[Intel] 엣지 AI SW 아카데미 _ 절차 지향 프로그래밍

Python으로 구현한 GrayScale Image Processing

임지원

프로젝트 개요

목표

OpenCV 없이 파이썬을 기반으로 영상 처리 프로그램을 구현

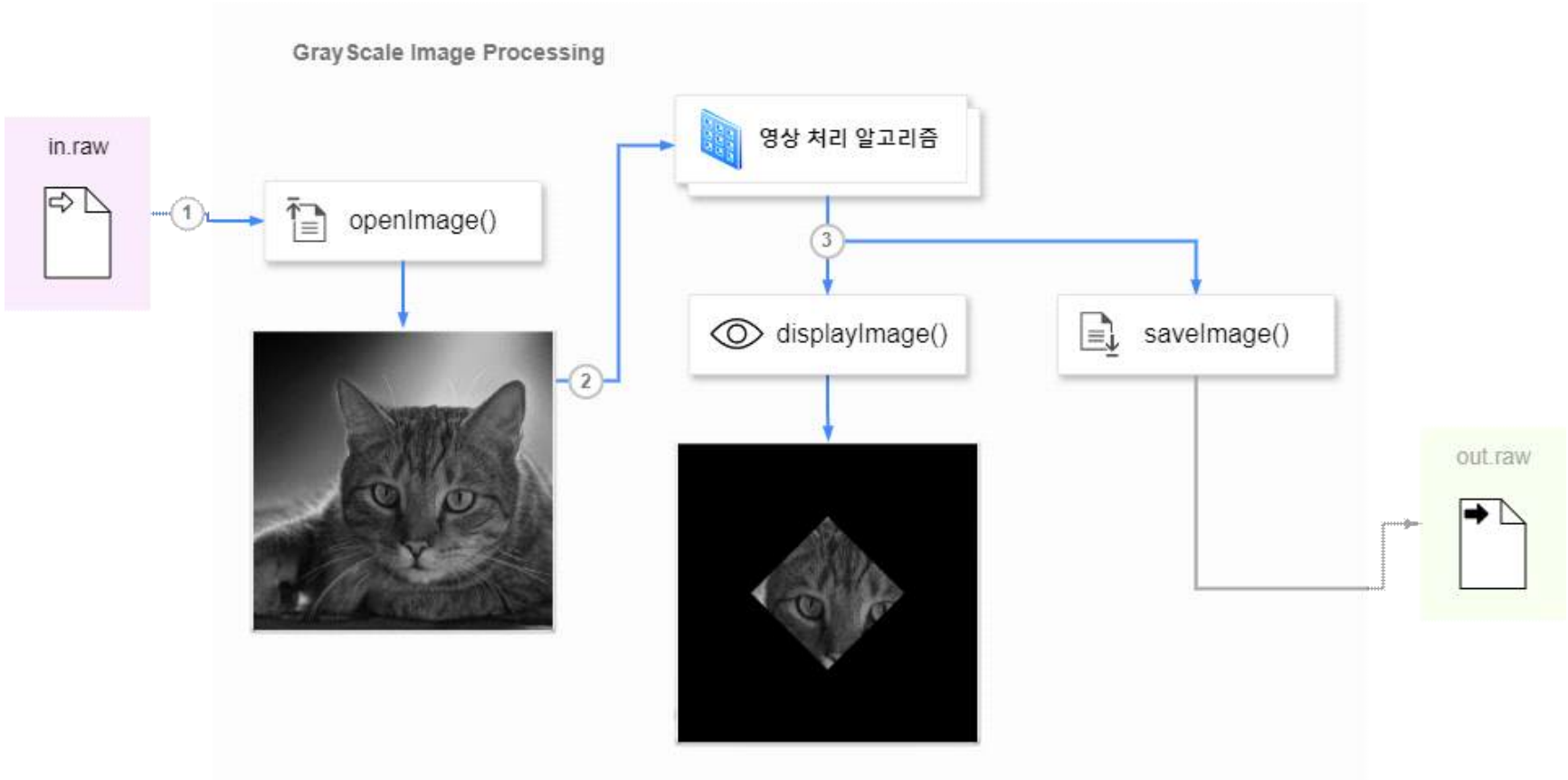
개발 환경

OS : Windows 11 Pro
Tool : Pycharm 2023.3.4 (Community Edition)

코드

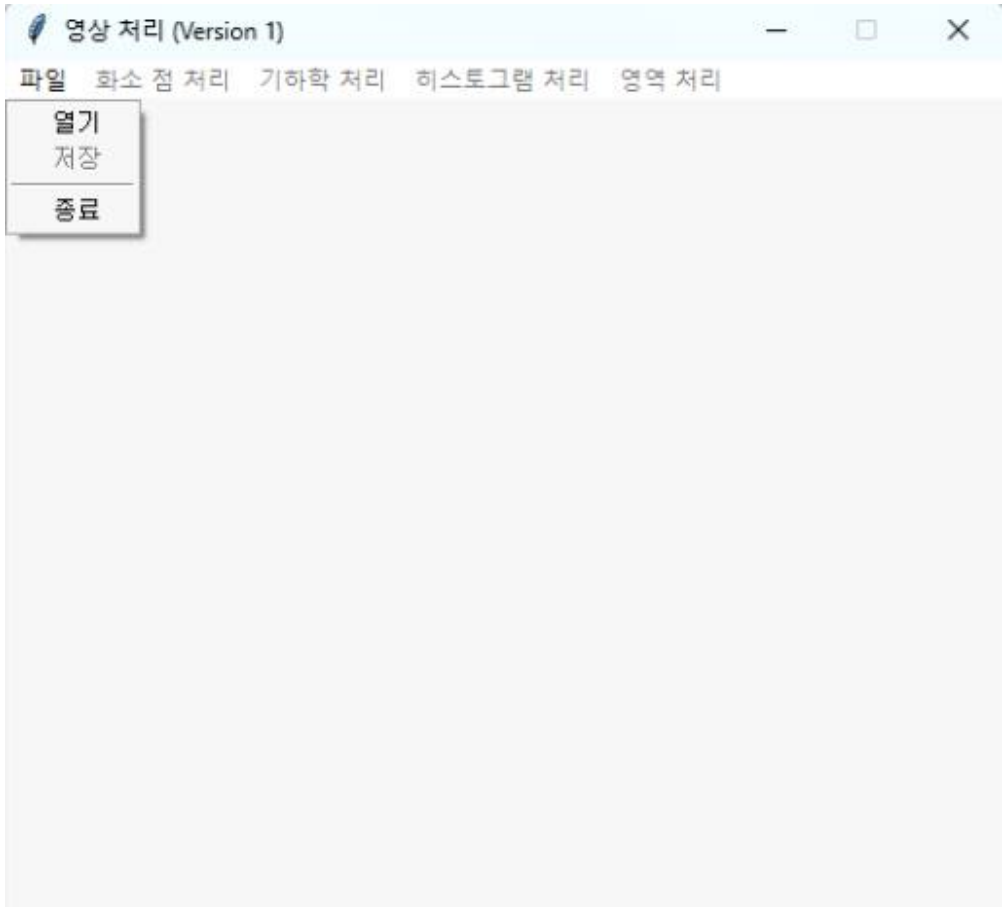
https://github.com/Jday4612/GrayScale-Image-Processing

프로젝트 구조도

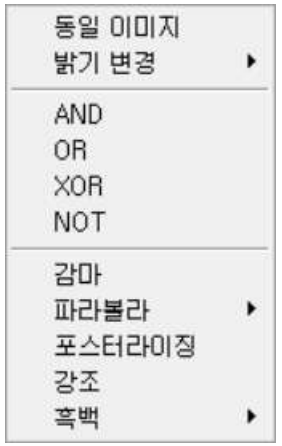

Gray Scale Image Processing
in.raw
1
openImage()
영상 처리 알고리즘
3
displayImage()
saveImage()
2
out.raw

메인 화면

*대화 상자에서 [취소] 시, 발생하는 에러 예외 처리

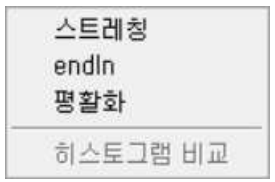


화소 점 처리

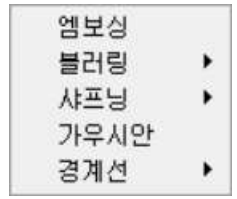


기하학 처리

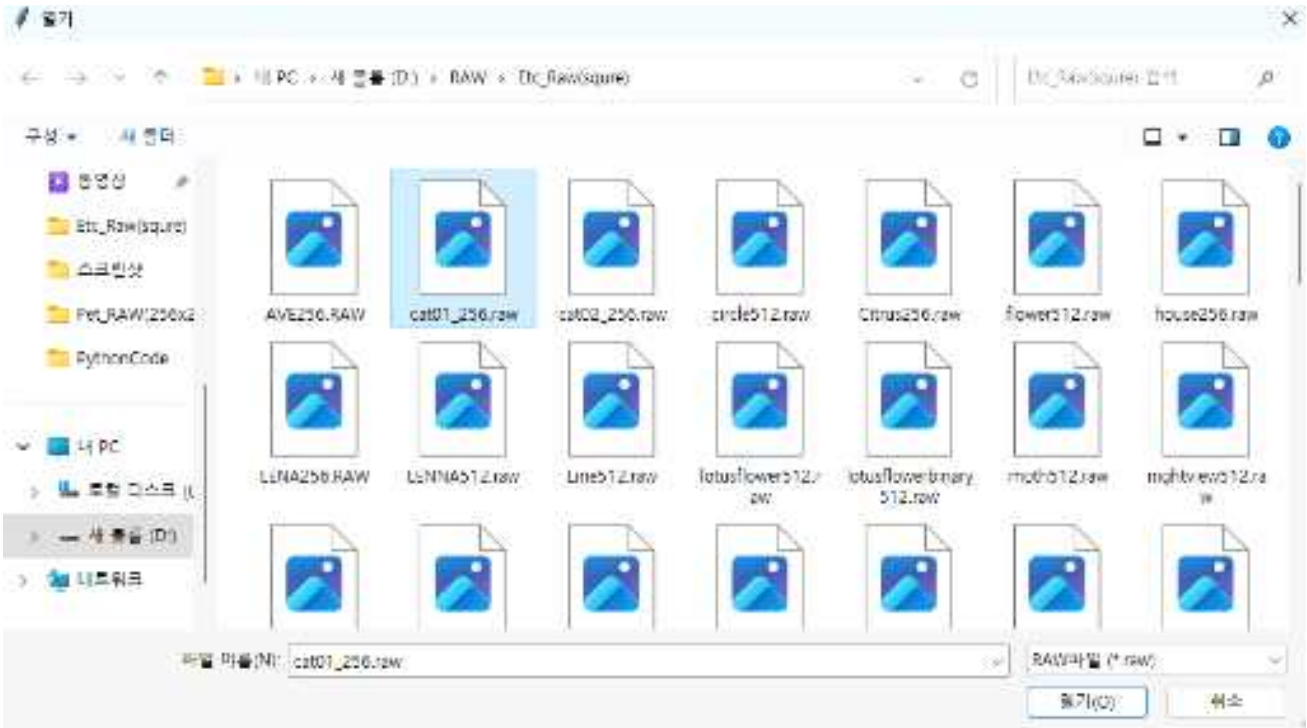
화소 점 처리



영역 처리

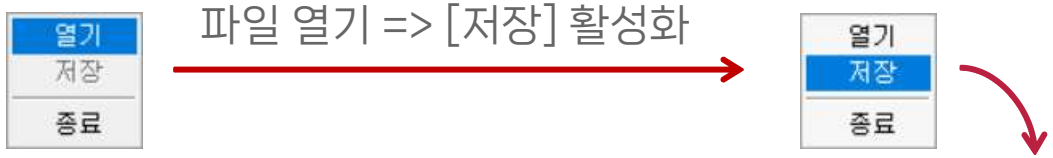


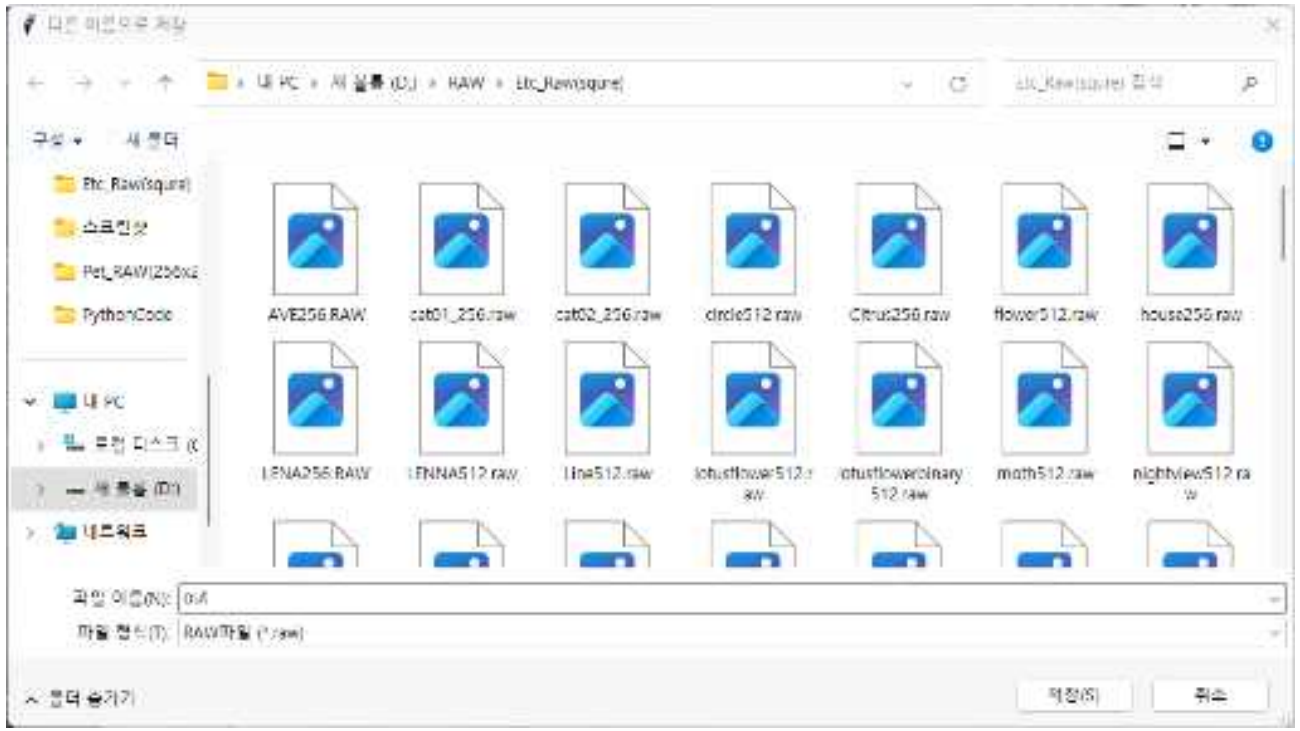
파일 - 열기



선택된 raw 파일 출력

저장





원하는 이름으로 파일 저장

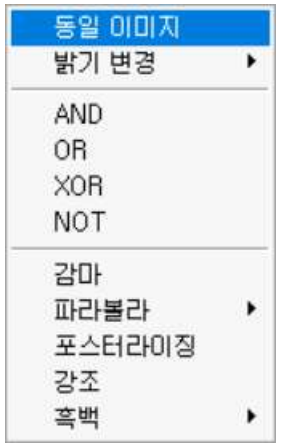
out.raw

화소 점 처리 - 동일

: 원 화소의 값이나 위치를 바탕으로 단일 화소 값을 변경하는 기술

입력 값을 그대로 출력 값으로 변환

```
for i in range(inH):
    for j in range(inW):
        outImage[i][j] = inImage[i][j]
```



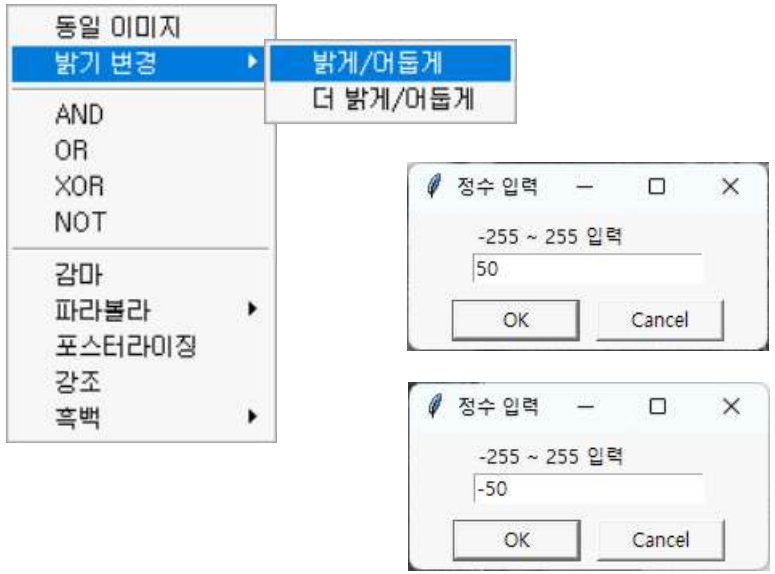
Input

Output

화소 점 처리 - 밝게/어둡게

정수를 입력 시, 해당 값을 더해 밝기 변경

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

if px < 0:
    px = 0
elif 255 < px:
    px = 255

outImage[i][j] = px
```



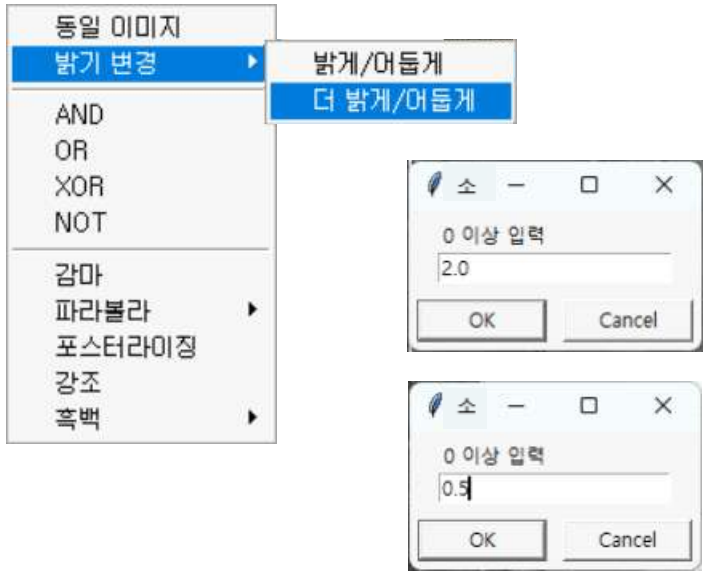
Output1

Output2

화소 점 처리 - 더 밝게/어둡게

소수를 입력 시, 해당 값을 곱해 밝기 변경

```
px = int(inImage[i][j] * value)

if px < 0:
    px = 0
elif 255 < px:
    px = 255

outImage[i][j] = px
```

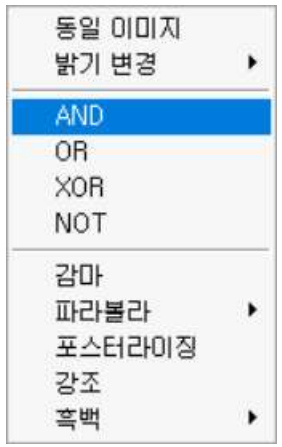


Output1

Output2

화소 점 처리 - AND

특정 데이터와 AND 연산 수행



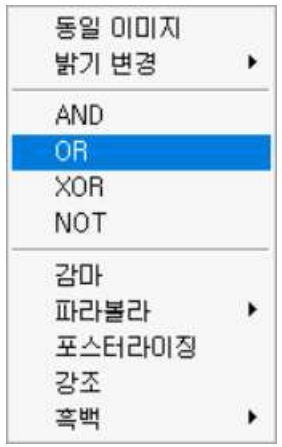

```
px = inImage[i][j] & dataImage[i][j]

if px < 0:
    px = 0
elif 255 < px:
    px = 255

outImage[i][j] = px
```

Input

Output

화소 점 처리 - OR

특정 데이터와 OR 연산 수행



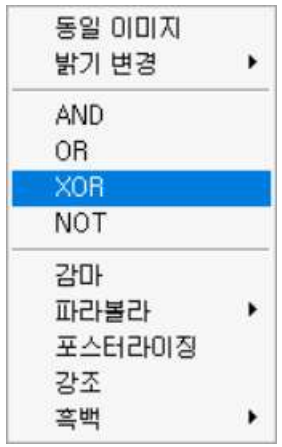

```
px = inImage[i][j] | dataImage[i][j]

if px < 0:
    px = 0
elif 255 < px:
    px = 255

outImage[i][j] = px
```

Input

Output

화소 점 처리 - XOR

특정 데이터와 OR 연산 수행



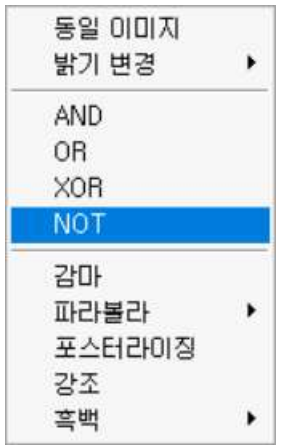

```
px = inImage[i][j] ^ dataImage[i][j]

if px < 0:
    px = 0
elif 255 < px:
    px = 255

outImage[i][j] = px
```

Input

Output

화소 점 처리 - NOT

화소 비트가 반전됨



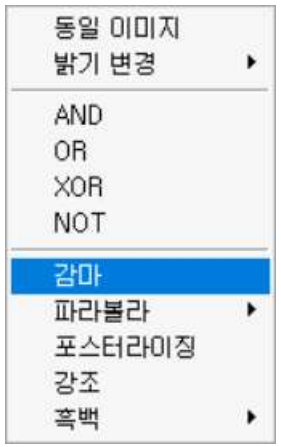

```
outImage[i][j] = 255 - inImage[i][j]
```

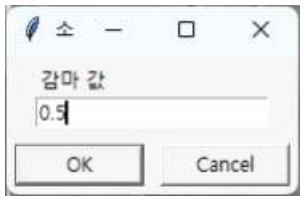
Input

Output

화소 점 처리 - 감마

소수를 입력 시, 해당 값에 따라 밝기 조절





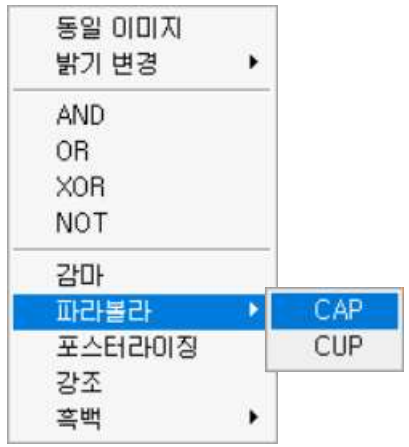

```
x = inImage[i][j]
outImage[i][j] = int(255.0 * math.pow(x / 255.0, gamma))
```

Input

Output

화소 점 처리 - 파라볼라(CAP)

파라볼라 CAP 수식을 통해 변환



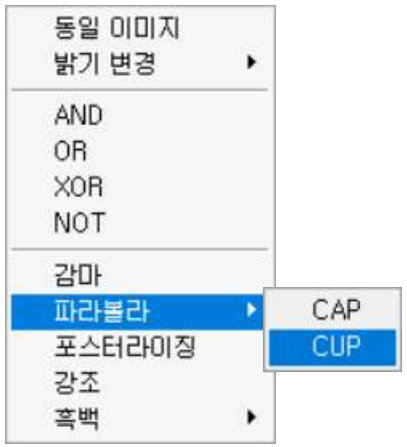

```
x = inImage[i][j]
outImage[i][j] = int(-255.0 * math.pow(x / 127.0 - 1.0, 2) + 255.0)
```

Input

Output

화소 점 처리 - 파라볼라(CUP)

파라볼라 CUP 수식을 통해 변환




```
x = inImage[i][j]
outImage[i][j] = int(255.0 * pow(x / 127.0 - 1.0, 2))
```

Input

Output

화소 점 처리 - 포스터라이징

정수를 입력 시,
해당 값으로 명암 값 수를 변경

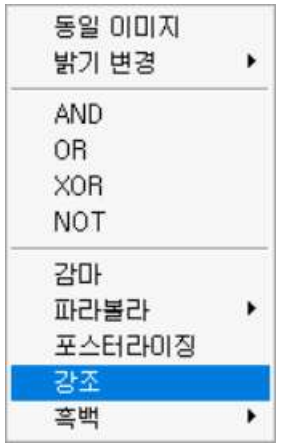

```
for k in range(1, value, 1):
    if k is (value - 1):
        outImage[i][j] = 255
    elif inImage[i][j] <= 255 // (value - 1) * k:
        if k == 1:
            outImage[i][j] = 0
        else:
            outImage[i][j] = 255 // (value - 2) * (k - 1)

        break
```

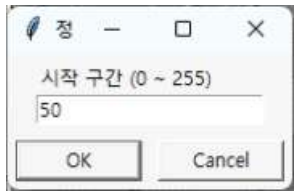
Input

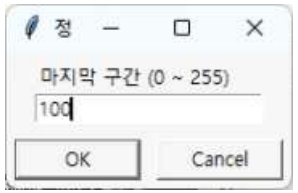
Output

화소 점 처리 - 강조

**정수 2개를 입력 시,
해당 범위 내의 화소만 강조**







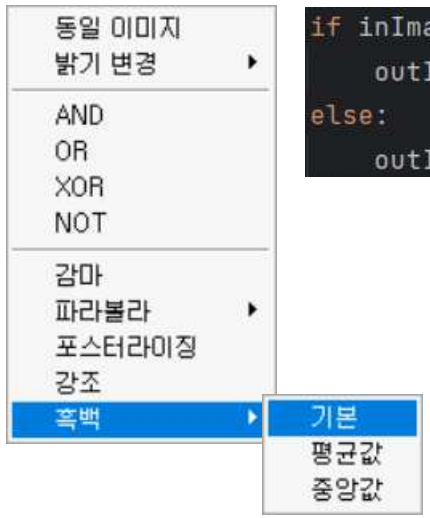

```
if value1 <= inImage[i][j] and inImage[i][j] <= value2:
    outImage[i][j] = 255
else:
    outImage[i][j] = inImage[i][j]
```

Input

Output

화소 점 처리 - 흑백

128을 기준으로 이진화 적용



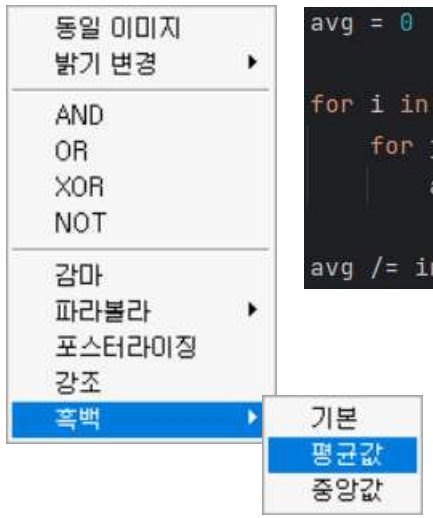

```
if inImage[i][j] < 128:
    outImage[i][j] = 0
else:
    outImage[i][j] = 255
```

Input

Output

화소 점 처리 - 흑백(평균값)

평균값을 기준으로 이진화 적용




```
avg = 0

for i in range(inH):
    for j in range(inW):
        avg += inImage[i][j]

avg /= inH * inW
```

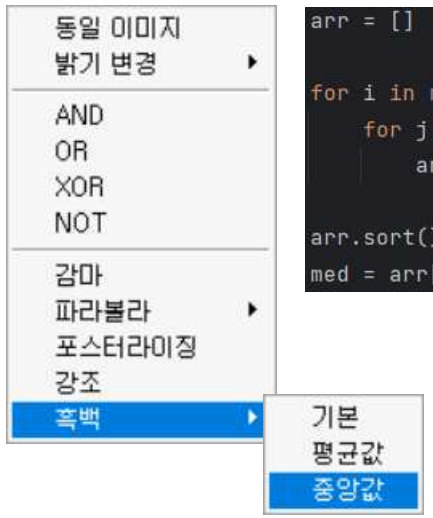
```
if inImage[i][j] < avg:
    outImage[i][j] = 0
else:
    outImage[i][j] = 255
```

Input

Output

화소 점 처리 - 흑백(중앙값)

중앙값을 기준으로 이진화 적용




```
arr = []

for i in range(inH):
    for j in range(inW):
        arr.append(inImage[i][j])

arr.sort()
med = arr[inH * inW // 2]
```

```
if inImage[i][j] < med:
    outImage[i][j] = 0
else:
    outImage[i][j] = 255
```

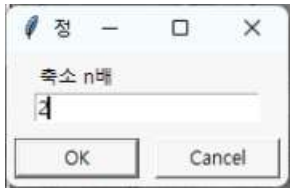
Input

Output

기하학 처리 - 축소

: 영상을 구성하는 화소의 공간적 위치를 재배치하는 기술

정수를 입력 시,
포워딩으로 (해당 값)배 축소



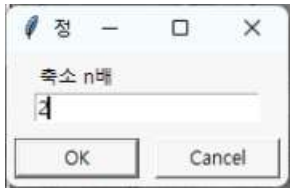

```
outImage[i // scale][j // scale] = inImage[i][j]
```

Input

Output

기하학 처리 - 축소(평균값)

정수를 입력 시,
평균값을 기준으로 (해당 값)배 축소



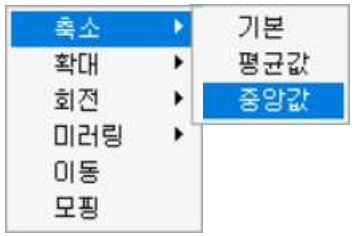

```
avg = 0

for n in range(0, scale, 1):
    for m in range(0, scale, 1):
        avg += inImage[i + n][j + m]

outImage[i // scale][j // scale] = avg // (scale * scale)
```

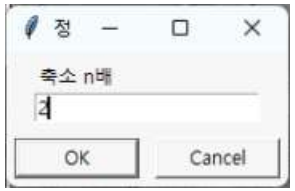
Input

Output

기하학 처리 - 축소(중앙값)

정수를 입력 시,
중앙값을 기준으로 (해당 값)배 축소





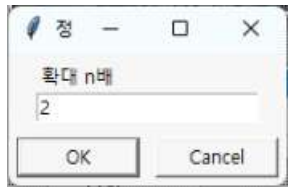

```
arr = []

for n in range(0, scale, 1):
    for m in range(0, scale, 1):
        arr.append(inImage[i + n][j + m])

arr.sort()

outImage[i // scale][j // scale] = arr[scale * scale // 2]
```

Input

Output

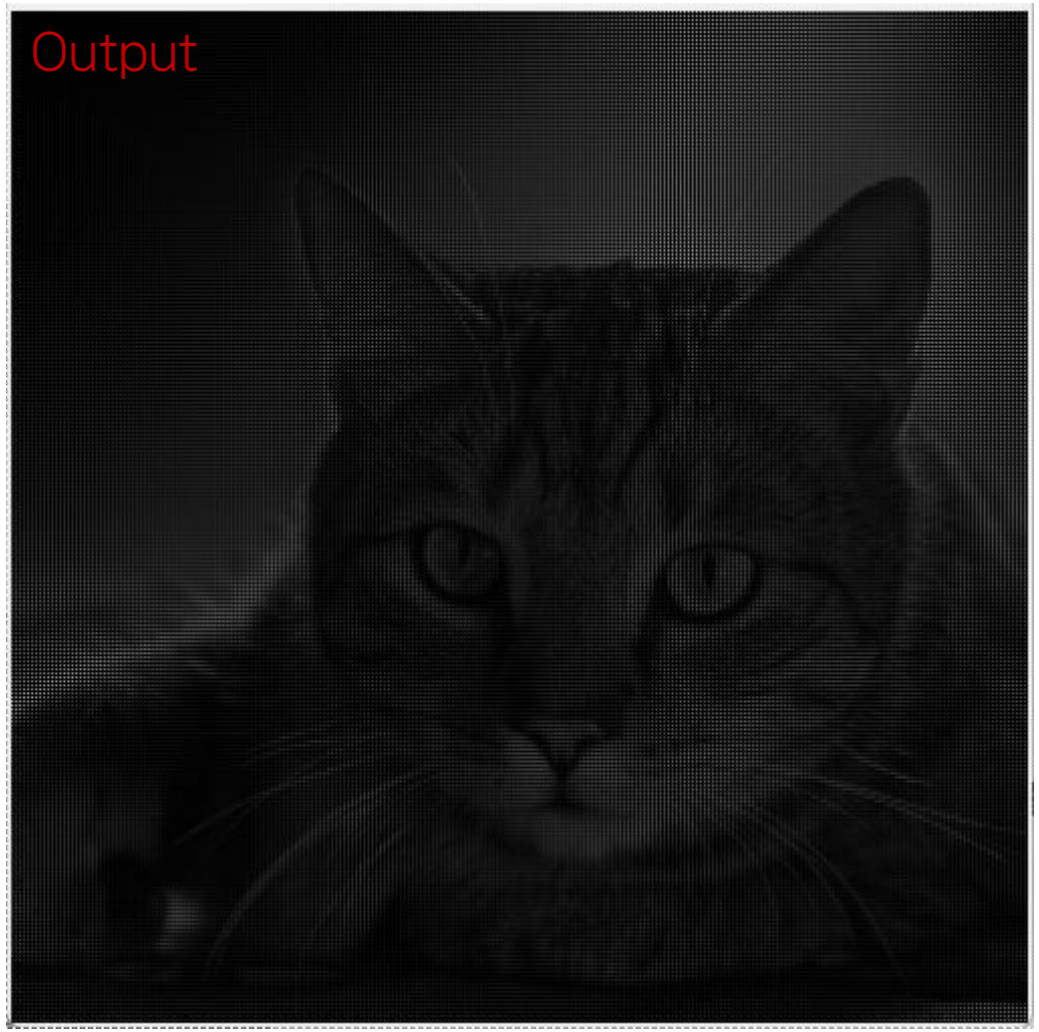
기하학 처리 - 확대(포워딩)

정수를 입력 시,
포워딩으로 (해당 값)배 확대



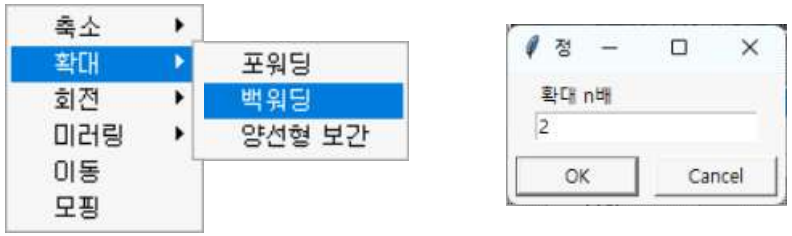

```
outImage[i * scale][j * scale] = inImage[i][j]
```

Input



기하학 처리 - 확대(백워딩)

정수를 입력 시,
백워딩으로 (해당 값)배 확대

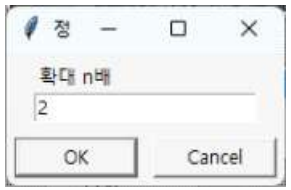



```
outImage[i][j] = inImage[i // scale][j // scale]
```

Input

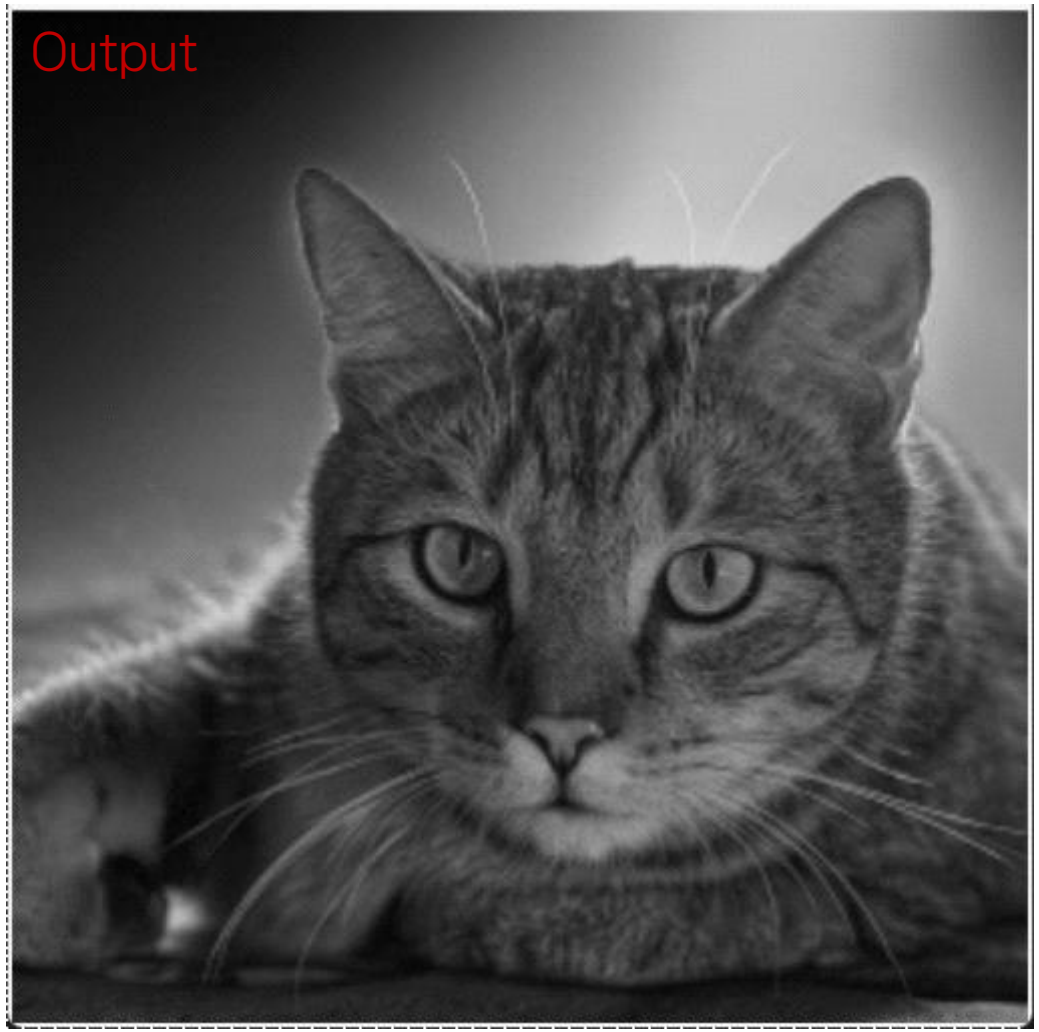
기하학 처리 - 확대(양선형 보간)

정수를 입력 시,
양선형 보간으로 (해당 값)배 확대



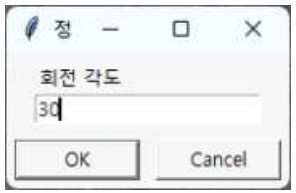

```
rH = i / scale
rW = j / scale

iH = int(math.floor(rH))
iW = int(math.floor(rW))

sH = rH - iH
sW = rW - iW

if iH < 0 or inH - 1 <= iH or iW < 0 or inW - 1 <= iW:
    outImage[i][j] = 255
else:
    C1 = float(inImage[iH][iW])
    C2 = float(inImage[iH][iW + 1])
    C3 = float(inImage[iH + 1][iW + 1])
    C4 = float(inImage[iH + 1][iW])

    outImage[i][j] = int(C1 * (1 - sH) * (1 - sW) + C2 * sW
                         * (1 - sH) + C3 * sW * sH + C4 * (1 - sW) * sH)
```

Input



기하학 처리 - 회전

정수를 입력 시, 해당 값만큼 회전

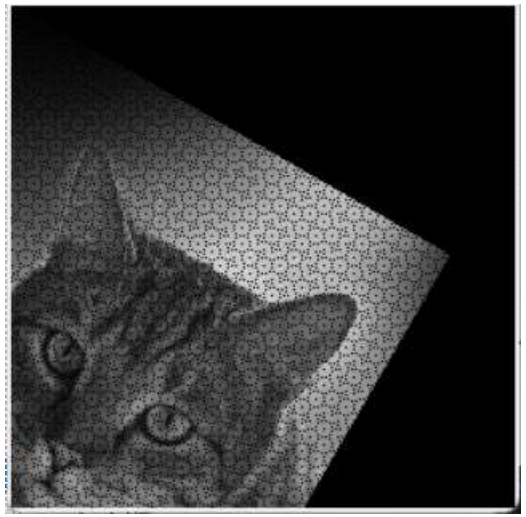



```
radian = -degree * 3.141592 / 180.0
```

```
xs = i
ys = j

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]
```

Input

Output



기하학 처리 - 회전(중앙&백워딩)

정수를 입력 시,
중앙 & 백워딩으로 해당 값만큼 회전


```
radian = -degree * 3.141592 / 180.0
```

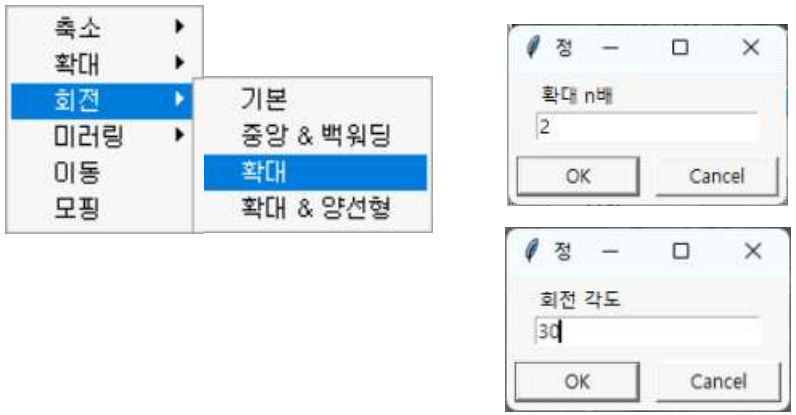
```
xd = i
yd = j

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

xs += cx
ys += cy

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

Input

Output

기하학 처리 - 회전(확대)

**정수 두 개를 입력 시,
해당 값만큼 회전 및 확대**

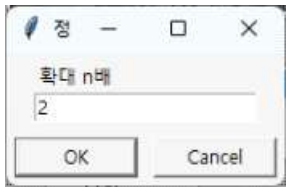


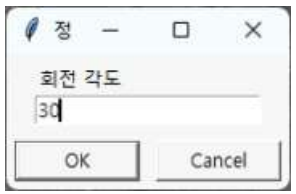
Input

Output

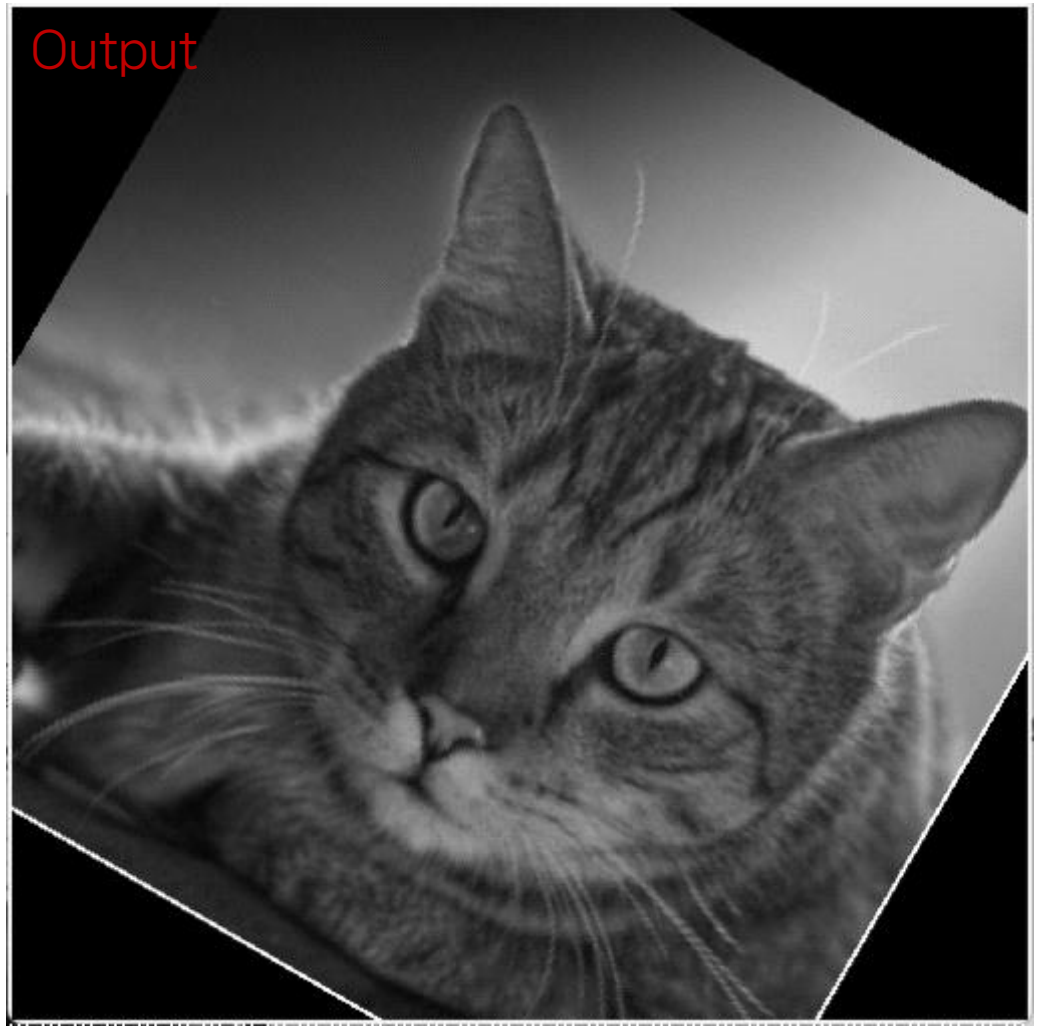
기하학 처리 - 회전(확대&양선형)

정수 두 개를 입력 시,
해당 값만큼 양선형 보간으로 확대 및 회전



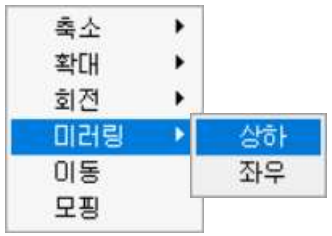


Input



기하학 처리 - 미러링(상하)

상하 대칭 적용




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

Input

Output

기하학 처리 - 미러링(좌우)

좌우 대칭 적용


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

Input

Output

기하학 처리 - 이동

정수 2개를 입력 시,
기존 위치에서 해당 값만큼 이동

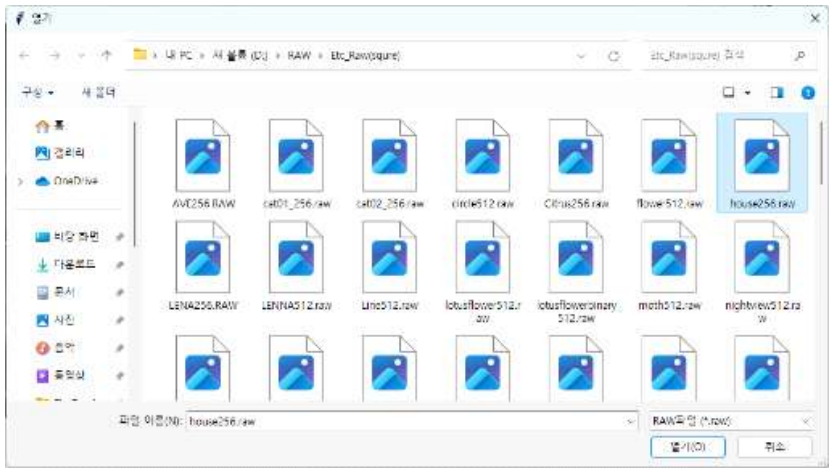

```
nx = i + posH
ny = j + posW

if (0 <= nx and nx < outH) and (0 <= ny and ny < outW):
    outImage[nx][ny] = inImage[i][j]
```

Input

Output

기하학 처리 - 모핑

다른 파일을 선택 시,
모핑 변환 (두 파일을 혼합)



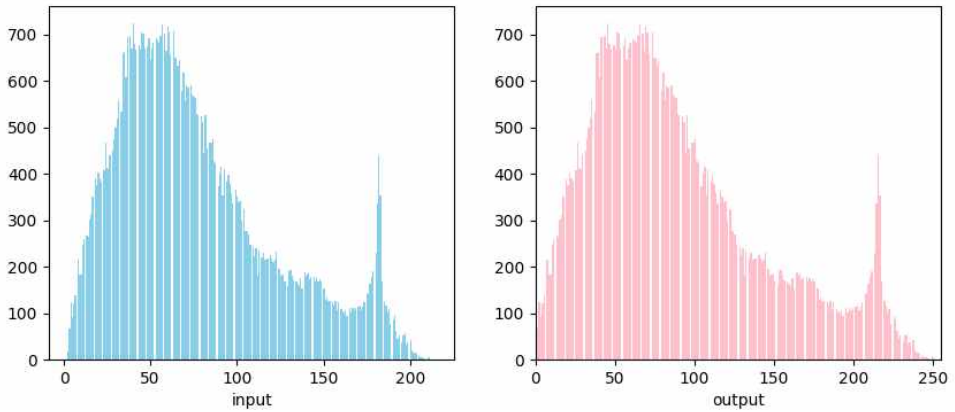
```
u = ((i * inH) + (j + 1.0)) / (inH * inW)
tmp = (int)((1.0 - u) * inImage[i][j] + u * tmpImage[i][j])

if 255 < tmp:
    outImage[i][j] = 255
elif tmp < 0:
    outImage[i][j] = 0
else:
    outImage[i][j] = int(tmp)
```

Input

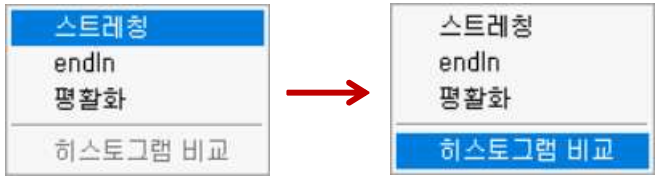
Output

히스토그램 처리 - 스트레칭

히스토그램 : 데이터를 막대 그래프 모양으로 나타낸 것



특정 부분에 집중된 히스토그램을 모든 영역으로 확장



Input

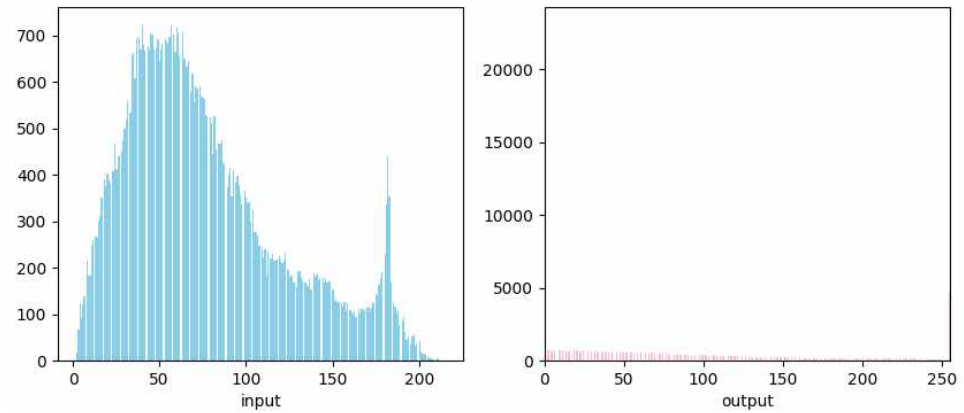
Output

히스토그램 처리 => [비교] 활성화
(*다른 기능 사용 시, 비활성화)

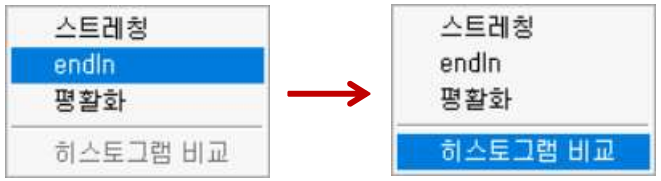
```
if inImage[i][j] < low:
    low = inImage[i][j]

if high < inImage[i][j]:
    high = inImage[i][j]
```

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

if new < 0:
    new = 0

if 255 < new:
    new = 255

outImage[i][j] = new
```

히스토그램 처리 - endIn



일정한 양의 화소를 흰색이나 검정색으로 지정



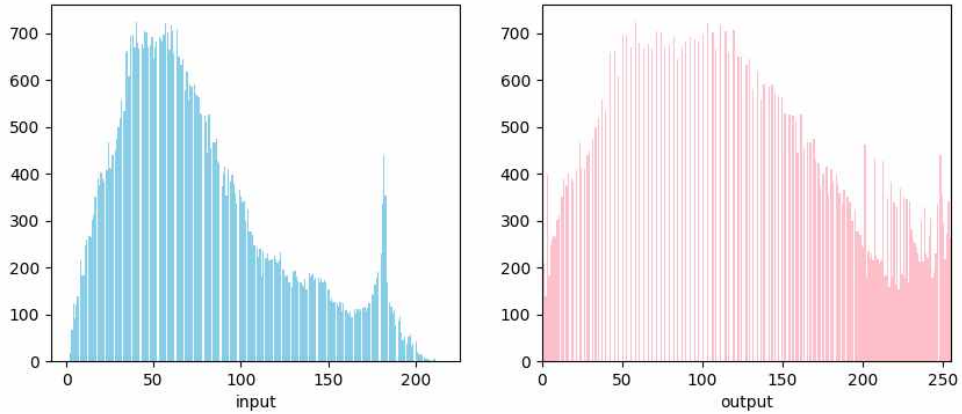

```
        if inImage[i][j] < low:
            low = inImage[i][j]

        if high < inImage[i][j]:
            high = inImage[i][j]
high -= 50
low += 50
```

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

if new < 0:
    new = 0

if 255 < new:
    new = 255

outImage[i][j] = new
```

Input

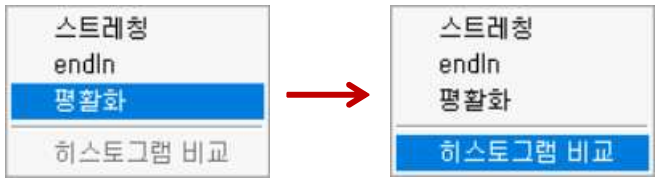
Output

히스토그램 처리 - 평활화



Input Output

영상의 밝기 분포를 재분배하여 명암 대비를 최대화




```
histo[inImage[i][j]] += 1
sumHisto[i] = sumHisto[i - 1] + histo[i]
normalHisto[i] = sumHisto[i] * (1.0 / (inH * inW)) * 255.0
outImage[i][j] = int(normalHisto[inImage[i][j]])
```

영역 처리 - 엠보싱

양각 형태로 변환

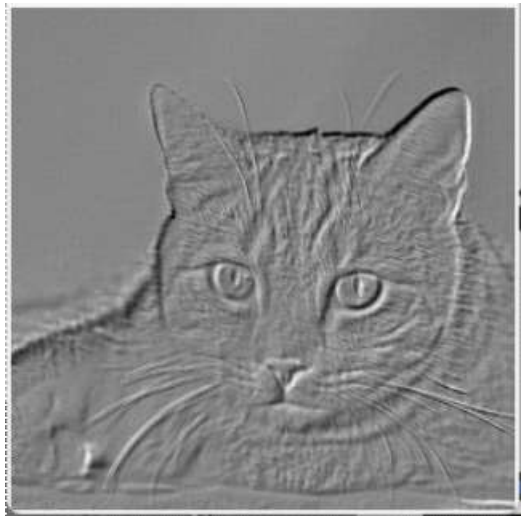

```
mask = [[-1.0, 0.0, 0.0],
        [0.0, 0.0, 0.0],
        [0.0, 0.0, 1.0]] # 엠보싱 마스크
```

```
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][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

```
if tmpOutImage[i][j] < 0.0:
    outImage[i][j] = 0
elif 255.0 < tmpOutImage[i][j]:
    outImage[i][j] = 255
else:
    outImage[i][j] = int(tmpOutImage[i][j])
```

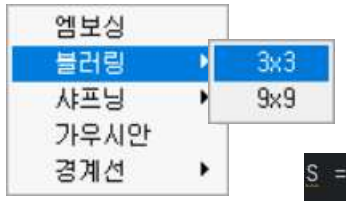
Input

Output



영역 처리 - 블러링(3x3)

3x3 마스크로 영상의 세밀한 부분을 제거




```
mask = [[1.0 / 9, 1.0 / 9, 1.0 / 9],
        [1.0 / 9, 1.0 / 9, 1.0 / 9],
        [1.0 / 9, 1.0 / 9, 1.0 / 9]] # 블러링 마스크
```

```
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][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

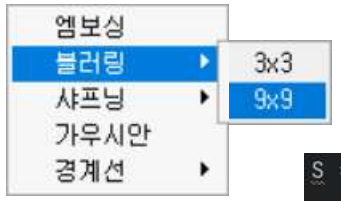
```
if tmpOutImage[i][j] < 0.0:
    outImage[i][j] = 0
elif 255.0 < tmpOutImage[i][j]:
    outImage[i][j] = 255
else:
    outImage[i][j] = int(tmpOutImage[i][j])
```

Input

Output

영역 처리 - 블러링(9x9)

9x9 마스크로 영상의 세밀한 부분을 제거




```
for i in range(9):
    for j in range(9):
        mask[i][j] = 1.0 / 81
```

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

for m in range(9):
    for n in range(9):
        S += tmpInImage[i + m][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

```
if tmpOutImage[i][j] < 0.0:
    outImage[i][j] = 0
elif 255.0 < tmpOutImage[i][j]:
    outImage[i][j] = 255
else:
    outImage[i][j] = int(tmpOutImage[i][j])
```

Input

Output

영역 처리 - 샤프닝

영상의 상세한 부분을 더욱 강조


```
mask = [[0.0, -1.0, 0.0],
    [-1.0, 5.0, -1.0],
    [0.0, -1.0, 0.0]] # 샤프닝 마스크
```

```
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][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

Input

Output

영역 처리 - 샤프닝(고주파)

고주파 통과 필터를 적용해 고주파 성분 강조


```
mask = [[-1.0 / 9, -1.0 / 9, -1.0 / 9],
    [-1.0 / 9, 8.0 / 9, -1.0 / 9 ],
    [-1.0 / 9, -1.0 / 9, -1.0 / 9]] # 샤프닝(고주파) 마스크
```

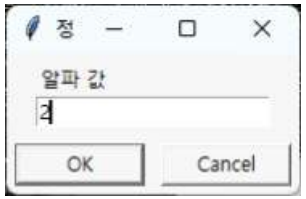
```
S = 0.0  # 마스크 9개와 입력값 9개를 각각 곱해서 합한 값

for m in range(3):
    for n in range(3):
        S += tmpInImage[i + m][j + n] * mask[m][n] * 20

tmpOutImage[i][j] = S
```

Input

Output

영역 처리 - 샤프닝(저주파)

정수를 입력 시,
(원 영상) – (저주파 통과 필터링 결과 영상)




```
mask = [[1.0 / 9, 1.0 / 9, 1.0 / 9],
    [1.0 / 9, 1.0 / 9, 1.0 / 9],
    [1.0 / 9, 1.0 / 9, 1.0 / 9]] # 샤프닝(저주파) 마스크
```

```
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][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

```
unsharp = alpha * inImage[i][j] - tmpOutImage[i][j]
```

Input

Output

영역 처리 - 가우시안

고주파 성분을 제거해 영상을 부드럽게 만듦


```
mask = [[1.0 / 16, 1.0 / 8, 1.0 / 16],
    [1.0 / 8, 1.0 / 4, 1.0 / 8],
    [1.0 / 16, 1.0 / 8, 1.0 / 16]] # 가우시안 마스크
```

```
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][j + n] * mask[m][n]

tmpOutImage[i][j] = S
```

Input

Output

영역 처리 - 경계선(수직)

수직 경계선 검출

(*edge(경계선) : 영상에서 밝기가 급격하게 변하는 부분, 영상을 구성하는 객체 간의 경계)


```
mask = [[0.0, 0.0, 0.0],
    [-1.0, 1.0, 0.0],
    [0.0, 0.0, 0.0]] # 수직 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(수평)

수평 경계선 검출


```
mask = [[0.0, -1.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0]] # 수평 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(유사 연산자)

화소를 감산한 값에서
최대값을 결정하여 경계선 검출


```
S = 0.0

for m in range(3):
    for n in range(3):
        if S < math.fabs(tmpInImage[i + 1][j + 1] - tmpInImage[i + m][j + n]):
            S = math.fabs(tmpInImage[i + 1][j + 1] - tmpInImage[i + m][j + n])

tmpOutImage[i][j] = S
```

Input

Output

영역 처리 - 경계선(차 연산자)

중앙을 기준으로 마주보는 값끼리 감산한 값에서 최대값을 결정하여 경계선 검출
(*뺄셈 연산이 유사 연산자와는 달리 화소당 네 번만 사용되어 빠르게 경계선 검출)

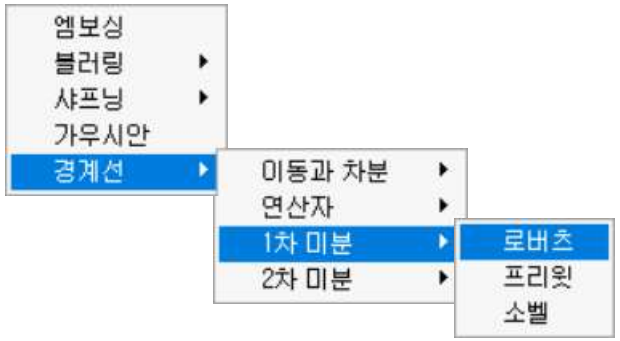

```
S = 0.0

for n in range(3):
    if S < math.fabs(tmpInImage[i][j + n] - tmpInImage[i + 2][j - n + 2]):
        S = math.fabs(tmpInImage[i][j + n] - tmpInImage[i + 2][j + -n + 2])

if S < math.fabs(tmpInImage[i + 1][j + 2] - tmpInImage[i + 1][j]):
    S = math.fabs(tmpInImage[i + 1][j + 2] - tmpInImage[i + 1][j])

tmpOutImage[i][j] = S
```

Input

Output

영역 처리 - 경계선(로버츠)

로버츠 마스크를 적용해 경계선 검출
(*크기가 작아 매우 빠른 속도로 동작하나,
돌출된 값을 잘 평균할 수 없으며, 잡음에 민감함)




```python
maskRow = [[-1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0]] # 로버츠 행 에지 검출 마스크

maskCol = [[0.0, 0.0, -1.0],
    [0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0]] # 로버츠 열 에지 검출 마스크
```

```python
S1 = S2 = 0.0

for m in range(3):
    for n in range(3):
        S1 += tmpRowInImage[i + m][j + n] * maskRow[m][n]
        S2 += tmpColInImage[i + m][j + n] * maskCol[m][n]

tmpRowOutImage[i][j] = S1
tmpColOutImage[i][j] = S2
```

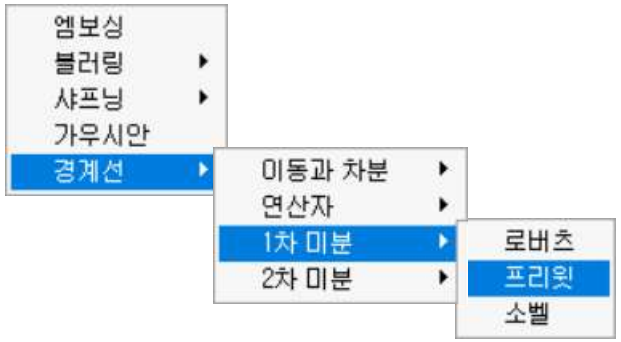
```python
S1 = tmpRowOutImage[i][j]
S2 = tmpColOutImage[i][j]

if math.sqrt(S1 * S1 + S2 * S2) < 0.0:
    outImage[i][j] = 0
elif 255.0 < math.sqrt(S1 * S1 + S2 * S2):
    outImage[i][j] = 255
else:
    outImage[i][j] = int(math.sqrt(S1 * S1 + S2 * S2))
```

Input

Output

영역 처리 - 경계선(프리윗)

프리윗 마스크를 적용해 경계선 검출
(*돌출된 값을 비교적 잘 평균화하나,
대각선보다 수평과 수직에 놓인 에지에
더 민감하게 반응함)



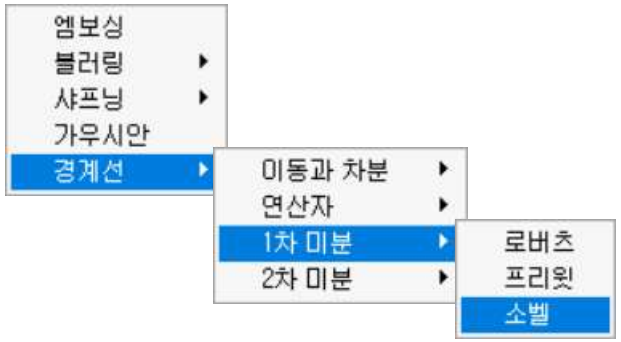

```
maskRow = [[-1.0, -1.0, -1.0],
    [0.0, 0.0, 0.0],
    [1.0, 1.0, 1.0]] # 프리윗 행 에지 검출 마스크

maskCol = [[1.0, 0.0, -1.0],
    [1.0, 0.0, -1.0],
    [1.0, 0.0, -1.0]] # 프리윗 열 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(소벨)

소벨 마스크를 적용해 경계선 검출
(*돌출된 값을 비교적 잘 평균화하나,
대각선 방향에 놓인 에지에 더 민감하게 반응함)



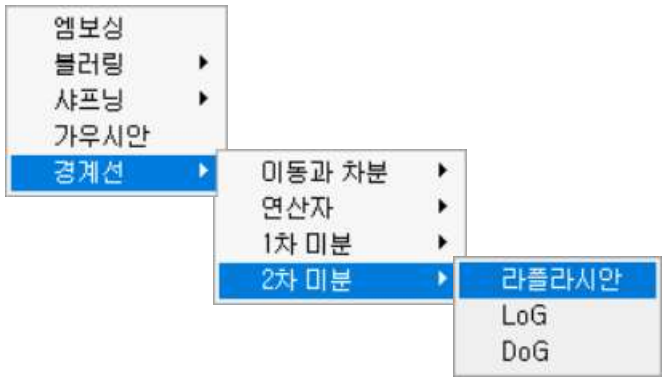

```
maskRow = [[-1.0, -2.0, -1.0],
    [0.0, 0.0, 0.0],
    [1.0, 2.0, 1.0]] # 소벨 행 에지 검출 마스크

maskCol = [[1.0, 0.0, -1.0],
    [2.0, 0.0, -2.0],
    [1.0, 0.0, -1.0]] # 소벨 열 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(라플라시안)

라플라시안 마스크를 적용해 경계선 검출
(*모든 방향의 에지를 강조하나,
잡음 성분에 매우 민감하여
실제보다 더 많은 에지를 검출함)




```
mask = [[0.0, -1.0, 0.0],
    [-1.0, 4.0, -1.0],
    [0.0, -1.0, 0.0]] # 라플라시안 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(LoG)

LoG 마스크를 적용해 경계선 검출
(*가우시안 스무딩으로 잡음 제거한 뒤,
에지를 강조하기 위해 라플라시안을 이용)


```
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]] # LoG 에지 검출 마스크
```

Input

Output

영역 처리 - 경계선(DoG)

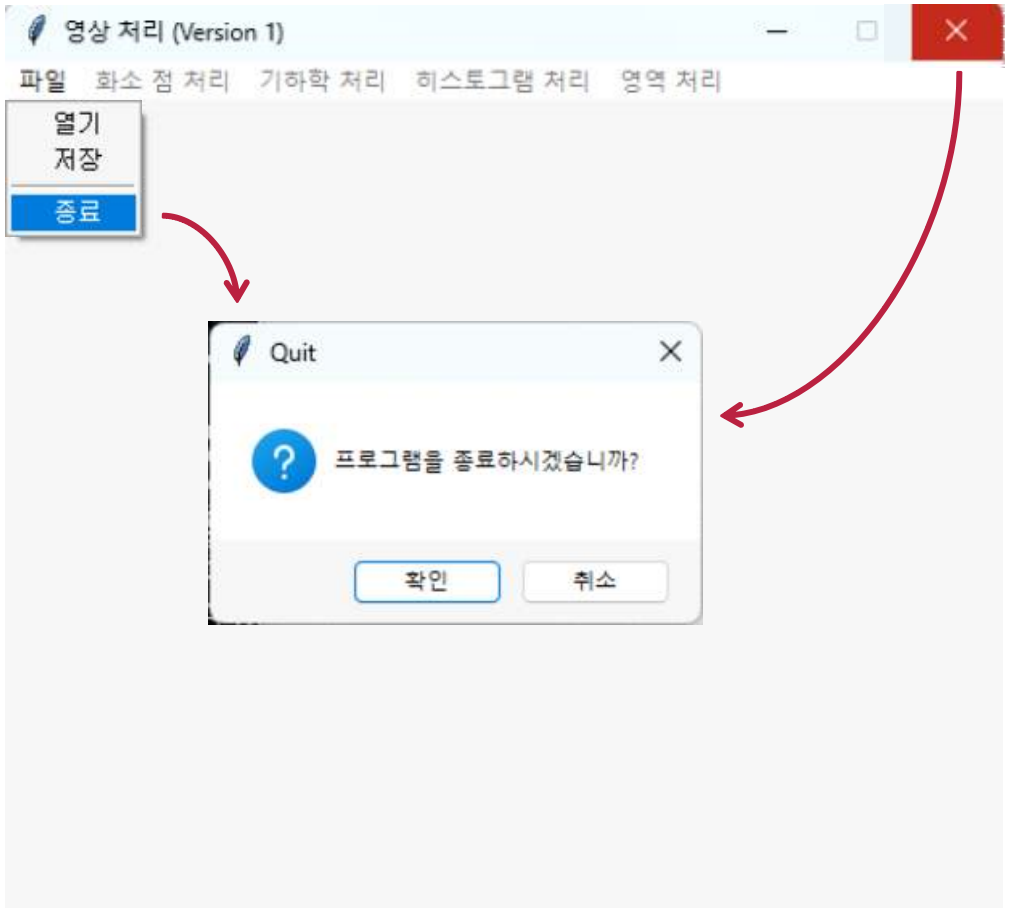
DoG 마스크를 적용해 경계선 검출
(*각 가우시안 연산에 분산 값을 서로 다르게 줘
이 차를 이용해 에지 맵을 구함, 계산 시간이
느린 LoG 연산자의 단점을 보안)


```
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]] # DoG 에지 검출 마스크
```

Input

Output

종료



1. [종료]
2. 윈도우 창 [X]

클릭 시, 종료 확인 창으로
종료 여부 재확인

[확인] -> 종료
[취소] -> 종료 취소

정리

느낀점

직접 영상 처리 프로그램을 구현해 봄으로써 영상 처리에 있어서 어떠한 기능들이 있고, 각각의 기능들이 어떤 과정을 통해 작동하는지를 보기만 했을 때보다 더 잘 알고, 이해할 수 있는 기회가 되었던 것 같다.
C 언어로 이미 개발했던 프로그램을 다시 파이썬으로 보다 쉽게 구현할 수 있었고, 일반 사용자를 기준으로 UI가 더 편하고 직관적인 프로그램이 된 것 같다.

향후 발전 방향

- 효과 누적 및 되돌리기가 가능하도록 기능 추가
- 더 다양한 영상 처리 기능 추가
- 필터 크기 입력 받아 적용