20190002_강다혜_HW2

1. Spatial Filtering

a. conv_2D() 함수 구현

함수 이름은 숫자로 시작할 수 없으므로 conv_2D() 함수를 작성한다.

```
% 원본 이미지의 모든 픽셀에 대하여 컨볼루션 작업을 수행한다.
for ir = 1:imgRow
    for ic = 1:imgCol
        ret(ir, ic) = 0;

        % 한 픽셀에 대하여 커널의 모든 값을 곱한다.
        for kr = 1:kerRow
            for kc = 1:kerCol
                nexR = ir - kr + floor(kerRow / 2) + 1;
                nexC = ic - kc + floor(kerCol / 2) + 1;

                % 현재 픽셀 인덱스가 이미지 범위 안인 경우 해당 픽셀을 참조한다.
                if (1 <= nexR && nexR <= imgRow &&...
                        1 <= nexC && nexC <= imgCol)
                    currentPixel = img(nexR, nexC);
                else
                    % 이미지 바깥에 있는 경우 zero padding을 수행한다.
                    currentPixel = 0;
                end

                ret(ir, ic) = ret(ir, ic) + double(currentPixel) * kernel(kr, kc);
            end
```

```
        end
    end
end
```

box filter를 쉽게 생성할 수 있는 함수를 작성했다.

호출은 아래와 같이 한다. 인자로 box filter의 사이즈를 a를 입력하면 a * a 사이즈의 필터가 생성된다.

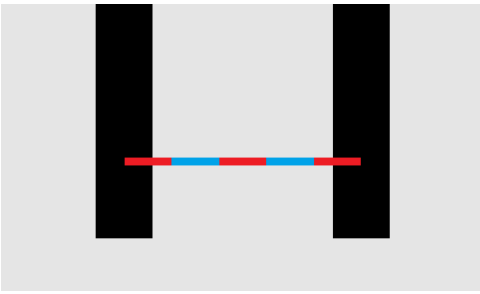
```
kernel = makeBoxFilter(25);
```

구현은 아래와 같다.

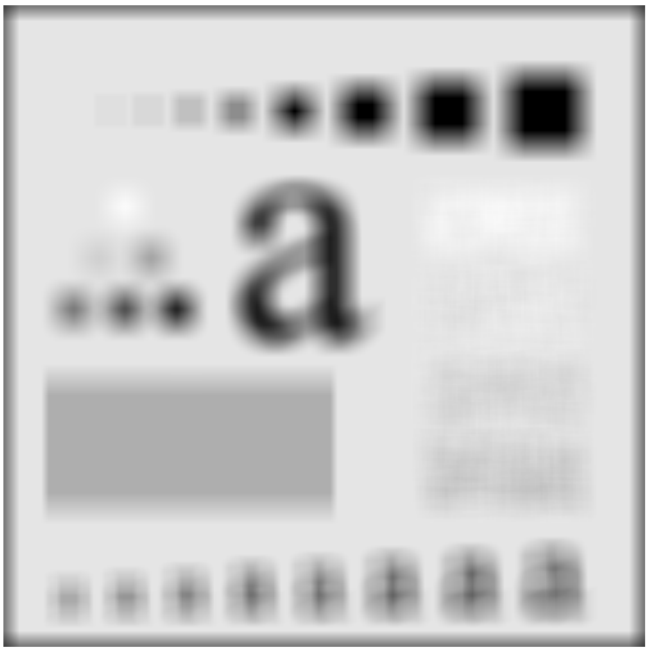
```
function ret = makeBoxFilter(size)
ret = zeros(size, size);
for r = 1:size
    for c = 1:size
        ret(r, c) = 1 / (size * size);
    end
end
end
```

b. conv_2D() 함수 + box filter

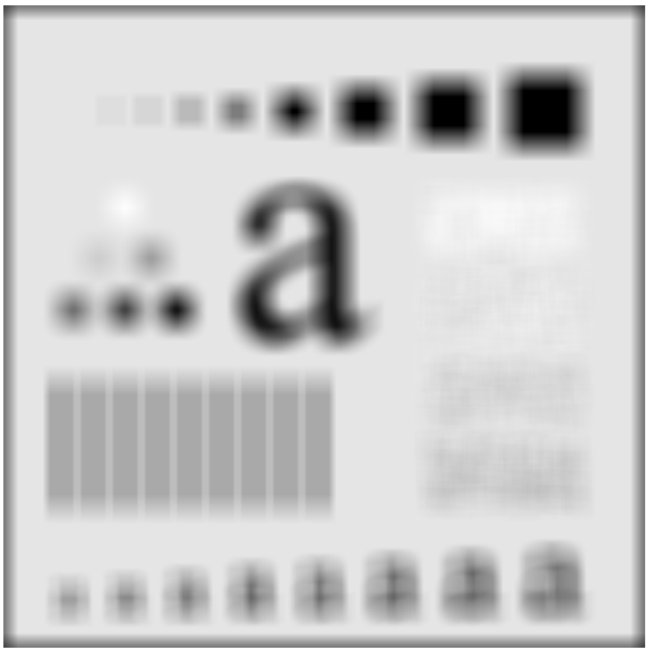
5픽셀씩 나누어 세어 본 결과 세로선과 세로선 사이의 간격은 25픽셀이었다. 따라서 25픽셀 사이즈의 box filter를 적용한다면 세로선들이 하나의 직사각형으로 보일 것이다.



`kernel = makeBoxFilter(25);`로 생성된 필터를 적용시킨 경우 결과물은 이렇다.



`kernel = makeBoxFilter(23);`로 생성된 필터를 적용시킨 경우 결과물은 이렇다. 직사각형이 2픽셀씩 떨어져 있는 것을 확인할 수 있다.



c. conv_2D() 함수 + box filter (separable)

위 코드에서 box filter는 $\frac{1}{\text{size}^2} \times \begin{bmatrix} 1 & 1 & ... & 1 \\ 1 & 1 & ... & 1 \\ ... & & & \\ 1 & 1 & ... & 1 \end{bmatrix}$ 처럼 사용되는데, 이 2D 행렬은 아래와 같이 1D 행렬 두 개의 곱으로 나타낼 수 있다.

$$\frac{1}{\text{size}^2} \times \begin{bmatrix} 1 & 1 & ... & 1 \\ 1 & 1 & ... & 1 \\ ... & & & \\ 1 & 1 & ... & 1 \end{bmatrix}$$

$$= \frac{1}{\text{size}^2} \times \begin{bmatrix} 1 & 1 & ... & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}$$

$$= \frac{1}{\text{size}} \times \begin{bmatrix} 1 & 1 & ... & 1 \end{bmatrix} \times \frac{1}{\text{size}} \times \begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}$$

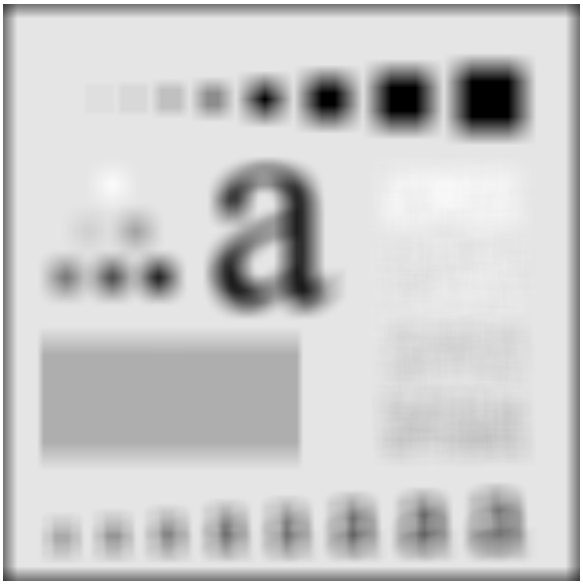
이렇게 행렬을 행벡터와 열벡터의 곱으로 나타낼 수 있다면, 원본 이미지에 행벡터와 벡터를 순서대로 적용한 결과와 원본 행렬을 적용한 결과가 같다.

```
BOX_FILTER_SIZE = 25;
img = imread('test_pattern.tif');

row_vector = zeros(BOX_FILTER_SIZE, 1);
for i = 1:BOX_FILTER_SIZE
    row_vector(i, 1) = 1 / double(BOX_FILTER_SIZE);
end
box = conv_2D(img, row_vector);

col_vector = zeros(1, BOX_FILTER_SIZE);
for i = 1:BOX_FILTER_SIZE
    col_vector(1, i) = 1 / double(BOX_FILTER_SIZE);
end
box = conv_2D(box, col_vector);
```

위 코드처럼 순서대로 행벡터와 열벡터를 convolution 시켜주었고, 결과는 아래 이미지와 같다.



Filter가 separable한 경우, 행렬을 그대로 이용하기보다 행벡터와 열벡터를 이용하는 쪽이 연산을 덜 하게 된다. 원본 이미지의 크기를 $(imgRow, imgCol)$, filter의 크기를 $(kerRow, kerCol)$로 두자. 행렬을 그대로 이용한 경우에는 $imgRow * imgCol * kerRow * kerCol$만큼의 연산이 필요하다. 행백터와 열벡터를 이용한 경우에는 $imgRow * imgCol * kerRow + imgRow * imgCol * kerCol = imgRow * imgCol * (kerRow + kerCol)$만큼의 연산이 필요하기 때문에 횟수가 줄어들게 된다.

d. conv_2D() 함수 + sobel filter

교과서에서 정의된 sobel operator는 다음과 같다.

$$M(x,y) \approx \\ |(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)| \\ + \\ |(z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)|$$

이 연산 결과를 얻으려면 아래의 sobel filter를 취한 결과를 각각 절댓값을 취한 뒤 더해야 한다.

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

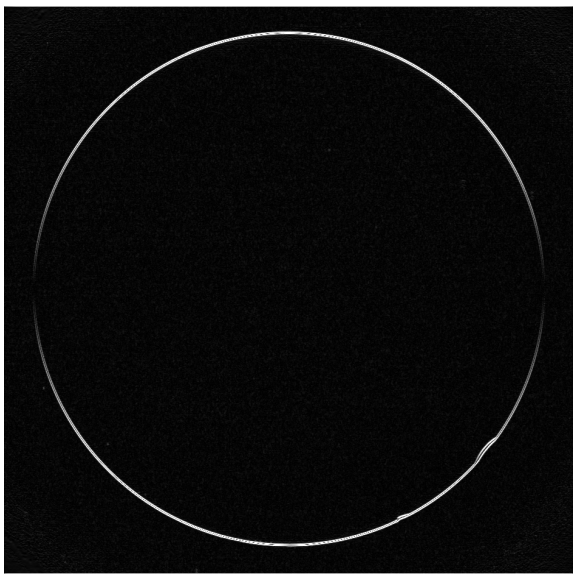
-1	0	1
-2	0	2
-1	0	1

sobel filter를 취한 결과는 이전에 구현한 `conv_2D()` 함수를 이용하여 다음과 같이 얻을 수 있다.

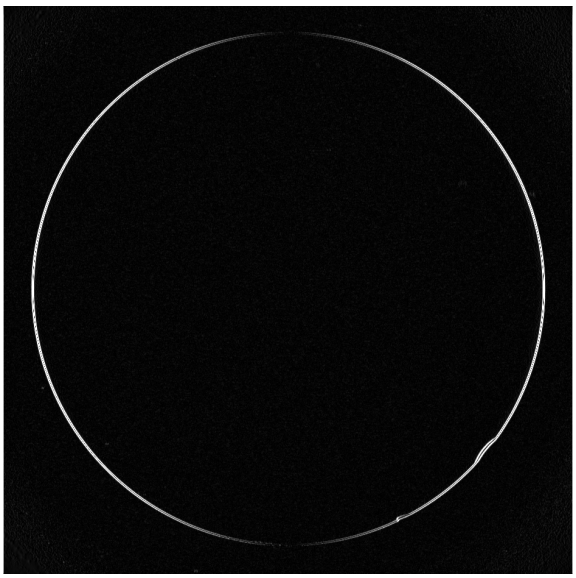
```
sobel_filter_y = [-1 -2 -1; 0 0 0; 1 2 1];
sobel_y = conv_2D(img, sobel_filter_y);

sobel_filter_x = [-1 0 1; -2 0 2; -1 0 1];
sobel_x = conv_2D(img, sobel_filter_x);
```

이 단계를 수행한 이후 0~255 범위를 벗어나는 픽셀 값을 정리하고 이미지를 출력하면 아래와 같다.



sobel_y : y축의 방향으로 테두리를 검출한 이미지

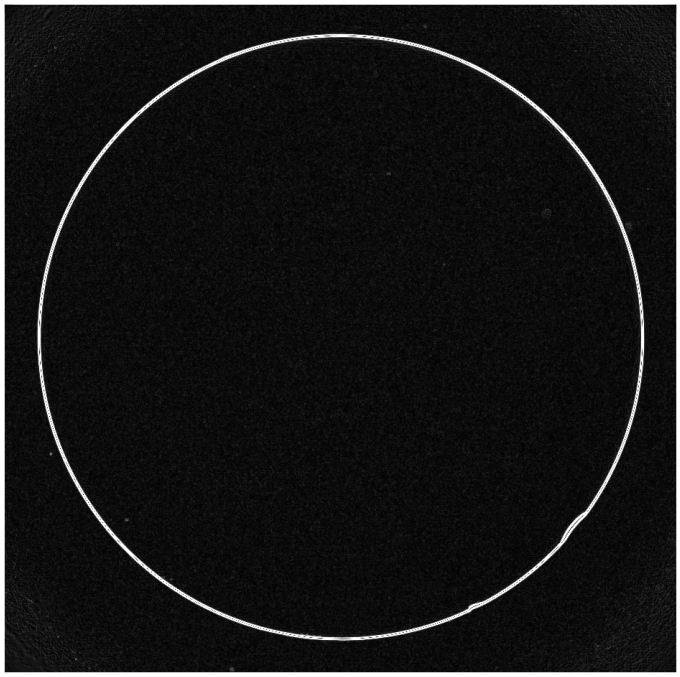


sobel_x : x축의 방향으로 테두리를 검출한 이미지

이 결과값을 절댓값을 취하고 더하면 최종적인 결과를 얻을 수 있다.

```
sobel_y = abs(sobel_y);
sobel_x = abs(sobel_x);

sobel_final = sobel_y + sobel_x;
```



최종적인 이미지는 위와 같다.

e. conv_2D() 함수 + sobel filter (separable)

sobel filter의 각 행렬 $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$, $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 는 separable하고, 다음과 같이 표현될 수 있다.

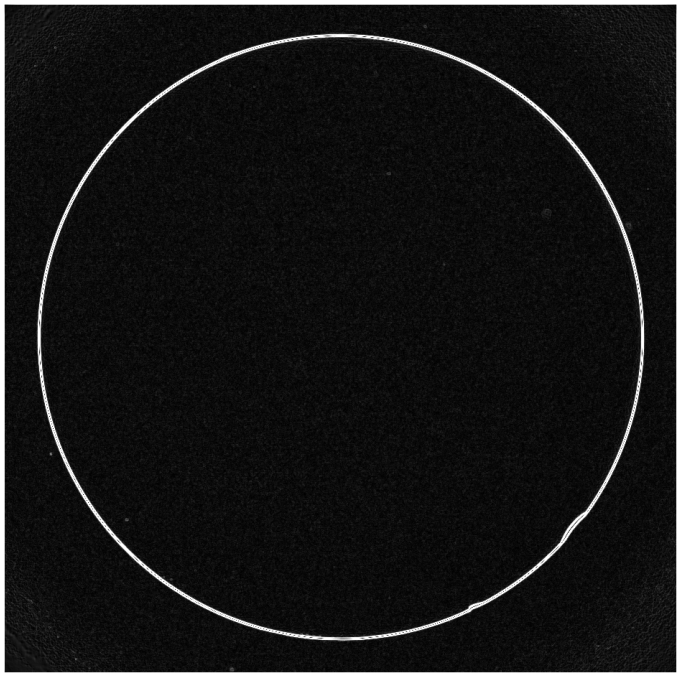
$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

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

```
sobel_y = conv_2D(img, [1 2 1]);
sobel_y = conv_2D(sobel_y, [-1; 0; 1]);

sobel_x = conv_2D(img, [-1 0 1]);
sobel_x = conv_2D(sobel_x, [1; 2; 1]);
```

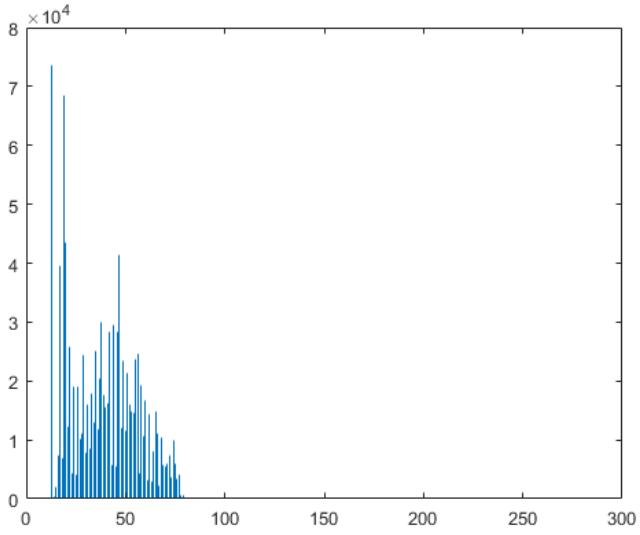
실제 구현은 위처럼 순서대로 conv_2D 함수에 넘겨주었고, 최종적인 이미지는 아래와 같다.



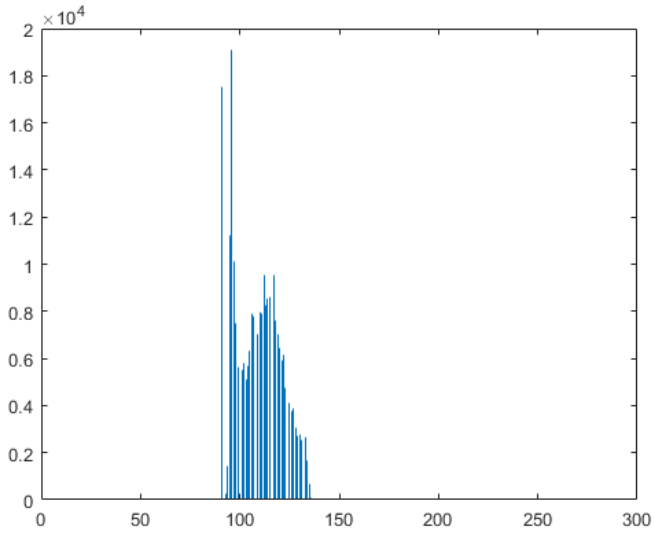
2. Histogram equalization

a. 이미지의 histogram 분석하기

“pollen1.tif”의 픽셀값을 히스토그램으로 나타내면 다음과 같다. 픽셀값은 0부터 255까지일 수 있는데, “pollen1.tif” 이미지는 0 ~ 80 사이에 픽셀값이 분포하고 있는 모습을 보인다.



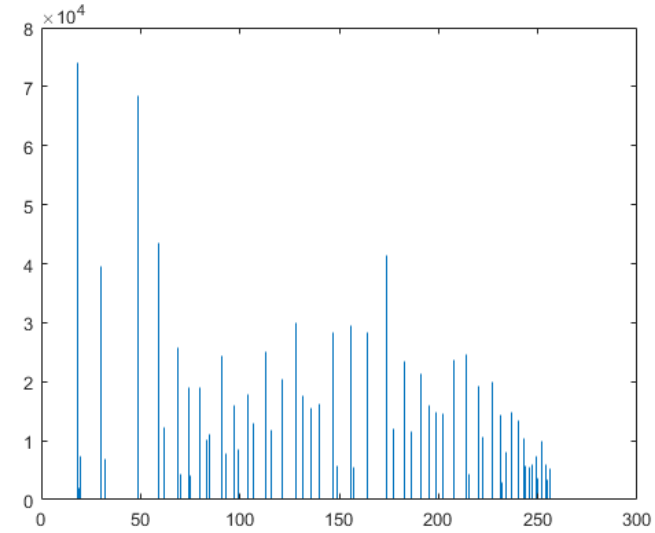
“pollen2.tif”의 픽셀값을 히스토그램으로 나타내면 다음과 같다. 이 이미지는 대략 100 ~ 150 사이에 픽셀값들이 분포하고 있다.



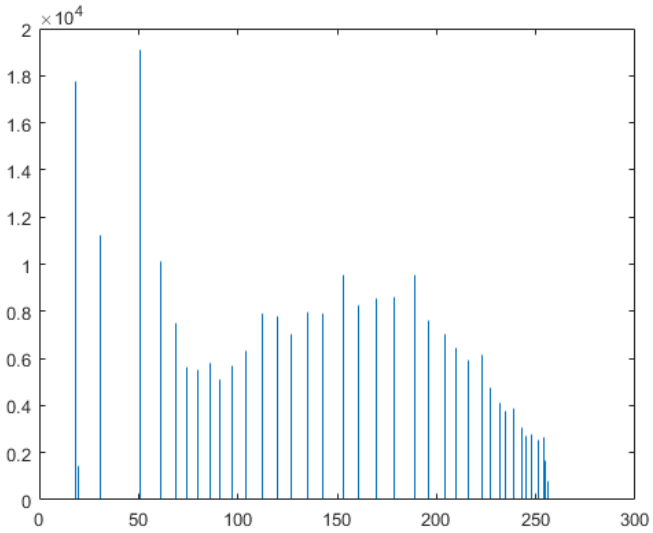
“pollen1.tif”보다 “pollen2.tif”가 더 255쪽으로 치우쳐 있으므로, 전체적으로 밝다.

b. histogram equalization 수행 후 histogram 분석하기

“pollen1.tif”에 histogram equalization을 수행한 후 히스토그램을 나타내면 아래와 같다. 픽셀값이 전체적으로 0에서 255 사이에 분포하고 있다.



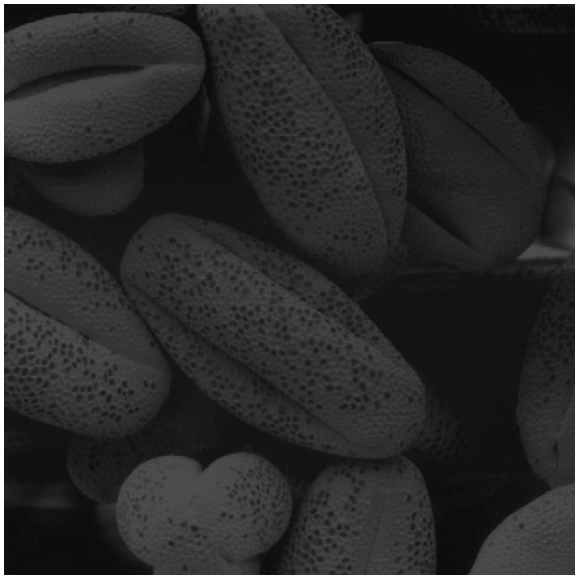
“pollen2.tif”에 histogram equalization을 수행한 후 히스토그램을 나타내면 아래와 같다. 역시 픽셀값이 0에서 255 사이에 분포하고 있다.



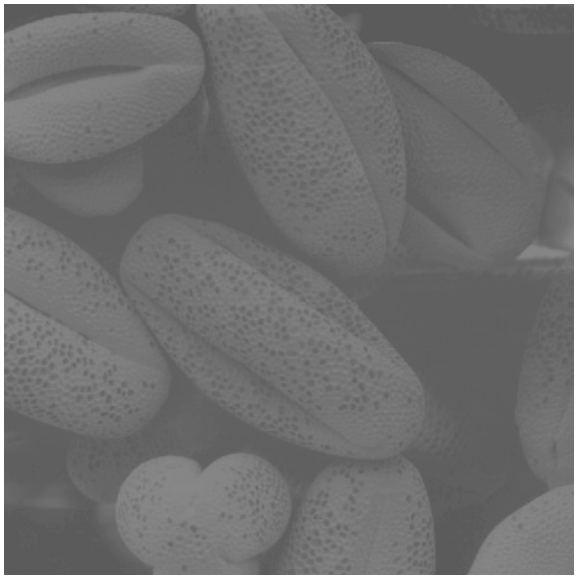
c. 원본과 결과 이미지의 histogram의 차이에 따른 시각적 변화

“pollen1.tif”의 equalization 결과와 “pollen2.tif”의 equalization 결과를 살펴보면 두 히스토그램의 분포가 유사함을 알 수 있다.

원본 이미지는 공통적으로 150 이상의 값을 가지는 픽셀이 없기 때문에 전체적으로 어둡다. 또한 pollen1의 픽셀값들이 pollen2의 픽셀값들보다 0쪽으로 더 가깝게 분포하고 있으므로 pollen1이 더 어둡다.

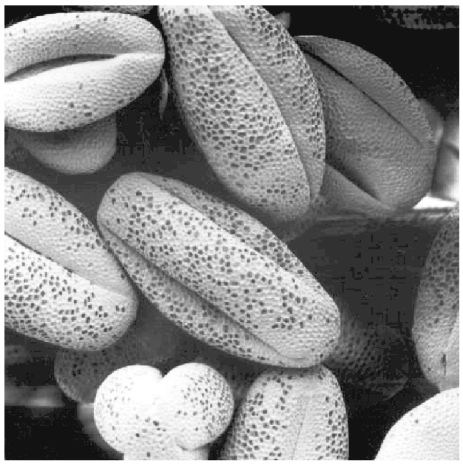


pollen1.tif 원본 이미지

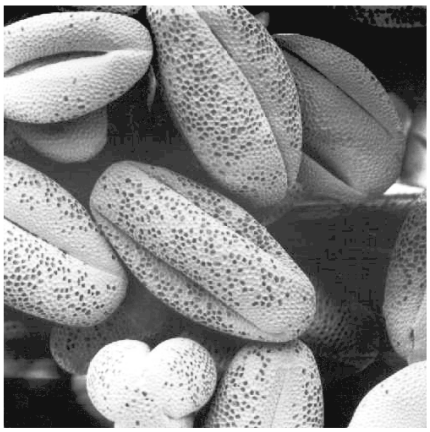


pollen2.tif 원본 이미지

결과 이미지는 아래와 같다.



pollen1.tif에 histogram equalization을 적용한 결과



pollen2.tif에 histogram equalization을 적용한 결과

두 이미지에 histogram equalization을 실시한 결과, 픽셀값의 분포가 거의 비슷하게 드러났다. 결과 이미지에서도 두 이미지가 비슷하게 드러났다.

3. 검토사항

a. 2D filter에서 separable의 조건

2D filter가 separable하다는 것은 2D filter가 1D의 행벡터와 1D 열벡터의 외적으로 표현될 수 있다는 뜻이다. 이 때, 두 개의 1D 벡터는 서로 독립적이어야 한다. 즉, 두 개의 필터가 서로의 작업에 영향을 미치지 않아야 한다.

b. 작은 filter를 여러 번 사용하유

필터의 사이즈가 $n * n$이고 원본 이미지가 $N * M$인 경우, 필터를 한 번 적용시키려면 n^2NM만큼의 연산량이 필요하다. 반면에 크기가 $3 * 3$인 필터를 한 번 적용시킨다면, 3^2NM만큼의 연산량이 필요한데, 이를 반복하면 $k * 3^2NM$만큼의 연산량이 필요하다. 시간복잡도의 측면에서 보았을 때, 전자는 $Olog(n^2)$이고, 후자는 $Olog(n)$이므로 후자가 더 이득이다. 또한, 후자는 $3 * 3$ 크기의 필터만 메모리에 올리면 되지만, 전자는 $n * n$크기의 필터를 올려야 하므로 메모리 소비가 심하다.

c. separable한 filter의 연산량

원본 이미지의 크기를 $m * n$이라고 할 때, $M * N$ 크기의 2D filter를 적용시키려면 총 $mnMN$만큼의 연산이 필요하다. 반면에 해당 2D filter가 separable하다면, 연산량은 $mnM + mnN = mn(M + N)$만큼으로 줄어든다. 결과적으로 $1 : (\frac{MN}{M+N})$의 비율로 연산량에 이득이 있다.

d. gamma correction과 histogram equalization

Gamma correction은 비선형 전달 함수를 사용하여 빛의 강도 신호를 비선형적으로 변형하는 기법이다. histogram equalization에 비해서 빠르고, 계산량이 적지만 대비가 적은 영상에서는 정보가 손실될 수 있다.

Histogram equlization은 대비가 낮은 영상에서 대비를 향상시킬 수 있다. 하지만 대비가 이미 높은 영상에서 노이즈를 발생시킬 수 있고, 큰 이미지에서는 연산량이 많아 느리다.

Gamma correction과 histogram equalization의 장점을 결합한 Adaptive Gamma Correction이라는 기법이 있다.