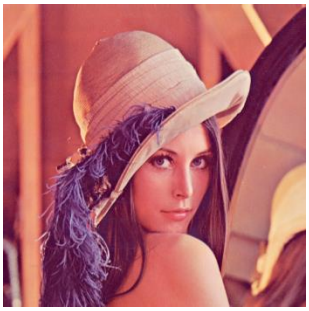
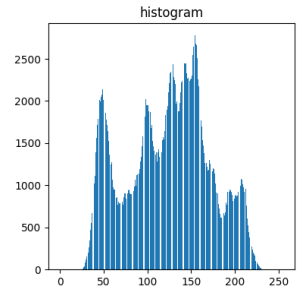
영상처리 Project 1

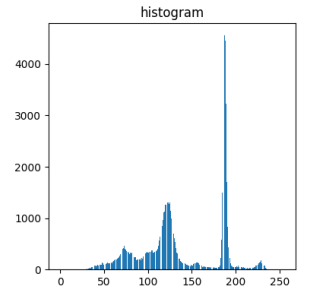
2016121150 윤준영

0. Original images.

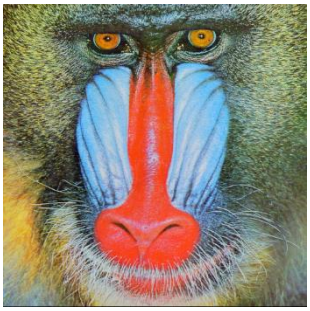
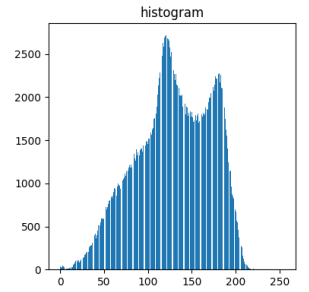
본 과제에서 사용한 영상과 각 영상의 히스토그램은 다음과 같다.

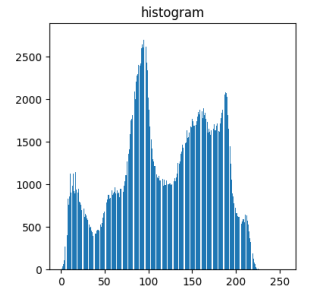
<lena>

<house>

<babbon>

<pepper>

1. Image denoising.

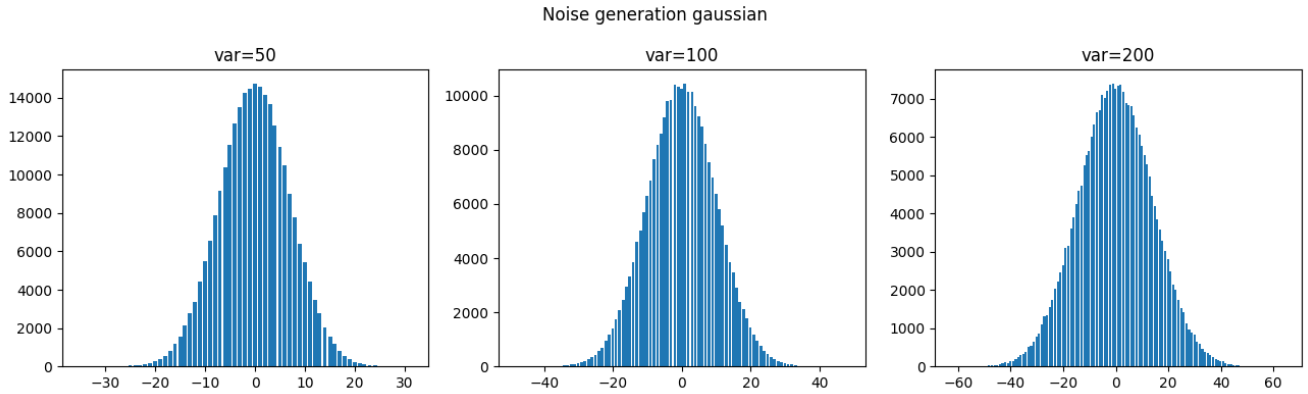
1.1. Generate the following noises, and add them to the ground-truth images.

a) Gaussian noise with (mean, var) = (0, s^2)

Gaussian noise는 다음과 같은 분포를 가진 잡음이다.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x-\mu^2}{2\sigma^2}}, \qquad \mu = \text{mean}(0), \sigma = \text{standard deviation(s)}$$

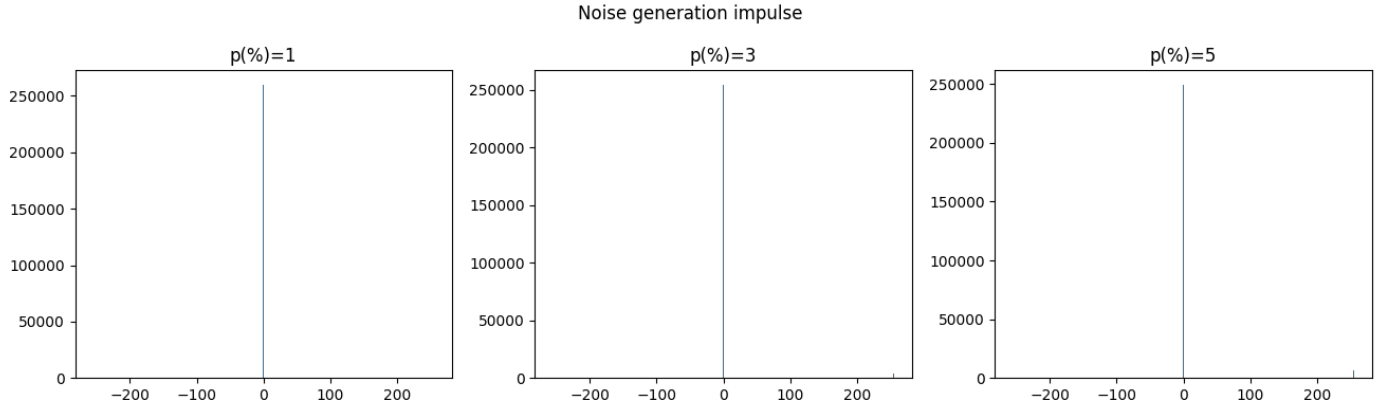
Gaussian 분포는 평균값에 가장 많은 분포가 있고 평균값에 가까울수록 많은 분포를 가진다. 표준 편차가 작을수록 분포의 폭이 좁아지며, 평균값에 더 많은 표본이 모여있으며, 표준 편차가 커지면 분포의 폭이 넓어지고, 표본이 평균값과 비교적 떨어지게 된다. Gaussian noise의 히스토그램은 다음과 같다.



b) Impulse noise (p%) ; Generated by uniform random number

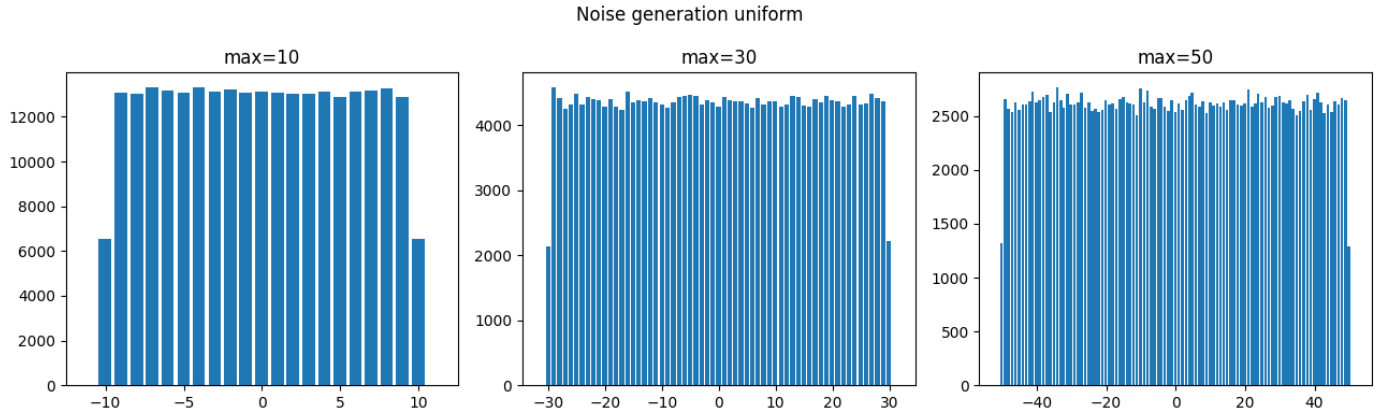
Impulse noise의 경우 p%의 표본이 크거나 작은 특정 값을 가지며, 나머지의 표본은 평균값(0)을 가지게 된다.

$$I(x) = \begin{cases} 255 & (p/2)\% \\ 0 & (100-p)\% \\ -255 & (p/2)\% \end{cases}$$

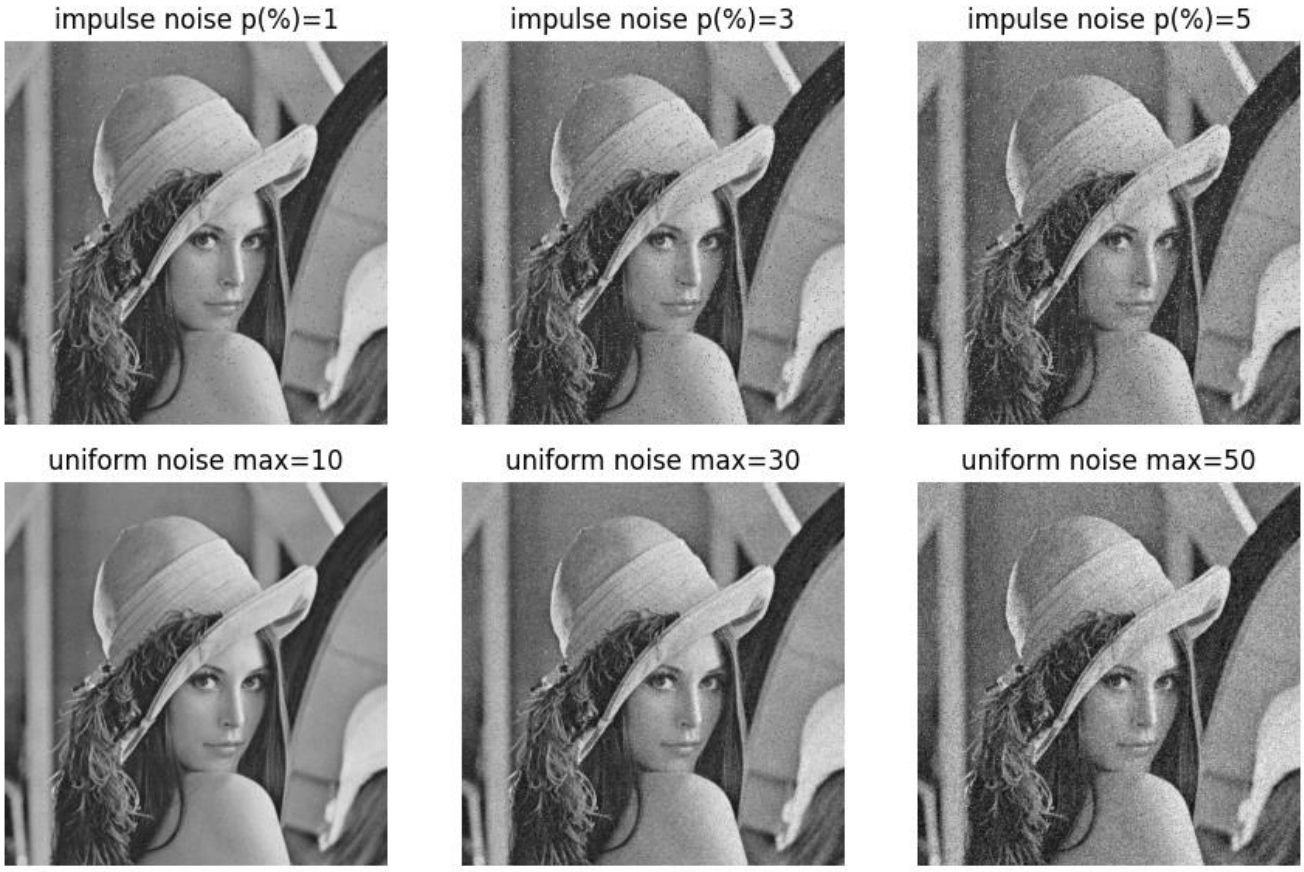


c) Uniform noise

Uniform noise는 최댓값과 최솟값 사이에서 균일한 분포를 가진다.



위 세 가지 잡음을 서로 다른 세 변수를 지정하며 적용한 결과는 다음과 같다.



Gaussian noise의 경우 대부분 평균에 근접한 잡음이 생성되어 분산이 낮은 경우에는 원본 영상과 유사한 영상을 얻을 수 있었다. Impulse noise가 적용된 영상은 흰 점과 검은 점이 나타나는 것을 확인할 수 있었으며, 눈으로 평가하였을 때 1%의 잡음만 들어가도 품질이 나빠지는 것을 체감할 수 있었다. Uniform noise를 적용한 경우 균일하게 분포된 잡음이 생성되어 max값이 높은 잡음이 적용된 영상의 경우 밝아진 부분과 어두워진 부분이 크게 차이나는 것을 확인할 수 있다.

1.2. Compute PSNR between ground-truth and noisy image.

영상의 잡음 정도를 수치로 나타내는 정량평가 방법 중 하나인 PSNR과 MSE는 다음과 같이 구할 수 있다.

$$\text{PSNR} = 10\log_{10}\frac{255^2}{MSE}(dB), \qquad MSE = \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W}[X(i,j)-Y(i,j)]^2$$

원본 영상을 이용하여 잡음이 적용된 영상의 PSNR을 구한 결과는 다음과 같다.

```
gaussian noise var=50, PSNR : 31.10dB
gaussian noise var=100, PSNR : 28.13dB
gaussian noise var=200, PSNR : 25.13dB
impulse noise p(%)=1, PSNR : 25.49dB
impulse noise p(%)=3, PSNR : 20.77dB
impulse noise p(%)=5, PSNR : 18.39dB
uniform noise max=10, PSNR : 32.89dB
uniform noise max=30, PSNR : 23.35dB
uniform noise max=50, PSNR : 18.95dB
```

모든 잡음이 적용된 영상은 잡음의 변수인 분산, p%, max값에 따라 달라지는 것을 확인할 수 있으며, 변수가 증가할수록 PSNR이 줄어들어 영상의 품질이 저하되는 것을 확인할 수 있다.

1.3~4. Implement the following denoising filters, and apply them for denoising. & Analyze

0) Filtering

잡음 제거를 하기 위하여 filtering 함수를 작성하였으나, 많은 반복문이 사용되어 잡음 제거 처리 속도가 느린 것을 체감할 수 있었다. 따라서 반복문을 최대한 적게 사용하는 행렬 연산 함수를 작성하였다.

```
def apply_filter(img, fil):
    ### appling filter to image (conventional) ###
    # input : image (MxN array), filter (mxn array)
    # output : filtered image (MxN array)
    result = np.copy(img)
    h = fil.shape[0]//2
    for y in range(h, img.shape[0]-h):
        for x in range(h, img.shape[1]-h):
            sub_img = img[y-h:y+h+1, x-h:x+h+1]
            result[y,x] = np.sum(fil * sub_img).astype(np.uint8)
    return resultas
```

기존 함수의 경우 mask의 크기와 같은 sub 행렬을 만들어 sub 행렬와 mask와 연산을 반복하여 픽셀을 하나하나 작성하는 방법이었지만, 새로운 함수는 영상 외곽을 mask 크기의 반 만큼 확장한 후 mask의 해당 위치로 shift 시켜 해당 mask 요소를 곱한 뒤 적층시켜 더하는 구조로 되어있어 mask의 크기 만큼만 반복 연산을 하면 된다. 상세한 내용은 아래와 같다.

```
def apply_filter2(img, fil, i=1, ab=1):
    ### appling filter to image (Matrix calculate) ###
    # input : image (MxN array), filter (mxn array),
    #         i=1:output to integer, ab=1:output to absolute value
    # output : filtered image (MxN array)
    h1 = fil.shape[0]//2; h2 = fil.shape[1]//2
    result = np.zeros((img.shape[0]+2*h1, img.shape[1]+2*h2))
    ar1 = np.arange(-h1,h1+1); ar2 = np.arange(-h2,h2+1)
    for y in ar1:
        for x in ar2:
            sub = np.pad(img,((h1-y,h1+y),(h2-x,h2+x)),constant_values=0)
            result += fil[y+h1,x+h2] * sub
    return result[h1:-h1,h2:-h2].astype(np.uint8)
```

새로운 함수를 사용하여 7x7 크기의 Gaussian filter를 적용해 보니 큰 속도 차이를 얻을 수 있었다.

```
1.3.0) Faster filtering
apply_filter : 1.489sec
apply_filter2 : 0.028sec
```

a) Gaussian filter

Gaussian filter는 위에 사용한 Gaussian noise를 생성할 때와 마찬가지로 Gaussian 분포를 가진 mask를 이용하여 filtering하는 방법이다. 3x3 Gaussian filter는 다음과 같이 나타낼 수 있다. 중앙값이 (0,0)의 좌표를 가지고, 행렬의 합은 1이 되도록 K로 정규화 시켜준다. Filter의 크기는 6σ보다 큰 가장 작은 홀수로 한다.

$$G(y,x)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{y^2+x^2}{2\sigma^2}},\qquad M=\frac{1}{K}\begin{pmatrix} e^{-\frac{(-1)^2+(-1)^2}{2\sigma^2}} & e^{-\frac{(-1)^2+0^2}{2\sigma^2}} & e^{-\frac{(-1)^2+1^2}{2\sigma^2}} \\ e^{-\frac{0^2+(-1)^2}{2\sigma^2}} & e^{-\frac{0^2+0^2}{2\sigma^2}} & e^{-\frac{0^2+1^2}{2\sigma^2}} \\ e^{-\frac{1^2+(-1)^2}{2\sigma^2}} & e^{-\frac{1^2+0^2}{2\sigma^2}} & e^{-\frac{1^2+1^2}{2\sigma^2}} \end{pmatrix},\qquad \Sigma M=1$$

Gaussian noise($\sigma = 200$)가 적용된 영상에 Gaussian filter($\sigma = 0.5,\ 1,\ 3$)를 적용하여 계산해본 결과는 다음과 같다.

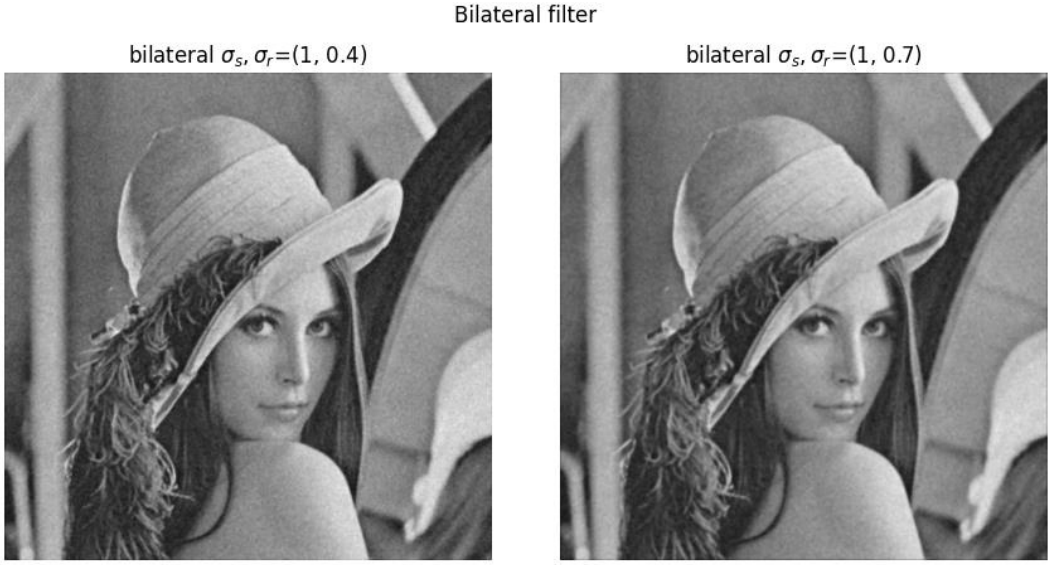

```
1.3.a) Gaussian filter
gaussian filter, sigma=0.5(3x3), PSNR : 28.64dB
gaussian filter, sigma=1(7x7), PSNR : 30.30dB
gaussian filter, sigma=3(19x19), PSNR : 25.01dB
```

Gaussian filter를 적용한 결과 표준 편차를 0.5로 두었을 때 보다 1로 두었을 때 PSNR값이 더 오른 것을 확인할 수 있었으며, 표준 편차를 3으로 두었을 때에는 육안으로도 영상이 흐려진 것을 확인할 수 있었고, PSNR값도 filtering 하지 않은 잡음 영상보다 낮아진 것을 확인할 수 있었다. 따라서 적절한 표준 편차를 선택하고, 그에 따른 filter size를 결정하는 것이 중요하다는 것을 알 수 있었다.

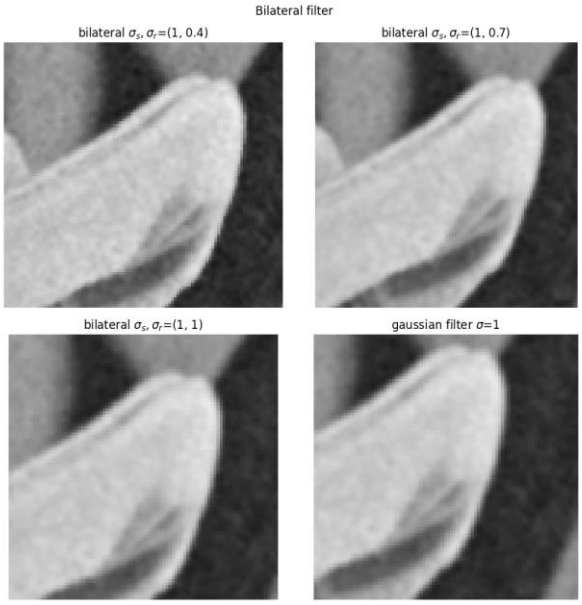
b) Bilateral filter

Bilateral filter는 픽셀값의 차이로 이루어지는 intensity term과 픽셀 간의 거리 차이로 이루어지는 Gaussian filter인 spatial term으로 이루어져 있다. Bilateral filter은 intensity term으로 인해 픽셀 값이 많이 차이나는 곳은 filter의 영향을 받지 못하여, 에지와 같이 픽셀 값의 차이가 많이 나는 부분을 보존할 수 있는 것이 장점이다. 하지만 intensity term또한 픽셀마다 계산하여야 하기 때문에 계산 시간이 오래걸리는 것이 단점이다.

$$\hat{I}(\mathrm{x}) = \frac{1}{K}\sum_{y} \mathrm{e}^{-\frac{\left\|I(y)-I(x)\right\|^2}{2\sigma_r^2}} \mathrm{e}^{-\frac{\left\|y-x\right\|^2}{2\sigma_d^2}} I(y)$$




```
1.3.b) Bilateral filter
bilateral filter, sigma_r,sigma_s=(1, 0.4), PSNR : 31.43dB, 18.332sec
bilateral filter, sigma_r,sigma_s=(1, 0.7), PSNR : 30.88dB, 18.277sec
bilateral filter, sigma_r,sigma_s=(1, 1), PSNR : 30.59dB, 18.212sec
```



Gaussian 표준 편차(σ_s)의 경우 Gaussian filter에서 가장 좋은 값을 가진 1을 사용하였다. Bilateral filter(σ_r)의 경우 range term(intensity term)의 표준 편차가 작을 때 PSNR이 가장 크게 나온 것을 확인할 수 있었다. PSNR 또한 모든 결과에서 소폭 상승한 것을 확인할 수 있었으며, $\sigma_r = 0.4$일 때 가장 좋은 결과를 얻을 수 있었다. 모자 부분을 확대해 본 결과 그냥 Gaussian filter만 사용하였을 때에는 에지 부분이 흐려지는 현상이 있었는데 $\sigma_r = 0.4$일 때에는 에지가 거의 남아있는 것을 확인할 수 있었다. 하지만, Gaussian filter를 사용할 때에는 0.1초 미만으로 수행할 수 있었지만, bilateral filter를 사용할 때에는 18초 이상의 시간이 소요되었으므로, 실시간 영상 처리나 동영상의 처리 등의 속도가 중요한 영역에서는 사용하기 힘들 것으로 판단된다.

c) Average filter

Average filter는 평균을 이용하는 filter로 mask 행렬의 모든 요소가 같은 값을 가지는 filter로, 주위 픽셀의 평균 값을 나타내는 filter이다.

$$M = \frac{1}{K}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \qquad \Sigma M = 1$$

```
average filter(3x3), PSNR : 29.99dB
average filter(5x5), PSNR : 27.88dB
average filter(7x7), PSNR : 26.15dB
```

Average filter의 경우 filter size가 커질수록 PSNR이 줄어드는 것을 확인할 수 있었으며, 7x7 크기의 mask를 사용하였을 때에는 blocking 현상이 일어나는 것도 확인할 수 있었다. Average filter는 주변의 평균값을 나타내는 특성상 impulse noise가 적용된 영상에 유용할 것이라 생각된다.

2. Image demosaicking

2.1. Design an algorithm to implement the CFA patterns.

작성한 mosaick 함수는 다음과 같다. RGB 이미지와 단위 CFA 행렬(R:0, G:1, B:2)을 입력하면, 단위 CFA 행렬의 크기로 나눈 나머지 이미지를 단위 CFA 행렬의 R이 있는 부분은 R를, G가 있는 부분은 G를, B가 있는 부분은 B를 입력한다.

```
def mosaick(rgb, cfa):
    # intput : rgb image (MxNx3 array), cfa array #
    # output : mosaicked image
    mo = np.zeros(rgb.shape,dtype=np.uint8)
    [y, x] = cfa.shape
    for i in range(y):
        for j in range(x):
            mo[i::y, j::x, cfa[i,j]] = rgb[i::y, j::x, cfa[i,j]]
    return mo
```

CFA 패턴으로 모자이킹을 수행한 결과는 다음과 같다.

(a), (b), (c), (f)의 경우 G의 pixel이 많아 전체적으로 초록빛을 띄는 것을 확인할 수 있고, (d), (e)의 경우엔 모두 1:1:1의 비율로 CFA 패턴이 이루어져 있기 때문에 밝기만 줄어든 것을 확인할 수 있었다.

2.2. Build demosaicking algorithm

Demosaicking은 다음과 같이 수행하였다. 먼저 filter를 만들어주는 함수에 bilinear interpolation 계산을 할 수 있는 filter를 추가로 작성하였다. Bilinear interpolation filter는 interpolation 계산을 하려고 하는 요소가 있을 수 있는 최대 거리 par1(y방향), par2(x방향)이 있다고 하면 filter의 크기는 $(2 \times \mathrm{par1} - 1) \times (2 \times \mathrm{par2} - 1)$이 된다. 여기서 거리에 따른 bilinear interpolation을 수행하므로 filter의, y 방향으로는 $\left[\frac{1}{\mathrm{par1}}, \frac{2}{\mathrm{par1}}, \dots, 1, \dots, \frac{2}{\mathrm{par1}}, \frac{1}{\mathrm{par1}}\right]$, x 방향으로는 $\left[\frac{1}{\mathrm{par2}}, \frac{2}{\mathrm{par2}}, \dots, 1, \dots, \frac{2}{\mathrm{par2}}, \frac{1}{\mathrm{par2}}\right]$의 크기를 가지게 된다. 따라서 meshgrid를 사용하여 filter를 작성할 수 있다. 예를 들면 크기에 따라 다음과 같은 mask를 가지게 된다.

$$\mathrm{M} = \begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & 1 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{pmatrix}, \qquad \mathrm{M} = \begin{pmatrix} 0.1111 & 0.2222 & 0.3333 & 0.2222 & 0.1111 \\ 0.2222 & 0.4444 & 0.6667 & 0.4444 & 0.2222 \\ 0.3333 & 0.6667 & 1 & 0.6667 & 0.3333 \\ 0.2222 & 0.4444 & 0.6667 & 0.4444 & 0.2222 \\ 0.1111 & 0.2222 & 0.3333 & 0.2222 & 0.1111 \end{pmatrix},$$

```
def filter_gen(par1, par2=1, type='gaussian', norm=1):
...
...
    elif type=='bilinear':
        # bilinear interpolation filter
        # size = (2 x par1 - 1) x (2 x par2 - 1)
        ar1 = np.arange(1, par1)/par1
        ar1 = np.hstack((ar1,(1), np.flip(ar1)))
        ar2 = np.arange(1, par2)/par2
        ar2 = np.hstack((ar2,(1), np.flip(ar2)))
        [y, x] = np.meshgrid(ar2,ar1)
        fil = x*y
...
...
    return fil
```

Bilinear filter를 이용하여, demosaick를 하는 방법은 다음과 같다. Bilinear interpolation의 경우 interpolation을 수행하는 점의 위치에 있는 mask의 요소의 합이 항상 1이 되기 때문에 mask를 normalize할 필요가 없지만, demosaick의 경우 단위 CFA 행렬에 색 요소가 여러 번 들어가는 경우가 있다. 따라서 R, G, B마다 다르게 mask를 normalize 하여하여 한다. 먼저 cfa를 3차원(CFAx3)으로 늘린 cfa_행렬을 생성하여, cfa_[:,:,n]에 R, G, B가 있는 부분을 1로 표시한다, 그 후 mask가 cfa_보다 y방향, x방향으로 크기 때문에 cfa_를 y, x 방향으로 stack한 후 mask의 크기와 같게 잘라낸다. 그 후 1로 이루어진 cfa_를 filtering하여 그 결과의 크기만큼 mask를 normalize 해 주었다. 그 후 각각 R, G, B 색상에 적용하여 bilinear interpolation을 하면, demosaick된 결과를 얻을 수 있다.

```
def demosaick(mo, cfa):
    [y, x] = cfa.shape
    cfa_ = np.zeros((y,x,3))
    cfa_[:,:,0][cfa==0] = 1
    cfa_[:,:,1][cfa==1] = 1
    cfa_[:,:,2][cfa==2] = 1
    cfa_ = np.vstack((cfa_, cfa_))
    cfa_ = np.hstack((cfa_, cfa_))
    result = np.zeros(mo.shape)
    fil = filter_gen(y, x, type='bilinear')
    [y, x] = fil.shape
    result[:,:,0] = apply_filter2(mo[:,:,0], fil/np.sum(cfa_[:y,:x,0]*fil))
    result[:,:,1] = apply_filter2(mo[:,:,1], fil/np.sum(cfa_[:y,:x,1]*fil))
    result[:,:,2] = apply_filter2(mo[:,:,2], fil/np.sum(cfa_[:y,:x,2]*fil))
    return result.astype(np.uint8)
```

2.3. Compute PSNR and analyze.

단위 CFA 패턴의 크기가 (a)~(f)가 각각 2x2, 2x4, 4x2, 2x3, 2x3, 4x4였으므로, 단위 CFA 패턴의 크기가 클 경우 mask의 크기가 커져 영상 품질의 저하가 있을 것으로 예상할 수 있다. 앞에서 mosaick한 영상을 demosaick한 결과는 다음과 같다.


```
Demosaick (a), PSNR : 31.21dB
Demosaick (b), PSNR : 28.88dB
Demosaick (c), PSNR : 29.81dB
Demosaick (d), PSNR : 29.32dB
Demosaick (e), PSNR : 30.27dB
Demosaick (f), PSNR : 27.84dB
```

(a)의 경우 mask의 크기가 작아 깔끔한 영상을 확인할 수 있었고, (f)의 경우 mask의 크기가 커져 약간의 blur가 있는 것을 확인할 수 있었다. 하지만 가장 품질이 좋은 (a)와 가장 품질이 나쁜 (f)를 제외하면, (b)~(e)는 육안으로 품질의 높고 낮음을 구분하기 어려웠다. PSNR 또한 (a)의 경우가 가장 큰 값을 가졌고, 두번째로 mask 크기가 작은 (e)가 두번째로 큰 PSNR값을 가지는 것을 확인할 수 있었다. 단위 CFA 행렬의 크기가 가장 큰 (f)가 가장 작은 PSNR 값을 가졌다.

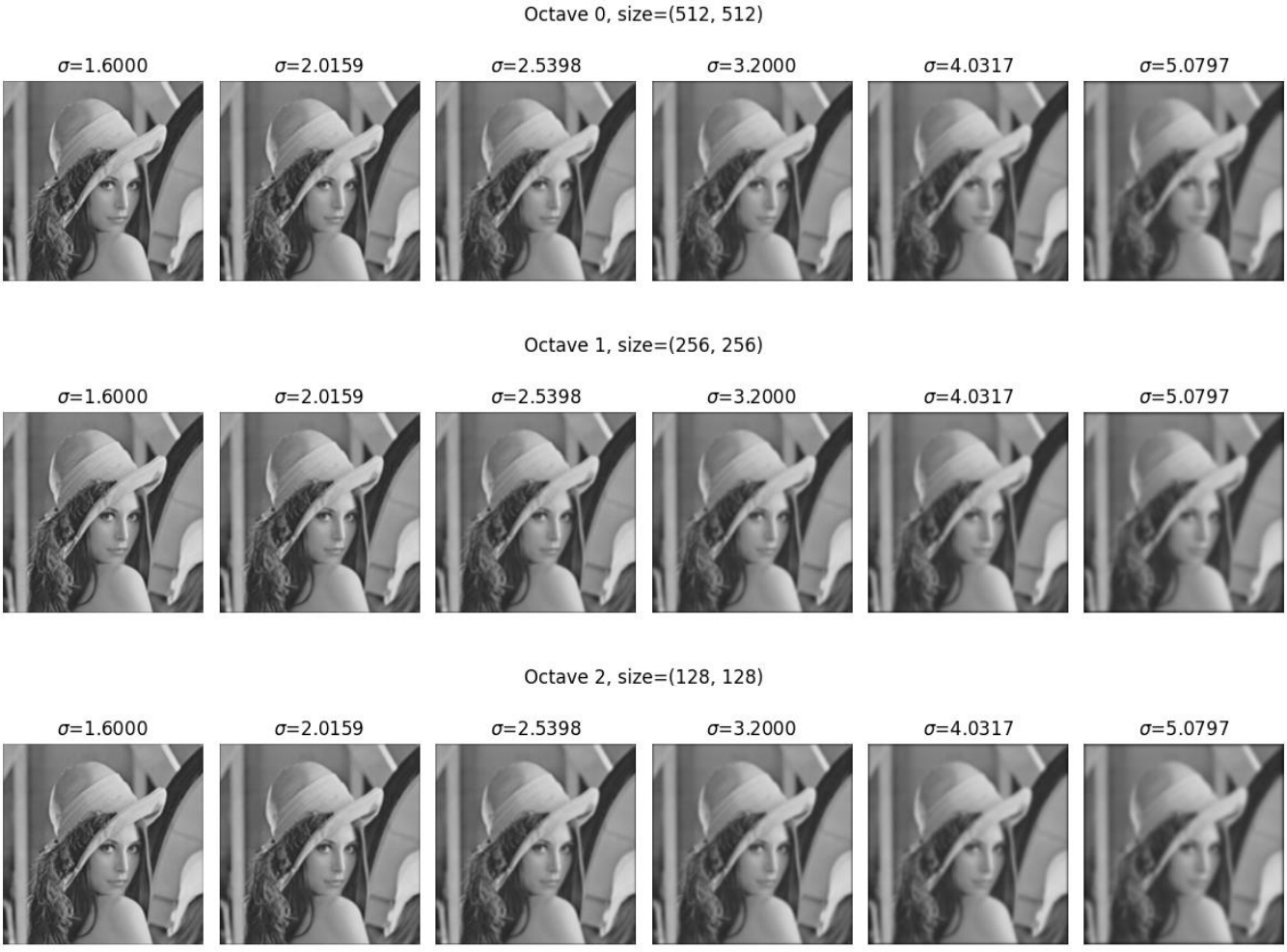
3. Local Descriptor

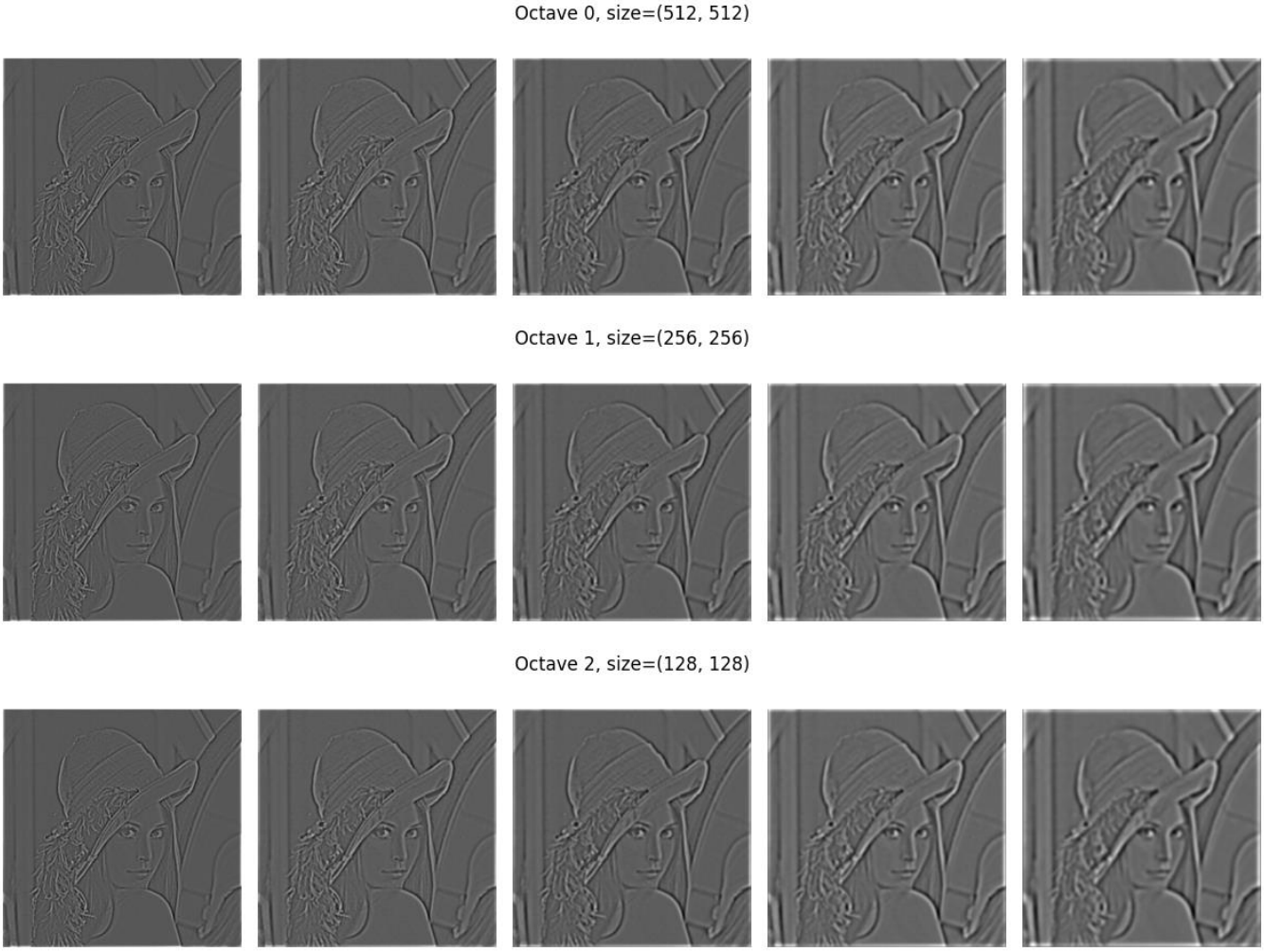
3.1) Implement the SIFT image pyramid generation function.

SIFT는 영상의 크기와 회전에 무관한 keypoint를 검출하는 방법으로, LoG와 유사하지만 계산속도가 빠른 DoG 맵을 이용하여 keypoint를 검출한다. DoG 맵은 같은 원본 영상을 σ값을 k배 증가시킨 Gaussian을 취하는 것을 반복하여 filtering된 영상으로 이루어진 Octave와, 원본 영상을 1/2로 축소시키며 Gaussian filtering을 반복하고, 다시 1/2 축소시켜 반복한다. 각 Octave에서의 인접 영상을 서로 빼면 DoG를 얻을 수 있고 그 DoG맵에서 인접한 28개의 값(9+8+8)에 비해 극점인 값을 keypoint로 검출한다.

SIFT pyramid를 얻기 위해 먼저 σ_i값을 구하였다. $\sigma_{i+1} = k\sigma_i$이지만, 원본 영상을 $k\sigma_{i-1}$ 로 filtering하는 것과, 이전에 σ_{i-1}로 filtering된 영상을 $\sigma = \sqrt{\sigma_i^2 - \sigma_{i-1}^2}$로 filtering하는 것이 같기 때문에 $\sigma = \sqrt{\sigma_i^2 - \sigma_{i-1}^2}$를 이용하여 mask size를 줄여 빠르게 계산할 수 있는 방법으로 계산하였다. 앞서 작성한 filter_generation 함수를 이용하여 3차원(y,x,i) Octave로 이루어진 pyramid list를 얻었다. 함수와 얻은 pyramid와 DoG를 plot한 결과는 다음과 같다.

```
def sift_pyramid_gen(img, sigma=1.6, k=6, o=3):
    ### SIFT pyramid generation ###
    # input : img : M x N image, th : threshold, sigma : scaling factor
    #         k : number of images, o : number of octave
    # output : pyramid(list of octaves), ( ex) pyramid[0] = 0 octave(M x N x k) )
    pyramid = list()
    for i in range(o):
        mx = 0
        img_o = downscale(img, 2**i)
        img_f = np.zeros([img_o.shape[0], img_o.shape[1], k])
        for n in range(k):
            if n==0 : sig = np.sqrt(sigma**2 - 0.5**2)
            else : sig = np.sqrt((sigma*2**(1/3*n))**2 - (sigma*2**(1/3*(n-1)))**2)
            size = sig*6//2*2+1
            fil = filter_gen(size, sig)
            if n==0 : img_f[:,:,n] = apply_filter2(img_o, fil, i=0)
            else : img_f[:,:,n] = apply_filter2(img_f[:,:,n-1], fil, i=0)
        pyramid.append(img_f)
    return pyramid
```





3.2) Verify that response of DoG is almost similar to it of LoG.

$$LoG \;=\; \nabla^2 G = \left(\frac{\partial^2 G}{\partial y^2} + \frac{\partial^2 G}{\partial x^2}\right) = \frac{\partial^2}{\partial y^2}\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2+x^2}{2\sigma^2}}\right) + \frac{\partial^2}{\partial x^2}\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2+x^2}{2\sigma^2}}\right)$$

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

$$DoG \;= \frac{\partial G}{\partial \sigma} = \frac{\partial}{\partial \sigma}\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2+x^2}{2\sigma^2}}\right) = \left(-\frac{2}{\sigma} + \frac{y^2+x^2}{\sigma^3}\right)\frac{1}{2\pi\sigma^2} e^{-\frac{y^2+x^2}{2\sigma^2}} = \sigma \times LoG$$

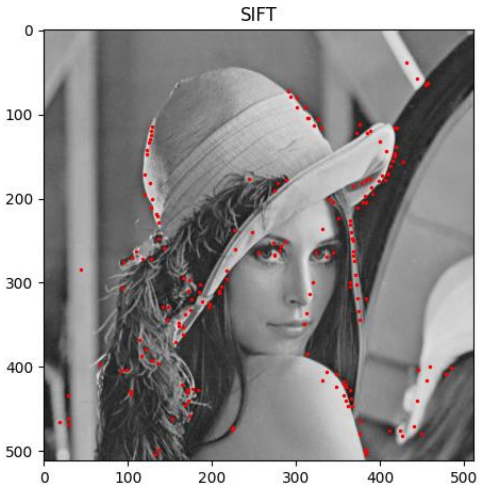
LoG와 DoG를 전개하면 다음과 같다. 따라서 정규화 하게되면 LoG≅ DoG이라고 할 수 있다.

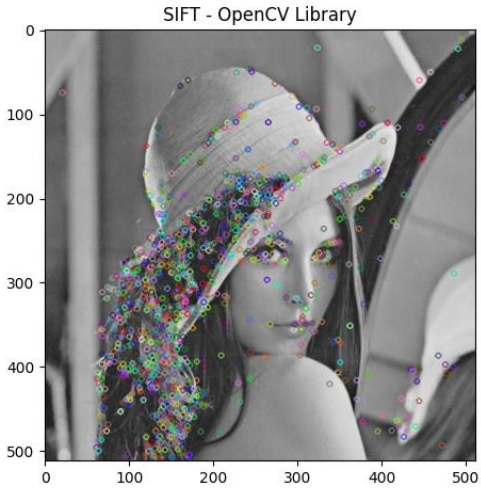
3.3) Detect key points using generated image pyramid and check whether its valid.

위에서 얻은 pyramid를 통해 DoG맵을 구한 후, 근접한 28개의 값 중, 극값을 구하면 key-point를 구할 수 있다. 극값을 구하는 작업 또한 위 filtering에서 사용한 방법처럼 행렬을 이용하였다. Indexing을 이용하여 주위 값을 3번째 차원으로 쌓아올려, dogn($y \times x \times 27$) 행렬을 만들었다. dogn행렬에 절대값을 취하고, 정규화한 하였다. 바로 최댓값을 이용하면, dogn[:,:,0]의 값이 최댓값이어도 다른 27개의 값 중 같은 값이 있을 수 있으므로, dogn[:,:,1:]의 최댓값과 dogn[:,:,:]의 최댓값이 다른 경우에 keypoint 행렬로 편입시켰다. 이 때 y, x좌표는 Octave에 따라 scaling된 값이므로, 2^o를 곱해주었다. Threshold는 전체 DoG의 최댓값의 일정% 이상으로 설정하였다. 검출된 keypoint의 좌표는 [y, x, octave, index]이다.

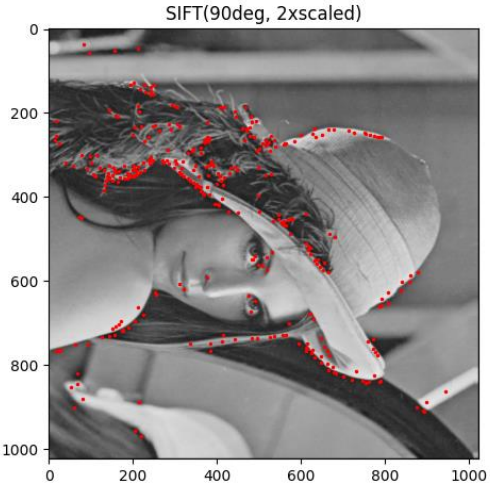
```
def sift_kp(pyramid, th):
    ### keypoint detection from sift pyramid ###
    # input : pyramid(list of octaves), threshold
    # output : keypoint(y, x, o, i)
    mx = [5, 7, 9, 11, 15, 19]
    kp = np.array([[0, 0, 0, 0]])
    dog = list()
    for o in range(len(pyramid)):
        gaussian = pyramid[o]
        dog.append(gaussian[:,:,:-1] - gaussian[:,:,1:])
        #kp = np.array([[0, 0, 0, 0, 0]])
        for n in range(1,4):
            [y, x] = dog[o].shape[0:2]
            off = mx[n]
            dogn = dog[o][off:y-off,off:x-off,[n,n-1,n+1]]
            for i in [0, -1, 1]:
                for j in [0, -1, 1]:
                    dogn = np.append(dogn, dog[o][off+i:y-off+i,off+j:x-off+j,[n,n-1,n+1]], axis=2)
            dogn = dogn[:,:,3:]
            dogn = abs(dogn)
            dogn = dogn/np.max(dogn)
            pos = np.where((np.max(dogn[:,:,1:], axis=2)!=np.max(dogn, axis=2))*(dogn[:,:,0]>th))
            kp_n = np.transpose([(pos[0]+off)*2**o, (pos[1]+off)*2**o])
            kp_n = np.pad(kp_n, ((0,0),(0,1)), constant_values=o)
            kp_n = np.pad(kp_n, ((0,0),(0,1)), constant_values=i)
            kp = np.append(kp, kp_n, axis=0)
    return kp[1:,:], dog
```

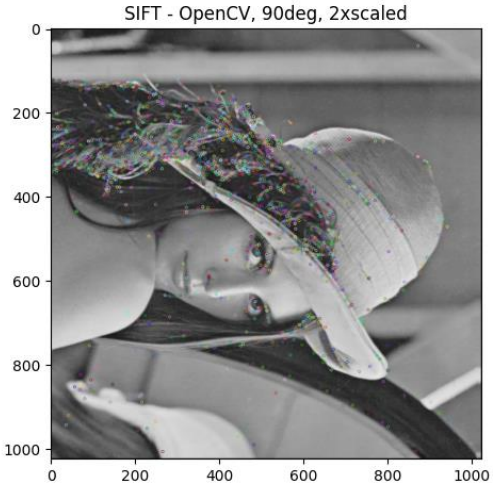
검출된 keypoint를 plot하면 다음과 같다. 좌측이 위 함수를 사용한 결과이고, 우측이 OpenCV library를 사용한 결과이다. 머리카락 부분과 모자, 어깨 부분에 keypoint가 검출된 것을 확인할 수 있다. 작성된 함수가 OpenCV libraray와 비교해도 keypoint가 잘 검출되는 것을 확인할 수 있다.



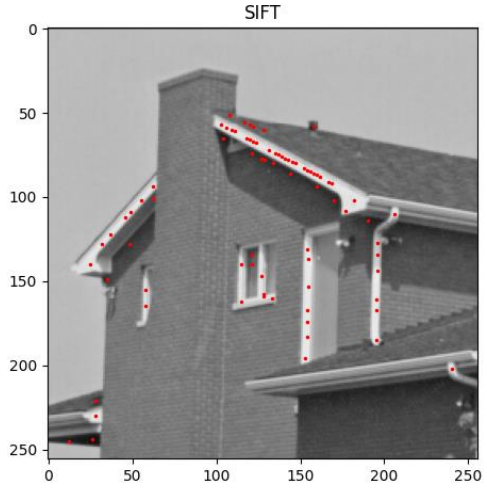


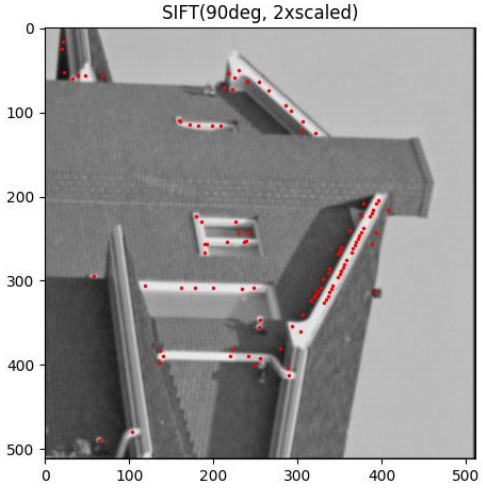
다음은 원본 영상을 2배 확대한 후, 90° 회전시킨 영상을 SIFT 알고리즘으로 keypoint를 검출한 결과이다. 여전히 머리카락과 모자, 어깨 부분에 keypoint가 검출된 것을 확인할 수 있으며, 결과적으로 검출된 keypoint가 유효하다는 것을 알 수 있다.

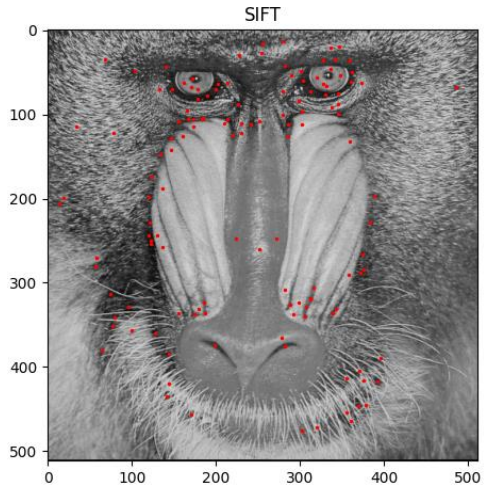


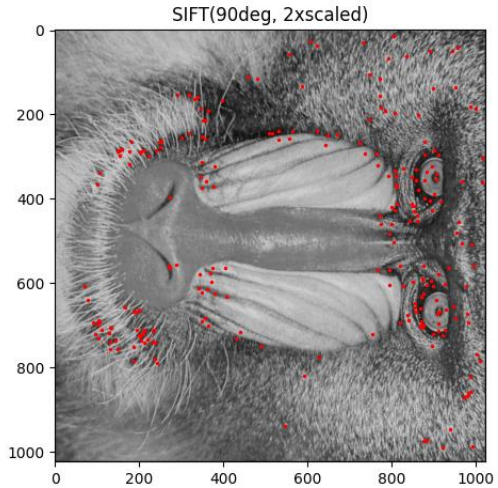


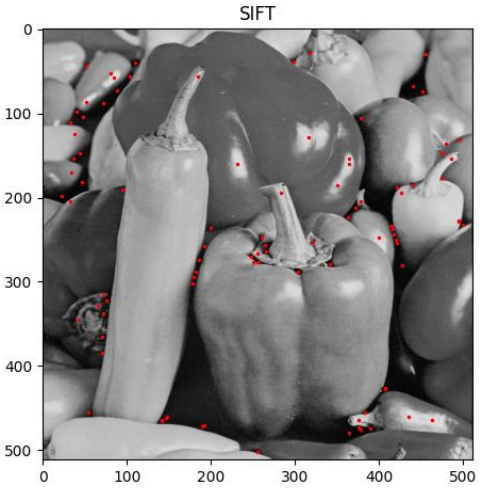
'house'의 경우에는 지붕과, 창문에 keypoint를 확인할 수 있었으며, 'babbon'의 경우 눈과 입 주변에서 keypoint를 확인할 수 있었다. 'pepper'에서는 pepper 꼭찌 부분에서 keypoint를 찾았다. 다른 영상에서도 마찬가지로 2x scaling과 90° rotation 하여도 keypoint가 대다수 검출되었으므로, SIFT 알고리즘은 scale, rotation invariant 하다는 것을 알 수 있었다.

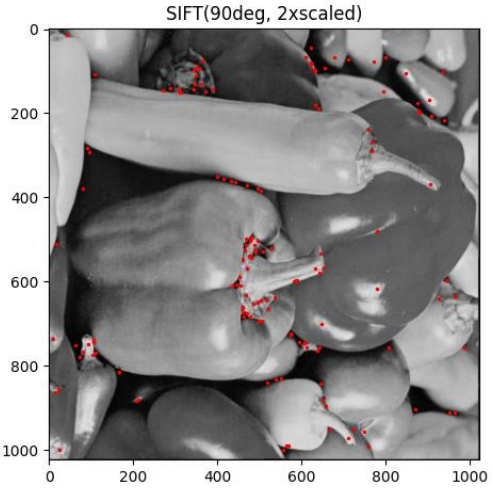









SIFT
0
100
200
300
400
500
0
100
200
300
400
500


SIFT(90deg, 2xscaled)
0
200
400
600
800
1000
0
200
400
600
800
1000