

[Intel] 엣지 AI SW 아카데미

절차지향 프로그래밍

OpenCV 없이 Python으로 구현한 GrayScale Image Processing

이지원

프로젝트 개요

• 구현 목적

- ✓ OpenCV 없이 Python으로 구현한
GrayScale Image Processing
- ✓ 개발 기간: 2024.3.20~3.22 (C언어 버전 포팅)

• 개발 환경

- ✓ OS : Windows 11, Rocky9
- ✓ Tool : PyCharm 2023.3.4
- ✓ Language : Python

• 주요 기능

- ✓ 화소점 처리
- ✓ 기하학 처리
- ✓ 히스토그램 처리
- ✓ 화소 영역 처리
- ✓ 경계선 검출

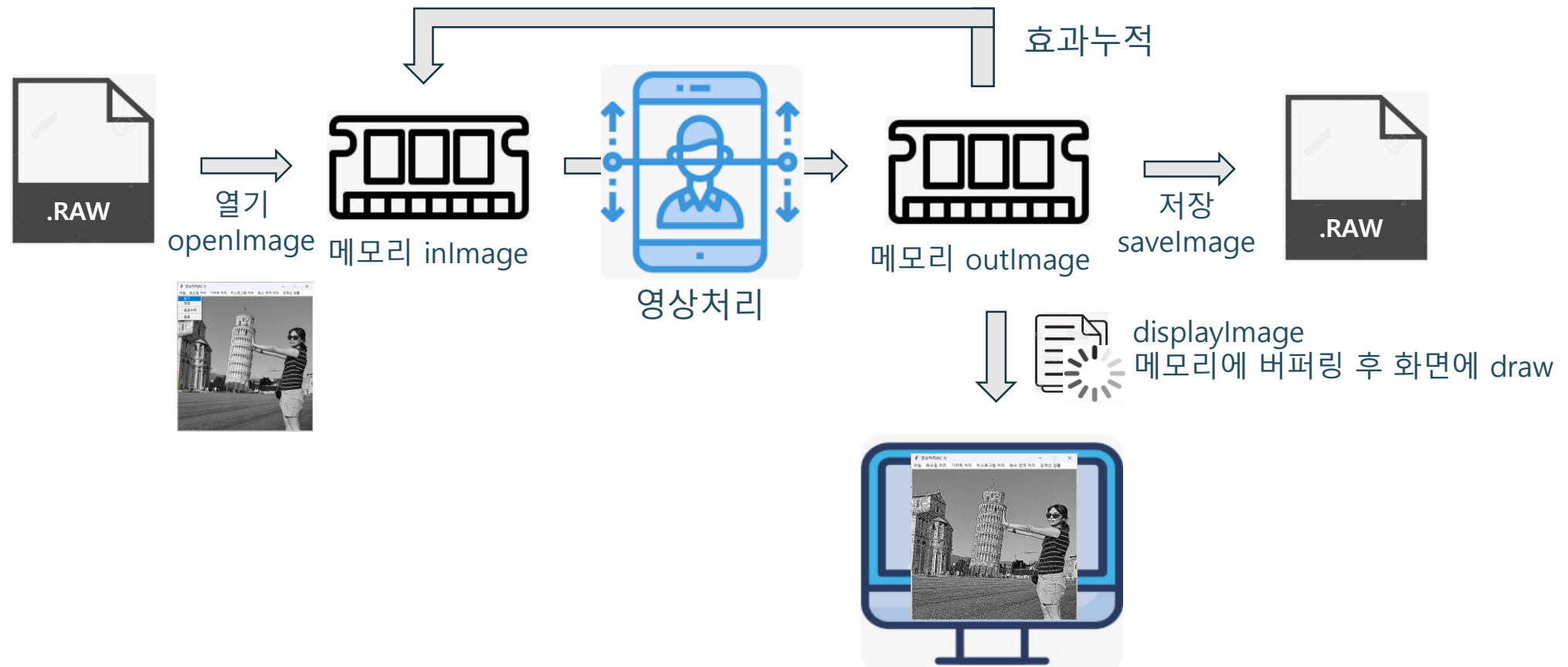
Windows 버전



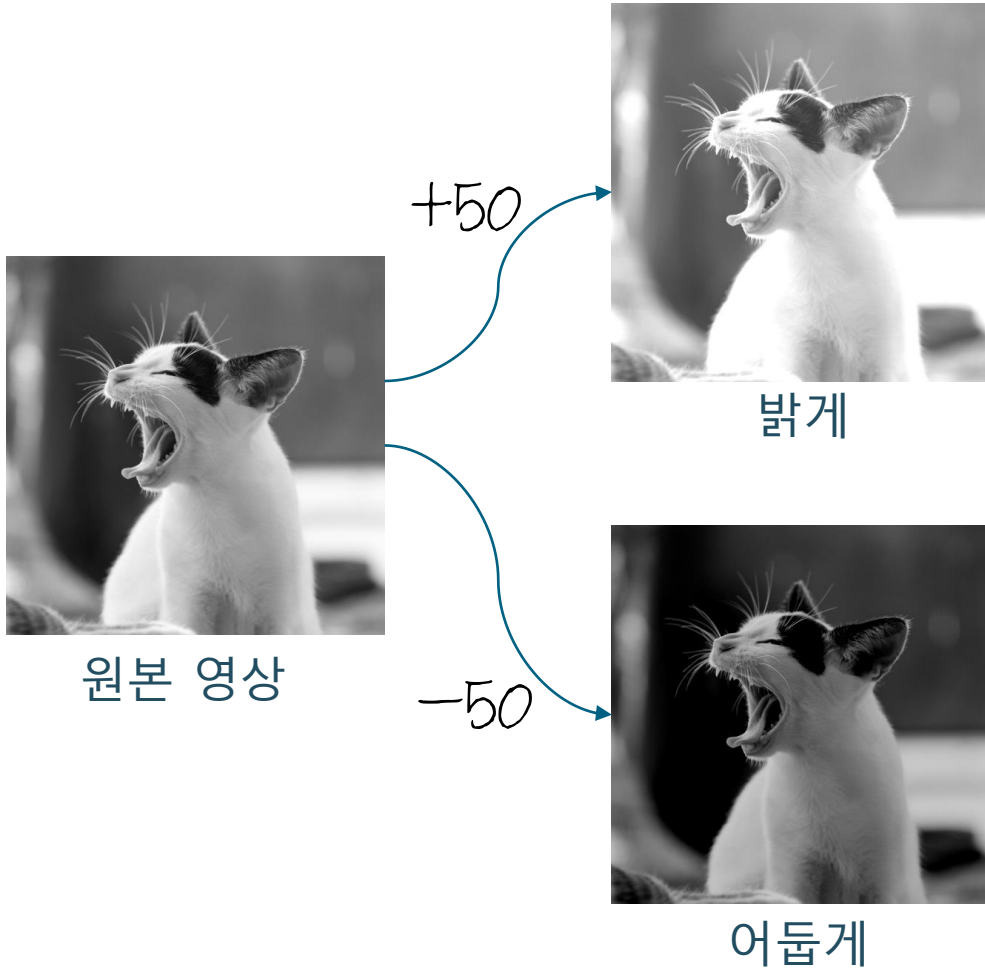
Linux 버전



프로그램 구조



1. 화소점 처리 ▶ 산술연산



✓ 화소점 처리란?

화소 점의 원래 값을 기준으로 화소 값을 변경하는 기술

1. 덧셈 연산

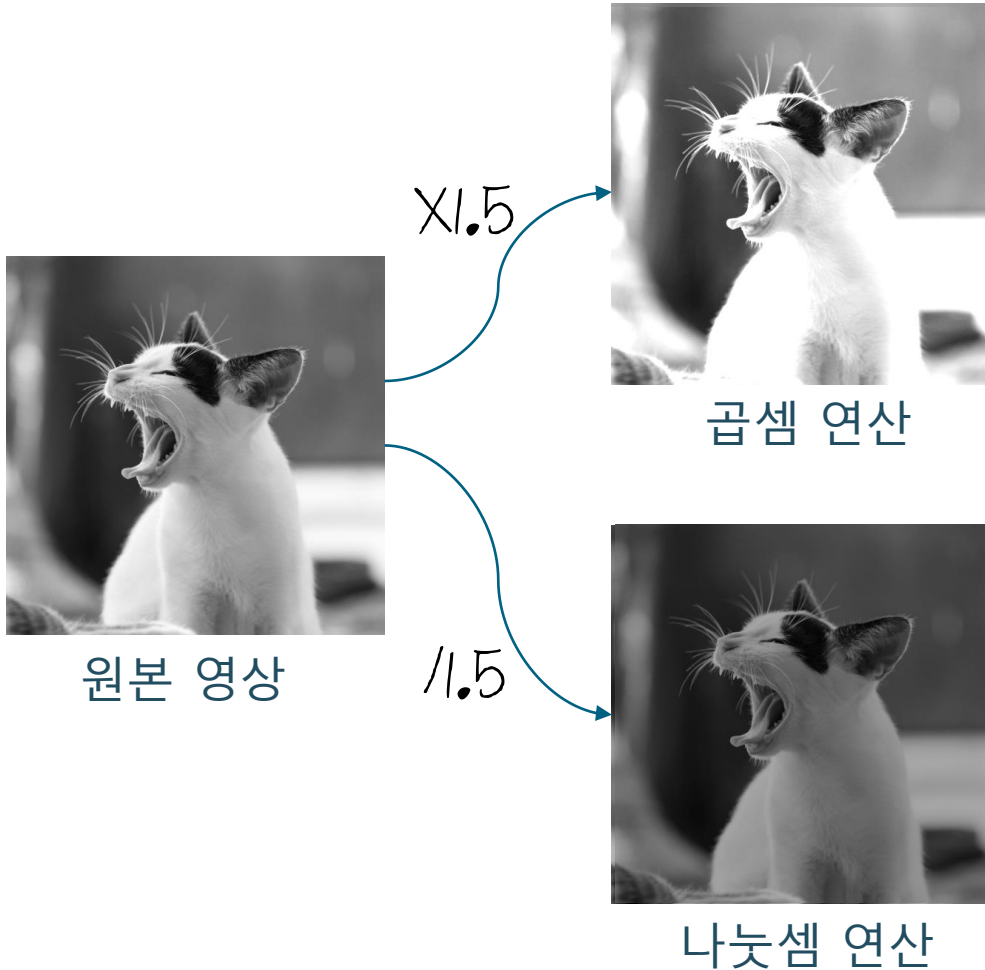
→ 이미지를 밝게

2. 뺄셈 연산

→ 이미지를 어둡게

```
px = inImage[i][k] + value
```

1. 화소점 처리 ▶ 산술연산



3. 곱셈 연산

→ 이미지를 밝게

```
px = int(inImage[i][k] * val)
```

4. 나눗셈 연산

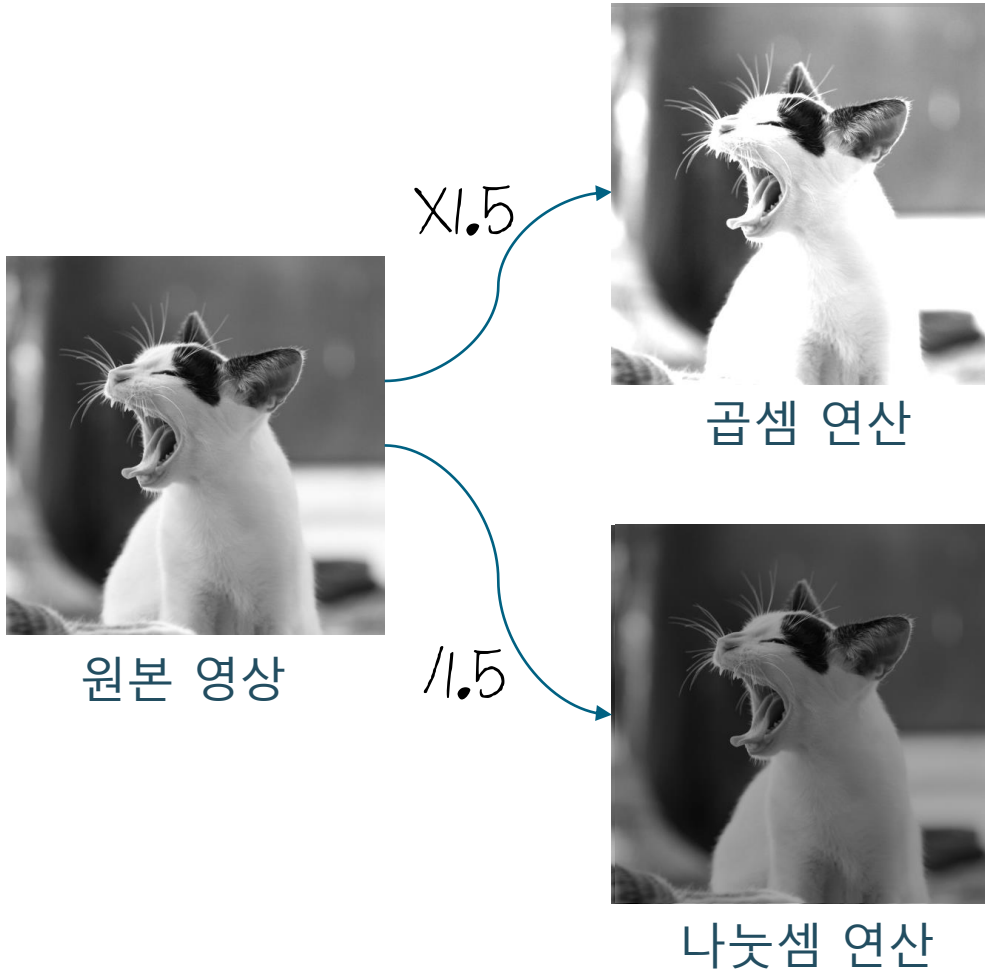
→ 이미지를 어둡게

```
px = int(inImage[i][k] / val)
```

```
if (px > 255) :  
    outImage[i][k] = 255  
elif(px < 0) :  
    outImage[i][k] = 0  
else :  
    outImage[i][k] = px
```

예외처리

1. 화소점 처리 ▶ 산술연산



3. 곱셈 연산

→ 이미지를 밝게

```
px = int(inImage[i][k] * val)
```

4. 나눗셈 연산

→ 이미지를 어둡게

```
px = int(inImage[i][k] / val)
```

```
if (px > 255) :  
    outImage[i][k] = 255  
elif(px < 0) :  
    outImage[i][k] = 0  
else :  
    outImage[i][k] = px
```

예외처리

1. 화소점 처리 ▶ 논리연산



원본 영상



AND 128



OR 128



XOR 128



NOT (반전)

5. AND 연산

```
px = inImage[i][k] & val
```

6. OR 연산

```
px = inImage[i][k] | val
```

7. XOR 연산

→ 입력이 다를 때만 1로

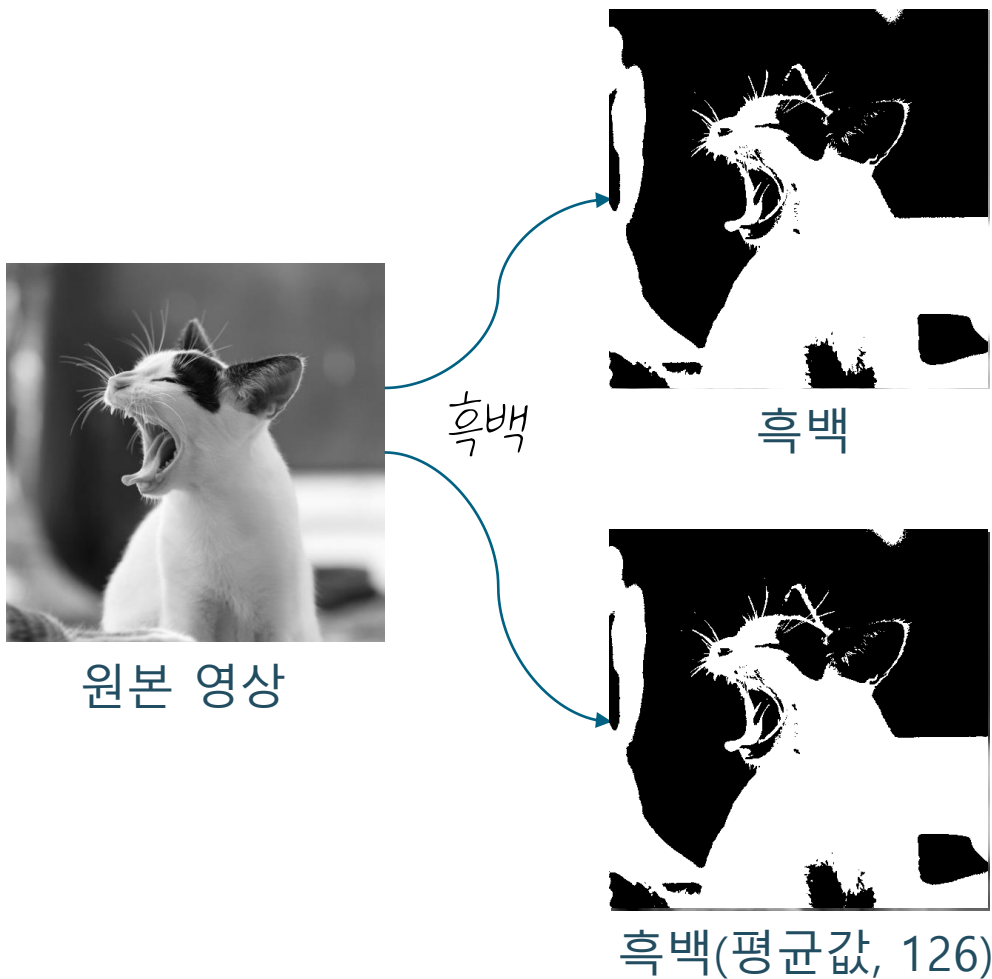
```
px = inImage[i][k] ^ val
```

8. NOT 연산 (반전)

→ 검정색은 흰색으로, 흰색은 검정색으로

```
outImage[i][k] = 255 - inImage[i][k]
```

1. 화소점 처리 ▶ 흑백 (이진화)



✓ 경계 값을 이용해 값이 두 개만 있는 영상으로 변환

$$\text{Output}(q) = \begin{cases} 255 & \text{Input}(p) \geq T \\ 0 & \text{Input}(p) < T \end{cases}$$

9. 흑백

→ $T = 128$

```
if (inImage[i][k] > 127) :  
    outImage[i][k] = 255  
else :  
    outImage[i][k] = 0
```

10. 흑백(평균값)

→ $T = \text{평균값}$

```
for i in range(inH):  
    for k in range(inW):  
        sum += inImage[i][k]  
    avg = sum / (outW * outH)
```


1. 화소점 처리 ▶ 흑백 (이진화)



원본 영상



흑백(중앙값, 113)

11. 흑백(중앙값)

→ T= 중앙값

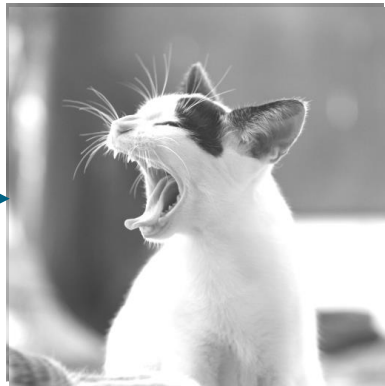
```
for i in range(inH):
    for k in range(inW):
        array.append(inImage[i][k])
array.sort()

if (inImage[i][k] < median):
    outImage[i][k] = 0
else:
    outImage[i][k] = 255
```

1. 화소점 처리 ▶ 감마보정



원본 영상

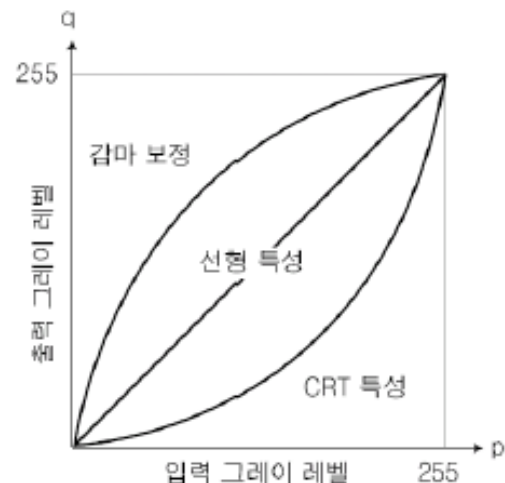


감마 값 0.7로 보정



감마 값 1.2로 보정

(d) 감마 보정 변환 함수 그래프



12. 감마보정

- ✓ 인간의 시각이 비선형적이기 때문에 비선형 전달 함수를 거쳐 빛의 강도에 변화를 주는 과정
- ✓ 함수의 감마 값에 따라 밝기 조정
- ✓ 감마 값이 1보다 크면 영상이 어두워지고, 1보다 작으면 영상이 밝아짐

```
px = inImage[i][k]  
outImage[i][k] = int(255.0 * (px / 255.0) ** (val))
```

감마 값

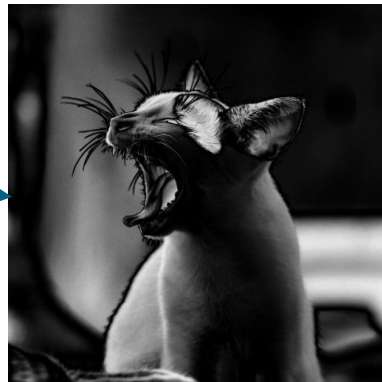
1. 화소점 처리 ▶ 파라볼라



원본 영상



파라볼라 CAP 보정



파라볼라 CUP 보정

✓ 영상에 입체감을 줌

13. 파라볼라 CAP

✓ 밝은 곳 입체형

```
px = inImage[i][k]  
val = 255.0 - 255.0 * pow((px / 128.0 - 1.0), 2)
```

14. 파라볼라 CUP

✓ 어두운 곳 입체형

```
px = inImage[i][k]  
val = 255.0 * pow((px / 128.0 - 1.0), 2)
```

1. 화소점 처리 ▶ 포스터라이징



원본 영상



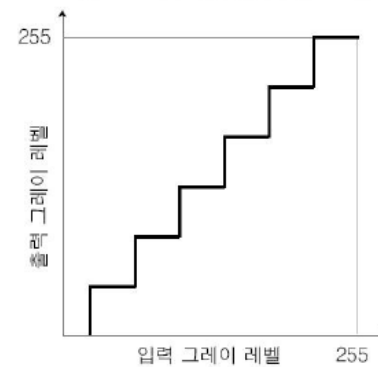
6단계로 변환된 영상

15. 포스터라이징

✓ 명암 값의 범위를 경계 값으로 축소

```
for i in range(val-1): # 영역 입력값에 비례하여 할당
    for j in range(inH):
        for k in range(inW):
            if (inImage[j][k] > (interval * i) and inImage[j][k] < interval * (i + 1)):
                outImage[j][k] = interval * (i+1)
```

(c) 포스터라이징 변환 함수 그래프



1. 화소점 처리 ▶ 범위 강조 변환



원본 영상

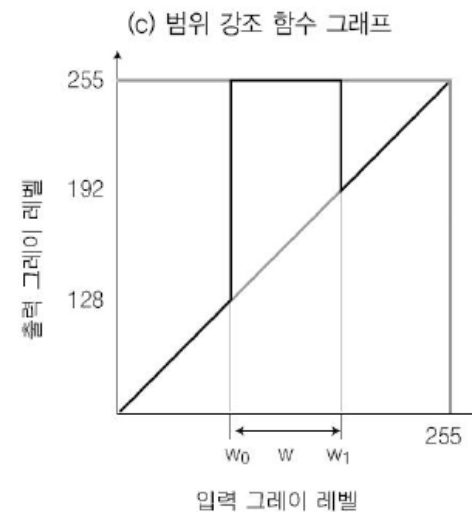


강조 변환
 $W0=100, W1=140$

16. 범위 강조 변환

✓ 일정 범위의 화소만 강조하는 변환

```
px = inImage[i][k]
if (px >= start and px <= end) :
    outImage[i][k] = 255
else :
    outImage[i][k] = px
```



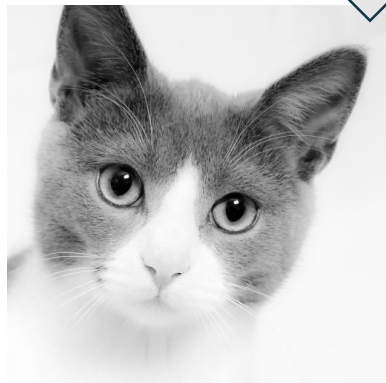
2. 기하학 처리



원본 영상



이동



좌우대칭



상하대칭

1. 이동

```
for i in range(inH) :  
    for k in range(inW) :  
        x = k - moveX  
        y = i - moveY  
        if (x >= 0 and x < inW and y >= 0 and y < inH)  
            outImage[i][k] = inImage[y][x]
```

2. 좌우대칭

- ✓ 세로축을 중심으로 뒤집음

```
outImage[i][k] = inImage[i][outW - 1 - k]
```

3. 상하대칭

- ✓ 가로축을 중심으로 뒤집음

```
outImage[i][k] = inImage[inH - 1 - i][k]
```

2. 기하학 처리 ▶ 확대



원본 영상

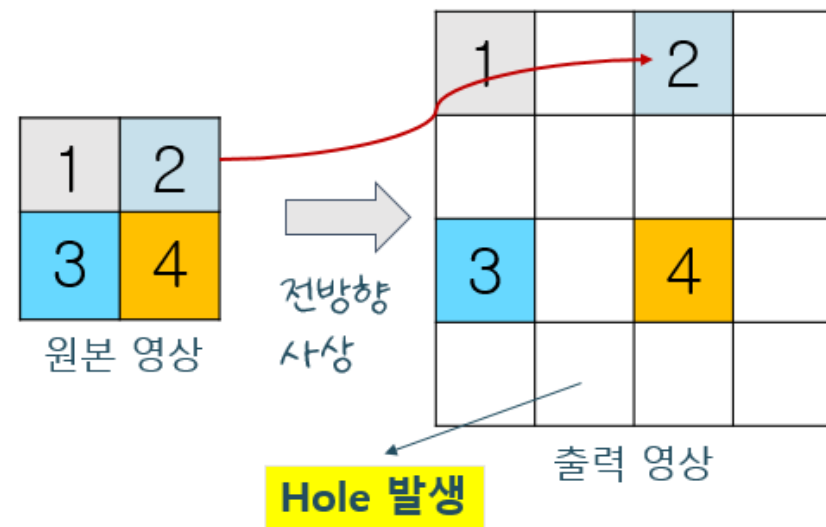


2배 확대, Hole 발생

```
outImage[i*scale][k*scale] = inImage[i][k]  
outImage[i*scale+1][k*scale+1] = inImage[i][k]
```

4. 확대(포워딩)

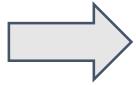
- ✓ 전방향 사상
- ✓ Hole 문제: 임의의 화소가 목적 영상의 화소에 사상되지 않음



2. 기하학 처리 ▶ 확대



원본 영상

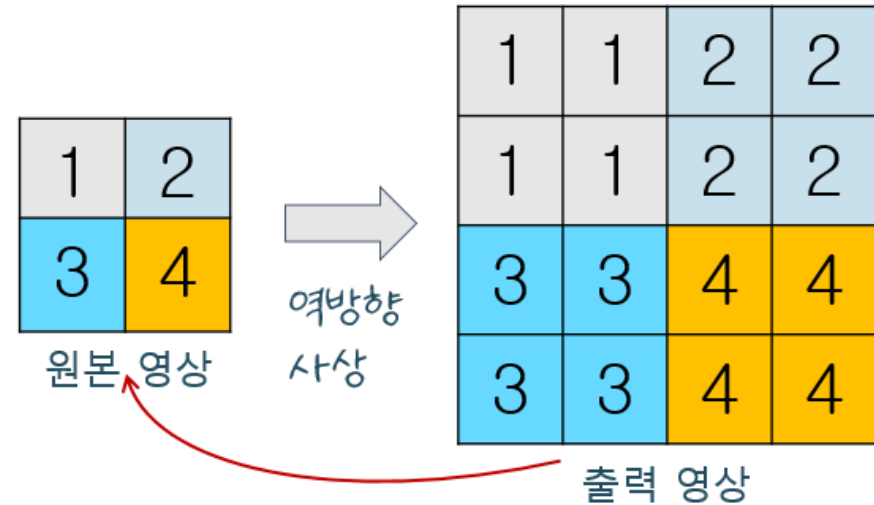


2배 확대, Hole 발생 X

```
for i in range(outH) :  
    for k in range(outW) :  
        outImage[i][k] = inImage[int(i/scale)][int(k/scale)]
```

5. 확대(백워딩)

✓ 역방향 사상: 홀 문제가 일어나지 않음



2. 기하학 처리 ▶ 확대



원본 영상

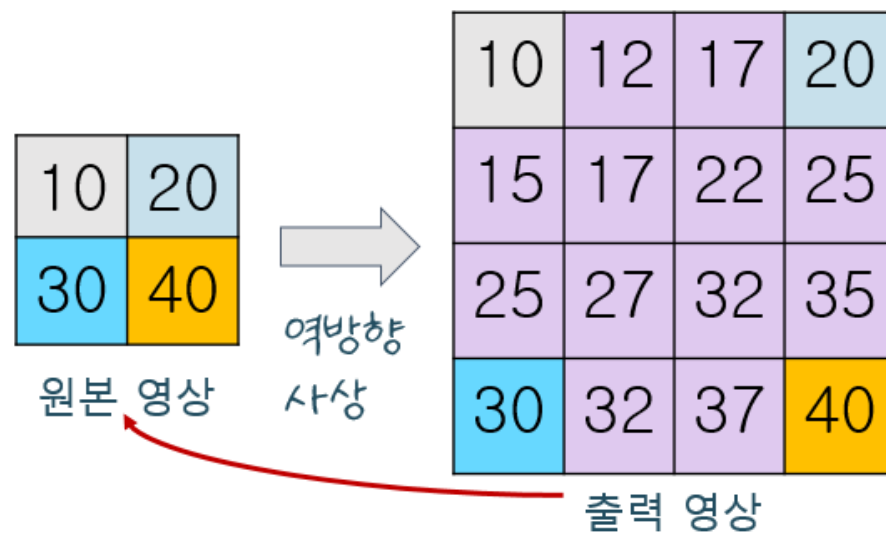


2배 확대, 양선형 보간

```
if (i_H < 0 or i_H >= (inH - 1) or i_W < 0 or i_W >= (inW - 1)) :  
    outImage[i][k] = 255  
else : # 소수점 값 보간하기  
    C1 = float(tempImage[i_H][i_W])  
    C2 = float(tempImage[i_H][i_W + 1])  
    C3 = float(tempImage[i_H + 1][i_W + 1])  
    C4 = float(tempImage[i_H + 1][i_W])  
  
    newValue = int(C1 * (1 - s_H) * (1 - s_W) + C2 * s_W * (1 - s_H) + C3 * s_W * s_H + C4 * (1 - s_W) * s_H)  
    outImage[i][k] = newValue
```

6. 확대(양선형 보간법)

✓ 양선형 보간법: 화소당 선형 보간을 세 번하여, 가장 가까운 화소 네 개에 가중치를 곱한 값을 합해서 얻음



2. 기하학 처리 ▶ 축소



원본 영상



```
for i in range(inH) :  
    for k in range(inW) :  
        outImage[int(i/scale)][int(k/scale)] = inImage[i][k]
```

7. 축소

- ✓ 에일리어싱: 영상의 크기를 많이 축소하면, 세부 내용을 상실하게 되는 현상

10	12	17	20
15	17	22	25
25	27	32	35
30	32	37	40

원본 영상

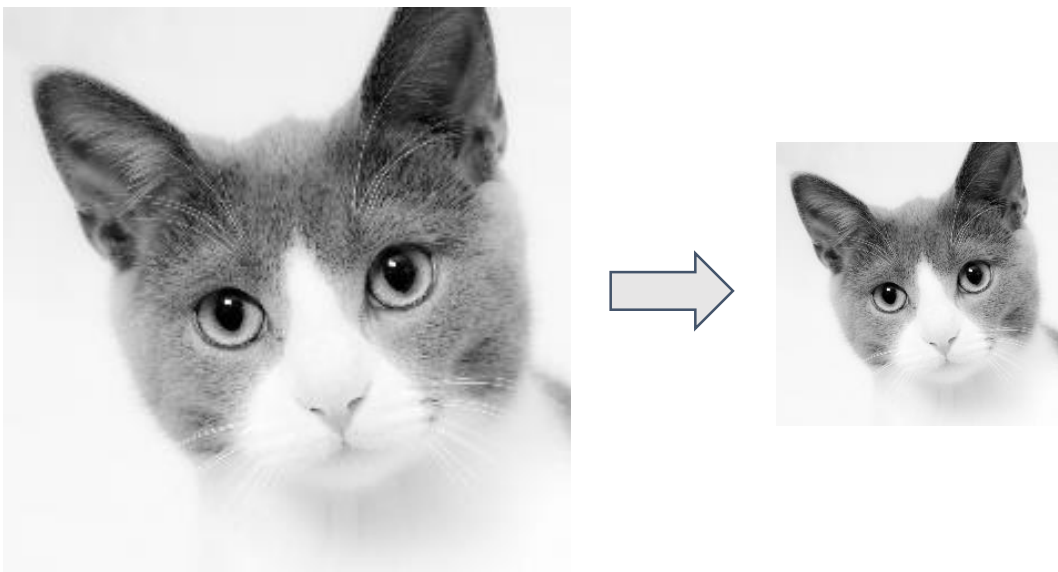


전방향
사상

10	17
25	32

출력 영상

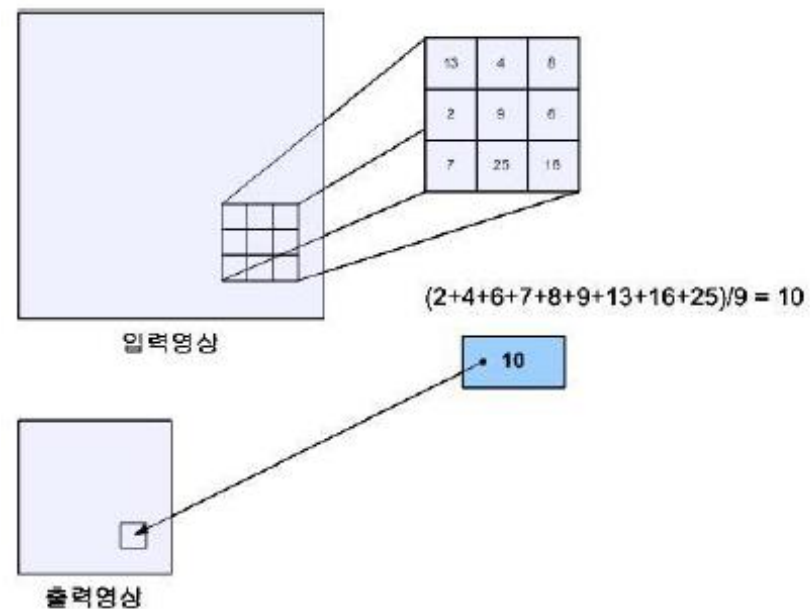
2. 기하학 처리 ▶ 축소



원본 영상

```
sum, avg = [0] * 2
for x in range(scale):
    for y in range(scale):
        sum += inImage[i+x][k+y]
avg = int(sum/(scale*scale))
outImage[int(i/scale)][int(k/scale)] = avg
```

8. 축소 (평균값)



[그림 8-29] 평균 표현을 이용한 서브 샘플링의 동작

2. 기하학 처리 ▶ 축소



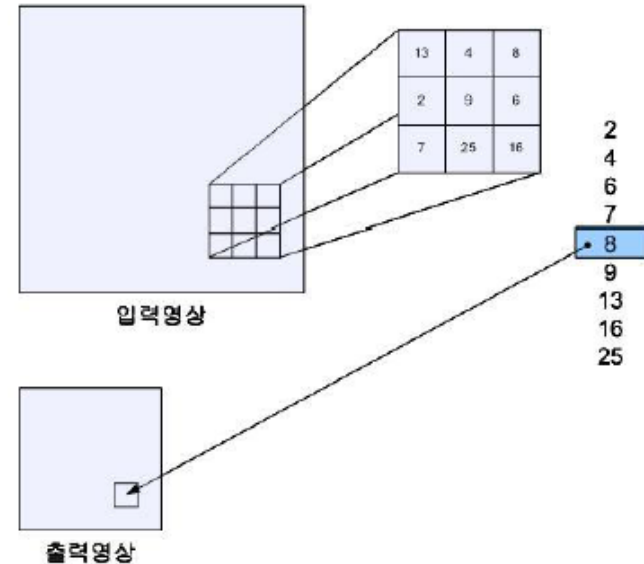
원본 영상

```
number = 0
for x in range(scale):
    for y in range(scale):
        array[number] = inImage[i+x][k+y]
        number += 1
array.sort()
med = array[int(scale*scale/2)]
outImage[int(i/scale)][int(k/scale)] = med
```



2배 축소

9. 축소 (중앙값)

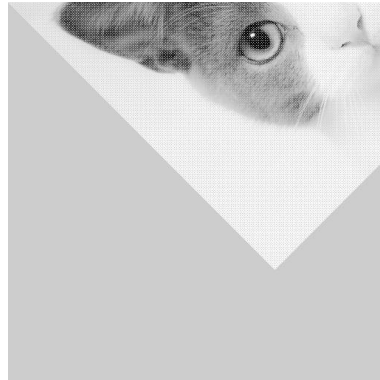
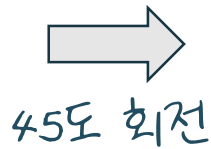


[그림 8-28] 미디언 표현을 이용한 서브 샘플링 동작 과정

2. 기하학 처리 ▶ 회전



원본 영상



전방향사상 시 홀 문제 발생
원점 기준 회전

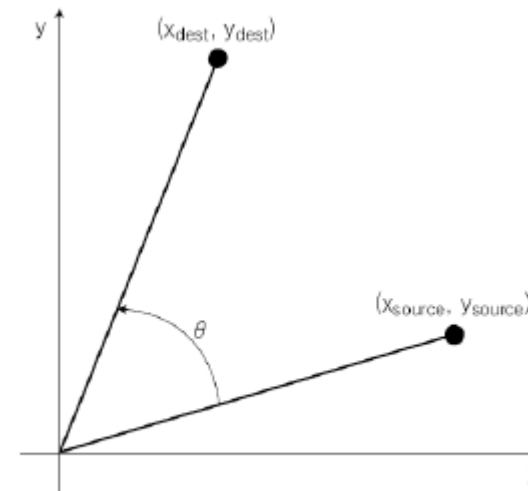
```
xs = i
ys = k
xd = (int)(math.cos(radian) * xs - math.sin(radian) * ys)
yd = (int)(math.sin(radian) * xs + math.cos(radian) * ys)

if ((0 <= xd and xd < outH) and (0 <= yd and yd < outW)) :
    outImage[xd][yd] = inImage[xs][ys]
```

10. 회전

✓ 회전: 영상을 특정한 각도만큼 회전시키는 것

$$\begin{bmatrix} x_{dest} \\ y_{dest} \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_{source} \\ y_{source} \end{bmatrix}$$

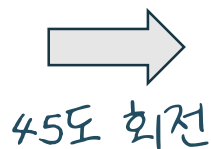


[그림 9-6] 회전 변환하여 좌표 변화

2. 기하학 처리 ▶ 확대



원본 영상



중심점을 기준 회전

```
cx = int(inH / 2)
cy = int(inW / 2)
xs = int(math.cos(radian) * (xd - cx) + math.sin(radian) * (yd - cy))
ys = int(-math.sin(radian) * (xd - cx) + math.cos(radian) * (yd - cy))
if ((0 <= xs and xs < outH) and (0 <= ys and ys < outW)) :
    outImage[xd][yd] = inImage[xs][ys]
```

11. 회전 (중앙, 백워딩)

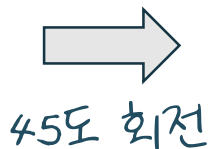
- ✓ 영상의 중심점을 기준으로 회전한 결과 보이는 부분이 줄어드는 것을 방지
- ✓ 역방향 사상: 홀 문제가 일어나지 않음
- ✓ 영상의 중심점이 (C_x, C_y) 이고, 역방향 사상을 고려한 공식

$$\begin{bmatrix} x_{source} \\ y_{source} \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_{dest} - C_x \\ y_{dest} - C_y \end{bmatrix} + \begin{bmatrix} C_x \\ C_y \end{bmatrix}$$

2. 기하학 처리 ▶ 확대



원본 영상



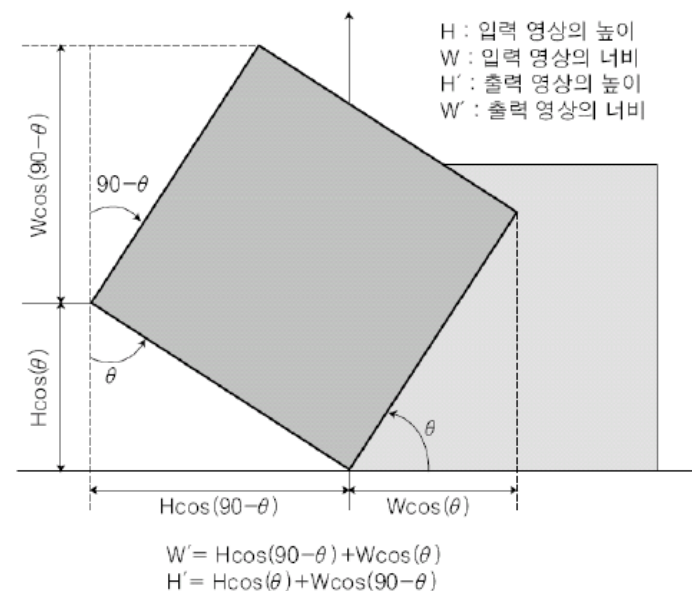
확대 회전

```
outH = int(inH * math.cos(radian2) + inW * math.cos(radian))
outW = int(inH * math.cos(radian) + inW * math.cos(radian2))
centerX = int(outH / 2)
centerY = int(outW / 2)
xs = int(math.cos(radian) * (xd - centerX) + math.sin(radian) * (yd - centerY))
ys = int(-math.sin(radian) * (xd - centerX) + math.cos(radian) * (yd - centerY))
if ((0 <= xs and xs < inH) and (0 <= ys and ys < inW)) :
    outImage[xd][yd] = inImage[xs][ys]
```

12. 회전 (확대)

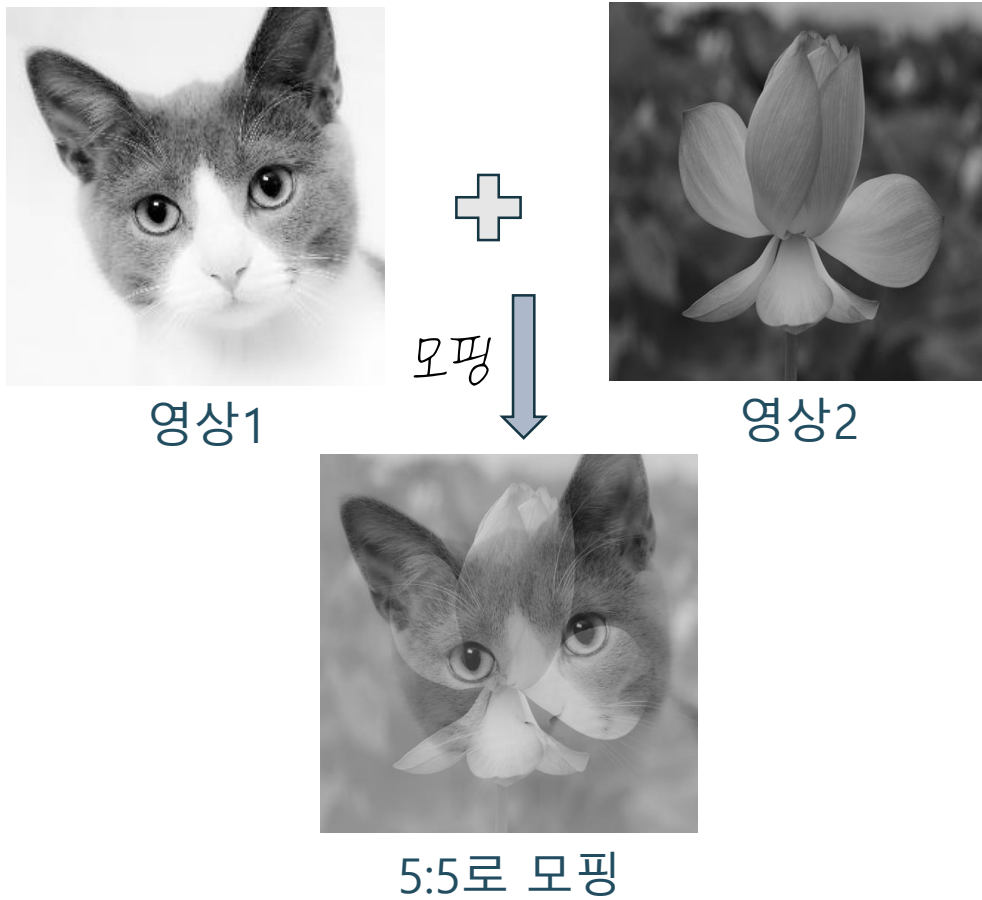
✓ 영상의 크기를 고려한 회전 변환

✓ 잘려 나가는 부분이 없게 출력 영상 크기 계산



[그림 9-14] 회전 기하학적 변환의 출력 영상 크기

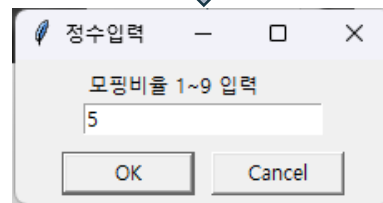
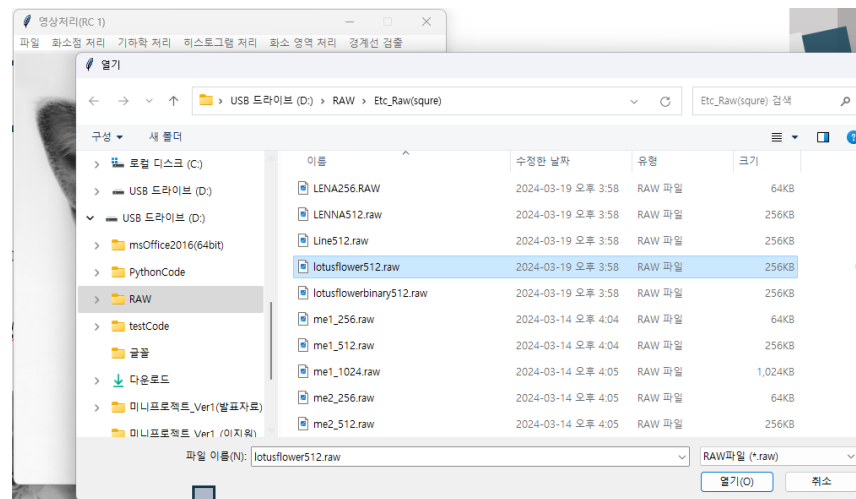
2. 기하학 처리 ▶ 모핑



```
outImage[i][k] = int((inImage[i][k] * (10. - morRate) / 10.))
```

13. 모핑

✓ 영상을 서서히 다른 영상으로 변환하는 기술

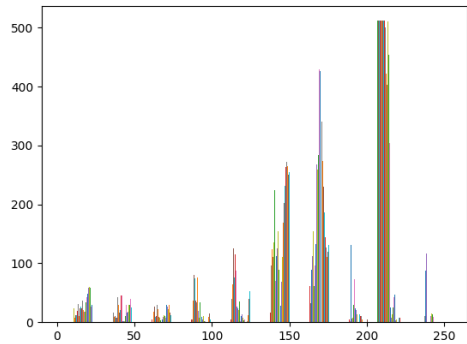


모핑 비율 선택 (1~10)

3. 히스토그램 처리 ▶ 스트레칭



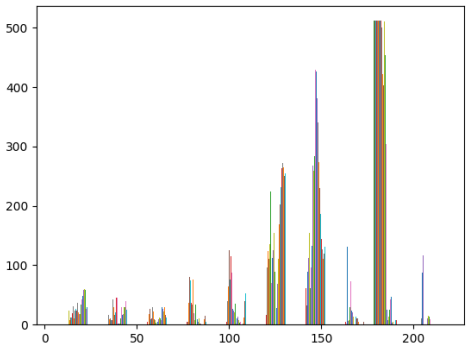
원본 영상



원본 히스토그램



스트레칭 영상



변환 히스토그램

✓ 히스토그램? 관측한 데이터가 분포된 특징을 한 눈에 볼 수 있도록 기둥 모양으로 나타낸 것

1. 히스토그램 스트레칭

✓ 명암 대비를 향상시키는 연산

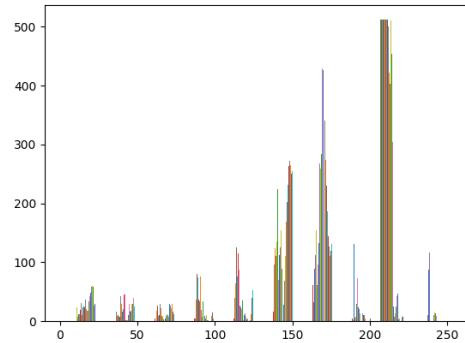
$$new\ pixel = \frac{old\ pixel - low}{high - low} \times 255$$

- old pixel은 원 영상 화소의 명도 값
- new pixel은 결과 영상 화소의 명도 값
- low는 히스토그램의 최저 명도 값
- high는 히스토그램의 최고 명도 값

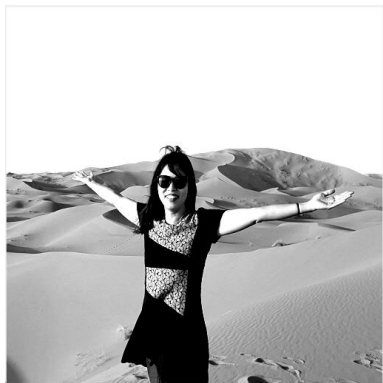
3. 히스토그램 처리 ▶ 앤드-인



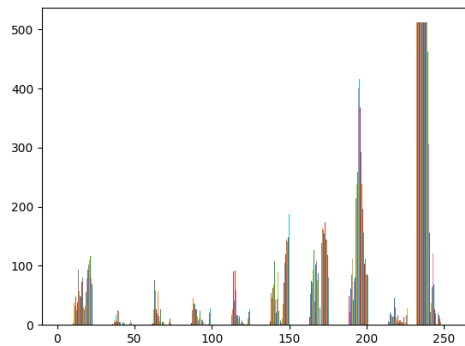
원본 영상



원본 히스토그램



앤드-인 영상



변환 히스토그램

2. 히스토그램 앤드-인

✓ 두 개의 임계 값을 사용하여 히스토그램의 분포를 좀더 균일하게 만듦

$$\text{new pixel} = \begin{cases} 0 & \text{old pixel} \leq \text{low} \\ \frac{\text{old pixel} - \text{low}}{\text{high} - \text{low}} \times 255, & \text{low} \leq \text{old pixel} \leq \text{high} \\ 255 & \text{high} \leq \text{old pixel} \end{cases}$$

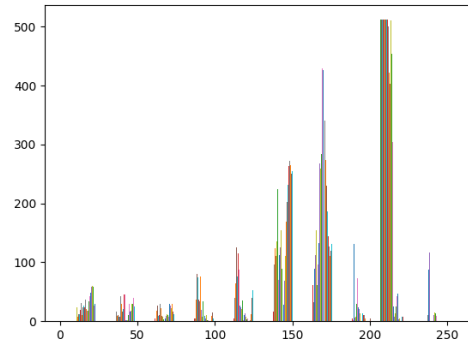
```
high -= 50
low += 50

old = inImage[i][k]
new = int(float(old - low) / float(high - low) * 255.0)
outImage[i][k] = new
```

3. 히스토그램 처리 ▶ 히스토그램 평활화



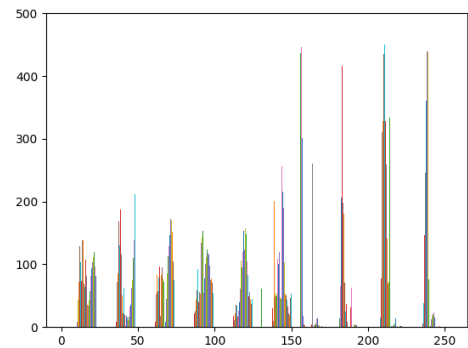
원본 영상



원본 히스토그램



평활화 영상



변환 히스토그램

3. 히스토그램 평활화

- ✓ 분포가 빈약한 영상을 균일하게 만듦
- ✓ 영상의 밝기 분포를 재분배하여 명함 대비 최대
- ✓ 1단계 : 히스토그램 빈도수 세기

```
histo[inImage[i][k]] += 1
```

- ✓ 2단계 : 누적히스토그램 생성

```
sumHisto[i] = sumHisto[i - 1] + histo[i]
```

- ✓ 3단계 : 정규화된 히스토그램 생성

```
normalHisto[i] = sumHisto[i] * (1.0 / (inH * inW)) * 255.0
```

- ✓ 4단계 : 입력 이미지를 정규화된 값으로 치환

```
outImage[i][k] = int(normalHisto[inImage[i][k]])
```

4. 화소 영역 처리

- ✓ 원래 화소와 이웃한 각 화소의 가중치를 곱한 합을 출력 화소로 생성

$$Output_pixel[x,y] = \sum_{m=(x-k)}^{x+k} \sum_{n=(y-k)}^{y+k} (I[m,n] \times M[m,n])$$

- **Output_pixel[x, y]**: 회선 처리로 출력한 화소
- **I[m, n]**: 입력 영상의 화소
- **M[m, n]**: 입력 영상의 화소에 대응하는 가중치

I ₁	I ₂	I ₃
I ₄	I ₅	I ₆
I ₇	I ₈	I ₉

(a) 입력 영상

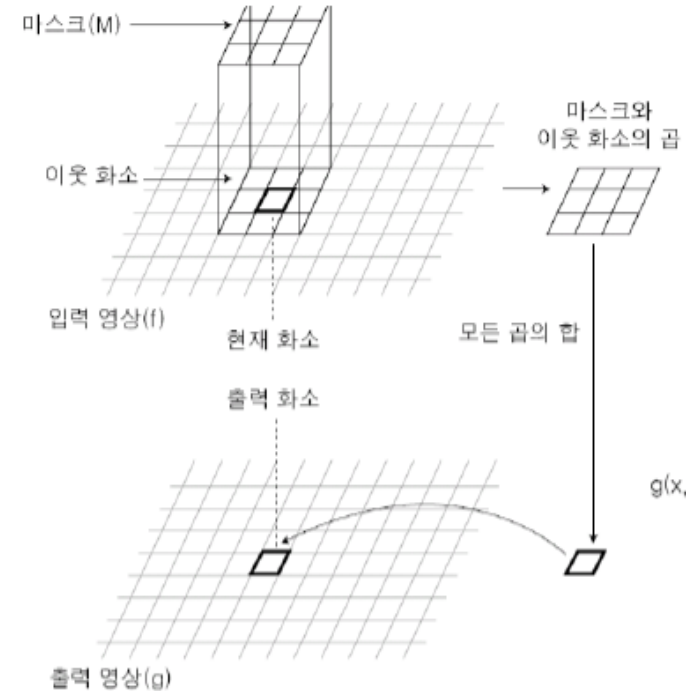
M ₁	M ₂	M ₃
M ₄	M ₅	M ₆
M ₇	M ₈	M ₉

(b) 회선 마스크

출력 픽셀 값 :

$$I_1 \times M_1 + I_2 \times M_2 + I_3 \times M_3 + I_4 \times M_4 + I_5 \times M_5 + I_6 \times M_6 + I_7 \times M_7 + I_8 \times M_8 + I_9 \times M_9$$

- ✓ 가중치를 포함한 회선 마스크가 이동하면서 수행

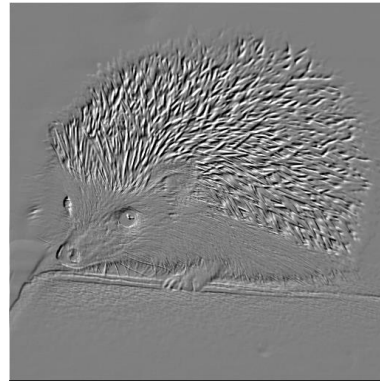


[그림 6-7] 디지털 영상에서 회선을 처리하는 과정

4. 화소 영역 처리 ▶ 엠보싱



원본 영상



결과 영상

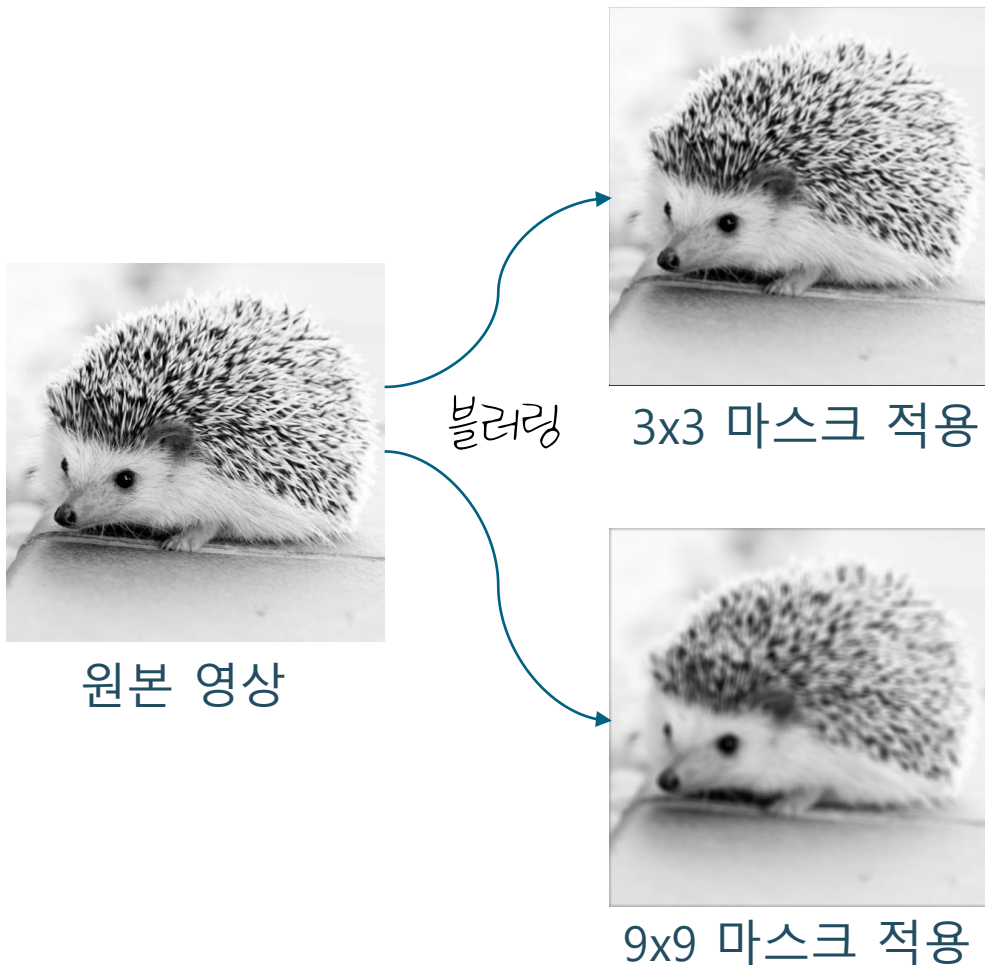
```
mask = [[-1.0, 0.0, 0.0], # 엠보싱 마스크  
        [0.0, 0.0, 0.0],  
        [0.0, 0.0, 1.0]]
```

1. 엠보싱

✓ 적절하게 구분된 경계선으로 영상이 양각된 것처럼 느껴짐

```
# ** 회선 연산 **  
for i in range(inH):  
    for k in range(inW):  
        # 마스크(3x3)와 한점을 중심으로 한 3x3을 곱하기  
        S = 0.0 # 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
        for m in range(3):  
            for n in range(3):  
                S += tmpInImage[i + m][k + n] * mask[m][n]  
        tmpOutImage[i][k] = S
```

4. 화소 영역 처리 ▶ 블러링



2. 블러링(3x3), (9x9)

- ✓ 영상의 세밀한 부분을 제거하여 영상을 흐리거나 부드럽게 하는 기술
- ✓ 영상의 세밀한 부분은 고주파 성분인데, 고주파 성분을 제거해 줌.
- ✓ 블러링 회선 마스크는 모든 계수가 양수로 전체 합은 1

```
mask = [[1./9, 1./9, 1./9],  
        [1./9, 1./9, 1./9],  
        [1./9, 1./9, 1./9]]
```

3x3 마스크

```
[[1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81],  
 [1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81, 1./81]]
```

9x9 마스크

4. 화소 영역 처리 ▶ 샤프닝

3. 샤프닝

- ✓ 영상의 상세한 부분을 더욱 강조하여 표현
- ✓ 상세한 부분은 고주파 성분이므로, 저주파 성분을 제거하면 샤프닝 효과



원본 영상



결과 영상

```
mask = [[-1., -1., -1.],  
        [-1., 9., -1.],  
        [-1., -1., -1.]]
```

4. 고주파 샤프닝

- ✓ 고주파 통과 필터링: 낮은 주파수 성분이 제거되어 경계선이 확연하게 보임
- ✓ 고주파 강조: 중요 성분이 남은 채 경계선 부분이 강조됨



고주파 통과



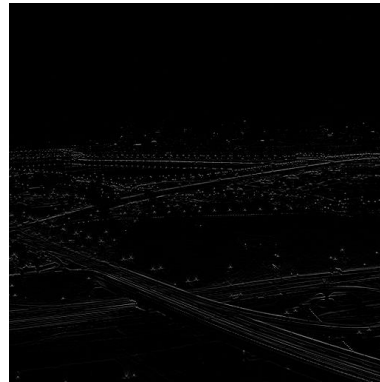
고주파 강조

```
mask = [[-1./9., -1./9., -1./9.],  
        [-1./9., 8./9., -1./9.],  
        [-1./9., -1./9., -1./9.]]
```


5. 경계선 검출 ▶ 에지 검출기



원본 영상



수직 에지 검출

```
mask = [[0.0, 0.0, 0.0],  
        [-1.0, 1.0, 0.0],  
        [0.0, 0.0, 0.0]]
```

✓ 에지(edge)

- 디지털 영상의 밝기가 높은 값에서 낮은 값으로 변하는 지점
- 영상을 구성하는 객체 간의 경계(=경계선)

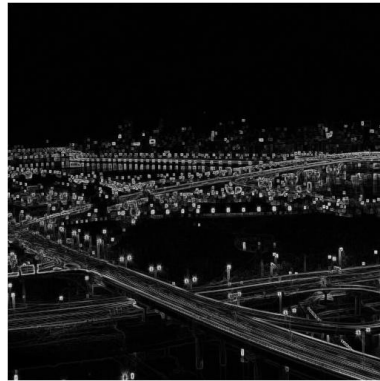
1. 에지 검출기 : 수직 에지 검출

- ✓ 화소 간의 차이를 이용하는 것

5. 경계선 검출 ▶ 유사 연산자



원본 영상



유사연산자로 에지 검출

```
for m in range(3):
    for n in range(3):
        if (math.fabs(tmpInImage[i + 1][k + 1] - tmpInImage[i + n][k + m]) >= max) :
            max = math.fabs(tmpInImage[i + 1][k + 1] - tmpInImage[i + n][k + m])
tmpOutImage[i][k] = max
```

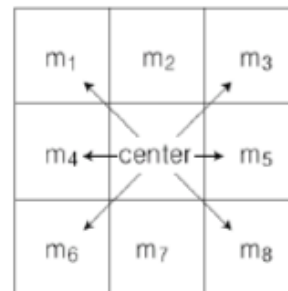
✓ 간단한 에지 추출 기법

- 연산 자체가 간단하고 빠름
- 유사 연산자(Homogeneity Operator)와 차 연산자(Difference Operator)가 있음

2. 유사 연산자

✓ 가장 단순한 에지 검출 방법

✓ 뿔셈연산이 여러 번 수행되어 계산 시간이 많이 소요

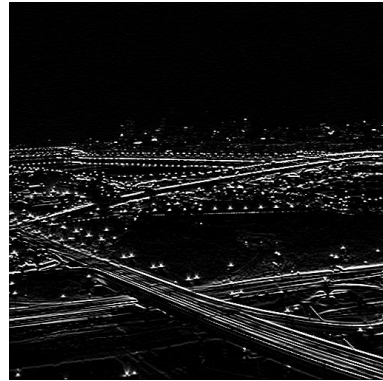


New Pixel = $\max(|\text{center}-m_1| \dots |\text{center}-m_8|)$
총 8번 수행

5. 경계선 검출 ▶ 차 연산자



원본 영상

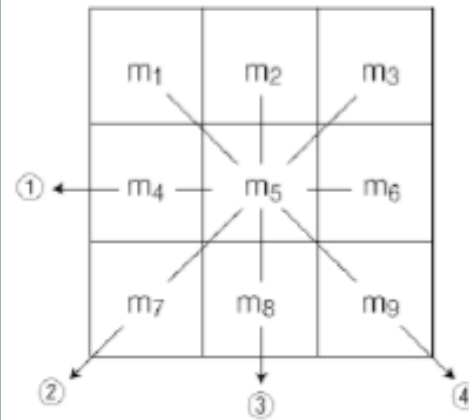


차연산자로 에지 검출

```
temp = math.fabs(tmpInImage[i][k] - tmpInImage[i + 2][k + 2])
if (temp >= max) :
    max = temp
temp = math.fabs(tmpInImage[i][k + 1] - tmpInImage[i + 2][k + 1])
if (temp >= max) :
    max = temp
temp = math.fabs(tmpInImage[i][k + 2] - tmpInImage[i + 2][k])
if (temp >= max) :
    max = temp
temp = math.fabs(tmpInImage[i + 1][k] - tmpInImage[i + 1][k + 1])
if (temp >= max) :
    max = temp
tmpOutImage[i][k] = max
```

3. 차 연산자

- ✓ 유사 연산자의 계산 시간이 오래 걸리는 단점을 보완
- ✓ 뿔셈 연산이 화소당 네 번으로 연산 시간이 빠름



5. 경계선 검출 ▶ 로버츠(Roberts)



원본 영상



로버츠 행 검출 마스크
로 에지 검출

```
mask = [[-1.0, 0.0, 0.0],  
        [0.0, 1.0, 0.0],  
        [0.0, 0.0, 0.0]]
```

마스크(3x3)와 한점을 중심으로 한 3x3 곱하기

```
S = 0.0
```

```
for m in range(3):
```

```
    for n in range(3):
```

```
        S += tmpInImage[i + m][k + n] * mask[m][n]
```

```
tmpOutImage[i][k] = S
```

✓ 미분을 이용한 에지 검출 방법

- 영상의 에지는 화소의 밝기 값이 급격히 변하는 부분
- 함수의 변화분을 찾는 미분 연산이 이용됨

4. 로버츠 마스크

✓ 1차 미분을 이용한 회선 마스크

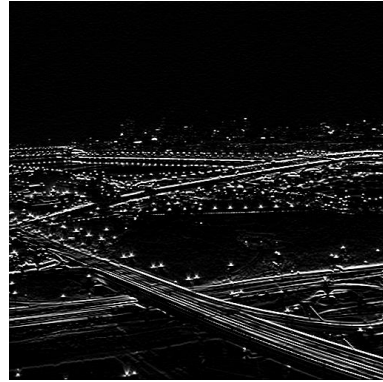
✓ 장점: 크기가 작아 빠른 속도로 동작

✓ 단점 : 돌출된 값을 잘 평균할 수 없으며, 잡음에 민감함

5. 경계선 검출 ▶ 소벨(Sobel)



원본 영상



소벨 행 검출 마스크로
에지 검출

```
mask = [[-1.0, -2.0, -1.0],  
        [0.0, 0.0, 0.0],  
        [1.0, 2.0, 1.0]]  
  
# 마스크(3x3)와 한점을 중심으로 한 3x3 곱하기  
S = 0.0 # 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
for m in range(3):  
    for n in range(3):  
        S += tmpInImage[i + m][k + n] * mask[m][n]  
tmpOutImage[i][k] = S
```

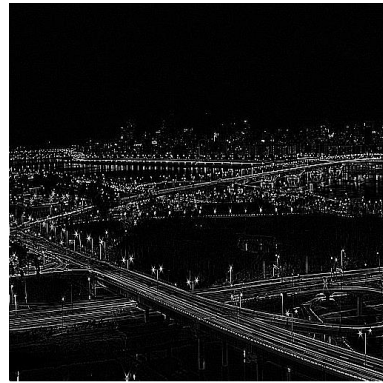
5. 소벨 마스크

- ✓ 1차 미분을 이용한 회선 마스크
- ✓ 장점: 돌출된 값을 비교적 잘 평균화 함.
- ✓ 단점: 대각선 방향에 놓인 에지에 더 민감하게 반응함

5. 경계선 검출 ▶ 라플라시안



원본 영상



라플라시안 회선 마스크로 에지 검출

```
mask = [[-1.0, -1.0, -1.0],  
        [-1.0, 8.0, -1.0],  
        [-1.0, -1.0, -1.0]] # 라플라시안 회선 마스크  
                                회선 마스크의 합은 0  
  
# 마스크(3x3)와 한점을 중심으로 한 3x3 곱하기  
S = 0.0 # 마스크 9개와 입력값 9개를 각각 곱해서 합한 값.  
for m in range(3):  
    for n in range(3):  
        S += tmpInImage[i + m][k + n] * mask[m][n]  
tmpOutImage[i][k] = S
```

✓ 2차 미분을 이용한 에지 검출

- 장점: 미분을 한번 더 수행하므로, 1차 미분의 단점(에지가 있는 영역을 지날 때 민감하게 반응)을 완화시킴
- 단점: 고립된 잡음에 민감하고, 윤곽의 강도만 검출하지 방향은 구하지 못함.

6. 라플라시안 연산자

- ✓ 대표적인 2차 미분 연산자로, 에지 검출 성능이 우수함
- ✓ 에지의 방향은 검출하지 못하고, 실제보다 많은 에지를 검출

5. 경계선 검출 ▶ LoG



원본 영상



LoG로 검출된 에지

```
mask = [[0.0, 0.0, -1.0, 0.0, 0.0],
        [0.0, -1.0, -2.0, -1.0, 0.0],
        [-1.0, -2.0, 16.0, -2.0, -1.0],
        [0.0, -1.0, -2.0, -1.0, 0.0],
        [0.0, 0.0, -1.0, 0.0, 0.0]]

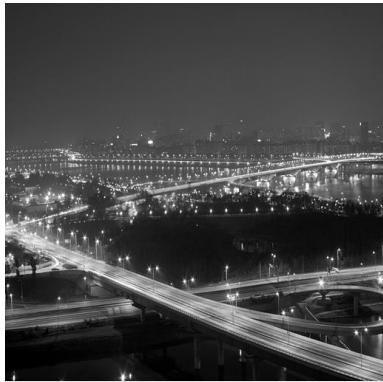
# 마스크(5x5)와 한점을 중심으로 한 5x5 곱하기
S = 0.0 # 마스크 25개와 입력값 25개를 각각 곱해서 합한 값.
for m in range(5):
    for n in range(5):
        S += tmpInImage[i + m][k + n] * mask[m][n]
tmpOutImage[i][k] = S
```

7. LoG(Laplacian of Gaussian) 연산자

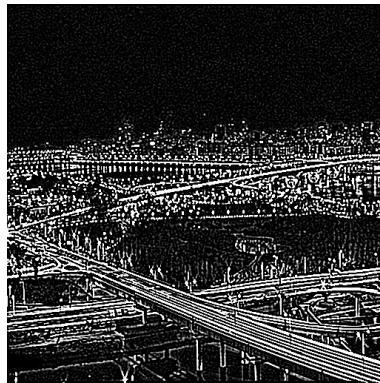
- ✓ 2차 미분 이용한 회선 마스크
- ✓ 잡음에 민감한 라플라시안의 문제를 해결하기 위해 만듦.
- ✓ 계산 시간이 많이 소요됨
- ✓ LoG 연산자 공식

$$\text{LoG}(x, y) = \frac{1}{\pi\sigma^4} \left[1 - \frac{(x^2 + y^2)}{2\sigma^2} \right] - e^{\frac{-(x^2 + y^2)}{2\sigma^2}}$$

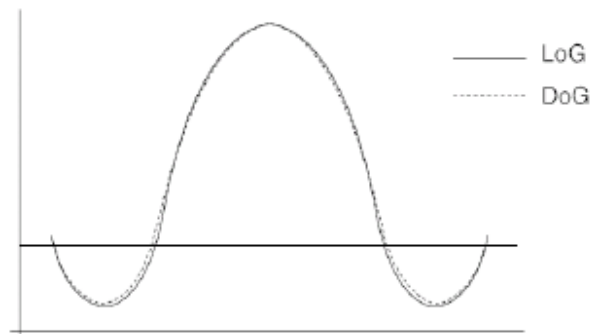
5. 경계선 검출 ▶ DoG



원본 영상



DoG로 검출된 에지



LoG와 DoG 그래프 비교

8. DoG(Difference of Gaussians) 연산자

✓ 2차 미분 이용한 회선 마스크

✓ 계산 시간이 많이 걸리는 LoG의 단점 보완 위해 등장

```
mask = [[0.0, 0.0, -1.0, -1.0, -1.0, 0.0, 0.0],
        [0.0, -2.0, -3.0, -3.0, -3.0, -2.0, 0.0],
        [-1.0, -3.0, 5.0, 5.0, 5.0, -3.0, -1.0],
        [-1.0, -3.0, 5.0, 16.0, 5.0, -3.0, -1.0],
        [-1.0, -3.0, 5.0, 5.0, 5.0, -3.0, -1.0],
        [0.0, -2.0, -3.0, -3.0, -3.0, -2.0, 0.0],
        [0.0, 0.0, -1.0, -1.0, -1.0, 0.0, 0.0]]
```

마스크(7x7)와 한점을 중심으로 한 7x7 곱하기

S = 0.0 # 마스크 81개와 입력값 81개를 각각 곱해서 합한 값.

```
for m in range(7):
```

```
    for n in range(7):
```

```
        S += tmpInImage[i + m][k + n] * mask[m][n]
```

```
tmpOutImage[i][k] = S
```

마무리

- 느낀 점
 - 영상 처리의 종류와 구현 방법에 대한 알게 됨
 - 라이브러리가 제공하는 기능을 low level로 구현함으로써 자신감을 얻음
 - C언어 버전을 Python 언어로 포팅하여 개발 시간 차이에 대해 비교
- 부족한 점
 - 컬러 이미지 영상 처리 불가능
 - 다양한 경우의 테스트 및 Error handling 필요
- 발전 방향 구현 내용
 - C++로 구현
 - Ctrl+Z (취소하기) 기능 구현

감사합니다

