ 히프 변환으로 해결! 끊긴 에지를 모아 선분, 원 검출

◆ 원리

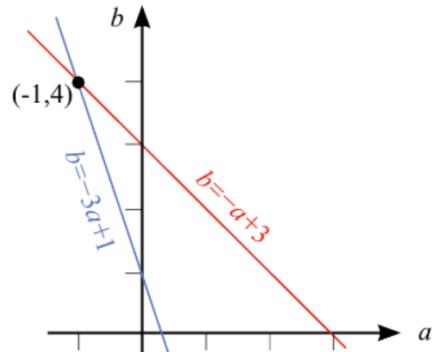
1. 입력된 각각의 점에 대해 (a, b) 공간에 직선을 그린다.
2. 이들 직선이 만나는 점 (a, b) 을 찾아 a 를 기울기, b 를 y 절편으로 취한다. 만나는 점은 투표로 알아낸다.

▼ 상세설명

- $(x, y) = (1, 3)$ 을 지나는 직선 $\rightarrow b = -a + 3$
- a, b 를 변수로 간주하면 새로운 공간 (a, b) 형성되며 기울기 -1 , 절편이 3 인 직선이 됨.
- a, b 공간에서 두 직선이 만나는 점은 $(-1, 4)$ 이며 이는 원래 공간에서 두 점을 지나는 직선의 기울기와 절편이다 $\Rightarrow y = -x + 4$
 - 각각의 직선은 자신이 지나온 점에 1만큼 투표함 \Rightarrow 2표를 받은 점이 만나는 점.



(a) (y, x) 로 표현되는 영상 좌표



(b) (b, a) 로 표현되는 공간으로서 매핑

그림 4-10 히프 변환의 원리

◆ 실제 상황에서 구현

1. 현실은 매우 많은 점, 일직선X

- (a, b) 공간을 이산화 → (a,b)의 2차원 누적 배열 v를 만들고 0으로 초기화 → 각각의 직선은 자신이 지나온 점에 1만큼 투표

2. 투표가 이뤄진 누적 배열에 노이즈 많음

- NMS로 극점만 남기고 임곗값을 같이 적용.

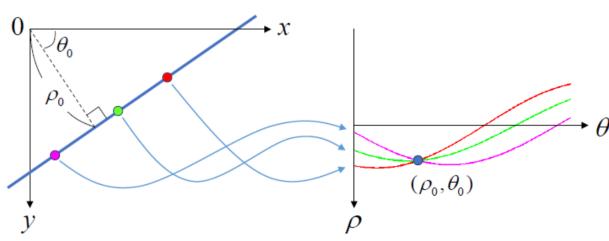
0	1	0	0	0	0	0	0
0	2	2	0	1	3	0	0
0	3	5	3	2	0	0	0
0	2	4	2	6	7	0	0
0	2	3	3	5	8	6	0
0	1	0	0	0	4	5	3

그림 4-11 비최대 억제로 찾은 극점 2개

3. $y = ax+b$ 에서 a가 무한대일 경우 투표 불가능

- 극좌표계 도입하여 해결(각도와 거리)

$$x\sin(\theta) + y\cos(\theta) = \rho$$



- 원 방정식 이용하여 원 검출가능(3차원 누적배열 사용)

$$(x-a)^2 + (y-b)^2 = r^2$$

◆ 허프 변환을 이용해 사과 검출하기

```
import cv2 as cv

img=cv.imread('apples.jpg')
gray=cv.cvtColor(img, cv.COLOR_BGR2GRAY)

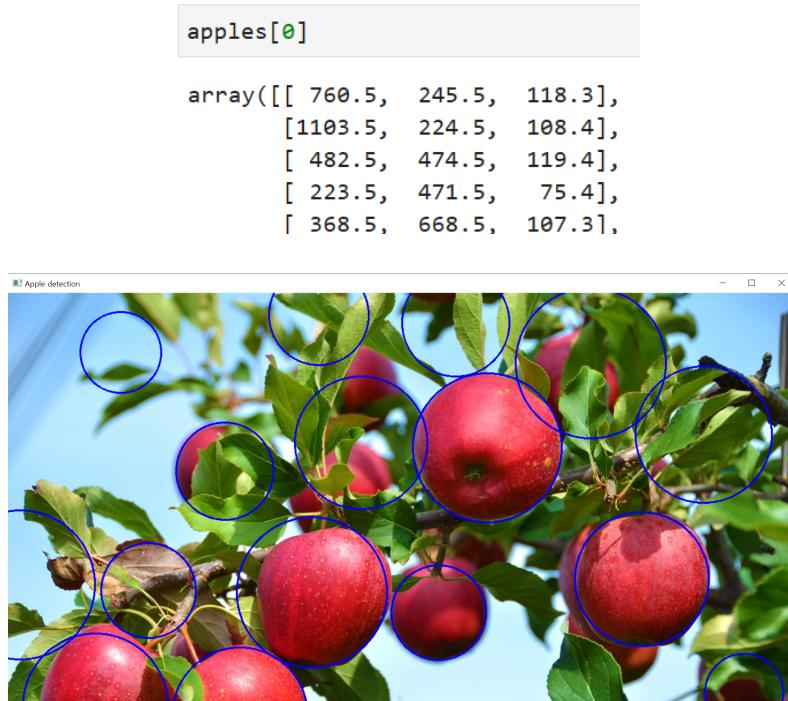
apples=cv.HoughCircles(gray, cv.HOUGH_GRADIENT, 1, 200, param1=150, param2=20, minRadius=50, maxRadius=120)
```

```
# cv.HOUGH_GRADIENT: 에지 방향 정보를 추가로 사용하는 방법

for i in apples[0]:
    cv.circle(img,(int(i[0]),int(i[1])),int(i[2]),(255,0,0),2)
# apples리스트가 가진 정보 이용하여 원본에 원 그림

cv.imshow('Apple detection',img)

cv.waitKey()
cv.destroyAllWindows()
```



- HoughCircles 함수 - 명암 영상에서 원을 검출, 중심과 반지름을 저장한 리스트 반환.

```
cv.HoughCircles( src, method, dp, min_dist, parameter1, parameter2, min_Radius, max_Radius)
```

▼ 인수 설명

- **src** : 입력할 이미지 변수, grayscale 의 이미지를 입력해야 함
- **method** : 원 검출방법, HOUGH_GRADIENT 를 사용하면 됨
- **dp** : 누적 배열의 크기 지정,
 - 값이 1이면 원본 해상도를 사용함, 웬만하면 1을 쓰면 됨
 - 값이 2이면 절반 해상도를 사용함
- **min_dist** : 검출할 원의 최소 거리
 - 작을수록 많은 원이 검출
- **parameter1** : Canny edge detection 에서의 높은 threshold 값
- **parameter2** : 비최대 억제 적용 시 임곗값
- **min_Radius** : 검출 될 원의 최소 반지름

- 모든 원을 검출하려면 0을 입력
 - max_Radius : 검출 될 원의 최대 반지름
- 모든 원을 검출하려면 0을 입력
- circle함수

```
cv2.circle(img, 원의 중심좌표, 원의 반지름, (B,G,R), 선 두께, 선 종류)
```

RANSAC(Random sample consensus)

허프 변환: 누적 배열에 노이즈가 많으면 직선 2개를 출력할 수 있음.

최소평균제곱오차(LMSE): 모든 점을 대상으로 최소 오류를 범하는 직선을 찾음.

→ 모든 샘플이 동등하게 오류 계산에 참여하므로 아웃라이어의 영향 크게 받음.

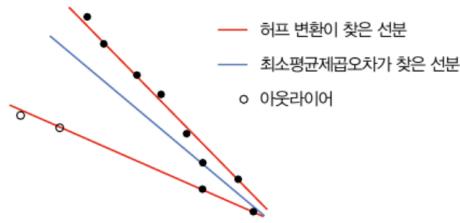
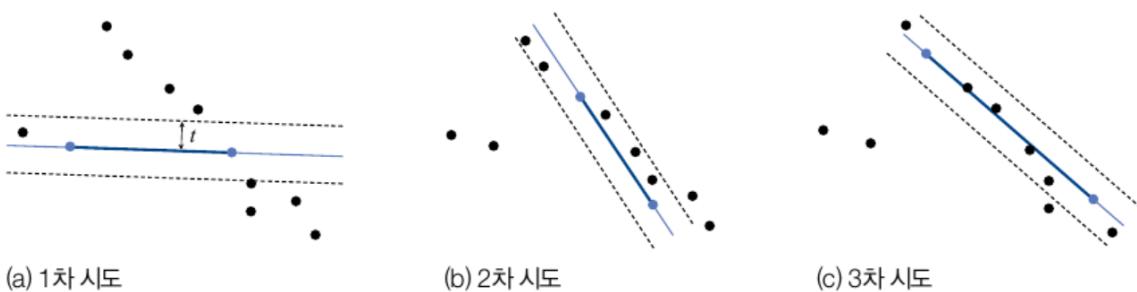


그림 4-12 강민하지 않은 기법의 선분 추정

- Robust estimation(강민한 추정): 아웃라이어를 걸러냄. ex) [1 6 1 1 1]
- RANSAC: 인라이어를 찾아 최적 근사하는 기법.
 - 임의의 두 점을 선택하고 그것을 지나는 직선 계산한다. 오차 t 를 허용해 직선에 일치하는 점의 개수를 센다.
 - 개수가 임곗값 d 를 넘지 못하면 버림. 넘으면 일치하는 점들만 가지고 다시 최적 직선을 추정한다.
 - 추정 오류가 임곗값 e 보다 작으면 후보군에 추가, 그렇지 않으면 버림.

⇒ 반복할수록 효과가 크지만 시간이 더 걸리므로 적절한 값을 설정해야함.

⇒ 난수 사용



(a) 1차 시도

(b) 2차 시도

(c) 3차 시도

그림 4-13 RANSAC으로 선분 추정(1차, 2차, 3차, 4차, … 시도를 반복)

4.4 영역 분할

region segmentation : 물체가 점유한 영역을 구분한다. 에지가 완벽하면 따로 필요하지 않지만 폐곡선을 형성하지 못하는 경우가 많음.

배경이 단순한 영상의 영역 분할

- 이진화 알고리즘: 오츄 이진화
- 군집화 알고리즘: r, g, b값으로 표현된 픽셀을 샘플로 3차원 공간에서 클러스터링 수행 → 연결 요소 찾아 영역으로 간주
- 워터셰드: 비가 오면 오목한 곳에 웅덩이가 생기는 현상을 모방. 낮은 곳부터 물 채워 서로 다른 웅덩이 찾음.

슈퍼 화소 분할

슈퍼 화소(super-pixel): 영상을 아주 작은 영역으로 분할하여 입력으로 활용. (픽셀보다 크고 물체보다 작음)

- SLIC(Simple Linear Iterative Clustering)
 - 입력 영상에서 k개의 pixel을 군집 중심으로 지정
 - pixel을 5차원 벡터로 표현
 - (R,G,B,x,y) : 화소의 색상, 위치
 - 화소 할당 단계 : pixel 각각에 대해 주위 4개 군집 중심과 자신까지의 거리를 계산해서 가장 유사한 군집 중심에 할당.
 - 군집 중심 갱신 단계: 화소 할당 이후 각 군집 중심은 자신에게 할당된 픽셀을 평균해 중심을 갱신
 - 위 두 단계 반복하다가 모든 군집 중심의 이동량의 평균을 구하고 평균이 임계치보다 작으면 수렴 판단 → 알고리즘 멈춤

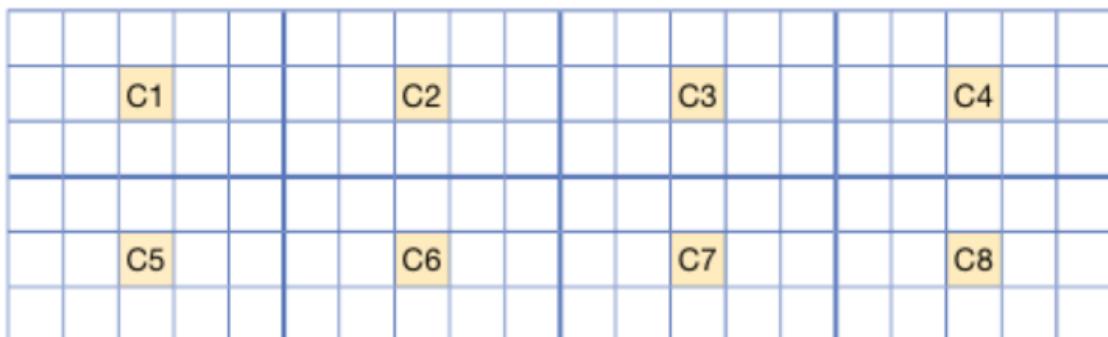


그림 4-15 SLIC 알고리즘의 초기 군집 중심

- ◆ SLIC 알고리즘으로 입력 영상을 슈퍼 화소 분할하기

```
import skimage
import numpy as np
import cv2 as cv

img=skimage.data.coffee() #skimage 내부의 coffee영상 읽어 저장
#img.size = 400 X 600
cv.imshow('Coffee image',cv.cvtColor(img, cv.COLOR_RGB2BGR))
#skimage는 RGB, OpenCV는 BGR순서로 저장하므로 cvtColor이용해 변환하여 출력
```

```

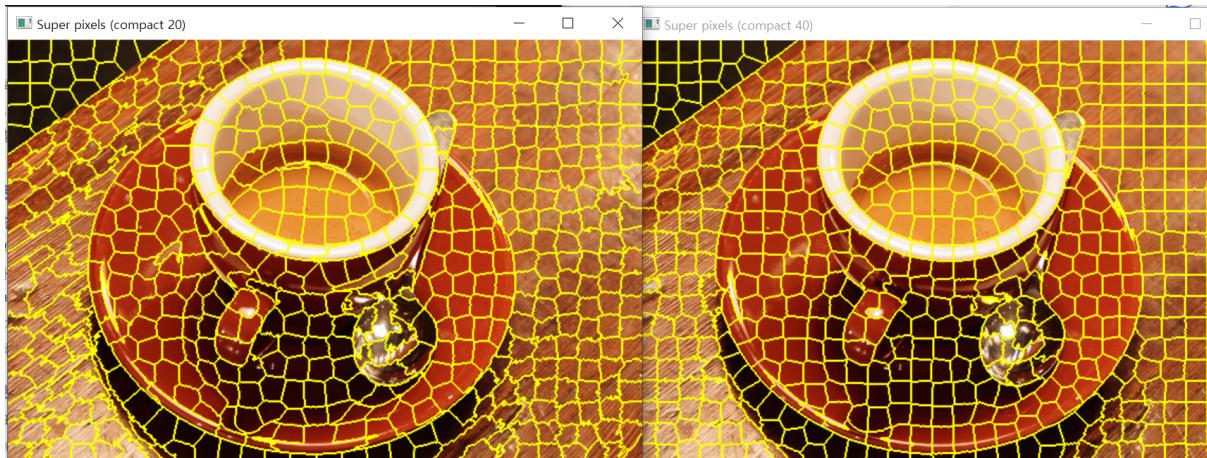
slic1=skimage.segmentation.slic(img,compactness=20,n_segments=600)
sp_img1=skimage.segmentation.mark_boundaries(img,slic1)
# slic1 객체의 분할 정보를 img에 표시하고 결과를 저장
sp_img1=np.uint8(sp_img1*255.0)
# 0~1 사이의 실수로 표현된 sp_img1을 0~255 사이로 바꾸고 uint8형으로 변환

slic2=skimage.segmentation.slic(img,compactness=40,n_segments=600)
sp_img2=skimage.segmentation.mark_boundaries(img,slic2)
sp_img2=np.uint8(sp_img2*255.0)

cv.imshow('Super pixels (compact 20)',cv.cvtColor(sp_img1,cv.COLOR_RGB2BGR))
cv.imshow('Super pixels (compact 40)',cv.cvtColor(sp_img2,cv.COLOR_RGB2BGR))

cv.waitKey()
cv.destroyAllWindows()

```



→ compactness 인수를 크게하면 네모 모양은 잘 유지되는 반면 슈퍼 화소의 색상 균일성이 낮아짐.

- slic 함수: 슈퍼 화소 분할을 수행

▼ 인수 설명

slic함수: 슈퍼 화소 분할 수행

img: 분할할 영상

compactness: 슈퍼 화소의 모양 조절. 값이 클수록 네모에 가깝고 색상 균일성은 희생됨.

n_segments: 슈퍼 화소의 개수 지정

최적화 분할

지역적 명암 변화(앞의 분할 알고리즘들) + 전역적 정보를 같이 고려하는 분할

영상을 그래프로 표현하고 분할을 최적화 문제로 푼다.

◆ 영상의 그래프 표현

- pixel을 노드로 사용하면 계산 효율 떨어짐.
⇒ 슈퍼 화소를 구하면 노드 개수를 현저히 줄일 수 있다.
- 두 노드를 연결하는 에지의 가중치로 유사도 사용
- 두 노드의 유사도를 아래와 같이 정의한다. (거리 \leftrightarrow 유사도)
 - $f(v)$: 노드 v 에 해당하는 픽셀의 색상과 위치를 결합한 벡터
 - D : 거리가 가질 수 있는 최댓값

$$\begin{aligned} \text{거리} & \begin{cases} d_{pq} = \|f(v_p) - f(v_q)\|, \text{ 만일 } v_q \in \text{neighbor}(v_p) \\ \infty, \text{ 그렇지 않으면} \end{cases} \\ \text{유사도} & \begin{cases} s_{pq} = D - d_{pq} \text{ 또는 } \frac{1}{e^{d_{pq}}}, \text{ 만일 } v_q \in \text{neighbor}(v_p) \\ 0, \text{ 그렇지 않으면} \end{cases} \end{aligned}$$

- 두 노드가 8-이웃을 이루거나 둘 사이의 거리가 사용자 지정값 r 이내면 참.

◆ 정규화 절단 알고리즘

- 정규화 절단(normalized cut)은 pixel을 노드로, $f(v)$ 로 색상과 위치를 합한 5차원 벡터로, 유사도를 에지 가중치로 사용한다.
- cut: 영역 분할의 좋은 정도를 측정해주는 목적함수. (나쁜 분할이면 크다)

$$cut(C_1, C_2) = \sum_{v_p \in C_1, v_q \in C_2} s_{pq}$$

- C_1, C_2 의 영역이 클수록 둘 사이의 에지가 많아 cut도 커짐 ⇒ cut을 정규화하여 영역의 크기에 종립이 되도록 만듦

$$ncut(C_1, C_2) = \frac{cut(C_1, C_2)}{cut(C_1, C)} + \frac{cut(C_1, C_2)}{cut(C_2, C)}$$

- ncut을 목적 함수로 사용해 분할을 최적화 문제로 풀 수 있다.

◆ 정규화 절단 알고리즘으로 영역 분할하기

```
import skimage
import numpy as np
import cv2 as cv
import time
```

```

coffee=skimage.data.coffee()

start=time.time()
#분할하는 데 걸리는 시간을 측정해 출력
slic=skimage.segmentation.slic(coffee,compactness=20,n_segments=600,start_label=1)
# 영상을 600개의 슈퍼 화소로 분할해 slic에 저장
g=skimage.future.graph.rag_mean_color(coffee,slic,mode='similarity')
# rag_mean_color함수: 슈퍼 화소를 노드로 사용하고, 유사도를 에지 가중치로 사용한 그래프를 저장
ncut=skimage.future.graph.cut_normalized(slic,g) # 정규화 절단
#ncut은 픽셀에 영역의 번호를 부여한 맵
print(coffee.shape,' Coffee 영상을 분할하는데 ',time.time()-start,'초 소요')

marking=skimage.segmentation.mark_boundaries(coffee,ncut)
# 영역 분할 정보를 담은 ncut 맵 이용해 영역 경계 표시
ncut_coffee=np.uint8(markng*255.0)
# 0~1 사이의 실수를 가진 marking을 0~255사이의 uint8형으로 변환
cv.imshow('Normalized cut',cv.cvtColor(ncut_coffee,cv.COLOR_RGB2BGR))

cv.waitKey()
cv.destroyAllWindows()

```

4.5 대화식 분할

능동 외곽선

- 초기 곡선을 지정하고 점점 확장하면서 물체 외곽선으로 접근하는 방법 - 스네이크
- 스네이크 알고리즘 - 에너지를 최소로 하는 최적의 곡선을 찾는 최적화문제

$$\hat{g} = \operatorname{argmin} E(g)$$

$$E(g) = \sum_{l=0}^n (e_{image}(g(l)) + e_{internal}(g(l)) + e_{domain}(g(l)))$$

영상 e : 곡선이 에지에 위치하게 유도(에지 강도 사용), 내부e : 곡선이 매끄럽게 변하는 방향이 되도록 유도(곡률 사용), 도메인e: 분할하려는 특정 물체의 모양 정보를 잘 유지하도록 유도.

- 사용자 지정해준 초기 곡선에서 g를 찾아가는 과정을 수렴할 때까지 반복

[알고리즘 4-1] 스네이크로 물체 분할

입력: 명암 영상, 임계값 T

출력: 최적 곡선 \hat{g}

1. 사용자 입력을 받아 초기 곡선 g 를 설정한다.
2. while TRUE
3. $moved=0$
4. for $i=0$ to $n-1$
5. for $g(i)$ 의 9개 이웃점 각각에 대해 // 자신과 8-이웃을 포함한 9개 점
6. $g(i)$ 를 이웃점으로 이동한 곡선의 에너지 E 를 식 (4.11)로 구한다.
7. if 에너지가 최소인 점이 $g(i)$ 와 다르면
8. $g(i)$ 를 최소점으로 이동하고 $moved$ 를 1 증가시킨다.
9. if $moved < T$ // 곡선의 이동량이 임계치보다 작으면 수렴했다고 간주하고 탈출
10. break

GrabCut

1. 사용자가 붓으로 [파란 pixel - 물체, 빨간 pixel - 배경]을 초기 지정
2. 이를 pixel로 물체 히스토그램과 배경 히스토그램을 만든다
3. 나머지 pixel들은 두 히스토그램과 유사성을 따져 물체/배경일 확률을 추정하고 물체/배경 영역 갱신
4. 새로운 정보로 히스토그램 다시 만들고 영역 갱신하는 과정 반복
5. 영역이 거의 변하지 않으면 수렴으로 간주하고 멈춤

◆ GrabCut으로 물체 분할 - 사용자가 봇칠한 정보를 이용하여 물체를 분할한다.

```
import cv2 as cv
import numpy as np

img=cv.imread('soccer.jpg') # 영상 읽기
img_show=np.copy(img) # 봇 칠을 디스플레이할 목적의 영상
# 원본 영상 img는 분할 알고리즘을 위해 원래 내용을 유지해야 하므로 복사해 별도의 배열 만들

# 사용자의 봇칠에 따라 물체 or 배경인지에 대한 정보를 기록할 mask
mask=np.zeros((img.shape[0],img.shape[1]),np.uint8) # 원본과 크기가 같은 mask배열 생성
mask[:, :] = cv.GC_PR_BGD # 모든 화소를 배경일 것 같음으로 초기화
# GC_PR_BGD: 배경일 것 같음_2. GC_PR_FGD: 물체일 것 같음_#
# GC_BGD: 배경으로 판정_0.GC_FGD: 물체로 판정_1

BrushSiz=9 # 붓의 크기
LColor,RColor=(255,0,0),(0,0,255) # 파란색(물체)과 빨간색(배경)

def painting(event,x,y,flags,param):
    if event==cv.EVENT_LBUTTONDOWN:
        cv.circle(img_show,(x,y),BrushSiz,LColor,-1) # 왼쪽 버튼 클릭하면 파란색을 기록
        cv.circle(mask,(x,y),BrushSiz,cv.GC_FGD,-1) # mask배열에 확실히 물체라는 표시를 봇칠한 곳에 기록.
    elif event==cv.EVENT_RBUTTONDOWN:
        cv.circle(img_show,(x,y),BrushSiz,RColor,-1) # 오른쪽 버튼 클릭하면 빨간색
        cv.circle(mask,(x,y),BrushSiz,cv.GC_BGD,-1)
```

```

        elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_LBUTTON:
            cv.circle(img_show,(x,y),BrushSiz,LColor,-1)# 左쪽 버튼 클릭하고 이동하면 파란색
            cv.circle(mask,(x,y),BrushSiz, cv.GC_FGD, -1)
        elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_RBUTTON:
            cv.circle(img_show,(x,y),BrushSiz,RColor,-1) # 오른쪽 버튼 클릭하고 이동하면 빨간색
            cv.circle(mask,(x,y),BrushSiz, cv.GC_BGD, -1)

        cv.imshow('Painting',img_show)

cv.namedWindow('Painting')
cv.setMouseCallback('Painting',painting)

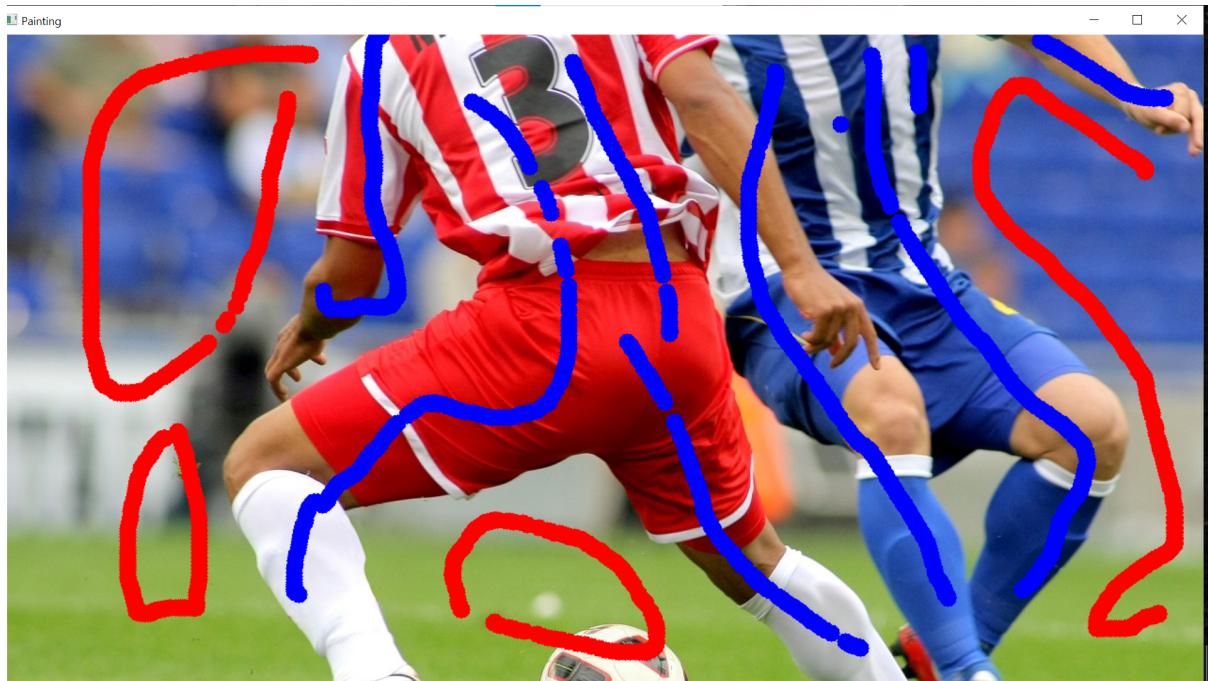
while(True):      # 끝 칠을 끝내려면 'q' 키를 누르고 분할을 시도하게 됨.
    if cv.waitKey(1)==ord('q'):
        break

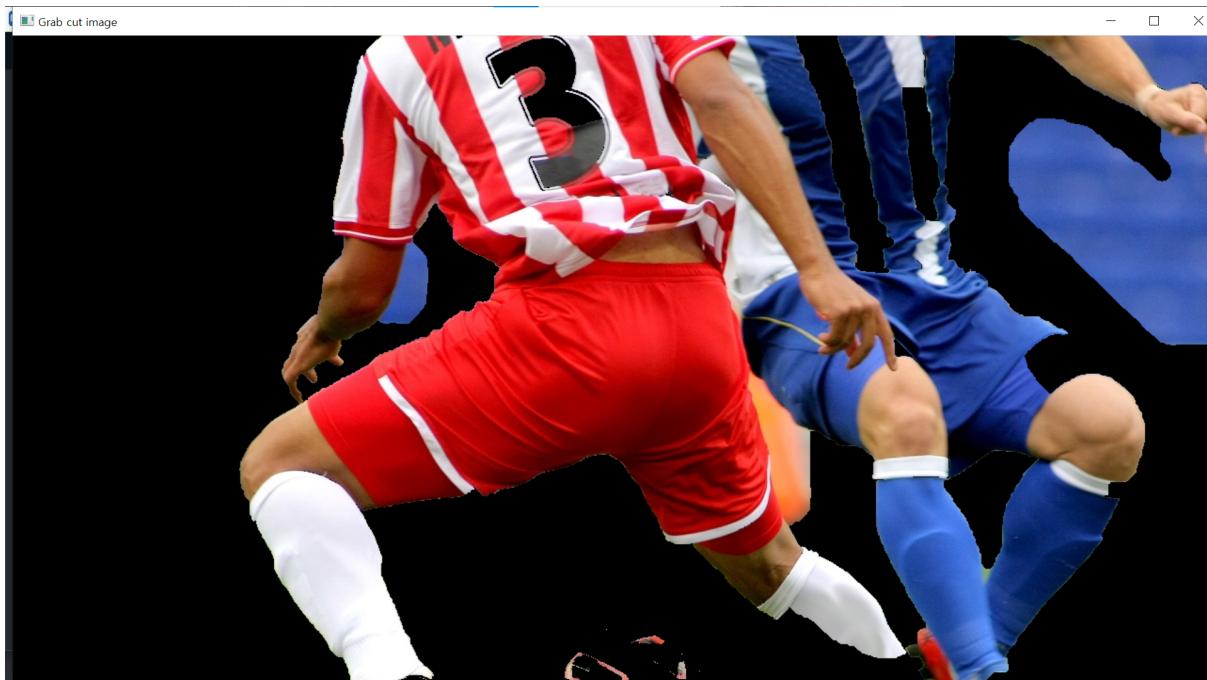
# 여기부터 GrabCut 적용하는 코드
background=np.zeros((1,65),np.float64) # grabcut함수가 내부에서 사용할 배경 히스토그램 0으로 초기화
foreground=np.zeros((1,65),np.float64) # 물체 히스토그램 0으로 초기화

cv.grabCut(img,mask,None,background,foreground,5,cv.GC_INIT_WITH_MASK)
# 함수 실행 끝나면 mask가 분할한 정보를 가진다.
mask2=np.where((mask==cv.GC_BGD)|(mask==cv.GC_PR_BGD),0,1).astype('uint8')
# 배경 또는 배경일 것 같음 = 0, 물체 또는 물체일 것 같음 = 1
grab=img*mask2[:, :, np.newaxis]
# 원본 영상과 mask2를 곱해서 배경에 해당하는 pixel을 검게 바꿈.
cv.imshow('Grab cut image',grab)

cv.waitKey()
cv.destroyAllWindows()

```





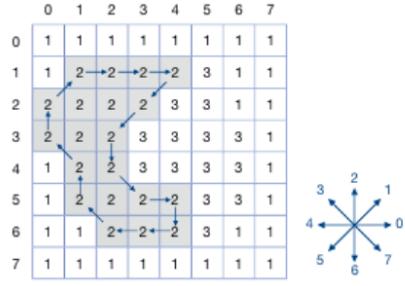
- Grabcut함수

```
cv.grabCut(img, mask, ROI, background, foreground, 반복횟수, cv.GC_INIT_WITH_MASK)
```

▼ 인수 설명

- img = 원본 영상
- mask = 사용자가 지정한 물체와 배경 정보를 가진 맵
 - 배경으로 지정한 픽셀은 0, 물체 지정한 픽셀은 1, 나머지 픽셀은 배경일 것 같음에 해당하는 2를 가짐
- ROI = 관심 영역 지정. None이면 전체영상을 대상으로 함
- background = 배경 히스토그램
- foreground = 물체 히스토그램
- cv.GC_INIT_WITH_MASK = 배경과 물체를 표시한 맵을 사용하라고 지시

4.6 영역 특징



(a) 영역의 레이블링



(b) 영역의 기하 변환

그림 4-17 영역의 레이블링과 기하 변환

◆ 특징의 불변성과 등변성

- 물체는 다양한 변환을 거쳐 영상에 나타남
 - 불변성(invariant): 변환해도 특징의 값이 변하지 않는 것
 - 성별은 나이에 불변, 근력은 나이에 불변X
 - 면적은 회전에 불변, 축소에는 불변X
 - 주축(물체의 중심축) 회전에 불변X, 축소에는 불변
 - 등변성(equivariant): \leftrightarrow 불변성. 특징이 어떤 변화에 따라 변한다.
 - 면적은 축소에 등변, 회전에는 등변X
 - 상황에 따른 특징 선택이 중요함.

◆ 모멘트와 모양 특징

- 영역 R의 모멘트. (y, x) 는 영역 R에 속하는 픽셀

$$m_{qp}(R) = \sum_{(y,x) \in R} y^q x^p \quad (4.13)$$

- 영역의 면적, 중심 그리고 중점을 이용한 중심 모멘트

$$\left. \begin{array}{l} \text{면적: } a = m_{00} \\ \text{중점: } (\dot{y}, \dot{x}) = \left(\frac{m_{10}}{a}, \frac{m_{01}}{a} \right) \end{array} \right\} \quad (4.14)$$

$$\mu_{qp} = \sum_{(y,x) \in R} (y - \dot{y})^q (x - \dot{x})^p \quad (4.15)$$

- 중심 모멘트를 이용한 열분산, 행분산, 일행분산

- 열 분산: pixel이 수직 방향으로 퍼진 정도
- 행 분산: pixel이 수평 방향으로 퍼진 정도
- 2번 영역은 열 분산은 크고 행 분산은 작다

$$\left. \begin{array}{l} \text{열 분산: } \nu_{cc} = \frac{\mu_{20}}{a} \\ \text{행 분산: } \nu_{rr} = \frac{\mu_{02}}{a} \\ \text{열행 분산: } \nu_{rc} = \frac{\mu_{11}}{a} \end{array} \right\} \quad (4.16)$$

- 중심 모멘트는 이동 불변, 크기나 회전 불변은 아님. 아래 식은 크기 불변__?

$$\eta_{qp} = \frac{\mu_{qp}}{\mu_{00}^{\left(\frac{q+p}{2}+1\right)}} \quad (4.17)$$

- 영역의 둘레(perimeter)와 둥근 정도(roundness)

$$\left. \begin{array}{l} \text{둘레: } p = n_{even} + n_{odd} \sqrt{2} \\ \text{둥근 정도: } r = \frac{4\pi a}{p^2} \end{array} \right\} \quad (4.18)$$

- 주축을 중심으로 영역을 회전하면 가장 적은 힘을 받고, 주축의 방향은 다음과 같다.

$$\text{주축의 방향: } \theta = \frac{1}{2} \arctan \left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}} \right) \quad (4.19)$$

♦ 텍스처 특징

- 텍스처: 일정한 패턴의 반복
- 텍스처가 세밀하면 많은 에지가 발생하고 거칠면 적게 발생하는 성질을 이용한다.
- 에지 정보의 통계를 텍스처 특징으로 취함
 - busy는 에지 픽셀 수를 전체 픽셀 수로 나누어 세밀함을 측정.
 - 에지의 강도와 방향의 분포도 텍스처 성질을 잘 반영한다.
 - mag: 에지 강도를 q단계로 양자화하여 구한 히스토그램
 - dir: 에지 방향을 8단계로 양자화해 구한 히스토그램

$$T_{edge} = \{busy, mag(i), dir(j)\}, 0 \leq i \leq q-1, 0 \leq j \leq 7 \quad (4.20)$$

- LBP(Local Binary Pattern): 중심 픽셀과 주위 픽셀의 픽셀값을 비교해 텍스처를 구함
 - 회색 칠한 픽셀을 조사하는 상황에서 주위 8개 픽셀과 값을 비교해서 회색 픽셀보다 크면 1, 작은 픽셀은 0으로 표시.
 - 빨간 선분이 가리키는 순서대로 이진수를 구성하고 십진수로 변환. 발생 가능한 수는 0~255
 - 영역에 속한 모든 픽셀을 조사한 다음 256개 값에 대한 발생빈도를 센 히스토그램을 구함 → 이렇게 256차원의 특징벡터인 LBP구함
- LBP는 작은 픽셀 값 변화에 민감함. 어떤 특정한 픽셀 값인 곳이 있을 때 이웃화소는 랜덤한 잡음이 발생하여 랜덤한 비트열이 생성될 수 있기 때문
- LTP(Local Ternary Pattern)는 임계값을 정하고 회색 픽셀 값-t보다 작으면 -1
회색 픽셀 값+t보다 크면 1
그렇지 않으면 0
을 부여해 3진 코드를 얻음 3진코드를 이진 코드 2개로 각각 변환하고 각각을 십진수로 변환한다. → 각각에 대해 히스토그램을 구하면 256개 칸인 히스토그램 2개를 얻음. 512차원 특징벡터 LTP 구함.

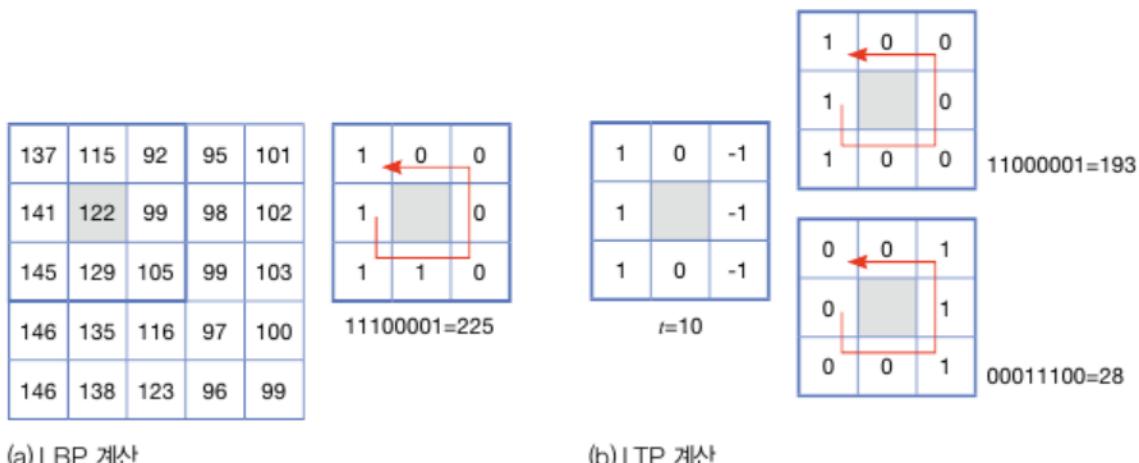


그림 4-18 LBP와 LTP 구하기

◆ 이진 영역의 특징을 추출하는 함수 사용하기

```

import skimage
import numpy as np
import cv2 as cv

orig=skimage.data.horse()
# 이 영상은 말이 차지한 영역을 False, 배경 영역은 True로 표시한 영상
img=255-np.uint8(orig)*255
cv.imshow('Horse',img)

contours,hierarchy=cv.findContours(img,cv.RETR_EXTERNAL,cv.CHAIN_APPROX_NONE)
# 경계선 추출
# len(contours) = 1->경계선 하나뿐

```

```

#영상에 경계선 표시하고 디스플레이
img2=cv.cvtColor(img, cv.COLOR_GRAY2BGR)    # 컬러 디스플레이용 영상

cv.drawContours(img2,contours,-1,(255,0,255),2)
# 3번째 인수 -1은 contours에 있는 경계선을 모두 표시하라
# 4,5 인수는 색깔과 선의 두께
cv.imshow('Horse with contour',img2)

contour=contours[0]
#이후 코드는 경계선 하나를 처리하므로 0번 요소를 꺼내 저장함

m=cv.moments(contour)      # 몇 가지 특징
area=cv.contourArea(contour)
cx,cy=m['m10']/m['m00'],m['m01']/m['m00']
perimeter=cv.arcLength(contour,True)
roundness=(4.0*np.pi*area)/(perimeter*perimeter)
print('면적=',area,'중점(','cx',' ','cy',')', '\n둘레=',perimeter,'\n둥근 정도=',roundness)

img3=cv.cvtColor(img, cv.COLOR_GRAY2BGR)    # 컬러 디스플레이용 영상

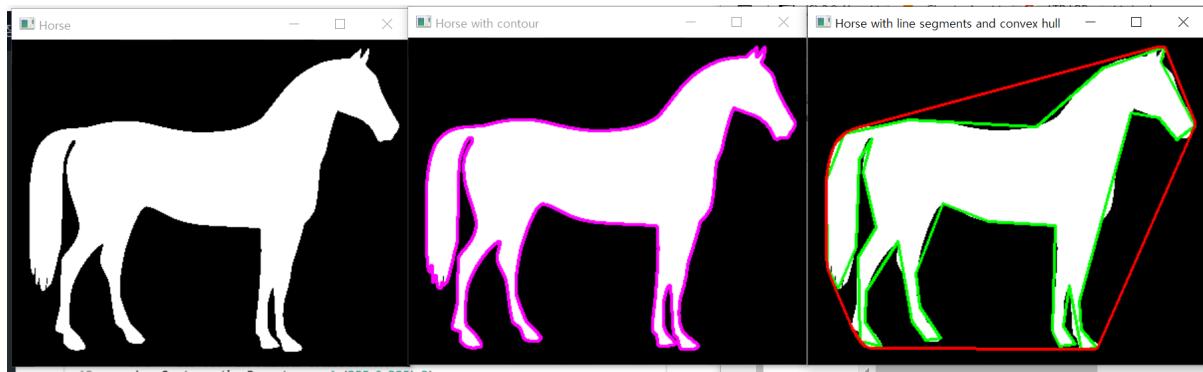
contour_approx=cv.approxPolyDP(contour,8,True)  # 경계선을 직선 근사
# 8은 임계값, True는 폐곡선
cv.drawContours(img3,[contour_approx],-1,(0,255,0),2)
# 근사한 결과를 녹색으로 표시

hull=cv.convexHull(contour)      # 볼록 헬
hull=hull.reshape(1,hull.shape[0],hull.shape[2])
#drawContours 함수에 입력가능한 형태로 바꿈.
cv.drawContours(img3,hull,-1,(0,0,255),2)

cv.imshow('Horse with line segments and convex hull',img3)

cv.waitKey()
cv.destroyAllWindows()

```



- 외곽선 잘 추출됨. 물체 경계를 다각형으로 잘 표현하는 중

Chapter 5 지역 특징

- 대응점 문제: 이웃한 영상에 나타난 같은 물체의 같은 곳을 쌍으로 묶어주는 일
- 에지 특징, 영역 특징은 대응점 찾기에 부족 → 지역 특징(local features)

5.1 발상

- 버스 추적에 효과적인 방법: 두 영상에서 특징점을 추출하고 매칭을 통해 해당하는 특징점 쌍 찾기(대응점 문제)
- 왼쪽 영상에서 추출된 특징점이 오른쪽 영상에도 추출되어야 함 → 이러한 조건을 높은 확률로 만족하면 반복성이 좋음
- 특징점에서 추출한 두 특징 벡터는 비슷해야 함. → 매칭을 통해 둘이 쌍이라고 알아낼 수 있음.
- 물체에 변화가 발생해도 이러한 조건이 만족하는 특징 → 불변성이 좋다.



그림 5-2 대응점 찾기(MOT-17-14-SDP 동영상의 70번째와 83번째 영상)

- 지역 특징: 좁은 지역을 보고 특징점 여부를 판정.
 - 특징점이 물체의 실제 모퉁이에 위치할 필요는 없음
 - 물체의 같은 곳이 두 영상 모두에서 특징점으로 추출되어야 함 → 반복성 중시
- 지역특징의 종류
 - 위치, 스케일 → detection 단계
 - 방향, 특징기술자(feature descriptor) → 기술 단계
- 지역 특징의 조건
 - 반복성
 - 불변성
 - 분별력: 물체의 다른 곳에서 추출된 특징과 두드러제가 달라야 함
 - 지역성: 작은 영역을 중심으로 특징 벡터를 추출해야 물체에 가림이 발생해도 매칭이 안정적으로 동작 함
 - 적당한 양: 물체 추적하려면 몇 개의 대응점만 있으면 된다. 대응점은 오류를 내포할 가능성이 있으므로 특징점이 많으면 더 정확하게 추적 가능. 대신 너무 많으면 계산 시간 과다
 - 계산 효율

→ 넓은 영역에서 특징 벡터 추출하면 분별력이 높아지지만 지역성이 낮아져 가림에 대처하지 못함.

5.2 이동과 회전 불변한 지역 특징

5.2.1 모라벡 알고리즘

5.3 스케일 불변한 지역 특징

5.4 SIFT

5.5 매칭

5.6 호모그래피 추정