OpenCV 없이 구현한

Linux 기반의 Python 영상처리

[Intel] 엣지 AI SW 아카데미
객체지향 프로그래밍

허찬욱

프로젝트 개요

프로젝트 목표

이미지 파일을 Python로 구현한
알고리즘으로 보정하기

프로젝트 기간

2024.03.20 ~ 2024.03.23

프로그램명

GrayScale Image Processing
(Version 1.0 Spin-off)

주요 기능

화소점 처리, 기하학 처리, 화소영역 처리,
히스토그램 처리

개발환경

Rocky Linux 9.0 (Blue Onyx) 64bit
PyCharm Community Edition 2023.3.4

전체 소스코드 링크

https://blog.naver.com/hew0916/

특이사항

Raw 형식의 파일만 저장 가능
정방형 & GrayScale Image 사용

구조도

D:/Mypic

mypet.raw
(512X512)

openImage()

메모리(RAM)

inW

inH

inImage 배열

영상처리 알고리즘

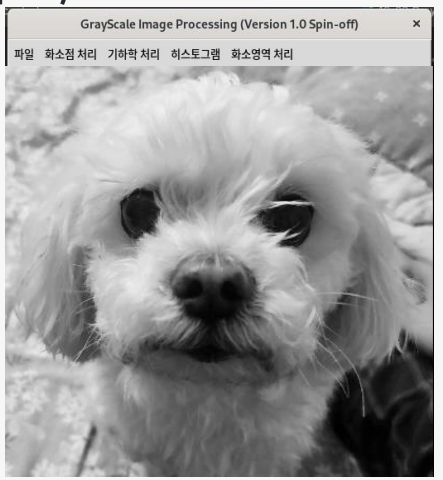
outH

outImage 배열

outW

displayImage()

Display

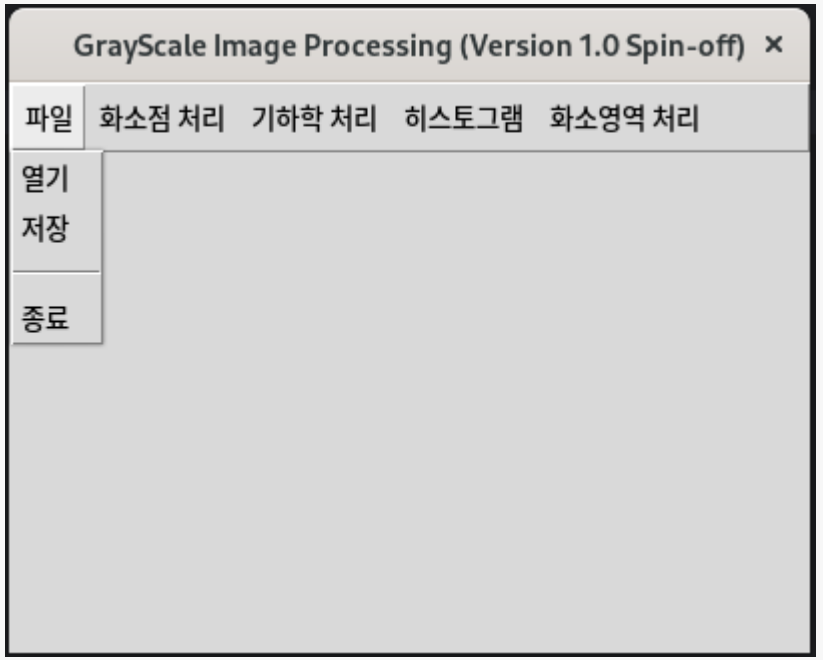


out1.raw
(256x256)

out2.raw
(512x512)

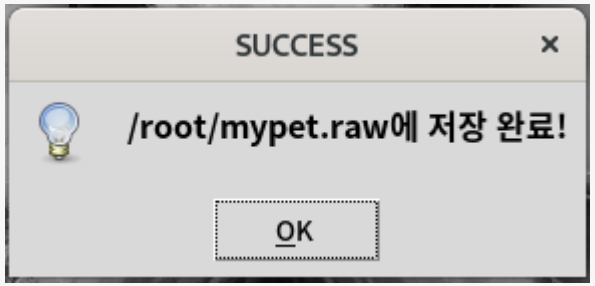
saveImage()

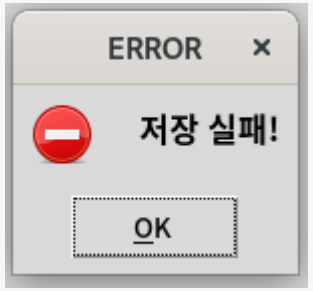
메인화면 구성



[열기]를 누른 후 파일을 선택하면,
해당 파일이 Print 됨

저장 알림 창

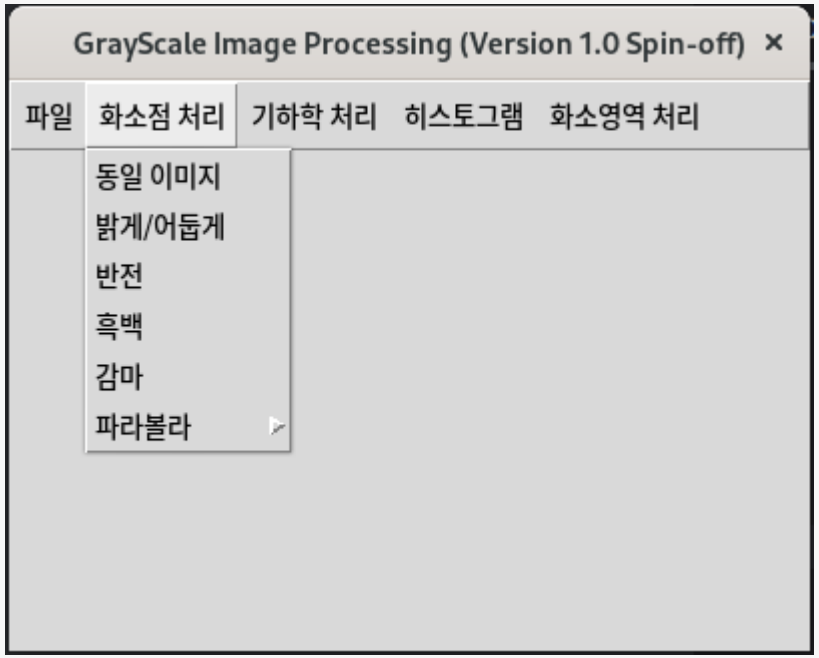




영상처리 알고리즘을 활용해 이미지 편집 후
[파일] – [저장]을 눌러 원하는 곳에
raw 파일로 저장 가능

1. 화소점 처리

- 원 화소의 값이나 위치를 바탕으로 단일 화소값을 변경하는 기술
- 종류 :
  - 동일
  - 밝게/어둡게
  - 반전
  - 흑백
  - 감마
  - 파라볼라(CAP/CUP)



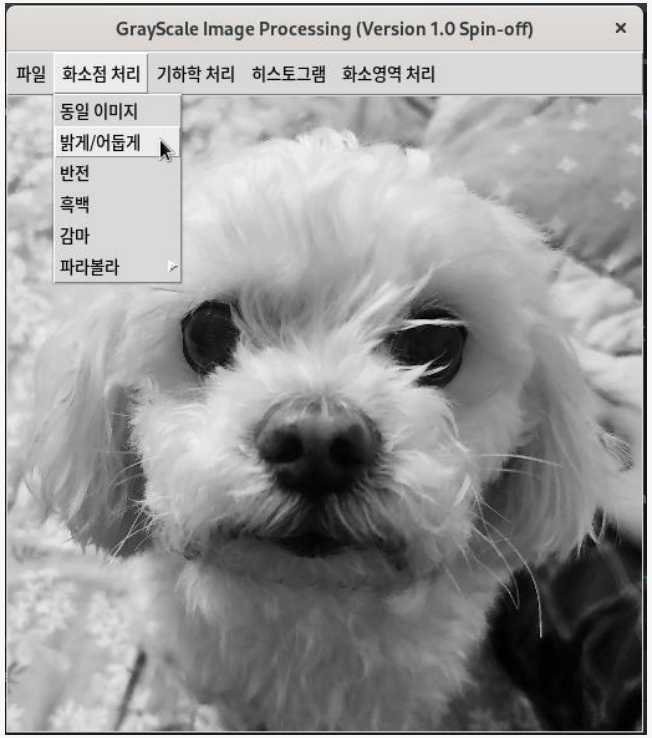
원본 이미지

- 동일

출력 이미지

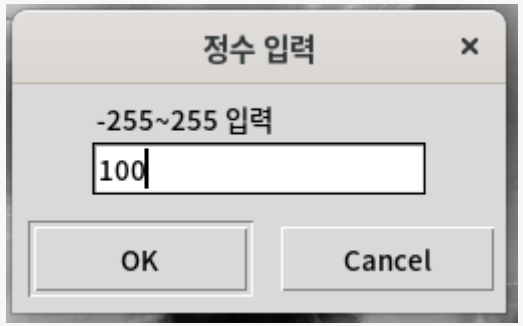
```
def equalImage():

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

1. 화소점 처리

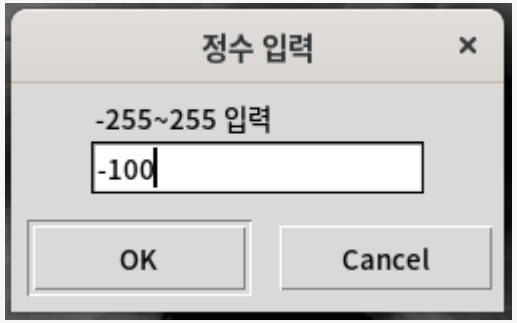


원본 이미지

- 밝게



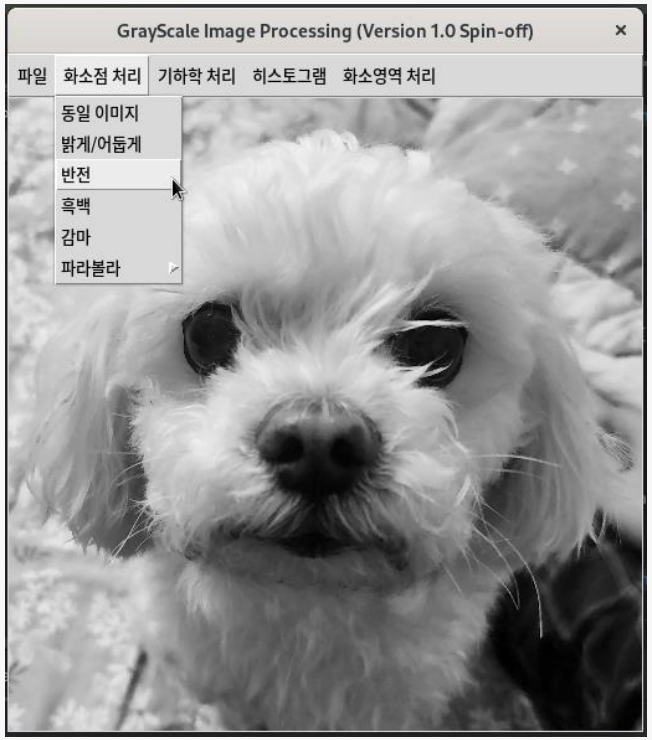
- 어둡게



출력 이미지

```
def addImage():

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

1. 화소점 처리



원본 이미지

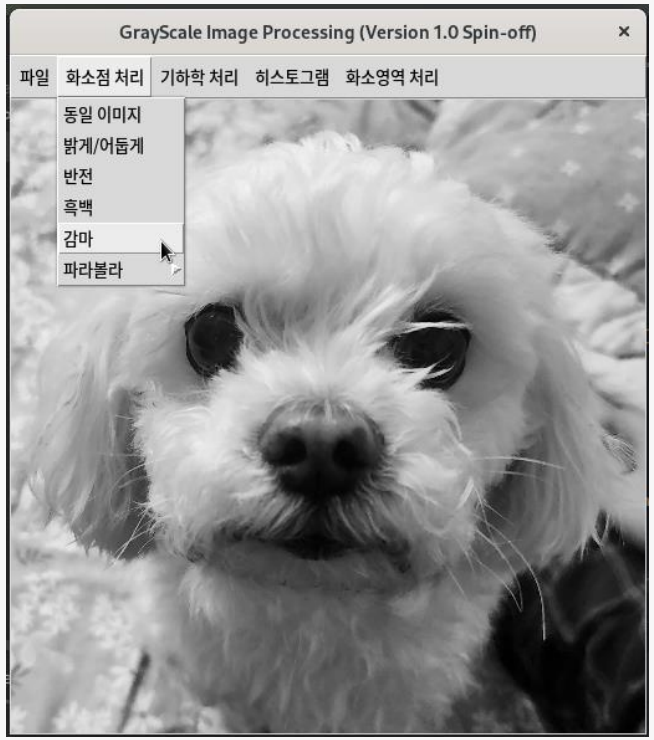
- 반전

```
def reverseImage():

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

- 흑백

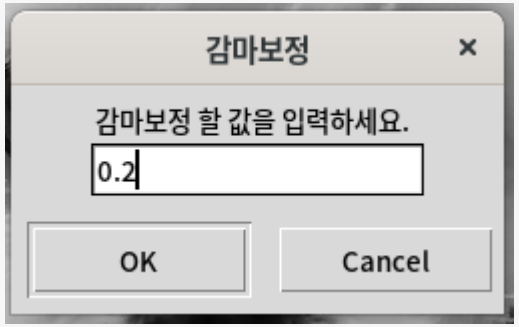
```
def bwImage():

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

1. 화소점 처리

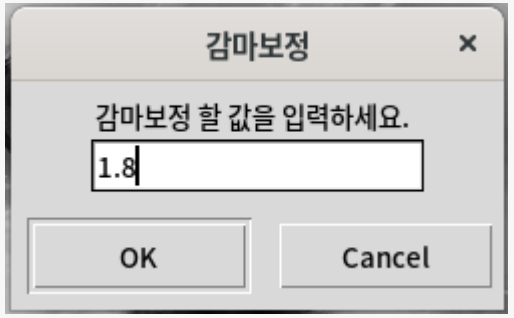


원본 이미지

- 감마 (0.2 입력)



- 감마 (1.8 입력)



출력 이미지

```
def gamma():

outImage[i][k] = int(255.0 * ((float(inImage[i][k]) / 255.0) ** gamma))
```

1. 화소점 처리

- 파라볼라 (CAP)

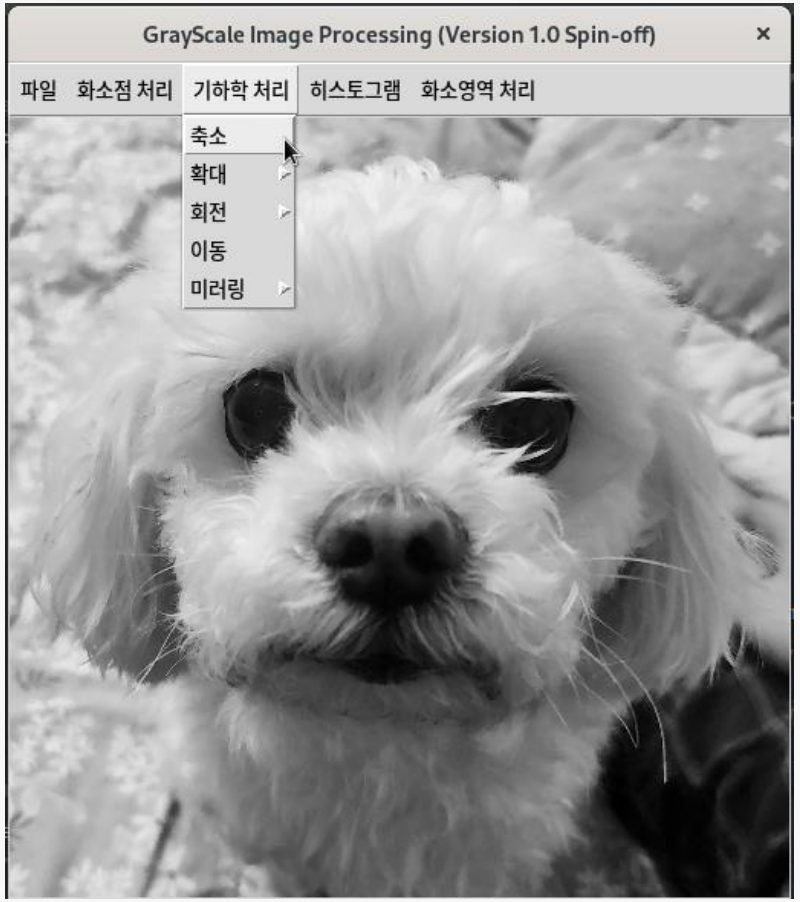

```
def para1Image():


outImage[i][k] = int(255. *
math.pow((inImage[i][k] / 127.) - 1, 2))
```

- 파라볼라 (CUP)


```
def para2Image():


outImage[i][k] =int(255. - 255 *
math.pow((inImage[i][k] / 127.) - 1, 2))
```

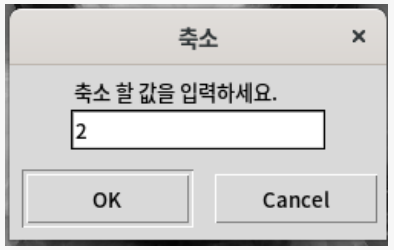
2. 기하학 처리

- 영상을 구성하는 화소의 공간적 위치나 배열을 재배치 하는 기술
- 종류 :
  - 축소
  - 확대
    (포워딩/ 백워딩)
  - 회전
    (기본/중앙)
  - 이동
  - 미러링
    (좌우,상하,상하좌우)



원본 이미지(512x512)

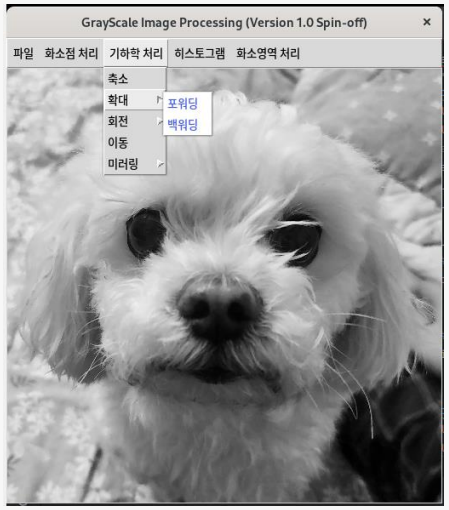
- 축소



출력 이미지(256x256)

```
def zoomOut():

outH = int(inH / value)
outW = int(inW / value)

outImage[int(i/value)][int(k/value)] =
inImage[i][k]
```

2. 기하학 처리



원본 이미지(512x512)

▪ 확대(포워딩)

홀 문제가 발생한 포워딩 출력 이미지(1024x1024)

```
def zoomin():

outImage[int(i*value)][int(k*value)] = inImage[i][k]
```

▪ 확대(백워딩)

```
def zoomin2():

outImage[i][k] = inImage[int(i/value)][int(k/value)]
```

보간 처리한 백워딩 출력 이미지(1024x1024)

2. 기하학 처리

- 회전

degree=45 입력

출력 이미지

```
def rotateImage():

radian = degree * math.pi / 180.0
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 < outH) & (0 <= yd < outW)):
outImage[xd][yd] = inImage[xs][ys]
```

- 회전(중앙, 백워딩)

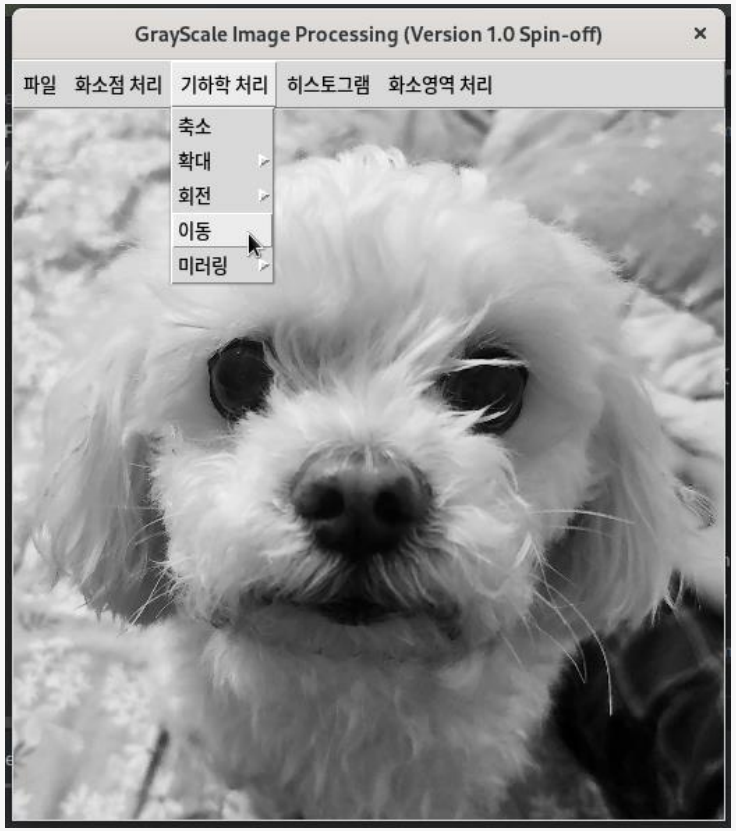
degree=30 입력

출력 이미지

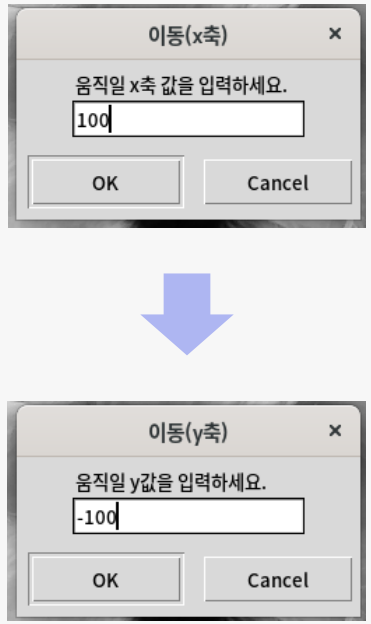
```
def rotateImage2():

radian = degree * math.pi / 180.0

cx = inH // 2 ; cy = inW // 2
xd = I ; yd = k

xs =int(math.cos(radian)*(xd-cx)+math.sin(radian)*(yd-cy))+cx
ys =int(-math.sin(radian)*(xd-cx)+math.cos(radian)*(yd-cy))+cy

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

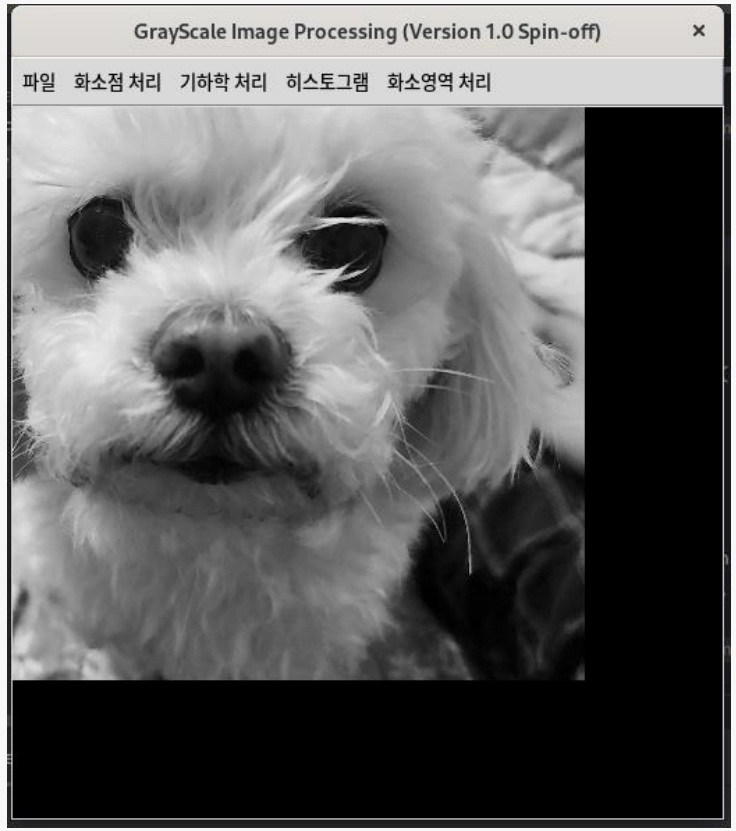
2. 기하학 처리

- 이동
  - 입력 받은 변수 만큼 이동



원본 이미지

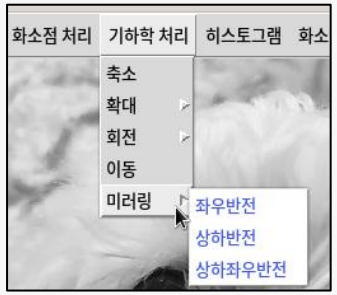




출력 이미지

```
def moveImage() :

outImage[i][k] = 0

if ((0 <= move_x and move_x < inH) and (0 <= move_y and move_y < inW)):
    outImage[move_x][move_y] = inImage[i][k]
```

2. 기하학 처리



- 좌우반전

```
def RightLeftMirror():

outImage[i][(inW-1)-k]=inImage[i][k]
```

- 상하반전

```
def UpDownMirror():

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

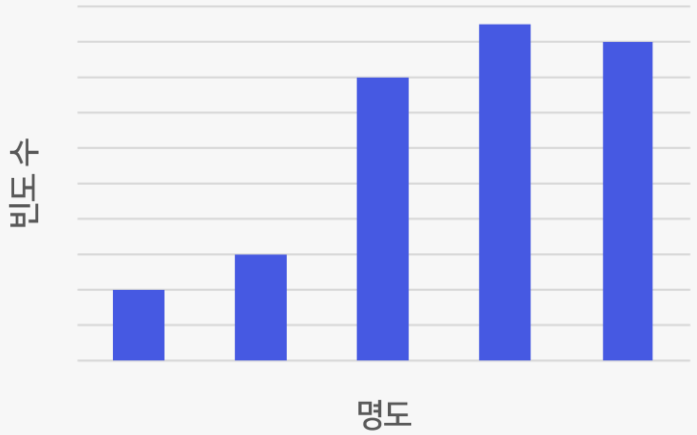
- 상하좌우반전

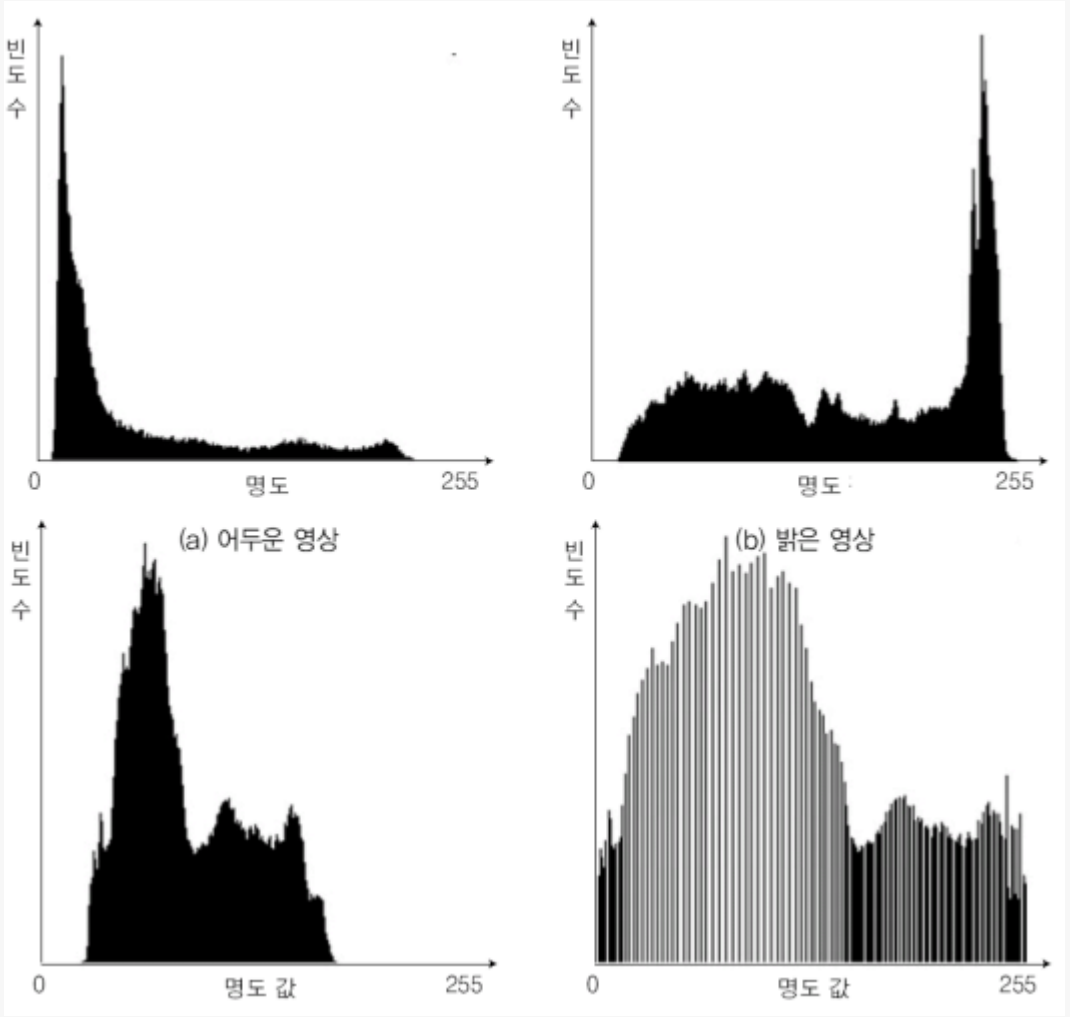
```
def UpDownRightLeftMirror():

outImage[(inH-1)-i][(inW-1)-k]=inImage[i][k]
```

3. 히스토그램이란?

- 표로 되어 있는 분포데이터를 막대 그래프로 나타낸 것
- 종류 :
  - 스트래칭
  - 엔드-인
  - 평활화

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

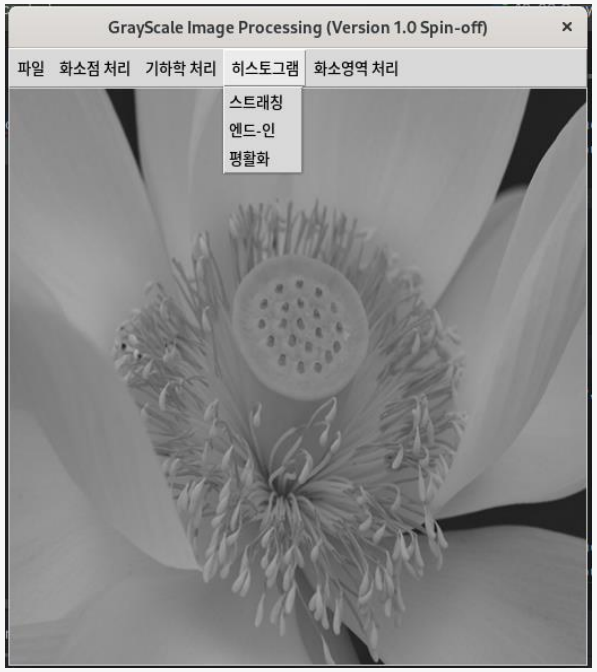




(C)명암 대비가 낮은 영상 (d)명암 대비가 높은 영상

3. 히스토그램 처리

➢ 원본 비교 사진

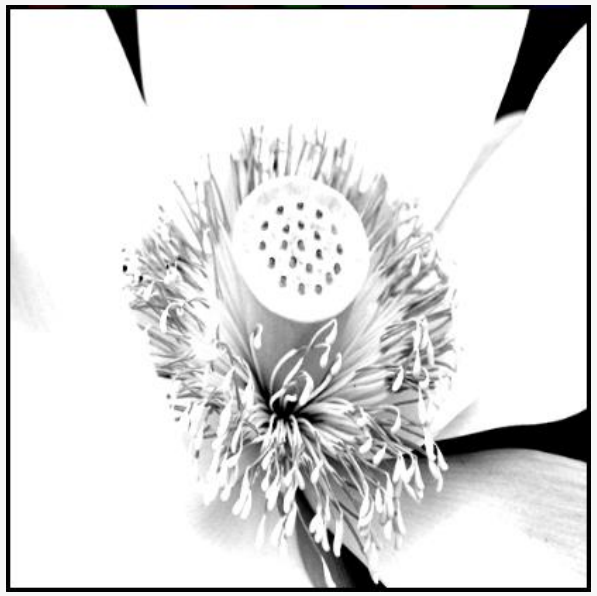


원본 이미지

▪ 스트래칭

출력 이미지

▪ 엔드-인



출력 이미지

▪ 평활화

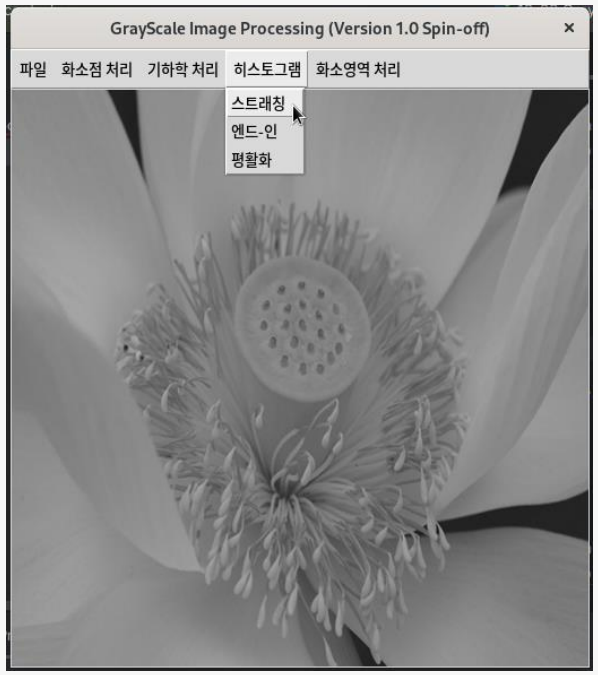
출력 이미지

3. 히스토그램 처리

- 명암 대비 스트래칭

- 히스토그램을 모든 영역으로 확장시켜 영상의 모든 범위의 화소값을 포함

$$outImage = \frac{inImage - low}{high - low} \times 255$$



원본 이미지

출력 이미지

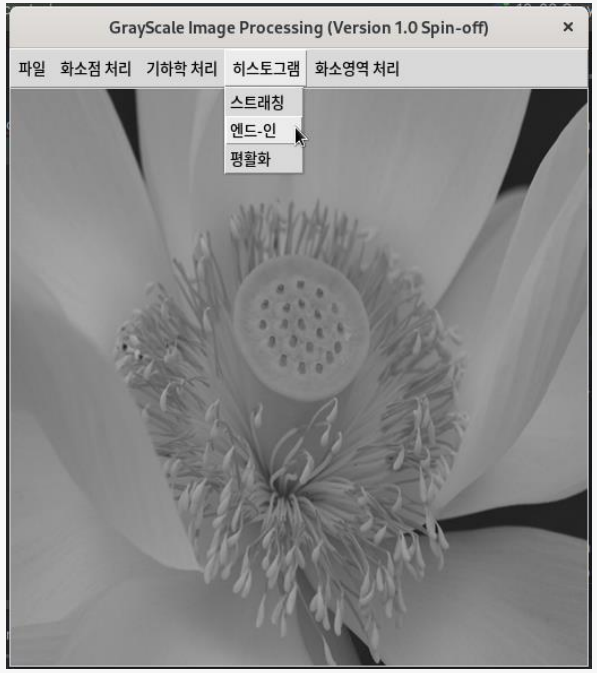
```
def histoStretch():


high = inImage[0][0]; low = inImage[0][0]

if (inImage[i][k] < low):
    low = inImage[i][k]
if (inImage[i][k] > high):
    high = inImage[i][k]

old , new = 0, 0
old=inImage[i][k]
new = (int)((old - low) / (high - low) * 255.0)
if (new > 255):
    new = 255
if (new < 0):
    new = 0
    outImage[i][k] = new
```

3. 히스토그램 처리

- 엔드-인 탐색

- 명암 대비 스트레칭에서 일정한 양의 화소를 흑백으로 지정하여 히스토그램의 분포를 더 균일하게 만드는 방법

$$outImage = \begin{cases} 0 & inImage \le low \\ \dfrac{inImage - low}{high - low} \times 255 & low \le inImage \le high \\ 255 & high \le inImage \end{cases}$$



원본 이미지

출력 이미지

```
def endIn():

high = inImage[0][0]; low = inImage[0][0]

if (inImage[i][k] < low):
    low = inImage[i][k]
if (inImage[i][k] > high):
    high = inImage[i][k]

high = high-50 ; low = low+50 ;        //스트레칭에서 이부분만 추가

old , new = 0, 0
old=inImage[i][k]
new = (int)((old - low) / (high - low) * 255.0)
if (new > 255):
    new = 255
if (new < 0):
    new = 0
    outImage[i][k] = new
```

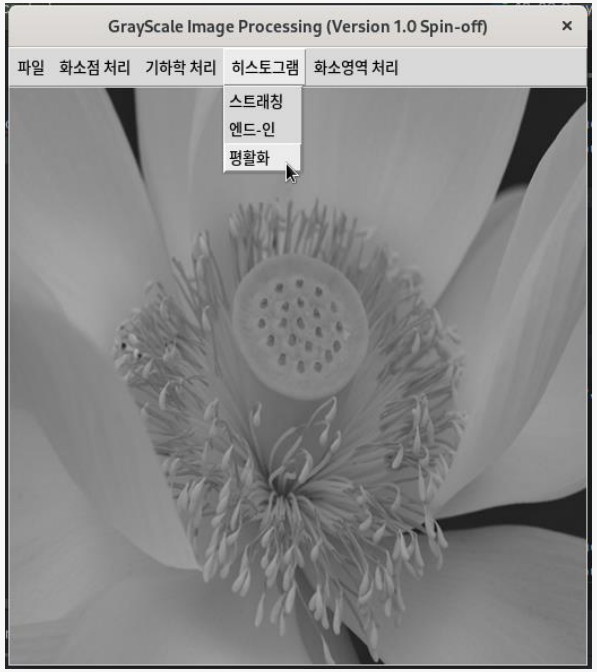
3. 히스토그램 처리

- 평활화

- 어둡게 촬영된 영상의 히스토그램을 조절하여 명암 분포가 빈약한 영상을 균일하게 만드는 방법

$$sum[i] = \sum_{j=0}^{i} hist[j]$$

$$n[i] = sum[i] \times \frac{i}{N(\text{총화소수})} \times I_{\max}(\text{최대 명도값})$$



원본 이미지

출력 이미지

```
## 1단계 : 히스토그램 생성
histo = []
histo = [0 for _ in range(256)]
for i in range(inH):
    for k in range(inW):
        histo[inImage[i][k]] += 1
## 2단계 : 누적 히스토그램 생성
sumhisto = []
sumhisto = [0 for _ in range(256)]

for i in range(256):
    sumhisto[i] = sumhisto[i - 1] + histo[i]

## 3단계 : 정규화된 누적 히스토그램
nomalhisto = [0 for _ in range(256)]
for i in range(256):
    nomalhisto[i]=int(sumhisto[i]*(1.0 / (inH * inW)) * 255.0)

## 4단계 : 원래값을 정규화 값으로 치환
for i in range(inH):
    for k in range(inW):
        outImage[i][k] = nomalhisto[inImage[i][k]]
```

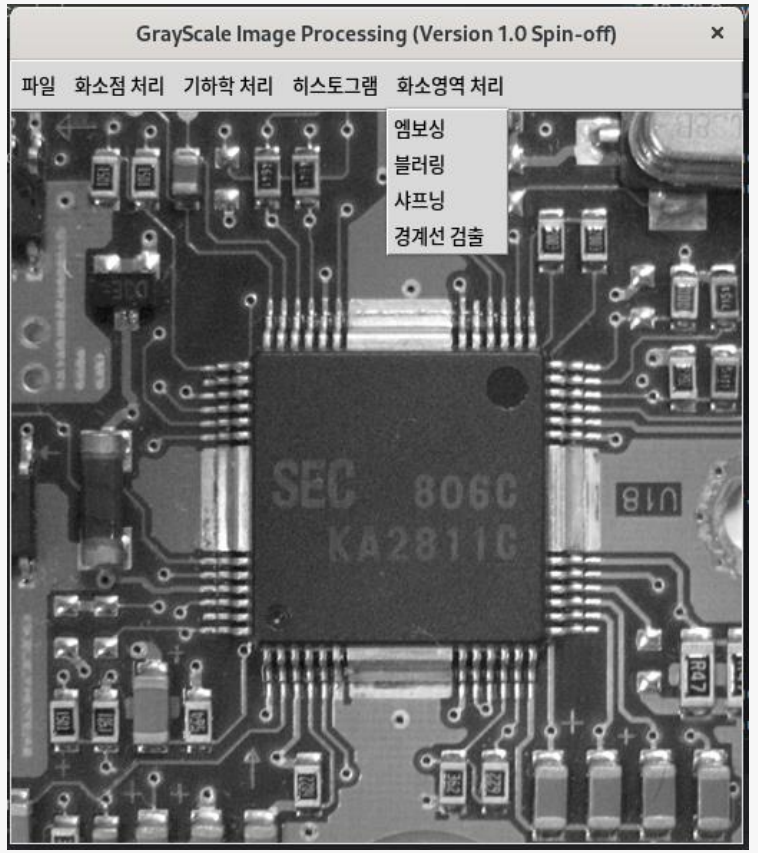
4. 화소영역 처리란?

- 화소점 처리(입력 화소) + 주변 화소 값도 고려하는 공간 영역 연산
- 회선 처리 또는 컨벌루션 처리(Convolution Processing)라고 함

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

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

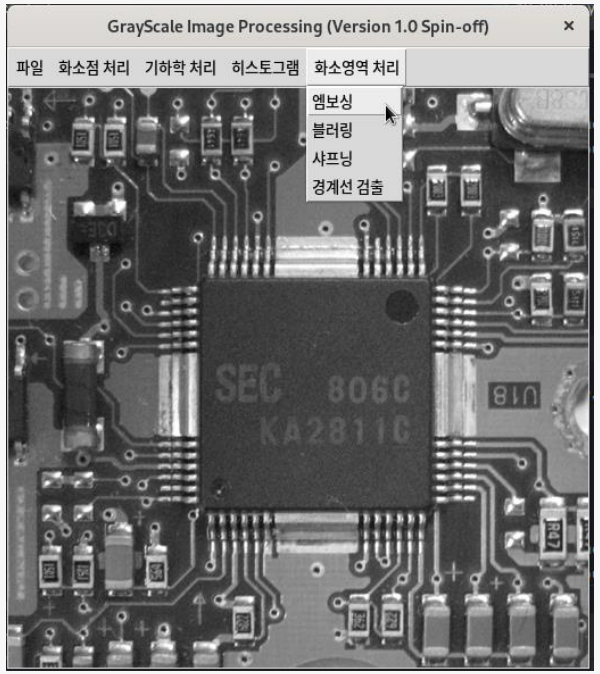
- 원시 화소와 이웃한 각 화소에 가중치를 곱한 합을 출력 화소로 생성
- 종류:
  - 엠보싱
  - 블러링
  - 샤프닝
  - 경계선 검출



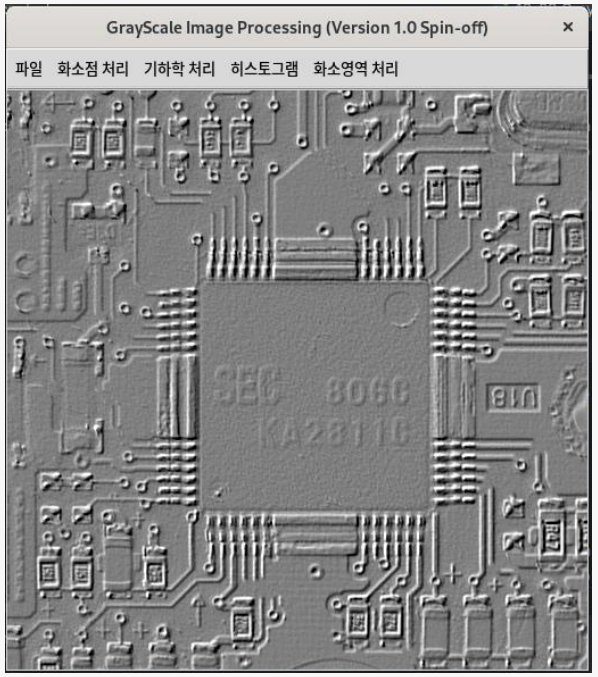
4. 화소영역 처리

- 엠보싱 (Embossing) 효과

- 입력 영상을 양각 형태로 보이게 하는 기술



원본 이미지



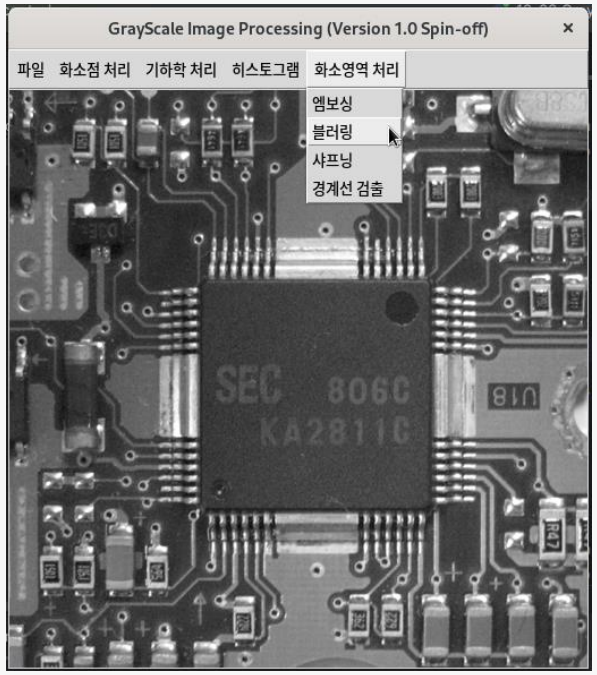
출력 이미지

```
def Emboss():

MSIZE = 3
mask = [[-1, 0, 0],
        [ 0, 0, 0],
        [ 0, 0, 1] ] #엠보싱 마스크

#임시 메모리 할당
tmpInImage = malloc2D(inH + 2, inW + 2)
tmpOutImage = malloc2D(outH, outW)

# 입력 영상 --> 임시 입력 영상
tmpInImage[i + 1][k + 1] = inImage[i][k]

# 회선 연산
for i in range(inH):
    for k in range(inW):
        SR = 0
        for m in range(MSIZE):
            for n in range(MSIZE):
                SR += int(tmpInImage[i+m][k+n] * mask[m][n])
          tmpOutImage[i][k] = SR

# 임시 출력 --> 출력 메모리
outImage[i][k] = tmpOutImage[i][k]
```

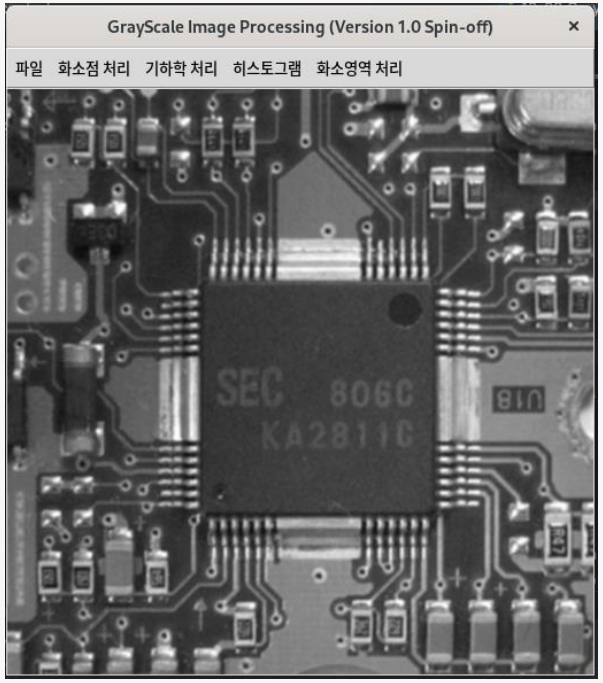
4. 화소영역 처리

- 블러링 (blurring) 효과

- 영상의 세밀한 부분을 제거하여 흐리게 하거나 부드럽게 하는 기술



원본 이미지



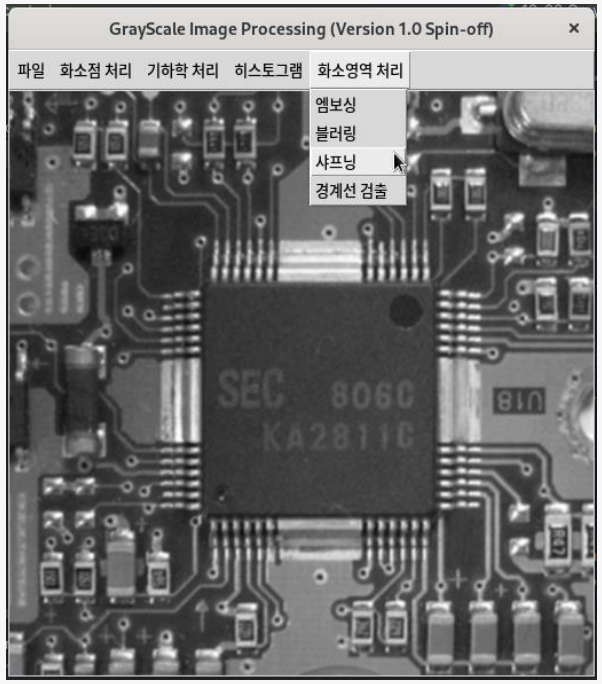
출력 이미지

```
def Blur():

MSIZE = 3
mask = [[1 / 9, 1 / 9, 1 / 9],
        [1 / 9, 1 / 9, 1 / 9],
        [1 / 9, 1 / 9, 1 / 9] ]  #블러링 마스크 //마스크만 변경


#임시 메모리 할당
tmpInImage = malloc2D(inH + 2, inW + 2)
tmpOutImage = malloc2D(outH, outW)

# 입력 영상 --> 임시 입력 영상
tmpInImage[i + 1][k + 1] = inImage[i][k]

# 회선 연산
for i in range(inH):
    for k in range(inW):
        SR = 0
        for m in range(MSIZE):
            for n in range(MSIZE):
                SR += int(tmpInImage[i+m][k+n] * mask[m][n])
          tmpOutImage[i][k] = SR

# 임시 출력 --> 출력 메모리
outImage[i][k] = tmpOutImage[i][k]
```

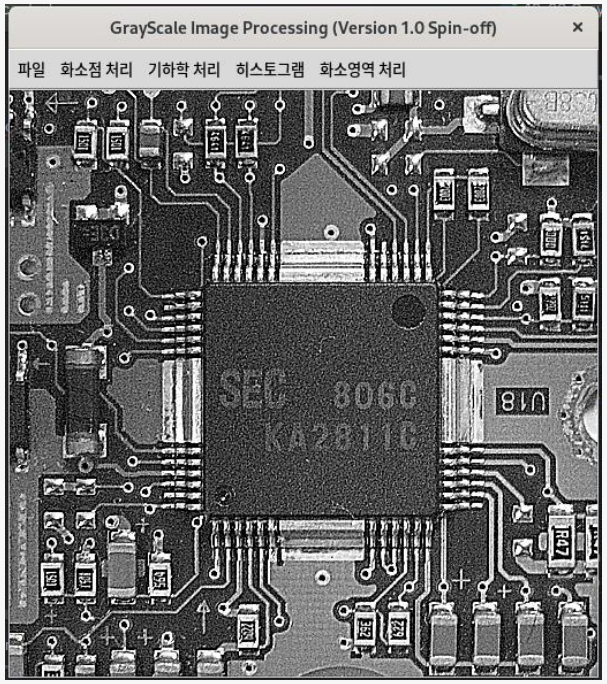
4. 화소영역 처리

- 샤프닝 (Sharpening) 효과

- 블러링과는 반대로 디지털 영상에서 상세한 부분을 더욱 강조하여 표현하는 기술



원본 이미지



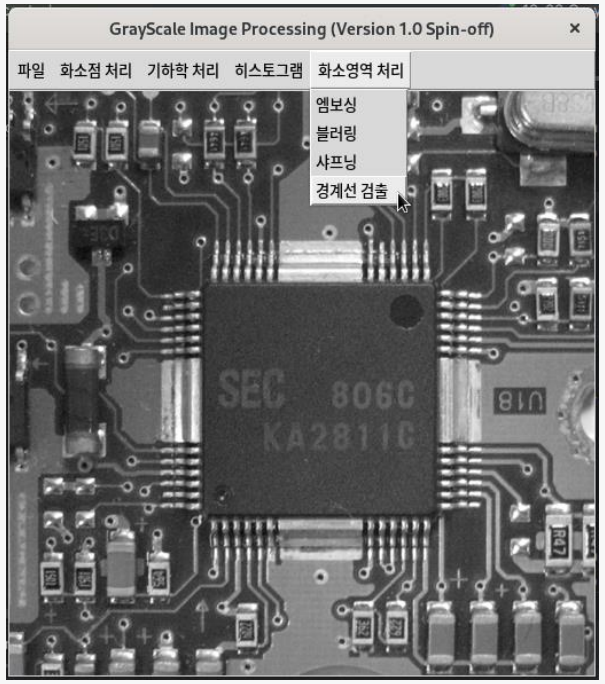
출력 이미지

```
def Sharp():

MSIZE = 3
mask = [[-1, -1, -1],
        [-1,  9, -1],
        [-1, -1, -1] ] # 샤프닝 마스크          //마스크만 변경


#임시 메모리 할당
tmpInImage = malloc2D(inH + 2, inW + 2)
tmpOutImage = malloc2D(outH, outW)

# 입력 영상 --> 임시 입력 영상
tmpInImage[i + 1][k + 1] = inImage[i][k]

# 회선 연산
for i in range(inH):
    for k in range(inW):
        SR = 0
        for m in range(MSIZE):
            for n in range(MSIZE):
                SR += int(tmpInImage[i+m][k+n] * mask[m][n])
          tmpOutImage[i][k] = SR

# 임시 출력 --> 출력 메모리
outImage[i][k] = tmpOutImage[i][k]
```

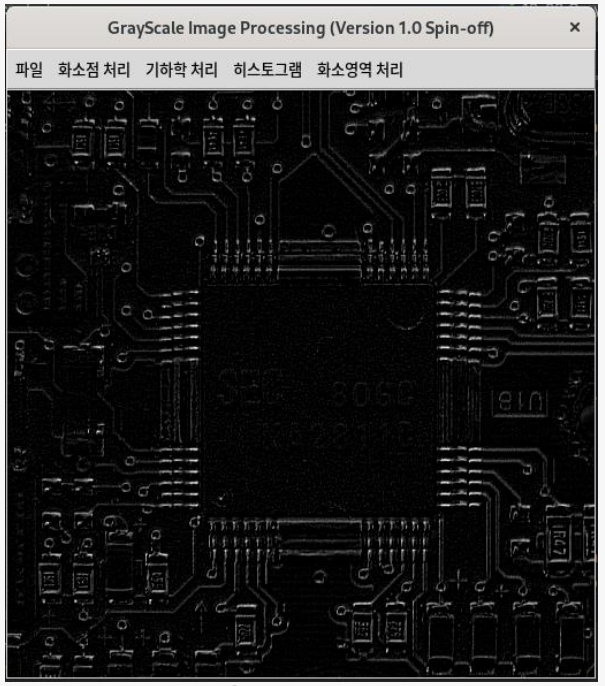
4. 화소영역 처리

- 경계선 검출

- 디지털 영상의 밝기가 낮은 값에서 높은 값으로,
또는 반대로 변하는 지점을 검출하는 과정



원본 이미지



출력 이미지

```
def Edge():

MSIZE = 3
mask = [[-1, 0,-1],
        [ 0, 2, 0],
        [ 0, 0, 0]] # 엣지 마스크          //마스크만 변경


#임시 메모리 할당
tmpInImage = malloc2D(inH + 2, inW + 2)
tmpOutImage = malloc2D(outH, outW)

# 입력 영상 --> 임시 입력 영상
tmpInImage[i + 1][k + 1] = inImage[i][k]

# 회선 연산
for i in range(inH):
    for k in range(inW):
        SR = 0
        for m in range(MSIZE):
            for n in range(MSIZE):
                SR += int(tmpInImage[i+m][k+n] * mask[m][n])
          tmpOutImage[i][k] = SR

# 임시 출력 --> 출력 메모리
outImage[i][k] = tmpOutImage[i][k]
```

향후 발전 방향

- 더욱 다양한 영상처리 알고리즘 생성
- 입력값 오류 처리
- 효과 누적 및 취소 기능 추가
- 정방향이 아닌 다양한 크기의 영상 입력이 가능하게 조정