컴퓨터 비전과 딥러닝

챕터 3, 4 정리

AI 학과 2444337 손주안

Chapter 1

3.1 디지털 영상 기초

3.2 이진 영상

3.3 점 연산

3.4 영역 연산

3.5 기하 연산

3.6 OpenCV의 시간 효율

Chapter 2

4.1 에지 검출

4.2 캐니 에지

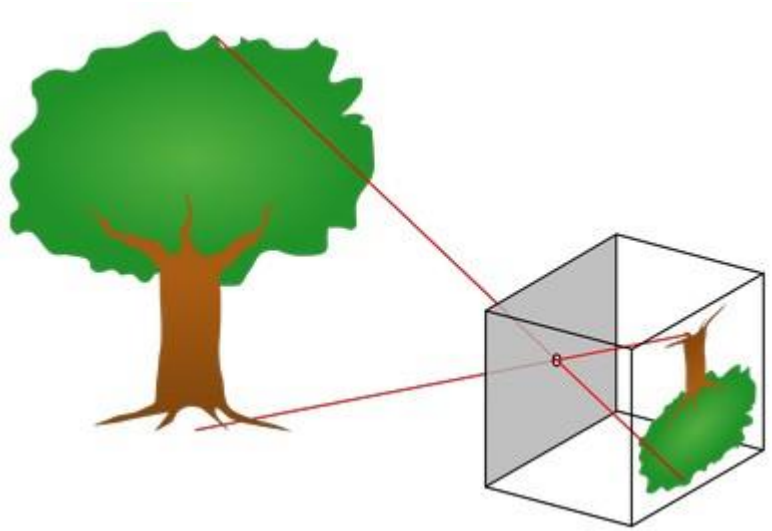
4.3 직선 검출

4.4 영역 분할

4.5 대화식 분할

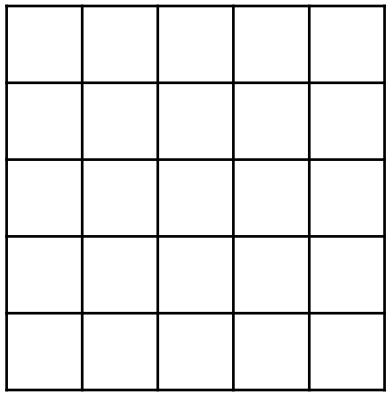
4.6 영역 특징

디지털 영상 기초



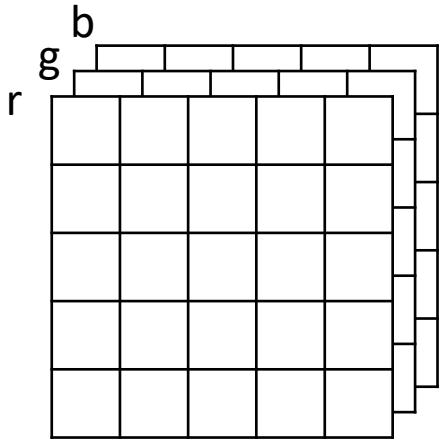
- 디지털 가메라는 실제 세상에 존재하는 피사체를 일정한 간격으로 샘플링하고 명암을 일정한 간격으로 **양자화**하는 과정을 통해 디지털 영상을 획득한다.
- 물체에서 반사된 빛은 카메라의 작은 구멍을 통해 카메라의 CCD 센서에 맺히고, CCD 센서는 빛을 디지털 신호로 변환하여 메모리에 저장한다. 이 과정에서 **샘플링**과 **양자화**를 수행한다.
- 디지털 영상의 좌표는 원점이 왼쪽 위에서 시작하며 (y,x)식 표기를 사용한다. 하지만 영상을 저장하는 배열에서 화소의 위치를 지정할 때 외에는 주로 (x,y) 로 표기한다.

디지털 영상 기초 | 다양한 종류의 영상



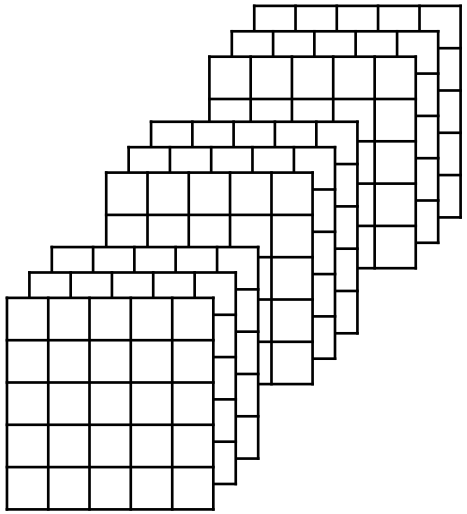
명암 영상

- 채널이 하나로, 2차원 구조로 표기한다.



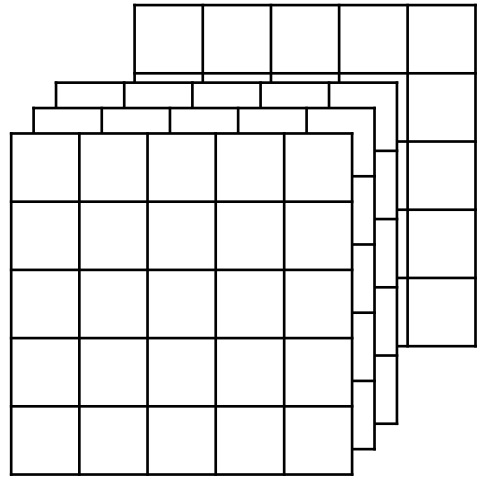
컬러 영상

- 채널이 3개로, 3차원 구조로 표기한다.



컬러 동영상

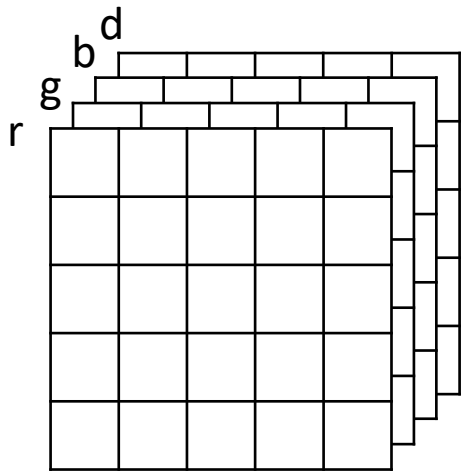
- 4차원의 구조로 표기한다.



다분광, 초분광 영상

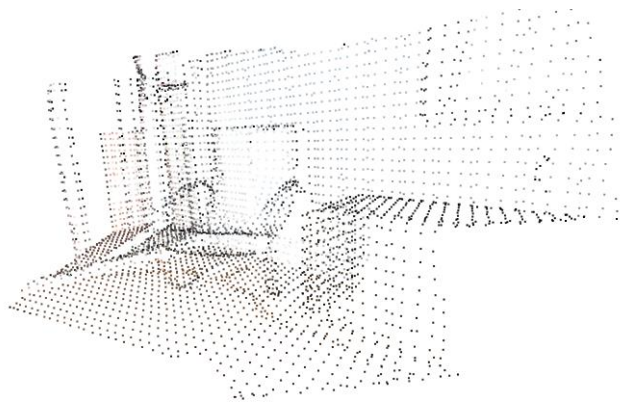
- 다분광은 10개 가량, 초분광은 수백, 수천 개의 채널이 있다.
- 이 역시 3차원으로 표현한다.

디지털 영상 기초 | 다양한 종류의 영상



RGB-D 영상

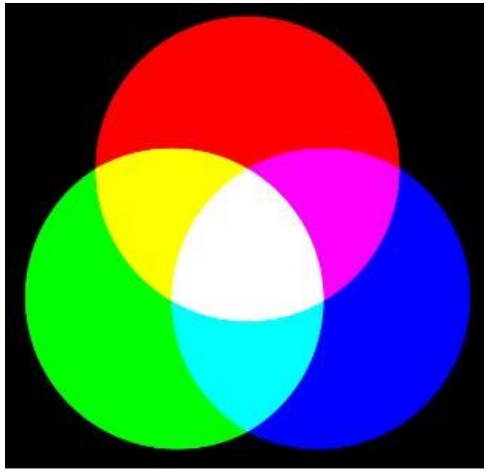
- RGB 센서와 깊이 센서가 통합된 카메라로 획득한다.

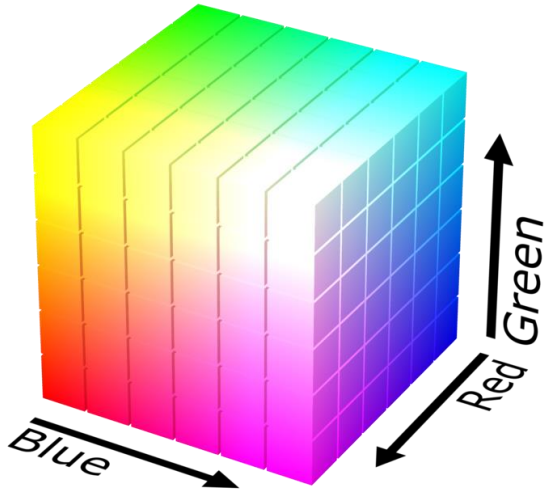


점 구름 영상

- 거리는 **깊이** 또는 **레인지**라고 부르며 거리를 표현한 영상을 깊이 영상 또는 레인지 영상이라고 부른다.
- 라이다는 어떤 조건을 만족하는 점의 거리만 측정하므로 영상이 완벽한 격자 구조가 아니다. 그러므로 획득한 거리 데이터를 점 구름 영상으로 표현한다.
- 점 구름은 점의 좌표 (x,y)와 해당 점까지의 거리 d를 (x_i, y_i, d_i)의 형식으로 저장한다.

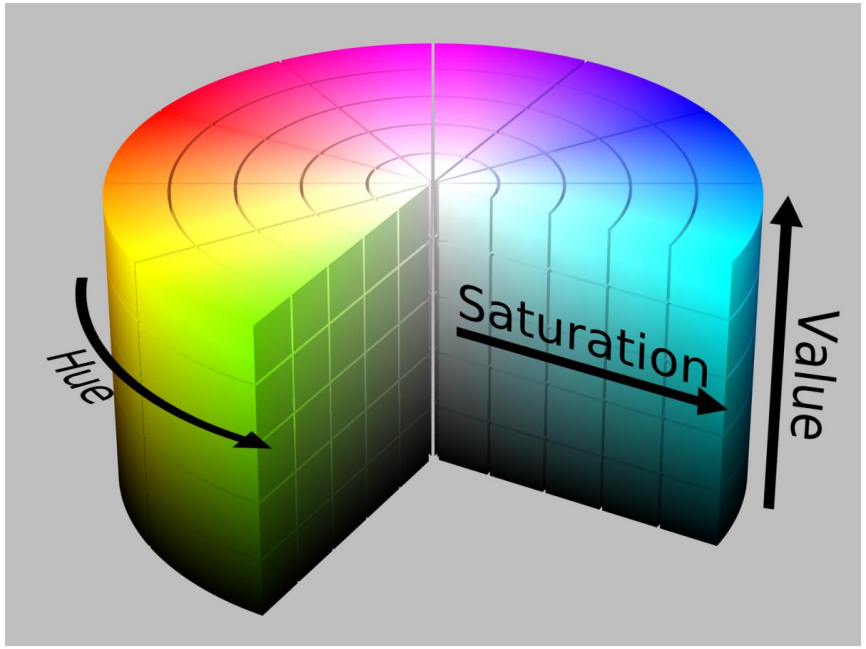
디지털 영상 기초 | 컬러 모델





- 빨강, 녹색, 파랑의 세 요소의 범위를 **0부터 1**까지로 설정해 세상 모든 색을 나타낼 수 있다. 이 모든 색을 **RGB 큐브**에 넣는다.
- RGB 큐브를 디지털 영상으로 표현할 때는 주로 **256 단계**로 양자화하여 한 바이트로 표현한다.

디지털 영상 기초 | 컬러 모델



HSV 컬러 모델

- H는 색상, S는 채도, V는 명암을 표현한다.
- RGB 모델에서는 빛의 밝기가 R,G,B 세 요소에 섞여 있으므로 빛이 약해지면 세 값이 모두 작아진다.

디지털 영상 기초 | 컬러 모델

```
cv2.imshow('upper left half',
img[0:img.shape[0]//2, 0:img[1]//2,:])

cv2.imshow('r_channel', img[:,:,0])
```

| img[0:img.shape[0]//2, 0:img[1]//2,:]

- 영상의 왼쪽 위 ¼를 출력할 수 있다.
- **B 채널 영역은 0번**, **G 채널은 1번**, **R 채널은 2번**을 입력하여 출력할 수 있다.
- R 채널을 출력했을 땐 붉은 부분이, G 채널은 초록색 부분이, B 채널은 푸른 부분이 밝게 나타난다.

디지털 영상 기초 | 실행 결과

```python
import cv2
import sys

img = cv2.imread('soccer.jpg')

if img is None:
    sys.exit('nothing')

cv2.imshow('img',img)
cv2.imshow('img2',img[0:img.shape[0]//2, 0:img.shape[1]//2,:])

cv2.imshow('blue',img[:,:,0])
cv2.imshow('green',img[:,:,1])
cv2.imshow('red',img[:,:,2])

cv2.waitKey()
cv2.destroyAllWindows()
```

| img

| img2

| blue

| green

| blue

이진 영상

$$B(j,i) = \begin{cases} 1, f(j,i) \geq T \\ 0, f(j,i) < T \end{cases}$$

이진 영상

- 화소가 **0(흑) 또는 1(백)**인 영상
- 컴퓨터 비전에서는 에지를 검출한 후 에지를 1로 표현하거나 물체를 검출한 후 **물체를 1**, **배경을 0**으로 표시하는 등의 일에 이진 영상을 활용한다.
- 명암 영상을 이진화하려면 임곗값 T보다 큰 화소는 1, 그렇지 않은 화소는 0으로 바꾸면 된다.
- 위 식으로 이 과정을 정의하는데, f는 기존 명암 영상이고 B는 이진 영상이다.

이진 영상

```
Import matplotlib.pyplot as plt
```

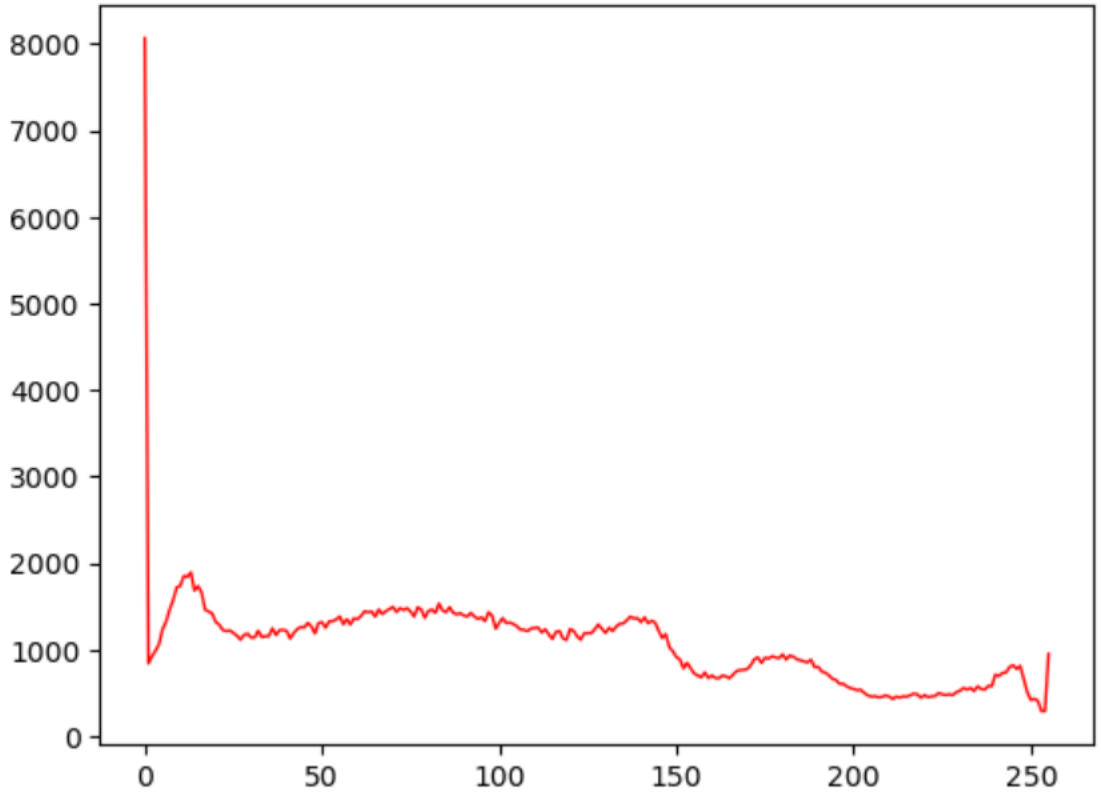
```
H = cv2.calcHist([img], [2],None,[256],
[0,256])
Plt.plot(h,color = ‘r’, linewidth = 1)
```

calcHist 함수

- 히스토그램을 구한다. 인수를 리스트로 주어야 하며, 첫 번째 인수는 영상, 두 번째 인수는 영상의 채널 번호다.
- 세 번째 인수는 히스토그램을 구할 영역을 지정하는 마스크로, None일 경우 전체 영상에서 히스토그램을 구한다.
- 네 번째 인수는 히스토그램의 칸의 수를 지정하는데, 명암 단계가 256이므로 256을 지정한다.
- 다섯 번째 인수는 세어 볼 명암값의 범위를 지정한다. [0,128]로 지정하면 128 이상의 값은 세지 않는다.

이진 영상

```python
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('soccer.jpg')

h = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(h,color='r',linewidth=1)
```



이진 영상

$$t = argmin\, J(t)$$
$$t \in \{0,1,2,\dots,L-1\}$$

```
T, b=
cv2.threshold(img[:,:,2],0,255,cv2.THRESH.BINARY + cv2.THRESH_OTSU)
```

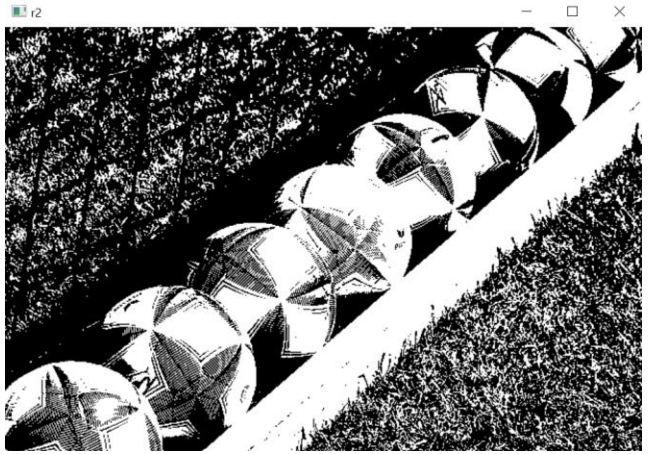
- 식은 명암값에 대해 목적 함수 J를 계산하고 J가 최소인 명암값을 최적값 t로 정한다. 이 t를 입곗값 T로 사용해 이진화한다.
- Threshold 함수는 알고리즘이 찾은 최적의 임곗값과 이진화된 영상을 반환한다. 각각 t와 b에 저장된다.

이진 영상

```python
import cv2
import sys

img = cv2.imread('soccer.jpg')

t, bin_img = cv2.threshold(img[:,:,2],0,255,cv2.THRESH_BINARY +cv2.THRESH_OTSU)
print('오츄 알고리즘이 찾은 최적 임곗값',t)

cv2.imshow('r',img[:,:,2])
cv2.imshow('r_binarization', bin_img)

cv2.waitKey()
cv2.destroyAllWindows()
```

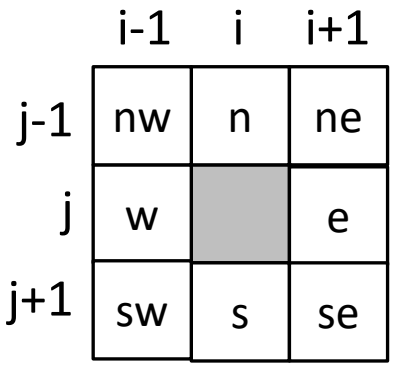
오츄 알고리즘이 찾은 최적 임곗값 **112.0**

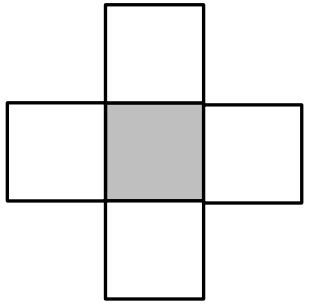
r



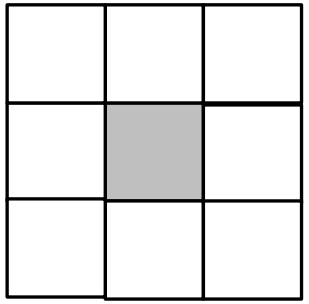
r_binarization

이진 영상 | 연결 요소





| 4-연결성



| 8-연결성

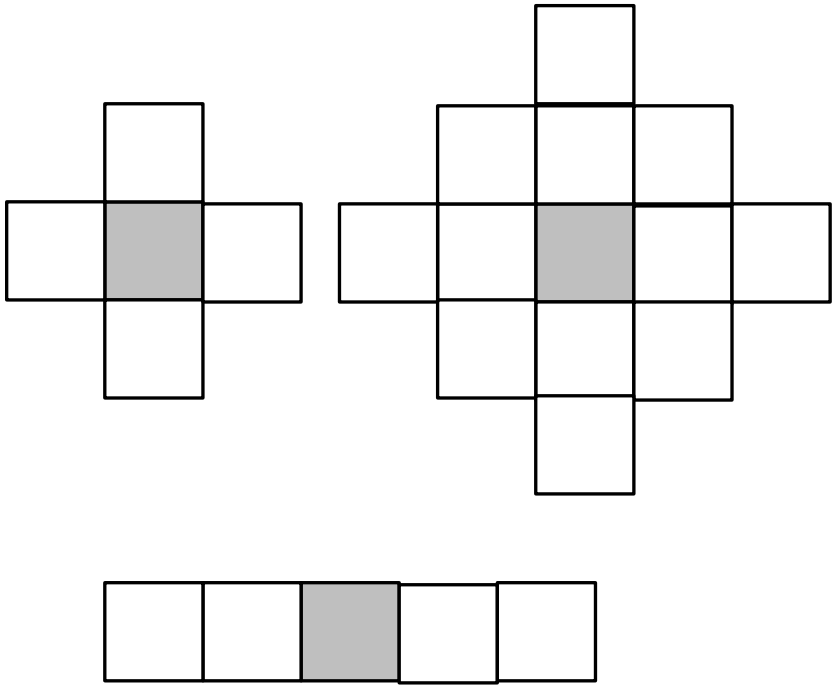
| 4-연결성

- 상하좌우에 있는 4개 이웃만 연결된 것으로 간주한다.

| 8-연결성

- 대각선에 위치한 화소도 연결된 것으로 간주한다.

이진 영상 | 연결 요소



모폴로지가 사용하는 구조 요소

모폴로지 연산

- 영상을 하나의 물체가 여러 영역으로 분리되거나 다른 물체가 한 영역으로 붙는 경우 등이 발생하는데 이런 부작용을 누그러뜨리기 위해 사용한다.
- 구조 요소를 이용해 영역의 모양을 조작한다.

팽창

- 입력 영상에 특정 위치에 구조 요소를 씌어 좌우 이웃 화소가 1로 바뀐다.

침식

- 구조 소에 해당하는 점에 1이 아닌 화소가 있으면 0으로 바뀐다.

열림

- 침식한 결과에 팽창을 적용하는 연산

닫힘

- 팽창한 결과에 침식을 적용하는 연산

이진 영상

```
Se = np.uint8([[0,0,1,0,0],
               [0,1,1,1,0],
               [1,1,1,1,1],
               [0,1,1,1,0],
               [0,0,1,0,0]])

b_dilation = cv2.dilate(b,se,iterations = 1)

b_erosion = cv2.erode(b,se,iterations = 1)

b_closing = cv2.dilate(cv2.erode(b,se,iterations = 1),se, iterations=1)
```

- se는 구조 요소를 저장한 것이다.
- **Dilate**는 **팽창** 연산, **erode**는 **침식** 연산으로 맨 아래의 경우 침식한 영상을 팽창하므로 열림 연산에 해당한다.
- iterations는 적용 회수이다.

이진 영상 | 실행 결과

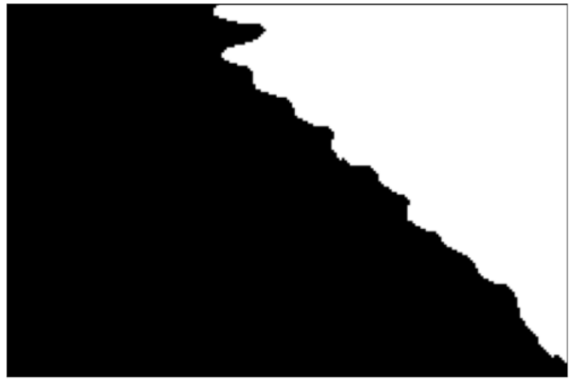
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

wing = cv2.imread('wing.png', cv2.IMREAD_UNCHANGED)

t,bin = cv2.threshold(wing[:,:,3],0,255,cv2.THRESH_BINARY + cv2.THRESH_OTSU)
plt.imshow(bin,cmap = 'gray'), plt.xticks([]), plt.yticks([])
plt.show()

b = bin[bin.shape[0]//3:2 * bin.shape[0]//3,0:bin.shape[0]//2+1]
plt.imshow(b,cmap='gray'),plt.xticks([]),plt.yticks([])
plt.show()

se = np.uint8([[0,0,1,0,0],
               [0,1,1,1,0],
               [1,1,1,1,1],
               [0,1,1,1,0],
               [0,0,1,0,0]])

b_dilation = cv2.dilate(b,se,iterations=1)
plt.imshow(b_dilation,cmap='gray'),plt.xticks([]),plt.yticks([])
plt.show()
```

```
b_erosion = cv2.erode(b,se,iterations=1)
plt.imshow(b_erosion,cmap='gray'),plt.xticks([]),plt.yticks([])
plt.show()

b_closing = cv2.erode(cv2.dilate(b,se,iterations=1),se,iterations=1)
plt.imshow(b_closing,cmap='gray'),plt.xticks([]),plt.yticks([])
plt.show()
```

이진 영상 | 실행 결과

| 잘라낸 영상

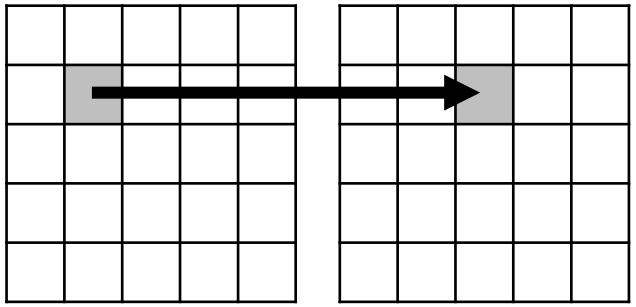
| 팽창



| 침식

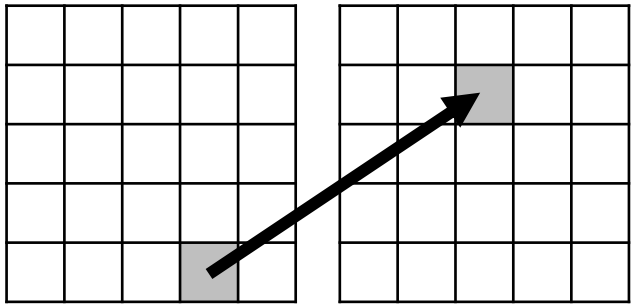
| 닫힘

점 연산



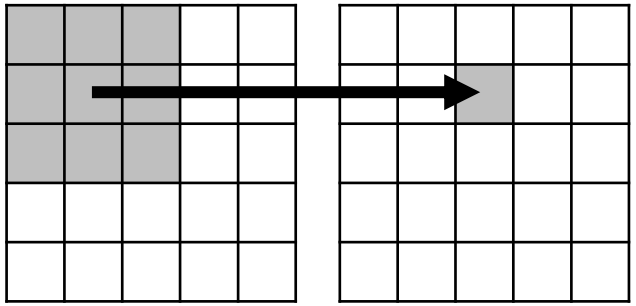
점 연산

자기 자신으로부터 값을 받는다.



기하 연산

기하학적 변환에 따라 다른 곳으로부터 값을받는다.



영역 연산

이웃 화소의 값을 보고 새로운 값을 결정한다.

- 화소 입장에서 영상 처리 연산이란 화소가 새로운 값을 받는 과정이다.
- 새로운 값을 어디에서 받는지에 따라 점 연산, 영역 연산, 기하 연산의 세 종류로 구분한다.

점 연산

$$f'(j,i) = \begin{cases} \min(f(j,i) + a, L-1) \\ \max(f(j,i) - a, 0) \\ (L-1) - f(j,i) \end{cases}$$

- 위 식을 이용해 영상의 밝기를 조정할 수 있다.
- 첫 번째 식은 **양수 a**를 더해 밝게 만들며 화소가 가질 수 있는 최댓값 L-1을 넘지 않도록 **min**을 취한다.
- 두 번째 식은 **양수 a**를 빼서 어둡게 만들며 음수를 방지하기 위해 **max**를 취한다.
- 마지막 식은 L-1에서 기존 명암값을 빼서 반전시킨다.

점 연산

```
def gamma(f,gamma = 1.0)
    f1 = f/255.0
    return np.uint8(255 *(f1**gamma))

gc = np.hstack((gamma(img,0.5),
(gamma(img,0.75), (gamma(img,1.0),
(gamma(img,2.0), (gamma(img,3.0)))
```

- Gamma의 값을 변화시켜 함수를 적용한 결과 원본 영상인 gamma = 1.0일 때를 기준으로 값이 낮을수록 밝고 높을수록 어둡다.

점 연산 | 실행 결과

```python
import cv2
import numpy as np

img = cv2.imread('soccer.jpg')
img = cv2.resize(img,dsize=(0,0),fx=0.25,fy=0.25)

def gamma(f,gamma=1.0):
    f1 = f/255.0
    return np.uint8(255*(f1**gamma))

gc = np.hstack((gamma(img,0.5),gamma(img,0.75),gamma(img,1.0),gamma(img,2.0),gamma(img,3.0)))
cv2.imshow('gamma',gc)
cv2.waitKey()
cv2.destroyAllWindows()
```

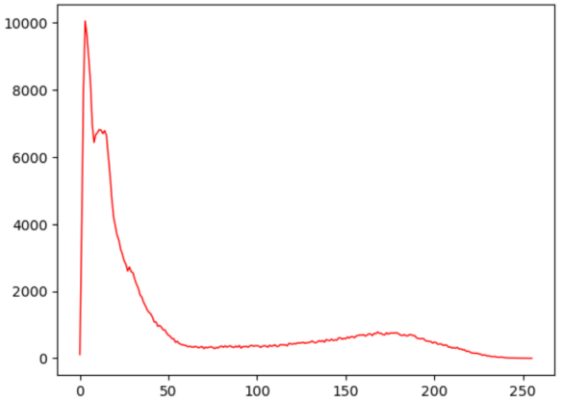
점 연산

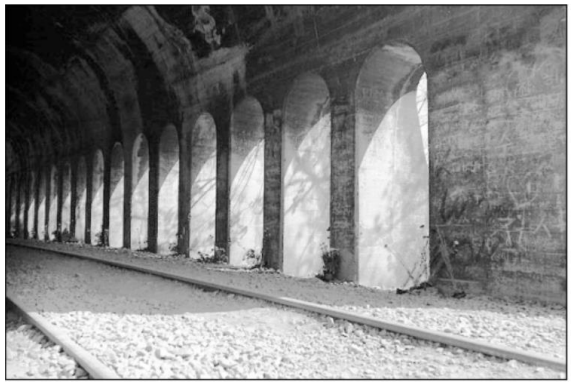
```
gray = cv2.cvtColor(img,cv.COLOR_BGR2GRAY)

plt.imshow(gray,cmap='gray'), plt.xticks([]),
plt.yticks([]), plt.show()

h = cv2.calcHist([gray], [0], None, [256], [0, 256])
plt.plot(h, color = ‘r’, linewidth = 1), plt.show()

equal = cv2.equalizeHist(gray)
plt.imshow(equal, cmap = 'gray’), plt.xticks([]),
plt.yticks([]), plt.show()

h = cv2.calcHist([equal], [0], None, [256], [0,256])
plt.plot(h, color='r’, linewidth=1), plt.show()
```

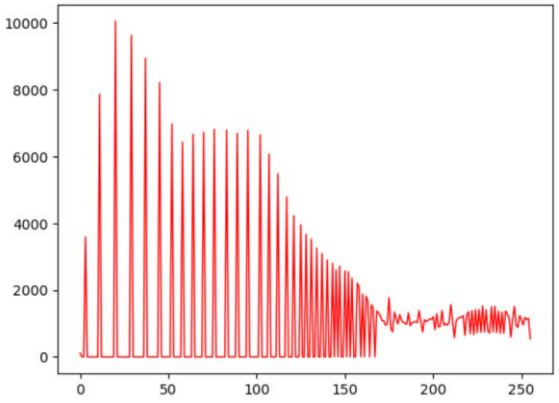
- 히스토그램 평활화는 히스토그램이 평평하게 되도록 영상을 조작해 영상의 **명암 대비를 높이는 기법**이다.
- 히스토그램 평활화를 적용한 영상은 **선명해지며**, 히스토그램이 더 평평해졌음을 알 수 있다.

점 연산 | 실행 결과

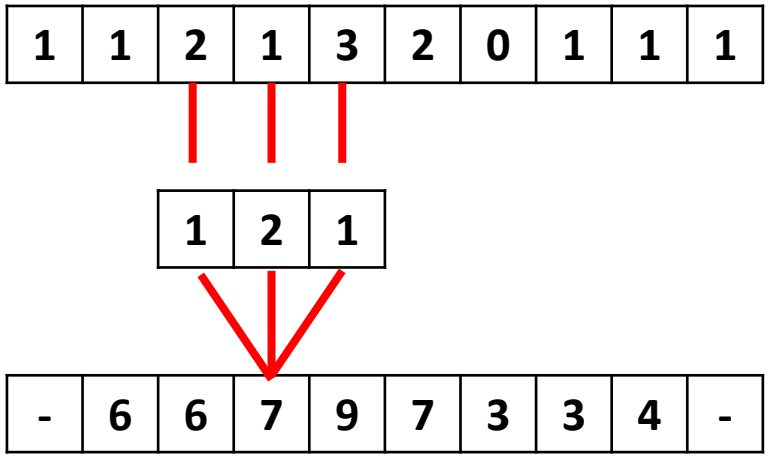
```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('dark2.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
plt.imshow(gray,cmap='gray'), plt.xticks([]), plt.yticks([]),plt.show()

h = cv2.calcHist([gray],[0],None,[256],[0,256])
plt.plot(h,color='r',linewidth=1), plt.show()

equal = cv2.equalizeHist(gray)
plt.imshow(equal,cmap='gray'),plt.xticks([]),plt.yticks([]),plt.show()

h = cv2.calcHist([equal],[0],None,[256],[0,256])
plt.plot(h,color='r',linewidth=1),plt.show()
```







영역 연산



컨볼루션

- 입력 영상 f의 각 화소에 필터를 적용해 곱의 합을 구하는 연산이다.
- 필터는 **가장자리 화소에 씌우면** 필터의 일부가 밖으로 나가기 때문에 적용할 수 없어 가장자리를 –로 표시한다.
- 가장자리에 적용하려면 원래 영상 f에 덧대기를 하면 된다.
- **0 덧대기**는 필요한 만큼의 가장자리를 확장하여 0으로 채우고, **복사 덧대기**는 가장자리 화소의 값으로 채운다.

영역 연산

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

1차원 가우시안

$$g(y,x) = \frac{1}{\sigma^2\sqrt{2\pi}} e^{-\frac{y^2+x^2}{2\sigma^2}}$$

2차원 가우시안

- 영상의 잡음은 스무딩 필터로 컨볼루션하여 누그러뜨릴 수 있다.
- 가우시안 함수는 보통 **6σ** 이상의 정수 중에 **가장 작은 홀수**를 필터 크기로 정한다.
- $\sigma = 0.7$일 경우 4.2 이상의 정수 중 가장 작은 홀수는 5이므로 5*5 필터를 사용한다.

영역 연산

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

0.0030	0.0133	0.0219	0.0133	0.0030
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0030	0.0133	0.0219	0.0133	0.0030

스무딩 필터

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

샤프닝 필터

스무딩 필터

- 잡음을 제거하는 효과가 있지만 물체의 경계를 흐릿하게 만드는 **블러링**이라는 부작용이 있다.

샤프닝 필터

- 스무딩 필터와 반대 작용을 하여 에지를 선명하게 하여 물체의 식별을 도우나 **잡음을 확대**한다는 부작용이 있다.

영역 연산

-1	0	0
0	0	0
0	0	1

-1	-1	0
-1	0	1
0	1	1

엠보싱 필터

엠보싱 필터

- 오른쪽 아래 화소에서 왼쪽 위 화소를 빼는 연산으로 -255~255 범위의 값이 발생한다.

- 0~255 범위를 사용하기 위해서는 -255~255 사이의 일정 부분을 포기하여 -127~383 범위로 변환한다. 이후 np.clip 함수를 통해 0~255 범위로 축소한다.

np.clip

- np.clip(a,p,q)는 a가 p보다 작으면 p로, q보다 크면 q로 바꾸고 그렇지 않으면 원래 값을 유지한다.

- 엠보싱 필터의 경우 np.clip(a+128, 0, 255)를 적용해 0 ~255 범위로 변환한다.

영역 연산

```
Smooth = np.hstack((cv2.GaussianBlur(gray, (5,5), 0.0), cv2.GaussianBlur(gray, (9,9), 0.0), cv2.GaussianBlur(gray, (15, 15), 0.0)))
```

- GaussianBlur 함수의 2번째 인수는 필터의 크기로, 필터가 커질수록 영상의 세부 내용이 사라진다.
- 3번째 인수는 표준편차 σ로, 0.0으로 설정할 경우 필터 크기를 보고 자동으로 설정하게 된다.

영역 연산

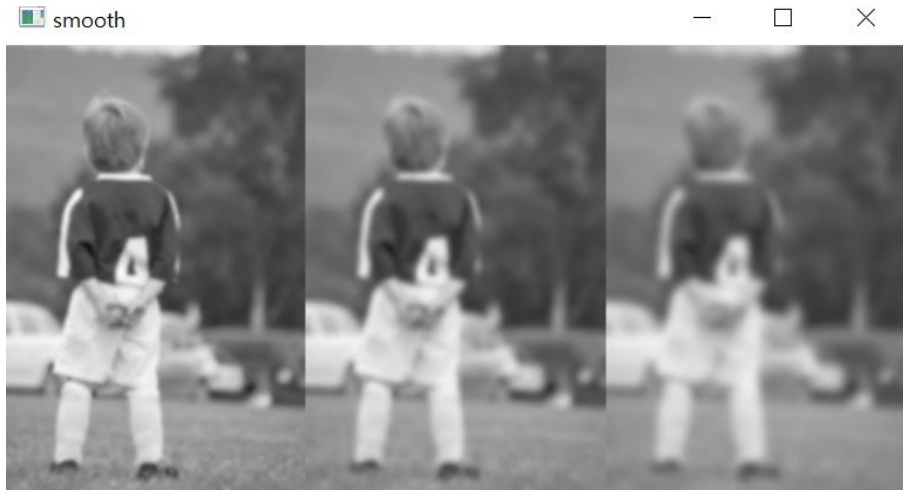
```
femboss = np.array([[-1.0,0.0,0.0],
                    [0.0,0.0,0.0],
                    [0.0,0.0,-1.0]])

gray16 = np.int16(gray)

emboss =
np.uint8(np.clip(cv2.filter2D(gray16,-
1,femboss)+128,0,255))

emboss_bad =
np.uint8(cv2.filter2D(gray16,-
1,femboss)+128)

emboss_worse =cv2.filter2D(gray16,-
1,femboss)
```

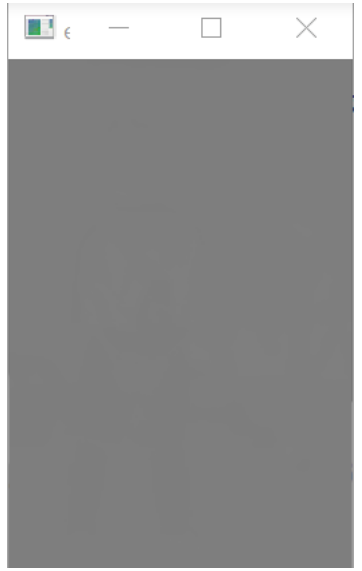
- Femboss는 엠보싱 필터를 정의한다.
- np.uint8 형의 경우 부호가 없는 1바이트 정수 형이다. np.int16 함수를 적용하면 부호가 있는 2바이트(16비트) 형으로 변환한다.
- filter2D 함수는 np.int16 형에서 동작해 음수까지 저장한다.
- Emboss_bad는

영역 연산 | 실행 결과

```
import cv2
import numpy as np

img = cv2.imread('soccer2.jpg')
img = cv2.resize(img,dsize=(0,0),fx=0.4,fy=0.4)
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
cv2.imshow('original',gray)

smooth = np.hstack((cv2.GaussianBlur(gray,(5,5),0.0),cv2.GaussianBlur(gray,(9,9),0.0),cv2.GaussianBlur(gray,(15,15),0.0)))

cv2.imshow('smooth',smooth)

femboss = np.array([[-1.0,0.0,0.0],
                    [0.0,0.0,0.0],
                    [0.0,0.0,-1.0]])

gray16 = np.int16(gray)
emboss = np.uint8(np.clip(cv2.filter2D(gray16,-1,femboss)+128,0,255))
emboss_bad = np.uint8(cv2.filter2D(gray16,-1,femboss)+128)
emboss_worse =cv2.filter2D(gray16,-1,femboss)

cv2.imshow('emboss',emboss)
cv2.imshow('emboss_bad',emboss_bad)
cv2.imshow('emboss_worse',emboss_worse)

cv2.waitKey()
cv2.destroyAllWindows()
```

영역 연산 | 실행 결과





기하 연산 | 동차 좌표, 동차 행렬

$$T(t_x, t_y) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

| 이동

- x 방향으로 t_x, y 방향으로 t_y만큼 이동한다.

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

| 회전

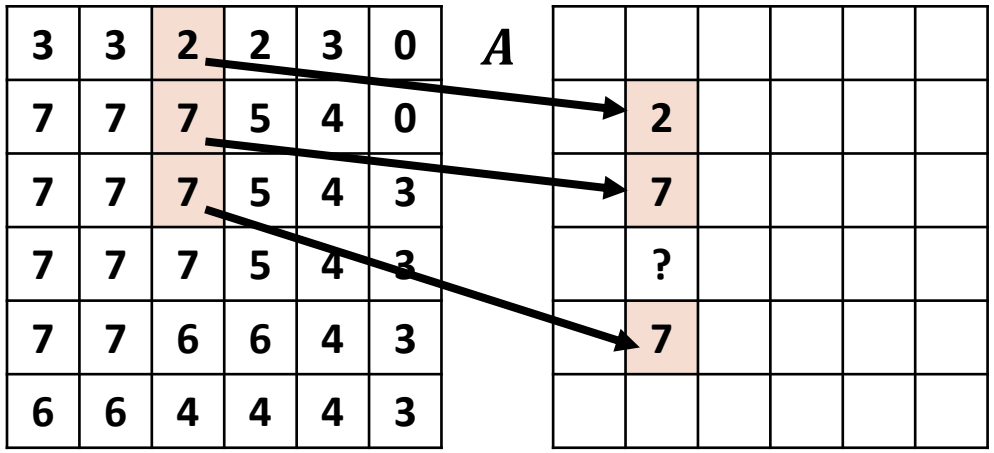
- 원점을 중심으로 반시계 방향으로 θ만큼 회전

$$S(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

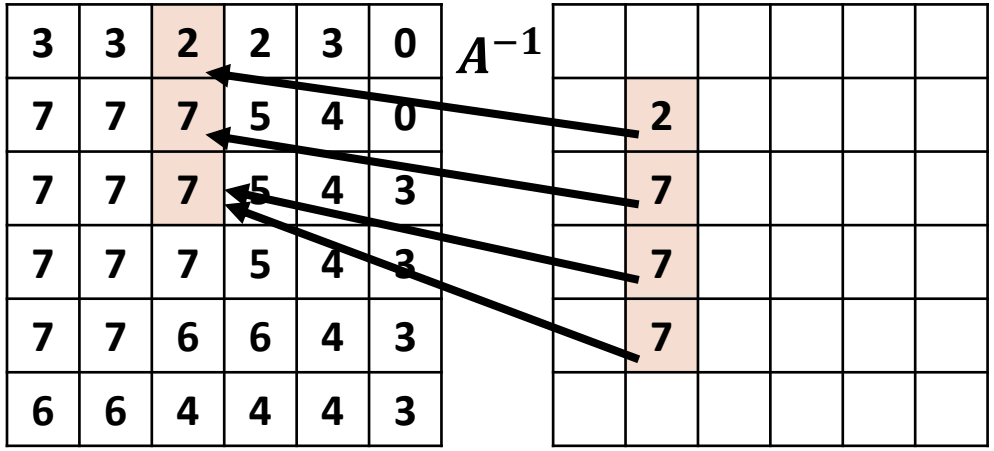
| 크기

- x 방향으로 s_x, y 방향으로 s_y만큼 크기를 조정한다.
- 1보다 크면 확대, 작으면 축소이다.

기하 연산 | 동차 좌표, 동차 행렬



| 전방 변환



| 후방 변환

- 화소에는 동차 변환을 적용해 회전, 크기 조정이 가능하나 화소의 위치를 정수로 지정하기에 문제가 생긴다.

| 에일리어싱

- 기존 영상을 새 영상으로 변환하는 과정에서 실수 좌표를 정수로 반올림하면서 값을 받지 못하는 화소가 생겨 곳곳에 구멍이 뚫린 이상한 영상이 되는 현상

| 안티에일리어싱

- 에일리어싱을 누그러뜨리는 방법
- 변환 영상이 원래 영상의 해당 화소를 찾는 후방 변환을 사용하면 된다.

기하 연산 | 동차 좌표, 동차 행렬

```
Img = cv2.imread('rose.jpg')
Patch = img[200:250, 200:250,:]
```

```
patch1 = cv2.resize(patch, dsize = (0,0), fx
= 5, fy = 5, interpolation =
cv2.INTER_NEAREST)
```

최근접 이웃 INTER_NEAREST

- 실수 좌표를 정수로 변환할 때 반올림을 사용해 가장 가까운 화소에 배정하는 기법
- 그러나 여전히 에일리어싱이 심하다

기하 연산 | 동차 좌표, 동차 행렬

```
patch2 = cv2.resize(patch, dsize = (0, 0), fx = 5, fy = 5, interpolation = cv2.INTER_LINEAR)
patch3 = cv2.resize(patch, dsize = (0, 0), fx = 5, fy = 5, interpolation = cv2.INTER_CUBIC)
```

보간

- 변환된 어떤 화소가 4개 화소에 걸린 경우 4개 화소와 걸친 비율에 따라 가중 평균하여 화솟값을 계산한다.
- 겹치는 비율을 곱하기 때문에 **선형 보간법**에 해당한다.

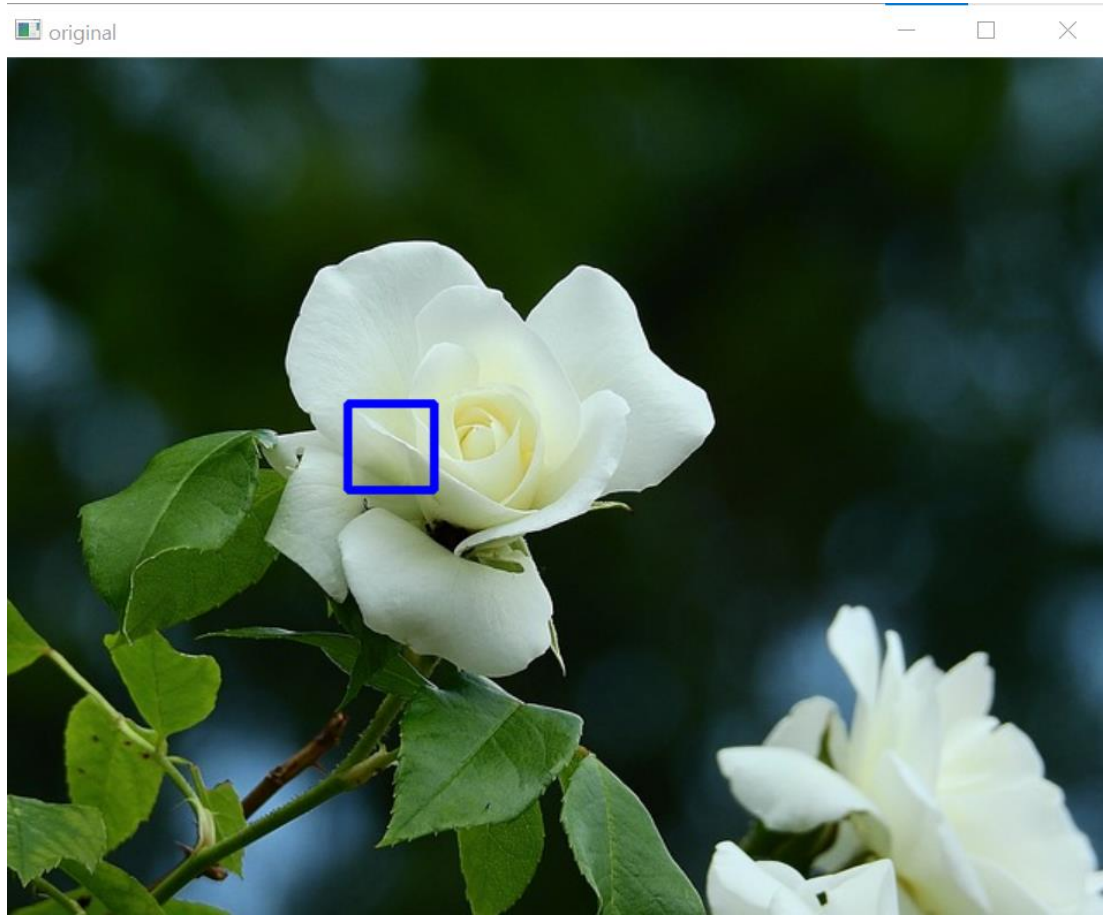
양선형 보간 INTER_LINEAR, 양3차 보간 INTER_CUBIC

- 최근접 이웃에 비해 화질이 월등히 좋으나 계산 시간이 더 걸린다.

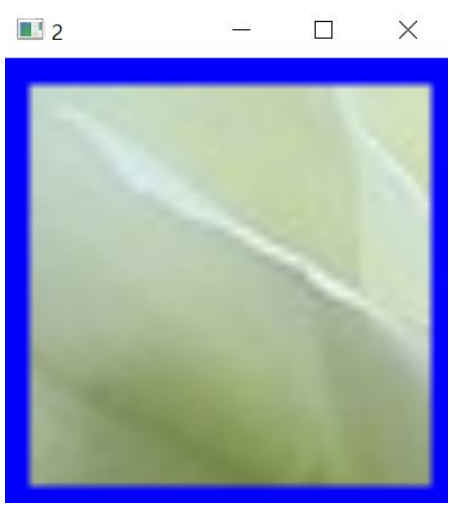
기하 연산 | 실행 결과

```
import cv2

img = cv2.imread('rose.jpg')

patch = img[200:250,200:250,:]

img = cv2.rectangle(img,(200,200),(250,250),(255,0,0),3)

patch1 = cv2.resize(patch,dsize=(0,0),fx=5,fy=5,interpolation=cv2.INTER_NEAREST)
patch2 = cv2.resize(patch,dsize=(0,0),fx=5,fy=5,interpolation=cv2.INTER_LINEAR)
patch3 = cv2.resize(patch,dsize=(0,0),fx=5,fy=5,interpolation=cv2.INTER_CUBIC)

cv2.imshow('original',img)
cv2.imshow('1',patch1)
cv2.imshow('2',patch2)
cv2.imshow('3',patch3)

cv2.waitKey()
cv2.destroyAllWindows()
```

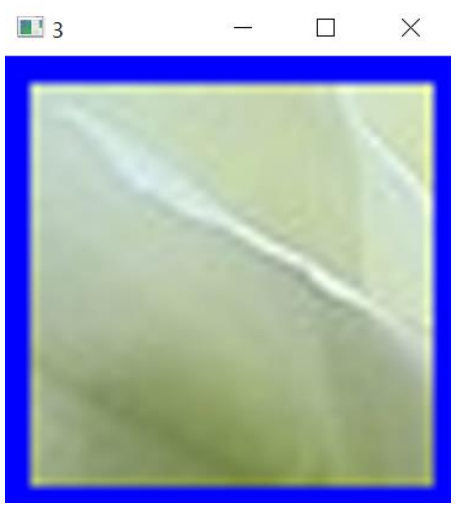
기하 연산 | 실행 결과



|최근접 이웃



|최근접 이웃



|양3차 보간

OpenCV의 시간 효율

```
Import cv2
Import numpy as np
Import time
```

```
def my_cvtgray1(bgr_img):
        g = np.zeros([bgr_img.shape[0], bgr_img.shape[1]])
        for r in range(bgr_img.shape[0]):
                for c in range(bgr_img.shape[1]):
                        g[r, c] = 0.114 * bgr_img[r, c, 0] + 0.587 * bgr_img[r, c, 1] + 0.299 * bgr_img[r, c, 2]
        return np.uint8(g)
```

- For 문 2개를 사용하여 아래 식을 통해 모든 화소의 컬러를 명암으로 변환한다.

$$I = round(0.299 * R + 0.587 * G + 0.114 * B)$$

OpenCV의 시간 효율

```
Import cv2
Import numpy as np
Import time
```

```
def my_cvtgray2(bgr_img):
    g = np.zeros([bgr_img.shape[0],
bgr_img.shape[1]])
    g = 0.114 * bgr_img[:, :, 0] + 0.587 *
bgr_img[:, :, 1] + 0.299 * bgr_img[:, :, 2]
    return np.uint8(g)
```

- 배열 연산을 통해 구현한다.

$$I = round(0.299 * R + 0.587 * G + 0.114 * B)$$

OpenCV의 시간 효율

```
Start= time.time()
Mycvtgray1(img)
Print(time.time()-start
```

- 함수를 호출해 명암 영상 변환을 시작하여 함수가 시작되는 시점과 끝나는 시점을 빼서 소요 시간을 출력하게 된다.

OpenCV의 시간 효율 | 실행 결과

```python
import cv2
import numpy as np
import time

def myg1(bgr):
    g = np.zeros([bgr.shape[0],bgr.shape[1]])
    for r in range(bgr.shape[0]):
        for c in range(bgr.shape[1]):
            g[r,c] = 0.114*bgr[r,c,0] + 0.587*bgr[r,c,1] + 0.299*bgr[r,c,2]
    return np.uint8(g)

def myg2(bgr):
    g = np.zeros([bgr.shape[0],bgr.shape[1]])
    g = 0.114*bgr[:,:,0] + 0.587*bgr[:,:,1] +0.299*bgr[:,:,2]
    return np.uint8(g)

img = cv2.imread('soccer.jpg')

start=time.time()
myg1(img)
print('myg1: ', time.time()-start)
```

```python
start=time.time()
myg2(img)
print('myg2: ', time.time()-start)

start=time.time()
cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
print('opencv: ', time.time()-start)
```

```
myg1:   2.500335693359375
myg2:   0.005983829498291016
opencv:   0.0
```

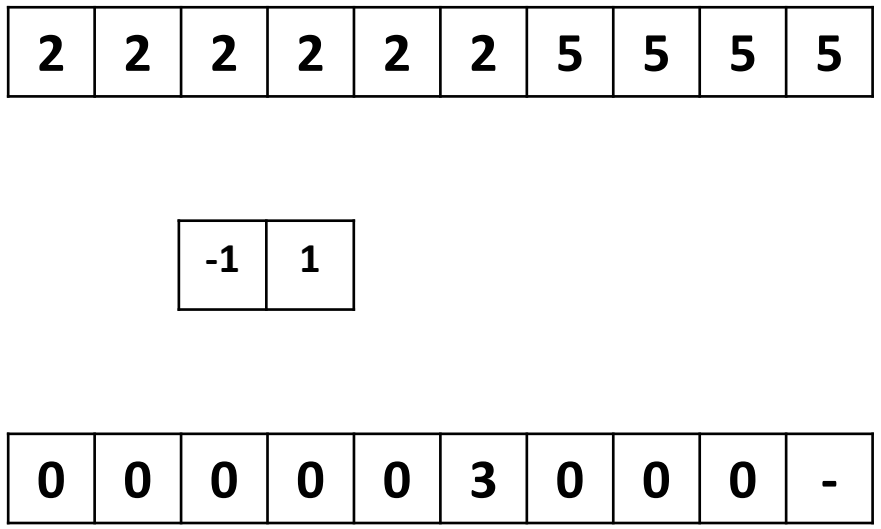
에지 검출

$$f'(x) = \lim_{\delta \to 0} \frac{f(x + \delta x) - f(x)}{\delta x}$$

$$f'(x) = \lim_{\delta \to 0} \frac{f(x + \delta x) - f(x)}{\delta x} = f(x+1) - f(x)$$

- 정수 좌표를 쓰는 디지털 영상에서 x의 최소 변화량은 1이므로 $\delta x = 1$로 하면 디지털 영상을 미분하는 식은 아래 식이 된다.

에지 검출



- 식을 영상 f에 적용하는 건 필터 u의 컨볼루션으로 구현한다.
- 필터 u를 **에지 연산자**라고 한다.
- 명암 변화가 없는 곳은 0, 급격한 부분은 3을 가진다.
- 명암 값이 커지면 미분 값이 **양수**, 작아지면 **음수**다.

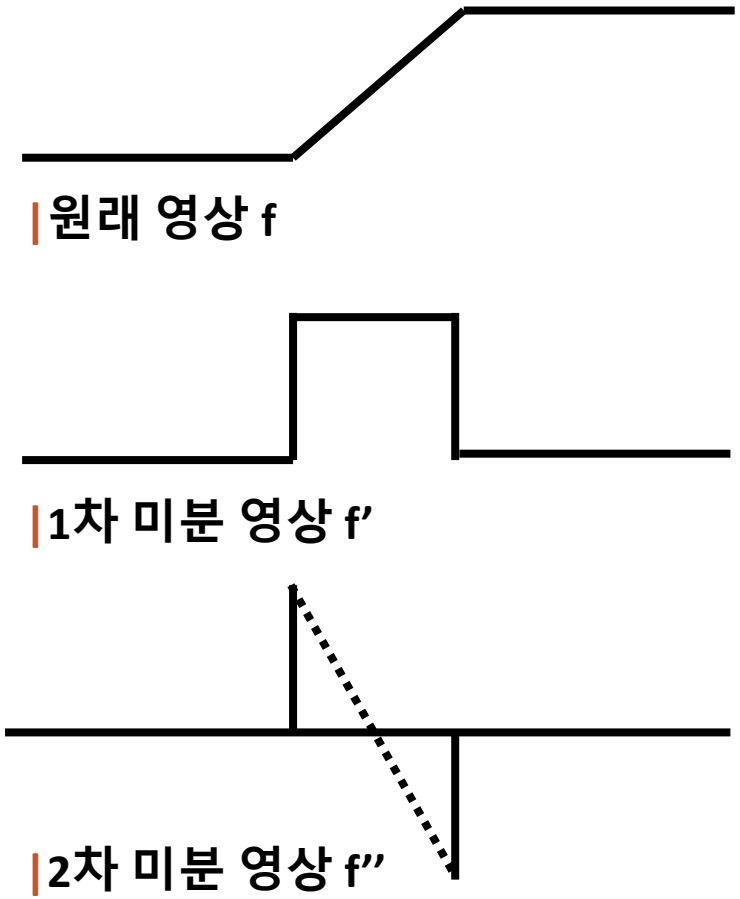
에지 검출

$$f''(x) = \lim_{\delta \to 0} \frac{f(x) - f(x - \delta x)}{\delta} = f'(x) - f(x - 1) = (f(x+1) - \big(f(x)\big) - \big(f(x) - f(x-1)\big) = f(x+1) - 2f(x)f(x-1)$$

1	-2	1

- 기존 영상을 2번 미분한 2차 미분 영상은 위 식에 따른 연산자로 1번에 구할 수 있어 2배 빠르게 구할 수 있다.

에지 검출



- 1차 미분 영상은 에지에서 봉우리가 발생하고 2차 미분 영상은 영교차가 발생한다.
- 영교차: 왼쪽과 오른쪽에 부호가 다른 반응이 나타나고 자신은 0을 갖는 위치를 뜻한다.

- 에지 검출이란 1차 미분에서 봉우리를 찾거나 2차 미분에서 영교차를 찾는 일이다.

에지 검출

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

$u_x =$

-1	0	1

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

$u_y =$

-1
0
1

- 위 식을 통해 1차원에서 설계한 연산자를 2차원으로 확장한다.
- 2차원에서는 x, y 방향의 두 연산자를 사용한다.

에지 검출

$u_x =$

-1	0	1
-1	0	1
-1	0	1

$u_y =$

-1	-1	-1
0	0	0
1	1	1

프레윗 연산자

$u_x =$

-1	0	1
-2	0	2
-1	0	1

$u_y =$

-1	-2	-1
0	0	0
1	2	1

소벨 연산자

- 필터를 3 * 3 크기로 확장하면 잡음을 흡수하여 더 좋은 성능을 보인다.
- 두 필터는 가장 널리 쓰이는 에지 연산자다.

에지 검출

$$s(y,x) = \sqrt{f_x'(y,x)^2 + f_y'(y,x)^2}$$

$$d\,(y,x) = \arctan\left(\frac{f_y'(y,x)}{f_x'(y,x)}\right)$$

- 위 식으로 에지일 가능성을 나타내는 에지 강도와 에지의 진행 방향을 나타내는 에지 방향을 구할 수 있다.
- f_x'와 f_y'는 프레윗 또는 소벨 연산자를 적용한 결과 영상이다.

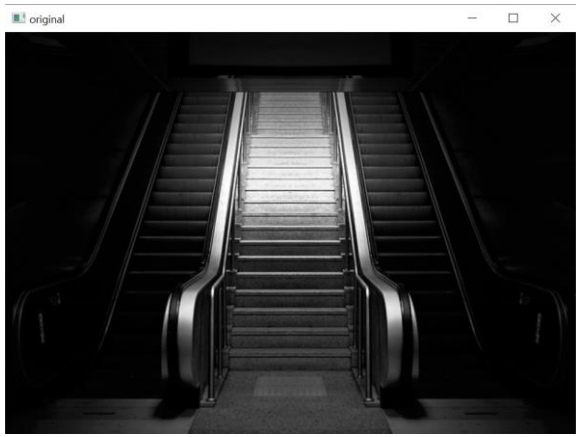
에지 검출

```
img = cv2.imread('dark.jpg’)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

gradx = cv2.Sobel(gray, cv2.CV_32F, 1, 0, ksize = 3)
grady = cv2.Sobel(gray, cv2.CV_32F, 0, 1, ksize = 3)

sobelx = cv2.convertScaleAbs(gradx)
sobely = cv2.convertScaleAbs(grady)

edge_strength = cv2.addWeighted(sobelx, 0.5,
sobely, 0.5, 0)
```

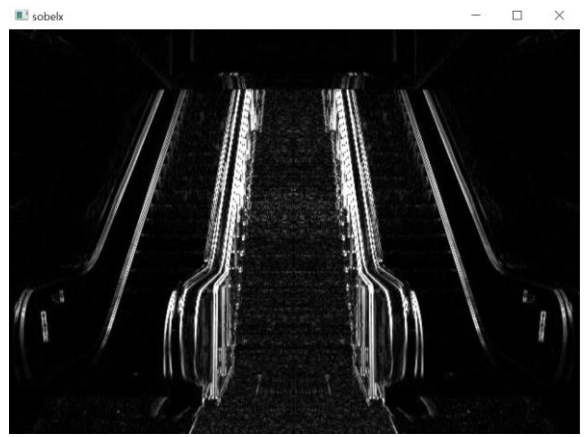
- 명암으로 만든 영상을 Sobel 함수로 x 방향의 연산자를 사용한다.
- Sobel 함수의 두 번째 인수는 영상을 32비트 실수 맵에 저장하도록 지시하며, 세, 네 번째 인수는 x 방향 연산자를 사용하도록 지시한다. 1과 0의 위치를 바꿀 경우 y 방향 연산자를 사용한다. 마지막 인수는 3 * 3 크기를 사용하도록 지정한다.

에지 검출 | 실행 결과

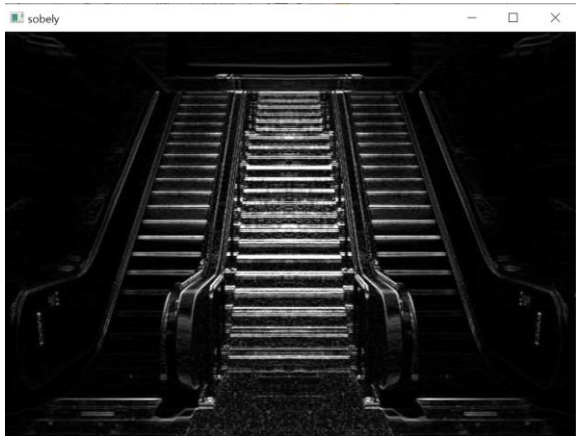
```python
import cv2

img = cv2.imread('dark.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

gradx = cv2.Sobel(gray,cv2.CV_32F,1,0,ksize=3)
grady = cv2.Sobel(gray,cv2.CV_32F,0,1,ksize=3)

sobelx = cv2.convertScaleAbs(gradx)
sobely = cv2.convertScaleAbs(grady)

edge_strength = cv2.addWeighted(sobelx,0.5,sobely,0.5,0)

cv2.imshow('original',gray)
cv2.imshow('sobelx',sobelx)
cv2.imshow('sobely',sobely)
cv2.imshow('edge_strength',edge_strength)

cv2.waitKey()
cv2.destroyAllWindows()
```



| original



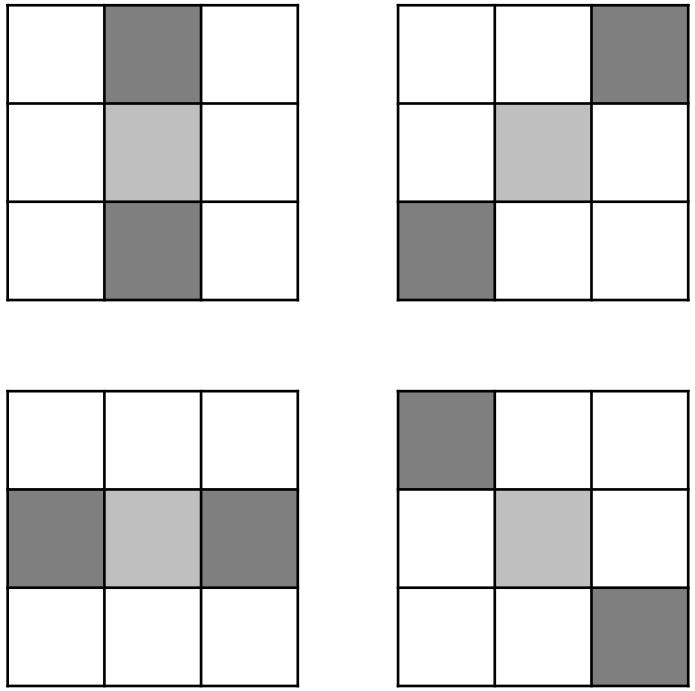
| sobelx



| sobely

| edge_strength

캐니 에지



비최대 억제

- 에지 방향에 수직인 두 이웃 화소의 에지 강도가 자신보다 작으면 **에지로 살아남고** 그렇지 않으면 **에지 아닌 화소로** 바뀐다.
- 두 이웃 화소와 비교해 자신이 최대가 아니면 억제된다.

캐니 알고리즘

- 거짓 긍정을 줄이기 위해 2개의 이력 임곗값을 이용한 에지 추적을 추가로 적용한다.
- **거짓 긍정**이란 실제 에지가 아닌데 에지로 검출된 화소를 뜻한다.
- 추적은 실제 에지일 가능성이 높은 곳에서 시작하며 이후 추적은 임곗값을 넘는 에지를 대상으로 진행한다.

캐니 에지

```
Img = cv2.imread('soccer.jpg')
Gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

Canny1 = cv2.Canny(gray,50,150)
Canny2 = cv2.Canny(gray,100,200)
```

- 이력 임곗값이 높으면 더욱 확실한, 에지 강도가 큰 화소만 추적하기 때문에 더 적은 에지가 발생한다.
- 이력 임곗값을 낮게 설정한 경우엔 잡음 에지가 많이 발생한다.

캐니 에지 | 실행 결과

```python
import cv2

img = cv2.imread('soccer2.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

canny1 = cv2.Canny(gray,50,150)
canny2 = cv2.Canny(gray,100,200)

cv2.imshow('original',gray)
cv2.imshow('Canny1',canny1)
cv2.imshow('Canny2',canny2)

cv2.waitKey()
cv2.destroyAllWindows()
```

| original

| Canny1

| Canny2

직선 검출

	0	1	2	3	4	5	6
0							
1			O				
2		O					
3		O					
4		O			O		
5			O	O			
6							

경계선 (1,2)(2,1)(3,1)(4,1)(5,2)(5,3)(4,4)

- 에지 맵에서 에지 화소는 1, 에지가 아닌 화소는 0으로 표시한다.

직선 검출

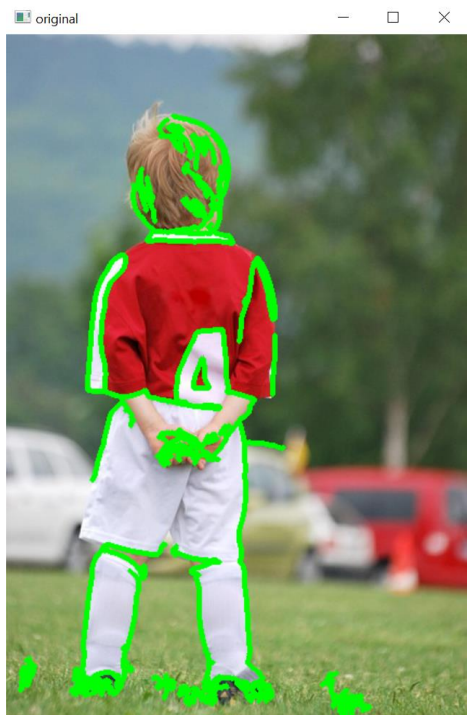
```
Img = cv2.imread('soccer.jpg')
Gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
Canny = cv2.Canny(gray,100,200)
```

```
Contour, hierarchy = cv2.findContours(canny,
cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)

Lcontour = []
For I in range(len(contour)):
        if contour[i].shape[0] > 100:
                lcontour.append(contour[i])

Cv2.drawContours(img, lcontour, -1, (0,255,0), 3)
```

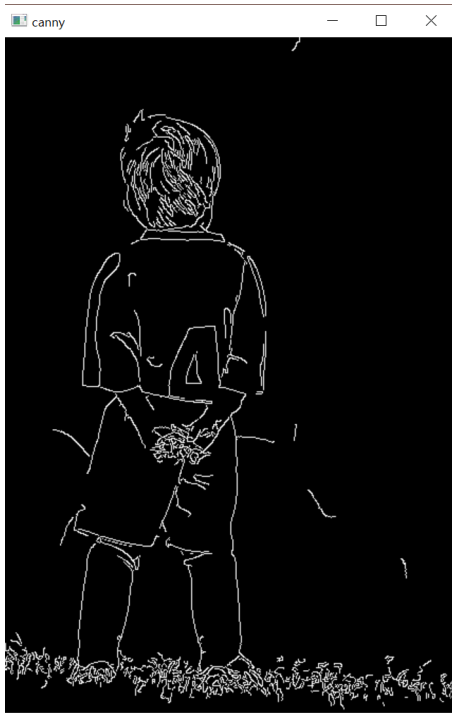
- Canny 함수로 우선 에지 맵을 구한다.
- findContours 함수를 통해 경계선을 찾는데 검출된 경계선을 contour에 저장하고, hierachy에는 경계선의 계층 구조를 저장한다.

직선 검출 | 실행 결과

```python
import cv2
import numpy as np

img = cv2.imread('soccer2.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
canny = cv2.Canny(gray,100,200)

contour, hierarchy = cv2.findContours(canny,cv2.RETR_LIST,cv2.CHAIN_APPROX_NONE)

lcontour = []

for i in range(len(contour)):
    if contour[i].shape[0] > 100:
        lcontour.append(contour[i])

cv2.drawContours(img,lcontour,-1,(0,255,0),3)

cv2.imshow('original',img)
cv2.imshow('canny',canny)

cv2.waitKey()
cv2.destroyAllWindows()
```

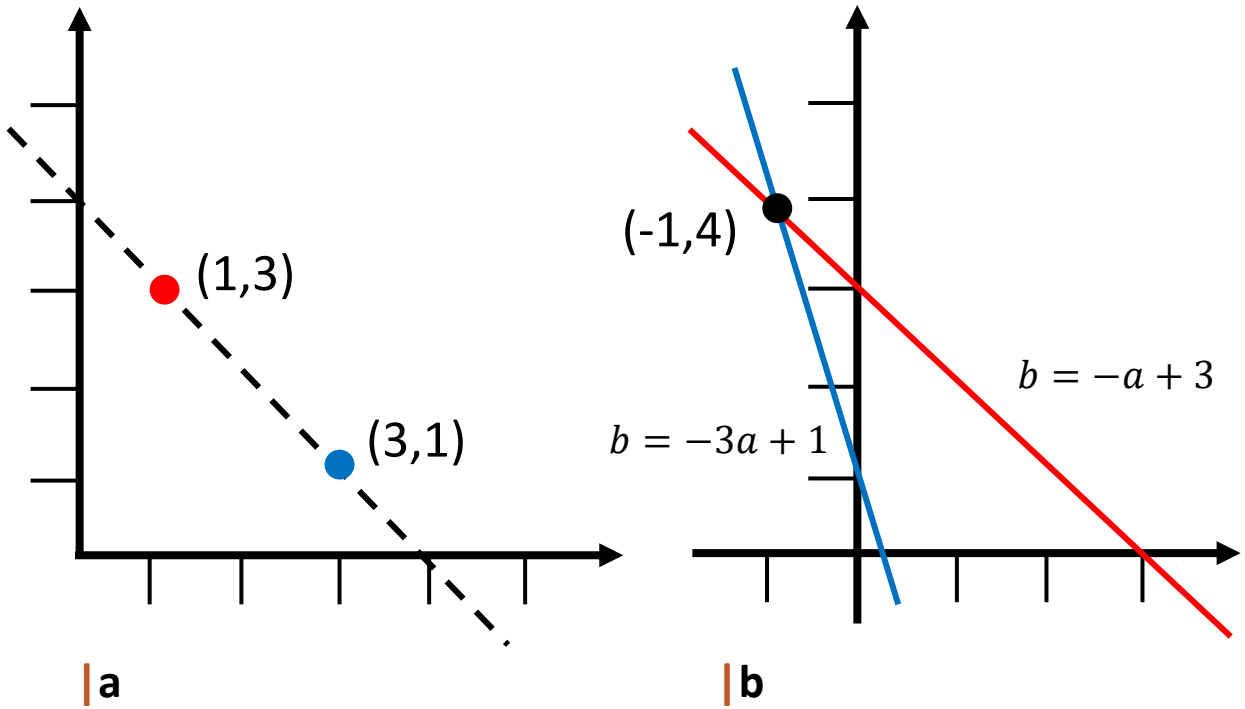


| original



| canny

직선 검출



- |a에서의 좌표 (1,3)과 (3,1)을 $y = ax + b$에 대입하여 그 식에서 b와 a을 변수로 간주하면 |b와 같이 새로운 공간 (a,b)가 형성된다.
- |b에서 두 직선이 만나는 점 (-1,4)는 원래 공간에서 직선의 기울기와 y절편이다.

- 그러므로 |a에서의 직선 방정식은 $y = -1x + 4$다.

- (a,b) 공간에서 두 직선이 만나는 점은 **투표**로 알아낸다. 직선은 자신이 지나는 점에 1만큼씩 투표를 한다.
- |b의 경우 직선이 지나지 않는 곳은 0, 지나는 곳은 1, 두 직선이 지나는 곳은 2표를 받는다.

직선 검출 | 실제 상황에서 구현하기 위해 고려할 점

$$xsin(\theta) + ycos(\theta) = \rho$$

첫째

- 현실에는 많은 점이 있고 점들이 완벽히 일직선을 이루지 못한다.
- 그러므로 (a,b) 공간을 이산화하여 해결한다.

둘째

- 투표가 이뤄진 누적 배열에 잡음이 많다.

셋째

- 직선 방정식 $y = ax + b$로는 기울기 a가 무한대인 경우엔 투표가 불가능하다.
- 극좌표에서 직선의 방정식을 표현하는 다음 식을 도입해서 해결한다.

직선 검출 | 동차 좌표, 동차 행렬

- 허프 변환은 직선의 방정식을 알려주지만 직선의 양 끝점은 알려주지 못한다.
- 이를 위해 비최대 억제 과정에서 극점을 형성한 화소를 찾아 가장 먼 곳에 있는 두 화소를 계산하는 추가적인 과정이 필요하다.
- 허프변환은 이론적으로 어떤 도형이라도 검출할 수 있다.

직선 검출

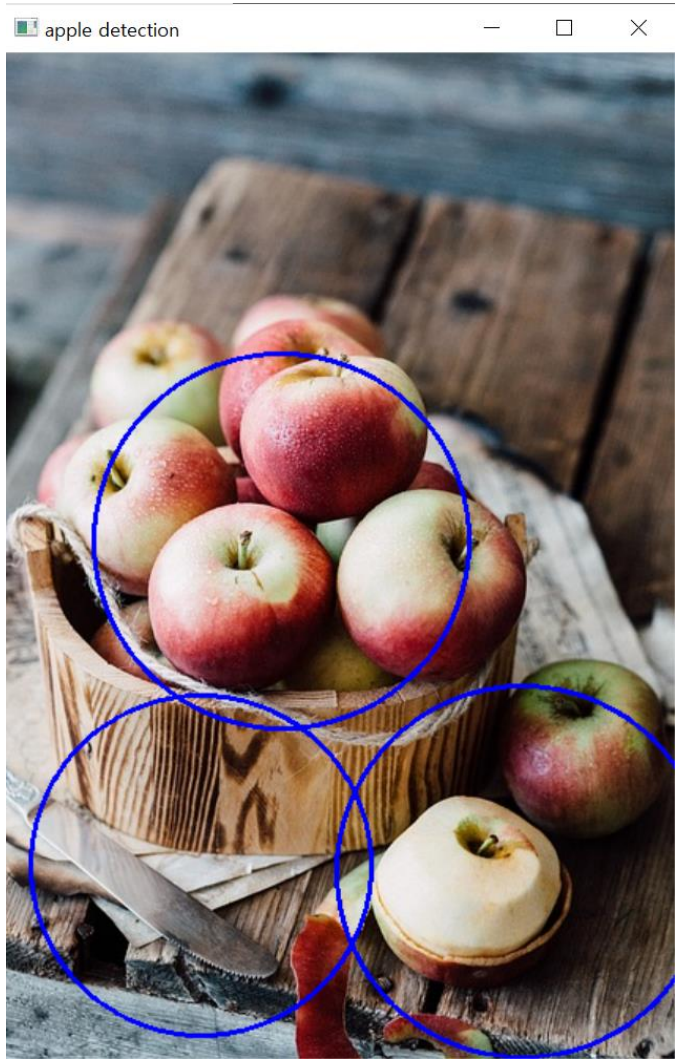
```
img = cv2.imread('apple.jpg’)
gray =
cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

apples =
cv2.HoughCircles(gray,cv2.HOUGH_GRADIENT,
1, 200, param1 = 150, param2 = 20,
minRadius = 50, maxRadius = 120)

for i in apples[0]:
    cv2.circle(img, (int(i[0]), int(i[1])), int(i[2]),
(255, 0, 0), 2)
```

HoughCircles 함수

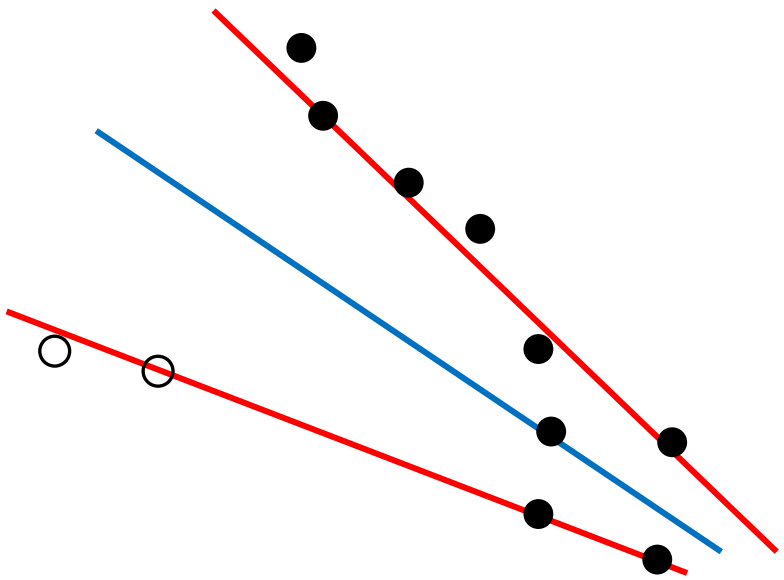
- 영상에서 원을 검출해 중심과 반지름을 저장한 리스트를 반환한다.
- Cv2.HOUGH_GRADIENT는 에지 방향 정보를 추가로 사용하는 방법이다.
- **세 번째 인수**에서는 누적 배열의 크기를 지정하는데 1로 설정할 시 입력 영상과 같은 크기를 사용한다.
- **네 번째 인수**는 원 사이의 최소 거리를 지정하는데 작을수록 많은 원이 검출된다.
- **다섯 번째 인수**는 캐니 알고리즘이 사용하는 T_{high}, **여섯 번째 인수**는 비최대 억제를 적용할 때 쓰는 임곗값이다. 이후 2개는 원의 최소와 최대 반지름을 지정한다.

직선 검출 | 실행 결과

```
import cv2

img = cv2.imread('apple.jpg')
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

apples = cv2.HoughCircles(gray,cv2.HOUGH_GRADIENT,1,200, param1 = 150, param2 = 20, minRadius = 50, maxRadius = 120)
for i in apples[0]:
    cv2.circle(img,(int(i[0]),int(i[1])),int(i[2]),(255,0,0),2)

cv2.imshow('apple detection',img)

cv2.waitKey()
cv2.destroyAllWindows()
```



직선 검출



허프 변환이 찾은 선분
최소평균제곱오차가 찾은 선분
아웃라이어

- 허프 변환은 어떤 선분이 있는지 몰라 **모든 점에 같은 투표 기회**를 주기에 누적 배열에 **잡음이 많이** 발생한다.

최소평균제곱오차 알고리즘

- 모든 점을 대상으로 오류를 계산하고 최소 오류를 범하는 직선을 찾는다.
- 인라이어, 아웃라이어의 구별없이 모든 샘플이 동등한 자격으로 오류 계산에 참여하여 아웃라이어의 영향이 크다.

- 둘 다 아웃라이어에 민감한 강인하지 않은 기법이다

직선 검출

강인한 추정

- 아웃라이어를 걸러내는 과정을 가진 추정 기법
- ex) 같은 물건의 길이를 5번 측정한 결과 {16,1,1,1,1}을 얻었다면 16은 아웃라이어일 가능성이 높다.

- **평균**으로 길이를 추정할 경우 **4**가 되므로 평균은 아웃라이어에 대처할 능력이 전혀 없다고 볼 수 있다.

- **중앙값**은 아웃라이어를 배제하여 길이를 **1**이라고 추정하기 때문에 **강인한 추정 기법**이다.

RANSAC

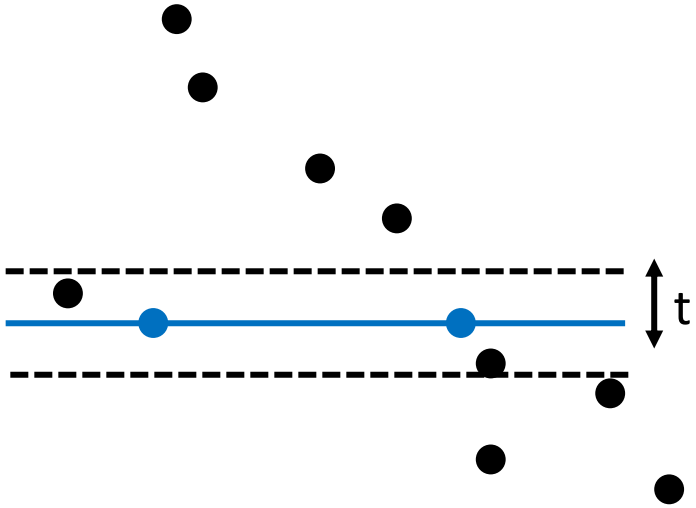
- 인라이어와 아웃라이어가 섞여 있는 상황에서 인라이어를 찾아 최적 근사하는 기법

- 랜덤하게 두 점을 선택하여 두 점을 지나는 직선을 계산한다.

- **난수**를 사용하여 **실행할 때마다 다른 결과**를 출력한다.

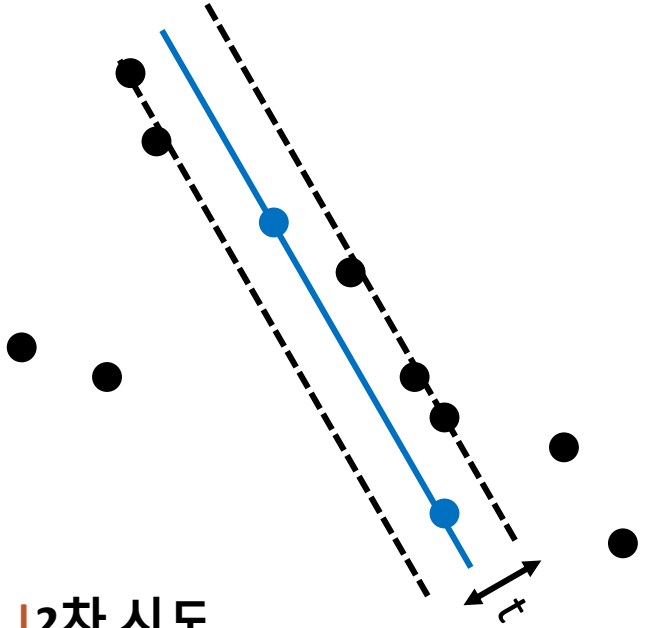
직선 검출

임곗값 d = 5
허용하는 오차 = 5



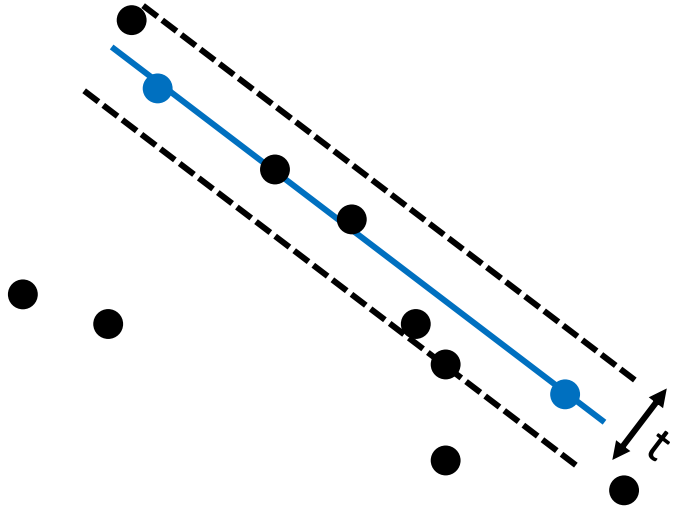
1차 시도

- 개수가 임곗값을 넘지 못했으므로 버린다.



2차 시도

- 임곗값을 넘었으므로 일치하는 점 7개를 가지고 최적 직선을 추정한다.



3차 시도

- 이런 시도를 많이 반복하여 후보군 중 최적을 찾아 출력한다.
- RANSAC은 반복 횟수가 많을수록 진짜 직선을 찾을 가능성이 높아진다.

영역 분할

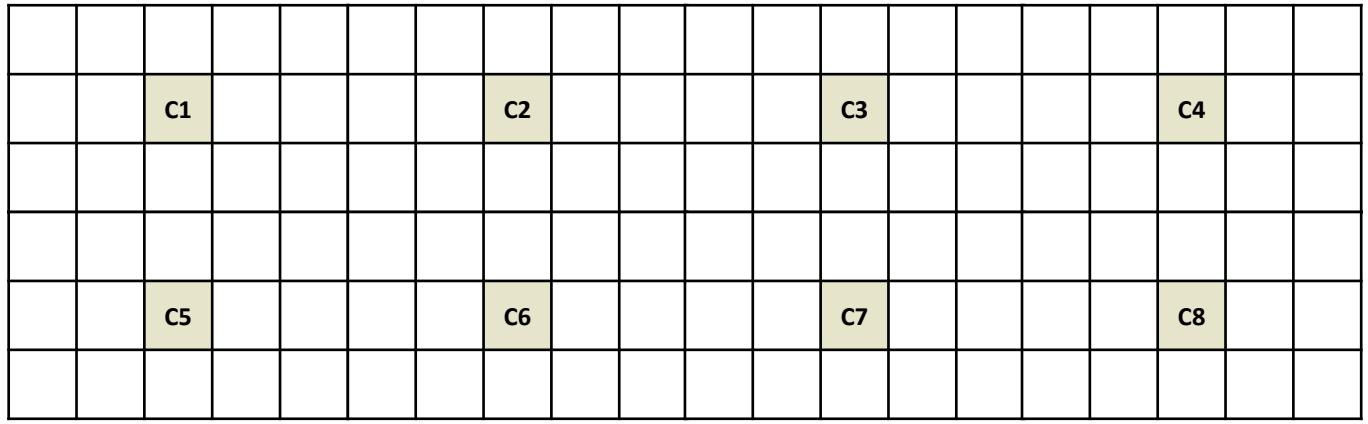
영역 분할
- 물체가 점유한 영역을 구분하는 작업
- 에지는 물체의 경계를 지정하므로 에지가 완벽하다면 영역 분할이 따로 필요 없다.

의미 분할
- 의미 있는 단위로 분할하는 방식
- ex) 관심이 있는 물체에 집중하는 선택적 주의집중 자굥ㅇ을 통해 사람 영역, 눈과, 입을 분할하여 표정까지 인식한다.

워터 셰드
- 비가 왔을 때 **오목한 곳에 웅덩이**가 생기는 현상을 모방하는 연산
- 이를 확장해 영역 분할에 활용 가능하다.

슈퍼 화소
- 영상을 아주 작은 영역으로 분할해 다른 알고리즘의 입력으로 사용할 때 화소보다 크지만 물체보다 작은 영역

영역 분할



SLIC

- 여러 슈퍼화소 알고리즘 중 하나
- K평균 군집화와 비슷하게 작동하나 처리 과정이 단순하고 성능이 좋다.
- 입력 영상에서 K개 화소를 군집 중심으로 간주한다.
- 화소를 색상과 위치 값을 결합해 **5차원 벡터**로 표현한다.
- **화소를 가장 가까운 군집 중심에 할당하는 단계, 군집 중심을 갱신하는 단계**를 반복한다.

영역 분할

```
pip install scikit-image
```

```
Img = skimage.data.coffee()
cv2.imshow('coffee', cv2.cvtColor(img,
cv2.COLOR_RGB2BGR))

Slic1 = skimage.segmentation.slic(img,
compactness = 20, n_segments = 600)
sp_img1 =
skimage.segmentation.mark_boundaries(img,
slic1)
Sp_img1 = np.uint8(sp_img1 * 255.0)
```

- skimage 라이브러리를 설치한다
- OpenCV로 skimage 영상을 불러올 때, skimage는 RGB 순서로 저장하지만, OpenCV는 반대로 저장하기에 cvtColor 함수로 BGR 변환하여 출력해야 한다.
- Slic 함수의 첫 번째 인수는 분할할 영상이고, compactness 인수는 슈퍼 화소의 모양을 조절한다. 값이 클수록 **네모에 가깝게** 만들어지고 대신 슈퍼 화소의 색상 균일성이 희생된다.
- **n_segments** 인수는 슈퍼 화소의 개수를 **600개**로 지정한다.(기존에 설명했던 k)
- 0~1 사이의 실수로 표현된 sp_img1을 0~255 사이로 바꾸고 uint8 형으로 변환한다.

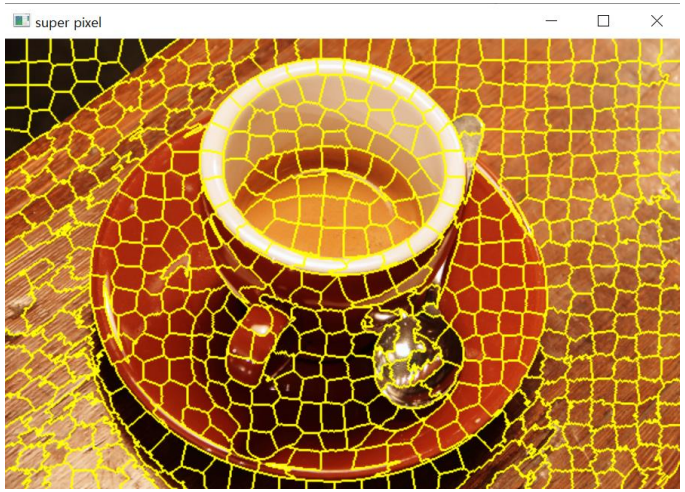
영역 분할 | 실행 결과

```
pip install scikit-image
```

```
Requirement already satisfied: scikit-imag
Requirement already satisfied: numpy>=1.23
Requirement already satisfied: scipy>=1.9
Requirement already satisfied: networkx>=2
Requirement already satisfied: pillow>=9.1
Requirement already satisfied: imageio>=2.
```

```python
import cv2
import skimage
import numpy as np
img = skimage.data.coffee()

cv2.imshow('coffee', cv2.cvtColor(img, cv2.COLOR_RGB2BGR))

slic1 = skimage.segmentation.slic(img, compactness = 20, n_segments = 600)
sp_img = skimage.segmentation.mark_boundaries(img, slic1)
sp_img = np.uint8(sp_img * 255.0)

cv2.imshow('super pixel', cv2.cvtColor(sp_img,cv2.COLOR_RGB2BGR))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

coffee



super pixel

영역 분할

최적화 분할

- 다른 분할 알고리즘은 지역적 명암 변화만 살피기에 배경색과 물체의 색이 비슷하면 **경계가 형성되지 않아** 배경과 물체의 **영역이 섞이게** 된다.
- 지역적 명암 변화와 전역적 정보를 같이 고려하여 이 문제를 해결할 수 있다.

유사도

- 거리와 반대되는 개념으로, 거리 d_{pq} 가 가질 수 있는 최댓값 D에서 거리 d_{pq} 를 뺀 값 또는 거리 d_{pq} 의 역수를 사용한다.
- f(v)는 v에 해당하는 화소의 색상, 위치를결합한 벡터다.

$$\text{거리} \begin{cases} d_{pq} = \|f(v_p) - f(v_q)\|, \text{만일 } v_q \in neighbor(v_p) \\ \infty, \text{그렇지 않으면} \end{cases}$$

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

영역 분할

정규화 절단

- 화소를 노드로 취하고 f(v)로 색상과 위치를 결합한 5차원 벡터를 사용하고, 유사도를 에지 가중치로 사용한다.
- 기존 영역을 2개 영역 , C_1, C_2로 분할했을 때 cut을 위 식으로 정의한다.
- cut 식은 C_1, C_2 영역이 클수록 둘 사이에 에지가 많아 덩달아 커지며 이를 목적 함수로 사용한 알고리즘은 영역을 **자잘하게 분할**하는 경향을 띤다.
- 아래 식은 cut을 정규화하여 영역의 크기에 중립이 되도록 만든다.

$$cut(C_1, C_2) = \sum_{v_p \,\epsilon C_1, v_q \,\epsilon C_2} S_{pq}$$

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

영역 분할

```
coffee = skimage.data.coffee()

slic = skimage.segmentation.slic(coffee,
compactness = 20, n_segments = 600)
g =
skimage.future.graph.rag_mean_color(coffee,slic,
mode ='similarity')

ncut =
skimage.future.graph.cut_normalized(slic,g)

marking =
skimage.segmentation.mark_boundaries(coffee,
ncut)
ncut_coffee = np.uint8(marking * 255.0)
```

- Rag_mean_color 함수는 슈퍼 화소를 노드로 사용하고 'similarity'를 에지 가중치로 사용한 그래프를 구성하여 g 객체에 저장한다.
- Cut_normalized 함수는 slic1과 g 객체 정보를 이용하여 정규화 절단을 수행하고 결과를 ncut 객체에 저장한다.

대화식 분할

2	3	3	3	2	2	3	0
3	7	7 (0, 6)	7	7 (5)	5	4	0
3	7	7	7	7	5	4	3
3	6	7	7	7	5 (4)	4	3
3	6 (1)	7	7	6	6	4	3
3	6	6	6	4	4	4	3
3	3	4	4 (2)	4	2 (3)	2	2
3	3	2	2	2	2	2	2

디지털 공간에서의 스네이크 곡선 표현

능동 외곽선

- 사용자가 물체 내부에 초기 곡선을 지정하면 곡선을 점점 확장하면서 물체 외곽선으로 접근하는 방법
- 곡선이 꿈틀대면서 에너지가 최소인 상태를 찾아가기 때문에 **스네이크**라는 별명을 얻었다.

스프라인 곡선

- 매끄러운 곡선의 모양을 모델링하는 데 쓸 수 있다.
- $g(l) = (y(l), x(l))$로 표현하며 매개변수 l은 $[0,1]$ 범위의 실수로, 폐곡선을 이루기 위해 $g(0) = g(1)$이다.
- 그림은 $n = 6$일 때의 사례다.

대화식 분할

```
mask = np.zeros((img.shape[0], img.shape[1]),
np.uint8)mask[:,:] = cv2.GC_PR_BGD
```

```
background = np.zeros((1,65), np.float64)
foreground = np.zeros((1,65), np.float64)
```

GrabCut

- 사용자가 붓으로 물체와 배경을 초기 지정하고 물체 히스토그램과 배경 히스토그램을 만드는 알고리즘이다.
- 원본 영상과 같은 크기의 mask 배열을 생성하고 모든 화소를 **배경일 것 같음(cv2.GC_PR_BGD)**으로 초기화한다.
- Grabcut 함수가 내부에서 사용할 배경 히스토그램과 물체 히스토그램을 새성하고 0으로 초기화 한다.

대화식 분할

```
cv2.grabCut(img, mask, None, background,
foreground, 5, cv2.GC_INIT_WITH_MASK)
```

GrabCut

- Grabcut 함수의 **두 번째 인수**는 사용자가 지정한 물체와 배경 정보를 가진 맵이다.
- 배경으로 지정한 화소는 **0(cv2.GC_BGD)**, 물체는 **1(cv2.GC_BGD)**, 나머지 화소는 배경일 것 같음에 해당하는 **2(cv2.GC_PR_BGD)**를 가지고 있다.
- 세 번째 인수는 관심 영역을 지정하는 ROI로 None으로 설정하여 전체 영상을 대상으로 하도록 한다.
- **네, 다섯 번째** 인수는 배경, 물체 히스토그램이다.
- **여섯 번째** 인수는 5번 반복하도록 지시하며, **마지막** 인수는 배경과 물체를 표시한 맵을 사용하라고 지시한다.

대화식 분할

```
Mask2 = np.where((mask == cv2.GC_BGD)|(mask
== cv2.GC_PR_BGD), 0,1).astype('uint8’)
grab = img * mask2[:,:,np.newaxis]
```

GrabCut

- 배경, 배경일 것 같음인 화소를 0, 물체, 물체일 것 같음인 화소를 1로 바꾸어 mask2 객체에 저장한다.
- Grab에 원본 img와 mask2를 곱하여 배경에 해당하는 화소를 검게 바꾸어 저장한다.

대화식 분할 | 실행 결과

```
import cv2
import numpy as np


img = cv2.imread('soccer2.jpg')
img_show = np.copy(img)

mask = np.zeros((img.shape[0], img.shape[1]), np.uint8)
mask[:,:] = cv2.GC_PR_BGD

brushsiz = 9
lcolor, rcolor = (255,0,0), (0,0,255)

def painting(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(img_show, (x,y), brushsiz, lcolor, -1)
        cv2.circle(mask, (x,y), brushsiz, cv2.GC_FGD, -1)

    elif event == cv2.EVENT_RBUTTONDOWN:
        cv2.circle(img_show, (x,y), brushsiz, rcolor, -1)
        cv2.circle(mask, (x,y), brushsiz, cv2.GC_BGD, -1)
```

```
    elif event == cv2.EVENT_MOUSEMOVE and flags == cv2.EVENT_FLAG_LBUTTON:
        cv2.circle(img_show, (x,y), brushsiz, lcolor, -1)
        cv2.circle(mask, (x,y), brushsiz, cv2.GC_FGD, -1)

    elif event == cv2.EVENT_MOUSEMOVE and flags == cv2.EVENT_FLAG_RBUTTON:
        cv2.circle(img_show, (x,y), brushsiz, rcolor, -1)
        cv2.circle(mask, (x,y), brushsiz, cv2.GC_BGD, -1)

    cv2.imshow('painting', img_show)

cv2.namedWindow('painting')
cv2.setMouseCallback('painting',painting)


while True:
    if cv2.waitKey(1) == ord('q'):
        break

background = np.zeros((1,65), np.float64)
foreground = np.zeros((1,65), np.float64)
```

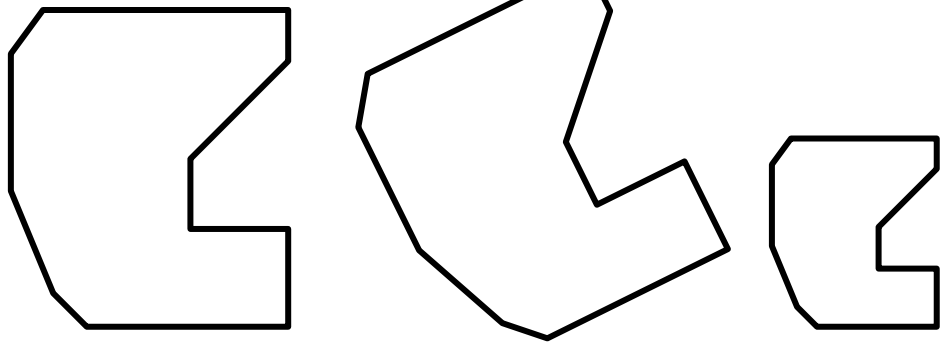
```
cv2.grabCut(img, mask, None, background, foreground, 5, cv2.GC_INIT_WITH_MASK)
mask2 = np.where((mask == cv2.GC_BGD)|(mask == cv2.GC_PR_BGD), 0,1).astype('uint8')
grab = img * mask2[:,:,np.newaxis]
cv2.imshow('grab cut image', grab)

cv2.waitKey()
cv2.destroyAllWindows()
```

대화식 분할 | 실행 결과

| painting

| grab cut image

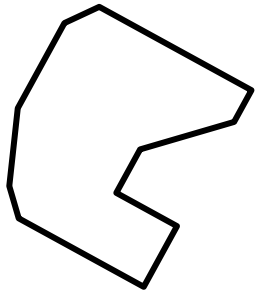
영역 특징



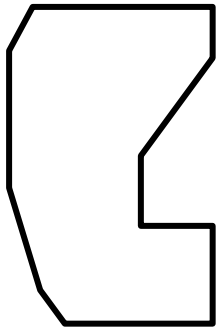
기준

회전

축소



회전 + 축소



찌그림

불변성

- 물체는 다양한 변환을 거쳐 영상에 나타날 수 있는데 변환으로도 특징의 값이 변하지 않는 걸 의미한다.
- 면적이나 주축은 각각 회전과 축소에 대해 불변이다.

등변성

- 불변성과 반대되는 개념으로 특징이 어떤 변환에 따라 변하는 걸 의미한다.

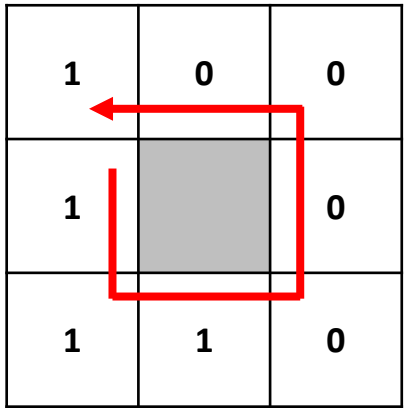
영역 특징

137	115	92
141	122	99
145	129	105

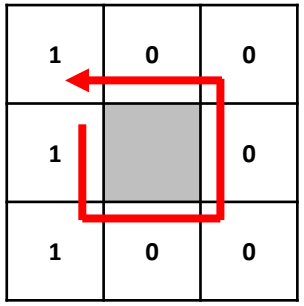
1	0	-1
1		-1
1	0	-1

LTP 계산

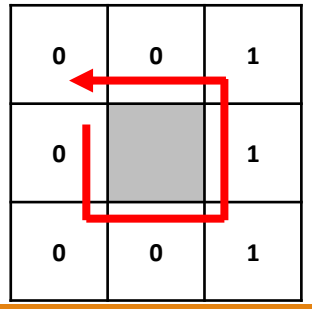
LBP 계산



11100001 = 225



11000001 = 193



00011100 = 28

LBP

- 회색 칠한 화소를 조사할 때 주위 8개 화소와 값을 비교하여 회색 화소보다 크면 1, 작으면 0으로 표시한다.
- 빨간 선분을 따라 순서대로 이진수를 구성하여 십진수로 변환한다(0~255까지 발생 가능).
- 작은 명암 변화에 민감하다.

LTP

- 임곗값이 t = 10이라면 회색 화솟값 122를 기준으로 122-10보다 작은 화소에 -1, 122+10보다 큰 화소에 1, 외에는 0을 부여해 3진 코드를 얻는다.
- 3진 코드를 이진 코드 2개로 변환하여 각각을 십진수로 변환한다.
- LBP의 문제를 누그러뜨릴 수 있다.

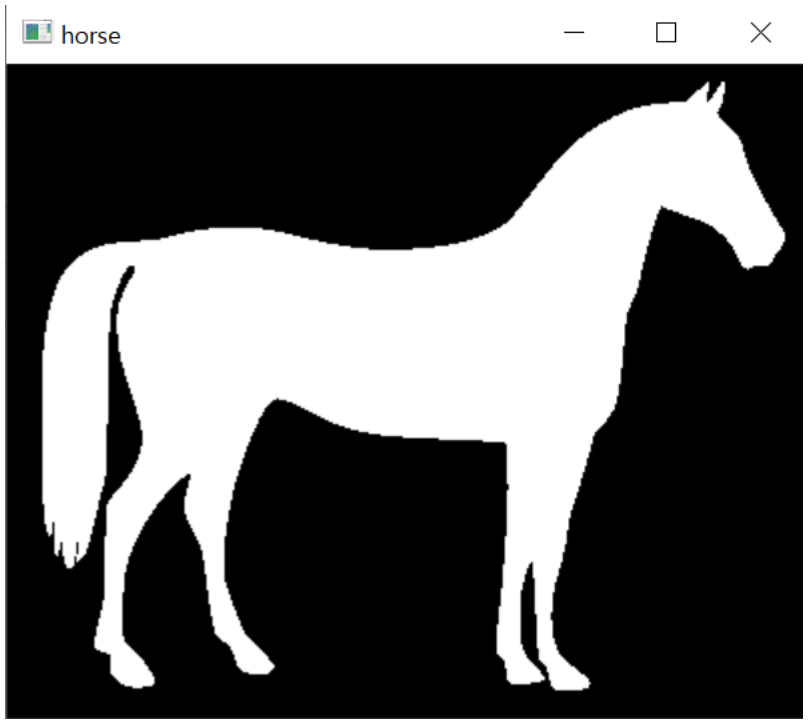
영역 특징 | 실행 결과

```python
import skimage
import numpy as np
import cv2

orig = skimage.data.horse()
img = 255 - np.uint8(orig) * 255
cv2.imshow('horse', img)

contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

img2 = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
cv2. drawContours(img2, contours, -1, (255,0,255), 2)
cv2.imshow('horse with contour', img2)

contour = contours[0]

m = cv2.moments(contour)
area = cv2.contourArea(contour)
cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
perimeter = cv2.arcLength(contour, True)
roundness = (4.0 * np.pi * area)/(perimeter * perimeter)
print(f'면적 = {area}\n중점 = ({cx}, {cy})\n둘레 = {perimeter}\n둥근 정도 = {roundness}')

img3 = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
```

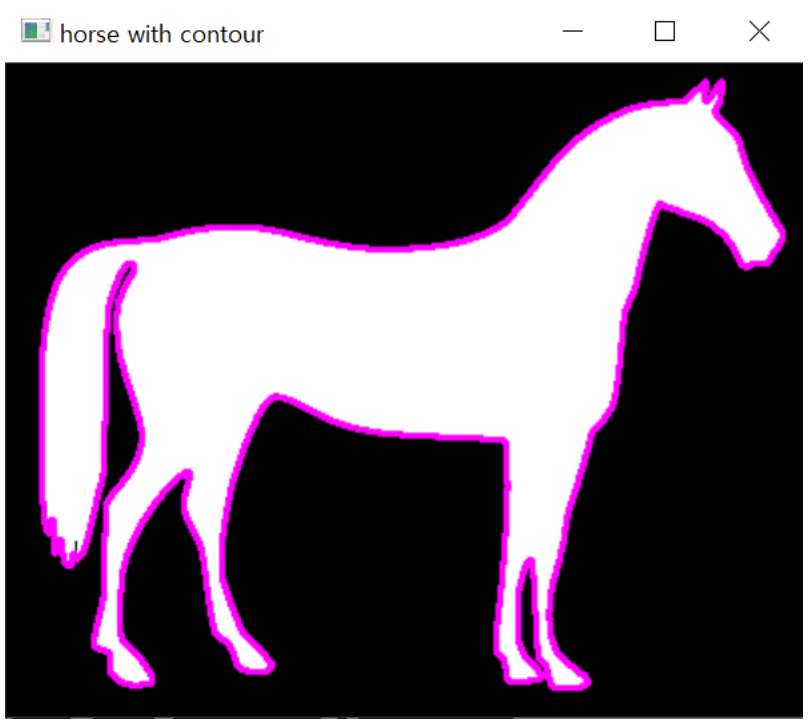
```python
contour_approx = cv2.approxPolyDP(contour, 8, True)
cv2.drawContours(img3, [contour_approx],-1, (0,255,0),2)

hull = cv2.convexHull(contour)
hull = hull.reshape(1, hull.shape[0], hull.shape[2])
cv2.drawContours(img3,hull, -1, (0,0,255),2)

cv2.imshow('horse with line segments and convex hull', img3)

cv2.waitKey()
cv2.destroyAllWindows()
```

```
면적 = 42390.0
중점 = (187.72464024534088, 144.43640402610677)
둘레 = 2296.7291333675385
둥근 정도 = 0.1009842680321435
```

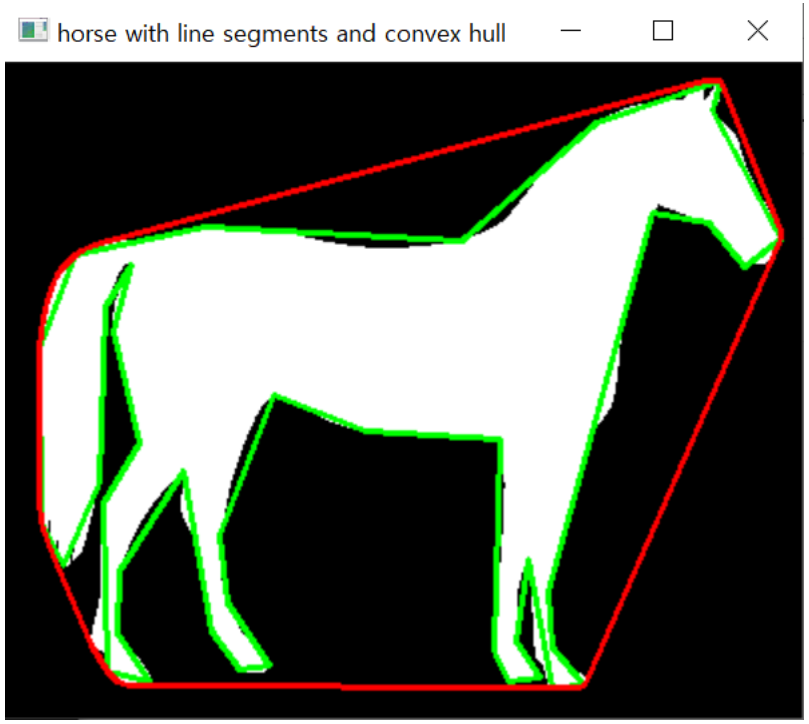
영역 특징 | 실행 결과



| horse



| horse with contour



| horse with segments and convex hull