1. (Image denoising) Please implement the followings and analyze the results.

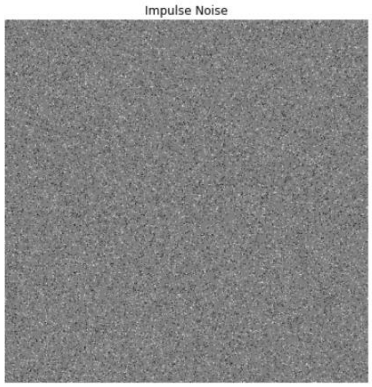
1) Generate the following noises, and add them to the ground-truth images.

a) Impulse noise

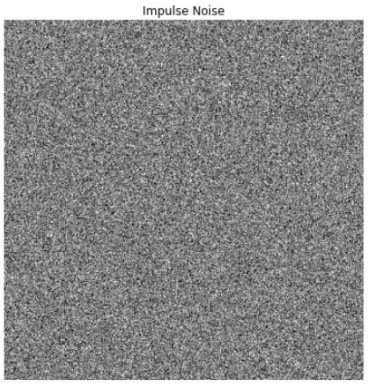
ground truth이미지와 같은 크기의 zero 이미지를 생성하고 0~1사이의 값을 uniform distribution을 갖는 랜덤변수를 통해 얻어내서 그 입력으로 넣어준 noise Percent에 따라 반은 -255, 반은 255의 값을 갖게 하였다. 이후 noise를 이미지에 더하는 과정에서 np.clip()을 사용해 uint8 자료형이 나타낼 수 있는 범위를 넘어가는 픽셀 값을 갖는 경우 0, 255의 값을 가질 수 있게 했다. 아래 사진에서 p=0.1, p=0.3 노이즈를 생성했는데 noise Percent값에 따라 품질이 더 나빠진 모습을 눈으로도 확연하게 관찰할 수 있었다.

```
def ImpulseNoiseImage(M,N,p):
    noise_img = np.zeros((M,N))
    for i in range(M):
        for j in range(N):
            t = np.random.uniform(0,1)
            if t < p/2 :
                noise_img[i][j] = -255
            elif t >= p/2 and t<p:
                noise_img[i][j] = 255
    return noise_img

# noise Image
I_noise = ImpulseNoiseImage(h,w,0.3)
# noise Added Imasge
INoiseAddedImage = I_noise + img
INoiseAddedImage = np.clip(INoiseAddedImage,0,255)
```



<p=0.1>

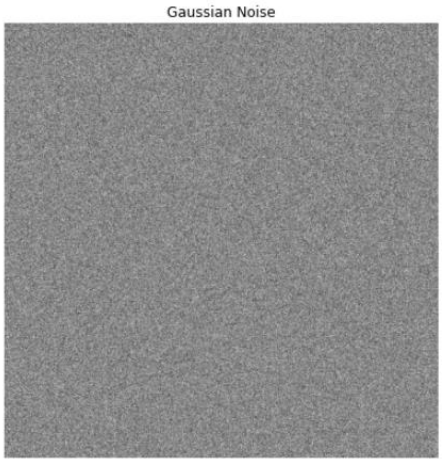


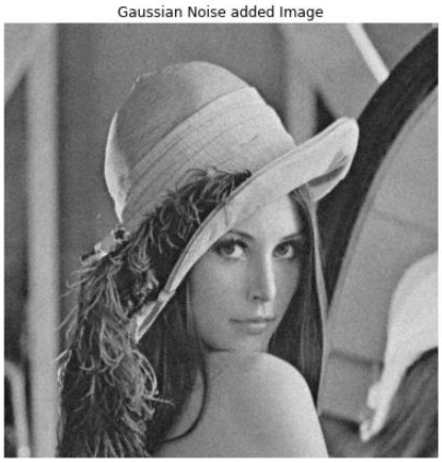
<p=0.3>

b) Gaussian noise

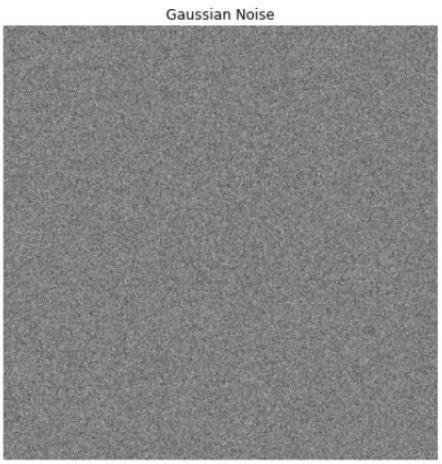
Gaussian distribution을 갖는 랜덤 변수를 통해 원하는 사이즈와 분산을 가지는 noise 이미지를 생성할 수 있다. 분산 값을 더 크게 넣어주는 경우 이미지의 품질이 더 나빠지는 것을 확인할 수 있다.

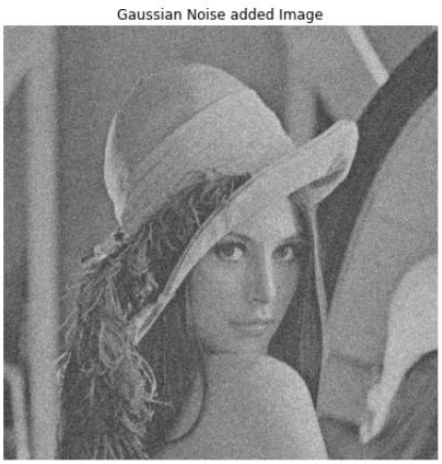
```
def GaussianNoiseImage(M,N,std):
    noise_img = np.random.normal(0, std, (M,N))  # mean, std, shape
    return noise_img
```





<std = 10>





<std=30>

2) Compute PSNR between ground-truth and noisy image.

PSNR값이 클수록 좋은 품질의 이미지, 노이즈가 적게 포함되어 있다고 할 수 있는데 예상한 대로 노이즈의 크기를 크게 해줄수록 PSNR값이 낮게 나온 것을 확인할 수 있었다.

```
def PSNR(img1, img2):
    mse = ((img1-img2)**2).mean()
    psnr = 10*math.log10(255**2 / mse )
    return psnr
```

```
GNoise(std=30) Added Image's PSNR:  18.595937357404242
GNoise(std=10) Added Image's PSNR:  28.125772169846506
INoise(p = 0.3)  Added Image's PSNR:  10.689400614592762
INoise(p = 0.1) Added Image's PSNR:  15.48157892821501
```

3) Implement the following denoising filters, and apply them for denoising

4) Please compare and analyze the results with subjective and objective (PSNR) measurements.

먼저 Filtering을 위해 공통적으로 사용되는 filtering함수를 구현했다.

```
def filtering(img, filter):
    h, w = img.shape
    fSize = len(filter)        # filter Size
    pSize = (fSize-1) // 2     # padding Size

    img = np.pad(img, ((pSize,pSize),(pSize,pSize)), 'constant', constant_values=0)
    filteredImg = np.zeros((h,w))

    for i in range(pSize,h+pSize):       # operate on ground-truth pixel
        for j in range(pSize,w+pSize):
            product = img[i-pSize:i+pSize+1,j-pSize:j+pSize+1] * filter
            filteredImg[i-pSize][j-pSize] = product.sum()

    return filteredImg
```

a) Gaussian filter

Gaussian distribution을 가지는 kernel을 생성해서 전체 이미지에 대해 같은 kernel을 사용해 filtering을 수행한다.

```
def gaussianFilter(size, std):
    gfilter = np.zeros((size,size))
    offset = size // 2
    for i in range(-offset,offset+1):
        for j in range(-offset,offset+1):
            gfilter[i+offset][j+offset] = math.exp( -1 * ( i**2 + j**2 ) / (2*(std**2)) )
    return gfilter / gfilter.sum()
```

Ground truth이미지에 std=30을 갖는 노이즈를 생성해 더해준 이미지에 대해 denoising을 진행하였다.

Kernel size과 std는 3~7까지 변화시키며 PSNR값을 관찰했다. Size를 증가시키거나 std를 증가시킨다고 해서 PSNR값이 비례해 증가하진 않았다는 점에서 이미지에 따라 특정 파라미터를 갖는 filter에서 좋은 성능을 가진다는 것을 짐작할 수 있다.

Gaussian Filteed Image VS Ground-Truth: 26.68528163348016 Gaussian Filteed Image VS Ground-Truth: 26.66279144134481

Gaussian Filteed Image VS Ground-Truth: 27.19681994610204 Gaussian Filteed Image VS Ground-Truth: 27.033047190839405

Gaussian Filteed Image VS Ground-Truth: 26.42779431438937 Gaussian Filteed Image VS Ground-Truth: 26.051827984550734

b) Bilateral filter

Gaussian filter의 단점으로 filtering을 진행하면서 edge가 blur되는 것을 뽑을 수 있는데 이를 개선에서 만들어낸 filter가 Bilateral filter이며 비선형적 특징을 가진다.

bilateral filter는 픽셀 거리 뿐 아니라 픽셀 값까지 필터에 반영해서 필터를 만든다. 따라서 Gaussian filter와 가장 큰 차이점으로 각 픽셀마다 다른 값을 가지는 filter가 생성되고 이를 통해 filtering을 진행한다. 매 픽셀마다 새로운 필터를 만들어서 filtering을 진행하므로 Gaussian filter에 비해 시간이 오래 걸리는 단점이 있지만 뛰어난 성능을 시각적으로, PSNR 수치로도 확인할 수 있다.

```
def bilateralFilter(image, x, y, size, std):   # 이미지, 픽셀위치, 필터 크기, 분산
    kernel = np.zeros((size,size),dtype='float')
    offset = size // 2
    for i in range(-offset, offset+1):
        for j in range(-offset, offset+1):
            kernel[i+offset][j+offset] =  math.exp( -1 * (image[x][y]-image[x+i][y+j])**2 / ( 2* std**2) ) * #
                                          math.exp( -1 * ( i**2 + j**2 ) / ( 2 * std**2) )
    return kernel / kernel.sum()

# padding
kernel_size=5
pSize= kernel_size //2
pimg = np.pad(img, ((pSize,pSize),(pSize,pSize)), 'constant', constant_values=0)

# 이미지의 각 픽셀에 대한 커널 생성 및 filtering
BFilteredImage = np.zeros((h,w))
for i in range(pSize,h+pSize):
    for j in range(pSize,w+pSize):
        BFilter = bilateralFilter(pimg,i,j,5,5)
        product = pimg[i-pSize:i+pSize+1,j-pSize:j+pSize+1] * BFilter
        BFilteredImage[i-pSize][j-pSize] = product.sum()
```

PSNR값이 Gaussian filtering을 진행했을 때에 비해 상당히 높게 나왔으며 size=3, std=3를 parameter로 넣어주었을 때 가장 좋은 성능을 보였다.

Bilateral Filteed Image VS Ground-Truth: 46.793672991835756 Bilateral Filteed Image VS Ground-Truth: 42.65805069632667

Bilateral Filteed Image VS Ground-Truth: 45.05103364720085 Bilateral Filteed Image VS Ground-Truth: 40.78087502510932

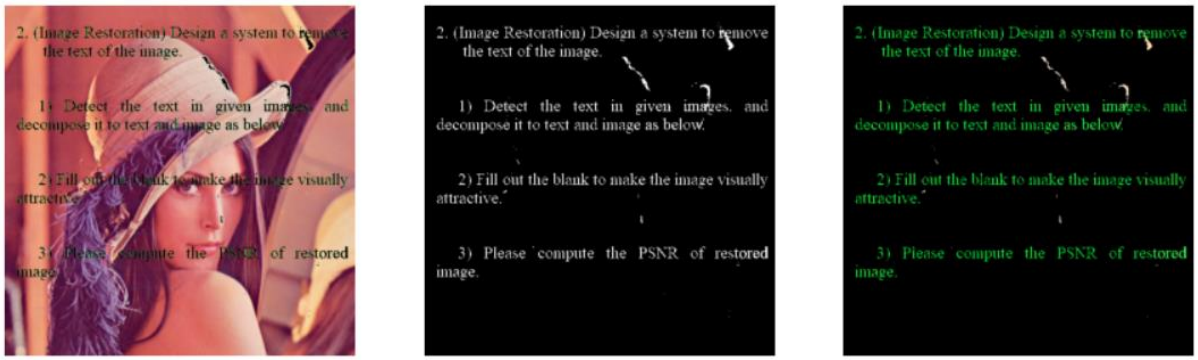
2. (Image Restoration) Design a system to remove the text of the image

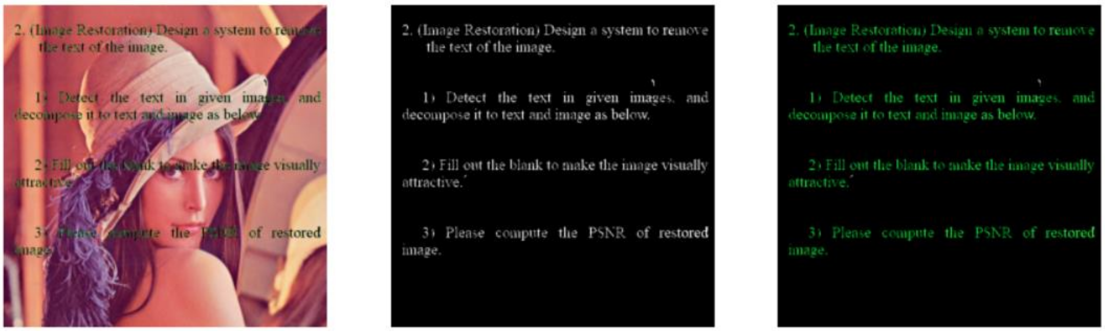
1) Detect the text in given images, and decompose them to the text and image.

Text와 image를 분리하기 위해 이미지를 관찰했을 때 가장 쉽게 찾을 수 있는 특징은 색상이었다. 따라서 이미지의 Green 색영역을 이진화해서 text를 분리하기 위한 mask를 생성했다.

순서대로 Green channel의 threshold를 각각 220, 230으로 이진화 한 결과이다. 220으로 이진화 한 결과 텍스트 영역이외 다른 영역의 공간까지 검출하는 단점이 있었고 230까지 threshold를 올리면 그 현상은 줄었지만 두 가지 모두 공통적으로 초록색 글씨를 완벽하게는 검출하지 못했다.

```
threshold = 230
text_mask = img[:,:,1].copy()
text_mask[text_mask[:,:]>threshold] = 255
text_mask[text_mask[:,:]<threshold] = 0
```





Threshold를 230까지 올린 뒤에 초록색 글씨를 완벽하게 분리하지 못하는 문제를 해결하기 위해 마스크에 모폴로지 연산 중 팽창연산을 적용한 뒤 다시 분리를 진행했다. 모폴로지에 사용한 커널은 3x3크기를 사용했다. 결과를 확인하면 마스크의 범위가 증가하면서 초록색 글씨를 보다 정확하게 검출하는 것을 확인할 수 있다.

```
def morphology(img, k): # dilate
    h,w = img.shape
    newimg = np.zeros((h,w),dtype='uint8')
    offset = len(k) // 2
    for i in range(offset, h-offset):
        for j in range(offset, w-offset):
            if(img[i][j]==255):
                newimg[i-offset:i+offset+1,j-offset:j+offset+1] = np.bitwise_or(img[i-offset:i+offset+1, \
                                                                    j-offset:j+offset+1], k)
    return newimg
```

다음은 OpenCV를 사용해 모폴로지연산, 마스킹을 수행한 결과이다. 유사한 것을 확인 가능하다.

```
k = cv2.getStructuringElement(cv2.MORPH_RECT, (3,3))
text_mask = cv2.dilate(text_mask, k)
img_result = cv2.bitwise_and(img, img, mask = ~text_mask)
```

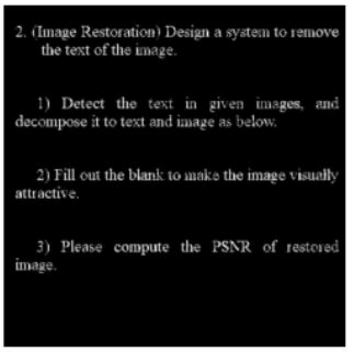
다음은 마스킹을 위해 직접 작성한 코드이다. 마스크의 값을 1과 0만 가지게 한 뒤에 각 원소끼리 곱했다. 곱은 각 채널에 대해 각각 수행했다.

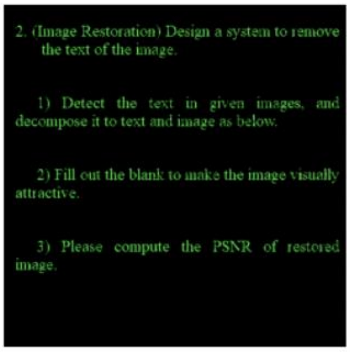
```
h,w = img.shape[:2]
img_result = np.zeros((h,w,3), dtype='uint8')
text = np.zeros((h,w,3), dtype='uint8')
img_result[:,:,0] = np.multiply(img[:,:,0], ~text_mask // 255)
img_result[:,:,1] = np.multiply(img[:,:,1], ~text_mask // 255)
img_result[:,:,2] = np.multiply(img[:,:,2], ~text_mask // 255)
text[:,:,0] = np.multiply(img[:,:,0],text_mask // 255)
text[:,:,1] = np.multiply(img[:,:,1],text_mask // 255)
text[:,:,2] = np.multiply(img[:,:,2],text_mask // 255)
```

이미지의 색을 다루는 공간으로 RGB외에도 HSV 공간이 널리 사용된다는 사실을 알아냈다.

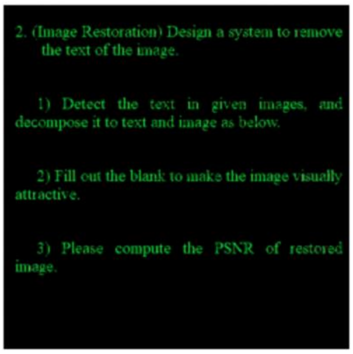
HSV공간에 대해 간단하게 적으면 H(Hue,색상)는 0~360의 범위를 가지고, 가장 파장이 긴 빨간색을 0으로 나타낸다. S(saturation, 채도)는 0~100의 범위를 가지고, 색상이 진한 상태를 100으로 나타내며 진함의 정도를 나타낸다. V(Value, 명도)는 0~100의 범위를 가지고 흰색을 100 검은색을 0으로 밝기를 나타낸다. HSV 공간에서 색상을 보다 직관적으로 분리할 수 있기 때문에 색상 검출에 보다 용이하게 사용된다고 한다.

```
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
lower_blue = (60-10, 30, 30)
upper_blue = (60+10, 255, 255)
img_mask = cv2.inRange(img_hsv, lower_blue, upper_blue)
text = cv2.bitwise_and(img, img, mask = img_mask)
img_result = cv2.bitwise_and(img, img, mask = ~img_mask)
```





분리 결과를 RGB 공간을 통해 분리한 결과와 비교하면 확실히 text외에 다른 영역에서 마스크가 생성되는 부분이 없어진 것을 확인할 수 있지만 text가 제거된 원본 이미지를 확인하면 여전히 초록색 부분이 남아 있는 것을 확인할 수 있다. 따라서 HSV공간을 통해 분리한 이미지에 대해서 동일하게 모폴로지 팽창연산을 진행했다.



분리 결과를 확인하면 초록색 부분이 거의 모두 검출된 것을 확인할 수 있다.

2) Fill out the blank to make the image visually attractive.

글씨를 분리해낸 이미지에 대해 restoring을 진행한다. 복구해야 하는 픽셀을 발견하면 box kernel 혹은 gaussian kernel을 사용해 주변 픽셀 값들을 사용해서 중심에 존재하는 픽셀 값을 찾을 수 있도록 했다. 이때 일반적으로 사용하는 이중 반복문을 통해 순회하며 hole을 찾게 되면 위쪽에 존재하는 픽셀 값에 의해 hole들의 픽셀 값이 편향될 수 있다는 생각이 들었다. 따라서 hole을 발견하면 가장자리부터 시작해서 중심으로 값을 복구해 나가는 것을 idea로 재귀 함수를 통해 Restoring을 구현했다. Hole을 발견하면 해당 위치의 주변, 0이 아닌 픽셀 값을 적절히 조합해서 그 위치의 픽셀 값을 먼저 복원하고 다음 위치의 값을 복구할 수 있도록 했는데 그 순서는 오른쪽, 오른쪽 아래 대각선, 아래, 왼쪽 아래 대각선, 왼쪽, 왼쪽 위 대각선, 위, 오른쪽 위 대각선 방향으로 함수를 계속 호출하면서 가장자리에서 중심으로 값을 채워나간다.

```
def inpaintImage(img,x,y, kernel):
    h,w = img.shape
    ksize = len(kernel) // 2
    if 1 > ksize or x> w-ksize or ksize > y or y>h-ksize:
        return
    if img[x,y]!=0:
        return

    offset = len(kernel) // 2
    subImage = img[x-offset:x+offset+1,y-offset:y+offset+1].copy()
    nonZeroR = subImage.copy()
    nonZeroR[nonZeroR!=0]=1
    nonZeroR = nonZeroR * kernel
    nonZeroR = nonZeroR / nonZeroR.sum()
    value = (subImage * nonZeroR).sum()

    img[x,y] = value

    inpaintImage(img, x+1,   y,  kernel)
    inpaintImage(img, x+1, y+1,  kernel)
    inpaintImage(img,   x, y+1,  kernel)
    inpaintImage(img, x-1, y+1, kernel)
    inpaintImage(img, x-1,   y, kernel)
    inpaintImage(img, x-1, y-1, kernel)
    inpaintImage(img,   x, y-1,  kernel)
    inpaintImage(img, x+1, y-1, kernel)
```

다음은 kernel을 구현하는 함수이다.

```
def BoxFilter(size):
    kernel = np.ones((size,size))
    kernel = kernel / kernel.size
    return kernel

def gaussianFilter(size, std):
    kernel = np.zeros((size,size))
    offset= size//2
    for i in range(-offset,offset+1):
        for j in range(-offset,offset+1):
            kernel[i+offset][j+offset] = math.exp( -1 * ( i**2 + j**2 ) / (2*(3**2)) )
    return kernel / kernel.sum()
```

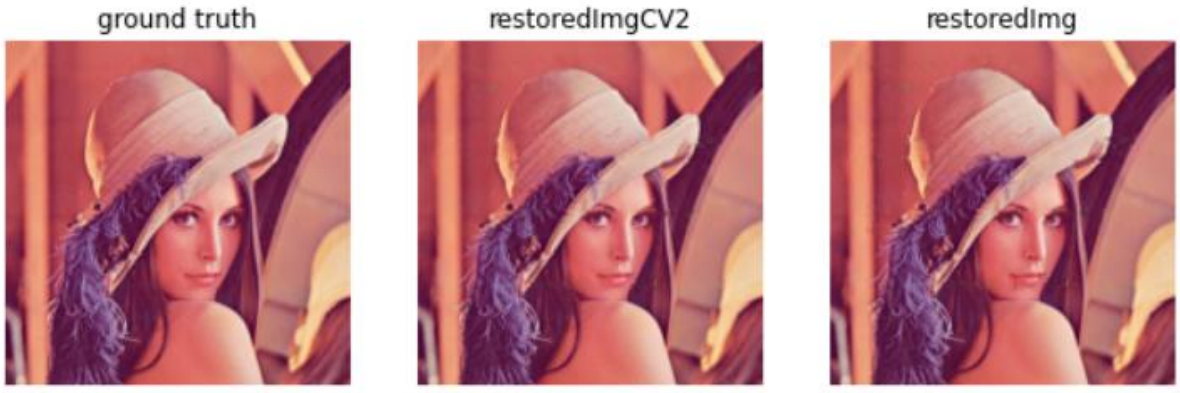
다음은 각 채널의 픽셀 값을 복원하는 과정이다. Gaussian kernel을 사용해 복원했다.

```
h,w = img_result.shape[:2]
kernel1 = BoxFilter(3)
kernel2 = gaussianFilter(3, 3)
restoredImg = img_result.copy()
for i in range(1,h-1):
    for j in range(1,w-1):
        if restoredImg[i,j,0]==0:
            inpaintImage(restoredImg[:,:,0], i, j, kernel2)
        if restoredImg[i,j,1]==0:
            inpaintImage(restoredImg[:,:,1], i, j, kernel2)
        if restoredImg[i,j,2]==0:
            inpaintImage(restoredImg[:,:,2], i, j,  kernel2)
```

다음은 OpenCV 라이브러리를 통해 복원하는 코드이다.

```
restoredImgCV2 = cv2.inpaint(img_result.astype('uint8'),text_mask,3,cv2.INPAINT_TELEA)
```

다음은 복원 결과이다. (HSV를 통한 분리)



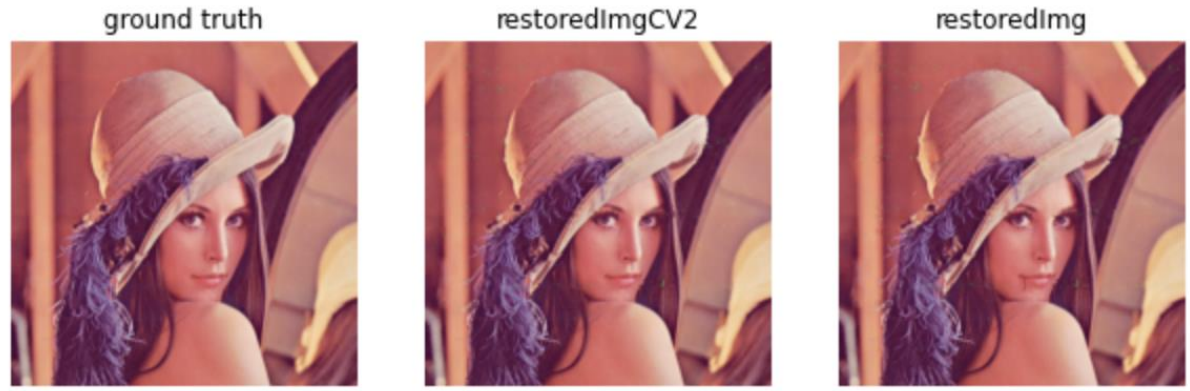
겉보기에 ground-truth이미지와 상당히 유사하게 복원된 것을 확인할 수 있었다.

3) Please compute the PSNR of restored image.

<18 point> 이미지의 복원 결과와 ground-Truth 이미지의 PSNR결과이다.

```
restored Image(OpenCV) VS Ground-Truth:  41.9956321805179
restored Image(my_method) VS Ground-Truth:  41.53065838026811
```

다음은 RGB공간을 통해 분리해서 복원한 결과이다. 18point에서는 거의 비슷한 성능을 보였다.



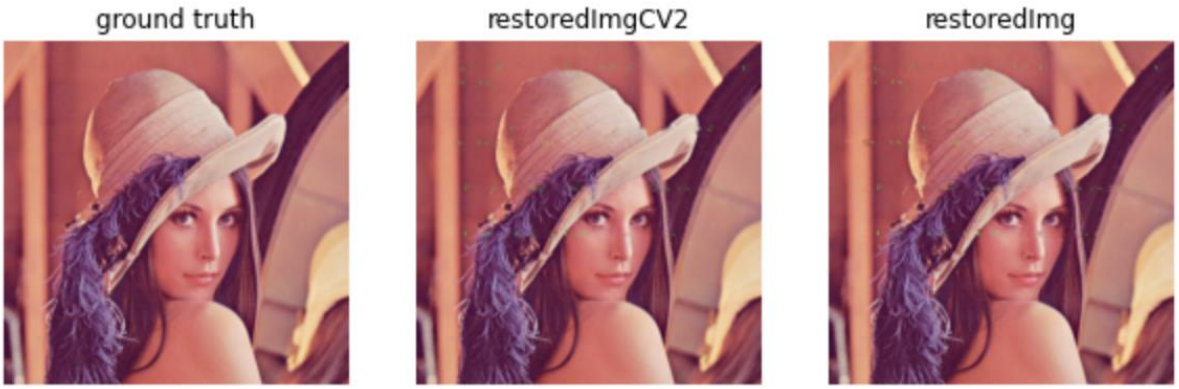

```
restored Image(OpenCV) VS Ground-Truth:  42.23146301871034
restored Image(my_method) VS Ground-Truth:  41.888786441591044
```

 # 나머지 글씨 크기에 대한 처리 결과 (RGB를 통한 분리 결과)

<10 point>

```
restored Image(OpenCV) VS Ground-Truth:  45.4908202142827
restored Image(my_method) VS Ground-Truth:  44.93517054230438
```

<14point>




```
restored Image(OpenCV) VS Ground-Truth:  43.62001196539442
restored Image(my_method) VS Ground-Truth:  43.337614464009235
```

3. (Image Transformation) Implement the followings and analyze and compare the results.

1) Transform the image with arbitrary angle and scale.

회전 행렬을 통한 이미지의 회전 변환은 이미지의 중심이 아닌 좌표축의 원점 기준이기 때문에 원점으로 translation한 이후에 회전을 취하고 다시 translation해야 한다. 또한, 축의 방향이 일반적인 좌표축의 방향과는 다르기 때문에 회전 행렬을 구할 때 유의한다.

이미지의 scaling과 회전을 위한 절차는 이미지 중심의 원점 이동, 스케일링, 회전, 이미지 중심의 원래 위치 이동으로 나눌 수 있다. 행렬 곱을 사용해서 4가지 변환을 하나의 행렬로 나타냈다.

이미지를 순회하며 transform을 적용하고 해당하는 위치에 픽셀 값을 옮겼다.

```
tx, ty = w//2, h//2
theta = np.radians(30)
k = 3
# Rotation Matrix
RMat = np.array([[math.cos(theta), math.sin(theta),0],
                 [-math.sin(theta),  math.cos(theta),0],
                 [              0,                0,1]])
# Scaling Matrix
SMat = np.array([[k,0,0], [0,k,0], [0,0,1]])
#  Translation Matrix
TMat = np.array([[1,0,tx], [0,1,ty], [0,0,1]])
#  inverse Translation Matrix (원점으로 위치시킴)
TMat_inv = np.linalg.inv(TMat)
# 이미지 중심의 원점이동 -> 스케일링 -> 회전 -> 중심 원위치
finalMat =  TMat @ RMat @ SMat @ TMat_inv

transformedImage = np.zeros((h,w))
for i in range(h):      # y
    for j in range(w): # x
        point = finalMat @ np.transpose([j,i,1]) # x,y,1
        toX = round(point[0]/point[2])
        toY = round(point[1]/point[2])
        if toX>=0 and toX<w and toY>=0 and toY<h:
            transformedImage[toY,toX] = img[i][j]
```

정상적으로 변환이 이루어졌지만 scaling을 1보다 크게한 경우 hole이 발생한다. 이를 해결하기 위해 2번에서 interpolation을 진행한다.

2) Use the nearest neighborhood (NN) and bilinear method for interpolation.

NN interpolation의 경우 backward mapping을 통해 찾아낸 좌표에서 가장 가까운 픽셀의 값을 사용하고 bilinear interpolation의 경우 찾아낸 좌표 근처 값들의 weighted sum을 픽셀 값으로 사용한다. Transform matrix의 inverse matrix 를 사용해서 원래 좌표를 찾아낼 수 있다. 그 값을 반올림해서 픽셀 값을 찾으면 NN interpoation을 구현할 수 있고 주변 픽셀까지 거리를 구해서 weighted sum을 하면 bilinear interpolation을 구현할 수 있다.

```
def NNinterpolation(t_img, img, tMat):
    h, w = img.shape
    t_img2 = t_img.copy()
    tMat_inv = np.linalg.inv(tMat)
    for i in range(h): # y
        for j in range(w): # x
            if t_img2[i][j]==0:
                point = tMat_inv @ np.transpose([j,i,1])
                y = round(point[1]/point[2])
                x = round(point[0]/point[2])
                if x>=0 and x<w and y>=0 and y<h:
                    t_img2[i][j] = img[y][x]
    return t_img2

# bilinear method interpolation
def bilinearInterpolation(t_img,img,tMat):
    h, w = img.shape
    t_img2 = t_img.copy()
    tMat_inv = np.linalg.inv(tMat)
    for i in range(h): # y
        for j in range(w): # x
            if t_img2[i][j]==0:
                point = tMat_inv @ np.transpose([j,i,1])
                x = point[0]/point[2]
                y = point[1]/point[2]
                x1, y1 = math.floor(x), math.floor(y)
                x2, y2 = x1 + 1, y1 + 1
                a = x-x1
                b = y-y1
                value = (1-a) * ( (1-b)*img[y1,x1] + b*img[y1, x2] ) #
                                + a * ((1-b) * img[y2,x1]  + b*img[y2, x2])
                t_img2[i][j] = value
    return t_img2
```

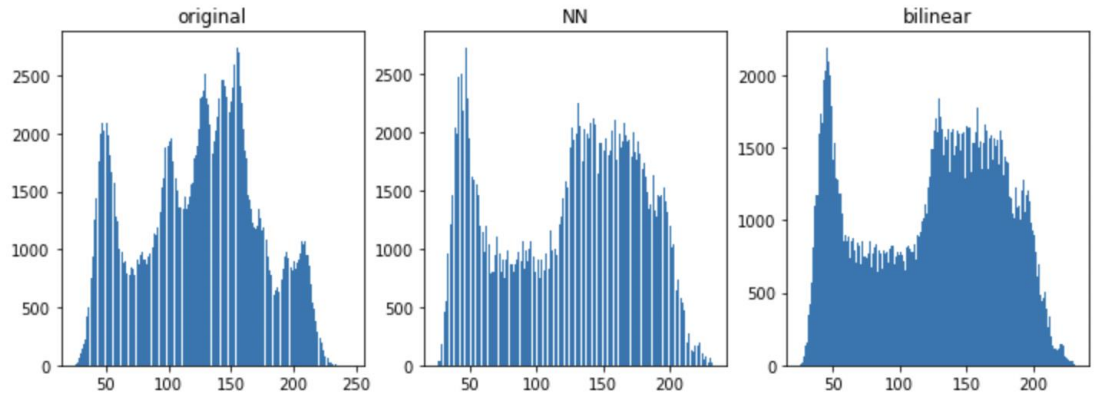
1번에서 transform을 했을 때 hole이 발생했던 theta=30, k=1.5변환에 대해 interpolaton을 진행했고 다음은 그 결과이다.

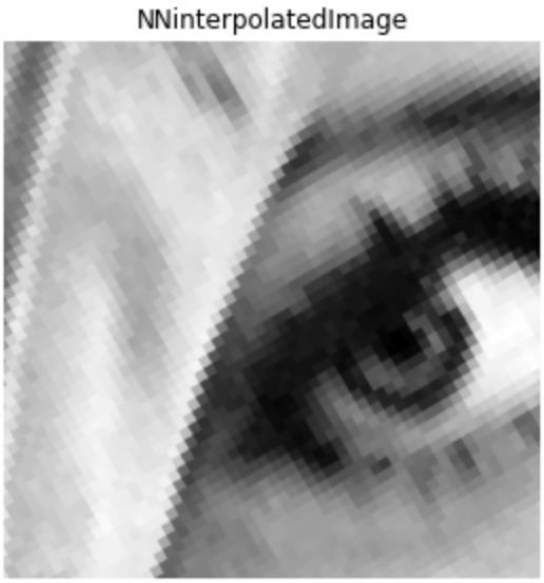
3) Compare the interpolation results.

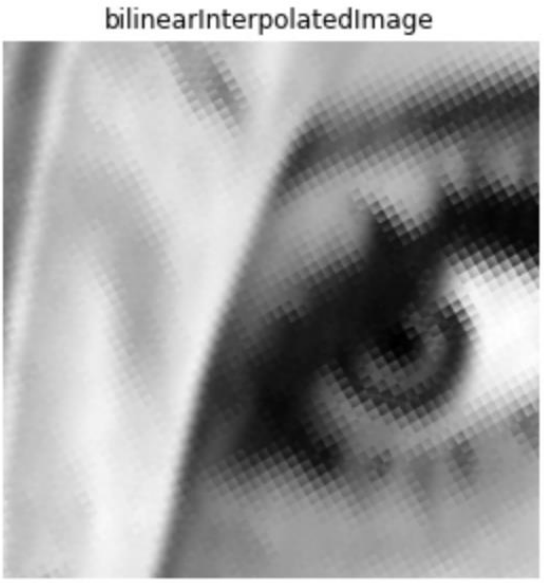
다음은 interpolation 결과를 비교하기 위해 plt를 통해 histogram을 그린 결과이다. FL#1에서 했던 대로 histogram을 직접 구현할 수도 있지만 plt.bar를 통해 histogram을 plot하면 막대그래프 사이사이 나타나는 빈 칸이 다음에서 설명할 두 가지 interpolation의 큰 차이를 명확하게 보여주지 못해서 plt.hist를 사용했다. 결과를 살펴보면 뚜렷하게 나타나는 차이점으로 bilinear의 경우 존재하지 않는 픽셀 값이 없고 픽셀 값들이 연속적으로 나타난다. Bilinear interpolation을 하면 원본 영상에서 나타나지 않던 픽셀 값들이 생겨날 수 있기 때문이다. 그 결과 이미지가 보다 부드럽게 나타날 것으로 예상된다.

```
plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
plt.hist(img.ravel(), 256)
plt.title('original')
plt.subplot(1,3,2)
plt.hist(NNinterpolatedImage .ravel(), 256)
plt.title('NN')
plt.subplot(1,3,3)
plt.hist(bilinearInterpolatedImage.ravel(), 256)
plt.title('bilinear')
```

```
Text(0.5, 1.0, 'bilinear')
```







예상 결과를 확인하기 위해 Scaling을 10배로 해서 이미지를 확인했다, 이미지에서 나타나는 픽셀 block들이 NN interpolation을 통한 결과에서 더 크게 나타났고 Bilinear interpolation에서는 상대적으로 작게 느껴져 보다 풍부한 픽셀 값을 갖는 것을 확인할 수 있었다. 이로써 예상 결과가 맞음을 확인할 수 있다.

4. (Hough Transform) Please implement the followings and analyze the results.

Hough transform을 통한 직선 검출 결과를 보다 뚜렷하게 확인하기 위해 인터넷에서 오각형 이미지를 새로 다운 받아 결과를 확인했다.

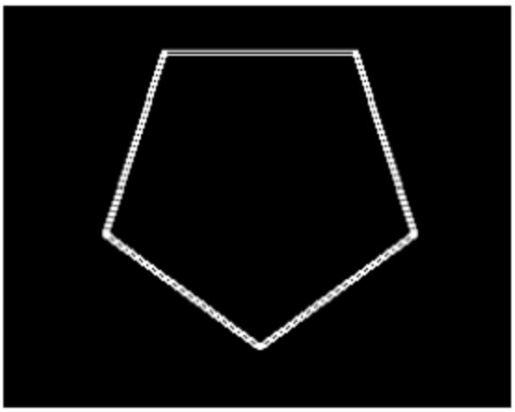
1) Find the edge map using Sobel mask.

Sobel mask를 이용한 filtering을 통해 에지를 검출했는데 오른쪽 결과이미지에서 확인할 수 있다.

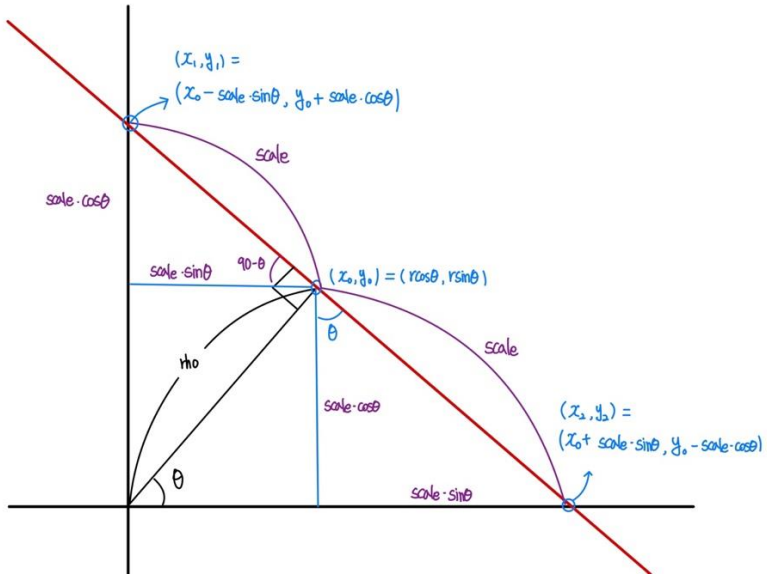
filtering함수는 앞에서 구현했던 방식과 동일한 함수를 사용했다.

```
def sobelEdgeDetection(img, threshold):
    mx = [ [-1,0,1],
           [-2,0,2],
           [-1,0,1] ]
    my = [ [-1,-2,-1],
           [ 0, 0, 0],
           [ 1, 2, 1] ]

    dx = filtering(img,mx)
    dy = filtering(img,my)
    s = np.sqrt(dx**2 + dy**2)  #strength
    o = np.arctan2(dx, dy)      #orientation
    f = s.copy()                #final
    f[ f>=threshold ] = 255
    f[ f<threshold ] = 0
    return dx, dy, s, o, f
```



기본적으로 hough transform은 직선을 나타내기 위해서 일반적으로 많이 사용하는 y=mx+n 꼴 방정식을 사용하지 않고 rho = a * cos(theta) + b *sin(theta) 꼴을 사용하는데 rho, theta를 통해 직선을 나타낼 수 있다. 다음은 rho, theta를 이용해 직선을 그리기 위한 두 점을 찾아내기 위해 직접 작성한 그림이다.



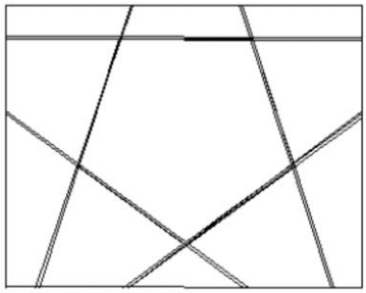
먼저 OpenCV를 통해 구현한 내용인데 cv2.HoughLines 함수를 사용해 threshold를 넘는 rho, theta pair 값들을 얻어내고 두 점의 좌표를 구한 뒤에 직선을 그어주었다. 결과 이미지를 보면 오각형의 에지를 포함한 직선이 사각형 위에 그려진 모습을 확인할 수 있다.

```
img = Image.open('pentagon.jpg').convert('L')
img = np.array(img)
h, w = img.shape
      # cv2.HoughLines(image, rho, theta, threshold  # rho : rho 정밀도 # theta : theta 정밀도
lines = cv2.HoughLines(f.astype('uint8'),1,np.pi/180,90)

for line in lines:
    rho,theta = line[0]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    scale = h+w #적당히 크게
    x1 = int(x0 - scale*b)
    y1 = int(y0 + scale*a)
    x2 = int(x0 + scale*b)
    y2 = int(y0 - scale*a)
    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),1)

plt.imshow(img, 'gray')
plt.axis('off')
```

```
(-0.5, 267.5, 211.5, -0.5)
```



2) From binary edge map, apply the Hough transform.

```
def hough_line(img):
    h, w = img.shape

    # theta: -90 ~ 180
    thetas = np.deg2rad(np.arange(-90, 180))

    # rho 범위 지정, 대각선 최대 거리
    diag_len = int(np.ceil(np.sqrt(h * h + w * w)))
    rhos = np.arange(0, diag_len)

    # hough domain
    hough_domain = np.zeros((len(rhos),len(thetas)))
    # non-Zero index 추출
    yindex, xindex = np.nonzero(img)

    for i in range(len(yindex)):
        x = xindex[i]
        y = yindex[i]
        for t in range(len(thetas)):
            rho = round(x * np.cos(thetas[t]) + y * np.sin(thetas[t]))
            hough_domain[rho, t] += 1
    return  hough_domain, thetas, rhos

hough_domain, thetas, rhos = hough_line(f)
```

Hough domain을 생성하고 Edge map에 나타난 좌표를 변환해서 hough domain에 voting한 뒤 그 결과를 thetas, rhos 배열과 함께 반환하도록 구현했다.

3) Analyze the result in Hough domain, and find the major lines.

4) Draw the lines on top of the image as below.

다음은 hough domain을 plot한 내용이다. 중간 중간에 점들이 몰려 흰색 점처럼 나타나는 지점이 있는데 이 지점의 index를 뽑아내면 rho와 theta값을 얻어낼 수 있으며 이를 통해 직선의 방정식을 얻어 직선을 그릴 수 있다. 직접 설정한 threshold에 대해 다음과 같은 theta, rho pair가 나왔다.

```
theta:  -1.5707963267948966 /  rho:  0
theta:  0.0 /  rho:  0
theta:  1.5707963267948966 /  rho:  0
theta:  1.5707963267948966 /  rho:  24
theta:  1.5707963267948966 /  rho:  26
theta:  2.199114857512855 /  rho:  64
theta:  2.199114857512855 /  rho:  66
theta:  0.3141592653589793 /  rho:  88
theta:  0.3141592653589793 /  rho:  89
theta:  0.33161255787892263 /  rho:  90
theta:  -1.5707963267948966 /  rho:  131
theta:  -0.33161255787892263 /  rho:  166
theta:  -0.3141592653589793 /  rho:  167
theta:  -0.3141592653589793 /  rho:  169
theta:  2.827433388230814 /  rho:  173
theta:  2.827433388230814 /  rho:  175
theta:  2.8099800957108707 /  rho:  176
theta:  1.5707963267948966 /  rho:  211
theta:  0.9424777960769379 /  rho:  223
theta:  0.9599310885968813 /  rho:  223
theta:  0.9250245035569946 /  rho:  224
theta:  0.9424777960769379 /  rho:  225
theta:  0.9250245035569946 /  rho:  226
theta:  0.0 /  rho:  267
theta:  -0.9424777960769379 /  rho:  276
theta:  -0.9424777960769379 /  rho:  278
theta:  -1.5707963267948966 /  rho:  316
theta:  -1.5707963267948966 /  rho:  318
```

```
threshold = 90
idx = np.where(hough_domain >= threshold)
thetas_idx = idx[1]
rhos_idx = idx[0]
for i in range(len(rhos_idx)):
    theta = thetas[thetas_idx[i]]
    rho   = rhos_idx[i]
    print('theta: ',theta,'/  rho: ', rho)
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    scale = h+w                 #적당히 크게
    x1 = int(x0 - scale*b)
    y1 = int(y0 + scale*a)
    x2 = int(x0 + scale*b)
    y2 = int(y0 - scale*a)

    # 직선 그리기
    try:
        m = (y2-y1)/(x2-x1)
    except:
        pass

    for x in range(w):
        try:
            y = round( m*(x-x1)  ) + y1
            if x>0 and y>0:
                img[y,x] = 0
        except:
                pass

    try:
        m = (x2-x1)/(y2-y1)
    except:
        pass

    for y in range(h):
        try:
            x = round( m*(y-y1)  ) + x1
            if x>0 and y>0:
                img[y,x] = 0
        except:
                pass
```

threshold를 초과하는 rho, theta 값에 대해 직선을 그리는 부분을 구현한 함수이다.

OpenCV에서 구했던 방식과 동일하게 두 점의 좌표를 구하고 이를 통해 직선의 방정식을 구해 x-y축으로 한 번 y-x축으로 한번 직선을 그린다.

이렇게 그리는 이유는 기울기가 무한대가 나오는 경우 직선이 그려지지 않기 때문이다. 축을 바꿔 그리면 다시 정상적으로 그릴 수 있다.

좌표 값이 이미지의 범위에 포함되지 않거나 기울기가 무한대가 되는 경우 except가 동작해서 픽셀 값을 넣지 않고 다음으로 넘어가게 구현했다.

다음은 원본 이미지에 직선을 그린 결과이다.